Efficient Protection of the Register File in Soft-Processors Implemented on Xilinx FPGAs

Alexis Ramos, Anees Ullah, Pedro Reviriego, and Juan Antonio Maestro

Abstract—Soft-processors implemented on SRAM-based FPGAs are increasingly being adopted in on-board computing for space and avionics applications due to their flexibility and ease of integration. However, efficient component-level protection techniques for these processors against radiation-induced upsets are necessary otherwise as system failures could manifest. A register file is one of the critical structures that stores vital information the processor uses related to user computations and program execution. In this paper, we present a fault tolerance technique for the register file of a microprocessor implemented in Xilinx SRAM-based FPGAs. The proposed scheme leverages the inherent implementation redundancy created by the FPGA design automation tools when mapping the register file to on-chip distributed memory. A parity-based error detection and switching logic are added for fault masking against single-bit errors. The proposed scheme has been implemented and evaluated in lowRISC, a RISC-V ISA soft-processor implementation. The effectiveness of the proposed scheme was tested using fault injection. The fault masking overhead required in terms of FPGA resources was much lower than a traditional Triple Modular Redundancy protection. Therefore, the proposed scheme is an interesting option to protect the register file of soft processors that are implemented in Xilinx FPGAs.

Index Terms—Fault injection, fault tolerance, internal reconfiguration, reliability, soft-processor

1 Introduction

The successful deployment of satellite missions based on commercial SRAM-based FPGAs has not only shown the feasibility of their usage but also the sensitivity-levels of the reconfigurable fabric resources to radiation effects [1], [2]. Therefore, motivating a resource-level analysis of radiation effects and its consequences along with design-mitigation approaches at a modular-level to enhance system-level reliability. This bottom-up approach to reliable electronic systems design needs to consider the heterogeneous composability of the state-of-the-art reconfigurable fabrics and the mapping of computational structures to it by the design automation tools. Applying such bottom-up design and test approach to soft-processors implemented on SRAM-based FPGAs is an interesting research direction [3]. In particular, combined with the recent trends towards a standard open Instruction Set Architectures (ISA) [4], reliability in soft-processors could be investigated and mitigated from the circuit and system level to architectural level and from operating system (OS) to application layers. Reliability techniques and re-configurability approaches could be integrated along the way to improve the fault-tolerance, fault diagnosis and recovery aspects.

One of the most critical structures to consider for radiation hardening is the register file when soft-processors are implemented on SRAM-based FPGAs. Since the register file stores the user computation intermediate results and the critical program flow execution information, radiation-induced upsets could corrupt these temporal contents. Consequently, the system-level effects could produce a silent data corruption (SDC) [5], a wrong branch address or an output failure. From the aspect of mapping to FPGA fabric resources, the potential on-chip targets for a register file structure could be Block RAMs (BRAM) or Distributed RAMs [6]. This mapping is determined by the design automation tools. However, the user can constrain it to the desired memory resource. Since recent in-orbit experiments have shown that BRAM interconnect is particularly sensitive to radiation-induced upsets [2], distributed RAMs could be a more reliable option for design mapping.

In this work, we propose a novel protection technique for a dual port register file structure leveraging the inherent duplication redundancy inferred by the FPGA design automation tools when mapping to the on-chip distributed memory. This inherent duplication redundancy is introduced by the design tools as an artifact to support the simultaneous read out of two register contents usually required by arithmetic operations in processors. Exploiting it for protection could considerably reduce the cost in terms of hardware overhead compared to Triplication [7] or even more complex Error Detection and Correction (EDAC) codes [8]. The proposed protection technique utilizes this redundancy with parity-based error detection and a multiplexing logic to provide feedforward error-free results. Thus, providing fault tolerance capabilities at considerably lower cost than TMR. The feasibility of the register file protection technique is evaluated through a lowRISC implementation [9] of the RISC-V ISA mapped onto a 28-nm Artix-7 FPGA. A fine-grain fault injection campaign in the register file contents is used to test the proposed approach.

The rest of the paper is organized as follows. Section 2 describes previous work. Section 3 presents how the lowRISC register file is implemented in the FPGA and the proposed solution to protect it. Section 4 describes the experimental setup, the fault injection campaign, the results and their analysis. Section 5 presents the conclusions.

2 Related Work

Protection techniques for critical components of processors, like register files, are increasingly important due to their potential use in safety-critical systems. We can broadly classify them into design level and implementation level techniques. The first group are proposals which could be used for ASIC and FPGAs. The second group is focused on solutions that utilize the inherent FPGA resources to protect the information storage modules.

