Microprocessors and Microsystems 49 (2017) 64–76

Contents lists available at ScienceDirect

Microprocessors and Microsystems

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

# A novel BRAM content accessing and processing method based on FPGA configuration bitstream

J. Gomez-Cornejo[a,*], A. Zuloaga[a], I. Villalta[a], J. Del Ser[b,c], U. Kretzschmar[a], J. Lazaro[a]

[a] Dept. of Electronics, University of the Basque Country UPV/EHU, Alameda Urquijo S/N, 48013 Bilbao, Spain
[b] OPTIMA Unit. TECNALIA. P. Tecnologico Bizkaia, Ed. 700, 48160 Derio, Spain
[c] Dept. of Communications Engineering, University of the Basque Country UPV/EHU, Alameda Urquijo S/N, 48013 Bilbao, Spain

ARTICLE INFO

*Article history:*
Received 13 June 2016
Revised 15 September 2016
Accepted 30 January 2017
Available online 2 February 2017

*Keywords:*
FPGA
Bitstream processing
BRAM
System on Chip
Scrubbing,

ABSTRACT

This paper presents a new approach to manage data content of memories implemented in FPGAs through the configuration bitstream. The proposed approach is able to read and write the data content from Block RAMs (BRAMs) in FPGA based designs by reading and processing the information stored in the bitstream. Thanks to this method it is possible to extract, load, copy or compare the information of BRAMs without neither resource overhead nor performance penalty in the design. It can also be applied to existing designs without the need of re-synthesizing. Due to its advantages it becomes an interesting tool to carry out several applications, such as error detection and recovery or fault injection. It also opens the doors to the design of cutting-edge applications. The approach has been implemented in a Xilinx ZYNQ System-on-Chip (SoC) device, which combines an FPGA and an ARM9 microprocessor. The access to the configuration bitstream has been performed using the ZYNQ's Processor Configuration Access Port (PCAP). Nevertheless, the flow presented in this article can be adapted to devices from other Xilinx families or vendors. The proposed approach has been fully tested and compared with specifically designed memory controllers. The results obtained in the experimental tests confirm that the proposed approach works properly without increasing the resource overhead but at a penalty in terms of processing time.



## 1. Introduction

New System-on-Chip (SoC) systems which combine reprogrammable fabric logic with integrated embedded processor cores are lately gaining momentum, enabling higher levels of integration, differentiation and flexibility. These hybrid FPGA systems pave the way towards a new concept of hardware and software integration. As a consequence, designers can take advantage of the complex computation power of an embedded processor and the well-known benefits of FPGAs, such as flexibility, performance, low-cost design, short time to market and reconfiguration capabilities. Furthermore, these SoC systems also are also suitable to create custom accelerators that enhance the performance of designs adapted to the requirements of particular applications. One remarkable example of this new SoC concept is the ZYNQ architecture from Xilinx [1], which integrates a dual ARM9 processor with the reconfigurable fabric logic of the Xilinx 7 series FPGAs. Following this idea, the ZYNQ is divided in two main regions: the processing system and the programmable logic. The main advantage of the ZYNQ architecture over previous FPGAs with hard processors is that, thanks to the huge number of traces that connect the processing system to the FPGA fabric, it provides the device with a connection between the processing system and the programmable logic, hence requiring less infrastructure. In this way, the ZYNQ SoC makes it possible to exploit the logic as an auxiliary resource that may be used to increase the performance of the deployed applications.

Memories are essential elements in SoC designs as a standard way to store data on a temporary or permanent basis. This data may have different purposes; in the case of FPGA systems the most common functionality is to store information of specific applications, entire programs or sequences of instructions, program state information and/or configuration information of the device. When using FPGAs, apart from the external memories, memory elements can be implemented using dedicated Block RAM (BRAM) modules or distributed general-purpose logic fabric [2]. BRAMs are the *de facto* design selection if the application stores and manages lots of data. To this end, they are provided with customizable characteristics, such as, data width and depth, single or dual input port, different control ports and protection based Error-Correcting Code (ECC). For the same storage purpose, the Look-Up Tables (LUT) of the slices of an FPGA can be used as distributed RAMs or as Shift

* Corresponding author.
*E-mail address:* julen.gcb@ehu.eus (J. Gomez-Cornejo).

http://dx.doi.org/10.1016/j.micpro.2017.01.009
0141-9331/

Register Lookup Tables (SRL). Distributed memories, such as shift registers, are more appropriate for storing small amounts of data and allow for a faster access to them. Hence, the designer should select the most adequate memory implementation depending on the specifications of the design. Otherwise, an improperly designed application with overestimated storage requirements would make the synthesizer reserve more memory space out of the FPGA logic slices than really needed. It needs to be remarked that the resource overhead comes with a higher power consumption and usually with a performance penalty, due to the increase of the length of critical paths.

The standard way to access and manage data in memories is to use their different ports, such as `data output`, `data input`, `write-enable` and `address`. Evidently, this access method requires a control mechanism to manage the inputs and read the outputs in a coordinated fashion. This is often accomplished by a memory controller, which can be implemented in different ways such as using soft-core or hard processors, specific IP cores or custom Finite State Machine (FSM) based modules. In addition, frequently additional elements are also required, like bus interfaces or auxiliary memories to store processed data. The implementation of all the above elements demands the use of different resources from the FPGA, increasing further the resource overhead. Besides, if it is required to read or write large amounts of data, resources committed to storing such data will be blocked and unavailable for other purposes. For example, it is not possible to read the data from the first two memory addresses while the last ten data addresses are being written.

In order to alleviate this noted overhead in the design of embedded memory resources, this work proposes a novel approach to access and manage BRAM based memories' content using the configuration bitstream. Thanks to the Bitstream Based Approach (BBA), the management of memory resources can be carried out without using any logic resource of the FPGA. Besides, it gives the chance to access the memory even when it is being read or written. Therefore, some of the common drawbacks of classical memory management strategies can be solved by using our proposed BBA approach, such as the increase of utilized resource, the appearance of single points of failure or the lower availability of memory blocks. This and several other advantages unleashes a myriad of possibilities when working with BRAM memories. These aspects are deeply discussed in the next sections.

## 2. Application scope of the proposed approach

In some specific situations it is necessary to work with memories in special ways, apart from the standard data reading and writing procedures based on the management of input and output ports. This section introduces several of those scenarios where BBA offers solutions and where standard procedures are not effective enough or even valid. Apart from the particular benefits for each application, the common advantage in all scenarios using the BBA scheme is the *zero* resource overhead. This mostly yields higher maximum frequencies (shorter data paths), less power consumption (less active elements), better fault tolerance (less susceptible elements) and more resources available for implementation purposes. On the downside, since BBA utilizes partial reconfiguration, which is a time demanding technique. Depending on the application this could be a negative or a negligible aspect.

Fault tolerance applications gather several examples where existing data manipulation techniques face problems when working with memory elements. FPGAs designs are susceptible to space radiation induced faults [3] known as Single Event Effects (SEEs). Among all the SEE, Single Events Upsets (SEUs) are the most common ones [4,5]. SEUs are non-permanent soft errors that can affect FPGAs flipping bits in user memories or in the configuration memory. SEUs located in the configuration memory are the most critical case because they can alter the implemented design by changing the configuration of crucial elements or their interconnections. However, faults in user memories can also be critical depending on the design, because as a result of an upset the entire system or a critical part of it can be unavailable or present a malfunction. In certain cases a single error can be unacceptable, specially in fields related to human safety or very expensive technologies, such as railway, avionic or spacial applications. Hence, in order to harden FPGA designs against faults specific methods mostly based on configuration memory scrubbing and/or software or hardware redundancy have been proposed in the literature [6–10].

### 2.1. State of the art and contributions

The most straightforward method to account for and overcome potential faults is the so-called bitstream scrubbing [11–13] which is performed thanks to the partial reconfiguration capability of FPGAs, by periodically writing a known bitstream with a correct configuration to clean upsets from the configuration memory. The scrubbing is usually performed by a dedicated auxiliary processor (hard or soft) with access to the reconfiguration port of the FPGA. The content of BRAMs and registers is masked in the clean copy of the bitstream used for the scrubbing, because this information changes continuously during the operation. In [14], the memory coherence problem is described, which occurs when configuration data contain user information that is updated by system-level user operations between the configuration readback and writeback operations. It proposes a *dirty-bit* technique which deals with this problem. Although it shows to outperform previous approaches, it also comes along with a performance penalty. Thanks to BBA, which provides the location of the data content in the bitstream, the memory coherence problem can be solved in a straightforward way without any resource overhead. In addition, since the configuration scrubbing is done in runtime without stopping the system, there is no performance penalty. The time demanded by BBA could only affect to the time between consecutive scrubbings, which is less the upper bound for the majority of the applications.

In [15] a user memory scrubbing approach to clean errors in memories is proposed. Unlike the bitstream scrubbing, this user memory scrubbing hinges on the addition of a FSM based module to the design. Thus, this method increases the logic overhead and needs to use a second memory port. Since Xilinx FPGA BRAMs can only be implemented as single or dual-port memories, if a BRAM memory block is already being used as a dual-port memory this approach not be feasible. In contrast to what this research states, by using BBA it is feasible to perform a user memory scrubbing in BRAM based blocks by using the bitstream information. Just as it occurs with the configuration scrubbing, neither resource overhead nor performance penalty are incurred by the design.

