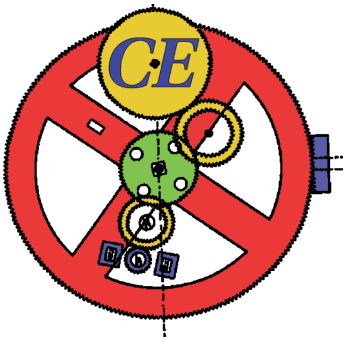


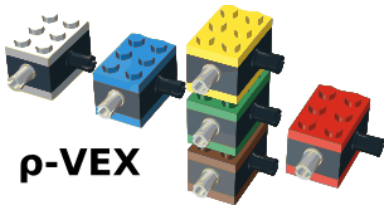
Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2015

MSc THESIS

Fault tolerance on the ρ-VEX processor

KLAAS MEUN



CE-MS-2015-10

Abstract

Increased technology scaling not only resulted in a performance increase of the microprocessor, but also led to increasing device vulnerability to external disturbances. Scaling accelerates ageing induced failures of CMOS devices and the average lifetime of electronic devices diminishes. This thesis describes the design and implementation of a fault-tolerant method to the ρ-VEX processor capabilities in order to counter these effects. The ρ-VEX processor is a Very Long Instruction Word (VLIW) softcore processor with the capabilities to reconfigure the issue width dynamic during run-time. Furthermore, this thesis discusses the implementation of not traditional Triple Modular Redundancy (TMR) with the existing issue width of the core by reconfiguring to a four times 2-issue width. In this method. an important section in the code can be marked and executed in three lock-stepped 2-issue width pipelines. Majority voters are implemented between these three pipelines to vote the results from the executions. At the cost of a 6 % increase in resources and without great structural changes, a new capability is added which allows the ρ-VEX processor to continue running even when it suffers from single-event upsets (SEUs). With the implemented design the amount of errors is monitored using majority voters. Moreover, it is possible to initiate a switch from 2-issue width to the unused issue-width without introducing delay. The design is compared with the original ρ-VEX processor and tested by injecting errors in several units during executing. During these fault injections the TMR was able to correct these errors and complete the program without adding extra cycles.

Project Website: http://rvex.ewi.tudelft.nl/

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Fault tolerance on the ρ-VEX processor

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

KLAAS MEUN
born in URK, NETHERLANDS

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

Fault tolerance on the ρ-VEX processor

by KLAAS MEUN

Abstract

Increased technology scaling not only resulted in a performance increase of the microprocessor, but also led to increasing device vulnerability to external disturbances. Scaling accelerates ageing induced failures of CMOS devices and the average lifetime of electronic devices diminishes. This thesis describes the design and implementation of a fault-tolerant method to the ρ-VEX processor capabilities in order to counter these effects. The ρ-VEX processor is a VLIW softcore processor with the capabilities to reconfigure the issue width dynamic during run-time. Furthermore, this thesis discusses the implementation of not traditional TMR with the existing issue width of the core by reconfiguring to a four times 2-issue width. In this method. an important section in the code can be marked and executed in three lock-stepped 2-issue width pipelines. Majority voters are implemented between these three pipelines to vote the results from the executions. At the cost of a 6 % increase in resources and without great structural changes, a new capability is added which allows the ρ-VEX processor to continue running even when it suffers from SEUs. With the implemented design the amount of errors is monitored using majority voters. Moreover, it is possible to initiate a switch from 2-issue width to the unused issue-width without introducing delay. The design is compared with the original ρ-VEX processor and tested by injecting errors in several units during executing. During these fault injections the TMR was able to correct these errors and complete the program without adding extra cycles.

Laboratory : Computer Engineering
Codenumber : CE-MS-2015-10

Committee Members :

Advisor:	Stephan Wong, CE, TU Delft
Chairperson:	Stephan Wong, CE, TU Delft
Member:	Chris Verhoeven, ECTM, TU Delft
Member:	Arjan van Genderen, CE, TU Delft

Dedicated to Ev.

