Reliability Engineering and System Safety 151 (2016) 1–9

Contents lists available at ScienceDirect

Reliability Engineering and System Safety

journal homepage: www.elsevier.com/locate/ress

Synchronization of faulty processors in coarse-grained TMR protected partially reconfigurable FPGA designs

U. Kretzschmar [a], J. Gomez-Cornejo [a], A. Astarloa [a], U. Bidarte [a,*], J. Del Ser [b,c]

[a] Department of Electronics, University of the Basque Country UPV/EHU, 48013 Bilbao, Bizkaia, Spain
[b] OPTIMA Area, TECNALIA Research & Innovation, 48160 Derio, Bizkaia, Spain
[c] Department of Communications Engineering, University of the Basque Country UPV/EHU, 48013 Bilbao, Bizkaia, Spain

ARTICLE INFO

Article history:
Received 31 August 2015
Received in revised form
27 November 2015
Accepted 23 December 2015
Available online 16 February 2016

Keywords:
Reliability
TMR
FPGA
Synchronization
Fault-recovery
Processor

ABSTRACT

The expansion of FPGA technology in numerous application fields is a fact. Single Event Effects (SEE) are a critical factor for the reliability of FPGA based systems. For this reason, a number of researches have been studying fault tolerance techniques to harden different elements of FPGA designs. Using Partial Reconfiguration (PR) in conjunction with Triple Modular Redundancy (TMR) is an emerging approach in recent publications dealing with the implementation of fault tolerant processors on SRAM-based FPGAs. While these works pay great attention to the repair of erroneous instances by means of reconfiguration, the essential step of synchronizing the repaired processors is insufficiently addressed. In this context, this paper poses four different synchronization approaches for soft core processors, which balance differently the trade-off between synchronization speed and hardware overhead. All approaches are assessed in practice by synchronizing TMR protected PicoBlaze processors implemented on a Virtex-5 FPGA. Nevertheless all methods are of a general nature and can be applied for different processor architectures in a straightforward fashion.

© 2016 Elsevier Ltd. All rights reserved.

1. Introduction

The great flexibility, high achievable system speeds and the large number of available design resources make SRAM-based FPGAs a good choice for a wide variety of electronic designs. Especially the low non-recurring engineering cost of FPGA based designs, where the high initial expenses of ASICs or ASSPs cannot be compensated by very high production volumes. Such designs realized using FPGAs have been shown to outperform standard CPUs [1–3]. Nevertheless, modern FPGA implementations typically utilize soft- or hard-core processors to enable an efficient implementation of the overall system, as well as to take advantage of existing hardware modules [4,5].

An additional advantage of SRAM-based FPGAs is Dynamic Partial Reconfiguration (DPR), which allows designers to change parts of the implemented design while keeping the overall system operational. This time-multiplexing method of FPGA resources can be used in a variety of ways such as e.g. adapting a cache architecture to specific application requirements [6], implementing a multi-protocol network switch or enabling software defined radio [7].

Due to the advantages that it provides, the number of fields of application that make use of the FPGA technology continues growing. In certain of these fields, in which a faulty operation can jeopardize human life or the integrity of valuable technology, high levels of reliability are required. Railway [8,9], automotive [10,11] and space [12,13] systems are remarkable examples. In such systems robustness is one of the most relevant aspects and the high susceptibility of SRAM-based FPGA technology to Single Event Effects (SEE) becomes a crucial factor. SEE faults can be caused by high energy particles impacting on the FPGA [14,15]. The most critical among all SEEs are Single Event Upsets (SEUs) [16], specially when they affect the configuration memory [17]. Unlike ASICs where the interconnections and the logic elements on the die are fixed, FPGAs use a configuration bitstream for defining the function of the configurable logic elements or the interconnection matrix. Configuration memory upsets can consequently alter the implemented design by changing its elements. SEUs in user memories such as FPGA internal block RAM (BRAM) and flip-flops are not as critical for the overall system, because the SEU occurrence rate per bit of BRAM is in the same order of magnitude as for the configuration memory. By contrast, a FPGA typically has one order of magnitude more configuration memory than BRAM [18–20].

To avoid this issue, methods for mitigating the susceptibility of FPGA designs against SEUs have been thoroughly investigated in the literature by resorting to Error Correction Codes (ECC) [21,22]

* Corresponding author.
E-mail address: unai.bidarte@ehu.eus (U. Bidarte).

http://dx.doi.org/10.1016/j.ress.2015.12.018
0951-8320/© 2016 Elsevier Ltd. All rights reserved.

or Duplication With Comparison (DWC) [23,24]. In particular, the so-called Triple Modular Redundancy (TMR) method results to be the most frequently addressed by both industry and academia in diverse technological architectures [25]. The rationale for this trend is threefold: (1) the possibility of fault masking by implementing the process of *voting*; (2) the method of scaling the TMR protection by changing its granularity [26]; and (3) the availability of tools allowing for a completely automated TMR generation [27]. TMR is typically combined with configuration scrubbing [28–30], a process which corrects configuration memory upsets. In this combination TMR enables the design to continue operating correctly in presence of faults, whereas scrubbing avoids the accumulation of multiple faults.

Although the combination of TMR and scrubbing is the fault handling strategy recommended by the FPGA vendors [31], several recent publications have proposed a step beyond this established method. This new strategy is based on coarse-grained TMR where each triplicated instance of the design is partially reconfigurable. Any error in one of such instances can be corrected by reconfiguring the corresponding module [32–34]. By providing distinct implementations of the same instance, this solution can also repair instances where the corresponding FPGA region has suffered a permanent error. Notwithstanding the great extent to which the TMR setup and the partial reconfiguration aspect of this approach have been studied in the related literature, the required synchronization of the reconfigured instance has been either neglected or it has been analyzed in an incomplete manner.

This work addresses this scarcity of investigations around synchronization in TMR systems by proposing, implementing and evaluating four different synchronization methodologies. The four approaches span a broad spectrum of possible alternatives from minimal hardware overhead to completely hardware-based synchronization. This allows balancing the trade-off between implementation cost and synchronization speed, depending on the requirements of the target application at hand. The performance of the proposed techniques is verified and compared to each other on the PicoBlaze [38] processor. However, all four approaches are of general nature and can easily be migrated to other processor architectures. These papers synchronization methods are furthermore not restricted to a set-up implementing TMR and DPR. They are applicable to any TMR protected processor system to recuperate a processor element, which was forced out of sync by a SEE.

The remainder of this work is structured as follows. Section 2 surveys recent advances on fault protection using TMR and DPR. Next, Section 3 proposes the aforementioned four different synchronization approaches, whereas implementation details are outlined in Section 4. Practical results are presented in Section 5, and finally Section 6 ends the paper by drawing some concluding remarks.