In redundancy based hardware techniques the implemented logic is replicated and the error detection and/or correction is done by using a voter or a comparison mechanism. These techniques are useful to make designs more robust against both user and configuration memory upsets. Different levels of hardware redundancy can be adopted. Higher redundancy levels usually provide higher fault tolerance but also a higher usage of resources. For these reasons, a trade-off decision must be made in the design process. The most utilized redundancy level is the Triple Modular Redundancy (TMR) [16,17]. However, in this approach a critical problem may occur if an upset affects the voter, since an erroneous output can be deemed correct. The common way to deal with this issue is to also replicate the voter [18]. Although this solution increases the fault tolerance level of the protected design, there is always a single output that can be a single point of failure. Another relevant drawback of using hardware-based modular redundancy

based techniques is that errors are masked but not corrected. In this context, in [15,19] the application of different fault tolerance techniques in user memory elements have been thoroughly analysed. Besides modular redundancy based techniques, Error Correction Codes (ECC) or/and Error Detection Codes (EDC) have been also studied to harden memories. These techniques also increase the usage of resources, and similarly to the redundancy based methods, ECC based approaches only mask errors. In this way, the use of BBA scheme avoids single points of failure, performing the voting process of data contents in the processor that reads the bitstream. Which entails a significant performance penalty. For these reasons, BBA is advisable to opt for the voting in memory redundancy schemes where power consumption and fault tolerance are crucial and the execution time is not critical, such as space applications.

Another fault tolerance strategy when using Dual Modular Redundancy (DMR) schemes is to use the combination of checkpointing and rollback techniques [20–22]. Since in DMR designs errors can be only detected but not masked nor corrected, the combination of both strategies is the prevailing alternative to recover the target system from an error. Rollback consists of restarting the operation in both used modules from a previous error free state. Checkpointing is used to go back to the previous error free state by periodically saving the correct states. Checkpointing can be done 1) by software, by adding instructions for data saving to the program; or 2) by hardware, by implementing data saving mechanisms. In both alternatives, a memory space to store the saved checkpoint is always required. The checkpointing frequency determines the fault tolerance level: higher checkpointing frequency means higher fault tolerance but also lower both operation speed and availability, because checkpointing requires stopping the system so as to read memory modules and registers. In [23] an approach without frequency penalty was presented; however, it does not override the previously noted resource overhead, but increases it further. In all cases checkpointing requires software or hardware overhead. In this context, the BBA proposed in this paper is an interesting alternative to perform both checkpointing and rollback in the memories of the design without changing the original design and resource overhead. As a downside, the time demanded by BBA limits the maximum checkpointing frequency. The relevance of this drawback depends on the design requirements.

When using redundancy based methods, a widely established practice is to divide the design in reconfigurable modules [24,25]. When an error is detected in one of the modules, it can be corrected using the partial reconfiguration capability of FPGAs. The synchronization of the reconfigured module with the rest of the system after a recovery is also a common issue, specially in soft-core processor based systems. This synchronization has been studied in [26–30]. BBA is an interesting alternative to synchronize data memories and also can repair damaged program memories based on BRAMs using the memory scrubbing. Bearing in mind that the damaged soft-core processors are fixed using partial reconfiguration, the performance impact of BBA can be reduced if a part of the data content is copied directly to the correct partial bitstream when reconfiguring the faulty processor.

Another interesting application related with data manipulation in BRAMs and fault tolerance is the emulation of SEUs by injecting errors in the bitstream of the design under test. Techniques based on this concept are a valuable, computationally efficient alternative to expensive physical error injection based methods [31–36] conventionally used to evaluate fault tolerance. Xilinx provides a so called Soft Error Mitigation Controller (SEM) [37] that can be used to inject, detect and correct errors in the configuration memory of the 7 series devices. Nevertheless, it is only suitable for the configuration memory and not for BRAMs or distributed memories. Regarding BRAMs, Xilinx offers a fault injection mechanism for BRAMs with its CORE Generator. However, this mechanism increases the overhead, decreases the performance and it can be also a single point of failure by itself. In this way, fault injection techniques based on the manipulation of the bitstream are common practice in the literature [4,38]. Most of them use logic resources to access the configuration port, generating single points of failures that may impact on the effectiveness of the fault injection. Since BBA uses the Processor Configuration Access Port (PCAP) of the ZYNQ architecture (which does not belong to the reconfigurable logic) any possible damage to the interface controller during the fault injections is avoided. Apart from this avoidance of single points of failure during error injections, BBA infers relevant knowledge about the structure of the BRAM content from within the bitstream. The exploitation of this knowledge improves the precision of fault injections, reduces their time requirements and helps understanding better the obtained results.

The impossibility of modifying the data content of ROM memories, widely used as program memories in soft-core processors [39–41] represents another relevant issue since this functionality may be needed in order to change the purpose of a processor or to recover after a SEU in the program memory. The regular way to modify the content of a ROM memory in a FPGA design is to re-synthesize and re-deploy the entire design with the new data onto the FPGA. In the best case scenario, if the program memory has been implemented as a reconfigurable partition, only this part should be re-synthesized and re-implemented. Following this approach, in [42] the program memory is implemented as a RAM and a dedicated IP core loads new memory content during the system operation by using a serial interface. However, this scheme increases requirements in terms of logic resources, and an upset in the input port of the memory may lead to changes in the memory content, hence increasing the probability of malfunction due to SEUs. With the latest FPGA series (Virtex-4, Virtex-5, Virtex-6, Spartan-3A, Spartan-3AN, Spartan-3A DSP, Spartan-6 and 7 Series), Xilinx offers the possibility to use the Data2MEM [43] data translation software to initialize BRAMs. Among other features, Data2MEM can replace the contents of BRAMs in configuration bitstreams in straightforward fashion, without requiring any implementation tool. However, this application undergoes several limitations: one of the most relevant ones is that this software must to be executed by an operating system (Windows or Linux). In addition, this program requires previously generated files to create the output files, such as BRAM Memory Map (BMM) files or Linkable Format (ELF) files. Finally, the configuration bitstream files to be updated by the tool must be created without compression and/or encryption. To sum up, this is a quite complex solution, not supported for partial bitstreams, feasible only for data writing in BRAMs and not for reading. It is therefore not a proper data management approach for autonomous systems based on partial reconfiguration. By utilizing BBA and (if necessary) making a bit-level analysis of the structure, it is feasible to modify entire programs or single instructions without making any new synthesis or implementation, and without any impact on the hardware overhead. Bearing in mind the time required by the synthesis and implementation procedures, BBA is postulated to be very advantageous in terms of timing.

An additional potential consequence of SEUs is the damage on the reading interface of a memory resource. Such an eventuality could make it impossible to recover the information stored in memories when resorting to conventional, off-the-self methods. This scenario can be a critical issue when the information stored has a special operational relevance. The BBA technique proposed in this work of allows recovering this data content in a straightforward manner.

Besides fault tolerance related scenarios, the approach proposed in this paper paves the way for other avant garde applications. For

**Table 1**
Packet header types (R: reserved bit; x: data bit).

(a) Type 1.

| Header Type | Opcode | Register Address | Reserved | Word Count |
|---|---|---|---|---|
| [31:29] | [28:27] | [26:13] | [12:11] | [10:0] |
| 001 | xx | RRRRRRRRRxxxxx | RR | xxxxxxxxxxx |

(b) Type 2.

| Header Type | Opcode | Word Count |
|---|---|---|
| [31:29] | [28:27] | [26:0] |
| 010 | RR | xxxxxxxxxxxxxxxxxxxxxxxxxxx |

instance, it may serve as a novel method to store data under privacy requirements. Since by using BBA, memories can be managed without using any port, the existence of such memory resources can be hidden in the design, implementing them as isolated memory blocks that can be accessed with no buses.

Another relevant application is the management with multiprocessor systems shared memory. In order to guarantee the memory coherence and minimize the performance overhead, different protocols have been proposed [44,45]. BBA is an interesting alternative to avoid the use of complex protocols, buses and interconnections. This could simplify these designs by enabling new ways of exchanging data between processors. It even makes it possible for one processor of the system to change or adjust the functionality of another by only changing a number of instructions of its program memory.

## 3. Discovering the bitstream structure of data in BRAM

Bearing in mind that this work's approach is based on a Xilinx Z-7020 ZYNQ-7000 All Programmable SoC and makes use of its reconfiguration capability, some aspects of the configuration bitstream of the Xilinx 7 series family's must be introduced.

The information around the bitstream is limited, specially since 1998 when the XC6200 series were discontinued [46]. Vendors have published scarce information about the bitstream composition of their devices, but the majority of them is still unknown. Thanks to different studies the knowledge about the bitstream continues growing over the last years [47,48], however there is a lack of studies dealing with the new Xilinx 7 series devices.

In [49] a bitstream generator that does not use reverse-engineering is presented. This generator is able to imprint small changes onto existing bitstreams and to embed the bitstream generation inside the system that it targets. Although it does not support the full set of device resources, it is a good example of the potential of this field. In [50] a bitstream manipulation tool for Xilinx FPGAs based on C language and capable of create partial bitstreams was presented. Although this work was implemented in a Xilinx Virtex II Pro FPGA, several aspects of the structure of the bitstreams exposed in this contribution are useful to understand the bitstream's composition of Xilinx 7 series FPGAs. Following this baseline, this section presents several interesting aspects of 7 series bitstream based on previous literature, Xilinx's documentation [51,52] and several off-line tests performed within for this research work.

