On-line Self-test Mechanism for Dual-Core Lockstep System-on-Chips

On-line Self-test Mechanism for Dual-Core Lockstep System-on-Chips

Andrea Floridia[1], Ernesto Sanchez

Dipartimento di Automatica e Informatica, Politecnico di Torino

Abstract

The Dual-Core Lockstep configuration is largely employed in safety-critical System-on-Chips for the sake of compliance with functional safety standards. Such configuration includes two processor cores paired together, always fed with the same identical inputs and their outputs are continuously compared by a set of comparators. However, permanent faults affecting the comparators may invalidate the system functionalities, thus in-field self-test mechanisms are mandatory. In this paper, different in-field self-test solutions are first discussed. Then, a hybrid hardware-software scheme for the on-line testing of the lockstep logic is proposed. Such a solution leverages test programs developed according to the Software-Based Self-Test (SBST) approach, used in conjunction with a specialized hardware module. The effectiveness of this approach was assessed on a modified version of the OpenRISC 1200 processor. Exhaustive experiments demonstrated that it is possible to achieve a fault coverage of stuck-at faults greater than 99%, while at the same time significantly reduce the area overhead of classical approaches.

Keywords: Multi-core Systems, On-line Testing, Self-Test, Safety-Critical Applications, Software-Based Self-Test, Functional Safety, Lockstep Computing

[1] Corresponding Author:
Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy
Corso Duca degli Abruzzi, 10129
Tel: +39 011 090 7091
Email Address: andrea.floridia@polito.it (Andrea Floridia)

1. Introduction

In the last decade, the high-quality requirements imposed by the automotive market radically changed the manufacturing process of the System-on-Chips (SoCs). These requirements not only imply an end-of-manufacturing testing plan able to yield a defect level (expressed in Defective Parts Per Million, DPPM) close to zero, but also the capability to detect hardware random failures when in mission mode.

For this reason, the usage of electronics devices in the automotive domain is regulated by the ISO 26262 [1] functional safety standard.

The standard covers the entire spectrum of the functional safety of electronics components in automotive applications, from the software to the hardware. Specifically, regarding the hardware, the ultimate goal is to avoid that a failure in a given hardware component lead to a catastrophic consequence (i.e., damage to human beings or properties). Towards this end, the standard defines the so-called *safety mechanisms*. A safety mechanism is a portion of the digital system intended for detecting faults, controlling system failures in order to achieve or maintain a *safe state*. Failures are generated due to the occurrence of a fault (either transient or permanent), which are classified as *Single-point faults* or *Multi-point Latent faults*. The former are immediately effective faults, since they would directly cause critical failures without a suitable safety mechanism guarding them. For the most critical systems, the predominant safety mechanisms commonly used against these faults are based on *redundancy*:

- End-to-End Error Correction Code (ECC) for memories [2], [3], [4];
- Redundant hardware execution units, Dual-Core Lockstep (DCLS): two processor cores (main and checker) are paired together, and their output is continuously monitored by a set of comparators [5], [6].

The additional circuitry introduced by those safety mechanisms must be equally tested. Indeed, the accumulation of *Multi-point Latent faults* could invalidate their functionalities. When these faults arise within the safety mechanisms, they do not cause directly a failure. However, they can become dangerous if a second fault arise in the module guarded by the safety mechanism. For this reason, additional

diagnostic safety mechanisms are required. Such safety mechanisms are mainly intended for implementing in-field self-test functionalities. During the power-up phase, the preferred self-test mechanism is based on Logic and Memory Built-In Self-Test (LBIST and MBIST respectively). The former targets mainly the permanent faults in the digital logic, while the latter in the embedded memories [7], [8]. In most of the cases, the MBIST can be executed transparently with respect to the content of the memory [9]. Since the LBIST is based on the already existing scan logic, it usually requires a full system reset after its completion. Therefore, its applicability is limited to the *Power-On Self-Test* (POST), that is the in-field test performed when the device is turned on.

This could become problematic if the time interval between two power-on events is too long, as in the case of several hours of continuous operations. Indeed, latent faults should be checked even when the system is fully on-line, that is with the mission software already running. This kind of in-field test is called *on-line test*. For this purpose, Software Test Libraries (STLs) are increasingly becoming adopted [10], [11], [12], [13], [14], [15]. An STL consists in a set of software self-test procedures, and the main target are permanent faults within the processor. This technique, also known in literature as Software-Based Self-Test or SBST, converts test patterns into instructions, and then accumulate their results to generate a test signature. Such signature is then used in field to determine whether the test passed or failed. Normally, this kind of self-test exclusively relies on the already existing on-chip resources.

In the context of this work, since the main target of an STL is the processor itself, SBST test programs are particular useful when used in conjunction with DCLS for avoiding latent fault accumulation in both the checker and the main core at run-time. It is important to highlight that the software approaches can produce pure functional stimuli, only. This is generally positive, since it complements the scan-based LBIST test, providing additional defect coverage for the processor core (being most critical module of a processor-based system) [16]. However, from the lockstep comparators standpoint it means that some critical faults cannot be addressed with the support of this method only. Indeed, some latent faults might escape the test. Specifically faults

within the lockstep comparators might cause faults within the main core being masked, inhibiting the lockstep functionalities. While most of the latent faults are detected during the POST with the application of the LBIST, when on-line usually a specific circuitry is added to the comparators for implementing the self-test. This additional hardware has the penalty of additional system area to be devoted exclusively for test purposes.

The aim of this paper is manifold. First, it discusses the different software and hardware self-test strategies commonly adopted by the industries for testing lockstep comparators. Then, it describes a novel hybrid hardware-software approach, to be used as an effective alternative to the classical mechanisms. The experiments demonstrated that it is possible to achieve almost the same fault coverage of hardware-based self-test mechanisms, while at the same time halving the area requirements.

The proposed approach extends the architecture proposed in [17], presenting an alternative hybrid software-hardware on-line testing strategy for *all the comparators* of a lockstep system, targeting *permanent latent faults* (e.g., stuck-at faults). The key idea is to move parts of the self-test features from the hardware to the software, leveraging the flexibility of a software approach, in conjunction with a specialized hardware module. Specifically, regarding to this new architecture, the paper provides:

1. Detailed description of a low-area overhead hardware module to be integrated within the SoC for assisting the execution of self-test routines oriented to the on-line test of the lockstep logic;
2. Guidelines for generating effective self-test routines;
3. Guidelines for performing the Failure Mode Effects and Diagnostic Analysis (FMEDA) of the additional hardware integrated within the SoC and possible countermeasures.

The rest of the paper is organized as follows: Section 2 discusses the most relevant related works to the one presented in this paper. Section 3 provides additional details about the ISO 26262 requirements and the FMEDA process, with the emphasis on the dual-core lockstep configuration and its possible failure modes: the goal is to present

the main motivations leading to this research; in Section 4, the descriptions of the hardware module and the proposed testing strategy are presented; Section 5 presents the case study, the experimental results and the results of the FMEDA process; finally, Section 6 concludes the paper.

