Low-Cost Strategy to Detect Faults Affecting Scrubbers in SRAM-Based FPGAs

*Published Version:*
M. Grossi, M.B. (2022). Low-Cost Strategy to Detect Faults Affecting Scrubbers in SRAM-Based FPGAs. MICROPROCESSORS AND MICROSYSTEMS, 89, 1-9 [10.1016/j.micpro.2022.104437].

*Availability:*
This version is available at: https://hdl.handle.net/11585/893391 since: 2022-09-13

*Published:*
DOI: http://doi.org/10.1016/j.micpro.2022.104437

# Low-Cost Strategy to Detect Faults Affecting Scrubbers in SRAM-Based FPGAs

Marco Grossi[1], Meryem Bouras[2], Martin Omaña[1], Hassan Berbia[2]

Corresponding author : marco.grossi8@unibo.it , Tel. 0039-0512093038

[1] Department of Electrical Energy and Information Engineering "Guglielmo Marconi" (DEI), University of Bologna, Bologna, Italy

[2] Department of Embedded Systems Engineering, Mohammed V University in Rabat, Rabat, Morocco

## Abstract

SRAM-based Field Programmable Gate Arrays (FPGAs) are vulnerable to SEUs. For applications demanding high reliability this problem is often solved by integrating in the system a scrubber, a circuit that periodically scans the FPGA configuration memory and reconfigures it if an error is detected. Since the scrubber is usually implemented in the same FPGA device, it is also vulnerable to SEUs, thus the scrubber reliability is increased by adopting standard fault tolerance techniques. These solutions guarantee the scrubber reliability, but generally require a large area overhead.

In this paper, we present a novel low-cost strategy capable to detect faults in the FPGA configuration memory implementing the scrubber. The proposed technique is based on time redundancy, forcing the scrubber output to produce an error indication for each word read from the FPGA memory, in order to detect the faults affecting the portion of FPGA memory implementing the scrubber. The implementation of our proposed strategy presents a negligible impact in terms of area overhead (4.17%) and a limited increase in power consumption (22.9%) over the original (unprotected) scrubber. As for the impact on system performance introduced by our strategy, it is of approximately the 38.2% over the unprotected scrubber, but it can be significantly lowered by reducing the frequency at which the scrubber is applied to test the FPGA.

**Keywords**: fault tolerance; reliability; SRAM based FPGA; configuration memory; time redundancy.

## 1. Introduction

The Attitude Determination and Control System (ADCS) is an on-board component of satellites, whose correct operation is essential to meet the satellite mission. In fact, the ADCS performs the spacecraft attitude control and maneuvers, tracking a predefined, nominal orbit and maintaining a preferred orientation in space. Usually, the ADCSs are implemented by means of SRAM based Field Programmable Gate Arrays (FPGAs) [1], in order to reduce costs and enable the possibility to reconfigure the system in the field. However, as known, the configuration memory of SRAM-based FPGAs is vulnerable to SEUs, especially for on-board satellite applications, where external disturbances, such as trapped particles, cosmic and solar radiations, geomagnetic field interferences, etc. are very likely to occur [2]. Considering that the ADCS is a crucial element for the satellite operation, it is of utmost importance to increase its robustness against SEUs, in order to guarantee its correct operation, thus the reliability of the whole space mission [1]. Moreover, transient and permanent faults can also affect the operation of FPGA based Networked Control Systems (NCSs) used in harsh industrial environments, with possible cathastropich consequences to users and/or the environment [3].

Consequently, several approaches have been presented in literature to increase the robustness of SRAM-based FPGAs against SEUs. In particular, the use of low-cost Error Detection and Correction codes and Interleaving has been largely studied in the literature (e.g., codes in [4, 5]). On the other hand, the use of the scrubbing techniques has been also studied in literature (e.g., the techniques in [5, 6, 7]). These techniques have been proven to be effective to protect the configuration memory of SRAM-based FPGAs [4, 7, 8, 9, 10, 11, 12, 13, 14].

Scrubbing techniques usually adopt an Error Detecting Code (EDC) to detect the presence of SEUs affecting the configuration memory. They read periodically byte after byte (actually two simultaneous bytes) of the FPGA memory and verify the presence of erroneous bit(s). If an error is detected, then the portion of the FPGA containing the erroneous bit(s) is reconfigured [15, 16].

There are two main types of scrubbing techniques: the internal and the external scrubbers [5, 6, 7, 17]. The external scrubber uses a second FPGA, different from the FPGA that implements the main circuit, for the scrubbing circuit, while the internal scrubber implements the scrubber circuitry in the same FPGA of the main circuit. Internal scrubbers are more effective in terms of time performance and area occupation [17].

More in details, commercial FPGAs scrubbers usually employ the Cyclic Redundancy Check (CRC) code as EDC to detect the presence of SEUs affecting the configuration memory [7]. In such scrubbers, a signature (or checksum) is added to each word (16 bits of information) stored in the configuration memory of the FPGA. During FPGA in-field operation, this scrubbing technique reads the FPGA memory periodically and verifies the checksum of the stored words. If the checksum is incorrect, the scrubber generates an error indication, and the portion of the FPGA containing the erroneous word is reconfigured [16, 18].

A problem of existing scrubbing techniques is that they are implemented within FPGAs, thus they are also vulnerable to SEUs [4, 7, 8, 9, 10, 11, 12, 13, 14]. In fact, SEUs can affect both the part of the FPGA memory implementing the main circuit as well as part implementing the scrubber. These latter SEUs may change the functionality of the scrubber [4, 6, 12, 19, 20], making it unable to detect successive SEUs affecting the FPGA, with possible catastrophic results for the system functionality.

In order to cope with this problem, different solutions have been proposed in literature to enhance the reliability of scrubbers [7, 8, 14, 21, 23]. Most of these solutions are based on the triplication (or duplication) of the scrubber, in order to tolerate (or detect) SEUs affecting the portion of the FPGA memory implementing them [6, 20, 22, 23]. A general limitation of these solutions is the significant area overhead they require, which may prevent their use in some applications with strict area requirements (e.g., these solutions are too expensive for on-board ADCS applications, like the one described above).

Based on these considerations, in this paper we propose a novel low-cost strategy to detect SEUs affecting the part of the FPGA memory implementing the scrubber. We consider scrubbers using the CRC code as EDC, since it is the EDC most widely used by scrubbers. However, our strategy can be straightforward modified to be used also with other kind of EDCs. Rather than using space redundancy, our strategy employs time redundancy to detect SEUs affecting the part of the FPGA memory implementing the scrubber, thus our strategy requires a significant smaller area overhead compared to space redundance approaches (e.g., like Triple Modular Redundancy – TMR).

Our strategy periodically tests the correctness of the words stored in the FPGA memory and the behavior of the scrubber by executing in sequence the following two steps: 1) check the correctness of the checksums of the words read from the FPGA memory (i.e., we verify the absence/presence of errors on the word being tested by the scrubber); 2) check the ability of the scrubber in detecting incorrect words affected by SEUs (i.e., our approach purposely induce bitflips on the words during this step to emulate the presence of SEUs). In step 2) an error indication is expected at the scrubber output in case of scrubber correct behavior (i.e., in case of no SEU affecting the portion of the FPGA memory implementing the scrubber).

As shown in the paper, the implementation of our proposed strategy requires a negligible area overhead (4 NOR and 3 AND gates) over the original (unprotected) scrubber, area overhead that is also negligible compared to that required by alternative solutions based on TMR. Moreover, the power consumption required by our proposed approach is also a small fraction of the power consumption of the FPGA memory.

The rest of this paper is organized as follows. In Section 2 an overview of the scrubbing techniques for SRAM based FPGAs is presented. In Section 3, some techniques in literature to mitigate the occurrence of SEUs on scrubbers for SRAM based FPGAs are presented and discussed. In Section

4, the proposed technique for low-cost detection of SEUs in CRC based scrubbers is discussed and its performances are presented. Finally, conclusions are drawn in Section 5.