The 7 series configuration memory is organized in frames that are tiled about the FPGA. A frame is the smallest addressable segment of the configuration memory and, for this reason, all the operations have to act upon whole configuration frames. The frame size is 101 words (32 bits), but may vary across different families. Frames are organized with an interleaving mechanism, which imposes that adjacent bits do not belong to the same frame. This is simple a protection strategy against SEUs in consecutive bits, avoiding two errors in the same frame, but also making more difficult the inference of the location of the data content in the bitstream. There is a correspondence between the information in the bitstream and the tile map of the device. Therefore, the number of frames of the bitstream depends on the configuration array size and the additional options of the bitstream. In most cases of the different FPGA's resources, the mapping of memory addresses to resources is unknown because this relation is usually extremely complex. However, some specific resources such as CLBs, DSP blocks and BRAM columns are represented explicitly in the bitstream addressing. The efforts in this work have been focused on understanding the distribution of data in BRAM columns.

**Table 2**
Frame address register format.

| Address type | Bit index |
|---|---|
| Block type | [25:23] |
| Top/Bottom bit | 22 |
| Row address | [21:17] |
| Column address | [16:7] |
| Minor address | [6:0] |

The bitstream file starts with a header consisting of a concatenation of different command words. All commands are executed by reading or writing the different configuration registers of the FPGA. The commands can send two packet types: Type 1 and Type 2 (used to write long blocks). Both packets consist of a header followed by a specific data section. This data section contains the number of 32-bit words specified by the word count portion of the header. Table 1 presents the format of the header of each packet type. The opcode field defines the function mode, which can be *read, write* or *no operation*. The register address field in the Type 1 packet indicates the address of the target register of the operation at hand. The Type 2 does not use the register address field, because it must always follow a Type 1 packet and it uses its packet address. Between the available 20 registers, the most interesting ones for this research are the Cyclic Redundancy Check (CRC), the Frame Address Register (FAR), the Frame Data Input (FDRI), the Frame Data Output (FDRO) and Command (CMD). The data stored in the CRC register is used to carry out error detection over the bitstream data. The FAR is used to set the address of the FPGA logic over which the reading or writing process is performed.

As shown in Table 2 the FAR register is split into 5 parts: block type (CLB, I/O, CLK, BRAM content or CFG_CLB), top/bottom bit, row address, column address and minor address. The FDRI register is used to configure frame data at the address specified in the FAR register. The FDRO register provides readback data for configuration following the address stored in the FAR register. Finally, the CMD is used for the different commands, such us, RCRC (CRC reset), START (FPGA initialization) and DESYNC(end of configuration to desynchronize the device).

The first words of the bitstream header define the boot sequence and provide additional information for synchronization purposes. The next commands specify the width of the configu-

ration bus. Then, a set of optional commands specify parameters related with transfer bitrate, encryption, authentication and CRC check. The last part of the header defines the address in the configuration memory and the amount of words that are going to be loaded. The header is followed by the body, which contains the information that is stored in the device. The structure of this is usually not specified by the FPGA manufacturers, and the only way to obtain information is reverse engineering. Finally, the trailer defines the finishing sequence, which contains CRC verification, the start command (optionally) and finally the DESYNQ command to end the communication. Along the entire bitstream many no operation (0x20000000) words are included for synchronization purposes.

Bearing in mind that the FPGA configuration can be complete or partial, complete or partial bitstreams are accordingly produced. Both options share many common aspects, but several particular differences deserve special attention. To begin with, the complete bitstream uses the START command for starting the device, which it is not necessary for its partial counterpart. This occurs because the partial reconfiguration is usually performed dynamically, i.e. while the device is running. In the case of a complete bitstream, the FAR is filled with 0's and the number of words stored in the FDRI register corresponds to the total words of the entire bitstream. The FAR and number of words of the bitstream are strictly related to the defined reconfigurable region. As a result, if the reconfigurable region is deployed over no consecutive zones following the address of the FAR, the bitstream is divided in different fragments. Each fragment is defined by its particular FAR and word amount. Therefore, the partial bitstream indicates the FAR and word amount of each resource column used in the design. For this reason, partial bitstreams have been used in this work to extract information about the data content of BRAMs.

The word amount stored in the FDRI register is a multiple of the frame size. Each frame is related to the configuration bits and initial values of a particular column of resources. Each resource may have a different number of frames, and these values may also vary among device families. In the case of the Z7020 device used in this study, the total number of frames is 10,008. After several partial configuration experiments it has been observed that each column of slices includes 36 frames, each column of DSPs contains 28 frames and each column of BRAMs embeds 28 configuration frames and 128 data content frames.

The Z7020 device has total number of 133 CLB columns, 11 DSP48 columns and 14 BRAM columns. Hence, the total number of configuration frames in this device is 5502 and the total number of BRAM data frames is 1972. Given the total number of frames of the complete bitstream it can be assumed that the remaining frames are used for the configuration of another resources, such as IOBs or BUFGPs. However, since these resources can not be used in partial bitstreams it is complex to determine the exact number of frames that they posses.

Reconfigurable FPGAs are programmed by loading a configuration bitstream into the internal configuration memory. The 7 series FPGA devices can program themselves from an external memory or can be configured by an external device (e.g. a microprocessor or a PC), through special configuration pins. These pins serve as an interface for different configuration: master and slave serial configuration configuration, master and slave SelectMAP parallel configuration, JTAG/boundary-scan configuration, master serial and byte peripheral interface flash configuration. Moreover, the ZYNQ can reconfigure itself by the programmable logic, or instead by any other processing system through the device configuration interface (DevC). This interface is supported by a dedicated DMA controller capable of transferring bitstreams from an external memory through the Processor Configuration Access Port (PCAP). The 7 series FPGAs also have an Internal Configuration Access Port (ICAPE2) [53], which provides the user logic with access to the 7 series FPGA configuration interface. In [54] both existing internal configuration interfaces, PCAP and ICAPE2, have been extensively evaluated in other to select the most convenient alternative. An additional architecture has been implemented in order to dynamically reconfigure the FPGA without the processing system at the maximum bandwidth of 400 MB/s. In [55], an alternative reconfiguration controller coined as ZyCAP is presented, which improves the reconfiguration throughput in Zynq when compared to standard methods. This controller allows overlapped executions, enhancing as a result the system performance. ZyCAP can be used with soft-processors, but driver software modifications are required. In this research the PCAP has been used, taking advantage of the ARM processor of the ZYNQ for bitstream reading, processing and writing, and avoiding the generation of single points of failure.

### 3.1. Xilinx's BRAMs

BRAMs in Xilinx's FPGAs are very flexible; each BRAM can be configured as a single or dual-port memory and can be set as a single 36 KB memory block or two independent 18 KB memory blocks that share nothing but the stored data. The 36 KB BRAMs are composed by a pair of 18 KB BRAMs (top and bottom). Different 36 KB or 18 KB BRAMs can be chained together to form larger memories. Moreover, when using the dual port configuration, both ports can be implemented with different width. In this way, each port can be configured as 32K× 1, 16K× 2, 8K× 4, 4K× 9 (or × 8), 2K× 18 (or × 16), 1K× 36 (or 32) or 512× 72 (or × 64).

BRAMs are also equipped with an optional ECC protection based on a Hamming code. Each 64-bit-wide block RAM utilizes eight additional Hamming code bits to carry out a single-bit error correction and double-bit error detection during readings. The use of this feature can save logic resources from the FPGA, but also presents several drawbacks as introduced in [23]. The ECC requires two clock cycles to complete the reading process, which may penalize the system with an extra clock cycle lantency in many cases. Furthermore, when the data width is not a multiple of 64, a double-bit upset can point out errors in the unused bits.

Memory elements can be also implemented using distributed resources. For this option Xilinx offers the possibility to capture the state of user registers using the CAPTUREE2 primitive [53]. By virtue of this functionality, current register values can be stored in the configuration memory by triggering an input on this primitive. These register values can be read out from the device along with all other configuration memory contents. The approach presented in [56] is capable of capturing, storing and copying the content of flip-flops within an FPGA device on a particular reconfigurable area by utilizing the bitstream. In a later work [57] the previous approach is extended to enable the copying of the content of flip-flops between different reconfigurable areas by processing captured bitstreams. Although these approaches are implemented in a Virtex V device with a MicroBlaze soft-core processor running a Linux OS, they can be adapted to the scheme presented in this work in a straightforward fashion, thus avoiding the use of soft-core processors and computationally demanding OSs by taking advantage of ZYNQ's SoC capabilities. For this reason, our efforts have been conducted towards on data managing of BRAMs.

