Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2018

MSc THESIS

Dynamically Reconfigurable Fault-Tolerant Design of $\rho-$VEX Softcore Processor

MUHAMMAD USMAN SALEEM

CE-MS-2018-16

Abstract

Over the past many years, technology scaling has resulted in a continuous reduction of lateral and vertical dimensions of transistors. The technology scaling, on the one hand, has led to a commensurate performance gain for very-large-scale integration (VLSI) circuits, but on the other hand, has also made such circuits more vulnerable to ionizing radiations which can cause single event effects(SEEs). These SEEs may cause the underlying user circuitry to deviate from its normal behavior. Devices that are destined for space missions need special protection for such kind of anomalies as space environment is filled with massive amount of high energy particles and ionizing radiations. In this thesis, the design, implementation, and verification of a fault-tolerant ρ-VEX, a softcore processor, is presented, so that it could be used as an attractive alternative to expensive radiation-hardened processors for space-based applications. ρ-VEX is a VLIW based, dynamically reconfigurable processor. Keeping in line with its inherent attribute, a dynamically reconfigurable fault-tolerant mode is presented in this work, which provides the running application an option to activate and deactivate the fault-tolerant mode multiple times. In this mode, for the protection of processor pipeline, a non-traditional TMR approach that requires 3 lanegroups running in 2-way mode is implemented. For the reliability of user memories, Hamming codes are implemented as an ECC coding scheme. The functionally of our fault-tolerant design is verified by using both a simulation-based platform (ModelSim) and an on-board FPGA platform (ML605 development kit). To measure the fault-tolerant capabilities of the ρ-VEX core, saboteurs are used to artificially inject faults at various predefined locations in the core. The obtained results have shown that our design can mitigate all injected single faults in the pipeline and double faults in the caches, without triggering any failure. The dynamically configurable fault-tolerant feature is obtained at the cost of about 30% additional resource utilization and 20% reduction in the maximum operating frequency.

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Dynamically Reconfigurable Fault-Tolerant Design of $\rho-$VEX Softcore Processor

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

MUHAMMAD USMAN SALEEM
born in PAKISTAN

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Dynamically Reconfigurable Fault-Tolerant Design of $\rho-$VEX Softcore Processor

by MUHAMMAD USMAN SALEEM

Abstract

Over the past many years, technology scaling has resulted in a continuous reduction of lateral and vertical dimensions of transistors. The technology scaling, on the one hand, has led to a commensurate performance gain for very-large-scale integration (VLSI) circuits, but on the other hand, has also made such circuits more vulnerable to ionizing radiations which can cause single event effects(SEEs). These SEEs may cause the underlying user circuitry to deviate from its normal behavior. Devices that are destined for space missions need special protection for such kind of anomalies as space environment is filled with massive amount of high energy particles and ionizing radiations. In this thesis, the design, implementation, and verification of a fault-tolerant ρ-VEX, a softcore processor, is presented, so that it could be used as an attractive alternative to expensive radiation-hardened processors for space-based applications. ρ-VEX is a VLIW based, dynamically reconfigurable processor. Keeping in line with its inherent attribute, a dynamically reconfigurable fault-tolerant mode is presented in this work, which provides the running application an option to activate and deactivate the fault-tolerant mode multiple times. In this mode, for the protection of processor pipeline, a non-traditional TMR approach that requires 3 lanegroups running in 2-way mode is implemented. For the reliability of user memories, Hamming codes are implemented as an ECC coding scheme. The functionally of our fault-tolerant design is verified by using both a simulation-based platform (ModelSim) and an on-board FPGA platform (ML605 development kit). To measure the fault-tolerant capabilities of the ρ-VEX core, saboteurs are used to artificially inject faults at various predefined locations in the core. The obtained results have shown that our design can mitigate all injected single faults in the pipeline and double faults in the caches, without triggering any failure. The dynamically configurable fault-tolerant feature is obtained at the cost of about 30% additional resource utilization and 20% reduction in the maximum operating frequency.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2018-16

Committee Members :

Advisor:	Dr .ir. J.S.S.M. Wong, CE, TU Delft
Chairperson:	Dr .ir. J.S.S.M. Wong, CE, TU Delft
Member:	Dr. ir. A. van Genderen, CE, TU Delft
Member:	Dr. A. Menicucci, SE, TU Delft

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
BRAM	Block Random Access Memory
CMOS	Complementary Metal Oxide semiconductor
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DLP	Data-Level Parallelism
DRAM	Dynamic Random Access Memory
DWC	Duplication With Compare
ECC	Error Correcting Code
EDAC	Error Detection and Correction
FF	Flip-flop
FPGA	Field-Programmable Gate Array
GF	Galois Field
GPU	Graphics Processing Unit
HDL	Hardware Description Language
IC	Integrated Circuit
ILP	Instruction-Level Parallelism
LCM	Least Common Multiple
LEO	Low Earth Orbit
LUT	Lookup Table
PC	Program Counter
SAA	South Atlantic Anomaly
SEC	Single Error Correction
SECDED	Single Error Correction, Double Error Detection
SED	Single Error Detection

SEE	Single Event Effect
SEFI	Single Event Functional Interrupt
SEL	Single Event Latch-up
SET	Single Event Transient
SEU	Single Event Upset
SIMD	Single Instruction Multiple Data
SoC	System on Chip
SRAM	Static Random Access Memory
TID	Total-Ionizing Dose
TLP	Thread-Level Parallelism
TMR	Triple Modular Redundancy
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
VLSI	Very-Large-Scale Integration

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Stephan Wong, without his help this work could not have been completed. His continuous guidance and support throughout the span of this thesis kept me motivated and committed. Additionally, I would also like to extend my gratitude to Jeroen van Straten who provided technical assistance and helped me in getting started with the project. Although he had a lot of work of his own, I could always drop by and ask for his help.

I also want to express thanks to my colleague and seniors that include Jacko, Omar, Imran and Nauman for their company as I could often visit them to have a little chit-chat. Many thanks also go to the QCE Feestcommissie who kept organizing various social events and gatherings to help us remember that there's a whole world outside TU.

And finally, the most important one, I wish to thank my parents and siblings, without them I would never have come to the end of this master studies. They have been there for me since day one, and continue to be by my side no matter what.

MUHAMMAD USMAN SALEEM
Delft, The Netherlands
August 21, 2018

Introduction 1

Since the beginning of mankind, humans have felt a primordial urge to explore - to blaze new trails, map new lands, and answer profound questions about ourselves and our Universe. The quest to unravel the mysteries led to the launch of a first artificial satellite named *Sputnik 1* in 1957 [2]. This successful launch, a remarkable achievement, thus began the space age, and since then, spacecrafts are being launched into and beyond the Earth's orbit. These robotic emissaries follow in the footsteps of their predecessors and are testaments to our long-lived desire to understand our place in the Cosmos. However, these space missions come with their own intrinsic challenges. The harsh space environment in which spacecraft operates poses many challenges to the spacecraft designers. The spacecrafts are subject to faults caused by equipment failure or environmental impacts, such as radiations, vibrations or temperature extremes. The anomalies introduced in the spacecraft electronics have been known since the very first day of space exploration [3] [4] [5] [6]. Therefore, to sustain the harsh environment of space, systems must be robust enough to operate reliably for the desired duration with little or no maintenance.

Space missions are subject to a heavy dose of radiations in the form of cosmic rays and the solar wind, exposing them to alpha particles, protons, heavy ions, X-rays and ultraviolet radiations. Interaction of these radiations with matter results in atomic displacement (rearrangement of atoms in a crystal lattice) or ionization, with the potential of causing soft errors which may lead to permanent or transient damage to the system. Soft errors are caused by transistors changing their state unintended due to significant amounts of energy disposed of by striking particles. Initially, only deep-space missions such as interplanetary missions were the main concern for such soft-errors as compared to missions close to Earth/Moon because they operate in relatively very high radiations enriched environment. Missions targeted at lower orbits were not very vulnerable because their operating environment has relatively low radiations. Larger footprints of transistor provided benefits to such low-Earth orbit missions in the sense that they could sustain a strike by a low energy particle. But with the advancement in technology, the transistor size is becoming smaller and smaller (Moore's law), which means the amount of energy required for a state change of transistor is also becoming smaller. Although, providing huger performance improvements, this decrease in the footprint of a transistor is making devices more vulnerable to radiation-induced faults. These effects were first reported in 1975 for ICs used in satellites in space [5]. Over the past years, the transistor size has further decreased so much, that nowadays soft errors are also being experienced at LEO satellites [7] and normal flight altitudes for civilian aviation [8] . Using older technology (larger transistor footprint) is not a very feasible option these days, as the demands for more advanced missions require more computational ability and lower power consumption. Therefore, other solutions need to be explored to get rid of anomalies introduced by the radiations.

The radiation-hardened electronics components are available commercially in the market and can mitigate the effects of radiations to an extent. The problem with such solutions is that besides providing very limited options for customization, they are also highly expensive. For the last several years, we have observed an increasing trend in student satellites. Student satellite projects like *AAUSAT4* from Aalborg University(Denmark), *e-st@r*-II from Politecnico di Torino (Italy), *OUFTI-1* from University of Lige (Belgium), *Delfi-C*^{3} and *Delfi-n3Xt* from Delft University of Technology (Netherlands) are few examples in this regard [9] [10]. All such satellite projects usually share a common problem of funding and limited resources. Use of expensive radiation-hardened components is not a very feasible option. Therefore, low-cost COTS are preferred for these projects. The issue with such low-cost solutions is that they come with limited reliability assurance. Therefore, there is a high need for non-expensive solutions that can be trusted for their reliability in a harsh space environment.

Traditionally, many space-based applications use microprocessors for on board processing as the development of software is relatively less expensive. In 1977, the RCA 1802 was the first microprocessor selected for spaceflight when it was used for the Galileo probe mission to Jupiter [11] [12]. Since then microprocessors are considered an integral part of most space missions. For most applications, the performance of a modern CPU is sufficient, but for real-time computationally intensive applications, microprocessors alone are not enough. Customized application specific hardware is a way to go for such applications. FPGA, a reconfigurable chip used for designing dedicated hardware for computationally intensive tasks, provides a more appropriate platform for such applications. They can provide solutions in an order of magnitude faster than the software algorithms developed for commercial general purpose microprocessor. In addition to providing high-performance gain, FPGA also offers benefits of allowing in-orbit customization (design changes), which helps in correcting errors or updating system design in order to cope with new missions requirements after launch. All these attributes make FPGA an attractive option for the remote space missions.

Specialized designs on FPGA often include a softcore, a customized CPU that based on its design can possess reduced or extended functionality of a commercial microprocessor. Today, there are multiple softcore processors available in the market, developed by open source communities and (large) companies. Open source softcores provide the user more possibilities for modification or customization of the core as source files of the design are available. However, the softcores developed by companies are often proprietary and expensive licenses are required to use them. Regardless of the source of these softcore processors, if these are intended to be used in space missions, these must possess the ability to sustain the harsh environment and handle soft errors, as any lapse might lead to a mission failure.

The ρ-VEX, a dynamically reconfigurable softcore processor, is developed by Computer Engineering department of the Delft University of Technology(Netherlands). It is an open-source VLIW based processor. The distinguishing feature of ρ-VEX is its ability to adapt itself to available ILP and TLP to utilize the available computational resources more efficiently. The goal of this thesis is to modify the softcore such that it can handle soft errors for future opportunities in the high safety and security-critical domain, and provide student satellites a reliable option as an alternative to expensive solutions. This document will describe the way in which the fault-tolerant attribute of ρ-VEX is designed, implemented and verified.

1.1 Research question & thesis objectives

In this work, the existing design of ρ-VEX softcore is taken as the starting point and efforts are put to make it fault-tolerant. To be developed variant of this softcore is targeted at high safety and critical space-based applications, such that correct execution flow could not be influenced by single event effects without such anomalies being detected. The research question/problem statement can be formulated as:

How can the ρ-VEX softcore be extended so that it becomes a reliable option for space-based critical missions?

Based on the problem statement, thesis objectives are defined as:

- The design must be able to detect and correct errors in the execution stages of ρ-VEX.
- The design must make on-chip memories robust against soft-errors.
- The design must be dynamically (runtime) reconfigurable.
- The design must be implemented on an FPGA while taking platform independence into account.
- The design must be verified for its correctness.

1.2 Methodology

A research methodology is defined to find a solution for the proposed research question and to achieve the thesis objectives. This methodology is specified as follows:

- Investigate the possible causes and impacts of single event effects in soft-cores.
- Investigate existing solutions for the mitigation of single event effects.
- Explore the ρ-VEX platform to analyze possible and feasible solutions.
- Implement the most suitable solution in a modular and efficient way

- Verify the implemented functionality comprehensively using conformance tests in simulation and benchmarks in hardware for its correctness
- Analyze the results of the implementation.
- Propose recommendations for future development of the implemented solution.

1.3 Thesis outline

The thesis is organized in the following way:

Chapter 2 provides the background knowledge required for this thesis. It provides details about the ρ-VEX platform and describes its distinguishing features. Details about space environment and its impact on electronics are also provided. Based on these impacts and their causes, different solutions to mitigate their effects in processors are discussed. Lastly, methods for verification of fault-tolerant systems are investigated.

Chapter 3 analyses the architecture of the baseline processor for its susceptibility to radiation-induced faults. It then performs a comparative analysis of various available options to explore their reliability and suitability for our design. Fault mitigation techniques for pipeline stages and memory elements are evaluated and best-suited options for our platform are identified. After finalizing the entire design, fault injection mechanism is selected from different possible options to validate the desired functionality of our design.

Chapter 4 provides the implementation level details of the fault-tolerant design. After providing a high-level description of the design, it explains the design of each basic building block added to make the core robust. Finally, this chapter discusses the implementation of fault injection mechanism and the addition of status-monitoring registers to thoroughly validate the core functionality and fault-tolerant behavior.

Chapter 5 deals with the verification of our design and presents the results associated with the fault-tolerant core. It provides details about the platforms and benchmark used to check the correctness of the design. Statistics of fault injection tests are also discussed in the chapter. Besides testing the functionally of the core, the results of synthesis are also presented as they provide an indication of the cost of the design and its performance.

Finally, the chapter 6 summarizes the whole thesis, presents the main contributions and lists recommendations for the future work.

2 Background

This chapter details the background knowledge required for this thesis. In Section 2.1, a brief overview of FPGA will be provided, and in Section 2.2, the details of the processor platform that is used in this thesis, i.e., ρ-VEX will be provided. Its implementation, features, configuration modes and working will be discussed. Afterwards, in Section 2.3, details about the space environment and how it induces anomalies in electronics especially in processors will follow. Subsequent sections will then deal with mitigation techniques found in the literature to make processors robust for space missions, and finally, various methods to validate the resilience of fault-tolerant systems will be discussed.

2.1 FPGA

Knowledge of the underlying architecture is helpful in understanding both how the user design is implemented and how the space environment can it. As an FPGA platform is used for this thesis project, it is helpful to present its overview first. Field-programmable gate arrays (FPGAs) are configurable integrated circuit based on a high logic density regular structure, which can be customized by the end user to realize different designs [13]. It can support designs varying from simpler logic gates (AND, OR etc) to much complicated designs such as processors. The circuitry of an FPGA contains two-dimensional arrays of logic blocks and interconnects. Logic blocks are programmed to implement some functionality, while interconnects are programmed using switch boxes to establish connections among these logic blocks. The name "Field Programmable" comes from the unique attribute of FPGA that end-user can configure it after its manufacturing. Based on the technology by which design data is stored on FPGA, they can be divided into various categories that include SRAM-based, flash based and antifuse-based FPGA.

FPGA fills the performance gap between application-specific integrated circuits (ASICs) and general purpose processors. ASICs are the fastest processing elements for computationally intensive workload and can utilize a lot of parallelism. They are tailored especially for some target workload. However, the problem with ASICs is that they are only economical if produced in bulk quantity, otherwise they provide a very expensive solution. On the other hand, general-purpose processors provide an easy and relatively inexpensive solution, as building software is comparatively easy and economical. But the performance of such processors is constrained by the inherent sequential nature of software that runs on them. FPGA provides the parallel nature of ASIC along with the ease of implementation of software. They can provide relatively high performance as compared to microprocessors for compute-intensive applications. Moreover,

few classes of FPGA (e.g., SRAM, flash based) also possess the distinguishing feature that circuit design on them can be reconfigured multiple times. This re-programmable nature of FPGA makes it very suitable for space-based missions, as besides providing application specific hardware, it can also be reconfigured during the mission to remove any anomalies or design faults, and new design can also be added to cope with changing mission requirements.
Design to be implemented on FPGA can be created in various ways. This could be schematic based, hardware descriptive language (HDL) based or combination of both. Most commonly used HDLs are VHSIC hardware description language (VHDL), Verilog and SystemC. Selection of a design method and HDL is a designer's prerogative. The ρ-VEX, a soft-core processor platform that will be used in this thesis work is written entirely in VHDL. Therefore, to maintain the consistency, new design to turn existing one into a fault-tolerant design is also implemented in VHDL.

2.2 The ρ-VEX architecture

The processor platform that is used in this work is ρ-VEX. It stands for reconfigurable VLIW example (VEX). Its architecture is based on VEX, i.e., the example architecture from the book "Embedded computing: a VLIW approach to architecture, compilers and tools" by Joseph A. Fisher, Paolo Faraboschi, and Cliff Young [14]. ρ-VEX is a VLIW based soft-core processor that has been developed to exploit parallelism in an application to achieve better performance. Before explaining it further in detail, there are a few concepts that need to be understood. Considering that, subsequent sections will provide a brief overview of various types of computational parallelism, VLIW architectures and softcore processors, followed by the design, features, and working of the ρ-VEX processor.

2.2.1 Exploiting parallelism

In the domain of processors, parallelism refers to the opportunities in an application to find independent operations and execute them simultaneously rather than running sequentially. Exploiting parallelism in an application can increase its performance manifold times and this increase in performance is proportional to the degree of parallelism found in the application. In this section three widely known parallelism mentioned in [15] are discussed:

Instruction-level parallelism
Instruction-level parallelism (ILP) refers to the existence of independent instructions in an application program. These independent instructions can be run simultaneously with other instructions in the same clock cycle. The amount of ILP that can be extracted from an application depends on the data dependencies and branches present in the instruction stream. Super-scalar and VLIW architectures are the platforms that exploit ILP in an application. In case of VLIW processors, finding independent instructions is the job of a compiler, while in case of super-scalar processors dedicated run-time control hardware is responsible for it. ILP can also be combined with any other type of

parallelism to further complement the performance gain.

Data-level parallelism
Data-level parallelism (DLP) refers to distributing the data across different computing nodes and executing them in parallel. These different computing nodes receive a small chunk of data and perform the identical operation on it in parallel, and the results are then combined to get a single finalized result. Single instruction multiple data (SIMD) is a form of DLP as it exploits parallelism in a data stream. Graphics processing unit (GPU) is another example that exploits DLP.

Thread-level parallelism
Thread-level parallelism (TLP) refers to executing multiple tasks of an application in different threads of a multi-threaded system or in different cores of a multi-core system. Multi-core systems can have multiple homogeneous or heterogeneous processing elements on which subprograms of a bigger program can be run simultaneously to exploit parallelism and get better performance. These subprograms can either communicate with and wait for each other, or they can also run completely independent of each other.

2.2.2 VLIW architecture

Processors based on a very-long instruction word (VLIW) architecture exploits ILP to get better performance. VLIW architecture contains multiple independent functional units that are capable of executing multiple instructions in parallel. By packing multiple instructions in a single long instruction word, these instructions can be executed in parallel in the same clock cycle to take advantage of ILP and reduce the overall execution time of an application. The maximum number of instructions that can be packed in a single instruction word and are supported by a given VLIW based processor is known as an Issue-Width.
Finding data dependencies and packing of independent instructions in an instruction word is done at the compile time and thus in VLIW architecture, the code is statistically scheduled. The burden of scheduling the instructions lies on the compiler and is done only once. This behavior is better off in both energy and logic area of a processor as compared to other platforms that do dynamic scheduling, e.g., superscalar processors. The downside of a VLIW processor is that the code is compiled for a specific issue width and it cannot run on a different issue width. It needs to be recompiled for a different issue width. Furthermore, in practice, the full issue-width cannot always be used as it is limited by the level of dependencies present among the instructions. It is the responsibility of the compiler to schedule instructions as efficiently as possible with the help of various scheduling algorithms.

2.2.3 Soft-core processors

Embedded systems based on soft-core processors are becoming very popular these days. A soft-core processor is a microprocessor that can be fully implemented in logical primitives of an FPGA and it provides the user a substantial amount of flexibility in design through the configurable nature of an FPGA.

When designing some complex embedded system using an FPGA, quite often we need some kind of processor to handle various system tasks. One way is to use commercial-off-the-shelf (COTS) microprocessor and mount it on the same board with FPGA and use some standard interface to communicate with FPGA. This is a viable option and most commonly used, but it comes with few limitations. For example, an application that needs some additional peripheral functionality that a COTS microprocessor do not provide, can not benefit from this discrete solution.
The other option is to embed the hardcore processor on the chip, which means it gets dedicated silicon on the chip. This allows the processor to run at the same frequency as it would run in a discrete way and theoretically provides the same level of performance. Many such solutions exist in the market nowadays. One example is Zynq-7000 SoC family. Zynq-7000 devices come with dual-core ARM Cortex-A9 processors integrated with Artix-7 or Kintex-7 based programmable logic for better performance and maximum design flexibility[16]. However, the issue with hard processors is that they can not be customized to better meet the needs of a particular system.
Soft-core processors, on the other hand, provides user much flexibility of design. They can entirely be implemented on an FPGA. The user, as per the requirements of its system, can make the design which has reduced or extended functionality than a hard processor. However, these soft-core processors also have few constraints. Because of the implementation in reconfigurable logic, they do not beat hard processors in operating frequency and run at a relatively lower frequency. In fact, they can not beat the performance, area, and power of hard-core processors, but still, for many embedded applications, soft-core processors are a preferred option, e.g., the applications which prefer expanded functionality over increased frequency. Soft-core provides a low-cost solution and can also be re-targeted to a new technology without much effort. Few examples of soft-core processors are LEON3, MicroBlaze, and OpenRISC. Many major FPGA vendors also provide soft-core processors in their product offerings.

