Mejora de la tolerancia a fallos de un procesador RISC-V mediante reconfiguración dinámica en FPGAs
Improving Fault Tolerance of a RISC-V Processor through Dynamic Reconfiguration in FPGAs

Trabajo de Fin de Grado
Curso 2024–2025

Autor
Isabel Román Mikkilä

Director
Óscar Garnica Alcázar

Grado en Ingeniería de Computadores
Facultad de Informática
Universidad Complutense de Madrid

Mejora de la tolerancia a fallos de un procesador RISC-V mediante reconfiguración dinámica en FPGAs
Improving Fault Tolerance of a RISC-V Processor through Dynamic Reconfiguration in FPGAs

Trabajo de Fin de Grado en Ingeniería de Computadores

Autor

Isabel Román Mikkilä

Director

Óscar Garnica Alcázar

Convocatoria: *Junio 2025*
Calificación: *10*

Grado en Ingeniería de Computadores
Facultad de Informática
Universidad Complutense de Madrid

20 de junio de 2025

Dedicatoria

A mi perrita Aysha. Ojalá fueras eterna.

Agradecimientos

Quiero agradecer enormemente a mis padres, que siempre han priorizado mis estudios, pero sobre todo mi felicidad, y me han enseñado lo que realmente es importante en la vida.

A Miguel, mi novio y mi mayor apoyo desde hace años, por acompañarme en cada paso con motivación, paciencia y amor incondicional.

A mis queridas hermanas, son mis mejores amigas y una constante fuente de inspiración.

A mis abuelos, los que están y los que ya no. Sin su apoyo no estaría donde estoy. En especial, a mi abuelo Luis, gracias a él empecé esta carrera, y sé que ahora estaría muy orgulloso.

También quiero agradecer a mis profesores de la academia Maths, sobre todo a Pilar, por darme el empujón inicial que necesitaba y por ver mi futuro en el hardware.

Y a todos los profesores de la Facultad de Informática, especialmente al tutor de este TFG, Óscar, por guiar el proyecto con dedicación y confianza en mi trabajo.

Resumen

Mejora de la tolerancia a fallos de un procesador RISC-V mediante reconfiguración dinámica en FP-GAs

El propósito de este Trabajo de Fin de Grado es estudiar la posible mejora de la tolerancia a fallos de un procesador RISC-V mediante reconfiguración parcial dinámica.

Para ello, se desarrolla un modelo de simulación en Python que permite evaluar el comportamiento del sistema ante fallos en el hardware, comparando su fiabilidad en escenarios con y sin la capacidad de reubicar dinámicamente los módulos afectados.

El proyecto se basa en trabajos previos de los compañeros Davó Laviña (2022) y Carpio Cuenca (2024), quienes integraron una red NoC (Network-on-Chip) en el procesador RISC-V para interconectar distintos módulos funcionales.

En este contexto, se analiza el flujo de diseño de un proyecto con Dynamic Function eXchange (DFX), identificando los cambios necesarios en el diseño RTL para adaptar el procesador a esta técnica sin comprometer la conectividad a través de la NoC.

Las FPGAs son los dispositivos ideales para implementar este tipo de diseño, ya que la reconfiguración dinámica es una de sus características más representativas. Esta capacidad permite añadir, sustituir o eliminar módulos lógicos en tiempo de ejecución sin reiniciar el sistema.

Finalmente, se presentan los resultados obtenidos mediante simulación, se analizan posibles mejoras combinando DFX con otras técnicas de redundancia, y se reflexiona sobre la viabilidad de convertir el diseño en uno completamente adaptado a DFX, así como sobre la idoneidad del uso de la NoC como infraestructura de comunicación en este tipo de arquitecturas reconfigurables.

Palabras clave

RISC-V, Reconfiguración Parcial Dinámica, NoC, DFX, RTL, FPGA

Abstract

Improving Fault Tolerance of a RISC-V Processor through Dynamic Reconfiguration in FPGAs

This Bachelor's Final Project aims to explore the potential improvement of fault tolerance in a RISC-V processor through partial dynamic partial reconfiguration.

To this end, a Python-based simulation model has been developed to evaluate the system's behavior in the presence of hardware faults, comparing its reliability in scenarios with and without the ability to relocate faulty modules dynamically.

The project builds upon previous work by Davó Laviña (2022) and Carpio Cuenca (2024), who integrated a Network-on-Chip (NoC) into the RISC-V processor to interconnect various functional modules.

In this context, the design flow of a project using Dynamic Function eXchange (DFX) is analyzed, identifying the required modifications in the RTL design to adapt the processor to support this technique without compromising connectivity through the NoC.

FPGAs are the perfect devices to implement this type of architecture, as dynamic reconfiguration is one of their most distinctive features. This capability enables logic modules to be added, replaced, or removed at runtime without restarting the system.

Finally, the simulation results are presented, potential improvements are discussed, such as combining DFX with other redundancy techniques, and the feasibility of fully adapting the design to DFX is evaluated, as well as the suitability of using the NoC as a communication infrastructure in such reconfigurable architectures.

Keywords

RISC-V, Dynamic Partial Reconfiguration, NoC, DFX, RTL, FPGA

Contents

1. Introduction **1**
1.1. Motivation 1
1.2. Objectives and Work Plan 2

2. State-of-the-art **5**
2.1. Fault Tolerance 5
2.1.1. Faults, Errors and Failures 5
2.1.2. Fault Classification 6
2.1.3. Dependability Metrics 8
2.1.4. Redundancy 11
2.2. FPGAs and Dynamic Reconfiguration 12
2.2.1. Structure of an FPGA 13
2.2.2. DFX Overview 14
2.2.3. Other Common Applications 15
2.3. RISC-V Processor 16
2.3.1. RISC-V 16
2.3.2. VeeR EL2 Core 17
2.3.3. Integrated NoC 19

3. Tools and Technologies **23**
3.1. Modeling and simulation 23
3.2. Hardware implementation 24
3.3. Project management 24
3.4. Report writing 25

4. Project Development **27**
4.1. Python Simulation Model 28
4.1.1. Scope 28
4.1.2. Files 29
4.1.3. Execution Flow 30
4.1.4. Classes and Logic 33
4.1.5. Updates and Improvements 35

4.2. NoC Integration and Dynamic Function eXchange 36
4.2.1. VeeRwolf Nexys 36
4.2.2. VeeRwolf Sim 37
4.2.3. Vivado DFX Poject 40

5. Results 43
5.1. Implementation of RISC-V with NoC 43
5.1.1. Vivado Reports 43
5.1.2. Verilator Simulator 46
5.2. Analysis of the Python Simulation 47
5.2.1. Injected Faults 47
5.2.2. Reconfiguration Performance 50
5.2.3. Dynamic Reconfiguration on Fault Tolerance 51

6. Conclusions and Future Work 53
6.1. NoC Limitations and Reinforcements 53
6.2. Design Adaptations for DFX 54
6.3. NoC Advantages for Partial Reconfiguration 56

Bibliography 57

A. DFX Supported Devices 59

B. Vivado DFX Project Flow 65

C. VeeRwolf Guide 67

List of figures

1.1. Gantt chart of the project timeline (final update). 3

2.1. The cause-and-effect relationship of faults (Johnson, 1984). 7
2.2. Barriers constructed by fault avoidance, fault masking, and fault tolerance (Johnson, 1984). 7
2.3. The reliability (survivor) function R(t) (Rausand and Høyland, 2004). 8
2.4. The bathtub curve (Rausand and Høyland, 2004). 10
2.5. Triple Modular Redundancy (Mitra and McCluskey, 2000). 11
2.6. Basic Premise of Dynamic Function eXchange (AMD Xilinx, 2024). . 12
2.7. Generic view of FPGA architecture (Li et al., 2019). 14
2.8. VeeR EL2 Core Pipeline (CHIPS Alliance, 2022). 18
2.9. VeeR EL2 Core Complex (CHIPS Alliance, 2022). 19
2.10. Network Architecture (Carpio Cuenca, 2024). 19
2.11. Connection of the sender (S) and receiver (R) modules, and the divider and multiplier wrappers to the NoC. (Carpio Cuenca, 2024). . . 21

4.1. Python model system architecture 28
4.2. Python model program flow 32
4.3. VeeRwolf Nexys A7 target (CHIPS Alliance) 37
4.4. VeeRwolf Simulation target (CHIPS Alliance) 38
4.5. Simulation output with the default `hello.vh` program loaded. 38
4.6. Simulation output with `mul_test.vh` loaded. 39
4.7. Simulation output with `div_test.vh` loaded. 39
4.8. Reconfigurable areas implementation 41

5.1. Verilator error due to unsupported interface vectors in `mesh_FT` . . . 46
5.2. Cumulative Frequency Curve: Failed Simulations (%) vs Total Faults (with dynamic reconfiguration) 48
5.3. Cumulative Frequency Curve: Failed Simulations (%) vs Total Faults (without dynamic reconfiguration) 48
5.4. Jitter plot: Total Faults per Simulation (with dynamic reconfiguration) 49
5.5. Line plot: Reconfigurations Performed vs Number of Simulations . . . 51
5.6. Box plot: Tolerated Faults vs Number of Reconfigurations 52

6.1. NoC design with DPR and TMR 55
6.2. Testbench run before DFX 56

List of tables

2.1. The 9s (IBM Corporation, 2021) 9

4.1. Program files and descriptions 29

5.1. Design Timing Summary 43
5.2. Slice Logic from Resource Utilization Report 44
5.3. Utilization by Hierarchy from Resource Utilization Report 45

6.1. Faulty partitions that caused system failure 53
6.2. Error type that caused system failure 54

A.1. DFX Supported Devices 59

Chapter 1

Introduction

1.1. Motivation

FPGAs (Field-Programmable Gate Arrays) were initially used mainly for prototyping, logic emulation systems, and low-volume applications. After significant advances, they now play a fundamental role in embedded systems, hardware acceleration, communications, and networking.

Due to the continuous miniaturization of transistors, the capabilities of these devices have increased dramatically over the past decades, sparking growing interest in highly demanding industries such as automotive, railway, industrial, aeronautics, aerospace, and defense. However, technological advancements have also introduced new challenges. As device sizes shrink, electronic current density in metal traces increases, leading to a higher risk of electromigration and thermal noise. Additionally, smaller transistors require less charge to switch, making them more vulnerable to ionizing radiation, especially gamma particles, which increases the probability of transient faults.

When an ionizing particle strikes an electronic device, it can alter the logic state of its signals (soft error) or even cause permanent damage if the energy is sufficiently high. Among the various types of hardware errors, soft errors are considered the most dominant for today's digital systems. This issue is particularly critical in safety-sensitive applications, such as autonomous vehicles, space missions, medical systems, and cyber-physical systems with strict security requirements. In these contexts, hardware designs must ensure high reliability and comply with specific safety standards, which require rigorous fault estimation and mitigation methods. Given this scenario, fault tolerance in FPGAs is crucial to maintaining hardware reliability in critical applications.

Traditionally, static redundancy or replicate circuitry has been used to address these issues, duplicating or triplicating system components to ensure continuous operation in the presence of faults. However, this technique comes with high area and power consumption costs, making it impractical for resource-constrained systems. A powerful alternative approach is to implement designs in reconfigurable logic and retain unused cells for use as spares in case of a fault. These spares can then be used to repair the circuit's functionality via device reconfiguration, providing a

more scalable and efficient solution. This concept is known as dynamic (or active) redundancy.

FPGAs are particularly well-suited for these strategies because of their programmability and adaptability. In particular, dynamic reconfiguration allows parts of the circuit to be modified at runtime, while the rest of the system continues to operate normally. This capability enables fault mitigation without rebooting the system or interrupting critical functionalities. When a fault is detected, the FPGA can load a new configuration that reassigns logic blocks and routing channels to avoid damaged areas, effectively self-repairing without external intervention. This feature is especially valuable in applications where downtime must be minimized, such as real-time embedded systems or systems where hardware access is difficult or costly, such as satellites, industrial infrastructures, or devices in remote environments. Dynamic reconfiguration improves system reliability without the need for unnecessary hardware replication. It is a flexible, cost-effective solution that improves fault tolerance without sacrificing performance or energy efficiency.

This project focuses on applying this technique to the design of a RISC-V processor with an integrated NoC (Network-on-Chip). RISC-V is an open and modular architecture that allows hardware customization, making it an ideal candidate for implementing dynamic reconfiguration in its functional units.

Additionally, the incorporated NoC in the design enables efficient and scalable communication between the processor's modules, which is crucial for maintaining system integrity during reconfiguration processes. This combination offers a promising approach to improving fault tolerance in modern architectures, particularly in environments where reliability and adaptability are essential.

1.2. Objectives and Work Plan

The objective of this project is to demonstrate how dynamic reconfiguration can improve the fault tolerance of a RISC-V processor. The starting point is a RISC-V processor that integrates a Network-on-Chip (NoC) as an interconnection mechanism, enabling the relocation of functional units to reconfigurable regions at runtime while maintaining the circuit's operability. The target platform is the Nexys 4 board, which features an Artix-7 FPGA.

To achieve this goal, these steps will be followed:

- Study and understand the RTL design of the VeeR EL2 RISC-V processor and the integrated NoC.

- Learn about dynamic partial reconfiguration and define dynamic reconfigurable regions in the processor's design to optimize fault tolerance.

- Develop a Python model to simulate fault injection in the processor's functional units and evaluate the effectiveness of dynamic reconfiguration.

- Learn specifically about the Dynamic Function Exchange (DFX) design approach for FPGAs and its implementation with Vivado.

- Define potential reconfiguration regions in the processor's architecture and determine the required design changes to integrate DFX into the project.
- Assess the role of the NoC in facilitating dynamic reconfiguration and analyze whether its use is advantageous in improving the processor's fault tolerance when applying DFX.

A schedule outlining the main tasks has been created to plan and structure the project. A Gantt chart is used for this purpose and will be updated as the project progresses. Figure 1.1 shows the final version of the chart.

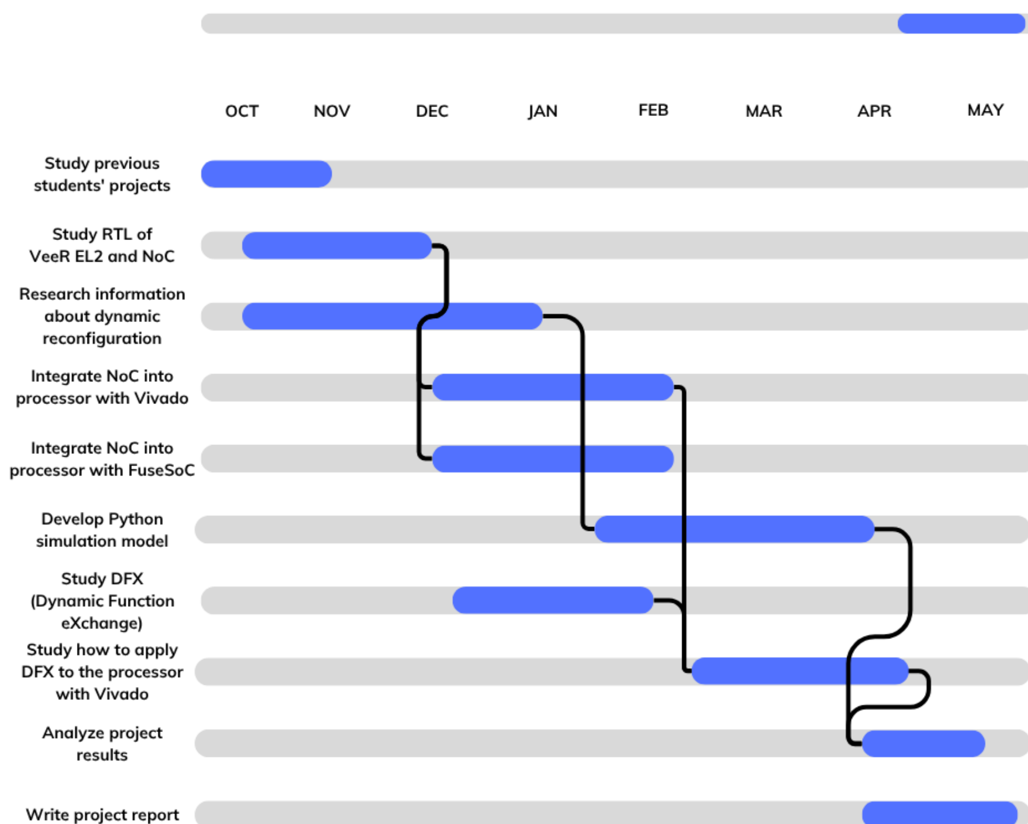

Figure 1.1: Gantt chart of the project timeline (final update).

Chapter 2

State-of-the-art

2.1. Fault Tolerance

Fault tolerance is the ability of a system to continue performing its specified tasks after the occurrence of faults (Johnson, 1984). A fault-tolerant system should still meet its goals even in the presence of hardware or software problems, power failures, or other kinds of unexpected events.

With new generations of technology offering higher transistor densities, the complexity of computer systems continues to grow, and as a consequence, the probability of faults also increases. Computer scientists and engineers have attempted to minimize the occurrence of faults in the hardware they design using various tools and techniques, such as well-defined design specifications, careful component selection, and extensive testing. However, it is practically impossible to avoid all hardware bugs, as unexpected environmental conditions, user mistakes, and other unpredictable factors can never be completely accounted for. As Koren and Krishna (2007) noted, we must build systems that acknowledge the existence of faults as a fact of life and incorporate techniques to tolerate them while still delivering an acceptable level of service.

2.1.1. Faults, Errors and Failures

The terms fault, error, and failure are often used interchangeably, but each has a specific meaning in the context of system reliability. While all three indicate that something has gone wrong, they refer to different levels. A fault occurs at the physical level, an error at the computational level, and a failure at the system level. Based on Dubrova's (Dubrova, 2013) definitions, the three terms can be differentiated as follows:

- A fault is a physical defect, imperfection, or flaw in some hardware or software component. Examples are a short-circuit between two adjacent interconnects, a broken pin, or a software bug.
- An error is a deviation from correctness or accuracy in computation, which occurs as a result of a fault. Errors are usually associated with incorrect values

in the system state. For example, a circuit or a program computed the wrong value, or inaccurate information was received while transmitting data.

- A failure is a non-performance of some due or expected action. A system is said to have a failure if the service it delivers to the user deviates from compliance with the system specification for a specified period of time. A system may fail either because it does not act according to the specification or because the specification does not adequately describe its function.