### 3.2. Proposed flow to obtain the bitstream structure of data in BRAMs

The aim of this work is to manage different BRAMs of the device by solely resorting to the bitstream. For that purpose it is necessary to read and generate bitstreams containing the data to be loaded in such memories. Since manufacturers do not provide detailed information about the bitstream composition, reverse engineering has been performed so as to find out the structure of data

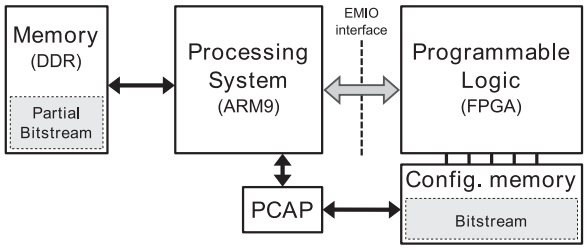



**Fig. 1.** Block diagram of the implementation scheme.

**Table 3**
FAR address of BRAM columns in Z7020.

| Resource | FAR (hex) | Resource | FAR (hex) |
|---|---|---|---|
| BRAM1 | 0x00c20000 | BRAM8 | 0x00c00180 |
| BRAM2 | 0x00c20080 | BRAM9 | 0x00c00200 |
| BRAM3 | 0x00c20100 | BRAM10 | 0x00c00280 |
| BRAM4 | 0x00c20180 | BRAM11 | 0x00800100 |
| BRAM5 | 0x00c20200 | BRAM12 | 0x00800180 |
| BRAM6 | 0x00c20280 | BRAM13 | 0x00800200 |
| BRAM7 | 0x00c00100 | BRAM14 | 0x00800280 |

in BRAMs. In this section, the utilized setup and the different flows and software programs are described in depth.

As has been previously introduced, the target device selected for this work is the Xilinx Z-7020 ZYNQ-7000 All Programmable SoC. Fig. 1 shows a simplified block diagram of the overall scheme utilized. The I/O peripheral signals of the programmable logic and the processing system are connected through the Extended Multiplexed I/O (EMIO) interface, allowing the I/O peripheral controller to access the programmable logic. The processing system reaches the configuration data through the PCAP interface in order to perform a readback, which stands for the process of reading the bitstream data from the configuration memory. The software running in the processing system used for the readback has been developed using the *XDevCfg* library. During the bitstream processing a DDR memory peripheral has been used to store the bitstream and additional application data. This simple approach permits isolating the configuration memory from any other component in the design.

The first step to determine in which words of the bitstream the data content of BRAMs is stored to identify the FAR address of each of the 14 BRAM columns. This is achieved by creating a reconfigurable region for each BRAM column, and thereafter examining the generated partial bitstreams. The FAR addresses of all BRAM columns obtained are shown in Table 3. In the next step each BRAM column has been studied under different implementations within the FPGA section. Considering that each column of the Z7020 contains 20 18K BRAMs, this set of implementations consists of 20 memories of 512 Kb depth and 32 bits each. Using this memory size each 18K-sized BRAM is utilized in its entirely. In order to ease the interpretation of the obtained data, all BRAMs have been specifically arranged by following an increasing order by using location constraints. The implementation also includes a FSM based memory filler module to write the data in all addresses in the memories and a multiplexer to selectively activate the write-enable ports.

Fig. 2 depicts an example of the utilized flow to determine in which words of the bitstream is located the data of each 18K BRAM. Starting with the first (BRAM which is located in the bottom part of the device), the memory filler module writes zeroes in all data positions within the memory. Right after that, the portion of bitstream of the column defined by the specific FAR and the number of frames of a BRAM column are read by using the readback functionality. This content is stored in the DDR memory. Subsequently, a similar process is done but writing all ones

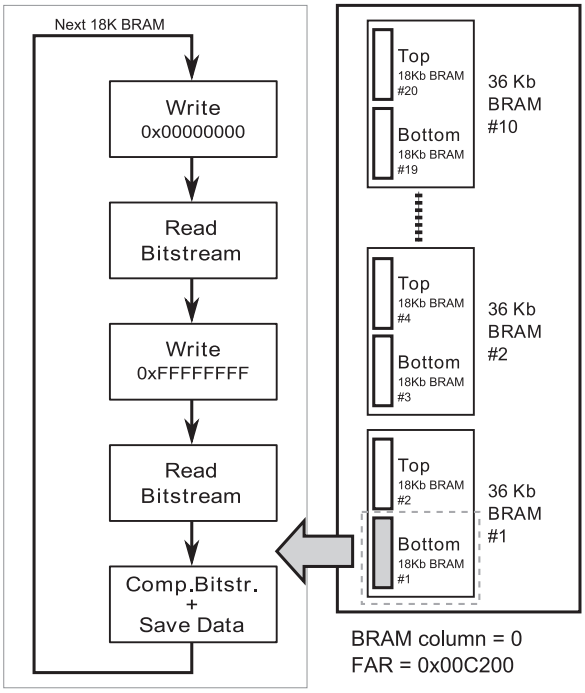



**Fig. 2.** Example of the flow used to determine the BRAM data location in the bitstream.

**Table 4**
Bit distribution example of the first data word of the first 18K BRAM in Z7020.

| N. bit | Word addr. | N. bit in word | N. bit | Word addr. | N. bit in word |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 16 | 2 | 0 |
| 1 | 0 | 16 | 17 | 2 | 8 |
| 2 | 1 | 0 | 18 | 1940 | 0 |
| 3 | 1 | 16 | 19 | 1940 | 16 |
| 4 | 2 | 16 | 20 | 1941 | 0 |
| 5 | 3 | 0 | 21 | 1941 | 16 |
| 6 | 3 | 16 | 22 | 1942 | 16 |
| 7 | 4 | 0 | 23 | 1943 | 0 |
| 8 | 0 | 8 | 24 | 1943 | 16 |
| 9 | 0 | 24 | 25 | 1944 | 0 |
| 10 | 1 | 8 | 26 | 1940 | 8 |
| 11 | 1 | 24 | 27 | 1940 | 24 |
| 12 | 2 | 24 | 28 | 1941 | 8 |
| 13 | 3 | 8 | 29 | 1941 | 24 |
| 14 | 3 | 24 | 30 | 1942 | 24 |
| 15 | 4 | 8 | 31 | 1943 | 8 |

(0xFFFFFFF) instead. The data content obtained in both bitstreams is compared, storing the address and the content of each different word in a spreadsheet. These steps have been performed for every of the 20 memories of each column, after which new implementations have been carried out by changing the location constraints and by using a proper FAR to study all BRAM columns.

Different conclusions have been reached drawn from the analysis of the obtained information. To begin with, it has been confirmed that the content of each BRAM is distributed along the partial bitstream of its BRAM column. This is probably due to the interleaving mechanism. Moreover, there is no direct correspondence between the data bits of an implemented memory and the data bits within the bitstream. Table 4 shows an example of data distribution for a single 32-bit data word over the bitstream. In this plot, the memory is constrained to the first 18K BRAM of the first column and the data content written is all ones. The first column of the table indicates the bit position in the data word. The second column shows the relative position of the word in the data portion of the partial bitstream. Finally, the last column denotes which bit

**Table 5**
Data organization example of one frame of a 18K BRAM column in Z7020.

| Data word address | Data in bottom BRAM | Data in top BRAM |
|---|---|---|
| Init Addr. + (frame number × 101) + 0 | FFFFFFFF | FFFF0000 |
| Init Addr. + (frame number × 101) + 1 | FFFFFFFF | FFFFFFFF |
| Init Addr. + (frame number × 101) + 2 | FFFFFFFF | FFFFFFFF |
| Init Addr. + (frame number × 101) + 3 | FFFFFFFF | FFFFFFFF |
| Init Addr. + (frame number × 101) + 4 | 0002FFFF | FFFFFFFF |

**Table 6**
Init addresses (hex) of 18K BRAMs in Z7020.

| BRAM num. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Init address | 0 | 5 | A | F | 14 | 19 | 1E | 23 | 28 | 2D | 33 | 38 | 3D | 42 | 47 | 4C | 51 | 56 | 5B | 60 |

of the particular word of the bitstream is affected by changes in the memory. The next memory words follow different distribution rules to place their bits in the bitstream. In some cases changes in a single bit of the user memory may lead to changes in two bits of the bitstream. The amount of data bits to handle and inferring the complex relation between them require a big processing effort. Even though working at bit level is interesting in certain applications where the memory space of words to be managed is not so big, in general it is not as practical as working with entire BRAMs. For this reason this approach has focused on managing data content of entire BRAMs instead of controlling changes over single bits or words, hence paving the way towards future developments.

The remaining conclusions related to the location of data words within the bitstream. First, the location scheme of data words in each column is the same. That is to say, all BRAM columns have the same word organization structure to store data within the bitstream. Second, the data content of each 18K BRAM is placed by performing cyclic jumps along the frames of the bitstreams of BRAM columns. These jumps can be noted in Table 5, which depicts an example of the data organization within one frame of the two 18K BRAMs (top and bottom) that compose a 36K BRAM. The data content written in the memories for this example is all ones, thus the data content in all words of the frame is composed of 'F's. The jump sequence begins with an address jump defined by the product of the frame number (relative to the partial bitstream of the BRAM column) and by the initial address of the particular BRAM. Table 6 displays the initial addresses for each 18K BRAM of a BRAM column in the Z7020. The next four addresses are also consecutively used to store data. Thereafter a jump of 101 words to continue in the next frame. As can be also observed in Table 5, the data organization in the top 18K BRAM differs from that in the bottom 18K BRAM. For instance, referring to Fig. 2 the data in the first 18K BRAM is arranged differently when compared to the second or the fourth BRAM, but similar to its third or fifth counterpart. In the case of 36K BRAMs, since they are composed by a 18K BRAM pair (top and bottom), the organization in all of them is the same. Albeit feasible the inference of the relationship between the top and bottom 18K BRAM has been discarded due to its complexity (it is necessary to process at bit level) and subsequent lack of practicability for real applications. In this way, this approach supports the inner data exchange between BRAMs at the same deployment level (i.e. top-top or bottom-bottom).