2. Fault tolerant systems based on DPR and TMR

The combination of Triple Modular Redundancy and Dynamic Partial Reconfiguration is a very attractive solution for the implementation of fault tolerant systems. Among many different possible implementation forms of TMR, the so-called coarse-grained TMR [26] implements three instances of the same module and a final voter. This method provides a slightly lower protection than fine-grained TMR [26,39], where modules needing protection are broken into smaller parts. Some fine-grained TMR approaches even provide synchronization of the three modules [40,27]. However, coarse-grained TMR is ideal for its combination with DPR as it results in a small amount of reconfigurable partitions. A block diagram of this combination is depicted in Fig. 1.

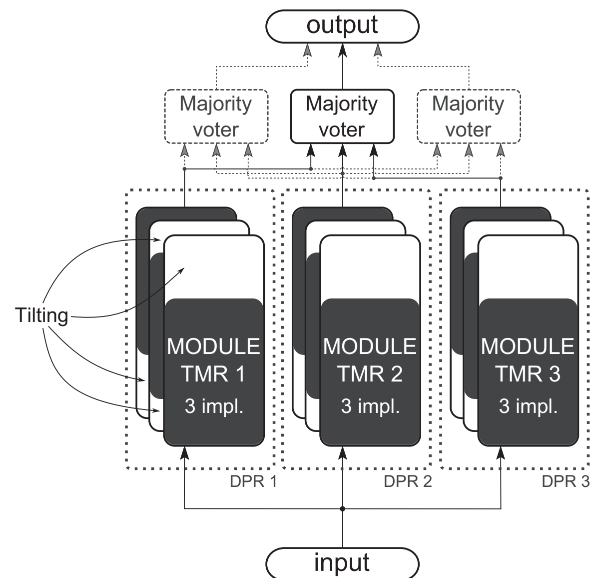

Fig. 1. Typical architecture of the combination of TMR and DPR.

The above figure summarizes the high level architecture proposed in a number of recent publications [32–37]. Three instances of the same module are placed in three partially reconfigurable areas. Configuration errors in one of these modules can consequently be repaired by reloading the bitstream of the faulty module by using partial reconfiguration once a voting step has identified the module providing the incorrect input. This voting step can be implemented either as a single voter or as a triplicated voter as suggested in Fig. 1. In addition to this protection against configuration errors, the architecture in Fig. 1 accommodates the idea of *tilting* [24].

In tilting the three reconfigurable regions are enlarged to enable different implementations of the same logic, leaving selected parts (the white spaces marked by arrows) of the partially reconfigurable area unused. This strategy provides a means to avoid permanent errors: if e.g. one of the reconfigurable regions fails to operate correctly due to a permanent error, reloading the same partial bitstream will not recover this corresponding instance. But if a different tilted implementation of the module is loaded, the region can be repaired if the tilted bitstream does not use the region with the permanent fault.

Nevertheless, in the majority of cases a reconfiguration by itself does not suffice for recovering a faulty TMR instance. If this instance features some kind of internal state, this state needs to be synchronized to the other instances after the reconfiguration. In this line of reasoning, a synchronization method valid for small Finite State Machines (FSM) is proposed in [36] introducing the notion of *state prediction*. State prediction suggests that each FSM has (at least) one state to which the machine always returns after a finite amount of time. Therefore, by setting the FSM of a reconfigured module to this state it is possible to wait for the other two instances to reach this point during their normal operation, and thereafter continue seamlessly operating with all three instances. It should be clear that this method is obviously only applicable for small state machines with a reduced number of possible states.

Synchronization has also been considered in [35], where a fault tolerant MicroBlaze architecture using TMR and DPR was presented, as summarized in Fig. 2.

In this work three MicroBlaze processors sharing peripherals and memory were implemented in partially reconfigurable areas. The peripherals and the shared memory are protected by TMR and ECC, respectively. Sharing one memory between the three processors

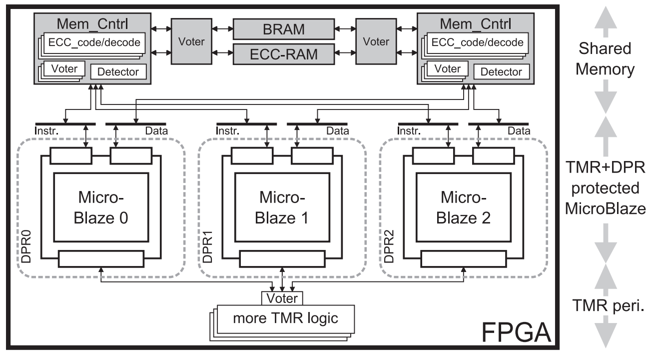

Fig. 2. Architecture combining TMR and DPR with shared memory (simplified from [35]).

reduces synchronization to a process of reading and writing to the memory: whenever the processors write data to the memory a voting process is started, which will mask wrong data from the newly reconfigured instance, and store the correct values sent from the two remaining functional instances. In a subsequent read cycle the three processors can read the synchronized value back to their memories.

While this synchronization approach is suitable for MicroBlaze processors, it is not applicable to all processor architectures. For the MicroBlaze processor it is possible to access all registers of relevance for the synchronization process, such as e.g. the stack pointer, the status register and the program counter. In this manner a synchronization by reading and writing to the shared memory becomes feasible. On the contrary, many other popular processor architectures (e.g. PicoBlaze or PIC) do not provide reading access to all registers representing the state of the CPU.

Another related contribution is the work in [24], where the synchronization between two MicroBlaze processors operating in Double Modular Redundancy (DMR) is addressed. After one of such processors is partially reconfigured, a similar technique to *state prediction* is used. After identifying the faulty processor, the so-called *roll-forward* procedure is executed to set the processor to a state the other MicroBlaze will reach in the future. Since the state is assumed to only consist of the program counter, a synchronization similar to the one in [35] is required after the roll-forward to update the register contents. This leaves this method subject to the same drawbacks as those previously identified for the shared memory approach in [35].

The well known approach of *checkpointing* and *rollback* as used for the synchronization of duplicated processors in the so-called *Lockstep* technology [24,23,41] is only mentioned here for completeness sake, as it does not translate well to the fault protection method represented in Fig. 1. In a system using DPR and TMR the periodic checkpointing (copying of the processors state) is unnecessary because in case of a fault in a single processor the correct system state can still be obtained by comparing all three processor instances (voting).

3. Proposed synchronization approaches

When implementing a combination of Triple Modular Redundancy and Dynamic Partial Reconfiguration for the realization of fault tolerant systems, a pure reconfiguration of a faulty module is not sufficient given that the reconfigured module comprises of an internal state. This synchronization is especially critical for processors, because their state is composed of a number of different registers. For a typical processor the following parts need to be synchronized: the program counter, the stack-content, the stack-pointer, the flag-status, the processor registers and the internal memory, which is provided by some processors such as the PicoBlaze. These elements will be referred to as *synchronization objects* in the remainder of this work. The synchronization of caches will not be evaluated in the following approaches, but a synchronization strategy for caches can be as easy as clearing their contents in all processors, in case of a write-through strategy.

