POLITECNICO DI TORINO

MASTER's Degree in Computer Engineering

MASTER's Degree Thesis

Enhancing the Robustness of System on FPGA by Routing Isolation

Supervisors

Prof. Luca STERPONE

PhD. Corrado DE SIO

Candidate

Davide NICOLINI

DECEMBER 2023

Abstract

Due to their high performance and flexibility, FPGA are gaining popularity as a solution for space application. However, electronic system operating in space are subjected to interaction with ionizing radiation that is a source of malfunctions in the device.

SRAM-based FPGA are particurarly sensitive to Single Event Effect that can lead to failure due to configuration memory corruption. To cope with this problem N-Modular Redundancy (NMR) is one of the mitigation approach commonly adopted. In this thesis, a new methodology for increasing reliability of circuit hardened by modular redundancy technique is proposed. The methodology statically analyze the implementation of a circuit implemented on the programmable hardware in order to identify single points of failure that can propagate faults to different replication-domain leading to application failure.

Acknowledgements

Grazie mille dell' enorme supporto che mi hanno dato i miei supervisors Luca Sterpone e Corrado De Sio durante tutto il lavoro della tesi, sempre disponibili a darmi consigli ed aiutarmi quando ne avevo bisogno.

Un enorme ringraziamento va alla mia famiglia per avermi dato sempre fiducia e spronato a dare il meglio di me stesso, e che è difficile esprimere la gratitudine in poche righe.

Grazie a tutti del ASAC lab per la compagnia, i pranzi insieme, e il supporto che mi hanno offerto senza esitazione, fino all'ultimo.

Infine grazie a tutti i miei amici che mi hanno accompagnato in questo viaggio, dall'inizio dell'università fino alla fine, specialmente i gruppi Pandawave, RsaR e Drop Table, sia all'università che fuori.

Table of Contents

List of Tables

List of Figures

Acronyms

FPGA

Field-Programmable Logic Array

TMR

Triple Modular Redundancy

SEU

Single-Event Upset

LUT

Look-up Table

ECC

Error Correction Codes

SRL

Shift Register LUT

Chapter 1

Introduction

FPGAs are reconfigurable integrated circuits that offer a flexible platform for digital circuit design and implementation. They consist of a matrix of programmable logic blocks interconnected through programmable routing resources. This architecture allows users to configure the FPGA according to specific application requirements, enabling the implementation of diverse digital logic functions, including arithmetic operations, signal processing, and complex control systems.

For those characteristics, FPGAs are proficient at delivering exceptional results pivotal in various industries, including space exploration. But in space, electronic systems face a constant threat from radiation, which can induce errors like bit flips, altering digital data. FPGAs are particularly crucial in this realm due to their reconfigurable nature, enabling them to adapt and mitigate such errors.
The impact of radiation-induced bit flips on FPGAs in space applications is a critical concern, demanding robust designs and error-correction mechanisms to ensure the reliability and functionality of systems operating in these challenging

environments.

Specifically, a bitflip in the configuration memory can alter the circuit structure, which can cause malfunctions. This is why systems such as Triple Modular Redundancy (TMR) are used, which consist of three systems performing the same identical task independently.

The system then compares the results; if one module produces a different outcome due to an error or fault (such as radiation-induced bit flips), a voting mechanism among the three modules allows the system to identify and discard the erroneous result, ensuring the system functions correctly despite potential faults. TMR significantly reduces the likelihood of system failure by providing a robust mechanism to handle errors.

But even in this case, a Single-Event Upset (SEU) may lead to malfunction by short-circuiting two different modules, propagating an error from one system to another, making the TMR system useless.

1.1 Motivation

The primary objective of this thesis is to devise a methodology for pinpointing all components susceptible to Single Event Upsets (SEUs) within a specific FPGA design incorporating Triple Modular Redundancy (TMR). Achieving this goal necessitates the creation of software capable of extracting comprehensive architectural information from the targeted FPGA. The choice of Vivado as the FPGA development and programming software introduces a challenge, as it is known for its sluggish performance when tasked with gathering such architectural details.

This characteristic poses a particular obstacle to conducting the in-depth study required for identifying and addressing vulnerabilities in the design

Collecting the information, however, was not trivial, given the amount of data it was not simple to transfer everything to the database, but had to use generalization techniques wherever possible. In fact, a lot of the data was redundant, allowing only a fraction of the data to be stored.

This thesis delves into two distinct architectures, namely Versal and Ultrascale+, aiming to showcase the versatility of the proposed approach. Ultrascale+ serves as a platform where established methodologies and research advancements facilitate the evaluation of the approach. On the other hand, Versal introduces novel challenges due to its innovative architecture and the inherent complexity in its utilization and validation. Consequently, the applicability of the approach to Versal signifies a notable strength, representing one of the initial strides toward enhancing the reliability of systems built upon this cutting-edge architecture.

After constructing the database, it is possible to analyze all the critical parts of a given implementation and eventually change the design to reinforce the stability of the system.

Due to the configurable nature of FPGAs, simulating radiation-induced bitflips becomes feasible by programming the FPGA with a faulted bitstream. This approach enables systematic testing of diverse fault combinations through multiple

injections, with the flexibility to repeat specific injections as needed. This methodology provides a means to thoroughly evaluate the reliability of the design under varying fault scenarios.

Chapter 2

Background

2.1 Field-programmable gate array

2.1.1 FPGA description

A Field-Programmable Gate Array (FPGA) is a reconfigurable integrated circuit that allows users to customize and program its functionality after manufacturing. Unlike Application-Specific Integrated Circuits (ASICs), FPGAs offer flexibility and adaptability for various digital circuit designs.

At the core of an FPGA are programmable logic blocks, each containing elements like look-up tables and flip-flops. These blocks can be configured to perform specific digital logic functions. The interconnection between these blocks is configurable, allowing designers to create custom digital circuits by defining connections. While the architecture is different for every FPGA, the core structure is composed of two main elements: the logic block and the switch matrix. The former is where

all the logic is computed, and will define what the FPGA will do. Specifically, is composed of Look-up Tables (LUT) and Flip-Flops: The LUT is a fundamental building block that can be programmed to implement specific logic functions, and the flip-flops store the intermediate and final results of these logic operations.

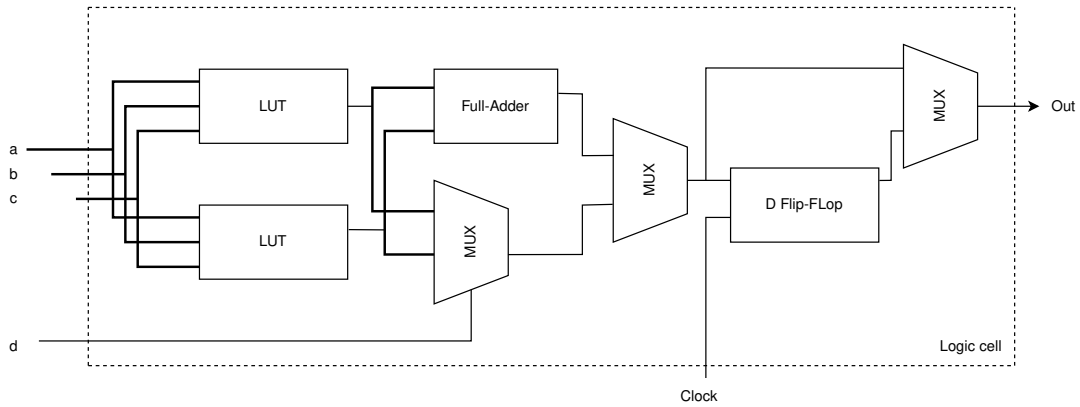

Figure 2.1: FPGA's Logic cell schema

The latter is the component responsible for managing the routing of signals between various logic elements within the device and consists of a network of switches that can be dynamically configured to establish connections between different inputs and outputs. These connections are pivotal for creating the desired digital circuit within the FPGA. When a user programs the FPGA, the switch matrix adapts to the specified configuration, enabling the interconnection of logic elements in a way that aligns with the intended functionality.

Programming an FPGA involves creating a hardware description using a Hardware Description Language (HDL), such as Verilog or VHDL. This description is

synthesized into a configuration bitstream, a binary file specifying the logic configuration. This bitstream is then loaded onto the FPGA, defining its functionality.

FPGAs find applications in digital signal processing, communication systems, embedded systems, and rapid prototyping. They excel in scenarios where adaptability and quick development cycles are crucial. Despite their flexibility, FPGAs present challenges related to power consumption, design complexity, and cost, factors that must be carefully considered in their application

2.1.2 FPGA in the modern industry

The realm of digital electronics is marked by a diverse array of devices catering to specific needs and functionalities. Among these, Field-Programmable Gate Arrays (FPGAs) stand out as versatile and reconfigurable integrated circuits. As technology advances, the choice between FPGAs and other electronic devices becomes a critical consideration for designers and engineers. This exploration delves into the comparative landscape of FPGAs against other electronic devices, unraveling the distinctive features, advantages, and applications that set FPGAs apart from conventional alternatives. From Application-Specific Integrated Circuits (ASICs) to microcontrollers and digital signal processors, understanding the strengths and limitations of each category is paramount in making informed decisions for diverse electronic design endeavors.

The primary distinction between FPGAs (Field-Programmable Gate Arrays) and ASICs (Application-Specific Integrated Circuits) lies in their programmability.

FPGAs are reconfigurable and can be programmed and modified after manufacturing, offering flexibility but typically at the cost of power efficiency and performance. In contrast, ASICs are custom-designed for specific tasks during manufacturing, optimizing for performance and power efficiency but sacrificing the adaptability inherent in FPGAs. This means that ASICs will prove more useful, especially in a context where there is a need for high-volume production and long-term commitment to the design.

2.2 Radiation-inducing effects on Electronics

2.2.1 Single-Event Upset

Radiation effects on electronic systems are a critical consideration in aerospace and high-reliability applications. Two primary forms of radiation impact are Single Event Effects (SEE)[1] and Total Ionizing Dose (TID)[2][3]. SEE involves the instantaneous alteration of a single memory or logic cell, while TID accumulates damage over time. Radiation sources encompass solar and cosmic radiation, as well as particles trapped in the Earth's magnetosphere. Understanding and mitigating these effects are paramount for the robust performance of electronic components in challenging environments.

In the context of space exploration, satellites, and spacecraft requiring dependable electronics is a crucial task. These systems face a unique challenge known as Single Event Upsets (SEUs), incidents where a single energetic particle from space momentarily disrupts the normal functioning of electronic components. This phenomenon is not confined to the depths of space but extends to other contexts,

including high-altitude flights, particle research facilities, and even advanced avionics systems. Understanding SEUs and their potential consequences is crucial in these environments, where the reliability and integrity of electronic systems can determine the success or failure of a mission.

Within the realm of electronics, SEUs [4][5][6] pose a particularly intriguing challenge for FPGAs, especially in contexts where these devices are exposed to cosmic radiation or high-energy particles.

FPGAs, known for their adaptability and reconfigurability, are key components in various applications, from satellite systems to aerospace technology. However, this flexibility comes with a vulnerability to SEUs, as the reconfigurable nature of FPGAs makes them more susceptible to temporary disturbances induced by energetic particles.

In the context of space missions, where FPGAs are extensively used for their versatility, the impact of SEUs can be critical. A single energetic particle striking an FPGA in a satellite's circuitry could potentially lead to erroneous configurations, affecting the proper functioning of the device and, consequently, the mission's success.