2. Related Work

Hybrid solutions for testing purposes are not new. In [18], a memory core storing an SBST-like test sequence is inserted in the system and connected to the system bus. During the testing phase, the processor is forced to execute the instructions provided by this module. Also the authors of [19] proposed a solution which consists of a software-hardware-cooperated BIST as an attempt to reduce the test time for DRAM memories. Another hybrid solution that leverages the on-chip programmable resources has been presented in [20]. The approach suggests the usage of the embedded DMA for RAM memory testing. It is also worth to mention that hybrid approaches have been extensively adopted also as hardening mechanism against transient faults [21], [22].

In [23] additional instructions are added to the ISA for testing purposes only (e.g., accessing to particular flip-flops within the processor), whereas in [24], observation points are inserted within the processor for increasing the effectiveness of self-test programs. As an attempt to mitigate the cost and performances penalties introduced by the safety mechanisms, in [25] the authors proposed a cooperation between hardware and software modules. In order to achieve the targeted safety level for a given SoC: for the main computational units (e.g., the CPU), they suggest the usage of hardware-based application-independent safety mechanisms. For the remaining of the system (e.g., peripherals) a combination of hardware-software mechanisms (application-specific). Our previous works already explored hybrid approaches: in [17] the targeted module was the set of comparators that checks the integrity of data and address signals directed to the data and instruction memory. In this work, we

target also the comparators that monitor the control signals, along with a detailed analysis on the safety of the hybrid structure itself (not previously addressed). To achieve that, in this paper we resorted to a test structure similar to the one presented in [26]. In the aforementioned paper, the goal was to test computational modules only.

3. Background

3.1 The ISO 26262

Depending on the risk associated to the failure of a given hardware module, the ISO 26262 defines the Automotive Safety Integrity Levels (ASILs). These ASILs range from A (the lowest, indicating that the module is not safety relevant) to D (the most critical). For each of these levels, two metrics define the reliability requirements: the Single-Point Fault Metric (SPFM) and the Latent Fault Metric (LFM). For assessing whether or not a given hardware module is compliant with the targeted ASIL, the Failure Mode Effects and Diagnostic Analysis (FMEDA) is performed [27]. The FMEDA is a structured approach to define failure modes, failure rate, and diagnostic capabilities of a hardware component. In particular, failure rate and diagnostic capabilities directly contribute to determine the SPFM. While the failure rate (measured in Failure in Time, FIT) depends on the technology node used, the diagnostic capabilities are expressed with the Diagnostic Coverage (DC). The DC indicates the number of critical faults leading to a failure (i.e., single-point faults) detected by the safety mechanism: the most commonly used fault models for computing this metric are the stuck-at fault model and the transient fault model (i.e., Single-Event Upset, SEU). In early analyses, the SPFM can be approximated with the DC (computed via fault injections campaigns [28], [29]). Table 1 reports the requirements in terms of DC for each of the ASILs.

The LFM indicates instead the number of latent faults in the safety mechanisms covered by an in-field test mechanism (e.g., LBIST, MBIST or STL). For this metric, the test coverage is computed typically with respect to the stuck-at fault model. Depending on the ASIL, the standard defines also the Fault Tolerant Time Interval

(FTTI), that is the time required for detecting a fault and react accordingly. For ASIL D, the single-point fault FTTI is in the range 10-150 ms, while the latent fault FTTI is expressed as multiple of 10 hours.

Considering for example a processor core, in order to achieve these metrics, it is common to use DCLS for achieving the SPFM metric, while a combination of LBIST (for the detection of latent faults during the power-on phase) and STL (for the detection of latent faults during the on-line phase) for the LFM.

Table 1: ISO 26262 Diagnostic Coverage Requirements for Safety-relevant Modules

ASIL	SPFM/DC	LFM
B	At least 90%	At least 60%
C	At least 97%	At least 80%
D	At least 99%	At least 90%

3.2 The Dual-Core Lockstep Configuration

Dual-Core Lockstep (DCLS) is one of many techniques aiming at enhancing the reliability of processor-based systems. Major IP and Semiconductor companies involved in the automotive market provide DCLS-based solutions for safety-critical applications [6], [30]. Briefly, a system including DCLS consists of two identical processor cores, both initialized to the same initial state and fed with identical inputs. As a consequence, identical outputs should always be produced. A logic failure (due to permanent or transient faults) reaching the output in one of the two cores can be detected by continuously comparing their output. Once a failure is detected, the system reacts depending on the application requirements. Usually, one of the two cores is named the *main core*, while the other the *checker core*. The only purpose of the checker core is to confirm the correctness of the main core outputs, being fed with the very same instructions and data of the main core.

While extremely efficient to detect single-point faults, The DCLS configuration described above cannot detect failures that occur at the same point in both cores. These failures are normally called *common mode failures*, which cause comparators to

produce a false match. A common technique for reducing such risk of failure is to provide *temporal diversity* to the two cores composing the system. This strategy consists on delaying the inputs fed to the checker core by means of a bank of shift registers. Obviously, the outputs must be resynchronized before being compared. This is achieved by delaying the main core outputs by the same amount of clock cycles of the checker core inputs. It is worth noting that the whole architecture is completely transparent from the application code perspective: indeed, the checker core does not have any direct access to the system resources. The overall architecture is shown in Figure 1. For the sake of better comprehension, all the bits belonging to the same signal are grouped in a unique comparator (e.g., *DATA RAM* wires). The control signals (i.e., the system bus interface) are also grouped in a single comparator (*CTRL CMP*). All the comparators are then organized in a cluster, whom output is comparators' outputs OR-ed each other's. Hardware faults in the comparators (Figure 2) could either cause a false alarm or an undetected critical failure. In the former, the *ALARM* signal is fired even though the two inputs match, while in the latter the signal is not fired even though the two inputs differ. Clearly, a false alarm is positive since it means that the hardware fault is detected. Instead, the second effect is potentially dangerous since a failure of the main core is not reported correctly. Therefore, it is evident that in order to obtain complete system dependability, it is necessary to devise a suitable self-test mechanism for latent faults to be applied when on-line.

For simplicity, in the following it is assumed that the cluster has the structure of Figure 1 but without any particular knowledge of the comparators implementation.

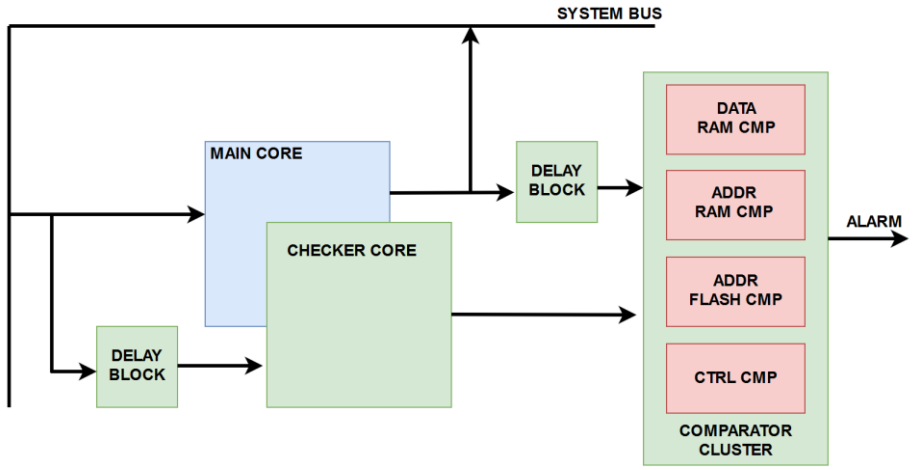


Figure 1: Typical delayed DCLS architecture.