2.2.4 The ρ-VEX processor

The ρ-VEX is a softcore processor based on VLIW architecture. It is a 32-bit big endian architecture. The distinct feature of ρ-VEX is that its architecture is designed such that the key metrics that include the issue width and the number of available multiplication units are configurable. This feature allows the processor parameters to be tailored to suit the ILP and arithmetic instruction mix of a certain application once the software is available. The need for ρ-VEX arose because the existing VLIW based softcore processors, e.g., [17] [18] [19] suffered from one of the following drawbacks [20]:

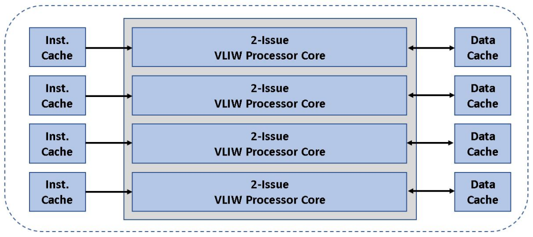
- Either compiler or processor design is not open source
- Toolchain lacks in good support
- Limited options for parametrized customization or extension

To overcome all these shortcomings, ρ-VEX was introduced in [21]. It is developed entirely by the students and Ph.D. candidates of the TU Delft. Over the period of time, there have been many revisions of it. However, for this thesis work, the latest version of

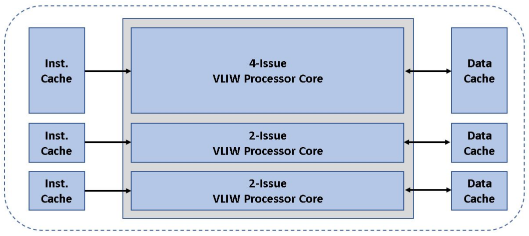
ρ-VEX has been used.
Let us here introduce some terminology that will be used very often in this thesis in reference to ρ-VEX. As it is a VLIW processor, this means that each instruction can specify multiple independent operations. Such operations are called *syllables*, a full instruction is called a *bundle*. The part of a VLIW processor that executes a syllable is called a lane (also referred as pipelane in this thesis). It contains computational resources to execute a syllable. The use of word *instruction* in this thesis may be used for either a bundle or a syllable, depending on the context. A VLIW processor capable of executing n-syllables per cycle is referred as *n-way* VLIW processor and the number of lanes running together as a part of a single processor are known as *issue width* of the processor.

Reconfigurability
The ρ-VEX is a parametrized processor that can be configured at design time and reconfigured at run-time. It must be noted here that this is not an FPGA reconfiguration, which means that we don't need to reconfigure and load the bitstream every-time we request reconfiguration. Here *reconfiguration* means a process within the system described by a single FPGA bitstream. There is no full or partial reloading of bitstream required. All resources required to do the switching are inferred from the hardware description, and the overhead of reconfiguration is mere pipeline flush.
The distinguishing feature of ρ-VEX is its ability to adapt itself to available ILP and TLP to utilize the available computational resources more efficiently. It can do so by dynamically changing the mapping between threads and issue slots. Though the total number of pipelanes are fixed, pipelanes can be distributed among different programs running in parallel, and this redistribution can be done at run-time. The core can behave as a large VLIW processor when high ILP is available, or in case of high TLP, it can behave as multiple smaller VLIW processors.

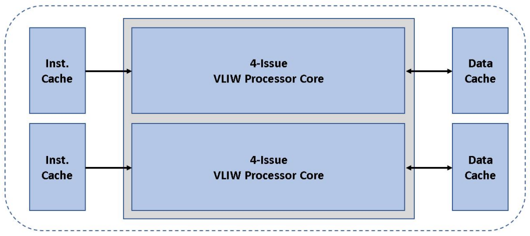
The default configuration of ρ-VEX consists of eight execution lanes called pipelanes. Not all of these pipelanes can be separated and operated independently. At least two consecutive pipelanes, often referred as lanepairs, must run together. Thus the minimum configuration that ρ-VEX can achieve is a 2-way (often referred as 2-issue) configuration. Figure 2.1 depicts various configuration modes of ρ-VEX. The core can be split up to four smaller cores (2-way) as depicted in Figure 2.1a or it can run as a single larger core (8-way) as depicted in Figure 2.1d. If an application has high ILP then to get better performance and utilize the available resources more efficiently, the core is run in full 8-way configuration, executing up to eight independent instructions in parallel. On the other hand, if an application has high TLP and lesser ILP, then running core in full mode will not provide a performance increase. Additionally, it will also be a wastage of available resources. Therefore for such scenario, the core has the ability to split up to four smaller 2-way VLIW cores, that can run four threads independently in parallel to each other resulting in increased performance. Each independent thread has its own state, called a *context*, consisting of the register files, PC, and other control registers. The core can also be reconfigured into two 4-ways cores (2.1c or one 4-way and two 2-way cores (2.1b, if required.
The parameters that can be configured at design time are listed in Table 2.1 [20].


Inst. Cache
2-Issue VLIW Processor Core
Data Cache
Inst. Cache
2-Issue VLIW Processor Core
Data Cache
Inst. Cache
2-Issue VLIW Processor Core
Data Cache
Inst. Cache
2-Issue VLIW Processor Core
Data Cache

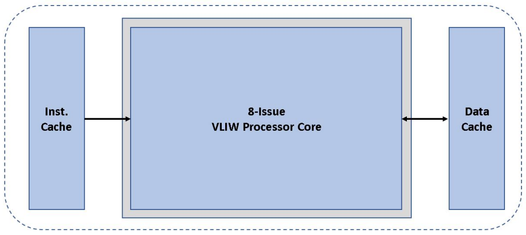
(a) 4x2-way


Inst. Cache
4-Issue VLIW Processor Core
Data Cache
Inst. Cache
2-Issue VLIW Processor Core
Data Cache
Inst. Cache
2-Issue VLIW Processor Core
Data Cache

(b) 1x4-way and 2x2-way



(c) 2x4-way



(d) 1x8-way

Figure 2.1: ρ-VEX configuration modes

Table 2.1: Design-time configuration parameters of ρ-VEX processor

Resources	**Parameters**
General	Issue width, Number of hardware context
Functional units	Number, Type and location, Support operations
Register file	Register file size
Interconnect	Presence of forwarding logic, memory bandwidth
Caches	Presence of caches, cache size and cache line size

Configuration word encoding

The encoding for the value that is written to the reconfiguration register is called configuration word. The size of configuration word is at most 32-bits, but in the current version of ρ-VEX, only least significant 16 bits are required to describe the configuration. The encoding is done in hexadecimal form. Each lane group (pipelane pair) of ρ-VEX is mapped to a nibble (four bits). The three least significant bits of each nibble specify the context it needs to be connected if the fourth most significant bit is zero, or a special mode if it is one. At this point, the only special mode defined is disabling the lane group, and it corresponds to nibble 8. Values 9 through F are reserved for future work.
An example would explain the configuration word encoding better. Consider an eight-way ρ-VEX with four lane groups and four contexts. 0x0000 then specifies a 1x8 lane configuration, with eight lanes working on context 0. 0x3210 specifies a 4x2 lane configuration, with first lane pair working on context 0, second on context 1, third on context

2 and fourth on context 3. 0x0013 specifies a 1x4 and 2x2 way lane configuration, with four lanes working on context 0, two lanes working on context 1 and two lanes working on context 3. 0x8800 would map to a 1x4 way configuration with four lanes running context 0 while remaining four lanes are deactivated or in power-down mode.
One must be careful when requesting a reconfiguration as not all the possible nibble combinations are valid. Following guidelines must be followed:

- Any context should be mapped to the power of two contagious lane groups. For example, 0x2030 and 0x8111 are invalid configuration words because these do not follow this rule.

- The nibble in configuration word can take value from zero to the number of hardware contexts minus one in order to map to a context. Nibble 8 is an exception which is explained earlier. Configuration words 0x 7711 is invalid for a configuration with four hardware contexts. Besides 8, the maximum nibble value that can be taken in this scenario is 3.

- The nibble in the configuration word corresponding to the non-existent lanegroups should be set to zero. For example, for a configuration with four hardware contexts, 0x00008210 is a valid configuration but 0x88888210 is not.

- A set of lane groups assigned to a single context should be aligned properly. For example, 0x0110 is invalid while 0x0011 is a valid configuration word.

Requesting a reconfiguration
There are three ways in which a reconfiguration of ρ-VEX processor can be requested.

- Writing new configuration word to the *context control register* (CRR) from a program running on the core.

- Writing new configuration word to the *bus reconfiguration request* (BCRR) global control register from the debug bus. This mechanism is similar to the first, except it is triggered externally, from outside the core.

- Using the sleep and wake-up system of the ρ-VEX. This involves writing new configuration word to *wake-up configuration* (WCFG) register and setting the flag in *sleep and wake-up control* (SAWC) register.

Usually, the new configuration is committed within something in the order of ten of cycles after its request, depending on how long it takes the configuration controller to pause and store the state of affected contexts. Reconfiguration can also be rejected sometimes. The reasons for the rejections can be following:

- Another context or the debug is requesting a new configuration simultaneously. The intended context might lose arbitration in this case.

- Configuration word does not comply with the encoding guidelines and is rejected.

2.3 Space environment

Anomalies in the spacecraft electronics have been known since the very beginning of space era. Space environment is a critical scenario for electronics in the sense that it contains a massive amount of radiations, and remote maintenance of space electronics is also not a feasible option in most of the cases. There are mainly two reasons that induce faults in electronic circuits namely radiations and device aging[22].

2.3.1 Radiation effects

The one prominent feature that distinguishes the space environment is the presence of huge amount of radiations. Radiation can be defined as energy in transit in the form of high-speed particles and electromagnetic waves[23]. Radiations can be divided into two categories: ionizing and non-ionizing

- **Ionizing radiations** are radiations that possess enough energy to remove electrons from the orbits of atoms, resulting in charged particles. Examples include gamma rays, neutrons, and protons. Effects of ionizing radiation are different than normal ions formation that happens in an ordinary chemical reaction, such as the formation of table salt from Sodium and Chlorine. In such reactions, electron(s) is(are) released from outer most orbit to form a positively charged ion. While in case of ionizing radiations (if energy is sufficient), electrons from the inner orbits can be released, resulting in a very unstable atom which is highly chemically reactive.
- **Non-ionizing radiations** are radiations that do not possess sufficient energy that is required to remove electrons from their orbits. Examples are visible light, radio waves, and microwaves. Such radiations are not a concern for electronics equipment.

Space radiations are mainly ionizing radiations which contain highly energetic charged particles. Three naturally occurring sources of space radiations are trapped radiations, galactic cosmic radiations, and solar particle events [23].

- **Trapped radiations**
  The Sun releases a stream of charged particles, known as solar wind, out into the space. The intensity of it depends on the amount of activity on the surface of Sun. This solar wind contains ions of many elements, however, the major chunk is of protons and electrons. When these particles tend to penetrate in Earth's atmosphere, Earth's magnetic field provides a hindrance to it. Earth's magnetic field is produced because of the rotation of Earth's iron core and it extends thousands of kilometers from Earth's interior out into the space. Most of the charged particles get deflected by the Earth's magnetic field, however, some become trapped in it. They are contained in one of the two magnetic rings surrounding the Earth commonly known as Van Allen radiation belts. The inner belt, extending from an altitude of about 1,000 to 8,000 miles, contains high concentration of electrons (hundreds of keV) and energetic protons (hundreds of MeV), while outer belt, extending from 12,000 to 25,000 miles, contains mainly

high energy electrons (0.1-10 MeV) [4][24].
Apart from the Apollo missions, NASA's manned spaceflight missions have stayed well below the altitude of the Van Allen belts[23]. However, there is an area where a part of inner Van Allen belt comes really close to Earth's surface, dipping down to an altitude of about 200 km. It is known as south atlantic anomaly(SAA) and is caused by the non-concentricity of the Earth and its magnetic dipole. This is the region where Earth's magnetic field is weakest relative to an idealized Earth-centered dipole field, which leads to an increased concentration of energetic particles in it. The largest part of radiation exposure to spaceflight missions occurs in SAA. Lower earth orbit flights traverse a portion of SAA six or seven times a day[23].

- **Galactic cosmic radiations**
  Galactic cosmic rays (GCR) are highly energetic background source of energetic particles that constantly bombard the Earth. They originate outside the solar system, very likely from the explosive events like supernova. These high energy particles consist of ionized atoms ranging from Hydrogen (accounting for 89% of GCR spectrum) to Uranium [25]. These particles travel at the large fractions of the speed of light and have tremendous energy. Earth's magnetic field provide shielding for spacecraft for most of GCR, however, they have access over the polar regions where the magnetic fields are open to interplanetary missions[23].

- **Solar particle events**
  Solar particle events (SPE) are injections of high energetic particles emitted by the Sun into interplanetary space. These particles include mainly protons, electrons and alpha particles. SPE occurs when the particles emitted by the Sun become accelerated either close to the Sun during a solar flare (highly concentrated, explosive release of energy) or in interplanetary space by coronal mass ejection (huge bubbles of plasma threaded with magnetic field emitted by the Sun) shocks. These particles impose significant operational constraints on space missions. Storm shelters with a significant amount of shielding are required to lower the radiation dose to tolerable levels for astronauts, and the critical equipment sensitive to such high dose need to be turned off to avoid soft errors or other radiation-induced damages [26].

The effects caused by the space radiations on the spacecraft depend on its orbit and the source of radiations. Energy level of main components that constitute the space environment mentioned in [3] are presented in Table 2.2.

The particles hitting the space electronics can either result in temporarily change in the behavior of some circuit (a soft error) or they permanently damage the circuit (a hard error). Two most widely known radiation effects on spacecraft electronics mentioned in [27] are TID and SEEs.

Table 2.2: Main sources of the space radiations

Radiation belts	Electrons	eV - 10 MeV
	Protons	keV - 500 MeV
Solar flares	Protons	keV - 500 MeV
	Ions	1 to a few 10 MeV/n
Galactic cosmic rays	Protons and ions	Max flux at about 300 MeV/n

2.3.1.1 TID

For long space missions or missions for extremely high radiations environment, such as interplanetary missions, accumulation of ionizing particles over the period of time could cause failure of components/FPGAs. This accumulation of ionizing radiation in electronics is referred as the total-ionizing dose (TID).
The amount of radiations which a particular component gets depend on a number of factors: orbit, duration of the mission, placement inside the spacecraft and the amount of outer shielding around the spacecraft. A short mission in the low-Earth orbit might only expose the FPGAs to 1-5 krad per year, while a mission to Jupiter might accumulate 10-100 krad per week [27]. This accumulation of ionizing radiation causes degradation in transistors. As mentioned in [28], the oxide trapped charges lead to a decrease in the threshold voltage of the n-channel transistor and cause an increase in the case of the p-channel transistor. This threshold voltage (the minimum voltage that is required to turn on the device) is a crucial factor in determining characteristics of a transistor and corrupting its value can entirely change the behavior of circuitry.
Different types of FPGAs can withstand a different dose of TID before the components failures. Table 2.3 lists TID tolerance limit for Xilinx FPGAs provided in [27]. Mitigation against TID will not be discussed in this thesis further and it is proposed that FPGA selection must be done carefully as per the mission needs, to avoid any TID based anomalies.

Table 2.3: Listing of TID limits for Xilinx FPGAs

FPGA	**TID limit**
	krad
Virtex	100
Virtex-II	200
Virtex-4	250
Virtex-5	340
Virtex-5QV	1000
Virtex-6	380

2.3.1.2 Single event effects

SEEs are unintended effects caused by the interaction of a single ionizing, energetic particles with electronic components. These ionizing particles can be primary, like heavy ions and alpha particles or secondary, created by a nuclear reaction of a particle with silicon or any other atom of the die. SEEs occur when the accumulation of charge liberated by the ionizing particles become more than the electric charge stored on a sensitive node[29] (a node in a circuit whose electric potential can be changed by accumulation or internal injection of electrical charges).
SEEs induced by the deposition of energy from ionizing particles can either by non-destructive or destructive, based on their effects. Non-destructive SEEs are transient effects and a device can be recovered by resetting or reconfiguring, while, destructive SEEs are permanent in nature and have a persistent effect even after resetting or reconfiguring the device [22]. Four widely known single event effects are SEL, SEU, and SET. [27].

- **SEL**
  Single event latch-up (SEL) is a radiation-induced version of latch-up [1]. It affects the behavior of parasitic thyristors in CMOS technology. A single energetic particle can switch PNPN structures from high impedance state to a low impedance state, which causes an abnormal amount of current flow through the sensitive regions of the device structure causing it to lose its functionality. This current increase happens in a very small period of time (milliseconds) making to difficult to detect the current increase before the component gets damaged. In some cases, components retain some partial functionality after the event, but most components do not function at all after the event [27]. All CMOS components have a potential for SEL sensitivity.
  FPGAs provided by all the main manufacturers these days are latch-up immune. Xilinx and Microsemi have published reports verifying latch-up immunity in high radiation environment[30, 31]. In back days, Altera devices had very low SEL immunity and, therefore, was not recommended for space-based applications[32] but now modern SRAM based Altera devices such as Stratix-IV possesses very high SEL immunity[33]. By careful selection of FPGA, single event latch-ups can be avoided, so they will not be discussed further in this thesis.

- **SEU**
  Single event upsets (SEUs) are anomalies caused in the memory cells because of radiations. The susceptibility to SEU depends on the type of memory elements (SRAM, DRAM etc). For SRAM-based FPGAs, SEU corrupts the values stored in[27]:
  - LUTs
  - Routing
  - On-chip SRAM

[1]Generation of a low-impedance path in CMOS chips between VDD and GND due to the interaction of parasitic PNP and NPN bipolar junction transistors

  – User flip-flops

  If it is the dynamic data that gets corrupted in on-chip SRAM or user flip-flops, then this corrupted value will be overwritten. In case, routing logic or LUT gets affected, then it will not recover on its own. SEU will remain persistent until or unless that part is reprogrammed via on-line or off-line reconfiguration.

- **SET**
  Single event transient (SET) is a voltage pulse in a combinatorial logic, that gets introduced by a single ionizing particle strike. This voltage pulse results in an erroneous data propagating through the circuit. Unlike SEUs, measuring SETs in SRAM based FPGAs is a challenging task. SET cross-section is 10-1000 times smaller than the SEU cross-section of the reconfigurable fabric[27]. The cross-section is a measurement of the sensitivity to SEU or SET from heavy ions and protons. Dealing with SETs is more challenging as it is observed that even in modern SEU immune FPGAs, such as Mircrosemi ProASIC3 and Xilinx Virtex-5QV, the reconfigurable fabric and input/output blocks are SET sensitive[27]. For SRAM based FPGAs, PLLs and multipliers are SET sensitive.

2.3.2 Aging effects

Aging effects can incur in all electronics regardless of the environment in which they operate. The reason these aging effects get more serious attention for long-lasting space missions is that system maintenance or substitution of some faulty component is very difficult and in many cases, not possible at all. Errors induced because of the aging are permanent in nature and their four main causes mentioned in [34] are TDDB, EM, NBTI, and HCE.

- **TDDB**
  Time-dependent dielectric breakdown (TDDB) is reduction in the gate oxide thickness caused by the trapping of charges in the oxide that creates an electric field.

- **EM**
  Electromigration (EM) is development of void in metal lines caused by heavy current densities over a period of time.

- **NBTI**
  Negative bias thermal instability (NBTI) is an increase in threshold and consequent decrease in drain current and trans-conductance in MOSFETs, caused by interface traps and some preexisting traps located in the bulk of dielectric.

- **HCE**
  Hot-carrier effects (HCE) is a creation of traps at the oxide surface, affecting the I-V characteristics of a transistor caused by electrons trapped in the oxide.

These aging effects for a spacecraft mission can be avoided or greatly reduced by carefully selecting the components/FPGAs, which can tolerate such effects throughout the lifetime of the mission. Mitigation and tolerance against such age-related anomalies will not be covered in this thesis and is declared out of scope for this thesis work.

2.4 Fault-tolerance in processors

Microprocessors are very important components for space missions as they control life-support equipment, navigation, on-board processing and data handling. Their failure can have catastrophic consequences. Therefore, before sending them into space, it must be made sure that they can sustain the space environment and operate reliably in it.

2.4.1 Fault mitigation techniques

For a harsh radiation environment like space, mitigation techniques must be employed to protect softcore processors from SEEs. These mitigation approaches can be broadly categorized in two ways: software based and hardware based fault-tolerance.

Software based:
Software-based fault-tolerance techniques provide less expensive and more flexible solutions. They are becoming more popular following the race in satellite miniaturization and trending use of COTS in space applications. In such cases, redundancy in hardware in either not possible or not a feasible option, thus software solutions are the savior in such cases. The main idea behind software-based fault-tolerance approaches is executing the same critical program multiple times and having the results evaluated by some checkpoint mechanism. These executions can either be in parallel to each other or in a sequential manner. Checkpoint to determine correct results of multiple executions is always sequential as it can only operate after results from all the executing elements are received.
A software-based fault-tolerance approach for shared memory multicore platform is presented in [35]. This approach is based on using redundant multithreaded processes to detect soft errors. There are also many other approaches that include securing conditional branches by encoding-based comparison result with the redundancy of control-flow-integrity (CFI) protection mechanism, software fault-tolerance via vectorization, error detection by selective procedure call duplication and multi-stage software solution incorporating various techniques together. [36] [37] [38] [39]. However, the focus of this thesis is on hardware based solutions and software solutions (if needed) can always be included later to further enhance the fault-tolerant capabilities.

Hardware based:
Hardware-based fault-tolerance techniques are based on incorporating additional hardware to detect or correct the errors in the system. Although it incurs additional area overhead, they are becoming more popular and considered more reliable. Hardware redundancy can be static or dynamic: static redundancy means all additional hardware works simultaneously and any error detected is masked immediately, while dynamic redundancy means additional hardware in only activated when current hardware detects an anomaly and starts malfunctioning. Subsequent sections will discuss various hardware fault-tolerance approaches in particular to the softcore processors.

2.4.1.1 Fault-tolerance in pipeline

Pipeline in a processor is the part where most of the data get manipulated. Therefore, protection of this part is a must in order to make a processor fault-tolerant. Two most widely used approaches to introduce redundancy in the pipeline are triple modular redundancy (TMR) and duplication with compare (DWC).