Contents

List of Figures

List of Tables

List of Acronyms

ALU Arithmetic Logic Unit

AMBA Advanced Microcontroller Bus Architecture

ASIC Application Specific Integrated Circuit

BTI Bias Temperature Instability

CPU Central Processing Unit

CXPLIF ConteXt PipeLine InterFace

DMEM Data Memory

ECC Error Correction Code

EDAC Error Detection and Correction

EDDI Error Detection by Duplicated Instructions

FIT Failure In Time

FPGA Field-Programmable Gate Array

FPGAs Field-Programmable Gate Arrays

FSM Finite State Machine

FTSs Fault Tolerant Systems

GPREG General Purpose Register

HFSM Hierarchical Finite State Machine

HCI Hot Carrier Injection

HWIFI Hardware Implemented Fault Injection

ILP Instruction Level Parallelism

IMEM Instruction Memory

IP Intellectual Property

ISA Instruction Set Architecture

MEFISTO Multi-level Error/Fault Injection Simulation TOol

MMU Memory Management Unit

NOP NO Operation

NOPs NO Operations

PC Program Counter

RAM Random Access Memory

RTN Random Telegraph Noise

SER Soft Error Rate

SEU single-event upset

SEUs single-event upsets

SRAM Static Random-Access Memory

SWIFI Software Implemented Fault Injection

SWIFT Software Implemented Fault Tolerance

TDDB Time Dependent Dielectric Breakdown

TMR Triple Modular Redundancy

VFIT VHDL-based Fault Injection Tool

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

VLIW Very Long Instruction Word

Acknowledgements

First of all, I would like to thank my fellow master student. We all came from different bachelors but we share the same interest. Hugo, thank you for all the help with the CLI and your superior knowledge of linux. Jens, you helped me a lot with VHDL, with your help my VHDL looked better and ran smoother.
I also want to express my thanks to the the PhD candidates. Joost and Anthony, although you had a lot of work on your own I could always come in and ask for help, or just for a silly talk.
And, of course, I want to thank my supervisor Stephan Wong for guiding me through my MSc project. You gave me idea's when I was without any.
My employer, who facilitated this study for me and was patient with my graduation delay.
And finally, the most important one, my girlfriend Evelien. Without you I would have been lost. You would listen to my endless talks about bits that had the wrong value and weird signals. You probably never had a clue what I was talking about, but you kept listening and supported me no matter what.

KLAAS MEUN
Delft, The Netherlands
November 16, 2015

1 Introduction

In the last couple of decades continuous scaling of the transistor resulted in an performance increase of the microprocessor. This trend of scaling from one generation to the next generation in CMOS technology increases the transistor count per unit area. In 1965, Gorden Moore, co-founder of Intel, predicted that the number of transistors in an integrated circuit doubles approximately every two years. This trend is today commonly known as Moore's law [7]. The trend has the benefit, among others, of smaller power consumption and fast switching. On the other hand, it also leads to increasing device vulnerability to external disturbances such as radiation, internal problems such as crosstalk and other reliability problems [8]. More importantly, scaling accelerates ageing induced failures of CMOS devices. The reliability bathtub curve shown in Figure 1.1 elaborates the shrinking of the useful life time of CMOS devices from generation to generation. Every new generation has a shorter lifetime compared to the older technology.

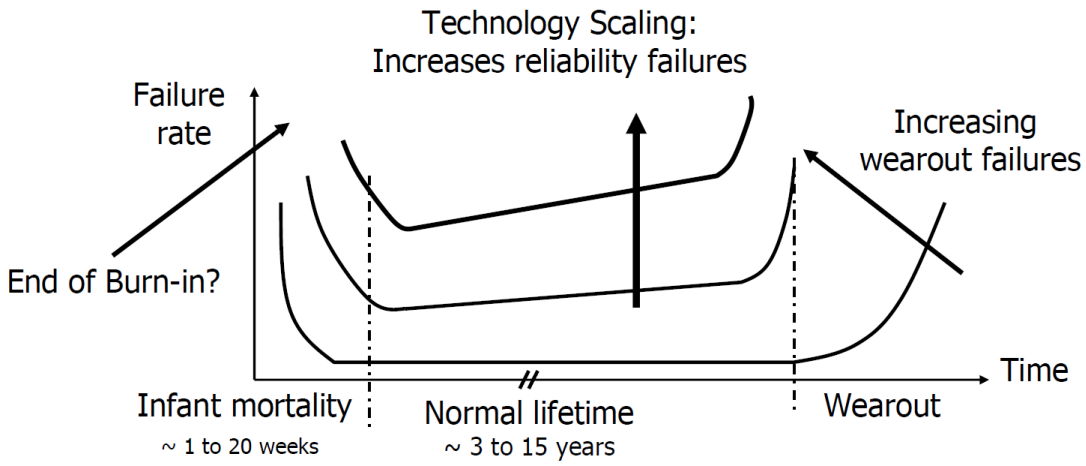


Figure 1.1: Reliability failure shift

These issues of ageing induced failures, radiation, and crosstalk contribute to the increase of *soft errors*, which are also called transient faults or *single-event upset (SEU)s*. These errors in processor execution are due to electrical noise or external radiation rather than design or manufacturing defects [9].
The satellite Rosetta with the Philae lander is a perfect example to illustrate these issues. The Rosetta is powered with an CPU from 1980. [10] Philae is powered with a 16-bit stack processors, again a design dating back to the 1980s. Why was this old technology chosen? For the simple reason that it was radiation hardened and would last long enough