An SEU can manifest in diverse scenarios. Often, it strikes an unused or redundant section of the FPGA, causing minimal impact. However, in certain instances, it may cause a system to crash. The most concerning outcome, though, is the potential introduction of a silent error. This subtle anomaly could result in incorrect data or delayed malfunctions, posing a significant challenge to the integrity and

functionality of the system.

As we venture further into space exploration and rely on FPGAs for their adaptability, the mitigation of SEUs becomes a critical consideration in ensuring the reliability and success of electronic systems in these demanding environments.

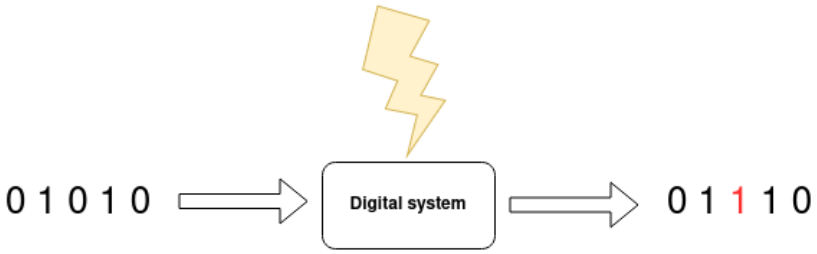

Figure 2.2: Bitflip caused by SEU

2.2.2 Radiation Hardening Techniques

To address this vulnerability, designers working with FPGAs in space applications implement strategies such as Triple Modular Redundancy (TMR) and Error Correction Codes (ECC). TMR involves triplicating key components and comparing their outputs to detect and correct errors induced by SEUs. ECC, on the other hand, introduces redundant bits in data storage, enabling the identification and correction of errors.

The decision to use TMR over ECC in FPGA design hinges on factors such as the criticality of the system, real-time performance requirements, simplicity of implementation, and the nature of potential errors. TMR is often preferred in applications demanding the utmost reliability, especially in critical systems with low-latency needs. Its simpler implementation may be advantageous, particularly

when addressing specific, known types of errors is crucial. However, TMR comes with increased resource requirements, potentially impacting power consumption and costs. The choice between TMR and ECC depends on the specific demands of the application and the desired balance between fault tolerance and resource overhead.

The thesis employs TMR as the primary error correction method. Additionally, an exploration is conducted into an alternative design featuring two components with an XOR gate comparing their outputs, with the ultimate goal of only detecting an error.

These designs, when impacted by SEUs, exhibit a high likelihood of either correcting or detecting errors effectively. However, there exist marginal cases where faults may persist, specifically when an SEU propagates an error from one domain to another. In such instances, the error might be treated as reliable data, potentially posing a risk to the mission.

2.3 Modern FPGA Architectures

The evolution of FPGA systems has been marked by significant advancements, shaping their capabilities and applications over the years. In the early stages, emerging in the 1980s, FPGAs introduced the concept of post-manufacturing hardware configuration. Throughout the 1990s, these devices became commercially available, featuring reconfigurable logic elements.

As the technology progressed into the late 1990s and early 2000s, FPGAs

witnessed an increase in logic density, accommodating more complex designs. This evolution continued with the introduction of advanced functionalities, including embedded memory blocks in the mid-2000s, making FPGAs applicable to a broader array of use cases.

In the mid-2000s and beyond, FPGAs found a place in high-performance computing (HPC), leveraging their reconfigurable nature for parallel processing. This period also saw the incorporation of high-speed serial interfaces like PCIe, enhancing data transfer rates.

Moving into the 2010s, FPGAs began integrating specialized processing elements, such as DSP blocks, enabling efficient signal processing tasks. The demand for accelerated artificial intelligence (AI) workloads further propelled the adoption of FPGAs due to their parallel processing capabilities.

In the late 2010s, a notable shift occurred with the integration of processor cores, leading to the emergence of System-on-Chip (SoC) FPGAs. This integration seamlessly combined reconfigurable logic with traditional processor functionalities, opening new possibilities for versatile and efficient applications. The continuous evolution of FPGA systems reflects their adaptability and relevance in the rapidly advancing field of digital hardware.

In this thesis, two specific architectures were used: Ultrascale+ and Versal.

2.3.1 Versal

Versal is an innovative family of FPGAs developed by Xilinx, representing a significant advancement in reconfigurable computing technology. It introduces a holistic approach by integrating adaptive hardware, specialized processing units, and advanced connectivity features into a unified platform. This architecture is

designed to cater to a diverse range of applications, from traditional FPGA use cases to complex computing tasks, including artificial intelligence and high-performance computing.

At its core, Versal combines programmable logic with dedicated processing elements, including AI engines and Arm Cortex processors. The adaptability of the programmable logic allows developers to configure and reconfigure the hardware according to specific application requirements. The inclusion of AI engines enhances the platform's capability to efficiently handle machine learning and neural network workloads.

Versal FPGAs are characterized by their flexibility, scalability, and ability to deliver high performance across various domains. With features like versatile connectivity options, optimized power efficiency, and support for advanced technologies, Versal offers a comprehensive solution for developers seeking to address the challenges of modern computing and accelerate innovation in a wide range of industries.

2.3.2 Ultrascale+

Ultrascale+ is a highly advanced family of FPGAs developed by Xilinx. Building on the success of previous FPGA generations, Ultrascale+ represents a significant evolution in terms of performance, capacity, and versatility. These FPGAs feature a scalable architecture that includes reconfigurable logic, high-speed transceivers, and dedicated processing elements.

Ultrascale+ FPGAs are known for their high logic density, enabling the implementation of complex designs. They incorporate features like high-speed serial

interfaces, DSP (Digital Signal Processing) blocks, and memory resources, making them suitable for a wide range of applications, including telecommunications, networking, data centers, and more.

One notable aspect of the Ultrascale+ family is the integration of processing units, including Arm Cortex-A53 or Cortex-R5 processors. This combination of programmable logic and embedded processing cores provides a versatile platform for both hardware acceleration and software-based tasks.

Furthermore, Ultrascale+ FPGAs support advanced technologies such as 3D IC integration, enabling better performance and power efficiency. The family is designed to address the increasing demands of modern computing applications, making it a preferred choice for developers seeking high-performance and adaptable solutions in the FPGA space.

Chapter 3

State of the Art

In the rapidly evolving landscape of FPGA (Field-Programmable Gate Array) technology, the reliability and robustness of these programmable devices are critical considerations, particularly in safety-critical and mission-critical applications. Single Event Upsets (SEUs), arising from radiation-induced effects, pose challenges to the integrity of FPGA-based systems[7].

This chapter delves into the state of the art concerning SEUs in FPGAs, with a specific focus on routing architectures, domain isolation strategies, and the application of statistical analysis methods.

As we navigate through the current research and advancements, our exploration aims to provide a comprehensive understanding of the intricate interplay between SEU vulnerabilities, routing structures, domain isolation techniques, and the evolving landscape of statistical analysis in FPGA design.

In the thesis work, one fundamental aspect is the reinforcement of the design via TMR and custom layout, and works like [8] and [9] show that this approach

could lead to benefits.

Reliability analyses are a fundamental part of understanding how the reliability of a board or a design could change. Different methods for various applications could be done, such as the work [10] via software, which simulates the SEUs by manipulating the bitstream and by programming the FPGA with the faulted bitstream, with an emulation of the board and manipulating the signals or directly with dedicated infrastructure to irradiate the board [11] [12], simulating the same radiation an FPGA would be subjected in space.

For custom injection faults, utilizing software emerges as a straightforward method, demanding only a computer and not dedicated infrastructure. While the latter is considered the most realistic approach, the availability of these tools is limited due to the substantial costs associated with the required instruments.

Libraries such as PyXEL[13] significantly facilitate the examination of the bitstream, enabling the direct simulation of faults on the physical board, providing an interface with Vivado, and much more, all in Python.

Various studies are done on the bitstream not only for direct manipulation but also to recover the original netlist, an information that is not directly carried inside the bitstream, but needs a grade of reverse engineering. A study [14] recovered the exact netlist and also the code with the goal of spotting eventual Hardware Trojan injected. But also works for other company's bitstreams are done for the same purpose, pointing out the need to reverse the bitstream [15]

Focusing on SEUs and their potential implications for FPGAs, various studies have sought to comprehend the impact of these effects on diverse and contemporary technologies. Additionally, these investigations delve into the effects of Single Event Transients (SETs) to gain a comprehensive understanding of their consequences. In work like [16] a study of those events is applied to a 7 nm FinFET technology.

One particularly vulnerable part to an SEU is the configuration memory since it can reassemble the circuit, and cause not intended behavior or silent errors, which can corrupt the data without being aware. Preventative techniques could be applied to improve the general robustness in case of those events: [17] presents a passive configuration memory scrubbing scheme.

A multitude of FPGA models exists, each with its distinct features, advantages, and limitations. Some are better suited for specific tasks than others. Consequently, the architecture is crafted not only to enhance overall performance but also to excel in particular applications. This implies that an SEU's impact on a board could differ remarkably from others. To cover this issue works are done [18] with the specific intent of understanding the difference between different FPGAs.

Chapter 4

Experimental Analysis

4.1 Versal architectural analysis

This thesis aims to investigate the impact of SEUs on a specific FPGA architecture, namely Versal. Versal, introduced by Xilinx, represents a cutting-edge family of Adaptive Compute Acceleration Platforms (ACAP). Unveiled in 2019, Versal signifies a departure from traditional FPGA designs, integrating programmable logic with specialized processing elements on a single chip. This architecture, blending versatility with AI and machine learning acceleration, is designed to address diverse compute-intensive workloads. Versal embodies Xilinx's vision for adaptable and high-performance computing solutions in the modern era.

The board is fairly recent, meaning that not much documentation, also from Xilinx themselves, was found. So, to proceed in this task I also studied the board using the tool Vivado, which allowed me to analyze directly the circuitry of the board. With the combination of the tool and the documentation found[19] it was

possible to reasonably understand the architecture.

The design of Versal is characterized by its integration of diverse components on a single chip. It combines programmable logic, adaptive engines, AI and machine learning accelerators, memory controllers, high-bandwidth interfaces, and security features. The architecture is designed to be highly versatile and adaptable, allowing for a unified solution to address a wide range of compute-intensive applications. The study will not focus on the AI chips, not only because little to nothing can be found as documentation, but also because are not relevant to the thesis work. I will instead focus on the Configurable Logic Blocks (CLB) structure.

4.1.1 Difference between previous generation

The are various differences between the versal and the previous generation but the main and most relevant could be summarized as:

- Logic capacity quadruplicated, from 8 LUT / 16 flip-flops to 32 LUT/ 64 flip-flops
- Dedicated LUT-LUT cascade path inside the CLB, to reduce lag and external routing requests
- The Super Long Line connections (SLL) are now inside the CLB
- Three outputs per LUT/FF instead of four
- Only a single type of CLB of which half of the LUTs support LUTRAM and SRL configurations

4.1.2 CLB description

The structure of the CLB is composed of two SLICEL, two SLICEM, and one CLB interconnect. Each SLICEM and SLICEL have 8 LUTs, a Carry Lookahead, and flip-flops. The difference between SLICEM and SLICEL is in the LUT: specifically SLICEL's LUT are SRL/LUTRAM capable.