In general finding an adequate synchronization strategy for a given application implies balancing a trade-off. On the one hand, adding specialized hardware for the processor synchronization will enable a very fast synchronization process. On the other hand, implementing the synchronization with little extra hardware combined with software will result in less implementation overhead and a lower impact on the critical path of the design.

The structure of the synchronization method impacts the duration of two sub-steps of the whole SEU recovery process. Firstly the time required to copy the correct values of the different synchronization elements to a recently reconfigured processor instance in the coarse-grain TMR setup. This time will be called *copy-time*. The second aspect of the synchronization speed is the time from the detection of an error by the voter until the point in time where the system is ready to start the synchronization process. This second time is named *wait-for-sync*. Some approaches cannot begin directly with the synchronization, but they first need to finish ongoing calculations before CPU time can be spent for updating a reconfigured processor instance in the system. The time from the detection of an error to the re-synchronization has implications on the overall system robustness, because in this time period the TMR system operates only with two functional instances, making it vulnerable to consecutive SEUs.

The whole SEU recovery process is illustrated in Fig. 3, where the time requirement is defined by the sum of four components: the time needed to detect the error, the *wait-for-sync* time, the time consumed for repairing the SEU by partial reconfiguration and the *copy-time*. Whereas the time for partial reconfiguration is proportional to the size of the reconfigured partition and the reconfiguration speed, the *time to detection* is not affected by the synchronization approach, but only application dependent and hence is considered beyond the scope of this paper.

In the following, different synchronization approaches are presented for the example of the PicoBlaze processor. The synchronization objects of this specific architecture are summarized in Table 1. This table also contains the synchronization objects of the MicroBlaze processor, a more powerful and complex processor architecture. Despite the simpler architecture of the PicoBlaze, it is more demanding in terms of synchronization. For the MicroBlaze all synchronization objects are accessible via software, however the PicoBlaze has an inherent need for additional hardware when a complete synchronization is desired because the stack, the flags and the program counter are neither readable nor writable by software.

It needs to be noticed that this work does not consider the instruction memory as a synchronization object, because it is external to the processor. In many cases it is outside of the FPGA or designers might consider other protection methods, such as ECC as done in [35].

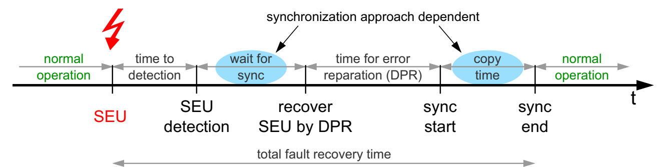

Fig. 3. SEU recovery process and impact of synchronization times.

Table 1
General synchronization objects and accessability for the PicoBlaze and MicroBlaze processors.

General synchronization object	PicoBlaze	MicroBlaze
ALU-flags	No access	Readable/writable
Stack-pointer	No access	Readable/writable
Stack-content	No access	Readable/writable
Program counter	No access	Readable/writable
CPU registers	Readable/writable	Readable/writable
Internal memory	Readable/writable	–

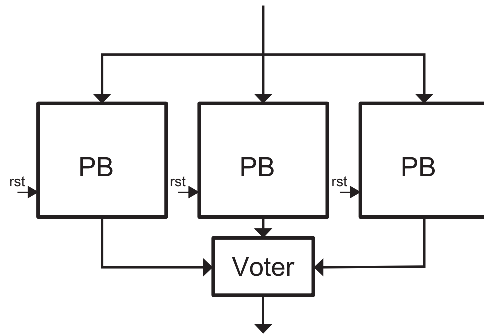

Fig. 4. Synchronization approach 1: cyclic resets.

3.1. Approach 1: cyclic resets

The first approach proposed in this work is to synchronize the three PicoBlaze processors by applying a cyclic reset as indicated in Fig. 4.

This reset allows bringing all TMR instances of the PicoBlaze back to a known state and restarting the program execution from the scratch with all three PicoBlazes in parallel. No changes in the actual PicoBlaze implementation are required. In this approach the BRAM memory on Xilinx devices is not cleared by a reset. The software on the PicoBlaze processors consequently needs to initialize all registers or scratchpad locations before the first usage.

There are different possibilities for triggering the synchronization reset:

- The PicoBlazes set a flag in a shared memory upon termination of their calculations. When at least two instances have set the flag, a reset may be issued.
- A timer is programmed to the worst-case runtime of the given software and is started together with the PicoBlazes. The expiring timer will trigger a reset.
- The instruction bus addresses of the PicoBlazes are supervised to detect the end of the program.
- Very simple applications, where it is admissible to lose one packet, may trigger the reset directly upon detection of a SEU.

Details on the software requirements of this approach are evaluated in Section 4. For the first three possible trigger methods in *Appr*. 1, the worst-case of the synchronization time *wait-for-sync* is one complete algorithm runtime. The *copy-time* does not exist because no values are copied.

3.2. Approach 2: synchronization memory and address force

Appr. 1 implies a high number of restrictions to the software running on the hardware. One important restriction is that it is not possible to leave permanent data in the processor registers or in its internal memory, in case of the PicoBlaze the scratchpad memory.

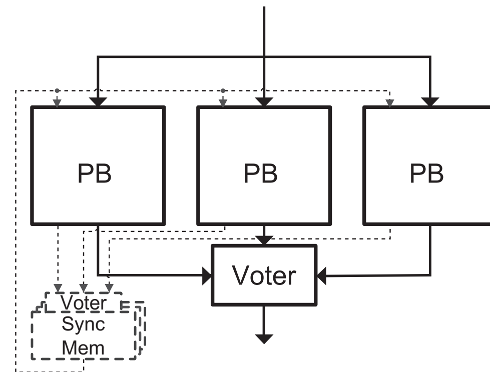

Fig. 5. Synchronization approach 2: synchronization memory.

Although this memory is not affected by the resets, it cannot be assumed to be error-free, because it is not synchronized by any means.

This drawback is addressed in *Appr*. 2 (Fig. 5), which builds upon and enhances a synchronization idea presented in [35] based on a shared memory. Adding a TMR-protected shared memory to the system allows a synchronization of the processors by concurrent writing to this memory, followed by a concurrent reading of the data. The actual synchronization of the data is executed when all PicoBlazes write concurrently to the TMR memory, where the voters on the input of the TMR memory are able to mask an incorrect input from one processor. The method proposed herein utilizes a synchronization memory of only one byte (i.e. the datawidth of the PicoBlaze) yielding in a minimal hardware overhead.

The synchronization process is triggered by externally forcing a `JUMP` instruction to all three PicoBlaze instances. The destination of this `JUMP` is the synchronization software sequence, which is executed by all PicoBlazes simultaneously. At the same time the program counters of the processors are synchronized using this simple method. In the synchronization sequence all PicoBlaze memory locations requiring synchronization are first written to and then read from the synchronization memory. It needs to be ensured that the synchronization is only triggered when the processors are not serving any interrupt. Details on the software requirements accompanying this approach are provided in Section 4.