to operate in 2014 when the landing on Comet 67P was executed. As shown in Figure 1.1, NASA had to make sure the equipment would last for at least the ten years it took to get there. The lander also includes an Analog Devices ADSP-21020 and a pair of 80C3x micro controllers as well as multiple Field-Programmable Gate Arrays (FPGAs) on board[11]. During the time of the launch more modern equipment was available. Why is this this "old" equipment from 1980 used in satellites launched in 2004? Is there

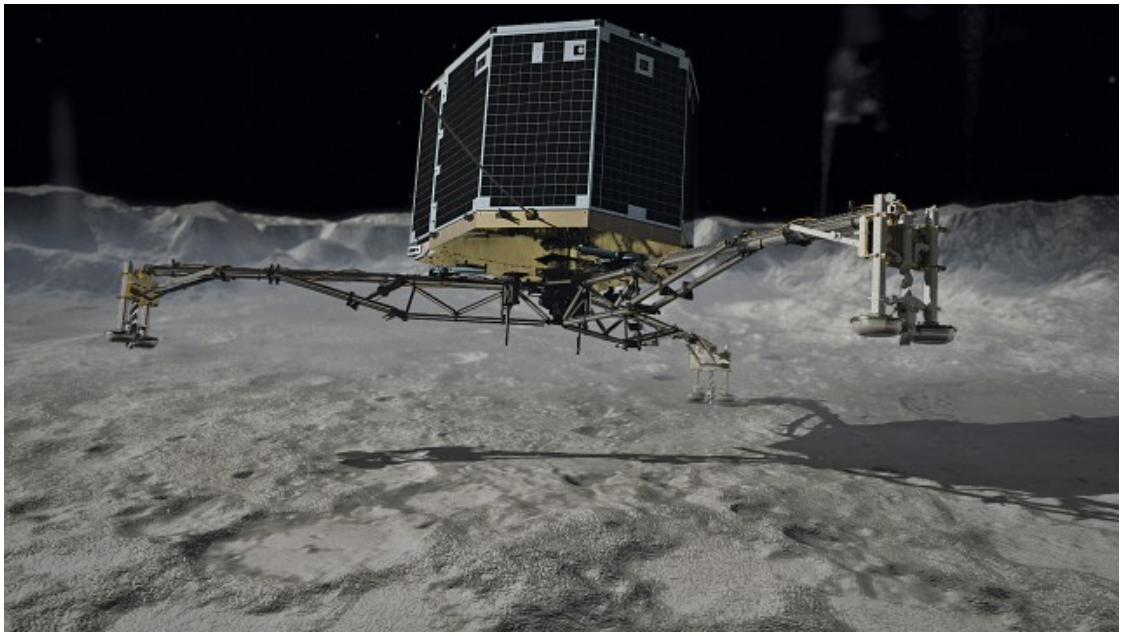


Figure 1.2: Artist Render of the Philae lander (from [1]).

no better solution than this? Can we use modern techniques to counter and protect the equipment from the effects of radiation and accelerated aging?

1.1 Problem Statement and Thesis Goals

The problem statement and thesis goals are meant to be implemented on the ρ-VEX processor Very Long Instruction Word (VLIW) platform. This platform is explained in Chapter 2. The design is build in hardware using VHSIC Hardware Description Language (VHDL). The problem statements is:

How can the ρ-VEX processor be used to enable hardware redundancy against single-event upsets (SEUs) in a fast and efficient way for critical parts that is essential for running a program?

The main goals of this thesis are:

- To find a **dynamic** method to execute a function in a fault tolerant way.
- To implement a low-cost, modular, and easy to use method in a working and running environment.
- To test the proposed solution and prove its correctness.