## 4. Managing memories with the bitstream based approach

As argued in the introductory, BBA presents a great potential for a number of applications. Apart from data reading and writing, the most essential operations required to implement such applications are data copy and comparison. This section delves into these two operations, with emphasis on how to implement them using the bitstream.

### 4.1. BRAM content copy using the bitstream

In this subsection the process to copy the data stored in BRAMs using the bitstream is described. This process can be held in two main scenarios: 1) data copy of entire BRAM columns and 2) data copy of individual BRAMs. Although the concepts of the latter case are also directly applicable to groups of consecutive BRAMs of 18K or 36K, for the sake of simplicity and clarity, this manuscript will refer to individual 18K BRAMs. At this point is also important to recall that, in the case of 18k BRAMs, if the source BRAM is a top 18K BRAM the destination must be also a top BRAM and vice versa. Moreover, it is important to remark that it is not necessary to implement the destination memory in a reconfigurable area of the design, this approach works properly in both, reconfigurable and non reconfigurable areas.

When dealing with single BRAMs, two subcases can be also distinguished depending on whether the data stored in the destination BRAM can be overwritten. Overwriting data is useful when the content of all the BRAMs of the column is not relevant, since this method writes the copied data in the destination BRAM and resets the rest of data bits of the other BRAMs within the column. The second sub-case arises when the content of other BRAMs from the destination column want to be preserved, which comes along with penalties in terms of code and execution overheads. From a global perspective, all the aforementioned cases comply with the flow described in Fig. 3, which hinges on reading the partial bitstream of the source BRAM and copying its data content into the destination BRAM column. The main difference between the three cases resides in the frames reading and data processing tasks.

The first task of the flow consists of creating the header of the new partial bitstream that will be transferred with the copied data. The header of the BRAM column's partial bitstream is composed by 30 words of 32 bits. This header is almost the same for each BRAM column, the difference being that the 27th word of this header contains the FAR address of the destination BRAM column.

After creating the header, the data content must be stored. This task is carried out in two steps that are performed for each frame within the partial bitstream of the BRAM column. First, the frames of the source and, if needed (only necessary when the data overwriting has to be avoided), the destination BRAMs have to be read by utilizing the readback functionality. The data content from the source frame has to be processed in order to organize it properly for the destination BRAM. After performing a readback, several bits of the obtained bitstream are set to 1. These bits must be reset in order to transfer the bitstream successfully. Each of these bits is located in the 18th position of the 4th, 14th, 24th, 34th, 44th, 55th, 65th, 75th, 85th and 95th words of the data part of each frame.

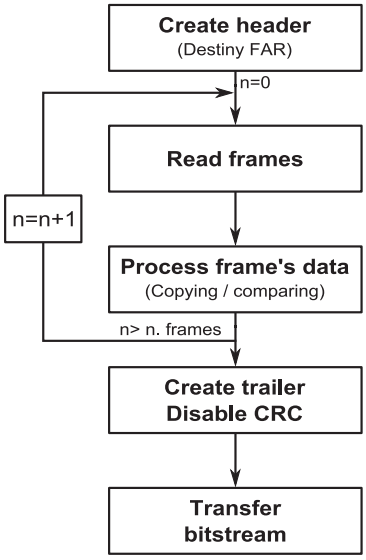



**Fig. 3.** Flow diagram of the BRAM copy procedure.

For this reason, during the processing task all set bits have been reset by resorting to a binary mask.

The specific data processing performed on the data content differs among application cases. Fig. 4 exemplifies the three most basic scenarios: entire BRAM column copy, single BRAM copy overwriting the rest of the BRAMs of the column and single BRAM copy without overwriting. However, based on these cases it is possible to implement more complex applications, such as copying various BRAMs into different locations or swapping data between different columns etc.

The most elementary case is to copy entire BRAM columns. Taking into account 1) that the data organization has found to be equal in all BRAM columns and 2) that the content of the destination BRAM column is not necessary, only the source frames have to be read, copying them directly onto the destination frames. Obviously, in this case all BRAMs from the destination column are overwritten.

In the case of copying single BRAMs with overwriting, the process is slightly different because the data read from the source frame must be reorganized to be placed properly in the destination BRAM. Thus, the exact placing of the source and destination BRAMs must be known. The use of placement constraints is very useful in this context, but it is also possible to determine the placement by analysing the implementation results. Based on the known positions of the BRAMs in the columns and by using Table 6 the initial addresses can be obtained. With this information and following the address jump pattern described in the previous subsection data can be properly arranged and written in the new partial bitstream. It is important to note that if the source BRAM and the destination BRAM are placed in the same column, the content of the source BRAM will be lost.

Copying single BRAMs without overwriting is more complex, specially in terms of program code (in quantitative more than twenty times more lines of code than with overwriting), because it is necessary to take into account several hypotheses. The data of the destination column must be read and copied in the new partial bitstream, and the bit `set` must be reset in all the words where they are set in the readback. By contrast, the cases with overwriting only require resetting the words of the read BRAM because the rest of data words are sent empty. Another singular aspect when copying without overwriting is that if source and destination columns are the same, the program is more efficient since only one readback must be performed for each frame. In any case, the initial addresses of source and destination BRAMs and the address jump pattern must be known beforehand.

After organizing and writing all the data, the trailer must be added in the end of the created partial bitstream. Most of the trailer words mean “no operation” (NOP) words, but it also contains CRC data. Considering that as has been mentioned the CRC method is still unknown [48], these words related to the CRC are written with the NOP word. In this manner, the trailer is the same for all created partial bitstreams. Finally, the produced partial bitstream is transferred to the device via the PCAP port, reconfiguring dynamically and partially the specific logic portion.

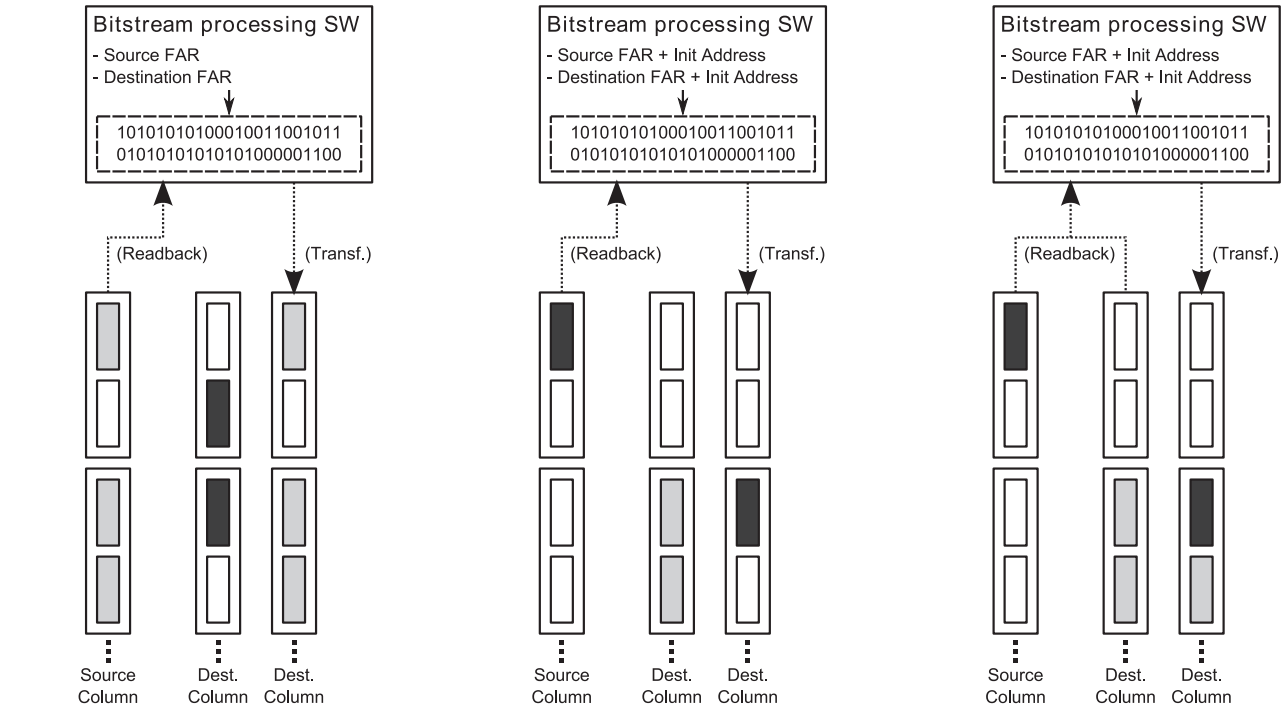
### 4.2. BRAM content comparison using the bitstream

BBA can be also used to compare the data content of different BRAM columns or single BRAMs. In both cases, if the data content is changing during the running process it is advisable to halt the system so also avoid data changes during the bitstream reading process and obtain a correct result. However, in those cases where the information stored in BRAMs remains unchanged for certain time (such as program memories) the comparison can be performed without stopping the system. Depending on the requirements the comparison can be performed in terms of different levels of depth. When the comparison aims at only determining whether the contents of the memories are equal to each other (as in e.g. the detection of SEUs) the detection of the first discrepancy suffices for stopping the comparison process. In other cases, a deeper comparison can be required for e.g. determining the fraction of unequal bits. Despite the fact that different comparison depth levels can be carried out by using our BBA approach, for the sake of simplicity the comparison in this work will be stopped right after the detection of the first discrepancy. To perform this comparison between entire BRAM columns the only information required is the FAR of each BRAM column. By using the readback function all the frames can be read and compared directly.