TMR
Triple modular redundancy (TMR), as the name suggests is a technique in which the hardware module is replicated three times. All three modules then execute the same process and their results are fed to a checkpointing module, which then decides the correct value. Checkpointing module usually works on the majority principle, which means that if one out of three modules produces erroneous value, then the checkpointing module will mask the erroneous value and produce a correct result because two modules (out of three) produced correct values. This behavior is also depicted in Table 2.4

Table 2.4: Truth table for checkpoint mechanism

A	**B**	**C**	**Output**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

All the three modules running under TMR must be in strict lockstep. Lockstep means that all the processes run the same program, exactly same instruction sequences, memory loads/stores and interrupts. A strict lockstep needs the processor to run identically on a cycle-by-cycle basis. TMR in processor pipeline can be implemented in twofold ways: either the whole pipeline is triplicated or the flip-flops in the pipeline are triplicated. [40] propose an approach in which the complete pipeline is triplicated and all the signals of the pipelines that could be from/to general purpose registers, data memory, and instruction memory pass through some kind of checkpointing mechanism. On the other hand, [41], as depicted in figure 2.2, propose an approach in which all the flip-flops that are used for pipeline latches/registers are triplicated and after each TMR, outputs pass through checkpointing unit before entering the next pipeline stage.

DWC
Duplication with compare (DWC) is a technique which uses duplication of the module and a checkpointing mechanism to detect upsets. This technique provides relatively less area overhead as compared to TMR, but in addition to just duplication, it also needs a rollback mechanism to take the process back to a previous synchronized stage every

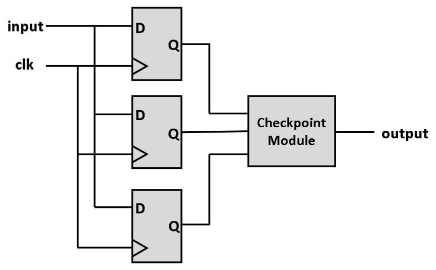


Figure 2.2: TMR for flip-flop

time an upset is detected.
Processes must also be in strict lockstep for this approach. Results from both modules are fed to the checkpointing mechanism which just passes the data across in case both inputs are same, but in case of conflicting outputs, it (temporarily) suspends the further execution of the process. One way to tackle the upset after detection is to reset the processor and restart the program from the beginning. The other way is to reset the state of the processor to a previously stored stable state. This approach provides a better and more efficient solution and is mentioned in [42]. In this, state of the processor is periodically saved when a program executes. State of processor involves state registers, contents of general purpose register files and other memory hierarchies such as caches and main memory. When an anomaly is detected, the rollback mechanism takes the process back to the most recently stored state to recover. If the anomaly is detected more frequently then after suspending the processor execution, FPGA bitstream scrubbing [43] [44] [45] is performed to repair any upsets that may exist in FPGA configuration memory. This periodically saving of state in DWC approach can be expensive in terms of both area and performance.
Another slightly modified DWC approach is mentioned in [46] and depicted in figure 2.3. In this, the checkpoint mechanism is used to compare output signals inside the pipelines that are executing duplicated instructions. These signals include results of arithmetic operations, jump address of a branch or values of memory operations. Whenever an anomaly is detected, a rollback mechanism flushes the pipeline and fetches the uncommitted instruction again. As the anomaly is detected before the memory and register files are modified, flushing the pipeline and executing the last uncommitted instruction works. The only state that needs to be saved in this case is the program counter (PC) value for the last instruction.

2.4.1.2 Fault-tolerance in memory cells

Besides the pipeline, the processor also includes several memory elements that constitute register file, instruction memory, and data memory. These memory elements are often implemented in dynamic random access memory (DRAM), static random access memory (SRAM) or flip-flops (FF). These memories are known as user memories. Besides user memories, FPGAs also have configuration memory that stores the design information. Both of these memories need to be protected against single event ef-

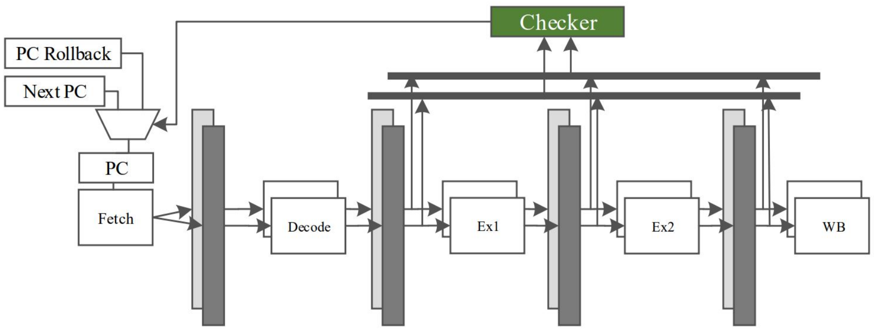


Figure 2.3: DWC with comparison

fects(SEEs). Protection schemes for both kinds of memories will be discussed separately.

User memory
User memory stores the application data that is being processed. Modular redundancy approaches explained earlier for pipeline protection can be used to protect this memory against SEEs. Individual register and flip-flop can be duplicated or triplicated, along with some comparison mechanism to detect and correct errors. However, it is not very economical in terms of area overhead to replicate memory cells twice or thrice along with adding correction logic. [47] shows that TMR is economical for control and datapath circuits or for single registers as in pipeline, but when it comes to a group of registers, caches, and embedded memories, error correcting codes are better options, even though encoders and decoders introduce a performance penalty due to extra delay on the critical path.
Error correcting codes (ECCs) are the most commonly used protection scheme for memories. They are based on the principle of adding extra bits to the data word to form a code word. These extra bits are computed based on the bits in the data word, thus providing redundancy such that errors in the data word can be detected or corrected. Error correcting codes can be categorized into two categories:

- **Block code**
  Information is considered as blocks and these codes are applied on a block-by-block basis. Blocks might be independent of each other.

- **Convolution code**
  Information is considered as a stream and these codes along with current data might also depend on preceding data.

For memory elements of the processor, block codes are usually applied. Convolution codes are difficult/almost impossible to apply as values in register files and data memory are usually independent of each other. Instructions can also be considered mainly independent to each other although there might be some form of dependency between consecutive instructions. Subsequent sections will provide an overview of most commonly used error correcting codes.

- **Hamming code**
  One of the most well-known error correcting code is Hamming code, that was introduced by R. W. Hamming in 1950 [48]. In his paper, three types of codes namely single error detecting (SED), single error correcting (SEC)and single error correcting, double error detecting (SECDED) were introduced.

  - **Single error detecting (SED) code**
    A single parity bit is added at the end of the data word. This parity bit is decided based on the concept to keep the number of 1's in the data word even. For example, if the number of 1's in the data word are odd, then the parity bit will be 1, otherwise, it will be 0. At the decoding side, the number of 1's are counted again, if they are even, it means data word is correctly received. Reception of an odd number of 1's implies an error.
  - **Single error correcting (SEC) code**
    These codes can correct one bit error in a dataword. For k bit data word, an additional m parity bits are added to make n bit code word where n and k must follow the inequality:

    $$2^k <= \frac{2^n}{1+n}$$

    If a code word is numbered from the least significant bit to the most significant bit as *1* to *n*, then parity bits are placed at the positions with the index equal to the power of 2, i.e., at positions 0,2,4,8 and so on. Rest of the bits are filled with data word bits in the same order in which they appear. Parity bit number t is computed by parity checks of all positions in the code which have a 1 at the position t of the binary index. At the decoding side, parity checks are computed for the complete code word. It must be noted that at decoding side, parity checks also apply on parity bits that were added at encoding side. If there is no error, then all parity checks should give a value *0*. If there is an error then at least one parity check value will be 1. Based on the non-zero parity check value(s), the position of the erroneous bit is located, which is then flipped to correct the error. Mathematically, for any positive integer, m, the SEC code parameters are presented in Table 2.5.

    Table 2.5: SEC code parameters

Parameter	**Equation**
Code length, n	$n = 2^m - 1$
Number of parity-check digits	$n - k = m$
Hamming distance	$d_{min} \geq 4$

  - **Single error correcting, double error detecting (SECDED) code**
    These codes are an extension to SEC codes mentioned in the earlier section. It involves the addition of another even parity check on all the previous bits. These are known as modified or extended Hamming codes, while SECs are

known as conventional Hamming codes. At the decoding side, there can be either one of three cases:

* All parity checks are satisfied, indicating no error has occurred.
* Parity checks based on both SEC and additional parity check fail, indicating there is one error.
* Parity checks based on SEC fails, but additional parity check over all previous bits succeeds (or vice versa), indicating there is a double error in the code word.

Note that, in the presence of more than two errors in a single codeword, these codes will not be helpful. In fact, they can misguide by staying silent or by giving false indications of single error (hence it will be miscorrected) or double error. For every positive integer m , the SECDED code parameter are listed in Table 2.6[49] .

Table 2.6: SECDED / Hsiao code parameters

Parameter	**Equation**
Code length, n	$n = 2^m$
Number of parity-check digits	$n - k = m$
Hamming distance	$d_{min} \geq 3$

- **Hsiao code**
  This class of code also belongs to single error correction, double error detection (SECDED) category and was introduced by M.Y. Hsiao in 1970 [50]. Along with Hamming codes, these codes are also very popular for use in embedded memories. For Hamming codes, the encoding and decoding procedures are not very optimal. Hsiao codes are based on the same principle as Hamming codes and provide a better approach for implementation of encoder and decoder. The generator and parity check matrices are constructed differently in this class of code and follow the below mentioned constraints [50]:

  – There are no all 0 columns.
  – Every column is distinct.
  – Each column contains an odd number of 1's.

  These constraints ensure a minimum number of 1's in the rows of parity check matrix, leading to a faster generation of check bits. This rapid generation of check bits leads to better performance of the encoding and decoding mechanism. The parameters of Hsiao code are the same as of Hamming SECDED codes, mentioned in Table 2.6.

- **BCH code**
  BCH codes were initially discovered by Hocquenghem in 1959 and subsequently by Bose and Chaudhuri in 1960 [51] [52]. This class of code possesses a higher

level of error detection and correction capabilities than the SEDDED based codes discussed earlier. SEDDED codes realize a maximum Hamming distance [2] of four, while BCH codes theoretically cover all Hamming distances.
BCH codes are a more generalized form of Hamming codes with the ability to correct multiple bit errors. The generator polynomial g(X)of BCH code is defined as:

$$g(X) = LCM[\phi_1(X), \phi_3(X), \phi_5(X), ..., \phi_{2t-1}(X)]$$

where LCM is least common multiple and ϕ_i(X) is the minimal polynomial of some element α_j, which are primitive elements of the Galois field $GF(2^m)$ [53]. A Galois field has the property that arithmetic operations on field elements always have a result in the field. For any positive integers m ($m >= 3$) and t ($t < 2^{m-1}$), the binary BCH code parameters are listed in Table 2.7[49]. The further details of constructing an arbitrary BCH code and the mathematical theory behind it will not be discussed here. Although efficient decoding methods exist due to the special algebraic structure's involvement in the codes [54], the implementation of these codes is considered too complicated and time-consuming in the context of this thesis project.

Table 2.7: BCH code parameters

Parameter	**Equation**
Code length, n	$n = 2^m - 1$
Number of parity-check digits	$n - k \leq mt$
Hamming distance	$d_{min} \geq 2t + 1$

- **RS code**
  Reed Solomon (RS) codes were proposed by Irving S. Reed and Gustave Solomon in the year 1960 [55] and are a special example of a more generalized class of BCH codes. These codes are block-based error correcting codes and have a wide range of applications (CD's, DVD's, etc). The RS codes operate on a block of data treated as a set of finite field elements called symbols. These codes are able to detect and correct multiple symbol errors and are usually preferred for multiple-burst bit-error correcting.
  RS codes are also based on Galois fields (GF). The RS code defined with symbols from GF(q), and a positive integer m, has the parameters mentioned in Table 2.8[49]:

Configuration memory
This memory stores the bitstream that defines the functionality of the underlying FPGA device. Errors within the configuration memory are especially troublesome as they may

[2]number of places at which two codewords of similar length differs

Table 2.8: RS code parameters

Parameter	**Equation**
Code length, n	$n = q - 1$
Number of parity-check digits	$n - k = 2m$
Hamming distance	$d_{min} = 2t + 1$

change the functionality of the device. Upsets may alter the functions of the configurable logic blocks, routing network, and input/output blocks. Moreover, errors in the configuration are not transient, they are permanent in nature. In the literature, many techniques have been proposed and tested for the mitigation of SEEs in the configuration memory. Few techniques are based on modular redundancy to reduce the probability of failure [56]. Replication of hardware and comparison of results, as explained earlier in Section 2.4.1.1, is used. However, in practice, not all SEEs in configuration memory can be mitigated by TMR [57]. Errors in configuration memory may accumulate and eventually lead to multiple faults breaking the redundancy protection.

An alternate approach that is widely used to halt error accumulation in configuration memory is scrubbing. It involves a periodic refresh of memory data while FPGA is operational. Extra golden copy of configuration data is stored on a radiation-hardened platform that might be ASIC or anti-fuse based FPGA. This golden copy can either be exact complete data or a golden configuration check code (ECC). To perform memory scrubbing, the configuration data is usually read sequentially from start to end. A discrete block of memory data is read and checked against respective golden data or golden check code. If any discrepancy is found, the discrete block in configuration memory is replaced by the data from the golden copy. In case, no discrepancy is found, scrubber moves to the next discrete block in memory. When the scrubber reaches the end of configuration memory data, the same process is repeated from the beginning. To preserve the configuration data, reading of data and correction of upsets (if any) is performed indefinitely.

There are many variations of this scrubbing techniques. Most commonly known are blind scrubbing and partial scrubbing. Blind scrubbing is the fastest implementation of scrubbing. In this, configuration data is overwritten with the golden data continuously, without reading and crosschecking the configuration memory data for possible upsets. It is favorable for the devices operating in an environment with higher upset rates, as it is the quickest method [58]. Partial scrubbing, on the other hand, is a technique in which faulty module in the design is identified via unit-level TMR and repaired with the scrubbing of only affected configuration data using dynamic partial reconfiguration [59].

2.4.2 Fault injection

After fault mitigation techniques are implemented in a soft-core processor, it must be rigorously tested and verified to validate that core behaves as intended and can mitigate errors. One way to validate such a system is by injecting the faults into it and observing the behavior of the system. Fault injection helps the designer to debug and fine-tune the design before it is actually operated in the real environment. [60] classifies the fault injection in three categories:

Hardware-based fault injection
There are many hardware-based fault injection techniques. One way is to disturb the signals on IC pins. The signals can be controlled by a general purpose fault inserter [61] and the value of signals can be changed randomly or on some pre-defined pattern. This approach provides a good control on fault insertions and can imitate different kinds of errors, however, it does not provide any control over the internal signals of the chip.
The other way to introduce errors in the system is by disturbing the power supply. This approach is usually used to model power surges and disturbances common in industrial applications. However, this can also be used in addition to other fault injection techniques to analyze the fault-tolerant system. [62] and [63] used this method in addition with high-ion radiation on a MC6809 processor. MOS power transistor was used to cause short voltage drops at the power supply pin of the CPU. A test CPU and reference CPU were run in lockstep and external buses were compared to analyze the fault-tolerance of test CPU.
A more realistic approach is to simulate a space environment, by placing the device under a heavy-ion radiation beam. Any angle of incidence can be used but usually, beams are targeted perpendicular (at 90 degrees) to achieve maximum penetration. This approach is closest to the real environment as compared to other approaches, however, the issue with this approach is that developing such a facility is very expensive and usually designers and researches do not have access to such facilities.

Software-based fault injection
As the name suggests, fault injections are done entirely in software. The idea is to reproduce such faults at software level that would have occurred in case of upsets in hardware. [64] describes the methodologies and guidelines for developing a flexible software base fault injector. A software tool *FERRARI* is also been introduced that emulates transient and permanent errors. However, these faults provide only a limited coverage and do not cover all possible faults that could occur in hardware.

Simulation-based fault injection
Simulation of the system under test is performed on some other computer system, and logical values of signals are altered to emulate faults. These logical values can be altered directly via simulator platform or source code can be modified using hardware description language (HDL) to introduce errors. [60] categorizes VHDL based fault injection techniques into simulator commands and VHDL code modifications. Simulator commands will only be used in this thesis work to verify intermediate results, while for formal and more rigorous fault injections, VHDL code modifications will be used. To accomplish this, there are two popular options presented in [60], mutants and saboteurs.

- **Mutant** is a component that is added to replace some other component. When inactive, it works similar to the component which is replaced, but when activated, it works as a faulty component. The characteristics of the faults are tuned using VHDL such that it imitates radiation-induced faults. Mutants can be generated by modifying behavioral descriptions or synchronization and timing clauses.

- **Saboteur** is a dedicated fault-injection component with the aim to alter the value or timing characteristics of one or more signals when activated, and when deactivated, it just passes the same values across without manipulating them. Based on the way saboteurs are implemented, they could be classified as serial saboteurs or parallel saboteurs. Serial saboteur is placed between the driver and corresponding receptor of the signal, while parallel saboteurs are added as an additional driver for a particular signal.

2.5 Conclusion

In this chapter, the required background information for making the ρ-VEX processor fault-tolerant for the space environment is presented. First, the ρ-VEX platform including its design, distinguishing features and working has been explored, followed by the study of space environment and its effects on electronics. Afterwards, fault mitigation techniques for processor found in the literature are explored. These include redundancy based TMR and DWC techniques for processor pipeline stage and error correcting codes and scrubbing for memory elements. Finally, the verification techniques for the fault-tolerant systems are explored with the focus on fault injection. Different types including hardware-based, software-based and simulation-based fault injections are explored.

Design 3

In Chapter 2, space environment and its impacts on electronics systems were discussed. Various implementation schemes to make softcore processors fault-resilient were also presented. This chapter provides details and rationale for the design strategy used to achieve the goals of this thesis. Based on the background information, the baseline processor platform is thoroughly evaluated to identify the vulnerabilities in it and a fault model for the ρ-VEX is constructed in Section 3.3. Following that, in Section 3.4 efficient and best-suited fault mitigation techniques from the pool of available techniques are shortlisted for our platform. But, before discussing all the design decisions, the architecture of the baseline processor platform will be discussed in Section 3.1.

3.1 Baseline processor platform

The latest release of ρ-VEX processor is the starting point for this thesis. This release does not come with any inherent fault mitigation techniques and is, therefore, very defenseless for the space environment. Before adapting this processor to a fault-tolerant processor, it is important to first look at its architectural level details.

3.1.1 Processor architecture

The simplified block diagram of ρ-VEX with caches is given in Figure 3.1. The default version of processor contains eight five-stage pipelines. However, for simplicity and more clarification, only one pipeline is shown in the figure.
The brief description of all the pipeline stages and other main blocks present in the architecture is as follows:

- **Instruction fetch** is in charge of requesting instructions from the instruction cache and routing the fetched syllable to the appropriate pipelanes.
- **Instruction decode** is responsible for decoding instruction syllables into control signals for functional units and data paths.
- **Execute** stage holds functional units such as ALUs and multipliers.
- **Memory** stage allows read and write operations to be performed on memory and control registers.
- **Writeback** stage is responsible for writing values to general purpose register files.
- **Instruction cache** is an on-chip reconfigurable buffer memory structure that lies between the external instruction memory and core pipeline to shorten the instruction fetch time.

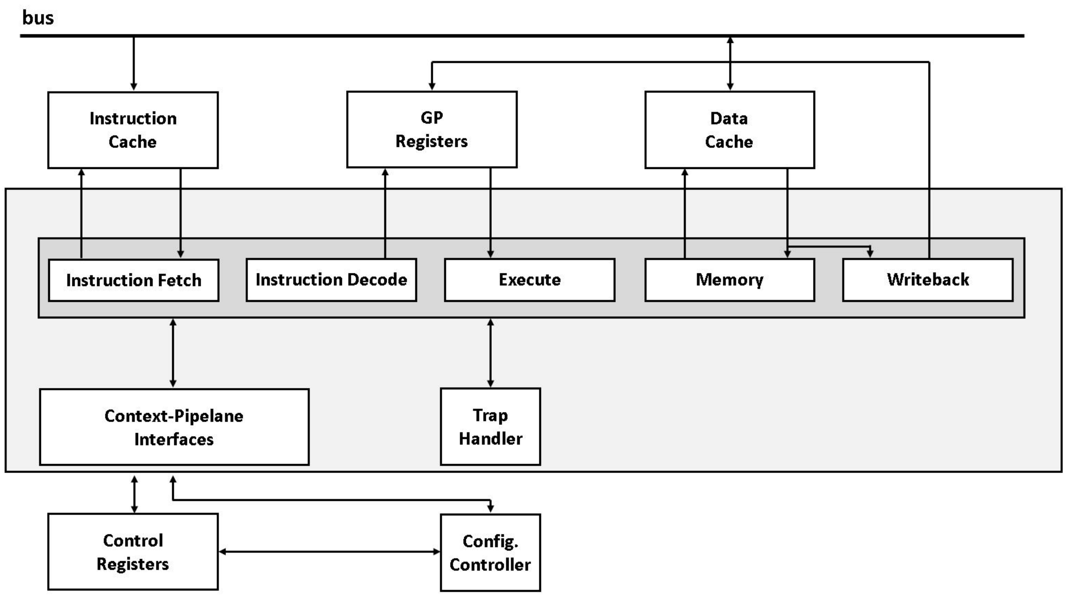


Figure 3.1: Block level overview of ρ-VEX along with caches

- **Data cache** is an on-chip reconfigurable buffer memory structure that lies between the external data memory and core pipeline to shorten the data access time.
- **GP registers** are quickly accessible registers that store transient data required by the running program.
- **Context-pipelane interface** connects the pipelane-based resources with the context resources based on the current configuration.
- **Trap Handler** handles pipeline stage invalidation if a trap occurs and ensures that the right trap information is forwarded to the branch unit and control registers in case of multiple simultaneous traps.
- **Control registers** holds information such as program counter, configuration vector and other context-specific details.
- **Configuration controller** arbitrates between the incoming reconfiguration requests, and synchronize the running contexts that get affected by the reconfiguration.