1.2 Methodology

In order to achieve these goals a research methodology is defined. This methodology represent the steps how to achieve the mentioned goals of the thesis. An abstract representation of the way to achieve these goals are listed as follows:

- Explore the ρ-VEX processor to comprehend the possibilities its capabilities of this special characterized platform.
- Define and research the problem, in this case the SEUs, to choose the best possible solution.
- Explore existing solutions in order to enhance or adapt these techniques to be used in this thesis.
- Invent custom solutions to fit our needs and implement this redundancy technique.
- Test the implemented solution and benchmark this solutions to prove correctness and measure performance.
- Analyze the added hardware, compare the results with the added functionality and determine if this was worth it.

1.3 Thesis Outline

The remainder of this thesis is organized as follows:
Chapter 2 starts with an introduction to the ρ-VEX processor platform and a brief history of the development. It continues with a classification into different categories of all the known errors and its cause, with an emphasis on the SEUs. Then, a basic explanation of the known fault tolerant techniques is presented. Finally, the chapter ends with a overview of how to inject faults in the system to test it's correctness.

Chapter 3 uses the knowledge of the SEUs and combines the characteristics of these errors to the vulnerable structures of the ρ-VEX processor. Together with known techniques and the capabilities of the ρ-VEX processor a solution is motivated for the ρ-VEX processor. A list of required functionalities with the required control mechanisms is filled.

Chapter 4 describes in detail every component that is added tot the design and

the functionality in the whole system. Also the control mechanism is implemented in a Hierarchical Finite State Machine (HFSM) and is explained in detail.

Chapter 5 gives an overview how all components are tested. After this individual test, the complete design is put to the test and compared with the original design. Finally, the system is injected with faults to measure how the system responds to errors during execution.

Chapter 6 summarizes the whole thesis and mentions the main contribution together with the future work.

2 Background

This chapter gives the relevant knowledge to understand the thesis. It starts with describing the platform on which the fault tolerance is implemented, the ρ-VEX processor. Then a classification of the known errors in electronic design are listed. Most categories are only briefly mentioned, but the most interesting errors, the single-event upsets (SEUs) are more elaborate discussed. The second last section discusses the state of the art in fault tolerant design. It is impossible to summarize of all the techniques so only a few are mentioned. Finally methods to inject faults in electronic designs are described.

2.1 The ρ-VEX processor platform

This section is a summary from [12] [4] and [13] to get a basic concept of the operation and architecture of the ρ-VEX processorand its special capabilities.

2.1.1 Very Long Instruction Word (VLIW)

The platform used this thesis is called the ρ-VEX processor. The ρ-VEX processor is a special configuration of a VLIW. A traditional processor has advanced logic to exploit the Instruction Level Parallelism (ILP). ILP is the amount of instructions that can be executed in parallel. A VLIW leaves all the dependencies to be analyzed by the compiler and has no complicated logic to exploit this during run-time. The processor has an issue-width that determines the amount of instructions that are executed in one cycle. The compiler is responsible for determining the dependencies between these instructions. This saves a lot of complicated logic in the core itself. In code that has low dependencies a lot of instructions can be executed in parallel, but if there are a lot of dependencies the majority of the issues are doing nothing[13] or executing a NO Operation (NOP). The lack of the logic to determine the dependencies saves much power and, therefore, makes the VLIW an interesting candidate for embedded applications[14].

2.1.2 VEX and VLIW

The compiler for a VLIW processor is very complex and requires significant efforts and time to develop from scratch. Fortunately, for the VEX ISA, a toolchain is freely available from HP. VEX stands for VLIW Example. The VEX toolchain includes three components[12][4]:

- The VEX Instruction Set Architecture (ISA)
- A parametrized C compiler

- A simulator which can be used for design space exploration and code generation for different implementations of the VEX processor.

This architecture was designed for academic and research purposes. The Computer Engineering group of Delft University of Technology has implemented this VEX architecture using reconfigurable hardware which will be discussed in the next subsection.

2.1.3 Reconfigurable Computing

Convectional computing is performed using two methods. The first,the most commonly known, uses an Application Specific Integrated Circuit (ASIC). ASICs are designed specifically to perform a given operation, therefore they are very fast and efficient when executing this exact operation for which they were designed. After fabrication the design can no longer be altered. When a change is desired, the complete fabrication process must be re-initiated.
The second method is the software programmed microprocessor. Better known as the general-purpose processor which can be found in almost every desktop computer. However, the downside of this flexibility is that the performance of specific application is not as fast as an ASIC.
Reconfigurable Computing fills this gap between the ASIC and the general purpose pro-