The rationale behind forcing a `JUMP` to execute the synchronization code as opposed to using an Interrupt Service Routine (ISR) is that *Appr*. 2 follows a “black box” approach, which does not require changes in the implementation of the used processor. Using an ISR for synchronization would require a synchronization of the processor stack, which would imply changes to be made to the HDL of the PicoBlaze.

The synchronization time *wait-for-sync* in *Appr*. 2 has a worst-case of the complete algorithm runtime, whereas the *copy-time* equals the best case of four cycles per each byte of synchronized data (one read- and one write-instruction, which respectively take two cycles).

3.3. Approach 3: synchronization interrupt with synchronization memory

The third synchronization approach uses a synchronization memory and synchronization software residing in an ISR.

Under this approach the processors do not need to wait for the current calculation to finish, but it is possible to almost directly react on a detected SEU by assigning the synchronization interrupt. Along with this improvement comes the need for modifications in the PicoBlaze implementation. When using an ISR for synchronization it is essential to synchronize the processor stack in order to synchronize the return addresses of all processors,

because before entering the ISR the program counter of the reconfigured processor will have a different value than the other two.

In Fig. 6 the *Sync Control* block represents the logic added to the overall system for the synchronization of the stack-pointer, the stack-address and the shadow registers of the ALU-flags. As these registers are not software accessible, it is not possible to implement *Appr*. 3 in the same "black box" fashion as *Appr*. 1 & 2.

The advantages of this synchronization approach become visible when focusing on the *wait-for-sync* synchronization time. This is reduced to be only a few cycles long, namely the time needed for triggering the interrupt. The *copy-time* remains the same as in *Appr*. 2, since the software for the copying is identical.

3.4. Approach 4: complete hardware-based synchronization

The last proposed approach is a completely hardware based synchronization method, which is illustrated in Fig. 7. The *Sync Control* module is expanded to also control the synchronization of the registers and the scratchpad memory without requiring any specific synchronization software. Fig. 7 depicts this approach with the largest hardware overhead.

When the hardware based synchronization is executed it needs to be ensured that no software is accessing the registers or the scratchpad memory simultaneously to avoid errors. As almost all PicoBlaze instructions use registers in their operations, this work proposes to stall the processor while the hardware based synchronization of *Appr*. 4 takes place. An easy implementation of this stalling can be achieved by externally forcing all PicoBlaze instruction addresses to the last memory position, namely `0x3ff`. This address is the interrupt entrance point and thus always contains an unconditional `JUMP` instruction to the ISR.

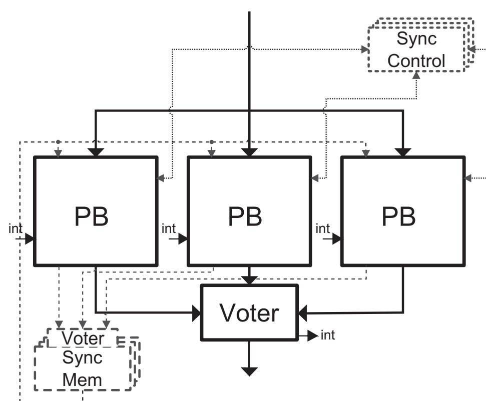

Fig. 6. Synchronization approach 3: synchronization memory + interrupt.

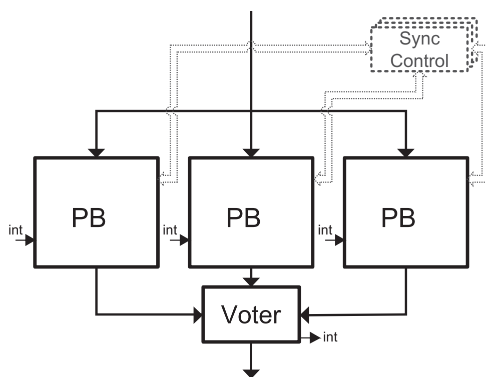

Fig. 7. Synchronization approach 4: hardware based synchronization.

Table 2
Synchronization method overview.

Synchronization object	Appr. 1	Appr. 2	Appr. 3	Appr. 4
ALU-flags	–	–	HW	HW
Stack-pointer	–	–	HW	HW
Stack-content	–	–	HW	HW
CPU registers	–	SW	SW	HW
Scratchpad memory	–	SW	SW	HW
Program counter	Reset	Forced `JUMP`	On ISR return	On ISR return

A tailored optimization of the stall time can be achieved in a variety of ways. If e.g. the usage of the scratchpad memory in the first part of the ISR is forbidden, the stalling can be as short as 16 cycles (the register synchronization time). The stalling can also be shortened, when only a subset of scratchpad locations require synchronization.

Appr. 4 has the same short *wait-for-sync* time as *Appr*. 3, but it has a significantly reduced synchronization time for the registers and scratchpad (*copy-time*). Whereas software based synchronization takes 4 cycles per memory cell, hardware based synchronization needs only one cycle. An additional benefit of hardware synchronization is that the copying of registers and scratchpad memory can be executed in parallel.

An overview of all the four proposed approaches is given in Table 2. It summarizes the synchronization strategy for all synchronization elements of the PicoBlaze processor. The entry *SW* represents the synchronization via the synchronization memory, *HW* represents a modification to the processor and a dash indicates that the element is not synchronized. The synchronization methods for the program counter manifest the biggest differences. Whereas *Appr*. 1 relies on resetting this register, *Appr*. 2 works by forcing a `JUMP` instruction to update this value. On the contrary *Appr*. 3 & 4 do only synchronize the stack while being in the ISR and the program counter is updated upon returning from the interrupt.

4. Implementation details

4.1. Software restrictions for approaches 1 & 2

When working with approaches which do not synchronize the device in its entirety (*Appr*. 1 & 2), this fact needs to be considered for the software running on the processors. On this purpose two different software flows are hereafter outlined enabling the utilization of *Appr*. 1 and *Appr*. 2.

The flow in Fig. 8(a) schedules a reconfiguration after each iteration of the algorithm implemented on the processor. After the software finishes its execution a waiting state is entered (e.g. an endless loop). After it has been detected that at least two processors reached this waiting state or after the expiration of a synchronization timer a resynchronization is triggered. This is executed independently of whether a fault occurred or not, in order to leave the waiting state and prepare the following iteration of the algorithm.

A software flow triggering reconfigurations only if necessary is presented in Fig. 8(b). After finishing one calculation iteration the processors read the TMR-protected information of the system voter if a resynchronization is required. If this is the case, then the software flow proceeds to a waiting state as in Fig. 8(a) and waits for synchronization by reset or by a forced `JUMP`. If on the other hand no synchronization is required, the processors may directly continue with the next calculation.