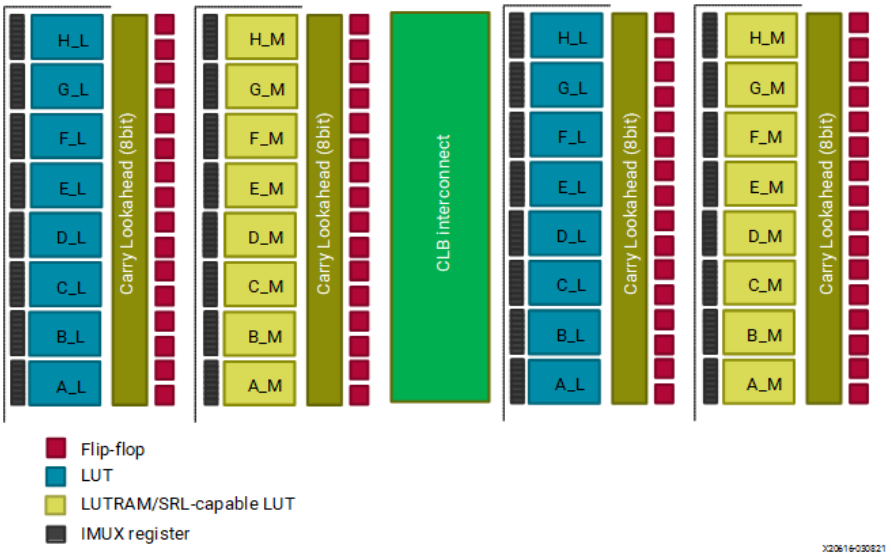

Figure 4.1: CLB Block Diagram [19]

LUT

Every LUT in the CLB is 6LUT, composed of 4 4LUT. Each one can be used in mode 6LUT or dual 5LUT. The multiplexers arranged up and down are new to the versal architecture: in fact, they can allow cascade connections between LUTs, enables two-LUT function for up to 6 input, and are also used for the carry logic.

Every LUT have 4 outputs: *prop*, used for the carry logic, $O6$ output for the standard mode 6LUT, $O5\_1$ and $O5\_2$ the two outputs for the dual mode 6LUT.

Adjacent LUTs are cascaded in one direction (From A ->H): more specifically the output *O*6 is linked to the input *casc* of the next LUT, but every two LUTs the output is not linked directly but it goes through a carry-lookahead.
The detailed schema is shown in figure 4.1. The two multiplexers near the top and bottom of the diagram are new to Versal architecture. They are static memory cell-controlled muxes. These multiplexers are used to carry logic paths, cascadable LUT -> LUT connections (O6 -> A5), and to enable dual LUT functions of up to six inputs (five in prior architectures).

Figure 4.3 represents the cascade connections for one half of the CLB (red arrows). The solid red lines represent actual wires. The dotted red lines represent logical connections that only exist in cascade mode

IMUX Register

Input Multiplexer (IMUX) registers are integrated into the CLB. These registers are present on specific CLB inputs, totalling 192 IMUXs and 64 bypasses. Positioned near the interconnect/CLB boundary, the IMUX registers can be bypassed. They provide support for a subset of CLB flip-flop features, each equipped with clock enable and synchronous or asynchronous reset capabilities. Notably, these registers lack readback/writeback functionality and do not possess synchronous/asynchronous set capability. While initialization is programmable, the options are limited to either init=0 or init=data input (with no init=1 option).

Shift Register

Similar to the previous architecture, with 6-LUT supporting 32-bit SRL and the 5-LUT/6-LUT pair supporting 16-bit SRL. The 32-bit SRL is connected by a dedicated shift chain using the SIN and SOUT pins of the 6-LUT, which is present only in the SLICEM. Also, each slice has the SIN input and SOUT output are connected to the CLBs below and above, respectively.

Storage Elements

A CLB contains 64 flip-flops, each of which can be influenced by one of several sources:

- **O5/sum:** Derived from the LUT output in dual LUT mode and the SUM output when utilizing carry logic.
- **O6/cout:** Originating from the LUT output in 6LUT mode and the Carry Out output when employing carry logic (only for even bits).
- **Bypass:** Each flop is associated with an individual bypass signal.
- **Miscellaneous:** Receives inputs from super long line (SLL) connections or the carry-out of odd bits.

Control Signals

In each CLB there are 4 clocks, 4 SRs (set/reset), and 16 CEs (clock enable). Clocks and SR are shared among all Flip-Flops in a Slice.

Storage Elements

There are 64 slice flip-flops in the CLB. Each can be driven by one of multiple sources:

- O5/sum: LUT output in dual LUT mode and SUM output when using carry logic
- O6/cout: LUT output in 6LUT mode and Carry Out output when using carry logic (only on even bits)
- Bypass: each flop has an independent bypass signal
- Miscellaneous: inputs from super long line SLL connections or carry-out of odd bits

Control Signals

In each CLB there are 4 clocks, 4 SRs (set/reset), and 16 CEs (clock enable). Clocks and SR are shared among all Flip-Flops in a Slice as shown in 4.2.

Interfaces

Each interface connects to other interfaces or to the LUT/FF complex with a range of +-15 rows (For every 4 interfaces there is an empty tile, so +-12 interfaces, or 3 groups of interfaces) and +-24 columns (every 4 columns there is an interface so up to +-6 interfaces)

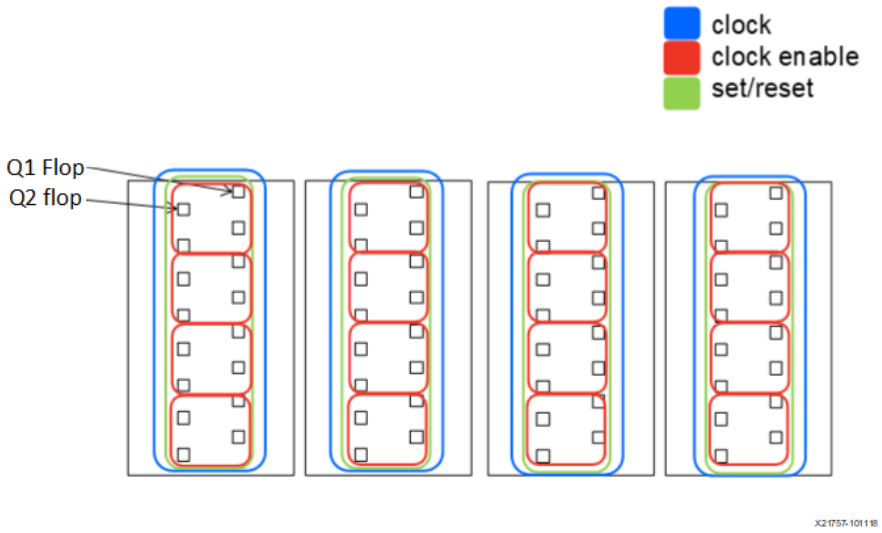

Figure 4.2: Control Signals [19]

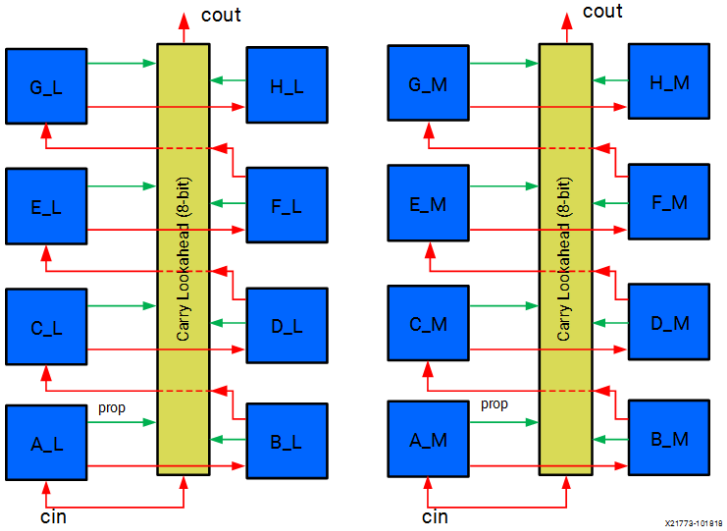

Figure 4.3: LUT Cascade[19]

4.2 FPGA Database

As already mentioned in 1, part of the thesis work was to create software that would gather architectural data of the Versal (namely PIPs, Nodes, Tiles, etc...)

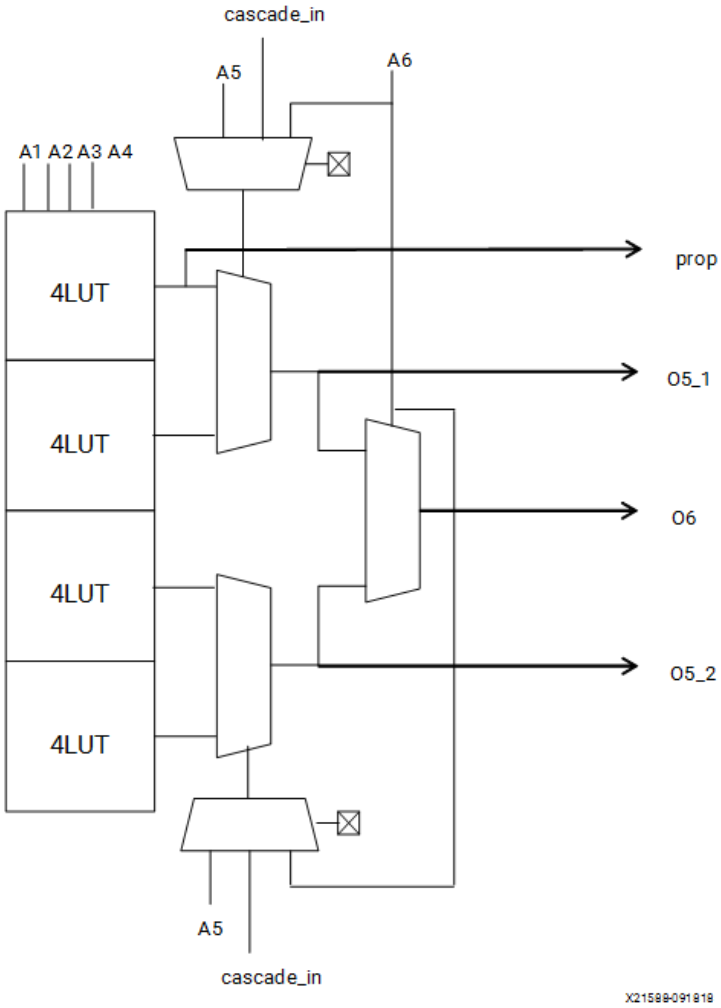

Figure 4.4: LUT Features[19]

into a faster, accessible local file, that could be queried without worrying about speed and missing or redundant data. In fact, the tool AMD provides to interface and program their FPGAs, Vivado, is notably computationally expensive, making extensive use of computer resources, implying that intricate queries might exceed the acceptable time frame for our objectives.

To make this task feasible, not only was necessary the comprehensive study done

in 4.1, but also required a way to interface with Vivado, since all the requested information can be retrieved from there. To help me with this, a Python library called PyXEL was used.

PyXEL has many functions that vary from interfacing to Vivado, calling directly TCL scripts and returning the result in Python format, to giving utilities for FPGA's bitstream. Those are the features mainly used for this work but are not limited to those.