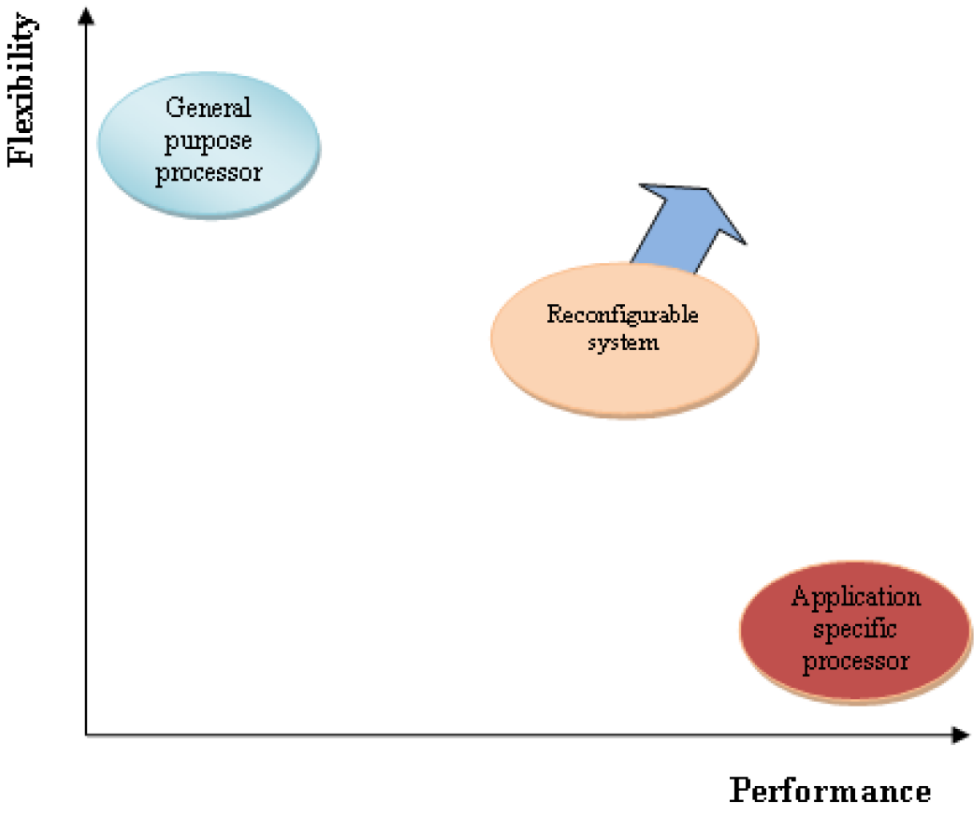


Figure 2.1: Fulling the gap between a GP processor and ASIC (from [2])

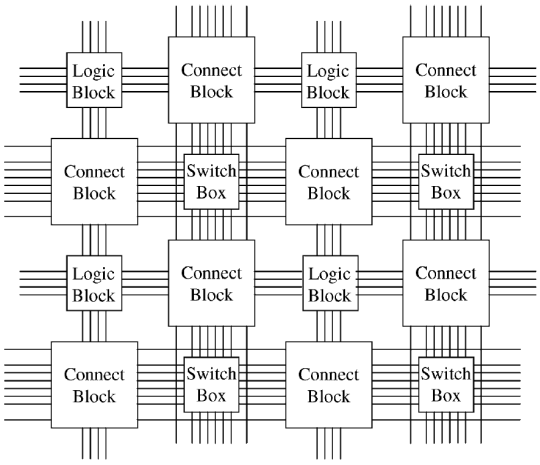


Figure 2.2: A generic island-style FPGA routing architecture (from [3])

cessor as is visualised in Figure 2.1. It has higher performance than software, and a higher level of flexibility than hardware. Reconfigurable devices, such as Field-Programmable Gate Arrays (FPGAs), exist mainly out of arrays of computational elements. These elements, or logic blocks, are connected through separate routing hardware that are also programmable. This way, custom digital circuits can be mapped to form the necessary circuit[3]. A visualisation of these arrays are shown in 2.3. FPGAs are often used for prototyping hardware design using VHSIC Hardware Description Language (VHDL). Another application for FPGAs is accelerating specific applications and algorithms by exploiting specific parallelism. Some High Performance computer incorporate FPGAs like the IBM POWER8 [15] to make use of this capability to run parts of a program on the general purpose processor and parts on the Field-Programmable Gate Array (FPGA).

