University of Piraeus - Department of Informatics
Postgraduate Studies
«Advanced Information Systems»

**Master Thesis**

| | |
|---|---|
| Thesis | **Development of a soft error vulnerability analysis framework for FPGA devices** |
| Students Name | **Dimitrios Agiakatsikas** |
| Fathers | **Sarantos** |
| Number of Records | **MPSP/10064** |
| Supervisor | **Mihalis Psarakis, Assistant Professor** |

Delivery Date **10 / 2013**

**Committee**

| (signature) | (signature) | (signature) |
|---|---|---|
| M.Psarakis<br>Assistant Professor | A.Pikrakis<br>Lecturer | P.Kotzanikolaou<br>Lecturer |

## Abstract

As the features sizes of the FPGA devices are moving aggressively to the nanometer regime, the single-event upsets (SEUs) are expected to become a major reliability concern for the SRAM-based FPGAs. Given the limited information provided by the FPGA vendors about the susceptibility of the FPGA designs to soft errors, the research community requires SEU analysis tools to accommodate the development and assessment of SEU mitigation approaches. On the other hand, open-source CAD tools, such as RapidSmith [1] and Torc [2], have been recently proposed that target industrial FPGA architectures without escaping the boundaries of proprietary issues in contrast with the traditional open-source FPGA CAD tools. In this thesis, an open-source framework is presented for the soft error vulnerability analysis of Xilinx FPGA devices. The proposed framework will allow researchers to evaluate their reliability-aware CAD algorithms and estimate the soft error susceptibility of the designs at early stages of the implementation flow for the latest Xilinx architectures. Furthermore the well-known simulated-annealing placement algorithm is implemented in RapidSmith - where a limited random placer is currently supported - in order to evaluate the proposed post-placement sensitivity analysis method. To demonstrate the vulnerability analysis framework, a rich set of experiments is carried out. The thesis compares the soft error awareness of different packing/mapping tools (VTR and Xilinx tools) and different place tools (simulated annealing and Xilinx placers). The proposed method is evaluated by correlating its sensitivity analysis results with the Xilinx sensitivity report.

## Περίληψη

Καθώς το μέγεθος των χαρακτηριστικών των FPGA κινείται επιθετικά στην περιοχή των νανομέτρων, τα μεμονωμένα σφάλματα αναμένονται να γίνουν μείζον ανησυχία για την αξιοπιστία των SRAM FPGA. Λόγω των περιορισμένων πληροφοριών που παρέχουν οι κατασκευαστές των FPGA για την ευπάθεια των FPGA κυκλωμάτων σε παροδικά σφάλματα, η ακαδημαϊκή κοινότητα απαιτεί εργαλεία ανάλυσης των SEU, ώστε να αναπτυχθούν τεχνικές μετρίασης τους. Από την άλλη, πρόσφατα έχουν προταθεί εργαλεία CAD ανοιχτού λογισμικού, όπως το RapidSmith [1] και το Torc [2] που σε αντίθεση με κλασσικά εργαλεία CAD ανοιχτού λογισμικού, υποστηρίζουν πραγματικά FPGA χωρίς όμως να παραβιάζουν τα πνευματικά δικαιώματα των κατασκευαστών. Στην εργασία αυτή παρουσιάζουμε ένα πακέτο εργαλείων ανοιχτού κώδικα για την ανάλυση της ευπάθειας των Xilinx FPGA σε παροδικά σφάλματα. Το προτεινόμενο πακέτο εργαλείων θα επιτρέψει στους ερευνητές να αξιολογούν τους SEU αλγόριθμους αξιοπιστίας και να εκτιμούν την ευπάθεια των κυκλωμάτων σε παροδικά σφάλματα σε πρώιμα στάδια της υλοποίησης τους για τις πιο πρόσφατες αρχιτεκτονικές της Xilinx. Επίσης έχει αναπτυχθεί ο simulated-annealing αλγόριθμος τοποθέτησης στο περιβάλλον του Rapidsmith, όπου παρείχε μόνο ένα τοποθετητή τυχαίας επιλογής, ώστε να αξιολογηθεί η προτεινόμενη μέθοδος ανάλυσης της ευαισθησίας του κυκλώματος μετά την τοποθέτηση του. Για τη επίδειξη του πακέτου ανάλυσης της ευαισθησίας των κυκλωμάτων έχουν εκτελεστεί μία πλούσια πληθώρα πειραμάτων. Η εργασία συγκρίνει το πόσο προσεκτικά είναι διάφορα εργαλεία packing/mapping (VTR και εργαλεία της Xilinx) και διάφοροι τοποθετητές (simulated annealing και Xilinx τοποθετητές) στα παροδικά σφάλματα. Τα αποτελέσματα της ευαισθησίας από την προτεινόμενη μέθοδο έχουν αξιολογηθεί, συσχετίζοντας τα με τα αποτελέσματα από την αναφορά της ευαισθησίας από την Xilinx.

## Acknowledgments

I would like to express my gratitude to the persons who helped me through the realization of this thesis. First and foremost, I wish to thank my advisor professor Mihalis Psarakis, for his constant support, enthusiasm, patience and insights. He helped me come up with the thesis topic and guide me for almost a year of development. I would not be able to implement this framework and write this dissertation without his support. It was an honor for me to collaborate with such a brilliant person and scientist. I also own special thanks to Aitzan Sari, PhD candidate of my advisor. He passionately helped through the process of this framework. We spent together lot of days and nights implementing this framework. I will not forget the night he fell asleep while we were talking on skype and implanting the framework for hours. It was so funny. I will always be thankful to my best friends Dr.Grigorios Koulouras, who was my advisor back in the bachelor studies and Kyriakos Kontakos. Throughout these years, they have taught me topics of the cutting-edge technologies used in the embedded systems world. My sincere thanks goes to my bachelor fellow student Evangelos Tasoulas, who is one of my best friends and always supports me in the Linux environments. He is a guru in his field and I wish him to excel in his PhD at the University of Oslo. I also thank my friend Tasos Vereses, another Linux guru who always helps me in Linux topics. I thank Dr.Christos Evangelidis, Dr.Nikolaos Melis, Dr.Ioannis Kalogeras, professor Constantinos Nomicos and especially professor Nikolaos Thomaidis. They do not stop to support me and give me the most helpful advices for winning a place in the academic community. I also thank my current work, Geodynamics Institute of the National Observatory of Athens for its economic support while I was conducting this research. Last but not least, I would like to thank my dad, mom and little sister. My family has always been with me when I needed them, providing me emotional and economic support. I love them so much.

# Table of Contents

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 Motivation

Over the last two decades, the research community has made significant efforts trying to find fault tolerant techniques in order to keep Field Programmable Gate Arrays (FPGAs) operational in high radiation environments. Such hostile environments can be found in space e.g. avionics in spacecraft, high-energy physics experiments e.g. CERN and many others. Although FPGA vendors provide high availability and reliability devices, using them in mission-critical applications [1] (i.e. can cause an environmental catastrophe or affect a human life) is a big challenge, due to their susceptibility to soft-errors [2][3]. The designers not only have to develop qualified critical systems, but also have to keep the design cost low, forcing them to use commercial-off-the-shelf (COTS) FPGAs combined with emerging failure systems. FPGA vendors have introduced radiation-hardened devices (e.g. antifuse-based) to solve this problem. However, these devices are much more expensive less technologically developed than COTS FPGAs. For example an Airbus 380 has more than 700 antifuse-based FPGAs [4] (Actel SX-A family). The cost of 700 Actel SX-A FPGAs (if we consider that they have used the biggest FPGA in SX-A family, i.e. A54SX72A) is about 157000 US dollars (2013). If indeed, they used Xilinx COTS FPGAs with almost equal specifications (Virtex4 xc4vlx15) the cost would be almost the half, i.e. 84000 US dollars (2013).

Soft-errors have been a meaningful matter of the research community, since spacecraft electronics were affected from radiation in the early 1975s [5]. Spacecraft and airplane electronic systems have a variety of analog and digital components sensitive to radiation, making Single Event Upsets (SEUs) a major concern. SEUs are caused when charged particles (heavy ions and protons) hit a silicon atom transferring enough energy to produce a failure in the system. The amount of energy and the location of the strike in the device can cause transient or permanent errors. An SEU can produce transient soft errors in the combinational logic components, which can possibly be captured from Flip-Flops(FFs). Moreover, transient soft errors can directly affect the FFs of the FPGA and its hard block resources, such as RAM. Permanent failures are divided to hard errors or recoverable errors. Hard errors occur when charged particles bring on a lutch-up producing a short-cut between source and drain in mosfet technology that is commonly used in FPGA architectures. In case of recoverable errors, the configuration bitstream remains erroneous until it is downloaded again to the FPGA. These tradeoffs are a major problem in FPGA technology due to their high reliance on SRAM memory to store the configuration data [6].

Although, many fault tolerant techniques have been developed the last years, high reliability solutions are still a big challenge for the academic and industry research. FPGA vendors provide high capacity and performance devices while keeping the power consumption low. In order to develop FPGAs with these specifications, programmable logic industry uses silicon nanometer technologies and low operating voltages. However, shrinking of circuit dimensions to nanometer regime or shrinking noise margins [7], has revealed the susceptibility of the FPGA devices to emerging failure mechanisms raising several reliability issues [13]. Therefore, given that the feature sizes of the future beyond nanometer technologies will continue to shrink and the packaging cannot effectively shield the devices against SEUs [8], the implications caused by soft errors are expected to deteriorate drawing the attention of more researchers and practitioners from both domains of fault-tolerant computing and FPGA design automation. For the development and assessment of SEU mitigation methodologies and SEU-aware CAD tools for FPGAs, the research community needs the existence of soft error analysis tools able to measure the vulnerability of the designs and provide useful insights.

Recently, the research community provided open-source CAD tools that support commercial complex FPGA devices [9][10]. Motivated by this work, this thesis aims to provide a collection of open-source tools for the vulnerability analysis of Xilinx FPGA devices. The proposed framework will benefit the upcoming research providing valuable feedback to SEU mitigation approaches about the sensitivity[1] of

[1] Configuration bits are categorized into *sensitive* and *non-sensitive* bits depending on the impact of soft errors to the circuit behavior [16], [47]. When a soft error in a configuration bit affects the circuit operation the bit is classified as sensitive (or essential according to Xilinx terminology) for the particular implementation, otherwise as non-sensitive. The actual failure rate of an FPGA design depends on the number of sensitive configuration bits or in other words the dynamic cross section of the design.

the FGPA configuration bits or the criticality of the design modules. For example, a TMR methodology could take advantage of the criticality analysis to reduce the area overhead by applying selectively the redundancy technique to the design modules [11]. Moreover, being able to estimate the soft error susceptibility of the design at various stages of the FPGA implementation flow, the proposed framework could be used for the development of SEU-aware PaR algorithms.

Several approaches have analyzed in the past the vulnerability of SRAM-based FPGAs into soft errors. These approaches are based either on fault injection experiments [6], [12],[13], [14], [15] or analytical methods [6],[16], [17], [18] to measure the sensitive configuration bits. The experimental approaches inject soft errors in the configuration bitstream of the design under test and, hence, they cannot apply during the FPGA design flow in order to provide an early sensitivity estimation. On the other hand, most research groups that have proposed analytical methods have developed proprietary vulnerability analysis tools targeting specific FPGA architectures which cannot be easily reproduced for another FPGA family. Furthermore, recent approaches have proposed SEU-aware placement and routing algorithms [8], [18], [19], [20], [21] in order to reduce the dynamic cross section of the FPGA designs. However, almost all these approaches have been demonstrated on the commonly used, academic VPR tool targeting virtual FPGA architectures. The proposed framework will enable the evaluation of such reliability-aware algorithms for off-the-shelf FPGA devices. The soft error vulnerability analysis framework is based on the recently proposed FPGA CAD platform, RapidSmith [9]. RapidSmith is a set of tools and APIs written in Java that manipulates a Xilinx human readable file format (XDL) and enables researchers to develop tools for the packing, placement, and routing of FPGA designs and parse/export configuration bitstreams. The proposed framework:

- evaluates the vulnerability of FPGA designs to soft errors analyzing the sensitivity of the configuration bitstreams. It classifies the sensitive bits according to their configuration type: block configuration bits (CLBs, IOBs, DSPs, etc.) and interconnection configuration bits.
- estimates the vulnerability of FPGA designs to soft errors at early phases of the FPGA implementation flow. In particular, it supports post-mapping analysis of the sensitive block configuration bits, post-placement analysis of the sensitive interconnection bits and final (post-routing) analysis of the total sensitive configuration bits. For the estimation of the sensitive configuration bits, the following methods are combined: theoretical analysis of the structure of Xilinx Virtex-5 programmable resources, analytical methods previously proposed in the literature [6], [18] for the estimation of sensitive interconnection bits and extraction of related information from the Xilinx sensitivity analysis results[2].
- visualizes the sensitive configuration bits in the FPGA physical layout taking advantage of RapidSmith APIs. The sensitivity bitmap of the proposed method is compared with the sensitivity bitmap of Xilinx analysis.

## 1.2 Observed soft-errors failures in space missions

Before continuing with the rest of this thesis it is worth to describe some observed failures in spacecraft electronic systems caused by cosmic radiation [22]. Back in 1989 Galileo mission was launched on a planetary exploration mission to Jupiter. All its electronic parts were fully tested and were radiation hardened with system-level redundancy and error detection capabilities. Despite the soft-errors mitigation techniques, safe holds failures where observed on some analog switches which fortunately did not have impact on the mission. These failures were believed to be due to SEUs. Another failure was experienced at the TOPEX/Poseidon mission, launched on August 1992. Proton radiation affected some 4N49 optocouplers of vendor Texas Instruments causing failures on some status signals and the circuits of the thruster command. On December 4, 1996, Cassini mission was lunched. Instead of a mechanical tape recorder they used a solid state recorder (SSR) constructed with high density RAM. Despite the single-bit correction and double-bit correction circuits it had, SSR experienced single-bit errors. On October 24, 1998, Deep Space 1 mission was lunched. A SEU caused failure in a FPGA due to a latch-up, while the recovery time of the system required 28 minutes which was an unexpected long time. Furthermore, on

[2] FPGA vendors provide utilities for the sensitivity analysis of the configuration bitstream. The Xilinx tool generates a map file (.ebd for essential bit description) for the characterization of the essential (sensitive) bits of the configuration bitstream. However, since it is not feasible to decode the raw bitstream data due to proprietary issues these sensitivity analysis results cannot provide an in-depth sensitivity analysis of the circuit.

April 24, 2001, Odyssey mission went into safe mode after 2 weeks in space due to failures in RAM, caused by a cosmic ray ion.

As FPGAs become more dencer and more powerful, they offer to the designers the integration of high-availability systems with hard real-time performance into a FPGA, such as avionics in airplanes and spaceships or medical systems. This raises need of new inventions from the academic and industrial research community to provide more robust SEU mitigation solutions that meet system reliability goals [23].

## 1.3 Thesis Structure

The remainder of this master thesis is organized as follows. In chapter 2 the related work and literature is collected giving a better understanding in chapter 3, where information about the implemented soft-error vulnerability framework is provided. In more detail, chapter 3 gives an explanation of the methods used to evaluate the soft-error estimation tools. Furthermore, a description of the framework packages is provided. Experimental results are presented in chapter 4, and last come the conclusions of this master thesis.

# 2 Background and literature

This chapter contains the background information about the fundaments of modern FPGA architectures, the state-of-the-art CAD technology used in nowadays FPGAs and finally the related work section that presents the mitigating soft-errors techniques for SRAM based FPGAs, proposed from the research community.

## 2.2 What is an FPGA

Field is the key word for defying a Field Programmable Gate Array (FPGA). FPGAs are Intergraded Circuits (ICs) that can be programmed in the field after manufacture. This means that FPGAs vendors must have some pre-fabricated digital circuity in the chip, enabling it to implement any given digital function from the user, simply by being appropriately programmed.

As described in [24], *FPGAs are semiconductor devices that are based around a matrix of configurable logic blocks or Cluster of Logic Blocks (CLBs), connected via programmable interconnects* (illustrated in Figure 2). *FPGAs can be programmed to desired application or functionality requirements after manufacturing. This feature distinguishes from Application-specific integrated circuits (ASICs), which are custom manufactured for specific design tasks. Although One-Time Programmable (OTP) FPGAs are available, the dominant types are Static Random Access Memory (SRAM) based which can be re-programmed as the design evolves.*

This innovative idea of a FPGA that would reduce the manufacturing time and cost of an IC from months to hours was introduced back in 1986 [25]. As a result, FPGAs are very popular these days, since faster design of complex products can be achieved, in contrast with ASICs. Because FPGAs are software configured, modifying a design is very fast, less risky and can be made in some hours, rather than months that is required for ASIC prototypes to be manufactured. FPGAs provide lower non-recurring engineering (NRE) costs, faster time to market and no expensive penalties at the verification phase [26] than ASICs. FPGAs are suitable for rapid prototype design, specialized digital systems (i.e. reconfigurable designs, System-On-Chip (SOC) designs) and low-volume IC production. Custom ASIC design is commonly used for high volume production, while Standard-Cell ASIC design for middle-volume production. Figure 1 illustrates the total cost as a function of IC parts of a design implemented in a FPGA, a Standard-Cell ASIC and custom ASIC device.

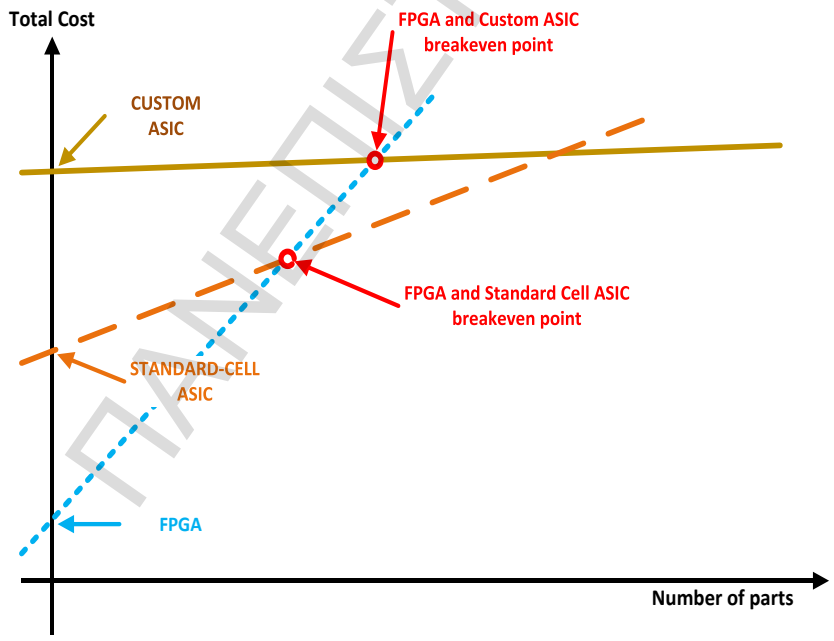


**Figure 1 - Cost vs. Volume.**

## 2.3 FPGA Architecture

Traditionally, an FPGA can be composed of three fundamental elements depicted in Figure 2:

- The Logic Blocks (LBs), named as CLBs in Xilinx terminology. The LBs contain a group of Block Logic Elements (BLEs) named as Slices in Xilinx terminology, which implement combinational and sequential logic.
- The Input-Output Blocks (IOBs).
- The Programmable Routing, which is a matrix of wires that interconnect the LBs and the IOBs via Connection Blocks/Boxes (CBs) and Switch Blocks/Boxes (SBs). More detail is provided in the programmable routing section.

Commercial FPGAs also include extra memory, multipliers, memory controllers, high speed IOBs, Digital signal processing (DSP), Phase Locked Loops (PLLs), clock management, even embedded processors. These special blocks are referred in literature as hard blocks giving more logic utilization and speed at a FPGA.