With all preparations complete, coding commenced to store architectural data in a JSON file. For each object requiring storage, individual files were generated, featuring data structures that could vary based on necessity, deviating from the original structure employed in Vivado as per specific requirements.

4.2.1 Database Software

Tiles

One of the first tasks for the construction of the database was to understand how interfaces, one of the two kinds of switch matrix, are linked to each other via nodes: specifically, since every interface have a set of coordinate, the list of the other interfaces that could be reached, and with what node, using a system of relative coordinate.

Every name of the tile is composed as "INT_X<X coord>Y<Y coord>", where <X coord> and <Y coord> are the coordinates used by Vivado to identify the CLB, and it is greatly helpful for this particular job, since not only I already have

to coordinate for the tile directly in the name, but also mean I can retrieve a specific tile just from their coordinates.

For example, if I have an interface with the name *INT_X200Y100* which is linked to another named *INT_X210Y100*, the information that I store is only X: 10 and Y: 0 since I have already the coordinate for that tile. This was done because the goal was to find a subset of interfaces with identical sets of relative coordinates, avoiding redundancy in the database.

However, after some testing, I noted that the coordinate system provided by the name changed every tile, even the empty ones. This was not what was needed since if an interface was linked to another interface there could be a situation where an identical tile, in terms of the same node to linked tiles, may have different coordinates caused by an empty tile.

I then switch the coordinate system to another, always provided by Vivado, defined as INT_TILE_X and INT_TILE_Y, that correspond to the coordinates of the CLB instead: every component inside the same CLB have the same coordinate, counting only CLBs, ignoring then empty and other kind of tiles.

By employing this system, it became impossible to extract coordinates directly from the name; instead, retrieval had to be done through Vivado using a call. Given the substantial number of calls needed for coordinates, it became essential to establish a simple database mapping each tile to its corresponding integer coordinate system.

Not every tile on the board underwent analysis; instead, the focus was directed solely towards those situated within a preselected clock region chosen arbitrarily.

This deliberate choice aimed to mitigate redundancy, given that the tiles exhibited nearly identical connections. In certain exceptional scenarios, such as when a tile bordered the edge of the board, it could lack nodes connecting to tiles situated beyond that boundary, but that was not a source of issues since the coordinate range of the tiles was known, meaning that if a node would lead to a tile outside the boundary, it simply does not exist.

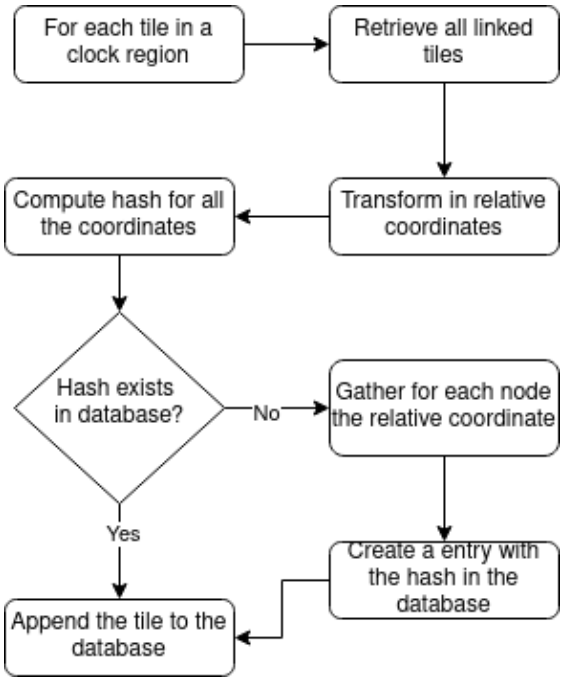

Figure 4.5: Flow for retrieving data from tiles

After receiving the list of names, retrieving the list of coordinates proved to be straightforward, given that they were directly in the name.

But what was missing was what node could lead to what tile, because despite this information being processed inside Vivado during the script to print the list of tiles, everything that is not the output is not saved, requiring another query, that would inevitably slow the process.

But this was exceptionally time expensive, requiring to iterate to all the tiles of one clock region was proven to not be feasible, also considering the common possibility of changing the algorithm or the data retrieved, or simply just an error, that would require running again the script.

So an optimisation of the code was fundamental for the prosecution of the work. The program's bottleneck was the retrieval of the tile connected to a node, iterated for each node within a specific tile. Not only was the non-trivial number of nodes in a tile a factor but the performance was further impeded by the slowdown caused by the repeated TCL calls in each iteration.

To cope with this problem was necessary to avoid if possible calling all this iteration, and this was feasible due to a repetition of coordinate pattern between the tiles: although there were variations in tiles with different nodes and linked tiles, excluding marginal situations, all the tiles could be consolidated based on identical relative coordinates.

After obtaining the linked tiles, before gathering all the data about what node leads to what tile, I have the list of all the tiles ordered alphabetically, transform every tile in its relative coordinate, put in a tuple and then hash the value. In this way, tiles that share the same set of relative coordinates will have the same hash, meaning I don't need to compute the expensive call to associate every node to their relative coordinate.

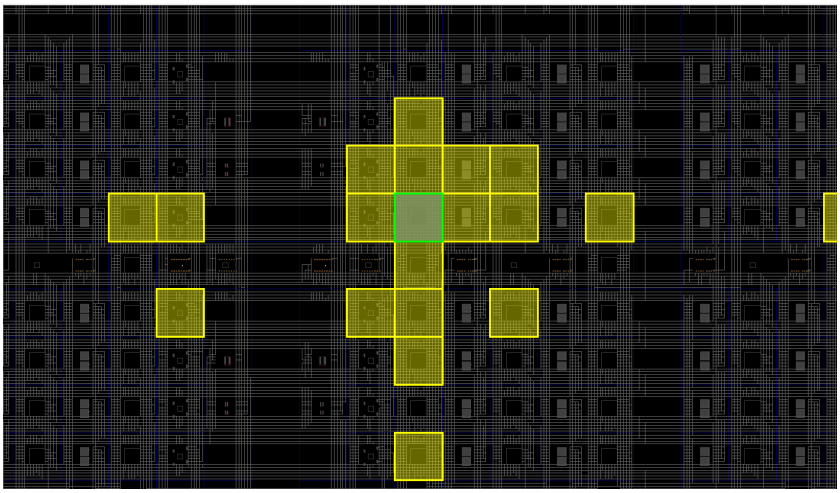

Figure 4.6: Some of the tiles (yellow) connected to a single tile (green)

Nodes

Gathering data for nodes, was a simpler task, but still not trivial: this was because the name of each node is composed of two parts, separated by a '\' where the first part is the name of the tile where the node started (carried the signal from), and the second part is the name of the nodes.

This means that a node could have the same second part of the name, but a different origin tile (in this case it's because one of the nodes starts from the analysed tile) and this should be addressed (Fig. 4.7).

Additionally, it's crucial to note that a single node could terminate in multiple tiles. The focus is exclusively on nodes leading to either a CLE block or an Interface block, as these are the entities responsible for routing.

The last necessary differentiation was separate input and output nodes but this was not a problem since Vivado already provided a parameter to filter an object in one of those categories.

The end result was a list of nodes separated by input and output (uphill and

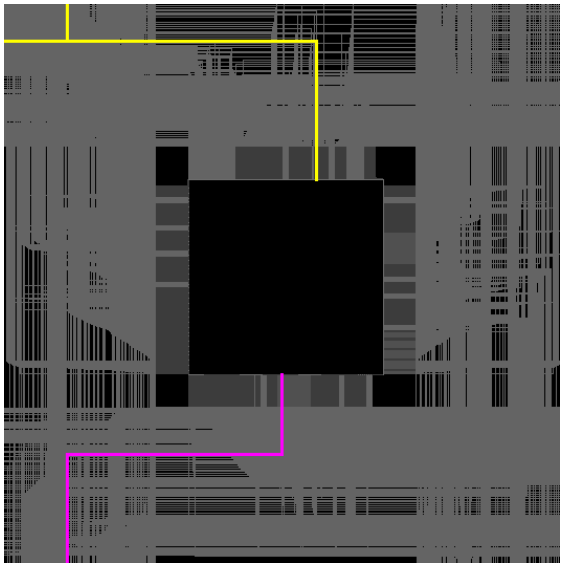

Figure 4.7: A Interface Block with two nodes with the same second part of the name

downhill in Vivado terms) and identified by the generic name (the name without the tile part) listing all the destinations, if output, or showing the source, if input, using again the relative Interface coordinates, and the type of the tile.

Wires

While there was no direct use for wires, mapping them helped greatly the overall work because they are a junction where all the PIPs start and/or end.

This, as explained better in 4.4, is fundamental data needed to conduct the study on problematic PIPs.

A wire may be categorized as either 'Bounce' or not. In the former scenario, it serves solely to redirect PIPs, remaining unattached to any node and featuring both input and output PIPs. In the latter case, the wire functions either as an

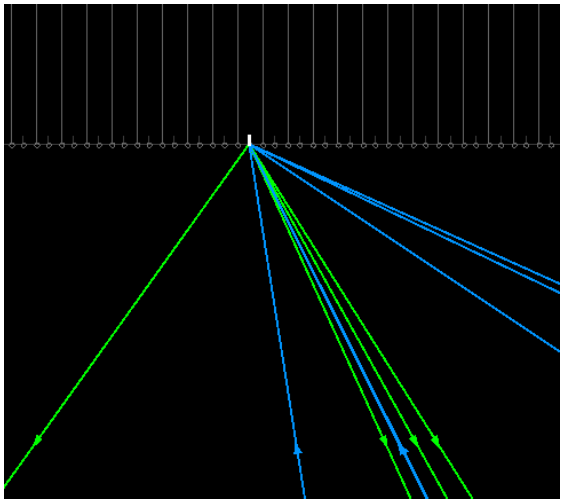

Figure 4.8: Bounce wire with input(uphill) PIPs in blue and output (downhill) PIPs in green

input or output for PIPs, exhibiting a unidirectional flow. This distinction arises from the association with a node, determining whether the wire sends or receives signals.

Gathering data for wire was straightforward, with no particular considerations about optimisations or complex queries.

The result was stored separated by the type of tile analysed (INT and CLE), identified by the generic name, listing the PIPs connected separated by the direction and the eventual node attached.

PIPs

The last component needed for the thesis work is the PIP: the PIP lays inside the switch matrix and is simply a connection between one wire to another, and is the one responsible for routing.

To elaborate further, given the existence of multiple PIPs per wire, when a PIP is activated at the input, the signal can be transmitted to a wire connected to the corresponding PIP. The receiving wire may either act as a bounce, in which case the process is repeated, or it may transfer the signal to a node leading to another switch matrix or a logic block.