## 2. FPGA Scrubbing Techniques

In order to implement scrubbing techniques in FPGAs, a dedicated Cyclic Redundancy Check (CRC) generator is used during the FPGA configuration to calculate a checksum for each word (or frame) to be stored on the FPGA memory. Such checksums are stored on the FPGA memory together with their associated frames, and they are used later during the FPGA in-field operation to detect the presence of SEUs in the configuration memory.

The configuration of the blocks composing SRAM based FPGAs (e.g., the CLBs, the routing resources, the blocks of RAM, the IO blocks, etc.) is programmed through a bitstream of words (frames), whose size depends on the particular FPGA device and the considered application [18]. For example, for the Virtex-5 FPGAs from Xilinx, the bitstream of the configuration memory is composed by 41 words of 32 bits each (1,312 bits).

Each frame [8] has a unique address that is related to the physical position in the FPGA floorplan, and the position in the floorplan is related to a specific resource (e.g. CLB, RAM, DSP, IOB, etc.). Each column of configuration memory defines a specific type of resource (e.g., CLB, DSP, etc.) [7, 8].

In order to protect the configuration memory of SRAM-based FPGAs against SEUs or MBUs, scrubbing techniques are usually adopted. These techniques read continuously (scrub), frame by frame, the FPGA configuration memory to detect the presence of SEUs. If an SEU is detected in a frame, the portion of the memory affected by the SEU is reconfigured without interrupting the normal FPGA operation. The circuit that performs scrubbing is commonly called scrubber [4, 15]. There are different kinds of scrubbing techniques, such as blind scrubbing, readback scrubbing, frame level scrubbing and model scrubbing [4, 15]. In this paper, we consider the readback scrubbing, which is the scrubbing technique requiring the lowest power consumption [4, 15], thus being the most suitable for the considered on-board ADCS application.

During the FPGA configuration process, a golden copy of the bitstream is stored in a non-volatile memory (PROM or flash ROM) that is immune to SEUs. Then, during normal operation in the field the memory is readback frame by frame. For each frame read from memory, the scrubber recalculates the CRC checksum, and compares it with the CRC generated during the configuration phase, and stored together with the frames. If due to an SEU the regenerated checksum is different from that stored in memory, an error indication is generated by the scrubber, and the part of the configuration memory of the FPGA containing the erroneous frame is rewritten with the data stored in the golden copy [16].

A problem of this scrubber is that SEUs affecting the part of the FPGA memory implementing the scrubber may change its functionality, which in turn may prevent the detection of successive SEUs affecting the portion of the FPGA memory implementing the main circuit, with consequent catastrophic results for the system functionality.

In order to avoid this problem, we propose a novel low-cost strategy that is able to detect SEUs affecting the part of the FPGA memory implementing the scrubber itself.

## 3. Related works

In the last years, many scrubber designs have been proposed to mitigate the effects of SEUs affecting the part of the FPGA memory implementing the scrubber itself. Most scrubbers are based on hardware redundancy, mainly adopting the conventional Triple Module Redundancy (TMR) technique, where three copies of the scrubber feed a majority voter. This solution guarantees a high reliable scrubber, but also requires high area occupation and power consumption. Zhang et al. in 2018 presented a scrubbing strategy based on TMR and implemented it on a Xilinx FPGA [24]. The results have shown that the proposed approach provides a quick repair of the SEUs and can improve the reliability of SRAM-based FPGAs. Sielewicz et al. in 2017 proposed an experimental method for the evaluation of TMR-based mitigation techniques on the Xilinx Kintex-7 FPGA [25]. The proposed architecture was evaluated under different redundancy topologies, such as no mitigation methods, triplication of the combinational logic, triplication of the output registers, triplication of the voter circuits as well as combination of these techniques. Irradiation experiments have been carried out at the isochronous cyclotron at the Nuclear Physics Institute of the Academy of Sciences of the Czech Republic and the reliability of the different designs evaluated.

On the other hand, Giordano et al. in 2018 introduced a scrubber that is implemented in a PicoBlaze 8-bit microcontroller running at 100 MHz [26]. The scrubbing algorithm is implemented by software in the microcontroller and the microcontroller reliability is guaranteed by TMR implementation on different modules of the processor. The proposed system has been implemented on an electronic board based on the Xilinx Kintex-7 70T FPGA and the results have shown that the reliability is increased by 42% and 290% if compared to a standard TMR approach and no mitigation techniques, respectively. Wilson et al. in 2021 also proposed a FPGA system based on a 32-bit pipelined VexRiscv processor [27] implemented on the Digilent Nexys Video development board integrating also the XC7A200T-ISBG484C FPGA. Two different versions of the processor were designed, one with unmitigated design and the other with TMR approach and triplicated voters. The results have shown how the TMR based soft-core processor provides a 33x improvement in reliability at the cost of 5x resource utilization and decreased operating frequency. Shaker et al. in 2020 presented a FPGA system implementing a penta modular redundancy (5MR) approach capable to detecting SEUs and multiple event upsets (MEUs) [28]. The system is implemented using the Kintex7 7k410tfbg676 FPGA device and adopts a dynamic partial reconfiguration to increase the system reliability.

While TMR provides a good protection against SEUs in SRAM-based FPGA designs, this technique requires a significant increase of the resource utilization, which can be unacceptable in particular design with tight constraints on the available resources. Thus, alternative approaches requiring lower resource overhead have been proposed, at the cost of a lower protection against

SEUs. Machado Matsuo et al. in 2018 proposed a Dual Modular Redundancy (DMR) mitigation scheme for an heterogeneous CPU-FPGA platform [29]. Keller and Wirthlin in 2018 presented a partial triple modular redundancy (pTMR) for fault mitigation in an FPGA system [30]. The pTMR technique consists in the logic triplication of only a few sub-modules that represent a small fraction of the total area but are particularly vulnerable to SEUs. The paper shows that this approach enables 6x increase in the system reliability compared to the unmitigated design, at the cost of only 2.8% increase in terms of area overhead.

A different approach to protect SRAM-based FPGA designs against SEUs is the adoption of time redundancy strategies. Time redundancy strategies are characterized by a negligible area overhead, but they require that system operations are executed multiple times in sequence, resulting in a non negligible impact on system performance, that can conflict with the requirements in terms of execution time of some real-time systems. As discussed in [31], time redundancy approaches are particularly suited for applications where erroneous results can be discarded and individual operations can be re-executed, or where an application can be restarted without serious consequences for the system.

Villa et al. in 2019 presented a fault tolerant technique based on time redundancy for SEUs detection and recovery in soft-core processors [32]. The architecture of the soft-core processor LEON3 designed on FPGA was modified to implement a fault tolerant technique based on checkpoint recovery. Checkpoints are saved during the program execution and, when an error is detected, program execution stops and returns to the last safe checkpoint. Bahramali et al. in 2011 proposed a fault detection scheme of secure hash algorithm (SHA-1 and SHA-512) for implementation in FPGA [33]. The computation is broken in two parts with a pipeline inserted in between. Each part is computed twice and the results compared to detect potential faults. Ibrahim et al. in 2014 presented a comparative study on the performance of FPGA based systems where SEUs are mitigated with time redundancy and hardware redundancy [34]. The solutions were implemented by using the Xilinx FPGA Virtex 5 LX50T. The paper shows that TMR requires 3x