In short, an error is the manifestation of a fault in the system, and a failure is the manifestation of an error on the service (Laprie, 1985). For example, consider an autonomous car that uses a sensor to measure the distance to nearby objects. Suppose the distance sensor gets misaligned because of a minor physical shock (fault). As a result, it starts giving wrong distance readings (error). The system then incorrectly calculates that there's enough space to change lanes, but there isn't. The car swerves and triggers the emergency brakes to avoid a collision, stopping the car's regular service because of the incorrect behavior (failure).

2.1.2. Fault Classification

Faults can occur at different stages of a system's development. At the highest level, specification errors arise from mistakes in the system's initial design, such as algorithmic or architectural flaws, potentially causing software and hardware faults.

In the next phase, while transforming specifications into physical software or hardware, implementation errors may appear. These are often introduced through poor design, incorrect component selection, low-quality construction, or software coding errors.

Another source of faults is component failure, which can typically stem from fabrication defects, such as flaws during the manufacturing process, random device faults, or the natural wear-out of components.

The fourth and final category comes from external factors. These include environmental disturbances such as radiation, extreme temperatures, vibrations, or electromagnetic interference, which can alter the system's behavior, for example, by flipping memory values. As illustrated in Figure 2.1, along with component failures, these environmental effects are responsible only for hardware faults. However, other external disturbances may be introduced by users, either accidentally through incorrect inputs or intentionally through malicious actions like hacker intrusions, which may also lead to software errors.

Johnson (1984) defines the three most important techniques to maintain or improve the normal performance of a system as fault avoidance, fault masking, and fault tolerance. These techniques focus on different levels or ways to handle faults. To better understand their application, Figure 2.2 illustrates where each technique operates.

Fault avoidance aims to prevent faults from occurring in the first place, whereas fault masking works to prevent already occurring faults from affecting the system's data. Fault tolerance, on the other hand, does not hide the faults; instead, it allows the system to recover from them and continue regular operation.

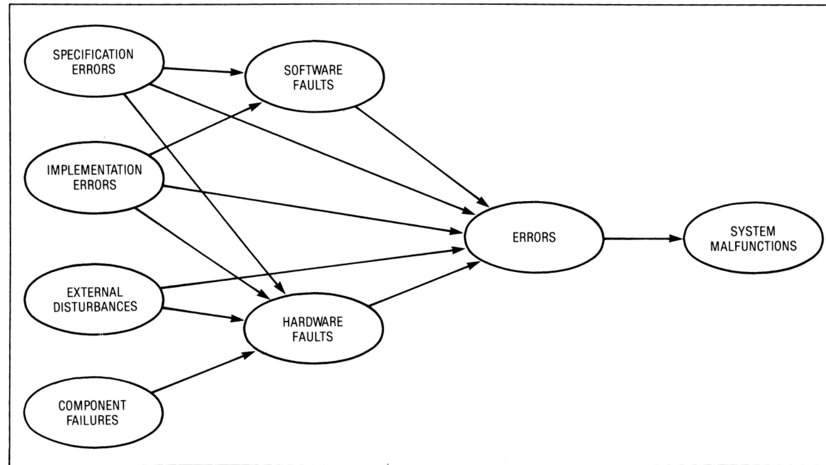

Figure 2.1: The cause-and-effect relationship of faults (Johnson, 1984).

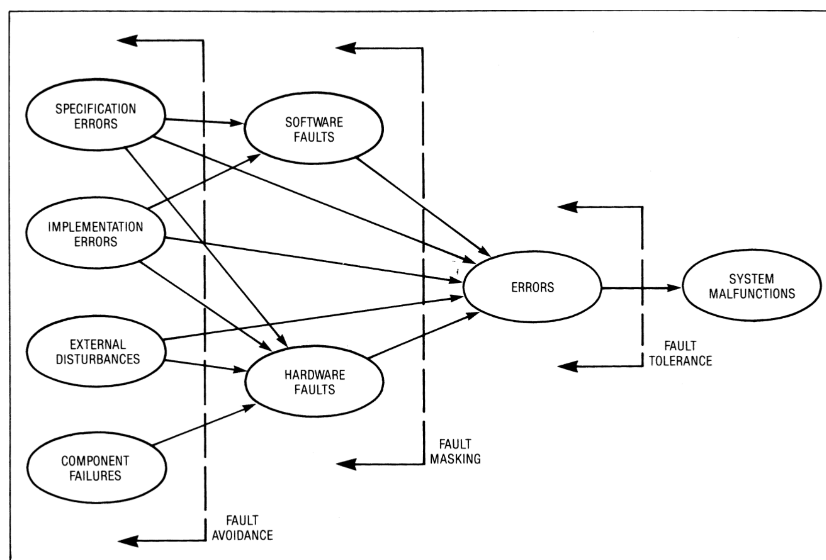

Figure 2.2: Barriers constructed by fault avoidance, fault masking, and fault tolerance (Johnson, 1984).

In addition to knowing the cause of a fault, it is important to determine its nature (software or hardware) and duration. This project focuses on hardware faults, which can be classified as permanent, transient, or intermittent, depending on how long they persist (Avizienis, 1976).

Permanent faults remain active after their occurrence until they are repaired. They are usually caused by physical damage to the hardware, such as broken components or connections, short circuits, or stuck-at bits in the architecture.

In contrast, transient faults are active only for a short period of time. During this brief interval, the fault can cause a component to malfunction, but its functionality is fully restored afterward. Due to their short duration, transient faults are difficult to detect directly, but they can often be inferred from the errors they propagate. These faults are caused mainly by environmental factors such as alpha particles, temperature fluctuations, electromagnetic interference, or power issues. They are

also known as soft errors and have historically been one of the most common types of hardware faults in digital systems. For example, Random Access Memory (RAM) is particularly susceptible to transient faults, which can cause bit flips and lead to data corruption or system crashes. Although advances in manufacturing and error mitigation techniques have helped reduce their impact, soft errors remain a relevant challenge in designing reliable systems.

When a transient fault becomes active periodically, it is classified as intermittent. Implementation flaws, hardware wear-out, or unexpected operating conditions typically cause these faults. Unlike transient faults, they do not disappear completely. Instead, they alternate between active and inactive states. When inactive, the component functions normally; when active, it malfunctions. An example is a loose electrical connection that causes the fault to appear intermittently.

2.1.3. Dependability Metrics

To develop dependable systems, fault tolerance is usually used with other methods, such as fault avoidance, fault removal, and fault forecasting. Dependability refers to the ability of a system to deliver its intended level of service to its users. Fault avoidance and tolerance are considered preventive tools to either avoid faults or continue providing service despite their occurrence. On the other hand, error removal and forecasting are validation tools aimed at minimizing and estimating faults, respectively (Laprie, 1985).

Because fault tolerance is about making machines or systems more dependable, it is essential to have proper metrics to evaluate such dependability. The three primary attributes or metrics of dependability are reliability, availability, and safety.

Reliability

Denoted by $R(t)$, reliability is the probability that a system operates without a failure in the interval $[0, t]$, given that the system was performing correctly at time 0. This metric reflects the system's ability to deliver correct service continuously. The reliability function $R(t)$ is also called the survivor function and is illustrated in Figure 2.3. High reliability is required when a system is expected to operate without interruptions. For example, typical requirements for flight control systems are a reliability of 0.9999999 during a three-hour time period.

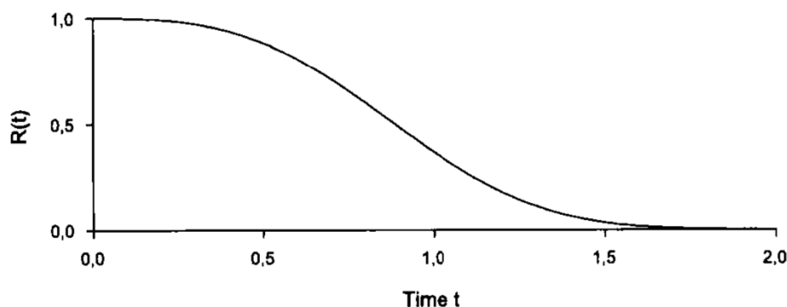

Figure 2.3: The reliability (survivor) function R(t) (Rausand and Høyland, 2004).

Availability

Denoted by $A(t)$, availability is the average fraction of time over the interval $[0, t]$ during which the system is operational. A closely related metric, point availability $A_p(t)$, represents the probability that the system is functioning correctly at a specific instant in time t. It reflects how frequently a system fails and how quickly it recovers. This metric is more suitable for applications where occasional brief downtimes are tolerable, but prolonged unavailability would cause high costs or risks.

According to the well-known availability metric of "The 9s", if an online banking system achieves a four-nines uptime (99.99%), the total annual downtime would be less than 53 minutes. Table 2.1 shows how each additional nine significantly reduces the downtime of an application.

Uptime (%)	Downtime (%)	Downtime/Year	Downtime/Month
98.0%	2%	7.3 days	14.6 hours
99.0%	1%	3.7 days	7.3 hours
99.8%	0.2%	17.5 hours	1.5 hours
99.9%	0.1%	8.8 hours	43.8 minutes
99.99%	0.01%	52.6 minutes	4.4 minutes
99.999%	0.001%	5.3 minutes	26.3 seconds
99.9999%	0.0001%	31.5 seconds	2.6 seconds

Table 2.1: The 9s (IBM Corporation, 2021)

Unlike reliability, a system can exhibit high availability even if it experiences frequent failures, as long as each failure is resolved quickly. For instance, a device that fails every hour but recovers within one second can still be considered highly available but not highly reliable.

Safety

Denoted by $S(t)$, safety is the probability that a system either performs its function correctly or discontinues its operation in a fail-safe manner during the interval $[0, t]$, assuming the system was operating correctly at time 0. Safety can be considered an extension of reliability, but focused specifically on catastrophic failures. While reliability treats all failures equally, safety distinguishes between fail-safe and fail-unsafe failures. Safety is especially critical in applications where a failure could lead to human injury, loss of life, or environmental disaster. Levels of Hardware Fault Tolerance (HFT) are defined in functional standards IEC 61508 (International Electrotechnical Commission, 2010) and IEC 61511 (International Electrotechnical Commission, 2016), mainly for safety purposes. Broadly speaking, the higher the required Safety Integrity Level (SIL), the more hardware faults the system must be able to tolerate. Systems or functions with one level of hardware fault tolerance (HFT = 1) are designed to endure a single dangerous (fail-unsafe) failure.

Other common dependability metrics related to reliability include:

1. **Failure Rate ($\lambda(t)$)**: is the expected number of failures per unit of time. For hardware, the typical evolution of failure rate over the lifetime of a system is illustrated by the bathtub curve shown in Figure 2.4.

$$\lambda(t) = \frac{f(t)}{R(t)}$$

   Where:

   - $f(t)$: failure density function (the probability that a failure occurs exactly at time t)
   - $R(t)$: reliability function (the probability that the system has not failed up to time t)

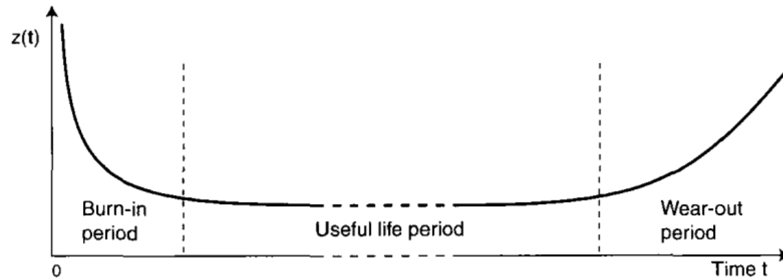

Figure 2.4: The bathtub curve (Rausand and Høyland, 2004).

2. **Mean Time to Failure ($MTTF$)**: is the expected time until the first failure occurs in a system.

$$\text{MTTF} = \int_0^\infty R(t)\,dt$$

   Where:

   - $R(t)$: reliability function

3. **Mean Time to Repair ($MTTR$)**: is the average time required to repair a failed system and return it to operational condition.

$$\text{MTTR} = \frac{\text{Total repair time}}{\text{Number of repairs}}$$

4. **Mean Time Between Failures ($MTBF$)**: is the average time between two consecutive failures in a repairable system. It includes both uptime and downtime (repair time).

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

5. **Fault Coverage (FC)**: is the probability that a system successfully detects and handles a fault, preventing it from causing a failure or incorrect behavior.

$$\text{Fault Coverage} = \frac{\text{Number of detected}}{\text{Total number of faults}}$$

2.1.4. Redundancy

Common to all fault tolerance approaches is a certain amount of redundancy. Although there are different forms, such as software, information, and time redundancy, hardware redundancy is the most commonly used technique. It consists of the physical replication of hardware components to detect, mask, or recover from faults during system operation.

In the past, one of the main limitations of hardware redundancy was the extra cost of adding additional components. With the continued reduction in hardware prices, this is no longer a significant drawback. However, other constraints, especially power consumption, may still limit its use in many applications. The increased weight, size, and time required for design, fabrication, and testing must also be carefully considered to determine the most effective way to incorporate redundancy into a system.

There are basically three forms of hardware redundancy or replication. First, passive or static redundancy techniques aim to mask faults immediately. They do not involve any detection, isolation, or repair of a faulty component. A classic example is Triple Modular Redundancy (TMR), illustrated in Figure 6.1, in which three identical modules execute the same operation and a voter determines the final output based on a majority vote, masking the effect of a single faulty module. Second, active or dynamic redundancy relies on fault detection and localization, followed by the replacement of the faulty component with a spare. Instead of masking the fault, the system is reconfigured using standby components, which are activated when a failure affects the currently active component. This project focuses on dynamic redundancy through dynamic partial reconfiguration, with further details later in the document. Third, hybrid redundancy combines passive and active approaches, using fault masking to prevent the fault from affecting the system and fault detection to allow a spare module to be switched in to replace the faulty module. It enables reconfiguration with no system downtime. Hardware redundancy can therefore range from simple duplication to complex architectures capable of dynamic reconfiguration. As previously mentioned, it incurs significant overheads. Hence, it is reserved for systems where physical access for maintenance is costly or limited, and especially used in critical systems where failure is not an option or downtime must be minimized.

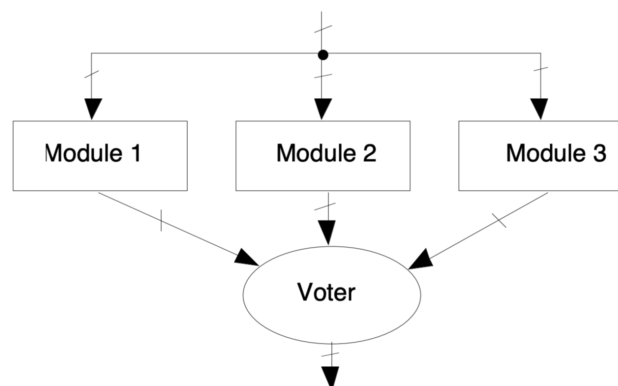

Figure 2.5: Triple Modular Redundancy (Mitra and McCluskey, 2000).

2.2. FPGAs and Dynamic Reconfiguration

FPGAs (Field Programmable Gate Arrays) are highly flexible integrated circuits that can be reprogrammed to implement different digital logic functions. Since their introduction in the 1980s, they have evolved rapidly and are now used in a wide range of applications across many industries, such as aerospace, automotive, and telecommunications. Unlike ASICs (Application-Specific Integrated Circuits), which are fixed once manufactured, FPGAs allow the reprogramming of their logic and routing even after deployment. Dynamic Partial Reconfiguration (DPR) takes this flexibility a step further by allowing part of the device to be reconfigured while the rest continues operating normally. As FPGAs are initially configured through a full configuration bit file, DPR enables the modification of reconfigurable regions by loading partial bit files without compromising the integrity of the applications running on the portions of the device that are not being updated. Xilinx rebranded Partial Reconfiguration as Dynamic Function eXchange (DFX) in 2020. The Figure 2.6 illustrates the premise behind Dynamic Function eXchange.

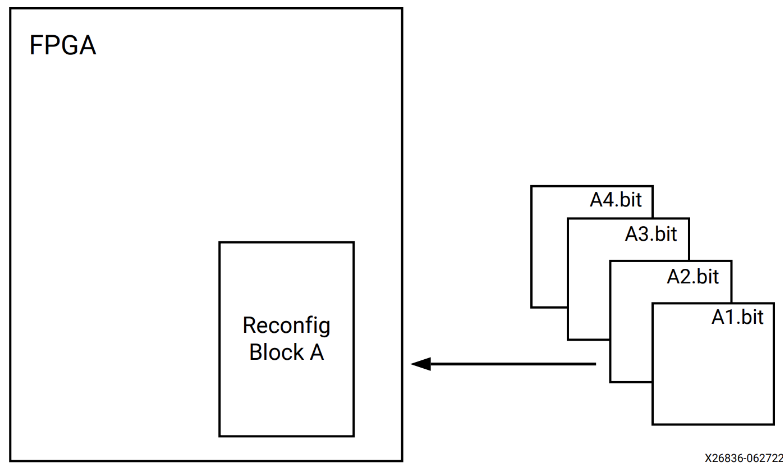

Figure 2.6: Basic Premise of Dynamic Function eXchange (AMD Xilinx, 2024).

DFX capability brings several advantages, such as:

- Reducing size, weight, and power consumption, as multiple applications can share a single FPGA, dynamically loading only the necessary functions.
- Enabling new techniques in design security that combine DFX and asymmetric cryptography.
- Improving FPGA fault tolerance.
- Keeping deployed systems updated with fixes and new features.
- Accelerating updates by reconfiguring only specific regions instead of the entire device.
- Improving reliability by maintaining uninterrupted operation during updates.

Dynamic Function eXchange enables new types of FPGA designs that would otherwise be impossible to implement. However, it is not supported on all FPGAs. Currently, the only two FPGA vendors commercially supporting partial reconfiguration are AMD and Intel.

AMD, which acquired Xilinx in 2022, has been a pioneer in supporting partial reconfiguration for over two decades, starting with the XC6200 series and continuing through FPGA families such as Virtex (from Virtex-II to Virtex-7), as well as Artix-7 and Kintex-7, which are part of the 7-series devices. Over time, architectural improvements such as finer-grain reconfiguration, flexible floorplanning, and faster configuration interfaces like Internal Configuration Access Port (ICAP) and the Media Configuration Access Port (MCAP) have enhanced DPR capabilities. Newer devices like Zynq-7000 and Ultrascale(+) even enable partial reconfiguration of high-speed peripherals. See Appendix A (AMD Xilinx, 2024) for a complete list of AMD Xilinx supported devices.

Intel supports partial reconfiguration on Stratix V, Arria 10, and Cyclone V devices, using Adaptive Logic Modules (ALMs) and frame-based programming. Newer devices like Stratix 10 introduce independent sectors with Secure Digital Managers (SDMs) to enhance reconfiguration speed and flexibility (Vipin and Fahmy, 2018).