Design level techniques are those traditionally based on redundancy, for example, Double Modular Redundancy (DMR), Triple Modular Redundancy (TMR), and Error Correction Codes (ECC). Neither of these options are perfect: on one hand TMR has a low latency cost but with a large area overhead; on the other hand, ECC codes are good at saving area but increase the latency of the system [10]. For these reasons, researchers try to modify one of these traditional techniques into a more efficient solution. An example of this redundancy is the solution presented in [11], which implements a hardware module that works as a temporal data storage module, saving the most recently used register data. The output from the register file is compared with one of the copy. If there is a mismatch, the pipeline is stalled and a Cyclic Redundancy Check (CRC) error is raised to choose which one of the values is correct. The problem with this proposal is that the microprocessor is stalled until the error is corrected. Another redundancy based technique is presented in [12]. It saves a copy of the used registers in spare registers. At read stage, if a register is retrieved with

• *The authors are with ARIES Research Center, Escuela Politécnica Superior, Universidad Antonio de Nebrija, C. Pirineos 55, Madrid 28015, Spain. E-mail: {aramosam, aullah, previrie, jmaestro}@nebrija.es.*

Manuscript received 3 Apr. 2017; revised 11 July 2017; accepted 2 Aug. 2017. Date of publication 10 Aug. 2017; date of current version 22 Jan. 2018.
(Corresponding author: Alexis Ramos.)
Recommended for acceptance by D. Kagaris.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2737996

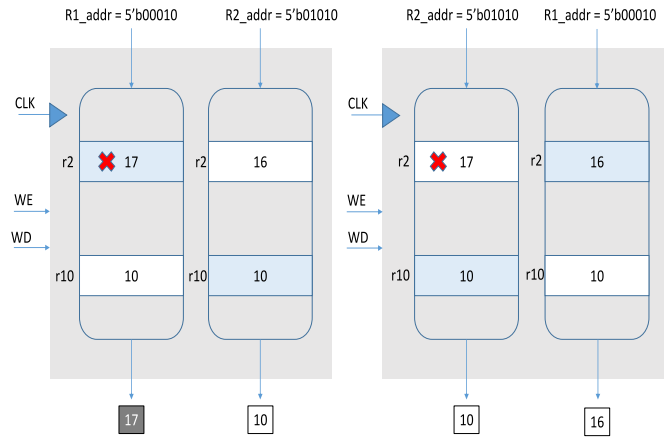

Fig. 1. Fault masking using switching of register file read addresses.

incorrect information and detected (using for example parity check) the correct data is read from the alternative copy. The advantage of this solution is that it has little impact on performance when the register file is not fully utilized. The weak point appears when the register file is at full capacity and there are not enough spare registers to save the copies. In that situation, the last used registers are overwritten to save the new data. Because less registers are available, the performance of the microprocessor decreases. Related with ECC codes, the authors in [13] propose to use the upper unused bits of the register to store the SEC Hamming Code [14] of the word inside it. This approach works fine in cases where there is enough available bits but if the word inside the register uses all bits, it cannot be protected with this method. To conclude with the general design level techniques, it is worth mentioning software techniques like the one presented in [15]. The authors propose to schedule the read operations as soon as possible and the write operations as late as possible because the registers are susceptible to suffer a transient error during the Write-Read and Read-Read interval.

Implementation level techniques are the ones which apply directly to the FPGA storage resources (Flip-Flops, LUTRAMs and BRAMs). The dual port memories of soft-processors are mapped to these aforementioned resources. These techniques at the implementation level include hardware redundancy, for example, Dual Modular Redundancy (DMR) or Triple Modular Redundancy (TMR), or memory scrubbing or ECC codes combined with redundancy [16], [17], [18], [19], [20]. In a bid to reduce the area and the power consumption, the authors in [16] propose to use the capabilities of the Xilinx FPGA LUTs and the structure of the SLICE combined with ECCs to protect the flip-flops from single bit errors. However the benefits come at a delay cost compared with TMR. Memory scrubbing is another technique to protect FPGA memories, in particular, the smaller and faster memories, Shift-Registers (SRLs) and LUTRAMS [17], [18]. Although the larger on-chip memories, the BRAMs, are protected with Single Error Correction Double Error Detection (SEC-DED) codes, they suffer from a read latency when dual-port memory structures are mapped to them. An interesting solution is presented by the authors in [19] for solving the above mentioned problem with prediction-oriented logic which incurs delay and area overhead. In contrast, the proposal presented in this paper manages to protect the dual-port register file of soft-processors without significant time delay and with very little area overhead. This is achieved by exploiting the already-present duplication for supporting dual-port reads when dual port memories, like the register file, are mapped in Xilinx FPGAs