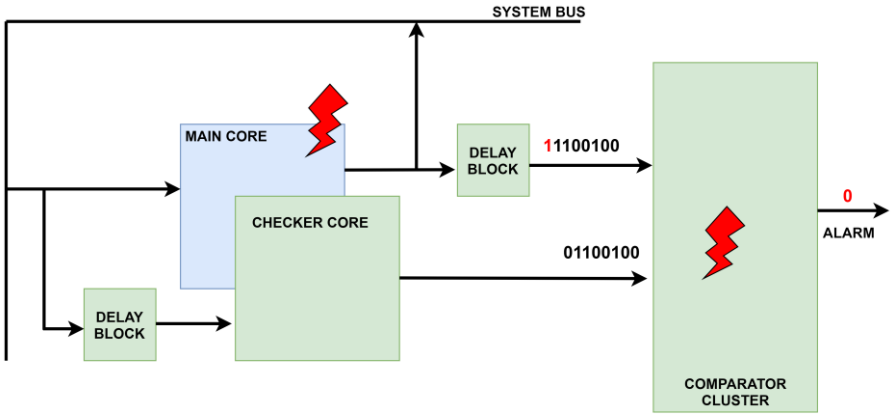


Figure 2: Example of latent fault causing a failure of the main core being masked.

4. Limitations of pure Hardware and Software Strategies

As already shown in Section 3, the basic elements composing a DCLS system are the two processor cores and a set of comparators. Authors of [31] already proposed an effective test strategy for the comparators. To fully test an m-bit wide comparator (i.e., two m-bit wide inputs), $2m+2$ test patterns are required. As stated by the authors,

240 the effectiveness of such patterns is independent on the low-level implementation of the comparators. The 2*m* patterns generate a mismatch in only one bit at a time, and in practice this correspond to a walking 0 (or 1), starting from the MSB (or LSB) up to the LSB (or MSB). The last two patterns correspond to the case in which the two comparator inputs are equal, namely both inputs with all bits at 1 and then at 0. An
245 example of the test patterns applied by this algorithm to a set of 4-bit comparators is shown in Table 2.

Table 2 Test algorithm for a 4-bit wide comparator

# pattern	Input A	Input B
1	0111	1111
2	1011	1111
3	1101	1111
4	1110	1111
5	1111	0111
6	1111	1011
7	1111	1101
8	1111	1110
9	1111	1111
10	0000	0000

Intuitively, when considering a pure software approach (e.g., execution of self-test
250 procedures belonging to an STL), the aforementioned test stimuli can be hardly generated in the system under analysis. As the reported experimental results confirm, the structure of a DCLS configuration imposes a constraint on the generation of those test patterns. In particular, both the checker and the main cores are fed with the very same inputs and always produce the same outputs (unless a fault is present). As a
255 consequence, it is not possible to create the difference needed by the 2*m* patterns. Moreover, a further challenge derives from the fact that the DCLS comparators do not only check for the integrity of data, but also *all the processors' outputs*. Therefore, the previous algorithm must be translated in a proper sequence of instructions that force

the processor to generate those values (which is not feasible with a pure software approach).

On the other hand, a pure hardware self-test (e.g., [32]) guarantees the completeness of the test in terms of generated test patterns. Nevertheless, it suffers the problem of excessive additional hardware devoted exclusively for testing purposes.

5. Proposed Approach

The main intent of this study is to propose an efficient low-area overhead solution for the on-line test of the most crucial part of the lockstep system, namely the DCLS comparators.

5.1 Hybrid self-test architecture

During the in-field test of the DCLS, a pure functional-based approach is only able to produce 2 out of the required $2m+2$ test patterns, reaching in this way an insufficient fault coverage. Thus, in order to overcome this problem, the proposed test strategy is composed of two main elements: a set of test programs and a hardware module. The latter, called the Lockstep Self-test Management Unit (LSMU), supports the test programs during their execution. Figure 3 depicts the overall architecture.

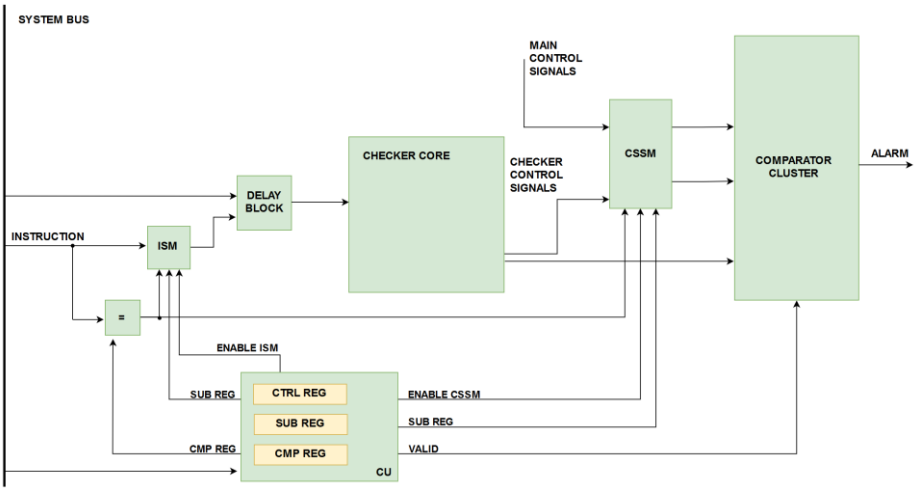


Figure 3: Architecture of the system including the LSMU. For simplicity, the main core is not included in this figure.

The LSMU is composed of two parts, a control unit (CU) and a datapath (DP). The CU includes the bus interface logic, an FSM and a set of registers; the DP is made of a comparator, the Instruction Substitution Module (ISM) and the Control Signals Substitution Module (CSSM). The LSMU is directly connected to the system bus and it exclusively intercepts all the instructions that the checker core receives as input. At each clock cycle, it monitors whether a particular instruction (hereinafter *target instruction*) is going to be fed to the checker core. If and only if the instruction going to be fed to the checker core is the target instruction, the ISM replaces such instruction with a so-called *substituted instruction*. Similarly, the LSMU receives as input the output control signals of both the checker and main cores. When the CSSM is active and the *target instruction* is fed to the cores, the value of the control signals is substituted with a specific *test pattern* intended for testing the control comparators. As it is demonstrated later in this section, while for address and data comparators the ISM is enough for generating the required test patterns, the CSSM is required for control comparators since controlling the value of control signals is not always feasible via software. Relying exclusively on the ISM would yield a lower fault coverage, since the control comparators would not be correctly addressed. Moreover, the reader should note that the output control signals are substituted right before being

fed to the comparators. Thus, the SoC is transparent to this substitution, since the real output directed to the system (namely the main core outputs) are left unchanged.