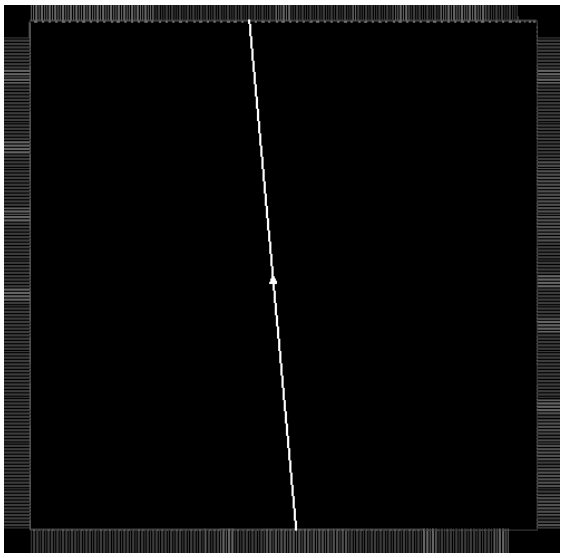

Figure 4.9: A single PIP inside a switch matrix

Inside a board, there are millions of PIPs, but they share the same connections between tiles of the same type, permitting to reduction of the redundancy in the data collection.

By choosing two tiles beforehand one can retrieve all the needed data.

The data stored was similar to the one of the wires, where a single PIP is stored for uphill and downhill respectively a list of PIPs and the attached wire.

It is noted that unlike wires, uphill and downhill PIPs are the following and previous to a queried PIP, meaning that could belong to another switch matrix

concerning the tile of the original PIP.

Net

To find problematic PIPs of a circuit in a specific board is necessary to store all the custom routing of the programmed circuit, called Net.

A Net is an aggregate of nodes, wires, and PIPs that compose a routing defined previously with HDL, and they can be hierarchical, meaning that a net could be composed of multiple smaller nets.

As I will specify better in 4.3 the study and the data collection will not focus on the whole circuit (or the highest net in the hierarchical level) but on specific sub-nets that represent modules of my interest.

This is also caused by nets not relevant to the routing such as ground or the power. Despite being composed of various objects, I store in the database only PIPs and wires, associated with a specific net, that correspond to a domain.

To improve the speed of the data collection, I eventually opted to retrieve only PIPs and then use the PIPs database to obtain the wires, since the information was already stored.

4.3 Design study

The focus of the thesis is to study and find any problematic PIPs related to specific designs.

Those designs are composed of two or three identical modules, running the same program with the goal of increasing robustness in case of fault.

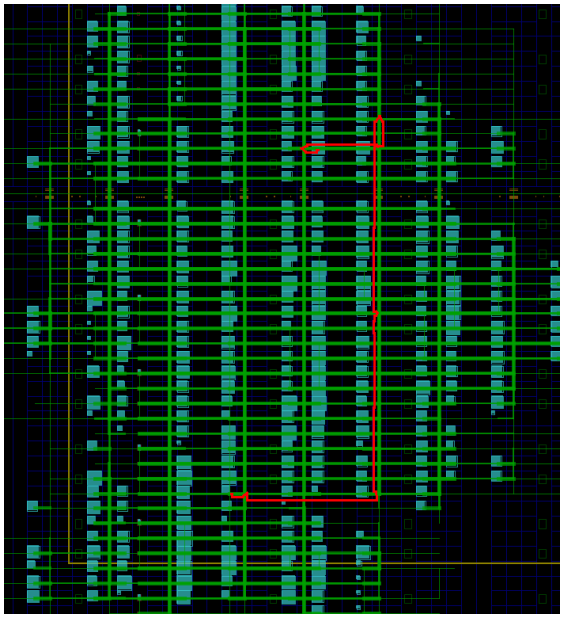

Figure 4.10: A module composed of multiples nets. A single net is highlighted in red

The two types of design used are a Triple Module Redundancy (TMR) and two modules with an XOR gate at the end, and the module used is a processor called Neorv32[20], while also was studied for a short period the B12[21].

4.3.1 TMR

Triple Modular Redundancy (TMR) is a fault-tolerant technique employed in digital systems, particularly in safety-critical applications. In TMR, a computation or data is triplicated, and three identical circuits operate concurrently. The system produces a consistent output as long as at least two of the three components agree; if one component deviates due to a fault or error, the other two can outvote it, ensuring reliability and error detection. TMR is commonly used in scenarios where high reliability and fault tolerance are paramount, such as in aerospace, automotive,

and critical infrastructure applications.

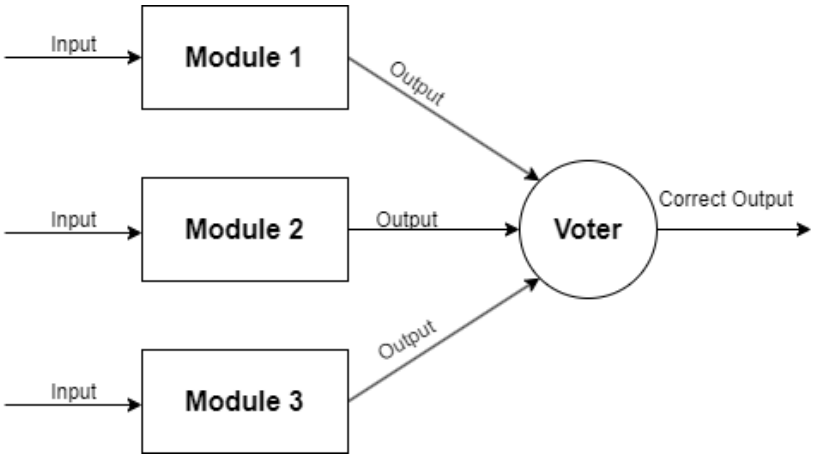

Figure 4.11: TMR schema

This design allows a single module to faults without affecting the result, increasing the overall reliability, meaning that to obtain a wrong output, faults in at least two modules are required.

The identification of problematic PIPs is particularly critical since a short-circuit PIP could transfer the error of a faulted domain to another, or a fault directly in the voter.

out1	out2	out3	voter output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 4.1: Voter truth table

4.3.2 XOR

Another design subject to study was an XOR gate that would compare the results of the modules, detecting eventual differences.

This would mean that, unlike the voter, no correction is applied, and is used only for detection, and the output needs to be retrieved from a module.

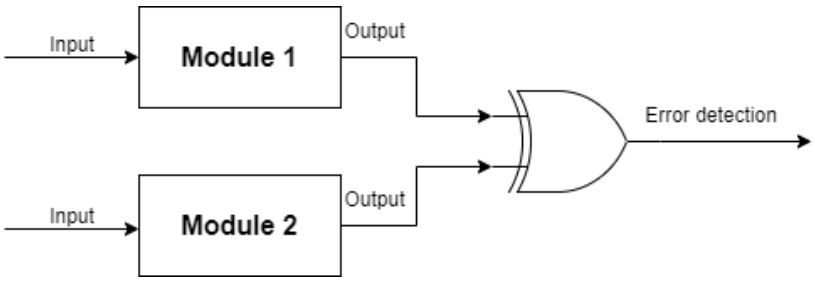

Figure 4.12: XOR schema

4.3.3 B12

B12 is the name of one of a series of benchmarks, written in VHDL, and easy to implement. In particular, B12 was defined as "1-player game (guess a sequence)".

The use of this benchmark however was only in the initial phase of the work, because its simplicity was the right candidate to develop the basis for the database, but was not complex enough to do proper testing.

4.3.4 Neorv32

Neorv32 is an open-source RISC-V processor core designed for FPGAs. It is a lightweight, configurable, and scalable processor architecture that adheres to the RISC-V instruction set architecture (ISA). Neorv32 is intended for educational purposes, research, and use in FPGA-based projects.

Neorv32 also put particular interest in execution safety ensuring always a predictable behaviour.

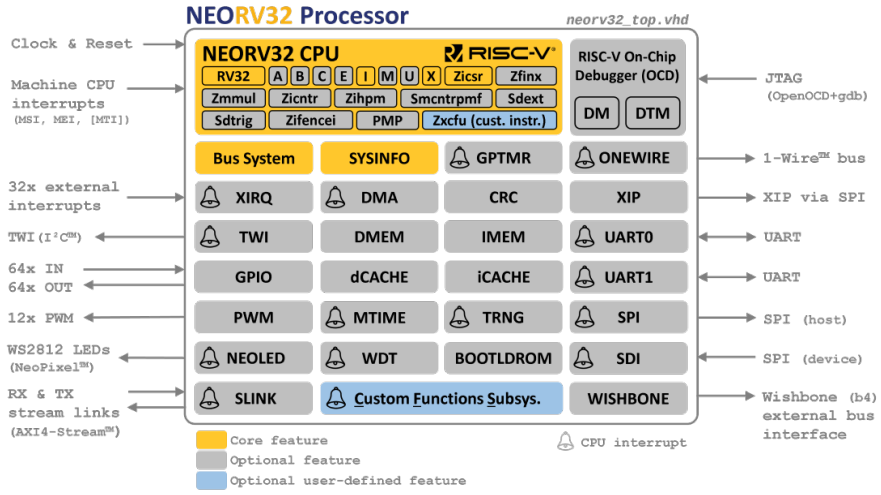

Figure 4.13: Neorv32[20]

Neorv32 gives multiple configurations, permitting it to adapt to the needed situation. Some common features and configurations available are:

- **Base Integer ISA**: Neorv32 typically supports the RV32I base integer instruction set architecture as specified by the RISC-V standard
- **Memory Configuration**: allowing configuration of the memory architecture, including the size and type of instruction and data memories
- **Debugging Support**: Debugging features such as JTAG support and hardware breakpoints may be available to facilitate software development and debugging
- **Peripheral Modules**: may include or support various peripheral modules like UART, GPIO, timers, and more, depending on the specific configuration

- **Boot Configuration**: Users can configure the boot process and memory initialization to suit their specific application requirements

The ultimate goal of the processor was to run a program to be used as a benchmark: one way to run a program is to set a bootloader of the processor and flash the program directly via serial. This way is without doubt flexible but not feasible due to the need for three identical modules.

An alternative approach was to directly write the program into the processor's memory, enabling the selected program to initiate the moment the processor starts. This method guarantees uniform execution of the same program across all modules, ensuring synchronization.

Also, there are many easy-to-implement benchmarks from which to choose the one best suited to the situation.

4.3.5 Complessive Design

Multiple designs were created with different characterizations for each one. Also designs with pblocks are considered to reinforce domain isolation and increase overall reliability.
Pblocks are constraints that can be applied to a design in synthesis to enforce a net to be confined inside a designed zone on the board. This was done to prevent cross-domain PIPs that if enabled would cause to transfer of the signal from one domain to another.

While there were some designs that were used for testing, the main designs were a combination of those characteristics:

- TMR with three modules or XOR gate with two modules
- with or without pblock constrain

In each module, there was a neorv32 processor that would run the benchmark Whetstone. The Whetstone benchmark is a widely used synthetic benchmark designed to measure the performance of a computer system, particularly its floating-point arithmetic capabilities. Developed by Harold Curnow and Brian Wichmann in 1972, Whetstone is designed to evaluate the speed of both integer and floating-point arithmetic operations, providing a numerical score that reflects the system's overall processing power.

The benchmark consists of a set of mathematical routines, including iterations of floating-point arithmetic, trigonometric functions, conditional statements, and array manipulations.

Whetstone has been widely used over the years as a standard benchmark in the field of computer performance testing. While it may not represent real-world applications, it provides a standardized metric for comparing the numerical processing capabilities of different computer systems.