3.1.2 Design cost

It is important to have a design cost estimate for the baseline ρ-VEX core before applying fault mitigation techniques to it. Fault-tolerance is an expensive feature and it will introduce more hardware resources in the design. In order to get a fair estimate of the cost at the end of our design, we must first know the design cost of our baseline core. To find that, synthesis of the baseline processor is done by *Xilinx ISE Design Suite* software and retrieved results are listed in Table 3.1.

Table 3.1: Synthesis results for the baseline ρ-VEX softcore

Resource	**Value**
Slices	23989
Registers	27179
LUTs	65841
DSPs	32
BRAMs	345
f_{max}	37.5 MHz

3.2 Previous work

An effort was made previously to make ρ-VEX fault-resilient for the space environment [40]. That work is thoroughly analyzed and few shortcomings are identified.

1. Missing Features
   - No fault-tolerance for memory elements.
   - Only context 0 could run in the fault-tolerant mode. There was no provision to run either of other contexts in the fault-tolerant mode.
   - No provision to run a second context in parallel to the fault-tolerant context.
   - Was implemented on a ρ-VEX version which didn't include caches in it.
2. Design Flaws & Drawbacks
   - Not enough coverage was provided. Many critical components were not made robust, e.g., configuration controller and context-pipeline interfaces
   - Use of less reliable voter
   - From the fault-tolerant mode, it was not possible to switch back to normal mode
   - Undesirable delays were introduced in the design.
   - Triple modular redundancy (TMR) was implemented at the cost of four lane-groups. All four lanegroups needed to be in active mode.

After analyzing the existing design, it is decided to not take it as a starting point. Although some features and ideas from it will be adopted in our design, the entire code will be rewritten taking the latest release of ρ-VEX as the starting point.

3.3 Fault model

Section 2.3.1 discussed the impacts of radiations on the electronics. Based on these impacts, ρ-VEX is carefully evaluated to develop a fault model, depicted in Figure 3.2. The fault model shows that which parts of the core are susceptible to which kind of single event effects. Orange color blocks represent susceptibility to single event upsets (SEUs), while brown color blocks represent susceptibility to single event transients (SETs). These fault types and their locations will be targeted by the fault-tolerant design being implemented in this thesis project.

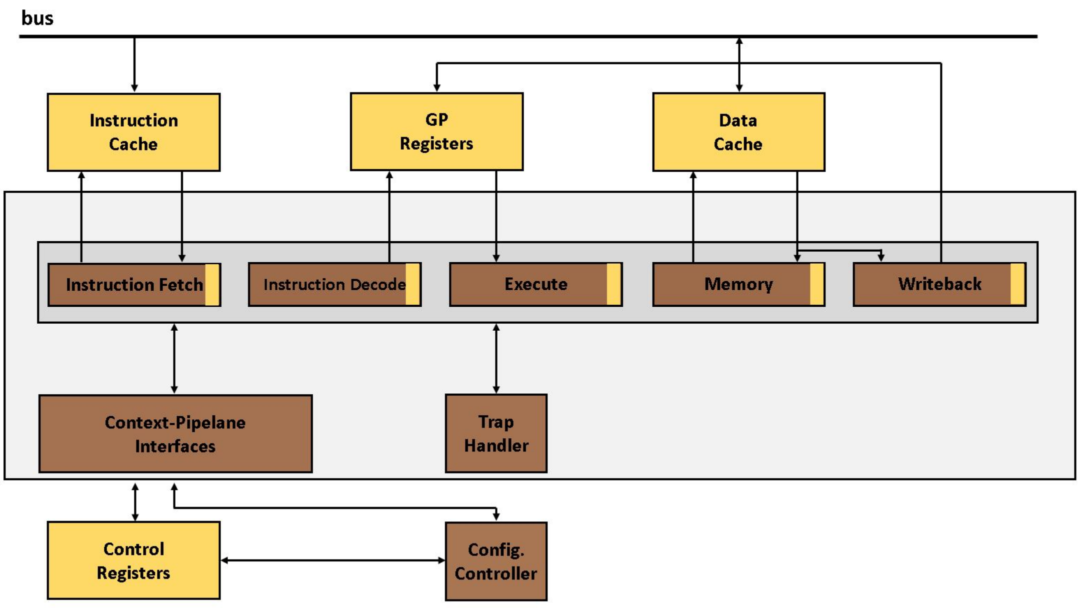


Figure 3.2: Block level diagram indicating susceptibility to radiations (yellow color shows susceptibility to SEUs, while brown color shows susceptibility to SETs)

3.4 Design options

Various techniques for fault mitigation in softcore processors were discussed in Section 2.4.1. This thesis will consider only hardware-based fault-tolerant techniques, and software-based solutions are declared out of scope for this thesis. However, if needed, software solutions can always be included in the future to further enhance the fault-tolerant capabilities. Following sections will discuss hardware-based design options for both processor pipeline and memory elements.

3.4.1 Design of fault-tolerance in pipeline

To implement fault-tolerance at the pipeline level, various options are carefully evaluated before finalizing the design. This sections will provide details about the decisions made and, the justifications and rationale behind them.

Redundancy level

Redundancy in a pipeline can be envisioned in two ways. One way is to do it at lower level, i.e., inside the pipeline at multiple locations as mentioned in [46] [65]. Individual components are replicated and a voting mechanism is implemented at each pipeline

stage. The other way of redundancy is to do it at a higher level, i.e., replicate the entire pipeline and implement voting mechanism on all the signals entering into or going out of the pipeline. The first approach detects faults faster as compared to the second approach. For example, if an upset occurs in the instruction-decode stage, the first approach will detect this anomaly in the same clock cycle, while the second approach will detect it when the effect of this approach propagates out of the pipeline; that can be in various forms such as wrong instruction fetch command or wrong data memory access. For TMR, it does not matter in terms of performance whether lower-level redundancy approach is used or higher level. TMR is a passive approach and it masks the error immediately in the same cycle as soon as it propagates through the voting mechanism. However, in the case of DWC, when an upset is detected, the rollback mechanism flushes the pipeline and redo the instruction fetch from the previously stored stable state. Therefore, the sooner the upset gets detected, the sooner the correction process starts. If it gets detected at the end of a pipeline stage as in the case of higher level redundancy approach, the correction process will start few cycles after the upset occurrence (if an upset occurred at the earlier stage of the pipeline) and thus incur relatively more performance degradation. Moreover, low-level redundancy approach also consumes more hardware as it needs a voting mechanism to be implemented at multiple locations inside the pipeline.
The ρ-VEX core provides eight pipelanes in its default configuration. This implies that implementation of redundancy at the pipelane level is more appropriate, economical and better suited for our platform. If we consider implementing low-level redundancy then it requires an addition of much more hardware to ultimately achieve the same thing which we can do by exploiting the already existing pipelanes. Therefore, it is decided to exploit higher-level redundancy, i.e., redundancy at the entire pipeline level.

DWC vs TMR
The decision is made that redundancy will be at the pipeline level, now we have to decide whether duplication with compare (DWC) should be implemented or triple modular redundancy (TMR). Section 2.4.1.1 provides detail of both approaches and it can be inferred that the selection of a duplication approach results in a trade-off between performance and additional hardware cost.
DWC is the approach that comes with low redundancy requirements but incurs more performance degradation. This additional performance degradation arises because of the rollback mechanism which requires the core to start execution from some previous stable state, each time an upset is detected. Its implementation is also relatively complicated (because of the rollback mechanism).[46] provides implementation details of DWC on ρ-VEX. Every-time an upset is detected, core flushes the pipeline and starts executions from some previously saved state. This correction process comes at the cost of few clock cycles, and the number of this additional clock cycles depends on the implementation approach and location of fault occurrence. On the other hand, triple modular redundancy (TMR) does not incur performance degradation in terms of the number of clock cycles, but it comes with relatively more additional hardware requirements (as it requires modules to be replicated three times). Whenever an upset gets detected by voting mechanism, it gets masked immediately without rolling back

the execution process to some earlier stage.
As mentioned earlier, our ρ-VEX platform contains eight pipelanes and we plan to exploit existing pipelanes to achieve redundancy, we should be indifferent about additional hardware cost while deciding between DWC and TMR. The other criterion for decision making besides additional hardware cost is performance. Considering that TMR provides less performance degradation as compared to DWC, TMR will be implemented in our design. The flow graph presented in Figure 3.3 highlights the decisions (in green color) which we took regarding redundancy at the pipeline level.

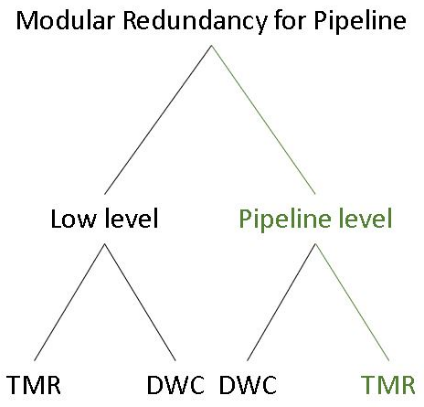


Figure 3.3: Flow graph showing modular redundancy options at the pipeline level

Realization of redundancy
It must be noted that redundancy in ρ-VEX pipelane cannot be envisioned in a similar manner as it is done in a normal RISC processor having just one pipeline. ρ-VEX is a VLIW based processor and we can not run a context on a single pipelane. At least two pipelanes are required to run a context. To get a clear picture of pipelanes behavior, consider the Figure 3.4. Bold border lines indicate that the pipelane pair is inseparable. It must also be noted that these inseparable pipelane pair does not include identical pipelanes. For example, in a pair, only one pipelane has the capability to access the data cache. Therefore, special attention must be paid while implementing TMR checkpoints among pipelanes.

For implementing TMR, we need three lane pairs (six pipelanes) out of four lane pairs (eight pipelanes). Lane pairs are represented in Figure 3.4 as A, B, C and D. Now for the three lane pairs under TMR, first pipelane of each lane pair must be compared with the first pipelanes of other lane pairs and similarly, the second pipelane must be compared with the second pipelanes of other lane pairs. Consider a scenario that A, B, and D are selected for TMR mode. One TMR check system will be implemented for pipelanes numbered as *i*, *iii* and *vii*, and another TMR check system will be implemented for pipelanes numbered as *ii*,*iv* and *viii*.

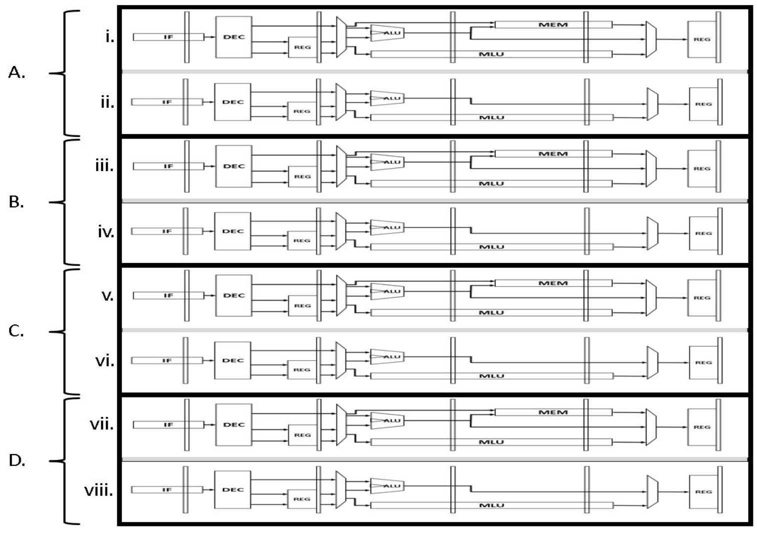


Figure 3.4: ρ-VEX pilelanes

3.4.1.1 TMR design checkpoints

The decision is made that TMR will be implemented at pipelane level and it means that all incoming and outgoing signals will have to pass through some kind of TMR checks. In reference to Figure 3.1, this section will present details about the locations and types of checkpoints.

Pipelane-instruction cache interface
Three lanegroups in TMR mode will run in strict lockstep mode. It implies that all three will perform the same operations on a cycle-by-cycle basis. All three lanegroups will send the same instruction fetch commands to the instruction cache and expect the same data simultaneously. However, the cache can only respond to one request at a time. Thus out of three lanegroups, one lanegroup who wins arbitration over others will receive the instruction word in the same cycle, while other two lanegroups will be stalled. After responding to a request from the first lanegroup, the remaining two lanegroups will be handled subsequently. This behavior will cause the TMR lanegroups to lose strict lockstep mode and start behaving haphazardly.
To address this issue, it is decided that only one lanegroup will send request to the instruction cache. All the signals that are intended to go from all three TMR lanegroups to instruction cache will pass through a voter. This voter compares the information from all three sources and produces one output that is selected based on the majority principle. This voter helps in resolving two issues at hand. First, it helps in merging three requests to one request, resulting in no stall and no divergence from strict lockstep mode. Secondly, it also helps in masking errors. If one pipelane gets corrupted by SEE then its effect will not be propagated further and will be masked by a majority voter.
For the instruction word that is intended from Instruction cache to lanegroup, we need a replication unit. As after majority voter, only one request is sent to the instruction

cache, the cache will respond to only that. Now we need this instruction word to be distributed to all three lanegroups running under TMR. To achieve this, we need a replication unit that will replicate the instruction three times and then each lanegroup will be assigned a copy.

Pipelane - data cache interface
Data cache interface will also follow the same reasoning as provided for instruction cache interface. All the data read commands sent from the TMR lanegroups will pass through a majority voter to avoid stalling and mask errors(if any). Data from data cache will pass through a replication unit before entering the TMR lanegroups.
However, in contrary to the instruction cache, langroups can also write data to the data cache. All such data and their addresses will also pass through a majority voter and only one lane will write to the data cache.

Pipelane - GP registers interface
General-purpose registers store transient data for the running contexts, and lanegroups access them for both writing to and reading from them. The implementation of general-purpose registers for ρ-VEX is very complex as two read ports and one write port is implemented for each lane. For the full ρ-VEX it means that 16 read ports and 8 write ports are implemented. Having this many ports helps us in the implementation of fault-tolerance, as read requests to and read data from registers do not need to be passed through a majority voter or a replication unit. All lanes running under TMR can request and read data simultaneously without causing any stalls.
However, multiple write ports do not help us much. In the baseline implementation of the core, if multiple lanes try to write data at the same address, then the lane with highest index number wins arbitration over others and writes to the register file. We do not want this behavior, as under radiation environment if the lane with the highest index gets corrupted then as per the baseline implementation, an erroneous value will be written to the register file. To resolve this issue, we need a majority voter that will vote on all write requests and data, and output of this voter will then be written to the register file.

Trap hander
When any kind of trap occurs, syllables in all pipeline stages up to and including the one in which trap occurs get invalidated. They can not commit to the register file or memories anymore and control is handed over to trap handler. In the fault-tolerant mode, when three lanegroups are running in lockstep mode, an anomaly in anyone lane could trigger a trap. This will result in halting the execution of the corrupted lanegroup, while the other two remaining lanegroups under TMR will continue their executions. At this moment, the core will keep executing the context correctly because all the majority voters implemented at various instances in the core will get two correct values and one faulty value, so they will produce correct value. The problem is that the faulty lanegroup has lost the lockstep mode and although the context is running correctly at the moment, the core will not be able to handle any more error in the two correctly running lane groups. Any errors in these lanes will result in a faulty behavior

of core.
Thus trap hander is considered as a sensitive component and it is decided that all signals to and from trap handler will pass through a majority voter. In this case, even if faulty lanegroup tries to trigger trap hander, it won't be able to do so because the fault will be masked be majority voter and the transient error in the faulty lane will fade away on its own, as all the incoming signals are coming after being checked by a majority voter.

Context-pipelane interface
The context-pipelane interface handles the vast majority of the reconfigurable interconnect between the lanes and contexts. Under the fault-tolerant mode, all the lanegroups running under TMR will access the context resources simultaneously. To make this interface fault tolerant, the same solution is applied as is for general-purpose registers and data cache. All the signals going from lanegroups to context resources will pass through a majority voter, and on the other way round, a replication unit is needed for all the signals coming from context resources to lanegroups.

Configuration controller
This controller can be considered as the brain of all reconfiguration logic. It deals with the incoming reconfiguration requests, decode them and handles arbitration among multiple requests. It is also responsible for synchronizing the running contexts that are affected by the reconfiguration before reconfiguring the core. Based on its functionality, it is deemed as a highly critical component as any anomaly might cause loss of synchronization, and in the worst case might even reconfigure the core. All the efforts being put in to make the core fault-tolerant will go in vain if an upset in the controller is able to reconfigure the core from the fault-tolerant mode to any other non-fault-tolerant mode.
Considering the vulnerability of this component and its possible impacts, it is decided to triplicate this unit. It is relatively a small component in terms of area consumption, thus it will not incur a very high design cost. All the signals from these three configuration controllers will pass through a majority voter before entering into lanes and control registers. All the incoming signals previously intended for one controller will pass through a replication unit, which will replicate them three times and send a copy to all the three configuration controllers.

Program counter
This unit is not shown explicitly in the block diagram of ρ-VEX core. It lies in the first pipeline stage. Although a decision was made that components and executions inside the pipeline will not be focused on and TMR will be implemented at a higher level, i.e., on all incoming and outgoing signals of the pipeline, this PC unit requires special attention. All the lanegroups under TMR have their own units that compute the next PC value, and in the normal case (no fault), all units produce the same PC value. These PC values then pass through a majority voter and corresponding instruction is fetched. The idea is that the majority voter masks the error and execution does not get disturbed, and as the error is transient in nature, it fades away for the later execution instances. However, if one PC unit gets corrupted due to SEE, then it will not be fixed

on its own. As the new PC value is just an increment on the current PC value (not in case of jump or branch), corrupted PC unit will keep on computing wrong PC value. At this moment, the core will be executing the context correctly, as for the instruction fetch PCs value will pass through the majority voter which will produce correct results as two redundant PC units under TMR are working correctly. The core will keep working correctly unless single event effect occurs in PCs unit of the other two cores. In that case, majority voter will produce false PC value resulting in a trap initialization or faulty execution of context.
To resolve this issue, outputs of all PC units running under TMR will pass through a majority voter before computation of next PC value. After this solution, if upsets occur in one PC value then it will have its impact on instruction fetch command in only that clock cycle (which will be masked by majority voter) and the fault will not sustain as computations for next PC will rely on the output of the majority voter.

3.4.2 Design of fault-tolerance in memory elements

As discussed earlier, for softcore processors running on SRAM based FPGA, two type of memories require attention; user memory and configuration memory.

3.4.2.1 User memory

Memory elements present in Fig 3.2 are instruction cache, data cache, general-purpose registers and control registers. These memory elements are susceptible to single event upsets (SEUs) and as was discussed in Section 2.4.1.2, error correcting codes will be used to make them resilient against upsets.
Different types of codes discussed in Chapter 2 are RS, BCH, Hsiao, and Hamming. A comparative analysis of these error correcting codes and TMR is done and presented in the form of a score matrix in Table 3.2. TMR provides the best error correcting capabilities as it can correct up to n errors in a n-bit word as long as the errors are located in a distinct position/unit and it is also comparatively easier to implement. However, this approach comes with a major drawback and that is resource utilization. As compared to the baseline design of a particular component, this TMR approach requires more than 200% [1] of additional area. RS and BCH provides better error handling capabilities as they provide the provision to correct multiple-bit upsets in a single data word. The additional area they require varies depending on the number of bits and can be in the range of 13-75 % [1]. The drawback of these codes is that they are complex to decode and implement in hardware,and also incurs a negative effect on system clock resulting in performance degradation. Hamming and Hsiao, both codes are favorable, as for single error correction, they incur a minimum cost to the system. The additional area they require depends on the number of bits and can be in the range of 7-32 % [1]. The implementation complexity of these codes is lower as compared to RS and BCH.
Hsiao belongs to the SECDED class and works on the same principle as Hamming codes, but provides a better approach for implementing encoders and decoders. Although Hsiao codes provide slightly better performance than SECDED Hamming codes, we will implement Hamming codes in our design. We need both SEC and SECDED classes in

our code (as will be explained in a later section) and Hamming codes provides provision for both. Therefore, to keep the design consistent and reuse the same encoders and decoders for SEC and SECDED after minor modifications, Hamming codes are finalized for implementation.

Table 3.2: Comparison of TMR and ECC schemes (adopted from [1])

Characteristic	**Area**	**Performance**	**Error Correction**	**Implementation**
TMR	- -	++	++	++
Hamming (SEC)	++	+	+	+
Hamming (SECDED)	+	+	+	+
Hsiao	+	++	+	+/-
RS(DEC-TED)	-	+/-	++	- -
BCH(DEC-TED)	-	-	++	- -

General-purpose registers
These are 32-bit registers files and to provide one error detection and correction ability per word, SEC Hamming codes will be implemented per 32-bit word. All the data intended to be written to general-purpose registers in the writeback stage of the pipeline will be first encoded by Hamming code encoder and then forwarded to general purpose registers. The size of general purpose registers will also be increased in accordance with the specifications of the Hamming code used. In the execute stage of pipeline, before using data from general-purpose registers, it will be first decoded back to the 32-bit word.

Instruction cache
Instruction syllables in ρ-VEX are encoded as 32-bit words and instead of SEC, SECDED codes will be implemented per syllable. The reason for using SECDED over SEC will be discussed in a later section. Instructions coming from external instruction memory will be encoded first before writing them to the instruction cache. And subsequently, before executing the instructions, these will be decoded in the instruction fetch stage of the pipeline.

Data cache
In case of data cache, there is a provision to perform read or write operations per 8-bit of data. Therefore, applying error correction codes per 32-bit word is not a favorable option, hence, SECDED Hamming codes will be applied per 8-bit data word. All the data being written to data cache by the memory stage of pipeline or by external data memory will be encoded. The data must be, therefore, decoded before being used by pipeline or written to the external data memory.

Increased cache protection
For the protection of memory elements, we are targeting single error correction. Although SEC codes can achieve this target, we have chosen SECDED code for caches because these codes besides correcting single error, provide the ability to detect two

errors as well. We will use this double error detection attribute and also exploit the underlying cache functionality to enable it to correct two errors as well. It is decided that in case of dual error detection, cache invalidation of the corrupted data will be done. Corresponding data will be fetched again from external memory before sending it to the pipeline. Although this new design feature will cost us some performance degradation in the form of cache miss penalty, we will be able to achieve increased resilience against radiation-induced anomalies.