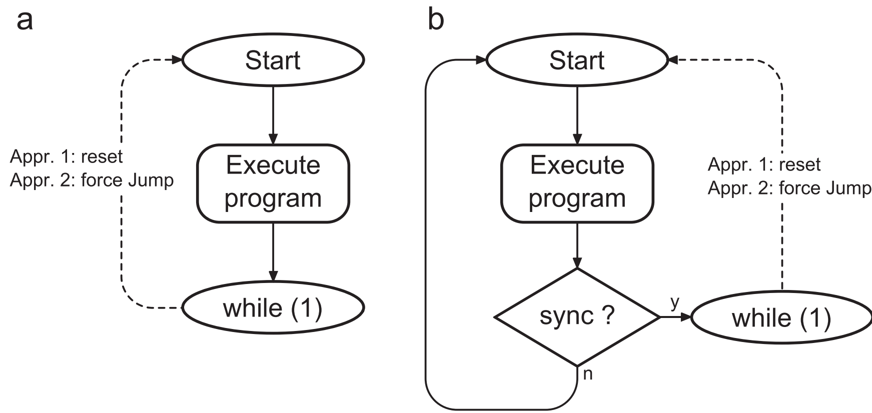

Fig. 8. Required program style when using *Appr.* 1 or *Appr.* 2.

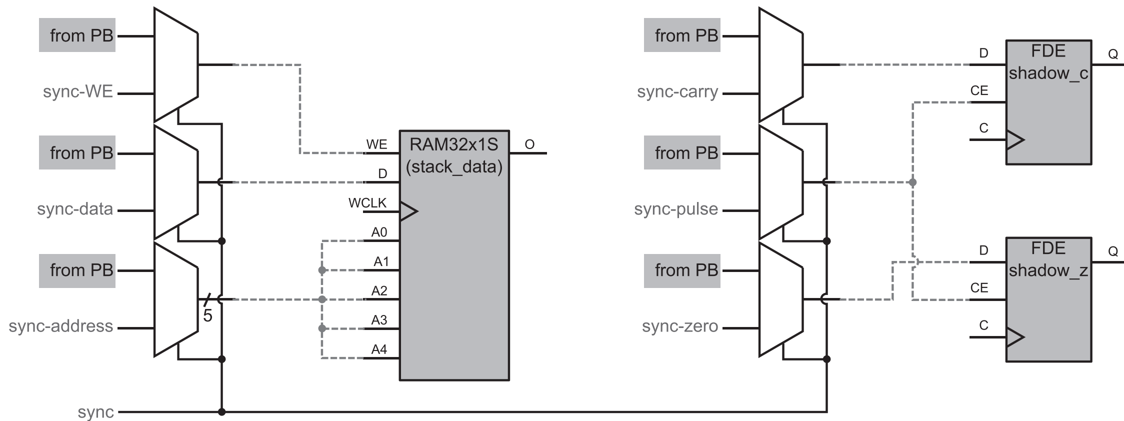

Fig. 9. Implementation of stack-data and flag synchronization (elements with grey background are original Picoblaze design, the muxes and the lightgrey signals have been added).

4.2. Implementation of the hardware synchronization interface

Both approaches 3 and 4 require modifications on the implemented PicoBlaze processor. Fig. 9 illustrates the implementation of this hardware synchronization interface on the example of the stack-data and the shadow flag registers.

The PicoBlaze elements requiring synchronization are divided into two groups:

- First the shadow registers for the flags and the stack pointer, which are elements are implemented as registers on the FPGA. To synchronize these, multiplexers are added for each of these registers, allowing a synchronization time of one-cycle.
- Secondly the processor registers, the stack-data and the scratchpad memory, which are all implemented using block ram (BRAM). Multiplexers and a synchronization counter are implemented for each element enabling the synchronization of one memory element in one cycle.

The synchronization approaches for the different PicoBlaze state elements result in the synchronization times (*copy-times*) given in Table 3.

It can be observed that hardware based synchronization (*Appr.* 4) results in the fastest synchronization. Table 3 reveals furthermore that the scratchpad memory has the biggest impact on the synchronization times. As a consequence big synchronization speed-ups can be achieved when the application allows synchronizing only a subset of the scratchpad memory locations.

Table 3
Cycles requirements for synchronization.

Synchronization object		Appr. 1	Appr. 2	Appr. 3	Appr. 4
Reg.	ALU-flags	–	–	1	1
	Stack-pointer	–	–	1	1
BRAM	Stack-content	–	–	32	32
	CPU registers	–	64[a]	64	16
	Internal memory	–	512[b]	512	64
	Total	–	576	576	64[c]

[a] 16 registers, 2 instructions per register, 2 cycles per instruction.
[b] 64 locations, 4 instructions per location, 2 cycles per instruction.
[c] For HW-based synchronization the overall synchronization time is determined by the highest BRAM synchronization time.

4.3. Test procedure

Partial reconfiguration techniques enable a very straightforward testing method of the proposed synchronization approaches. This test approach is summarized in Fig. 10.

In addition to one functional bitstream for each partition containing a PicoBlaze processor, some corrupted bitstreams are prepared to force certain processor elements out of sync. In Fig. 10 this is showcased for the processor instance *PB0*.

A synchronization test is executed as follows:

1. Configure the complete device
2. Emulate a failure by loading a *bad* partial bitstream
3. Repair the failure by loading *good* partial bitstream

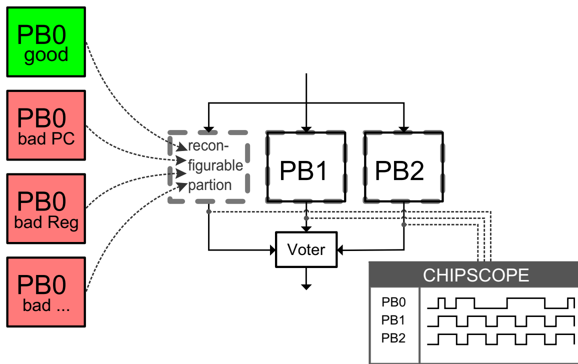

Fig. 10. Test procedure for the synchronization validation.

4. Trigger synchronization routine
5. Check correct synchronization using ChipScope

The use of ChipScope [42] permits to validate the synchronization process on the actual FPGA by monitoring the external processor signals such as the instruction address for observation of the program counter and also internal signals like the stack pointer for validation of *Appr*. 3 & 4. In this work problems with the trigger settings were encountered when using ChipScope for both partial reconfiguration and logic analyzer. To circumvent these issues the synchronization was triggered manually instead of invoking this step automatically after a partial reconfiguration.

As opposed to a complete fault injection (as executed for the PicoBlaze in [43]) the simplified injection method of altered partial bitstreams proposed in this work focuses mainly on the correct synchronization of all synchronization objects presented in Table 1, rather than determining the robustness of the complete setup.

5. Implementation and test results

All four proposed synchronization methods were implemented on a Xilinx Virtex-5 FPGA to validate their correct operation. Fault injection tests were executed as defined in the previous section.