Instructions, test patterns and other functionalities can be programmed by the main core via a set of registers at runtime, as a standard memory-mapped peripheral. As mentioned before, testing the comparators requires to create a difference in just one of the bits fed to the DCLS comparators. This is not always possible since normally the comparators are grouped into a single cluster and the execution of different instructions by the two cores could lead to different control signals activated contemporary. More than one comparator active at the same time may cause masking problems, since their outputs are OR-ed together. For this reason, during the test phase, the main core can program the LSMU to disable the comparators that should not be tested (*VALID* signals in Figure 3) during the current test session so that they will not influence the targeted comparator. For example, let us assume that both the main and the checker cores execute a store byte but accessing to different addresses. As a consequence, a different byte should be selected which means that different control signals are activated, hence two different comparators are fired (*CTRL CMP* and *ADDR RAM CMP* in Figure 1).

For implementing the behavior described above, three registers are required:

- *COMPARE REGISTER (CMP REG)*: containing the target instruction. The instruction written in this register must be encoded as in the program memory;
- *SUBSTITUTE REGISTER (SUB REG)*: containing the substituted instruction or the test pattern to be applied to the control comparators;
- *CONTROL REGISTER (CTRL REG)*: this register drives the behavior of the FSM and DCLS comparators to be disabled.

The bit-width of the first register depends on the considered processor architecture (i.e., 32/64-bit architecture), the second one should have a bit-width at least equal to the *CMP REG* (for fitting the substituted instruction). Possibly, if the number of control signals is higher, additional bits should be comprised in this register. The

CTRL REG depends on the number of comparators. The first three bits of *CTRL REG* are dedicated to the substitution mode. To provide as much flexibility as possible, without bounding the test programs to any particular implementation, two substitution modes are provided:

- *ONE-SHOT SUBSTITUTION*: it substitutes the target instruction only once, and then the LSMU disables itself;
- *CONTINUOS SUBSTITUTION*: the ISM continuously substitutes the target instruction, until the LSMU is disabled by the main core.

The remaining bits are dedicated to the *TEST MODE*, which allows disabling the comparators. The fourth bit specifies whether *TEST MODE* should be entered or not, while there are as many bits as the number of comparators to be disabled. During the normal operational phase, all the comparators are enabled. If the *TEST MODE* bit is set, then only those comparators for which the corresponding bit is set are enabled. Finally, one bit should be reserved for enabling the ISM or the CSSM. As an example, let us consider again the architecture of Figure 1. Given such architecture (once again, all the control signals are grouped in a single comparator), the *CTRL REG* has the structure shown in Figure 4. In the aforementioned figure, it is assumed that the *ADDRESS FLASH* comparators are enabled when the test mode is entered. Thus, it is not necessary to have a further dedicated bit within the *CTRL REG*.

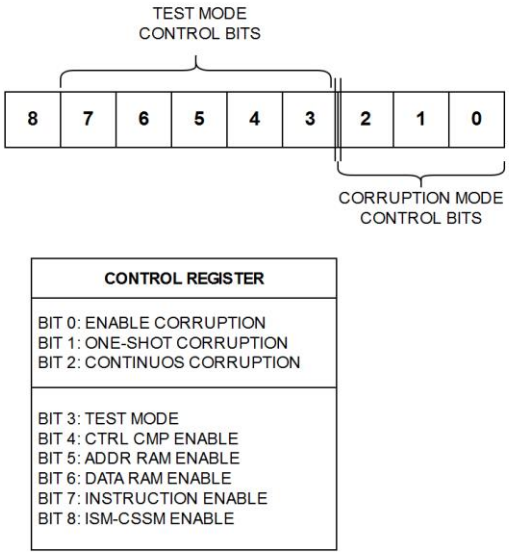


Figure 4: Control register (CTRL REG) description. The ADDRESS FLASH CMP are automatically enabled when the TEST MODE bit is set.

5.2 On-line testing flow

In order to test the DCLS comparators, it is necessary to execute a suitable test program while enabling the LSMU, providing in this way, the desired stimuli to the targeted hardware. Let us consider first the case in which the test of DATA RAM comparator (each input is 32-bit wide) is performed. The test program structure is the one reported in Figure 5. For sake of compactness, it was assumed that the *mov* instructions support the sign-extension (as it happens with many modern RISC-based processors).

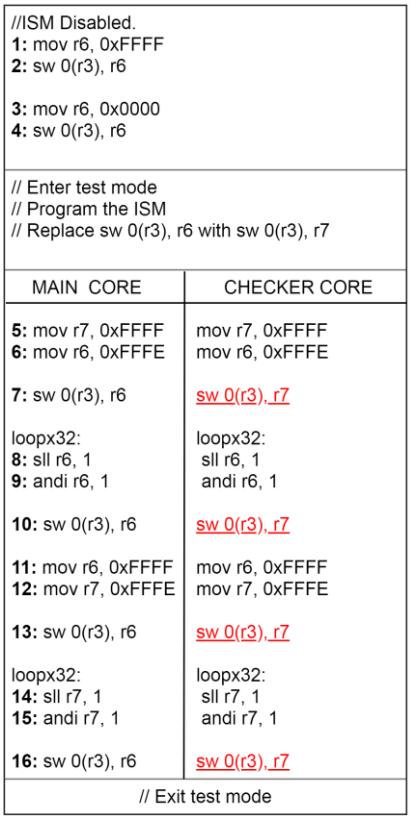


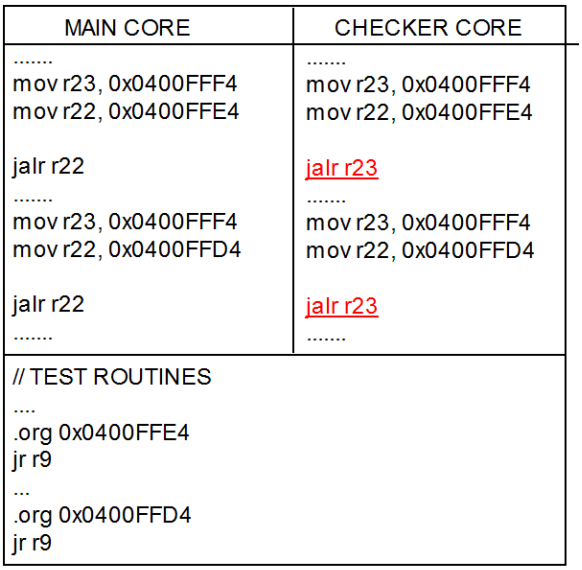
Figure 5: A possible SBST program that leverages the ISM for testing the DATA RAM CMP. Before ISM activation, both cores execute the same instructions. After its activation, the instructions underlined in red are those substituted in the checker core.

Testing that comparator can be achieved by the execution of a sequence of store operations to a fixed address, each of these providing one of the $2m$ patterns. The first two store operations (lines 1 to 4 in Figure 5) correspond to apply the hexadecimal patterns 0xFFFFFFFF, 0x00000000. Since the same values should be applied to both inputs, the ISM is not active.

In the next step, since it is required to apply a difference to the comparator, the ISM must be activated.