Control registers
These registers are special-purpose registers and can be classified into global and context control registers. Instead of designing them in VHDL, a script is written to auto-generate these control registers and keep them synchronized with the documentation. The implementation of these control registers does not follow a regular memory architecture such as BRAM, instead, they are implemented in general-purpose FPGA fabric. If these registers were implemented using BRAM, error correction codes were a suitable protection scheme for them. Now, as these are made using logic elements and does not follow a regular structure, error correcting codes cannot be applied to them.
Considering their nature, there are two options possible to make them fault-resilient. One way is to triplicate all the control registers and apply TMR or DWC on them. This approach is not very tempting as it will incur huge area overhead as control registers occupy a significant amount of area in the core. The other approach is to re-design all the control registers such that they are implemented in a regular memory fashion, e.g., using BRAM and then implement error correcting codes. Considering the time constraints for this thesis, re-designing the entire control registers module and then implementing fault-tolerance is not possible. Therefore, it is left as a future work and declared out of the scope of this thesis.

3.4.2.2 Configuration memory

Upsets in the configuration memory can modify the FPGA design and alter its functionality. Among all type of FPGAs available in the market, SRAM based FPGAs are most susceptible to it. As discussed in Section 2.4.1.2, scrubbing of configuration memory is usually recommended for its fault resilience, be it complete scrubbing or partial scrubbing. To an extent, upsets in the configuration memory of our design will be covered by fault resilience of pipelines and error correction in user memories. However, a dedicated protection scheme for configuration memory such as scrubbing will not be implemented in this work. Implementing scrubbing along with all the other design features which we have discussed earlier is too time demanding. Therefore, considering the timing constraints and quantity of workload, it is decided to declare implementation of fault-tolerance in configuration memory out of the scope of this thesis and is left for the future work.

3.4.3 Summary of design decisions

Section 3.4.1 and 3.4.2 evaluates various design options for implementing fault tolerance in pipeline and memory elements. The decisions taken in this regard are summarized as

follows:
For pipeline stage, TMR is selected as a modular redundancy option and it will be implemented at a higher level, i.e., over the entire pipeline and all the signals entering into or going out of the pipeline will pass through TMR checkpoints. The TMR will be implemented for the smallest possible configuration of ρ-VEX, i.e., 2-way. This means that we will be using three lane pairs (6 pipelanes) out of four lane pairs (8 pipelanes) to implement TMR at the pipelane level. The interfaces of the pipelane with the instruction cache, data cache, GP registers and context resources will include TMR checkpoints. Besides these interfaces, TMR will also be implemented for trap handler, configuration controller, and program counter logic.
For memories, Hamming codes are selected as an ECC encoding scheme. SEC codes will be used for protection of general-purpose registers and SECDED codes will be used for caches. For general-purpose registers and instruction cache, encoding will be applied per 32-bit word, while for data cache it will be applied per 8-bit word. For caches, an additional protection scheme to correct two errors per word will also be implemented. Dedicated protection of control registers and configuration memory will not be implemented in this work and is left for the future work.

3.5 Reconfiguration

This section will describe how the fault-tolerant mode will be added to the reconfigurable ρ-VEX processor. One way is to make this feature design-time configurable, which means that before synthesizing and generating a bitstream, it must be decided whether we want a normal core or a fault-tolerant core. Once configured, the design cannot be switched to another mode unless new bitstream is generated and loaded to the FPGA. The other way of implementation is to make this fault-tolerant mode dynamically configurable, which means that this mode can be triggered run-time without loading the bitstream again. This design option also incorporates the design time configurable feature in itself. As it was declared in the research question of this thesis that dynamically reconfigurable fault-tolerant mode will be added, therefore, the second design option is chosen.
Regarding reconfiguration, the design of the new ρ-VEX mode will include the following features:

- If multiple contexts are running in parallel, then any context can request reconfiguration to the fault-tolerant mode.
- Out of four lanegroups, any three lane groups can be selected to run under TMR mode. Which lanes to be selected for TMR is encoded in the reconfiguration word.
- Fourth lane group (not under TMR) can be disconnected or sent to power-down mode to save power.
- Fourth lane group (not under TMR) can be used to run a second context in parallel to a context running in the fault-tolerant mode. Note that, this second context will run in non-fault-tolerant mode.

Requesting a reconfiguration
In the existing design, a pre-defined control register is responsible for issuing reconfiguration requests. The same behavior is adopted for the fault-tolerant mode to make the design consistent with the baseline ρ-VEX design. Reconfiguration can be requested via either of the following ways:

- Writing new configuration to the *context control* register (CRR).
- Writing new configuration word to the *bus reconfiguration request control* register (BCRR).
- Using the sleep and wake-up system of the ρ-VEX core.

3.6 Design verification

The purpose of a fault-tolerant design is that in case of fault occurrence, the design must continue its correct execution. When the design is made fault-tolerant, it is not sent directly for the desired mission without being tested thoroughly. These tests are performed to verify that design is capable to mitigate upsets and can continue its correct execution in a harsh environment. In Section 2.4.2, various fault injection techniques were discussed. Selection of a fault injection mechanism for this design will be done based on its feasibility, level of design modifications required, area coverage and time required for verification.

3.6.1 Fault injection method

It is obvious that the ideal way for fault injection is to simulate a space environment by having a radiation beam targeted at the design. But as our university department does not possess such a facility, and outsourcing it will be expensive, we will not go for this option. The other possible options for fault injection are evaluated and summarized in Table 3.3[65].

Table 3.3: Evaluation of various fault injection techniques

Criterion	**Software Based**	**Simulation Based**	**Hardware Based**	
			Saboteur	Mutant
Design Modifications	No	No	Yes	Yes
Modification Level	N/A	N/A	Medium	High
Simulation time	Medium	Low	High	Medium
Resource Coverage	Low	High	High	High
On-board Testing	Yes	No	Yes	Yes

Software solutions are relatively inexpensive solutions. They also do not require design modifications, but the issue with these solutions is that they do not provide

enough coverage. Not all the faults that are possible in hardware because of radiations, can be emulated in software. Considering this, we will not go for this option.
Simulation techniques provide a wide area of coverage and more control over fault injections. The injection of faults via simulation techniques is also relatively easy. But the problem with these solutions is that simulation is very time-consuming. As a comprehensive test is required to thoroughly validate the system, longer simulation times are evident. In addition, a simulation may not necessarily catch all design flaws as it is just an approximation of the actual design. Although it is not a very attractive option, intermediate results and fault injection at module level during their implementation will be done via simulation-based solutions.
For the formal verification of complete design, we need to choose between saboteurs and mutants. Both are hardware-based solutions and provide extensive coverage for fault injection. Choosing between them is just a matter of preference. We will choose saboteurs as they are implemented on interfaces, while for mutants design of modules need to be modified which is not desirable.

3.6.2 Fault injection locations

The locations at which faults will be injected must be chosen wisely so that all vulnerable positions mentioned in the fault model (Figure 3.2) could be tested. Inserting saboteurs at all possible locations where fault could occur is not a feasible option as it will require a lot of modifications in the design (which we do not want) and also take a longer time to run a program. Inserting fault injections randomly can solve this long time issue, but it will not provide satisfactory coverage. As it is possible that design contains some flaws, and random fault injection couldn't test those locations. Further, there is another problem associated with these random fault injections that we cannot reproduce exactly the same injections if needed. Therefore, we must carefully decide fixed fault injection points such that a minimum amount of saboteurs could cover the maximum area.
Fault injections will mainly be done at interfaces. For the pipeline, we will not insert fault at each stage, we will insert faults at a higher level, e.g., at locations where pipeline interacts with outer components. For example, faults will be inserted at interfaces that access general-purpose registers and caches. Faults at such places can emulate the effects of faults happening inside the pipeline. Inside pipeline stages, fault insertion will only be done at the output of the program counter. Faults will also be introduced at interfaces of context resources, trap handler and configuration controller. Besides these, memory elements that are protected by ECC will also be tested for their fault resilience.

3.7 Conclusion

In this chapter, we have presented the baseline design of ρ-VEX processor and analyzed its various components for their susceptibility to faults. After thorough analysis, a fault model is constructed that shows which components of the processor are susceptible to which kind of faults. Afterwards, a comparative analysis of various design options to make the processor fault-tolerant is done. For pipeline protection, the pros and cons of

implementing DWC and TMR are evaluated before finalizing TMR as a design option. All the locations inside the core are identified where TMR checkpoints will be placed. For memory elements, a comparison is done among various types of error correcting codes, and SEC Hamming codes are selected for general-purpose registers and SECDED codes for caches.
Keeping in line with the inherent nature of ρ-VEX code, a reconfigurable fault-tolerant mode is finalized that can be activated or deactivated dynamically. In addition, any three lanegroups out of four lanegroups can be selected to run in TMR mode.
Finally, for the verification of our fault-tolerant design, saboteurs are decided as a fault injection mechanism. The locations where saboteurs will be placed are also identified.

Implementation 4

In the previous chapter, design to turn the existing ρ-VEX core into a fault-resilient core was decided after evaluating multiple design options. This chapter discusses the implementation of this design. In Section 4.1, the high-level design of the fault-tolerant core will be presented, followed by the implementation details of the additional modules introduced in the existing design. These additional modules include majority voters, replication units, and ECC encoders and decoders. Finally, Section 4.2 will discuss the implementation of saboteurs and how they are incorporated in the design to inject faults.

4.1 Fault-tolerant ρ-VEX core

This section presents the implementation details of the fault-tolerant design. The block level diagram of the fault-tolerant ρ-VEX core is provided in the Figure 4.1. New modules that are added to the design are distinguished by the green border line. *MV* blocks represent majority voter, *RU* represents replication unit, *ECC Enc.* and *ECC Dec.* blocks represent Hamming code encoder and decoder respectively. Subsequent sections will provide details about all these newly added modules.

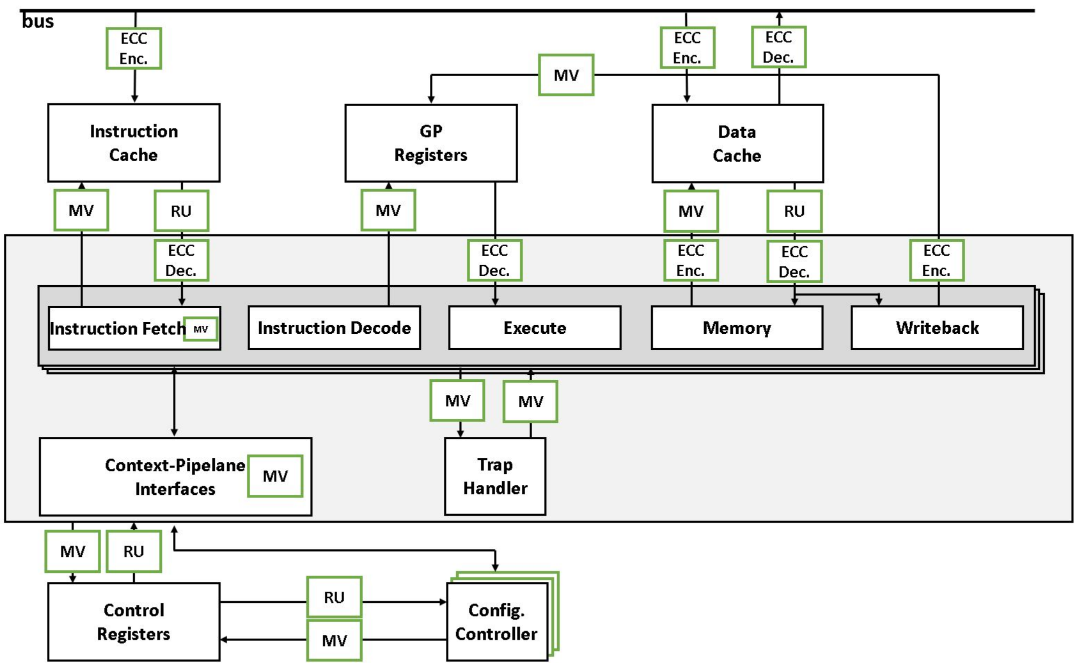


Figure 4.1: Top-level block diagram of the fault-tolerant ρ-VEX core

4.1.1 Majority voters & replication units

Whenever signals leave or enter the TMR domain they need to pass through majority voters or replications units. Implementation details of both these components will be discussed here.

Majority voter

Majority voters are used at all the places where signals exit the TMR domain. The purpose of this module is to compare the input signals coming from three sources and produce a single output based on the majority principle. If out of three modules, one module produces erroneous value, then it will be masked at this stage. Figure 4.2 presents the structure of most commonly used majority voter. The truth table for this voter is the same as was presented in Section 2.4.1.1. Here it is presented again in Table 4.1 for the better understanding of the majority voter functionality. In our design, we want to achieve the same functionality but we can not rely on this majority voter. We have secured pipelane stages by introducing TMR, and memory elements by ECC, but the majority voter presented in Figure 4.2 is not reliable itself. It is a single point of failure and any fault in its structure can sabotage all our design efforts. Therefore, to remove this vulnerability, a triplicated voter presented in Figure 4.3 is implemented in our design. This new voter follows the same functionality depicted by Table 4.1.

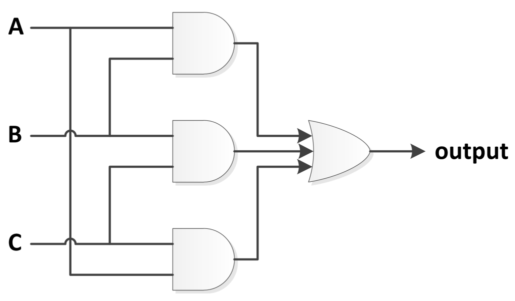


Figure 4.2: 1-bit majority voter

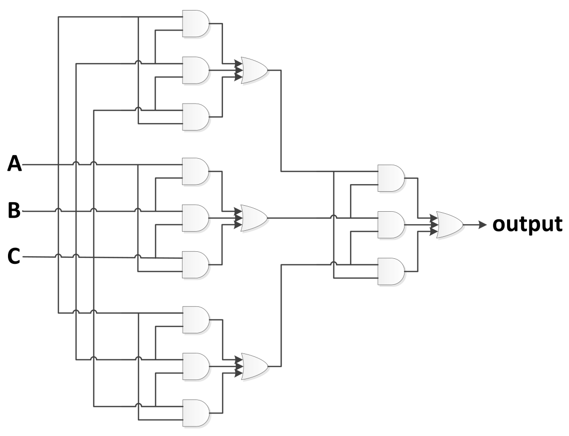


Figure 4.3: Triplicated 1-bit majority voter

As depicted in Figure 4.1, the majority voters are implemented at multiple locations. They are implemented on all signals that exit the TMR domain, e.g., signals that go to the instruction cache, data cache, and general-purpose registers. Majority voter is also implemented for context-pipelane interfaces because all the three lanegorups running under TMR sends the same signals to context related resources and therefore, these signals need to be voted on. Following the decision made in Section 3.4.1.1, majority voter is also implemented for program counter, trap handler and for signals that go from configuration controller to control registers.

One thing must be noted here, that the implemented voter is a one-bit voter and it is replicated multiple times as per the length of the desired signal. Applying the majority voting procedure on a per bit basis provides a significantly high level of protection. For

Table 4.1: Truth table for the majority voter

A	**B**	**C**	**Output**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

a n-bit word, it can correct up to n errors as long as the errors are located in the distinct positions. This is quite a high level of redundancy as compared to ECC codes implemented in memories, as in memories a data-word can correct only one erroneous bit. However, this additional redundancy provided by the majority voter is justified, as a pipeline containing a fault is likely to produce a completely different outcome than the other pipeline running under TMR. This is especially the case when a fault occurs in a control signal in the pipeline and force the pipeline to manifest a different behavior than the other pipelines.
Although the majority voter depicted in Figure 4.3 is incorporated in our final design, a slight (temporary) modification is done to majority voter for the system validation part. In the system validation part, we will insert a number of faults and want to know how many of these faults get detected and corrected. Therefore, additional logic is added to the majority voter structure to obtain the statistics of error correction, as depicted in Figure 4.4.

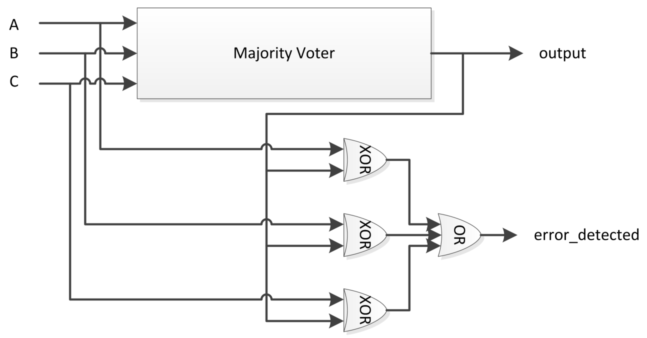


Figure 4.4: Error detection logic for the majority voter

Replication unit
All the signals that go from a single module to TMR domain pass through a replication unit. This unit simply replicates the signal data three times and feed each receiving module with a copy. As depicted in Figure 4.1, signals that go from the instruction cache, data cache and control registers to pipelanes(operating under TMR), pass through a

replication unit. Also, the signals that pass from control registers to triplicated configuration controllers pass through a replication unit.

4.1.2 EDAC implementation

It was decided in Chapter 3 that SEC and SECDED classes of Hamming codes will be used for protecting memory elements. The specifications for the Hamming ECC codes for different data widths are presented in Table 4.2. For general-purpose registers, we have implemented SEC Hamming codes per 32-bit word. This means that per word we have consumed an additional 6 bits to get a feature of single error correction. For instruction cache, we have implemented SECDED Hamming codes per 32-bit word which needs 7 additional parity bits per word. The reason for using SECDED over SEC for caches was given in Section 3.4.2.1, and Section 4.1.3 will provide details about how this additional double error detection feature is exploited to get double error correction. For data cache, as ρ-VEX provides the provision to read or write data per 8-bit word, SECDED codes are implemented per 8-bit word requiring 5 extra parity bits. This means that for 32-bit data-word we have to store 52-bits, which can provide error correction up to 8 bits, i.e., double error correction per 8-bit of data.

Table 4.2: Specifications for the Hamming code

	data width			**error handling**		
	n	k	r	ED	EC	d_{min}
	12	8	4	1	1	3
SEC	38	32	6	1	1	3
	13	8	5	2	1	4
SECDED	39	32	7	2	1	4

The basic components for establishing the ECC protection domain are the encoder and decoder.

Hamming code encoder
Hamming code encoder is responsible for encoding the data by adding parity bits to it. To implement it, the input data word is numbered from the least significant bit to the most significant bit as *1* to n, and then parity bits are placed at the positions with the index equal to the power of 2, i.e., at positions 0,2,4,8 and so on. Rest of the bits are filled with data word bits in the same order in which they appear. Parity bit number t is computed by parity checks of all positions in the code which have a 1 at the position t of their binary index. However, in our design, we have customized the placement of parity bits rather than placing them in the standard way. Figure 4.5 presents the block design of Hamming code encoder. All parity bits are placed together at the end of data word so that they could be identified easily.

As depicted in Figure 4.1, whenever data is written by the pipelane to the data cache and general-purpose registers, it passes through Hamming code encoder. Likewise, all the data written to the instruction cache and data cache from external bus is encoded

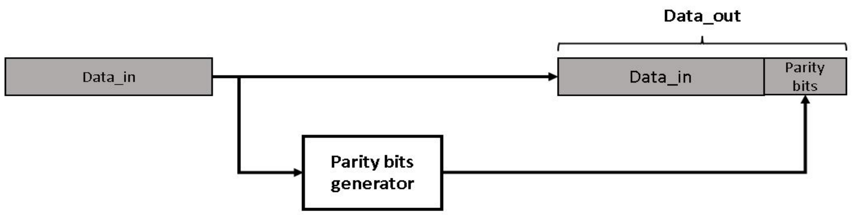


Figure 4.5: Hamming code encoder

first by the encoder.

Hamming code decoder:
In Figure 4.6, the block diagram of the Hamming code decoder is presented. The decoder decodes the incoming data and can correct one bit or detect two bit upsets. Decoder applies parity checks on complete input word (including data word and parity bits). If no error is detected, the data word is forwarded after discarding parity bits. In case of one error detection, error corrector based on the parity checks finds the erroneous bit and then flips it to remove the error before forwarding the data. In case of double error detection, the decoder does not change the data and raises an *error_detected* flag.

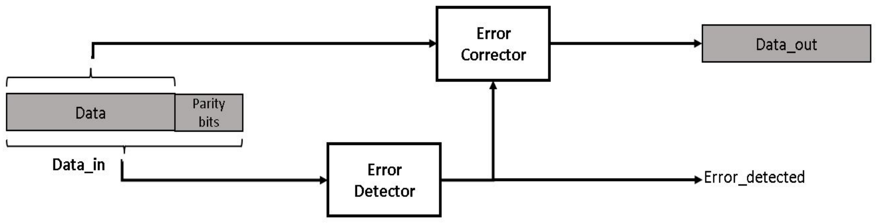


Figure 4.6: Hamming code decoder

As depicted in Figure 4.1, whenever pipelane reads data/instruction from the data cache, instruction cache or general-purpose registers, it passes through Hamming code decoder first. Likewise, data written to the external bus by data cache is also decoded first by the decoder.

4.1.3 Double error correction in caches

Considering the double-error detection capabilities of SECDED Hamming codes, it was decided to exploit the existing cache structure such that it could achieve the capability of double-error correction per data word. To do so, existing cache logic that decides whether the data request leads to a cache hit or a cache miss is modified. An additional parameter which indicates whether a double error is detected or not is incorporated in this design logic. As both instruction and data cache blocks have a similar structure, the same design modifications are implemented for both. Figure 4.7 represents the mechanism for double error correction. When double-error is detected in the requested data by the Hamming decoder, the design logic invalidates the corresponding erroneous entry from the cache memory and declares the request as a cache miss. The cache then requests the fresh copy of data from the external memory via ρ-VEX bus and serve the pipelane request with this fresh data. In this way, we have attained the double error correction ability by using simple Hamming SECDED codes. As the increased protection schemes always come with an additional cost, we have to bear cache miss penalty every time a double error is corrected.