All the functional behaviour was validated and a selection of these test results is displayed in Fig. 11, visualizing the final synchronization step of the tests. The figure contains synchronization details for *Appr*. 2, 3 and 4 in Fig. 11(a)–(c). Within the first seven signals a uniform structure was established for all waveforms. The first three waves contain the instruction addresses (representing the PC) of the three PicoBlazes. In the next three waveforms the value, which was forced out of sync in the test. Finally in the seventh line the synchronized data is displayed, which is written back to PicoBlaze number 0 (which in the tests is always the faulty instance). A visualization of the voting was added in form of an arrow starting from a dashed ellipse containing the values of all three PicoBlazes.

Fig. 11 (a) gives an insight into the synchronization of 4 bytes of scratchpad memory and the PC using *Appr*. 2. This figure's second ellipse marks the different values the PicoBlazes send to the synchronization memory and the arrow marks the voted result, which afterwards is read by all PicoBlazes in the next cycle. The software assembler commands implementing the software based synchronization are given in the tilted boxes. The stack content synchronization, which allows indirectly correcting the program counter while an ISR is processed is illustrated in Fig. 11(b). This hardware based stack synchronization is used in *Appr*. 3 & 4. Finally the last example of Fig. 11(c) contains the completely hardware based register synchronization used in *Appr*. 4. Issuing an interrupt is necessary in *Appr*. 4 although no synchronization software is required. However, the processor needs to enter this state because the processors require to be stalled for register synchronization. The program counter synchronization happens indirectly by the synchronization of the stack.

Comparing Fig. 11(a) and (c) drastic differences in terms of synchronization speed between software based- and hardware based approaches arise. The four instructions used for synchronizing one byte of scratchpad RAM require eight clock cycles. By contrast Fig. 11(c) illustrates the fact that the hardware based copying of BRAM content, such as registers, scratchpad RAM and stack content, requires one clock cycle per byte. In addition to that the hardware synchronization for all different BRAM based elements – e.g. registers and scratchpad – can be performed in parallel.

The drawbacks of hardware based synchronization are unveiled in Table 4, where the FPGA resource requirements of the different implementations are summarized as absolute values and in relation to *Appr*. 1. Correspondingly *Appr*. 4 uses 1.81 times the LUTs resources compared to *Appr*. 1 and the possible operation frequency is reduced to 46%.

The increase in resource requirements is significant, especially if a high degree of hardware synchronization is supported. When implementing a processor as small as the PicoBlaze, adding modules such as the *sync memory* or the *sync FSM* has a stronger relative weight as compared to bigger processors. The VHDL source code of the PicoBlaze provided by Xilinx has been optimized by hand implementing FPGA resources (such as LUT, BRAM, MUXes) directly to fit perfectly the FPGA architecture. When augmenting this hand-optimized design by synchronization logic, the optimization level is deteriorated.

Besides the additional FPGA resource requirements, *Appr*. 3 and 4 undergo a significant impact on the system frequency. This is also a consequence of the hand-optimized VHDL description, which does not allow the synthesis tool to optimize the design efficiently after the synchronization logic is added. But for *Appr*. 4 a certain system frequency drop cannot be avoided, because adding a MUX to the processor registers extends the critical path of the processor.

Based on the trade off between the three factors of the synchronization approaches (the FPGA resource overhead, the synchronization speed and the elements which support synchronization) different applications might benefit from different synchronization techniques. A simple algorithm, such as the Advanced Encryption Standard (AES), does not require a very elaborate synchronization. If the key does not need to be stored, a cyclic reset (*Appr*. 1) might be a sufficient solution. If only a small amount of data needs to be kept on the processor (for example in cyclic sensor readouts) *Appr*. 2 is a suitable option. The last readouts can be kept in the scratchpad memory and the rest of the system can be forced to a sync whenever this is required. Both approaches 3 and 4 represent almost no restrictions to the device software and only minor considerations need to be taken into account for the development of the ISR. Hence almost all applications are candidates to be synchronized using *Appr*. 3 or *Appr*. 4. In very high radiation environments the faster synchronization time (copy-time) of *Appr*. 4 might favour this method despite of its higher resource requirements.

6. Conclusions and future works

In this work the need for synchronization techniques has been underlined when using TMR and partial reconfiguration for building reliable systems. Four different synchronization approaches have been defined so as to cover a wide spectrum of

