LECTURE NOTES IN ELECTRICAL ENGINEERING | 26

Luca Sterpone

Electronics System Design Techniques for Safety Critical Applications

Springer

Electronics System Design Techniques
for Safety Critical Applications

Lecture Notes in Electrical Engineering

Volume 26

For other titles published in this series, go to
www.springer.com/series/7818

Luca Sterpone

Electronics System Design Techniques for Safety Critical Applications

Springer

Luca Sterpone
Politecnico di Torino
Corso Duca Degli Abruzzi, 24
10129 Torino
Italy

ISBN: 978-1-4020-8978-7 e-ISBN: 978-1-4020-8979-4

Library of Congress Control Number: 2008934322

Printed on acid-free paper.

9 8 7 6 5 4 3 2 1

springer.com

To my parents Gianfranco and Primarosa

To my wife Silvia

CONTENTS

CONTRIBUTING AUTHOR

Luca STERPONE, Ph. D. is actually a research assistant in the Department of Automatic Control and Computer Engineering at Politecnico di Torino university, Torino, Italy. He has published widely in the area of dependable systems and fault tolerance techniques and he is involved in research on dependable designs for aerospace and automotive systems as well as innovative biological research for study the fault tolerance and dependable characteristics of genomic.

He is the winner of the EDAA (European Design Automation Association) Outstanding Monograph Award in the Reconfigurable Electronics section in the 2007.

PREFACE

What is exactly “Safety”? A safety system should be defined as a system that will not endanger human life or the environment. A safety-critical system requires utmost care in their specification and design in order to avoid possible errors in their implementation that should result in unexpected system’s behavior during his operating “life”. An inappropriate method could lead to loss of life, and will almost certainly result in financial penalties in the long run, whether because of loss of business or because the imposition of fines. Risks of this kind are usually managed with the methods and tools of the “safety engineering”. A life-critical system is designed to lose less than one life per billion (10^9).

Nowadays, computers are used at least an order of magnitude more in safety-critical applications compared to two decades ago. Increasingly electronic devices are being used in applications where their correct operation is vital to ensure the safety of the human life and the environment. These application ranging from the anti-lock braking systems (ABS) in automobiles, to the fly-by-wire aircrafts, to biomedical supports to the human care. Therefore, it is vital that electronic designers be aware of the safety implications of the systems they develop.

State of the art electronic systems are increasingly adopting programmable devices for electronic applications on earthling system. In particular, the Field Programmable Gate Array (FPGA) devices are becoming very interesting due to their characteristics in terms of performance, dimensions and cost.

FPGAs use a grid of logic gates, based on *gate array* technology, and the programming is done by the customer, not by the manufacturer. The term

"field-programmable" may result obscure to somebody, but "field" is just an engineering term for the world outside the factory built, where the customers live. FPGAs are usually programmed after being soldered. In the most larger FPGAs, such as the RAM-based devices, since the configuration is volatile, their configuration must be re-loaded into the device whenever power is applied or different functionality is required.

During the last decade, the new manufacturing technologies made feasible the development of SRAM-based FPGAs that became very popular thanks to their capability of implementing complex circuits with a very short development time. However, nowadays SRAM-based FPGAs are really not considered enough reliable to be used in safety critical applications such as avionic and space ones. The main obstacle to their applications in these contexts is represented by the high sensitivity to the radiation effects such as Single Event Upsets (SEU): device shrinking coupled with voltage scaling and high operating frequencies correspond to significantly reduced noise margin, which makes FPGAs more sensitive to radiation effects, as well as to other phenomena (such as cross talk or internal noise sources) that provoke transient faults. The strong needs to evaluate the possible applications of the programmable logic devices in safety critical applications need the usage of the new techniques oriented to the evaluation of the reliability of such devices and to the development of hardening techniques for enable the usage of SRAM-based FPGAs in safety critical fields.

The main purpose of the present book addresses the development of techniques for the evaluation and the hardening of designs on SRAM-based FPGAs against the radiation induced effects such as SEUS. The set of analysis and design flows proposed in this work are aimed at defining a novel and complete design methodology solving the industrial designer's needs for implementing electronic systems in critical environments using SRAM-based FPGA devices.

Regarding the analysis flow, the present book contribution consists in a set of algorithms performing the fault injection for the evaluation of the soft-errors sensitivity of designs implemented on SRAM-based FPGAs. Two kind of fault injection environments are provided:

1. *Simulation based*: The simulation environment is able to predict the SEU effects in circuit mapped on SRAM-based FPGAs combining radiation testing data with simulation. The former is used to characterize (in term of device sensibility to the radiation particles) the technology on which the FPGA device is based, the latter is used to predict the probability for a SEU to alter the expect behavior of a given circuit.
2. *Hardware-based*: this environment is able to inject SEU directly in the configuration memory of SRAM-based FPGA devices. The environment is composed of all the module necessary to perform the complete analysis

of the circuit. A *Fault List Manager* generates the list of SEUs to be injected within the circuit under analysis; a *Fault Injection Manager* manages the fault injection process, by selecting one fault from the fault list, performing its injection in the DUT and the observing and analyzing the obtained results to provide the fault-effect classification.

In order to deploy successfully commercially-off-the-shelf (COTS) SRAM-based FPGA devices in safety critical applications, designers need to adopt suitable hardening techniques, as well as methods for validating the correctness of the obtained as far as the system's dependability is considered. An innovative algorithm based on an analytical model of the FPGA architecture is able to estimate the effects of SEUs when redundancy-based techniques are adopted in order to mask the effects of SEUs in SRAM-based FPGAs, has been provided. The main novelty this approach introduces is the possibility it offers of analyzing any SEU location within a design and of identifying whether the SEU provokes any observable effect to the system's outputs. This approach has been implemented in a tool called *STAR* (Static Analyzer).

This book presents also a novel contribution in the FPGA design flow. A new reliability-oriented place and route algorithm is illustrated in details. By coupling its hardening capability with the Triple Modular Redundancy (TMR) it is able to effectively mitigate the effects of soft-errors within FPGA devices especially based on Static-RAM's configuration memory. The effectiveness of the reliability-oriented place and route algorithm has been demonstrated by extensive fault injection experiments showing that the capability of tolerating SEU effects in the FPGA's designs increases up to 85 times with respect to a standard TMR design technique. The developed algorithm has been implemented in a tool called *RoRA*, (Reliability-Oriented Place and Route Algorithm). The available tools STAR and RoRA have been included in a new design tool-chain.

The present book offers a contribute also to the analysis of several applications field where the usage of reconfigurable logic devices introduces several advantages. In particular, two applications are considered: reconfigurable computing for multimedia applications and biomedical applications.

Considering reconfigurable computing, a novel reconfigurable structure has been proposed, also called *Reconfigurable Mixed Grain*, ReCoM. This structure is based on the novel Reconfigurable Compute Fabric (RCF) concept, it implements a mixed-grain reconfigurable array which combines a RISC microprocessor and a reconfigurable hardware for computation-intensive applications.

The feasibility of reconfigurable devices in biomedical applications is also investigated in this book showing the drastic advantages both related to the computational performance and on the dependability of the process.

In this book, the implementation of a new Deoxyribonucleic Acid (DNA) microarray analyzer is provided. DNA microarray technologies are an essential part of modern biomedical research. The analysis of DNA microarray images allows the identification of gene expressions in order to drawn biologically meaningful conclusions for applications that ranges from the genetic profiling to the diagnosis of oncology disease. This book describes an architecture that uses several computational units working in a single instruction-multiple data fashion managed by a microprocessor core. An FPGA-based implementation of the developed architecture has been evaluated using several realistic DNA microarray images. A reduction of the computational time of one order of magnitude and an increasing of the data quality of the analyzed images has been demonstrated.

PART I

Chapter 1

AN INTRODUCTION TO FPGA DEVICES IN RADIATION ENVIRONMENTS

From the architecture to the model

Electronic devices are sensitive to radiation that may happen both in the space environment and at the ground level. Nowadays, the continuous evolution of manufacturing technologies makes Integrated Circuits (ICs) even more sensitive to radiation effects: Devices shrinking coupled with voltage scaling and high operating frequencies correspond to significantly reduced noise margins, which make ICs more sensitive to radiation, as well as to other phenomena (such as cross-talk or internal noise sources) that provoke transient faults.

In the last decade, the new manufacturing technologies made feasible the development of SRAM-based FPGAs that became very popular thanks to their capability of implementing complex circuits with a very short development time. Today, manufacturers are producing very complex and resourceful FPGAs. State-of-the-art SRAM-based FPGAs embed megabits of RAM modules and plenty of configurable logic and routing resources, which are making feasible the implementation of circuits composed of millions of gates. SRAM-based FPGAs are used for different applications, such as signal processing, prototyping, and networking, or wherever reconfiguration capabilities are important.

The architecture of SRAM-based FPGAs is composed of a fixed number of routing resources (wires and programmable switches), memory modules, and logic resources (i.e., lookup tables or LUTs, flip-flops or FFs). All these components are programmed by downloading into an on-chip configuration memory a proper bitstream, giving the FPGA the capability of implementing nearly any kind of digital circuit on the same chip. In SRAM-based FPGA, both the combinational and sequential logic are controlled by several

L. Sterpone, *Electronics System Design Techniques for Safety Critical Applications*,

customizable SRAM cells that are extremely sensitive to radiation that may cause Single Event Upsets [1, 2].

If an upset affects the combinational logic in the FPGA, it provokes a bit-flip in one of the LUTs cells or in the cells that control the routing. This upset has a persistent effect that could be propagated in other parts of the circuit since the implemented hardware is modified. This upset is correctable only at the next load of the configuration bitstream (which is often performed in some critical space applications), but the effect may still remain in the circuit until the next reset is performed. On the other hand, when an upset affects the user sequential logic, it may have a transient effect if the flip-flops next load corrects it and if the effect is not propagated to other parts of the circuit or a persistent effect if the effect is propagated to other parts of the circuit. For instance, a counter hat is affected by an SEU cannot return to its original counting sequence until it undergoes to a reset.

In this case, SEU can have more persistent effects in the implemented user circuit.

SEUs may also affect the configuration control logic registers that are used during the download of the bitstream within the configuration memory. An experimental analysis based on heavy ion beam is described in [3] that shows the criticalities of such registers and that demonstrates that they have a sensitivity to SEUs several orders of magnitude lower with respect to the configuration memory.

The half-latch structures used to generate constant logic values may be also affected by SEUs. This problem has been addressed and fixed according to the work presented in [4], in the presented hardening technique the reliability-oriented placement algorithm is driven in order to solve this problem by means of a technology based placement.

Researchers both from academia and industry investigated on developing solutions able to mitigate the effects of SEUs in the FPGA's configuration memory. These methods could be divided in two main categories: reconfiguration-based and redundancy-based. The formers aim at restoring as soon as possible the original values into configuration bits after an SEU happened [5], the latters are oriented at masking the propagation of SEUs effects to the circuit's outputs [6–[8]. Fault masking techniques are usually achieved through redundancy-based techniques which purpose is to remove all the single point of failure a circuit may have. The widely known redundancy-based technique is the Triple Modular Redundancy (TMR), where three identical replicas of the same circuit work in parallel and the outputs they produce are compared and voted through a majority voter. TMR is an appealing technique for hardening designs implemented on SRAM-based FPGAs. Since all the resources embedded by these devices such as memory

elements, routing resources and logic resources are all susceptible to SEUs, the redundancy technique must be adopted to all of them.

The resources that are most likely to be affected by SEUs are those controlling the routing, indeed about 90% of the configuration memory bits are devoted to storing information about routing resources. Previous works, essentially based on a simulation tool, have experimentally tested the TMR's capability of tolerating SEUs [9]. The criticalities induced by SEUs within the configuration memory provoke an intrinsic behavior to the circuit implemented by the FPGA device. The configuration memory of such devices undergo a detailed analysis of each singular FPGA resource [10, 11] followed by injection experiments [12] able to probe the behavior of each resource induced by the single bit modification. The results gained from these analysis shown that any single modification of a configuration memory cell is capable of producing multiple errors when affecting the portion of the FPGAs configuration memory that stores some kinds of routing and logic resources. Furthermore, the experimental analysis shows that a faulty behavior is produced when a SEU hits either a programmed bit or a non programmed memory bit that may have side effects on the resources configured by the programmed ones. As a result of this effect, the TMR architecture is able to only partially mitigate the effects of SEUs in routing resources. This phenomenon depends on many factors: the architecture of the adopted FPGA family, the organization of the configuration memory, the kind of application that is implemented on the FPGA device, and the bit of the configuration memory affected by the SEU. Given this scenario, redundancy-based techniques are not sufficient by themselves to ensure complete reliability against single-error induced by radiation particles. In order to give a metric to the reader, we considered several benchmark circuits designed according to the TMR architecture and we observed about the 14% of the configuration memory bits upset that affect the portion of the configuration memory storing the information about the routing resources produce multiple errors that the TMR is not able to mask [11]. In this book is presented an analysis of the distribution of SEUs within the FPGA's configuration memory and affecting the TMR behavior. Furthermore, as shown in [13] a clever selection of the TMR architecture helps in reducing the number of escaped SEUs, but it is unable to reduce them to zero.

In order to identify the reasons that limit the effectiveness of TMR, the resources of the FPGA have been systematically analyzed. The case study devices considered by the present research is the Xilinx Virtex family. Independently from the circuit mapped on the FPGA architecture, each FPGA's resource has been analyzed identifying all the possible configuration memory bits controlling its behavior. For example, for a programmable interconnection point, all the possible configuration bits that can be used by the place and

route algorithm are used for implementing any given circuit. The study presented in this book identifies all the critical situations, where SEU hitting the configuration memory may modify the configuration of two or more FPGA's resources. The theoretical explanation and experimental probe of the criticalities affecting circuit implemented through the TMR is the results of this analysis.

After presenting an analysis of the SEU's effects in the FPGA's configuration memory, this part presents a reliability-oriented place and route algorithm, called RoRA, that has been developed for implementing dependable circuits, based on redundancy techniques such as TMR, on SRAM-based FPGAs. The RoRA algorithm is able to place and route the logic functions and the signals of a design in such a way that the number of SEUs affecting the configuration memory and possibly causing FPGA wrong behavior is drastically reduced with respect of a common redundancy-based approach adopting the TMR technique. For the considered benchmark circuits, the capability to tolerating SEU effects in the FPGAs configuration memory increases up to 85 times with respect to a standard TMR approach. In order to achieve an higher level of reliability, the RoRA algorithm introduces penalties both in terms of area overhead and speed of the original circuit. Furthermore, the fulfillment of the routing problem needs more computational time due to the reliability rules inserted both to the placement and routing phases.

The reduction of the circuit's running frequency may range from 22% to 60% of the original (plain) circuit speed, while from the circuit area perspective, RoRA introduces an overhead of the routing resources with respect to the TMR standard solution. However, RoRA does not introduces any area overhead, with respect to the TMR, when logic resources are considered. The RoRA solution is the first place and route algorithm developed that is transparent to designers, which can trade off fault tolerance versus area and circuit's frequency overhead.

1. PREVIOUSLY DEVELOPED HARDENING TECHNIQUES

During the past years, several mitigation techniques have been proposed in order to increase the reliability of circuits of avionics and space applications and in particular, to remove single point of failure from the designs. When SRAM-based FPGA devices are considered, several SEU mitigation techniques have been proposed exclusively for these devices. These techniques can be organized into two categories: reconfiguration-based techniques and

redundancy-based techniques. The former are used to correct fault effects, while the latter are used to mask fault effects.

1.1 Reconfigurable-based techniques

The FPGA's configuration memory, if based on SRAM cells, may accumulate soft error or SEU over the usage time in an harsh environment, for this reason the configuration memory is periodically rewritten. This approach is called *Scrubbing* and it is the simplest technique that may be used to remove SEU effects accumulated within the configuration memory [14]. The implementation of a scrubbing system introduces a limited overhead that essentially corresponds in the circuit needed to control the bitstream loading process, as well as the memory for storing an error-free bitstream. The systems also needs a mechanism to control how often the scrubbing must take place. The occurrence frequency of the scrubbing operations is normally referred to the *scrub rate* and it is determined on the basis of the expected SEU rate, i.e., on the basis of a figure predicting how often an SEU may appear in the FPGA configuration memory.

An improvement of the Scrubbing mechanism consists in applying the partial reconfiguration capability of the latest generation of SRAM-based FPGAs, which allow reconfiguring only a user-selected portion of the configuration memory (known as *frame*) while leaving the remaining part of the circuit unmodified [5]. This technique uses a readback process to read one frame at a time and compares it with the expected one, which is stored in an error-free off-chip memory. Another commonly used technique to detect errors by means of readback is to use Cyclic Redundancy Check (CRC) on each frame storing only the check word rather than the entire frame of the configuration data [5].

When a SEU is detected, only the faulty frame is rewritten. The readback is normally transparent to the circuit the FPGA implements, which continues to operate normally even while the readback process is running. The presence of SEUs is thus checked online and the FPGA is set offline only for the amount of time needed for rewriting the faulty configuration memory frame. The normal activity of the circuit the FPGA implements is stopped for a shorter period of time than in the scrubbing case. The partial configuration mechanism is employed in state-of-the-art Xilinx SRAM-based FPGA devices, such as the Virtex family, with the further advantage that consists in having the possibility to rewrite the configuration data without putting the devices offline. This makes possible online and transparent fault correction. If on one side, the scrubbing and the partial reconfiguration mechanisms represent a simple solution for protecting designs against the effects of SEU, on the other side these techniques are mandatory for adopting SRAM-based FPGA

in the presence of SEU. In fact, these techniques are the only viable solution for removing the accumulation of soft error within the configuration memory, thus whatever is the system used in an harsh environment and embedding SRAM-based FPGAs, it must adopt reconfigurable or scrubbing mechanism in order to avoid the accumulation of SEU within the configuration memory.

1.2 Redundancy-based techniques

The redundancy-based techniques presented in this section adopts additional hardware components or additional computation time for detecting the presence of SEUs modifying the expected circuit operations and/or masking SEUs propagation to the circuits outputs. It is worthwhile to underline here that the techniques presented in this section are not intended for removing SEUs from the configuration memory, but only for mitigating the SEUs effects. SEUs may be removed from the configuration memory by resorting to those techniques presented in the previous section.

Fault detection can be achieved by duplicating the circuit the FPGA implements. The outputs the two replicas produce are continuously compared and an alarm signal is raised as soon as a mismatch is found [14]. This solution is fairly simple and cost-effective; however, it is not able to mask the SEUs effects.

When fault masking is mandatory, designer may resort to the Triple Modular Redundancy (TMR) approach. The basic concept of the TMR architecture is that a circuit can be hardened against SEUs by designing three copies of the same circuit and building a majority voter on the outputs of the replicated circuits. Implementing TMR to prevent the effects of SEUs in technologies such as ASICs is generally applying the protecting capabilities only the memory elements since combinational logic and interconnections are less sensitive to SEUs. When the configuration memory of FPGAs is considered, the TMR implementation should be revisited since a modification in the configuration memory may affect every FPGAs resource: routing resources implementing interconnections, combinational resources, sequential resources, I/O logic. This means that three copies of the whole circuit, including I/O logic, have to be implemented to harden it against SEUs [14].

The optimal implementation of the TMR circuitry inside SRAM-based FPGAs depends on the type of circuit that the FPGA implements. As described in [14], the logic may be grouped into four different types of structure: throughput logic, state-machine logic, I/O logic, and special features (embedded RAM modules, DLLs, etc.). The throughput logic is a logic circuit of any size or functionality, synchronous or asynchronous, where the entire logic path flows from the inputs to the outputs of the module without

ever forming a logic loop. The TMR architecture for a module M is implemented as shown in Figure 1.1.

Three copies of M are connected to a majority voter V, which computes the output of throughput logic. In order to prevent common-mode failures, the inputs feeding the throughput logic have to be replicated, too. This implies that, when M is fed directly from I/O pins, the adoption of TMR must be accomplished tripling the circuit I/O pins.

State-machine logic is, by definition, state dependent. For this reason, it is important that the TMR voting is performed internally rather than externally to such a module. Thus, applying TMR to a state machine consists of tripling all circuits and inserting a majority voter for each of the replicated feedback paths. The use of three redundant majority voters eliminates there as single points of failure, as shown in Figure 1.2.

Hardening the I/O logic through TMR causes a severe increase in the number of required I/O pins and this method can be used only when there are enough I/O resources to achieve tripling of all the inputs and outputs of the design. Therefore, as illustrated in Figure 1.3, each redundant module of a design that uses FPGAs inputs should have its own set of inputs. Thus, if one input is affected by an SEU, it only affects one module of the TMR architecture.

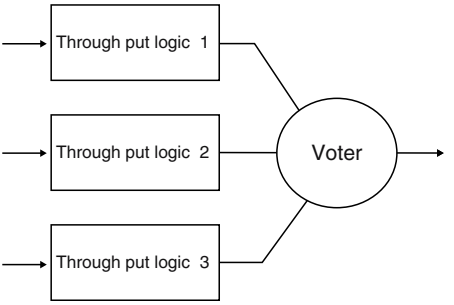

Figure 1.1. TMR architecture for throughput logic.

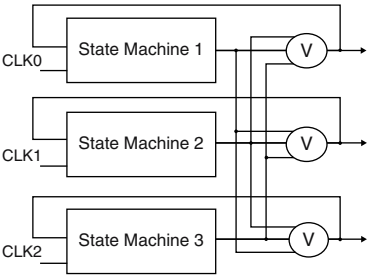

Figure 1.2. TMR scheme for State-machine logic.

The majority of any logic design can be realized by using look-up tables (LUTs), flip-flops (FFs), and routing resources that can be hardened against SEUs in the configuration memory through the previously outlined methods. However, there are other special FPGA resources that allow the implementation of more efficient and performing circuit implementations. These include block RAM, LUT RAM, shift-register, and arithmetic cores. For each of these features, there are particular recommendations to be followed to guarantee an accurate TMR architecture. A detailed presentation of these recommendations is out of the scope of this manuscript. Reader interested in these subjects may refer to [5, 14].

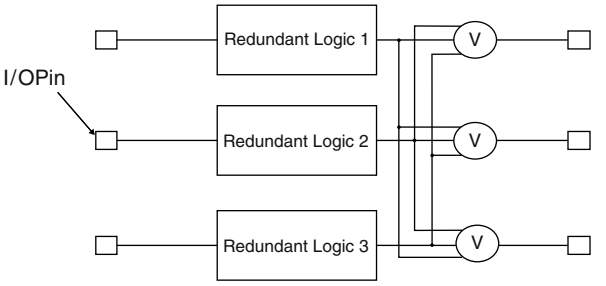

Figure 1.3. TMR scheme for I/O logic.

Other methodologies to implement redundant architectures on SRAM-based FPGAs are available. One of these techniques is oriented in performing all mitigations using the description language to provide a functional TMR methodology [8]. According to this methodology, interconnections and registers are tripled and internal voters are used before and after each register in the design. The advantage of this methodology is that it can be applied in any type of FPGAs.

Another approach is based on the concept that a circuit can be hardened against SEUs by applying TMR selectively (STMR) [15]. This approach extends the basic TMR technique by identifying SEU-sensitive gates in a given circuit and then by introducing TMR selectively on these gates, only. Although this approach optimizes TMR by replicating only the most sensitive portions of a circuit (thus saving area), it needs a high number of majority voter since one voter is needed for each SEU-sensitive circuit portion.

To reduce both the pin count and the number of voters used to implement the TMR approach, Lima at al. proposed a technique based on time and hardware redundancy to harden combinational logic [6, 7]. This technique combines duplication with comparison (DWC) with a concurrent error detection (CED) machine based on time redundancy that works as a self-checking block. DWC detects faults in the system and CED detects which

blocks are fault-free. Although this fault-tolerant technique aims to reduce the number of I/O pads and the power dissipation, it is applied on a high-level description of the circuit, and, thus, if their components are not properly placed and routed on the FPGAs, they may suffer the multiple effect induced by SEU in the FPGAs configuration memory. In order to address the multiple effects induced by SEUs in the FPGAs configuration memory, it is mandatory to select a clever placement and routing of the design. To attach the problem, we abstracted the physical characteristics of FPGA by using a generic FPGA model.

2. PRELIMINARIES OF SRAM-BASED FPGAS ARCHITECTURE

The basic FPGA architecture consist of a two-dimensional array of logic blocks and flip-flops interconnected by a network of interconnections. Families of FPGAs differ from each other by the physical means for implementing user programmability, interconnection wires and the basic characteristics of the logic blocks. In order to describe the general characteristics of modern SRAM-based FPGAs, a generic model is introduced. This model permits to focus attention on only those components that are affected by the multiple faults induced by SEUs. On these components, SEUs induce multiple effects that are permanent until the corrupted bitstream is refreshed through the download of the new one. Thus, place and route algorithms must be enhanced in order to introduce redundancies that are resilient to multiple effects, too.

2.1 Generic SRAM-based FPGA model

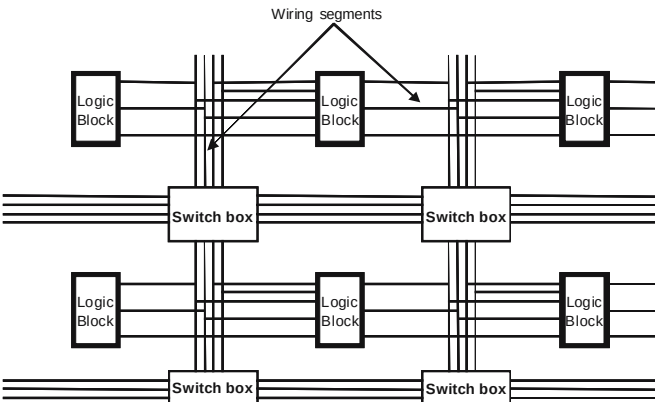

Figure 1.4. Generic FPGA architecture model.

A Field Programmable Gate Array consists of an array of logic blocks that can be interconnected selectively to implement different designs. An FPGA logic block is typically capable of implementing many different combinational and sequential logic functions. Today, commercial FPGAs use logic blocks that are based on transistor pairs, basic small gates such as two-input NANDs or exclusive ORs, multiplexers, look-up tables (LUTs), and wide-fanin AND-OR structures. An FPGA routing architecture incorporates wire segments of varying length that can be interconnected via electrically programmable switches. The distribution of the length of the wire segments directly affects the density and performance achieved by an FPGA.

The SRAM-based FPGA generic model used in this work is shown in Figure 1.4. This model is common to the architecture of several families of SRAM-based FPGAs [16, 17]. The model consists of three kinds of resources: wiring segments, logic blocks, and switch boxes.

Wiring segments are chunks of wiring devoted to transfer information among logic blocks. Wiring segments are organized in the horizontal plane, traversing an FPGA from east to west, and the vertical plane, traversing the FPGA from north to south. Wiring segments are used in conjunction with switch boxes to deliver information between any locations inside FPGAs. Logic blocks contain the combinational and sequential logic required to implement the user circuit, which is defined by writing proper bit patterns inside the FPGAs configuration memory.

Figure 1.5 shows an example of simple logic block, where we can recognize a look-up table (LUT) to implement combinational functions, a flip-flop (FF) to implement memory elements, and two multiplexers (MUX) needed for implementing different signal forwarding strategies.

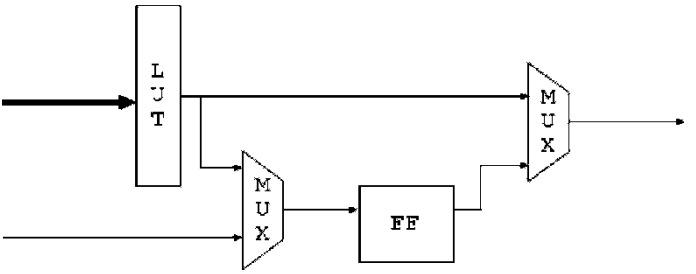

Figure 1.5. Simple FPGA's logic block.

Each logic block has a number of input and output signals connected to adjacent switch boxes and logic block through wiring segments. The SRAM programming technology uses static RAM cells to control pass gates or multiplexers.

The programmable interconnection network consists of wiring segments that can be connected or disconnected by several programmable interconnect points (PIPs). The PIPs are organized to form switch matrices that are located inside switch boxes, which are controlled by the FPGAs configuration memory. PIPs (also called routing segments) provide configurable connections between pairs of wiring segments. The basic PIP structure consists of a pass transistor controlled by a configuration memory bit. There are several types of PIPs: cross-point PIPs that connect wire segments located in disjoint planes (one in the horizontal plane and one in the vertical plane), break-point PIPs that connect wire segments in the same plane, decoded and non-decoded multiplexer (MUX) PIPs, and compound PIPs, which consist of a combination of *n* cross-point PIPs and *m* break-point PIPs, each controlled separately by groups of configuration memory bits [18]. Decoded MUX PIPs are groups of 2^k cross-point PIPs sharing common output wire segments controlled by *k* configuration memory bits. Conversely, non-decoded MUX PIPs consist of *k* wire segments controlled by *k* configuration bits.

2.2 FPGA routing graph

A model that abstracts most of the details of SRAM-based FPGAs has been developed. It is general enough to describe any FPGA architecture and it conveys only the meaningful information for the dependability-oriented analysis. Indeed, it is particularly important to capture information about which logic blocks are used by a circuit mapped on an FPGA, as well as all the information about the interconnections between used logic blocks (i.e., how wiring segments and switch matrices are configured for implementing a circuit). Conversely, it is not important to know which function (combinational or sequential) a logic block implements.

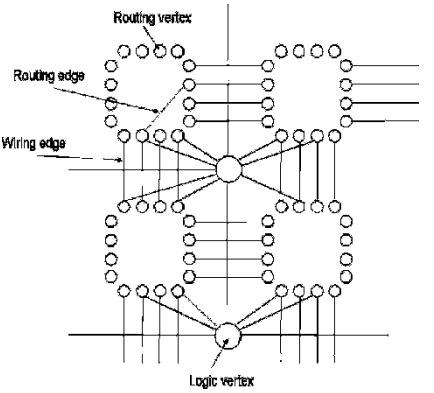

Figure 1.6. FPGA routing graph.

The resources in an SRAM-based FPGA that are used to implement a circuit can be described by resorting to a routing graph, where the graphs vertices model logic blocks and switch boxes while the graphs edges model wiring segments. As shown in Figure 1.6, the routing graph has two types of vertices: logic vertices that model the FPGAs logic blocks and routing vertices that model the input/output ports of each switch box. For each switch box having I inputs and O outputs, the routing graph has $I + O$ routing vertices. Moreover, the routing graph has two types of edges: routing edges that model the FPGAs PIPs as edges between two different routing vertices and wiring edges that model the FPGAs wiring segment as edges between logic vertices and routing vertices.

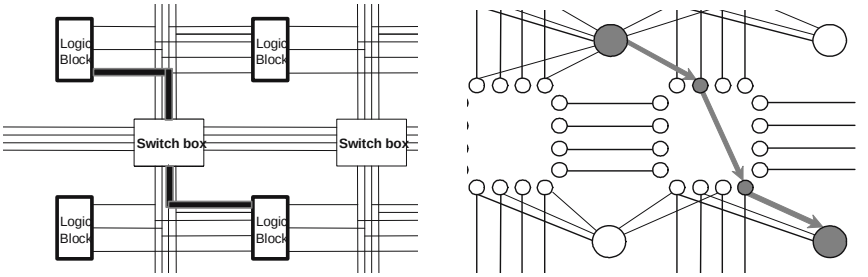

Figure 1.7. Modeling of a FPGA implementing a circuit by means of the routing graph.

An FPGA switch box is described by the graph model in different routing edges forming a structure known as a Universal Switch Module (USM) [19]. The number of vertices and edges modeling switch boxes and logic blocks depends on the selected FPGAs architecture.

According to our model, a logic signal connecting two logic blocks in the circuit the FPGA implements is modeled by the routing graph as a path that may span over different wiring edges and routing edges. As illustrated in Figure 1.7, edges and vertices are colored to indicate that the corresponding FPGAs resource is used to implement a circuit. In case the FPGA implements different circuits or different replicas of the same circuit, different colors are used to mark edges and vertices of each circuit or replica.

Moreover, a direction is associated to any edge to describe the direction of the information flow. The proposed graph model is very flexible and can be adopted to describe any type of FPGAs architecture.

REFERENCES

[1] M. Nikolaidis, *Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies*, Proceedings IEEE 17th VLSI Test Symposium, Apr. 1999, pp. 86–94.

[2] E. Normand, *Single Event Upset at Ground Level*, IEEE Transactions on Nuclear Science, Vol. 43, No. 6, Dec. 1996, pp. 2742–2750.

[3] M. Alderighi, A. Candelori, F. Casini, S. D'Angelo, M. Mancini, A. Paccagnella, S. Pastore, G. R. Sechi, *Heavy Ion Effects on Configuration Logic of Virtex FPGAs*, IEEE 11th On-Line Testing Symposium, 2005, pp. 49–53.

[4] P. Graham, M. Caffrey, D. E. Johnson, N. Rollins, M. Wirthlin, *SEU Mitigation for Half-Latches in Xilinx Virtex FPGAs*, IEEE Transactions on Nuclear Science, Vol. 50, No. 6, Dec. 2003, pp. 2139–2146.

[5] C. Carmichael, M. Caffrey, A. Salazar, *Correcting Single Event Upset Through Virtex Partial Reconfiguration*, Xilinx Application Notes, XAPP216, 2000.

[6] F. Lima Kanstensmidt, G. Neuberger, R. Hentschke, L. Carro, R. Reis, *Designing Fault-Tolerant Techniques for SRAM-Based FPGAs*, IEEE Design and Test of Computers, Nov.–Dec. 2004, pp. 552–562.

[7] F. Lima, L. Carro, R. Reis, *Designing Fault Tolerant System into SRAM-Based FPGAs*, IEEE/ACM Design Automation Conference, June 2003, pp. 650–655.

[8] S. Habinc Gaisler Research, *Functional Triple Modular Redundancy (FTMR) VHDL Design Methodology for Redundancy in Combinational and Sequential Logic*, www.gaisler.com

[9] N. Rollins, M. J. Wirthlin, M. Caffrey, P. Graham, *Evaluating TMR Techniques in the Presence of Single Event Upsets*, poster MAPLD 2003.

[10] M. Bellato, P. Bernardi, D. Bortolato, A. Canderlori, M. Ceschia, A. Paccagnella, M. Rebaudengo, M. Sonza Reorda, M. Violante, P. Zambolin, *Evaluating the Effects of SEUs Affecting the Configuration Memory of a SRAM-Based FPGA*, IEEE Design Automation and Test in Europe, 2004, pp. 188–193.

[11] M. Ceschia, M. Violante, M. Sonza Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, A. Candelori, *Identification and Classification of Single-Event Upsets in the Configuration Memory of SRAM-Based FPGAs*, IEEE Transactions on Nuclear Science, Vol. 50, No. 6, Dec. 2003, pp. 2088–2094.

[12] P. Bernardi, M. Sonza Reorda, L. Sterpone, M. Violante, *On the Evaluation of SEUs Sensitiveness in SRAM-Based FPGAs*, IEEE 10th On-Line Testing Symposium, 2004, pp. 115–120.