2.1.4 Reconfigurable Softcores

One of the first mentioned reconfigurable VLIW softcore is the Spyder [16]. There are other well known softcores as the MicroBlaze from Xilinx [17] or the NIOS from Altera[18]. These are better known, but are not reconfigurable. Other examples of reconfigurable softcores are [19] or [20]. These implementations lack the extensibility like like adjusting the issue-width and changing the number of functional units, or the absence of a good software toolchain[21].

2.1.5 ρ-VEX processor: The Delft reconfigurable VLIW softcore processor

The main purpose for a reconfigurable architecture has been touched upon in the previous sections. When a program has high ILP more instruction can be performed in parallel. When this is not the case and a specific program has lot dependencies, the larger part of the processor is not executing instructions. This idling of the processor is of course a waste of power and space which could be put to better use. This is were the reconfigurable

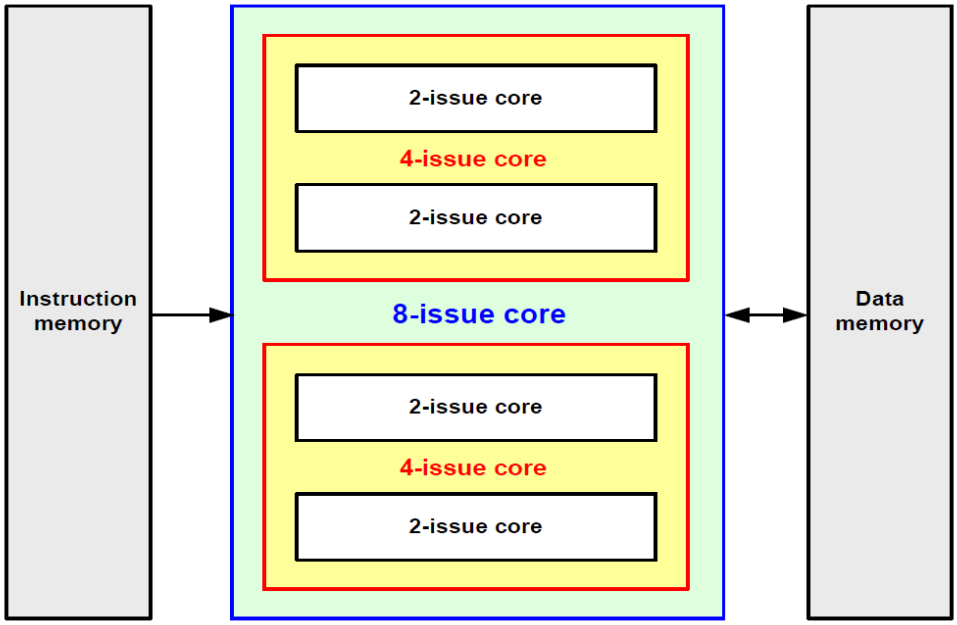


Figure 2.3: The 2-4-8-issue ρ-VEX processor (from [4])

part becomes interesting. If a program has a high ILP, the VLIW is left untouched, but when the ILP is low the idle part of the processor could be used to executed another program.
The ρ-VEX processor can reconfigure during run-time in any desired configuration from the biggest version as an 8-issue width to the smallest version as an 2- issue width as is mentioned in [4] and [13]. During an operation the core can reconfigure from an 8-way to an 2 times 4-way to continue the program which has lower ILP and start running a program in the other 4-way that is available. The two times 4-way has an higher ILP than the first program in an 8-way configuration. This is all possible due to the use of generic binaries[22].
The ρ-VEX processor has currently been rewritten in order to get a more homogeneous style of coding. This has become the starting point for this thesis. Furthermore, progress is being made in the development of an Memory Management Unit (MMU) to sustain an newer version of Linux than the current version of 2.0.

2.2 Target fault

To choose the best technique for our fault tolerant application a deeper understanding of the faults is deemed necessary. In order to achieve this a classification and identification is important in finding and solving the root cause of the faults. Figure 2.4 shows there are three main classifications for fault errors[23][24][25].

1. **Design and Manufacturing failures**. These are faults in the design and manufacturing period.