2.2.1. Structure of an FPGA

Conceptually, in FPGAs, we can distinguish between an application or hardware logic layer and a configuration memory layer (Becker et al., 2007). The application layer contains the logic and routing hardware resources needed to implement a circuit physically. On the other hand, the configuration memory layer stores the data in a bitstream, a binary file that fully defines the system's functionality and controls the configuration of the application layer. FPGAs achieve their unique reprogrammability and flexibility thanks to this structure.

Hardware resources in a FPGA include (Eastland, 2025) (Figure 2.7):

- Configurable Logic Blocks (CLB): Basic building blocks of an FPGA and what give it its flexibility. FPGAs contain thousands of them and can be programmed to implement almost any logic function. Each CLB is made up of LUTs (look-up tables) and flip-flops.

- Digital Signal Processing (DSP) Slices: Specialized blocks for digital signal processing tasks like multiplication or filtering. Using a DSP slice for these operations is much more efficient than implementing them using many CLBs.

- Block Random Access Memory (BRAM): The internal memory of the FPGA. Each BRAM block has a fixed size but can be split or combined to get the necessary memory size. It is also capable of various operational settings and can support special functionality such as error correction.

- Transceivers: Dedicated blocks for sending and receiving high-speed serial data. They are useful for moving large amounts of data in and out of the FPGA without using up logic resources.

- Input/Output (IO) Blocks: Blocks that handle input and output between the FPGA and the outside world. They belong to IO banks and are more flexible than transceivers, but operate at lower speeds.

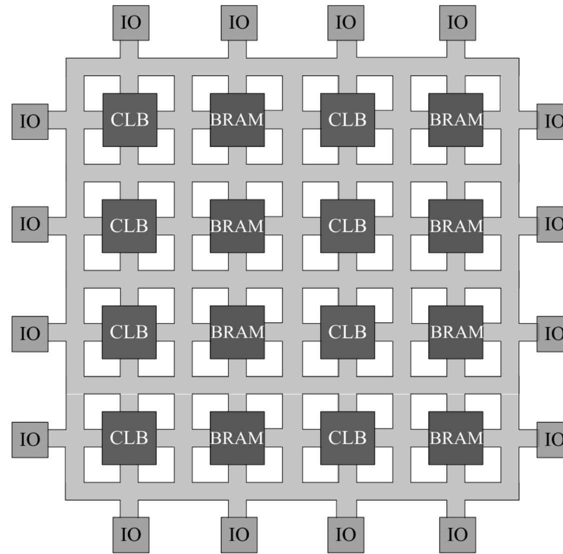

Figure 2.7: Generic view of FPGA architecture (Li et al., 2019).

2.2.2. DFX Overview

An FPGA is programmed with a bitstream, a sequence of commands and data that tells the device how to connect its internal logic gates and configure its resources to implement the desired functionality. It includes the description of the hardware logic, routing, and initial values for both registers and on-chip memory. Typically, the bitstream configures the entire chip, but it is actually composed of smaller commands that target specific elements of the device. Because of this structure, it is possible to create a bitstream that modifies only part of the FPGA while keeping the rest unchanged and operational. This capability is the foundation of Partial Reconfiguration or Dynamic Function eXchange (DFX) (Billauer, 2019).

A Partial Reconfiguration bitstream can be delivered to the FPGA using any interface that supports loading bitstreams, such as JTAG, or directly from within the FPGA's logic via the Internal Configuration Access Port (ICAP), as long as the process occurs while the FPGA is running.

The part of the FPGA that remains unchanged is referred to as static logic. It is static in two ways: functionally, because it runs without interruption from the FPGA's start; and physically, because it is fixed in regions where no further modifications are allowed. In contrast, there is reconfigurable logic, which can dynamically change the logic functions implemented. For Partial Reconfiguration to be possible, there must be a clear distinction between static and reconfigurable logic. This separation is achieved through hierarchical design, which ensures that static logic remains unaffected during reconfiguration.

The design is organized as a collection of components, where those with reconfigurable logic are explicitly assigned, through floorplanning, to a specific physical region on the FPGA, while static logic fills the remaining areas. These regions with reconfigurable logic are called reconfigurable partitions (RPs), and each can host different reconfigurable modules (RMs) but only one at a time. RMs implement different functions and are loaded dynamically. In Vivado, the unit used for floorplanning to define each RP's physical region is called a Pblock.

As multiple RMs can exist for a single RP, it is important to understand that a configuration is a complete design that includes one RM per RP. There may be several configurations in a DFX project, each corresponding to a different combination of RMs. The parent configuration refers to the base design that contains all the static logic and one initial RM for each RP. All remaining configurations are defined as children of that parent, where one or more RMs are replaced with different versions.

The parent configuration is implemented first: Vivado synthesizes the static logic and the initial RMs separately, followed by a global place and route (PnR) of the entire design. The result of this implementation serves as a reference point for all subsequent child implementations. For each additional RM, a child implementation is carried out individually, with one significant difference from the parent implementation: only the reconfigurable logic is synthesized, as the static logic was already completed during the parent implementation. Moreover, the PnR of the static logic remains fixed, inherited from the parent's static design checkpoint, and only the new RM is placed and routed within its RP. All implementations (parent and child) must satisfy timing constraints for the complete design. Meeting these constraints ensures that functionality is preserved both before and after reconfiguration. Therefore, even though static and reconfigurable logic are synthesized separately, implementation always considers timing closure globally. Additionally, all implementations generate a complete and a partial bitstream per RM, ensuring flexibility for both complete configurations and dynamic reconfigurations.

One final element to consider is the routing interface between static and reconfigurable logic. This interface is managed through partition pins, which define the precise connection points between static and reconfigurable logic, marking where the static routing ends and the reconfigurable routing begins. Both the parent and all child implementations must agree exactly on the locations of these partition pins.

For a detailed description of the Vivado DFX Project Flow, see Appendix B.

2.2.3. Other Common Applications

Several scenarios benefit significantly from using DFX, beyond improving the tolerance to hardware faults. One of the most representative examples is a multiport network interface, such as a 40G OTN (Optical Transport Network) muxponder application. In such systems, each port may need to support different communication protocols, but it is impossible to predict which one will be used in advance. Without DFX, all possible interface protocols must be implemented for every port, resulting in excessive resource usage, even though only one protocol is active at a time. With DFX, each protocol interface can be implemented as a reconfigurable module that is

dynamically swapped in as needed. This approach significantly reduces logic usage, removes unnecessary multiplexers, and improves overall design efficiency. A wide range of systems can benefit from this principle. Software-Defined Radio (SDR), for example, is a well-known case where mutually exclusive functionality greatly benefits from this kind of dynamic multiplexing.

In packet processor applications, DFX allows the FPGA to adapt to the type of traffic it receives. For instance, the FPGA can receive a packet whose header contains a partial BIT file. This file is processed and used to reconfigure a specific region of the FPGA to perform a particular function. This mechanism makes the system far more flexible and removes the need to predefine all possible functionalities, enabling highly adaptive processing.

Some applications are only made possible through DFX, such as secure configuration schemes based on asymmetric encryption. In these cases, the FPGA initially loads a minimal configuration that contains no sensitive information and internally generates a public-private key pair. The public key is sent to the host, which uses it to encrypt a partial bitstream containing the main system logic. This bitstream is then returned to the FPGA, decrypted internally, and used to complete the reconfiguration. This setup offers several security advantages: the private key never leaves the device, it is stored in volatile SRAM (and therefore lost when power is removed), and it is not confined to any fixed region of the FPGA, making it extremely difficult to extract, even with physical access to the FPGA.

2.3. RISC-V Processor

This project aims to build upon the work initiated by my colleague Davó Laviña (2022) and later extended by Carpio Cuenca (2024).

Davó Laviña set out to design a low-power, scalable, and configurable Network-on-Chip (NoC), and to use it to interconnect at least two components within a RISC-V processor. To do so, he explored different RISC-V implementations. He selected the VeeR family, specifically the SweRV EL2 core (now VeeR EL2), due to its implementation in SystemVerilog, known license, and simple, modular, easy-to-understand, well-documented architecture. Carpio Cuenca subsequently modified the designed NoC to detect and tolerate link failures.

2.3.1. RISC-V

RISC-V is an open standard instruction set architecture (ISA) based on established Reduced Instruction Set Computer (RISC) principles. Unlike commercial ISAs such as ARM, MIPS, or x86, RISC-V is open and free, meaning anyone can use, modify, and implement it for any purpose without paying license fees or royalties. This openness has encouraged its adoption in both academic and industrial settings.

The ISA was defined with a clear goal: to be suitable for nearly any computing device (Waterman, 2016). To achieve this, RISC-V was deliberately not over-architected for any particular microarchitectural pattern, implementation fabric, or

deployment target. Its design avoids architectural techniques that might benefit certain implementations but limit others.

Another key consequence of the goal to make RISC-V truly ubiquitous was that the ISA must remain open and free to implement. The benefits of an open standard are numerous, but perhaps the most important is the potential for a wide variety of processor implementations. Open access to the specification fosters the development of both open-source and proprietary processors, encourages innovation in microarchitecture, facilitates collaboration between academia and industry, and reduces development costs.

One of the features that makes RISC-V suitable for a wide range of devices is its modular design. The instruction set is built around a simple and minimal base ISA that can be extended with various optional features, such as advanced arithmetic operations, floating-point support, atomic instructions, and vector extensions. This modularity allows RISC-V to be tailored to applications ranging from low-power embedded systems to high-performance servers, without introducing unnecessary complexity, cost, or power consumption.

2.3.2. VeeR EL2 Core

As previously mentioned, the selected core for integrating the Network-on-Chip (NoC) was VeeR EL2, an open-source 32-bit CPU developed by Western Digital in collaboration with the CHIPS Alliance. It is the smallest member of the VeeR family and complies with RV32IMC specification, supporting integer, compressed, multiplication/division, and bit manipulation instructions. It is designed to run at a target frequency of 600 MHz when implemented using 16nm technology, offering a well-balanced compromise between performance and power efficiency.

VeeR EL2 core contains a 4-stage, scalar, in-order pipeline with a branch predictor. The four stages depicted in Figure 2.8 are:

1. Fetch (F): retrieves instructions and calculates the program counter parallel to instruction memory access.

2. Decode (D): aligns and decodes the instructions, and prepares addresses for memory operations.

3. Execute/Memory (X/M): performs arithmetic and memory operations.

4. Retire (R): updates the registers with the results of the executed operations.

In addition, the processor includes a divider unit that operates outside the main pipeline and incorporates stall points in the Fetch and Decode stages to manage execution flow.

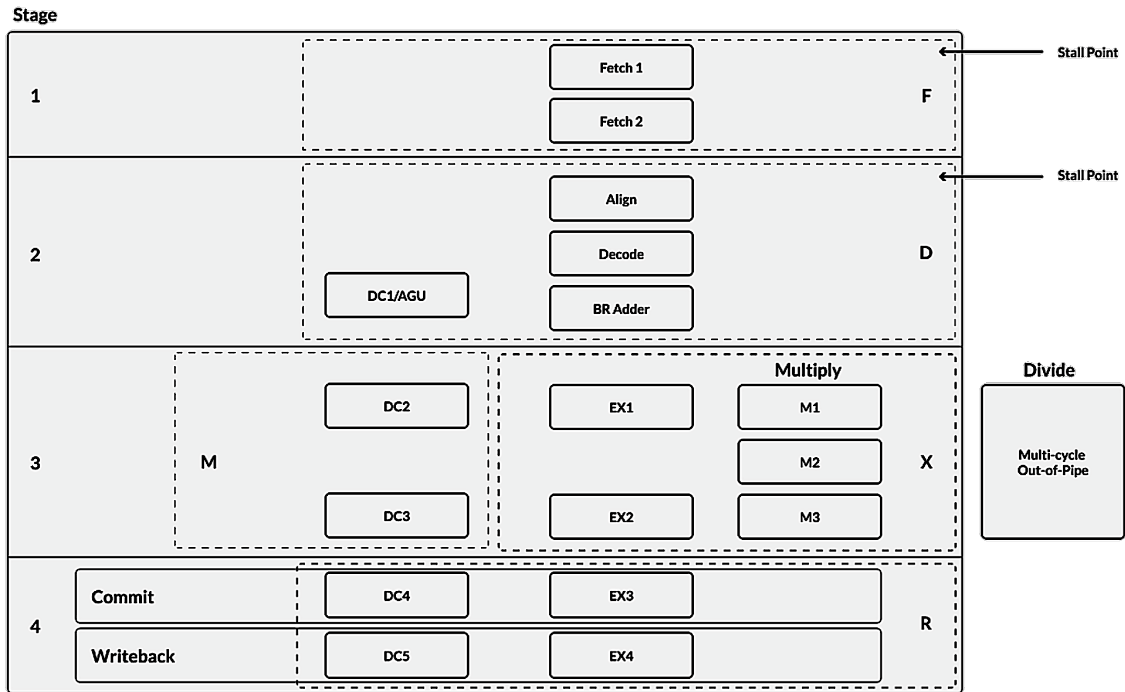

Figure 2.8: VeeR EL2 Core Pipeline (CHIPS Alliance, 2022).

The core complex is composed of the following main functional units:

- **IFU** (Instruction Fetch Unit): responsible for the fetch phase, where the instruction is retrieved from memory using the address provided by the program counter.

- **DEC** (Decoder): interprets the instructions fetched by the IFU, identifying the operands needed for execution in the EXU and generating the corresponding control signals.

- **EXU** (Execution Unit): executes the instruction using different submodules depending on the execution pipeline type. It is the functional unit where the NoC is integrated.

- **LSU** (Load-Store Unit): handles load and store instructions, accessing memory to read or write data as needed.

The core complex also includes key components such as closely-coupled instruction and data memories (ICCM and DCCM), an instruction cache, a programmable interrupt controller, and four system bus interfaces for instruction fetch, data access, debugging, and DMA transfers (configurable as 64-bit AXI4 or AHB-Lite buses) (CHIPS Alliance, 2022). Figure 2.9 depicts VeeR EL2 Core's functional blocks and feature set.

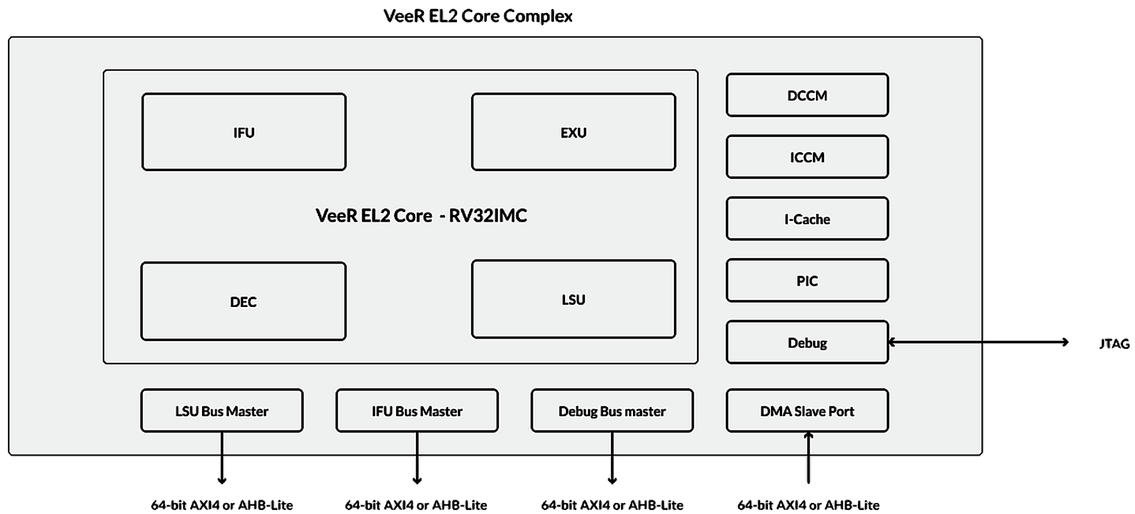

Figure 2.9: VeeR EL2 Core Complex (CHIPS Alliance, 2022).

2.3.3. Integrated NoC

The Network-on-Chip (NoC) was designed to be a low-cost and resource-efficient communication infrastructure, minimizing its impact on the overall system. The architecture (Figure 2.10) is based on a two-dimensional indirect mesh topology with full-duplex links, allowing simultaneous bidirectional data transmission. It is parameterizable in height and width, making it adaptable to different system sizes and design constraints. Routers are addressed using Cartesian coordinates, which simplifies routing decisions.

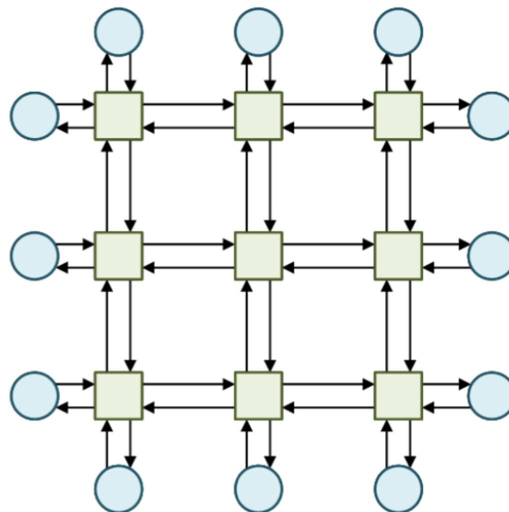

Figure 2.10: Network Architecture (Carpio Cuenca, 2024).

Two routing algorithms are implemented. The primary one is Deterministic Dimensional Order Routing (DOR), which always chooses the shortest path following a specific order to traverse the network (first moving horizontally, then vertically). As a fallback, a Static Priorities (SP) algorithm is included, which provides alternative, non-minimal paths based on fixed priorities depending on the router's position

in the network. Using the two algorithms allows adaptation to network conditions in case of link failures or congestion.

The switching strategy is segmented circuit switching, where data is broken down into flits that are sent in sequence. Three types of flits (header, data, or tail) travel in parallel over dedicated buses. Routers use a round-robin arbitration method to manage simultaneous transmission requests fairly.

Communication between nodes follows a simple handshake protocol using `en` (enable) and `ack` (acknowledge) signals to ensure data validity and delivery confirmation. Additionally, two new control signals have been introduced: `alive`, which informs whether a link is operational, and `rej` (rejected), which indicates that no valid route to the destination was found.

This NoC was designed to connect components of the VeeR EL2 core. In particular, two modules from the execution unit (EXU) were selected: the multiplier and the divider. The EXU is responsible for executing instructions, meaning it receives operands (provided by the fetch and decode stages) and outputs the result of the operation to be used in the retire stage. It also handles internal tasks such as branch prediction and program counter calculations. To execute these operations, the EXU initially instantiated three modules (an ALU, a multiplier, and a divider), all connected through point-to-point signal buses. So, for them to communicate through the NoC, their interfaces had to be adapted using a set of additional modules that convert the standard signals into network packets sent flit by flit, following the NoC's communication protocol. These modules are:

- Sender: takes a fixed-length data bus, splits it into flits, and sends them across the network.