3 Register File Protection

Our proposed methodology leverages the inferred architectural redundancy by the FPGA design automation tools when mapping a processor register file. This existing redundancy is augmented with an efficient error detection and error masking method to provide TMR like capabilities at considerably low area and delay overhead. The following sections illustrates our proposed method in details.

3.1 Read-Port Address Switching Based Fault Masking

As stated before, the Xilinx design tool implements a dual port register file as two separate physical copies, RD1 and RD2, each one containing all the register values (see Fig. 1). Although the presented technique will be explained in depth in the next sections, a general overview is presented in this section so that the readers can get the top level approach in advance. The strategy of the presented technique is as follows. If an instruction reads two operand registers and one of the parity checks is wrong, that means that the register associated to that parity check has suffered an error. If, after this, the two operands are swapped in the read lines, then their actual values would be retrieved from the opposite register copy, thus avoiding the instance that has suffered the error. Since we are assuming a single error mechanism, there will always be at least one valid copy of every register in the system. An example of this technique is shown in Fig. 1. On the left, register r2 in RD1 (whose initial correct value was 16) has been affected by an error in its least significant bit, changing its content to 17. A read operation of the register reflects this (using a parity check), but its second copy in RD2 is error-free. An error detection logic is added (Section 3.3) to detect such an error and control the switching of read address lines. In the example, before the switching, r2 was read from the first copy RD1, but after switching r2 is now read from the error-free second copy RD2, thus avoiding the wrong value. Similarly, notice that r10 was read before from RD2, but it is swapped to be read from RD1. In this case the output of r10 is still valid, since none of its copies has suffered any error. With this technique, no matters where the error affects, the system is able to swap the read lines and get a good copy of both registers. Notice that with this procedure faults are masked, although not corrected. This is not a problem because the register file is frequently updated and the corrupted content would be replaced by another value. The implementation of this scheme in Xilinx FPGAs and the error detection and switching logic is described in the following sections.

3.2 Register File Implementation in Xilinx FPGAs

Register files are an integral part of any processor architecture and are vital to provide support for instructions operating on multiple registers simultaneously. Usually, most operations require two registers as inputs. Therefore, simultaneous readings of two values from the register file is an important consideration. Since memories available inside SRAM-based FPGAs are designed with one read port, design automation tools replicate the same data in more than one instance of the architectural memory primitives to support multi-read ports. The write port of the register file remains connected to both the replicated instances as the same data has to be written in a synchronous manner while storing data in the register file. In Fig. 2, the Logical view, illustrates the scheme for a dual (read) port register file that has a word width of 64 bits and depth of 32 registers. This view shows the content organization with respect to individual registers arranged horizontally in rows and then divided into equal size chunks vertically. Each chunk is housed by a fundamental block annotated in the logical view. Consequently, each fundamental building block acts as a smaller register file of 32 registers ($r0, r1, \ldots, r31$ in Fig. 2, Logical view) with word width of 6 bits (3 LUTs each having 2 bits outputs). Therefore, the entire register file contents are divided into equal-size chunks of 6 bits denoted as $C1, C2 \ldots C11$ in Fig. 2 and stored across multiple fundamental blocks. The physical view in Fig. 2 illustrates that the two instances of the FPGA memory are used (COPY A and COPY B), each one connected to different read