When working tackling single BRAMs different scenarios can be held. When the BRAMs to be compared are placed in the same position of different columns the way to proceed can be the same to the column comparison. The main difference is that is not necessary to compare all words within each frame. Provided that the initial address and the address jump pattern are known it is possible to know which specific words should be compared. If the BRAMs to be compared are placed in different positions within the same (or different) BRAM columns, the data content has to be processed in the same way that the copying cases in order to compare the right data words. In this case also, comparing BRAMs of the same column is more efficient than comparing BRAM from different column, since it is only necessary to read the partial bitstream of a single column. It must be remembered again that this approach works with top or bottom 18K BRAMs separately. For this reason, top 18K BRAMs are compared with other top BRAMs and vice versa with bottom 18K BRAMs.

## 5. Experimental setup

In order to assess in practice the performance of the proposed approach, a ZedBoard with Xilinx Zynq-7000 All Programmable SoC has been utilized as a experimental platform. The validation of the proposed approach has been carried out by means of several tests based on the scheme presented in Fig. 1. The systems implemented for these tests aim at reading, copying, writing and comparing different data blocks from different BRAM columns, single 18K BRAMs and 36K BRAMs. The performance scores analysed in this study are the resource overhead, the execution time and the




(a) Entire BRAM column copy. (b) BRAM copy with overwriting. (c) BRAM copy without overwriting.

**Fig. 4.** Different copying types.

availability. The ARM processor and the FSM run at 666.7 MHz 100 Mhz respectively.

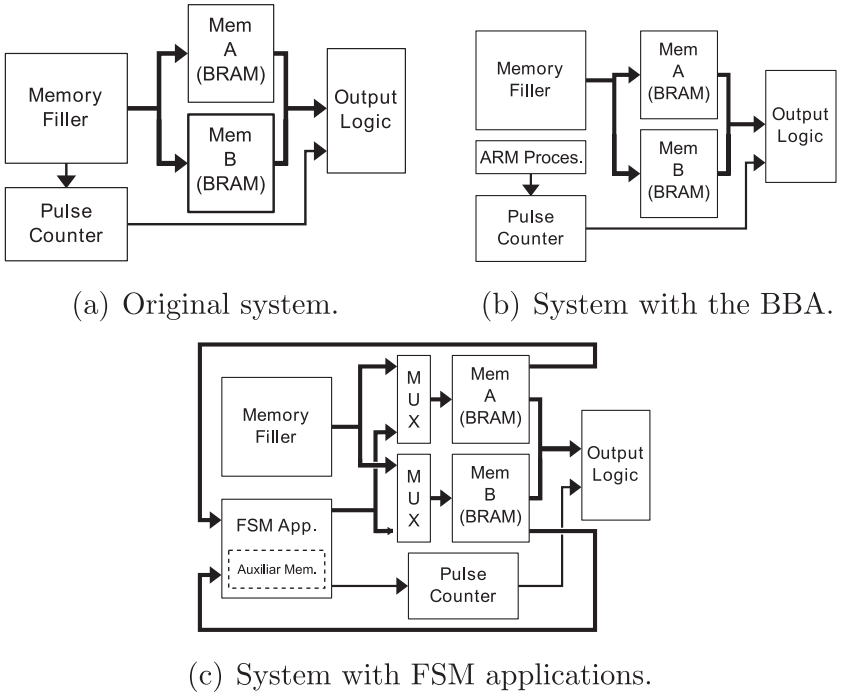
In the next step the bitstream approach has been compared with a common implementation. As has been argued before, most of the memory controllers are based on the use of processors, previously generated generic IP cores or specifically built FSMs. In order to account for the worst case scenario in terms of execution time, the content written in both memories has been equal so as to make the comparing procedure traverse the entire memory space and hence yield the highest execution time. A dedicated soft-core processor would be a waste of resources, therefore in this case a FSM based controller is the most efficient and fastest method to control the memories.

The implemented test applications are memory copying and memory comparing. For each application, different tests have been performed including entire BRAM column and single BRAM experiments. Both applications have been designed to be commanded by a `ready` signal, which permits choosing when the action at hand is executed over the second memory. In this way both approaches (the FSM based and the bitstream based) feature the same functionality. However, in all tests the `ready` signal has been set to *active* so as to avoid the effect of the waiting time in the measurement of processing time. All applications begin by fully writing the source memory, which has not been taken into account when measuring the processing time. Thereafter, the next step to perform the data copy process is to read the source memory, for which the application waits the `ready` signal from the destination memory. Upon receiving this signal, the data content is copied to the destination memory and cross-checked with the original data of the source memory in order to detect data discrepancies. To this end, once the writing process is done the cross-check starts by reading the first memory. After receiving the `ready` signal from the second memory, the data therein are read and compared to the previously read memory. In order to account for the worst case scenario in terms of execution time, evaluate the highest execution time case, the content written in both memories has been equal. Thanks to that no errors have been detected and all the memory addresses have been analysed during the comparing processes.

Fig. 5 depicts the simplified block diagram of the different testing designs implemented in the FPGA part of the ZYNQ. The original system has been implemented as a reference in order to evaluate the resource overhead. As shown in 5(a), it consists of different modules where the most relevant one are the two memory blocks, the FSM based memory filler, the pulse counter and the output logic. The two memory blocks are the target memories, which are equally implemented with different sizes depending on the test to be addressed. The memory filler is a FSM based module used to fill the memories with specific test data. The pulse counter, which is composed by a counter and FSM, measures the time spent by each approach. In this case this module is not relevant; however it has been included so as to present this functionality from impacting on the resource utilization overhead. Finally, the output logic selects the signals to be displayed as outputs.

As evinced Figure in 5(b), the proposed BBA is similar to the original scheme. This is because it does not need any logic resource. The only difference is that the pulse counter is controlled by the ARM processor, which is only necessary to measure the time. The number of instructions to be executed by the ARM vary depending on the location of the BRAMs and the possibility of data overwriting. Both applications - memory copying and memory comparing - have been implemented with source and destination BRAMs in the same or different column and with or without overwriting.

Fig. 5(c) presents the scheme for the FSM based memory copying and memory comparing operations. In these cases, several further modifications have been incorporated with respect to the




(a) Original system. (b) System with the BBA.

(c) System with FSM applications.

**Fig. 5.** Block diagrams of the logic implementations.

**Table 7**
Results of BRAM data copy and comparison tests.

| | | Slice registers | Slice LUTs | BRAMs | Time (ms) |
|---|---|---|---|---|---|
| Orig. | BRAM | 88 | 84 | 2 | – |
| | BRAM columns | 110 | 529 | 40 | – |
| FSM | BRAM copy | 135 (53%) | 235 (80%) | 3 (50%) | 0.001 |
| | BRAM column copy | 165 (50%) | 703 (33%) | 60 (50%) | 0.02 |
| | BRAM comp. | 111 (26%) | 198 (35%) | 3 (50%) | 0.001 |
| | BRAM column comp. | 141 (28%) | 614 (16%) | 60 (50%) | 0.02 |
| Bitstream | BRAM copy(Same column / Overwr.) | 88 (0%) | 84 (0%) | 2 (0%) | 3.6 |
| | BRAM copy(Same column / No Overwr.) | 88 (0%) | 84 (0%) | 2 (0%) | 4.6 |
| | BRAM copy(Dif. column / Overwr.) | 88 (0%) | 84 (0%) | 2 (0%) | 3.6 |
| | BRAM copy(Dif. column / No Overwr.) | 88 (0%) | 84 (0%) | 2 (0%) | 4.6 |
| | BRAM column copy | 110 (0%) | 529 (0%) | 40 (0%) | 4.3 |
| | BRAM comp.(Same column) | 88 (0%) | 84 (0%) | 2 (0%) | 0.27 |
| | BRAM comp.(Dif. column) | 88 (0%) | 84 (0%) | 2 (0%) | 0.47 |
| | BRAM column comp. | 110 (0%) | 529 (0%) | 40 (0%) | 1.25 |

original scheme. The most remarkable one is the addition of the FSM application module. For each test (copying and comparing) a specific FSM has been implemented, minimizing the number of states for an efficient coupling. In order to wait the `ready` signal, an auxiliary BRAM memory has been allocated in these modules. Both FSM applications also manage the pulse counter module. Finally, additional multiplexer modules have been added to manage the inputs of the target memories. As opposed to BBA, when using FSMs there is no functional difference related to the location of the BRAMs, nor any the problem of data overwriting of the rest of BRAMs from the same BRAM column.

Table 7summarizes the results of copying and comparing tests. In both test the analysed parameters are the same. The first five columns (Slice Registers, Slice LUTs, Occupied Slices, LUT FF pairs and 18K BRAMs) quantify the usage of resources of the FPGA. The number in parenthesis indicates the relative resource overhead with respect to the original system. The remaining parameter is the time used to perform each application extracted from the pulse counter module. It should be noted that since the processing system presents a non-deterministic response (due to uncontrollable side conditions e.g. hardware interruptions, hardware-software communication...) there are small variations in the measured time for each experiment. Consequently, the provided scores represent averaged values after 10 experiments. In the same table, the first two rows correspond to the parameters of the original system without any additional element which these result are used as a reference. In this case, since no data operation is involved, there is no information for the time parameter. The next four rows are the values of the monitored parameters corresponding to the FSM based applications. Last but not least, the results of the bitstream manipulation based applications are shown in the last set.