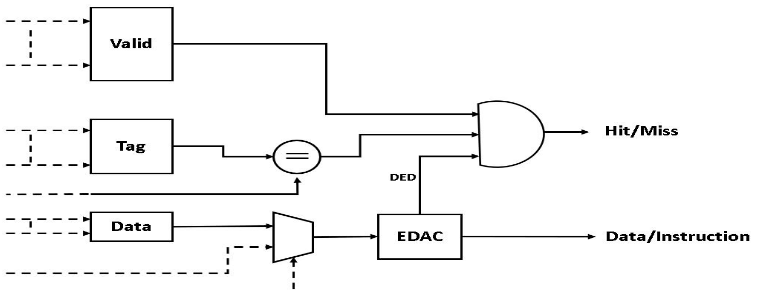


Figure 4.7: Double error correction mechanism for caches

4.1.4 Interaction between TMR and EDAC domains

Placement of EDAC components, majority voters, and replication units is done carefully to avoid any possible single point of failures. Figure 4.8 provides the behavioral description of the design.

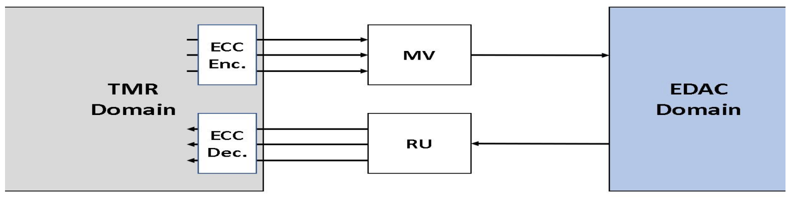


Figure 4.8: Interaction between TMR and EDAC domains

All the signals that pass from the TMR domain to EDAC domain are first passed through the error correcting encoders and then they pass through majority voters. Applying Hamming encoder in TMR domain incurs increased hardware cost as compared to the cost if applied after majority voter, still it is chosen because it provides more protection and prevents provision for the single point of failures. If we apply majority voter first, followed by Hamming encoder then the region that lies between the majority voter and the Hamming encoder is left vulnerable. Any fault occurring in this region will go undetected and corrupt the data word stored in the memory element. Following this erroneous value stored in memory elements, the core might behave haphazardly. Considering that, Hamming encoders are implemented in TMR domain before data is forwarded to majority voters. Likewise, following the same reasoning, Hamming decoders are implemented in TMR domain after data comes from replication units. This positioning of additional components can be seen in the top-level block diagram of the fault-tolerant core presented in Figure 4.1.

4.1.5 Reconfiguration

For configuring the core into the fault-tolerant mode, the same mechanism is adopted which is in place to configure the baseline core in any other reconfigurable modes. A dedicated reconfiguration-word is allocated for it such that it could indicate which context we want to run in the fault-tolerant mode and which lanegroups we want to use for the TMR purpose. As discussed in Section 2.2.4, values from 9 to F were left for the future work and does not correspond to any existing configuration-word encoding scheme. Therefore, value 9 is chosen to indicate the fault-tolerant mode. Figure 4.9 realizes the configuration-word encoding scheme for the fault-tolerant mode. Each marked region corresponds to one nibble. To activate the fault-tolerant mode, 9 must be written to the least significant nibble. If 9 is written to the first nibble, then the second nibble corresponds to the context number which we intend to run in the fault-tolerant mode. The third nibble indicates the lanegroup number, which will not run under TMR and the fourth nibble represents what we want to do with the fourth lanegroup (that is not running under TMR). It can either be deactivated to save power or a second context can be run on it in parallel to the fault-tolerant context.

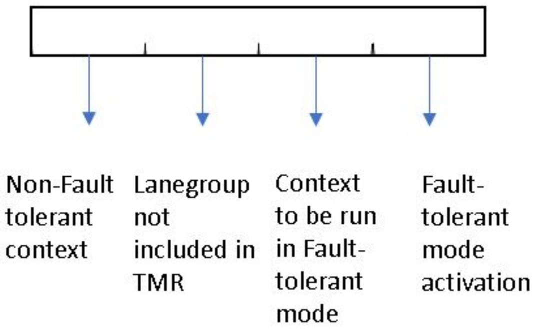

Non-Fault tolerant context
Lanegroup not included in TMR
Context to be run in Fault-tolerant mode
Fault-tolerant mode activation

Figure 4.9: Configuration word encoding for the fault-tolerant mode

To clarify the encoding scheme, few examples will be discussed. Consider 0x1309, 9 here indicates that this configuration scheme corresponds to the fault-tolerant mode and 0 indicates that context 0 should be run in the fault-tolerant mode. The third entry 3 indicates that lanegroup 3 will not be a part of TMR, which means lanegroup 0, 1 and 2 will run under TMR. The fourth entry 1 represents that context 1 will run in the lanegroup 3. To sum up, 0x1309 means context 0 will run in the fault-tolerant mode and will use the first three lanegroups for TMR purpose, and context 1 will run in the last lanegroup. Likewise, the encoding word 0x8219 specifies that context 1 will run in the fault-tolerant mode and use lanegroup 0,1 and 3 for TMR purpose, while lanegroup 2 will be deactivated to save power. Figure 4.10 shows the ρ-VEX core running in 8-way mode. After this core requests reconfiguration via configuration word 0x8309, it will be reconfigured to a fault-tolerant core presented in Figure 4.11, which depicts context 0 running in the fault-tolerant mode using first three lanegroups, while the last lanegroup is in power off mode.

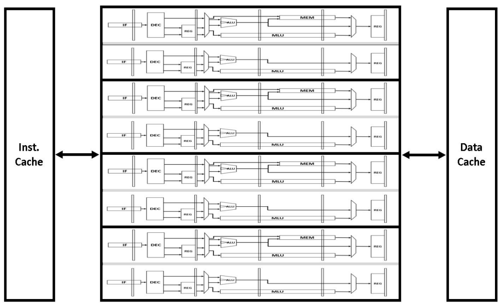


Figure 4.10: ρ-VEX in 8way non-fault-tolerant mode

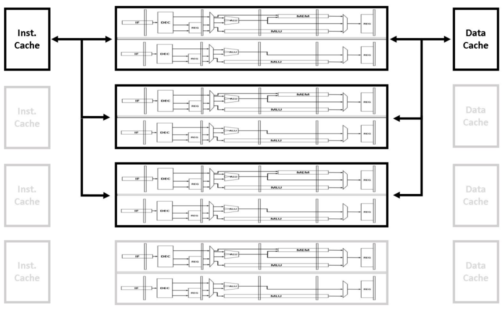


Figure 4.11: ρ-VEX in fault-tolerant mode

4.2 Design validation

In Section 3.6, it was decided that saboteurs will be used as an artificial fault injection method to validate the robustness of our design. This section will deal with the implementation of the saboteur and the mechanism by which we will monitor the error correcting capabilities of our design.

4.2.1 Saboteur

The saboteur is designed as per our requirements such that it inserts faults only when the core is in the fault-tolerant mode and remains silent in the normal modes. Figure 4.12 shows the block diagram of the saboteur which is implemented in this work and it can be configured for any length of the input signal. The faulty signal is produced by XORing the input signal with a *mask signal*. This *mask signal* defines which bit(s) of the input signal we want to corrupt. A counter is used to output a faulty signal after a pre-defined number of clock cycles surpasses. After inserting the fault, the counter resets to zero and start counting again. The saboteur inserts faults only when the core is running in the fault-tolerant mode and the target lanegroup is the one running under TMR, otherwise, it just forwards the input data across without manipulating it. Besides producing faulty signals (when active), saboteur also sets a flag every time fault is inserted so that total numbers of insertions could be monitored.

4.2.2 Status monitoring registers

When the core runs on an FPGA, it is not possible to track the signals inside. Therefore, to monitor the working of our fault-tolerant core, we have introduced a few additional registers. All the internal signals which we want to monitor are led to these registers and can be accessed from outside the core via debug port using ρ-VEX debugger (rvd). These registers provide the following information:

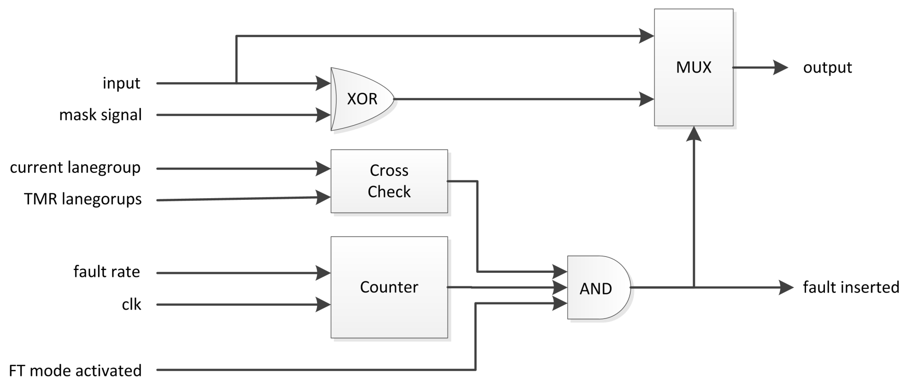


Figure 4.12: Block diagram of saboteur

- whether the fault-tolerant mode is activated or not.
- which lanegroups are running under TMR mode in the fault-tolerant mode.
- which lanegroup out of three lanegroups(running under TMR) can access caches.
- total number of faults injected by the saboteur.
- total number of faults corrected by the majority voters and ECC decoders.

The number of fault insertions by the saboteurs and their correction by the fault-tolerant core is necessary to evaluate the robustness of our design. This evaluation is done comprehensively and results are presented in the Section 5.3.

4.3 Conclusion

In this chapter, the implementation of the fault-tolerant design of ρ-VEX core is presented. Firstly, the high-level block design is discussed followed by the implementation details of each additionally included component. The design of basic building blocks; the majority voter, and the replication unit is presented. Furthermore, the specifications of the Hamming code used are presented along with the design of Hamming encoder and decoder. In addition to this, a new approach used to implement the double-error correction in caches is introduced. Subsequently, the encoding scheme implemented for the fault-tolerant reconfiguration word is explained in the chapter. Finally, the implementation of saboteur that is used as a fault injector at various locations inside the core is presented, followed by the details of status-registers added to monitor the internal signals of the core.

Verification and Results 5

In Chapter 3, we discussed the design of the fault-tolerant ρ-VEX core and Chapter 4 provided its implementation details. In this chapter, we will discuss how the functionality of the designed and implemented core is verified. Section 5.1 will discuss the details of the platforms and benchmark used to test the basic functionality of the core. In Section 5.2 and Section 5.3, details about the functional testing and verification of the fault-tolerance capabilities will be presented. Finally, the cost incurred to the system in the form of resource utilization and clock cycle limitations will be presented in Section 5.4.

5.1 Test environment

The test environment established to validate the functionality of our modified design is comprised of testing platforms and a standardized benchmark suite.

5.1.1 Test platform

For verification of our design, both simulation-based and on-board test environment are used. The simulation-based test setup is used throughout the development phases of the design as it provides provision to monitor intermediate or partial results. To validate the correctness of our entire design, simulation platform alone is not enough as it is quite difficult to precisely emulate the hardware environment. Therefore, along with the simulation platform, on-board test setup is also used to validate the finalized design.

Simulation-based test platform
The platform used for the simulation of the system is ModelSim-Intel FPGA software. It provides support for inter gate-level libraries and includes behavioral simulations, HDL testbenches, and Tcl scripting. A comprehensive test bench to verify the working of ρ-VEX core was developed by my predecessor(s) which is used in this work without any modifications. The test-bench tries to imitate the on-board environment as closely as possible. An application written in C-language can be run and evaluated using this simulation platform.

On-board test platform
The on-board platform used to validate the design is ML605 evaluation kit. It is produced by Xilinx and depicted in Figure 5.1. It features Virtex-6 XC6VLX240T-1FFG1156 FPGA along with various peripherals.

Xilinx provides its own synthesis toolchains for its products. The toolchain that is used in this thesis work is Xilinx ISE 14.7. Although Vivado has superseded ISE 14.7 toolchain, it is not used in this work because it lacks support for 6-series FPGAs [66].

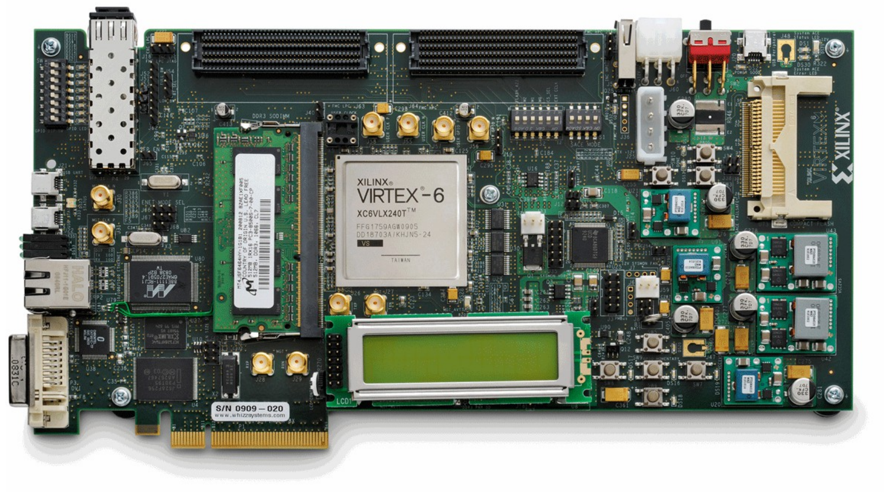


Figure 5.1: ML605 - The FPGA development board used in this project

5.1.2 Test applications

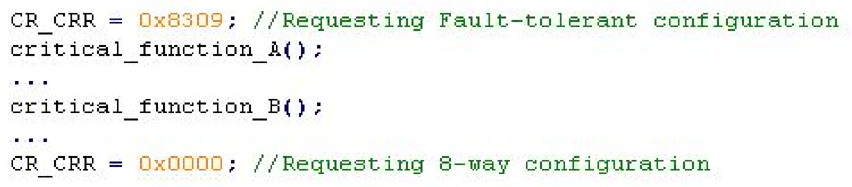
In order to verify the correct functioning of our design, the *Powerstone Benchmark suite* is used. The reason for selection of this benchmark suite is that applications in it are relatively simple and do not rely significantly on any advanced libraries (which have not been ported to ρ-VEX yet). Another reason for using this benchmark suite is that it has already been ported to ρ-VEX architecture in earlier work. Thirteen applications from the Powerstone suite are used in this work and two applications namely *auto* and *whetstone* could not be used. The reason for not using these two applications is that their source code could not be found. The brief description of the thirteen applications used are listed in Table 5.1 [67].

5.2 Functional testing

The purpose of functional testing is to validate the correct working of the core and it does not focus on its fault-tolerance abilities. Along with checking the correct execution of the programs, it also tests the reconfigurable nature of the core. To perform functional testing, the applications were run on all ρ-VEX modes, i.e., 8-way, 4-way and 2 -way and a reconfiguration request to configure the core into the fault-tolerant mode was made inside the application source code. To request the fault-tolerant mode inside the application, we just needed to write the appropriate configuration word to the context control register. Figure 5.2 shows an example where fault-tolerance is requested for a chunk of code. $CR\_CRR = 0x8309$ configures the core into the fault-tolerant mode, while $CR\_CRR = 0x0000$ reconfigures the core back into the normal (non-fault-tolerant) 8-way mode.

Table 5.1: Powerstone benchmark suite

Applications	**Description**
bcnt	Bit shifting & ANDing through 1K array
blit	Graphics applications
compress	A UNIX utility
crc	Cyclic redundancy check
des	Data encryption standard
engine	Engine control application
fir	Integer FIR filter
g3fax	Group three fax decode(single level image decompression)
jpeg	JPEG 24-bit image decompression standard
pocsag	POCSAG paging communication protocols
qurt	Square Root calculation
ucbqsort	U.C.B quick sort
v42	Modem encoding/decoding



```
CR_CRR = 0x8309; //Requesting Fault-tolerant configuration
critical_function_A();
...
critical_function_B();
...
CR_CRR = 0x0000; //Requesting 8-way configuration
```

Figure 5.2: Part of an application showing how reconfiguration can be requested

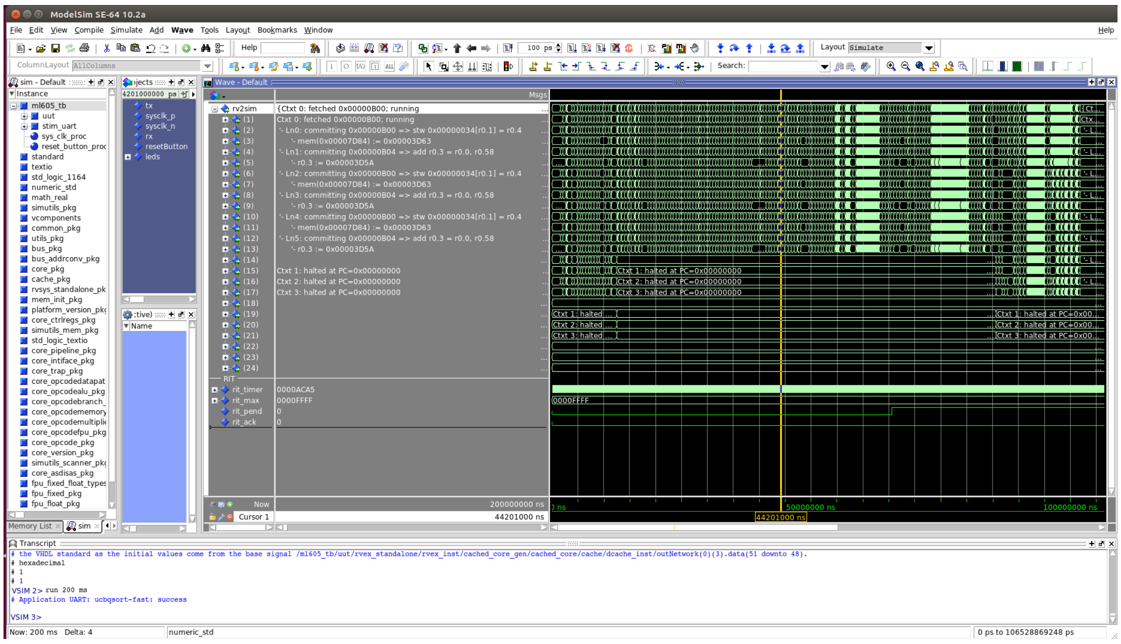
System configuration
The standalone processing system is used throughout in this work for verification of our design. This system provides minimal dependencies and deterministic timing for the memory system, to provide a platform for the experimentation. This system comes in two versions; one with a cache and one without. However, for our work, we used the system that comes with a cache. The system configuration which was used is provided in Figure 5.3 . Complete details about the instantiation can be found in the source code.

Simulation-based testing
Figure 5.4 presents the simulation results for the core which requested reconfiguration to a fault-tolerant mode from the 8-way mode, and after running in the fault-tolerant mode for a while, again requested reconfiguration to the 8-way mode. In the fault-tolerant mode, first three lane-groups started executing exactly the same instructions in lock-step mode. This behavior imitates execution of an application which needs some critical function to run in a fault-tolerant mode.

Similarly, Figure 5.5 shows a core which requested the fault-tolerant mode from a 2-way mode, and after running in the fault-tolerant mode for a while, requested the

```
constant CFG                : rvex_sa_generic_config_type := rvex_sa_cfg(
   core => rvex_cfg(
     numLanesLog2            => 3,
     numLaneGroupsLog2       => 2,
     numContextsLog2         => 2,
     traceEnable             => 1
   ),
   core_valid                => true,
   cache_enable              => 1,
   cache_config => cache_cfg(
     instrCacheLinesLog2     => 8,
     dataCacheLinesLog2      => 8
   ),
   cache_config_valid        => true,
   dmemDepthLog2B            => 18
 );
```

Figure 5.3: Instantiation template used for the system



```
rv2sim  {Ctxt 0: fetched 0x00000B00; running
(1)     Ctxt 0: fetched 0x00000B00; running
(2)      '- Ln0: committing 0x00000B00 => stw 0x00000034[r0.1] = r0.4
(3)         '- mem(0x00007D84) := 0x00003D63
(4)      '- Ln1: committing 0x00000B04 => add r0.3 = r0.0, r0.58
(5)         '- r0.3 := 0x00003D5A
(6)      '- Ln2: committing 0x00000B00 => stw 0x00000034[r0.1] = r0.4
(7)         '- mem(0x00007D84) := 0x00003D63
(8)      '- Ln3: committing 0x00000B04 => add r0.3 = r0.0, r0.58
(9)         '- r0.3 := 0x00003D5A
(10)     '- Ln4: committing 0x00000B00 => stw 0x00000034[r0.1] = r0.4
(11)        '- mem(0x00007D84) := 0x00003D63
(12)     '- Ln5: committing 0x00000B04 => add r0.3 = r0.0, r0.58
(13)        '- r0.3 := 0x00003D5A
(14)
(15)    Ctxt 1: halted at PC=0x00000000
(16)    Ctxt 2: halted at PC=0x00000000
(17)    Ctxt 3: halted at PC=0x00000000
(18)
(19)
(20)
(21)
(22)
(23)
(24)
RIT
rit_timer  0000ACA5
rit_max    0000FFFF
rit_pend   0
rit_ack    0
Now        200000000 ns
Cursor 1   44201000 ns
```

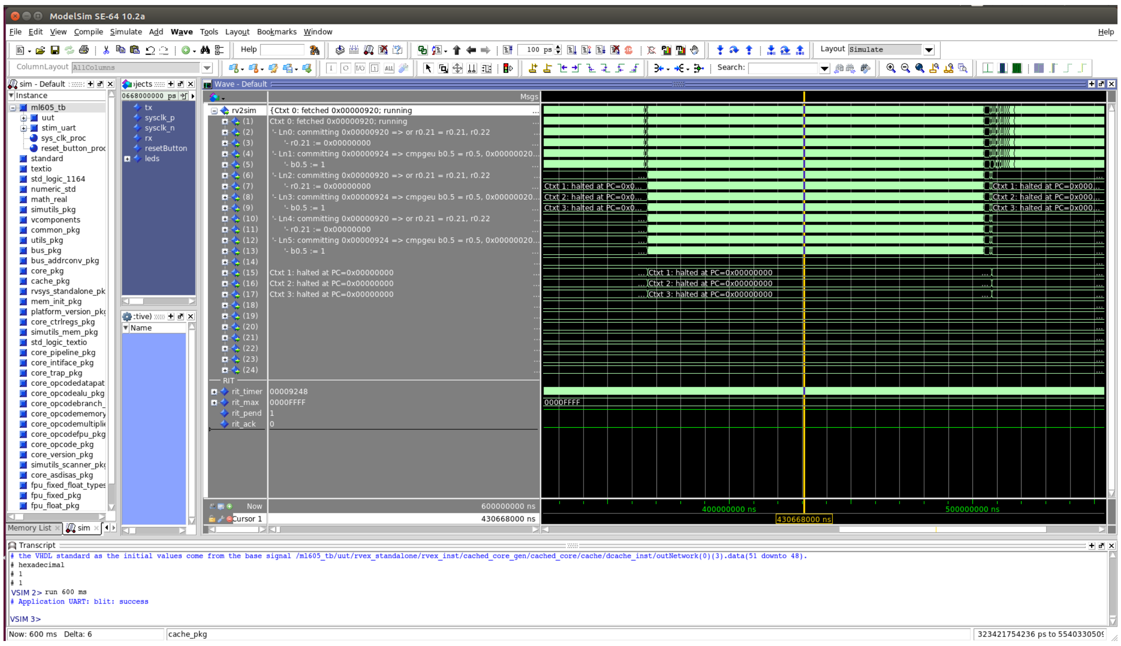
```
# the VHDL standard as the initial values come from the base signal /ml605_tb/uut/rvex_standalone/rvex_inst/cached_core_gen/cached_core/cache/dcache_inst/outNetwork(0)(3).data(51 downto 48).
# hexadecimal
# 1
# 1
VSIM 2> run 200 ms
# Application UART: ucbqsort-fast: success

VSIM 3>
```

Figure 5.4: Screenshot of ModelSim showing reconfigurations from 8-way mode to a fault-tolerant mode and then back to the 8-way mode

normal 2-way mode. This sequence of ρ-VEX modes was also followed for the on-board test environment.