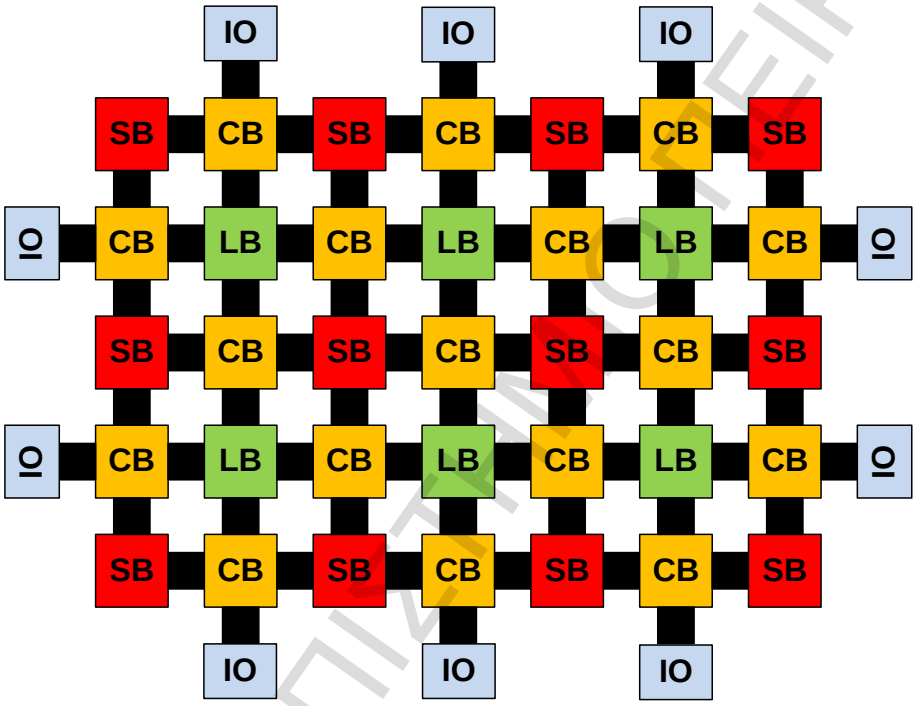


**Figure 2 – FPGA architecture.**

### 2.3.1 Basic Logic Element (BLE)

The BLE is built from the following components depicted in Figure 4:

- A Look-Up Table (LUT) to implement combinational logic.
- A Flip-Flop (FF) providing sequential behavior.
- A multiplexer for bypassing the FF if only combinational logic is needed.

Figure 3 depicts the LUT which is the well-known truth table from digital design. A K-input LUT is typically composed from $\mathbf{2^k}$ SRAM that holds the configuration memory (LUT-mask) and $K-1$ multiplexers implemented as a tree to select the bit from SRAM and pass it to the LUT output. To simplify, we give an example of a LUT (illustrated in Figure 4) which can implement any combinational function of 4-inputs (A, B, C, D). It has a 4-inputs, 16bit SRAM ($\mathbf{2^4 = 16bit}$) and 15 x 2:1 multiplexers. Programming the LUT-MASK with the appropriate bit will assemble the desired function [27].

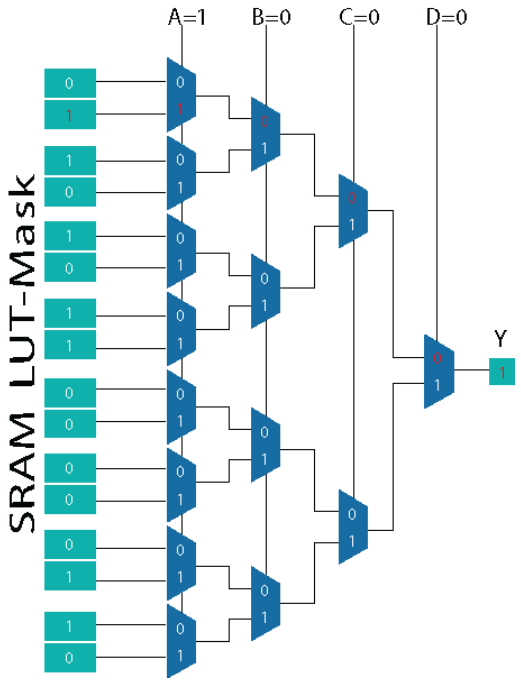


**Figure 3 - Look Up Table (LUT).**

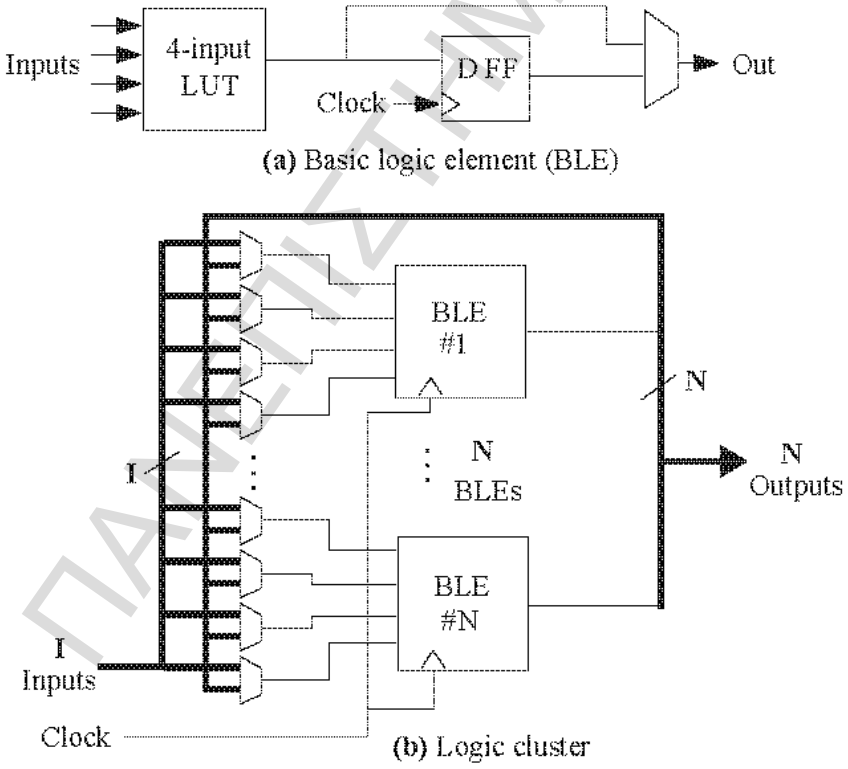


**Figure 4 - Structure of basic BLE and LB** [28].

### 2.3.2 Logic Block (LB)

Typically, commercial FPGAs have groups/clusters of BLEs that contain LUTs and FFs. Each LE has size N, while it consists of N interconnected BLEs as shown in Figure 4. Furthermore, a LB has ***I*** external pins that are passed via multiplexers to the BLEs inputs. In addition, the multiplexers provide the flexibility of interconnecting the BLEs in the same block/cluster. The ***N*** output pins of the LB are connected to the routing resources via the CBs.

### 2.3.3 Programmable Routing

Figure 5 depicts the routing resources. As high resource utilization and need for high bandwidth is necessary in FPGA designs, the routing resources occupy the largest area of silicon in a FPGA. The routing resources are constructed by three fundamental components [29].

- **Switch Blocks/Boxes (SBs):** SB is a hub that programmable connects horizontal and vertical metal lines of the routing channels. This is done via Programmable Interconnection Points (PIPs). Its flexibility depends on the property ***Fs,*** which defines the number of connection wiring segments it can handle.
- **Connection Blocks/Boxes (CBs):** CB is a switch in between the LB and the SB. It is responsible to connect the input and output pins of the LB with the routing wires of the SB. It has a property ***Fc*** describing the number of wires a LB pin can handle.
- **Routing channels:** Routing channels are horizontal and vertical metal wires that span between LBs, IOBs and Hard Blocks. In order to route the design, the appropriate PIPs must be programmed to interconnect the pins of the LBs, IOBs and Hard Blocks with the available metal wires. The amount of wires in a routing channel is referred in literature as size/width and is represented with the letter ***W***.

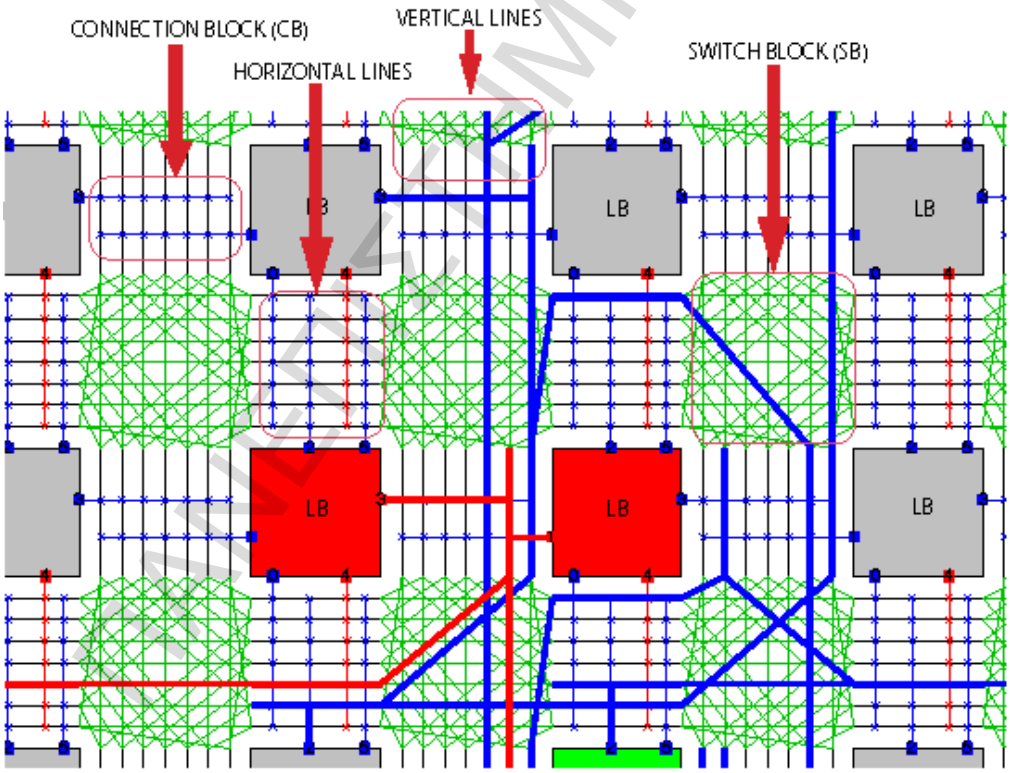


**Figure 5 - Routing Resources.**

## 2.4 FPGA Design Automation

Implementing a circuit in a modern FPGA is a big challenge, requiring millions of configuration bits to be set on proper state, high or low. Although in 1960s, IC designs were hand-drawn, nowadays the complexity of FPGA architectures clearly prohibits those design procedures. Indeed, circuits are described in higher abstraction languages referenced as hardware description languages (e.g. Verilog and VHDL) and then converted from CAD tools into FPGA configuration bitstreams, which specify the state of every bit in the FPGA in order to assemble the described circuit. A way to keep the complexity of this problem low is to break it into some sub-problems. In the following sections a description of the sequential stages (depicted in Figure 6) that are involved in the procedure of mapping a circuit into a FPGA is presented.

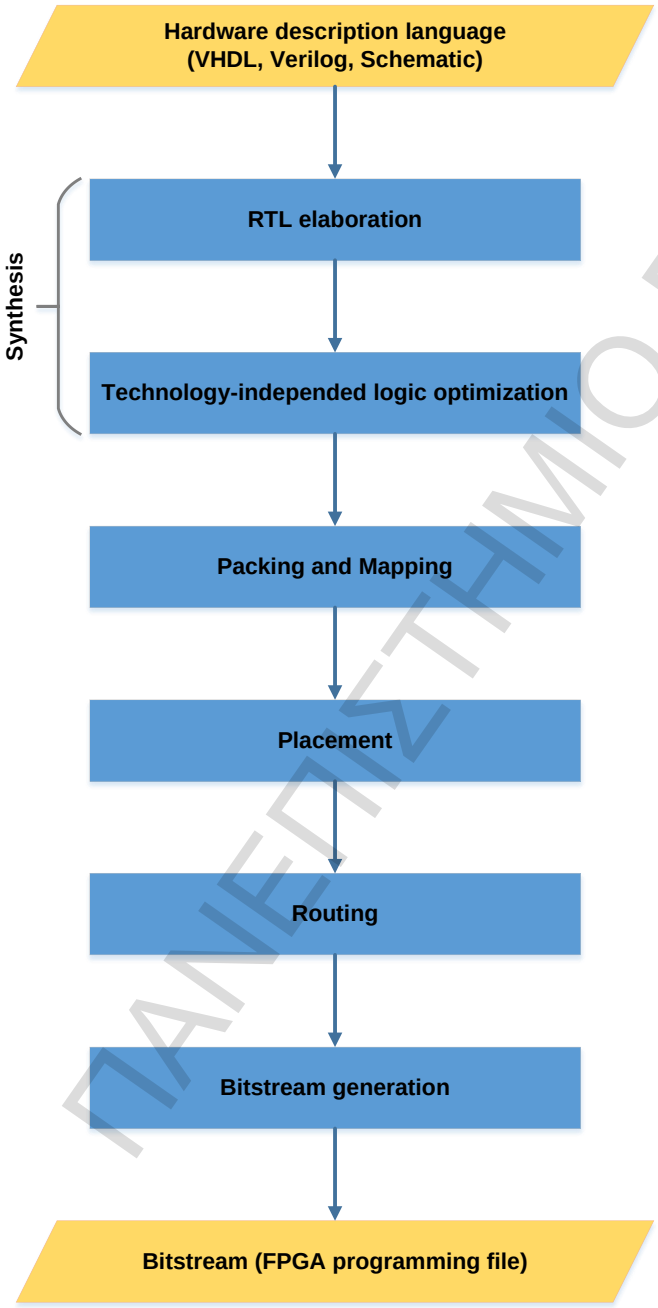


**Figure 6 - FPGA CAD flow.**

### 2.4.1 Synthesis

The synthesis procedure is a very complex task. In this stage the synthesis tool performs two steps. First, it converts the hardware description language into a neltlist of gates and second converts the neltlist of gates into a netlist of FPGA logic blocks, while trying to optimize the area and performance of the design. In more detail, the synthesis tool executes the following two sub-stages:

- RTL elaboration. This converts the hardware description language into gates, implementing datapath operations (such as additions and multiplications) and control logic (such as a set of finite-state machines or Boolean networks). It must be mentioned that in this sub-stage the synthesizer recognizes functions that can be handled by hard blocks (e.g. DSP) in the FPGA and forwards them to the packing and mapping stage. These functions are referenced as black-boxes.
- Technology-independed logic optimization. This optimizes both datapath and control logic, independed of the targeting FPGA architecture. There is a rich set of optimization techniques, which are performed in this sub-stage, such as removing redundant logic or sharing resources, don't care based optimization and many others.

### 2.4.2 Packing and Mapping

In this stage the packer packs several LUTs and Flip flops into one LB, respecting the targeting architectural limitations, such as the maximum number of LUTs and FFs the LB may contain and its available inputs and outputs. Afterwards, the packed instances of the design are technology mapped into the available LBs and also the black box functions into the available FPGA hard blocks. The optimization goals in this stage are to pack each LB to its maximum capacity, therefore maximize the FPGA resource utilization and also attempt to minimize the inputs and outputs signals of the LB, in order to route effectively the design in the routing stage.

### 2.4.3 Placement

As described in [30], *placement algorithms determine which logic block within an FPGA should implement each of the logic blocks required by the circuit. The optimization goals are to place connected logic blocks close together to minimize the required wiring (wire-length-driven-placement), and sometimes to place blocks to balance the wiring density across the FPGA (routability-driven-placement) or to minimize circuit speed (timing-driven placement).*

In literature there are four different categories of placement methods for FPGA CAD tools [31]:

- simulated annealing
- min-cut
- quadratic
- parallel

This thesis will focus on the well-known simulated annealing (SA) placement algorithm with adaptive schedule [30][32], as it is implemented it in the Rapidsmith framework, in order to replace the uncompleted placer package provided by Rapidsmith and evaluate the proposed SEU vulnerability framework. The main reason this algorithm has been chosen, is that its cost function can be easily modified from researchers to implement an SEU-aware placement algorithm [8], [18], [20]. SA placers mimic the natural process of a metal to be easily shaped in high temperatures while local improvement of the shape can be made as its temperature decreases. A linear congestion cost function is determined to calculate the quality of the placement. A wire-length cost function $\mathbf{C}$ is selected to be used, which is the summary of all nets costs of a design. The cost of a net is the half perimeter of the bounding box (Figure 7) that encapsulates all the attached logic blocks to the net and is given from the following equation:

$$\mathbf{N_{cost}} = \mathbf{q(i)} * \left( \frac{\mathbf{bb_x(i)}}{\mathbf{c_{av,x}(i)^{\beta}}} + \frac{\mathbf{bb_y(i)}}{\mathbf{c_{av,y}(i)^{\beta}}} \right) \quad (1)$$

where $\mathbf{q(i)}$ is a constant value ranging from 1 for a net with less than 3 terminals, to its maximum value 2.73 for more than 50 terminals. The $\mathbf{c_{av,x}}$ and $\mathbf{c_{av,y}}$ are the average horizontal and vertical routing tracks respectively, trying to minimize the routing congestions of overloaded switch matrixes. The exponent $\boldsymbol{\beta}$

has default value 1 and allows the relative cost of using narrow and wide routing channels to be adjusted. When giving a larger value to **β** parameter, more wiring in narrow channels is penalized. Setting **β** to zero reverts the linear congestion cost function to standard bounding box cost function, forcing to a more shrinked placement. In general the SA algorithm tries to swap random LBs in the FPGA pre-fabricated circuitry and accepts the swap if the design cost decreases. This would eventually produce a high quality placement. The SA algorithm executes the following steps:

- An initial placement (without taking in account the design cost) of the netlist logic blocks (LBs) is performed, assigning all LBs to the available and compatible resources in the FPGA fabric.
- The initial parameters of the SA algorithm are calculated. These include temperature **T** that controls the probability of accepting a random swap, **m** which is the number of swaps that will be executed in every temperature and finally **Rlimit** which determines how close must be the LBs for swapping. This leads to a procedure where random LBs for swap are selected from the entire area of the FPGA and as the temperature decreases only close LBs are selected for swapping.
- A large number of swaps are then made to gradually improve the placement quality. At high temperatures almost all swaps are accepted and as the temperature drops the acceptance probability decreases.
- The placement algorithm terminates when the temperature drops under a threshold.

In more details, the algorithm consists of two nested loops as depicted in Figure 8. The inner loop is executed **m** times and swaps two random LBs in the **Rlimit** area at every step. A swap is accepted when cost **C** is decreased. However, there is chance to accept the move, even if cost **C** is increased. This probability is computed by the equation $\mathbf{r} < \mathbf{e}^{-\frac{\Delta C}{T}}$ , where $\Delta\mathbf{C} = \mathbf{C_{new}} - \mathbf{C_{old}}$ and **r** is a random number ranging from 0 to 1. This feature gives the ability to escape local minima. The outer loop updates **T**, **m** and the **Rlimit** parameters. Finally the placement terminates when $\mathbf{T} < \boldsymbol{\varepsilon} * \frac{\mathbf{C}}{\mathbf{N_{nets}}}$, where **ε** is a constant ranging from 0.005 to 0.05 and $\mathbf{N_{nets}}$ is the amount of nets contained in the design.

The proposed SA placer supports the movement of all programmable resources, (e.g. logic slices, DSP slices, IOBs, BRAMs) but it is not capable to handle carry chains. Thus, someone should use either benchmarks without carry chains or the proposed SA placer will not move the tiles that contain instances with curry chains.