The conclusions drawn by observing the above table support the research hypothesis stared in Section 1. The BBA based applications obtain significantly better results in terms of resources overhead. While the overhead in BBA is essentially zero, the overheads of the FSM based approaches are between 50% and 180%. However, the results also confirm that the BBA based applications are the most time consuming options in the benchmark, specially in the case of the copying as because the transmission of the 128 frames of a BRAM columns takes about 3.3 ms. Although it could

be possible to reduce this time by optimizing the bitstream transfer functions, there is a limitation imposed by the maximum PCAP frequency (100 MHz). This limitation of the maximum PCAP frequency has a direct impact on the velocity of the bitstream writing and reading processes, since it limits the communication speed between the bitstream processing block and FPGA. The increase of this maximum frequency would be an interesting improvement, since it could increase the execution speed of the approach. Moreover, it has been also confirmed via off-line experiments that the processing time varies depending on the relative location of the BRAMs and the possibility of overwriting the information of the destination memory, hence elucidating that the quickest cases are copying and comparing BRAMs allocated in the same column.

## 6. Conclusions and future work

In this work a novel approach to access and manage data in BRAM based memory designs for FPGA SoC implementations has been presented. This approach utilizes the accessing to the configuration bitstream to manage the data content of different BRAMs. Since this method does not add any additional element to the original design, there is no impact on the resource overhead. This is relevant because the resource overhead involves longer datapaths, more power consumption, worst susceptibility and less resources available. In addition, thanks to the use of dynamic partial reconfiguration this new data management strategy can be done in runtime. For this reason, despite its higher time demand the proposed technique does not degrade the system overall performance in a number of applications. Finally, by virtue of the use of the PCAP it avoids single point of failures. Based on this rationale, this new methodology to work with memories improves a number of applications, specially in scenarios related with fault tolerance, i.e. memory scrubbing, error detection, synchronization of reconfigured modules in redundancy based schemes, checkpointing, rollback, fault injection or recovering information from memories with damaged reading interfaces. It also permits modifying program memories without synthesizing any part of the design. In addition, the proposed scheme can be interesting for other special applications such as the management of shared memory multiprocessor systems without using any logical interconnection, or the concealing of memory modules in the design.

Since the weakest aspect of BBA is its higher demand of time, future work will be focused on improving this aspect. A possible strategy in this direction could be to enhance the program functions for the bitstream reading and transfer. In its current implementation the bitstream reading is done frame by frame, and consequently is responsible for a significant fraction ot the overall processing time of the proposed method. Therefore, the optimization of the functions to read more than one frame at once (i.e. bulk read) should make the system faster. Another possible direction to improve the velocity of the approach would be to adapt the system to use the ZyCAP [55], since it is reported that this approach achieves a reconfiguration throughput of 382 MB/s, improving over PCAP by almost 3x.

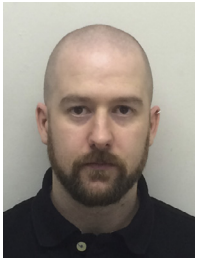
## Acknowledgments

This work has been carried out within the Research and Education Unit UFI11/16 of the University of the Basque Country UPV/EHU and supported by the Department of Education, Universities and Research of the Basque Government within the fund for research groups of the Basque University System IT394-10.

## References

[1] Xilinx Corp., Zynq-7000 all programmable SoC technical reference manual UG585 (v1.9.1), 2014, (Xilinx Documentation, http://www.xilinx.com).
[2] M. Bales, Xilinx implementation tutorial, in: IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2007.
[3] P. Dodd, L. Massengill, Basic mechanisms and modeling of single-event upset in digital microelectronics, IEEE Trans. Nucl. Sci. 50 (3) (2003) 583–602.
[4] U. Legat, A. Biasizzo, F. Novak, SEU recovery mechanism for SRAM-based FPGAs, IEEE Trans. Nucl. Sci. 59 (5) (2012) 2562–2571.
[5] L. Entrena, A. Lindoso, M. Valderas, M. Portela, C. Ongil, Analysis of SET effects in a PIC microprocessor for selective hardening, IEEE Trans. Nucl. Sci. 58 (3) (2011) 1078–1085.
[6] A. Sari, M. Psarakis, D. Gizopoulos, Combining checkpointing and scrubbing in FPGA-based real-time systems, in: IEEE VLSI Test Symposium (VTS), 2013, pp. 1–6.
[7] N. Avirneni, A. Somani, Low overhead soft error mitigation techniques for high-performance and aggressive designs, IEEE Trans. Comput. 61 (4) (2012) 488–501.
[8] Z. Qian, Y. Ichinomiya, M. Amagasaki, M. Iida, T. Sueyoshi, A novel soft error detection and correction circuit for embedded reconfigurable systems, IEEE Embed. Syst. Lett. 3 (3) (2011) 89–92.
[9] Y. Ichinomiya, M. Amagasaki, M. Iida, M. Kuga, T. Sueyoshi, Improving the soft--error tolerability of a soft-core processor on an FPGA using triple modular redundancy and partial reconfiguration, J. Next Gener. Inf. Technol. 2 (3) (2011) 35–48.
[10] S. Rezgui, G. Swift, K. Somervill, J. George, C. Carmichael, G. Allen, Complex upset mitigation applied to a re-configurable embedded processor, IEEE Trans. Nucl. Sci. 52 (6) (2005) 2468–2474.
[11] A. Sari, M. Psarakis, Scrubbing-based SEU mitigation approach for systems-on-programmable-chips, in: International Conference on Field-Programmable Technology (FPT), 2011, pp. 1–8.
[12] A. Vavousis, A. Apostolakis, M. Psarakis, A fault tolerant approach for FPGA embedded processors based on runtime partial reconfiguration, J. Electron. Testing: TheoryAppl. (JETTA) (2013) 1–19.
[13] Z. Jing, L. Zengrong, C. Lei, W. Shuo, W. Zhiping, C. Xun, Q. Chang, An accurate fault location method based on configuration bitstream analysis, in: NORCHIP, 2012, pp. 1–5.
[14] W.J. Huang, E.J. McCluskey, A memory coherence technique for online transient error recovery of fpga configurations, in: ACM/SIGDA International Symposium on Field Programmable Gate Arrays, ACM, 2001, pp. 183–192.
[15] N. Rollins, M. Fuller, M. Wirthlin, A comparison of fault-tolerant memories in SRAM-based FPGAs, in: IEEE Aerospace Conference, 2010, pp. 1–12.
[16] K. Siozios, D. Soudris, A low-cost fault tolerant solution targeting commercial FPGA devices, J. Syst. Archit. (2013) 1255–1265.
[17] H. Pham, S. Pillement, S. Piestrak, Low overhead fault-tolerance technique for dynamically reconfigurable softcore processor, IEEE Trans. Comput. 62 (99) (2013) 1179–1192.
[18] F. Veljkovic, T. Riesgo, E. de la Torre, Adaptive reconfigurable voting for enhanced reliability in medium-grained fault tolerant architectures, in: NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2015, pp. 1–8.
[19] J. Gomez-Cornejo, A. Zuloaga, U. Kretzschmar, U. Bidarte, J. Jimenez, Study of implementation alternatives for a lockstep approach in FPGA soft core processors, in: Conference on Design of Circuits and Integrated Systems (DCIS), 2013.
[20] M. Amin, A. Ramazani, F. Monteiro, C. Diou, A. Dandache, A self-checking hardware journal for a fault-tolerant processor architecture, Int. J. Reconfig. Comput. (2011) 11:11–11:11.
[21] F. Abate, L. Sterpone, C. Lisboa, L. Carro, M. Violante, New techniques for improving the performance of the lockstep architecture for SEEs mitigation in FPGA embedded processors, IEEE Trans. Nucl. Sci. 56 (4) (2009) 1992–2000.
[22] F. Abate, L. Sterpone, M. Violante, A new mitigation approach for soft errors in embedded processors, IEEE Trans. Nucl. Sci. 55 (4) (2008) 2063–2069.
[23] J. Gomez-Cornejo, A. Zuloaga, U. Kretzschmar, U. Bidarte, J. Jimenez, Fast context reloading lockstep approach for SEUs mitigation in a FPGA soft core processor, in: Conference of the IEEE Industrial Electronics Society, IECON, 2013, pp. 2261–2266.
[24] X. Iturbe, M. Azkarate, I. Martinez, J. Perez, A. Astarloa, A novel SEU, MBU and SHE handling strategy for Xilinx Virtex-4 FPGAs, in: International Conference on Field Programmable Logic and Applications (FPL), 2009, pp. 569–573.
[25] A. Astarloa, A. Zuloaga, U. Bidarte, J.L. Martin, J. Lazaro, J. Jimenez, Tornado: a self-reparablee control system for core-based multiprocessor CSoPCs, J. Syst. Archit. 53 (9) (2007) 629–643.
[26] S. Tanoue, T. Ishida, Y. Ichinomiya, M. Amagasaki, M. Kuga, T. Sueyoshi, A novel states recovery technique for the TMR softcore processor, in: International Conference on Field Programmable Logic and Applications (FPL), 2009, pp. 543–546.
[27] C. Pilotto, J.R. Azambuja, F.L. Kastensmidt, Synchronizing triple modular redundant designs in dynamic partial reconfiguration applications, in: Symposium on Integrated Circuits and Systems Design (SBCCI), 2008, pp. 199–204.
[28] Y. Ichinomiya, S. Tanoue, M. Amagasaki, M. Iida, M. Kuga, T. Sueyoshi, Improving the robustness of a softcore processor against SEUs by using TMR and partial reconfiguration, in: IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2010, pp. 47–54.
[29] C. Stoif, M. Schoeberl, B. Liccardi, J. Haase, Hardware synchronization for embedded multi-core processors, in: IEEE International Symposium on Circuits and Systems (ISCAS), 2011, pp. 2557–2560.
[30] I. Safarulla, K. Manilal, Design of soft error tolerance technique for FPGA based soft core processors, in: International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), 2014, pp. 1036–1040.