On-board testing

A rigorous testing of the system was done on on-board FPGA platform. Firstly, the core was configured into the fault-tolerant mode and applications were run on it entirely in the fault-tolerant mode without requesting any run-time reconfigurations. All the applications in Powerstone benchmark suite successfully completed their executions under this mode. The active cycles taken by the applications were the same as they take in normal 2-way mode of ρ-VEX. In the fault-tolerant mode, the application basically runs on the 2-way mode, and it is just replicated three times to introduce appropriate



```
rv2sim    {Ctxt 0: fetched 0x00000920; running
(1)       Ctxt 0: fetched 0x00000920; running
(2)        '- Ln0: committing 0x00000920 => or r0.21 = r0.21, r0.22
(3)          '- r0.21 := 0x00000000
(4)        '- Ln1: committing 0x00000924 => cmpgeu b0.5 = r0.5, 0x00000020...
(5)          '- b0.5 := 1
(6)        '- Ln2: committing 0x00000920 => or r0.21 = r0.21, r0.22
(7)          '- r0.21 := 0x00000000
(8)        '- Ln3: committing 0x00000924 => cmpgeu b0.5 = r0.5, 0x00000020...
(9)          '- b0.5 := 1
(10)       '- Ln4: committing 0x00000920 => or r0.21 = r0.21, r0.22
(11)         '- r0.21 := 0x00000000
(12)       '- Ln5: committing 0x00000924 => cmpgeu b0.5 = r0.5, 0x00000020...
(13)         '- b0.5 := 1
(14)
(15)      Ctxt 1: halted at PC=0x00000000
(16)      Ctxt 2: halted at PC=0x00000000
(17)      Ctxt 3: halted at PC=0x00000000
RIT
rit_timer 00009248
rit_max   0000FFFF
rit_pend  1
rit_ack   0

Now       600000000 ns
Cursor 1  430668000 ns
```

```
# the VHDL standard as the initial values come from the base signal /ml605_tb/uut/rvex_standalone/rvex_inst/cached_core_gen/cached_core/cache/dcache_inst/outNetwork(0)(3).data(51 downto 48).
# hexadecimal
# 1
# 1
VSIM 2> run 600 ms
# Application UART: blit: success

VSIM 3>
```

Figure 5.5: Screenshot of ModelSim showing reconfigurations from 2-way mode to a fault-tolerant mode and then back to the 2-way mode

redundancy for implementing TMR.
In order to test the dynamically reconfigurable feature on-board, the core was first configured to a 2-way mode. Inside the application, the fault-tolerant mode was requested for a chunk of code (e.g., some function) which means that before entering to that particular chunk of code, the core configured itself into the fault-tolerant mode and after exiting from it, configured itself back into the original 2-way mode. As reconfiguration was requested twice inside the application, this behavior incurred double reconfiguration overhead. Table 5.2 lists the number of active cycles taken by the applications when they requested the fault-tolerant mode for one function (dual reconfiguration) inside the source code.

It can be seen that the overhead cost for reconfiguring the core comprises of two factors; reconfiguration logic cost and cache-flush penalty. Reconfiguration logic cost depends on how long it takes the reconfiguration controller to pause the affected contexts and flush the pipelines. The cache-flush penalty depends on the state of cache and temporal locality of the application data. The overhead of both these factors is variable.

[68] states that the reconfiguration logic cost of ρ-VEX can be in the order of tens of cycles and the maximum cost that can be incurred is 14 cycles. The cost is variable as it depends on various factors including stalls from the memory subsystem, the state of the cache write buffers, the requested configuration, its difference between the requested and current configuration, and the pipeline configuration. However, our statistics also show that cost of reconfiguration logic for a single reconfiguration is always less than 14. It must be noted that entries in Table 5.2 showing reconfiguration logic overhead, is providing the cost of twice reconfiguration. In the programs, reconfiguration was

Table 5.2: Active clock cycles taken by the benchmark suite

Applications	**Non-FT**	**FT over one function**	**Overhead**			
	2-way mode	(Dual Configuration)	Reconfiguration	Cache Flush	Total	Total %age
bcnt	105245	107875	15	2615	2630	2.5
blit	377152	379967	16	2799	2815	0.7
compress	2246347	2249046	17	2682	2699	0.1
crc	81960	98316	16	16340	16356	20
des	928005	931778	16	3757	3773	0.4
engine	6387291	6390928	16	3621	3637	0.06
fir	713186	717501	15	4300	4315	0.6
g3fax	8154246	8160409	16	6147	6163	0.08
jpeg	24498088	24500908	15	2805	2820	0.01
pocsag	244871	250271	17	5383	5400	2.2
qurt	152445	157568	15	5108	5123	3.4
ucbqsort	848954	852794	17	3823	3840	0.5
v42	17662996	17665288	15	2277	2292	0.01

requested before entering the critical section and then after exiting the critical section, therefore, incurring twice reconfiguration cost. The second factor in overall overhead cost is the cache-flush penalty. This is the major cost and reconfiguration logic cost is almost negligible as compared to it. Whenever the core reconfigures itself, then depending on the requested configuration, related caches are flushed. One of the reasons for flushing cache is that cache size is not constant and varies as per the current configuration. As an example, if a single context is running on an 8-way mode, then reconfiguring it in a 2-way mode results in resizing of its cache size and it is reduced to about 4 times smaller cache. To comply with this cache-resizing feature and keep the design generic, cache is flushed every time reconfiguration is requested. The overhead for cache-flush is a quite dominant factor in overall reconfiguration cost.
It is assumed that during the mission, either the core will be run entirely in fault-tolerant mode (no reconfiguration needed) or reconfiguration to the fault-tolerant mode will be requested only for critical applications. As this reconfiguration will not be very frequent, therefore, reconfiguration overhead can be accepted. The overhead increases in direct relation to the number of times reconfiguration is requested. This behavior is tested using two different type of applications. Each application was run multiple times and each time multiple reconfiguration requests were made dynamically. The results gathered can be seen in Figures 5.6 and 5.7. Figure 5.6 represents total overhead while Figure 5.7 represents reconfiguration logic overhead for the applications when reconfiguration was requested multiple times. Again it can be seen that the reconfiguration logic is almost negligible as compared to total overhead as the cache-flush penalty is quite dominant. The upward slope of both the graphs is in-line with our inference that cost of reconfiguration is directly proportional to the number of times it is requested.

Running two contexts in parallel
Our design provides the provision that out of four lane-groups we can choose any three lane-groups to run under TMR in the fault-tolerant mode. The fourth lane-group can either be de-activated to save some power or can run another context in parallel. To

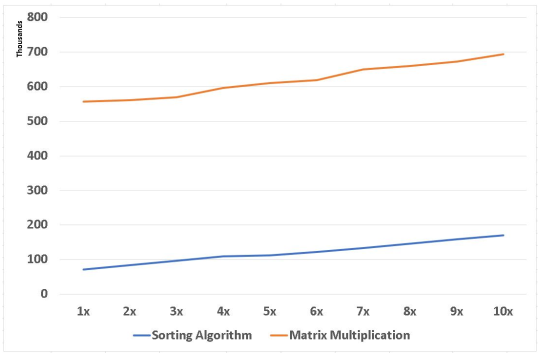


Figure 5.6: Total overhead in terms of clock cycles when multiple reconfigurations are requested

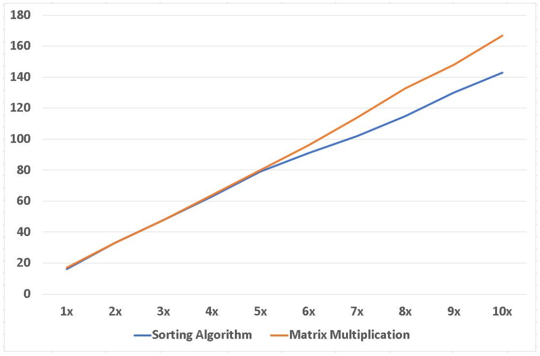


Figure 5.7: Reconfiguration logic overhead in terms of clock cycles when multiple reconfigurations are requested

perform a rigorous testing of running multiple contexts and monitoring their switching, an operating system support is needed. Because of the unavailability of this support, an alternate approach is used. A single context is run independently on both TMR lane-groups and spare lane-group.
A configuration word 0x0309 was written to the configuration-register file. As per encoding scheme, this word indicates that context zero to be run in the fault-tolerant mode using lane-groups 0,1 and 2. This word also indicates context 0 to run on lane-group 3 in normal (non-fault-tolerant) mode. Thus, this encoding word imitates two contexts running in parallel, one being in fault-tolerant mode and the other being in normal (non-fault-tolerant) mode. Multiple programs were run in this configuration. All the programs successfully completed the execution and printed the *success* output twice (one from the fault-tolerant and one from the normal context) via UART interface. Other configuration words also used to verify this behavior were 0x0009, 0x0109 and 0x0209. These words represent the same functionality as represented by 0x0309, but configures distinct possible combinations of lane-groups to be run under TMR. All of these combinations provided the desired results.

5.3 Fault-tolerance verification

A verification procedure is followed in order to test the fault-tolerant capabilities and to check whether the core is capable to mitigate faults and can continue its correct execution in the presence of faults. Saboteurs, whose implementation was discussed in Section 4.2.1, were inserted at various locations in the core. Status-monitoring registers explained in Section 4.2.2 were used to collect statistics of fault insertions, detections, and corrections.
Saboteurs were inserted on the interfaces between pipeline and instruction cache, data cache and general-purpose registers. These were also inserted inside the program counter logic, trap handler, configuration controller, majority voter and the debug circuitry. After a pre-defined number of clock cycles, these saboteurs inserted faults at these places. Error detection and correction logic then detected these anomalies and reported the status by

updating the status-monitoring registers. Before injecting errors, it was checked that core was running under the fault-tolerant mode and the target lane-group was a part of TMR. This was done to restrict saboteurs from inserting errors when the core was running in normal (non-fault-tolerant) mode and also to not interfere with the spare lane-group (not running under TMR).
Figure 5.8 provides the results of fault injection tests. At least five thousand faults were inserted in each of the entity, and none of them was able to cause a failure. Almost all the faults were detected and corrected by the ECC decoder or a majority voter. In case of debug circuitry, only 9% faults were detected and corrected. This behavior was certain because only partial debug circuitry is made fault-tolerant, rather than the entire debug circuitry. The use of debug circuitry is for on-ground testing and debugging and it is of no use during the mission. However, the critical signals such as *write enable* and *reset* signals that might cause the core to misbehave are made fault-tolerant. 9% faults that were corrected in the debug circuitry lied in these critical signals. The remaining faults go undetected, but it must be noted that these faults did not have any impact on the correct execution of the program and the program completed its execution successfully. All faults inserted in other regions that included input and output to both caches, general-purpose registers, configuration controller, trap handler, majority voter itself, and the program counter got detected and corrected by the fault-tolerant core.

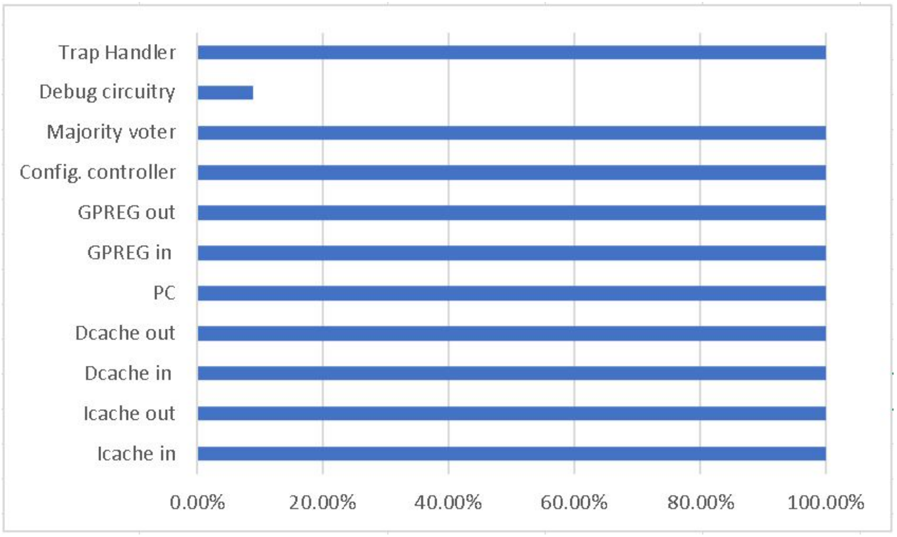


Figure 5.8: The results from the fault injection tests showing the percentage of corrected faults

Error insertions inside the cache memory and pipelines were also done and it was verified that a program successfully completes the execution. However, statistics of such insertions are not documented in the report because their statistics do not provide a clear picture whether all faults are corrected or not. In caches, if a bit inside the memory is flipped, then before this entry is accessed and passed through ECC decoder, this entry might get invalidated or replaced when the cache fills up. In such a case, the inserted error will go undetected, though it will not have any impact on the correct execution of

the program. However, to cover such errors, faults were inserted into the data when it was read from the cache and was about to pass through ECC decoder. This behavior emulated the effects of faults occurring inside the cache and also helped us in keeping count. In the case of pipelane, the situation is quite opposite to that of caches. One bit error might lead to multiple bits errors as an erroneous data address might fetch entirely different data, or operations on faulty data in *Execution* stage of the pipeline might produce totally different results. As long as redundant pipelanes produce correct results, it does not matter even if all the bits in a faulty pipelane are erroneous. After voting, correct results will be forwarded. Thus if a fault is inserted in the pipelane, it can lead to detection and correction of multiple errors, and in case of a design flaw, might also go undetected. Thus we can not deduce any valuable information from fault insertions inside the pipeline. Therefore, faults were inserted on the output data of pipeline, which besides emulating effects of faults originating from inside the pipeline, also helped in validating the system by comparing the number of faults insertions to the number of faults detections.
For instruction and data caches, we have implemented a protection mechanism which can detect and correct two errors per word, as explained in Section 4.1.3. To verify the functionality of this added behavior, two faults on consecutive bits in a single data word were introduced. These faults were included when data was being read and before it passed through the ECC checks. A total of one thousand double-bit errors were introduced in both instruction and data cache, and none of these errors could trigger a failure. All of them were detected and respective cache entries were invalidated, and status-monitoring registers were updated accordingly.
As the platform used for system validation is standalone processing system, the access latency of the memory could be configured at runtime. This feature of standalone processing system provides a provision to mimic a more realistic memory access latency for cache tests. It was observed during tests that the overhead due to access latency was directly proportional to the number of double-bit faults insertions per word in caches.

5.4 Cost and overhead of fault-tolerance

Fault-tolerance is an expensive feature and our improved design which makes the core fault-tolerant also comes with a price. This price can be categorized into additional hardware cost and system performance degradation cost.

5.4.1 Additional hardware / Resource utilization

The design is synthesized with Xilinx ISE 14.7 and details of resource utilization are acquired. The absolute values of the resources used in the baseline core and the fault-tolerant core are presented in Table 5.3 and their relative resource utilization is presented graphically in Figure 5.9. The maximum difference is for LUTs which are 29% more than that of the baseline core, followed by the number of slices and BRAM which are 28% and 25% more respectively. The increase in LUTs and registers corresponds to the additional logic introduced in the form of majority voters, replication units, configuration controller alteration, and other similar modifications, while the increase in BRAM corresponds to the addition of ECC check words to the data words stored in caches and general-purpose registers.

Table 5.3: Resource utilization comparison of the original core and the fault-tolerant core

Resource	**Baseline Core**	**Fault-Tolerant Core**	**Difference**
Slices	23989	30617	6628
Registers	27179	30043	2864
LUTs	65841	84689	18848
DSPs	32	32	0
BRAMs	345	433	88
f_{max}	37.5 MHz	30.0 MHz	7.5 MHz

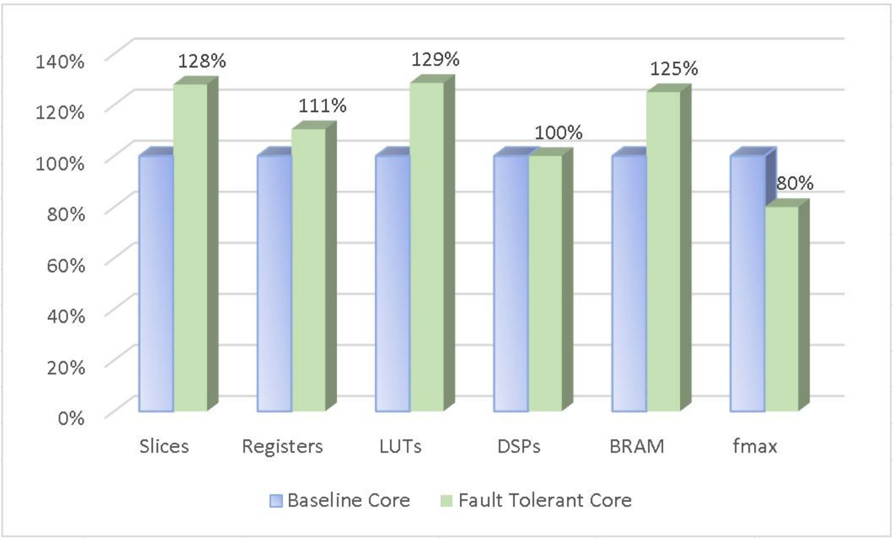


Figure 5.9: The relative resource utilization

[69] and [70] represents two different works that incorporated TMR in some different processor platforms. Both of these designs lists the resource utilization of the fault-tolerant processor to be more than 400% of that of original processor. TMR implementations and design features might be different than our work, but these examples are mentioned to give a rough estimate of additional hardware demand in a fault-tolerant core. Now, one might wonder why our modified design only takes less than 130% resources utilization to that of the baseline design. The reason is that we have exploited the underlying design to implement TMR in pipelines, instead of adding new pipelines in the design. ρ-VEX provides 8 pipelanes and we have used the same pipelanes for TMR whenever the fault-tolerant mode is activated. This careful and intelligent use of already existing pipelanes in the core is the reason, why our resource utilization is far less than the 400%. We have achieved the goal of fault-tolerance implementation by the inclusion of just less than 30% additional resources.

5.4.2 Timing results

Besides the inclusion of additional hardware, our new design also affects the critical path of the system. The critical path in the design can be defined as the path in the entire design with a maximum delay. This critical path defines the maximum frequency at which a core can operate. Figure 5.9 shows that the maximum frequency that the fault-tolerant core can operate on is 20% less than that of the baseline core. The five longest paths in our design are highlighted in red color in Figure 5.10. Though there are other paths in the core as well which defy the original f_{max} of 37.5 MHz but the highlighted paths have major impacts on the frequency and limit the fault-tolerant core to operate at a maximum frequency of 30MHz. This increase in critical path is understandable as the longer paths in the baseline core are further elongated by the inclusion of either ECC decoder or ECC encoder and triplicated majority voter in them.

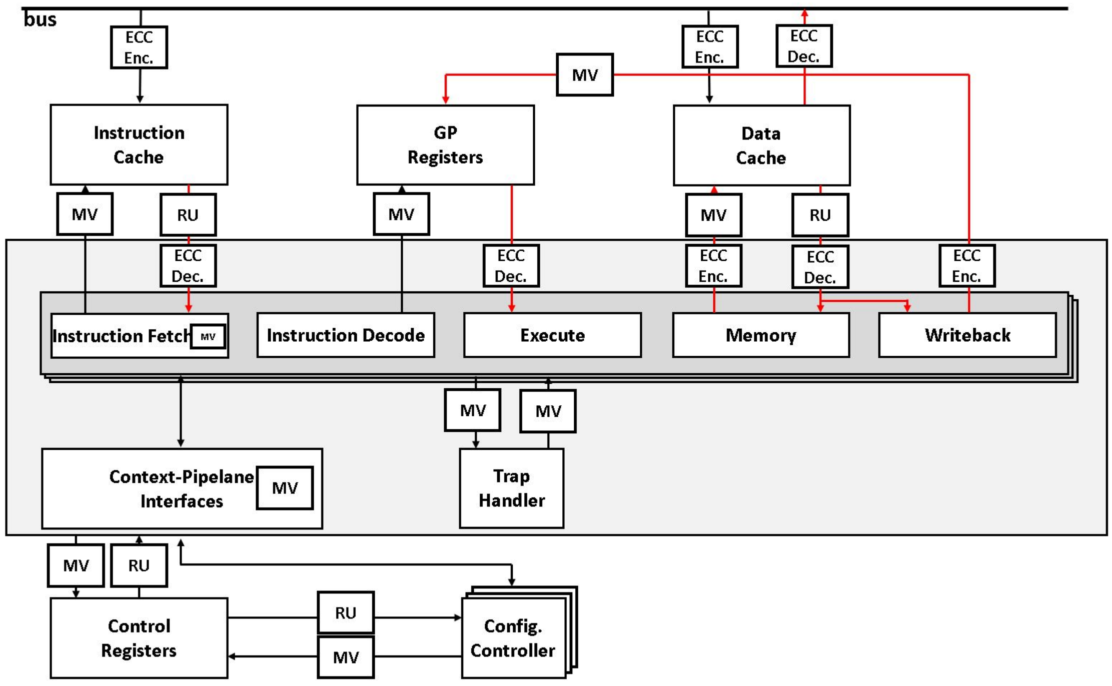


Figure 5.10: The long delay paths in the fault-tolerant ρ-VEX core

5.5 Conclusion

This chapter deals with the verification of our design and presents some results associated with our fault-tolerant core. For verification, a simulation-based test platform using ModelSim, and an on-board test platform using a Virtex-6 FPGA is described. Powerstone benchmark suite is used to test the basic functionality of the design. The on-board test platform is also used to perform fault injection tests and it is observed that none of the faults could trigger any failure.
The results mentioned in the chapter provides details of reconfiguration overhead which

incurs whenever the fault-tolerant mode is requested. This overhead is the same as in the case of normal mode switching in the baseline core. The additional hardware utilized to implement new design is less than 30% to that of the baseline design. The new design also has its impact on the system clock cycle and the minimum clock period allowed is 20% more than that is allowed in the baseline core.