Clearly, the reader should note that the three registers are programmed depending on how the test patterns are generated by the test program. Assuming that the continuous substitution mode is enabled, the *CTRL REG* is programmed with the value 001001011. Considering the example, the *CMP* and *SUB* registers are programmed to replace the store operation with the store of a different value (i.e., register R7 instead of R6 as in line 7, 10, 13, 16). Then, the patterns are generated with a loop that implements the walking 0 followed by a store operation of the newly generated pattern (lines 5 to 10). It is worth noting that one input must vary while the other one must be maintained unchanged; thus, the generation loop must be applied twice: first on R6, then on R7 (lines 11 to 16). Given the presence of the ISM, the final effect is to have the main core producing the first *m* patterns (walking 0 on the first input) and then the checker core producing the remaining *m* patterns (walking 0 on the second input). Once all the patterns are applied, the module is disabled, and the test routine terminates.

The same reasoning applies to the *ADDRESS RAM* comparators. The test program structure is similar to the one discussed before, but the value stored is fixed while the addresses are generated resorting to a walking 0 strategy. In most of the cases, the two patterns 0xFFFFFFFF, 0x00000000 cannot be applied as the accessibility to these addresses depends on the particular memory map of the system under analysis. Indeed, normally RAM addresses are restricted to a particular range. Thus, to maximize the fault coverage, the two patterns correspond to storing data at the lowest possible address (with as many 0 as possible) and at the highest possible address (with as many 1 as possible). Then, starting from the highest address, walking 0 is applied. The same strategy pertains also for *ADDRESS FLASH*, but instead of a sequence of store operations, function calls are required. As depicted in Figure 6, each jump forces a different address (at each generated address, a valid function or piece of code must be present) so that all the required patterns are generated (clearly, within the address range of the program memory).



MAIN CORE	CHECKER CORE

.....
mov r23, 0x0400FFF4
mov r22, 0x0400FFE4
 	
jalr r22
.....
mov r23, 0x0400FFF4
mov r22, 0x0400FFD4
 	
jalr r22
.....
	.....
	mov r23, 0x0400FFF4
	mov r22, 0x0400FFE4
	
	jalr r23
	.....
	mov r23, 0x0400FFF4
	mov r22, 0x0400FFD4
	
	jalr r23
	.....

// TEST ROUTINES .... .org 0x0400FFE4 jr r9 ... .org 0x0400FFD4 jr r9	

Figure 6: Fragment of SBST program testing the ADDRESS FLASH CMP.

It is important to note that forcing the main and the checker cores to jump at two different addresses in the program memory is totally safe, since in a DCLS system (Figure 1), the checker outputs are directed to the comparators only, while its inputs are driven by the main core. Therefore, it is always the main core that drives the execution flow in both cores.

When dealing with the control comparators (*CTRL CMP* in Figure 1), the CSSM is required to be active. In this case, the self-test routine should: 1) setup the LSMU so that CSSM is active and all the other comparators but the CTRL CMP disabled; 2) configure the *CMP REG* with the target instruction; 3) configure *SUB REG* with the test pattern to be applied; 4) Execute the target instruction, which in this case behaves as a trigger for the substitution. If the *CONTINUOUS SUBSTITUTION* mode is used, step 3 and 4 are repeated until all test patterns are applied. The LSMU allows for any test pattern to be applied, without any particular restriction. For obtaining a short and efficient test, they can be generated internally to the self-test routine following the walking bit strategy presented so far (e.g., as in Figure 5). Clearly, patterns can be stored in the memory Flash as constants. However, this would require additional space for the testing routine.

Finally, the adoption of the LSMU requires the insertion of the *VALID* signal (which acts as an enable) within the comparators. To achieve a complete fault coverage, it is required to test the hardware introduced for implementing such signals. For each comparator, the self-test routine should maintain the other comparators disabled and: 1) enable the target comparator and force the inputs such that they differ and then correspond; 2) disable the target comparator and force the inputs such that they differ and then correspond once again. In the fault-free scenario, the outcome of step 2 should be independent from the value of the inputs. Step 1 is intrinsically implemented in the self-test algorithms presented in this section. Step 2 would instead require a custom routine.

5.3 Observation mechanism

Alarms raised by DCLS comparators are normally handled by a specific module integrated within the SoC, that reacts depending on the application requirements. Clearly, in order to adopt the strategy described above, such module must include a configuration setup that keeps track of the DCLS alarms raised during the on-line testing procedure and report any unexpected misbehavior.

6. Experimental Results

The following section is organized as follows: initially, the description of the case study is provided. Successively, the gathered experimental results for both pure software and hardware self-test methods are discussed. Finally, the effectiveness and overhead of the proposed hybrid approach is reported, along with a detailed FMEDA process.

6.1 Case Study

Experiments were conducted on an in-house modified version of the OpenRISC 1200 (OR1200) soft-core processor. It consists of a 32-bit pipelined RISC microarchitecture, with MMU and basic DSP functionalities. It includes also data and instruction caches. For the purpose of this work, caches and MMU were not included in the final synthesized version. The RTL source files are described in Verilog and are available from GitHub [33], [34]. Finally, the system includes also a behavioral description of a Flash and RAM memory. All the logic simulations were performed using Synopsys VCS, whereas Synopsys Design Compiler as logic synthesis tool. The effectiveness was assessed by means of fault simulation campaigns, using Z01X by Synopsys. Z01X is used widely for functional safety verification, providing an extremely flexible environment for the fault simulation. All the experiments were performed on a workstation with an Intel Xeon CPU running at 2.5 GHz, equipped with 12 cores, and 256 GB of RAM.

A delayed DCLS configuration was implemented at the CPU level (i.e., all the logic within the CPU was duplicated), with a temporal diversity of two clock cycles between the two cores. A further bank of flip-flops was added for main and checker outputs to isolate any critical path from the comparators.

Concerning the fault simulation campaigns, stuck-at faults were exclusively considered (1,374 faults), being the most commonly used fault model for this kind of analyses. Nevertheless, the reader should note that the method is easily extensible to other fault models. Although the set of considered faults is relatively small, from the safety viewpoint they are extremely relevant. Indeed, the processor core is one of the main components of modern SoC and any failure of this unit is likely to affect the main application.

6.2 Evaluation of pure software self-test

In the following, the fault coverage results of a set of software programs are analyzed. The programs used during the experiments fall into two categories:

- Application programs;
- Test programs targeting stuck-at faults within the CPU core only.

For sake of conciseness, stuck-at faults located within *DATA RAM*, *ADDRESS RAM* and *ADDRESS FLASH* comparators were considered. These three modules account for 936 stuck-at faults. In order to show the ineffectiveness of a software approach when addressing the type of faults mentioned above, four application programs were initially considered (Table 3). They implement simple applications: vector sorting (*bubble_sor*}, *quick_sort*), minimum path identification in a graph (*dijk*) and random number generation (*lfsr_32*).