[13] F. Lima Kanstensmidt, L. Sterpone, L. Carro, M. Sonza Reorda, *On the Optimal Design of Triple Modular Redundancy Logic for SRAM-Based FPGAs*, 2005, pp. 1290–1295.

[14] C. Carmichael, *Triple Modular Redundancy Design Techniques for Virtex FPGAs*, Xilinx Application Notes, XAPP197, 2001.

[15] P. K. Samudrala, J. Ramos, S. Katkoori, *Selective Triple Modular Redundancy (STMR) Based Single Event Upset (SEU) Tolerant Synthesis for FPGAs*, IEEE Transactions on Nuclear Science, Vol. 51, No. 5, Oct. 2004.

[16] S. Brown, *FPGA Architecture Research: A Survey*, IEEE Design and Test of Computers, Nov–Dec 1996, pp. 9–15.

[17] J. Rose, A. El Gamal, A. Sangiovanni-Vincetelli, *Architecture of Field-Programmable Gate Arrays*, IEEE Proceedings, Vol. 81, No. 7, July 1993, pp. 1013–1029.

[18] C. Stroud, J. Nall, M. Lashinsky, M. Abramovici, *BIST-Based Diagnosis of FPGA Interconnect*, International Test Conference, 2002, pp. 618–627.

[19] Y. W. Chang, D. F. Wong, C. K. Wong, *Universal Switch Modules for FPGA Design*, ACM Transaction on Design Automation of Electronic System, Jan. 1996, pp. 80–101.

Chapter 2

RADIATION EFFECTS ON SRAM-BASED FPGAS

Modeling and simulation of radiations effects

The past 30 years have seen the discovery that electronic circuits are sensitive to transient effects such as Single Event Upsets (SEUs) provoked by ionizing radiation [1]. Since the discovery of SEUs at aircraft altitudes, researchers have made significant efforts to monitor the environment. The space and the earth environment contain various ionizing radiations, generated by natural phenomena such as sun activity and manmade radiation that interacts with silicon atoms. If, at ground level, neutrons and alpha particles are the most frequent causes of SEUs, in a space environment, they are protons and heavy ions. When a particle hits the surface of a silicon area, it loses its energy through the production of free electron-hole pairs, resulting in a dense ionized track in the struck region [2]. Interestingly, when the struck silicon area implements a static memory cell, the transient pulse may induce permanent changes: it can indeed activate the inversion of the stored value. In SRAM-based FPGAs, transient faults originating in the FPGAs configuration memory have dramatic effects since the circuits the FPGAs implement are totally controlled by the content of the configuration memory, which is composed of static RAM cells [3, 4]. In this chapter, the effects of the SEUs within the configuration memory of SRAM-based FPGAs will be accurately described, thanks to the graph model presented in the previous chapter, the effects of SEUs within the internal FPGA's resources is modeled and analyzed.

L. Sterpone, *Electronics System Design Techniques for Safety Critical Applications*,

1. RADIATION EFFECTS

The radiation effects may be classified in two categories: energetic particles (such as electrons, protons, alpha particles), neutrons, heavy ions (that are influenced by the electromagnetic field, and electromagnetic radiations such as photon, gamma ray, X-ray or ultra-violet. The effects of radiations can be distinguished depending on the terrestrial or extra-terrestrial environment.

On the Earth the principal radioactive sources are represented by the radioactive material and by the cosmic ray. The materials used during the productive process of integrated circuits, such as the aluminum and gold, can contain traces of radioactive material or to be exposed to environmental consequences. The cosmic rays are mainly due to the solar wind, that consists of the particles flux at low energy and the galactic cosmic rays, composed by high energy particles emitted by remote sources in the universe.

Radiations coming from the space are influenced by the terrestrial magnetic field that decrease their effects. The particles that pass the terrestrial magnetic field and hit the atmosphere provoke the production of secondary particles that are able to reach the Earth surface. The influences of protons and heavy ions at an high altitude is not negligible. The radio between the amount of radiations that hit an aircraft at high altitude with respect to the amount of radiations at the sea level is 100 times [5].

In the space is absent the filter effect provided by the atmosphere, however the terrestrial magnetic field influence the radioactive particles hitting the space vehicles working in this environment. The source of radiation in the earth space are principally due to three factors: the Van Allen belts, solar wind and galactic cosmic rays.

The Van Allen belts are two regions in which the electrically charged particles are attracted by the terrestrial magnetic field in a stronger measure. Within the Van Allen belts the major causes of electronic circuits malfunctions is composed by high energy protons.

Vice versa the solar wind is formed by the *Coronal Mass Ejection* (CME) that are able to pass the Sun gravity. The solar wind consists of a long flux of particles at high energy that influence the behavior of the Van Allen belt. The galactic cosmic rays are composed by heavy ions at high energy with an isotropic flux, similar for each directions. They hit the space crafts operating outside the influence of the terrestrial magnetosphere.

The two principal mechanisms through radiations interact with the matter are the *atomic displacement* and the *ionization* or *electronic charge displacement*.

The atomic displacement takes place when a particle hits an atom changing its original position. If this atom belongs to the crystalline structure, it may change the properties of the material. The effects on the semiconductor is

similar to the one artificially produced thanks to the ionic implantation process executed during the manufacturing of integrated circuit, and thus it can provoke the equivalent variation of drug in the semiconductor.

The ionization causes the move of charge, forming couple of electron-holes. Within the semiconductor the electric field produced by these particles determine the generation of an internal current, that in some cases may modify the functionalities of the circuit. These kind of errors are defined as *soft-error*, since they do not damage the electronic circuit, but causes only the temporary variation of the functionality. The ionization may be provoked also by photons. The energy transmitted to electrons in the valency band may move them to the conduction band. This iteration produces hole within the small dielectrics, provoking their slow degradation. This is an example of permanent error also known as *hard error*.

The damage provoked by radiations may be classified in two principal categories:

1. *Long terms cumulative degradation*: it is divided in *Total Ionizing Dose* (TID) effects, the accumulation of ionizing radiations over the time, that provokes degradation within the electrical circuit, and *Displacement Damage Dose* (DDD), the accumulation over the time of the atomics' material movements.
2. *Single Event Effects* (SEE): kind of event that happens locally following an action of single ionizing particles. These events are classified as SEE and in particular as *Single Event Upset* or *Single Event Latchup*.

1.1 Single Event Upset (SEU)

The Single Event Upset (SEU) is a change of condition or a transition, induced by an high charged particle. An SEU consist of the change of the logic state or, more in general, in a transitory error and it is classified by the scientific literature in the category of *soft-error* since it can provoke the *reset* or the *rewriting* of the device normal behavior.

The Figure 2.1A shows a simple storage cell of a single bit and it illustrates the effect of an SEU also known as bit-flip. The circuit in Figure 2.1A is designed in order to maintain to stable state: stored '0' and stored '1'. In each state two transistors are activated and two are put off. A bit-flip happens when an high-charged particle provoke the inversion of the circuit transistor state. This phenomena happens in all microcircuits, from memory chips to microprocessors. The occurrence of a bit-flip can generate a random change of the processor state and may provoke the crash of the system. The Figure 2.1B illustrates how an high-charged particle may provoke a spurious electronic signal. The particle produces a charge along its path in the form of electron-hole couple. These are collected within the source and drain

generating an effect similar to a current pulse that may be sufficiently wide to produce an effect comparable to a normal signal applied to a transistor.

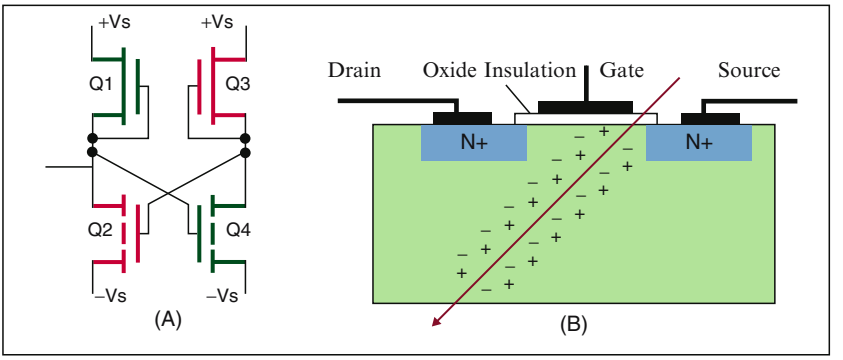

Figure 2.1. (A) storage cell for a single bit (S-RAM). (B) junction crossed by an high-charged particle.

The SEUs are drastically relevant for SRAM-based FPGA since the configuration memory is sensible to ionizing radiations. The effects of SEUs within SRAM-based FPGA devices depend on the technology and on the architectural choice. The malfunction provoked by an SEU is classified as Single Event Functional Interrupt (SEFI).

The SEFI phenomena is used for the first time in the 1996 within the Standard EIA/JEDEC[2]. The SEFI is the first anomaly within integrated circuits provoked by a bump of a single ion, similarly to the SEU, that introduces a temporary malfunction or interruption of the device standard operations. While the SEU is a phenomena that produces a temporary change of the device physical conditions, the SEFI is a phenomena that happens in the temporary change of the implemented functionality and may remain until the power supply is interrupted. The SEFI are observable in several devices, however until it is not related to a single cause, this phenomena remains hardly definable [6].

1.2 Single Event Latch-Up (SEL)

The ionizing radiations may provoke other kinds of effects called *Single Event Latch-up* (SEL), that is produced activating the parasitic transistor present between the junctions N-P of the CMOS transistors. The activation of such kind of transistor create a low frequency path between the power supply (Vcc) and the ground, crossed by an high current. For this reason, the SEL effects are potentially destructive for an electronic circuit. In parallel with the progressive reduction of the physical dimensions, the supply current and the threshold voltages applied to the manufacturing techniques of

SRAM-based FPGAs, the malfunctions due to radiations are proportionally increased.

2. SEU EFFECTS ON FPGA'S CONFIGURATION MEMORY

SRAM-based FPGAs suffer from radiation as other semiconductor devices. Designer and users have to consider these radiation effects before including an SRAM-based FPGA within a space application. SRAM-based FPGAs, as other devices, that contain several arrays of memory cells, are extremely sensitive to SEUs due to the large amount of memory within a relatively small amount of silicon area.

SRAM-based FPGAs contain a lot of memory cells within a single device, implementing the configuration memory, which are sensitive to SEUs. The SEU upset rate is related to the kind of radiation environment where the device will be used. To mention an estimation, in the Cibolla flight experiment using a SRAM-based FPGA Xilinx Virtex 1000 containing more than six million bits, it has been calculated that worst-case SEU upset rate on an average orbit ranges from 0.13 SEUs per hour under a quiet sun, up to 4.2 SEUs per hour under a peak upset rate [7]. The effects induced by SEUs on SRAM-based FPGAs have been recently investigated thanks to radiation experiments [8–10]. More recently, an analysis that combines the results of radiation testing with those obtained while analyzing the meaning of every bit in the FPGAs configuration was presented in [11].

Although SEUs are transient by nature, when they originate in the configuration memory, their effects are permanent since SEUs remain latched until the configuration memory is rewritten with new configuration data. The errors produced by SEUs in the FPGAs configuration memory can be classified into two different categories: errors that affect logic blocks and errors that affect the switch boxes.

As far as logic-block errors are concerned, several different phenomena may be observed, depending on which resource of the logic block is modified by the SEU:

- *LUT error*. The SEU modified one bit of a LUT, thus changing the combinational function it implements.
- *MUX error*. The SEU modified the configuration of a MUX in the logic block, as a result, signals are not correctly forwarded inside the logic block.
- *FF error*. The SEU modified the configuration of a FF, for example, changing the polarity of the reset line or that of the clock line.

In order to model faulty logic blocks in the routing graph previously described, we assumed using the black color to mark each vertex corresponding to a faulty logic block.

As far as switch boxes are concerned, different phenomena are possible. Although an SEU affecting a switch box modifies the configuration of one PIP, both *single* and *multiple* effects can be originated.

Single effects happen when the modifications induced by the SEU alter only the affected PIP. In this case, one situation may happen. The SEU changes the configuration of the affected PIP, and the existing connection between the two routing segments is opened, provoking an open effects. Considering the routing graph, this situation is modeled by deleting the routing edge corresponding to the PIP that connects the two routing vertices.

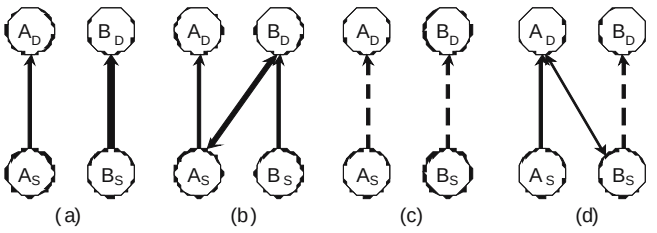

Figure 2.2. Possible multiple effects induced by one SEU.

In order to describe the multiple effects in terms of modifications to the routing graph, let us consider the two routing edges A_S/A_D and B_S/B_D connecting the routing vertices A_S, A_D, B_S, B_D, as shown in Figure 2.2a. Considering this routing situation, the following modification could be introduced by an SEU:

1. *Short* between A_S/A_D and B_S/B_D. As shown in Figure 2.2b, a new routing edge is added to the graph that connects either one end of A to one end of B. This effect can happen if A_S/A_D and B_S/B_D belong to the same switch box and the SEU enables the non-decoded or decoded PIP that connects B with A.
2. *Open* correspond to the deletion of both routing edges A_S/A_D and B_S/B_D as shown in Figure 2.2c. This situation may happen if a decoded PIP controls both A_S/A_D and B_S/B_D.
3. *Open/Short*, which corresponds to the deletion of either the routing edge A_S/A_D or the one B_S/B_D and to the addition of the routing edge A_S/A_D or B_S/B_D, as shown in Figure 2.2d. This situation may happen if a decoded PIP controls both A_S/A_D and B_S/B_D.

The short effects, as shown in Figure 2.2b, may happen if two nets are routed on the same switch box and a new edge is added between them. This kind of faulty effect happens when a cross-point PIP, which is non-buffered and has bidirectional capability, links two wire segments located in disjoint planes. Conversely, the Open and the Open/Short effects, as shown in Figure 2.2c, d, may happen if two nets are routed using decoded PIPs.

3. SIMULATION-BASED ANALYSIS OF SEUs

Researchers have investigated the use of simulation-based approaches for predicting the effects of SEUs. The methods proposed so far [12, 13], although effective and accurate, are intended for the analysis of applications implemented on ASICs only. Considering the SRAM-based FPGA devices, two complementary aspects should be considered:

1. SEUs may alter the memory elements the design embeds. For example, a SEU may alter the content of a register in the data-path, or the content of the state register of a control unit.
2. SEUs may alter the content of the memory storing the devices configuration information. For example, a SEU may alter the content of a Look-Up Table (LUT) inside a logic resource of the FPGA, or the routing signals.

As far as the former aspect is concerned, the available approaches are adequate. Conversely, the latter aspect demands much more complex analysis capabilities. The effects of SEUs in the devices configuration memory are indeed not limited to modifications in the design memory elements, but may produce modifications to the interconnections inside a logic resource and among different logic resources.

A Simulation-based approach to address the aforementioned problem has been developed: through suitably defined fault models and an ad-hoc developed simulation tool, the procedure is able to predict the effects of SEUs in the device configuration memory. The approach provides experimental results that can be compared to the predicted SEU cross-section with those obtained from radiation testing. These comparisons show that our method is quite accurate and that it can be used to predict the result of radiation testing.

3.1 Simulation environment

In the developed environment the FPGA-based system is composed of two independent layers: the *application layer* and the *physical layer*. The

application layer corresponds to the digital circuit that implements the functionalities the system is intended to carry out. The application layer is a VHDL model that codes the netlist implementing the desired circuit. Its building blocks are the components available within the adopted FPGA: LUTs that store the truth table of the Boolean functions the circuit implements, routing resources, and memory elements (flip-flop, register, etc.). Conversely, the physical layer corresponds to the FPGA device on which the circuit is implemented. The two layers are analyzed independently by the proposed approach.

The application layer is analyzed using a simulation-based analysis tool which computes the predicted error rate. The figure is the probability that an SEU modifies the circuit implemented by the application layer in such a way that it produces SEFIs, i.e., erroneous output results. The computation of the predicted error rate is performed by resorting to fault-injection experiments, which are based on fault models that emulate accurately the effects of SEUs in the configuration memory of FPGAs.

The physical layer is analyzed using the test-bed we introduced in [14]. The purpose of this analysis is to characterize the FPGA devices manufacturing technology from the point of view of sensitivity to radiation. For this purpose, radiation-testing experiments are performed to measure the *cross-section* of the adopted FPGA device, which gives the probability for a particle to produce an SEU.

The important aspect of this approach is that the computation of the cross section does not depend on the application layer: in fact it may be performed by configuring the FPGA device with test circuits that are different from the application layer. The cross section obtained by this method is associated with the FPGA device and it is independent respect to the application using it. The analysis of the physical layer is required each time a new technology is exploited: once the FPGA cross-section has been computed, it may be exploited for any application using that technology.

As soon as both analyses are completed, we can compute the predicted cross-section of the whole system, as follows:

$$\sigma_{Predicted} = \epsilon_{Predicted}\, \sigma_{FPGA} \tag{2.1}$$

This figure gives the sensitivity to radiation of the whole systems. It thus combines the effects of SEUs in the application layer. A similar approach was proposed in [15] for analyzing processor-based systems.

The core of the tool is the fault-injection environment outlined in Figure 2.3. Starting from an initial description of the circuit the system implements, we use the tools provided by the FPGA vendor for performing place and route operations. This preliminary step is typical of any design flow based on FPGA devices, and produces a configuration file where the content of the devices configuration memory is stored, i.e., the bitstream. This information

defines the application layer. Starting from the information stored in the bitstream, two ad-hoc developed tools are used.

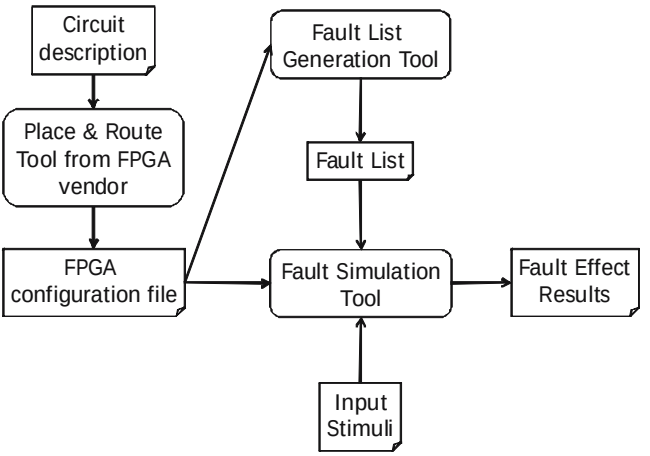

Figure 2.3. Architecture of the fault-injection approach we developed. It combines both ad-hoc developed tools with commercial tools provided by the FPGA vendor for place and route operations, and independent suppliers for simulation operations.

The *Fault List Generation Tool* identifies the FPGAs resources in the application layer (for logic implementation, signal routing, etc.) that are used and it generates the list of faults (Fault List) to be injected, accordingly to the fault models described in the section 2 of the present chapter. Each fault is described by the couple (fault injection time, fault location) describing when the SEU appears, and which resource it modifies.

The *Fault Simulation Tool* simulates serially the faults in the *Fault List*. During simulations the outputs produced by the faulty application layer are compared with those of the fault-free one. As soon as a mismatch is found, the simulation is stopped and the effect provoked by the injected fault is classified as *wrong answer*. Conversely, in case the simulation of the Input Stimuli set concludes, and no mismatch is found, the fault is classified as *Effectless*.

The tools produce the following figures:

- B_{used}. The number of configuration memory bits that needs to be programmed on the physical layer to implement the application layer.
- B_{total}. The total number of configuration memory bits for the physical layer. It includes the bits that need to be programmed for implementing the application layer, as well as those left unprogrammed since the resource they control are not used.

- N_ϵ. The percentage of injected faults whose effects are classified as *Wrong Answers*.

The aforementioned figures are combined by means of (2.2) to estimate the predicted circuit error rate:

$$\varepsilon_{PREDICTED} = N_\varepsilon \cdot \frac{B_{USED}}{B_{TOTAL}} \tag{2.2}$$

The term N_ϵ is the percentage of faults provoking Wrong Answers, while the ratio estimates the probability for an SEU to appear in the used portion of the physical layer.

Given an SRAM-based FPGA device, its configuration memory consists of two types of bits: some controlling *signal-routing* resources, and some controlling *logic* resources. Signal-routing resources are all those resources concerned with the transmission of information within the physical layer. In general these resources include: *wire segments*, which are wires unbroken by programmable switches (each end of a wire segment typically has a switch attached), and *tracks*, which are sequences of one or more wire segments [16].

Conversely, logic resources are all those resources concerned with the implementation of combinational or sequential logic functions.

By considering the typical architecture of SRAM-based FPGAs, we can observe the modifications induced by SEUs to the FPGA resources configuration described in the previous sections.

The tool we developed for Fault List Generation analyzes the device configuration file produced by the place and route tools, and it identifies the bits used to route the (N_{route} bits), and those controlling the logic resources used by the mapped circuit (N_{CLB} bits). It then generates all the possible couples (*fault-injection time*, *fault location*), where fault-injection time ranges from the time of application of the first input stimuli to the last one, while fault location corresponds to all the possible SEUs in $N_{\text{route}} + N_{\text{CLB}}$ bits. Fault sampling is exploited to reduce the number of faults to be simulated by the Fault Simulation tool: if N is the number of simulated faults, then (N_{route} x N) / ($N_{\text{route}} + N_{\text{CLB}}$) faults will be injected in the routing resource, while (N_{CLB} x N) / ($N_{\text{route}} + N_{\text{CLB}}$) will be injected in the CLB ones. Similarly, fault-injection time will be randomly selected between the first and the last input stimuli.

3.2 Fault simulation tool

In the present section, it is described the fault simulation tool developed while addressing Xilinx devices. The tool can be adapted easily to other devices from different manufacturers, since it works on commonly used

hardware description languages (HDL) model of a circuit mapped on an FPGA available (i.e., the application layer).

In order to help designers to evaluate the correctness of their designs after place and route, FPGA vendors usually provide this type of tool.

TABLE 2.1 Summary of the mutations inserted in the VHDL model of the considered circuit to mimic the effects of seus in the device configuration memory

Faulty resource	Fault effect	Corresponding mutation
Routing	Open	Stuck-at-zero or Stuck-at-one, depending on the affected resource.
	Bridge	The signal source is modified and connected to a new source depends on the affected resource.
	Conflict	Wired-AND or Wired-OR
Logic	Combinational defect	Bit-flip in a Look-Up Table
	Routing defect	The signal source is modified and connected to a new source. The choice of the new source depends on the affected resource.
	Sequential defect	Bit-flip in a flip-flop.

The developed tool exploits the ModelSim VHDL simulator for evaluating the outputs that the faulty application layer produces. For this purpose, the application layer is first obtained by executing the ncd2vhdl tool provided by Xilinx. Where NCD stands for Native Circuit Description language, and in details, is the file containing all the information of the circuit mapped on the FPGA's physical level. Let's consider to refer on the fault-free application layer as C_{gold}. Before fault simulation can start, for each fault in the Fault Lists a new model, called C_{faulty}, is computed as a mutation of C_{gold}. During this process the set of VHDL instructions that model the fault are inserted in C_{gold}. In particular, using the mutations reported in Table 2.1.

Table 2.1 shows an overview of the test-bed, including its main components. A Control Host, located outside the irradiation chamber, is used to monitor the experiment execution. It is provided with an IP connection with the set-up inside the irradiation chamber through which it sends commands and receives information about the status of the experiments, as well as data to be logged for elaboration purposes. Inside the irradiation chamber, it has been located a Test CPU (a Power-PC MPC860) that communicates with the Control Host as well as with the device under test. Its purpose is to perform the low-level operations needed for running an experiment: programming the device under test, applying input stimuli, collecting output responses, and

reading back the configuration memory of the device under test. A Control Hardware is also used for adapting the Test CPU to the FPGA Under Test.

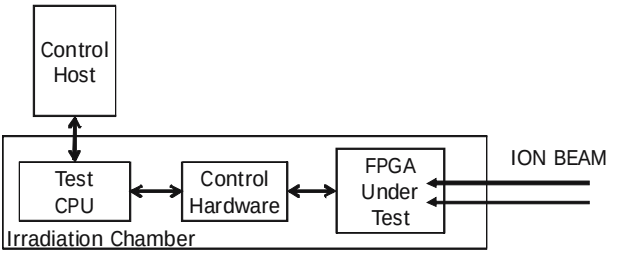

Figure 2.4. Overview of the test-bed we developed for performing radiation-testing experiments on FPGA devices.

The test-bed, illustrated in Figure 2.4, can be used for two purposes. It can be exploited for measuring the cross section of an FPGA-based system, obtaining the measured cross section of the whole systems. For this purpose, the typical test session consists in configuring the physical layer with the application layer, and then in continuously stimulating the FPGA device with a given set of input stimuli. The output responses are continuously collected and compared with the expected ones. As soon as a mismatch between the expected output values and the read ones is observed, i.e., when a SEFI is detected, the test is stopped and the configuration of the FPGA Under Test is read back and sent to the Control Host for data logging. Following this operation, the test is restarted from the beginning. By relating the number of observed SEFIs with the estimated number of particles hitting the devices surface is then possible to compute the device cross section.

Similarly, the test-bed can be used to measure the cross section of the physical layer. In this case, the FPGA is initially programmed with an empty bitstream, and then its configuration memory is periodically read back. By comparing the read information with the fault-free bitstream, it is possible to measure the number of observed SEUs. As previously done, the device cross-section is computed relating this figure with the estimated number of particles hitting the device surface.

3.3 Experimental results

In order to evaluate the accuracy of the presented approach, several experimental analysis have been executed.

The first one, aims at evaluating the accuracy of the simulation-based approach while modeling the effects of SEUs in the device configuration memory.

The second one, aims at evaluating the accuracy of estimation of the predicted cross section of a circuit mapped on a device with respect to that measured by means of radiation testing.

The Xilinx Virtex XCV300 device has been used as physical layer. The device has been exposed to various ion species ranging from 84 MeV *Carbon* to 210 MeV *Nickel* featuring linear energy transfer (LET) values between 1.6 and 30 MeV cm/mg. Radiation testing experiments were carried out at the Tandem Van De Graff Accelerator of INFN-LNL, Legnaro (PD), Italy.

The application layer was a circuit composed of four 16 × 16 bit binary multipliers. Inputs of the four multipliers were connected in parallel, while the outputs were connected to an XOR gate array. The multiplier occupies 2,524 out of 3,072 slices of the adopted XCV300 device and operates at 10 Mhz.

To assess the accuracy of the developed simulation tool, the output responses have been compared during the radiation testing with those computed by the simulation tool. For each SEFI recorded during radiation testing, the SEU causing it has been identified. The SEU is modeled in terms of the modification it introduces in the application layer, and finally it has been injected in the application layer by means of the developed simulation tool.

For this purpose, an initial set of radiation testing experiments is performed. During the radiation experiments the physical layer was configured with the application layer, it was continuously stimulated by a given set of input stimuli, and the resulting outputs observed. As soon as a mismatch on the output values was observed between the expected values and the measured ones, the test was stopped, and the content of the physical layer configuration memory was read back. By analyzing the faulty bitstream, the FPGA's resources affected by SEUs have been identified.

For each SEU observed during radiation testing, which forced the system to produce the faulty outputs, it is executed a simulation experiment. The SEU observed in the device configuration memory is modeled accordingly to the proposed technique by injecting a SEU into the application layer through the simulation-based approach described in the previous section. Then, the resulting output traces are recorded. Finally, the output observed during the radiation experiments have been compared with those obtained by simulations: for all the injected faults, the resulting traces predicted and always matched the measured ones.

The cross section of the FPGA-based system (a multiplier implemented on a Xilinx Virtex device) predicted by simulations is compared with that measured during radiation testing.

TABLE 2.2 Comparison between the cross section obtained during radiation testing experiments and that obtained by means of simulations

Ion	LET [MeV·cm^2/mg]	*Measured* circuit cross section [cm^2/bit]	*Predicted* circuit cross section [cm^2/bit]
^{12}C	1.6	$1.78 \cdot 10^{-13}$	$1.08 \cdot 10^{-12}$
^{16}O	3.0	$1.98 \cdot 10^{-11}$	$4.44 \cdot 10^{-11}$
^{19}F	4.1	$3.53 \cdot 10^{-10}$	$5.28 \cdot 10^{-10}$
^{28}Si	8.5	$1.80 \cdot 10^{-9}$	$1.82 \cdot 10^{-9}$
^{58}Ni	29.0	$2.57 \cdot 10^{-9}$	$4.45 \cdot 10^{-9}$

In computing the predicted error rate, we injected 10,000 SEUs. For the application layer, 9,712 faults have been identified in the routing resources and 288 faults into the logic resources using the Fault List Generation Tool.

By multiplying the predicted circuit error rate by the cross section of the physical layer the predicted cross section is obtained. Table 2.2 gives the predicted cross section obtained during radiation testing for the specific ions used in the experiment. Table 2.2 also gives the measured cross section obtained during radiation testing for the specific ions used in the experiment.

4. HARDWARE-BASED ANALYSIS OF SEUs

As the reader can observe, predicted values are within a factor of two of the measured ones. The effects induced by SEUs on a SRAM-based FPGA have been recently investigated through radiation experiments [8–10], where the predominant effect that was observed was the Single Event Functional Interrupt (SEFI). More recently, an analysis that combines the result of radiation-testing with those obtained while analyzing the meaning of every bit in the FPGAs configuration memory were reported in [11, 14], which identified the bits responsible for each SEFI and that classified the observed SEFIs according to the affected FPGAs resource.

As an alternative to radiation testing, several fault injection approaches were recently proposed. All these approaches emulate the effects of SEUs in the FPGA's configuration memory as bit-flips in the memory content, i.e., the bitstream, downloaded in the FPGA at power up. Some of them exploits run-time reconfiguration [17], while others modify the bitstream before downloading it in the configuration memory [18] or during download operations [19].

Several techniques have been developed in the past years in order to avoid the incidence of SEUs on the behavior of the implemented circuits.

Some of them aim at correcting the effects of SEUs in the device configuration memory. For example the techniques proposed by Xilinx and known as Scrubbing consists in periodically reloading the whole content of the configuration memory [20]. A more complex system used to correct the information in the configuration memory exploits the *readback* and *partial configuration* process. Through the readback operation, the content of the FPGAs configuration memory is read and compared with the expected value, which is stored in a dedicated memory located outside the FPGA. As soon as a mismatch is found, the correct information is download in the FPGAs memory. During the reconfiguration only the faulty portion of the configuration memory is rewritten [20].

Alternative techniques were also proposed that do not aim at identifying and correcting the modification introduced by SEUs, but just aim at avoiding the propagation of SEU effects to the observable outputs, mainly by introducing hardware redundancy in the circuit mapped on the FPGAs. Triple Modular Redundancy (TMR) is an attractive solution for SRAM-based FPGAs because it provides full hardware redundancy of the users combinational and sequential logic, the routing, and the I/O pads [8, 21].

The basic idea of the TMR scheme is that a circuit can be hardened against SEUs by designing three copies of the same circuit and building a majority voter on the outputs of the replicated circuits. Implementing triple redundant circuits in other technologies, such as ASICs, is generally limited to protecting only the memory elements, because combinational logic is hard-wired and correspond to non-configurable gates. Conversely, full module redundancy is required in FPGAs, because memory elements, interconnections and combinational gates are all susceptible to SEUs. This means that three full copies of the users design have to be implemented to harden the circuit against SEUs. In order to prevent fault accumulation, TMR is often coupled with techniques like scrubbing or readback and partial reconfiguration to remove SEUs from the FPGAs configuration memory.

Although effective, the overheads TMR mandates may overcome the available resources, e.g., the number of available I/O pads, and thus some applications exist where it can hardly be exploited. To solve this problem a new method was proposed in [22], aiming at reducing the overhead of a full TMR implementation.

Even if optimized, these kinds of methods come with very high design penalties: besides the area overhead due to the TMR design, removing SEUs from the configuration memory mandates the adoption of ad-hoc circuit for supporting the readback and the partial reconfiguration procedure, and additional energy consumption.

4.1 Details on the Xilinx Triple Modular Redundancy

The suggested optimal implementation of the TMR circuitry inside a SRAM-based FPGA provided by Xilinx depends on the type of the circuit that is mapped on the FPGA device. There are three types of structures: combinational logic, state machines or special devices.

The primary purpose for using the TMR methodology is to remove all single points of failure from the design. This starts with the FPGA inputs. If the same input is connected to all the three domains of the redundant logic within the FPGA, then a failure at the input would propagate through all the domains, escaping the TMR protection capability. Therefore, each replica of the redundant logic should have its own set of inputs, as illustrated in Figure 2.5.

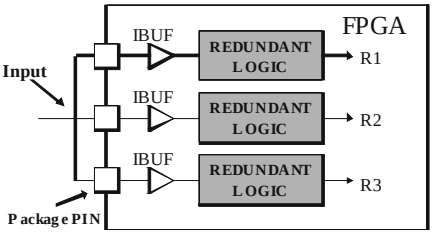

Figure 2.5. Triple Modular Redundancy (TMR) FPGA inputs.

As far as the implementation of the majority voter is concerned, Xilinx proposed to build it by using the Output Buffer Three-state cell (OBUFT) provided by Xilinx library primitives as shown in Figure 2.6.

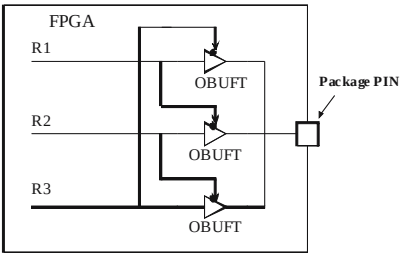

Figure 2.6. Triple Modular Redundancy (TMR) BUFT majority voter outputs.

4.2 Analysis of TMR architecture

In order to assess the effects of SEUs in the FPGA configuration memory, a fault-injection system is used to inject SEUs internally to the configuration

memory and to record the circuit's output. The fault injection system used is composed of the following modules:

1. *Fault List Manager (FLM)*: it generates the list of faults to be injected within the circuit under analysis, i.e., the Device Under Test (DUT).
2. *Fault Injection Manager (FIM)*: it manages the fault injection process, by selecting one fault from the fault list, performing its injection in the DUT and then observing and analyzing the obtained results to provide the fault-effect classification.

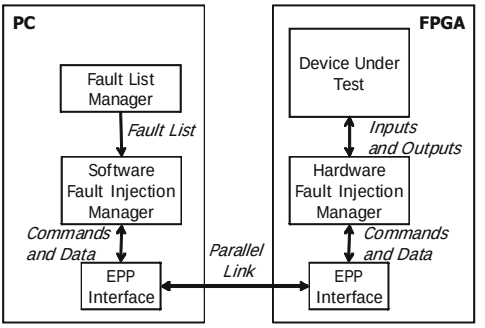

Figure 2.7. The architecture of the fault-injection system.