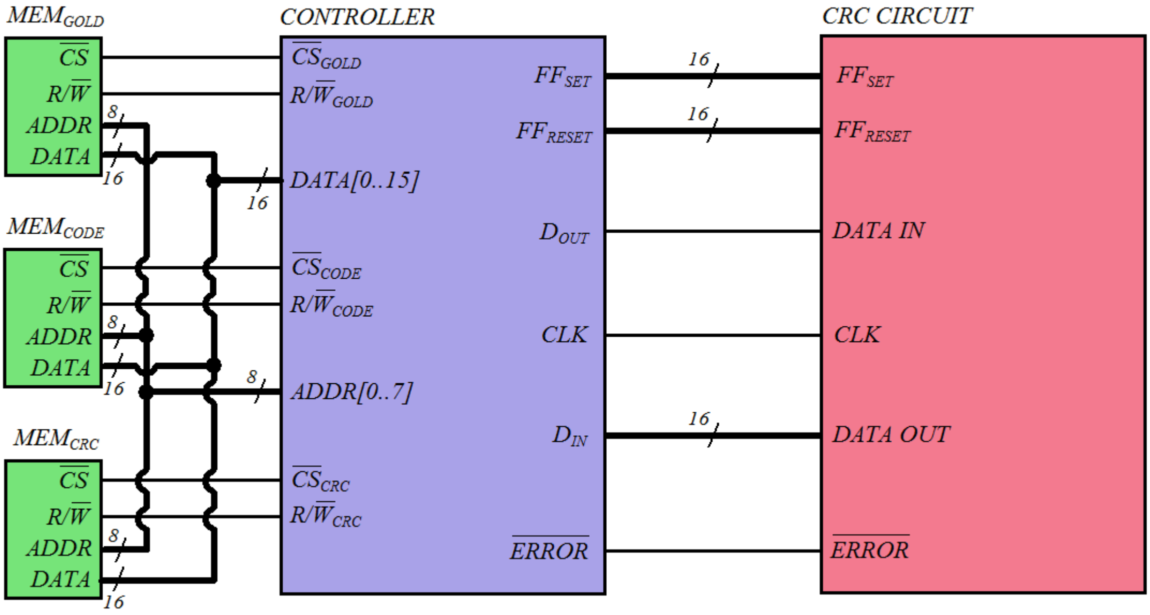


**Fig. 1** Simplified scheme of the scrubber hardware.

resources utilization and 28% increase of the power consumption, but minimally impacts the processing time. On the other hand, the paper shows that time redundancy implies an increase of approximately 3x in the processing time compared to the unprotected system.

Generally, the choice between hardware redundancy and time redundancy depends on the type of application and the type of FPGA device. In fact, different applications may have different requirements in terms of reliability, expressed as failures in time (FIT) per billion hours, and different FPGA technologies can be characterized by different SEUs error rate [7]. For example, Xilinx Virtex-II FPGAs have a soft error rate of 405 FIT/Mb, while more recent devices are characterized by improved reliability (soft error rate of 160 FIT/Mb and 100 FIT/Mb in the case of Virtex-6 and Virtex-7 FPGAs, respectively).

## 4. Proposed Solution

In this Section, we present a low-cost scrubber for SRAM based FPGAs that is capable to detect SEUs affecting the part of the FPGA memory implementing the scrubber itself. We implemented the proposed scrubber in Verilog RTL, and synthesized it by means of the Quartus II tool. We then performed logic level simulations by means of the Icarus Verilog (iVerilog) tool to verify the operation of the proposed scrubber.

Our scrubber employs a dedicated Cyclic Redundancy Check (CRC) generator to verify the correctness of both the words stored in the FPGA memory, as well as the correct behavior of the scrubber itself. This is achieved by executing the following two steps in sequence:

1) To verify the correcteness of each word read from the memory, we first regenerate the checksum from the read word by using the CRC generator. Then, the regenerated checksum is compared with the checksum of the word being checked (that is also stored in the FPGA memory).
2) To verify the correct behavior of the scrubber, we check its ability in detecting incorrect words read from the FPGA memory by purposely inducing bitflips on the words (to emulate the presence of SEUs). Therefore, during this step, for the case of scrubber correct behavior we expect to obtain an error indication at the scrubber output.

A simplified schematic representation of the proposed scrubber is illutrated in Fig. 1. It includes a non-volatile memory ($MEM_{GOLD}$) that is immune to SEUs, where the golden copy of the circuit implemented by the FPGA is stored. The volatile memories $MEM_{CODE}$ and $MEM_{CRC}$ represent, respectively, the part of the FPGA memory where the words of the FPGA (implementing the main circuit) and the corresponding checksums are stored. As a simple case study, the size of such memories has been set to 256 words of 16 bits. At the system boot, the volatile memory $MEM_{CODE}$ is initialized with the data from $MEM_{GOLD}$, while the volatile memory $MEM_{CRC}$ is initialized with the checksums calculated using the CRC circuit. The CRC circuit block is the circuit used to the checksum calculation and error verification, while the controller block generates the control signals required for the operation of the scrubber (i.e. memory operations and initialization, generations of the input signals for the CRC circuit, acquisition of the error signal, etc.). All the blocks of the scrubber in Fig. 1 (except for the non-volatile memory $MEM_{GOLD}$) are implemented inside the FPGA device.

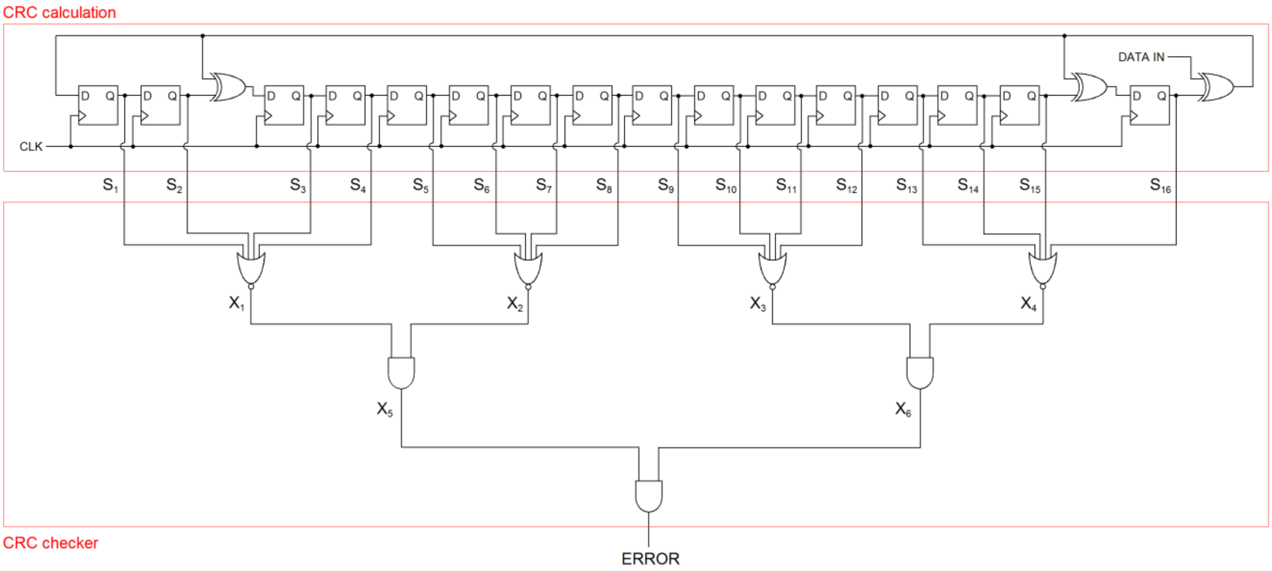


**Fig. 2** Schematic representation of the CRC calculation and CRC checker circuits of the scrubber.

In the following Subsections, we present a possible implementation for the blocks composing the proposed scrubber.

*4.1 CRC generator and checker*

Fig. 2 shows a schematic representation of a 16-bits CRC generator for the considered case of 16-bit words, which represents a realistic example of CRC generators used in modern FPGAs [16]. The 16-bit CRC generator is based on a Linear Feedback Shift Register (LFSR) with characteristic polynomial given by:

$$CRC_{16} = X^{16} + X^{15} + X^{2} + 1 \qquad (1)$$

In the FPGA configuration phase, the 16 flip-flops of the CRC checker are reset and the 16-bit word of the FPGA is serially given as input (most significant bit first) at the DATA IN line. After 16 clock cycles the DATA OUT array ($S_{16}S_{15}$…..$S_1$) contains the checksum for the corresponding word. The obtained checksum is stored in the FPGA volatile memory $\mathrm{MEM_{CRC}}$ to be used later during the scrubbing of the FPGA in the field.