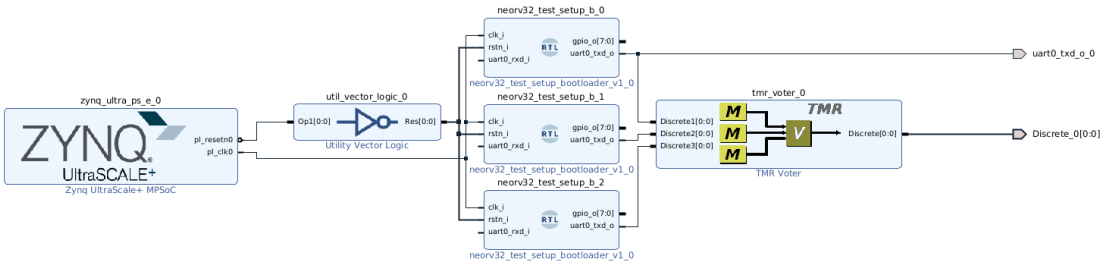

Figure 4.14: Voter block design

Whetstone was not however used for performance testing, but as a benchmark to

let the processor work for a short period of time, so that many resources were used and so the chance of a fault in the case of bitflip was relevant.

As the benchmark concludes with a printed output, determining the presence of a fault relies on comparing this output to the expected result. Any variations or deviations from the expected output indicate the possibility of a fault.

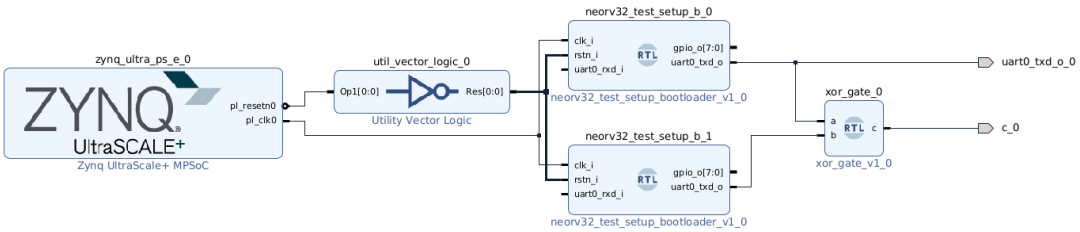

Figure 4.15: XOR gate block design

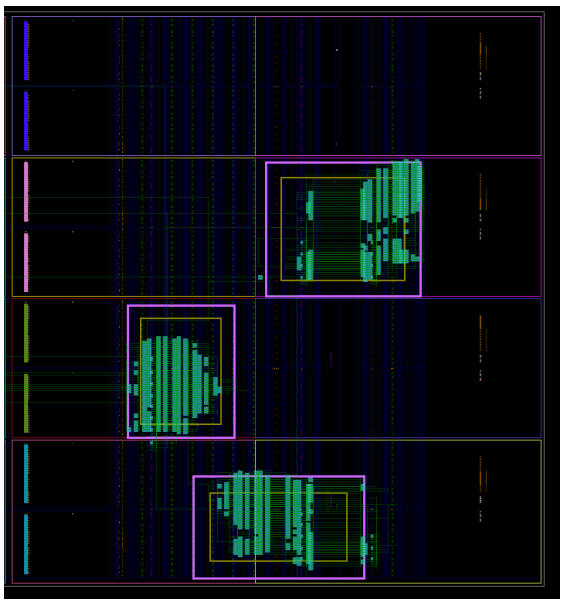

Figure 4.16: Implemented design with pblock contrains

Difference between a design with pblock instead of one without can be seen in 4.16 and 4.17 respectively. In the former design, two domains share a conspicuous

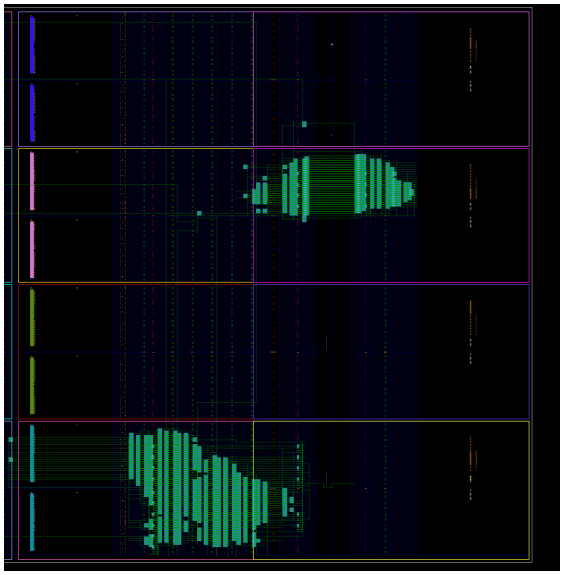

Figure 4.17: Implemented design without pblock contrains

number of switch matrices, resulting in a high number of cross-domain problematic PIPs.

4.4 Static Detection of Single Point of Failure

The main goal of the thesis is, given a specific design, to analyse and find every problematic PIPs. A problematic PIP is defined as a PIP that is not enabled in the design but has the wires to which it is connected active because other enabled PIPs lead or start in those wires.

Particular concern was put on cross-domain PIPs, PIPs that start from a wire of one domain and end in a wire of another domain, causing not only the possibility to transfer a signal to another domain but also an error, causing two domains to be faulted and bypassing the robustness granted by TMR and XOR gate.

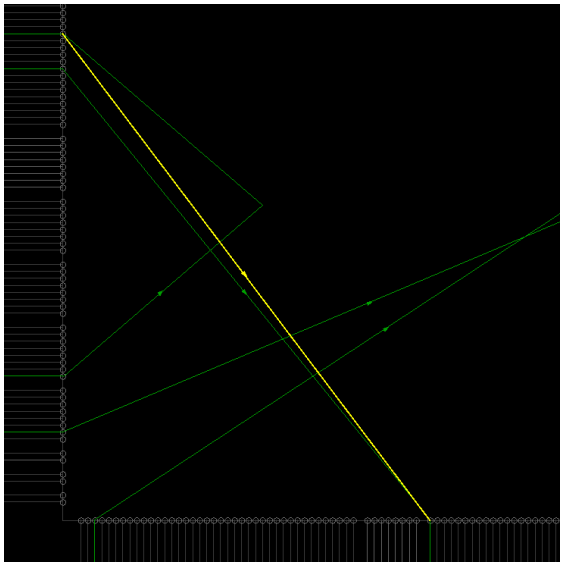

Figure 4.18: In yellow a critical PIP

To complete such task, extensive use of the database created (4.2) was performed.

In particular, databases for PIPs, Wires, and Nets are indispensable. PIPs and Wires data are inherent to an FPGA architecture, and adaptable for various circuits within the same board. On the contrary, Nets are contingent on the circuit under analysis, necessitating reconstruction from scratch whenever the design undergoes changes.

While it was possible to perform such an algorithm with an intricate query in TCL via Vivado, using the local database resulted in $30\times - 300\times$ times faster depending on the design.

4.4.1 Critical PIP Identification Algorithm

The algorithm developed to find all the critical PIPs within a design is structured as follows:

1. For each wire in the design (in all the nets), collect every uphill and downhill PIPs in two sets, one for each direction.

2. After the iteration, I will have two sets containing all the possible uphill and downhill PIPs.

3. Find the PIPs that are present in both sets. Since are sets it's just the intersection between them.

4. Remove from the intersection set all the PIPs present in the net

The result is a set of PIPs that are not already present in the net (so are not enabled) that have an active wire at both the end and the start. Those are the problematic PIPs that could cause faults if enabled.

This algorithm checks problematic PIPs in the whole design, but another similar algorithm was developed to address the case where we want to find only problematic PIPs inside each domain.

In that case, the slightly modified algorithm is composed as follows:

1. For each domain collect PIPs and wires that belong to it.

2. For each wire in the domain, collect every uphill and downhill PIPs in two sets, one for each direction.

3. After the iteration, I will have two sets containing all the possible uphill and downhill PIPs related to a domain.

4. Find the PIPs that are present in both sets. Since are sets it's just the intersection between them.

5. Remove from the intersection set all the PIPs present in the net

6. Update a generic set with all the problematic PIPs with the PIPs found in the previous step.

7. Repeat for each domain

After this procedure, the problematic PIPs that belong to a single domain are obtained.

If we run both algorithms we will end with two sets: all the critical PIPs inside the design, and critical PIPs that belong to a specific domain.

This implies that all the PIPs that belong to the first set, but that are not present in the second one, are necessarily cross-domain critical PIPs.

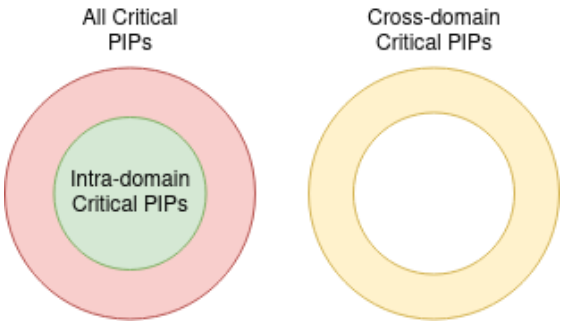

Figure 4.19: With the sets obtained from the algorithm (left) is possible to obtain the Cross-domain Critical PIPs (right)

As mentioned in 4.2 the data stored in the database collected are generic, meaning that every PIP and wire does not have the full name, with the associated tile, but just the second part of the name, where to identify the behaviour of the object.

This is not, of course, true for the net database, since knowing exactly what PIPs and wires are present is imperative.

But the end result must have a list of specific PIPs, that can be identified on the board.

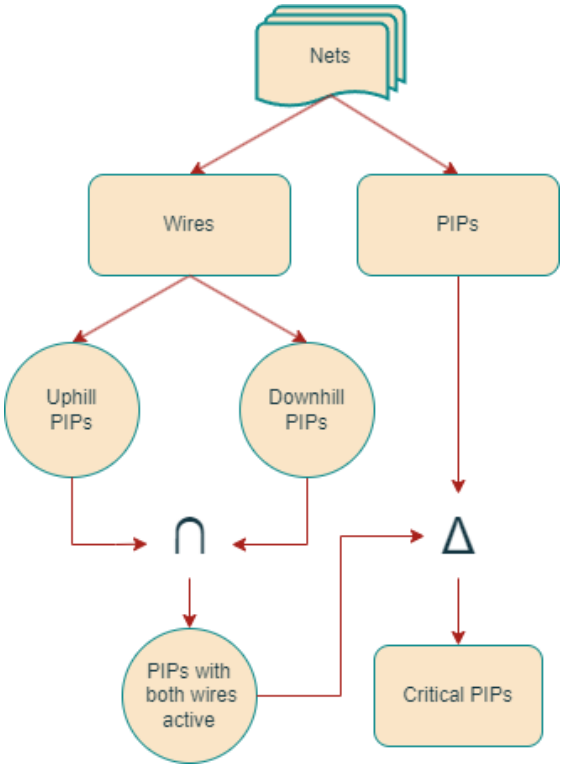

Figure 4.20: Algorithm to find all the critical PIPs in a design

So to obtain a specific PIP during the algorithm, the following procedure is performed:

1. During the iteration of the wires, is obtained for the current wire the tile name (first part of the name) and the generic wire name (second part)

2. the type of the tile is acquired from the tile name

3. with the tile type and the generic name the entry for the wire can be retrieved from the database, and so the uphill and downhill PIPs.