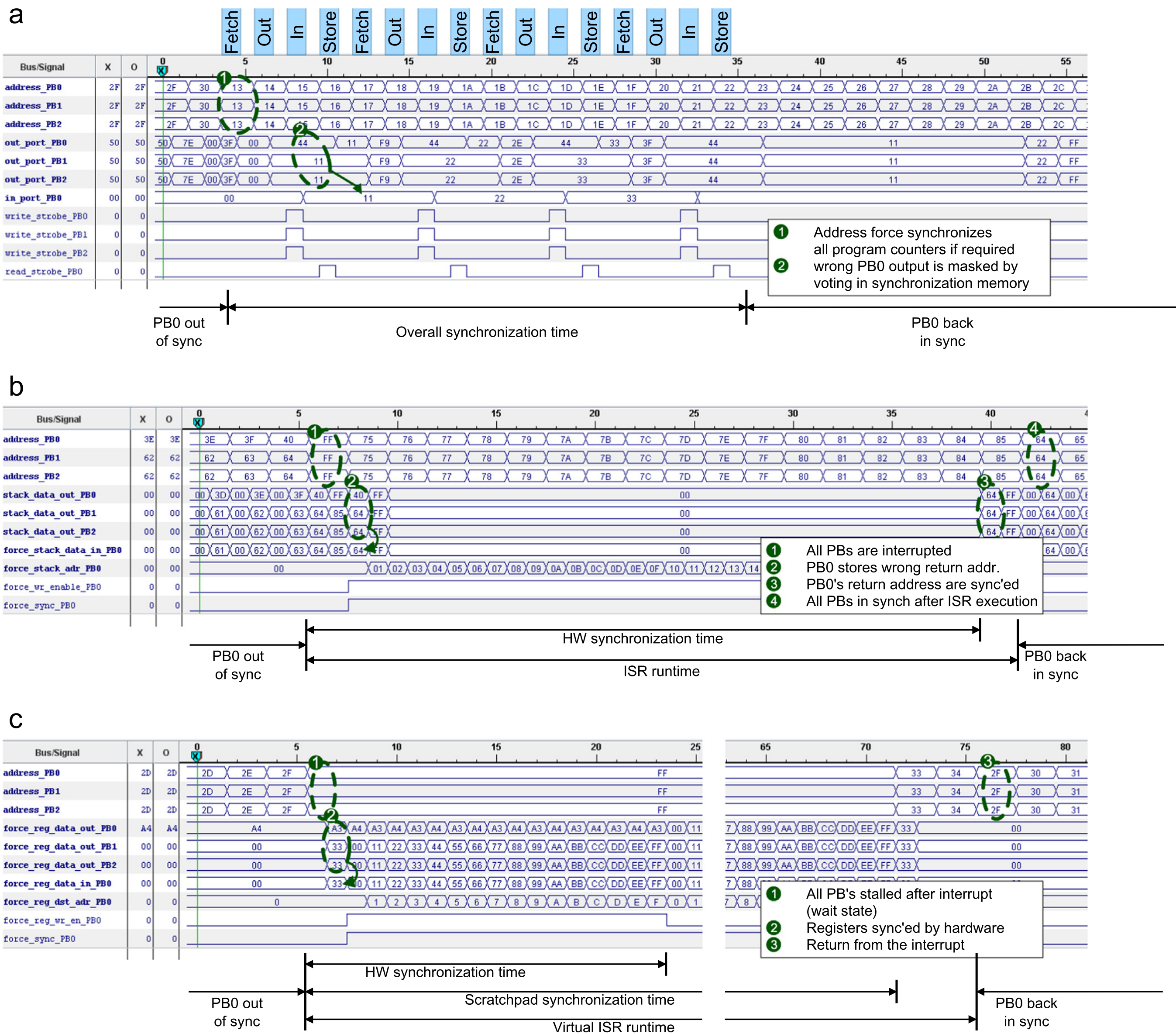

Fig. 11. Screenshots of ongoing synchronizations: (a) shows the synchronization of 4 byte scratchpad memory using *Appr.* 2, (b) shows the stack data synchronization from *Appr.* 3 and (c) shows the register synchronization using *Appr.* 4.

Table 4
Resource consumption on Xilinx XC5VFX70T (without Chipscope, without partial reconfigurable areas).

Resource	Appr. 1	Appr. 2	Appr. 3	Appr. 4
LUT	372 (100%)	470 (126%)	579 (156%)	672 (181%)
Reg	204 (100%)	215 (105%)	251 (123%)	240 (118%)
BRAM	3 (100%)	3 (100%)	3 (100%)	3 (100%)
f_{max}	226 (100%)	211 (93%)	142 (63%)	104 (46%)

synchronization complexity levels. These approaches are, in order of increasing complexity: a minimal hardware approach (*Appr.* 1), a software based memory synchronization (*Appr.* 2), a complete synchronization using a mix of hardware and software (*Appr.* 3) and a completely hardware based approach (*Appr.* 4). The correct operation of all suggested methods has been proven in actual FPGA implementations and the advantages and disadvantages have been thoroughly discussed. Different application scenarios have been suggested depending on the implementation and synchronization features of the proposed approaches.

Future efforts will focus on the application of the synchronization methods in different contexts. Both the synchronization of other processors using these methods and the execution of a complete fault injection campaign on a system of coarse-grained TMR with DPR will be pursued as particularly promising research lines derived from this work.

Acknowledgements

This work has been supported by the Ministerio de Economa y Competitividad of Spain within the project TEC2014-53785-R and it has been carried out inside de Research and Education Unit UFI11/16 of the UPV/EHU and partially supported by the Basque Government within the fund for research groups of the Basque university system IT394-10. Also, FEDER funds are acknowledged.

References

[1] Stitt G, George A, Lam H, Reardon C, Smith M, Holland B, et al. An end-to-end tool flow for FPGA-accelerated scientific computing. Des Test Comput 2011;28 (4):68–77. http://dx.doi.org/10.1109/MDT.2011.46.

[2] Kumar Jaiswal M, Chandrachoodan N. FPGA-based high performance and scalable block lu decomposition architecture. IEEE Trans Comput 2012;61 (1):60–72. http://dx.doi.org/10.1109/TC.2011.24.

[3] Fife W, Archibald J. Improved census transforms for resource-optimized stereo vision. IEEE Trans Circuits Syst Video Technol 2013;23(1):60–73.

[4] Xilinx Corp., EDK concepts, tools and techniques. Xilinx documentation, UG683; April 2012. ⟨http://www.xilinx.com⟩.

[5] Stevens D, Chouliaras V, Azorin-Peris V, Zheng J, Echiadis A, Hu S. BioThreads: a novel vliw-based chip multiprocessor for accelerating biomedical image processing applications. IEEE Trans Biomed Circuits Syst 2012;6(3):257–68. http://dx.doi.org/10.1109/TBCAS.2011.2166962.

[6] Gil AS, Latorre FQ, Calvino MH, Gomez EH, Benitez JB. Optimizing the physical implementation of a reconfigurable cache. In: International conference on reconfigurable computing and FPGAs (ReConFig); 2012. p. 1–6. http://dx.doi.org/10.1109/ReConFig.2012.6416768.

[7] Iturbe X, Benkrid K, Hong C, Ebrahim A, Torrego R, Martinez I, et al. R3TOS: a novel reliable reconfigurable real-time operating system for highly adaptive, efficient, and dependable computing on FPGAs. IEEE Trans Comput 2013;62 (8):1542–56. http://dx.doi.org/10.1109/TC.2013.79.

[8] Bernardi S, Flammini F, Marrone S, Mazzocca N, Merseguer J, Nardone R, et al. Enabling the usage of UML in the verification of railway systems: the dam-rail approach. Reliab Eng Syst Saf 2013;120:112–26. http://dx.doi.org/10.1016/j.ress.2013.06.032.

[9] Brizuela J, Ibañez A, Fritsch C. NDE system for railway wheel inspection in a standard FPGA. J Syst Archit 2010;56(11):616–22. http://dx.doi.org/10.1016/j.sysarc.2010.07.015.

[10] Guilbert D, Guarisco M, Gaillard A, N'Diaye A, Djerdir A. FPGA based fault-tolerant control on an interleaved DC\DC boost converter for fuel cell electric vehicle applications. Int J Hydrogen Energy 2015;40(45):15815–22. http://dx.doi.org/10.1016/j.ijhydene.2015.03.124.

[11] T. Hoppe, S. Kiltz, J. Dittmann, Security threats to automotive CAN networks – practical examples and selected short-term countermeasures. In: Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics), Lecture notes in computer sciences, vol. 5219; 2008. p. 235–48. Cited by 6. http://dx.doi.org/10.1007/978-3-540-87698-4.

[12] Prieto-Alfonso H, Del Peral L, Casolino M, Tsuno K, Ebisuzaki T, Rodrguez Fras M. Radiation hardness assurance for the jem-euso space mission. Reliab Eng Syst Saf 2014;133:137–45. http://dx.doi.org/10.1016/j.ress.2014.08.014.

[13] Park D, Yeom K, Ahn S, Lim H. Validation of an active transponder for KOMPSAT-5 SAR image calibration. Adv Space Res 2014;54(8):1552–62. http://dx.doi.org/10.1016/j.asr.2014.06.016.

[14] Xilinx Corp., Considerations surrounding single event effects in FPGAs, and processors. Xilinx documentation, WP402; March 2012. ⟨http://www.xilinx.com⟩.

[15] Entrena L, Garcia-Valderas M, Fernandez-Cardenal R, Lindoso A, Portela M, Lopez-Ongil C. Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection. IEEE Trans Comput 2012;61(3):313–22. http://dx.doi.org/10.1109/TC.2010.262.

[16] Weulersse C, Miller F, Carrire T, Mangeret R. Prediction of proton cross sections for SEU in SRAMS and SDRAMS using the METIS engineer tool. Microelectron Reliab 2015;910:1491–5. http://dx.doi.org/10.1016/j.microrel.2015.06.117.