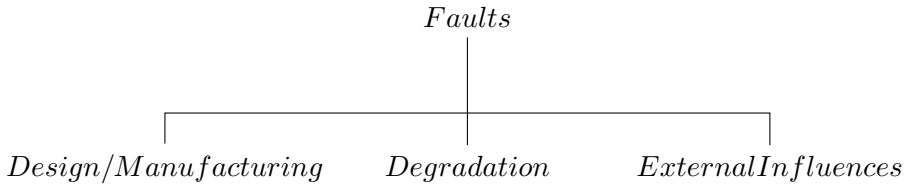


Figure 2.4: Fault classification

2. **Degradation faults**. These are fault due to the aging of the transistors.
3. **Environmental influences**. These are the faults we are interested in. These are cause by the radiations and operating environment of a device.

2.2.1 Design or Manufacturing Faults

Design and Manufacturing faults are faults generated by men in the design of the device, or are caused by the machines that are used in the process of making dies[26]. The most common are:

- Crosstalk
  The increase in transistors due to Moore's Law[7] result in the increase of noise from coupling capacitance between the interconnect wires know as crosstalk[27].
- Process Variations
  During the process of lithography the width, length, or thickness of some layers can vary. As a cause, the voltage and current can show some variations.

2.2.2 Degradation

Degradation are faults that find the transistor. Degradation of the transistor will gradually decrease the performance of device till the moment it will eventually fail when the intended lifetime is reached. The most common are:

- **Bias Temperature Instability (BTI)**
  BTI is the event where the threshold voltage gradually increases over a very long period of time. This will cause the current to decrease
- **Time Dependent Dielectric Breakdown (TDDB)**
  TDDB is also known as gate oxide breakdown. This phenomenon occurs more with the technology scaling. Together with the saturating trend this can cause gaps and lead to gate leakage.
- **Hot Carrier Injection (HCI)**
  HCI is the event where energetic electrons, or holes at the drain end of the channel, overcome the potential barrier at the silicon-oxide interface and inject into the gate oxide. This causes threshold voltage increment, trans-conductance degradation, saturation current decreasing and leakage current increasing.

- **Random Telegraph Noise (RTN)**
  RTN is the noise generated from changing resistance during switching actions.

- **Electro-migration**
  Electro-migration is the noise generated by the large migration of electrons in the interconnect materials. This causes a momentum and an electric field. This results interconnect resistivity change, and interconnect failure such as short and open faults may occur.

2.2.3 External Influences

The most interesting fault of these three are the external influences. These are the faults that are targeted in the fault tolerant mode on the ρ-VEX processor. The cause of these faults are, as the name suggest, found in the environment in which the device operates. Temperature, supply voltage and cosmic rays are the main factors that contribute to the faults in these devices. In the next sections will discussed the how the environmental influences contribute to the failure of these devices.

- **Temperature**
  CMOS devices increasingly dissipate power under the ongoing technology scaling[28]. Extreme heat contributes to the degradation of the device were as temporal and spatial heat variations can affect voltage threshold of transistors which can lead to delays.

- **Voltage**
  The device may suffer from a bad power supply with voltage variations. Voltage variation leads to uneven power dissipation which can result to the formation of temperature hot spot and temperature variation.

- **Latch Up**
  Latch up is a sequence of events occurring very quickly leading up to the SCR state change. Understanding this sequence in detail provides insight which can be used to prevent latch up. For an overvoltage bigger, the worst case would be while the inverter is in the "1" state[29].

- **Soft Errors**
  Soft errors are the errors we are looking for. In the early 80s, IBM conducted a series of experiments to measure the particle flux from cosmic rays[30]. These errors occur from energetic particles such as alpha particles, cosmic ray and thermal neurons. Transistors may accumulate these charges and when a significant amount of charge is collected the state of a logic device is inverted. With this inversion it introduces a logical fault in the circuit's operation. Because, it is not a permanent fault it is called a soft- or transient error[9].

  One aspect of the measurements conducted by IBM is that particles of lower energy occur far more frequently than particles of higher energy. With the current technology scaling[7] less energy from the particles is needed to invert the

transistor. Leading to much higher soft errors.

Two sources are identified as the source of the cosmic rays: First, the flood of low-energy particles from our sun called the solar wind. The solar wind disappears during the period of the quiet sun, and then builds into a torrential storm of particles during an active sun period. This period varies from 9 to 12 years and at it's peak is the main contributor to the rays. Second, is a flux of very energetic particles from far distant sources in the galaxy. The earth is protected against the raw radiation by the magnetic field which deflects these particles, but some may slip through into the atmosphere.

Soft Errors in Memory Circuits
High-energy particles that strike a sensitive region in a semiconductor device deposit a dense track of electron-hole pairs as they pass through a p-n junction. Some of the deposited charge will recombine to form a very short duration pulse of current at the internal circuit node that was struck by the particle. When a particle strikes a sensitive region of an Static Random-Access Memory (SRAM) cell, the charge that accumulates could exceed the minimum charge that is needed to flip the value stored in the cell, resulting in a soft error.