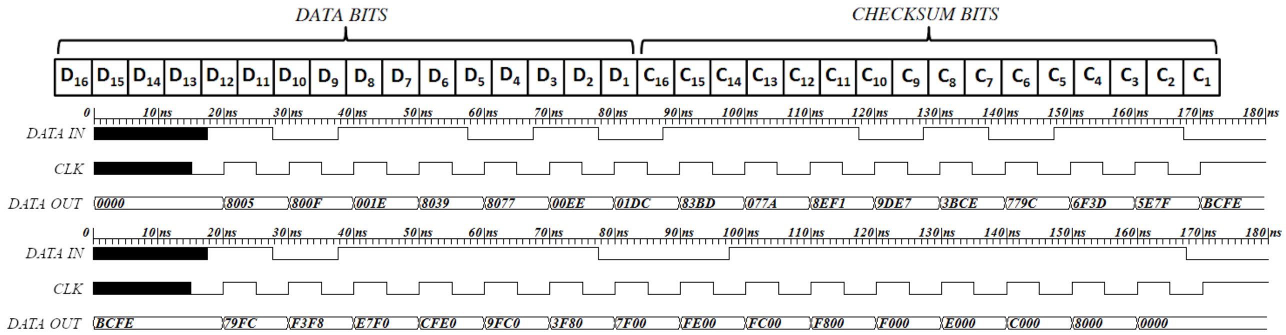


**Fig. 3** Waveforms for the scrubber control signals during the checksum calculation and test of a code word.

In particular, during scribbing, to verify the correcteness of the words read from the FPGA memory, the following steps are carried out:

- The 16 flip-flops of the CRC generator circuit are reset.
- A 32-bit word, obtained by appending the word under test from $\text{MEM}_{\text{CODE}}$ (most significant word) and the checksum from $\text{MEM}_{\text{CRC}}$ (least significant word), is fed as input (most significant bit first) at the DATA IN line (this step is executed in 32 clock cycles). After the first 16 clock cycles the DATA OUT array ($S_{16}S_{15}\ldots..S_1$) contains the recalculated checksum of the word being verified. In the second 16 clock cycles the checksum from $\text{MEM}_{\text{CRC}}$ is fed as input at the DATA IN line.
- After applying 32 clock cycles, the output of the CRC generator DATA OUT ($S_{16}S_{15}\ldots..S_1$) contains all 0s (00….0) only if the recalculated checksum is equal to the one read from the volatile memory $\text{MEM}_{\text{CRC}}$.

As can be seen from Fig. 2, the 16-bit CRC checker is implemented by a combinational circuit composed of 3 AND and 4 NOR gates whose output (ERROR) is equal to 1 if no errors are present in the tested code word, or is equal to 0 otherwise. The “CRC generator and checker” are shown in Fig. 2.

The working principle of the CRC circuit in Fig. 2 is illustrated in Fig. 3, for the case of absence of errors in the word read from memory. The waveforms for the signals DATA IN, CLK and DATA

OUT are shown for the case of the word #B5D6 and the checksum #BCFE. The upper waveform referrs to the checksum recalculation, while the lower waveform refers to the checksum verification. More in details, the following steps are illustrated in Fig. 3:

- The flip-flops of the CRC generator are reset (Data Out initially equal to 00….0).
- The code word #B5D6 (1011010111010110) from $\text{MEM}_{\text{CODE}}$ is fed as input at the DATA IN line. After 16 clock cycles the recalculated checksum #BCFE (1011110011111110) is present on DATA OUT ($S_{16}S_{15}$…..$S_1$).
- The checksum from $\text{MEM}_{\text{CRC}}$ is fed as input at the DATA IN line. Since this value (#BCFE) is the same as the value calculated during the first 16 clock cycles, after the second 16 clock cycles DATA OUT is equal to (00…..0) and the output of the CRC checker is equal to 1 (no error detected).

This approach is capable to detect errors due to occurrence of SEUs in the FPGA memory only if the part of the FPGA memory implementing the CRC generator and checker itself is error free (i.e., the CRC generator and checker is correctly configured in the FPGA). However, if a SEU induces an error in the part of the FPGA memory implementing the CRC generator and checker, the reliability of the scrubber may be seriously compromised. In fact, as a simple example, the SEU can make the output of the scrubber constant (ERROR=1, i.e. no error detected) during the FPGA normal operation, so it is not possible to detect SEUs affecting the FPGA memory implementing the main circuit. As clarified before, this critical situation is avoided by our self-checking scrubber based on time redundancy.

### *4.2 Proposed Scrubbing Strategy*

The algorithm of our novel low-cost self-checking scrubber strategy, based on time redundancy, is illustrated in the flow chart in Fig. 4. In the first phase "word test phase", the word under test and the corresponding checksum are read from memory (i.e., from $\text{MEM}_{\text{CODE}}$ and $\text{MEM}_{\text{CRC}}$ respectively) and given as input to the CRC circuit (the word and the checksum are first appended to obtain a word of 32 bits, as described in previous Subsection). Then we apply 32 clock cycles,