Table 3: Characteristics of applications programs

Programs	Duration [c.c.]	Size [Bytes]	FC [%]
dijk	5,170	604	71.34
bubble_sort	1,184	192	70.84
quick_sort	3,278	932	71.34
lfsr_32	3,304	388	71.09

Table 4: Stuck-at oriented SBST Programs

Test Programs	Duration [c.c.]	Size [Bytes]	FC [%]
rf_test	1,502	3,508	71.84
cu_test	538	808	71.34
opmux_test	308	484	71.09
alu_test	10,820	3,448	71.84
mac_test	3,248	2,596	71.84
lsu_test	2,108	4,216	72.08
genpc_test	24,914	2,690	71.84
wbmux_test	538	808	71.34

The DCLS modules are hard to be tested even when targeting the on-line testing of the CPU core via SBST programs. Indeed, it was considered also a set of eight handcrafted test programs (Table 4), each test program was developed to test a specific part of the processor core. The fault coverage of the entire test suite against the whole processor stuck-at faults is 80.79%. For each program in Table 4 and 3 the duration (in clock cycles, C.C.), the size (expressed in bytes) and the achievable fault coverage (FC) are reported.

By comparing Table 3 and 4, one could immediately observe that for both sets, the fault coverage saturates at around 71%. The only test program that reaches a fault coverage of 72% is the one addressing processor Load Store Unit (LSU). Such test program generates a considerable activity on the memory interface; thus, it is reasonable that the fault coverage is higher than any other program in the two sets.

6.3 Evaluation of pure hardware self-test

As hardware self-test mechanism for the test of the comparators, the architecture described in [32] was selected. For sake of a fair comparison, the hardware module was designed in order to generate the whole set of test patterns described in [31]. It is important to notice that the architecture described in [32] does not specify any test sequence to be used. The post-synthesis results are shown in Table 5. When applying patterns generated with this method, the overall fault coverage is 99.7% (obtained in about 500 clock cycles). The major drawback of this approach stems from the fact that the area overhead is directly proportional to the bit-width of the lockstep comparators (i.e., number of signals to be compared). The remaining faults not covered by this approach are related to the reset circuitry and as further explained in the next sub-section, they never produce a failure when in mission mode.

Table 5: Hardware Self-Test [32] Synthesis - Area Breakdown

Module	Area [μm^2]	Total Area [%]
DCLS CPU	140,891.39	100.0
Self-test module [32]	6,293.6	4.47

6.4 *Evaluation of the proposed hybrid self-test architecture*

The LSMU was designed in Verilog and included in the final RTL version. The overall system architecture is the same described in the previous section in Figure 3 and Figure 1, with all the control signals grouped in a single comparator. The entire system was synthesized and mapped to a 65*nm* CMOS technology library. The post-synthesis results are shown in Table 6.

Table 6: LSMU Synthesis - Area Breakdown

Module	**Area [µm^2]**	**Total Area [%]**
DCLS CPU	140,891.39	100.0
LSMU (ISM)	335.40	0.2
LSMU (CSSM)	204.36	0.1
LSMU (CU)	2,107.55	1.5

As it can be noticed, the LSMU accounts for the 1.88% of the of the entire DCLS CPU. The controller (CU) contains the *CMP*, *SUB*, *CTRL* registers and the finite-state machine (FSM). The latter accounts for the 0.2%. It includes also the system bus interface, which represents the most expensive part (in terms of logic gates) of the block. It is worth noting that the ISM includes also the comparator for the target instruction shown in Figure 3. For avoiding performance degradation, the ISM was placed in between the two banks of flip-flops that delay the checker inputs.

In the following, the effectiveness of test programs leveraging the LSMU are analyzed. Four self-test programs were developed, each of them targets a specific comparator (including the test of the *VALID* signal). Table 7 summarizes the achieved fault coverage. While the *data_ram* achieves quite high fault coverage, the remaining programs were not so effective. Indeed, they are mainly limited from the fact that some fault locations are not accessible due system memory map. Therefore, an analysis of the fault list was performed in order to identify possible *Safe Faults*. As specified in the ISO 26262, these are faults whose occurrence do not cause any

failure. Following the guidelines presented in [35], it was possible to remove faults due to the system memory map. It is important to notice that these are *on-line functionally untestable* faults, which are *application independent* and therefore they can be individuated in any device. In the system under analysis, the following valid addresses exist:

- RAM from 0x0000_2000 to 0x001F_FFFF;
- Flash from 0x0400_0000 to 0x0400_FFFC.

The reasoning behind this procedure stems from the fact that the processor is able to access only a portion of the available address space. As an example, according to the specifications mentioned above, when dealing with flash addresses the upper part of the address holds exclusively the value 0x0400. This causes having logic gates stuck to fixed value during the whole in-field behavior: as a consequence, it is not possible to change the value of some bits by executing any software. Such faults were identified resorting to TetraMax by Synopsys. The comparators inputs were connected to V_{dd} or ground and then, given these constraints, the tool was instructed to perform a structural untestability analysis on the modified netlist.

After this process, it was possible to remove about 20% of faults from *ADDRESS RAM* and *ADDRESS FLASH* comparators fault list. Then, further analyses were conducted on the remaining faults. Specifically, by using Inspect by Synopsys, it was possible to link the remaining faults to the reset signal of the comparators (that include flip-flops for breaking critical paths). Such faults would prevent the flip-flops from being initialized during the reset phase. However, when designing lockstep systems, it is common practice to invalidate comparators outputs for a certain number of clock cycles after the system leaves the power-on reset. This prevents receiving bogus data from both checker and main. As a consequence, those faults never provoke a failure (thus can be considered as safe) of the lockstep because when the comparators become active, they receive initialized data. The total percentage of safe faults removed is around 13% of the initial fault list. The final fault coverage after this pruning process is shown in the fifth column of Table 7.

Using the hybrid test strategy described in previous section it was possible to achieve an overall fault coverage of 99.5% for all the cluster, with reasonable test duration and program size. Considering a clock period of 40*ns* (as in the performed experiments), the test programs were executed in 198.8μs, while their overall memory footprint is about 4KB. Lastly, it is noteworthy that the test program *address_flash* has a memory footprint almost double with respect to the other two test programs due to the additional portions of code needed by the test strategy for testing those comparators.

Table 7: Characteristics of the ad-hoc test programs exploiting the LSMU

Test Program	Duration [c.c.]	Size [Bytes]	FC [%]	Safe FC [%]
data_ram	1,392	818	99.03	100.0
addr_ram	1,626	1,308	91.35	100.0
addr_flash	1,132	1,738	75.32	99.12
ctrl_cmp	1,820	706	91.22	100.0

6.5 LSMU Failure Mode Effects and Diagnostic Analysis

Normally, a complete FMEDA analysis involves also the computation of the failure rate. However, this depend on the technology used for the final implementation. Clearly, since the goal of the paper is to introduce a new architecture, these data are missing. For this reason, the focus of this analysis is mainly centered in determining the possible failure modes of the LSMU (and the related critical faults causing the failure), assessing their impact (i.e., whether they lead to a critical failure) and possible countermeasures against these faults.

The possible failure modes (denoted as FM) affecting the LSMU (and its submodules) *due to permanent faults* are:

- *FM1*: the ISM is active, but is not able to correctly substitute the target instruction with the substituted instruction;

- *FM2*: The ISM is active, but is not able to recognize the target instruction (namely it does not perform the substitution at all);
- *FM3*: The CSSM is active, but is not able to substitute the control signals with the test pattern at all;
- *FM4*: The CSSM is active, but is not able to correctly substitute the control signals with the test pattern;
- *FM5*: The LSMU is not active, but the CU disables the lockstep comparators;
- *FM6:* The ISM is not active, but it performs a substitution of the instruction;
- *FM7*: The CSSM is not active, but it performs a substitution of the control signals.