- Receiver: reconstructs the packet from the received flits and sends it over a bus.

- Wrapper: encapsulates either the multiplier or the divider. Each wrapper connects directly to the NoC and behaves like the original module it wraps, using both a sender and a receiver to handle internal communication.

To integrate the NoC into the processor and connect the multiplier and the divider to it, a 3×3 mesh (configurable via preprocessor definitions) was instantiated, with the required components placed around the network: `mul_sender`, `mul_receiver`, `div_sender`, `div_receiver`, `mul_wrapper`, and `div_wrapper`. Another necessary modification was the introduction of an internal clock signal, `clk_noc`, which drives all NoC components, including the senders and receivers. This separate clock is required because the NoC needs several cycles to transmit a complete packet. In the tests performed by David, this clock was configured to run 16 times faster than the main system clock.

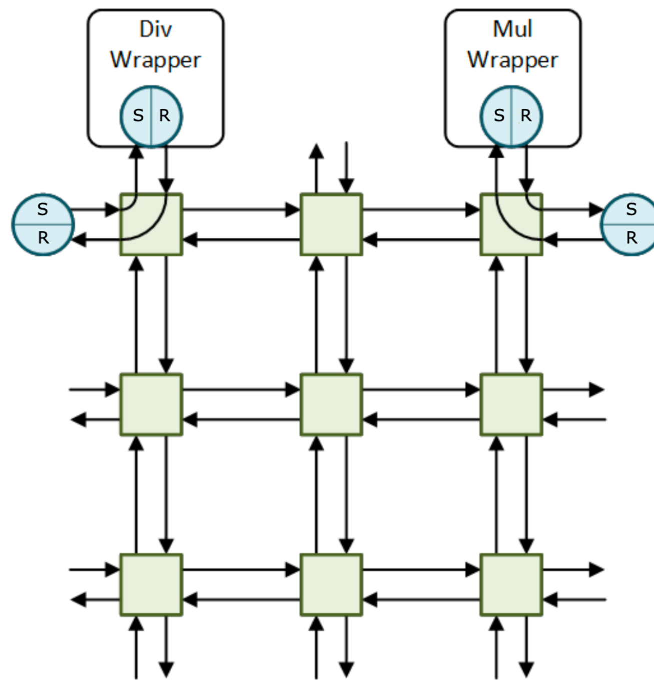

Figure 2.11: Connection of the sender (S) and receiver (R) modules, and the divider and multiplier wrappers to the NoC. (Carpio Cuenca, 2024).

Chapter 3

Tools and Technologies

Several tools and technologies have been used to design, model, simulate, and document the RISC-V processor with dynamic reconfiguration to carry out this project.

The following are the primary tools utilized in each phase.

3.1. Modeling and simulation

- **Python:** Programming language used to model the NoC and its behavior in response to fault injections. Chosen for its popularity in system simulation, its simple and expressive syntax makes code writing and debugging easier. It also has a wide collection of specialized libraries for various types of simulation and analysis that we used in a Python notebook to generate and visualize results. These libraries include:

  - **NumPy** and **SciPy** for numerical computations,
  - **Matplotlib** for data visualization,
  - **Pandas** for data manipulation.

- **XML:** Markup language used to describe the NoC structure, including its nodes and associated metadata, such as the number of LUTs and FFs per node. Its hierarchical structure made it easy to parse and integrate with the Python model.

- **Bash script:** File written to automate the execution of the simulation, making it easier to run a large number of tests consecutively (e.g., 5,000 iterations).

- **Visual Studio Code (VSCode):** Development environment that supports multiple languages (Python, XML, Bash) and integrates well with Python notebooks, which makes it suitable for both developing and visualizing the simulation model.

3.2. Hardware implementation

- **VeeRwolf:** FuseSoC-based reference platform for the VeeR family of RISC-V cores, specifically supporting VeeR EH1 and VeeR EL2. VeeRwolf provides a standardized environment to develop, simulate, and implement these cores. We used adapted versions of the VeeRwolf Nexys and VeeRwolf Sim VeeR EL2 cores for this project.
- **FuseSoC:** Package manager and build tool for HDL (Hardware Description Language) cores and IPs. FuseSoC was used to automate the VeeRwolf core build process for simulation and FPGA implementation, managing dependencies efficiently. It simplifies handling multiple modules and ensures reproducible synthesis and simulation workflows.
- **Verilator:** An open-source Verilog/SystemVerilog simulator used to perform fast, cycle-accurate simulations of RTL code. It enabled efficient debugging of the NoC and processor modules.
- **Vivado:** The official integrated design environment (IDE) developed by AMD (formerly Xilinx) for designing and implementing FPGA and SoC hardware. It was used to synthesize and implement the RISC-V processor and the NoC and generate bitstreams for the Artix-7 FPGA on the Nexys 4 development board. Vivado also generates useful reports to evaluate the design, such as physical resource usage, power consumption, and timing. In addition, Vivado's Dynamic Function eXchange (DFX) feature can be used to define and manage reconfigurable partitions.
- **YAML:** Data serialization format commonly used for writing configuration files. Since the VeeRwolf `.core` files were written in YAML, learning how to read and write this format was necessary to edit and customize these files.
- **SystemVerilog:** Hardware description language used in the design of the RISC-V processor, the NoC, and the modules required for communication with it. Understanding and working with SystemVerilog were essential for analyzing the design, identifying the needed modifications to enable DFX, and supporting the overall project development.
- **FPGA (Field-Programmable Gate Array):** The target hardware platform chosen to implement and test the design was the Nexys 4 Artix-7 FPGA. An essential feature for selecting this board was its support for partial reconfiguration. This capability would have provided a real environment to deploy and test the dynamically reconfigurable RISC-V processor with the NoC. However, actual fault injection tests on the hardware were not performed.

3.3. Project management

- **Git:** Version control system used while programming the simulation model. It helped to keep track of the different versions of the code, allowing us to go

back to previous states if something broke, and making it easier to experiment with changes without losing stable versions.

- **GitHub:** Git-based code hosting platform used through a repository to store all the files related to the simulation model, and other documents related to the project. It ensured that both the author and the tutor could access the latest versions.

  In addition, GitHub's Projects feature was used to organize the workflow with a Kanban board, where the status and priority of each task were indicated. This feature helped maintain a clear overview of what was pending, what was being worked on, and what had already been completed. The board was also used to record meeting notes and tutor feedback, ensuring that comments and suggestions were directly linked to the corresponding tasks.

  Project's repository available at:

  `https://github.com/ABSysGroup/TFG2024-25_RISC-V_reconfigurable`

- **Gmail Chat, Google Meet, and Calendar:** For quick questions or clarifications, the integrated chat in Gmail was used to maintain an agile line of communication. Additionally, Google Meet was used for remote meetings when necessary to follow up on the project's progress. Google Calendar was used to schedule and organize in-person and remote sessions, ensuring smooth coordination between the author and the tutor.

3.4. Report writing

- **LaTeX:** Typesetting system used for writing the report because it creates high-quality documents, especially for works with equations, references, and bibliographies. It is widely used in academia and allows the focus to be on the content rather than formatting, ensuring a professional result. Additionally, a template provided by the Faculty of Computer Science (FDI) for final degree projects was used as a starting point, ensuring that the document followed the required formatting and structure for submission.

- **Overleaf:** Online LaTeX editor chosen to edit and compile the document. It makes collaboration and real-time editing easier and eliminates the need for local installations. Overleaf has version control, which helps manage changes made to the document between the author and the project supervisor.

- **draw.io:** diagramming tool used to create illustrations included in the report. These visual representations were helpful to clarify specific processes and architectural designs, making the explanations more intuitive and easier to follow.

Chapter 4

Project Development

The development of the project began with a detailed study and analysis of the RTL design of the VeeR EL2 processor, along with the Network-on-Chip (NoC) developed and integrated by Davó Laviña (2022).

Some of the main reasons for integrating a NoC into the execution unit are scalability and modularity. Each functional unit or module is connected to the network as an independent node, simplifying the design, modification, and maintenance. It also makes adding new modules, such as additional multipliers or dividers, easier without significantly changing the overall architecture. Moreover, data traffic is optimized since multiple paths allow for more efficient communication management.

Another important benefit is improved fault tolerance. The enhancements by Carpio Cuenca (2024) to the NoC through link failure recovery mechanisms made the network more resilient. These improvements allowed the network to detect and adapt to physical link and router failures and temporary traffic congestion, thanks to a new `alive` signal and the support of the SP (static priorities) and DOR (dimensional order routing) algorithms, and through the use of a `rej` signal and dynamically rerouting traffic through alternative paths (explained in Subsection 2.3.3).

In this project, we aim to further enhance fault tolerance by applying dynamic reconfiguration, which allows functional units to be moved or relocated while the NoC continues to handle communications in a standardized way.

Once we had a solid understanding of dynamic reconfiguration, an essential preliminary task was clearly defining which parts of the design would be static and which would be reconfigurable. The idea is that when a fault occurs in either the divider or multiplier, the wrapper containing that unit can be dynamically relocated to another node in the NoC so that the functionality can continue working. The new route between the sender/receiver and the relocated wrapper is managed by the NoC itself, using its internal routing algorithms and redirection mechanisms. Other modules connected to the NoC include the sender and receiver for both the multiplier and the divider. These should remain part of the static logic, since, as their names suggest, they are responsible for sending and receiving data to and from the rest of the system, which is static.

The NoC topology is a two-dimensional mesh (3x3), which includes nine intermediate nodes (routers) and twelve endpoint nodes, where the modules required

by the multiplier and divider are connected. These modules use four of the twelve available endpoints, leaving eight nodes free for relocation in case of failure. The spare nodes, along with the wrappers, constitute the dynamic reconfigurable regions of the design.

4.1. Python Simulation Model

Before performing hardware tests on the FPGA or running heavy simulations, it is convenient to have a lightweight environment that allows the system's behavior under faults to be studied in a controlled way. In this specific case, a simulation model is used because it is neither feasible nor desirable to physically damage the FPGA to test fault behavior. For this purpose, a Python model was developed to replicate the structure of the RISC-V processor with the integrated NoC, including the separation between static and reconfigurable logic. This model enables flexible fault injection, analysis of error impact, and validation of fault-tolerance strategies in an automated way.

4.1.1. Scope

Before starting to code, it was essential to determine the extent of the architecture to be represented in the model. Since this project focuses on the use of dynamic reconfiguration in the modules connected to the NoC, it was decided to model only that part of the execution unit. In summary, the model, shown in Figure 4.1, represents the NoC with its intermediate nodes (routers) and endpoint nodes. These endpoint nodes include the sender and receiver modules for the multiplier and divider, the wrappers, and the spare reconfigurable areas.

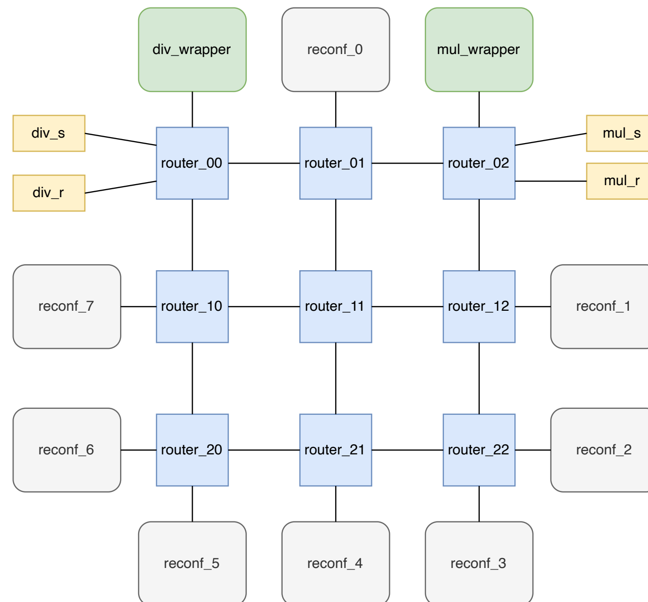

Figure 4.1: Python model system architecture

4.1.2. Files

The files that make up the model are described in Table 4.1 and are all available in the project GitHub repository at:

`https://github.com/ABSysGroup/TFG2024-25_RISC-V_reconfigurable/tree/main/python_model`

Filename	Description
`exu_noc.xml`	Input file to initialize the system. Defines NoC topology, size, module type (static or dynamic), needed resources (LUTs/FFs), and connections to routers.
`run_experiments.sh`	Bash script to automate multiple simulation executions (`simulation.py`).
`simulation.py`	Main simulation file. Parses the XML architecture, builds the model, injects faults, and logs results.
`classes.py`	Defines the main classes that model the system, including the EXU with its NoC, static and reconfigurable partitions, and fault injection, detection, and dynamic reconfiguration logic.
`reconf_results.csv`	Output file to store the number of dynamic reconfigurations performed in each simulation run before an error due to a non-tolerable fault. Each line corresponds to one simulation run.
`fault_results.csv`	Output file to store how many hardware faults were injected in each simulation run before the system halted due to a critical fault. Each line corresponds to one simulation run.
`error_type_results.csv`	Output file to log why the system failed in each simulation. Each line corresponds to one simulation run.
`faulty_partition_results.csv`	Output file to record the partition name that caused the error. Each line corresponds to one simulation run.
`plots.ipynb`	Jupyter Notebook for analyzing and visualizing results from the CSV files (jitter plots, box plots, etc.).

Table 4.1: Program files and descriptions

4.1.3. Execution Flow

The simulation process is implemented in `simulation.py` and follows a structured flow (Figure 4.2):

1. **Architecture loading and model initialization**

   The input file, `exu_noc.xml`, is parsed to extract the structure of the NoC, including the functional blocks (`mul_wrapper`, `div_wrapper`, senders, and receivers), the spare reconfigurable areas, their resource usage, and their connections to the routers. Since the multiplier and divider wrappers have different resource requirements (regarding LUTs and FFs), the reconfigurable areas are defined with sufficient resources to accommodate the larger of the two modules. Consequently, some resources remain unused (`spare`) when a smaller module is instantiated. This strategy enables the reuse of a single reconfigurable region for both modules.

   To represent routers and the rest of the modules, instances of the `Partition` and `Node` classes are created using the data from the XML file, marking their resources as either `used` or `spare` accordingly.

2. **Fault injection**

   To ensure that the simulation only stops when the system actually fails, a conditional `while` loop was implemented. This loop checks the value of the `sim` variable to decide whether the simulation should continue. If `sim` is `True`, the fault was tolerated and the simulation proceeds. If `sim` becomes `False`, the fault was not tolerable, and the simulation stops. In each iteration of the loop:

   - A random resource (LUT or FF) is selected using `get_defect()`.
   - The fault is injected using `inject_defect()`, which returns three values:
     - `sim`: (boolean) indicates whether the fault was tolerated.
     - `error_type`: (Enum) indicates the cause of the error, if any.
     - `faulty_partition`: name of the partition where the fault occurred.

3. **Fault tolerance**

   When a fault is injected into the system, the first step is to mark the affected resource as `faulty`. The resource's previous status is also verified. If the resource was marked as `spare` or was already `faulty`, the fault has no impact, is considered tolerable, and the system continues operating without the need for reconfiguration. However, additional actions are required if the resource is currently `used`.

   If the faulty resource belongs to a static region or a router, the affected node is removed from the NoC mesh, simulating a permanent fault. The system then verifies if the network can still work. On the other hand, if the fault affects a reconfigurable module, such as the multiplier or divider wrapper, the system attempts to recover by relocating the faulty module to a spare reconfigurable area that satisfies resource and connection constraints.

A wrapper may require reconfiguration in two situations:

- If a LUT or FF it uses becomes `faulty`.
- If it loses connection to its sender or receiver due to their failure or the failure of an intermediate router.

In such cases, the system begins scanning the list of available reconfigurable areas. For each candidate area, the following conditions are evaluated:

- It must offer a valid path to the corresponding sender and receiver within the NoC mesh.
- It must contain enough non-faulty LUTs and FFs to meet the wrapper's resource usage.
- It must not have been used previously by the same wrapper, avoiding repeated failures in the same node.

Once a suitable reconfigurable area is found:

- The wrapper's connection to the NoC is updated accordingly.
- Its internal list of resources is rebuilt, reflecting the current state of the new region.
- The previous area is updated to reflect its new spare state.
- The old node is recorded as tested by the wrapper, preventing repeated placements in the event of future faults.

This reconfiguration process is performed independently for the multiplier and the divider modules. If no valid area is found after checking all candidates, the system concludes that recovery is not possible and reports an error.

4. **Saving results**

   After each simulation run, the output files are updated:

   - The number of reconfigurations performed to tolerate faults is written to `reconf_results.csv` and printed in the terminal.
   - The total fault count is written to `fault_results.csv` and in the terminal as well.
   - The reason for failure is stored in `error_type_results.csv` and also displayed in the terminal.
   - The name of the last partition affected by an injected fault is stored in the file `faulty_partition_results.csv` and displayed in the terminal. This partition is the one that triggered the error.