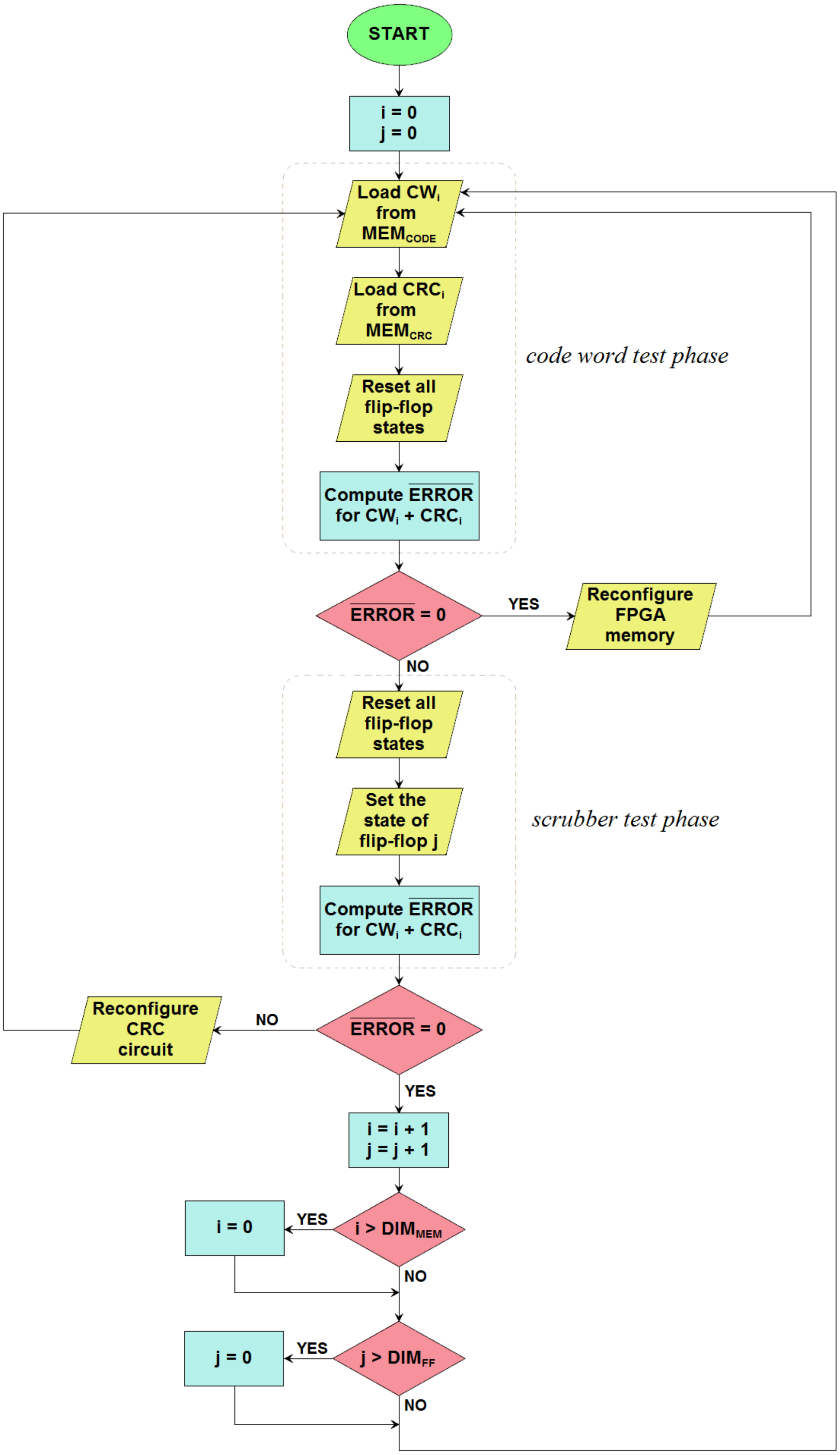

START
i = 0
j = 0
Load CW$_i$ from MEM$_{CODE}$
Load CRC$_i$ from MEM$_{CRC}$
Reset all flip-flop states
Compute $\overline{ERROR}$ for CW$_i$ + CRC$_i$
code word test phase
$\overline{ERROR}$ = 0
YES
Reconfigure FPGA memory
NO
Reset all flip-flop states
Set the state of flip-flop j
Compute $\overline{ERROR}$ for CW$_i$ + CRC$_i$
scrubber test phase
$\overline{ERROR}$ = 0
NO
Reconfigure CRC circuit
YES
i = i + 1
j = j + 1
i > DIM$_{MEM}$
YES
i = 0
NO
j > DIM$_{FF}$
YES
j = 0
NO

**Fig. 4** Flow-chart of the algorithm of the test phase implemented in the scrubber controller.

and then, if we obtain an error indication at the output of the CRC checker, the FPGA memory is reconfigured using the data in $\mathrm{MEM}_{\mathrm{GOLD}}$. Otherwise, the “scrubber test phase” begins to verify its correct operation. As described at the beginning of this Section, in the “scrubber test phase” the CRC circuit is checked by emulating an error in the 32-bit word given to the CRC checker, so that in case of correct behaviour we expect to obtain an error indication at the output of the CRC checker. To achieve this goal we induce a bit flip in the 32-bit word (obtained by appending the checksum to the memory word) before it is given to the CRC circuit. In particular, 15 flip-flops out of the 16 flip-flops of the CRC checker are reset, while one flip-flop is set, in order to modify the initial state of the CRC generator. This way, after 32 clock cycles, we expect to obtain a logic 0 (presence of error) at the ouput of the CRC checker for the case of scrubber correct behaviour. Otherwise, if after the 32 clock cycles we obtain a logic 1 at the output of the CRC checker, it means that the scrubber is unable to detect errors in the FPGA memory (words) and the scrubber circuit must be reconfigured.

As an example, Figs. 5, 6, 7 and 8 report some waveforms of the signals during the two steps of the algorithm presented in Fig. 4.

In particular, Fig. 5 reports the waveforms regarding the FPGA memory read operation of the word #B5D6 and the checksum #BCFE (both read at address 69 of $\mathrm{MEM}_{\mathrm{CODE}}$ and $\mathrm{MEM}_{\mathrm{CRC}}$, respectively), that are stored in the registers Codeword and CRCword, respectively.

The values of the registers Codeword and CRCword are used in the next phase of our approach, when the correctness of the word read from memory and the scrubber behaviour are verified.

In addition, Fig. 6 illustrates how the word under test and the corresponding checksum are checked (cases a, b and c), as well as how the functionality of the scrubber is verified (case d).

In Fig. 6a, the 32-bit word #B5D6BCFE (obtained by appending the content of the register CRCword to the register Codeword) is given as input to the scrubber after the 16 flip-flops of the CRC checker are reset. As expected, after 32 clock cycles, all bits of DATA OUT are equal to 0, thus the CRC checker output is 1 (i.e., no error detected).

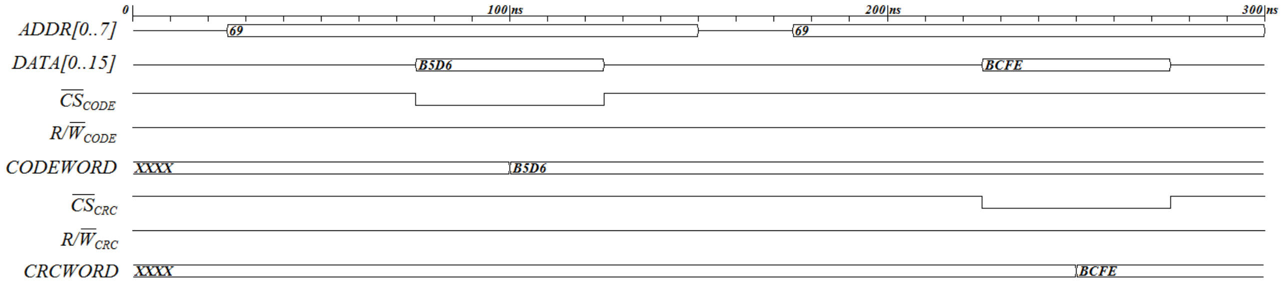
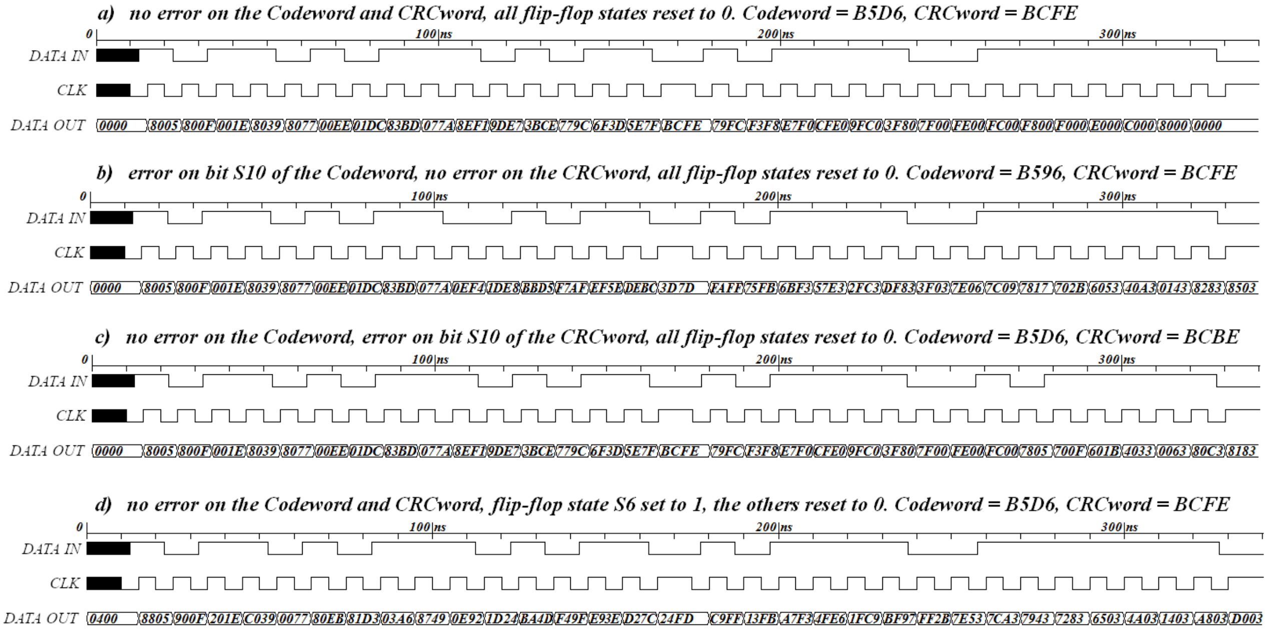