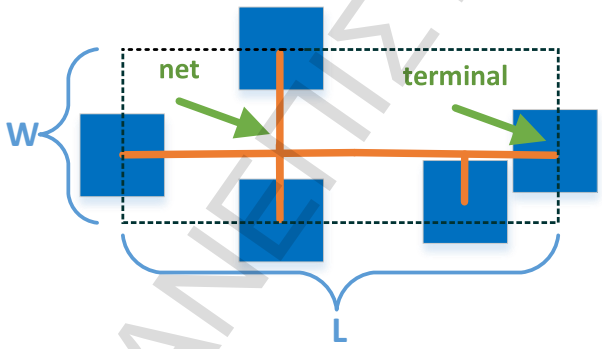


**Figure 7 - Bounding Box**

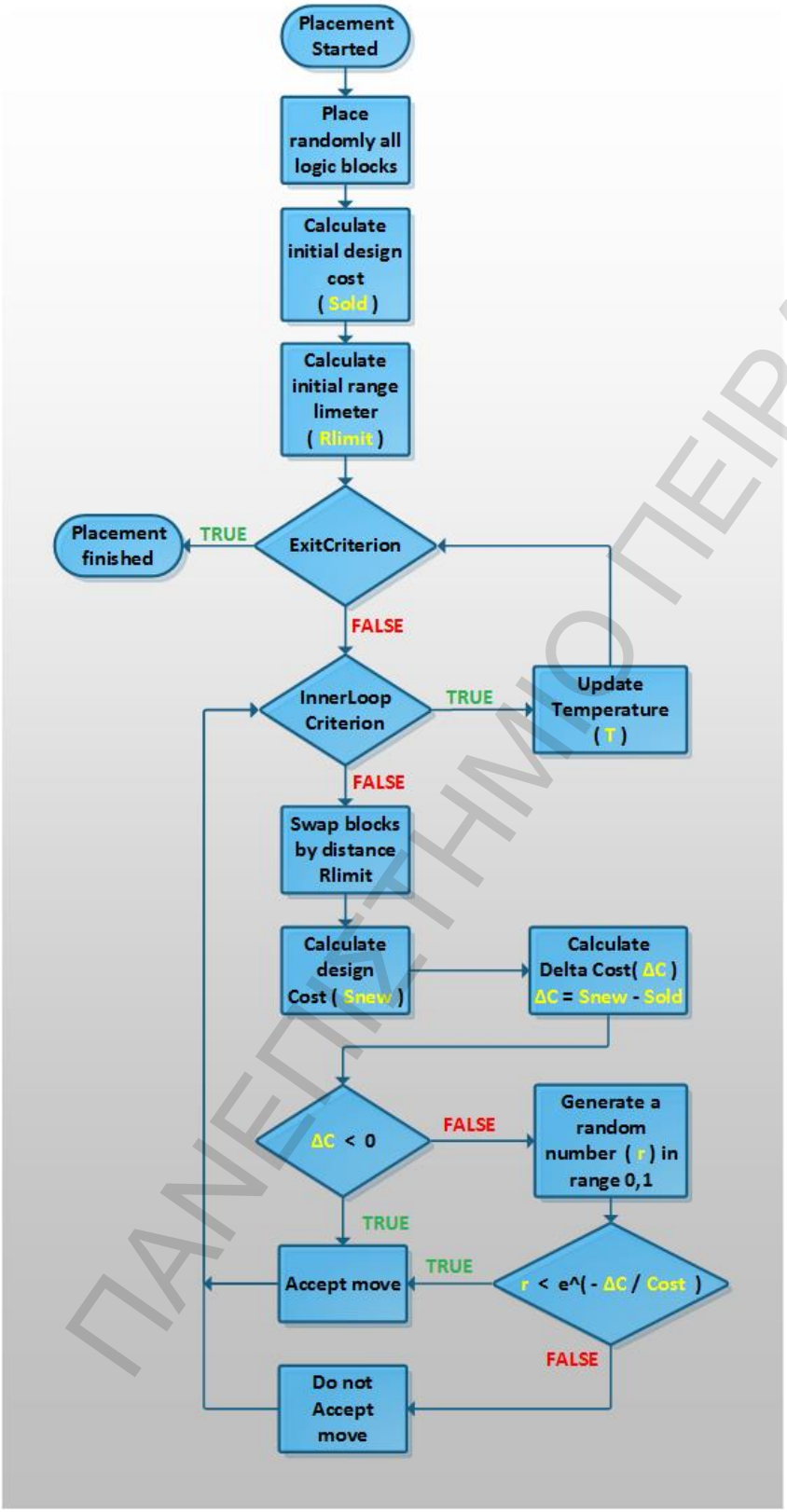




**Figure 8 - Adaptive Simulated Annealing**

### 2.4.4 Routing

In the previous stage the placer tried to choose the correct LB locations to place the instances with the aim that the router would need less effort to route the design. The router enables the required programmable interconnection points (PIPs) in order to connect effectively all LB input and output pins described in the design. Designed to be able to route successfully a design, a CAD flow must have an equal skillful placer and router. If the routing algorithm has excellent performance while the placement algorithm does not, it is obvious that the routing will fail and vice versa. Sometimes the placement and routing is performed simultaneously, as the cost of a design in the placement procedure is inherently weak at addressing both wirability and timing optimizations [33].

In general the routing architecture is represented as a directed graph. Each pin and wire of the design becomes a node and each potential connection an edge. The router algorithm has to find and connect all the nodes in the routing resource graph. There are two main router algorithms in the FPGA CAD technology. The routability-driven that try to find the shortest path and the time-driven which use more complex algorithms, giving priority mostly to the critical paths of the design.

### 2.4.5 Bitstream generation

The bitstream generation is the conversion of a routed design file into a sequence of bits, called bitstream. The bitstream is uploaded to the FPGA in order to configure every bit state in it. This will eventually produce the actual hardware.

## 2.5 The Rapidsmith framework

Rapidsmith is a set of APIs written in Java that read, manipulate and write the Xilinx human readable file format (XDL), whereas hiding syntactic details from the user. Rapidsmith gives the ability to researchers to try out new ideas in all fields of FPGA CAD tools on Xilinx FPGAs. A design must be converted first in the appropriate XLD file format in order to be imported to Rapidsmith. Rapid development of packing, placement or routing tools and parse/export configuration bitstreams can be achieved taking the advantage of the available APIs provided by the framework. It is argued that Java is a slow and memory-hungry programming language, in contrast with native machine languages like C, but no speed or memory issues where observed, while developing the proposed soft error vulnerability tools. Java is a free object-oriented programming language with useful libraries for big data structures which makes it very powerful for developing CAD tools. The researchers can focus at their algorithms implementation, while time-consuming memory management is taken care from the Java garbage collector, cleaning up the unused objects without big performance tradeoffs. In order to develop a tool in Rapidsmith someone must understand the basic syntax of XDL files. Therefore, in the following sections a description of the the Xilinx XDL file structure will be provided, over some examples.

### 2.5.1 The XDL file

Xilinx vendor provides the Xilinx Design Language (XDL) to interface and access the features of a design or a device. XDL has two main sides. One side is the description of the FPGA architecture, providing all the primitives and routing fabric of the FPGA. The other side is the description of the design. XDL is a human readable ASCII file, offering a representation of the proprietary Xilinx Netlist Circuit Description (NCD) file format in every CAD stage (i.e. mapped, placed or routed design). The user is free to insert or extract any desired information at different CAD stages of the Xilinx design flow. XDL can describe the following designs:

- A partially or full Mapped design.
- A partially or full Placed design.
- A partially or full Routed design.

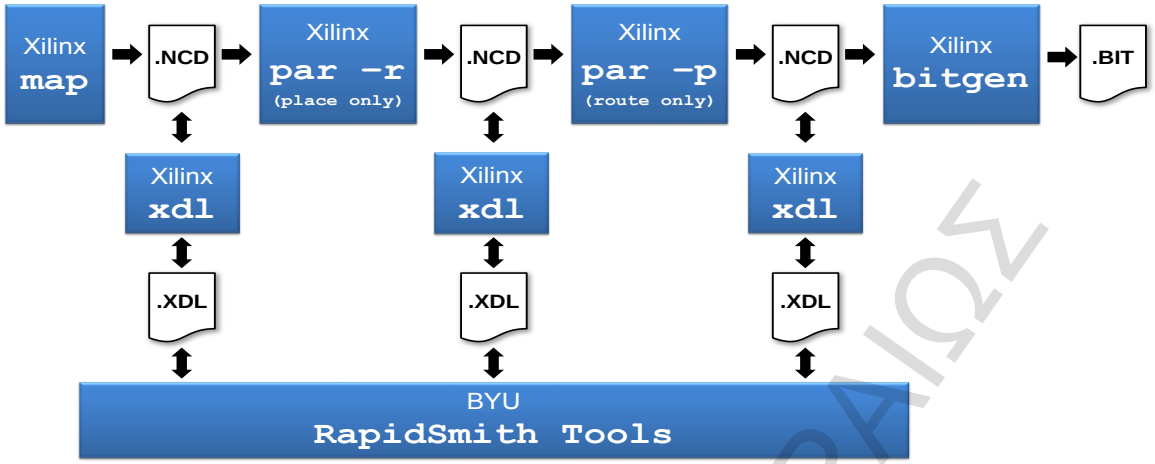


**Figure 9 - Block diagram of where XDL fits in the CAD flow** [9].

### 2.5.2 XDL Syntax

The XDL is a self a documented file format explaining every statement with an example comment. In order to understand the XDL syntax we will examine a design with two instances, while it is mapped, placed and routed.

**Design statement**

Every XDL file has a design statement which includes the name of the design, the part number of the FPGA and also a list with some attributes of the CAD tools. Below in Figure 10 we can see an example of the design statement with its attributes list. We observe that the name of the design is "lut" and the targeting FPGA is the xc5vlx110tff1136-1.

```
# XDL NCD CONVERSION MODE $Revision: 1.01$

# time: Sun Jul 14 22:26:07 2013

# The syntax for the design statement is:

# design <design_name> <part> <ncd version>;

# or

# design <design_name> <device> <package> <speed> <ncd_version>

# ======================================================

design "lut" xc5vlx110tff1136-1 v3.2 ,

  cfg "

      _DESIGN_PROP:P3_PLACE_OPTIONS:EFFORT_LEVEL:high

      _DESIGN_PROP::P3_PLACED:

      _DESIGN_PROP::P3_PLACE_OPTIONS:

      _DESIGN_PROP::PK_NGMTIMESTAMP:1373612597";
```

**Figure 10 - XDL design statement example**

**Instance statement**

An instance is represented in Rapidsmith with the design.Instance class. It always begins with the keyword “inst”. An instance can be placed or unplaced in the FPGA layout. The configurations of the instance are provided by a string that starts with the keyword “cfg”. It must be mentioned that the name of the instance must be unique in a design in order to avoid name conflicts in Rapidsmith. Figure 11 depicts an example with two instances in a mapped design. The name of the first instance is *out0_OBUF*. The second instance is *out0*. The two instances are not placed, while in the next CAD stage these instances have to be placed in a compatible Primitive site of *SLICEM* and *IOB* type, respectively.

```
inst "out0_OBUF" "SLICEM",unplaced  ,

  cfg " A5LUT::#OFF A5RAMMODE::#OFF A6LUT:LUT_U0:#LUT:O6=(A1*(~A2*(~A3*(~A4*(~A5*~A6)))))

      _BEL_PROP::A6LUT:BEL:A6LUT A6RAMMODE::#OFF ACY0::#OFF ADI1MUX::#OFF

      AFF::#OFF AFFINIT::#OFF AFFMUX::#OFF AFFSR::#OFF AOUTMUX::#OFF AUSED::0

      B5LUT::#OFF B5RAMMODE::#OFF B6LUT::#OFF B6RAMMODE::#OFF BCY0::#OFF

      BDI1MUX::#OFF BFF::#OFF BFFINIT::#OFF BFFMUX::#OFF BFFSR::#OFF BOUTMUX::#OFF

      BUSED::#OFF C5LUT::#OFF C5RAMMODE::#OFF C6LUT::#OFF C6RAMMODE::#OFF

      CCY0::#OFF CDI1MUX::#OFF CEUSED::#OFF CFF::#OFF CFFINIT::#OFF CFFMUX::#OFF

      CFFSR::#OFF CLKINV::#OFF COUTMUX::#OFF COUTUSED::#OFF CUSED::#OFF

      D5LUT::#OFF D5RAMMODE::#OFF D6LUT::#OFF D6RAMMODE::#OFF DCY0::#OFF

      DFF::#OFF DFFINIT::#OFF DFFMUX::#OFF DFFSR::#OFF DOUTMUX::#OFF DUSED::#OFF

      PRECYINIT::#OFF REVUSED::#OFF SRUSED::#OFF SYNC_ATTR::#OFF WA7USED::#OFF

      WA8USED::#OFF WEMUX::#OFF "
inst "out0" "IOB",unplaced  ,

  cfg " DIFFI_INUSED::#OFF DIFF_TERM::#OFF IMUX::#OFF OUSED::0 PADOUTUSED::#OFF

      PULLTYPE::#OFF TUSED::#OFF OUTBUF:out0_OBUF: PAD:out0:

        DRIVE::12  OSTANDARD::LVCMOS33  SLEW::SLOW "
```

**Figure 11 - Unplaced instances in the XDL file.**

We continue with the same instances being placed. We observe that the instance *out0_OBUF* is placed in the primitive site *SLICE_X0Y0*, while the slice is located in the primitive tile *CLBLM_X1Y0*. Furthermore, the instance *out0* is placed in the primitive site *LIOB_X0Y1* which is located in the primitive tile *AP21*.

```
inst "out0_OBUF" "SLICEM",placed CLBLM_X1Y0 SLICE_X0Y0  ,

  cfg " A5LUT::#OFF A5RAMMODE::#OFF A6LUT:LUT_U0:#LUT:O6=(A1*(~A2*(~A3*(~A4*(~A5*~A6)))))

      _BEL_PROP::A6LUT:BEL:A6LUT A6RAMMODE::#OFF ACY0::#OFF ADI1MUX::#OFF

      AFF::#OFF AFFINIT::#OFF AFFMUX::#OFF AFFSR::#OFF AOUTMUX::#OFF AUSED::0
```

```
      B5LUT::#OFF B5RAMMODE::#OFF B6LUT::#OFF B6RAMMODE::#OFF BCY0::#OFF

      BDI1MUX::#OFF BFF::#OFF BFFINIT::#OFF BFFMUX::#OFF BFFSR::#OFF BOUTMUX::#OFF

      BUSED::#OFF C5LUT::#OFF C5RAMMODE::#OFF C6LUT::#OFF C6RAMMODE::#OFF

      CCY0::#OFF CDI1MUX::#OFF CEUSED::#OFF CFF::#OFF CFFINIT::#OFF CFFMUX::#OFF

      CFFSR::#OFF CLKINV::#OFF COUTMUX::#OFF COUTUSED::#OFF CUSED::#OFF

      D5LUT::#OFF D5RAMMODE::#OFF D6LUT::#OFF D6RAMMODE::#OFF DCY0::#OFF

      DFF::#OFF DFFINIT::#OFF DFFMUX::#OFF DFFSR::#OFF DOUTMUX::#OFF DUSED::#OFF

      PRECYINIT::#OFF REVUSED::#OFF SRUSED::#OFF SYNC_ATTR::#OFF WA7USED::#OFF

      WA8USED::#OFF WEMUX::#OFF "
inst "out0" "IOB",placed LIOB_X0Y1 AP21  ,
 cfg " DIFFI_INUSED::#OFF DIFF_TERM::#OFF IMUX::#OFF OUSED::0 PADOUTUSED::#OFF

      PULLTYPE::#OFF TUSED::#OFF OUTBUF:out0_OBUF: PAD:out0:

        DRIVE::12  OSTANDARD::LVCMOS33  SLEW::SLOW "
```

**Figure 12 -Placed instances in the XDL file.**

**Net statement**

Moreover, in Figure 14 the above instances have been routed. The router tries to connect the outpins (sources) and the inpins (sinks) of the design primitive sites which are described with the net statement. The net statement is represented in Rapidsmith with the design.Net class. Nets have 3 different types: VCC, GND and WIRE. The keyword WIRE is the default type and is not required to be present in the XDL file. Nets are described with two components: The pins and the PIPs. The pips are only available when the design has been routed. As an example, Figure 13 depicts an unrouted net. Pins define the source and one or more sinks within the net. A pin is identified by the name of the instance it resides in and also with its internal name within the instance. Pips are represented with the design.PIP class in Rapidsmith and are used to connect the sources and sinks of the design using the prefabricated wires located in the matrixes. A PIP is uniquely described with the name of the tile that it resides followed with the internal coordinates that indicate the location of the PIP in the tile. Furthermore, a PIP has two wires with a connection between them. Figure 13 describes the unrouted net, while Figure 14 describes the same net, while it is routed. In the unrouted net example we can clearly see that the signal from output pin of instance '*out0_OBUF*' must be connected to the input pin of instance '*out0*'. In the routed net example, PIPS are added to the XDL file to describe the connection of the sources and the sinks of the two instances. Almost all PIPS are unidirectional and are described with the symbol ("->"). In some cases someone can find some bidirectional PIPS ("-=") used with long lines, which route the global nets of the design. However, Rapidsmith does not use this description, avoiding problems that can be caused by the XDL conversion.

```
net "out0_OBUF" ,
 outpin "out0_OBUF" A ,
  inpin "out0" O ,
```

**Figure 13 - Unrouted nets in the XDL file.**

```
net "out0_OBUF" ,

  outpin "out0_OBUF" A ,

  inpin "out0" O ,

  pip CLBLM_X1Y0 M_A -> SITE_LOGIC_OUTS12 ,

  pip INT_X0Y0 WN2END_S0 -> NW2BEG2 ,

  pip INT_X0Y1 NW2MID2 -> IMUX_B41 ,

  pip INT_X1Y0 LOGIC_OUTS12 -> WN2BEG0 ,

  pip IOI_X0Y1 IOI_IMUX_B41 -> IOI_O11 ,

  pip IOI_X0Y1 IOI_O11 -> IOI_O_PINWIRE1 , # _ROUTETHROUGH:D1:OQ
"XDL_DUMMY_IOI_X0Y1_OLOGIC_X0Y2" D1 -> OQ

  pip IOI_X0Y1 IOI_O_PINWIRE1 -> IOI_O1 ,

  ;
```

**Figure 14 - Routed nets in the XDL file.**

### 2.5.3 XDLRC Files

XDLRC report files are generated by the Xilinx XDL command line tool, i.e. xdl –report –pips –all_conns <partName>, describing the architecture of the corresponding FPGA device. The size of the generated files varies from some megabytes to some gigabytes for recent devices. Although there are only few different types of tiles in a Xilinx FPGA, the Xilinx resource descriptions files are gigantic and cannot be easily manipulated. The reason is that the same attributes for similar tiles is repeated in the XDLRC files. However, Rapidsmith uses a custom form of serialization and compression libraries to compress these files. For example the XDLRC report file of a Virtex 7 is compressed to 5965KB form an initial size of 73.6GB.

In order to describe the XDLRC report files, the general architecture description of a Xilinx FPGA. Xilinx is recalled. FPGAs consist of an array of tiles. The most frequent tile found in a Xilinx FPGA is the Configurable Logic Block (CLB) and the interconnection blocks (connection matrix and switch matrix). The CLB consists of two slices containing the look-up-tables and the flip-flops implementing combinational or sequential logic. Every CLB has two interconnection tiles on its left side. The first interconnection tile is called connection matrix and the second switch matrix. The switch matrix is a tile with PIPS that connects the horizontal and vertical wires of the routing channels. The connection matrix is a tile which is responsible to connect the input and output pins of the slices (contained in the CLBs) with the routing lines of the switch matrix. Figure 15 depicts a CLB with its interconnections tiles. Besides CLBs and interconnection tiles Xilinx provides and other tile types, such as Random Access Memory (RAM) blocks, Phase-locked loops (PLLs), digital signal processors (DSPs), even high performance embedded processors, providing high performance and capacitance to the FPGA. For example Xilinx vendor provides the new Zynq-7000 family (2013), which embeds a state-of-the-art ARM Cortex A9 micro-processor core in it, enabling designers to evaluate high performance systems-on-chip (SoC), while keeping the power consumption low.

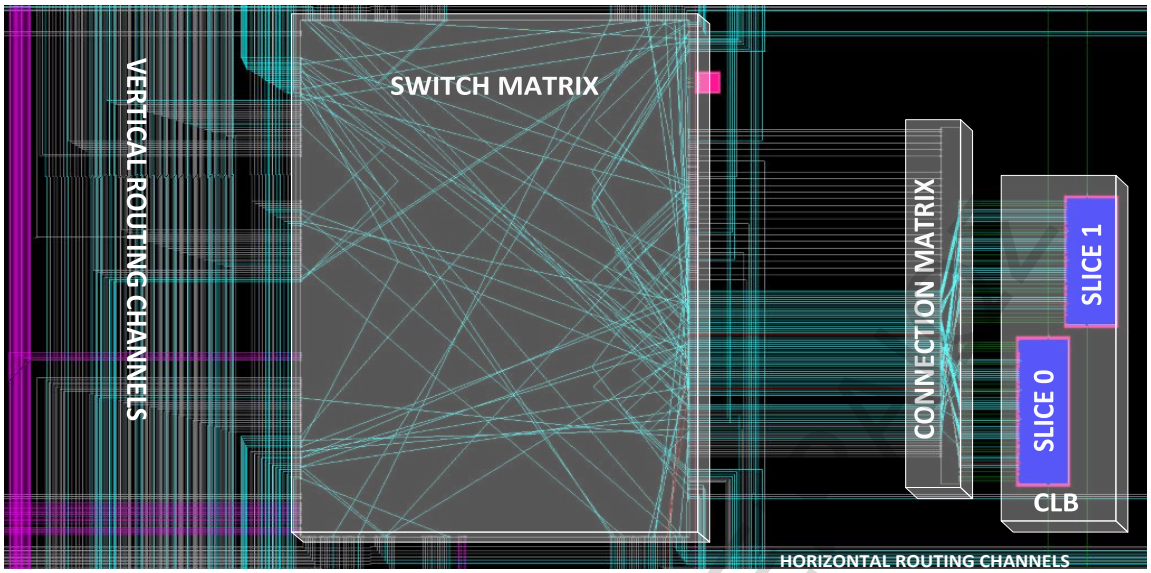


**Figure 15 - Xilinx general architecture**

In the next section the corresponding descriptions of tiles, primitive sites, wires and PIPS within the XDLRC report files is provided.