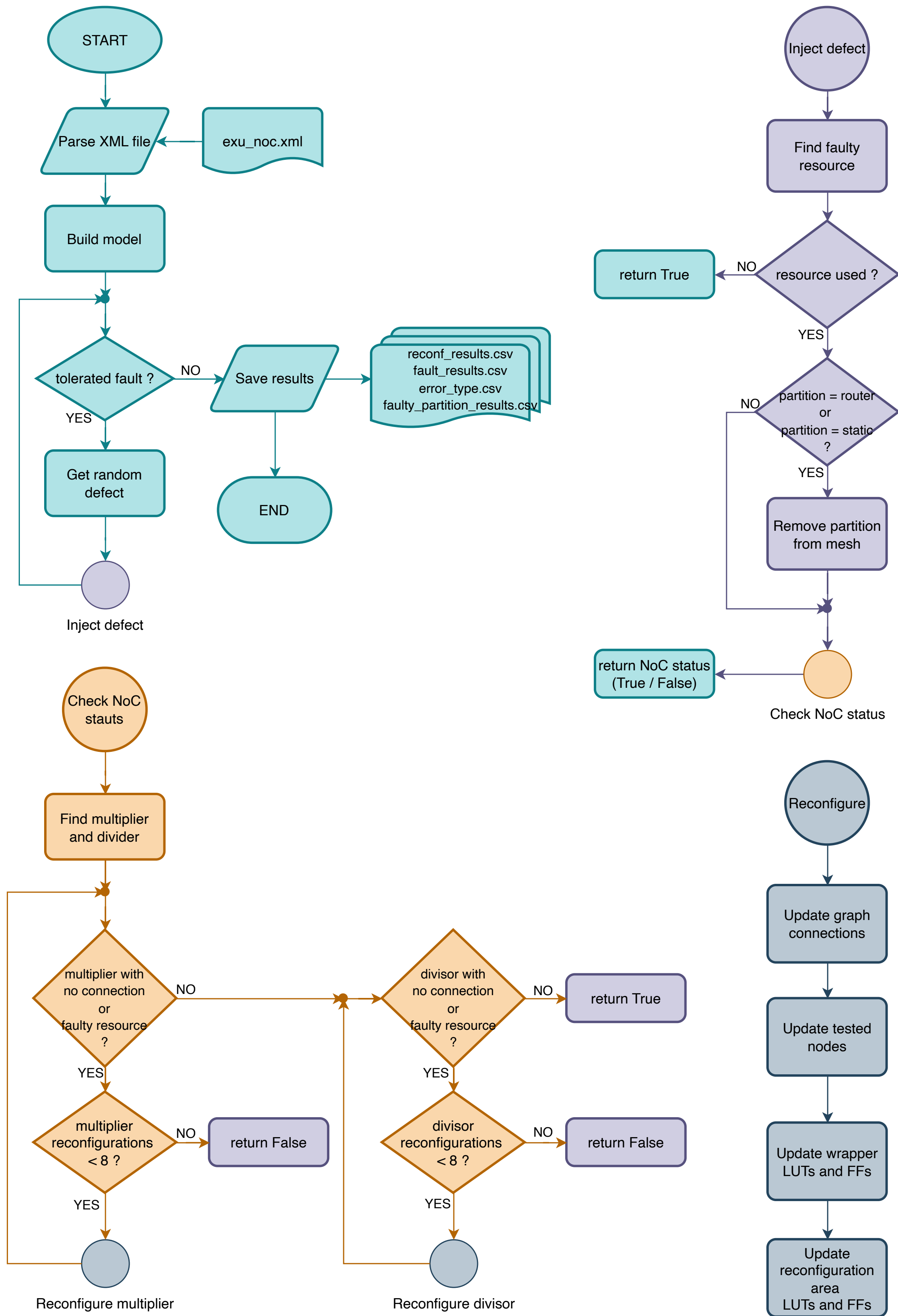

Figure 4.2: Python model program flow

4.1.4. Classes and Logic

The model's classes defined in `classes.py` include:

1. `EXU`

   It is the main class of the model. It represents (part of) the EXU (Execution Unit) of the processor. It contains a NoC composed of routers and several other connected modules, such as functional units and reconfigurable areas. Its methods allow fault injection and reconfiguration when necessary.

   Attributes:

   - `reconfiguration_mode`: boolean that turns dynamic reconfiguration on or off during fault handling.
   - `total_luts` / `total_ffs`: total number of LUTs and FFs in the system that faults can affect.
   - `partitions`: list of all partitions in the system (instances of `Partition` or `Node`).
   - `noc`: instance of the `NOC` class, which models the mesh interconnection.
   - `reconfig_count`: number of reconfigurations performed so far.
   - `error_type`: reason for system failure. Possible values are:
     - `static_resource`: a fault has affected a non-reconfigurable system module.
     - `connection_down`: there are no paths between wrappers and their corresponding senders or receivers.
     - `invalid_area`: the selected reconfigurable area has either already been used for reconfiguring that module, or contains faulty resources that interfere with the wrapper's correct placement.

   Methods:

   - `get_defect()`: randomly generates a fault in a LUT or FF, considering the total distribution.
   - `inject_defect(faulty_resource, resource_type)`: injects the selected fault into the system and determines if it causes a critical failure or not.
   - `check_noc_status()`: verifies whether the system is still operational after a fault. If not, it tries to reconfigure the faulty module.
   - `reconfigure(wrapper, reconf_area)`: swaps a faulty wrapper with a spare partition and updates resources and NoC connections accordingly.
   - `check_mul_connections()` / `check_div_connections()`: verify that `mul_wrapper` or `div_wrapper` are connected to their corresponding sender and receiver.
   - `check_partition_luts()` / `check_partition_ffs()`: check that none of the used LUTs/FFs in a partition are damaged.

- `check_tested_nodes()`: ensures that a wrapper is not reconfigured to a node already used in a previous reconfiguration.

2. `Partition`

   Represents a functional unit, a router, or a spare area.

   Attributes:

   - `name`: name of the partition.
   - `luts_list`: list of LUTs occupied by the partition. Each element is a tuple `(i, status)` where:
     - `i`: unique identifier of the LUT.
     - `status`: can be `used`, `spare`, or `faulty`.
   - `ffs_list`: list of FFs occupied by the partition, with the same format as `luts_list`.
   - `type`: partition type. It can be:
     - `static`: non-reconfigurable region.
     - `dynamic_mul` / `dynamic_div`: dynamically reconfigurable area.
     - `router`: intermediate node in the NoC mesh.
     - `reconf_area`: spare block used for reconfiguration.

3. `Node`

   Inherits from `Partition`. Represents an endpoint node connected to the NoC, such as senders, receivers, multiplier and divider wrappers, and reconfigurable areas.

   Attributes:

   - `used_luts`: number of LUTs currently in use ($\leq$ total in `luts_list`).
   - `used_ffs`: number of FFs currently in use ($\leq$ total in `ffs_list`).
   - `router_connection`: router to which this node is connected.
   - `node_number`: identifier used to distinguish nodes and track usage during reconfiguration.
   - `tested_nodes`: set of node numbers already used in previous reconfigurations (to avoid reuse).

4. `NOC`

   This class models the interconnection network (NoC) as a two-dimensional mesh of routers. It connects routers and modules.

   Attributes:

   - `mesh`: a `networkx.Graph` object that represents the full mesh structure, including routers and endpoint nodes.

Methods:

- `build_mesh(mesh_height, mesh_width, partitions)`: creates the mesh topology and adds all connections using the `networkx` library.
- `display_mesh()`: prints the mesh structure and the neighbors of each node (used only for debugging).

4.1.5. Updates and Improvements

After the initial version of the simulation model was implemented, we began running small-scale simulations (100 to 500 iterations) to verify that the system behaved as expected. As with any evolving software, incremental changes were made to refine its behavior or to obtain more specific insights from the simulations. In addition to minor code improvements for readability and reuse, several significant updates were introduced to improve the realism and reliability of the model.

Initially, simulations only considered faults in LUTs. One of the first significant enhancements was extending fault injection to include FFs. To randomly select a faulty resource from the EXU, the actual proportion of LUTs to FFs in the design had to be considered.

Another necessary refinement was the addition of a mechanism for each wrapper to keep track of the nodes where it had already been configured, which was implemented in the code as a Python `set` named `tested_nodes` (explained in Subsection 4.1.4). It prevented reconfiguring a wrapper in a node where it had previously failed. Even if there were still enough healthy LUTs and FFs in the area, the place-and-route phase of hardware design tools always generates a fixed layout for a module in a given region. Therefore, if a wrapper had failed in a node, reconfiguring it again in the same place would result in the same layout and cause it to fail again. This behavior better reflects what would happen in a real FPGA. However, this approach is not perfect. Ideally, the exact resources assigned by the layout would be known, and the status of those specific LUTs and FFs could be checked directly. The limitation is that, while reconfiguration is avoided in nodes where the module has already been tested and failed, when a new node is tested, the algorithm only verifies that there are enough available resources overall, without checking the exact resources that the module's fixed layout would consume. This imperfection means that some faults could still be missed during simulation.

In the early versions of the simulation, if a fault occurred in the static region (sender/receiver), the system would fail immediately without attempting any reconfiguration. This behavior was changed to more accurately represent real hardware behavior. In practice, the system would detect a communication loss via the `alive` signal, but it wouldn't know that the fault came from the static region. It would still trigger all possible reconfigurations before concluding that the fault was not tolerable. With this improvement, every simulation now attempts at least eight reconfigurations (corresponding to the number of available reconfigurable regions). Previously, some simulations failed after zero reconfiguration attempts (scenarios that wouldn't realistically occur in the actual system).

To gain better insight into the cause of system failure, an enumeration was intro-

duced with the possible values `connection_down`, `static_region`, and `invalid_area`, along with an `error_type` attribute in the `EXU` class (see Subsection 4.1.4). Depending on the stage at which the system failed, this variable would take on one of the enumerated values and be saved to an output file for post-analysis.

Finally, a command-line flag (`-r`) was added to control whether dynamic reconfiguration is enabled during the simulation. This flag is passed as an input argument when running the program. If it is present, the `EXU` object is initialized with `reconfiguration_mode = True`; otherwise, it defaults to `False`. This feature allows the same simulation setup to be run with or without reconfiguration, enabling a direct comparison between both scenarios. As a result, the impact of dynamic reconfiguration on fault tolerance can be evaluated in a more consistent and controlled way.

Once the model reached a stable and reliable state, larger-scale simulations were performed, first with 1,000, then 5,000, and eventually up to 6,000 iterations, to enable a more thorough analysis of the system's performance and fault tolerance capabilities. The resulting plots showed no significant changes between the runs with 5,000 and 6,000 simulations, confirming that 6,000 iterations were reasonable to ensure result stability. The obtained data and graphs based on these simulations are discussed in Section 5.2.

4.2. NoC Integration and Dynamic Function eXchange

Another important part of the project focused on integrating the Network-on-Chip (NoC) (described in Subsection 2.3.3) into the RTL code of the RISC-V processor to enable synthesis, implementation, and bitstream generation, to program the FPGA, run simulations, and verify that the system behaved consistently before and after partial reconfiguration. Although synthesis and implementation with Dynamic Function eXchange (DFX) enabled were not ultimately carried out in this project version, the design was analyzed to determine whether it would be suitable for application in future work.

4.2.1. VeeRwolf Nexys

To adapt the existing RISC-V processor to integrate our custom NoC and support the new communication protocol between units, we used VeeRwolf, a platform based on FuseSoC for the VeeR family. VeeRwolf allows running RISC-V software in simulation or on FPGA boards, as well as adapting the RISC-V System-on-Chip (SoC) to specific needs or porting it to new target devices. The SoC is built around a portable core independent of the target technology and is wrapped with modules that adapt it to specific platforms or FPGA boards.

Out of the available cores and wrappers, we started by using VeeRwolf Nexys (Figure 4.3), a version of the VeeRwolf SoC specifically adapted for the Digilent Nexys A7 board. It uses the on-board 128MB DDR2 for RAM, connects GPIOs to the LEDs, supports SPI Flash boot, and uses the microUSB interface for UART

and JTAG communication.

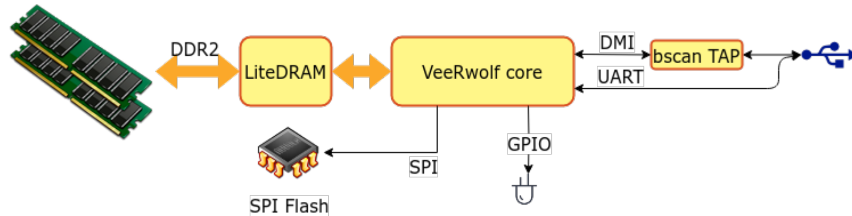

Figure 4.3: VeeRwolf Nexys A7 target (CHIPS Alliance)

To build the VeeRwolf SoC for the Nexys A7, we followed the steps described in Appendix C. The process generated a Vivado project with all the necessary files, including the top module (`veerwolf_nexys.v`) and the constraints file (`.xdc`), which maps the SoC's signals to the physical pins of the board. At this stage, the design included only the RISC-V processor.

To integrate the NoC and enable communication between the multiplier and divider units instantiated in the processor's EXU module, several modifications to the Vivado project were required. First, we added all the design source files developed by Davó Laviña and Carpio Cuenca for the NoC itself (mesh, routers, crossbars...), the communication modules (senders, receivers, wrappers...), and other necessary files such as interfaces, an edge detector, routing algorithm modules, and definition headers.

We also replaced the original RTL file of the EXU (`el2_exu.sv`) with a modified version that includes the NoC. In this version, the original multiplier and divider modules are no longer directly instantiated. Instead, the EXU instantiates the `mesh` module, along with the corresponding senders, receivers, and wrappers. It also instantiates the `edge_detector`, which triggers a flush signal that allows senders and receivers to unlock and restart data transmission and reception. Additionally, the EXU receives a new clock input, `clk_noc`, used by the mesh and all other connected modules.

This new clock input is propagated through the modules up to the top-level module (`veerwolf_nexys.v`). To ensure that this signal behaves as it did in the tests performed by Davó Laviña, where it operated 16 times faster than the main system clock, a Clocking Wizard IP was instantiated in the top module. This IP takes the 25 MHz system clock as input and generates a 400 MHz output clock using an MMCM. The generated output clock drives the NoC and all connected components.

With these modifications, we successfully synthesized and implemented the design of the RISC-V processor with the NoC integrated, but we encountered timing failure due to setup violation, discussed in Section 5.1.

4.2.2. VeeRwolf Sim

As a next step, we wanted to run simulations on the whole design and validate the processor's behavior modified with the NoC. Although Vivado includes its own

RTL simulator, the previously mentioned VeeRwolf platform also provides a specific simulation target: VeeRwolf Sim.

VeeRwolf Sim (Figure 4.4) wraps the VeeRwolf core in a testbench to be used with Verilator to enable full-system simulations that execute real programs on the VeeR processor. It also supports connecting a debugger through OpenOCD and JTAG VPI.

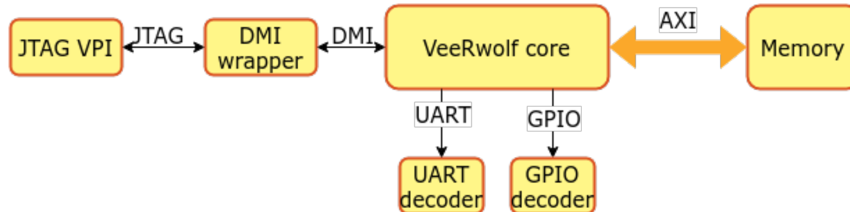

Figure 4.4: VeeRwolf Simulation target (CHIPS Alliance)

It is ideal for verifying the new communication architecture through the NoC, without building complex testbenches from scratch in Vivado. Another key advantage over Vivado's simulator is its speed and flexibility, as it allows for quick iteration over design changes and simplifies debugging.

The steps we followed to build the simulable VeeRwolf SoC can be found in Appendix C. Similarly to VeeRwolf Nexys, the RISC-V processor built does not include the NoC. By default, the simulator loads `hello.vh`, which prints the message "VeeR+FuseSoC rocks", shown in Figure 4.5.

```
================================================================================
INFO: Running
INFO: Running simulation
Loading RAM contents from hello.vh
Releasing reset
VeeR+FuseSoC rocks

Finito
- src/veerwolf_0.7.5/rtl/veerwolf_syscon.v:154: Verilog $finish
(veerwolf_env) isabelroman@isabelroman-VirtualBox:~/Desktop/tfg/veerwolf$
```

Figure 4.5: Simulation output with the default `hello.vh` program loaded.

However, VeeRwolf Sim allows preloading an application into memory using the `--ram_init_file` parameter. Since our processor modifications would directly affect the behavior of the multiplier and divider, we wrote two very simple test programs to verify that both units worked correctly before and after integrating the NoC:

- `mul_test.S`: computes the scalar product of two integer vectors using `mul` instructions, stores it in memory, and prints the result to the console. In Figure 4.6, the output result is the character "y", which corresponds to the ASCII code for 121, the actual result of the scalar multiplication.

- `div_test.S`: divides the elements of two vectors one by one using `div` instructions, adds the results, saves the total in memory, and prints a value to the console. In this case, in Figure 4.7, the output is the character "/", the ASCII code for the calculation result, 47.

```
================================================================================
INFO: Running
INFO: Running simulation
Loading RAM contents from /home/isabelroman/Desktop/tfg/veerwolf/fusesoc_libraries/veerwolf/sw/mul_test.vh
Releasing reset
y
Finito
- src/veerwolf_0.7.5/rtl/veerwolf_syscon.v:154: Verilog $finish
(veerwolf_env) isabelroman@isabelroman-VirtualBox:~/Desktop/tfg/veerwolf$
```

Figure 4.6: Simulation output with `mul_test.vh` loaded.

```
================================================================================
INFO: Running
INFO: Running simulation
Loading RAM contents from /home/isabelroman/Desktop/tfg/veerwolf/fusesoc_libraries/veerwolf/sw/div_test.vh
Releasing reset
y
Finito
- src/veerwolf_0.7.5/rtl/veerwolf_syscon.v:154: Verilog $finish
(veerwolf_env) isabelroman@isabelroman-VirtualBox:~/Desktop/tfg/veerwolf$
```

Figure 4.7: Simulation output with `div_test.vh` loaded.

These source files had to be converted into Verilog hex files (`.vh`) to preload them into memory during simulation. All files are available in the GitHub repository under the `veerwolf_dir/tests` directory.

Once the SoC was built and the test programs successfully executed, it was time to integrate the NoC. We had to modify several files inside the `fusesoc_libraries/veerwolf` folder, which define the SoC structure and module connections used by VeeRwolf to build the simulator.

The main entry point for the build process is the `veerwolf.core` file, which lists all relevant source files and their dependencies. By examining its contents, we identified `veer_el2_wrapper.sv` as the top-level wrapper and a dependency on the VeeR_EL2 core, described in the `veer_el2.core` file. This information helped us locate the wrapper and the processor core modules that needed modification.

The original `veer_el2.core` file defines the basic structure of the VeeR EL2 RISC-V core, listing the essential SystemVerilog source files for the processor's key components, including the instruction fetch unit (IFU), decoder, execution units (ALU, multiplier, divider), and load-store unit (LSU).

We extended this file by adding a set of source and header files required to integrate the NoC. These files were also physically added to the corresponding directories within the FuseSoC library structure, ensuring that all referenced paths in the core description remain valid. These additions mirror those included in the Vivado project, as detailed in Subsection 4.2.1. By including them in the core description, the build process understands the new dependencies and incorporates the NoC seamlessly into the processor design.

The files `cores.zip` and `rtl.zip` available in the `veerwolf_dir` directory of the project GitHub repository reflect the result of these modifications and can be consulted for further details.

After updating and adding the necessary files, we reran the FuseSoC command to build the simulator, this time with the NoC fully integrated into the processor. The results obtained from the Verilator simulation are detailed in Section 5.1.2.

4.2.3. Vivado DFX Poject

Dynamic Function eXchange (DXF) (explained in Subsection 2.2.2) is a technology that enables an incremental and partitioned hardware configuration of a programmable device like an FPGA. DFX enables partitioned designs, allowing one part of the system to be changed (reconfigured) while another remains running.