**Fig. 5** Waveforms for the memory read operation.


a) no error on the Codeword and CRCword, all flip-flop states reset to 0. Codeword = B5D6, CRCword = BCFE

b) error on bit S10 of the Codeword, no error on the CRCword, all flip-flop states reset to 0. Codeword = B596, CRCword = BCFE

c) no error on the Codeword, error on bit S10 of the CRCword, all flip-flop states reset to 0. Codeword = B5D6, CRCword = BCBE

d) no error on the Codeword and CRCword, flip-flop state S6 set to 1, the others reset to 0. Codeword = B5D6, CRCword = BCFE

0
100|ns
200|ns
300|ns
DATA IN
CLK
DATA OUT
0400 8805 900F 201E C039 0077 80EB 81D3 03A6 8749 0E92 1D24 BA4D F49F E93E D27C 24FD C9FF 13FB A7F3 4FE6 1FC9 BF97 FF2B 7E53 7CA3 7943 7283 6503 4A03 1403 A803 D003

**Fig. 6** Waveforms for the step of word checking and verification of the scrubber functionality.

On the other hand, Fig. 6b, shows the case in which one bit of the word is altered and the 32-bit words #B596BCFE is given to the scrubber. As can be seen, for this case, after 32 clock cycles, some bits of the signal DATA OUT are 1, thus the CRC checker output is 0 (i.e., error indicaton). Therefore, in this case the MEM$_{\text{CODE}}$ word must be reconfigured with the value on MEM$_{\text{GOLD}}$ as shown in Fig. 7, and the checksum calculated again and stored in MEM$_{\text{CRC}}$, as shown in Fig. 8. Similarly, Fig. 6c reports the case in which one bit of the checksum is altered and the 32-bit word #B5D6BCBE is given to the scrubber input. Also in this case, after 32 clock cycles, some bits of the signal DATA OUT are 1, thus the CRC checker output is 0 (i.e., error indicaton). As in the previous

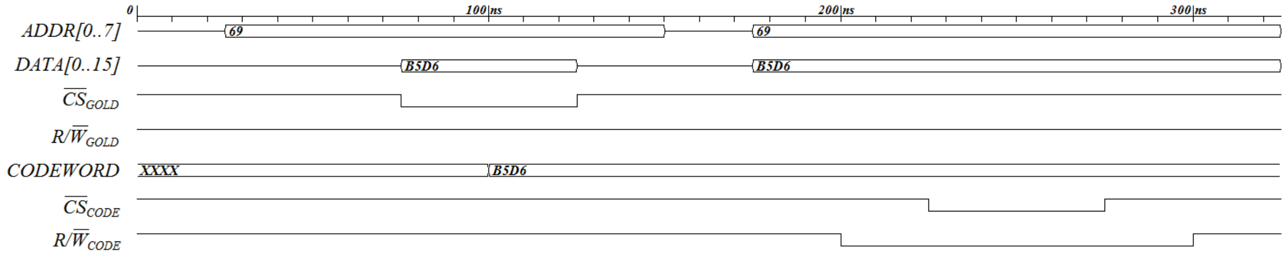


**Fig. 7** Waveforms during the reconfiguration of the FPGA memory containing an erroneous word, after it is detected by our scheme.

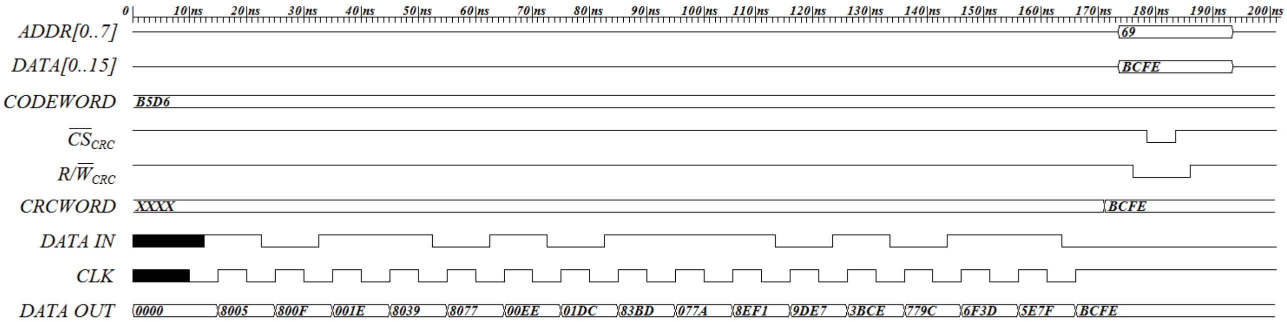


**Fig. 8** Waveforms for the checksum calculation and write operation of the checksum in memory.

case, the word on MEM$_{\text{CODE}}$ must be reconfigured with the value on MEM$_{\text{GOLD}}$ (Fig. 7) and the checksum calculated again and stored in MEM$_{\text{CRC}}$ (Fig. 8).

Finally, Fig. 6d, shows the case where the scrubber functionality is verified. The same checksum in Fig. 6a corresponding to the correct word is used, and the 32 bit word #B5D6BCFE is given as input to the scrubber. In this case, however, the state of the flip-flop S6 is set instead of reset, while all other flip-flops are reset (initially Data Out is set to #0400). As expected, after 32 clock cycles, DATA OUT has the value #D003, and the output of the CRC checker is 0 (indicating the presence of an error), indicating that the CRC checker is working properly, thus being able to detect errors on words read from the FPGA memory.

Fig. 7 reports the waveforms during the reconfiguration of the FPGA memory containing an erroneous word, after it is detected by our scheme. In particular, the correct word #B5D6 is read at the address #69 of MEM$_{\text{GOLD}}$ and written to the same address of MEM$_{\text{CODE}}$. In order to complete the reconfiguration process, the checksum (#BCFE) of the reconfigured word must be calculated and stored at the corresponding address (#69) in MEM$_{CRC}$. The waveforms of this latter operation are reported in Fig. 8.

### *4.3 Costs of the proposed scheme*

We have estimated the cost of our proposed scrubber in terms of time overhead, resource utilization and power consumption. In order to estimate such costs, the proposed FPGA scrubber has been implemented in Verilog and synthesized on a real FPGA device (Arria II GX EP2AGX45CU17I3) using the Quartus II (64 bit version) tool. For our evaluations, we have considerd as a realistic example a clock frequency of 100 MHz (clock cycle period of 10 ns).

Let us first report the cost in terms of time overhead of the proposed solution. The time required to read a word (16 bit) from the volatile memory (MEM$_{CODE}$ or MEM$_{CRC}$) is 100 ns, while the time required to write a word in such a memory is 150 ns.

As for our algorithm presented in Fig. 4, it first verifies the correctness of the word read from the FPGA, and then the ability of the scrubber to detect incorrect words.

As for the time required to verify the correctness of the word read from the FPGA memory, it is given by: 1) the time required to load the word from MEM$_{CODE}$ and the checksum from MEM$_{CRC}$ (100 ns each), plus 2) one clock period (10 ns) to reset the 16 flip-flops of the CRC generator, plus 3) 32 clock cycles (320 ns) to generate the error/no error indication at the CRC checker output. Therefore, the time required by our scheme to verify the correctness of a word read from the FPGA memory is:

$$T_{CODEWORD\_TEST} = 100ns + 100ns + 10ns + 320ns = 530ns \qquad (2)$$

Similarly, the time required by our scheme to detect the ability of the scrubber in detecting incorrect word is given by: 1) 1 clock period (10 ns) to reset the 16 flip-flops of the CRC generator, plus 2) one clock period to set the state of one flip-flop (10 ns), plus 3) 32 clock cycles to generate the error/no error indication at the CRC checker output. Therefore, the time required for the scrubber in this phase is:

$$T_{SCRUBBER\_TEST} = 10ns + 10ns + 320ns = 340ns \qquad (3)$$

The reconfiguration of a word in the FPGA configuration memory requires a read operation from $\text{MEM}_{\text{GOLD}}$ (100ns), a write operation to $\text{MEM}_{\text{CODE}}$ (150 ns), the calculation of the correct checksum (16 clock cycles for an operation time of 160 ns) and to write the checksum to $\text{MEM}_{\text{CRC}}$ (150 ns). Thus, the reconfiguration of a word in the FPGA configuration memory requires:

$$T_{CODEWORD\_FPGA\_RECONFIGURATION} = 100ns + 150ns + 160ns + 150ns = 560ns \qquad (4)$$