**Tiles**

Tiles are represented in Rapidsmith with the device.Tile class. A tile starts with the keyword "tile" followed by the X, Y coordinates that indicate the location of the tile in the FPGA layout. Furthermore, it contains the name of the tile and the number of the primitive sites that are hosted in it. The tile ends with a "tile summary", summarizing the name and type of the tile. The declaration finishes with some numbered statistics. Figure 16 depicts a tile which resides on the X = 1, Y = 14 coordinates of the FPGA fabric layout. It has a unique name *CLB_X6Y63*, while its type is CLB. Finally the tile carries 4 primitive sites (CLB 4).

```
(tile 1 14 CLB_X6Y63 CLB 4

(tile_summary CLB_X6Y63 CLB 122 403 148)
```

**Figure 16 - XDLRC tile declaration**

**Primitive sites**

Primitive sites are represented in Rapidsmith in the device.PrimitiveSite class. Furthermore, a primitive type enumeration is available in Rapidsmith. Only instances (design.Instance class) compatible to the primitive site type can be placed in it. Also a list with pinwires describing the name and direction of the pins contained in the site is provided in the XDLRC description file. In Figure 17 we depict an example of a primitive site declaration. From the attribute *SLICE_X9Y127* we can notice that the site type is *SLICEL*. Any instance compatible to this type can reside on *SLICE_X9Y127*. Finally, the description ends with a list containing the pins of the site, which are the BX, BY, CE input pins and the XMUX out pin with their corresponding internal tile names.

```
(primitive_site SLICE_X9Y127 SLICEL internal 27

(pinwire BX input BX_PINWIRE3)

(pinwire BY input BY_PINWIRE3)

(pinwire CE input CE_PINWIRE3)

(pinwire XMUX output XMUX_PINWIRE3)
```

**Figure 17 - XDLRC primitive site declaration**

**Wire**

A list of wires is declared in a tile that describes the routing resources which are used to connect the specific tile with other tiles. For example in Figure 18 the wire E2BEG0 is connecting the tile with 3 interconnection tiles (INT_X8Y63, INT_X9Y63, INT_X9Y62) and two CLBs (CLB_X7Y63, CLB_X8Y63). The connections are denoted with the keyword 'conn' and are described with the tile name and the wire name used for the connection. Connections which are located in the connection matrix are not programmable (fixed), in contrast with connections in the switch matrixes that use PIPS for programmable connection. Wire enumeration is provided by the device.WireEnumerator class. The enumeration is represented with integers, giving significant compression to the XDLRC description file. Another technique evaluated in the Rapidsmith, in order to reuse wire data structures is the use of relative tile offsets.

```
(wire E2BEG0 5

(conn CLB_X7Y63 CLB_E2BEG0)

(conn INT_X8Y63 E2MID0)

(conn CLB_X8Y63 CLB_E2MID0)

(conn INT_X9Y63 E2END0)

(conn INT_X9Y62 E2END_S0)
```

**Figure 18 - XDLRC wire declaration**

**PIP**

As mentioned above, a PIP is responsible to connect two wires in a switch matrix. Figure 19 outlines a PIP declaration, which describes that the wire '*BEST_LOGIC_OUTS0*' will be connected with the wire '*BYP_INT_B5'*, if the PIP is set high ("turned on") in the switch matrix INT_X7Y63.

```
(pip INT_X7Y63 BEST_LOGIC_OUTS0 -> BYP_INT_B5)
```

**Figure 19 - XDLRC PIP declaration**

**Primitive Definitions**

At the end of the XDLRC report file there is a collection of primitive definitions for the targeting Xilinx FPGA part number, which are used for reference and are not very frequent used from Rapidsmith, as some necessary information is not provided from Xilinx vendor. Father information for the XDL file format can be found in [9][34].

## 2.6 Related Work

Many previous approaches have analyzed the vulnerability of the configuration memory of SRAM-based FPGAs to soft errors and investigated their effects in the behavior of various applications. These approaches are based either on fault injection experiments [35], [12], [13], [14], [15] or analytical methods [6], [16], [17], [18].

The fault injection process is performed using either accelerated radiation testing [6], [12], [15] or fault injection tools [13], [14], [15]. In radiation-based approaches, the device under test is exposed to a controlled flux of radiation, emitted either by proton accelerators [12] or radioactive sources like proton beam [15] to slowly introduce upsets in the memory cells of the device. To reduce the high cost needed for the experimental setup of the radiation tests, fault injection approaches [13], [14], [15] emulate the effects of SEUs in the FPGA's configuration memory as bit-flips in the memory cells. The fault injection approaches can be used to analyze the susceptibility of the final configuration bitstream and thus they cannot apply during the FPGA design flow in order to provide an early estimation. To avoid the time-consuming fault simulations, several analytical approaches have been proposed [6], [16], [17], [18].

In [16] and [17] Asadi et al. present an analytical soft error rate estimation methodology which is based on the error propagation probability of the SEUs from the error sites to system outputs. In [18] the

authors propose methods for the estimation of the sensitive configuration bits after the placement process, which are adopted by our post-placement analysis method.

In [12] Johnson et al. have performed fault injection experiments on a Xilinx Virtex 1000 FPGA in order to compare the performance of simulation and accelerated radiation based tests. The simulator was built on the SLAACI-V PCI FPGA board. The board had tree Virtex 1000 FPGAs (PE0, PE1 and PE2). PE1 was used to emulate the effects of SEUs by changing the contains of the configurations memory throw partial reconfiguration, PE1 had the golden bitstream and the last PE2 FPGA was used to capture the output results of PE1 and PE2, which were operating identically under normal circumstances. To validate the simulation, acceleration test was used to slowly introduce upsets in the PE1 with proton beam. The output errors were captured with the same way in the simulation test, i.e. comparing the output of the PE1 and PE2 with the PE0 FPGA. The simulator predicted 97% of the output errors observed during the radiation tests.

Furthermore, in [6] Bellato et al. proposed an analytical method to investigate the effects of SEUs in the SRAM configuration memory of a Xilinx Virtex XCV300 FPGA and also introduced radiation fault injection experiments to validate the correctness of their results. In order to analyze the effects of SEUs on the FPGA resources, the authors first decoded the stored configuration memory in the device by continuously observing the generated bitstream outputs of all possible configuration modes of a single given resource. They accomplished to decode the 192 bits of the CLB resources and find out how they are affected from SEUs. A bit flip in a LUT could modify an implemented function, defected muxes could cause new exit paths from the CLB and last the initialization of the CLB could change the behavior of its internal components. The researchers categorized the possible interconnection soft-errors scenarios, by inserting or deleting nets in an initial design and afterwards observing the differences in the generated bitstreams. These scenarios are as follows:

- Open bit scenario: In order to emulate this scenario, they deleted a net that was connecting two pins in the CLB. The PIP which was connecting the two pins was set to open state producing an open-bit error.
- Short bit or bridge scenario: In order to emulate this scenario, they replaced an existing net with a new one, activating another PIP which connected an unknown logic value to the CLB.
- Input Antenna scenario: They inserted a new net starting from an unused input pin to a used output. The new input pin could influence the behavior of the CLB especially if it was connected at a high frequency output pin.
- Output Antenna scenario: They inserted a new net starting from a used input pin to an unused output. The new input pin did not influence the behavior of the CLB because the output pin was not used.
- Conflict scenario: A new net connected two used input and output pins, producing a conflict, since the unknown output values of the input pin were fed from the output pin.
- None scenario: They added a net in order to enable a PIP which connected two unused pins. This did not affect the functioning of the CLB.

Due to this analysis, they were able to understand the consequences of a soft error in the configuration memory and the programmable resources. However, the preliminary analysis has been done for a Virtex device and cannot be easily reproduced for other FPGA architectures. Recent approaches have proposed SEU-aware mapping, placement and routing algorithms [8],[18], [19], [20], [21] in order to reduce the vulnerability of the FPGA designs. In [8] and the [36] placement and routing algorithms of VPR tool are modified in order to reduce the susceptibility of the FPGA circuits to SEUs. In [19] the authors present a reliability-aware place and route algorithm to mitigate the effects of SEUs to TMR-based circuits. In [20] a modified version of the VPR algorithms is proposed to reduce the bridging faults caused by SEUs in the configuration memory. In [21] the proposed SEU-aware placement and routing algorithms incorporate both application level and physical level factors to reduce the soft error rate. Most of the above approaches have been demonstrated on the well-established academic VPR tool targeting only virtual FPGA architectures. The proposed framework will enable the evaluation of such reliability-aware algorithms for industrial FPGA devices.

# 3 Methodology

The methodology chapter will provide to the reader the description of the proposed soft-error vulnerability analysis framework, a deep analysis of the methods which are used to estimate the sensitive bits in placed and routed designs and finally a briefly analysis of the framework code structure that is implemented, as the proposed open-source framework (documented in Java-docs) and a technical report will be soon available on the internet.

## 3.1 Soft-error vulnerability analysis framework

The research community has intensively addressed the last years the problem of bridging the academic CAD tools with commercial CAD tools and therefore be able to apply them to real industrial FPGAs. For example, the authors in [37] connected an academic synthesis and verification tool (ABC) with the Xilinx's ISE CAD flow and compared it with the Xilinx Synthesis Technology (XST) tool. In [38], the JBits interface [39] was combined with the open-source VPR tool [32] to generate configuration bitstreams for Xilinx Virtex architectures (placed and routed by VPR) for the needs of a fault tolerant methodology. In more detail, the authors described the architecture of the Xilinx Virtex FPGAs and modified the source of the VPR tool in order to make it capable to place and route a design on a Virtex (XCV100, XCV300) FPGA. Finally, an interface was implemented to connect the Jbits interface for Virtex with the output of the VPR tool and therefore generate the bitstream configuration file. Recently, the authors in [40] developed an extension of the academic Verilog-To-Routing (VTR) [41] flow to synthesize, optimize and technology map a netlist with ODIN II [42] and ABC tools on a Virtex-6 Xilinx FPGA, pack and place it with the VPR tool and subsequently route and generate the bitstream with the Xilinx CAD flow. The drawback of these approaches is the extra effort needed to develop netlist models and interfaces to support new FPGA architectures, e.g. only a specific Virtex-6 device is supported in [40].

Motivated by the above approaches and the need of research community for FPGA reliability analysis tools, an open–source soft error vulnerability analysis framework based on RapidSmith [9] was developed, that is capable to target industrial FPGA architectures. RapidSmith is a set of open-source tools and APIs written in Java language that manipulate the Xilinx human readable XDL files, allowing researchers to try out new ideas in various fields of FPGA CAD domain. In order to be imported to RapidSmith, any design netlist must be first converted to the compatible XDL file format. Note that an XDL netlist can be easily exported in almost all implementation stages using Xilinx utilities. The advantage of RapidSmith compared to the traditional academic FPGA CAD flows is its ability to target the latest Xilinx FPGA architectures. This was the main reason that of using RapidSmith to build the vulnerability analysis tool. Figure 20 depicts the main functions supported by the proposed soft error vulnerability analysis framework. The sensitivity analysis of an FPGA design to soft errors can be performed at all stages of the FPGA design flow, while all different types of configuration bits, e.g. block configuration bits (CLBs, IOBs, DSPs, etc.) and interconnection configuration bits are considered. The user is free to run the entire flow and measure the dynamic cross section of the final FPGA design or run individual tools at earlier stages of the flow to pre-estimate the soft-error vulnerability of the design.

The functions supported by the framework are the following:

- Post–mapping analysis of the block configuration bits: It extracts the FPGA resource utilization data (e.g. number of utilized slices, DSPs, IOBs, BRAMs, LUT inputs, slice functional mode, I/O direction and attributes, etc.) from the XLD netlist produced by the packing/mapping step and analyses the sensitivity of the block configurations bits based on a precompiled resource usage profile (described in an xml file).
- Post–placement analysis of the interconnection configuration bits: It takes into consideration the actual sites of the used resources obtained by the placement process (extracted from the XDL netlist) and the goals of the routing algorithm and analyses the possibility of a net to become open-wired or short-wired with another net due to a soft error in a programmable interconnection point (PIP). So, it estimates the vulnerability of the interconnection configuration bits before the final routing. This tool is mainly based on sensitivity analysis methods previously proposed in the literature [6], [18].

- Post–routing analysis of the interconnection configuration bits: It provides a more accurate analysis since it relies on the final routed circuit. It considers all possible defects that can be caused by a soft error in a programmable interconnection point, i.e. open faults, bridging faults and antenna faults. The final analysis results are written in a text file (.rsba stands for routing sensitive bit analysis) for further processing.
- Analysis of the Xilinx report for essential configuration bits: Xilinx supports the generation of an essential (sensitive) bitmap along with the generation of the configuration bitstream (using flag –g EssentialBits:Yes in the bitgen command). The tool analyses the Xilinx report and parses the sensitive bitmap (.ebd file) and the bitstream (.bit file) using RapidSmith packages. After that, it classifies the sensitive bits as reported by Xilinx into block, interface and interconnection configuration bits and allocates them to configuration frames. This sensitivity bitmap analysis could facilitate scrubbing-based SEU mitigation approaches to prioritize the testing of most critical configuration frames of the FPGA device in order to reduce the mean repair time [43]. The results are written in a text file (.xsba stands for Xilinx sensitive bit analysis) for further processing.
- Visualization of soft-error vulnerable areas: A Graphic tool[3] built as extension of the Rapidsmith Device.Explorer class reads the results from the two previous analysis steps (.rsba and .xsba files) and illustrates the vulnerable areas of the FPGA layout. This allows the user to visualize the vulnerability analysis results of the proposed approach and compare them with the sensitivity report of Xilinx.

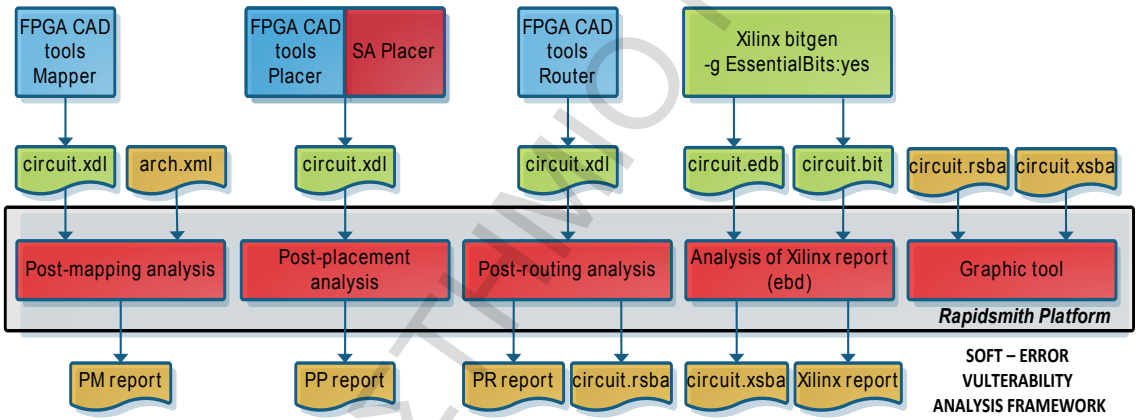


**Figure 20 - Soft error vulnerability analysis framework.**

Since the analysis tools require only the XDL description of the circuit under test, the framework can interact with any FPGA CAD tool (packer, mapper, placer, router) provided that the XDL circuit description is available. Also, given that RapidSmith can manipulate the XDL description of any FPGA device the proposed framework can support all existing Xilinx FPGA architectures. Currently, the tools have been tested for Virtex-5 and Virtex-6 families supporting all the available devices. The only restriction is that the analysis of Xilinx report (.ebd file) for Virtex-6 is not feasible due to undocumented internal frame structure from Xilinx. The .xml file provided to the post– mapping analysis tool includes the results of the usage profiling[4] of the programmable resources for a specific FPGA architecture. According to this profile, each primitive resource has been mapped with its theoretical configuration bits based on its possible usage mode. The well-known Simulated Annealing (SA) placement algorithm is also implemented, described in [30] in order to replace the uncompleted placer package provided by RapidSmith framework and to evaluate the proposed analysis methods. The main reason, this algorithm

[3] The graphic tool entirely implemented by Mr. Aitzan Sari.

[4] The analysis and implementation of usage profiling was entirely evaluated by Mr. Aitzan Sari. He has performed the profiling of Virtex-5 and Virtex-6 architectures and he plans to integrate the profiles of more Xilinx architectures in the future.

was selected, is that its cost function can be easily modified from researchers to implement an SEU-aware placement algorithm [8], [18], [20].

## 3.2 Estimation of sensitive configuration bits

In this section a description of the methods used to estimate the sensitive configuration bits is provided. Both programmable logic resources and routing resources are considered in the analysis since both categories contribute significantly to the total amount of sensitive configuration bits as shown by the experimental results. The analysis distinguishes the sensitive bits to interconnection and block configuration bits while the sensitive interconnection bits are being further classified to open, short and antenna sensitive bits.

### 3.2.1 Sensitive Block Configuration Bits

The estimation of sensitive block configuration bits can be applied as early as the mapping process on the FPGA design flow. The block configuration bits are classified into CLB, IOB, BRAM and DSP resource configuration bits estimation while two estimation methods are proposed: a black-box method and a structural analysis method. The former method depends only on the post-mapping resource utilization data while the latter uses a structural sensitivity analysis of the programmable resources to improve the estimation accuracy.

The black-box estimation approach [43] assumes that all configuration bits of a used programmable resource are sensitive. According to this pessimistic assumption, the number of sensitive bits per programmable resource can be extracted from the documented structure of configuration bitstream. The sensitive bits of programmable blocks **R** are calculated by dividing the configuration bits of a column for block R $\mathbf{ColumnBits_R}$ to the number of blocks in the column $\mathbf{ColumnBlocks_R}$ and multiplying with the number of blocks $\mathbf{N_R}$ used in the design.

$$\mathbf{Sensitive\ bits}\ (\mathbf{R}) = \frac{\mathbf{ColumnBits_R}}{\mathbf{ColumnBlocks_R}} * \mathbf{N_R} \quad (2)$$

For example, in the Virtex-5 family a CLB column needs 11 configuration frames (although it is documented that a CLB column requires 10 configuration frames, a careful CLB utilization and bitstream examination reveals that the actual number of frames is 11) and with 1280 bits per frame (excluding the ECC word since it is considered as non-sensitive) there are 14080 configuration bits which are uniformly distributed across the column. A CLB column in the Virtex-5 family consists of 20 CLBs and each CLB contains two slices. Applying Equation 2, gives 704 configuration bits per CLB or 352 configuration bits per slice. The sensitive configuration bits for all programmable resources are calculated similarly. Table I summarizes the results for the primary block resources of Virtex-5 architecture. The figure in parenthesis in the second column of Table I denotes the number of configuration frames per column, i.e. 30 configuration frames (28 block plus 2 interface frames) per IOB column, 4 configuration frames (2 block plus 2 interface frames) per BRAM column and 2 interface configuration frames per DSP column.

| Block (*R*) | $\mathbf{ColumnBits_R}$ | $\mathbf{ColumnBlocks_R}$ | Sensitive bits |
|---|---|---|---|
| CLB | 14080 (11) | 20 | 704 |
| IOB | 35840 (30) | 40 | 896 |
| BRAM | 5120 (4) | 4 | 1280 |
| DSP | 2560 (2) | 8 | 320 |

**Table I - Black-box estimation of sensitive bits per block.**

### 3.2.2 Sensitive Interconnection Configuration Bits

The fault modeling and the vulnerability analysis of the FPGA routing resources have been studied extensively in the past [6], [13], [14], [18]. Here, the fault modeling of routing resources proposed by several previous approaches are adopted, according to which the routing faults due to soft errors in the programmable interconnection points can be open, short or antenna. So, an interconnection configuration bit is termed as open-sensitive when a soft error causes an open wire, as short-sensitive when it causes the

bridging of two distinct nets, and finally as antenna-sensitive when it results to a hanging wire connected to a net.

The estimation method is based on the simple interconnection block model depicted in Figure 21 which describes adequately the interconnection blocks of Virtex-5 and Virtex-6 FPGA architectures. The interconnection block consists of a switch-matrix and an interface block. The interface block is used to connect the terminals (inputs/outputs) of a resource block (CLBs, IOBs, etc.) to the switch matrix which in turn provides access to the global interconnection network.

Consider the nets connected to the interconnection block of Figure 21: two nets are routed through the switch matrix, namely *NET A* and *NET B* which use the wires *N1W4* and *E4S4*, respectively and the connections made for these wires are realized with a single PIP (Programmable Interconnection Point) for each net. As shown in the example, *NET B* is connected to the configurable block through the interface block. These PIPs are open-sensitive PIPs since a soft error will result in disconnecting the starting wire from the end wire of the switch matrix. A soft error, for example, in the SRAM cell of *PIP_N1W4* will disconnect the *N1* wire of *NET A* from the *W4* wire.