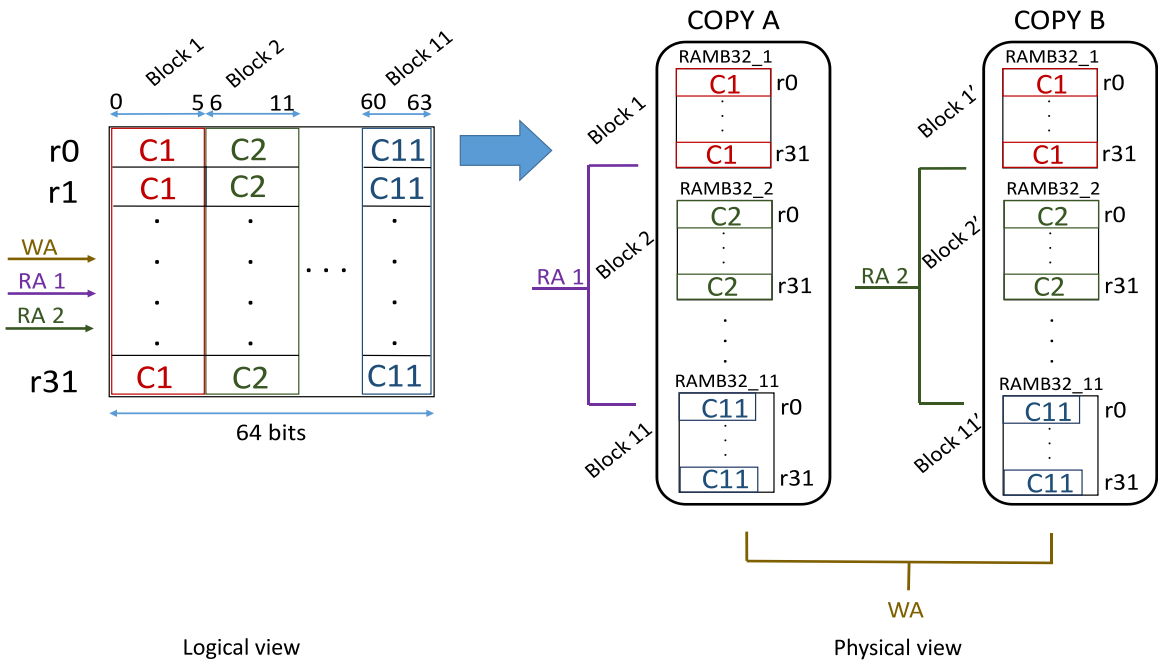

Fig. 2. Implementation of a register file in Xilinx FPGAs.

address lines (RA1 and RA2 respectively) but the same write port. It has to be mentioned that the write operation is synchronous but the read operation is asynchronous. Furthermore, the Block 1 in the Logical View is mapped to the FPGA as Block 1 for COPY A and Block 1′ for COPY B. This process is repeated for Block 2 and onwards until reaching the Block 11, which is mapped in the same way but with less resources. This mapping results in eleven (for COPY A and for COPY B) such memories to be used to compose the entire register file. As the considered register file has a word width of 64 bits and the utilized memories have the capacity to store 66 bits (11 memories x 6 bits for each copy of the register file), the spare bits could be used for error detection in the entire register file as will be discussed in Section 3.3. This implementation utilizes the available on-chip distributed memory structures in Xilinx FPGAs. The fundamental building block of this implementation is a SLICEM composed of four LUTs (termed as RAMB32M in Xilinx documentation [20]). In order to understand why one LUT is unavailable to use inside the fundamental building block, it is essential to understand how a LUT in Xilinx FPGAs works. Each LUT (6-inputs, 2 outputs) is further composed of two smaller LUTs (5-inputs, 1-outputs). Therefore, it supports two different logic or memory storage operations but with shared inputs. The architectural definition of the 32 X 6 Simple Dual Port Distributed RAM (RAM32M), our fundamental building block, dictates that one out of four LUTs cannot be used as it can be seen in the Xilinx 7 series CLB manual [20]. It is important to note that the distinction is made between the spare bits (or correspondingly the spare LUT) and the unavailable LUT. The unavailable LUT is due to the structure of the RAM32M primitive while the spare LUT results from the mapping of a memory with specific dimensions to FPGA. The amount of spare bits depends upon the word width of the memory and mostly resides in the last RAM32M instance. For example, while mapping a register file of 64 bits width, the amount of spare bits is two. In the general case of other word width, the spare bits can be calculated with the expression given:

$$6 - (Word_{with} \ \% \ 6) \ . \tag{1}$$

The next section (Section 3.3) utilizes these spare bits and the duplicated memories in an efficient manner to convert the inherent register file duplication into a real-time fault tolerant memory structure for soft-processors.

3.3 Proposed Protection Technique

The proposed register file protection technique is presented in Fig. 3. It can be noted that the register file implementation is modified to include a parity computation, storage and comparison module along with several multiplexers at the input and output of the register file. The parity module takes the advantage of the spare bits in the register file implementation for storage of the parity bits.