The proposed fault injection system consists of an FLM module implemented as a software process running on a host PC, and a FIM, that runs in part on an host PC and in part on the same FPGA device where the DUT is placed. The two portions of the FIM communicate through a parallel link that exploits the Enhanced Parallel Port (EPP) protocol. The scheme of the implemented fault injection system is implemented in Figure 2.7.

In the developed fault injection system the DUT, the EPP Interface and the FIM shared the same FPGA device. These modules should be placed on the FPGA device in such a way that any fault injected in the DUT does not interfere with the FIM and EPP interface. This requirement is complied by constraining the place and route algorithms to organize the FPGA-resource allocation as described in the Figure 2.8.

The developed fault injection system exhaustively injects faults in all the configuration memory bits, no matter if they are used or not. In order to speed-up and make more precise the fault-injection process, the developed FIM identifies the configuration memory bits that are actually programmed to implement the DUT, and generates faults only for them. Moreover, this solution prevents us from erroneously injecting faults in the FPGA resource implementing the FIM and the EPP interface.

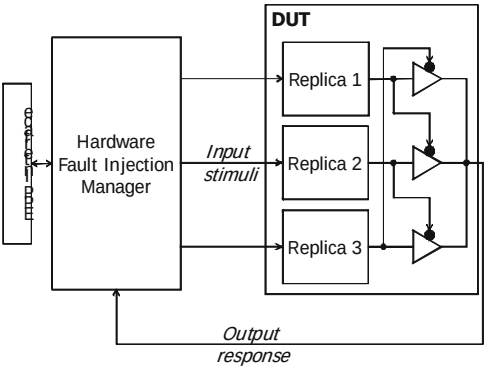

Figure 2.8. The adopted resource allocation of the fault injection system.

To implement such a solution, it is first generated the file storing the configuration memory bit for whole design, then, FIM identifies those bits related to the DUT. This step is possible thanks to a data-based developed by decoding the Xilinx bitstream. Finally, the list of faults for the DUT configuration memory bits is computed and stored. Each element of the fault list is a faulty bitstream for the FPGA where one bit at a time is modified to emulate the effects of a SEU.

The architecture of the Fault Injection Manager is the most crucial part in the whole fault injection system. It is divided in two parts, a software one implemented on a PC (SW-FIM) and an hardware one (HW-FIM) that runs on the FPGA.

The FIM executes the fault injection process in four steps:

1. The SW-FIM configures the FPGA with a faulty bitstream taken from the fault list.
2. The SW-FIM sends a start command to the HW-FIM placed on the FPGA.
3. The SW-FIM polls the EPP Interface waiting for the result of the execution, and then it performs the fault effect classification.
4. The SW-FIM resets the FPGA board and restarts from step 1.

The SW-FIM is a supervisor for the HW-FIM, which consist of three modules: a control unit, a test-pattern generator and an output analyzer.

The control unit inside of the HW-FIM communicates with the SW-FIM through the EPP Interface and implements the following steps:

1. It waits for the start command from the SW-FIM.
2. It puts EPP Interface on an idle state, and starts the test-pattern generator.
3. When all the stimuli have been applied, it sends to the SW-FIM the result observed by the output analyzer.

The test-pattern generator provides the input stimuli to the DUT. The output analyzer compares the output response coming from the faulty DUT with the expected one, which is computed by a fault-free replica of the circuit hardened via TMR.

4.3 Experimental results

The fault injection experiments are performed injecting SEUs in the configuration memory of FPGAs while implementing circuits hardened according to the Xilinx TMR architecture. The experimental setup presented in the previous sections has been used, the accuracy of this setup was confirmed by radiation testing experiments reported in [11]. The device used in the experiments is the Xilinx Spartan XC2S30-PQ144, whose configuration memory is composed of 336,768 bits organized in 1,165 frames of 288 bits each. The configuration memory controls 132 I/O blocks and an array of 12 x 18 slices [23].

The performed analysis consisted in study three purely combinational circuits: an adder computing the sum of two 8-bit wide operands, an adder working on two 16-bit wide operands and a multiplier working on two 8-bit wide operands.

During the experiments, a 16-bit wide up-counter has been used as test-pattern generator. It allows generating all the possible input combinations for both the 8-bit adder and multiplier. The same counter was also used for testing the 16-bit wide adder, while the two inputs ports were shortened together.

TABLE 2.3 Characteristics of the adopted circuits

Circ.	Slices [#]	Programmed bits [#]	CLB bits [#]	Routing bits [#]
Add8	100	9,785	2,560	7,225
Add16	103	11,963	2,656	9,307
Mul8	127	17,448	3,280	14,168

The characteristics of the adopted circuits are reported in Table 2.3, where Slice reports the number of FPGA slices that the circuit occupies, Programmed bits is the number of configuration memory bits that are actually used by the mapped circuit, CLB bits is the number of configuration memory bits used to program the configurable logic blocks the circuit exploits, and Routing bits is the number of configuration memory bits for signal routing the circuit exploits.

The described set-up was used for running three fault-injection campaigns, one for each circuit.

The fault-injection process took about 6 s for each fault. Being test-pattern generator and the output analyzer place in the same FPGA holding the DUT, the time needed for applying input stimuli and classify fault effects was negligible (4.5 ms on the average). Most of the time for processing each fault was indeed spent to download the faulty bitstream.

TABLE 2.4 Fault-injection results

Circuit	Injected faults [#]	Wrong answer [%]
Add8	9,785	9.01
Add16	11,963	11.28
Mul8	17,448	13.18

The results collected during the fault-injection campaigns are reported in the Table 2.4, where *Injected Faults* reports the number of injected SEUs, and *Wrong Answer* reports the percentage of SEUs provoking SEFIs, i.e., the obtained output response differs from the expected ones.

During the experiments, it is injected only one fault for each configuration memory bitstream actually programmed for implementing the mapped circuit. The faults were selected in such a way that common-mode faults were not possible.

These results are particularly interesting since they experimentally show that the TMR architecture is not able to effectively harden the considered circuits against SEUs affecting the configuration memory of SRAM-based FPGAs. Moreover, the percentage of *Wrong Answers* is related to the density of programmed bits within the slices used for implementing the TMR architecture.

TABLE 2.5 Comparison between programmed and fault bit

Circ.	Programmed-bit density [bit/slice]	Wrong answer [%]
Add8	97.85	9.01
Add16	116.75	11.28
Mul8	137.39	13.18

To better outline this effect, in Table 2.5 a comparison between the FPGA-resource usage and the percentage of *Wrong Answer*.

The column *Programmed-bit Density* reports the average number of programmed bits for the FPGAs slices actually used by the DUT. As the reader can observe, the percentage of *Wrong Answer* scales with the *Programmed-bit Density*.

This result suggested that SEU sensitivity in SRAM-based FPGAs is related to the number of used bits in each slice. The lower it is the number of used bits in each slice, the lower it is the probability that, when affected by SEUs, the bits of a slice provoke a SEFI.

5. ROBUSTNESS OF THE TMR ARCHITECTURE

The circuit mapped on a reconfigurable FPGA is totally controlled by the configuration memory, which in the case of SRAM-based FPGA, is composed of static RAM cells. Interestingly the effects induced by SEU affecting the configuration memory are permanent, since the SEU changes the mapped circuit until the device is programmed again. The result of a SEU that causes the devices to stop operating properly is generally defined as a Single Event Functional Interrupt (SEFI). One possible solution to this problem is to use radiation-hardened FPGAs, but since these devices are very expensive, alternative solutions allowing using non radiation hardened devices are currently investigated.

Triple Module Redundancy is often exploited for hardening digital logic against SEUs in safety-critical applications. As an instance, TMR is often used to design fault tolerant memory elements to be employed in sequential digital logic. Unfortunately, non-radiation-hardened FPGAs present insufficient protection of memory elements in both the mapped circuit, and the configuration memory. As a result, particles hitting the configuration memory can change dramatically the logic functionality of the mapped circuit, as well as the circuits memory elements. Evaluation techniques must be used to evaluate the impact of SEUs affecting FPGAs configuration memory, and to avoid undesired changes of the circuit mapped on the FPGA.

The purpose of this section is to deeply investigate how circuits designed according to the Triple Modular Architecture, and mapped on non-radiation-hardened SRAM-based FPGAs, behave when SEUs are injected in the configuration memory cells controlling the FPGA resources. For this purpose fault injection experiments are performed.

As the results of the experimental SEU's effects analysis illustrated, it is suggested that it is possible to reduce the effects of SEUs within the configuration memory bits of non-radiation-hardened SRAM-based FPGAs by placing the TMR circuit on the FPGA floorplan respecting constraints rules able to decrease the damaging effect of SEUs.

The experimental analysis assesses the effects of SEUs in the FPGA configuration memory of a real FPGA device, we injected faults in a Xilinx Spartan XC2S30PQ144 device, whose configuration memory is composed of 336,768 configuration memory bits organized in 1,165 frames of 288 bits

each. The configuration memory controls 132 I/O blocks and array of 12 x 18 slices [23].

The experimental analysis considers an extended set of circuits including two adders (one working on two 8-bit wide operands and the other on two 16-bit wide operands) and two multipliers (one working on 4-bit wide operands and one on two 8-bit wide operands. Furthermore an analysis on an elliptic filter in order to evaluate the sensitiveness to SEUs in a sequential circuit.

The characteristics of the set of circuit used in the experiments are reported in Table 2.6, where Slices reports the number of FPGA slices that the circuit occupies, Programmed bits the number of configuration memory bits actually used by the mapped circuit, CLB bits is the number of configuration memory bits used to program the configurable logic blocks of the circuit, and finally Routing bits is the number of configuration memory bits for signal routing the circuit exploits.

TABLE 2.6 Characteristics of the adopted circuits

Circuit	Slice [#]	Programmed bits [#]	CLB bits [#]	Routing bits [#]
Add8	100	9.785	2,560	7,225
Add16	103	11,963	2,656	9,307
Mul4 (a)	51	5,448	1,306	4,142
Mul4 (b)	42	5,443	1,107	4,336
Mul4 (c)	53	7,318	1,329	5,989
Mul8	127	17,448	3,280	14,168
Filter	132	20,501	3,401	17,091

We developed three different strategies of placement for the resources within the SRAM-based FPGA floorplan before running the fault injection campaigns. The strategies are the following:

1. *No constraints*: the place and route tool is let free to map the circuit in the whole FPGA area.
2. *Minimal Area Constraints*: the place and route tool is forced to produce the smallest possible design.
3. *Safe Area Constraints*: the place and route tool is forced to place each module of the TMR in a dedicated partition of the FPGA, so that two different module cannot share the same FPGA portion.

Due to the limited amount of resources of the adopted FPGA device, all circuits excepts the multiplier with 4-bit wide operands have been placed according to the strategy 1. Seven fault injection experiments are performed

using the above-described tool and strategies of placement. The results of the fault injection experiments are reported in the Table 2.7, where the *Injected Faults* column reports the number of SEUs we injected, and the *Wrong Answer* one reports the percentage of SEUs provoking SEFIs, i.e., the obtained output response differs from the expected ones. During the experiments, only one fault for each configuration memory has been injected, assuming that the fault injection time is equal to 0, i.e., SEUs affect the device right after it has been programmed. In the experiments a workload composed of all the possible input configuration was used. These results are particularly interesting since they experimentally show that the TMR architecture is not able to effectively harden the considered circuits against SEUs affecting the configuration memory. Indeed, the percentage of Wrong Answers can reach 13% for the largest circuits.

TABLE 2.7 Fault-injection results

Circuit	Injected faults [#]	Wrong answer [#]
Adder 8	9,785	982
Adder 16	11,963	1,349
Multiplier 8	17,448	2,300
Filter	20,501	2,708
Multiplier 4 (a)	5,448	33
Multiplier 4 (b)	5,443	27
Multiplier 4 (c)	7,318	17

5.1 Analysis of the fault effects

To better understand the causes of failure of the TMR architecture, each faulty configuration memory is analyzed using the developed classification tool [14, 24]. The result of the classification is reported in Table 2.8, where are reported the number of effects observed during the fault injection experiments, classified according to the affected resources (logic and routing) and to the produced effects (Open, Bridge, etc.). The effects are divided between *Routing faults*, provoked by any SEU that hits the bits controlling the programmable switches attached to the wire segments used to connect the FPGAs logic resource, and Logic faults, provoked by any SEU that hits the bit controlling the logic resources. For each kind of fault we can observe the following modifications induced by SEUs to the FPGA resource configuration. The effects classification report the following situation:

- *Routing*: the routing of a logic signal from the resource A to the resource B (track A/B) may be affected as follows:
  - *Open*: the track A/B is broken, and thus resource B is no longer fed with the expected logic value coming from resource A, which is instead left dangling.
  - *Bridge*: the track A/B is replaced with a new track C/B, and thus resource B is no longer fed with the expected logic value coming from resource A.
  - *Antenna*: a new track A/B is created linking a unused resource that could be connected far away in the FPGA routing topology, this may influence the behavior of the circuit since the resource associated to the output pad are driven to an unknown logic value.
  - *Conflict*: a new track C/B is created that overlaps with A/B. Resource B is driven by an unknown logic value which depends on the values coming from resources A and C.
  - *Others*: a modification of the track cannot be classified in any of the above classes.

- *Logic*: a logic resource may be affected as follows:
  - *LUT*: a bit controlling the LUT content is modified, this implies a modification of the logic function implemented.
  - *MUX*: a new MUX selection bit causes a new path of the signal.
  - *Initialization*: an initialization bit is modified provoking a modification of the behavior of the internal components of the CLB.

TABLE 2.8 Classification of the effects induced by SEUs

		Add8	Add16	Mul8	Mul4 (a)	Mul4 (b)	Mul4 (c)	Filter
		[#]	[#]	[#]	[#]	[#]	[#]	[#]
Logic	LUT	0	0	0	0	0	0	0
	MUX	206	52	112	2	1	0	293
	Initialization	50	22	66	1	1	0	331
Routing	Open	565	701	1,159	18	14	0	1,429
	Bridge	45	36	133	3	4	7	318
	Antenna	3	10	24	2	3	2	62
	Conflict	208	254	307	15	14	9	138
	None	0	0	0	0	0	0	0
	Others	236	383	501	7	4	4	450
	Total	1,313	1,458	2,302	48	41	22	3,021

In order to explain the effects we observed when analyzing the routing faults let us consider the fault-free configuration shown in Figure 2.9, where A_I, B_I, A_J, B_J, indicate four components, belonging to two different modules I and J, where I differs from J, of the TMR. The modifications that could be generated due to a SEU in the configuration memory are of two different types:

- *Conflict*: between track A_I / B_I and A_J / B_J of the TMR architecture as shown in Figure 2.10a.
- *Open and Bridge*: between track A_I / B_I and A_J / B_J of the TMR as shown in Figure 2.10b.

Analyzing the Logic faults, the effects that can provoke errors in the TMR architecture are mainly due to MUX shared between two different redundant modules I, and J. An error in such MUX may provoke multiple errors in both I, and J.

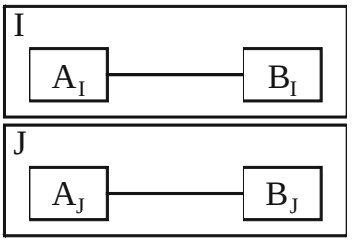

Figure 2.9. Fault-free configuration of two interconnections related to the redundant module I and J.

The Table 2.7 shows that the TMR architecture may fail to work properly because one SEU often produces one or more effects, at the same time, modifying the expected behavior of two or more different replicas of the TMR hardened circuit.

In terms of classification analysis, the main difference between combinational and sequential circuits is the increased number of effects within the logic resources, in particular in the MUX and initialization components. Moreover, the obtained result shows that most of the faults escaping the TMR architecture affect routing resources, and in particular, they are classified as Open.

The analysis of the gathered results, obtained from the study on the three different versions of the Multiplier, suggests a new approach to reduce the effects of multiple errors provoked by a single SEU. The obtained results show a progressive reduction of the number of faults: 33 for the experiment (a), 27 for (b) and finally 17 for the case (c). These results show a significant

reduction in the number of faults in relation to a smaller placing and routing that isolate each different module of the TMR architecture. In particular the version (c) of the multiplier is not affected by open errors.

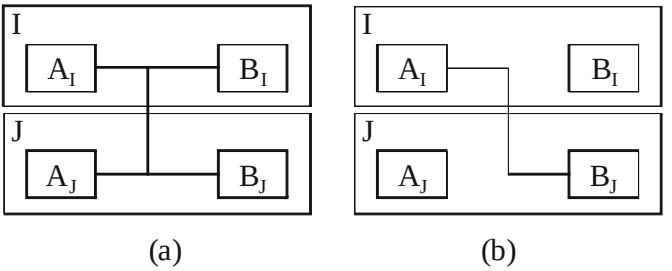

Figure 2.10. (a) Conflict between track A_I/B_I and A_J/B_J. (b) Open and bridge between track A_I/B_I and A_J/B_J.

These results support the conclusion that the key for a better implementation of the TMR architecture resides in the clever selection of the placement and routing of TMR components.

6. CONSTRAINTS FOR ACHIEVING FAULT TOLERANCE

Considering the experimental analysis performed and presented in this chapter, the main conclusion is that an SEU affecting the FPGA's configuration memory may provoke multiple errors by changing the configuration of routing resources. As a result, hardening techniques developed according to the single fault assumption are not adequate to cope with the multiple effects of SEUs in the configuration memory controlling routing resources. In the considered analysis, many situations exist where one SEU provokes multiple errors in such a way that the TMR scheme is no longer able to mask the SEUs effects [14].

As an example of this situation, referring to the Figure 2.2a, assuming that A_s/A_D and B_s/B_D are two routing edges belonging to two different replicas of the circuit hardened according to TMR. In this case, each SEU resulting in the erroneous configurations reported in Figure 2.2b–d violates the single-fault assumption.

This problem is particularly critical since 90% of the bits of FPGAs configuration memory are devoted to programming the routing resources. While it is possible that one upset may modify more than one routing edges, this becomes only when two routing edges from two different TMR replicas (i.e., domains) are affected.

In order to estimate the magnitude of the problem, it is considered one switch box and, for a given pair of routing edges (implemented by two PIPs of the same switch box) that belong to two different TMR replicas, then all the faulty configuration that are possible for a given routing architecture have been identified. In the presented study, the Xilinx Virtex family has been considered. For each faulty configuration, the corresponding image of the configuration memory has been computed. The faulty bitstream is then compared with the reference one and it is observed that they differ by one bit only. This means that one SEU may provoke multiple effects.

The aforementioned procedure has been repeated for all the faulty cases (i.e. short, open and open/short) and the computation reports that 72% of all the configuration memory bits controlling the considered switch box could produce critical situations if used for routing different TMR replicas. All the switch boxes with the FPGA are equal and, therefore, the above considerations are general. An example of such a kind of analysis related to the Short fault effect is illustrated in Figure 2.10a.

As a result, unless suitable countermeasures are developed, the TMR approach is no longer suitable for achieving fault tolerance.

Following the analysis performed on FPGAs architecture and on the organization of the FPGAs configuration memory, the constraints are used to enforce place and route algorithm in order to develop circuit implemented with TMR that are resilient to multiple errors:

1. All the circuit modules and connections must be replicated three times.
2. The outputs of the three circuit replicas must be voted according to the TMR principle.
3. The elements of the resulting TMR architecture (logic functions and connections among them) must be placed and routed in such a way that, given the corresponding routing graph, each new edge that is added (or deleted) to (from) the graph cannot provoke any fault belonging to the following categories:

   (a) Short between different connections belonging to different circuit replicas
   (b) Open affecting different connections belonging to different circuit replicas

REFERENCES

[1] T. P. Ma, P. V. Dressendorfer, *Ionizing Radiation Effects in MOS Devices and Circuits*, Wiley, New York, 1989, ISBN: 0-471-84893-X.

[2] J. L. Barth, C. S. Dyer, E. G. Stassinopoulos, *Space, Atmospheric, and Terrestrial Radiation Environments*, IEEE Transaction on Nuclear Science, Vol. 50, No. 3, June 2003, pp. 466–482.

[3] M. Ceschia, A. Paccagnella, S. -C. Lee, C. Wan, M. Bellato, M. Menichelli, A. Papi, A. Kaminski, J. Wyss, *Ion Beam Testing of ALTERA APEX FPGAs*, NSREC 2002 Radiation Effects Data Workshop Record, Phoenix, AZ, July 2002.

[4] R. Katz, K. LaBel, J. J. Wang, B. Cronquist, R. Koga, S. Penzin, G. Swift, *Radiation Effects on Current Field Programmable Technologies*, IEEE Transaction on Nuclear Science, Vol. 44, No. 6, Dec. 1997, pp. 1945–1956.

[5] Jih-Jong Wang, Brian E. Cronquist, Benny Sin, Jennifer J. Moriarta, Richard B. Katz, *Antifuse FPGA for space applications*, RADECS 1997.

[6] R. Koga, S. Penzin, K. Crawford, W. Crain, *Single Event Functional Interrupt (SEFI) Sensitivity in Microcircuits*, the Aerospace Corporation, 1998.

[7] M. Wirthlin, E. Johnson, N. Rollins, M. Caffrey, P. Graham, *The Reliability of FPGA Circuit Designs in the Presence of Radiation Induced Configuration Upsets*, 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003, pp. 133–142.

[8] E. Fuller, M. Caffrey, A. Salazar, C. Carmichael, J. Fabula, *Radiation Testing Update, SEU Mitigation and Availability Analysis of the Virtex FPGA for Space Re-configurable Computing*, presented at the IEEE Nuclear and Space Radiation Effects Conference, July 2000.

[9] M. Bellato, M. Ceschia, M. Menichelli, A. Papi, J. Wyss, A. Paccagnella, *Ion Beam Testing of SRAM-Based FPGA's*, IEEE Radiation Effects Data Workshop, July 2002.

[10] M. Alderighi, F. Casini, S. D'Angelo, F. Faure, M. Mancini, S. Pastore, G. R. Sechi, R. Velazco, *Radiation Test Methodology of SRAM-Based FPGAs by Using THESIC+*, IEEE 9th On-Line Testing Symposium, 2003, pp. 162.

[11] M. Bellato, P. Bernardi, D. Bortolato, A. Candelori, M. Cerchia, A. Paccagnella, M. Rebaudengo, M. Sonza Reorda, M. Violante, P. Zambolin, *Evaluating the Effects of SEUs Affecting the Configuration Memory of an SRAM-Based FPGA*, IEEE Design Automation and Test in Europe, 2004, pp. 188–193.

[12] B. L. Bhuva, J. J. Paulos, R. S. Gyurcsik, S. E. Kerns, *Switch-Level Simulation of Total Dose Effects on CMOS VLSI Circuits*, IEEE Transaction on Computer-Aided Design, Vol. 8, No. 9, Sept. 1989, pp. 933–938.

[13] M. P. Baze, S. Buchner, W. G. Bartholet, T. A. Dao, *An SEU Analysis Approach for Error Propagation in Digital VLSI CMOS ASICs*, IEEE Transaction Nuclear Science, Vol. 42, No. 6, Dec. 1995, pp. 1863–1869.

[14] M. Ceschia, M. Violante, M. Sonza Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, A. Candelori, *Identification and Classification of Single-Event Upsets in the Configuration Memory of SRAM-Based FPGAs*, IEEE Transaction on Nuclear Science, Vol. 50, No. 6, Dec. 2003, pp. 2088–2094.

[15] R. Velazco, S. Rezgui, R. Ecoffet, *Predicting Error Rate for Microprocessor-Based Digital Architectures Through C.E.U. (Code Emulating Upsets) Injection*, IEEE Transaction Nuclear Science, Vol. 47, Dec. 2000, pp. 2405–2411.

[16] J. Rose, A. El Gamal, A. Sangiovanni-Vincetelli, *Architecture of Field-Programmable Gate Arrays*, IEEE Proceedings, Vol. 81, No. 7, July 1993, pp. 1013–1029.

[17] F. Lima, C. Carmichael, J. Fabula, R. Padovani, R. Reis, *A Fault Injection Analysis of Virtex FPGA TMR Design Methodology*, IEEE European Conference on Radiation and Its Effect on Component and Systems, 2001, pp. 275–282.

[18] M. Alderighi, F. Casini, S. D'Angelo, M. Mancini, A. Marmo, S. Pastore, G. R. Sechi, *A Tool for Injecting SEU-Like Faults into the Configuration Control Mechanism of Xilinx Virtex FPGAs*, 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems, 2003, pp. 71–78.

[19] M. Alderighi, S. D'Angelo, M. Mancini, G. R. Sechi, *A Fault Injecting Tool for SRAM-Based FPGA*, 9th IEEE On-Line Testing Symposium, 2003, pp. 129–133.

[20] C. Carmichael, M. Caffrey, A. Salazar, *Correcting Single-Event Upset Through Virtex Partial Reconfiguration*, Xilinx Application Notes, XAPP216, 2000.

[21] C. Carmichael, *Triple Module Redundancy Design Techniques for Virtex FPGAs*, Xilinx Application Notes, XAPP197, 2001.

[22] F. Lima, L. Carro, R. Reis, *Designing Fault Tolerant System into SRAM-Based FPGAs*, IEEE/ACM Design Automation Conference, June 2003, pp. 650–655.

[23] Xilinx Inc., *Spartan-II 2.5 V FPGA Family: Introduction and Ordering Information*, Xilinx Product Specification Datasheets, 2003.

[24] M. Violante, L. Sterpone, M. Ceschia, D. Bortolato, P. Bernardi, M. S. Reorda, A. Paccagnella, *Simulation-Based Analysis of SEU Effects in SRAM-Based FPGAs*, IEEE Transactions on Nuclear Science, Vol. 51, No. 6, Dec. 2004, pp. 3354–3359.

Chapter 3

ANALYTICAL ALGORITHMS FOR FAULTY EFFECTS ANALYSIS

Single and multiple upsets errors

Reconfigurable FPGAs are very appealing as a replacement of ASICs for low-volume designs. FPGAs offer performance levels close to that of ASICs, plenty of resources to implement even very complex systems, as well as the possibility of performing in-the-field-reprogrammability.

In order to adopt successfully and safety these advantages, developers of safety-or mission-critical applications have to guarantee that the obtained FPGA-based systems meet the needed dependability levels. As deeply described in Chapter 2, SRAM-based FPGAs are particularly sensitive to upsets induced by energetic particles [1, 2] and thus they cannot be straightforwardly adopted in safety-or mission-critical applications, like space-borne ones.

As far as upsets affecting the memory elements the FPGA-based system embeds, two problem must be addressed: the protection of the users memory elements and that of the configuration memory.

Users memory elements (registers, memory arrays, etc.) must be hardened against these effect (either single and multiple) that may alter the information the system stores thus provoking temporary disruption of the service the system delivers. The disruption can be considered temporary since, assuming that the users memory is both read and written during systems activity, the disruption lasts as soon as a new (correct) value is written in the memory element the event affected.

Similarly, FPGAs configuration memory must be hardened against the occurrence of such effects, too. Being composed of SRAM cells, the configuration memory content may be altered by energetic particles hitting the FPGA, and therefore the vital information the memory holds, which defines which function the FPGA implements, may be altered. By changing the implemented function, upsets modifying the FPGAs configuration memory

L. Sterpone, *Electronics System Design Techniques for Safety Critical Applications*,

may alter dramatically the service system delivers. This type of disruption can be seen as permanent: the configuration memory is normally written once, at system power-up, and therefore when altered, the memory content cannot be restored until the next power-up (unless hardening strategies are used that periodically rewrite the configuration memory).

As described in the previous chapter, several approaches were proposed to cope with the abovementioned issues, making the deployment of SRAM-based FPGAs in mission or safety-critical applications feasible. However, no matter which type of hardening strategy is adopted, the designers have to validate the resulting system to prove that the needed dependability level is reached. For this purpose different approaches are available. These approaches may be grouped in the following categories:

- *Accelerated radiation ground testing*, where prototypes of the analyzed systems are exposed to suitable radiation beams.
- *Fault injection*, where upsets are inoculated either in prototypes or in simulation models of the analyzed system.
- *Analytical computation*, where models of the analyzed systems are studied by resorting to probabilistic techniques.

In this chapter, a new analytical algorithm is described that provides accurate estimation of the effects of Single Event Upsets (SEUs) and Multiple Cell Upsets (MCUs) inside FPGA-based designs. The main novelty the algorithm introduces is the possibility it offers of analyzing any SEU location within a design (by considering both user memory and configuration memory), and of identifying whether the SEU provokes any observable effect (i.e., the modifications induced by the SEU propagate from the SEU location to the systems outputs). The approach identifies all the memory elements (either belonging to the user memory, or to the configuration memory) that have to be hardened in order to make a design insensitive to SEUs effects. It thus improves the capabilities of already available analytical approaches that, to the best of our knowledge, provide only statistical estimations (although valuable) of SEUs effects.

Several approaches are available for analyzing the effects of SEUs in FPGA devices.

Accelerated radiation ground testing is an effective solution for estimating the effects on both the memory elements used by the design the FPGA implements as well as the FPGAs configuration memory [2, 3]. This kind of technique requires a prototype of the system under analysis, which is exposed to a flux of radiations, originated either by radioactive sources or by particle accelerators, which interacts with both the designs memory elements and the configuration memory. Radiation testing strategies aiming at validating the robustness of a design (i.e., computing its dynamic cross section) are usually based on the continuous monitoring of the outputs of the circuit

implemented on the FPGA under test, which is continuously stimulated by a given set of input stimuli. As drawbacks, radiations have the capability of permanently damaging the device under test and the costs needed by the experimental setup and by the beam time are not negligible.

As an alternative to radiation testing, several fault-injection approaches were recently proposed. Fault injection is an attractive technique for the evaluation of design characteristics such as reliability, safety and fault coverage [4]. The process involves inserting faults into particular targets in a system and monitoring the results to observe the produced effects. All these approaches emulate the effects of SEUs in the FPGAs memory as bit-flips in the bitstream that is downloaded in the FPGA in the programming phase. Some of them use run-time reconfiguration [5], while others modify the bitstream before downloading it in the device configuration memory or during download operations [6, 7]. Although the fault injection approaches permit to evaluate the effects of SEUs in all the memory bits, the time needed by the fault injection process is still huge, even in the case the process is optimized by the use of partial reconfiguration.

To overcome the time-consuming processes needed by the fault-injection approaches and to avoid the high cost of radiation testing, analytical approaches based on synthesis tools and software programs, only, are proposed in [8, 9]. In [8] a static estimation of the mapped designs susceptibility to SEUs is proposed assuming that all the bits of a design are susceptible at all times. Differently, in [9] an approach is proposed that identifies the paths sensitive to SEUs by calculating the probability error rate of all circuit nodes and by combining it with the error propagation probability of each net within the design. Then, the obtained information is coupled with the sensitivity of the FPGAs configuration memory bits. These approaches are either very pessimistic or able to provide only probabilistic estimations of SEU effects. In our approach, we analyze the topology of the design implemented on the SRAM-based FPGA and we couple this analysis with a set of reliability constraints. Thanks to this approach we are able to achieve the same accuracy of more time-consuming approaches like fault injection, while the execution time our approach demands is orders of magnitude lower.

Two different versions have been developed, the first one addressing the Single Event Upsets (SEUs) while the second ones addressing the Multiple Cell Upsets (MCUs).

1. OVERVIEW ON STATIC ANALYSIS ALGORITHM

The main purpose of the proposed algorithm is to analyze the effects of upsets in both the user memory elements and the FPGA's configuration

memory early in the design phase, as soon as the placed and routed model of the designed circuit is available.

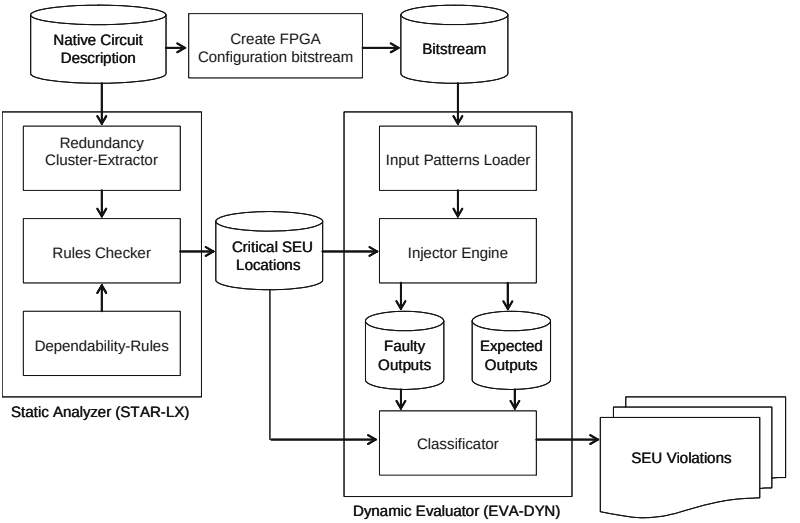

Figure 3.1. The architecture of the proposed approach. The figure reports the Static Analyzer and the Dynamic Analyzer tool chain.

When the SEUs are considered, the developed flow is illustrated in Figure 3.1, which depicts the *Static Analyzer* (STAR) algorithm. It is the tool that checks whether the placed and routed circuit is sensitive to upsets affecting either the memory elements the designers embed in the circuit, as well as the configuration memory of the SRAM-based FPGA implementing the circuit. It is composed of three modules:

- *Redundancy Cluster-Extractor*: it is a module that reads the Native Circuit Description and extracts the place and route information related to each cell of the FPGA architecture. That information is processed by means of a clustering process that groups the data depending on the FPGA topology architecture and on the redundancy structure of the adopted hardening technique.
- *Dependability-Rules*: it is a database of constraints related to the topology architecture of the FPGA that must be fulfilled by the placed and routed circuit in order to be resilient to the effects provoked by SEUs.
- *Rules-Checker*: it is the algorithm that reads each cluster and analyze every bit of the user memory and the configuration memory the FPGA has. It returns a list of SEUs (*Critical SEU Locations*) that introduces critical modification that may overcome the TMR hardening technique adopted.

Furthermore the flow is based on two description files: the *circuit description* and the *layout description*. The circuit description is a file containing the structural description of the circuit, which consists of logic functions (either combinational or sequential) and connections between them. Both the logic functions and the connections between them are described in terms of FPGAs resources. The layout description is a file containing the description of where each resource in the Circuit Description file is placed and routed on the FPGA area.

The classification is then performed by the *Dynamic Evaluator* (EVA-DYN). It is the platform that performs the dynamic evaluation of the SEU effects on the analyzed circuit. It is based on the fault injection approach. This platform applies the desired input patterns to the circuit description. An injector engine is devoted to create a faulty configuration memory bitstream according to the SEU location classified by the STAR tool. Finally a *report of violation* is generated that contain a list of all the violations of the Dependability Rules that the static analyzer identify. Each entry of the file describes the memory element, and the FPGAs resource responsible for the violations.

Given the circuit and layout descriptions, the static analyzer verifies whether all the constraints described in the dependability rules are fulfilled. In case any violation is found an entry is stored in the report of violations file.