Figure 21 depicts also a net-bridging scenario where two nets are shorted as a consequence of soft error in *PIP_E4W4* which connects wires *E4* and *W4*. Since these wires are used by nets *A* and *B*, respectively, a short connection is formed between these two nets. In the case of antenna-sensitive bit, a soft error causes a wire used by a net to be connected to an unused wire of the switch matrix. For example, *PIP_S4W6* is considered as antenna-sensitive bit, since an SEU will cause wire *S4* to be connected to *W6* producing a wire acting as antenna on *NET B*.

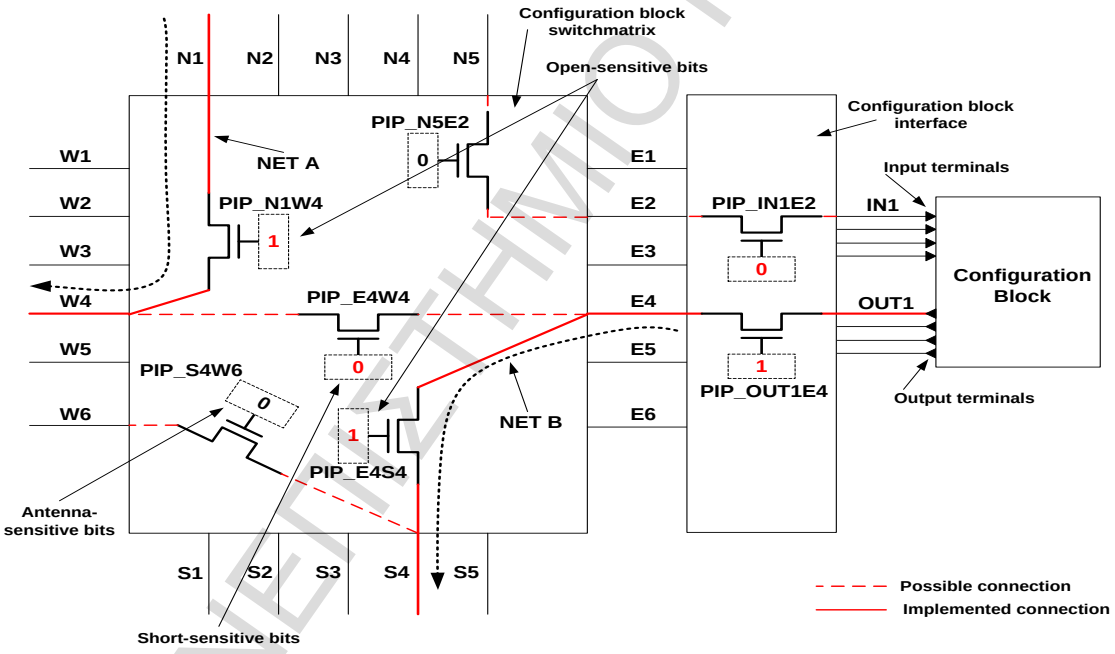


**Figure 21 - Sensitive bits of an interconnection block.**

Although in most cases an antenna will not lead to erroneous output, it will degrade the performance of a circuit, especially when it occurs in high-frequency nets such as a clock net. Finally, Figure 5 shows a non-sensitive PIP (*PIP_N5E2*) where a soft error will not affect the circuit operation since the bridging wires are not used. The information required by the above analysis (e.g. which PIPs are in use, which wires can be connected through PIPs, etc.) is extracted from the XDL model of the circuit.

As described in the related work section, the framework provides two roadmaps for the analysis of the sensitive interconnection bits: the post-placement and the post-routing analysis. The post-placement analysis provides to the designer a vulnerability estimation early in the design flow while it can also drive a reliability aware placement algorithm [18], [20], [8]. On the other hand, the post-routing analysis provides a more precise calculation of the sensitive configuration bits.

## 3.3 Post-placement analysis

The proposed framework has adopted the method introduced in [18] to calculate the open-sensitive and short-sensitive bits of an FPGA design. The open-sensitive bits for a single net are calculated using the Manhattan distance applied on its Bounding Box (BB) assuming $\mathbf{X_{min}Y_{min}}$ and $\mathbf{X_{max}Y_{max}}$ being the coordinates of the BB. Equation 3 is used to calculate the number of open-sensitive bits. The post-placement method adopts the *q* [44] factor used also in the simulated annealing placement algorithm to characterize the pin-count of the particular net.

$$\mathbf{Sens.bits\ (open)} = (|\mathbf{X_{max-}X_{min}}| + |\mathbf{Y_{max-}Y_{min}}| + \mathbf{1}) * \mathbf{q} \quad (3)$$

The method for the estimation of short-sensitive bits proposed in [18] is based on the usage probability of switch matrices within the BB of a net. To find the short-sensitive bits between two nets NET1 and NET2 the method just uses the product of their usage probabilities over the overlap area (Equation 4). A simple example is shown in Figure 22 where two nets have been considered to illustrate the estimation process.

$$\boldsymbol{Sens.bits\ (short)} = \sum_{\forall(i,j)in_{overlap_{area}}} \mathbf{p^{N1}\ (i,j)} * \mathbf{p^{N2}(i,j)} \quad (4)$$

where $\mathbf{p^N(i,j)}$ is the probability of net N routing through switch matrix **(i,j)**. For more details the reader can refer to [18].

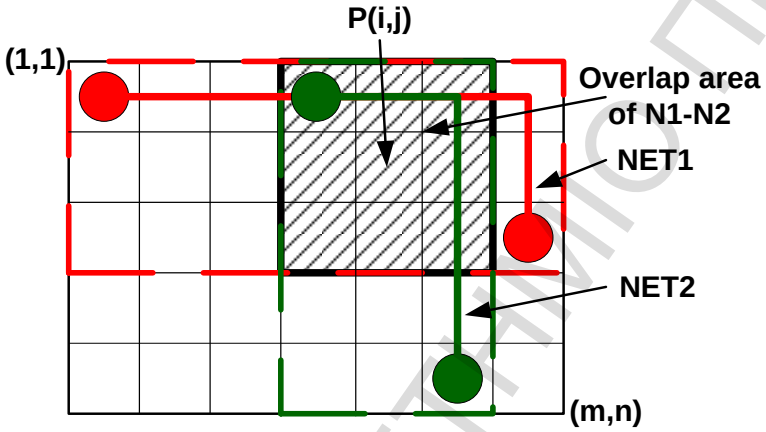


**Figure 22 - Short sensitive bits of two nets.**

## 3.4 Post-routing analysis

Although post-placement analysis provides a useful tool to estimate sensitive bits at an early phase of the design flow it lacks accuracy, overestimating the susceptibility of routing resources to soft errors. A more accurate calculation can be done analyzing the final routed circuit through its XDL netlist. The calculation of the open-sensitive bits is almost straightforward process since it requires a simple exploration of the nets and an aggregation of the used PIPs. The PIPs-counting process considers all PIPs belonging to switch-matrices and interconnection interface blocks.

In order to calculate the short sensitive bits, the post-routing analysis algorithm checks the wires used in the design and for each wire it identifys its possible connections. The possible connections for a given wire (i.e. connections supported by the switch matrix) are extracted using the appropriate APIs of the RapidSmith framework. Remember that a short circuit between two nets is possible when there is a potential, but unused connection of one net that, if it is activated due to an error in the corresponding PIP, the net will be connected to a wire used by the second net. Note that the calculation is carried out considering the PIPs of the switch-matrix and not the PIPs of the interface blocks, since the latter use point-to-point connections and the wires connected to each PIP does not provide an alternative routing path. This means that a short circuit cannot be formed among the wires of the interface block.

To illustrate the post-routing analysis, assume the simple example of the switch-matrix shown in Figure 23. There are four wires used to route the two nets, namely *N1*, *W4*, *E4* and *S4*. By examining each

wire, it can be observed that there is a possible connection of *W4* with *E4*, *S4* with *W6* and *E4* with *W4*. Only connections *W4-E4* and *E4-W4* can form a bridge between *NET A* and *NET B*. The last factor of the post-routing analysis is the antenna sensitive bits. An antenna sensitive bit, as previously mentioned, produces a wire that acts as a radiation medium to a used net which could reduce the performance of a circuit or even lead to circuit malfunction when occurs in critical nets, e.g. high-frequency nets, long-wire nets, etc. The antenna sensitive bits are calculated by finding the alternative connections of the used wires in the switch matrix which do not produce a short circuit. The pseudo-code of Figure 23 describes the calculation of short and antenna sensitive interconnection bits. Note that since the connections are bi-directional and each connection is considered twice in the loop (i.e. for both end points of the connection), the final number of short sensitive bits is divided by two.The total sensitive configuration bits for an FPGA design can be calculated summing-up the block configuration bits and the interconnection configuration bits using either the post-placement analysis or the post-routing analysis.

$$
\begin{aligned}
&short \ = 0 \\
&\\
&antenna = 0 \\
&\forall \ SwitchMatrix \\
&\quad W_{used} = \{wires\ used\ in\ the\ SwitchMatrix\} \\
&\quad \forall\ N\ \in\ W_{used} \\
&\quad\quad W_{alternative} = \{alternative\ connections\ of\ N\} \\
&\quad\quad\ short\ += |W_{alternative} \cap\ W_{used}| \\
&\quad\quad antenna += |W_{alternative} -\ W_{used}| \\
&\quad end \\
&end \\
&short\ /= 2
\end{aligned}
$$

**Figure 23 - Pseudo-code for the calculation of short & antenna-sensitive bits**

## 3.5 Soft error vulnerability analysis framework packages

This section describes the structure of the proposed soft-error vulnerability analysis framework. Several packages are provided for placing a design with the well-known simulated annealing algorithm, routing it and finally generating the bitstream configuration files (calling tools provided by Xilinx vendor). At each CAD stage, i.e. after mapping, placement, routing a design or generating the bitstream, the user can analyze the vulnerability of the design to soft-errors. A hierarchy of packages within our framework can be seen in Figure 24. The framework consists from the placer package, utilities package, userInterface package and finally the analysis package. In the following sections a description of all classes and methods which are available in the framework will be provided.

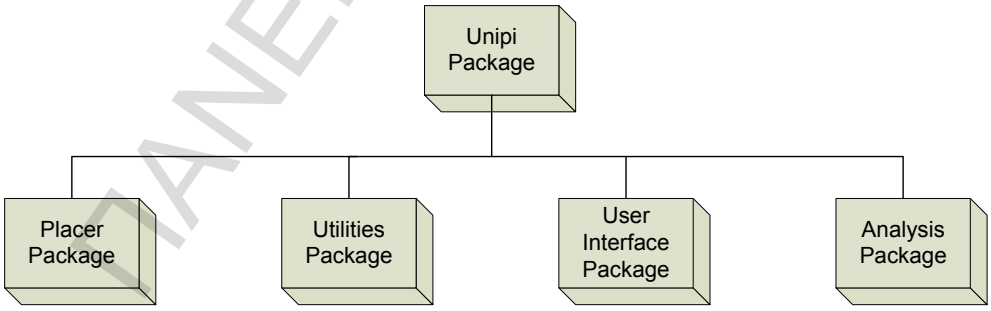


**Figure 24 - Unipi packages**

### Overview

Soft-error vulnerability framework is organized into several packages. All packages are prefixed with "unipi" (University of Piraeus):

| Package Name | Description |
|---|---|
| **userInterface** | A command prompt user interface providing to the end-user an easy way to access out proposed soft-error vulnerability tools. |
| **Placer** | Provides the well-known simulated annealing placement algorithm in order to replace the uncompleted placer package provided by Rapidsmith framework. |
| **Utilities** | Provides classes for performing mathematical functions needed from the unipi.placer package and the unipi.analysis package, utilities such as to load designs, save designs, route designs, generate bitstreams and exporting results to excel files. |
| **Analysis** | Provides classes to estimate the vulnerability of FPGA designs to soft-errors. In particular, it supports post-mapping analysis of the sensitive block configuration bits, post-placement analysis of the sensitive interconnection bits and post-routing analysis of the total sensitive configuration bits. Furthermore it provides a class for analysis of the Xilinx report for essential (sensitive) configuration bits. At last it provides a class to visualize the vulnerability analysis results of our and Xilinx sensitivity analysis approaches. |

**Figure 25 - Provided packages from the framework**

### 3.5.1 Placer Package

A hierarchy of the classes within the placer package can be seen in Figure 26.

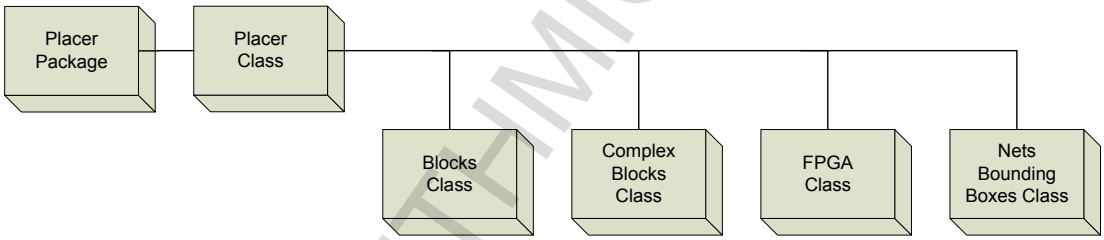


**Figure 26 - Hierarchy of the classes within the placer package**

The Placer class represents the simulated annealing placement algorithm. It reads a XDL file, unplaces all the instances of the design and continues with a simulated annealing placement. It can handle a Xilinx User Constraints File (UCF) indicating the IO pins that must be locked to specific locations and does not move them during the placement phase. It supports the placement of all programmable resources, but it is not capable to handle carry chains. Thus, you must provide designs without carry chains. If the design has carry chains, the CLBs containing curry chains will not be moved. The Placer class encapsulates following classes:

**Block class**

Represents a primitive site and the instance that resides in it. When an instance of the design is placed in this primitive site, the primitive site (location, type) is initialized with the properties of the instance.

| Block class private field | Detail |
|---|---|
| **PrimitiveSite site** | The primitive site of the block. |
| **Instance instance** | The Instance that is placed in this block primitive site. |

| Block class constructors | Detail |
|---|---|
| **Block()** | Constructor which initializes all member data structures. Sets site and instance to null |

| | |
|---|---|
| **Block(PrimitiveSite site)** | Creates a new Block and populates it with the given PrimitiveSite. |
| **Block(PrimitiveSite site, Instance inst)** | Creates a new Block and populates it with the given PrimitiveSite and Instance |

| Block class public methods | Detail |
|---|---|
| **clone()** | Creates a copy of the object, with the same class and with all the fields having the same values. |
| **addInstance(Instance instance)** | Adds a new instance to the block. |
| **Instance getInstance()** | Returns the instance of the block. |
| **removeInstance()** | Sets the instance within the block to null. |
| **setSlice_inst(Instance slice_inst)** | Creates a new instance and initializes its primitiveSite. |
| **PrimitiveSite getSite()** | Returns the PrimitiveSite of the block. |
| **HashSet<Net> getNetList()** | Returns a list with nets attached on the blocks instance. |

**ComplexBlock class**

Represents a FPGA tile. A complex block houses primitive sites. Placement occurs by assigning an instance to a specific primitive site. The instances have been grouped in a complex block in order to keep the packing of the design immutable.

| ComplexBlock class private field | Detail |
|---|---|
| **ArrayList<Block> blocks** | The list of blocks in the complexBlock. A block contains a primitive site and an instance if available. |
| **TileType tileType** | XDL Tile Type (INT,CLB,...) |
| **int x** | The horizontal coordinates of the tile in the FPGA layout. |
| **int y** | The vertical coordinates of the tile in the FPGA layout. |
| **boolean used** | A boolean indicating if the complexBlock (tile) is used or not. |

| ComplexBlock class constructors | Detail |
|---|---|
| **ComplexBlock()** | Constructor which initializes all member data structures. tileType is set to null. |
| **ComplexBlock class public methods** | **Detail** |
| **clone()** | Creates a copy of the object, with the same class and with all the fields having the same values. |
| **swap(ComplexBlock cb)** | Replaces this complexBlock with the given complexBlock cb. |
| **place(ArrayList<Block> blocks)** | Places a list of blocks within the complexBlock. |
| **add(Block block)** | 'Adds a block in the complexBlock. |
| **add(PrimitiveSite site, Instance inst)** | Creates a new block, initializes it with the given PrimitiveSite and Instance. Finally, it adds it in the complexBlock. |
| **HashSet<Net> getNets()** | Returns a hashest<Net> list with the nets attached to this complexBlock. |
| **ArrayList<Block> getBlocks()** | Returns the blocks contained in this complexBlock. |
| **TileType getTileType()** | Returns the tile type of this complexBlock. |

**NetBB class**

Represents a bounding box of a net. It has xmax, xmin, ymax, ymin dimensions. Its cost is the half perimeter of the bounding box that encapsulates all the attached logic blocks to the net.

| NetBB class private field | Detail |
|---|---|
| **int xMax** | The maximum horizontal (X) coordinate of the attached logic block to the net. |
| **int xMin** | The minimum horizontal (X) coordinate of the attached logic block to the net. |
| **int yMax** | The maximum vertical (Y) coordinate of the attached logic block to the net. |
| **int yMin** | The minimum vertical (Y) coordinate of the attached logic block to the net. |
| **double cost** | The half perimeter of the bounding box that encapsulates all the attached logic blocks to the net. |
| **int terminals** | The number of terminals attached on the net. |

| NetBB class constructors | Detail |
|---|---|
| **NetBB()** | Constructor which initializes all member data structures. |
| **NetBB (int xMax, int xMin, int yMax, int yMin)** | Constructor which initializes all member data structures with the given values. |
| **NetBB (int xMax, int xMin, int yMax, int yMin, int terminals, String netName)** | Constructor which initializes all member data structures with the given values. |
| **NetBB(NetBB netbb)** | Constructor which initializes all member data structures with the data structures values of the given netbb. |

| NetBB class public methods | Detail |
|---|---|
| **getxMax()** | Returns the maximum horizontal (X) coordinate of the attached logic block to the net. |
| **setxMax(int xMax)** | Sets the maximum (X) coordinate location of the attached logic block to the net. |
| **getxMin()** | Returns the minimum horizontal (X) coordinate of the attached logic block to the net. |
| **setxMin(int xMin)** | Sets the minimum horizontal (X) coordinate of the attached logic block to the net. |
| **int getyMax()** | Returns the maximum vertical (Y) coordinate of the attached logic block to the net. |
| **setyMax(int yMax)** | Sets the maximum vertical (Y) coordinate of the attached logic block to the net. |
| **int getyMin()** | Returns the minimum vertical (Y) coordinate of the attached logic block to the net. |
| **setyMin(int yMin)** | Sets the minimum vertical (Y) coordinate of the attached logic block to the net. |
| **double getCost()** | Returns the cost of the bounding box which is the half perimeter of the bounding box that encapsulates all the attached logic blocks to the net. |
| **setCost(double cost)** | Sets the cost of the bounding box which is the half perimeter of the bounding box that encapsulates all the attached logic blocks to the net. |
| **String getNetName()** | Gets the name of the net. |

| | |
|---|---|
| **int getTerminals()** | Gets the number of terminals attached on the net. |
| **setTerminals(int terminals)** | Sets the number of terminals attached on the net. |

**Fpga Class**

Holds information about the targeting FPGA layout and the instances of the design. It also contains a list with the bounding boxes of each net in the design and the design cost. Every tile of the targeting FPGA is loaded with the placed instances that rely in them. This class contains methods to swap the instances of two random tiles and automatically updates the design cost. Furthermore, this class provides methods to restore the design to its last state, i.e. before swapping two tiles. Finally, rich statistics about the numbers of slices, tiles, IOs, tiles with carry chains and resource utilization are provided by the class.

| **Fpga class public field** | **Detail** |
|---|---|
| **ArrayList<ComplexBlock> blocks** | An ArrayList with the complex blocks of the design. In other words the tiles of the design and their relying instances. |
| **HashMap<String,NetBB> netsBB** | A map containing the name of each net in the design with its bounding box. |
| **boolean ucf** | A boolean indicating if a UCF file is provided in order to be taken into account in the placement phase. |
| **ArrayList< String> ucfList** | An ArrayList containing the tiles that must not be moved in the placement phase. |
| **Fpga class constructors** | **Detail** |
| **Fpga()** | This constructor gets the design that must be placed. Afterwards it loads the complex blocks of the design and calculates the bounding boxes of every net. |

| **Fpga class public methods** | **Detail** |
|---|---|
| **swapTwoRandomComplexBlocks(boolean debug)** | Swaps the instances of two random tiles and automatically updates the current cost of the design. If debug flag is set true, the locations of the instances before and after swapping are printed. |
| **fallBack()** | Restores the design to its last state, i.e. before swapping the last two random tiles. |
| **int getUsedIOs()** | A list with the used I/O blocks of the design. |
| **int getUsedCLBs()** | The number of used CLBs in the design. |
| **HashMap<PrimitiveType, Integer> getTypeFreq()** | A list with the used Primitive Types. |
| **hasCarry(Instance inst)** | A flag designating if the given instance has a carry chain. |
| **int getCarryClbs()** | Returns the number of CLBs that have instances with carry chains. |
| **int getUsedSlices()** | Returns the number of used slices. |
| **loadCompexBlocks(boolean debug)** | Loads the complex blocks of the design in the block ArralList, except the tiles that contain instances with curry chains. If debug flag is true, the tiles of the targeting FPGA and their relying instances are printed. |
| **set_cost(double designCost)** | Sets the cost of the design. |
| **double get_cost()** | Returns the cost of the design. |
| **double calculateCost()** | Calculates and returns the cost of the design from scratch. More specific, it updates the cost of each net and returns the summary cost of all nets cost, i.e. the design cost. |

| | |
|---|---|
| **calculate_nets_cost()** | Updates the cost of each net. |
| **static Design getDes()** | Returns the design. |
| **static void setDes(Design des)** | Sets the design. |

### 3.5.2 Utilities package

A hierarchy of the classes within the utilities package can be seen in Figure 27 below. This class contains useful methods to manipulate XDL files, can call external tools like Xilinx PAR and bitgen, provides mathematical functions and finally can produce excel worksheets.