Failure modes FM1-4 do not impact the safety of the application, since they become relevant during the test mode only. Thus, the only inconvenient is a lower fault coverage. FM7 is likely to be detected by the lockstep comparators and do not cause any critical failure since CSSM outputs are directly connected to those comparators. To further eases the detection of these faults, the reset value of the *SUB REG* could be set to a value such that only 1 bit differ (e.g., all zeros and a bit at one only). By doing so, it is possible to immediately detect the occurrence of faults causing this specific failure mode.

Failure modes FM5 and FM6 are quite critical instead, since directly impact the normal execution of any user application. For identifying critical faults that cause these failures, it was built a functional safety verification environment with Z01X. For increasing the truthfulness of the experiments, *functional fault simulations* (that is fault simulation of the entire SoC including the memories) were performed. The chosen fault model was still the single stuck-at, and fault were injected in the LSMU logic only (3,286 faults). When performing these evaluations, it is important to specify *observation points* and *diagnostic points*. The former are points of the design (namely, ports or internal wires) where to observe the effect of the faults. The latter

instead are points of the design where to observe the reaction of the safety mechanism. Faults detected in both observation and diagnostic points can be labeled as *dangerous detected*. All the faults detected in an observation point but not in a diagnostic one are labeled as *dangerous undetected*. The remaining faults can be considered as safe faults.

For both failure modes, the same application programs used for the experiments in [17] were executed during the fault simulations. For FM6, using as reference Figure 3, the observation point was placed on the checker inputs. The FM6 causes a wrong instruction to be fed to checker, thus by observing its inputs, it is possible to identify faults leading to that failure mode. After running the fault simulations, the 5.17% of the total faults cause a wrong instruction to be fed to the checker. As safety mechanism (i.e., countermeasure) for these faults, it is possible to use the already existing comparators of the DCLS configuration. This can be done since it is assumed a single stuck-at fault only. The fault simulations were repeated, and the diagnostic point was placed on the *ALARM* signal. At the end of this campaign, all the 5.17% critical faults were detected also at the diagnostic point. Thus, they can be labeled as dangerous detected whereas the remaining 94.83% as safe since they do not cause this specific failure.

The same procedure was performed for FM5 as well. Since this failure mode causes lockstep comparators to be disabled, the observation point was situated on the *VALID* signal, output of the LSMU. In this case, 5.42% of the total faults lead to this failure and they all belong to the FSM within the CU. As countermeasures, two options are possible: leverage the self-test routines of Table 7 or use a Triple Module Redundancy (TMR) configuration for the FSM within the LSMU.

Considering the first option, by placing the diagnostic point on *ALARM* (namely the same when testing the lockstep comparators) 97.25% of the critical faults results being detected. This means that most of the faults are actually detected during the test of the DCLS comparators.

Concerning the second option, using FSM in a TMR configuration on one hand increases the fault coverage, the detection time, but also silicon area of the LSMU. As already shown at the beginning of this section, the FSM accounts for the 0.2% of the

total design. By using a lockstep variant, the overhead of this module increases up to the 0.66%. Although a TMR configuration is adopted, it is important to notice that few faults in the majority voter logic might still lead to a failure. However, such faults are less than the 0.2% and thus there is still compliance with the ISO 26262 standard. Table 8 summarizes the possible failure modes and the countermeasures, along with the achievable *Diagnostic Coverage* (indicated with DC, being the number of critical faults detected by the safety mechanism).

Table 8: FMEDA LSMU - Failure Modes and Countermeasures

Sub-module	Failure Mode	Critical	Safety Mechanism	DC [%]
ISM	FM1	NO	NONE	-
ISM	FM2	NO	NONE	-
CSSM	FM3	NO	NONE	-
CSSM	FM4	NO	NONE	-
CU	FM5	YES	Self-Test Programs	97.25
CU	FM5	YES	TMR-FSM	99.99
ISSM	FM6	YES	DCLS	100.0
CSSM	FM7	YES	DCLS + SUB reg safe reset	100.0

6.6 Discussion of the three possible approaches

Table 9 summarizes the three self-test alternatives described in this section. The first important observation is that the STL solely does not detect all the possible faults within this specific safety mechanism. Therefore, the STL must be necessarily complemented with either a pure hardware self-test module (e.g., [32]) or the proposed hybrid architecture. The former yields a complete fault coverage with a short test application time. Instead, the proposed one is still able to generate the

required test stimuli, while mitigating the area overhead introduced by a pure hardware self-test approach (halving the area overhead). On the other hand, since now part of the stimuli are generated via software, the test duration is higher.

Unlike pure hardware strategies, in which the test patterns are hardwired, the test performed with the proposed approach is much more flexible: test patterns are not anymore fixed and can be updated on-the-fly. Finally, it is worth mentioning that the area reported in Table 9 for the proposed approach includes also the additional circuitry that mitigates the critical failures.

Table 9: Area, Fault Coverage and Test Duration for the three considered approaches.

Self-test Approach	**Area w.r.t DCLS [%]**	**FC [%]**	**Duration [c.c.]**
Hardware [32]	4.47	99.7	500
STL	0.00	72.0	43,976
LSMU	2.10	99.5	5,970

7. Conclusions

This paper reports a detailed analysis of the possible self-test mechanisms to be used for the on-line testing of the comparators required for implementing a DCLS configuration. It was proven through the experiments the inadequacy of an STL to address all the possible permanent faults within the comparators. At the same time, it was shown the overhead in terms of area of a pure hardware self-test module. An alternative self-test approach is then introduced: it is based on the insertion into the system of a hardware module (LSMU) that assists test programs for the generation the required test stimuli. Therefore, the proposed strategy is hybrid, since it is based on both software and hardware.

The proposed strategy provides several advantages:

1. Low hardware overhead compared to a pure hardware approach: part of the test patterns generator is implemented in software;
2. Flexibility: as it is partially based on software, it inherits its flexibility: the test can be split in different test sessions to fit the test time budget when on-line. Moreover, the test patterns are not fixed anymore and can be updated as required;
3. Promotes IP re-usability: the hardware module is the least intrusive as possible. Indeed, it is not required any modification nor a detailed knowledge of the processor core. The system designer oversees the integration of the module in the final SoC, as it happens with a standard IP;
4. Scalability: the hardware overhead minimally depends on the complexity of the considered processor. The ISM depends only on the considered architecture (e.g. 32 or 64-bit), while the CSSM depends on the number of control signals of the processor core.

The paper provides also a set of countermeasures to be used against single-point faults that could arise within the LSMU, in order to improve the safety of the module. Latent faults within the LSMU can be addressed following the same approach of a pure hardware module (i.e., LBIST-based approaches).

Although the single stuck-at fault model was used as fault model, the applicability of the concepts presented in this work are easily extensible to other fault models.

References

[1] Road vehicles – functional safety, in: ISO 26262, 2011.