2. ANALYTICAL DEPENDABLE RULES

The dependability rules, as described in the first chapter, must be enforced by a circuit implemented on SRAM-based FPGA in order to be resilient to the effects of SEUs. In particular, the rules guarantee that any SEU affecting either the memory elements the circuit uses and the FPGAs configuration memory is not able to propagate to the circuit's outputs. The dependability rules implemented are the results of an in-depth investigation of the effects of upsets in the memory elements of SRAM-based FPGA's designs. It has been observed that one and only one configuration memory bit *B* modifies two or more routing segments provoking multiple effects. A detailed analysis of these effects can be found in [10]. However, when TMR hardening technique is used, further considerations should be done. A TMR circuit may include voter partition logics. A *voter partition logic* may be defined as the logic resources (both sequential and combinational) that is comprises between two voter's structures. Considering the TMR scenario described in Figure 3.2, a voter partition logic consists in the logic domain D_i with $I \in (1, 2, 3)$ comprises between voter structure V_i and V_{i+1}. The modification that

may be introduced are deeply investigated in [10] and can be grouped in three distinct cases: *Short*, *Open* and *Open/Short*.

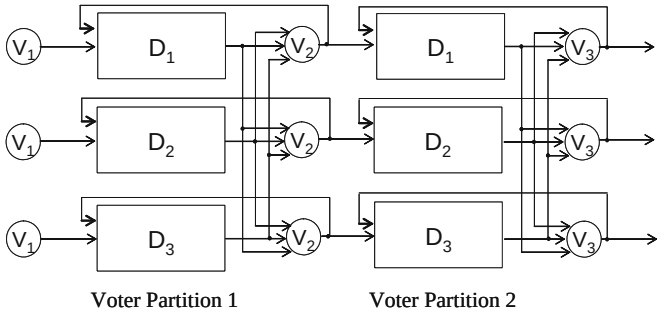

Figure 3.2. The Triple Modular Redundancy structure voter scenario.

These modifications may introduce critical behavior in the TMR structure illustrated in Figure 3.3. For example, considering the TMR scenario shown in Figure 3.3, an SEU may induce an open effect on two signals (i.e., the output signals of the FFs A1 and A2) provoking the multiple error in all the outputs of the TMR structure.

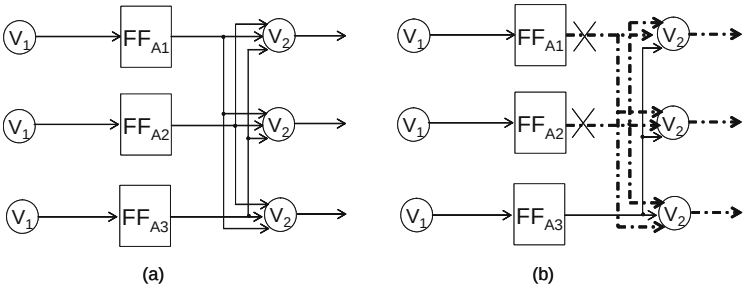

Figure 3.3. Open effect induced by an SEU example on a TMR scenario. (a) original condition, (b) open effect.

3. THE STAR ALGORITHM FOR SEU ANALYSIS

The STAR tool for the SEU analysis is based on a clustering algorithm which works on the FPGA architectural graph model described in the previous chapters. The flow of the cluster algorithm implemented is illustrated in Figure 3.4. When analyzing a circuit, the STAR algorithm performs three distinct phases.

The first phase reads the native circuit description and creates two sets: the first contains the routing resources related to each voter partition logic (P_i) while the second stores the logic resources related to each TMR domain (D_j).

```
STAR()
  {
   /*Reading_native_resources*/
   set_voter_partitions (Pi)
   set_tmr_domains (Dj)
   /*Redundancy-Clusters*/
   create_cluster_sets (CS(x,y), HS(x,y))
   for each voter_partition VP ∈ Pi
     for each tmr domains TD ∈ Dj
      {
       CS(x,y) = cluster_databits(VP, TD )
       HS(x,y) = cluster_hierarchy_tree(VP, TD)
      }
   /*Rules-Checker*/
   for each cluster C ∈ CS(x,y)
     for each bit location B ∈ C
       {
        UL = create_upset_list (B, HS(x,y))
        Check_dependability (UL, HS(x,y))
       }
```

Figure 3.4. The flow of the Static Analyzer algorithm (STAR) developed.

The second phase creates two cluster sets that groups information of a selected area of the FPGA matrix, where parameters x and y correspond to the row and column of the FPGA array: *CS*(x, y) which contains an array of the memory bit related to both the user memory and to the configuration memory controlling the logic and the routing resources, and *HS*(x, y) which contains the routing graph correspondent to the selected FPGA location. The routing graph contained in each cluster *HS*(x, y) is colored according to the information related to the voter partition logic and to the TMR domain. Two kind of marks are used, a first mark is assigned considering that the circuit is designed according to the TMR principle, three different colors are used for vertices belonging to each TMR domain. The second mark is an index that identifies the correspondent voter partition logic.

The third phase checks the effects that may be generated by SEUs that affect the memory bit contained in each cluster *CS*(x,y). This phase consists of the following steps:

1. A bit within the cluster set *CS*(x,y) is considered as SEU sensitive.
2. It is generated a list of SEU inducing modification into the circuit. This list includes the logic or routing vertices involved in the modification due to the SEU sensitive bit. These vertices are marked as faulty.
3. The dependability evaluation is performed by the function *Check_dependability* (). The routing tree contained within the cluster set *HS*(x,y) is updated generating a SEU propagation tree that contains all the paths stemming from the vertices marked as faulty, to the first voter's structure. If the leaves of the propagation tree include more than one graph color

and only one voter partition logic index, the correspondent bit is added to the Critical SEU Locations. Indeed, when this condition is met, the SEU effect is propagated to two or more circuit domains within the same voter partition logic, and therefore the TMR principle is no longer enforced.

3.1 The dynamic evaluation platform

The dynamic evaluation platform consists of a fault-injection environment which allows to evaluate the dynamic effects on the circuit under test of the critical SEUs reported by the static analyzer tool.

The fault injection system is composed of the following modules: an host computer, an FPGA board equipped with a Xilinx Virtex II-Pro device, and a serial communication link supported by a RS-232 cable that connects the FPGA board to the host computer. The host computer is preliminary used for configuring the Virtex-II Pro device and then for the generation of the input patterns.

The architecture of the proposed fault injection system is completely implemented on the FPGA device, which its layout is depicted in Figure 3.5.

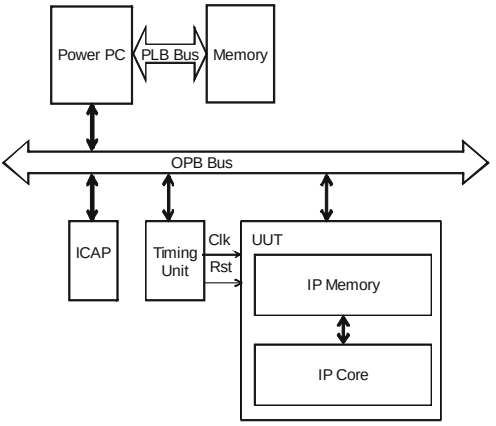

Figure 3.5. Architecture of the fault injection environment.

Four components are mapped on the Virtex-II Pro FPGA, all of them are interconnected by an On-chip Peripheral Bus (OPB):

- *Timing Unit*: it drives the UUT clock and reset.
- *Unit Under Test (UUT)*: it is the circuit under test and it may consist of an IP core and an own memory.
- *ICAP*: it is the Internal Configuration Access Port provided by last generations of Xilinx FPGAs. It allows to access to the FPGA

configuration memory through an internal port in order to perform partial reconfiguration without the support of an external hardware.

- *PowerPC microprocessor*: it is hardwired in the FPGA device and it controls the fault-injection process.

The fault injection process consists of several steps. At first some preliminary operations are executed:

1. The input patterns and the Critical SEU locations are load within the internal memory connected through the PLB bus to the PowerPC.
2. The PowerPC starts the execution of the UUT applying the input patterns. The obtained outputs are stored within the Expected output memory.

Once the preliminary operations are completed, the PowerPC selects the configuration memory bit from the Critical SEU locations and through the ICAP port performs partial reconfiguration of the frame where the SEU has to be injected. It starts the execution of the UUT application. During the execution, it stores within the faulty output the data generated by the UUT. Finally, at the end of the execution, the FPGA control board compares the expected and the faulty memories in order to identify if a mismatch is found. Once a mismatch is identified the bit-flip information are transferred to the SEU violations file. This process is repeated for all the Critical SEUs identified by the STAR algorithm.

3.2 Experimental results of SEU static analysis

In this section we describe the experiments has been performed to evaluate the efficiency of the proposed SEU estimation methodology.

A prototype of the STAR algorithm has been developed, that accounts for 64K lines of ANSI C code and of the dynamic evaluation platform using a Xilinx Virtex-II Pro XC2VP30 as FPGA under test. The developed experiments aim at analyzing the capability of the proposed methodology of detecting SEUs in the FPGA logic and routing structures in the configuration memory of FPGAs that implement circuits hardened according to the TMR approach. Three different circuits hardened through Xilinx TMR (X-TMR) approach [11] have been developed: a FIR filter, a microprocessor core implementing the Intel 8051, and the PicoBlaze microcontroller. During our experiment the FPGA has been configured at the working frequency of 100 MHz. Furthermore the internal ROM of the 8051 and PicoBlaze has been initialized with an Elliptic filter program working on 64 samples.

The results achieved are illustrate in Table 3.1 where are reported the number of SEUs identified as critical for the exhaustive fault injection analysis performed injecting in all the possible memory locations related to the circuit under test.

TABLE 3.1 Experimental results reporting the computational time needed and the critical SEUs identified

Circuit	Exhaustive fault injection	
	Time [min]	Critical SEUs [#]
FIR X-TMR	30.4	82
8051 X-TMR	180.2	103
Pico X-TMR	91.2	95
Circuit	STAR-LX	
	Time [min]	Σ SEUs [#]
FIR X-TMR	5.1	105
8051 X-TMR	7.5	120
Pico X-TMR	5.2	115
Circuit	EVA-DYN	
	Time [min]	Φ SEUs [#]
FIR X-TMR	9.2	82
8051 X-TMR	10	103
Pico X-TMR	9.3	95

It is reported also the list of SEUs generated by the STAR algorithm that may modify the system (Σ) and the critical SEUs finally identified by the dynamic evaluation platform (Φ).

The number of critical SEUs detected by the proposed algorithm is exactly that obtained by extensive fault injection. Besides, the proposed methodology is one order of magnitude faster on the average than exhaustive fault injection approach. Furthermore, the computational time is not proportional with the circuit complexity. The results have been validated comparing the location of critical SEUs obtained from the STAR algorithm and the others coming from fault injection experiments.

4. THE STAR ALGORITHM FOR MCU ANALYSIS

One critical issue to enable using not rad-hard SRAM-based FPGAs in the space environment is the capability of mitigating the effects induced by upsets within the device's configuration memory. In addition to Single Event Upsets (SEUs), Multiple Cell Upsets (MCUs) provoked by ionizing radiations have been observed in SRAM-based memory devices [12, 13]. MCUs within not rad-hard SRAM-based FPGAs have been observed during radiation experiments with proton and heavy ions [14]. In particular, a study that quantifies the occurrence of protons- and heavy-ions effects on four different Xilinx FPGA's families indicates that the newer families (such as Virtex-II and Virtex-IV) are increasingly sensitive to MCUs. As a result, the MCU cross-sections of the newer Xilinx Virtex-II increase by two orders of magnitude if compared with the previously manufactured family Xilinx Virtex-I.

Few data are recently observed on the effectiveness of TMR hardening technique when coping with MCUs. The research expects that as MCUs produce multiple upsets in the configuration memory, they are more likely than SCUs to induce domain-crossing events that may affect two or more TMR modules, thus limiting the effectiveness of TMR [15].

In this chapter is presented a new analytical-oriented methodology for the estimation of MCU-induced effects inside the FPGA's configuration memory while TMR design techniques are adopted. The developed analytical method allows to analyze 2-bits MCUs sensitiveness of TMR circuits implemented on SRAM-based FPGAs.

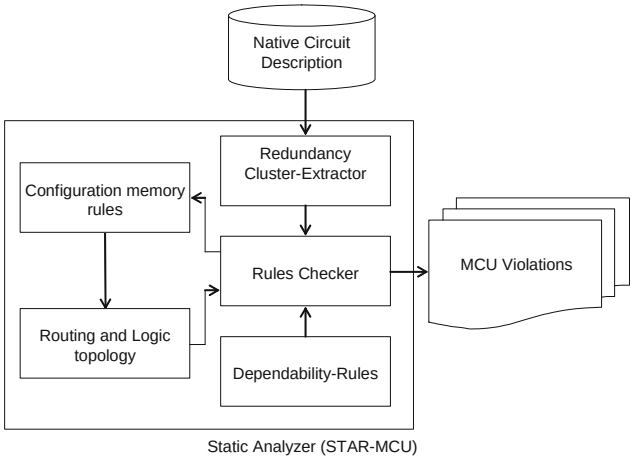

Figure 3.6. The flow of the STAR algorithm for the analysis of MCUs.

The flow of the proposed methodology is depicted in Figure 3.6. The *STAR* algorithm for the analysis of MCU is composed of the following elements:

- *Native Circuit Description*: It is a file containing the structural and layout descriptions of the circuit, which consists of logic functions (either combinational or sequential) and connections between them. Both the logic functions and the connections between them are described in terms of resources placed and routed on the FPGA.
- *Static Analyzer MCU*: It is the tool that checks the placed and routed circuit analyzing the sensitive MCU locations affecting the memory elements the design embeds and the configuration memory. It is composed by five modules: the *Redundancy Cluster-Extractor*, the *Configuration memory rules*, the *Routing and logic topology*, the *Dependability Rules* and the *Rules Checker*.

The Redundancy Cluster-Extractor is a module that reads the Native Circuit Description and extracts the place and route information related to

each cell of the FPGA architecture. This information are processed by a clustering algorithm that groups the data depending on the FPGA topology architecture and on the redundancy structure of the adopted hardening technique. The configuration memory rules is a data-base related on the physical layout of the FPGA's configuration memory cells. It contains the configuration memory coding of all the resources of a CLB, while the functionality of the CLB's logic and the interconnections architecture effectively programmed are identified by the configuration memory rules. The information about the routing and the logic internal structure of the SRAM-based FPGA device are stored within the Routing and Logic topology.

The Dependability-Rules is a data-base of constraints related to the topology architecture of the not rad-hard FPGA that must be fulfilled by the placed and routed circuit in order to be resilient to the effects provoked by MCUs. The Dependability-Rules are used by the Rules-Checker algorithm that reads each cluster and analyze all the bits of the FPGA's configuration memory. It returns a list of MCUs (*MCU Violations*) that provoke critical modifications that may overcome the adopted hardening technique.

4.1 Analysis of errors produced by MCUs

As discussed in the Chapter 2, the dependability rules must be adopted by circuits implemented on not rad-hard SRAM-based FPGA in order to be resilient to the effects of SCUs. In particular, the rules guarantee that any SCU affecting either the memory elements the circuit uses or the FPGAs configuration memory is not able to propagate to the circuit's outputs. When considering MCUs induced by a single particle affecting two cells of the FPGA's configuration memory further considerations are needed, which include how the redundancy structure is laid out on the FPGA.

The MCU's effects have been analyzed considering clusters of adjacent configuration memory bits as illustrated in Figure 3.7a. In Figure 3.7b, is illustrated the resources possibly affected by MCUs. They belong to the following sets: CLBs, Block RAMs (BRAMs), BRAMs interconnects, and IOBs. Each resource's set is controlled by a defined number of configuration memory frames where each frame corresponds to an FPGA's configuration column of SRAM cells [16]. Depending on the orientation of the MCU events (single column, row or diagonal adjacent cells), the provoked effects may simultaneously corrupt resources of a single set or two sets whose configuration's memory bits are adjacent.

Considering the TMR architecture represented in Figure 3.2, the modifications of SCUs can be grouped in two distinct cases: *Short* and *Open*. These modifications may introduce critical behavior in the TMR structure as described in the previous sections. The effects of MCUs can be modeled as

multiple SCUs that happen simultaneously. As an example considering the TMR scenario represented in Figure 3.2, an MCU may induce an open and short effects (i.e. the output signal of the FF_{A1} is opened, while the output signals of the FF_{A2} and FF_{A3} are shorted together) provoking multiple errors in all the outputs of the TMR structure. This effects is represented in Figure 3.8.

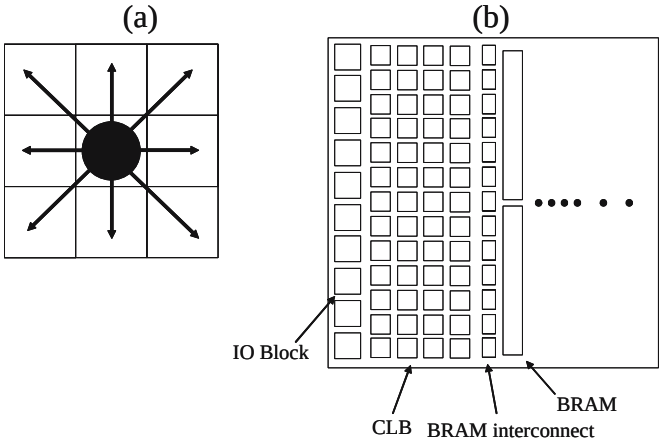

Figure 3.7. (a) Multiple Cell Upsets adjacent cells. (b) Configuration memory layout general organization of Virtex-II.

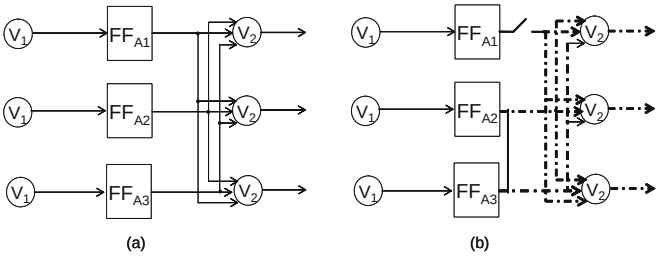

Figure 3.8. The TMR Voter Partition scenario. An example of MCUs effects (open/short).

The effects of MCUs can be defined considering the following parameters:

- *Orientation*: it defines the position of the MCU within the FPGA's configuration memory, as *single column*, *diagonal* or *single row.*
- *Case*: it defines the transitions induced by the MCU within the FPGA's configuration memory cells as 00→11, 01→10 / 10→01 or 11→00.
- *Effects*: it defines the effects induced by the MCU as *Short*, *Open*, *Short/Open*, *Logic* and *Logic-Routing* [17].

The classification of the effects can be further refined by considering the number of bits and the occurrence of the effects.

Considering a couple of vertices A_S/A_D and B_S/B_D linked by two distinct interconnection segments and controlled by two configuration memory bits each, as illustrated in Figure 3.9a. The following scenarios related to the interconnection resources is represented:

(a) *Open or Short 1-bit*: only one bit of the two cells affected by the MCU provokes a failure effect.
(b) *Double Open or Short*: both the bits of the two cells affected by the MCU provokes failure effects. In particular, each bit affects a distinct interconnection of the TMR structure. For example, it is reported in Figure 3.9b the double open effects when two different bits in a vertical orientation affect two separate interconnections.
(c) *Open or Short 2-bit*: both the bits of the two cells affected by the MCU provoke failure effects. In this case, both the bits are related to a single interconnection, and thus it does not corrupt the TMR structure. In the Figure 3.9c is described an example of an open 2-bit.
(d) *Open-Short*: both the bits of the two cells affected by the MCU provoke failure effects. In particular, one bit induces an Open effect and the other one a Short effect between distinct interconnections, as illustrated in Figure 3.9d.

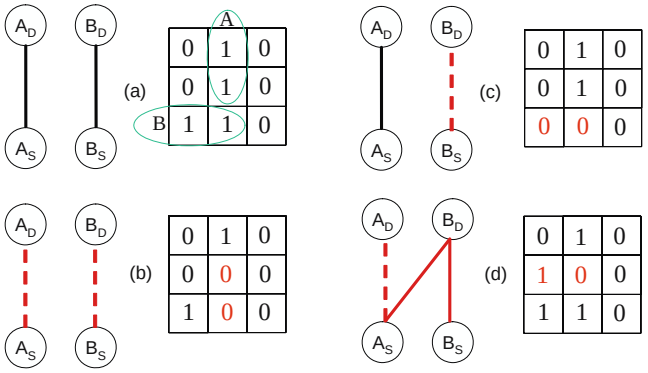

Figure 3.9. MCU fault effects scenario. The original configuration topology of the vertices A_S/A_D and B_S/B_D is defined by the configuration memory bits reported in (a). In (b) is illustrated a double open effects when two different bits in a vertical orientation affect two separate interconnections. In (c) is reported an open 2-bit; in this case both the involved bits are related to a single interconnection, while in (d) is reported an Open/Short effect.

When logic resources are considered, the following cases appear:

(a) *Logic Failure*: both the bits of the two cells affected by the MCU provoke a failure in a single logic block of the FPGA.

(b) *Logic-Routing Failure*: both the bits of the two cells affected by the MCU provoke failure effects. In particular, one cell controls logic resources and the other one control interconnections resources.

The STAR-MCU algorithm performs three distinct phases as illustrated in Figure 3.10: *reading native resources*, *redundancy clusters* and *rules checker*.

The reading native resources phase reads the native circuit description and creates two sets: one containing the routing resources related to each voter partition (P_i), and one containing the logic resources related to each TMR domain (D_j). In details, each *i* voter partition P_i contains the programmable interconnections, while each *j* TMR domain D_j contains all the logic resources such as LUTs, FFs or Multiplexers.

The redundancy cluster phase creates two clusters that store information about the configuration memory layout regarding any area (x, y) of the FPGA matrix, where x and y identify a row and a column of the FPGA array. In details a single Configuration Frame Rules *CFR(x,y)* cluster contains a matrix of bit related to both the user memory and the configuration memory controlling the logic and routing resources in the FPGA array (x,y). The bits are programmed reflecting the effective usage of that resources in

```
STAR_MCU()
  {
   /*Reading_native_resources*/
   set_voter_partitions (Pi)
   set_tmr_domains (Dj)
   /*Redundancy-Clusters*/
   create_cluster_sets (CFR(x,y), HS(x,y))
   for each voter_partition VP ∈ Pi
     for each tmr domains TD ∈ Dj
      {
       HS(x,y) = cluster_hierarchy_tree(VP, TD)
       CFR(x,y) = cluster_configuration_memory_rules(VP, TD )
      }
   /*Rules-Checker*/
   for each cluster C ∈ CFR(x,y)
     for each point location P ∈ C
       {
            /*MCU-engine*/
            for each orientation O
               {
             MCU_UL = create_MCU_upset_list (P, HS(x,y),O)
             RL_set = read_topology_rules(MCU_UL, C )
                 for each partition I ∈ RL_set
                  Check_dependability (RL_set, C, I )
             }
        }
    }
```

Figure 3.10. The flow of the STAR algorithm for the MCUs analysis.

the FPGA's CLB array at the coordinates (x,y). Each CFR cluster contains a bit matrix related to all the CLBs located at the coordinates (x,y) within the FPGA CLB array, and each matrix of bit contains the used bit marked accordingly with the correspondent routing signal or logic element. Vice versa, the cluster set Hierarchy Tree *HS(x,y)* contains the routing graph correspondent to the selected (x,y) FPGA location. The routing graph contained in each cluster *HS(x,y)* is colored according to the information related to the voter partition logic and to the TMR domain, where two nomenclatures are used. The first is a mark that is assigned considering that the circuit is designed according to the TMR principle, three different colors are used for all the vertices belonging to each TMR domain. The second, is an index that identifies the correspondent voter partition logic.

The rules-checker phase analyzes the effects that may be induced by MCUs affecting the user or the configuration memory cells. The core of this phase is characterized by the *MCU-engine*. It performs the analysis of all the MBU orientation reading the routing or logic topology from the topology data-base. This analysis is performed verifying if the dependability rules are satisfied for all the possible modifications of the routing graph description contained in the clusters *HS(x,y)* due to the MCU's induced modification.

The configuration memory rules contain all the configuration memory coding related to the FPGA's logic components and the routing topology of the implemented circuit. It is generated by the function `cluster_configuration_memory_rules` that reads the resources description belonging to a voter partition *VP* of a given logic domain *TD* for each location of the FPGA matrix architecture, it generates a cluster of bits *CFR*. Each cluster consists of a bit matrix where all the bits are organized reflecting the SRAM-based FPGA configuration and user memory architecture.

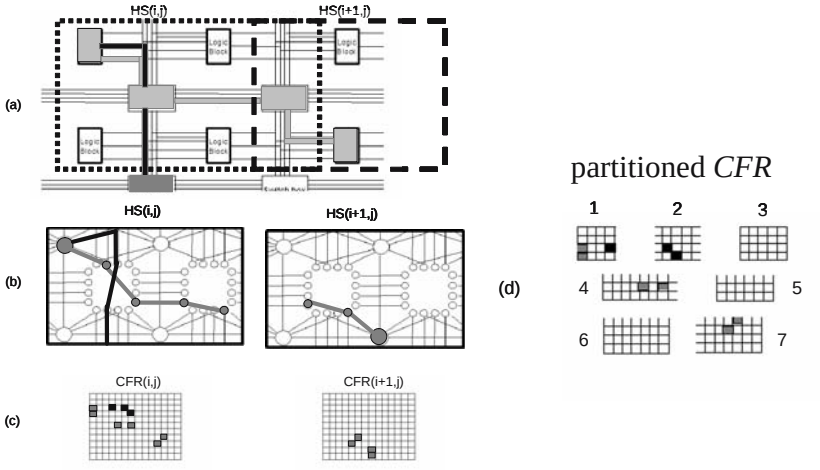

Figure 3.11. An example of generation of the cluster sets *HS* and *CFR*.

In Figure 3.11a is illustrated a model of an interconnection path between two logic vertices, while in Figure 3.11b the cluster *HS* generated according to the circuit model for each coordinates (i, j) describes a portion of the whole routing graph of the FPGA architecture considered. Vice versa, in Figure 3.11c the cluster *CFR* generated according to the circuit model, each cluster *CFR*(i,j) defines a bit matrix where each bit is marked as used (i.e. grey color) if the correspondent routing edge or logic vertex is used by the routing graph model of the implemented circuit. In Figure 3.11d each cluster *CFR* containing more than a logic components or routing signal is partitioned considering the FPGA topology architecture.

The segmentation of the CFR cluster data matrix allows to increase the speed of the analysis since it enables a rapid identification of the set of possible modifications affecting two or more routing signal. The reader should note that considering that the analysis will be focused on MCU, several bit-flip combinations need to be generated. The segmentation of the cluster CFR data matrix is aiming at reducing the computational time of the several MCU combination.

At the end of the execution of the redundancy cluster extractor phase, the clusters *CFR* and *HS* define an accurate model of the circuit that is mapped on the SRAM-based FPGA that consider both the user and configuration memory characteristics as well as the routing and logic organization of the FPGA adopted.

As illustrated in Figure 3.11, the clusters *HS* and *CFR* are generated according to the routing and logic topology of the circuit mapped on the FPGA architecture. Considering the circuit model illustrated in Figure 3.11a, which consists of a routing path between two logic vertices, the circuit description is read and stored within two clusters at the position *i, j* and *i + 1,j*. An example of the cluster generation is illustrated in the Figures 3.11b and c respectively. In Figure 3.11d is reported the segmentation of the CFR cluster *i, j* which contains more than one logic component or routing signal. Please note that the segmented matrices have different dimensions depending on the configuration memory bit organization that belong to any different FPGA's family. In details, each segmented matrix contains the configuration memory bits programming PIP or logic element with shared resources.

The rules-checker is the most crucial part of the developed Static Analyzer tool. It is the third phase of the STAR-MCU algorithm and it analyzes if the dependability rules are satisfied for all the possible MCUs affecting the user and the configuration memory bits.

This analysis is performed by three functions: `create_MCU_upset_list`, `read_topology_rules` and `Check_dependability`, that are executed for all the bits contained within the cluster set *CFR*.

In details the function `create_MCU_upset_list()` performs the following three steps:

1. It select a bit (*P*) in the position l, m within the bit-matrix of the cluster *CFR(x,y)*. As an example, in Figure 3.12 is illustrated the cluster *CFR(x,y)l,m* with l = f and m = 3.
2. It marks the selected bit i, m as SEU sensitive.
3. It generates a Multiple Cell Upset list (*MCU_UL*) that consists of the modifications introduced within the routing graph architecture. These modifications depend on the kind of resource interested:

   (a) *Routing*: the upset list is updated with the routing edge/edges that is/are added or deleted from the circuit routing graph model due to the modification induced by the bit-flip.
   (b) *Logic*: the upset list is updated with the kind of logic components interested. In that case, the modifications include: Look-Up Tables (LUTs), Multiplexers (MUXs) or Logic Configurations (CFGs).

As soon as the upset list MCU_UL is generated, the function `read_topology_rules` divides the considered set *CFR(x,y)* in several partition containing all the configuration memory bits that are controlling a given routing architecture or logic component within a CLB. Each bit is therefore associated to a proper segmented area of the CFR cluster. The segmentation is performed according to the technological characteristic of the analyzed FPGA device. These characteristics are related to the interconnection architecture (routing segments topology) and the CLB granularity (i.e. number of FFs embedded in each logic element or LUT's dimensions). Please consider that in the case the segmentation is performed on the configuration memory boundaries, the segmented areas are overlapped, including the bit locations that are located in the physically adjacent places.

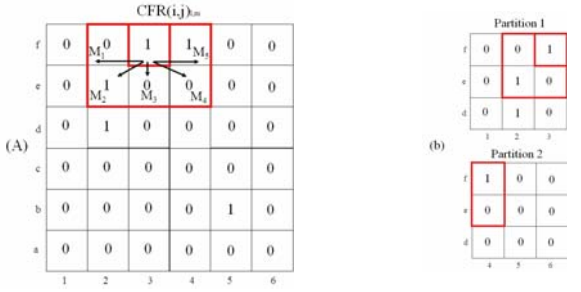

Figure 3.12. An example of generation of the partition set RL_set.

An example of the generation of those partitions is illustrated in Figure 3.13. Given a set of routing interconnections, represented by the routing

segments S_1, S_2 and S_3. The CFR cluster is generated by the function `cluster_configuration_memory_rules` as described in the previous section, according to the configuration memory bits that enable the three routing segments, as illustrated in the Figure 3.13a. The cluster is then partitioned according to the FPGA's configuration memory technological characteristics. In details, these characteristics are related to the routing granularity, since the dimension of each partition is defined considering the number of routing segments that may span from each programmable point.

In the example illustrated in Figure 3.13b, it is assumed that the configuration memory area is segmented in four parts.

Finally, this function generates the *RL_set* for the considered MCU orientation. Each set consists in one or more partitions of the cluster CFR that includes the configuration memory cells affected by the MCU. From the data contained within the set, the function `Check_dependability` performs the analysis of the MCU induced effects on the circuit.

The dependability rules are checked by the function `Check_dependability()`. This function executes the following steps:

1. It updates the cluster *HS(x,y)* introducing the modification included in the upset list RL_set.
2. The vertices of the clusters *HS(x,y)* involved in the modification are marked as faulty.

The routing tree contained within the cluster set *HS(x,y)* is updated generating a MCU propagation tree that contains all the paths stemming from the vertices marked as faulty, to the first voter's structure. If the leaves of the propagation three include more than one graph coloring and only one voter partition logic index, the correspondent bit is added to the Critical MCU Locations. Indeed, when this condition is met we have that the MCU effects propagated to two or more circuit domains within the same voter partition logic, and therefore the TMR principle is no longer enforced. The critical MCU locations contain for each MCU considered as critical, the position within the user or configuration memory of the SRAM-based FPGA used, as well as the indication of the kind of resources (logic or routing) and the name of the circuit component or net involved.

In order to give an example of this procedure, in Figure 3.12 we reported a data matrix correspondent to the generated cluster CFR(i, j). Considering all the possible orientation of a 2-cells MCU starting from the position (3, f) two partitions (i.e., the partition 1 and the 2) have been included in the *RL_set*, these partitions are illustrated in Figure 3.12b.

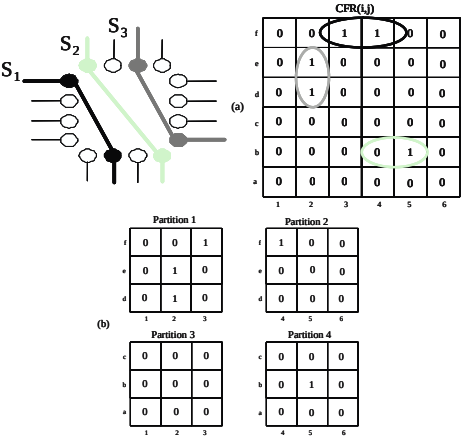

Figure 3.13. An example of the generation of the partition generated by the function `read_topology_rules`. Given a set of routing interconnection, the *CFR* cluster is generated (a). The cluster is then partitioned according to the FPGA configuration memory characteristics (b). As an example in the figure, the cluster is divided into four partitions.

On the basis of these two partitions the function `Check_dependability()` perform the analysis of each MCU effect. The several combination generated are depicted in the five cases M_i, as illustrated in Figure 3.14.

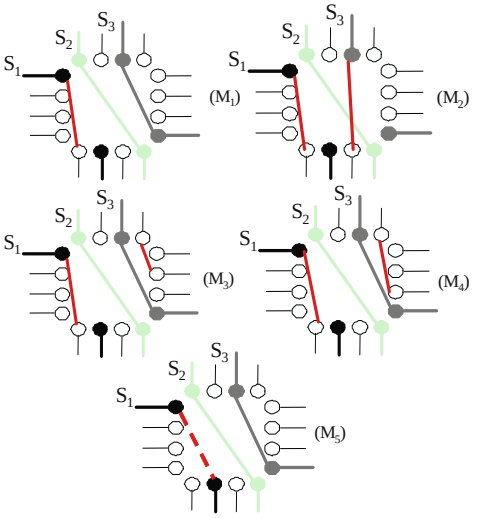

Figure 3.14. A list of modification generated according to the set RL_set related to the considered example.

4.2 Experimental results of MCU static analysis

In this section we describe the analytical analysis we performed to evaluate the MCUs sensitiveness of circuits implemented on not rad-hard SAM-based FPGAs when the TMR hardening technique is used. We implemented a prototype of the STAR-MCU tool that accounts for about 12K lines of ANSI C code.

We selected as case study two different circuits hardened according to the Xilinx TMR (X-TMR) approach [18] and implemented on a Xilinx Virtex-II XC2V1000 FPGA: a FIR Filter with 64-stages and a Cordic core DSP.

The characteristics of the considered benchmark applications are reported in the Table 3.2, where for each circuit we reported the number of Flip-Flops (FFs), 4-input Look-Up Tables (LUTs) and I/O Pins.