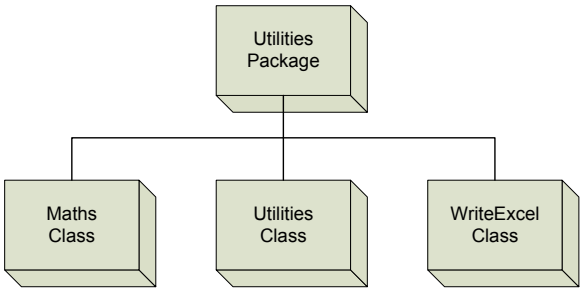


**Figure 27 - Hierarchy of the classes within the utilities package.**

**Maths class**

Maths class contains static methods for performing mathematical functions used in the placer package and the analysis package.

| Maths class field | Detail |
|---|---|
| **static FactCache factCache** | A static cache for saving calculated factorial numbers. |
| **protected static Vector<Double> table** | A vector table used from the factCache. |
| **static final double[] cross_count** | An lookup table providing the q(i) factors used to compensate the wire length model in the linear congestion cost function of the simulated annealing based placement . |

| Math class public static methods | Detail |
|---|---|
| **static synchronized double factorial(int x)** | Returns the factorial of int x, using BigIntegers cached in a Vector. This method uses arbitrary precision integers, so it does not have an upper-bound on the values it can compute. It uses a Vector object to cache computed values instead of a fixed-size array. A Vector is like an array, but can grow to any size. The factorial() method is declared "synchronized" so that it can be safely used in multi-threaded programs. Look up java.math.BigInteger and java.util.Vector while studying this method. |
| **static Double calcPnm(int n, int m)** | This method is used for finding combinations in probability theory. It uses Double numbers in the factorial calculation procedure p(n,m). |
| **static double calcPnmBigInt(int n, int m)** | This method is used for finding combinations in probability theory. It uses double numbers in the factorial calculation procedure p(n,m). |
| **static double calcPijmn(int i, int j, int m, int n, int bboxTerminal)** | Calculates the usage probability of the switch matrices that will be used is the routing phase of |

| | |
|---|---|
| | a design. This method is called from the unipi.analysis package for accurately estimation of the short bits. It tries to calculate first the factorials with double numbers and if the results are out of range, it re-calculates the factorial with BigIntegers. This technique in combination with cashing speedups the calculations. |
| **static int gcd(int a, int b)** | Calculates the Greatest Common Divisor between two integers. |
| **static double getCrossCount(int terminals)** | Returns the q(i) factor. Values are ranging from 1.0 to 2.7933 depending on nets terminals. |
| **static double get_std_dev(int n, double sum_x_squared, double av_x)** | This method calculates the standard deviation of the given values. It is called from the placer package in order to calculate the initial temperature of the simulated annealing schedule. |

**Utilities class**

The Utilities class provides static methods to load XDL designs, convert Xilinx XDL files to Xilinx NCD files and vice versa, to route and generate the configuration bitstream of the targeting FPGA device and to execute external command line programs within the framework. It also provides extra methods to generate reports of the targeting FPGA, time conversions and finally provides logging capabilities to the framework.

| Utilities class public methods | Detail |
|---|---|
| **String milliseconds2hms(long millis)** | Gets milliseconds and returns a string with the time in hh:mm:ss:ms format. This method throws java.text.ParseException. |
| **Design loadDesign(String xdlFile)** | Creates a new design and loads the XDL design. |
| **printDesignReport(Design design)** | Reports family PartName and the available columns and rows of the targeting device. |
| **routeDesign(String ncdPlaced, String ncdOutRouted)** | Gets the path of a NCD placed design, calls the Xilinx PAR tool to route it and afterwards saves it to the specified path. It also calls Xilinx reportgen, trce and xpwr Xilinx tools in order to perform timing and power analysis of the routed design. |
| **generateBitstream(String routedNCD)** | Calls the Xilinx bitgen tool to create the bistream and the essential (sensitive) bitmap of a routed NCD design. |
| **String convertXdl2Ncd(String myXdlFileName)** | Converts a file XDL file to NCD file by the same name but with an .ncd extension. |
| **String convertNcd2Xdl(String myNcdFileName)** | Converts a NCD file to XDL file by the same name but with an .xdl extension. |
| **redirectConsole(String fileName, PrintStream logFile, boolean redirect)** | If redirect flag is enabled, the output screen console is redirected to the given filename path. |

**WriteExcel class**

This class is used for the creation of an excel file in order to export the performance of the simulated annealing placement and the results of the soft-errors vulnerability analysis tools in it.

| WriteExcel class private fields | Detail |
|---|---|
| **WritableCellFormat timesBold10** | Bold times new roman fonts. |
| **WritableCellFormat times10** | Times new roman fonts. |
| **File file** | The file to create the excel file. |
| **WorkbookSettings wbSettings** | Settings of the excel workbook. |
| **WritableWorkbook workbook** | Writable excel workbook. |

| String pathname | We save the excel file to this path. |
|---|---|
| CellView cv | This is a bean which client applications may use to get/set various properties for a row or column on a spreadsheet. |
| **WriteExcell class constructors** | **Detail** |
| WriteExcell() | Constructor which initializes all member data structures. Creates a new CellView object and a new WorkbookSettings object. |

| WriteExcell class public methods | Detail |
|---|---|
| setPathName(String pathName) | Sets the path of the excel file. |
| createExcel() | Creates a new excel. You must first set the global variable pathname. |
| initialize() | Initializes the excel object with the workbook settings. |
| addCaption(int column, int row, String s) | Adds a string to the given column and row of the worksheet. |
| addInt(int column, int row, int integer) | Adds an integer number to the given column and row of the worksheet. |
| addDouble(int column, int row, double d) | Adds a double number to the given column and row of the worksheet. |
| addLong(int column, int row, long l) | Adds a long number to the given column and row of the worksheet. |
| addLabel(int column, int row, String s) | Adds a bold string to the given column and row of the worksheet. |
| setBold(boolean timesBold) | Sets the strings or the numbers fonts to bold. |
| saveExcel() | Saves the excel workbook. |

### 3.5.3 Analysis package

A hierarchy of the classes within the analysis package can be seen in Figure 28 below. In this package the following classes are provided to:

- Estimate the sensitive block configuration bits of mapped designs.
- Estimate the sensitive interconnection configuration bits of placed or routed designs.
- Analyze the report of Xilinx essential configuration bits.

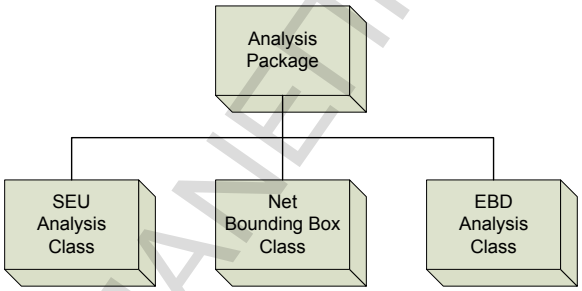


**Figure 28 - Hierarchy of the classes within the analysis package**

**SEUAnalysis class**

This class estimates the vulnerability of FPGA designs to soft errors at all phases of the FPGA implementation flow. In particular, it supports post-mapping analysis of the sensitive block configuration bits, post-placement analysis of the sensitive interconnection bits and post-routing analysis of the total sensitive configuration bits. Finally, it analyzes the report of Xilinx essential configuration bits and classifies the sensitive bits according to their configuration type: block configuration bits (CLBs, IOBs, DSPs, etc.) and interconnection configuration bits.

| SEUAnalysis class field | Detail |
|---|---|
| **Design design** | The design to be analyzed. |
| **HashMap<String, Bbox> netBboxes** | A map containing the name of each net in the design with its bounding box. |
| **ArrayList<Bbox> netBbList** | A list with the Bonding box of each net in the design. |
| **Maths maths = new Maths();** | An instance of Math class in order to use it in the post-placement analysis phase. |
| **Integer[][] fpga;** | A two-dimensional array representing the switch matrices of the targeting FPGA device. This field is used when we find the overlap between two Bounding Boxes. |
| **long resourceBits** | A variable to store the total resource configuration sensitive bits. |
| **long clbSensitiveBits** | A variable to store the total CLB configuration sensitive bits. |
| **long ramSensitiveBits** | A variable to store the total RAM configuration sensitive bits. |
| **long ioSensitiveBits;** | A variable to store the total IOB configuration sensitive bits. |
| **long dspSensitiveBits;** | A variable to store the total DSP configuration sensitive bits. |
| **int sliceBlocks;** | A variable indicating the number of used SLICES in the design. |
| **int ioBlocks** | A variable indicating the number of used IO tiles in the design. |
| **int bramBlocks** | A variable indicating the number of used BRAM tiles in the design. |
| **int dspBlocks** | A variable indicating the number of used DSP tiles in the design. |
| **String path** | The path of the design. |
| **String sbaFilePath** | The path for writing the .rsba file (stands for routing sensitive bits analysis). |
| **String xmlPath** | The path of the XML file (precompiled resource usage profile). |
| **ArrayList<ComplexBlock> blocks** | An ArrayList of the complex blocks of the design. In other words the tiles of the design and their relying instances. |
| **int sliceControlBitsTotal** | The number of sensitive bits (total) for control bits of the slices. |
| **long sensitiveBits_open** | The number of open sensitive bits (total) of the design analysis. |
| **long sensitiveBits_short** | The number of short sensitive bits (total) of the design analysis. |
| **int iobSensitiveInterface_open** | The number of open interface sensitive bits for the IO blocks. |
| **int clbSensitiveInterface_open** | The number of open interface sensitive bits for the CLB blocks. |
| **int bramSensitiveInterface_open** | The number of open interface sensitive bits for the BRAM blocks. |
| **int dspSensitiveInterface_open** | The number of open interface sensitive bits for the DSP blocks. |
| **int clkSensitiveInterface_open** | The number of open interface sensitive bits for the CLK blocks. |
| **int sliceControlBitsTotal** | The number of sensitive bits (total) for control bits of the slices. |

| | |
|---|---|
| **long sensitiveBits_short** | The number of short sensitive bits (total) of the design analysis. |
| **int iobSensitiveInterconnectionBits_short** | The number of short interconnection sensitive bits for the IO blocks. |
| **int clbSensitiveInterconnectionBits_short** | The number of short interconnection sensitive bits for the CLB blocks. |
| **int bramSensitiveInterconnectionBits_short** | The number of short interconnection sensitive bits for the BRAM blocks. |
| **int dspSensitiveInterconnectionBits_short** | The number of short interconnection sensitive bits for the DSP blocks. |
| **int clkSensitiveInterconnectionBits_short** | The number of short interconnection sensitive bits for the CLK blocks. |
| **enum ColumnType {CLB, IO, BRAM, DSP, CLK, UNKNOWN}** | Enumeration for the types of the column that we currently can handle |

| **SEUAnalysis class constructors** | **Detail** |
|---|---|
| **SEUAnalysis()** | Constructor which initializes all member data structures. |
| **SEUAnalysis(String ncdIn, String path, String xmlPath)** | Constructor which initializes all member data structures and performs the soft-error vulnerability analysis of the resource block configuration bits. |

| **SEUAnalysis class public methods** | **Detail** |
|---|---|
| **void printResources()** | It prints the results of the resource block configuration bits analysis. More specifically it prints the sensitive bits of the:<br><br>a)Clbs,<br><br>b)Ram,<br><br>c)DSP,<br><br>d)Total resource block configuration sensitive bits<br><br>Furthermore, the number of used:<br><br>a) SLICE blocks,<br><br>b) IO blocks,<br><br>c) BRAM blocks,<br><br>d) DSP blocks. |
| **void routeAnalysis()** | Post-routing analysis. It provides a more accurate analysis since it relies on the final routed circuit. It considers all possible defects that can be caused by a soft error in a programmable interconnection point, i.e. open faults, bridging faults and antenna faults. The analysis results are written in a text file (.rsba stands for routing sensitive bit analysis) for further processing. |

| | |
|---|---|
| | Final, this method outputs the routing Analysis results:<br><br>a)Open sensitive bits,<br><br>b)Short sensitive bits,<br><br>c)Sum of Short and Open sensitive bits,<br><br>d)Antennas sensitive bits,<br><br>e)Sum of Short, Open and Antenna sensitive bits,<br><br>f)Total sensitive bits (without antennas),<br><br>g)Total sensitive bits (with antennas)<br><br>Furthermore it outputs the discrimination of the interconnection sensitive bits to:<br><br>a)CLB interconnection sensitive bits,<br><br>b)IO interconnection sensitive bits,<br><br>c)DSP interconnection sensitive bits,<br><br>d)BRAM interconnection sensitive bits,<br><br>e)CLK interconnection sensitive bits, |
| **ColumnType getColumTileType(HashMap<String, PrimitiveSite> primitiveSites, int x)** | This mehtod returns a column type.<br><br>param primitiveSites: The sites of the FPGA device<br><br>param x: The x-position of the block.<br><br>return ColumnType: The column type of the block located in the x position of the FPGA layout. Returns UNKNOWN in case the column type cannot be found. |
| **long getAllShort()** | Returns the short sensitive bits of a routed design. (included the antennas sensitive bits) |
| **long getShortBits_route()** | Returns the short sensitive bits of a routed design (does not include the antenna sensitive bits). |
| **void placementAnalysis()** | Post-placement method analyzes the interconnection configuration bits of a placed design: It takes into consideration the actual sites of the used resources obtained by the placement process (extracted from the XDL netlist) and the goals of the routing algorithm and analyses the possibility of a net to become open-wired or short-wired with another net due to a soft error in a programmable interconnection point (PIP). So, |

| | |
|---|---|
| | it estimates the vulnerability of the interconnection configuration bits before the final routing. This tool is mainly based on sensitivity analysis methods previously proposed in the literature in the paper [Abdul-Aziz, M.A.; Tahoori, M.B., "Soft error reliability aware placement and routing for FPGAs," IEEE International Test Conference (ITC), Nov. 2010]. We output the following:<br><br>a)Open sensitive bits ( manhattan dist * 3 ),<br><br>b)Open sensitive bits ( manhattan dist * q(i)),<br><br>c)Open sensitive bits (manh dist * q(i) * 1.5),<br><br>d)Short sensitive bits,<br><br>e)Total sensitive bits. Sum of the [resource + Short + Open (manhattan distance * 3) ] sensitive bits,<br><br>f)Total sensitive bits. Sum of the [resource + Short + Open * q(i) ] sensitive bits,<br><br>g) Total sensitive bits. Sum of the [resource + Short + (Open q(i) * 1.5) ] sensitive bits. |
| **void getTypeFreq()** | Initializes the global variables that hold the statistics of the used primitive sites. |

| **SEUAnalysis class private methods** | **Detail** |
|---|---|
| **long getOpenBits_route()** | Returns the open sensitive bits of a routed design. |
| **long getCLBSensitiveBits()** | Returns the sensitive bits for the CLB resources of the targeting device. |
| **long getIOSensitiveBits()** | Returns the sensitive bits for the IOB resources of the targeting device. |
| **long getIOSensitiveBits(PrimitiveSite iobSite, Instance iobInstance)** | Overloading getIOSensitiveBits() method. Returns the sensitive bits for a specific IOB (PrimitiveSite and instance) resource of the targeting device. |
| **int getUsedSlices()** | Returns the number of used slices. |
| **int getSLICE_controlSensitiveBits(Instance inst)** | Gets an instance (only instances that reside on SLICES) and returns the control sensitive bits. |
| **Attribute find_attribute( Collection<Attribute> attributes, String attributeName)** | Finds a specific attribute (by name) from an attribute list and returns it. |
| **long getLUT_used()** | Calculates the number of the used LUT resources in the design. Furthermore it prints to the console the:<br>a)Number of used LUTs,<br>b)Sensitive bits of the design LUTs, |

| | |
|---|---|
| | c)Number of LUTs with 1 input,<br>d)Number of LUTs with 2 inputs,<br>e)Number of LUTs with 3 inputs,<br>f)Number of LUTs with 4 inputs,<br>g)Number of LUTs with 5 inputs,<br>h)Number of LUTs with 6 inputs. |
| **long calculateClock_antennas()** | Returns the sensitive antenna bits of a routed design. |
| **int calculateUserNet_antennas(String userfilePath)** | Calculates only the sensitive antenna bits specified from user (nets from the given text file path). |
| **double getTerminals()** | Returns the summary of all terminals of the tiles in the design. |
| **double[] getOpenBitsEst1_place()** | Returns a double[] array with the open sensitive bits of the placed design. We have implemented the first method of the paper "Soft error reliability aware placement and routing for FPGA" to calculate the open sensitive bits:<br>double[0]=Open sensitive bits (manh * 3),<br>double[1]=Open sensitive bits (manh * q(i)),<br>double[2]=Open sensitive bits (manh * (q(i) * 1.5)), |
| **Bbox calcOverLap(Bbox bboxA, Bbox bboxB)** | Returns a Bounding Box indicating the overLap of two Bounding Boxes in the device. If overlap does not exist, the method returns the bounding box with its initial values, i.e. xMax, xMin, yMax, yMax, terminals. |
| **Double getShortBitsEst2_place()** | Returns the short sensitive bits of a placed design. We have implemented the second method of the paper "Soft error reliability aware placement and routing for FPGA" to calculate the short sensitive bits. |
| **MultiKeyMap<Integer,Double> calcProp(Bbox bbox)** | Gets a bounding box of a net and returns a Map (key = XY cords, Value = probability) with the probability of a pip to be used in xy coordinates. Terminal cross count is taken into account.<br>Offset correction:<br>xMin = 0 and xMax = xMax - xMin.<br>yMin = 0 and yMax = yMax – yMin. |
| **void fillDesBboxes()** | Calculates the Bounding Box for each Net and adds them in the Global HashMap<String, Bbox> netBboxes map and ArrayList<Bbox> netBbList list. |
| **void loadFile(String ncdName)** | Loads the XDL design into the class. |
| **void loadCompexBlocks(boolean debug)** | Loads the complex blocks of the design in the block ArralList. This method is same with unipi.placer.FPGA loadCompexBlock method, with the one difference. It loads the instances containing curry chains. Param: debug while true, we output the tiles of |

| | |
|---|---|
| | the FPGA and their relying instances. |
| **PrimitiveSite findIOB(int xTile, int yTile, int x, int y)** | Returns the correlated IODELAY or ILOGIC or OLOGIC PrimitiveSites of an IOB PrimitiveSite in order to be packed in one complex block. |
| **long getRamSensitiveBits()** | Returns the RAM sensitive bits. This method is implemented for Virtex5 and Virtex6 architectures. |
| **long getDspSensitiveBits()** | Returns the DSP sensitive bits. This method is implemented for Virtex5 and Virtex6 architectures. |

| **Nested class wireSite** | **Detail** |
|---|---|
| **wireSite** | Nested class used to represent a wire |
| **Inner class wireSite public fields** | **Detail** |
| **String siteName** | The name of the site at which the wire resides. |
| **int wire** | The wire. |
| **String netName** | The name of the net at which the wire belongs. |
| **String wireName** | The name of the wire. |
| **Inner class wireSite constructors** | **Detail** |
| **wireSite()** | Constructor which initializes all member data structures. |
| **wireSite(String name, int wire)** | Initializes the wireSite class by name and wire. |
| **wireSite(String name, int wire, String netName)** | Initializes the wireSite class by name, wire and netName. |
| **wireSite(String name, int wire, String netName, String wireName)** | Initializes the wireSite class by name, wire, netName and wireName. |

**EBD_Analysis class**

This class analyzes the essential bits generated from the Xilinx bitgen tool.