Conclusion 6

This chapter summarizes and concludes this thesis work. Section 6.1 will summarize the work done in this thesis, and Section 6.2 will reiterate the research question and thesis objectives, and will list the main contributions made. Finally, in Section 6.3, recommendations for the future work will be provided.

6.1 Summary

Chapter 2 provides the background information required for this thesis work. Firstly, the ρ-VEX platform including its design, architecture, distinguishing features and working is explored. Then the study of space environment and its effects on electronics are provided. Subsequently, fault mitigation techniques for processor found in the literature are explored. These include redundancy based TMR and DWC techniques for processor pipeline stage and error correcting codes and scrubbing for memory elements. Finally, the verification techniques for the fault-tolerant systems are explored with the focus on fault injection. Different types including hardware-based, software-based and simulation-based fault injections are investigated.

Chapter 3 presents the design of the main components of the new fault-tolerant processor. It starts by presenting the baseline design of ρ-VEX processor and analyzing its various components for their susceptibility to faults. After thorough analysis, a fault model is constructed that shows which components of the processor are susceptible to which kind of faults. Afterwards, a comparative analysis of various design options to make the processor fault-tolerant is done. For pipeline protection, the pros and cons of implementing DWC and TMR are evaluated before finalizing TMR as a design option. All the locations inside the core are identified for placement of TMR checkpoints. For memory elements, a comparison is done among various types of error correcting codes, and SEC Hamming codes are selected for general-purpose registers and SECDED codes for caches.
Keeping in line with the inherent nature of ρ-VEX code, a reconfigurable fault-tolerant mode is finalized that can be activated or deactivated dynamically. In addition, any three lanegroups out of four lanegroups can be selected to run in TMR mode and the leftover lanegroup can be used to run a second context. After finalizing the entire design, insertion of saboteurs is decided as a fault injection mechanism to validate the functionality of our design, and the locations for saboteurs placement are also identified.

Chapter 4 presents the implementation of the fault-tolerant design of the ρ-VEX core. Firstly, the high-level block design is discussed followed by the implementation details of each additionally included component. The design of basic building blocks, the majority voter, and the replication unit is presented. Furthermore, the specifications of the Hamming code used are presented along with the design of Hamming encoder and decoder. In addition to this, a new approach used to implement double-error correction in caches is introduced and its implementation details are provided. Afterwards, the encoding scheme implemented for the fault-tolerant reconfiguration word is explained in the chapter. Finally, the implementation of saboteur that is used as a fault injector at various locations inside the core is presented, followed by the details of status-registers added to monitor the internal signals of the core.

Chapter 5 presents how the functionality of the designed and implemented fault-tolerant core is verified and provides some results associated with it. For verification, a simulation-based test platform using ModelSim, and an on-board test platform using a Virtex-6 FPGA is described. Powerstone benchmark suite is used to test the basic functionality of the design. The on-board test platform is also used to perform fault injection tests and it is observed that none of the faults could trigger any failure.
The results mentioned in the chapter provides details of reconfiguration overhead which incurs whenever the fault-tolerant mode is requested. This overhead is the same as in the case of normal mode switching in the baseline core. The additional hardware utilized to implement new design is less than 30% to that of the baseline design. The new design also has its impact on the system clock cycle and the minimum clock period allowed is 20% more than that is allowed in the baseline core.

6.2 Main contributions

In Section 1.1, the research question for this thesis was formulated as:

How can the ρ-VEX softcore be extended so that it becomes a reliable option for space-based critical missions?

We have responded to this question by presenting a fault-tolerant design, implementing it, and verifying its functionality on an FPGA development board. Particularly, we have demonstrated that core can be configured into a fault-tolerant mode which is capable of detecting and correcting faults introduced by single event effects. This mode can either be configured at design time resulting in a fault-tolerant core that could be used throughout the mission, or it can also be configured dynamically whenever the execution of some critical function is sought.
In Section 1.1, we proposed the thesis objectives which were kept under consideration throughout the design and implementation phases. These objectives are listed again here, along with a short summary of how the thesis objectives were achieved.

- *The design must be able to detect and correct errors in the execution stages of ρ-VEX.*
  Triple modular redundancy (TMR) was implemented for the pipeline stage to detect and correct errors in it. No additional pipelines were added in the design for this purpose, instead the underlying ρ-VEX architecture characteristics were

exploited such that out of four available lanegroups, any three lanegroups can be selected to run under TMR. The outputs of these TMR lanegroups subsequently pass through a voter which works according to the majority principle and masks a single fault. For a particular n-bit signal, majority voter can correct up to n errors, as long as the errors are located in a distinct position.

- *The design must make on-chip memories robust against soft-errors.*
  Hamming codes were implemented for the protection of on-chip memories. For general-purpose registers, SEC codes and for the instruction cache, SECDED codes were implemented per 32-bit word. For data cache, as we have the provision to access data per byte, therefore, SECDED codes were implemented per 8-bit word. For caches, a new approach was implemented to provide additional protection. This approach enables the caches to correct two bits per word, using SECDED codes and cache invalidation logic.

- *The design must be dynamically (runtime) reconfigurable.*
  A new fault-tolerant mode is added to the existing modes of ρ-VEX core such that it can be reconfigured dynamically. A dedicated encoding scheme for the configuration word was implemented in this work that can trigger the fault-tolerant mode. An application can request activation and deactivation of this mode multiple times during its execution. The overhead of the dynamic reconfiguration to this fault-tolerant occurs in terms of pipeline-flush and cache-flush penalty.

- *The design must be implemented on an FPGA while taking platform independence into account.*
  ML605 evaluation kit produced by Xilinx that features Virtex-6 XC6VLX240T-1FFG1156 FPGA was used for design implementation. Throughout the design, it was ensured that implementation was made as efficient as possible without losing platform independence out of sight. All newly added components were implemented with behavioral descriptions and structural descriptions using no platform specific building blocks.

- *The design must be verified for its correctness.*

  A simulation-based test platform using ModelSim, and an on-board test platform using a Virtex-6 FPGA was used to verify the correctness of our design. Powerstone benchmark suite was run on these platforms to test the basic functionality of the design. To verify the fault-tolerance capabilities, artificial fault insertions using saboteurs were performed. Status of these faults insertions, and their corrections by our design were monitored by status-monitoring registers. None of the faults could trigger a failure of the core.

To summarize, a dynamically reconfigurable fault-tolerant mode is implemented in the ρ-VEX processor which can be activated and deactivated multiple times during a program execution. In this fault-tolerant mode, Hamming codes are implemented for the protection of memory elements, and a non-traditional triple modular redundancy (TMR) approach that can select any three out of four available lanegroups, is implemented for the pipeline protection. A new approach to correct two concurrent errors in a cache word is also implemented in this work. The overhead of the dynamic reconfiguration to the fault-tolerant mode is comprised of pipeline-flush and cache-flush cost. The implementation of this new feature in ρ-VEX is obtained at the cost of about 30% additional resource utilization and 20% reduction in the maximum operating frequency.

6.3 Future work

This section lists recommendations for the future work:

- **Configuration memory scrubbing**
  The fault-tolerant design implemented in this work does not deal with the protection of the configuration memory. This memory stores the design bitstream which defines the functionality of the underlying FPGA device. Upsets in this memory can modify the FPGA design and alter its functionality. Among all types of FPGAs available in the market, SRAM based FPGAs are most susceptible to it. As discussed in Section 2.4.1.2, scrubbing of configuration memory is usually recommended for its fault resilience, be it complete scrubbing or partial scrubbing. To an extent, upsets in the configuration memory of our design will be covered by fault resilience of pipelines and error correction in user memories. However, a dedicated protection scheme for configuration memory like scrubbing is recommended for the future work.

- **Protection for control registers**
  Special-purpose registers also referred as control registers in the report are not made fault-tolerant in this project as explained in Section 3.4.2.1. To protect them, re-designing of all the control registers such that they are implemented in a regular memory fashion is required. This work is suggested for the future work.

- **Optimizing critical path**
  The maximum operating frequency for our fault-tolerant core is 20% less than that of the baseline core. This behavior is understandable as additional components are placed in the longer paths of the core. However, a dedicated effort can lead to optimization of these paths. Section 5.4.2 highlights the longest paths in the core and it can be seen that these paths can be reduced by optimizing the implementation of Hamming code encoders and decoders. Decoders can be optimized if we process the data in parallel with the decoding operation. If no error is detected, which will be the case in majority of the operation, the processor can proceed. If a correctable error is detected, the pipeline needs to be stalled for one cycle so the data processing operation can be performed with the corrected data. This could result in the reduction of longer paths. The only drawback of this approach is that it will affect the CPI as the pipeline needs to be stalled for one cycle each time a correctable error is detected. Anyhow this optimization part is left for the future work.

- **Random fault injection**
  In this work, fault injection at fixed locations at predefined intervals was chosen because of the advantages of controllability and reproducibility. However, this fixation of faults insertion might miss bugs, so fault injection at random locations at random times will be a good addition to the verification suite. If feasible, testing under some radiation test facility is recommended.

Bibliography

[1] V. Castano and I. Schagaev, *Resilient Computer System Design.* Springer International Publishing, 2015.

[2] "Sputnik and the dawn of the space age," https://history.nasa.gov/sputnik/, accessed: 2018-06-29.

[3] R. Ecoffet, "Overview of in-orbit radiation induced spacecraft anomalies," *IEEE Transactions on Nuclear Science*, vol. 60, 2013.

[4] R. H. Maurer, K. Fretz, M. P. Angert, D. L. Bort, J. O. Goldsten, G. Ottman, J. S. Dolan, G. Needell, and D. Bodet, "Radiation-induced single-event effects on the van allen probes spacecraft," *IEEE Transactions on Nuclear Science*, vol. 64, 2017.

[5] D. Binder, E. C. Smith, and A. B. Holman, "Satellite anomalies from galactic cosmic rays," *IEEE Transactions on Nuclear Science*, vol. 22, pp. 2675–2680, 1975.

[6] R. Neogy and C. Siu, "A satellite failure database system." IEEE, 1988.

[7] W. Suparta, "Space weather effects on microelectronics devices around the leo spacecraft environments," *Journal of Physics: Conference Series*, vol. 539, 2014.

[8] E. Normand, "Single-event effects in avionics," *IEEE Transactions on Nuclear Science*, vol. 43, pp. 461–474, 1996.

[9] "Student teams to launch," http://www.esa.int/Education/CubeSats_-_Fly_Your_Satellite/Student_teams_to_launch, accessed: 2018-06-29.

[10] "Tu delft launches delfi-n3xt satellite," https://www.tudelft.nl/en/2013/tu-delft/tu-delft-launches-delfi-n3xt-satellite/, accessed: 2018-06-29.

[11] J. Zipse, J. Flower, T. Mizuo, R. Yeung, B. Zimmerman, R. Morillo, and D. Olster, "A multicomputer simulation of the galileo spacecraft command and data subsystem." IEEE, 1991.

[12] J. E. Tomayko, *Computers in spaceflight: The NASA experience*, 1988.

[13] F. Kastensmidt and P. Rech, *FPGAs and Parallel Architectures for Aerospace Applications.* Springer Science + Business Media, 2016.

[14] J. A. Fisher, P. Faraboschi, and C. Young., *Embedded computing : a VLIW approach to architecture, compilers and tools*, 2005.

[15] D. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 5th ed. Elsevier, 2012.

[16] "Socs with hardware and software programmability," https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html, accessed: 2018-05-23.

[17] D. Saptono, V. Brost, F. Yang, and E. P. Wibowo, “Concept and development of modular vliw processor based on fpga.” IEEE, 2010.

[18] F. Campi, M. Toma, A. Lodi, A. Cappelli, R. Canegallo, and R. Guerrieri, “A vliw processor with reconfigurable instruction set for embedded applications.” IEEE, 2003.

[19] C. Pham-Quoc, B. Kieu-Do-Nguyen, and A.-V. Dinh-Duc, “Adaptable vliw processor: The reconfigurable technology approach.” IEEE, 2017.

[20] V. Nannen, “Exploiting the Recongurability of p-VEX Processor for Real-Time Robotic Applications,” Master’s thesis, TU Delft, the Netherlands, 2016.

[21] S. Wong, T. van As, and G. Brown, “-vex: A reconfigurable and extensible softcore vliw processor.” IEEE, 2008.

[22] C. Bolchini and C. Sandionigi, “Fault classification for sram-based fpgas in the space environment for fault mitigation,” *IEEE Embedded Systems Letters*, vol. 2, pp. 107–110, 2010.

[23] “What is space radiation?” https://srag.jsc.nasa.gov/spaceradiation/What/What.cfm, accessed: 2018-06-01.

[24] “How nasa’s twin radiation belt storm probes work (infographic),” https://www.space.com/17248-nasa-radiation-belt-storm-probes-mission-infographic.html, accessed: 2018-08-14.

[25] “Galactic cosmic rays,” https://www.swpc.noaa.gov/phenomena/galactic-cosmic-rays, accessed: 2018-06-02.

[26] “Solar particle events and radiation exposure in space,” https://three.jsc.nasa.gov/articles/hu-spes.pdf, accessed: 2018-06-02.

[27] H. Quinn, “Radiation effects in reconfigurable fpgas,” *Semiconductor Science and Technology*, vol. 32, no. 4, 2017.

[28] V. Bezhenova and A. Michalowska-Forsyth, “Total ionizing dose effects on mos transistors fabricated in 0.18 m cmos technology.” IEEE, 2016.

[29] M. Nicolaidis, Ed., *Soft Errors in Modern Electronics Systems.* Springer Science + Business Media, 2011.

[30] G. Allen, G. Swift, and C. Carmichael, “Virtex-4 vq static seu characterization summary,” Tech. Rep., 2008.

[31] “Mitigation of radiation effects in rtg4 radiation-tolerant flash fpgas,” https://www.microsemi.com/document-portal/doc_view/126494-msan107-appnote, accessed: 2018-05-31.

[32] G. R. Allen and G. M. Swift, “Single event effects test results for advanced field programmable gate arrays.” IEEE, 2006.

[33] G. R. Allen, G. Madias, and E. Miller, “Recent single event effects results in advanced reconfigurable field programmable gate arrays.” IEEE, 2011.

[34] S. Srinivasan, R. Krishnan, P. Mangalagiri, Y. Xie, V. Narayanan, M. J. Irwin, and K. Sarpatwari, “Toward increasing fpga lifetime,” *IEEE Transactions on Dependable and Secure Computing*, vol. 5, pp. 115–127, 2008.

[35] H. Mushtaq, Z. Al-Ars, and K. Bertels, “Efficient software-based fault tolerance approach on multicore platforms.” IEEE, 2013.

[36] R. Schilling, M. Werner, and S. Mangard, “Securing conditional branches in the presence of fault attacks.” IEEE, 2018.

[37] Z. Chen, R. Inagaki, A. Nicolau, and A. V. Veidenbaum, “Software fault tolerance for fpus via vectorization.” IEEE, 2015.

[38] N. Oh and E. McCluskey, “Error detection by selective procedure call duplication for low energy consumption,” *IEEE Transactions on Reliability*, vol. 51, pp. 392 – 402, 2002.

[39] C. M. Fuchs, T. P. Stefanov, N. M. Murillo, and A. Plaat, “Bringing fault-tolerant gigahertz-computing to space: A multi-stage software-side fault-tolerance approach for miniaturized spacecraft.” IEEE, 2017.

[40] K. Meun, “Fault tolerance on the -vex processor,” Master’s thesis, Delft University of Technology, Delft, Netherlands, 2015.

[41] F. Anjam, “Run-time adaptable vliw processors,” Ph.D. dissertation, Delft University of Technology, Delft, Netherlands, 2013.

[42] D. Pradhan and N. Vaidya, “Roll-forward and rollback recovery: performance-reliability trade-off.” IEEE, 1994.

[43] J. Heiner, B. Sellers, M. Wirthlin, and J. Kalb, “Fpga partial reconfiguration via configuration scrubbing.” IEEE, 2009.

[44] R. Giordano, S. Perrella, V. Izzo, G. Milluzzo, and A. Aloisio, “Redundant-configuration scrubbing of sram-based fpgas,” *IEEE Transactions on Nuclear Science*, vol. 64, pp. 2497 – 2504, 2017.

[45] A. Stoddard, A. Gruwell, P. Zabriskie, and M. J. Wirthlin, “A hybrid approach to fpga configuration scrubbing,” *IEEE Transactions on Nuclear Science*, vol. 64, pp. 497 – 503, 2016.

[46] A. L. Sartor, P. H. E. Becker, J. Hoozemans, S. Wong, and A. C. S. B. Filho, “Dynamic trade-off among fault tolerance, energy consumption, and performance on a multiple-issue vliw processor,” *IEEE Transactions on Multi-Scale Computing Systems (Early Access)*, 2017.

[47] R. Hentschke, F. Marques, F. Lima, L. Carro, A. Susin, and R. Reis, “Analyzing area and performance penalty of protecting different digital modules with hamming code and triple modular redundancy.” IEEE, 2002.

[48] R. W. Hamming, “Error detecting and error correcting code,” *The Bell System Technical Journal*, vol. 29, pp. 147–160, 1950.

[49] M. H. Sulaiman, S. I. M. Salim, A. Jaafar, and M. M. Ibrahim, “A survey of fault-tolerant processor based on error correction code,” in *2014 IEEE Student Conference on Research and Development*, Dec 2014, pp. 1–6.

[50] M. Y. Hsiao, “A class of optimal minimum odd-weight-column sec-ded codes,” *IBM Journal of Research and Development*, vol. 14, pp. 395–401, 1970.

[51] A. Hocquenghem, “Codes correcteurs d’erreurs,” *Chiffres (in French)*, vol. 2, pp. 147–156, 1959.

[52] R. Bose and D. Chaudhuri, “On a class of error correcting binary group codes,” *Information and Control*, vol. 3, pp. 68–79, 1960.

[53] J. C. Moreira and P. G. Farrell, *Essentials of error-control coding*, 2006.

[54] J.S.Chitode, “Information coding techniques,” Tech. Rep., 2007.

[55] I. S. Reed and G. Solomon, “Polynomial codes over certain finite fields,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, 1960.

[56] C. Carmichael, “Triple module redundancy design techniques for virtex fpgas,” Tech. Rep., 2006.

[57] L. Sterpone and M. Violante, “Analysis of the robustness of the tmr architecture in sram-based fpgas,” *IEEE Transactions on Nuclear Science*, vol. 52, pp. 1545 – 1549, 2005.

[58] A. Ahmed, “New fpga blind scrubbing technique.” IEEE, 2016.

[59] J. R. Azambuja, F. Sousa, L. Rosa, and F. L. Kastensmidt, “Evaluating large grain tmr and selective partial reconfiguration for soft error mitigation in sram-based fpgas.” IEEE, 2009.

[60] J. Gracia, J. Baraza, and D. Gil, “Comparison and application of different vhdl-based fault injection techniques.” IEEE, 2001.

[61] Schuette and Shen, “Processor control flow monitoring using signatured instruction streams,” *IEEE Transactions on Computers*, pp. 264 – 276, 1987.

[62] G. Miremadi and J. Torin, “Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection,” *IEEE TRANSACTIONS ON RELIABILITY*, vol. 44, 1995.

[63] G. Miremadi, J. Harlsson, U. Gunneflo, and J. Torin, "Two software techniques for on-line error detection." IEEE, 2002.

[64] G. Kanawati, N. Kanawati, and J. Abraham, "Ferrari: a flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, pp. 248–260, 1995.

[65] W. F. Heida, "Towards a fault tolerant risc-v softcore," Master's thesis, Delft University of Technology, Delft, Netherlands, 2016.

[66] "Vivado - are spartan-6, virtex-6 and older devices supported in the vivado design tools?" http://www.xilinx.com/support/answers/53109.html, accessed: 2018-05-08.

[67] A. Malik, B. Moyer, and D. Cermak, "A low power unified cache architecture providing power and performance flexibility," in *ISLPED'00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (Cat. No.00TH8514)*, 2000, pp. 241–243.

[68] J. van Straten, "A dynamically reconfigurable vliw processor and cache design with precise trap and debug support," Master's thesis, TU Delft, The Netherlands, 2016.

[69] P. Garcia, T. Gomes, F. Salgado, J. Cabral, P. Cardoso, M. Ekpanyapong, and A. Tavares, "A fault tolerant design methodology for a fpga-based softcore processor," *IFAC Proceedings Volumes*, vol. 45, no. 4, pp. 145 – 150, 2012, 1st IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1474667015404574

[70] M. Fujino, H. Tanaka, Y. Ichinomiya, M. Amagasaki, M. Kuga, M. Iida, and T. Sueyoshi, "Fault recovery technique for tmr softcore processor system using partial reconfiguration," 09 2012, pp. 392–404.