As previously mentioned, the components we aim to reconfigure in our design are the wrapped functional units: the multiplier and the divider. In this case, the reconfiguration consists of relocating them to different endpoint nodes of the NoC, rather than changing the functionality they implement.

Communication between the functional units and the rest of the system is done through the NoC. Since the NoC and the rest of the processor form part of the static logic, this connectivity remains unaffected during reconfiguration, preserving system coherence.

A dedicated Vivado project was created focusing exclusively on the execution unit (EXU), the submodule containing all the relevant logic for reconfiguration. This project uses a simplified version of the EXU as the top-level module, only instantiating the mesh of the NoC and the necessary modules for the multiplier and divider units to work. The inputs and outputs of the EXU were also reduced to the minimal set required to operate these components. This simplification ensures that dynamic reconfiguration can be applied without interference from unrelated logic.

After enabling the "Dynamic Function eXchange" option in a Vivado project, the first step is to define the reconfigurable partitions (RPs). These correspond to the `i_mul_wrapper` and `i_div_wrapper` instantiations in our design. One of the fundamental requirements of a partially reconfigurable design is consistency across all reconfigurable modules (RMs) targeting the same RP. As one module is swapped for another, the connections between the static design and the RM must be logically and physically identical. Since the static side of the interface becomes fixed after the first configuration, any changes in later modules cannot be adjusted for by the Vivado tools. It is therefore critical to ensure that the interface between static logic and RMs remains the same.

Fortunately, the wrapper modules designed by David already satisfy this requirement. Despite internally instantiating different functional units, the interfaces of `el2_exu_mul_wrapper` and `el2_exu_div_wrapper` are identical. This design choice makes them ideal candidates for module swapping in a DFX flow.

However, the wrapper modules are instantiated with non-static parameters (`.pt`), whose values are unknown at synthesis time. This aspect violates RP definition constraints and must be corrected. Consequently, the first required change is to eliminate these dynamic parameters or refactor the EXU module so that parameters are evaluated locally prior to partition creation.

Another critical aspect to address is the management of spare areas for reconfiguration and how to implement them. If we consider them reconfigurable partitions, the mesh would be connected to ten RPs in our design. Out of these ten RPs, only two will be loaded in any given configuration, one with the multiplier and one with the divider. The remaining RPs must still contain some valid logic design. Vivado's solution for this scenario is to implement a pseudo black box called a greybox. A

greybox is an RP without internal logic, but with LUT1 buffers placed on all its inputs and outputs. To implement the spare reconfigurable areas, we would need eight greyboxes, which are essentially "empty" modules that expose the same interface as the multiplier and divider wrappers. These modules should be instantiated in the EXU, using the remaining available `node_port` interfaces, as shown in Figure 4.8.

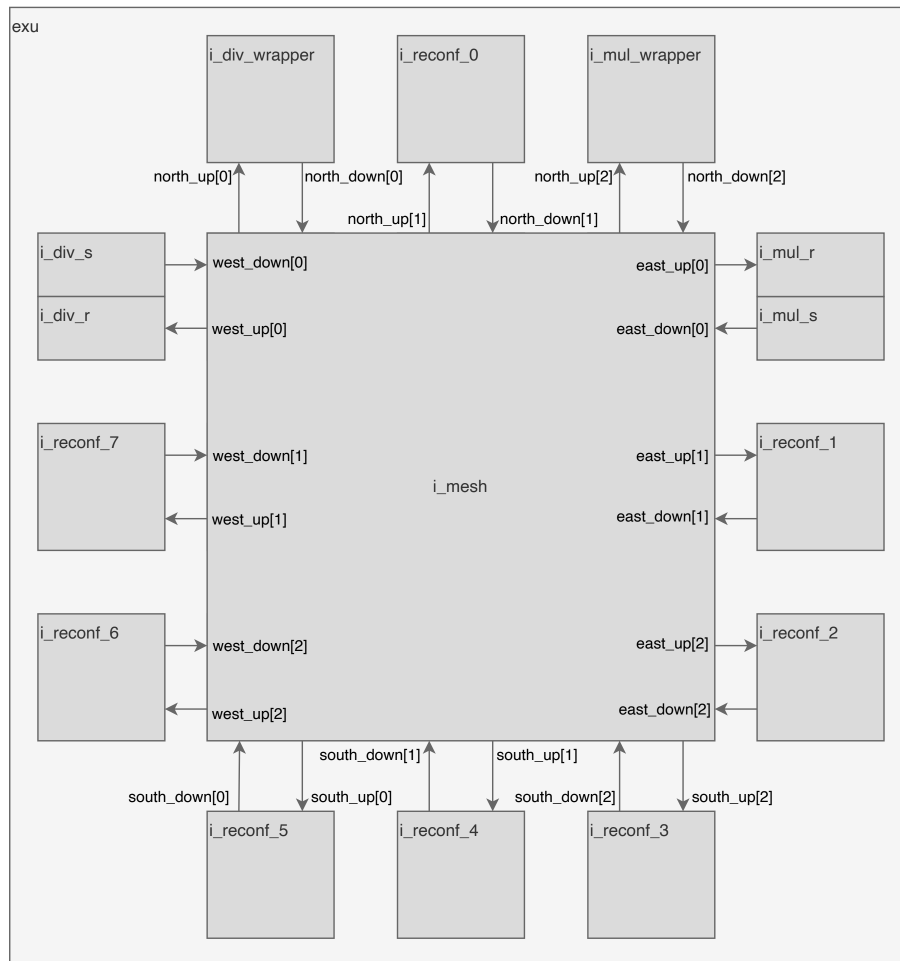

Figure 4.8: Reconfigurable areas implementation

One final but important challenge that requires changes to the design is ensuring that, when a wrapper is dynamically reconfigured and loaded into a new RP, the rest of the system, specifically the sender and receiver modules, should be informed of its new location in the NoC. This requirement is not a modification intended to support DFX itself, but rather a functional requirement to ensure that communication through the NoC continues to operate correctly, with packets routed to the appropriate destinations.

One idea is for the wrappers to periodically send their current position to their corresponding sender and receiver. Since the sender and receiver modules have fixed positions in the NoC, even after reconfiguration, the wrappers will always know where to send and receive the data through the NoC.

Chapter 5

Results

Here, the results of the project are presented, covering the hardware implementation and the simulation model, which were developed in parallel and often complement each other. Information from one process was used to guide and adjust the other. For example, resource usage data from the hardware helped define the model inputs, while simulation results provided insights to evaluate the RTL design, such as the number of reconfigurable areas and the impact of module sizes on fault tolerance.

5.1. Implementation of RISC-V with NoC

Although simulating real hardware faults on the FPGA was not within the project's objective, successfully synthesizing and implementing the RISC-V processor with the integrated NoC was crucial in evaluating its behavior and adaptability to DFX techniques. The analysis includes implementation reports generated by Vivado, a brief evaluation of simulation attempts using Verilator, and the study of the architectural modifications required to support dynamic partial reconfiguration.

5.1.1. Vivado Reports

The Vivado synthesis and implementation processes for the RISC-V with the NoC integrated were completed successfully. However, the design failed to meet timing constraints during implementation due to setup violations (detailed in Table 5.1).

Setup		**Hold**	
Worst Negative Slack (WNS)	-41.564 ns	Worst Hold Slack (WHS)	0.052 ns
Total Negative Slack (TNS)	-76,356.140 ns	Total Hold Slack (THS)	0.000 ns
Number of Failing Endpoints	6,266	Number of Failing Endpoints	0
Total Endpoints	39,067	Total Endpoints	39,067

Table 5.1: Design Timing Summary

Setup violations occur when a signal fails to reach the input of a flip-flop early enough before the active clock edge. They typically arise when the data path is too slow relative to the clock period, which is common in designs with high clock frequencies or long combinational delays. In contrast, hold violations occur when a signal changes too soon after the active clock edge, not allowing the previous value to be held long enough at the flip-flop's input.

We examined the Timing Report to investigate the origin of the setup violations encountered. This report allows a detailed analysis of the most critical paths in the design, identifying their start and end points, the logical elements involved, and the clock domains in use. Inspecting the ten most critical paths, we noticed that all go from a module driven by `clk_core` to another module driven by `clk_out1_clk_wiz_0`, which runs 16 times faster. These observations confirm that timing issues were caused by the data transfer between two different clock domains. The significant frequency difference imposes much tighter timing constraints, which explains the large negative slack. Consequently, a more robust synchronization strategy is needed between these two domains.

Knowing that poor synchronization between clock domains due to the significant frequency difference was probably the cause of timing failures, we nevertheless tried to fix the setup violations by lowering the `clk_core` frequency.

First, we lowered this frequency to 20 MHz as a test, but the timing issues persisted. Later, we calculated the theoretical maximum operating frequency based on the worst negative slack, which resulted in approximately 12.26 MHz:

If the original clock was 25 MHz, the clock period T is:

$$T = \frac{1}{25\,\text{MHz}} = 40\,\text{ns}$$

Given:

$$\text{WNS} = -41.564\,\text{ns}$$

So:

$$T_{\text{min}} = 40\,\text{ns} + 41.564\,\text{ns} = 81.564\,\text{ns}$$

Therefore,

$$f_{\text{max}} = \frac{1}{81.564 \times 10^{-9}} \approx 12.26\,\text{MHz}$$

After lowering `clk_noc` to the calculated maximum operating frequency, we continued to experience setup violations, and hold violations also emerged, confirming that the clock domains, synchronization, or logic structure may require a redesign.

In addition to timing analysis, we examined the Utilization Report, which details the usage of key FPGA resources such as LUTs and Flip-Flops (FFs). This data, shown in Table 5.2, allows us to understand the overall resource consumption of the system.

Site Type	**Used**	**Available**	**Util%**
LUTs	27029	63400	42.63
Flip Flops	13822	126800	10.90

Table 5.2: Slice Logic from Resource Utilization Report

The utilization report reveals that approximately 42.6% of the LUTs and only 10.9% of the Flip-Flops are currently used in the design. These figures indicate a large portion of the FPGA resources remain free, which is a favorable scenario for implementing Dynamic Function eXchange (DFX). Since DFX requires reserved or spare cells to host partial reconfigurable modules and enable dynamic swapping of hardware blocks, having over 57% of LUTs and almost 90% of Flip-Flops available means there is plenty of room to assign these spare resources.

Additionally, by using the `-hierarchical` flag, we were able to examine resource usage in greater detail, identifying how many LUTs and FFs were consumed by each module in the design hierarchy. In particular, from this report we extracted the resources usage corresponding to the EXU, consumed by the wrappers, senders, receivers, the NoC mesh, and the individual routers that form this mesh (Table 5.3).

Module	**Total LUTs**	**FFs**
el2_exu	6654	1700
el2_exu_div_receiver	18	35
el2_exu_div_sender	26	4
el2_exu_div_wrapper	1434	220
el2_exu_mul_receiver	1	35
el2_exu_mul_sender	69	5
el2_exu_mul_wrapper	755	186
mesh_FT	2826	944
router_FT (00)	425	130
router_FT (01)	249	98
router_FT (02)	387	134
router_FT (10)	433	104
router_FT (11)	364	134
router_FT (12)	332	104
router_FT (20)	85	68
router_FT (21)	375	104
router_FT (22)	189	68

Table 5.3: Utilization by Hierarchy from Resource Utilization Report

This information was used to generate the XML file that describes the structure of the NoC and its constituent modules, which serves as the input for the Python simulation model. It was also valuable for estimating the probability of faults affecting critical components (the senders and receivers of the multiplier and divisor, as well as the routers connected to them, router_FT(00) and router_FT(02)). These components are considered critical because any fault affecting them will inevitably lead to system failure, as there is no reconfiguration capable of maintaining communication with the wrappers. Collectively, they make up 5.91% of the total resources represented in the Python model, corresponding to the probability that a fault will occur in these areas.

Statistically, the expected number of faults before encountering one that affects a critical resource is the inverse of that probability, approximately 16.9 faults on

average, reasonably close to the MTTF value of 14.3 obtained from simulations, calculated in the next section (Subsection 5.2.1). The difference between these two values can be explained by the fact that the expected number of faults only considers the likelihood of a fault occurring in a critical resource and does not account for the accumulation of previous faults in the system, whereas the MTTF does.

5.1.2. Verilator Simulator

Hoping to run full-system simulations of the processor with the NoC integrated, we used the VeeRwolf Sim target, making the necessary modifications to the files required by the simulation tool. As a result, we obtained a version of the VeeRwolf core with the NoC successfully integrated and ready to be simulated with Verilator.

Initially, the simulation was tested using the version of Verilator that had been automatically installed (v5.020). However, this version produced several compilation errors, for instance, due to the lack of support for streaming operations (`<<`) on multidimensional arrays.

To address these issues, we explored more recent versions of Verilator, including v5.024 and v5.028. For each version, we evaluated the necessary code modifications and verified compatibility, allowing us to resolve some of the initial errors. Unfortunately, the error shown in Figure 5.1 persisted across all of them. The issue stems from Verilator's inability to handle interface vectors used in the `mesh_FT` module.

We tried to simplify the code to make it compatible with Verilator, but these efforts were ultimately unsuccessful. Finally, we tested the most recent available version at the time, v5.036, but the error could still not be resolved.

Future releases will hopefully address this limitation, finally enabling full simulation of the system with the integrated NoC, considering that all the necessary files are complete and ready to be used. The built simulable VeeRwolf core files are available in the project's GitHub repository.

```
%Error: Internal Error: src/chipsalliance.org_cores_VeeR_EL2_1.4/design/exu/noc/mesh_FT.sv:239:70: ..
/V3Param.cpp:1347: Could not find array index in unlinked text: '' for node: CELLARRAYREF 0x644d2ad5e
510 {cc239cs} @dt=0@  ports_up__Viftop
                                                                                                   : ..
. note: In instance 'veerwolf_core_tb.veerwolf.rvtop.veer.exu.i_mesh'
  239 |                                .down(nodes_h[y].nodes_w[x].ports_up[EAST]),
      |                                                                      ^
                        ... See the manual at https://verilator.org/verilator_doc.html?v=5.036 for mo
re assistance.
make: *** [Makefile:15: Vveerwolf_core_tb.mk] Error 1

ERROR: Failed to build ::veerwolf:0.7.5 : '['make']' exited with an error: 2
```

Figure 5.1: Verilator error due to unsupported interface vectors in `mesh_FT`

5.2. Analysis of the Python Simulation

To gain a deeper understanding of the model's behavior (described in Section 4.1) and to assess the effectiveness of dynamic reconfiguration, several statistical and graphical analyses were conducted based on a simulation of 6,000 runs. This analysis focuses on the total number of faults injected into each system until failure, the frequency and distribution of reconfigurations performed, and the correlation between the reconfiguration activity and the system's overall reliability.

5.2.1. Injected Faults

Data stored in `fault_results.csv`, indicating the number of injected faults in each simulation, were used to generate plots and metrics that illustrate the system's fault tolerance behavior and performance.

In this first graph (Figure 5.2), the line represents the cumulative percentage of failed simulations as a function of the total number of injected faults. In other words, it shows the percentage of simulations that failed after reaching that number of faults. Its shape reveals several key insights.

The first portion of the curve rises steeply and shows a concave downward trend, indicating that most systems fail after tolerating somewhere between 0 and 25 faults, over 80% of them, in fact. Approximately at 30 total faults, the growth slows down noticeably, and by the time it reaches 45, it's nearly flat. Such behavior suggests that almost all simulations have already failed by that point, and only a few manage to tolerate such a high number of faults. As expected in a cumulative distribution, the curve's tail asymptotically approaches 100%, confirming that all systems eventually fail as the number of faults increases. In this particular run of 6,000 iterations, the highest total faults observed was 88.

Additional statistical indicators can also be extracted from this graph. The median number of injected faults, corresponding to the point where 50% of the simulations have failed, is 12. The first quartile (Q1), marking the threshold below which 25% of simulations failed, is 5 faults, while the third quartile (Q3), where 75% have failed, is 20 faults. These values give an interquartile range (IQR) of 15 faults, meaning that half of the simulations failed after between 5 and 20 faults, offering a good estimate of the system's typical fault tolerance window.

To better understand the impact of dynamic reconfiguration, the same cumulative analysis was carried out using an identical number of iterations (6,000) but with the reconfiguration mechanism disabled (flag -r off). The resulting curve (Figure 5.3) shows a steeper initial rise and flattens earlier. In this case, more than 80% of the systems failed after only 12 injected faults, and the curve reaches saturation just past 16 faults. The highest number of tolerated faults was 61, compared to the 88 observed with reconfiguration enabled.

The difference is also reflected in the statistical data. The median without reconfiguration drops to just 5 total faults, while Q1 and Q3 shift to 2 and 9, respectively. The resulting IQR is just 7 faults, less than half of the 15-fault range in the reconfigurable setup.

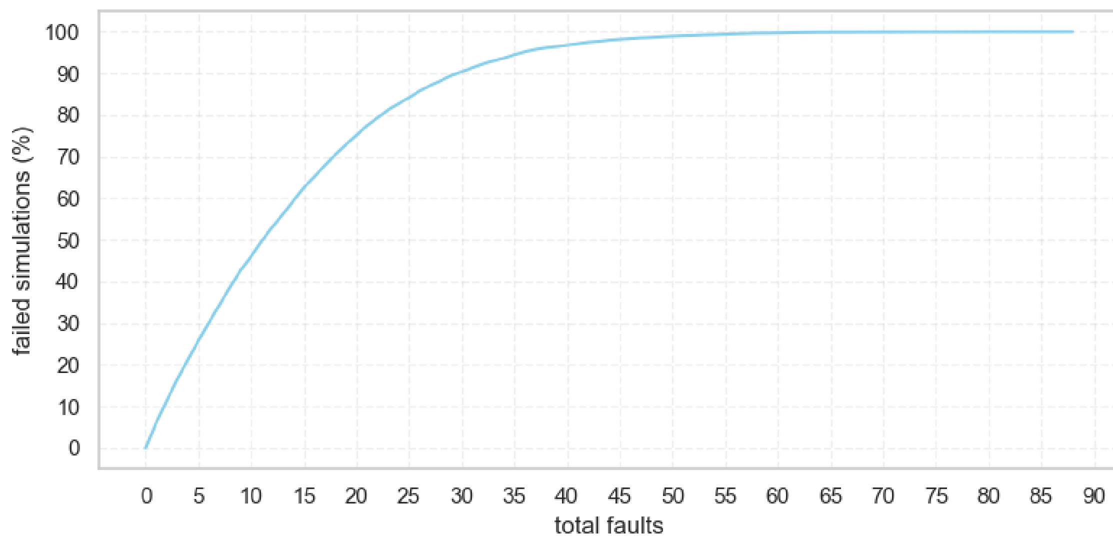

Figure 5.2: Cumulative Frequency Curve: Failed Simulations (%) vs Total Faults (with dynamic reconfiguration)

The improvement is not overwhelming, but the results still point to a noticeable gain in fault tolerance when dynamic reconfiguration is enabled.