| **EBD_Analysis class public field** | **Detail** |
|---|---|
| **Bitstream bitstream** | The bitstream representation, provided by Rapidsmith framework. |
| **String ebd_fileName** | The essentials configuration bits path. |
| **XilinxConfigurationSpecification spec** | Specifications of the targeting FPGA architecture. |
| **HashMap<Integer, Frame> ebdFrames** | A map of the FPGA frames. |
| **long sensitiveConfigurationBits_total** | The total sensitive configuration bits. |
| **long sensitiveConfigurationBits** | The sensitive configuration bits. |
| **long sensitiveInterconnectionBits** | The sensitive interconnection bits. |
| **long sensitiveInterfaceBits** | The sensitive interface bits. |
| **ArrayList<column> columnFrames** | A list with the column frames. |
| **FPGA fpga** | The targeting FPGA layout. |

| **EBD_Analysis class constructors** | **Detail** |
|---|---|
| **EBD_analysis(String bitStream_fileName)** | Constructor which initializes all member data structures.<br><br>param[in] String: The full path of the design bitstream file. |

| **EBD_Analysis class public methods** | **Detail** |
|---|---|
| **loadBitStream()** | Parses the bitstream file. The bitstream file |

| | |
|---|---|
| | should be in debug format. |
| **loadEBD()** | Parses the EBD file. The bitstream file should be in debug format. |
| **int get_IOB_columns()** | Returns the IO block sensitive bits of all frames. |
| **long get_sensitiveBits()** | Gets the total number of sensitive bits. |
| **get_sensitiveConfigurationBits()** | Gets the block configuration sensitive bits. |
| **long get_sensitiveInterconnectionBits()** | Gets the interconnection sensitive bits. |
| **int get_IOB_sensitiveInterconnectionBits()** | Gets the sensitive interconnection bits for the IOBs. |
| **int get_IOB_sensitiveInterfaceBits()** | Gets the sensitive interface bits for the IOBs. |
| **int get_IOB_sensitiveBlockConfigurationBits()** | Gets the sensitive block configuration bits for the IOBs. |
| **int get_CLB_sensitiveInterconnectionBits()** | Gets the sensitive interconnection bits for the CLBs. |
| **int get_CLB_sensitiveInterfaceBits()** | Gets the sensitive interface bits for the CLBs |
| **int get_CLB_sensitiveBlockConfigurationBits()** | Gets the sensitive block configuration bits for the CLBs. |
| **int get_BRAM_sensitiveInterconnectionBits()** | Gets the sensitive interconnection bits for the BRAMs. |
| **int get_BRAM_sensitiveInterfaceBits()** | Gets the sensitive interface bits for the BRAMs. |
| **int get_BRAM_sensitiveBlockConfigurationBits()** | Gets the sensitive block configuration bits for the BRAMs. |
| **int get_DSP_sensitiveInterconnectionBits()** | Gets the sensitive interconnection bits for the DSPs. |
| **int get_DSP_sensitiveInterfaceBits()** | Gets the sensitive interface bits for the DSPs. |
| **int get_DSP_sensitiveBlockConfigurationBits()** | Gets the sensitive block configuration bits for the DSPs. |
| **int get_CLK_sensitiveInterconnectionBits()** | Gets the sensitive interconnection bits for the CLKs. |
| **int get_CLK_sensitiveInterfaceBits()** | Gets the sensitive interface bits for the CLKs |
| **int get_CLK_sensitiveBlockConfigurationBits()** | Gets the sensitive block configuration bits for the CLKs. |
| **long get_sensitiveInterfaceBits()** | Gets the interface sensitive bits. |
| **getResults()** | Prints the results. |

| **EBD_Analysis class private methods** | **Detail** |
|---|---|
| **findFramesPerColumn()** | This function finds and maps the sensitive frames for each FPGA device column. The mapping is done by finding the start and end frame address for each column. This information is saved in the ArrayList "columnFrames" |
| **Byte[] bytesTrim(byte[] bytes)** | This function removes any padding bytes (0x0D) from the input byte array.<br><br>param[in] bytes : The byte array which contains the data read in ascii format.<br><br>returns Byte[]: The byte array without the padding bytes (0x0D). |
| **ArrayList<Integer> bytesToWords(Byte[] frameBytes)** | This function converts the data of the frame which is in ascii format to its binary equivalent (words).<br><br>param[in] frameBytes: The bytes of the frame in |

| | |
|---|---|
| | ascii format.<br><br>returns List<Integer>: The words of the frame in binary format. |
| **List<Byte> asciiBytes_toBinary(Byte[] frameBits)** | This function converts the data of the frame which is in ascii format to its binary equivalent.<br><br>param[in] frameBytes: The bytes of the frame in ascii format.<br><br>Returns List<Byte>: The bytes of the frame in binary format. |

| **Nested class columnFramesRange** | **Detail** |
|---|---|
| columnFramesRange | Nested class representing the column frames range. |
| Inner class columnFramesRange public fields | Detail |
| int startAddress | The start position address of the frame. |
| int endAddress | The end position address of the frame. |
| int frames | The number of frames in this start – end frame address range. |
| Inner class columnFramesRange constructors | Detail |
| columnFramesRange() | Constructor which initializes all member data structures with zero. |

| Inner class column | **Detail** |
|---|---|
| Column | Nested class representing a frame column. |
| Inner class columnpublic fields | Detail |
| int columnIndex | The start index of the frame. |
| int sensitiveConfigurationBits | The sensitive configuration bits of this column. |
| int sensitiveInterconnectionBits | The sensitive interconnection bits of this column. |
| int sensitiveInterfaceBits | The sensitive interface bits of this column. |
| columnFramesRange | Address details of the frame. |
| PrimitiveType columnType | The type of sites that reside in this frame. |
| Inner class columnFramesRange constructors | Detail |
| column(int columnIndex, columnFramesRange frameAddressInfo) | Constructor which initializes all member data structures with the given values. |

### 3.5.4 UserInterface Package

The userInterface package contains only one class which provides to the user an easy way to place, route, generate a bitstream configuration file or perform a soft-error sensitivity analysis at all CAD stages. The user is free to run only one desired stage or run the entire flow at once.

The usage of the console interface has the following arguments:

| **Console arguments** | **Detail** |
|---|---|
| **-b** | Generate the Bitstream. |
| **-c** | Redirect Console to the log file. |
| **-e** | Export placer performance to excel. |
| **-ebd** | Perform the ebd analysis. |
| **-ep <arg>** | Epsilon value. Default value = 0.005 |
| **-m <arg>** | Moves per temperature multiplier. Default 10 |
| **-oa** | Perform the analysis only (not placement). |
| **-p <arg>** | Path of the design (xdl or ncd file). |

| | |
|---|---|
| **-pa** | Place Analysis. |
| **-pl** | Place the design with Sa placer. |
| **-r** | Route the design. |
| **-ra** | Route Analysis. |
| **-resa** | Resource Analysis. |
| **-seu** | SEU awareness placement (beta). |
| **-ucf <arg>** | Path of UCF file |
| **-xdl2ncd** | Convert the placed xdl file to ncd file. |

# 4 Experimental results

A rich set of experiments are carried out to demonstrate all functions of our vulnerability analysis framework. The first experimental set has been performed to evaluate the post-mapping analysis method. The experiments compare the packing and mapping steps of the VTR and Xilinx tools in order to explore their effects to the block sensitive configuration bits. In order to compare the two packers some benchmark circuits are used from the VTR flow which have been synthesized for a Virtex-6 device (XC6VLX240T-1FFG1156) using the method described in [40]. The same circuits have been also implemented using the Xilinx flow. The results of the post-mapping analysis are shown in Table IV and Table V. The first rows present the FPGA resource utilization while the last four rows present the sensitive configuration bits per programmable resource type and the total. It is interesting to see that both packers result to almost the same sensitive bits for all the resources except slices. Xilinx ISE packer produces 3.15% more sensitive bits on average for the IOBs and 4.76% less sensitive bits on average for the DSP slices compared to VTR packer. Regarding the logic slices, ISE flow generates significantly less sensitive configuration bits by a factor of 34.31% on average which is due to the fact that ISE synthesis and packing tool produces less slices compared to the VTR synthesis and packing toolsets. An interesting point also is the LUT utilizations. From the experimental results it is clearly depicted that Xilinx ISE results in more low-utilized LUTs (LUTs with less than 4 inputs) but it generates less high-utilized LUTs (LUTs with more than 3 inputs) than the VTR flow. It is obvious that the LUT utilization affects substantially the slice block sensitive bits.

The second experimental set has been used to demonstrate the performance of the implemented simulated annealing placement algorithm and to evaluate the post-placement and post-routing analysis. The QUIP benchmarks [45] shown in Table II which have been adapted to the design flow of Xilinx ISE and synthesized on a Virtex-5 device FPGA (XC5VLX30FF67) are used in these experiments. For the above benchmarks, both post-placement and post-routing analysis methods have been executed. Table VI and Table VII present the results of the two steps in terms of sensitive interconnection bits (open-sensitive, short-sensitive and total) considering the SA placer and Xilinx ISE placer respectively. The two placers present the same behavior in terms of SEU awareness. Precisely, Xilinx ISE placer produces slightly less open-sensitive bits than the SA placer by a percentage of 0.79% and 3.24% less short-sensitive bits than SA placer for the post-routing analysis. Table VI and Table VII can be also used to evaluate the accuracy of the post-placement estimation of interconnection sensitive bits compared to the post-routing analysis. The post-placement estimation provides sufficient results: it introduces a small overestimation of 10% in case of the open-sensitive which is almost doubled for the short-sensitive bits estimation (18.54%). Table III presents the efficiency of the SA placer in comparison with Xilinx ISE placer. The SA placer has better performance by a percentage of 5.7% (wire-length-cost) in average, while it produced faster designs by a percentage of 5.8%. Finally the power consumption of the SA placer designs is 0.7% more than the ISE placer designs in average. Figure 29 depicts the placement of b4 benchmark (mux_128bit) with SA placer and Xilinx ISE placer respectively.

The last experimental set has been performed to evaluate the vulnerability analysis method compared to the sensitivity report of Xilinx (.ebd file). All benchmarks have been implemented using the SA placer and the Xilinx router. Table VIII and Table IX compare the sensitive block configuration bits and interconnection configuration bits, respectively, for the two methods. The results are similar for the block configuration bits of slices and IOBs. Specifically, the method presents a slight augmentation to the number of sensitive block configuration bits for the slices (2.21%) and for the IOBs (7.98%). However, in the case of DSP and RAM blocks, there is considerable difference to the number of sensitive bits. This is due to the *black-box* estimation approach used for these blocks which results in an overestimation of the sensitive bits. Regarding the sensitive interconnection bits, the results of the post-routing analysis are used, which are compared with the sensitivity data from Xilinx report. There are significant differences in the results of the two methods. The Xilinx sensitivity analysis results to a significantly larger number of sensitive interconnection bits for all categories. In future, fault injection experiments will be performed to identify whether the analysis framework underestimates or the Xilinx tool overestimates the vulnerability of the designs to soft errors. Figure 30 illustrates the sensitive bits of the mux_128bit circuit in the FPGA layout. The circuit has been placed using the SA placer. Figure 30 shows the circuit's sensitive sites using the proposed framework as well as the Xilinx report. Although the developed visualization tool supports sensitivity-level coloring of the sensitive sites, for visibility reasons only one color-level has been used. It

is clearly observed that both tools produce the same results with slight differences in the switch-matrix locations.

| | Benchmark | SLICEL | SLICEM | ILOGIG | OLOGIK | IOB | BUFG | RAMB18X2 | RAMB36_EXP | DSP48E |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | **oc_correlator** | 91 | 0 | 73 | 2 | 85 | 1 | 0 | 0 | 0 |
| **2** | **oc_des_des3perf** | 2157 | 122 | 120 | 64 | 298 | 1 | 0 | 0 | 0 |
| **3** | **barrel64** | 103 | 0 | 71 | 64 | 136 | 1 | 0 | 0 | 0 |
| **4** | **mux_128bit** | 327 | 0 | 11 | 128 | 140 | 1 | 0 | 0 | 0 |
| **5** | **oc_ata_ocidec3** | 182 | 6 | 40 | 28 | 130 | 1 | 0 | 0 | 0 |
| **6** | **oc_des_area_opt** | 88 | 0 | 0 | 0 | 189 | 1 | 0 | 0 | 0 |
| **7** | **fip_risc8** | 86 | 4 | 20 | 53 | 113 | 1 | 0 | 0 | 0 |
| **8** | **oc_fpu** | 430 | 3 | 69 | 40 | 110 | 2 | 0 | 0 | 2 |
| **9** | **oc_mem_ctrl** | 696 | 3 | 101 | 107 | 267 | 2 | 0 | 0 | 0 |
| **10** | **oc_pavr** | 584 | 3 | 1 | 0 | 52 | 1 | 0 | 1 | 0 |
| **11** | **oc_aquarius** | 888 | 5 | 1 | 18 | 35 | 1 | 10 | 0 | 2 |
| **12** | **oc_video_comp_sys_jpeg_log** | 1659 | 13 | 17 | 26 | 47 | 1 | 0 | 0 | 32 |

**Table II - QUIP benchmarks**

| Benchmark | Xilinx Bounding- Box Cost | Sa placer Bounding Box Cost | Efficiency (%) | Sa placer Placement time (hh:mm:ss:ms) | Sa plcer maximum frequency (MHz) | Xilinx maximum frequency (MHz) | Sa placer total Power Supply (mW) | Xilinx total Power Supply (mW) |
|---|---|---|---|---|---|---|---|---|
| **1** | 3007.29 | 3600.43 | 83.5 | 00:00:01:879 | 80.965 | 85.889 | 379.80 | 380.04 |
| **2** | 81764.3 | 76109.63 | 107.4 | 01:53:24:730 | 248.077 | 263.09 | 404.02 | 400.19 |
| **3** | 6755.77 | 6085.93 | 111.0 | 00:00:02:933 | 124.502 | 122.37 | 377.64 | 377.64 |
| **4** | 7858.46 | 6747.08 | 116.5 | 00:00:12:397 | 267.953 | 235.74 | 384.60 | 387.95 |
| **5** | 7817.56 | 7992.60 | 97.8 | 00:00:04:896 | 250.250 | 264.48 | 383.40 | 383.40 |
| **6** | 7040.79 | 6092.68 | 115.6 | 00:00:03:833 | 225.836 | 166.03 | 380.76 | 381.24 |
| **7** | 4655.01 | 4050.53 | 114.9 | 00:00:02:525 | 89.952 | 79.246 | 380.52 | 380.52 |
| **8** | 17310.6 | 17830.0 | 97.1 | 00:00:16:606 | 24.992 | 30.511 | 384.36 | 384.12 |
| **9** | 33770.5 | 29345.42 | 115.1 | 00:03:03:227 | 168.748 | 136.17 | 393.24 | 395.64 |
| **10** | 27644 | 28236.39 | 97.9 | 00:00:58:335 | 73.508 | 68.894 | 384.11 | 384.11 |
| **11** | 49194.3 | 44550.49 | 110.4 | 00:03:25:514 | 49.145 | 46.369 | 391.79 | 392.75 |
| **12** | 72257.6 | 71149 | 101.6 | 00:20:06:297 | 87.237 | 84.767 | 400.91 | 400.67 |

**Table III - Placement performance: SA placer vs Xilinx ISE placer**

| Benchmark | BGM | SYN7 | SHA | BFLY | DSCG |
|---|---|---|---|---|---|
| **Used SLICEs** | 6479 | 12009 | 378 | 2552 | 2454 |
| **Used IOs** | 289 | 289 | 74 | 257 | 193 |
| **Used DSPs** | 22 | 42 | 0 | 8 | 8 |
| **Used LUTs** | 15344 | 31031 | 1294 | 6271 | 6107 |
| **1-input LUTs** | 141 | 287 | 1 | 23 | 32 |
| **2-input LUTs** | 172 | 392 | 161 | 79 | 74 |
| **3-input LUTs** | 552 | 1149 | 4 | 183 | 202 |
| **4-input LUTs** | 731 | 1688 | 98 | 318 | 304 |
| **5-input LUTs** | 3130 | 5958 | 576 | 1241 | 1220 |
| **6-input LUTs** | 10618 | 21557 | 454 | 4411 | 4259 |
| **LUT sensitive bits** | 796794 | 1608646 | 49734 | 328930 | 318456 |
| **SLICE sensitive bits** | 1256103 | 2448864 | 77356 | 510800 | 489168 |
| **IOB sensitive bits** | 21230 | 22126 | 5508 | 19408 | 14944 |
| **DSP sensitive bits** | 7040 | 13440 | 0 | 2560 | 2560 |
| **Total sensitive bits** | 1284373 | 2484430 | 82864 | 532768 | 506672 |

**Table IV - Post-mapping analysis(block configuration bits) using the Xilinx ISE flow** [46]**.**

| Benchmark | BGM | SYN7 | SHA | BFLY | DSCG |
|---|---|---|---|---|---|
| **Used SLICEs** | 7948 | 18409 | 555 | 3151 | 3059 |
| **Used IOs** | 289 | 289 | 74 | 257 | 193 |
| **Used DSPs** | 22 | 50 | 0 | 8 | 8 |
| **Used LUTs** | 28844 | 66801 | 2071 | 11115 | 10769 |
| **1-input LUTs** | 6 | 14 | 0 | 4 | 2 |
| **2-input LUTs** | 250 | 655 | 22 | 83 | 95 |
| **3-input LUTs** | 376 | 982 | 33 | 136 | 136 |
| **4-input LUTs** | 5460 | 13981 | 521 | 2368 | 2326 |
| **5-input LUTs** | 5849 | 13377 | 624 | 2070 | 1975 |
| **6-input LUTs** | 16903 | 37792 | 871 | 6414 | 6195 |
| **LUT sensitive bits** | 1360340 | 3080952 | 84400 | 516052 | 498368 |
| **SLICE sensitive bits** | 1915016 | 4363005 | 118315 | 724849 | 690161 |
| **IOB sensitive bits** | 20642 | 23040 | 4932 | 19276 | 14054 |
| **DSP sensitive bits** | 7040 | 16000 | 0 | 2560 | 2560 |
| **Total sensitive bits** | 1942698 | 4402045 | 123247 | 746685 | 706775 |

**Table V - Post-mapping analysis(block configuration bits) using the VTR** [41]**,** [40]**.**

| | Sa placer Post-placement estimation | | | Xilinx ISE placer Post-placement estimation | | | Sa placer sensitive bits vs. Xilinx ISE placer sensitive bits (%) | | |
|---|---|---|---|---|---|---|---|---|---|
| **B** | Open | Short | Total | Open | Short | Total | Open | Short | Total |
| **1** | 8505 | 8185 | 16690 | 6468 | 4187 | 10655 | 131.49 | 195.49 | 156.64 |
| **2** | 166308 | 526535 | 692843 | 174462 | 570346 | 744808 | 95.33 | 92.32 | 93.02 |
| **3** | 14652 | 34534 | 49186 | 14286 | 23499 | 37785 | 102.56 | 146.96 | 130.17 |
| **4** | 14844 | 6336 | 21180 | 14421 | 4963 | 19384 | 102.93 | 127.66 | 109.27 |
| **5** | 18339 | 31261 | 49600 | 16905 | 21277 | 38182 | 108.48 | 146.92 | 129.90 |
| **6** | 16533 | 18615 | 35148 | 17778 | 16472 | 34250 | 93.00 | 113.01 | 102.62 |
| **7** | 7941 | 8266 | 16207 | 8529 | 8957 | 17486 | 93.11 | 92.29 | 92.69 |
| **8** | 38220 | 129462 | 167682 | 36204 | 110521 | 146725 | 105.57 | 117.14 | 114.28 |
| **9** | 57768 | 155132 | 212900 | 63387 | 151809 | 215196 | 91.14 | 102.19 | 98.93 |
| **10** | 62586 | 370032 | 432618 | 60699 | 376718 | 437417 | 103.11 | 98.23 | 98.90 |
| **11** | 92451 | 702652 | 795103 | 96978 | 786408 | 883386 | 95.33 | 89.35 | 90.01 |
| **12** | 94662 | 641509 | 736171 | 96612 | 655580 | 752192 | 97.98 | 97.85 | 97.87 |