TABLE 3.2 Characteristics of the benchmark circuits

Resources	Cordic core processor	FIR filter
FFs [#]	3,315	1,588
4-input LUTs [#]	3,246	170
I/O pins [#]	225	19

The results we gathered are shown in Table 3.3, where we reported the number of SCUs identified as critical using the STAR approach presented in [17], and in Tables 3.4, 3.5 and 3.6, are reported the number of MCUs that overcome the X-TMR protection capabilities. The results show that the number of MCUs corrupting the TMR is 2.6 order of magnitude higher than the SCUs one. In details, we can observe that the majority of the critical MCUs are provoked in diagonal orientation, while the most relevant effects is the double short provoked by the transition 00→11. We omitted the classification of Logic failures, Short 2-bit, Open 1-bit and Open 2-bit since no effects have been observed for these cases.

Furthermore we recorded the computational time needed by STAR-MCU to evaluate the considered circuits. In both cases STAR-MCU takes about 940 s to perform a complete analysis.

TABLE 3.3 Critical SCUs identified

Circuits	Critical SCUs [#]	
	Multiple open	Short
FIR Filter	0	8
Cordic Core	0	3

TABLE 3.4 Critical MCUs identified by the proposed approach (I)

Circuits		Critical MCUs [#]	
FIR filter	Orientation	Total	Single column
		3,650	853
	Effects		Case 00→11
		Double short	2,116
		Short 1-bit	8
		Open-short	0
		Double open	0
Cordic core	Orientation	Total	Single column
		2,454	574
	Effects		Case 00→11
		Double short	1,385
		Short 1-bit	3
		Open-short	0
		Double open	0

TABLE 3.5 Critical MCUs identified by the proposed approach (II)

Circuits		Critical MCUs [#]	
FIR Filter	Orientation	Total	Diagonal
			1,911
	Effects		Case 01→10/10→01
		Double short	0
		Short 1-bit	0
		Open-short	4
		Double open	1,350
Cordic Core	Orientation	Total	Diagonal
			1,263
	Effects		Case 01→10/10→01
		Double short	0
		Short 1-bit	0
		Open-short	1
		Double open	894

TABLE 3.6 Critical MCUs identified by the proposed approach (III)

Circuits		Critical MCUs [#]	
FIR filter	Orientation	Total	Single row 886
	Effects		Case 11→00
		Double short	0
		Short 1-bit	0
		Open-short	0
		Double open	172
Cordic core	Orientation	Total	Single row 617
	Effects		Case 11→00
		Double short	0
		Short 1-bit	0
		Open-short	0
		Double open	171

REFERENCES

[1] E. Fuller, M. Caffrey, P. Blain, C. Carmichael, N. Khalsa, A. Salazar, *Radiation Test Results of the Virtex FPGA and ZBT SRAM for Space Based Reconfigurable Computing*, presented at the MAPLD Conference, Sept. 1999.

[2] M. Ceschia, A. Paccagnella, S. -C. Lee, C. Wan, M. Bellato, M. Menichelli, A. Papi, A. Kaminski, J. Wyss, *Ion Beam Testing of ALTEERA APEX FPGAs*, NSREC 2002 Radiation Effects Data Workshop Record, Phoenix, AZ, July 2002.

[3] R. Katz, K. LaBel, J. J. Wang, B. Cronquist, R. Koga, S. Penzin, G. Swift, *Radiation Effects on Current Field Programmable Technologies*, IEEE Transaction on Nuclear Science, Vol. 44, No. 6, Dec. 1997, pp. 1945–1956.

[4] D. K. Pradhan, *Fault-Tolerant Computer System Design*, Upper Saddle River, NJ, Prentice-Hall, 1996.

[5] F. Lima, C. Carmichael, J. Fabula, R. Padovani, R. Reis, *A Fault Injection Analysis of Virtex FPGA TMR Design Methodology*, in Proceedings IEEE European Conference on Radiation and Its Effect on Component and System, 2001, pp. 275–282.

[6] P. Bernardi, M. Sonza Reorda, L. Sterpone, M. Violante, *On the Evaluation of SEUs Sensitiveness in SRAM-Based FPGAs*, IEEE 10th On-Line Testing Symposium, 2004, pp. 115–120.

[7] M. Alderighi, S. D'Angelo, M. Mancini, G. R. Sechi, *A Fault Injection Tool for SRAM-Based FPGA*, 9th IEEE On-Line Testing Symposium, 2003, pp. 129–133.

[8] P. Sundararajan, B. Blodget, *Estimation of Mean Time Between Failure Caused by Single Event Upset*, Xilinx Application notes, XAPP559, Jan. 2005.

[9] G. Asadi, M. B. Tahoori, *An Analytical Approach for Soft Error Rate Estimation of SRAM-Based FPGAs*, presented at the MAPLD Conference, 2004.

[10] M. Ceschia, M. Violante, M. Sonza Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, A. Candelori, *Identification and Classification of Single-Event Upsets in the Configuration Memory of SRAM-Based FPGAs*, IEEE Transaction on Nuclear Science, Vol. 50, No. 6, Dec. 2003, pp. 2088–2094.

[11] C. Y. Lee, *An Algorithm for Path Connections and Its Application*, IRE Transaction on Electronic Computers, Vol. 10, No. 3, Sept. 1961, pp. 346–365.

[12] A. G. M. Swift, S. M. Guertin, *In-Flight Observations of Multiple-Bit Upset in DRAMs*, IEEE Transactions on Nuclear Science, Vol. 47, No. 6, Dec. 2000, pp. 2386–2391.

[13] B. R. Koga, K. B. Crawford, P. B. Grant, W. A. Kolasinski, D. L. Leung, T. J. Lie, D. C. Mayer, S. D. Pinkerton, T. K. Tsubota, *Single Ion Induced Multiple-Bit Upset in IDT 256K SRAMs*, in Proceedings 2nd Euro Conference on Radiation and Its Effects on Components and Systems, St. Malo, France, Sept. 1993, pp. 485–489.

[14] C. R. Koga, J. George, G. Swift, C. Yui, L. Edmonds, C. Carmichael, T. Langley, P. Murray, K. Lanes, M. Napier, *Comparison of Xilinx Virtex-II FPGA SEE Sensitiveness to Protons and Heavy Ions*, IEEE Transactions on Nuclear Science, Vol. 51, No. 5, Oct. 2004, pp. 2825–2833.

[15] D. H. Quinn, P. Graham, J. Krone, M. Caffrey, S. Rezgui, *Radiation-Induced Multi-Bit Upsets in SRAM-Based FPGAs*, IEEE Transactions on Nuclear Science, Vol. 52, No. 6, Dec. 2005, pp. 2455–2461.

[16] B. Bridgford, C. Carmichael, C. W. Tseng, *Correcting Single-Event Upsets in Virtex-II Platform FPGA Configuration Memory*, Xilinx Application Notes, XAPP779, Feb. 19, 2007.

[17] L. Sterpone, M. Violante, *A New Analytical Approach to Estimate the Effects of SEUs in TMR Architecture Implemented Through SRAM-Based FPGAs*, IEEE Transactions on Nuclear Science, Vol. 52, No. 6, Part 1, Dec. 2005, pp. 2217–2223.

[18] "TMRTool User Guide", Xilinx User Guide UG156, 2004.

Chapter 4

RELIABILITY-ORIENTED PLACE AND ROUTE ALGORITHM

Dependable design on SRAM-based FPGAs

In general, the commonly used design-flow to map designs onto a SRAM-based FPGA consist of three phases. In the first phase, a synthesizer is used to transform a circuit model coded in a hardware description language into an RTL design. In the second phase a technology mapper transforms the RTL design into a gate-level model composed of look-up tables (LUTs) and flip flops (FFs) and it binds them to the FPGA's resources (producing the technology-mapped design). In the third phase, the technology mapped design is physically implemented on the FPGA by the place and route algorithm.

The problem of how to physically implement a circuit on a FPGA device is divided into two sub problems: *placement* and *routing*. The main reason behind such decomposition is to reduce the problem complexity. Our proposed reliability-oriented place and route algorithm, called *RoRA*, firstly reads a technology mapped design. Then, it performs a reliability-oriented placement of each logic functions, and finally it routes the signals between functions in such a way that multiple errors affecting two different connections are not possible.

The algorithm we developed is described in Figure 4.1, where the placement and routing steps are shown in a C-like pseudo-code. Our proposed RoRA Placement algorithm performs a robust placement, which implements the TMR principle, executing four distinct functions:

1. The `generate_functions_replicas()` firstly reads the design description produced after the technology mapping and identifies the logic functions in the design. Secondly, it generates three replicas of the logic functions belonging to the original design. Let F be the set of

L. Sterpone, *Electronics System Design Techniques for Safety Critical Applications*,

the original design's logic functions: at the end of this step the three sets F_1, F_2 and F_3 are produced.
2. The `generate_majority_voter()` analyzes the three logic function sets F_1, F_2 and F_3, and generates a logic functions set F_4 that performs the majority voting between them.
3. The `generate_partitions()` partitions the routing graph's vertices in four non-overlapping sets, where each set S_i (i = 1, 2, 3, 4) has enough logic vertices to contain the logic functions of each set F_i (i=1, 2, 3, 4).
4. Every logic function in set F_i is placed heuristically to the logic vertices in set S_i, where i = 1, 2, 3, 4. This phase takes care of marking the graph, by assigning each logic function to exactly one logic vertex in our routing graph.

The RoRA placement algorithm places each logic functions in F_i to the graph vertices belonging to S_i, as well as the majority voter on S_4. After the placement process, each set S_i contains exclusively the function of set F_i. This solution allows us to guarantee that single or multiple effects within one set S_i only do not provoke any misbehavior of the circuit. Indeed, accordingly to our placement, only multiple effects on the boundary of two different sets $S_i \neq S_j$ may generate multiple errors that affect two different replicas.

When all the logic functions are placed to the correspondent set of logic vertex, RoRA performs the routing of the interconnections between the logic vertices. Basically, the RoRA Routing algorithm works on the routing graph we developed, and it routes each connection between two logic vertices through the shortest path it can find. During path selection, the RoRA Routing algorithm labels dynamically the graph's routing vertices, in such a way that it avoids the instantiation of two connections that may be subject to Short effects. Each graph routing vertex (RV) are labeled as *free*, *used* or *forbidden*, with the following meanings:

1. *Free*: the routing vertex is not used by any connection.
2. *Used*: the routing vertex is already used by a connection.
3. *Forbidden*: a routing vertex RV is forbidden if and only if:

   (a) It belongs to set S_i ($RV \in S_i$).
   (b) At least one routing edge, or one wiring edge exists between RV and another vertex RV' belonging to S_j ($RV' \in S_j$), where $i \neq j$.

If RV is added to the circuit and a SEU affects the routing resources in such a way that both RV and RV' are affected, the TMR does no longer work as expected. The Forbidden Vertices Sets (FVSs), which are empty at the beginning of the RoRA routing, contain the vertices marked as forbidden and belonging to the correspondent graph routing vertices set S_i.

RoRA performs the routing of each net by taking into consideration all the graph's vertices labeled as free, and it updates progressively the FVSs adding the vertices marked as forbidden.

As soon as the net is routed, and the marking of the graph has been updated (i.e., the vertices in the routing graph, and the associated edges, have been marked as used by the circuit implementation), the `update()` function is used to modify the set i of forbidden vertices (FVS_i), which is empty at the beginning of RoRA routing.

```
/*Placement*/
generate_functions_replicas (F1, F2, F3)
generate_majority_voter (F4)
generate_partitions (S1, S2, S3, S4)
for each logic function LF ∈ Fi
 place LF on  Si  where i = {1, 2, 3, 4}

/*Routing*/
FVSi = Ø where i = {1, 2, 3}
for each source vertex SV ∈ Fi
{
   for each destination vertex DV of SV
     RT = route (SV, DV)
   update (FVSi,RT)
}
```

Figure 4.1. The flow of the proposed Reliability-Oriented Place and Route Algorithm RoRA.

1. RoRA PLACEMENT ALGORITHM

The developed algorithm starts by reading a description of the circuit which consists of unplaced logic blocks and a set of nets. While standard placement techniques are sufficient if the application mapped on the FPGA does not require any particular reliability constraints, special attention must be taken in FPGA placement algorithm for safety critical application where high reliability is a mandatory requirement.

The RoRA Placement algorithm, which is described in Figure 4.2 as C-like pseudo code, performs the placement of a logic function by using the concept of *window*. A window is defined as a rectangular portion of the logic vertices belonging to the routing graph space. More in details, the RoRA Placement algorithm uses two types of windows: the *place window PW* and the *nearby window W*. The place window PW defines a rectangular space containing the logic vertices already connected to the logic vertex being placed, while the nearby window W defines the space containing a whole of logic vertices labeled as *free* and candidate for the placement.

```
/*Placement*/

place LF on Si
  {
   for each logic vertex DLV ∈ Si and connected to LF
     generate place_window PW
   if PW has at least one free logic vertex
     nearby_window W = PW
   else
     update nearby_window W
   do until LF is placed
     {
      do until cost (local_density (W), global_constraints
(W)) < LIMIT_D_G
        update nearby_window W
      for each free logic vertex V ∈ Si in W
        for each logic vertex DLV ∈ Si and connected to LF
        MDLV = computing Manhattan distance (V, DLV)
        C = Manhattan_cost (MDLV)
        if C < MAX_DISTANCE
          place LF on logic vertex V
          label logic vertex V as used
      }
   }
```

Figure 4.2. The flow of the RoRA Placement algorithm.

The RoRA Placement algorithm implements different heuristic cost functions that measure the wirelength as well as the routability of the placement. The wirelength is based on the *Manhattan* distance that defines the distance between two points measured along axes at right angles that include horizontal and vertical components. Minimizing the wirelength minimizes the number of routing resources required, and thus reduces the existence of SEU sensitive routing resources; thus, the Manhattan distance is minimized. However, the minimization of the Manhattan distance does not guarantee that a signal can be routed successfully, since not all the available routing resources can indeed be used, since some of them must be avoided for satisfying to reliability constraints. To address this problem we added two metric functions: “local density” and “global constraints”, that are defined as follow:

1. The `local_density (W)` computes the number of routing resources available in the nearby window W. It returns the number of available edges that link two routing vertices labeled as *free*.
2. The `global_constraints (W)` computes the routing reliability constraints in the nearby window W. It returns the number of routing reliability constraints that may be generated between the routing vertices labeled as *free* and comprised in the nearby window W.

The local density addresses the degree of routability of the placement. It attaches a cost to the placement considering the capability of routing resources. Thus, it aims at avoiding any competition among signals for insufficient routing resources. The global constraints address the inadequacies of the routability computing the congestion provoked by the routing reliability constraints. These metrics consist of looking at the region contained in the nearby window W and to compute a cost calculating the number of net and routing reliability constraints that may exist in this region.

For a given placement phase the generated nearby window W in the routing graph is examined. This phase allows the RoRA routing algorithm to find easily a route for every signal, since the routing capability of the considered nearby window W where the signals have to be routed is computed during the placement phase.

The RoRA placement of a logic function LF on a partition set S_i is divided in two phases: pre-placement and placement.

During the pre-placement, the window PW is generated considering the logic functions connected to LF that have already been placed on the logic vertices DLVs.

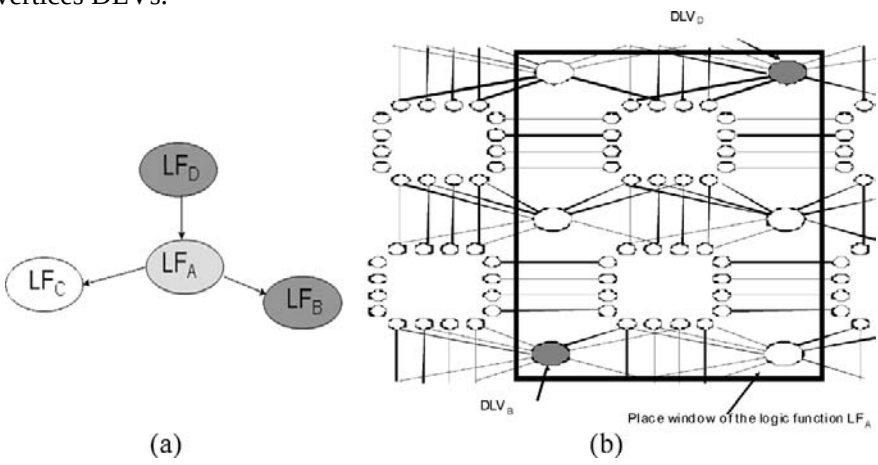

Figure 4.3. Example of Place Window.

In Figure 4.3 it is described an example of the PW generation. Supposing that a logic function LF_A is connected to the logic functions LF_B, LF_C and LF_D, as shown in Figure 4.3a. It is supposed that only LF_B and LF_D have already been placed on the logic vertices DLV_B and DLV_D; during the placement of the logic function LF_A, the place window PW will be generated as described in Figure 4.3b, it selects an area where a logic vertex could be used for the placement of the logic function LF_A. Moreover, W is initialized as equal as PW only if PW contains at least one logic vertex. Otherwise, W

is generated by adding from the same dimension of PW one row or column that contains at least one free logic vertex.

During the placing phase, the RoRA Placement algorithm executes three different steps until the logic function LF is placed on a logic vertex V. Firstly, the RoRA Placement algorithm computes the heuristic cost functions *local density* and *global constraints* on the nearby window W, and compares the respective values with their limits. The limits depend on the cardinality of the adopted routing graph, and thus on the kind of the used FPGA architecture. If the limits are not respected, the nearby window W is updated until the cost function is satisfied.

Secondly, a logic vertex labeled as free is selected from the nearby window W belonging to the partition set S_i. A cost M_{DLV} is associated with every logic vertex DLV that is already placed on the partition S_i and that is connected to the logic function LF. Each cost M_{DLV} is defined calculating the Manhattan distance between each DLV and the logic vertex V candidate for the placement of the logic function LF. Finally, the RoRA Placement algorithm calculates a Manhattan Cost C for the whole DLVs and if C satisfies the max length distance the logic function LF is placed on the candidate logic vertex V.

2. RoRA ROUTING ALGORITHM

The FPGA routing is a complex combinatorial problem. Basically, the RoRA router algorithm works on the routing graph, and routes each connection between two logic vertices through the shortest path it can find. During path selection, RoRA labels dynamically the graph's routing vertices, in such a way that it avoids the instantiation of two connections belonging to two different sets S that may be subject to multiple effects.

The general approach implemented in the RoRA router is a two-phase method composed of a global routing followed by a detailed routing. As shown in Figure 4.4, given a source vertex SV belonging to a logic function F_i, a connection between SV and all its destination vertices DVs is computed executing the global routing followed by the detailed routing. The global routing balances the density of all the routing structures in relation with the reliability constraints, while the detailed routing assigns to the paths specific wiring edges, routing edges and routing vertices.

The global routing is based on a Super-Routing graph architecture which is composed of logic vertices and super routing vertices (SRV) that are linked by a super edge (SE) as shown in Figure 4.5, where a super routing vertex models the whole of routing vertices of the FPGA routing graph,

while a super edge models the whole of routing edges between routing vertices or between a routing and a logic vertex.

```
Route (SV, DV)
  {
   /*Global_routing*/
   do until (SV, DV) is routed
      P = find global_route SV to DV
      L = computing length on route P
      F = computing forbidden node on route P
      if L,F are verified
        /*Detailed_routing*/
        RT = create_routing_tree (SV, DV)
          if (SV, DV) is routed
            return RT
  }
```

Figure 4.4. The flow of the RoRA global and detailed router algorithm.

The Super-Routing graph is used to execute the global routing. The global routing on the Super-Routing graph architecture is performed by the function `find global_route SV to DV`. This function generates a global route P that consists of a sequence of super edges and super routing vertices that link the source logic vertex SV to the destination logic vertex DV. Associating the Super-Routing graph architecture with the FPGA routing graph, a global route P is decomposed to a sequence of routing vertices, wiring and routing edges that connect SV to DV. Thus, the RoRA Global Routing generates a set of candidate paths that could be chosen by the RoRA Detailed Routing to connect SV to DV.

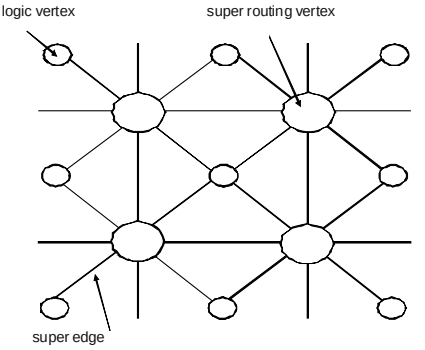

Figure 4.5. The super-routing graph architecture.

To determine whether a global route P is optimal, the RoRA Global Routing selects the super edges and the super routing vertices optimizing an heuristic cost function that consists of two components: the first component

aims at minimizing the length of the global route by selecting the shortest way to connect the source to the sink, while the second component computes the availability of the global route calculating the number of vertices labeled as *forbidden* out of the number of vertices labeled as *free***,** existing in it. The availability A_f of a global route P composed of *i* super routing vertices SRV is defined as:

$$A_f(P) = \sum_i \frac{free(SRV_i)}{avoid(SRV_i)}$$

where *avoid* (SRV_i) is the number of routing vertices labeled as *forbidden* belonging to the Super Routing vertex SRV_i and *free*(SRV_i) is the number of routing vertices labeled as *free*. The global router makes the routing problem easier, since it can estimate the routing congestion due to the routed interconnection and the forbidden vertices. When a global route P is selected, the RoRA Routing algorithm executes the detailed routing.

The RoRA detailed routing algorithm is split in two phases. In the first phase it expands each *routing tree*, where the root is associated to the logic vertex correspondent to the source of the connection, while the leaves are associated to the logic vertices correspondent to the destinations of the connection. The routing tree expansion is made by choosing wiring and routing edges linked by routing vertices labeled as free in our routing graph and belonging to the Global route selected by the RoRA Global routing. The RoRA detailed routing is based on the approach developed for the Pathfinder negotiated congestion algorithm [1, 2]. It is based on the construction of a routing tree. The maze routing, described in [3], is usually used for this purpose. The RoRA detailed router expands the routing tree progressively to the leaves and preserving the routing channel by the global routing: starting from a tree composed of the source vertex, only, new vertices are added, until all the destinations of the connection have been added to the tree. The previously executed global routing allows preserving memory and running time for the routing tree expansion, since the detailed router may choose the net paths on a limited space of solutions. The RoRA detailed router uses the routing tree construction developed for the maze routing approach with a fundamental difference in the creation of each routing tree: the key step of the RoRA detailed router is performed during the routing tree expansion, where those vertices that are labeled as forbidden are not used. Moreover, the set of forbidden vertices is updated in the second phase of the RoRA detailed router after the creation of the routing tree.

The detailed routing generates the routing tree computing the function `create_routing_tree()`. This function performs the computation of the routing tree by taking into consideration all the graph's vertices not labeled as forbidden and belonging to the global route P selected. After the

expansion each routing tree (SV, DVs) may contain a number of routing vertices that could have a routing edge that links them to other routing vertices in the routing graph model by the modification of a single configuration memory bit. The *update* function of the RoRA algorithm selects these routing vertices belonging to the set S_i, and checks if each of them could be linked, by changing a single configuration memory bit, to the routing tree routed on the routing graph belonging to the set S_j, where $i \neq j$. If this happens, the *update* function labels it as forbidden. By this way, no routing edge could link routing vertices belonging to a different set S, and thus no SEU affecting the configuration memory of the SRAM-based FPGA could affect more than one replica of the implemented TMR architecture.

3. EXPERIMENTAL ANALYSIS

In this section a series of experiments are performed to evaluate the effectiveness of the RoRA algorithm. For this purpose, a prototype of the RoRA algorithm is developed, which accounts to about 8K lines of ANSI C code. The developed RoRA prototype has been used for hardening four circuits mapped on a Xilinx Spartan II device.

To evaluate the robustness of the circuits obtained through RoRA against transient faults affecting the FPGA's configuration memory, and in particular against faults affecting the routing resource, the fault injection environment presented in [4] has been used.

The device used in the experiments is a Xilinx Spartan® XC2S30PQ144, whose configuration memory is composed of 336,768 bits organized in 1,165 frames of 288 bits each. The configuration memory controls 132 I/O blocks and an array of 12 x 18 slices [5].

Three purely combinational case studies have been considered: an adder with two 8-bit wide operands, an adder working on two 16-bit wide operands, and a multiplier with two 8-bit wide operands. An elliptic filter has been also considered in order to evaluate the sensitiveness to SEUs in the configuration memory of SRAM-based FPGAs implementing a sequential circuit. Besides, in order to evaluate the capability of RoRA on a real design we mapped an IP-core that implement the Control Area Network (CAN) that uses about 98% resources of a Spartan II XC2S200 [5].

In order to evaluate the effectiveness of the developed algorithm, the five circuits have been mapped using RoRA, as well as the TMR approach (i.e., each circuit is implemented by using three identical modules performing the same task and a majority voter). In the latter case, TMR circuits are placed and routed by standard tools, which do not pose any emphasis in enforcing dependability-oriented place and route rules.

The characteristics of the adopted circuits are reported in Table 4.1, where we report the number of FPGA slices that the circuits occupy (column Area), as well as their maximum working frequencies (column Speed), for the plain, the TMR, and the RoRA versions. It is interesting to observe that, for the considered benchmarks, RoRA does not introduce any area overhead with respect to the traditional TMR solution (which is about three times larger than the plain circuit), and in some cases it is even less resource demanding. Conversely, when placed and routed through RoRA, the circuits become 22% slower on the average than their TMR versions. This effect is the result of the dependability-oriented routing algorithm that RoRA implements: the shortest path is not always selected as the best solution, since it may not be acceptable from the dependability point of view.

TABLE 4.1 Characteristics of the adopted circuits

Circuit	Plain version		TMR version		RoRA version	
	Speed [Mhz]	Area [# slices]	Speed [Mhz]	Area [# slices]	Speed [Mhz]	Area [# slices]
Add8	105	26	86	100	64	96
Add16	105	28	85	103	62	105
Mul8	105	41	64	127	54	125
Filter	104	46	65	132	58	138
CAN	225	384	189	1,152	142	1,152

In order to measure the hardness of the obtained circuits, 15,000 randomly selected SEUs have been injected in the FPGA's configuration memory bits. These bits are selected among those configuration memory bits that define the designs we implemented.

Please note that they may be both programmed or not since both of them may be critical for the mapped design. The number of injected faults was selected to guarantee that the gathered results are statistically meaningful. For these purpose, the experiments have been repeated with 150,000 randomly selected SEUs. Negligible modifications have been observed with respect to the results already gathered with 15,000 faults. Considering that the used voters are not fault tolerant, no faults have been injected in the portion of the configuration memory that implements it. The results obtained are reported in Table 4.2, where *Injected Faults* is the number of injected SEUs, as well as *Wrong Answer* is the number of SEUs for which the faulty circuit produces outputs that differ from the fault-free one. In order to show the contribution of the different FPGA's resources, it has been reported the number of injected faults, and the observed wrong answers, for the FPGA's CLBs and Routing resources. During the experiments a workload of 100,000 randomly generated input stimuli has been applied.

TABLE 4.2 Fault injection results concerning

Circuit	Injected faults [#]		Wrong answer [#]					
			Plain		TMR		RoRA	
	CLB	Routing	CLB	Routing	CLB	Routing	CLB	Routing
Add8	2,558	12,442	2,550	12,037	97	1,255	29	1
Add16	2,410	12,590	2,408	12,190	83	1,609	37	4
Mul8	2,440	12,560	2,390	12,213	91	1,886	20	3
Filter	2,427	12,573	2,398	12,244	86	1,895	39	5
CAN	2,550	12,450	2,545	12,404	71	2,005	38	8

From the gathered results, it is possible observe that most of the injected faults provoke erroneous behaviors in the plain, un-hardened circuits. Moreover, even when the TMR architecture is adopted, a significant percentage of the injected faults still produce a wrong answer. The faults escaping the TMR have been analyzed carefully, and the results is that most of them correspond to multiple errors crossing the TMR domains. In particular the majority of faults escaping TMR are due to SEUs in the routing resources. A very limited number of faults escaping TMR do not fall in the scenario outlined by the dependable rules: these are faults that affect FPGA's resources that do not depend on the implemented circuit, and whose usage is independent from the place and route algorithm. For this very specific device-dependent type of faults different hardening strategies must be envisioned, possibly coming from the FPGA vendor.

From the results achieved it is possible to observe that RoRA drastically reduces the number of SEUs producing a Wrong Answer. In particular, as far as routing resources are concerned, RoRA is able to reduce the number of faults producing wrong answer by 3 orders of magnitude, while reductions by a factor of 2 were observed for fault affecting CLBs. The number of routing faults is reduced effectively thanks to the ability of RoRA of generate a reliability-aware routing topology able to avoid the propagation of multiple errors through the different circuit domains output. Although very effective, RoRA still produces circuits where few SEUs escape and provoke circuits misbehaviors. As the reader can notice, the numbers of faults within the CLBs are not widely reduced as the routing ones. This is due to critical faults that cannot be masked only through the usage of a reliability-oriented place and route algorithm, since they produce errors that exclusively influence those FPGA parts (such as the delivery of power or reset signals to CLB or routing resources) that can be hardened only by the usage of information provided by the vendor.

In the executed experiments, the performance of RoRA in terms of required FPGA routing resources have been measured. The results in Table 4.3, where *PIPs TMR* and *PIPs RoRA* report the number of PIPs in the circuits obtained by the TMR approach and those obtained by RoRA. Please

note that RoRA uses a higher number of PIPs for each circuit: the existence of forbidden graph's routing vertices (i.e., PIPs) forces RoRA to produce connections that are longer than those obtained in the TMR circuits, where all the PIPs are available to the router tool. However, the overhead in terms of PIPs is rewarded by a much higher degree of fault tolerance. The computation times needed by RoRA to perform the place and route process are reported in Table 4.4. The machine used for running RoRA was a SunUltra 250 equipped with 2 Gbytes of RAM, and running at 400 MHz. As a reference. In the Table 4.4 has been also reported the time needed by a commercial tool (Xilinx PAR) for placing and routing the considered circuits. As the reader can observe, the time needed by RoRA is higher than that of the commercial tool; however, the increased time for running the place and route process is rewarded by a much higher fault-tolerance capability.

TABLE 4.3 Summary of routing resources needed by TMR and RoRA circuits' implementations

Circuit	PIPs TMR [#]	PIPs RoRA [#]
Add8	3,864	4,194
Add16	4,874	5,390
Mul8	7,175	9,919
Filter	7,293	9,941
CAN	11,451	14,304

TABLE 4.4 CPU time needed by RoRA and Xilinx PAR

Circuit	CPU Time [min]	
	RoRA	Xilinx PAR
Add8	1.75	0.21
Add16	9.27	0.25
Mul8	21.87	0.34
Filter	25.22	0.40
CAN	83.01	2.86

REFERENCES

[1] V. Betz, J. Rose, *Directional Bias and Non-Uniformity in FPGA Global Routing Architectures*, ICCAD, 1996, pp. 652–659.

[2] C. Ebeling, L. McMurchie, S. A. Hauck, S. Burns, *Placement and Routing Tools for the Triptych FPGA*, IEEE Transaction on VLSI, Dec. 1995, pp. 473–482.

[3] C. Y. Lee, *An Algorithm for Path Connections and Its Application*, IRE Transaction on Electronic Computers, Vol. 10, No. 3, Sept. 1961, pp. 346–365.

[4] M. Ceschia, M. Violante, M. Sonza Reorda, A. Paccagnella, P. Bernardi, M. Rebaudengo, D. Bortolato, M. Bellato, P. Zambolin, A. Candelori, *Identification and Classification of Single-Event Upsets in the Configuration Memory of SRAM-Based FPGAs*, IEEE Transaction on Nuclear Science, Dec. 2003, Vol. 50, No. 6, pp. 2088–2094.

[5] *Spartan-II 2.5 V FPGA Family: Introduction and Ordering Information*, Xilinx Product Specification Datasheets, 2003.

Chapter 5

A NOVEL DESIGN FLOW FOR FAULT TOLERANCE SRAM-BASED FPGA SYSTEMS

Integrated synthesis design flow and performance otpimization

SRAM-based Field Programmable Gate Arrays (FPGAs) are programmable devices used for different applications, such as signal processing, prototyping and networking. They have fixed number of wires, switches and look-up tables (LUTs): all these components can be programmed by downloading a configuration memory with a proper bitstream, giving an FPGA the capability to implement nearly any kind of digital circuit on the same chip.

As widely illustrated in the previous chapters, the content of the configuration memory is vital for the correct operations of the circuit the FPGA implements. The circuit is indeed totally controlled by the FPGA's configuration memory, which is composed of static RAM cells. When energetic particles hit the surface of the SRAM-based FPGA, they can alter the bits composing the configuration memory, and therefore the circuit the FPGA implements may change its original behavior.

The problem of making SRAM-based FPGAs resilient to SEUs has been attacked in two ways. An earlier solution consisted in developing radiation-hardened FPGAs by resorting to special manufacturing technologies, as well as suitable SEU-immune architectures. Although effective, this solution is very expensive and therefore it can be exploited only in those applications where cost is not a primary concern (e.g., military applications). The solution that is currently under investigation by many researchers consists in adopting fault-tolerant architectures to implement hardened circuits while using commercial-off-the-shelf FPGA devices. This solution is very attractive since it is potentially able to combine the needed dependability level, offered by fault-tolerant architectures, with the low cost of commodity devices.

L. Sterpone, *Electronics System Design Techniques for Safety Critical Applications*,

As far as fault-removal techniques are considered, several solutions have been investigated in the past years. The one known as *Scrubbing* consists in periodically reloading the content of the whole configuration memory [1] with the correct bitstream. To minimize the number of needed reconfigurations, which limit the FPGA's availability, a more complex solution uses the *Readback* and the *Partial Reconfiguration* processes. Through the Readback, the content of the FPGA's configuration memory is read and compared with the expected one, which is stored in a dedicated memory located outside the FPGA. If a mismatch is found, the correct bitstream is downloaded in the FPGA's configuration memory. During re-configuration, only the faulty portion of the configuration memory is rewritten [1], thus reducing the re-configuration time.

Several architectures were also proposed, which are all based on introducing hardware redundancy in the circuit the FPGA implements. Among the available architectures, Triple Module Redundancy (TMR) is that attracted most of the attention of researchers. TMR can be implemented easily by using three identical logic blocks performing the same task while a majority voter compares their outputs and decides the correct one.

Although TMR is effective in protecting against SEUs the information the circuits elaborate, it showed some pitfalls when the effects of SEUs in the FPGA's configuration memory are considered. Through detailed analyses of FPGA resources [1], and extensive fault-injection experiments [2], it has been observed that one SEU affecting the FPGA's configuration memory, and in particular those portion of the configuration memory controlling routing resources, may originate multiple errors. This phenomenon depends on many factors: the architecture of the adopted FPGA family, the organization of configuration memory bit, the application that is mapped on the FPGA device, and the memory bit affected by the SEU. In our investigations we considered several test circuits designed according to the TMR architecture, and we observed that about 10% of the faults that may affect the configuration memory produce multiple errors that the TMR is not able to mask [2]. As shown in [3], a clever selection of the TMR architecture helps in reducing the number of escaped faults, but it is still unable to reduce them to zero [4]. To cope effectively with SEUs in the FPGA configuration memory, we presented in [5] an approach that makes use of TMR and of a dependability-oriented place and route algorithm, RoRA, to implement cleverly a circuit on SRAM-based FPGAs in such a way that the effects of SEUs are minimized. The approach is very effective in hardening FPGA-based circuits, but it may require high computational times, due to the complexity of executing the dependability-oriented place and route operations for the whole circuit.