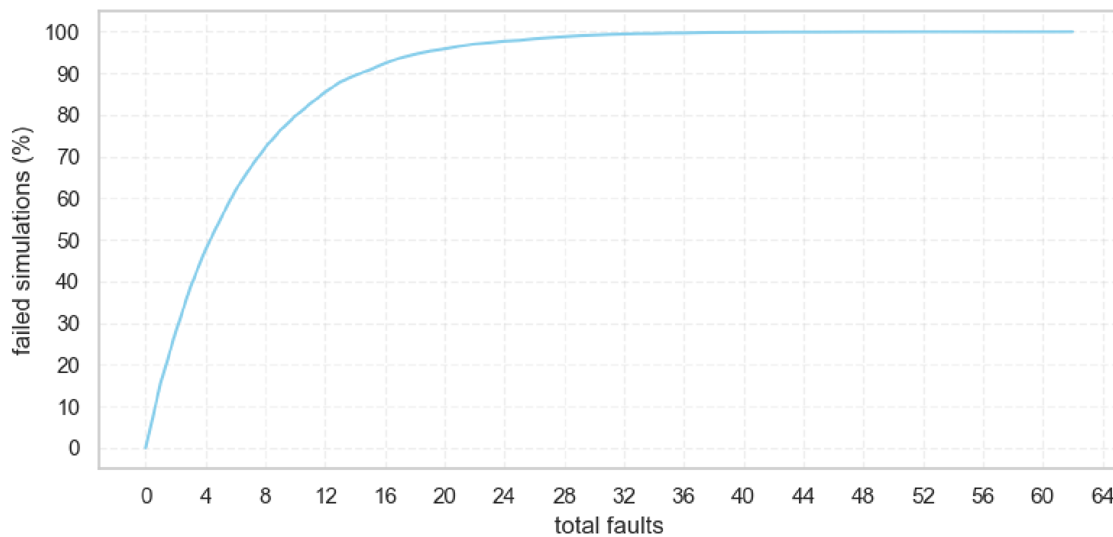

Figure 5.3: Cumulative Frequency Curve: Failed Simulations (%) vs Total Faults (without dynamic reconfiguration)

A jitter plot was used to visualize the distribution of total faults across a large number of simulations (Figure 5.4). Since only one variable is represented, some vertical noise was added so that overlapping points become visible and the density of observations is better illustrated.

Each dot represents the result of one simulation, with its horizontal position indicating the total number of faults injected before failure. The vertical position has no meaning; it only helps to separate the points visually. The color of the dots

varies with the number of faults, using a gradient colormap for better distinction.

Although the cumulative frequency curve already provides valuable insight into the system's fault-tolerance capabilities, this jitter plot makes it easy to see how the concentration of simulations becomes smaller as the total fault count increases. While this plot is not essential for the core analysis, it is a simple and accessible tool to obtain a first impression of the results. It is also particularly useful for spotting outliers that deviate significantly from the general trend. In the plot, one can clearly observe the simulation that reached 88 faults.

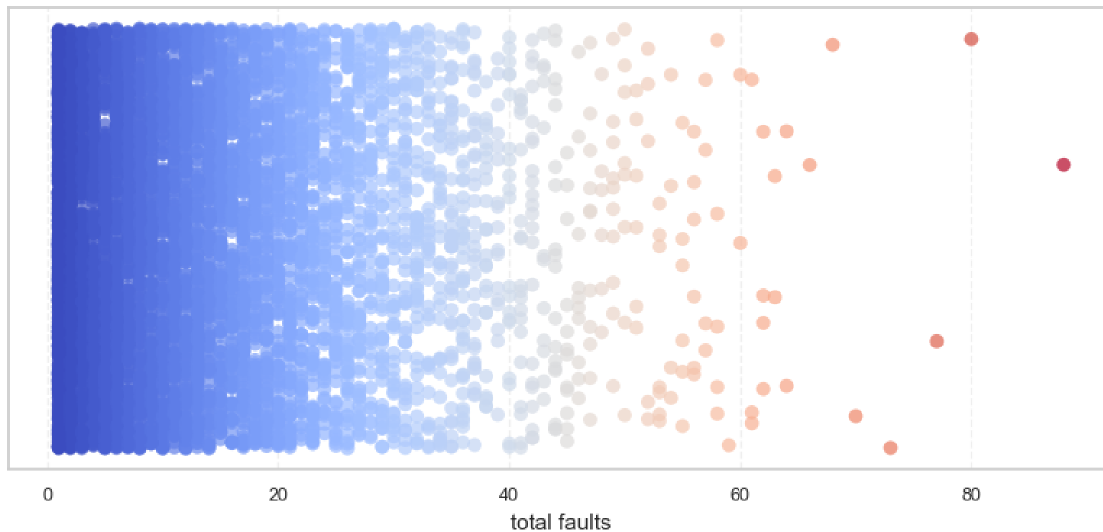

Figure 5.4: Jitter plot: Total Faults per Simulation (with dynamic reconfiguration)

Some of the traditional metrics described in Subsection 2.1.3 were not directly applicable or could not be measured with precision in the context of this simulation model, primarily because the model does not include explicit timing or duration for operation and repairs of non-tolerable faults. Nevertheless, if we consider one iteration of the fault injection loop as one unit of time (i.e., one injected fault equals one time unit), the simulation can be used to estimate adapted versions of the following metrics:

- Reliability $R(n)$: It can be approximated as the probability that a system remains functional after injecting n faults. Given a total of N simulations, and $F(n)$ as the number of simulations that failed at n faults:

$$R(n) \approx 1 - \frac{F(n)}{N}$$

  For example, the system's reliability at 12 injected faults, R(12), is approximately 0.5 when reconfiguration is enabled, and only 0.14 without it. This metric is implicitly visualized in the cumulative failure plots discussed earlier.

- Mean Time To Failure ($MTTF$): Assuming one fault is injected per unit time, MTTF corresponds to the average number of injected faults until system failure. This value can be directly computed from the entries in `fault_results.csv`

by calculating the mean. We observed MTTF $\approx$ 14.3 with reconfiguration enabled, and MTTF $\approx$ 6.7 without reconfiguration.

- Fault Coverage (FC): Defined as the ratio of successfully tolerated faults to the total number of injected faults. Since each simulation ends with an untolerated fault, the number of tolerated faults is the sum of all injected faults across simulations minus the number of simulations. Thus:

$$\mathrm{FC} = \frac{\sum_{i=1}^{N}(f_i - 1)}{\sum_{i=1}^{N} f_i} = 1 - \frac{N}{\sum_{i=1}^{N} f_i}$$

  where f_i is the number of injected faults in simulation i, and N is the total number of simulations.

  When dynamic reconfiguration was enabled, 79,816 out of 85,816 injected faults were tolerated, resulting in a fault coverage of FC = 0.93. Without reconfiguration, fault coverage dropped to FC = 0.85. This metric provides a general overview of the system's ability to handle faults but must be interpreted carefully. It does not follow the classical definition of fault coverage, which considers independent faults in isolation. In our model, faults are injected sequentially, and whether one is tolerated may depend on earlier ones within the same simulation.

  Moreover, the final FC value aggregates results from all 6,000 simulations. As such, it does not measure the probability of tolerating a single fault in a clean system, but rather summarizes how many faults were tolerated on average before failure, across many runs.

5.2.2. Reconfiguration Performance

To understand how often reconfiguration was needed to tolerate injected faults, a line plot was generated showing the number of simulations that ended with each specific reconfiguration count (Figure 5.5). The x-axis represents the number of reconfigurations performed in each simulation, and the y-axis indicates how many runs ended with that count.

The shape of the plot is strongly influenced by the design constraint of having only eight reconfigurable regions available. Because of this, a large portion of simulations reach exactly 8 reconfigurations (one for each available area) before failing. Smaller peaks around 9 or 11 reconfigurations likely correspond to cases where a few faults were tolerated through configuration before facing a critical fault.

Beyond these peaks, the number of simulations drops significantly. In most of the remaining cases, the count stays between 1 and 100, out of a total of 6,000. However, the presence of points across almost all reconfiguration counts up to the maximum, 55, suggests a fairly uniform spread and no significant outliers.

This behavior is consistent with the data in Table 6.1, which indicates that more than 70% of failures are ultimately due to a fault in `router_00` or `router_02`. Since the sender and receiver modules are statically connected to these two routers, any

fault affecting them breaks all viable routing paths with their corresponding functional unit wrappers. In such scenarios, the system may exhaust all 8 reconfigurable regions without ever finding a successful recovery strategy. Such a situation explains the concentration of terminations at the 8-reconfiguration mark.

Although increasing the number of reconfigurable areas might seem like a potential improvement, it would have little impact in practice. Table 6.2 shows how only 0.02% of failures were caused by running out of reconfigurable options (`invalid_area`). These results indicate that the current number of areas is already sufficient in most practical scenarios, and that most failures are due to structural limitations rather than a lack of available regions for reconfiguration.

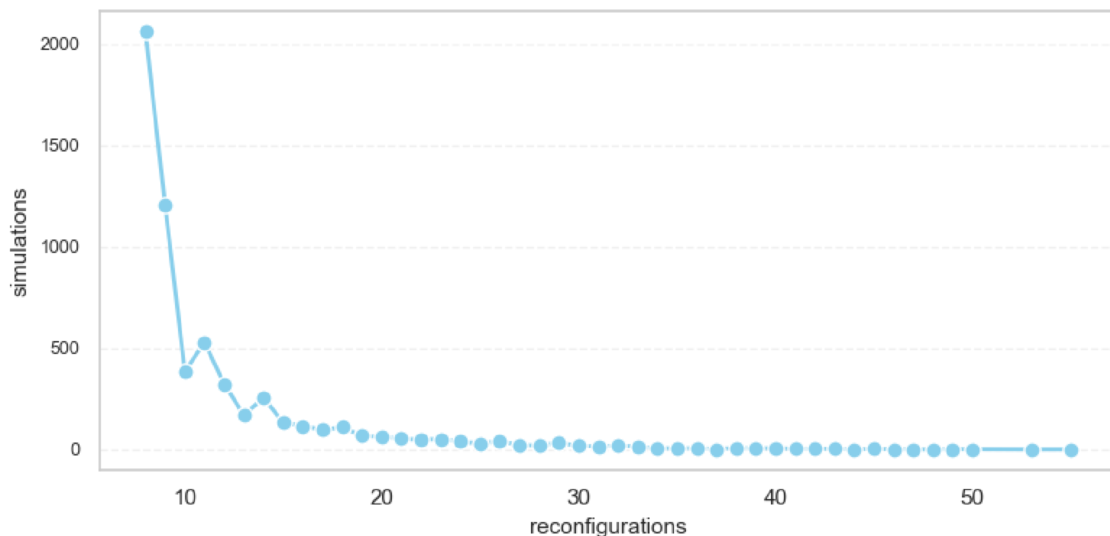

Figure 5.5: Line plot: Reconfigurations Performed vs Number of Simulations

5.2.3. Dynamic Reconfiguration on Fault Tolerance

This plot is likely the most important in the analysis, as it directly relates the number of reconfigurations to the number of tolerated faults. It essentially allows us to evaluate and demonstrate whether dynamic reconfiguration effectively improves fault tolerance.

The box plot shows the relation between the number of reconfigurations and total faults in each simulation (Figure 5.6). In general, the median number of faults increases as the number of reconfigurations grows, which aligns with the idea that dynamic reconfiguration helps the system recover from faults. However, this trend is not linear. The growth slows down between 25 and 35 reconfigurations, and beyond 35, no clear improvement is observed. This irregularity is likely due to the small number of simulations that reach such high reconfiguration counts, making the data in that range less reliable. One possible explanation for this stagnation is the structural limitation in the design discussed earlier, especially failures affecting routers that are essential for data transmission.

The early part of the plot (low reconfiguration counts) also shows a high number of outliers, which are simulations that manage to tolerate a surprisingly large number

of faults despite requiring very few reconfigurations. Such behavior likely occurs when the injected faults do not affect critical components, allowing the system to continue operating without reconfiguring.

Overall, the plot suggests that dynamic reconfiguration does improve fault tolerance, but only up to a certain point. Its effectiveness is mainly limited by failures in parts of the system that are not reconfigurable. Since the sender and receiver modules are statically connected to specific routers, reconfiguration can no longer restore functionality once those routers or the modules themselves are affected. Even if new routing paths are explored, they are blocked by these faulty static areas, and the system ends up failing regardless of how many reconfiguration attempts are made.

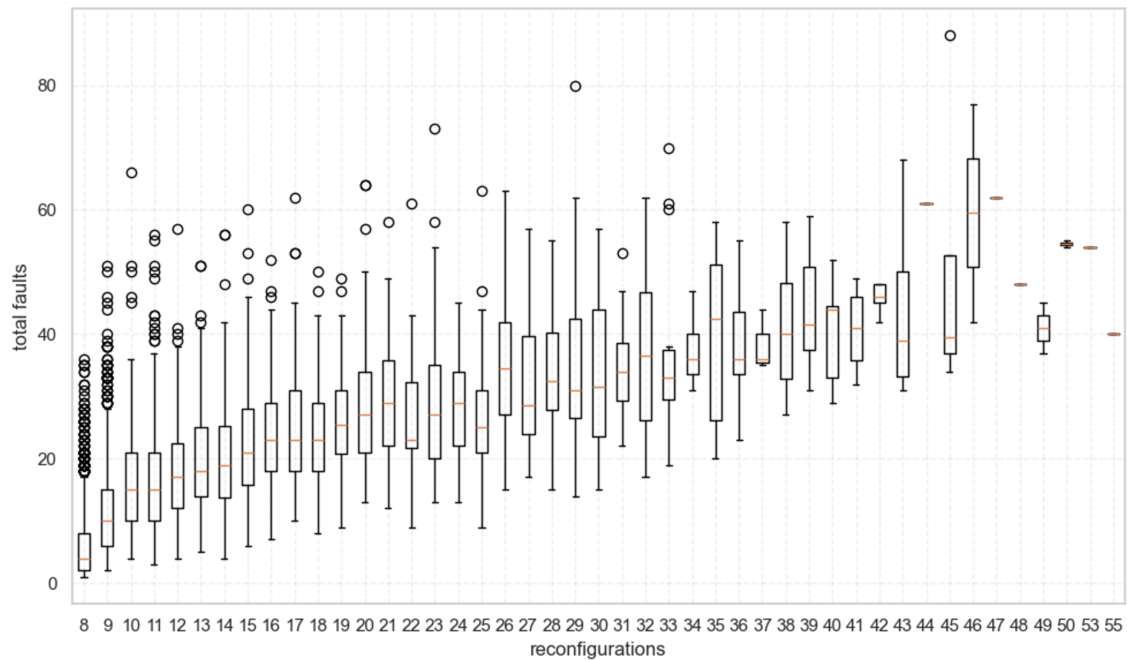

Figure 5.6: Box plot: Tolerated Faults vs Number of Reconfigurations

Chapter 6

Conclusions and Future Work

6.1. NoC Limitations and Reinforcements

Thanks to the analysis of the data and plots obtained from the simulation model, we conclude that dynamic reconfiguration indeed results in an improvement in the system's fault tolerance, although a modest one. This improvement is mainly limited by the architecture of the NoC itself. As shown in Table 6.1, using dynamic reconfiguration, more than 80% of the simulation failures were caused by a faulty router, among which `router_00` and `router_02` are the most frequent ones.

Faulty Partition	**Failed Simulations (%)**
`router_00`	36.68%
`router_02`	34.98%
`router_01`	4.32%
`mul_s`	4.22%
`div_wrapper`	4.03%
`div_r`	3.65%
`router_10`	3.17%
`mul_r`	2.55%
`router_12`	1.97%
`div_s`	1.88%
`mul_wrapper`	1.60%
`router_11`	0.35%
`router_22`	0.27%
`router_21`	0.27%
`router_20`	0.07%

Table 6.1: Faulty partitions that caused system failure

This behavior is also reflected in the error breakdown in Table 6.2, which shows that 87.68% of failures occurred because the system could not find a valid route between the sender or receiver and the wrapper, typically due to faults in NoC routers.

Error Type	Failed Simulations (%)
no_route	87.68%
static_resource	12.30%
invalid_area	0.02%

Table 6.2: Error type that caused system failure

We can therefore confirm that the main vulnerability in our NoC design lies in the routers connected to the static modules (senders and receivers), which in our case are router_00 and router_02.

As observed in Table 5.3, router_00 and router_02 together represent only about 5% of the total resources in the architecture modeled by the Python simulation. However, they account for more than 70% of the simulation failure causes.

The reason for these results, as explained earlier, is that the system can generally recover from a wrapper or any other router's failure by rerouting traffic or dynamically reconfiguring. However, when either router_00 or router_02 fails, the NoC cannot find an alternative path, breaking communication between senders and receivers and the wrappers, ultimately always leading to system failure. Other critical components (senders and receivers) are less likely to cause system failure as they represent less than 1% of the modeled architecture.

These findings suggest that while dynamic reconfiguration improves system reliability, it may not be sufficient on its own to ensure high fault tolerance. An additional method that reinforces the NoC is therefore strongly recommended, and possibly essential.

A promising approach would be to apply TMR (Triple Modular Redundancy) (explained in Subsection 2.1.4). By physically triplicating the critical routers (router_00 and router_02), we can significantly reduce the risk that a single fault compromises the communication path. In the presence of a transient or permanent fault in one of the triplicated routers, the remaining two could still provide correct functionality, allowing the NoC to maintain connectivity between the static and reconfigurable modules. This kind of redundancy is especially useful in systems where specific components represent a single point of failure, as is the case here. While implementing TMR increases area and resource usage, the potential gain in fault tolerance and system robustness may justify the overhead, especially in safety-critical applications or systems deployed in harsh environments with elevated radiation exposure. Figure 6.1 illustrates the proposed NoC design incorporating DPR and TMR techniques.

6.2. Design Adaptations for DFX

The necessary changes in the EXU design with the integrated NoC, analyzed in Subsection 4.2.3, can be summarized as follows:

- Eliminate the dynamic parameters of the wrapper instances or refactor the EXU so that these parameters are evaluated locally prior to partition creation.