4. Add to the generic name of the PIPs the tile name obtained in the first step

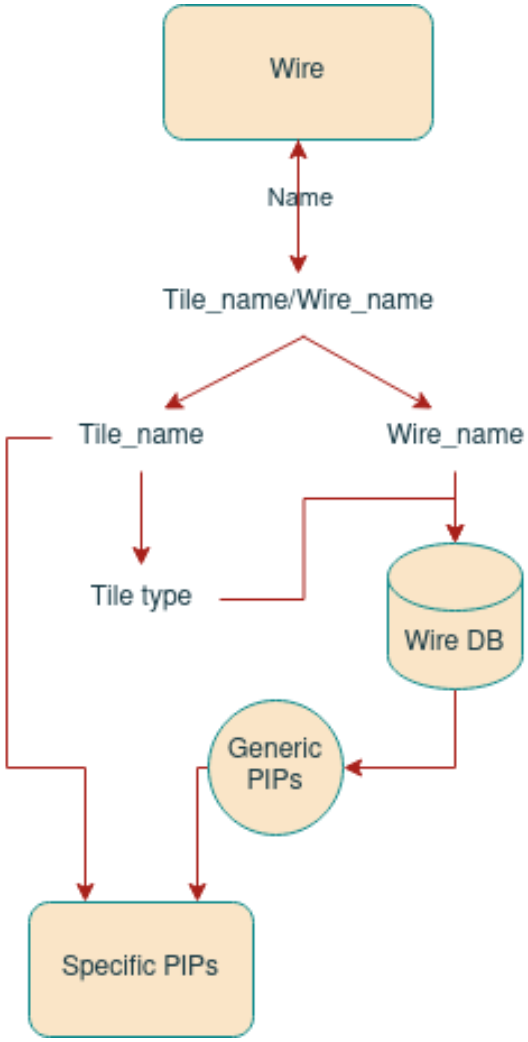

Figure 4.21: Procedure to get specific PIPs from the database

4.5 Experimental Analysis

One of the essential parts of the work is to verify that the proposed design is capable of improving the general robustness. There are multiple ways to achieve this, one with dedicated infrastructures that irradiate directly a board, but a more feasible one is to directly inject a corrupted bitstream to an FPGA.

To achieve the latter procedure, a bitstream of the chosen design is required, generated by software like Vivado. During the study of the versal, we assumed that the bitstream generated for this architecture would have had the same format as in the previous families, but ended up being a completely different way to program the FPGA: not only did it have an unknown format, but also instead of a single .bit file, multiple files where generated, with no defined purpose.

Various testing methods were explored for evaluating the enhancements, but the bitflipped approach emerged as the most favoured due to its close resemblance to real-world scenarios. Furthermore, comparable tests were conducted on the Ultrascale+ family. The transfer of the database to the new architecture posed the sole supplementary task, a task rendered straightforward by the scripts' inherent flexibility.

4.5.1 Fault injection approach

The injections were made using a board KRIA kv260, from the Ultrascale+ family. The reason for why this specific board was simply because it was available as a physical board, to program directly from a computer.

Software in Python was written for the purpose of injecting a faulted bitstream, and with libraries such as *serial*, to interface with the serial port, and PyXEL [13] to load and modify the bitstream and then to program the board with it.

Two types of injections were made: a single bitflip injection, to analyse if it will cause faults or not, and cumulated bitflips, with the aim of broke on purpose the system.

Before starting the analysis two bitstreams were created from the design with TMR, one without pblock constraints and the other with.

After the generation, it was possible to analyse a visualisation of the bitstream thanks to PyXEL (fig. 4.22) to achieve targeted injection, since as seen in the image, the majority of the bitstream is empty space that would cause nothing. So the injections were not made in the entire bitstream but only in a range where the information about the modules was likely to be stored. From 4.22 that zone corresponds to the white pixels.

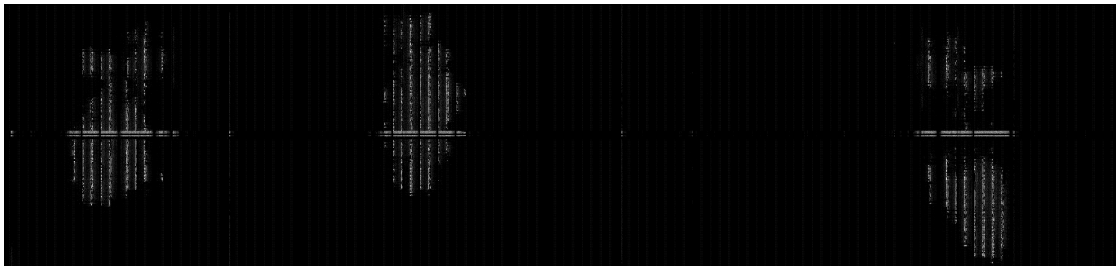

Figure 4.22: Visual representation of the bitstream of the TMR pblock enforced design

While the bitstream brings an important piece of information, it was not possible to reverse the information from the image to a general case, meaning that was not possible, at least directly, to retrieve the exact correspondence to the design of a pixel. So the injection proceeded with the intention of injecting a module zone, but without knowing what was modified.

By using the image it was possible to retrieve a specific range of pixels to inject at random. Since the modules were separated by blank space, using a single range was not optimal, instead three separated ranges were chosen. To inject those

domains completely at random a weighted random decision of which domain to inject is made. The weight is based on the number of pixels.

The output of the benchmark, which will end with a line of 'Done_Done', is used to recognise the correct execution of the FPGA programmation, ending the listening socket early and allowing saving time. This of course doesn't mean that the output is correct so the comparison with the gold run is done in any case after the socket ends.

Lastly, the runs were preseeded, to achieve replicability, and the possibility to rerun specific runs. The last is fundamental, considering the not-so-uncommon case of issues occurring during a run.

The script used to inject a single bitstream is the following:

1. The bitstream is loaded
2. A run is executed with the untouched bitstream, and the output is saved for later comparison
3. A range frame is chosen with the weighted distribution, and then the random coordinate for the bitflip within the range of the domain.
4. If a run already exists for that coordinates, skip the run. This is also useful to resume the runs session in case of interruption
5. Is created the bitstream with the bit flipped in the chosen coordinates.
6. A thread is created with the goal of listening to the serial, while the main execution will program the FPGA.

7. After a string is received (that announces the end of the whetstone) or a timeout is reached the thread with the socket will end, returning what listened.

8. the output received is saved in a file, and a comparison with the gold run is made to ensure that the output is the one expected or not.

As I will show in 4.5.2 with a single bitflip it's very unlikely to cause faults in the system, since the TMR voter helped to mask a possible error, so the only case where a fault could be caused is when a cross-domain PIP is flipped, which are a fraction of the total PIPs.

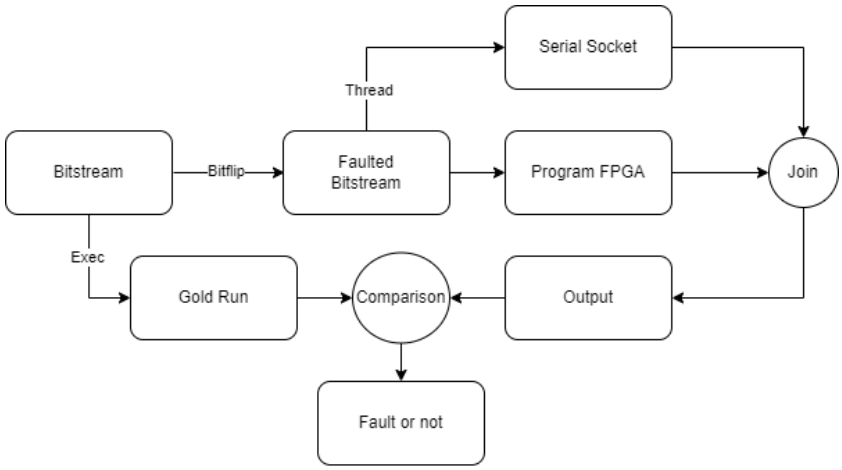

Figure 4.23: Injection flow for a single bitflip

For also this reason the runs with cumulative bitflips are more interesting. The procedure is similar to the single bitflip runs, but some expedients are required for the correct execution of the program. Those are the following steps:

1. The bitstream is loaded

2. A run is executed with the untouched bitstream, and the output is saved for later comparison

3. Starting from the number 0, check if a run already exists with that number, if so skip the run. This is also useful to resume the runs session in case of interruption

4. A range frame is chosen with the weighted distribution, and then the random coordinate for the bitflip within the range of the domain. This is repeated for a predefined number of times (usually 64)

5. Is created the bitstream with the bit flipped in the chosen coordinates and also all the bitflips are saved in order of execution.

6. A thread is created with the goal of listening to the serial, while the main execution will program the FPGA.

7. After a string is received (that announces the end of the whetstone) or a timeout has reached the thread with the socket will end, returning what listened.

8. If the output is identical to the one of the golden run, repeat the procedure from the 4, otherwise a fault is found and the exact number of bitflips needed are searched.

9. A binary search is used to find the exact number of bitflips. We start knowing a number of bitflips that did not cause faults and another that caused faults

10. we pick the number in the middle of those two and, if the bitstream has more bitflips than this number then bitflips are reversed until reach that number, otherwise bitflips are applied, using the previously saved bitflips coordinates.

11. If we got a fault we lower the number of bitflips required to cause a fault, otherwise we increase the number that did not cause the fault. This will decrease the search range.

12. Repeat from the 9 until the range is reduced to a single bitflip

13. Is saved in a file the bitflips required to cause a fault

For the cumulative bitflips runs another approach is used to assure replicability: two random number generators were used, the first one is in charge of generating a seed for every run, so every n run has the same seed, and the second one is using the seed generated from the first to generate the coordinate of the bitflips.

4.5.2 Results

In this section will be discussed the results obtained with the methods and the design explained in the previous section.

Should be noted that despite the numerous expedients, the injection campaign requested weeks of testing, for reasons mentioned before, like only a single board to test, slow time to program the FPGA, and the frequency of wrong runs. Usually, a single run for cumulated injections took about 100 seconds, meaning ~800 runs per day, but in the end, 100-200 of those runs were lost to some problems with the board and needed to be redone.

The single bitflip instead was faster and was possible to obtain 4000 runs in a day, tested in the TMR without pblock constraints. As expected, despite this being the most vulnerable design, the robustness granted by the TMR did not lead to any failure. The only possible way to cause a fault was to enable one of the critical PIPs, which are a small fraction of the entirety of the bitstream, and even with

that case not taken for granted that would cause an issue. For those motivations, the runs were not done in the design with pblocks, because it was highly unlikely to cause any issues.

Instead, the cumulated injections were chosen as the best metric, and 3000 runs for each design were made, testing the number of required bitflips to alter the output.

The plot 4.24 shows the probability of fault after a random number of bitflips for both designs, capped at 300 bitflips (since after, the number of faults is not relevant). As expected the design with pblock constraints performs slightly better than the one without. To give more insights another figure is plotted 4.25 showing the ratio between the probability of the design with pblock and the one without.

Appling the pblock constraints drastically reduces the number of cross-domain PIPs as shown in table 4.2, while also reducing the total number of PIPs in the design. The single PIP that is cross-domain in the pblock design is inside the voter and while could be prevented with other techniques, was not possible with the pblock method.