The reconfiguration of the CRC circuit is, of course, the most time consuming operation since it requires the reconfiguration of multiple code words in the FPGA used to configure the CRC circuit. Assuming the CRC circuit uses 25 words in the FPGA configuration memory, the required time is:

$$T_{CRC\_CIRCUIT\_RECONFIGURATION} = 560ns \times 25 = 14\mu s \qquad (5)$$

The total time required to perform a single loop of the algorithm of Fig. 4, in the case of absence of errors, is the sum of the time required for the word test phase (530 ns), the time required for the scrubber test phase (340 ns) and 2 clock cycle to increase the registers for the variables i and j (20 ns). Thus the total time required for a single loop of the algorithm is 890 ns. As an example, for the case of a configuration memory size of 256 words of 16 bits (the case study discussed in this paper) the total scrubbing time in absence of errors is 227.84 µs, that corresponds to a total scrubbing time of 58.33 ms for every Mbit of configuration memory. The overhead introduced by the scrubber test phase is 38.2% of the total scrubbing time. However, this overhead can be reduced by performing the scrubber test phase only 1 out of n loops (i.e., the scubber behavior is verified after *n* words of FPGA memory are scrubbed). For example, in the case of n=2 the time overhead introduced by the scrubber test phase is 23.6% of the total time, in the case of n=4 is 13.4% of the total time and in the case of n=8 is 7.2% of the total time. Thus, depending on the SEU error rate of the particular application, the time overhead of the scrubber test phase can be significantly reduced with a trade off between system performance and reliability. However, the additional time overhead required by our strategy over the original scrubber does not affect the reliability of the FPGA. In fact, as reported in [7], FPGA devices (like the Xilinx Virtex-II) may be characterized by a soft error rate of approximately 405 FIT/Mb, that is 405 soft errors in a billion hours of operation per Mbit of memory. Therefore, the soft error rate is low enough to guarantee the absence of multiple SEUs in the time required by our strategy, which is equal to 58.33 ms per 1 Mbit of configuration memory.

Regarding the resource utilization and the power consumption, such costs have been estimated considering the scheme in Fig. 1. As expected, the FPGA memory ($MEM_{CODE}$ and $MEM_{CRC}$) is the most demanding in terms of resource utilization and power consumption. Each 256 words of memory is responsible for a 20% logic utilization of the entire FPGA device, with 5797 combinational ALUTs out of 36100, and a power consumption of 19.83 mW. On the contrary, the CRC generator and checker circuit is responsible for less than 1% logic utilization of the entire FPGA device, with 84 combinational ALUTs out of 36100 and 16 registers, and a power consumption of 2.08 mW. Similarly, the control circuit is responsible for less than 1% logic utilization of the entire FPGA device, with 109 combinational ALUTs out of 36100 and 33 registers, and a power consumption of 2.47 mW. Thus, the resource utilization for the CRC generator and checker and the control circuits represents only 4.17% of the resource utilization of the FPGA memory and the power consumption for the CRC generator and checker and the control circuits represents only 22.9% of the power consumption of the FPGA memory.

Overall, the proposed strategy based on time redundancy is characterized by a very low resource occupation while still maintaining the total scrubbing time at acceptable levels. Reversely, solutions based on hardware redundancy, although very efficient in terms of time overhead (i.e. processing is carried out in parallel) are extremely expensive in terms of resource utilization. For example, the TMR approach proposed by Zhang et al. in 2018 that exploits the triplication of the circuit, results in a 300% resource increase [24]. The TMR approach on the Xilinx Kintex-7 FPGA proposed by Sielewicz et al. in 2017 triplicates not only the circuit but also the voter, resulting in an area increase > 300% [25]. An even more expensive solution in terms of resource has been proposed by Shaker et al. in 2020 where the circuit is replicated five times, resulting in a 500% increase of resource utilization [28]. In comparison, the proposed strategy is much more area efficient, requiring an increase of only 4.17% of resource utilization. Like in any time redundancy strategy, the low increase in resource utilization is balanced by a higher time overhead. The time redundancy strategy proposed by Villa et al. in 2019 is characterized by a 107% time overhead (compared to the 38.2% required by our strategy) and 93.26% area overhead (compared to the 4.17% of our strategy) [32]. The detection scheme proposed by Bahramali et al. in 2011 is characterized by an increase of resource utilization between 30% and 58% (compared to the 4.17% of our strategy) [33]. The time redundancy strategy proposed by Ibrahim et al. in 2013 is characterized by a 300% time overhead (compared to the 38.2% of our strategy) [34]. Moreover, differently from other strategies in literature, our detection strategy can also detect errors in the scrubber circuit.

## 5. Conclusions

In this paper a novel strategy to detect SEUs induced faults in SRAM based FPGAs is presented. The proposed technique is based on time redundancy and allows to detect faults both in FPGA configuration memory and the scrubber with negligible area overhead if compared with the unmitigated approach.

The working principle of the proposed strategy is to force the scrubber output to assume both possible value (presence or absence of error), thus testing the scrubber functionality to detect errors in the code word under test.

The performance of the proposed strategy has been evaluated in terms of time overhead, resource utilization and power consumption by synthesizing the circuit on a real FPGA device (Arria II GX EP2AGX45CU17I3). The results have shown how the scrubber  functionality test phase introduces a 38.2% time overhead over the unprotected design but this time overhead can be significantly lowered by decreasing the frequency of the scrubber test phase. The resource utilization overhead is negligible (4.17%) and the power consumption overhead is relatively small (22.9%) if compared to the original unmitigated scrubber.

In future works, the proposed stategy based on time redundancy to detect SEUs in SRAM-based FPGAs will be implemented on different FPGA devices from different producers to evaluate the performance differences based on different hardware. The system performance will then be evaluated by laboratory measurements on real hardware and compared with the standard mitigation techniques (TMR, DWC, etc.) in terms of execution times, occupation area and power consumption.

## References

[1] Bouras M., Berbia H., Nasser T., On Modeling and Fault Tolerance of NanoSatellite Attitude Control System. *In: El Oualkadi A., Choubani F., El Moussati A. (eds) Proceedings of the Mediterranean Conference on Information & Communication Technologies 2015*. Lecture Notes in Electrical Engineering, 380, (2016), Springer, Cham. https://doi.org/10.1007/978-3-319-30301-7_43.

[2] L.D. van Harten, M. Mousavi, R. Jordans, H.R. Pourshaghaghi, Determining the necessity of fault tolerance techniques in FPGA devices for space missions. *Microprocessors and Microsystems*, *63*, (2018), 1-10. https://doi.org/10.1016/j.micpro.2018.08.001.

[3] G.I. Alkady, R.M. Daoud, H.H. Amer, M.Y. ElSalamouny, I. Adly, Failures in fault-tolerant FPGA-based controllers–A case study. *Microprocessors and Microsystems*, *64*, (2019), 178-184. https://doi.org/10.1016/j.micpro.2018.11.003.

[4] J.-M. Yang, S. W. Kwak, Corrective control for transient faults with application to configuration controllers. *IET Control Theory Appl.*, 9 (8), (2015). https://doi.org/10.1049/iet-cta.2014.0532.

[5] M. Ebrahimi, P. M. B. Rao, R. Seyyedi, M. B. Tahoori, Low-Cost Multiple Bit Upset Correction in SRAM-Based FPGA Configuration Frames. *IEEE Trans. Very Large Scale Integr. Syst.*, 24 (3), (2016), 932–943. https://doi.org/10.1109/TVLSI.2015.2425653.

[6] X. Li, H. Lou, Z. Jin, A Fault-tolerant Method of SRAM FPGA Based on Processor Scrubbing. *IEEE 5th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, 5, (2021), 1024-1028. https://doi.org/10.1109/IAEAC50856.2021.9390706.