The proposed scheme is very efficient because all the LUTs inside the fundamental memory block are connected to the read/write ports. Therefore, the existing interconnections are utilized for synchronous parity computation, storage and retrieval with limited overhead. It can be noted from Fig. 3 that at the writing phase in the register file, a XOR-reduction is performed by X1 on the written word and a parity bit at a corresponding location is saved in the parity memory. Although, for sake of illustration, the parity memory in Fig. 3 is shown separately, it is implemented inside the same FPGA mapped register file memory structure. Therefore, the parity memory is not adding any resource overhead due to the exploitation of unused bits in the FPGA mapped register (spare bits are utilized) as discussed in the previous section). The error detection and masking logic is designed to deal with two distinct cases, i.e., when an instruction is using two separate operand registers (like ADD R1,R2,R3) and when an instruction is using the same register (e.g., AND R1, R2, R2 which is a common method to implement a MOV instruction in most RISC processors). In the first case, at the reading stage, the parity is calculated for the output words by X2 and X3, and compared with the stored parity bit. In case of a mismatch, a parity-check flag is generated by O1. This parity check-flag is further sampled by the negative edge triggered flip-flop FF_2, inserted to break the combinational feedback cycle necessary to avoid multiple writing of the register contents. This flip-flop is followed by two multiplexers M_5 and M_6. These multiplexers are inserted to deal with the aforementioned two cases. In the first case (different registers), the same_signal will be logic 0 leading to the selection of FF_2 output while in the second case (same register), the same_signal will be logic 1 resulting in selection of outputs from FF_1 and FF_3. Note that these flip-flops are directly sampling the inequality

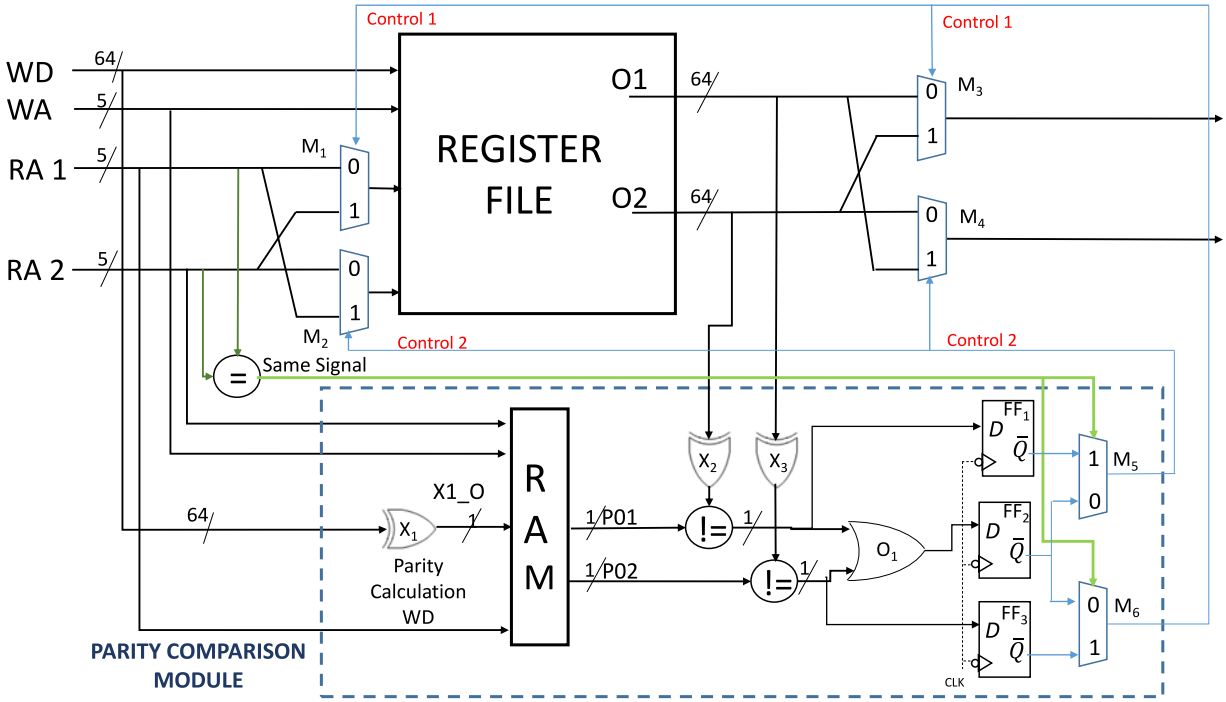

Fig. 3. Proposed error masking method.