[31] M. Niamat, D. Nemade, M. Jamali, Test, diagnosis and fault simulation of embedded RAM modules in SRAM-based FPGAs, Microelectron. Eng. (2007) 194–203.
[32] N. Kumar, C. Wai Chong, An automated approach for locating multiple faulty LUTs in an FPGA, Microelectron. Reliab. (2008) 1900–1906.
[33] A. Ruan, B. Jie, L. Wan, J. Yang, C. Xiang, Z. Zhu, Y. Wang, A bitstream readback-based automatic functional test and diagnosis method for xilinx FPGAs, Microelectron. Reliab. (2014) 1627–1635.
[34] J. Azambuja, G. Nazar, P. Rech, L. Carro, F. Lima Kastensmidt, T. Fairbanks, H. Quinn, Evaluating neutron induced SEE in sram-based FPGA protected by hardware- and software-based fault tolerant techniques, IEEE Trans. Nucl. Sci. 60 (6) (2013) 4243–4250.
[35] U. Kretzschmar, Estimating the Resilience Against Single Event Upsets in Applications Implemented on SRAM Based FPGAs, Basque Country University UPV/EHU, 2014 Ph.D. thesis.
[36] J. Tarrillo, J. Tonfat, L. Tambara, F.L. Kastensmidt, R. Reis, Multiple fault injection platform for SRAM-based FPGA based on ground-level radiation experiments, in: Latin-American Test Symposium (LATS), 2015, pp. 1–6.
[37] Xilinx Corp., Soft error mitigation controller UG036 (v4.1), 2014, (Xilinx Documentation, http://www.xilinx.com).
[38] U. Kretzschmar, A. Astarloa, J. Lazaro, G. M., Robustness of different TMR granularities in shared wishbone architectures on SRAM FPGA, in: International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2012.
[39] J. Gomez-Cornejo, A. Zuloaga, U. Bidarte, J. Jimenez, U. Kretzschmar, Interface tasks oriented 8-bit soft-core processor, in: Annual FPGAworld Conference, ACM, 2012, pp. 4:1–4:5.
[40] H.Y. Cheah, S. Fahmy, D. Maskell, iDEA: A DSP block based FPGA soft processor, in: International Conference on Field-Programmable Technology (FPT), 2012, pp. 151–158.
[41] F. Merchant, S. Pujari, M. Manish Patil, Platform independent 8-bit soft-core for SoPC, in: International MultiConference of Engineers and Computer Scientists, 2009, pp. 1541–1544.
[42] Z. Hajduk, An FPGA embedded microcontroller, Microprocess. Microsyst. 38 (1) (2014) 1–8.
[43] Xilinx Corp., Data2MEM. user guide UG658 (v2012.4), 2012, (Xilinx Documentation, http://www.xilinx.com).
[44] M.S. Papamarcos, J.H. Patel, A low-overhead coherence solution for multiprocessors with private cache memories, SIGARCH Comput. Archit. News (1984) 348–354.
[45] J. Archibald, J. Baer, Cache coherence protocols: evaluation using a multiprocessor simulation model, ACM Trans. Comput. Syst. (1986) 273–298.
[46] A. Megacz, A library and platform for FPGA bitstream manipulation, in: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2007, pp. 45–54.
[47] F. Benz, A. Seffrin, S. Huss, Bil: A tool-chain for bitstream reverse-engineering, in: International Conference on Field Programmable Logic and Applications (FPL), 2012, pp. 735–738.
[48] R. Chakraborty, I. Saha, A. Palchaudhuri, G. Naik, Hardware trojan insertion by direct modification of FPGA configuration bitstream, IEEE Des. Test 30 (2) (2013) 45–54.
[49] R. Soni, N. Steiner, M. French, Open-source bitstream generation, in: EEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2013, pp. 105–112.
[50] C. Morford, BitMaT - Bitstream Manipulation Tool for Xilinx FPGAs, Faculty of the Virginia Polytechnic Institute and State University, 2005 Master's thesis.
[51] Xilinx Corp., 7 series FPGAs configuration UG470 (v1.9), 2014, (Xilinx Documentation, http://www.xilinx.com).
[52] Xilinx Corp., Partial reconfiguration user guide UG702 (v13.4), 2012, (Xilinx Documentation, http://www.xilinx.com).
[53] Xilinx Corp., Vivado design suite 7 series FPGA and Zynq-7000 all programmable SoC libraries guide UG953 (v2014.4), 2014, (Xilinx Documentation, http://www.xilinx.com).
[54] J. Correa, K. Ackermann, Leveraging partial dynamic reconfiguration on Zynq SoC FPGAs, in: International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014, pp. 1–6.
[55] K. Vipin, S. Fahmy, ZyCAP: efficient partial reconfiguration management on the Xilinx Zynq, Embedded Syst. Lett., IEEE 6 (3) (2014) 41–44.
[56] A. Morales-Villanueva, A. Gordon-Ross, On-chip context save and restore of hardware tasks on partially reconfigurable FPGAs, in: International Symposium on Field-Programmable Custom Computing Machines (FCCM), IEEE, 2013a, pp. 61–64.
[57] A. Morales-Villanueva, A. Gordon-Ross, HTR: On-chip hardware task relocation for partially reconfigurable FPGAs, in: Reconfigurable Computing: Architectures, Tools and Applications, 7806, Springer Berlin Heidelberg, 2013b, pp. 185–196.



**Julen Gomez-Cornejo** received the qualification of Electronic Engineer and the Master degree in Advanced Electronic Systems at the University of the Basque Country UPV/EHU in 2010 and 2012 respectively. From 2007 to 2010 he worked as lecturer in different departments of the University of the Basque Country UPV/EHU. In 2011 he started at the Telecommunications Department of the University of the Basque Country UPV/EHU as a Ph.D. student and Researcher. In 2015, he did a research stay in the Ilmenau University of Technology (Germany). In 2016, he started at the Electrical Engineering Department of the University of the Basque Country as a lecturer. He has participated different in research projects supported by public institutions and private companies. He is author or coauthor off several papers in national and international conferences.

**Aitzol Zuloaga** received the M.S. degree in electronics engineering and M.S. degree in project management from the Simón Bolívar University, Venezuela, in 1985 and 1992, respectively, and Ph.D. degree in telecommunications engineering from the University of the Basque Country, Spain, in 2001. From 1985 to 1995 he worked in R & D for private companies in Venezuela. In 1995, he joined to the University of the Basque Country as a researcher. In 2000 he became Assistant Professor in electronic technology at the Electronics and Telecommunications Department. His main research interests are System-on-Chip and digital communications circuits.

**Igor Villalta** received the qualification of Telecommunication Engineer and the Master degree in Advanced Electronic Systems at the University of the Basque Country UPV/EHU in 2012 and 2014 respectively. In 2013 he started at the Applied Electronics Research team as a Ph.D student. He has taken part in several research projects and he is author of some papers in national and international conferences.

**Javier Del Ser** received his first Ph.D. in Telecommunications at the University of Navarra in 2006, and a second PhD degree (summa cum laude) in Information Technologies at the University of Alcala in 2013. He is currently the technology director of the OPTIMA department at TECNALIA, Spain, and a adjunct lecturer at the Department of Communications Engineering of the University of the Basque Country (EHU/UPV). His research activity gravitates on the use of descriptive, prescriptive and predictive algorithms for data mining, optimization and forecasting in a diverse range of application fields such as Energy, Transport, Telecommunications and Security, amongst others. In these fields he has published over 140 journal articles and conference contributions, supervised 6 Ph.D. theses, participated in over 30 research projects and authored 4 patents.

**Uli Kretzschmar** received the M.S. degree in computer science from the Ilmenau University of technology, Germany, in 2006. After that he worked for four years at Texas Instruments as a system engineer for the low power microcontroller MSP430, responsible for defining new digital modules including a new CPU architecture. In 2010 he joined University of the Basque Country, Bilbao, Spain, where he obtained his Ph.D. degree in electrical engineering in 2014, with the research interests including field programmable gate arrays, fault tolerance, fault injection, tiny microprocessors and reconfigurable computing.

**Jesús Lázaro** received the M.S. and Ph.D. degrees in telecommunications engineering from the University of the Basque Country, Spain, in 2001 and 2005, respectively. In 2001, he started at the Telecommunications Department of the University of the Basque Country as a researcher and lecturer. In 2010, he was a Research Visitor at the Configurable Computing Lab of Virginia Tech. He is member of the Applied Electronics Research Team and co-founder and entrepreneur of System-on-Chip engineering S.L. (soc-e.com). He has participated in more than 30 research projects supported by public institutions and private companies, in six of them as Main Researcher. He is author or coauthor of two patents, 20 articles in scientific international magazines and more than 40 papers in national and international conferences.