Design	number of PIPs	cross-domain PIPs
Normal Design	260019	1103
Pblock Design	254131	1

Table 4.2: Number of PIPs and cross-domain PIPs for each design

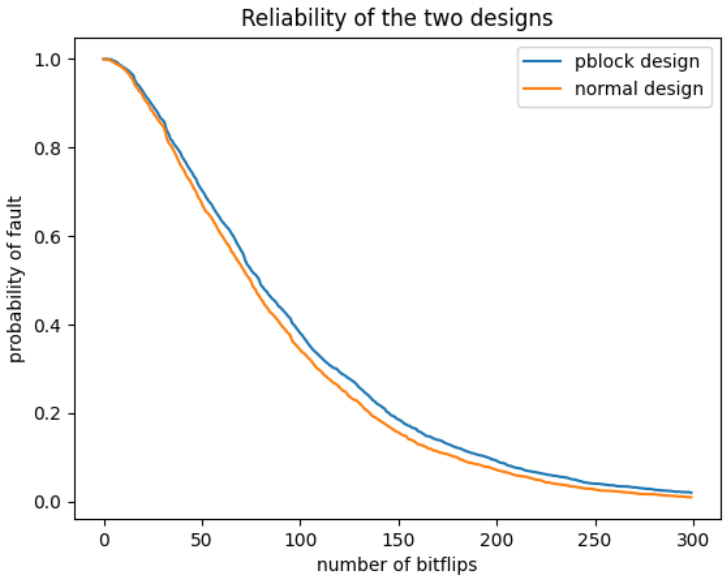

Figure 4.24: Reliability of the designs

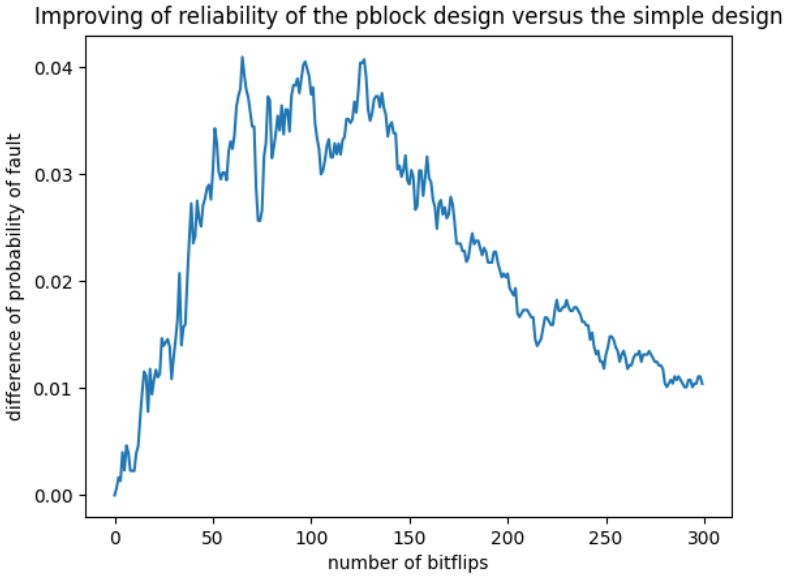

Figure 4.25: Difference between the probability of fault between the designs

Chapter 5

Conclusions

The primary objective of the thesis was to create a database eliminating the need for Vivado for architectural data queries on a particular FPGA board family. This initiative aimed at conducting statistical analyses to enhance overall reliability in scenarios involving SEUs.

The database not only proves to significantly improve query times but also the software's flexibility opens up the potential to extend these enhancements to other FPGA families.

The recognition of critical PIPs is a fundamental step for enhancing the robustness of TMR design, giving the possibility to manage a custom routing to avoid cross-domain PIP. This is proven by manually placing modules to avoid conflicts, using the pblock constraints, and increasing de facto the number of accumulated bitflips required to fault the system.

The pblock constraints are not always available, because they require physical space to be placed, and this is not always true considering that the TMR triples the space required, so future work to manage custom routing instead of placing

pblock should be the optimal case.

5.1 Future works

As highlighted in chapter 4.5, a primary challenge emerged early in the research the modification in the bitstream format for the Versal series. The absence of prior comprehensive studies added complexity to the task. Additionally, the Versal architecture's recentness presents another hurdle, as obtaining the physical board is challenging due to high demand and limited availability, consequently driving up prices.

To obtain the results a bitflipped bitstream was used to program the board. This implies that all the procedure to program the board was necessary, taking a notable amount of time. A different approach would be to use a Vivado module called SEM (Soft Error Mitigation), which allows for fault directly at runtime, not requiring the programmation of the board at every bitflip. Given that with the method used for the work 12 seconds are needed just to program the FPGA, this timeframe is no longer required, decreasing enormously the time for each injection, permitting a greater amount of injection.

To improve the robustness of a design, modules were separated to ensure no tiles were shared among them. This is an easy procedure and feasible only when there is an abundance of empty space available. That could be not true for every design, particularly for those that demand substantial space. With the work done, it was possible to forbid tiles that could cause problematic PIPs, but even in this situation,

the solution would be sub-optimal. Instead, the re-routing of every problematic net should be the optimal case, and the data gathered during the thesis work should be all of what would be needed to complete this task.

Bibliography

[1] Priya Peter and Manju Maheve. «Single Event Effects Analysis to Improve the System Safety and Fault Tolerance». In: *2021 Annual Reliability and Maintainability Symposium (RAMS)*. 2021, pp. 1–5. DOI: 10.1109/RAMS48097.2021.9605772 (cit. on p. 8).

[2] H. J. Barnaby. «Total-Ionizing-Dose Effects in Modern CMOS Technologies». In: *IEEE Transactions on Nuclear Science* 53.6 (2006), pp. 3103–3121. DOI: 10.1109/TNS.2006.885952 (cit. on p. 8).

[3] T.R. Oldham and F.B. McLean. «Total ionizing dose effects in MOS oxides and devices». In: *IEEE Transactions on Nuclear Science* 50.3 (2003), pp. 483–499. DOI: 10.1109/TNS.2003.812927 (cit. on p. 8).

[4] Daniel M. Fleetwood. «Radiation Effects in a Post-Moore World». In: *IEEE Transactions on Nuclear Science* 68.5 (2021), pp. 509–545. DOI: 10.1109/TNS.2021.3053424 (cit. on p. 9).

[5] Daisuke Kobayashi. «Scaling Trends of Digital Single-Event Effects: A Survey of SEU and SET Parameters and Comparison With Transistor Performance». In: *IEEE Transactions on Nuclear Science* 68.2 (2021), pp. 124–148. DOI: 10.1109/TNS.2020.3044659 (cit. on p. 9).

[6] P.E. Dodd and L.W. Massengill. «Basic mechanisms and modeling of single-event upset in digital microelectronics». In: *IEEE Transactions on Nuclear Science* 50.3 (2003), pp. 583–602. DOI: 10.1109/TNS.2003.813129 (cit. on p. 9).

[7] R.C. Baumann. «Radiation-induced soft errors in advanced semiconductor technologies». In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 305–316. DOI: 10.1109/TDMR.2005.853449 (cit. on p. 15).

[8] C. De Sio E. Vacca and S. Azimi. *Layout-oriented Radiation Effects Mitigation in RISC-V Soft Processor*. May 2022. URL: https://dl.acm.org/doi/10.1145/3528416.3530984 (cit. on p. 15).

[9] Andrea Portaluri, Corrado De Sio, Sarah Azimi, and Luca Sterpone. «A New Domains-based Isolation Design Flow for Reconfigurable SoCs». In: *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 2021, pp. 1–7. DOI: 10.1109/IOLTS52814.2021.9486687 (cit. on p. 15).

[10] Corrado De Sio, Sarah Azimi, and Luca Sterpone. «FireNN: Neural Networks Reliability Evaluation on Hybrid Platforms». In: *IEEE Transactions on Emerging Topics in Computing* 10.2 (2022), pp. 549–563. DOI: 10.1109/TETC.2022.3152668 (cit. on p. 16).

[11] S. Azimi, C. De Sio, A. Portaluri, D. Rizzieri, and L. Sterpone. «A comparative radiation analysis of reconfigurable memory technologies: FinFET versus bulk CMOS». In: *Microelectronics Reliability* 138 (2022). 33rd European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, p. 114733. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.

2022.114733. URL: https://www.sciencedirect.com/science/article/pii/S0026271422002578 (cit. on p. 16).

[12] E. Vacca, S. Azimi, and L. Sterpone. «Failure rate analysis of radiation tolerant design techniques on SRAM-based FPGAs». In: *Microelectronics Reliability* 138 (2022). 33rd European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, p. 114778. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2022.114778. URL: https://www.sciencedirect.com/science/article/pii/S002627142200302X (cit. on p. 16).

[13] Corrado De Sio, Sarah Azimi, Luca Sterpone, David Merodio Codinachs, and Filomena Decuzzi. «PyXEL: Exploring Bitstream Analysis to Assess and Enhance the Robustness of Designs on FPGAs». In: *2023 19th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*. 2023, pp. 1–4. DOI: 10.1109/SMACD58065.2023.10192116 (cit. on pp. 16, 48).

[14] Tao Zhang, Jian Wang, Shize Guo, and Zhe Chen. «A Comprehensive FPGA Reverse Engineering Tool-Chain: From Bitstream to RTL Code». In: *IEEE Access* 7 (2019), pp. 38379–38389. DOI: 10.1109/ACCESS.2019.2901949 (cit. on p. 16).

[15] Yongseen Kim, Eun-Gu Jung, and ChangKyun Kim. «Bitstream Reverse Engineering of Microsemi's VersaTile-based FPGAs». In: *2021 IEEE Physical Assurance and Inspection of Electronics (PAINE)*. 2021, pp. 1–8. DOI: 10.1109/PAINE54418.2021.9707700 (cit. on p. 16).

[16] S. Azimi, C. De Sio, and L. Sterpone. «Analysis of radiation-induced transient errors on 7 nm FinFET technology». In: *Microelectronics Reliability* 126 (2021). Proceedings of ESREF 2021, 32nd European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, p. 114319. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2021.114319. URL: https://www.sciencedirect.com/science/article/pii/S0026271421002857 (cit. on p. 17).

[17] Kyle W. Gear, Alfonso Sánchez-Macián, and Juan Antonio Maestro. «An analysis of FPGA configuration memory SEU accumulation and a preventative scrubbing technique». In: *Microprocessors and Microsystems* 90 (2022), p. 104467. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2022.104467. URL: https://www.sciencedirect.com/science/article/pii/S0141933122000357 (cit. on p. 17).

[18] Melanie D. Berg, Kenneth A. LaBel, Hak Kim, Mark Friendlich, Anthony Phan, and Christopher Perez. «A Comprehensive Methodology for Complex Field Programmable Gate Array Single Event Effects Test and Evaluation». In: *IEEE Transactions on Nuclear Science* 56.2 (2009), pp. 366–374. DOI: 10.1109/TNS.2009.2013857 (cit. on p. 17).

[19] *Versal ACAP Configurable Logic Block Architecture Manual*. AMD. 2023. URL: https://docs.xilinx.com/r/en-US/am005-versal-clb/Overview (cit. on pp. 18, 20, 24, 25).

[20] stnolting. *neorv32*. URL: https://github.com/stnolting/neorv32 (cit. on pp. 35, 38).

[21] F. Corno, M.S. Reorda, and G. Squillero. «RT-level ITC'99 benchmarks and first ATPG results». In: *IEEE Design and Test of Computers* 17.3 (2000), pp. 44–53. DOI: 10.1109/54.867894 (cit. on p. 35).