operators which means that these flip-flops reflect which of the copy is faulty to subsequently switch address and output lines for the faulty one only using the multiplexers at the inputs (M_1 and M_2) and outputs (M_3 and M_4) of the memory. Therefore, in both cases the faulty memory copy is avoided by reading from the alternative copy. As the register file at every granularity in this implementation uses an asynchronous read, the output data is almost immediately available. The propagation delay of the inserted error detection and masking logic is affordable compared to the operating clock cycle as will be reported in Section 4.1. Therefore, the entire swapping mechanism is completed before the next rising edge of the clock cycle, thus providing correct data to the upstream stages in the processor execution pipeline. In the case of a second read request to the fault memory location in the subsequent clock cycles, the parity mechanism will utilize the steering mechanism for error masking. However, if a different location is read again in the subsequent clock-cycle, the parity module will detect no mismatch under the single error fault model assumption (which is common to Redundancy-based approaches like TMR) and the multiplexers (M5 and M6) will ensure the selection of correct input addresses and outputs for the normal operation. Moreover, as the execution progresses, the faulty register file contents will be rewritten. Therefore, the faulty register file content is left uncorrected due to the small likelihood of fault accumulation over time. Compared to the TMR voter-logic, the proposed technique offers similar real-time SEU masking capabilities at a much lower resource overhead. However, the proposed approach cannot handle multiple bit errors in individual register contents as TMR does. This is because the parity-based detection is unable to detect errors that affect an even number of bits. To deal with multiple errors, the proposed scheme can be extended by using SEC or SEC-DED based error detection and subsequent readout port-switching. However, this increased protection would come at an increase in resource overhead mostly due to the additional parity bits required for the SEC or SEC-DED code on every register.

The added functionality for parity-calculation and comparison logic result in a timing delay that is small enough to provide safe margins to avoid setup time violations at the input of the inserted flip-flops and its post-Place and Route (PAR) static timing analysis.

4 Evaluation

The proposed protection technique is applied to the register file of a soft-processor implemented on a 28-nm Artix-7 FPGA. The following sections discuss the evaluation details pertaining to implementation and the protection effectiveness.

4.1 Implementation Resource Evaluation

An untethered implementation of the RISC-V ISA, the lowRISC processor [22], is used as a testbed for demonstrating the effectiveness of the proposed technique. In particular, our method was applied to the integer register file which supports dual-port asynchronous read and single-port synchronous write operations. The integer register file was mapped to the distributed memory resources at the FPGA implementation phases. The logic related with "Parity Comparison Module", "*Same Signal comparators*" and Multiplexers $M_0 - M_4$ in Fig. 3 are added in Chisel language. This is the language in which the processor was originally developed. This is a high-level language for hardware description and subsequent RTL Verilog generation. The activation of the spare bits for parity RAM storage and retrieval in the last LUT is straight forward as each RAM32M (each Block) is a Xilinx primitive. Therefore, the signals "X1_O" and "PO1" and "PO2" in Fig. 3 (ensured to be maintained through synthesis) are connected in the post-synthesis Verilog netlist by modifying the port connection DIC and DOC of the final RAM32M primitive (i.e Block 11 of COPY A and COPY B) corresponding to LUT-C. The modified Verilog netlist is loaded again for subsequent FPGA implementation phases.

As the added redundancy is not applied at a modular-level, it introduces considerably less area overhead compared to TMR while results in a slight degradation in timing delay. Table 1 shows the comparison between the TMR-version of the register file, unprotected version and our protection technique in terms of FPGA resources. It is worth mentioning that the resource comparison is made based on the register file as a standalone component for comparison purposes. It is evident from Table 1, that our protection techniques only needs one third of the LUTRAM resources than TMR uses. Our approach utilizes the same LUTRAM resources as the unprotected version with only 129 LUTs for error

TABLE 1
Register File Resource Comparisons

	Register File		Masking Resources		
	LUTRAMs	SliceM	LUTs	FF	SliceL
Default	88	22	0	0	0
TMR	264	66	128	0	43
Proposed	88	22	129	3	44

detection and masking logic for the 64-bit wide register file. The next parameter of evaluation to consider is the timing delay introduced by the inserted protection technique in comparison to the system clock period. A static timing analysis was performed with vendor tools to measure the post-Place and Route (PAR) implementation delay in the critical path. Our analysis shows that the delay across the error detection and masking logic is almost 4 ns compared to the clock period of 10 ns. Most of the delay contribution comes from the XOR-reductions, X_2 and X_3 in Fig. 3, as they operate on 64-bit entries and require several LUT levels. This time is well below the clock period length and does not violates the setup timings of the FPGA flip-flops which is typically 0.07- 0.12 ns [21].