[7] Xilinx and Inc, Mitigating Single-Event Upsets WP395 (v1.1) May 19, 2015. https://www.xilinx.com/support/documentation/white_papers/wp395-Mitigating-SEUs.pdf (accessed 23 June 2021).

[8] Xilinx and Inc, 7 Series FPGAs Configuration User Guide UG470 (v 1.13.1) August 20, 2018. https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf (accessed 20 June 2021).

[9] I. Herrera-Alzu, M. López-Vallejo, Design techniques for Xilinx Virtex FPGA configuration memory scrubbers. *IEEE Trans. Nucl. Sci.*, 60 (1), (2013). https://doi.org/10.1109/TNS.2012.2231881.

[10] U. Legat, A. Biasizzo, F. Novak, SEU recovery mechanism for SRAM-Based FPGAs. *IEEE Trans. Nucl. Sci.*, 59 (5), (2012), 2562–2571. https://doi.org/10.1109/TNS.2012.2211617.

[11] A. Ebrahim, T. Arslan, X. Iturbe, On enhancing the reliability of internal configuration controllers in FPGAs. *Proceedings of the 2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2014)*, (2014), 83–88. https://doi.org/10.1109/AHS.2014.6880162.

[12] M. Wirthlin, A. Harding, Hybrid Configuration Scrubbing for Xilinx 7-Series FPGAs. *FPGAs and Parallel Architectures for Aerospace Applications, Cham: Springer International Publishing*, (2016), 91–101. https://doi.org/10.1007/978-3-319-14352-1_7.

[13] T.S. Nidhin, A. Bhattacharyya, R.P. Behera, T. Jayanthi, A review on SEU mitigation techniques for FPGA configuration memory. *IETE Technical Review*, 35 (2), (2018), 157-168. https://doi.org/10.1080/02564602.2016.1265905.

[14] S. Fouad, F. Ghaffari, M. E. A. Benkhelifa, B. Granado, Reliability assessment of backward error recovery for SRAM-based FPGAs. *9th International Design and Test Symposium (IDT)*, (2014), 248–252. https://doi.org/10.1109/IDT.2014.7038622.

[15] F. Sahraoui, F. Ghaffari, M. E. Amine Benkhelifa, B. Granado, An efficient BER-based reliability method for SRAM-based FPGA. *2013 8th IEEE Design and Test Symposium (IDT),* (2013). https://doi.org/10.1109/IDT.2013.6727129.

[16] T. Bates, C.P. Bridges, Single event mitigation for Xilinx 7-series FPGAs, *IEEE Aerospace Conference*, (2018), 1-12. https://doi.org/10.1109/AERO.2018.8396520.

[17] F. Brosser, E. Milh, V. Geijer, P. Larsson-Edefors, Assessing scrubbing techniques for Xilinx SRAM-based FPGAs in space applications. *Proceedings of the 2014 International Conference on Field-Programmable Technology*, (2014). https://doi.org/10.1109/FPT.2014.7082803.

[18] Xilinx and Inc, Virtex-6 FPGA Configuration User Guide UG360 (v3.9) 18 November 2015, https://www.xilinx.com/support/documentation/user_guides/ug360.pdf (accessed 22 June 2021).

[19] D. Rossi, M. Omaña, C. Metra, Transient Fault and Soft Error On-die Monitoring Scheme, *IEEE 25th International Symposium on Defect and Fault Tolerance in VLSI Systems*, (2010), 391-398, https://doi.org/10.1109/DFT.2010.53.

[20] J. Tonfat, F.G. Kastensmidt, R.A. Reis, Energy efficient frame-level redundancy scrubbing technique for SRAM-based FPGAs. *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, (2015), 1-8. https://doi.org/10.1109/AHS.2015.7231160.

[21] R. Glein, F. Rittner, A. Heuberger, Adaptive single-event effect mitigation for dependable processing systems based on FPGAs. *Microprocessors and Microsystems*, *59*, (2018), 46-56. https://doi.org/10.1016/j.micpro.2018.03.004.

[22] N.T. Nguyen, D. Agiakatsikas, Z. Zhao, T. Wu, E. Cetin, O. Diessel, L. Gong, Reconfiguration Control Networks for FPGA-based TMR systems with modular error recovery. *Microprocessors and Microsystems*, *60*, (2018), 86-95. https://doi.org/10.1016/j.micpro.2018.04.006.

[23] F. Smith, J. Omolo, Experimental verification of the effectiveness of a new circuit to mitigate single event upsets in a Xilinx Artix-7 field programmable gate array. *Microprocessors and Microsystems*, *79*, (2020), 103327. https://doi.org/10.1016/j.micpro.2020.103327.

[24] R. S. Zhang, L. Y. Xiao, X. B. Cao, J. Li, J. Q. Li, L. Z. Li, A fast scrubbing method based on triple modular redundancy for sram-based fpgas. *14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, (2018), 1-3. https://doi.org/10.1109/ICSICT.2018.8565046.

[25] K. M. Sielewicz, G. A. Rinella, M. Bonora, P. Giubilato, M. Lupi, M. J. Rossewij, T. Vanat, Experimental methods and results for the evaluation of triple modular redundancy SEU mitigation techniques with the Xilinx Kintex-7 FPGA. *IEEE Radiation Effects Data Workshop (REDW)*, (2017), 1-7. https://doi.org/10.1109/NSREC.2017.8115451.

[26] R. Giordano, D. Barbieri, S. Perrella, R. Catalano, G. Milluzzo, Configuration self-repair in Xilinx FPGAs. *IEEE Transactions on Nuclear Science*, *65* (10), (2018), 2691-2698. https://doi.org/10.1109/TNS.2018.2868992.

[27] A. E. Wilson, S. Larsen, C. Wilson, C. Thurlow, M. Wirthlin, Neutron Radiation Testing of a TMR VexRiscv Soft Processor on SRAM-Based FPGAs. *IEEE Transactions on Nuclear Science*, *68* (5), (2021), 1054-1060. https://doi.org/10.1109/TNS.2021.3068835.

[28] M. N. Shaker, A. Hussien, G. I. Alkady, H. H. Amer, I. Adly, FPGA-Based Reliable Fault Secure Design for Protection against Single and Multiple Soft Errors. *Electronics*, *9* (12), (2020), 2064. https://doi.org/10.3390/electronics9122064.

[29] I. B. M. Matsuo, L. Zhao, W. J. Lee, A Dual Modular Redundancy Scheme for CPU–FPGA Platform-Based Systems. *IEEE Transactions on Industry Applications*, *54* (6), (2018), 5621-5629. https://doi.org/10.1109/TIA.2018.2859386.

[30] A. Keller, M. Wirthlin, Partial triple modular redundancy: low-cost resilience for FPGAs in space environments. *Orbital ATK Conference Center*, (2018). https://digitalcommons.usu.edu/spacegrant/2018/Session_one/2/ (accessed 5 July 2021).

[31] C. M. Fuchs, Fault-tolerant satellite computing with modern semiconductors (Doctoral dissertation, Leiden University), (2019). https://scholarlypublications.universiteitleiden.nl/handle/1887/82454 (accessed 5 July 2021).

[32] P. R. Villa, R. Travessini, R. C. Goerl, F. L. Vargas, E. A. Bezerra, Fault tolerant soft-core processor architecture based on temporal redundancy. *Journal of Electronic Testing*, *35* (1), (2019), 9-27. https://doi.org/10.1007/s10836-019-05778-z.

[33] M. Bahramali, J. Jiang, A. Reyhani-Masoleh, A fault detection scheme for the FPGA implementation of SHA-1 and SHA-512 round computations. *Journal of Electronic Testing*, *27* (4), (2011), 517-530. https://doi.org/10.1007/s10836-011-5237-4.

[34] M. M. Ibrahim, K. Asami, M. Cho, Time and Space Redundancy Fault Tolerance Trade-offs for FPGA Based Single and Multicore Designs. *Transactions of the Japan Society for Aeronautical*

*and Space Sciences, Aerospace Technology Japan*, *12* (29), (2014), 15-24. https://doi.org/10.2322/tastj.12.Pj_15.