[17] Quinn H, Graham P, Wirthlin M, Pratt B, Morgan K, Caffrey M, et al. Methodology for determining space readiness of Xilinx SRAM-based FPGA devices and designs. IEEE Trans Instrum Meas 2009;58(10):3380–95. http://dx.doi.org/10.1109/TIM.2009.2025469.

[18] Xilinx Corp., Device reliability report, fourth quarter 2010. Xilinx documentation, UG116; February 2011. ⟨http://www.xilinx.com⟩.

[19] Xilinx Corp., Xilinx-5 FPGA configuration user guide. Xilinx documentation, UG191; August 2010. ⟨http://www.xilinx.com⟩.

[20] Xilinx Corp., Virtex-5 family overview. Xilinx documentation, DS100; February 2009. ⟨http://www.xilinx.com⟩.

[21] Liu S, Sorrenti G, Reviriego P, Casini F, Maestro J, Alderighi M, et al. Comparison of the susceptibility to soft errors of SRAM-based FPGA error correction codes implementations. IEEE Trans Nucl Sci 2012;59(3):619–24.

[22] Jacobs A, Cieslewski G, George A. Overhead and reliability analysis of algorithm-based fault tolerance in FPGA systems. In: 22nd International conference on field programmable logic and applications (FPL); 2012. p. 300–6. http://dx.doi.org/10.1109/FPL.2012.6339222.

[23] Violante M, Meinhardt C, Reis R, Reorda M. A low-cost solution for deploying processor cores in harsh environments. IEEE Trans Ind Electron 2011;58 (7):2617–26. http://dx.doi.org/10.1109/TIE.2011.2134054.

[24] Pham H, Pillement S, Piestrak S. Low overhead fault-tolerance technique for dynamically reconfigurable softcore processor. IEEE Trans Comput 2013;62 (6):1179–92.

[25] Fort A, Mugnaini M, Vignoli V, Gaggii V, Pieralli M. Fault tolerant design of a field data modular readout architecture for railway applications. Reliab Eng Syst Saf 2015:456–62. http://dx.doi.org/10.1016/j.ress.2015.06.008.

[26] Kretzschmar U, Astarloa A, Lazaro J, Garay M. del Ser J. Robustness of different TMR granularities in shared wishbone architectures on SRAM FPGA. In: International conference on reconfigurable computing and FPGAs (ReConFig); 2012.

[27] Xilinx Corp., Xilinx TMRTool user guide. Xilinx documentation; September 2009. ⟨http://www.xilinx.com⟩.

[28] Berg M, Poivey C, Petrick D, Espinosa D, Lesea A, LaBel K, et al. Effectiveness of internal versus external seu scrubbing mitigation strategies in a Xilinx FPGA: design, test, and analysis. IEEE Trans Nucl Sci 2008;55(4):2259–66. http://dx.doi.org/10.1109/TNS.2008.2001422.

[29] Herrera-Alzu I, Lopez-Vallejo M. Design techniques for Xilinx Virtex FPGA configuration memory scrubbers. IEEE Trans Nucl Sci 2013;60(1):376–85.

[30] Nazar G. Improving FPGA repair under real-time constraints. Microelectron Reliab 2015;55(7):1109–19. http://dx.doi.org/10.1016/j.microrel.2015.04.003.

[31] Carmichael C, Bridgford B, Tseng Chen Wei. Single-event upset mitigation selection guide. Xilinx documentation, XAPP987; March 2008. ⟨http://www.xilinx.com⟩.

[32] Paulsson K, Hubner M, Jung M, Becker J. Methods for run-time failure recognition and recovery in dynamic and partial reconfigurable systems based on Xilinx Virtex-II Pro FPGAs. In: IEEE Computer society annual symposium on emerging VLSI technologies and architectures; 2006. p. 6. http://dx.doi.org/10.1109/ISVLSI.2006.62.

[33] Bolchini C, Miele A, Santambrogio M. TMR and partial dynamic reconfiguration to mitigate SEU faults in FPGAs. In: 22nd IEEE International symposium on defect and fault-tolerance in VLSI systems, DFT; 2007. p. 87–95. http://dx.doi.org/10.1109/DFT.2007.25.

[34] Iturbe X, Azkarate M, Martinez I, Perez J, Astarloa A. A novel SEU, MBU and SHE handling strategy for Xilinx Virtex-4 FPGAs. In: International conference on field programmable logic and applications (FPL); 2009. p. 569–73. http://dx.doi.org/10.1109/FPL.2009.5272410.

[35] Ichinomiya Y, Tanoue S, Amagasaki M, Iida M, Kuga M, Sueyoshi, T. Improving the robustness of a softcore processor against SEUs by using TMR and partial reconfiguration. In: 18th IEEE Annual international symposium on field-programmable custom computing machines (FCCM); 2010. p. 47–54. http://dx.doi.org/10.1109/FCCM.2010.16.

[36] Azambuja J, Sousa F, Rosa L, Kastensmidt F. Evaluating large grain TMR and selective partial reconfiguration for soft error mitigation in SRAM-based FPGAs. In: 15th IEEE international on-line testing symposium, IOLTS; 2009. p. 101–6. http://dx.doi.org/10.1109/IOLTS.2009.5195990.

[37] Straka M, Kastil J, Kotasek Z. Modern fault tolerant architectures based on partial dynamic reconfiguration in FPGAs. In: IEEE 13th international symposium on design and diagnostics of electronic circuits and systems (DDECS); 2010. p. 173–6.

[38] Chapman K. PicoBlaze 8-bit microcontroller for Virtex-E and Spartan II/IIE-Devices, Xilinx documentation, XAPP216; February 2003. ⟨http://www.xilinx.com⟩.

[39] Wang X. Partitioning triple modular redundancy for single event upset mitigation in FPGA. In: International conference on e-product e-service and e-entertainment (ICEEE); 2010. p. 1–4. http://dx.doi.org/10.1109/ICEEE.2010.5660842.

[40] Johnson JM, Wirthlin MJ. Voter insertion algorithms for FPGA designs using triple modular redundancy. In: Proceedings of the 18th annual ACM/SIGDA international symposium on field programmable gate arrays. New York, NY, USA: ACM; 2010. p. 249–58.

[41] Harn Hua Ng. PPC405 Lockstep system on ML310. Xilinx documentation, XAPP564; January 2007. ⟨http://www.xilinx.com⟩.

[42] Xilinx Corp., ChipScope Pro 12.1 software and cores. Xilinx documentation, UG029; April 2010. ⟨http://www.xilinx.com⟩.

[43] Kretzschmar U, Astarloa A, Lazaro J, Jimenez J, Zuloaga A. An automatic experimental set-up for robustness analysis of designs implemented on SRAM FPGAs. In: International symposium on system on chip (SoC); 2011. p. 96–101. http://dx.doi.org/10.1109/ISSOC.2011.6089684.