4.2 Protection Effectiveness Evaluation

The proposed scheme was implemented and tested in a Nexys 4 DDR board, which implements a XC7A100T-1CSG324C Xilinx Artyx-7 FPGA. The synthesis and implementation tools used to design this technique as well as to evaluate it are Vivado and Xilinx SDK 2016.4.

In order to evaluate the effectiveness of the proposed technique, a fault injection at a fine-granularity targeting the contents of the register file was carried out. The fault injection platform presented in [25] is adapted for SEU injection in LUTRAM. To accomplish that, a Microblaze-processor has been implemented alongside the lowRISC acting as a master. The DUT which is the register file, was isolated using the Vivado placement constraints. The Microblaze was chosen for fault injection because of the library support for accessing the configuration memory at a frame-level, thereby enabling the "*selective*" modification of corresponding mapped resources of interest [23]. The fault injector code for Microblaze was implemented using the Xilinx SDK. This program, stops the clock at a certain cycle. This is done because the LUTRAM context is dynamic and therefore the injection process could fail. With the clock stopped, the injector retrieves the part of the bitstream which corresponds to the register file using the ICAP port, commits the bit-flip, writes the bitstream again and resumes the execution. The process is repeated in all the cases of the experiment domain. Both processors run at 100 MHz.

In total, 4,160 errors covering the entire range of register file bits were injected at randomly selected clock cycles. These random numbers were generated by a Linear Feedback Shift Register (LFSR) logic connected to the Microblaze-processor. It should be noted that the injection cycle was selected from the range of clock cycles the benchmark takes to execute. During the injection, the lowRISC was halted and allowed to re-execute once the injection was completed. The considered software benchmarks running on the lowRISC processor were carefully selected based on certain characteristics which follows [24].

1. Quicksort: It has several jumps and conditions to calculate, thus it is a good candidate to test the program execution flow. It receives the input of a 20-element unsorted array with random numbers from −1000 to 1000. The output is the sorted input array.
2. FFT: This algorithm is widely used in signal processing applications and heavily utilizes data centric operations.
3. Matrix multiplication: It performs a lot of arithmetical operations that highly stress the register file, making the program execution more susceptible to failures. The matrices used in this test were 20×20 square matrices filled with random numbers from −1000 to 1000. The output is another square matrix with the result of the multiplication.

When the benchmark execution finishes, the results are logged to the PC for further analysis and comparison with the pre-stored golden benchmark outputs.

The result of the described injection campaign was that no error was detected at the output, which confirms that the presented protection technique of the register file is able to avoid 100 percent of the failures in the actual registers. On the other hand, the same injection campaigned was applied to the unprotected version of the register file, resulting in 10.17 percent of errors at the output.

5 Conclusions and Future Work

This paper presents an efficient technique to protect the register file of soft-processors implemented in Xilinx SRAM-based FPGAs. The method leverages the inferred redundancy incorporated by the design automation tools to come up with a low overhead approach to error detection and masking at considerably less cost than TMR. Our evaluations demonstrated that the delay introduced by the proposal is affordable in comparison to the processors clock period. The presented technique can be applied not only to register files in soft-processors but also to other similar dual-port memory structures implemented through distributed RAM resources on SRAM-based FPGAs.

The proposed scheme using parity based error detection is able to correct single bit errors. Future work will consider the protection against multiple bit errors by using the proposed technique with SEC or SEC-DED codes to perform error detection.

Acknowledgments

This work was supported by the Spanish Ministry of Economy and Competitiveness under Grant ESP2014-54505-C2-1-R.