[2] E. Fujiwara, Code Design for Dependable Systems: Theory and Practical Application, Wiley-Interscience, New York, NY, USA, 2006.

[3] C. L. Chen, M. Y. Hsiao, Error-correcting codes for semiconductor memory applications: A state-of-the-art review, IBM Journal of Research and Development 28 (2) (1984) 124–134.doi:10.1147/rd.282.0124.

[4] R. Mariani, F. Colucci, P. Fuhrmann, Safety integrity of memory sub-systems in automotive microcontrollers, SAE Transactions 116 (2007) 486–496.URLhttp://www.jstor.org/stable/44719916

[5] X. Iturbe, B. Venu, E. Ozer, S. Das, A triple core lock-step (tcls) armR©cortexR©-r5 processor for safety-critical and ultra-reliable applications, in: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), 2016, pp. 246–249.doi:10.1109/DSN-W.2016.57.

[6] Application Note - Cortex-M33 Dual Core Lockstep, in: ARM InfoCenter, 2017.

[7] G. Tshagharyan, G. Harutyunyan, Y. Zorian, An effective functional safety solution for automotive systems-on-chip, in: 2017 IEEE International Test Conference (ITC), 2017, pp. 1–10.doi:10.1109/TEST.2017.8242075.

[8] T. McLaurin, Periodic online lbist considerations for a multicore processor, in: 2018 IEEE International Test Conference in Asia (ITC-Asia), 2018, pp. 37–42.doi:10.1109/ITC-Asia.2018.00017.

[9] M. Nicolaidis, Theory of transparent bist for rams, IEEE Transactions on Computers 45 (10) (1996)1141–1156.doi:10.1109/12.543708.

[10](2019)ARM:[Online].Available:https://www.arm.com/products/development-tools/embedded-and-software/software-test-libraries.

[11](2019)Infineon:[Online].Available:https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/pro-sil-safetcore-safetlib/.

[12](2019)Renesas:[Online].Available:https://www.renesas.com/en-eu/products/synergy/software/add-ons.html#read.

[13](2019)Cypress:[Online].Available:http://www.cypress.com/file/249196/download

[14](2019)Microchip:[Online].Available:http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf.

[15] P. Bernardi, R. Cantoro, S. De Luca, E. Sanchez, A. Sansonetti, Development flow for on-line core self-test of automotive microcontrollers, IEEE Transactions on Computers 65 (3) (2016) 744–754.doi:10.1109/TC.2015.2498546.

[16] P. C. Maxwell, R. C. Aitken, V. Johansen, Inshen Chiang, The effect of different test sets on quality level prediction: When is 80% better than 90%?, in: 1991, Proceedings. International Test Conference,1991, pp. 358–.doi:10.1109/TEST.1991.519695.

[17] A. Floridia, E. Sanchez, Hybrid on-line self-test strategy for dual-core lockstep processors, in: 2018IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems(DFT), 2018, pp. 1–6.doi:10.1109/DFT.2018.8602982.

[18] P. Bernardi, L. M. Ciganda, E. Sanchez, M. S. Reorda, Mihst: A hardware technique for embedded microprocessor functional on-line self-test, IEEE Transactions on Computers 63 (11) (2014) 2760–2771.doi:10.1109/TC.2013.165.

[19] T. Hsieh, J. Li, K. Wu, J. Lai, C. Lo, D. Kwai, Y. Chou, Software-hardware-cooperated built-in self-test scheme for channel-based drams, in: 2017 International Test Conference in Asia (ITC-Asia), 2017, pp.107–111.doi:10.1109/ITC-ASIA.2017.8097122.

[20] P. Bernardi, R. Cantoro, L. Gianotto, M. Restifo, E. Sanchez, F. Venini, D. Appello, A dma and cache-based stress schema for burn-in of automotive microcontroller, in: 2017 18th IEEE Latin American Test Symposium (LATS), 2017, pp. 1–6.doi:10.1109/LATW.2017.7906767.

[21] S. Campagna, M. Violante, An hybrid architecture to detect transient faults in microprocessors: An experimental validation, in: 2012 Design, Automation Test in Europe Conference Exhibition (DATE),2012, pp. 1433–1438.doi:10.1109/DATE.2012.6176590.

[22] P. Bernardi, L. M. V. Bolzani, M. Rebaudengo, M. S. Reorda, F. L. Vargas, M. Violante, A new hybrid fault detection technique for systems-on-a-chip, IEEE Transactions on Computers 55 (2) (2006)185–198.doi:10.1109/TC.2006.15.

[23] Wei-Cheng Lai, Kwang-Ting Cheng, Instruction-level dft for testing processor and ip cores in system-on-a-chip, in: Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232),2001, pp. 59–64.doi:10.1109/DAC.2001.156108.

[24] M. Nakazato, S. Ohtake, M. Inoue, H. Fujiwara, Design for testability of software-based self-test for processors, in: 2006 15th Asian Test Symposium, 2006, pp. 375–380.doi:10.1109/ATS.2006.260958.

[25] R. Mariani, T. Kuschel, A flexible microcontroller architecture for fail-safe and fail-operational systems,2010.

[26] A. Floridia, G. Mongano, D. Piumatti, E. Sanchez, Hybrid on-line self-test architecture for computational units on embedded processor cores, in: 2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS), 2019, pp. 1–6.doi:10.1109/DDECS.2019.8724647.

[27] R. Mariani, G. Boschi, F. Colucci, Using an innovative soc-level fmea methodology to design in compliance with iec61508, in: 2007 Design, Automation Test in Europe Conference Exhibition, 2007, pp.1–6.doi:10.1109/DATE.2007.364641.

[28] A. Floridia, E. Sanchez, M. Sonza Reorda, Fault grading techniques of software test libraries for safety-critical applications, IEEE Access 7 (2019) 63578–63587.doi:10.1109/ACCESS.2019.2917036.

[29] A. Nardi, A. Armato, Functional safety methodologies for automotive applications, in:2017IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2017, pp. 970–975.doi:21

10.1109/ICCAD.2017.8203886.

[30] X. Iturbe, B. Venu, E. Ozer, J.-L. Poupat, G. Gimenez, H.-U. Zurek, The arm triple core lockstep(tcls) processor, ACM Trans. Comput. Syst. 36 (3) (2019) 7:1–7:30.doi:10.1145/3323917.URLhttp://doi.acm.org/10.1145/3323917

[31] H. Grigoryan, G. Harutyunyan, S. Shoukourian, V. Vardanian, Y. Zorian, Generic bist architecture for testing of content addressable memories, in: 2011 IEEE 17th International On-Line Testing Symposium,2011, pp. 86–91.doi:10.1109/IOLTS.2011.5993816.

[32] Rangachari, S., Jalan, S.: Self test for safety logic. United States Patent US9964597B2, 8 May 2018.

[33]https://openrisc.io/.

[34]https://github.com/openrisc/or1200.

[35] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, O. Ballan, On-line functionally untestable fault identification in embedded processor cores, in: 2013 Design, Automation Test in Europe Conference Exhibition (DATE), 2013, pp. 1462–1467.doi:10.7873/DATE.2013.298.22