**Table VI - Post-placement analysis (interconnection configuration bits): SA placer vs. ISE placer**

| | Sa placer Post-routing estimation | | | Xilinx ISE placer Post-routing estimation | | | Sa placer sensitive bits vs. Xilinx ISE placer sensitive bits (%) | | |
|---|---|---|---|---|---|---|---|---|---|
| **B** | Open | Short | Total | Open | Short | Total | Open | Short | Total |
| **1** | 8272 | 20426 | 28698 | 7853 | 19417 | 27270 | 105.34% | 105.20% | 105.24% |
| **2** | 151192 | 420107 | 571299 | 150059 | 417081 | 567140 | 100.76% | 100.73% | 100.73% |
| **3** | 10069 | 24174 | 34243 | 9562 | 21733 | 31295 | 105.30% | 111.23% | 109.42% |
| **4** | 16521 | 23802 | 40323 | 15874 | 22918 | 38792 | 104.08% | 103.86% | 103.95% |
| **5** | 13485 | 27260 | 40745 | 13343 | 22989 | 36332 | 101.06% | 118.58% | 112.15% |
| **6** | 8351 | 14834 | 23185 | 8323 | 13791 | 22114 | 100.34% | 107.56% | 104.84% |
| **7** | 7858 | 13275 | 21133 | 7932 | 13264 | 21196 | 99.07 | 100.08% | 99.70 |
| **8** | 36189 | 120672 | 156859 | 35778 | 120814 | 156592 | 101.15% | 99.88 | 100.17% |
| **9** | 49869 | 118468 | 168335 | 49990 | 96615 | 146605 | 99.76 | 122.62% | 114.82% |
| **10** | 53805 | 219767 | 273572 | 53116 | 219952 | 273068 | 101.30% | 99.92 | 100.18% |
| **11** | 86625 | 349154 | 435779 | 85858 | 333627 | 419485 | 100.89% | 104.65% | 103.88% |
| **12** | 103959 | 221027 | 324986 | 104194 | 219849 | 324043 | 99.77 | 100.54% | 100.29% |

**Table VII - Post-routing analysis (interconnection configuration bits): SA placer vs. ISE placer**

| | Proposed framework | | | | | Xilinx report | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **B** | **SLICEs** | **IOBs** | **BRA** | **DSPs** | **Total** | **SLICEs** | **IOBs** | **BRA** | **DSP** | **Total** |
| **1** | 19099 | 1019 | 0 | 0 | 29293 | 18136 | 9356 | 0 | 0 | 27492 |
| **2** | 402329 | 3069 | 0 | 0 | 433027 | 404062 | 28233 | 0 | 0 | 432295 |
| **3** | 27703 | 1637 | 0 | 0 | 44073 | 23540 | 14551 | 0 | 0 | 38091 |
| **4** | 38053 | 1589 | 0 | 0 | 53947 | 24536 | 13919 | 0 | 0 | 38455 |
| **5** | 29933 | 1339 | 0 | 0 | 43323 | 33297 | 12284 | 0 | 0 | 45581 |
| **6** | 21190 | 1499 | 0 | 0 | 36186 | 20756 | 14440 | 0 | 0 | 35196 |
| **7** | 17302 | 1162 | 0 | 0 | 28928 | 14995 | 10834 | 0 | 0 | 25829 |
| **8** | 99201 | 1316 | 640 | 0 | 113009 | 105877 | 11910 | 302 | 0 | 118089 |
| **9** | 118518 | 2994 | 0 | 0 | 148462 | 112997 | 27062 | 0 | 0 | 140059 |
| **10** | 132489 | 4584 | 0 | 1280 | 138353 | 145362 | 4383 | 0 | 251 | 149996 |
| **11** | 199847 | 2910 | 640 | 12800 | 216197 | 206708 | 3037 | 302 | 2450 | 212497 |
| **12** | 215635 | 5286 | 10240 | 0 | 231161 | 391503 | 4822 | 4832 | 0 | 401157 |

**Table VIII - Sensitive block configuration bits: Proposed framework vs. Xilinx report**

| | Proposed framework | | | | | Xilinx report | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **B** | **SLICEs** | **IOBs** | **BRA** | **DSPs** | **Total** | **SLICEs** | **IOBs** | **BRA** | **DSPs** | **Total** |
| **1** | 27184 | 1490 | 24 | 0 | 28698 | 41660 | 3911 | 738 | 16 | 46325 |
| **2** | 565295 | 3965 | 1138 | 901 | 571299 | 829088 | 15445 | 11952 | 9084 | 865569 |
| **3** | 33022 | 1203 | 12 | 6 | 34243 | 52157 | 4410 | 344 | 176 | 57087 |
| **4** | 38630 | 1648 | 7 | 38 | 40323 | 85198 | 5253 | 340 | 874 | 91665 |
| **5** | 39155 | 1502 | 36 | 52 | 40745 | 69252 | 4629 | 776 | 1150 | 75807 |
| **6** | 22212 | 882 | 84 | 7 | 23185 | 44130 | 3611 | 1295 | 152 | 49188 |
| **7** | 19133 | 1740 | 82 | 178 | 21133 | 38066 | 5166 | 1123 | 1975 | 46330 |
| **8** | 149518 | 1706 | 848 | 4787 | 156859 | 199696 | 5744 | 6768 | 8757 | 220965 |
| **9** | 162787 | 5200 | 129 | 219 | 168335 | 265612 | 16436 | 2082 | 3212 | 287342 |
| **10** | 268977 | 1247 | 1600 | 1748 | 273572 | 306162 | 6654 | 6789 | 1076 | 330371 |
| **11** | 412378 | 3227 | 14782 | 5392 | 435779 | 477918 | 11858 | 24679 | 9628 | 524083 |
| **12** | 263751 | 2417 | 618 | 58200 | 324986 | 511717 | 9635 | 9521 | 6389 | 594764 |

**Table IX - Sensitive interconnection configuration bits: Proposed framework vs. Xilinx report**

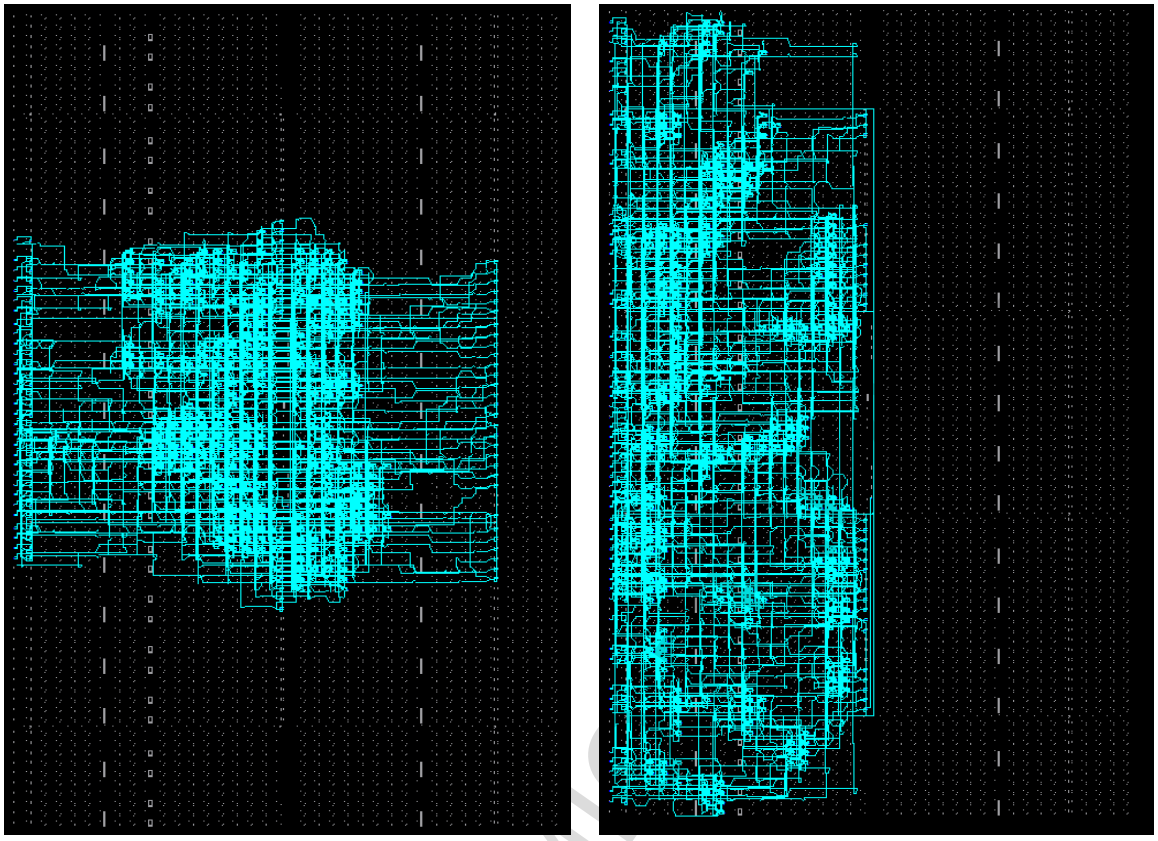

**Figure 29 - Placement with SA and Xilinx: Left: SA placer, Right: ISE placer.**

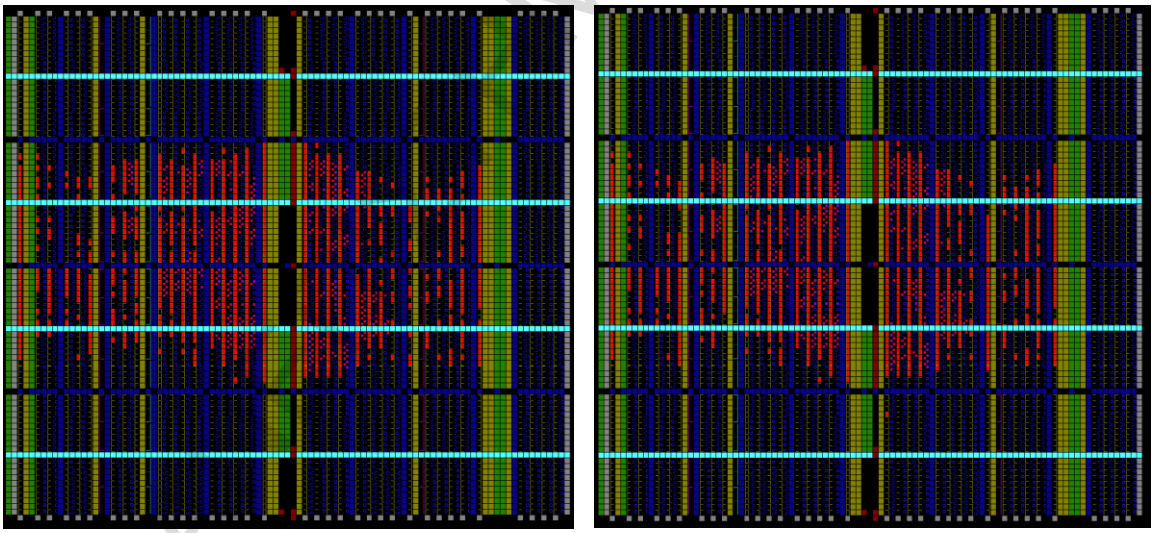

**Figure 30 - Visualization of the sensitive bits: Left: Proposed framework, Right: Xilinx report.**

# 5 Conclusions and Future Work

The problem statement of this thesis was to create a soft error vulnerability analysis framework for Xilinx FPGAs which is able to estimate soft-errors in the following enumerated CAD stages:

1. Post-mapping analysis of the sensitive block configuration bits.
2. Post-placement analysis of the sensitive interconnection bits.
3. Post-routing analysis of the sensitive interconnection bits.
4. Bitstream analysis: Classification of the configuration sensitive bits is provided, according to their configuration type: block configuration bits (CLBs, IOBs, DSPs, etc.) and interconnection configuration bits.

Furthermore, visualizition of the circuit's sensitive sites exported from Xilinx report and the proposed framework is available.

The results of this research seem to have raised several interesting directions for future work:

1. The implemented simulated annealing placer algorithm does not support the movement of CLBs that contain instances with carry chains. Therefore, carry chain handling could be evaluated on a newer version of the SA placer.
2. The proposed framework could be tested for all Xilinx FPGA families. Currently, it is has only been tested on Virtex5 and Virtex6 FPGAs.
3. The analysis of usage profiling is performed for Virtex-5 and Virtex-6 architectures. More Xilinx architectures profiles could be integrated in future.
4. Fault injection tools could be integrated in the soft-error vulnerability analysis framework.
5. The SEU aware placer described in [18] was implemented, but due to lack of time it was not fully tested. The SEU aware placer algorithm could be optimized and verified in future.

# 6 Bibliography


[1] F. Lima, C. Carmichael, J. Fabula, R. Padovani, and R. Reis, "A fault injection analysis of Virtex FPGA TMR design methodology," in *RADECS 2001. 2001 6th European Conference on Radiation and Its Effects on Components and Systems*, 2001, vol. 00, no. C, pp. 275–282.

[2] F. Lima, C. Carmichael, J. Fabula, R. Padovani, and R. Reis, "A fault injection analysis of Virtex FPGA TMR design methodology," in *RADECS 2001. 2001 6th European Conference on Radiation and Its Effects on Components and Systems (Cat. No.01TH8605)*, 2001, pp. 275–282.

[3] C. Carmichael, E. Fuller, J. Fabula, and F. Lima, "Proton testing of SEU mitigation methods for the Virtex FPGA," *MAPLD*, pp. 1–7, 2001.

[4] Actel Corporation, "Actel Provides Multiple Solutions for Airbus A380 Next-Generation Commercial Airliner," 2005. [Online]. Available: http://www.design-reuse.com/news/10703/actel-multiple-solutions-airbus-a380-generation-commercial-airliner.html.

[5] D. Binder, E. C. Smith, and A. B. Holman, "Satellite Anomalies from Galactic Cosmic Rays," *IEEE Trans. Nucl. Sci.*, vol. 22, no. 6, pp. 2675–2680, 1975.

[6] M. Bellato, P. Bernardi, D. Bortolato, A. Candelori, M. Ceschia, A. Paccagnella, M. Rebaudengo, M. S. Reorda, M. Violante, and P. Zambolin, "Evaluating the effects of SEUs affecting the configuration memory of an SRAM-based FPGA," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, 2004, pp. 584–589.

[7] P. Graham, M. Caffrey, and P. Cameron, "Consequences and categories of SRAM FPGA configuration SEUs," *MAPLD*, vol. 836, pp. 0–33, 2003.

[8] H. R. Zarandi, S. G. Miremadi, D. K. Pradhan, and J. Mathew, "SEU-Mitigation Placement and Routing Algorithms and Their Impact in SRAM-Based FPGAs," *8th Int. Symp. Qual. Electron. Des.*, pp. 380–385, Mar. 2007.

[9] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "RapidSmith : Do-It-Yourself CAD Tools for Xilinx FPGAs," *Language (Baltim).*, pp. 349–355, 2011.

[10] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11*, 2011, p. 41.

[11] O. Ruano, J. a. Maestro, and P. Reviriego, "A Methodology for Automatic Insertion of Selective TMR in Digital Circuits Affected by SEUs," *IEEE Trans. Nucl. Sci.*, vol. 56, no. 4, pp. 2091–2102, Aug. 2009.

[12] E. Johnson, M. Caffrey, P. Graham, N. Rollins, and M. Wirthlin, "Accelerator validation of an fpga seu simulator," *IEEE Trans. Nucl. Sci.*, vol. 50, no. 6, pp. 2147–2157, Dec. 2003.

[13] M. Violante, L. Sterpone, M. Ceschia, D. Bortolato, P. Bernardi, M. S. Reorda, and a. Paccagnella, "Simulation-based analysis of SEU effects in SRAM-based FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 51, no. 6, pp. 3354–3359, Dec. 2004.

[14] L. Sterpone and M. Violante, "A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 54, no. 4, pp. 965–970, Aug. 2007.

[15] M. Wirthlin, E. Johnson, N. Rollins, M. Caffrey, and P. Graham, "The reliability of FPGA circuit designs in the presence of radiation induced configuration upsets," *11th Annu. IEEE Symp. Field-Programmable Cust. Comput. Mach. 2003. FCCM 2003.*, pp. 133–142, 2003.

[16] G. Asadi and M. B. Tahoori, "Soft error rate estimation and mitigation for SRAM-based FPGAs," *Proc. 2005 ACM/SIGDA 13th Int. Symp. Field-programmable gate arrays - FPGA '05*, p. 149, 2005.

[17] H. Asadi, M. Tahoori, and B. Mullins, "Soft error susceptibility analysis of SRAM-based FPGAs in high-performance information systems," *IEEE Trans. Nucl. Sci.*, vol. 54, no. 6, pp. 2714–2726, 2007.

[18] M. A. Abdul-Aziz and M. B. Tahoori, "Soft error reliability aware placement and routing for FPGAs," in *2010 IEEE International Test Conference*, 2010, pp. 1–9.

[19] L. Sterpone and M. Violante, “A new reliability-oriented place and route algorithm for SRAM-based FPGAs,” *IEEE Trans. Comput.*, vol. 55, no. 6, pp. 732–744, Jun. 2006.

[20] S. Golshan and E. Bozorgzadeh, “Single-event-upset (SEU) awareness in FPGA routing,” *Proc. 44th Annu. Conf. Des. Autom. - DAC ’07*, p. 330, 2007.

[21] K. Huang, Y. Hu, and X. Li, “Cross-layer optimized placement and routing for FPGA soft error mitigation,” *2011 Des. Autom. Test Eur.*, pp. 1–6, Mar. 2011.

[22] B. E. Pritchard, G. M. Swift, and A. H. Johnston, “Radiation effects predicted, observed, and compared for spacecraft systems,” *IEEE Radiat. Eff. Data Work.*, pp. 7–13, 2002.

[23] Altera Corporation, “Robust SEU Mitigation With Stratix III FPGAs,” 2007.

[24] Xilinx Corporation, “Field Programmable Gate Array (FPGA),” 2013. [Online]. Available: http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm.

[25] W. Carter, K. Duong, R. H. Freeman, H. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo, and S. L. Sze, “A user programmable reconfigurable gate array,” in *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1986.

[26] Xilinx Corporation, “The programmable logic data book,” 1996.

[27] Altera Corporation, “White paper - fpga architecture,” 2006.

[28] V. Betz and J. Rose, “Cluster-based logic blocks for FPGAs: Area-efficiency vs. input sharing and size,” in *Custom Integrated Circuits Conference, 1997., Proceedings of the IEEE 1997*, 1997, pp. 551–554.

[29] G.-P. Daniel and C. Maciej, “A tutorial on fpga routing,” University of Massachusetts, Amherst, 2006.

[30] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for deep-submicron FPGAs*. Kluwer Academic Publishers, 1999.

[31] X. Shi, “FPGA Placement Methodologies: A Survey.”

[32] V. Betz and J. Rose, “VPR: A new packing, placement and routing tool for FPGA research,” in *Field-Programmable Logic and Applications*, 1997, pp. 213–222.

[33] S. K. Nag and R. A. Rutenbar, “Performance-driven simultaneous place and route for island-style FPGAs,” in *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, 1995, pp. 332–338.

[34] C. Beckhoff, D. Koch, and J. Torresen, “The Xilinx Design Language (XDL): Tutorial and use cases,” *6th Int. Work. Reconfigurable Commun. Syst.*, no. Xdl, pp. 1–8, Jun. 2011.

[35] M. M. Ibrahim, K. Asami, and M. Cho, “Evaluation of SRAM based FPGA performance by simulating SEU through fault injection,” *2013 6th Int. Conf. Recent Adv. Sp. Technol.*, pp. 649–654, Jun. 2013.

[36] M. A. Abdul-Aziz and M. B. Tahoori, “Soft error reliability aware placement and routing for FPGAs,” in *2010 IEEE International Test Conference*, 2010, pp. 1–9.

[37] L. Sterpone and M. Violante, “A new reliability-oriented place and route algorithm for SRAM-based FPGAs,” *IEEE Trans. Comput.*, vol. 55, no. 6, pp. 732–744, Jun. 2006.

[38] Weifeng Xu, “VPR for Virtex + JBits Interface,” *University of Massachusetts*. [Online]. Available: ttp://www-unix.ecs.umass.edu/~wxu/jbits/.

[39] Xilinx Corporation, “JBits 2.8 SDK for Virtex,” 1999.

[40] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, “The VTR project,” in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays - FPGA ’12*, 2012, p. 77.

[41] E. Hung, F. Eslami, and S. J. E. Wilton, “Escaping the Academic Sandbox: Realizing VPR Circuits on Xilinx Devices,” in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, 2013, pp. 45–52.

[42] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, “Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research,” *2010 18th IEEE Annu. Int. Symp. Field-Programmable Cust. Comput. Mach.*, pp. 149–156, 2010.

[43] A. Sari and M. Psarakis, "Scrubbing-based SEU mitigation approach for Systems-on-Programmable-Chips," in *2011 International Conference on Field-Programmable Technology*, 2011, pp. 1–8.

[44] C.-L. E. Cheng, "Risa: Accurate And Efficient Placement Routability Modeling," in *IEEE/ACM International Conference on Computer-Aided Design*, 1994, pp. 690–695.

[45] Altera Corporation, "Benchmark Designs For The Quartus University Interface Program (QUIP) Version 1.1." 2010.

[46] Xilinx Corporation, "ISE Design Suite." 2012.

[47] K. Morgan, M. Caffrey, P. Graham, E. Johnson, B. Pratt, and M. Wirthlin, "SEU-induced persistent error propagation in FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 52, no. 6, pp. 2438–2445, Dec. 2005.