In this section is presented a new flow that makes use of the results achieved in [5] to design complex circuits that are insensitive to SEU and that has optimized characteristics in terms of running frequency.

The design flow is based on standard tools for design entry, synthesis and design implementation.

1. THE DESIGN FLOW

The developed design flow adopts standard tools provided by FPGA or EDA vendors for performing the typical tasks needed for transforming a specification coded in HDL, or provided through schematic entry, into a bitstream suitable for being downloaded into the FPGA's configuration memory. Ad-hoc developed tools are used in combination to standard tools for guaranteeing that the obtained bitstream is resilient to SEUs: they are used to rework only those portions of the circuit that are particularly sensitive to SEUs, i.e., that modify the circuit behavior when affected by SEUs. The rationale behind this approach is that standard tools are very effective in performing synthesis, placement and routing, and produce high quality designs (in terms of clock frequency, area occupation, or power consumption). It is therefore worthwhile to use them to carry out most of the tasks involved in the design process. Moreover, only a limited subset of the whole circuit has to be addressed to cope with SEU-induced problems. For this limited subset it is worthwhile resorting to ad-hoc developed tools that provide robust circuits, although their computational cost is usually higher than that of standard tool. By exploiting standard tools for carrying out most of the design task, while resorting to computational-expensive ad-hoc tools only for a limited subset of the whole circuit, it is possible to minimize the design time, reduce the performance penalty overhead, while meeting high dependability levels.

The developed design flow is illustrated in Figure 5.1, and it is composed of three main modules:

1. *Xilinx ISE* is the collection of Xilinx's design tools that is normally used by designers for obtaining a circuit implemented by any Xilinx FPGA. The collection comprises tools for design entry, synthesis, and place and route. Please note that, although Xilinx's synthesis tool is used, other solutions can be adopted for this purpose (like for example Simplicity's Symplify, or Synopsys's FPGA compiler).
2. *STAR tools* is a collection of tools developed for analyzing the placed and routed circuit, to check whether critical area exist that may corrupt the correct operations of the TMR architecture when affected by SEUs.

3. *V-Place and RoRA router*: It performs the implementation phase of a design. The design flow is divided, as it refers to the standard FPGA implementation flow, in two sub-problems: the placement and the routing. The Placement algorithm (V-Place) is able to map fault tolerant circuits according to the TMR design techniques on SRAM-based FPGAs while optimizing the circuit's frequency. The routing algorithm (RoRA) is a reliability-oriented tool that modifies the critical circuit areas identified by the STAR tool, and that produces a new version of the circuit where all the criticalities have been resolved.

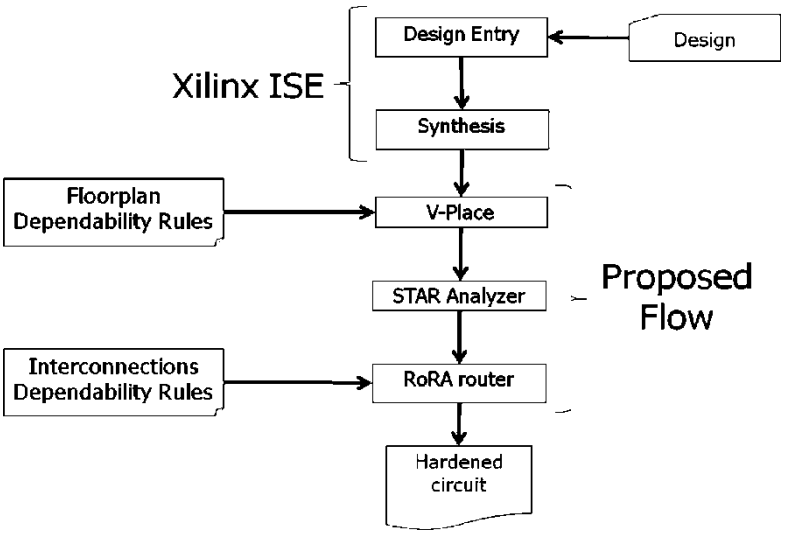

Figure 5.1. The proposed integrated design flow.

1.1 STAR analyzer

The main purpose of the STAR tool, as described in the Chapter 3, which is to analyze the SEU effects in SRAM-based FPGAs early in the design phase, in particular, as soon as the placed and routed model of the designed circuit is available.

The tool is based on the description of the circuit and on a data base coding the rules.

The circuit description is divided in two files:

1. *Circuit* DB. It contains the structural description of the circuit, which consists of logic functions (either combinational or sequential) and connections between them implemented through FPGA's resources.

2. *Floorplan* DB. It contains the description of where each circuit's resource is placed on the FPGA. The FPGA's floorplan is divided in four sets S_1, S_2, S_3, and S_4, and each circuit's resource is placed into one and only one of these sets.

The *Analyzer* checks if each resource complies with the data base of rules and, if a violation is found, it produces a report that shows the resources and the configuration memory bits that violated the rules when affected by SEUs. The produced report serves as input to the RoRA router that produces an equivalent circuit where all the violations are resolved.

1.2. RoRA router

The RoRA router, which was introduced in [5] to solve the problem of the Short effect, is based on the approach developed for the Pathfinder negotiated congestion algorithm [6, 7]. Basically, the RoRA router works on the graph model we developed, and it routes each connection between two logic vertices through the shortest path it can find. The path is composed of routing vertices, routing-to-routing edges, and logic-to-routing edges. During path selection, RoRA labels dynamically the graph's routing vertices, in such a way that it avoids the instantiation of two connections that may be subject to Short effects.

2. PERFORMANCE OPTIMIZATION OF FAULT TOLERANT CIRCUITS

In the past years, several fault tolerance methods have been proposed in order to mitigate the effects of SEs in the configuration memory of SRAM-based FPGAs. On one side a possible solution to this problem is to use radiation-hardened FPGAs, however these devices are much more expensive than Commercial-Off-The-Shelf (COTS) FPGAs. Vice versa, the viable solutions are represented by two methodologies: the reconfiguration-based techniques and the redundancy-based approaches.

The reconfiguration-based methods, aiming at restoring as soon as possible the proper values into the configuration bits after an SE happened [1], are a viable solution to detect and remove the upset within the configuration memory. However, this approach does not offer a complete immunity to the SE's effects, thus masking techniques are needed, such as redundancy-based ones that avoid the SE's effects propagation to the circuit's outputs [8–10]. These techniques are deployed through Triple Modular Redundancy (TMR), where three identical replicas of the same circuit work in parallel while the outputs are produced by comparing and majority voting their

signals. TMR is a mandatory hardening technique for SRAM-based FPGAs since memory elements, routing resources and logic resources are all sensitive to SEs and thus redundancy must be adopted for all of them. Although TMR techniques are drastically reducing the effects of SEs within the configuration memory of SRAM-based FPGAs, recent works demonstrated that some criticalities are still not protected from these techniques [11] and reliability-oriented place and route algorithms (RoRA) are needed to physically map the circuit on the FPGA's resources in order to guarantee complete robustness against the SE's effects within the FPGA's configuration memory [12]. Although effective, this technique introduces high degradation in terms of operational frequency to the implemented circuits, since they become 40–50% slower, on the average, with respect to their original versions [12]. The operational frequency reduction nullify the high performance offered by these devices, it is therefore necessary the development of techniques able to guarantee the circuit's fault tolerance without reducing its speed.

The proposed V-Place algorithm is based on a model-based topology heuristic that address the arithmetic modules implemented on the FPGA. The delay of the interconnection between these resources is reduced thus minimizing the critical paths of the circuit physically mapped on the FPGA architecture. The main novelty of the proposed algorithm lies in the technique used to address the physical placement of the resources that, differently from other investigated approaches, does not rely only on the netlist model of the implemented circuit but directly on the topology organization of the circuit elements physically interconnected on the SRAM-based FPGA device.

2.1 The congestion graph

The design complexity leads to an increasing routing congestion of a design implemented on SRAM-based FPGAs. The routing congestion may provoke several problems: it may degrade the performance of the design or it may add more uncertainty on the design closure process. The global increase of the delay is generally related to the unexpected increase of the delay of a single net due to routing congestion. In particular, congestion can affect design performance in two circumstances:

- A path may be detoured due to the presence of congested regions.
- A path may include several numbers of vias (interconnection points) if the router finds the shortest path through a congested region.

The resulting increase in the delays of the critical nets can cause timing violations on the paths through those nets. The routing algorithms generally

detour that nets increasing their delay. Furthermore, when reliability-oriented rules are considered [12] the routing-congestion dramatically increases due to the constraints inserted by the reliability rules that force the routing into specific path in order to prevent multiple errors in the implemented circuit. In order to take care of the congestion during the placement step, a congestion graph is introduced.

The congestion graph used is illustrated in Figure 5.2. The array area of the FPGA is modeled as a matrix of bin-squares where each one models the resources of an FPGA Configurable Logic Block (CLB). Every bin-square is characterized by a weight and a set of arcs. The weight *P* indicates the number of logic elements placed in the considered CLB. The arcs model the number of nets between two logic elements. Each arc coefficient is the ratio between the number of used routing resources and the number of available nets on horizontal plan (HL and HR), diagonal plans (DLT, DLB and DRT, DRB) and vertical plan (VT and VB).

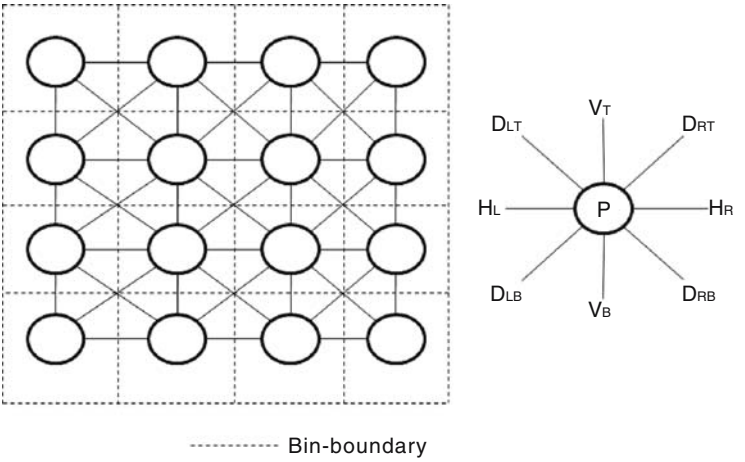

Figure 5.2. The congestion-graph and the vertex definition.

2.2 The voter architectures and arithmetic modules

TMR hardening techniques involve the usage of voting architectures. Two voting structures define a *voter partition logic*, the set of interconnection and logic resources (both combinational and sequential) located between them. Considering the TMR scenario described in Figure 3.2, a voter partition logic consists in the logic domains D_j with $j \in \{1,2,3\}$ comprises between voter structures V_i and V_{i+1}.

If on one side, the TMR voter partition structure increases the fault tolerance capability of a TMR architecture since introduces voter barriers, on the other side it is particularly critical for the placement task, since each voter partition introduces congestion on the routing interconnections. The developed algorithm treats the voter architecture as macros in order to prevent routing congestion.

Modern SRAM-based FPGA devices support the design of embedded arithmetic cores for general purpose operations such as high parallelism multiplications or floating-point units. These units are synthesized accurately in order to optimize their operational speed and computational precision. Unfortunately, when these units are placed and routed on the FPGA physical layout, they may lost their optimized characteristics due to the logic and routing congestion. In particular, when TMR hardening techniques are adopted, the topological characteristics of the programmable interconnections may drastically degrade the delay of each single routing path thus decreasing its computational speed.

The developed placement algorithm is able to address the routing and logic delay of the arithmetic units following a set of *arithmetic rules* and by modifying their placement positions within the FPGA's logic programmable array.

The arithmetic rules consist of a set of physical macros that are implemented by the placement algorithm with respect to the circuit's functionalities.

2.3 The V-Place algorithm

The developed placement algorithm (V-Place) is based on the implementation of a graph embedding a metric that contains information about the FPGA's regular physical architecture. The proposed algorithm directly considers the routing delays on the basis of a Manhattan distance heuristic. The algorithm is based on the routing graph presented in the previous section. The routing graph embeds interconnection's delay measured in terms of number of traversed routing switches. The flow of the proposed algorithm is divided in three phases. In the first phase V-Place computes an analytical distance metric of the total interconnection length for each input and output signals of a given logic resource. Secondly it constructs a metric space for computing the FPGA performances. The third phase consists in the optimization of the location of each logic resource. The optimization is performed taking care of three different placement organizations: voting structures, arithmetic cores and general purpose logic. The flow of the proposed algorithm is illustrated in Figure 5.3.

```
/*Phase 1*/
read_native_circuit_description()
generate_routing_graph()
L = reading_logic_vertex()
V = reading_logic_voter()
D = reading_TMR_domains()
for each logic vertex i ∈ L
  MD_i = create_manhattan_distance (i)
/*Phase 2*/
M_space = create_performance_metric (MD)
A_macros_j = read_arithmetic_rules (L)
Macro_SET_j = ø
/*Phase 3*/
do until ΔE(M_space) < min_delay
{
   v_place_optimizer (M_space ,D, MD_i, L, V, A_macros_j)
   {
   /*Place Macros*/
   mark_estimated_voter_space(V)
   if Macro_SET_j = ø
     do until ΔE(M_space) < min_delay
       for each L ∈ A_macros_j with max(MD_i)
         RR = available_reliability_rules(D)
           FP = find_free_location()
    place(L,FP)
          update(MD_i)
          update(Macro_SET_j)
   else
  move_macro (Macro_SET_j)
   /*Place Logic*/
   for each L not included in A_macros_j
       RR = available_reliability_rules(D)
         FP = find_free_location()
  place(L,FP)
     update(MD_i)
   }
}
```

Figure 5.3. The flow of the new V-Place algorithm.

3. EXPERIMENTAL RESULTS

The purpose of this section is to evaluate the effectiveness of the proposed design flow in designing circuits that are hardened against SEU effects and to estimate the performance costs. In particular, we focused on the effects of SEUs in the FPGA's configuration memory since it is the most critical aspect. The number of bit devoted to the configuration memory is indeed much higher than that devoted to the user memory (for implementing registers, or memory blocks). As a result, an SEU is more likely to happen within the configuration memory than in the user memory.

Three experiments were performed. The first one aimed at estimating the performance in terms of timing analysis of the placement algorithm. The second one consisted in designing three simple circuits according to the approach presented in [5], and according to the design flow presented in this paper. By comparing the attained results we can quantify the improvements that our new design flow allows with respect to the original solution we presented in [5]. The second experiment consisted in designing a realistic circuit with the intent of analyzing the viability of our design flow in attacking the design of a hardened real-life design.

3.1 Timing analysis

In order to estimate the improvements and the effectiveness of the proposed algorithm, two parameters of the circuit placed with the V-place algorithm have been evaluated: the SEs sensitivity and the speed. The improvements of the latter are shown by timing analysis reports while static analysis, by means of the approach developed in [13], proofs that the proposed algorithm does not affect the fault tolerance of the circuit itself.

Three real-case designs have been used to perform the experiments: a CORDIC processor core, usually exploited for real-time calculations of trigonometric functions and vector magnitude, a 24 × 24 parallel multiplier, and an 8051 Intel microcontroller core. The circuits have been hardened using the Xilinx XTMR tool [14] to provide a full tolerance against single SEs. The static analysis experiments have been run on a Xilinx Virtex-II XC2V1000-FG456 device which is characterized by 10,240 available Look-Up Tables (LUTs), 10,240 Flip-Flops (FFs) and 324 Input-Output Blocks (IOB) and whose configuration memory is composed of 4,082,592 bits. Table 5.1 summarizes the circuits' characteristics for this specific device, in terms of occupied LUTs, FFs and IOBs.

TABLE 5.1 Characteristics of the adopted circuits

Circuit	LUTs [#] (%)	FFs [#] (%)	IOBs [#] (%)
CORDIC core	6,258 (61)	2,478 (24)	303 (93)
Parallel Multiplier	3,597 (35)	0 (0)	288 (88)
8051	7,210 (70)	3,672 (35)	108 (33)

The performance of the placed circuits are evaluated using vendor's tool in order to estimate the delay of the critical paths within the circuit. The results we obtained are reported in Table 5.2 where we show the delay of the critical path of each hardened circuit (TMR and Proposed Flow) with respect to the unhardened version (Plain). The developed design flow is able to optimize the delay of the maximum critical path up to the 44% with respect

to the TMR version, thus increasing the operational frequency of the implemented circuit. As the reader can notice, the delay introduced by the developed flow with respect to the plain circuits is less than the 4% in the worst case.

Table 5.2 Timing analysis comparison

Circuit	Plain [ns]	TMR [ns]	Proposed flow [ns]
CORDIC core	5.230	9.702	5.404
Parallel multiplier	6.993	7.599	7.245
8051	5.840	8.932	6.020

Furthermore, two kinds of static analysis are executed in order to evaluate the effects both of the single upsets and the SEs accumulation.

A first experiment has been performed using the Xilinx ISE-generated version of the three circuits. We analyzed them with the Static analyzer. The results show that the X-TMR tool successfully hardened the circuits against single SEs; indeed no failures have been detected. Then we performed the same experiment on the circuits replaced with the proposed approach and the static analyzer shows that no failures have been produced. We can thus conclude that the proposed algorithm does not threaten the circuit's fault tolerance.

A second analysis has been performed in order to evaluate the SEs accumulation effects. This analysis consisted in estimating the average number of accumulated SEs, called $N_{\text{estimated}}$, within the configuration memory by running several analysis with the Static Analyzer considering the accumulation of SEs. $N_{\text{estimated}}$ is updated at the end of each generated distribution. The execution of the analysis is terminated once $N_{\text{estimated}}$ meets an estimation error β for a number of distribution γ defined by the user before the analysis. We evaluated the mean number of accumulated SEs by means of the static analysis algorithm fixing a tolerated precision error $\beta = 1\%$ for the number of iterations $\gamma = 100$.

Table 5.3 summarizes the results of this second analysis, in terms of mean number of accumulated SEs before a criticality is reported by the Static Analyzer.

The results show that different placement can modify the mean number of SEs need to produce a failure on the circuit's outputs; in particular for the adopted congestion-oriented placement we observed a reduction of the accumulated SEs. This means that it is possible to improve the fault tolerance of a circuit simply modifying its placement.

TABLE 5.3 Static analysis results for SEs accumulation

Circuit	Mean number of accumulated SEs before failure	
	ISE-generated version	Proposed flow
CORDIC core	16	12
Parallel multiplier	31	23
8051	28	13

3.2 Evaluating the proposed design flow

To evaluate the capability of the proposed design flow three purely combinational case studies have been considered: an adder with two 8-bit wide operands, an adder working on two 16-bit wide operands, and a multiplier with two 8-bit wide operands. We designed it according to two approaches: the one presented in this paper, and that presented in [5].

TABLE 5.4 Comparing execution times

Circuit	RoRA [s]	Proposed approach [s]
Add8	105.0	56.6
Add16	556.2	101.0
Mul8	1,312.2	265.4

The device used in the evaluation experiments is a Xilinx Spartan® XC2S30PQ144, whose configuration memory is composed of 336,768 bits organized in 1,165 frames of 288 bits each. The configuration memory controls 132 I/O blocks and an array of 12 × 18 slices.

Fault injection experiments [3] showed that in both cases we obtained hardened circuits with respect to SEUs affecting the configuration memory of the adopted FPGA. Conversely, as results reported in Table 5.4 shows, significant differences in execution times were observed. These results show that demanding the synthesis, and place and route operations, of most of the circuit to standard tools, while relying to reliability-oriented tools for addressing critical areas only, is far more efficient than routing the whole circuit with the approach presented in [5]: the execution times are indeed reduced by a factor ranging from about 2 to a factor of about 6.

3.3 Evaluating a realistic circuit

The realistic circuit that has been taken in consideration is the IP-core implementing the Control Area Network protocol. This IP-core has been developed and validated in [15] and it is compliant with the CAN protocol specifications.

As a first step of this experiment, it is designed the hardened version of the adopted IP-core according to the design flow proposed in this paper. To show the versatility of our approach, we considered several different FPGA families all coming from Xilinx: the Spartan II, the Virtex I, the Spartan 3, and the Virtex II. Similarly to what is normally done during the development of a realistic design, it is selected the smallest device able to hold the design.

Table 5.5 reports the selected FPGAs, the percentage of FPGA's resources that are used by the TMR-version of the CAN controller, as well as the circuit's maximum frequency. Please note that for all the devices considered, the IP-core uses at least the 98% of the total available resources. Although the selected FPGAs are almost full, RoRA was still able to find a different routing for the critical signals. Moreover, we analyzed the circuit's frequency (by exploiting the Xilinx's timing analyzer tool) before and after the execution of RoRA, which reworked the optimal circuit implemented produced by Xilinx's ISE to make it robust against SEUs. For all the considered architectures, we always observed negligible reduction (less than 1%) of the maximum frequency. These results suggest that the proposed design flow can be used effectively to attack the design of realistic circuits, even in those case were few resources are available for reworking the circuit produced by the standard design tools.

TABLE 5.5 Characteristics of the FPGA devices used

FPGA device	Resource occupation [%]	Frequency [MHz]
Spartan II XC2S200	98	225
Virtex I XCV200	98	174
Spartan 3 XC3S200	99	230
Virtex 2 XC2V250	100	180

TABLE 5.6 Comparing the execution time

FPGA device	Number of escaped faults		CPU time [min]
	Proposed approach	TMR approach	
Spartan II XC2S200	0	263	83
Virtex I XCV200	0	265	86

In order to quantify the effectiveness of the proposed design flow in protecting FPGA-based systems over a simpler approach where only the TMR architecture is adopted, we designed the selected IP-core according to the TMR, only. In this case, only the standard Xilinx ISE tools are used in combination with the TMR tool we developed, while the other components of the SEU-kit are not exploited (i.e., floorplan constraints, Analyzer and RoRA are not used). This scenario depicts the case of designers willing to adopt the TMR architecture and resorting only to standard tools for circuit design, plus ad-hoc tool for replicating the circuit and adder the needed majority voter.

Table 5.6 reports the number of faults that escape the TMR architecture obtained by using standard tools only, in comparison with the figures attained for the IP-core designed according to the approach presented in this paper (for simplicity, only the figures concerning the Spartan II and Virtex I architectures are reported. Similar figures were observed for the other architectures). As the reader can observe, the approach presented in this paper produced fault- tolerant circuits, while the TMR alone was not able to achieve this goal. Table 5.6 also reports the CPU time for completing the design of the circuit according the approach presented in this paper. This figures indicate that a designer can obtain a validated and hardened design in less than 1 h and a half.

REFERENCES

[1] Xilinx Application Notes XAPP216, *Correcting Single-Event Upset Through Virtex Partial Reconfiguration*, 2000.

[2] M. Bellato, P. Bernardi, D. Bortolato, A. Candelori, M. Cerchia, A. Paccagnella, M. Rebaudengo, M. Sonza Reorda, M. Violante, P. Zambolin, *Evaluating the Effects of SEUs Affecting the Configuration Memory of an SRAM-Based FPGA*, IEEE Design Automation and Test in Europe, 2004, pp. 188–193.

[3] P. Bernardi, M. Sonza Reorda, L. Sterpone, M. Violante, *On the Evaluation of SEUs Sensitiveness in SRAM-Based FPGAs*, IEEE International On-line Testing Symposium, 2004, pp.115–120.

[4] F. Lima Kastensmidt, L. Sterpone, L. Carro, M. Sonza Reorda, *On the Optimal Design of Triple Modular Redundancy Logic for SRAM-Based FPGAs*, IEEE DATE, 2005, pp. 1290–1295.

[5] M. Sonza Reorda, L. Sterpone, M. Violante, *Multiple Errors Produced by Single Upsets in FPGA Configuration Memory: A Possible Solution*, IEEE European Test Symposium, 2005, pp. 136–141.

[6] V. Betz, J. Rose, *Directional Bias and Non-Uniformity in FPGA Global Routing Architectures*, ICCAD, 1996, pp. 652–659.

[7] C. Ebeling, L. McMurchie, S. A. Hauck, S. Burns, *Placement and Routing Tools for the Triptych FPGA*, IEEE Transactions on VLSI, Dec. 1995, pp. 473–482.

[8] S. Habinc Gaisler Research, *Functional Triple Modular Redundancy (FTMR) VHDL Design Methodology for Redundancy in Combinational and Sequential Logic*, www.gaisler.com.

[9] P. K. Samudrala, J. Ramos, S. Katkoori, *Selective Triple Modular Redundancy (STMR) Based Single-Event Upset (SEU) Tolerant Synthesis for FPGAs*, IEEE Transactions on Nuclear Science, Vol. 51, No. 5, Oct. 2004.

[10] C. Carmichael, *Triple Modular Redundancy Design Techniques for Virtex FPGAs*, Xilinx Application Notes XAPP197, 2001.

[11] L. Sterpone, M. Violante, S. Rezgui, *An Analysis Based on Fault Injection of Hardening Techniques for SRAM-Based FPGAs*, IEEE Transactions on Nuclear Science, Vol. 53, No. 4, Part 1, Aug. 2006, pp. 2054–2059.

[12] L. Sterpone, M. Violante, *A New Reliability-Oriented Place and Route Algorithm for SRAM-Based FPGAs*, IEEE Transactions on Computers, Vol. 55, No. 6, June 2006, pp. 732–744.

[13] L. Sterpone, M. Violante, *A New Analytical Approach to Estimate the Effects of SEUs in TMR Architectures Implemented Through SRAM-Based FPGAs*, IEEE Transactions on Nuclear Science, Vol. 52, No. 6, Part 1, Dec. 2005, pp. 2217–2223.

[14] *TMRTool User Guide*, in Xilinx User Guide UG156, 2004.

[15] J. Perez, M. Sonza Reorda, M. Violante, *Accurate Dependability Analysis of CAN-Based Networked Systems*, 16th IEEE Symposium on Integrated Circuits and Systems Design, 2003, pp. 337–342.

PART II

Chapter 6

CONFIGURATION SYSTEM BASED ON INTERNAL FPGA DECOMPRESSION

A new configuration architecture

Nowadays Field Programmable Gate Arrays (FPGAs) are an improved technology for developing high-performance embedded systems. SRAM-based FPGAs offers the possibility of in-the-field reconfiguration that results in the ability to adapt the product to modified user's requirements, to enrich the product's features, or simply to correct bugs. With the advent of multi-million gate FPGAs, the size of the configuration information that defines what circuit the FPGA implements has increased drastically, and thus the amount of external memory needed to keep the configuration data is increasing dramatically. The work presented in this chapter describe a novel configuration compression system that exploits internal configuration mechanism of modern SRAM-based FPGAs and results in high compression efficiency. The proposed system is applicable to any modern SRAM-based FPGA devices having an embedded microprocessor core since the configuration data are processed as raw data. Moreover, the proposed approach does not require any external hardware support and allows high speed dynamic reconfiguration. Experimental results on Xilinx SRAM-based FPGAs platform implementing several real-world circuits demonstrated 82% savings in memory on the average.

1. INTRODUCTION TO THE DECOMPRESSION SYSTEMS

Field Programmable Gate Arrays (FPGAs) are reconfigurable platforms that can implement embedded systems with high processing rates while providing a high degree of flexibility required in dynamically changing environments.

L. Sterpone, *Electronics System Design Techniques for Safety Critical Applications*,

Today's FPGAs are suited for accelerating computing-intensive algorithms that can take advantage of massive hardware parallelism [1]. Moreover, SRAM-based FPGA devices offer the possibility of run-time reconfiguration. Finally, modern SRAM-based FPGA devices embed hardwired microprocessor cores that have drastically increased the computational capability of these devices.

In SRAM-based FPGAs, the content of the configuration memory is reloaded after power-up, therefore it does not store permanent data. As a result, SRAM-based FPGAs require external devices to initialize the configuration memory. A typical SRAM FPGA-based systems includes indeed a non-volatile memory for storing configuration data and the FPGA device itself. Each time the system is powered-up, the configuration data are loaded into the FPGA.

As the number of configurable blocks and the complexity of the routing resources increase, the amount of configuration memory needed to store the configuration data grows accordingly. It is worth noticing that the configuration data of the Xilinx Virtex-II FPGAs ranges from 0.4 Mbits to 43 Mbits [2]. Therefore, storing the configuration data in a FPGA-based system is a critical issue since it needs memory modules that could increase the overall system cost. The size of the configuration memory has a negative impact also on the configuration time, and it can limit the applicability of partial/total dynamic configuration in time-critical applications.

The memory size for storing configuration data may be reduced by exploiting suitable compression algorithms. As will be summarized in the following section, several works proposed techniques which exploit the peculiarity of the considered FPGA family [3–6], and therefore they are of limited applicability. Vice versa, a technique applicable to any SRAM-based FPGA device is presented in [7]. No matter which approaches are considered, all of them require an additional hardware component to be placed between the memory storing the configuration data and the FPGA, which decompresses the compressed configuration data and control the configuration operations. The additional hardware component represents an overhead for the system and it may introduce a not negligible design's cost. In fact, developers must modify their original designs in order to implement these techniques.

Other researchers investigated the implementation of on-chip FPGA decompressor. In [8] a specific decompressor hardware module is implemented using the internal configurable logic available within the FPGA device. Besides, in [9] is presented an approach able to support flexible FPGA-based run-time partial reconfiguration using a microprocessor mapped on the available resources of the FPGA device. Nevertheless these techniques present the advantage of fast run-time reconfiguration for small applications, these solutions present two major drawbacks. The first is that they are

specifically designed for Xilinx's FPGA devices, the second is that they present a low compression ratio for those applications use a large percentage of the FPGA available resources.

A novel configuration compression system is proposed and it is able to reduce the memory requirements for storing configuration data within a FPGA-based embedded system without introducing expensive overhead due to the implementation of the decompression module internally or externally the FPGA device.

Being based on the internal resources that most recent SRAM-based FPGAs offer, the approach does not require any external hardware, since all the operations needed to decompress and configure the FPGA are performed by an on-chip CPU and by an internal configuration mechanism. This does not add any cost to the configuration state-machine, since the configuration operations are performed by the internal configuration mechanism available on FPGA devices. The on-chip CPU is dedicated to run the decompression algorithm only during the configuration process, after that it is available to run the user applications. Thus, the effective area cost that has to be paid is minimal and it does not depend on the dimension of the FPGA device adopted. Moreover, the proposed approach uses a compression algorithm implementing an adaptive binary arithmetic coder working on raw data; therefore, since it is not based on specific configuration data organization, it is applicable to any kind of modern FPGA devices.

The capability of the approach implemented as a case study on the Xilinx Virtex II Pro device is evaluated considering several configuration data for the FPGA corresponding to real-world circuits. From our experiments we observed compaction ratio of 5.5 times on the average. Moreover, we recorded an average time of 0.6 ms for configuring the device, which makes possible the implementation of very efficient dynamically configurable systems.

2. OVERVIEW ON THE PREVIOUSLY DEVELOPED DECOMPRESSION SYSTEMS

Several compression techniques have been proposed for FPGA architectures. In order to reduce the memory requirements of their FPGA configuration data, Xilinx developed a compression algorithm based on a LZ77 scheme [3]. LZ77 is a dictionary-based text compression scheme that works by defining a fixed-size dictionary to hold bytes from an input source. As the compression progresses, the dictionary is updated by loading more bytes from the input source, thus forcing earlier entries out. Although this technique does not introduce any time overhead on the configuration process, it is

applicable only on Xilinx FPGAs and the compression ratios are extremely low.

In [5] dictionary-based techniques were adopted to reduce the time required to transfer configuration data to Xilinx Virtex series FPGAs. A compressed version of the configuration data is fed to the configuration circuitry of the FPGA and the decompression takes place inside the FPGA. Although this techniques reported high compression ratios, the decompression process of the configuration data is very time consuming. Besides, this techniques need the modification of the configuration mechanisms in order to support decompression.

An FPGA configuration data compression approach that take advantages of the characteristics of the configuration mechanism of an FPGA Xilinx XC6200 is presented in [5]. In [6], run-length compression techniques for FPGA configuration data have been presented. The addresses were compressed using run length encoding while data was compressed using LZ compression. However, this approach take advantages of the specific characteristics of the adopted FPGA. A dedicated hardware is required for both the previously referred methods.

A compression technique based on processing of raw configuration data is presented in [7]. This technique is applicable to any SRAM-based FPGAs, the compression algorithm is based on the principles of dictionary-based compression, and it does not depend on specific features of the configuration mechanisms. However, this approach requires an external hardware that executes the decompression process and controls the configuration operations of the SRAM-based FPGAs.

Vendors provide decompression solutions specifically oriented to their FPGA devices. However, these solutions can be grouped on two kinds of alternative configuration: active and passive, where the configuration is performed externally or internally the FPGA respectively. An approach based on active configuration that uses a PROM built-in decompression algorithm is described in [10], this solution can achieve a compression ratio of two times the original configuration data length. Vice versa, a technique based on passive configuration is presented in [11] where decompression is done by the FPGA itself, this solution achieves up 1.9 times the original configuration data length.

A solution based on a specific decompression module implemented on the FPGA chip using the available resources is presented in [8]. This module is inserted a part of the run-time system which controls the decompression and receives data only for performing partial dynamic reconfiguration. This approach is not completely implemented on the FPGA, since the system needs external chip interconnections that link the output pins of the decompression module to the pins of the external FPGA configuration port.

A solution that exploits internal configuration port that modern FPGAs offer is presented in [9]. This solution implements on the FPGA chip resources, both the decompression module and the configuration controller supporting flexible run-time partial reconfiguration. In order to execute the decompression both the approaches exploit the LZ77 algorithm. Nevertheless, the advantages concerning the fast run-time partial reconfiguration these solutions are specifically designed for Xilinx's FPGA devices, furthermore they present a low compression ratio for applications using a large percentage of the FPGA available resources, this is meanly due to the LZ77 compression algorithm the has a low compression ratio for complex configuration data.

In the approach presented, no external hardware is needed in order to perform the decompression and FPGA configuration, since all the operations are performed exploiting the FPGA's internal resources. Moreover, no interconnections are needed outside of the FPGA in order to perform the configuration, since these operations are performed through the internal configuration port the most recent FPGAs embed. From the computational compression algorithm point of view, our approach achieve high compression ratio thanks to an adaptive binary arithmetic coder that works on raw data without considering individual semantics of specific FPGA configuration data organizations. Thanks to this compression algorithm is achieved a savings in memory of 82% on the average for various circuit's configuration data. Moreover, the proposed compression system is applicable to any SRAM-based FPGA and do not require any modification of the external hardware in order to be adopted.