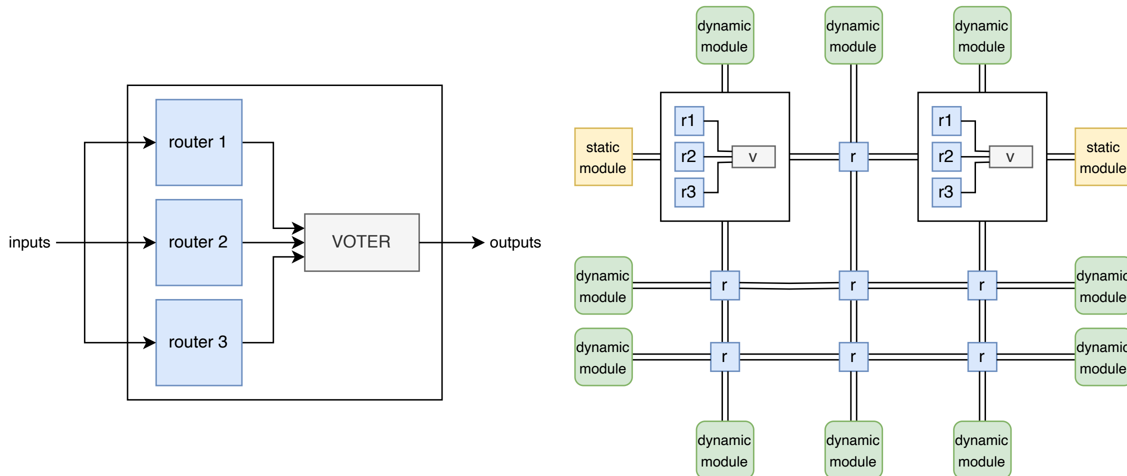

Figure 6.1: NoC design with DPR and TMR

- Create and instantiate greyboxes in the EXU to serve as spare reconfigurable areas.

- Design and implement a dynamic discovery mechanism to ensure end-to-end communication consistency between the dynamically reconfigured wrappers and the fixed-position senders and receivers. For example, by periodically sending messages with the wrapper's current position to the sender and receiver.

Although these are the main challenges identified so far, additional issues will likely arise during the synthesis and implementation steps with DFX.

A useful final extension to the design would be enabling the simulation of hardware faults on the FPGA without physically damaging the circuit. Such a simulation could be achieved by introducing a faulty RM that mimics the behavior of a module with defective hardware. Inspired by Sai Carpio's `node_unlink` module, which simulates the failure of a link between nodes in the NoC mesh by retransmitting a constant zero value regardless of the input, the faulty module would behave as if it neither receives data from the NoC nor returns any multiplication or division result. Instead, it would always return zero regardless of the operands received and, more importantly, it would send a signal filled with zeros rather than the periodic message previously proposed to announce its position in the NoC. This implementation would effectively emulate a hardware fault within the module.

We developed a very simple testbench to verify that the system behavior remains consistent before and after applying DFX (`exu_tb.sv`). It checks whether the multiplication and division of the operands 6 and 2 produce the correct results, 12 and 3 respectively, thus confirming that the multiplier and divider units operate correctly (Figure 6.2). The testbench ran successfully on the project before applying DFX and is available in the GitHub repository to be re-executed once the project fully implements dynamic partial reconfiguration.

Figure 6.2: Testbench run before DFX

6.3. NoC Advantages for Partial Reconfiguration

Integrating a NoC within the RISC-V processor has demonstrated clear advantages for enabling dynamic partial reconfiguration. The NoC-based architecture provides a modular and flexible communication backbone that simplifies the relocation and replacement of hardware modules at runtime. The design can decouple logic placement from communication by connecting reconfigurable areas through the NoC rather than instantiating all modules directly inside the EXU. This approach enables modules to be dynamically replaced without modifying the EXU or disrupting its operation, while the NoC handles data routing to maintain continuous communication regardless of module location.

Although timing challenges remain to be fully resolved, the architectural modifications and the proposed mechanisms for dynamic discovery and synchronization showcase the NoC's potential to maintain system consistency despite reconfigurations. Additionally, the availability of significant spare FPGA resources reinforces the feasibility of implementing DFX-based fault recovery techniques. In summary, this work supports the hypothesis that incorporating a NoC to communicate reconfigurable modules in a RISC-V processor enhances dynamic reconfiguration capabilities, providing a strong foundation for improved fault tolerance in FPGA designs.

Bibliography

AMD Xilinx. *Vivado Design Suite User Guide: Dynamic Function eXchange (UG909)*. AMD, Inc., 2024. Version 2024.2.

Avizienis, A. Fault-tolerant systems. *IEEE Transactions on Computers*, Vol. C-25, 1304–1312, 1976.

Becker, T., Luk, W. and Cheung, P. Y. Enhancing relocatability of partial bitstreams for run-time reconfiguration. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, 35–44. IEEE, 2007.

Billauer, E. Understanding partial reconfiguration with vivado. 2019. Available at: `https://www.01signal.com/vendor-specific/xilinx/partial-reconfiguration/part1-introduction/` (Accessed: 2025-04-27).

Carpio Cuenca, S. *Implementación de una red en chip tolerante a fallos en un procesador RISC-V*. Trabajo de fin de grado, Universidad Complutense de Madrid, Facultad de Informática, 2024.

CHIPS Alliance. *RISC-V VeeR EL2 Programmer's Reference Manual documentation*, 2022. `https://chipsalliance.github.io/Cores-VeeR-EL2/html/main/docs_rendered/html/overview.html`.

CHIPS Alliance. VeeRwolf: FuseSoC-based SoC for VeeR EH1 and EL2. Online, 2023. `https://github.com/chipsalliance/VeeRwolf`.

Davó Laviña, D. *Implementación de una red en chip en un procesador RISC-V*. Trabajo de fin de grado, Universidad Complutense de Madrid, Facultad de Informática, 2022.

Dubrova, E. *Fault-Tolerant Design*. Springer New York, 2013. ISBN 978-1-4614-2112-2.

Eastland, N. Structure of an fpga. *Digilent Blog*, 2025.

IBM Corporation. Availability design and principles: The 9s. `https://www.ibm.com/docs/en/warehouse-management/9.5.0?topic=principles-9s`, 2021. Accessed: 2025-03-24.

International Electrotechnical Commission. Iec 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems. https://webstore.iec.ch/publication/22273, 2010. 2nd Edition.

International Electrotechnical Commission. Iec 61511: Functional safety – safety instrumented systems for the process industry sector. https://webstore.iec.ch/publication/24355, 2016. Parts 1–3.

Johnson, B. Fault-tolerant microprocessor-based systems. *IEEE Micro*, Vol. 4(6), 6–21, 1984. ISSN 0272-1732.

Koren, I. and Krishna, C. M. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn., 2007. ISBN 0120885255.

Laprie, J.-C. Dependable computing and fault tolerance: Concepts and terminology. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, 2–11. IEEE Computer Society, 1985.

Li, T., Liu, H. and Yang, H. Design and characterization of seu hardened circuits for sram-based fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 27(6), 1276–1283, 2019.

Mitra, S. and McCluskey, E. J. Word-voter: a new voter design for triple modular redundant systems. In *Proceedings 18th IEEE VLSI Test Symposium*, 465–470. IEEE, 2000.

Rausand, M. and Høyland, A. *System Reliability Theory: Models, Statistical Methods, and Applications*. Wiley-Interscience, 2nd edn., 2004. ISBN 9780471471332.

Vipin, K. and Fahmy, S. A. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Comput. Surv.*, Vol. 51(4), 2018. ISSN 0360-0300.

Waterman, A. S. *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

Appendix A

DFX Supported Devices

The following table A.1 shows the devices supported for the current release (April 2025).

Table A.1: DFX Supported Devices

Device	Variants*
AMD Artix™ 7 Devices	
xc7a15t	A, L
xc7a35t	A, L
xc7a50t	A, L, Q
xc7a75t	A, L
xc7a100t	A, L, Q
xc7a200t	L, Q
AMD Kintex™ 7 Devices	
xc7k70t	L
xc7k160t	A, L
xc7k325t	L, Q, QL
xc7k355t	L
xc7k410t	L, Q, QL
xc7k420t	L
xc7k480t	L
AMD Virtex™ 7 Devices	
xc7v585t	Q
xc7v2000t	
xc7vx330t	Q
xc7vx415t	
xc7vx485t	Q
xc7vx550t	
xc7vx690t	Q
xc7vx980t	Q
xc7vx1140t	

Continued on next page

Continued from previous page	
Device	**Variants***
xc7vh580t	
xc7vh870t	
AMD Zynq™ 7000 Devices	
xc7z007s	
xc7z010	A
xc7z012s	
xc7z014s	
xc7z015	
xc7z020	A, Q
xc7z030	A, Q
xc7z035	
xc7z045	Q
xc7z100	Q
AMD Kintex™ UltraScale™ Devices	
xcku025	
xcku035	
xcku040	Q
xcku060	Q, QR
xcku085	
xcku095	Q
xcku115	Q
AMD Virtex™ UltraScale™ Devices	
xcvu065	
xcvu080	
xcvu095	
xcvu125	
xcvu160	
xcvu190	
xcvu440	
AMD Artix™ UltraScale+™ Devices	
xcau7p	A
xcau10p	A
xcau15p	A
xcau20p	
xcau25p	
AMD Kintex™ UltraScale+™ Devices	
xcku3p	
xcku5p	Q
xcku9p	
xcku11p	
xcku13p	
xcku15p	Q
Continued on next page	

Continued from previous page

Device	**Variants***
xcku19p	
AMD Virtex™ UltraScale+™ Devices	
xcvu2p	
xcvu3p	Q
xcvu5p	
xcvu7p	Q
xcvu9p	Q
xcvu11p	Q
xcvu13p	Q
xcvu19p	
xcvu23p	
xcvu27p	
xcvu29p	
xcvu31p	
xcvu33p	
xcvu35p	
xcvu37p	Q
xcvu45p	
xcvu47p	
xcvu57p	
AMD Zynq™ UltraScale+™ MPSoC Devices	
xczu1cg	
xczu1eg	A
xczu2cg	
xczu2eg	A
xczu3cg	
xczu3eg	A, Q
xczu3tcg	
xczu3teg	A
xczu4cg	
xczu4eg	Q
xczu4ev	A
xczu5cg	
xczu5eg	
xczu5ev	A, Q
xczu6cg	
xczu6eg	
xczu7cg	
xczu7eg	
xczu7ev	A, Q
xczu9cg	
xczu9eg	Q

Continued on next page

Continued from previous page	
Device	**Variants***
xczu11eg	A, Q
xczu15eg	Q
xczu17eg	
xczu19eg	Q
AMD Zynq™ UltraScale+™ RFSoC Devices	
xczu21dr	Q
xczu25dr	
xczu27dr	
xczu28dr	Q
xczu29dr	Q
xczu39dr	
xczu42dr	
xczu43dr	
xczu46dr	
xczu47dr	
xczu48dr	Q
xczu49dr	Q
xczu63dr	
xczu64dr	
xczu65dr	Q
xczu67dr	Q
AMD Versal™ AI Core Devices	
xcvc1502	
xcvc1702	Q
xcvc1802	
xcvc1902	Q, QR
xcvc2602	
xcvc2802	
AMD Versal™ AI Edge Devices	
xcve1752	A
xcve2002	A
xcve2102	A, Q
xcve2202	A
xcve2302	A, Q, QR
xcvc2602	A
xcvc2802	A
AMD Versal™ Prime Devices	
xcvm1102	Q
xcvm1302	
xcvm1402	Q
xcvm1502	Q
xcvm1802	Q
Continued on next page	

Continued from previous page	
Device	**Variants***
xcvm2202	
xcvm2302	
xcvm2502	
xcvm2902	
AMD Versal™ Premium Devices	
xcvp1002	
xcvp1052	Q
xcvp1102	
xcvp1202	Q
xcvp1402	Q
xcvp1502	Q
xcvp1552	
xcvp1702	Q
xcvp1802	
xcvp2502	Q
xcvp2802	
AMD Versal™ HBM Devices	
xcvh1522	
xcvh1542	
xcvh1582	
xcvh1742	
xcvh1782	

*Variants in addition to Commercial Grade (XC)

- L = Low Power
- A = Automotive
- Q = Defense Grade
- QL = Defense Grade, Low Power
- QR = Defense Grade, Radiation Tolerant

Note: All speed grades and temperature grades are supported.

Appendix B

Vivado DFX Project Flow

Xilinx's UG909 (AMD Xilinx, 2024) suggests two workflows for Partial Reconfiguration: the non-project flow, where implementation is carried out by explicitly writing and executing Tcl scripts; and the project flow, which involves using Vivado's graphical user interface and automatically generated scripts. For this project, the Project Flow was selected due to the design involving multiple reconfigurable modules and the centralized dependency management that it provides. Additionally, working with the graphical interface was more convenient and intuitive than using command-line tools.
The general steps for a Dynamic Function eXchange Project in Vivado are as follows:

1. **Create a Dynamic Function eXchange Project**
   On an existing project in Vivado, start by selecting **Tools > Enable Dynamic Function eXchange...** and confirming the irreversible conversion. The Tcl command actullay executed is:
   `set_property PR_FLOW 1 [current_project]`

2. **Create a Partition Definition**
   Right-click on the desired module and select **Create Partition Definition** to begin the process of Reconfigurable Partition (RP) creation. In the dialog box that appears, give this partition definition a unique name. Also define a name for the first Reconfiguration Module (RM). This RM is created from the RTL sources currently residing in this level of hierarchy. More RM are added or created later in the flow. Each instance of the module is shown in the **Hierarchy** view with a diamond, indicating that it is an RP. Repeat this step for all unique RPs required within the design.

3. **Define Parent and Child implementations**
   This consists of adding more RM for each of the RPs, defining a full set of Configurations that combine RMs with the static design, and declaring the set of runs that will be used to implement all the Configurations. All of these additions are done within the Dynamic Function eXchange Wizard. Select **Tools > Dynamic Function eXchange Wizard**. To add an RM in the **Edit Reconfigurable Modules** window, click +, enter a unique name for it, associate it with the appropriate partition definition, and add the required

source files. In the **Edit Configurations** window, click + to add a new Configuration, ensuring that each Configuration has a different RM. In the **Edit Configuration Runs** window, delete any existing runs (if present) and click **automatically create configuration runs**. Vivado will generate a parent run and child runs for each Configuration. Finally, click **Finish** to apply the changes. Repeat this process for all existing RMs for every Partition Definition. If a greybox module is desired, no action is required, because this is a built-in Vivado Design Suite feature that requires no sources.

4. **Floorplanning**
   Start the synthesis of the project. The synthesis of the reconfigurable module will automatically be done as an Out-of-Context (OOC) run. Afterward, draw a Pblock for the reconfigurable logic. To do this, with a post-synthesis design configuration open, go to the **Netlist hierarchy view**, right-click on the module corresponding to the reconfigurable partition (RP), and select **Floorplanning > Draw Pblock**. Once the Pblock for the RP is drawn, its properties can be accessed in the **Pblock Properties** window under the **Properties** tab. There, you'll find two options unique to RPs: `RESET_AFTER_RECONFIG` (available for Series-7 only) and `SNAPPING_MODE`. After creating a Pblock for each RP, the design can be implemented. Click the **Run Implementation** button in the Flow Navigator to initiate place and route for the parent run first. Once completed, all child runs will launch in parallel, using the static design results of the parent as a starting point.

5. **Generate Bitstreams**
   Once all the desired Configurations have been placed and routed, bitstreams can be generated. Similar to the implementation process, the **Generate Bitstream** button in the Flow Navigator can be used. This will launch the `write_bitstream` for all child runs, as well as the active parent. Additionally, a local right-click on any configuration allows you to call `write_bitstream` directly for that specific configuration.

Appendix C

VeeRwolf Guide

VeeRWolf GitHub Repository:
`https://github.com/chipsalliance/VeeRwolf`

Note: These steps were performed on Ubuntu 24.04.1. We initially tried using Windows, but encountered errors with FuseSoC commands, since it seems to be primarily designed for Linux systems.

Installation and Setup

1. Install dependencies:

   - Install Vivado
   - Install Verilator (recommended version: 5.036 or newer)

2. Create a working directory:
Create an empty directory (e.g., `veerwolf`) that will serve as the project root. From now on, it will be referred to as `$WORKSPACE`. Inside this directory, run:

```
export WORKSPACE=$(pwd)
```

3. Create and activate a virtual environment in the project root directory:

```
python3 -m venv veerwolf_env
source veerwolf_env/bin/activate
```

4. Install and configure FuseSoC:

   - Make sure FuseSoC version 1.12 or higher is installed:

   ```
   pip install fusesoc
   ```

   - Add the base FuseSoC library:

   ```
   fusesoc library add fusesoc-cores https://github.com/fusesoc/fusesoc-cores
   ```

- Add the VeeRwolf library:

```
fusesoc library add veerwolf https://github.com/chipsalliance/VeeRwolf
```

- After this step, your workspace should look like this:

  - $WORKSPACE
    - fusesoc_libraries
      - fusesoc-cores
      - veerwolf
    - veerwolf_env

- Set the environment variable `$VEERWOLF_ROOT` for convenience:

```
export VEERWOLF_ROOT=$WORKSPACE/fusesoc_libraries/veerwolf
```

Running the SoC

The VeeRwolf SoC can be run either in simulation or on hardware (Nexys A7 board).

VeeRwolf sim

VeeRwolf sim is a simulation target that wraps the VeeRwolf core in a testbench to be used by verilator or event-driven simulators such as QuestaSim.
To build and run the simulation model:

```
fusesoc run --target=sim --flag=cpu_el2 veerwolf
```

To rerun the program without rebuilding the simulation model:

```
fusesoc run --target=sim --run --flag=cpu_el2 veerwolf
```

Note: VeeRwolf allows to preload an application into memory using the `-ram_init_file` parameter. VeeRwolf includes some example applications under `$VEERWOLF_ROOT/sw`.
If `-ram_init_file` is not specified, `hello.vh` is loaded by default.
To convert your own `.S` file to `.vh`:

- Install the RISC-V toolchain (if not already installed):

```
sudo apt install gcc-riscv64-unknown-elf
```

- Edit the Makefile (in `$VEERWOLF_ROOT/sw`):
  At the end of the `%.elf:` rule, add: `-nodefaultlibs`

- Compile:

```
make example.vh
```

To build the simulation model and run your own program:

```
fusesoc run --target=sim --flag=cpu_el2 veerwolf \
--ram_init_file=$VEERWOLF_ROOT/sw/example.vh
```

VeeRwolf Nexys

VeeRwolf Nexys is a version of the SoC designed to run on the Diligent Nexys A7 FPGA development board.
To build (and optionally flash) an image for the Nexys A7:

```
fusesoc run --target=nexys_a7 --flag=cpu_el2 veerwolf
```