References

[1] H. Quinn, et al., "The Cibola flight experiment," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 1, Feb. 2015, Art. no. 3.
[2] H. Quinn, et al., "Flight experience of the Xilinx Virtex-4," *IEEE Trans. Nuclear Sci.*, vol. 60, no. 4, pp. 2682–2690, Aug. 2013.
[3] N. A. Harward, M. R. Gardiner, L. W. Hsiao and M. J. Wirthlin., "Estimating soft processor soft error sensitivity through fault injection," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, May 2015, pp. 143–150.
[4] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual," May 2017, [Online]. Available: https://riscv.org/specifications/
[5] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *ACM Trans. Storage*, vol. 4, no. 3, pp. 8:1–8:28, Nov. 2008.
[6] 7 series FPGAs memory resources (ver 1.12), in Xilinx Document No. UG473, Sep. 2016. [Online]. Available: https://www.xilinx.com/
[7] A. Manuzzato, S. Gerardin, A. Paccagnella, L. Sterpone, and M. Violante, "Effectiveness of TMR-based techniques to mitigate alpha-induced SEU accumulation in commercial SRAM-based FPGAs," *IEEE Trans. Nuclear Sci.*, vol. 55, no. 4, pp. 1968–1973, Aug 2008.
[8] P. Reviriego, S. Pontarelli, J. A. Maestro, and M. Ottavi, "A method to construct low delay single error correction codes for protecting data bits only," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 32, no. 3, pp. 479–483, Mar. 2013.
[9] LowRISC, lowRISC trace debugger project, Jul. 2016. [Online]. Available: http://www.lowrisc.org/docs/debug-v0.3/
[10] R. Naseer, R. Z. Bhatti, and J. Draper, "Analysis of soft error mitigation techniques for register files in IBM cu-08 90nm technology," in *Proc. 49th IEEE Int. Midwest Symp. Circuits Syst.*, Aug. 2006, pp. 515–519.
[11] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke, "Cost-efficient soft error protection for embedded microprocessors," in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2006, pp. 421–431.
[12] G. Memik, M. T. Kandemir, and O. Ozturk, "Increasing register file Immunity to transient errors," in *Proc. Des. Autom. Test Europe*, Mar. 2005, pp. 586–591.
[13] H. Amrouch and J. Henkel, "Self-immunity technique to improve register file integrity against soft errors," in *Proc. 24th Int. Conf. VLSI Des.*, Jan. 2011, pp. 189–194.

[14] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, Apr. 1950.
[15] J. Yan and W. Zhang, "Compiler-guided register reliability improvement against soft errors," in *Proc. 5th ACM Int. Conf. Embedded Softw.*, 2005, pp. 203–209.
[16] P. Reviriego, M. Demirci, J. Tabero, A. Regadío, and J. A. Maestro, "DMR +: An efficient alternative to TMR to protect registers in Xilinx FPGAs," *Microelectronics Rel.*, vol. 63, pp. 314–318, 2016.
[17] P. Adell, G. Allen, G. Swift, and S. McClure., "Assessing and mitigating radiation effects in Xilinx SRAM FPGAs," in *Proc. Eur. Conf. Radiation Effects Components Syst.*, Sep. 2008, pp. 418–424.
[18] N. H. Rollins, "Hardware and software fault-tolerance of softcore processors implemented in SRAM-based FPGAs," Ph.D. dissertation, Electrical and Computer Engineering Dept., Univ. Brigham Young, Provo, Utah, 2012.
[19] C. Hong, K. Benkrid, X. Iturbe, and A. Ebrahim., "Design and implementation of fault-tolerant soft processors on FPGAs," in *Proc. 22nd Int. Conf. Field Program. Logic Appl.*, Aug. 2012, pp. 683–686.
[20] 7 series FPGAs configurable logic block (ver 1.8), in Xilinx Document No. UG474, Sep. 2016, pp. 23–27, [Online]. Available: https://www.xilinx.com/
[21] Artix-7 FPGAs data sheet: DC and AC switching characteristics, DS181, Sep. 2016, [Online]. Available: https://www.xilinx.com/
[22] lowRISC, lowRISC Configuration Parameters, (2015, Dec.). [Online]. Available: http://www.lowrisc.org/docs/untether-v0.2/parameter/
[23] E. Sanchez, L. Sterpone, and A. Ullah, "Effective emulation of permanent faults in ASICs through dynamically reconfigurable FPGAs," in *Proc. 24th IEEE Int. Conf. Field Program. Logic Appl.*, Sep. 2014, pp. 1–6.
[24] H. Quinn, et al., "Using benchmarks for radiation testing of microprocessors and FPGAs," *IEEE Trans. Nuclear Sci.*, vol. 62, no. 6, pp. 2547–2554, Dec. 2015.
[25] A. Ullah, E. Sanchez, L. Sterpone, L. A. Cardona, and C. Ferrer, "An FPGA-based dynamically reconfigurable platform for emulation of permanent faults in ASICs," *Microelectronics Rel.*, vol. 77, pp. 110–120, Jun. 2017.