2.1 Generalities of SRAM-based FPGAs

Generally, FPGAs are characterized by an array of configurable logic blocks (CLBs) surrounded by input-output blocks (IOBs). Nowadays, state-of-the-art SRAM-based FPGA devices provide further resources that are scattered among the logic array. These resources comprise block RAM memories (BRAMs), multiple clock resources and hardwired modules. The hardwired modules consist of various kind of DSPs and microprocessor solutions depending on the manufacturer.

All the resources embedded within a SRAM-based FPGA are controlled by an internal configuration memory. The configuration time depends on the size of the configuration data and on its format. Moreover, the clock rate and the operation mode of the configuration mechanism may have an important role for the configuration speed-up, since they determine the rate at which the configuration data are delivered to the FPGA device. The semantic of the configuration data strictly depends on the characteristics of the configuration mechanisms as well as the characteristics of the FPGA architecture. Thus the

configuration data format varies among different vendors. In order to provide a compression system applicable to any FPGA, it is therefore needed an algorithm that does not take in consideration any peculiarity of the configuration data related to the FPGA architecture.

The configuration data is loaded within the FPGA configuration memory through external or internal configuration mechanisms. The external mechanism is based on parallel transfer of 8-bit configuration data words that results in faster configuration time. The configuration data is transferred to the internal configuration memory through an external configuration access port with the support of a specific external circuitry working at higher data rates. Typical transfer frequency can be as high as 60 Mhz [12, 13]. Vice versa, the internal configuration mechanism is based on an internal configuration port that allow FPGA's own logic resources to access the configuration data in such a way that the FPGA can dynamically reconfigure itself without demanding this process to an external hardware.

The internal configuration port is available in may recent devices from several FPGA vendors. For example, Atmel's SRAM-based FPGAs embed an Advanced Virtual RISC (AVR) microcontroller that can write configuration data to the FPGA configuration memory through the internal configuration port. Similarly, the most recent Xilinx FPGAs embed an Internal Configuration Access Port (ICAP) controller able to read and write the content of the configuration memory [14].

3. THE PROPOSED SYSTEM

A typical architecture of an FPGA-based embedded system consists of an FPGA device, support memories to store the data and the FPGA configuration data, and control modules aimed to supervise the configuration process and to manage the I/O interface to send and receive the data to and from the FPGA device. The proposed compression system does not require any external hardware for implementing the decompression process, since this task is entirely mapped on the SRAM-based FPGA resources and exploits the internal configuration port. The following subsection illustrates the detailed system architecture implementing the decompression algorithm, the compression algorithm adopted and the overview on the configuration process.

The architecture of the proposed decompression system is implemented using the configurable resources available on the modern SRAM-based FPGAs. The scheme of the proposed compression system is illustrated in Figure 6.1.

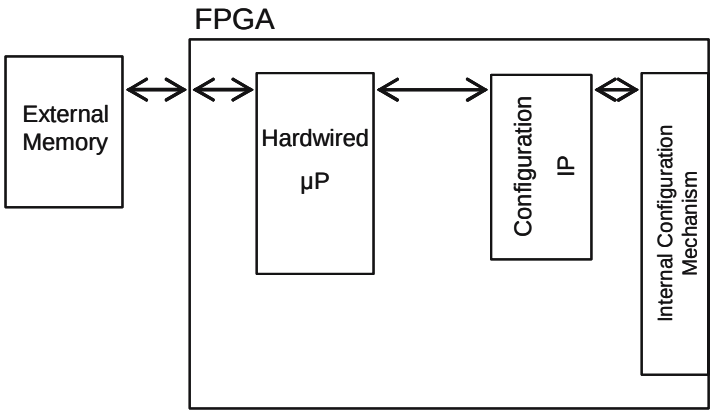

Figure 6.1. Architecture of the decompression system.

The proposed system is based on the assumption that the compressed configuration data are generated using a software compressor running on a PC, and then they are stored within the external memory belonging to the FPGA-based embedded system. The system implemented on the SRAM-based FPGA consists of three modules.
The system mapped on the SRAM-based FPGA consists of three modules:

- *Hardwired μP*: It is a hardwired microprocessor embedded within the SRAM-based FPGA architecture. It executes the decompression algorithm reading the compressed configuration data from the external memory and sending the decompressed data to the configuration IP.
- *Configuration IP*: It is a logic core implemented on the available logic resources provided by the SRAM-based FPGA device. It receives the decompressed data from the Hardwired μP and performs the configuration flow using the internal configuration mechanism.
- *Internal Configuration Mechanism*: It is an hardwired mechanisms that allow to access to the FPGA configuration memory reading or writing the configuration data. The configuration IP manages the operations of the Internal Configuration Mechanisms writing the decompressed configuration data into the FPGA configuration memory.

The proposed compression system is loaded on the FPGA during the bootstrap of the embedded system. Though partial configuration, we load the Configuration IP into the FPGA taking the uncompressed configuration data from the External Memory, and the compression algorithm within the hardwired μPas well. Then, the system is activated, and the processor starts executing the configuration of the rest of the FPGA by loading and decompressing the configuration data from the External Memory.

The compression algorithm we developed is based on a binary arithmetic encoder integrating an adaptive model that does not use a discrete number of bits for each symbol and may reach high compression ratio [15]. The main idea behind arithmetic coding with an adaptive model is to assign to each symbol an interval. The encoding task is then performed on a recursive probability interval partition that is progressively adapted to the changes of the symbol probabilities during the compression process.

```
/*Initialization*/
Set_Interval_Range (I)
Initialize_Adaptive_Model_Margin (I, Mi)
for each symbol SYM ∈ CONFIGURATION_DATA
  /*Arithmetic Coding*/
  Update_Interval(I)
  Coding(I)
  /*Data Out and Adaptive Scaling*/
  until Is_Scaling(I, Mi)
     Update_Interval(I)
     Scale_Data_Out(I)
Closing()
```

Figure 6.2. The flow of the proposed configuration data compression algorithm.

The flow of the proposed configuration data compression algorithm is described in Figure 6.2. The algorithm consists of two phases: the initialization and the arithmetic coding. During the initialization phase, the adaptive model interval is set with the interval value of [0,1[considering 256 symbols of 8 bit. The arithmetic coding phase elaborate each symbol belonging to the configuration data.

The encoding is performed on recursive probability interval partitions. The function `Update_interval ()` at each iteration splits into two sub-intervals the original adaptive model interval in such a way that the function `Coding()`adjusts the coded symbol, pointing to the base of the sub-interval that corresponds to the input symbol. At each iteration, an adaptive scaling is executed. This phase changes the symbol probabilities during the compression process in order to adapt to the changing contexts. Initially, the compression process starts with a basic model that does not produce any configuration data out. During the process, the function `Scaling_Data_Out()` adapts the interval model to the input symbols, and generates the correspondent compressed data out. Finally, the coding is ending by the function `Closing()`.

As far as the decoder side is concerned, the decompression flow is dual with respect of the encoder. The decoder performs the scaling and the arithmetic encoding achieving the original configuration data. The decoder algorithm is executed by the FPGA hardwired microprocessor that reads the

compressed configuration data from the external memory, and send to the configuration IP the original configuration data.

The configuration IP is a dedicated hardware that is mapped on the available logic resources provided by the SRAM-based FPGA device. The configuration IP is responsible for controlling the communication of the decompressed data to the FPGA configuration memory resorting to the internal configuration mechanism.

The architecture of the configuration IP consists of a data buffer that stores the decompressed configuration data received from the hardwired microprocessor. The data buffer has the dimension equal to the maximum configuration data that can be simultaneously written to the configuration memory. When the data buffer is full, the configuration IP enable the internal configuration mechanism and performs the writing of the decompressed configuration data into the selected frame of the FPGA configuration memory. The selection of the configuration memory frame is managed by the Configuration IP on the basis of the configuration architecture.

The execution flow of the proposed system can be summarized as follow. At the power-up of the system, the FPGA device is partially configured loading the configuration data that implement the Configuration IP. Once the FPGA is configured with the layout depicted in Figure 6.1, the code related to the configuration data decoder is loaded within the internal memory of the hardwired μP. After this operation the system is able to read the compressed configuration data from the external memory and to decompress it within the configuration memory of the FPGA. The logic resources related to the Configuration IP form a special boot-area that should not be over written by the decompressed configuration data. Since the architecture of the Configuration IP is extremely simple, it takes only a limited area of the device. The user-applications should be designed avoiding the using of the considered boot-area. This operation is very simple, and it can be performed automatically by the FPGA vendor floorplanning tools. By this way, the decompressed configuration data are written exclusively on the available FPGA resources without compromising the functionalities of the decompression system.

4. EXPERIMENTAL RESULTS

In order to show the feasibility and the characteristics of the proposed compression system, the performances are evaluated on a representative case study. At first the prototypal implementation of the developed decompression system is analyzed, and the performances in terms of area occupation and configuration time are analyzed. Secondly, the capabilities of the

compression algorithm on several real-world benchmark circuits are evaluated and compared versus the results coming from standard compression algorithms.

The cases study adopted are based on a prototype board consisting of a Xilinx Virtex-II Pro Platform [16]. In particular the system is mapped on a Xilinx SRAM-based FPGA XC2VP30 embedding two PowerPC microprocessor cores [17]. This device is characterized by a configuration memory of 11,589,920 bits that controls an array of 13,696 CLBs resources organized as a matrix array with 80 rows and 46 columns. This device has an internal configuration mechanism based on an Internal Configuration Access Port (ICAP). The ICAP module allows the internal FPGA logic resources to access to the configuration data reading or writing a specific configuration data.

The system is designed on the lower right side of the FPGA matrix array. In particular the configuration IP has been placed near the ICAP port that is located in the lower corner of the FPGA. The compression algorithm source code requires 8 KB of available memory within the hardwired μP, while the compression system mapped on the FPGA logic resources use 14 CLBs. During the execution the decompression algorithm uses only up to 12 KB of memory. The area overhead introduced by the proposed system is therefore of only 0.1%. In terms of external memory, 120 KB are required to store the bootstrap configuration data for the initialization of the decompression system.

4.1 Compression system results

The proposed configuration data compression system has been tested on the configuration data of several real-world circuits implemented on the XC2VP30 Xilinx SRAM-based FPGA. For this device we generated six configuration data for six different designs. We evaluated two computing cores consisting of a FIR filter and a Cordic Core, two controllers of CAN and USB interface, and two microprocessors, the Intel 8051 microcontroller and the Leon microprocessor. In particular, in order to estimate the compression ratio for applications that use high percentage of FPGA resources, we implemented two version of the microprocessors: plain and Triple Modular Redundancy (TMR). Furthermore, we initialized the BRAMs modules with random values in order to make difficult the work of the compression algorithm.

The characteristics of these designs are shown in Table 6.1, where we reported the differences between the designs in terms of used CLBs, BRAMs modules and percentage of used CLB resources with respected of the adopted FPGA device. Please note that the percentage of used resources considering the selected device, range from 2.3% to 92.3%

TABLE 6.1 Characteristics of the benchmark circuits

Circuit	CLBs [#]	BRAMs [KB]	Used CLBs [%]
FIR filter	321	0	2.3
Cordic	605	0	4.4
CAN Controller	1,611	4	11,8
USB Controller	1,920	6	14
8051	2,446	24	17.9
Leon	4,213	64	30.8
TMR 8051	7,338	72	53.6
TMR Leon	12,639	192	92.3

The results of the configuration data memory required from the developed system are illustrated in Table 6.3. We compared the original configuration data, the configuration data required by our system and those comings from the Xilinx system [3] and the data coming from the LZW compression system used in [7]. Please note, that the data reported for the proposed system includes also the amount of memory dedicated to the bootstrap configuration data. As it can be observed, the proposed compression algorithm shows an average memory reduction of about 5.5 times, compared to the 1.5 times of the Xilinx System. Besides, our approach achieves 82% savings in memory on the average, versus 41% obtained by the compression approach illustrated in [7] that needs an external hardware that performs the decompression. Besides, the bootstrap amount of memory is only needed one time for the first configuration, thus it is not a drawback when several applications configuration data are used.

In order to evaluate the performance characteristics in terms of configuration speed, the PowerPC has been configured at the running frequency of 300 MHz and the compressed data has been stored within the memory modules of the Xilinx Virtex-II Pro platform.

The configuration time needed by the developed system has been estimated in relation to decompress the configuration data and to configure the entire FPGA configuration memory. The obtained results are illustrated in Table 6.2, where are compared the time needed by the proposed configuration system versus the time needed by the Xilinx System configuration. With respect to the time, the overhead introduced by the developed system is proportional with the complexity of the circuits due to the arithmetic coding computation. For the considered benchmark circuits, the time overhead versus the Xilinx System ranges from 1.04 to 4.2 times. This limited drawback is overcome by the benefit of the compression ratio that can provide savings in memory ranging from 64% to 88%. However, considering small applications, those generally used in partial reconfiguration-based system, our system introduces a speed overhead of only 4%.

TABLE 6.2 Configuration time needed by our approach starting from compressed configuration data

Circuit	Proposed system configuration time [ms]	Xilinx system configuration time [ms]
FIR filter	125	120
Cordic	197	126
CAN controller	364	142
USB controller	388	144
8051	485	198
Leon	849	248
TMR 8051	895	310
TMR Leon	905	340

TABLE 6.3 Comparison of configuration data memory required by the analyzed compression systems

Circuit	Uncompressed [byte]	Proposed system [byte]	Xilinx system [byte]	LZW [byte]
FIR filter	1,448,812	167,812	998,320	854,742
Cordic	1,448,812	219,201	1,010,442	891,798
CAN controller	1,448,812	244,352	1,087,655	899,432
USB controller	1,448,812	276,910	1,122,990	912,560
8051	1,448,812	306,178	1,332,782	945,866
Leon	1,448,812	363,966	1,398,632	947,372
TMR 8051	1,448,812	399,453	1,420,844	998,560
TMR Leon	1,448,812	481,562	1,442,520	1,004,220

REFERENCES

[1] K. Compton, S. Hauck, *Reconfigurable Computing: A Survey of Systems and Software*, ACM Computing Surveys, Vol. 34, No. 2, June 2002, pp. 171–210.

[2] Xilinx Product Specification, *Virtex-II Platform FPGAs: Complete Data Sheet*, DS031 (v3.4) March 1, 2005.

[3] A. Khu, *Xilinx FPGA Configuration Data Compression and Decompression*, Xilinx – WP152, September 2001.

[4] Z. Li, S. Hauck, *Configuration Compression for Virtex FPGAs*, IEEE Symposium on Field-Programmable Custom Computing Machines, 2001, pp. 111–119.

[5] S. Hauck, Z. Li, E. J. Schwabe, *Configuration Compression for the Xilinx XC6200 FPGA*, IEEE Transactions on Computer Aided Design Integrated Circuits Systems, Vol. 18, No. 8, Aug. 1999, pp. 1107–1113.

[6] J. Pan, T. Mitra, W. Wong, *Configuration Bitstream Compression for Dynamically Reconfigurable FPGAs*, IEEE/ACM International Conference on Computer Aided Design, Nov. 2004, pp. 766–773.
[7] A. Dandalis, V. K. Prasanna, *Configuration Compression for FPGA-Based Embedded Systems*, IEEE Transactions on VLSI systems, Vol. 13, No. 12, Dec. 2005.
[8] M. Hübner, M. Ullmann, F. Weissel, J. Becker, *Real-Time Configuration Code Decompression for Dynamic FPGA Self-Reconfiguration*, 18th IEEE International Parallel and Distributed Processing Symposium, Apr. 2004, pp. 138.
[9] M. Ullmann, M. Hübner, B. Grimm, J. Becker, *An FPGA Run-Time System for Dynamical On-Demand Reconfiguration*, 18th IEEE International Parallel and Distributed Processing Symposium, Apr. 2004, pp. 135.
[10] A. Le, *Simplifying the FPGA Configuration Design Process*, Xilinx White Paper: Platform Flash PROMs, WP253, v1.0.1, Aug. 2006.
[11] Altera sheets, *Using Altera Enhanced Configuration Devices*, Chapter 14, Apr. 2003.
[12] Atmel FPGA, www.atmel.com
[13] Xilinx FPGA, www.xilinx.com
[14] V. Eck, P. Kalra, R. LeBlanc, J. McManus, *In-Circuit Partial Reconfiguration of Rocket-IO Attributes*, Xilinx Application Notes, XAPP662, May 26, 2004.
[15] A. Moffat, R. M. Neal, I. H. Witten, *Arithmetic Coding Revisited*, ACM Transactions on Information Systems, Vol. 16, No. 3, July 1998, 256–294.
[16] Xilinx Product Specification, *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, DS083 v4.5, Oct. 10, 2005.
[17] Xilinx Reference Guide, *PowerPC Processor*, EDK 6.1, Sept. 2, 2003.

Chapter 7

RECONFIGURABLE DEVICES FOR THE ANALYSIS OF DNA MICROARRAY

A complete gene expression profiling platform

A Deoxyribonucleic Acid (DNA) microarray is a collection of microscopic DNA spots attached to a solid surface, such as glass, plastic or silicon chip forming an array. DNA microarray technologies are an essential part of modern biomedical research. DNA microarray allows to compress in a little microscope glass, hundreds of thousands of different DNA nucleotide sequences, and permits to have all this information on a single image. The analysis of DNA microarray images allows the identification of gene expressions in order to drawn biological conclusions for applications that ranges from the genetic profiling to the diagnosis of cancer disease. Unfortunately, DNA microarray technology has a high variation of data quality. Therefore, in order to obtain reliable results, complex and extensive image analysis algorithms should be applied before actual DNA microarray information can be used for biomedical purpose. In this paper, we present a novel hardware architecture specifically designed to analyze DNA microarray images. The architecture is based on a dual core system implementing several units working in a single instruction-multiple data fashion. An FPGA-based prototypal implementation of the proposed architecture is presented in this chapter showing how reconfigurable devices can be used to increase the computation performances in biomedical applications.

1. INTRODUCTION TO THE DNA MICROARRAY

The Deoxyribonucleic Acid (DNA) microarray is a solid surface, such as glass, plastic or silicon chip studded with a large number of DNA fragments, each containing a nucleotide sequence that serves as probe of a specific

L. Sterpone, *Electronics System Design Techniques for Safety Critical Applications*,

gene. The purpose of DNA microarray is examining the expression of several thousands of genes simultaneously [1]. DNA microarray allows to identify and study the gene expression patterns that underlie cellular physiology, in particular in order to obtain the genetic profiling and identifying the differently expressed genes in cancer disease. DNA microarray images are generated by an automated scanning-laser microscope that scans a microarray slide with several blocks of two dimensional (2-D) arrays where the DNA fragments are localized. The purpose of DNA microarray data analysis is to draw biologically meaningful conclusions. In particular, the goal of the microarray image analysis is to extract absolute or relative intensity values from each DNA fragment (spot) that represents the gene expression level.

The results of a microarray experiment is presented in the form of an image, where the most expressed genes are indicated by high intensity spots with different channels ranging from the cyanine dyes, Cy3, that is the green, and the Cy5, that is the red. Several, microarray analysis steps need to be done before a conclusions is made.

The first stage of the analysis is called *gridding*, that is the process of assigning coordinates to the spot locations. The gridding (also known as addressing or grid alignment [2]) is a processing phase that aims to localize the positions of the spots that should be analyzed. This phase is fundamental since the localization off the exact spots positions allow to extract the correct information from the correspondent DNA fragment. Basically, the gridding process generate a grid where a set of spaces, parallel and perpendicular lines with the image content representing the 2-D array of the spots are registered. Thus, several squares of different dimensions are identified on the grid. Each square should be correctly placed on the correspondent spot. In order to find the spot on the 2-D array image, the gridding process is based on image processing algorithms which may identify objects into constituent regions. The edge detection algorithm is the most suitable solution in order to measure and recognize the spots positions [3]. This algorithm works on the microarray images placing the edges in the image with strong intensity contrast, considering that they occur at image locations representing spot boundaries.

The data is then *segmented* in order to separate the foreground pixels from the background. Once the spots are identified, this stage allow to select which pixels belong to the spot and which needs to be considered image noise. The third step is the *quality assurance* that corresponds in identify and avoid the analysis of low dependable spots. This step measures the quality of the previous steps by fixing some features to check, for example spot morphology, size, intensity, and homogeneity. Finally, comes the *intensity extraction* that corresponds to reading the intensity of expression of each

DNA fragments and classifying properly the DNA microarray information extracted.

The execution of a complete analysis flow of a DNA microarray image is a time consuming task. Previously developed approaches are mainly based on completely software-based solutions where the edge detection algorithm is executed by standard CPUs performing software routines on the whole pixels of the DNA microarray image [4–6]. In order to reduce the computational time some approaches try to down sample the microarray image, unfortunately this approaches have an high lost of accuracy that results in missing information about each DNA microarray spot [7]. All these solutions are extremely time expensive due to the complex and repeated CPU operations executed. Furthermore they have a computational speed that is inversely proportional to the image resolution, principally because edge detection algorithm is executed by software routines that elaborate groups of pixels, thus the elaboration time increases with the number of pixels of the processed microarray image. Previous works, essentially limited on the gridding process, have experimentally demonstrated the capability of parallel and multitasking architecture to drastically reduce the computational time [4].

A dual core architecture is presented in this chapter. It is able to perform a complete image analysis flow of a DNA microarray image. The aim of this work, is to support a full-automatic execution of the gridding and spot segmentation processes as well as the quality assurance and intensity extraction of ever DNA microarray image. The proposed approach is able to elaborate DNA microarray images in a fraction of time previously developed software-based approaches need. Furthermore, the proposed system is drastically increasing the capability of detecting spots in DNA microarray images since it does not loss any accuracy of the microarray image. In order to evaluate the effectiveness of the proposed architecture we use original DNA microarray images available from the Stanford University Microarray Database [8]. The experimental evaluation presents a maximum computational speed of one order of magnitude better than previously developed software-based approaches. Besides, detailed quality assessment analysis, demonstrated that the capability of detecting DNA spots, increases of more than 30% with respect to previously developed software-based approaches.

2. OVERVIEW ON THE PREVIOUSLY DEVELOPED ANALYSIS TECHNIQUES

Nevertheless the DNA fragment spots position should be prior known, since the DNA microarray devices are manufactured with regular structure, several issues during the biological process may influence the regularity of its

structure inserting noise and distortion of the scanned DNA microarray image. The major challenges result from the irregularity of the grid and the appearance of significant illumination noise that corrupt the expected illumination of the genetic markers where the DNA fragment probes are placed. The DNA microarray analysis requires finding the location of the microarray probes and the spots resulting from the biochemical reaction of the analyzed tissue. Once the locations have been determinate, various measurements technique can be performed in order to determine their discriminatory power and robustness [9].

The DNA microarray analysis has been addressed in two ways. First, the problem was attached by using a very accurate technology, for example, in the case of Affymetrix chips [10]. However, this technology is much more expensive than the commercial on-the-shelf (COTS) ones, and thus the need for solving the image analysis has remained. Second, the microarray analysis was addressed with template-based and data-driven approaches based on software routines. The template-based approach is the most relevant in the literature and it is based on complete software packages [12]. Vice versa, the data-driven approach has been based on statistical analysis of 1D image projections and by analysis using image segmentation algorithm [13, 14]. Several currently available software packages enable manual template matching by correcting the spot size, spot spacing and grid location [15]. However, the irregular grids morphology cannot be computed with template-based approaches unless a manual adjusted is defined. On the other sides the data-driven approaches are capable of finding irregular grids but provide low quality grids due to spurious or missing spots.

The purpose of the developed DNA microarray analysis system is to execute the image processes in a fully automated way, in order to reduce the human-operations and thus minimizing the measurement error introduced.

The contribute of this system consists of two major advantages. First, it is implemented only on a hardware platform and takes advantages of the parallel execution of the edge-detection algorithm during the several processing phases. Second, in comparison with other data-driven methods that work on all the image pixel reducing the computational time and the data accuracy, the analysis algorithm is based on a recursive spot segmentation algorithm that works on different DNA microarray image sub-regions thus avoiding the typical limits of the data-driven algorithms.

3. PRELIMINARIES OF DNA MICROARRAY IMAGE ANALYSIS

A DNA microarray image, as that depicted in Figure 7.1, is characterized by three main objects: the Spots, the Sub-grids and the Background. Theoretically, a DNA microarray image is characterized by deterministic grid geometry, known background intensity with zero uncertainty, pre-defined spot shape, and constant spot intensity that has to be different from the background. Finding such an ideal DNA microarray is almost impossible.

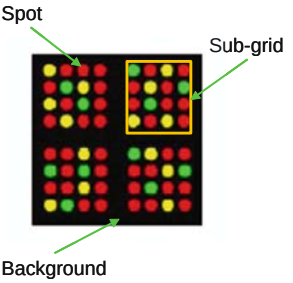

Figure 7.1. A slice of DNA microarray image.

A realistic DNA microarray image, due to the complex process which involves electrical, optical and chemical issues, is characterized by several differences with respect of an ideal image. These differences are mainly due to the variation of four parameters:

1. *Image Channels*: this is caused principally by the digital storage file formats. The digital image has to accommodate an analog signal, introducing sampling and quantization issues. The usage of lossy data compression techniques increases the variations of the original analog image. In our approach we do not use any data compression technique, therefore the proposed architecture does not increase the original image channel variation.
2. *Grid Geometry*: the irregularity of the DNA microarray sub-grid is often caused by the biological microarray preparation. This irregularity usually results in rotational offset. Our approach generates an *edge* output image which allows correcting the possible irregular spaced or rotated sub-grids.
3. *Background*: the presence of dust or dirty tissue during the acquisition procedure causes often a non linearity of the background intensity level. Thanks to a morphologic analysis of the sub-grids, our approach provides an *Intensity Background Ratio* that allows executing the appropriate noise-reduction filter to the whole image before its computation.

4. *Spot Morphology*: A large number of shape deviations exits, equals to the total number of spot cell within a sub-grid. Our approach identifies squares containing the spot morphology, whatever is the spot shape. The proposed approach is flexible and may be used also on not common DNA microarray technologies that adopt rectangular or triangular spots.

3.1 The edge detection algorithm

Edge detection is one of the key algorithms used in object recognition in images [16]. It consists in a 2-D first derivate operator applied to the grey-scale image to highlight regions of the image with high first spatial derivates. The edges are translated into ridges in the gradient magnitude of the image. The algorithm tracks along the top of these ridges and sets to zero all pixels that are not actually on the ridge top and give a thin line in the output. The edge detection of an image is the convolution products of the image pixels with different masks which result in the calculation of the horizontal and the vertical gradient. The two gradients are calculated using differences between adjacent pixels.

One way to find edges is to use the Prewitt kernels. The Prewitt kernels are based on the idea of the central difference, and are expressed by the following first order spatial derivates:

$$\frac{\partial I}{\partial x} \approx \frac{I(x+1, y) - I(x-1, y)}{2}$$

$$\frac{\partial I}{\partial y} \approx \frac{I(x, y+1) - I(x, y-1)}{2}$$

The two derivates correspond to a convolution kernel consisting in the horizontal convolution expressed by {-1,0,+1} and the vertical convolution expressed by {-1,0,1}. These convolutions are applied to the grey-scale image to get the horizontal and the vertical gradients.

$$\text{(a)} \quad \begin{Bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{Bmatrix} \qquad \text{(b)} \quad \begin{Bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{Bmatrix}$$

In order to perform the convolution on the entire image, the idea consists in building a $n \times n$ (typically 3×3) matrix of numbers called *kernel mask*, and in multiplying it with a portion of the image of the same dimension. Then, all the products are summed in order to determine the central pixel value. The kernel mask using Prewitt coefficients will be the matrix reported

in (a) for the vertical edge detection and the one reported in (b) for the horizontal edge detection. The results of the convolution will stay in the range defined by the pixel resolution in number of bit, thus if the pixel resolution is 8 bits, the range of the results will be [0:255]. If a resulting pixel exceeds the range, it has to be normalized.

4. THE PROPOSED DNA MICROARRAY ANALYSIS ARCHITECTURE

The main purpose of the proposed approach is to provide a faster and high-precise analysis of DNA microarray images in order to extract biologically valid information. As illustrated in Figure 7.2, the proposed methodology consist of two cores: the DNA-EDC (DNA-Edge Detection Core) and the DNA-QAC (DNA-Quality Assessment Core). The proposed system, on the basis of the DNA microarray image and of several data rules generates an image data gridding containing the information about the generated grid and segmentation of the considered DNA microarray image and the expression levels of the identified DNA fragments (spots).

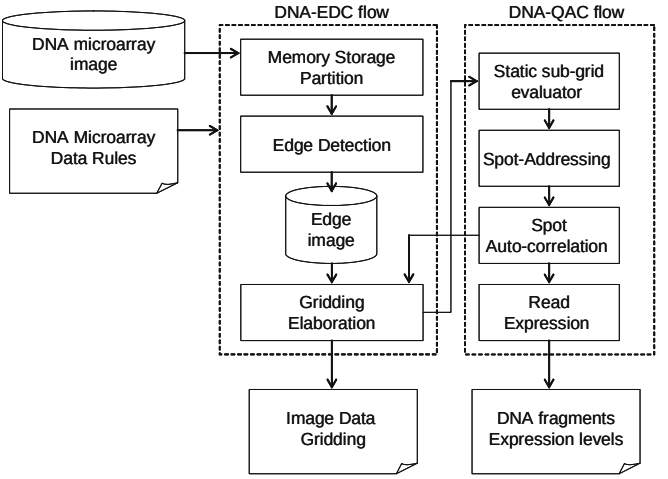

Figure 7.2. An overview of the proposed methodology.

The *DNA Microarray Data Rules* include the number of spots for each sub-grid in terms of rows and columns and the number of sub-grids within the DNA microarray image analyzed. Please note that all these information are provided by the manufacturer of the DNA microarray used.

The DNA-EDC flow consists of three phases:

1. *Memory Storage Partition*: this phase consists in dividing the original DNA microarray image in several frames. Each frame will be transferred in a single memory block within the DNA-Edge Detection Core.
2. *Edge Detection*: this phase performs the execution of the edge detection algorithm of the DNA microarray image. Each frame contained within the memory block is computed in parallel and the result of the computation is transferred to the output Edge image.
3. *Gridding Elaboration*: this phase generates the gridding matrix related to the analyzed image.

The result of the DNA-EDC flow is the *Image Data Gridding* that includes the following information:

(a) *Intensity Background Ratio*: it is the ratio between the image intensity of the background area and the image intensity of the several sub-grids area. This parameter is particularly important to estimate the quality of the image, and if needed to apply a post-processing noise reduction filter choosing the order on the bases of the Intensity Background Ratio.
(b) *Grid Matrix*: is a data matrix containing the coordinates of the lines that form the grid of the DNA microarray image. It consists of couples of data related to the horizontal and vertical positions.
(c) *Edge Image*: it is the output image after the computation of the edge detection phase. This image allows identifying the shape and the size of each spot.

The result of the DNA-QAC flow is the DNA fragments expression level of the identified spots. The expression level is reported as an index of the fluorescence for each individuated spot and can be used to extract biological information from the analyzed tissue. Considering the grid $G(x,y)$ and $\{x_1,.., x_n\},\{y_1,.., y_n\}$ the grid coordinates generated by the DNA-EDC module for all the spot locations, the DNA-QAC flow analyze each $G(x_i, y_i)$ and in case of the spot shape is not correctly identified, it re-computes the coordinate according to a Detailed Segmentation Spot algorithm (DeSSa).

The DNA-QAC consists of four phases:

(a) *Static Sub-Grid Evaluator*: this phase consists in estimate the spots position obtained after the gridding process performed by the DNA-EDC module.
(b) *Spot-Addressing*: this phase elaborates the spots that are not correctly identified. It modifies the spot coordinates in such a way that the spot *i* is correctly contained within the grid square $G(x_i, y_i)$.
(c) *Spot-Autocorrelation*: this phase consists in applying the correction of the grid square alignment performed in the previous phase, for all the spot coordinates corrected. It returns to the DNA-EDC gridding elaboration

phase the spot coordinates that have been fixed, and thus does not need to be re-computed by the DNA-EDC gridding elaboration phase.

(d) *Read Expression*: this is the last phase performed by the EDC-QAC flow. It is performed when all the spots are aligned in the best possible solution. It reports the final number of correctly identified spots, and for each spot *i* it returns the correspondent fluorescence value *V*.

The system operation may handle DNA microarray image of different dimension and with different spot shapes. In details, the system is based on two microprocessor core running in conjunction. The first processor, manages and synchronizes the task sequence of the DNA-EDC core. The second processor belongs to the DNA-QAC core and it is completely dedicated to run the DeSSa algorithm.

4.1 The edge detection architecture

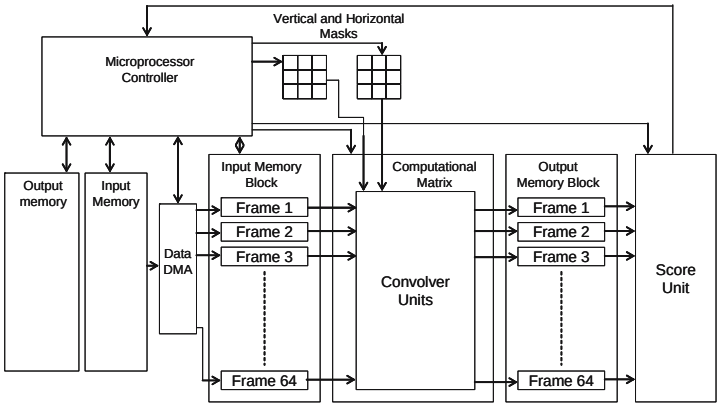

Figure 7.3. The architecture layout of the development DNA-Edge Detection Core (DNA-EDC).

The scheme of the developed architecture DNA-Edge Detection Core DNA-EDC is depicted in Figure 7.3. The DNA-EDC architecture consists of the following components:

1. *Master Microprocessor*: The Master Microprocessor manages the execution flow of the entire DNA-EDC architecture. It principally reads the image data from the input memory and controls the transfer and the computation operations.
2. *Input Memory*: The input memory stores the original DNA microarray image. It is organized in 32-bit data words, where each data word stores 4 data pixels. Considering that the original DNA microarray image consists of several pixels each one having an intensity value that ranges from the

level 0 to the level 255 (the color ranges from the black to the white considering the grey-scale), each pixel has been coded on 8 bits and the image is progressively stored in a raster way starting from the top-left to the bottom-right.

3. *Output Memory*: the output memory stores the image resulting from the edge detection process on the original DNA microarray image.
4. *Horizontal and Vertical Mask*: the horizontal and the vertical masks are two 3 × 3 matrices containing the coefficients that will be used for the computation of the edge detection process. Each matrix's cell is implemented by a 8-bit register.
5. *Input Memory Block*: the input memory block consists of 64 frames. Each frame is characterized by 16 buffer registers of 32 bits. The input data pixels are transferred from the input memory to each frame through the control of the Master Microprocessor.
6. *Data DMA*: the Data DMA unit is devoted to transfers the image data stream from the input memory to the Input memory block frames. This module is essential in order to guarantee faster data-transfer.
7. *Computational Matrix*: the computational matrix is the more complex part of the developed system. It consists of 64 convolver units. Where each convolver unit is able to compute the edge detection process on 9 pixels at every clock cycle. The inputs of each convolver unit are connected to the outputs of the corresponding frames within the input memory block and to the vertical or horizontal masks. Vice versa, the outputs from each convolver unit are connected to the correspondent frames within the output memory block.
8. *Output Memory Block*: the output memory block is formed by 64 frames of 16 32-bit buffer registers. Each frame is connected to the output signals coming from the correspondent convolver unit within the computational matrix. Vice versa, the output of each frame is connected to the Vertical and Horizontal Score unit.
9. *Score Unit*: the score unit computes the *vertical* and the *horizontal profile* of the edge detected data contained within each frame of the output memory block. The horizontal and the vertical profiles are the sum of each pixel intensity value on all the pixel columns and rows respectively.

The unit that executes the computation on the image and produces the correspondent edge detected image is the computational matrix. The computational matrix consists of 64 *convolver units* performing the computation of the convolution multiplication in parallel. The architecture of a convolver unit is illustrated in Figure 7.4.

It consists of an arithmetic architecture that executes the convolution operation between two 3 × 3 matrices of data pixels.

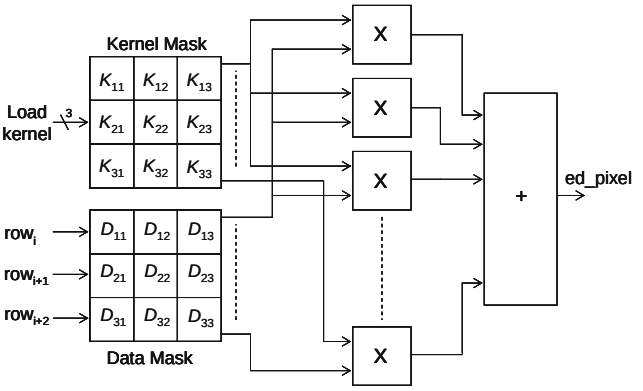

Figure 7.4. The architecture of the developed convolver unit.

The architecture is composed by two matrices: the Kernel Mask and the Data Mask. The Master Microprocessor transfers into the Kernel Mask the Horizontal or the Vertical Mask depending if the edge detection is computed on the horizontal or vertical direction. The convolution operations are executed by nine multipliers and one adder, where each multiplier perform the multiplication between the matrix cells K_{ij} and D_{ij}, and the adder perform the additions between the nine multiplier outputs. The adder output *ed_pixel* is linked with the correspondent frame within the output memory block.

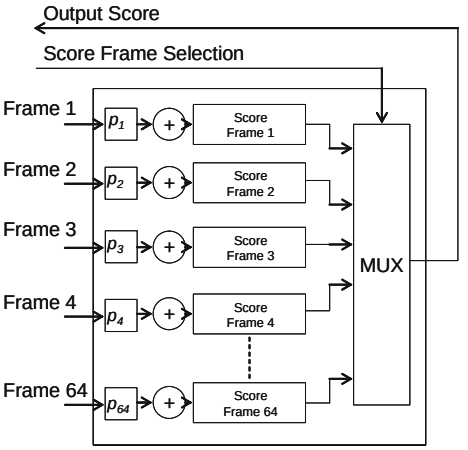

Figure 7.5. The architecture of the developed score unit.

The Score unit generates horizontal and vertical histogram projections that sum the values of the edge pixels in both the directions. These values are used by the Master Microprocessor in order to create the data matrix

containing the coordinates of the lines that form the gridding of the DNA microarray image. The architecture of the Score unit, which is depicted in Figure 7.5, consists of a multiplexer controlled by the Master microprocessor, a couple of 64 parallel registers of 32 bits and 64 adders.

The lines of the Output memory block frames are directed into the correspondent score unit input register P_i. The content of each register P_i is added to the Score Frame *i* at every computation. At the end of the elaboration of each pixel row, the Score Frame *i* contains the histogram projection of the correspondent pixel *i*. The Master Microprocessor transfers the content of the 64 Score Frame registers and computes the horizontal/vertical grid lines on the basis of the minimum score parameter and the number of searched lines. At the end of the computation the content of the Score Frame registers is reset.

4.2 The quality assessment core

The DNA-Quality Assessment Core (DNA-QAC) has been developed in order to extract the expression level of each identified DNA spot. It is based on a *Detailed Spot Segmentation Algorithm* (DeSSa) executed on a microprocessor core working in conjunction with the DNA-EDC core. The flow of the proposed DeSSa algorithm is reported in Figure 7.6. The algorithm consists of four steps, as introduced in the previous section: *static sub-grid evaluator, spot addressing, spot auto-correlation* and *read-expression*.

The static sub-grid evaluator consists in analyzing all the spot area (*SP*) identified by the coordinates *G(x,y)*. The function `analyze_spot_shape()` compares the pixels intensity contained in every grid square identified by the coordinates *G(x,y)* with a defined spot shape. The defined spot shape is set by the user, considering the manufacturing characteristics of the DNA microarray device used. In the case, the spot shape is not correctly fitted, the spot square coordinates are added to a Not-Defined-Spot (*NDS*) list.

The spot addressing phase modifies the spot square coordinates of the NDS list. This phase search the possible grid square position that allows to contain a single spot. The process is divided in two functions:

1. `extend_spot_area()`: this function progressively enlarges the grid's square dimension on all the four directions in order to fit the spot area completely.
2. `min_neighborhood()`: this function computes the intensity ratio (`threshold`) of the signal in the neighborhood area of the considered grid square. If the signal intensity ratio is lower than the minimum spot shape parameter set according to the DNA microarray manufacturing characteristics, the grid's square dimension is considered as fixed.

Once the spot addressing is finished, the grid's square positions need to be correlated with the other grids previously computed. This process is performed by the DNA-EDC core that starts the gridding elaboration phase. This phase is executed only for the grid area not belonging to the NDS list. The three algorithm steps are repeated until the update process is completely for all the grid squares.

Finally, the last step of the DeSSa algorithm consists in computing the expression level of all the detected spot area. The expression level is provided as an average intensity value computed for all the pixels belonging to spot area defined by the grid *G*.

```
DeSSa()
  {
   do
    {
    /*Static Sub-Grid Evaluator*/
    for each spot area SP ∈ G(x,y)
     {
      NDS(i) = analyze_spot_shape(SP, G(x,y))
     }
    /*Spot Addressing*/
     for each not-defined-spot i ∈ NDS
      {
       min=false
       while(!min)
        {
         extend_spot_area (i,G(x,y))
         threshold = min_neighborhood (i, G(x,y))
        if (threshold < min_spot_shape(i))
           min=true
        }
      }
    /*Spot Auto-Correlation*/
    update_NG_list(NDS,G(x,y))
    } while (update_NG_list(NDS) == not_completed)
    /*Read Expression*/
   for each spot area SP ∈ G(x,y)
     compute expression_level (SP)
  }
```

Figure 7.6. The flow of the developed Detailed Spot Segmentation Algorithm (DeSSA).

5. EXPERIMENTAL RESULTS

A prototype of the system architecture has been developed on a Xilinx Virtex-II Pro Development System board embedding a XC2CP30 SRAM-based FPGA device and using an external memory of 256 Mb [17]. This FPGA device embeds two hardwired microprocessors PowerPC 405 which consists of a 32-bit hardware architecture [18] and it consist of 136 memory

block of 18 Kb each one, 27,392 Flip-Flops (FFs) and 27,392 Look-Up Tables (LUTs) organized in a matrix of 13,696 logic cells. We implemented the architecture layout depicted in the section IV using the two PowerPC 405 as the controller for the DNA-EDC core and for the DNA-QAC core. We divides the external memory in two banks in order to implement the input and output memories. We set the clock frequency of the entire system at 200 MHz. The used resources of the implemented system are shown in Table 7.1, where we reported the number of used FFs, LUTs and BRAMs (divided in number of block and total K-Byte used) for each module of the developed system. In order to guarantee fast data computation, we mapped the internal registers of the input and output memory blocks exploiting the dual port Block-RAM resources of the Xilinx FPGA. In particular, we mapped two frame registers for each Block RAM.

TABLE 7.1 Prototypal characteristics of the developed system

Module	FFs [#]	LUTs [#]	BRAMs	
			[#]	KB
Data DMA	155	1,684	0	0
Input memory block	1,400	860	32	4
Computational matrix	25	12,032	0	0
Output memory block	1,408	894	32	4
Score unit	15	6,804	32	0.5
DNA-QAC	568	360	12	4

The performance capabilities of the developed system have been evaluated on original case study DNA microarray images available from the Stanford Microarray Database [11] and containing images of several kind of DNA microarray devices and image quality. On the considered images we configured the system in order to compute the edge detection algorithm using the Prewitt masks [5]. The characteristics of the analyzed images are illustrated in Table 7.2 while the results obtained are shown in Table 7.3. Where it is reported as DNA microarray ID, the reference identification number of the considered image form the Stanford University Database Category, the kind of DNA microarray image analyzed; Dimension, the image dimension in term of number of pixels for rows and columns; the Computational time, the computational time for the proposed system and for the pure software approach presented in [7] executed on a Pentium-II processor equipped with 1 Gbyte of RAM, and running at 1,6 GHz, and finally the performance quality of the obtained gridding considering the percentage of correctly individuated spot over the total number of spot belonging to the considered DNA microarray devices.

TABLE 7.2 DNA microarray images characteristics

DNA microarray ID [#]	Category	Dimension
10,029	Adenoma – liver	1,900 × 3,640
3,657	Brest – tumor tissue	1,992 × 1,870
12,507	Lymphoma – normal tissue	1,940 × 5,496
16,940	Lymphoma – follicular	1,940 × 5,548
12,485	Solid tumor – primary	1,920 × 5476
12,395	Metastatic tumor – liver	2,016 × 3,744
40,600	Neurobiology – amplification	2,048 × 5,680
34,905	Stress – drug treatment	1,888 × 5,500
67,549	Normal tissue – whole blood	1,894 × 5,512

TABLE 7.3 Experimental results of the proposed dual core system for the analysis of DNA microarray images

DNA microarray ID [#]	Performance [s]		Spot coverage [# identified spots / # existing spots]	
	Proposed approach	Software approach	Proposed approach	Software approach
10,029	10.9	194.3	0.97	0.61
3,657	4.9	73.8	1	0.87
12,507	15.4	138.5	1	0.3
16,940	16.3	136.4	1	0.64
12,485	16.5	171.8	0.98	0.68
12,395	10.4	104.2	0.99	0.58
40,600	22.7	166.1	0.98	0.64
34,905	15.2	182.4	0.97	0.78
67,549	16.2	145.7	1	0.82

On the considered case study, it has been recorded an average percentage of individuated spots of 98% versus the 66% obtained with the approach proposed in [7]. These results demonstrated that the proposed system is able to analyze DNA microarray images introducing only a minimal error in the obtained DNA microarray spots expression level. Besides, it is clearly illustrated a reduction of the computational time of more than one order of magnitude with respect to a pure software solution. This result demonstrates that the usage of hardware-accelerated architectures could drastically improve the analysis of DNA microarray images.

REFERENCES

[1] Amos Mosseri, Eitan Hirsh, *Analysis of Gene Expression Data*, Lecture 3, Tel Aviv University, 2005.

[2] Y. H. Yang, M. J. Buckley, S. Dudoit, T. P. Speed, *Comparison of Methods for Image Analysis on cDNA Microarray Data*, Dept. Statistic., University of California at Berkeley, Tech. Rep. 584, Nov. 2000.

[3] B. Fisher, S. Perkins, A. Walker, E. Wolfart, *Hypermedia Image Processing Reference*, Department of Artificial Intelligence, University of Edinburg, Available: http://www.cee.hw.ac.uk/hipr/html/index.html

[4] L. Sterpone, M. Violante, *A New FPGA-Based Edge Detection System for the Gridding of DNA Microarray Images*, IEEE Instrumentation and Measurement Technology Conference, 2007, pp. 1–6.

[5] P. Bajcsy, *An Overview of DNA Microarray Image Requirements for Automated Processing*, IEEE Conference on Computer Vision and Patter Recognition, Vol. 3, No. 1, 2005, pag. 147.

[6] Yuan-Kai Wang, Cheng-Wei Huang, *DAN Microarray Image Analysis Using Active Contor Model*, IEEE Computational Systems Bioinformatics Conference, 2005, pp. 12–13.

[7] P. Bajcsy, *Gridline: Automatic Grid Alignment DNA Microarray Scans*, IEEE Transactions on Image Processing, Vol. 13, No. 1, Jan. 2004, pp. 15–25.

[8] X. H. Wang, Robert S. H. Istepanian, Yong Hua Song, *Microarray Image Enhancement by Denoising Using Stationary Wavelet Transform*, IEEE Transactions on Nanobioscience, Vol. 2, No. 4, Dec. 2003, pp. 184–190.

[9] X. Wang, S. Ghosh, S. W. Guo, *Quantitative Quality Control in Microarray Image Processing and Data Acquisition*, Journal on Nucleic Acids Research, Vol. 29, No. 15, 2001.

[10] Affymetrix Inc., *Gene Chip Arrays*, Product Description at http://www.affymetrix.com/

[11] Stanford University, *Stanford Microarray Database* at http://smd.stanford.edu/

[12] Axon Instrument Inc., *GenePix Pro*, Product Description at http://www.axon.com/

[13] M. Steinfath, W. Wruck, H. Seidel, H. Lehrach, U. Radelof, J. O'Brien, *Automated Image Analysis for Array Hybridization Experiments*, Bioinformatics, 2001, pp. 634–641.

[14] A. N. Jain, T. A. Tokuyasu, A. M. Snijders, R. Segraves, D. G. Albertson, D. Pinkel, *Fully Automated Quantification of Microarray Image Data*, Genome Research, Vol. 12, No. 2, Feb. 2002, pp. 325–332.

[15] J. Buhler, T. Ideker, D. Haynor, *Dapple: Improved Technique for Finding Spots on DNA Microarrays*, UV CSE Technical Report UWRT 2000-08-05.

[16] J. F. Canny, *A computational approach to edge detection*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 8, No. 6, Nov. 1986, pp. 769–798.

[17] Xilinx Product Specification, *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheets*, DS084 v4.5, Oct. 10, 2005.

[18] Xilinx Reference Guide, *PowerPC Processor*, EDK 6.1, Sept. 2, 2003.

Chapter 8

RECONFIGURABLE COMPUTE FABRIC ARCHITECTURES

A new design paradigm

Re-Configurable Mixed grain (ReCoM) is a novel Reconfigurable Compute Fabric (RCF) architecture based on a mixed-grain reconfigurable array which combines a RISC microprocessor and a reconfigurable hardware for computation-intensive applications. ReCoM comprises a modified RISC microprocessor, a dynamically reconfigurable processing array including reconfigurable cells formed by a 64-bits ALU, Look Up Tables (LUTs), word-level arithmetic units, and an efficient configuration and data memory architecture.

High-performance execution of complex algorithms involves massive computations. In the past, custom application-specific architectures have been used to satisfy these demands. This implementation approach, while effective, is expensive and poorly flexible since hardwired application-specific architectures are extremely expensive to evolve and maintain. As a matter of that, a fixed, application specific architecture will require significant redesign in order to assimilate new algorithms and new hardware components. A flexible system must function in rapidly changing environments, resulting in multiple modes of operation. On the other side, efficient hardware architectures must match algorithms to maximize performance and minimize resources. Reconfigurable devices, such as Reconfigurable Compute Fabrics (RCFs) allow the implementation of architectures that change in response to the changing environment. In general, RCFs have wider applicability than Application Specific Integrated Circuits (ASICs) or general-purpose processors alone.

A novel model for RCFs targeted at computation-intensive applications, called ReCoM, is introduced in this chapter. The ReCoM architecture consists of a Tiny RISC microprocessor core [1], a dynamically reconfigurable array,

L. Sterpone, *Electronics System Design Techniques for Safety Critical Applications*,

a reconfigurable management unit and a memory interface. The main characteristic of ReCoM is given by the reconfigurable array based on a mixed-grain reconfigurable cell architecture including a 64-bits ALU, Look Up Tables (LUTs) and word-level arithmetic units, that may target both word-level and bit-level granularity applications.

The capabilities of the proposed reconfigurable system have been validated on a representative case study implementing a FIR filter. Furthermore, the performance obtained by ReCoM have been compared with those coming from a DSP and a previous developed reconfigurable system, showing an improvement of at least three times in term of computational speed.

1. INTRODUCTION TO RCF DEVICES

The range of existing reconfigurable architectures is divided in two main categories: fine and coarse grained approaches. Fine grained devices are optimized to implement glue logic or irregular structures like finite state machines. Conversely, coarse grained devices are optimized to implement word level computational intensive applications.

Fine grain prototypes are generally built on a computational model based on a unique processor. They include prototypes such as DPGA [2] or Garp [3] especially oriented to application domains such as bit-level computation or image processing and cryptography. On the other side, coarse grained prototypes are based on an array of processing units organized in a Multiple-Instruction Multiple-Data (MIMD) or in a Single-Instruction Multiple-Data (SIMD).

MIMD architectures may be used in a wide range of application areas, such as computer-aided design/manufacturing, simulation, modeling and communication switches. Examples of MIMD-based reconfiguration systems are MATRIX [4] or RAW [5].

The recent years have seen the introduction of many computation-intensive tasks as mainstream applications that manipulate large arrays and matrices in minimal time. These tasks are performed efficiently on SIMD architectures. Reconfigurable systems based on SIMD array are REMARC [6], MorphoSys [7] or DReAM [8]. REMARC is a reconfigurable coprocessor that is tightly coupled to a main RISC processor and consists of a global control unit and 64 programmable logic blocks called nano processors. Similarly, the MorphoSys architecture comprises five components: a core processor, a reconfigurable array, a context memory, a frame buffer and a DMA controller. A three layer interconnection network gives to the reconfigurable array high connectivity. Another coarse grained reconfigurable device is the Dynamically Reconfigurable Architecture for Mobile System (DReAM). It consists of an array architecture of reconfigurable processing units (RPUs) optimized for the requirements of mobile communication

system. Each RPU consists of two dynamically reconfigurable 8-bit data paths and two 16 by 8-bit dual port RAMs. The dual port RAMs are used as LUTs when performing multiplication operations.

A medium-grain reconfigurable cell array prototype has been previously developed in [9]. This prototype is based on a matrix of programmable 4-bit cells where each cell performs a small portion of the overall algorithm.

ReCoM has several enhancements if compared with previous SIMD-based or medium-grain reconfigurable systems. It has a configuration and data transfer architecture that could be controlled independently by the reconfigurable array and a multi domains dynamically reconfiguration unit that permits configuration swap oriented to multi tasking applications. Finally, the ReCoM's reconfigurable array incorporates mixed grained-based cells that could be used in order to implement word-level or bit-level granularity applications.

2. THE ReCoM ARCHITECTURE

The architecture of the proposed reconfigurable compute fabric ReCoM is illustrated in Figure 8.1. The ReCoM's main components include a *Reconfigurable Unit*, a RISC processor (*Tiny RISC*), two DMA controllers (one related to the *configuration*, and one to *data* stream) managed by the RISC processor and a *data* DMA controller managed by the reconfigurable unit.

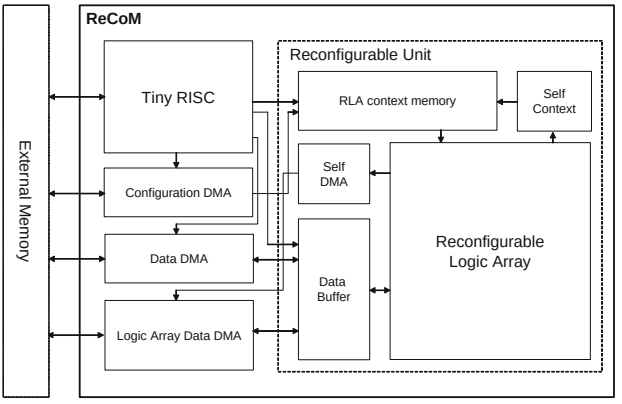

Figure 8.1.The architecture of the reconfigurable compute fabric system ReCoM.

The reconfigurable unit is composed of several sub-components: a *Reconfigurable Logic Array (RLA)*, a *context memory*, a *Data Buffer*, a *Self Context* and a *Self DMA* units.

The reconfigurable logic array is configured by the RLA context memory, while the Tiny RISC is the main processor that manages the DMAs

dedicated to the data/configuration flow towards the reconfigurable logic array and that drives the RLA context memory. Vice versa, the Self Context unit allows the reconfigurable logic array to partially or totally reconfigures itself independently from the control of the main processor. Furthermore, the Self DMA unit can manage a dedicated DMA (*Logic Array Data DMA*) in order to transfer data to/from the external memory without the participation of the main processor. This is extremely useful in order to exploit the parallelism available in an application's algorithm.

The main processor of ReCoM is a 32-bit processor, called TinyRISC [1]. Tiny RISC is a 4-stages pipelined processor with four registers in addition to the register file and the special register file. One is the program counter register, which contains the address of the program execution point. The other three are the pipeline registers, which provide the latched interface between each pipeline stage. For ReCoM, the Tiny RISC pipeline structure has been modified according to the scheme illustrated in Figure 8.2. Furthermore, several instructions are added to the original Tiny RISC ISA in order to manage the configuration/data DMA, the RLA context memory and the data buffer behavior.

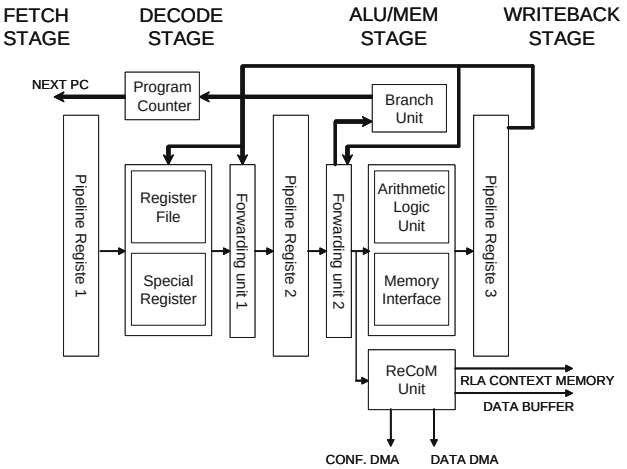

Figure 8.2. Tiny RISC pipeline stages modified with the ReCoM executing unit.

A *ReCoM unit* is included, that executes the instructions added to the original Tiny RISC ISA. These instructions and their correspondent operations are reported in Table 8.1. There are three different categories of these instructions: instructions related to the execution of the program by the reconfigurable array, instructions related on the behavior of the reconfigurable array and configuration/data DMA.

TABLE 8.1 Instruction set added to the ISA of ReCoM

Instruction	Description of operation
LOADCM	Load from the external memory to the RLA context memory the program to be executed by the reconfigurable unit
REXEC	Configure the reconfigurable unit cells transferring a configuration set from the RLA context memory to the context word registers
LOADB/SAVEB	Transfer the data from/to the external memory to/from the data buffer within the reconfigurable unit using the data DMA
LOADEX/SAVEEX	Configure the reconfigurable cells loading a context from the context memory and concurrently store/save the data from the data buffer to the reconfigurable cells considering the specified configuration table
LOADCT/SAVECT	Configure a reconfigurable cell in such a way to manage transfer data from/to the data buffer within the reconfigurable unit to/from the external memory using the Self-DMA unit
LUTC	Configure the content of a LUT's word within the RLA matrix

Where the reconfigurable array instructions control the execution of the RLA by specifying the memory context that will be executed, the address location within the RLA, the data address of the data buffer and the functionalities of the Self Context unit. Otherwise, the instructions related on the behavior of the reconfigurable unit define the functions of the LUTs embedded in each reconfigurable cell. Finally, the configuration/data DMAs initiate configuration and data transfer between the main memory and the data buffer.

The reconfigurable unit is the main component of the ReCoM system. It consists of a Reconfigurable Logic Array (RLA) of 8 × 8 reconfigurable cells placed in an interconnection net, an RLA context memory, a Data Buffer and two Self components dedicated to the context and to the data DMA.

The basic component of the RLA matrix is the reconfigurable cell. As is illustrated in Figure 8.3, the reconfigurable cell is composed by an ALU (64-bits fixed-point operations) working on two 32-bits wide operands, two LUTs of 8-bits wide input and 16-bits wide output, two 32-bits registers, a register file composed of 15 registers (where R13 is connected to Self-DMA unit and R14 is connected to the Self Context unit), and several multiplexers that controls the data path. Besides, a 32-bits context word register configures all the components excepting the two LUTs that are configured by the main processor through memory mapping.

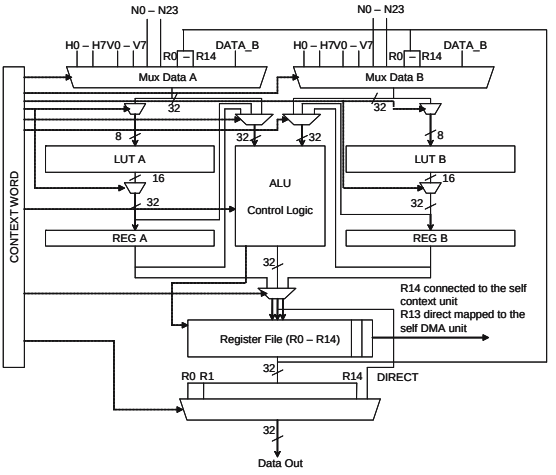

Figure 8.3. Reconfigurable cell architecture.

The ALU arithmetic unit implements three kinds of standard logic and arithmetic functions:

1. *Logic operations*: AND, OR, XOR and NOT
2. *Arithmetic operations*: ADD, SUB and MUL
3. *Other operations*: BYP (bypass operand to register file), RST (clear register file) and KEEP (no ALU operations)

The operation that will be executed by the ALU is specified through two fields: *opcode* (2 bits) and *sub-opcode* (4 bits). While the destination register file is specified by the field *ResultReg* (4 bits).

The inputs of a reconfigurable cell are selected by the multiplexers (*Data A* and *Data B*) that can be linked to two kinds of resources:

1. The data buffer or the register file, using the reconfigurable cell internal interconnection
2. The register file of another reconfigurable cell placed in the same row/column (H/V) or within the neighborhood (N)

Furthermore, the reconfigurable cell architecture includes two 4Kbits LUTs that are based on 8-bits wide inputs that select one of the 256 16-bits wide output words. The configuration words of the LUTs are memory mapped. Thus, the content of each LUT's word is load by the main processor defining one of the 2^{15} possible addresses.

Considering the configuration memory, ReCoM is based on the RLA context memory. It is organized in four blocks where each block contains eight sets. Each set can store eight context words. There are two possible ways to transfer the data into the context word registers: *context broadcast* and *selective context enabling*.

The context broadcast mode consists in transferring a single set in row-wise or column-wise operations to all the context words of the RLA matrix. Where in the case of row-wise operations all the reconfigurable cells of a row are configured with the same context word. Vice versa, in the case of column-wise operations, all the reconfigurable cells of a column are configured with the same context word.

The selective context enabling consists in transferring a single set to only one row or column of the RLA matrix. In this case each reconfigurable cell of the selected row/column may be configured in a various way.

The two different modes of transferring the configuration contexts permit to manage rapidly the context words reconfiguration in order to guarantee the effectiveness of the architecture's parallelism. The RLA context memory can be uploaded concurrently during the execution of the reconfigurable cells, since both the configuration modes may be executed in one clock cycle. Thus, the reconfiguration time is reduced to zero allowing the dynamic reconfiguration of the RLA matrix cells.

The ReCoM network is a hierarchical multi domains collection of 32-bit busses. The interconnect distribution is similar to traditional FPGA interconnections architecture. Differently from traditional FPGA, ReCoM has the possibility to dynamically switch the interconnection network between the reconfigurable cells.

The ReCoM's interconnection network includes two interconnection levels, as shown in Figure 8.4. The first interconnection level (Level 1) has a direct interconnection between the reconfigurable cells on the same row and column (H/V). The second interconnection level (Level 2) includes direct network interconnection provided between the reconfigurable cells within three Manhattan grid squares (N). The results are transmitted over local multiplexers and they are available in the destination reconfigurable cells in one clock cycle.

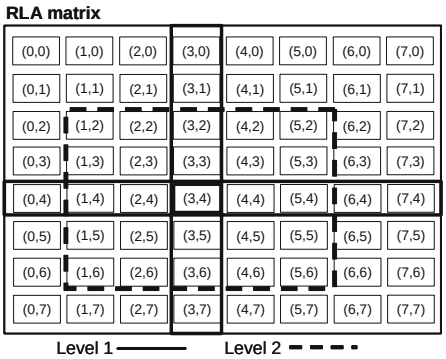

Figure 8.4. Interconnection network levels.

The Data Buffer is the component devoted to the transfer of the data to/from the external memory from/to the reconfigurable cells within the

RLA matrix. It consists of three parts, as is represented in Figure 8.5: a *Data Memory*, a *Configuration Table* and a *Selection Logic*.

The Data Memory is organized in 256 banks composed by 64 sets of 32-bits data words. Each set consists 2,048 bits. The division in sets supports the concurrent execution of the data transfers and computation operations: if one set provides a data stream of 2,048 bits for the RLA matrix computations and stores data results from the RLA matrix; another set stores data into the main memory through the control of one DMA controller and reloads data for the next computation. The configuration table is organized in 16 words of 384 bits. Each word is used to control a Selection Logic that determines the order in which the data are transferred to/from each reconfigurable cell within the RLA matrix.

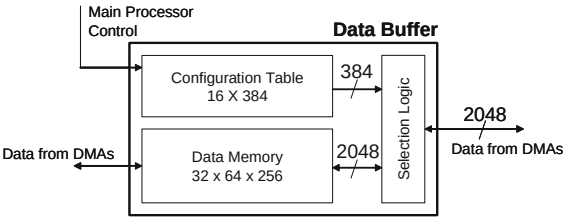

Figure 8.5. Data-buffer architecture.

The *Self-Context* and *Self-DMA* units allow the reconfigurable unit to reconfigure itself and to transfer data to the external memory independently from the Tiny RISC execution.

The Self-Context unit is controlled by an internal 32-bits register that can be addressed by each reconfigurable cell through the register file R14. The Self-Context unit generates the signals towards the RLA context memory in such a way to control the dynamic partial and total reconfiguration capability of the RLA matrix.

On the other side, the Self-DMA unit is controlled by an internal 32-bits register addressable by each reconfigurable cell through the register file R13. This unit controls a specific DMA (*Logic Array Data DMA*) in order to manage the data transfer from/to the reconfigurable array to/from the external memory, independently of the main processor functionality. These two units may be used effectively to increment the performance capability of the reconfigurable system, since the main processor can be discharged of the data transfers and configuration management.

The ReCoM system operation may handle application tasks of different nature. In details, the Tiny RISC processor manages the sequential tasks and controls the reconfigurable system, while the reconfigurable unit is used to support tasks with high data-parallel operations. The execution of such tasks is denoted by several steps. An overview on these steps is described as follow:

1. The context is loaded from the external memory and transferred into the RLA context memory through the execution of the function *LOADCM*.
2. The context related to the operations executable independently from the main processor is loaded by the function *AUTOCTX*. Otherwise, selective operations may be programmed by the functions *LOADCT* and *SAVECT*, while the LUTs are programmed by the function *LUTAC*.
3. The operation of the RLA matrix may be executed concurrently with the data transfer with the functions *LOADEX*, *SAVEEX*. Besides, the LUTs may be programmed with the function *LUTA*. Otherwise the parallel execution may be performed also using the functions *REXEC*, *LOADB* and *SAVEB* if the computation or the data transfer tasks have an independent length.

3. EXPERIMENTAL RESULTS

The functionality of ReCoM system has been specified in a prototypal behavioral VHDL. The entire system has been modeled along with external memory. The VHDL model of ReCoM has been used to simulate a simple benchmark application consisting in a FIR Filter. We selected two kinds of FIR Filters: one with 4 taps and another with 8 taps and we assume to work on 16-bit fixed-point data.

The methodology we adopted to map the FIR filters may be used for every N taps FIR Filter with $N \leq 64$.

The performance characteristics of the mapped FIR Filter implemented within ReCoM are shown in Table 8.2 assuming to have preload within the external memory 256 samples. In table 8.2, we reported the number of data input necessary for each computation (N_{valIN}), the number of instruction executed for the data computation (N_{Instr}) and the number of computational phase needed to generate all the output results (N_{elab}).

Table 8.2 Characteristics of the mapped FIR filters

# taps	N_{valIN}	N_{elab}	N_{Instr}
4	19	16	6
8	15	32	6

The performances of ReCoM are analyzed and compared versus a previous developed reconfigurable system called Morphosys [10] and versus the fixed-point DSP TM320C55X manufactured by Texas Instruments [11]. In order to make the comparison feasible we compute the Million Samples per Second (MSPS) considering a running frequency of 100 Mhz.

Table 8.3 Performances comparison of different system

	MSPS		
# taps	ReCoM	DSP TM [11]	MorphoSyS
4	267	25	89
8	133	17	80

The comparison results are illustrated in Table 8.3. From these results it possible to observe that ReCoM is about ten times faster versus the DSP TM320C55X that does not implements any reconfigurable computing features. Furthermore, ReCoM is three times faster with respect of the dynamically-reconfigurable system Morphosys.

REFERENCES

[1] A. Abnous, C. Christensen, J. Gray, J. Lenell, A. Naylor, N. Bagherzadeh, *VLSI Design of the Tiny RISC Microprocessor*, Custom Integrated Circuits Conference, May 1992, pp. 30.4.1–30.4.5.

[2] E. Tau, D. Chen, I. Eslick, J. Brown, A. DeHon, *A First Generation DPGA Implementation*, FPD'95, Canadian Workshop of Field-Programmable Devices, May 1995.

[3] J. R. Hauser, J. Wawrzynek, *Garp: A MIPS Processor with a Reconfigurable Co-Processor*, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, Apr. 1997.

[4] E. Mirsky, A. DeHon, *MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources*, Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, Apr. 1996, pp. 157–166.

[5] M. Taylor, *The RAW Prototype Design Document, Spread Sheet Documents*, Massachusetts Institute of Technology, Sept. 6, 2004.

[6] T. Miyamori, K. Olukotun, *A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications*, Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, Apr. 1998.

[7] H. Singh, M -H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, E. Chaves Filho, *MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications*, IEEE Transactions on Computers, Vol. 49, No. 5, May 2000.

[8] A. Alsolaim, J. Starzyk, J. Becker, M. Glesner, *Architecture and Application of a Dynamically Reconfigurable Hardware Array for Future Mobile Communication Systems*, IEEE Symposium on Field-Programmable Custom Computing Machines, 2000, pp. 205.

[9] J. G. Delgado-Frias, M. J. Myjak, F. L. Anderson, D. R. Blum, *A Medium-Grain Reconfigurable Cell Array for DSP*, Proceedings of Circuits, Signals and Systems, 2003, p. 391.

[10] H. Diab, E. Abdennour, F. Kurdahi, *FIR Filter Mapping and Performance Analysis on Morphosys*, 7th IEEE International Conference Electronic, Circuits and Systems, Vol. 1, No. 1, 2000, pp. 99–102.

[11] Texas Instruments, *DSP TMS320C55X fixed-point digital signal processing data sheet*, Feb. 1999.

Index