Soft Errors in Combinatorial Logic
If a particle strikes a p-n junction in a part of combinatorial logic it can change the value. However, this will not always affect the result unless it is captured in a memory component. A soft error can cause no effect if it is masked by one of the following three types of masking[9]:

1. Logical masking means when a particle strikes a part of the logic that is not responsible for the output.
2. Electrical masking happens when the charge is absorbed by other logic so that the value is not inverted.
3. Latching-window masking happens when the wrong value is propagated to a memory element, but this is not at the clock transition

The Failure In Time (FIT)/bit of a cell typically ranges between 0.001 - 0.010[31]. This means once every 100-1000 times a bit upset and wrongly represented. In the current technology this has no harm, because this will be restored in the flip-flips after a while. But in an aircraft flying at 10 km height the amount of radiation is 100 times higher. This will affect not only the static logic, but also the combinatorial logic.

Soft Errors on high altitude
Cosmic ray particles that penetrate the atmosphere collide with the atoms and disperse their energy. As we come closer to the earth surface the chance of particles colliding increases because the air become "thicker". When we fly away from the earth surface with an airplane the air becomes thinner. During 1988 and 1989, IBM conducted a series of tests on three different aircrafts in which single-event upset (SEU) in a large array of 64-k SRAM's were measured[32]. These tests

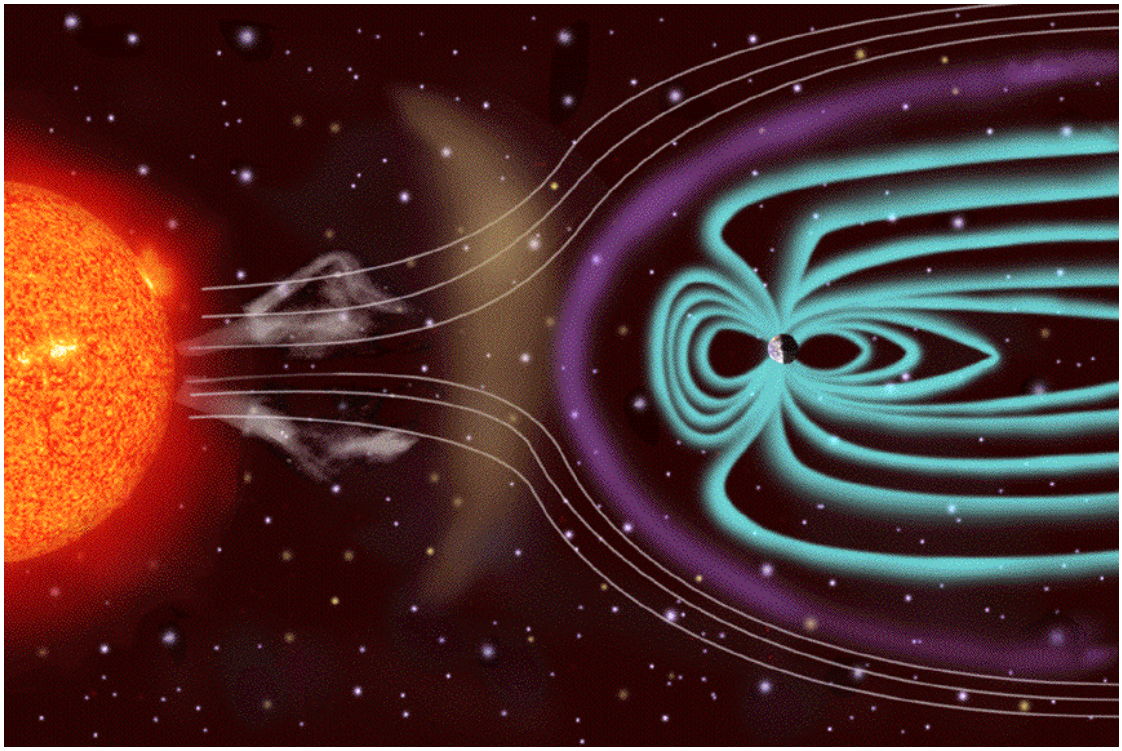


Figure 2.5: The sun's solar flare and the earths magnetic field (from NASA).

demonstrated that SEUs in avionics are real, that the higher a plain flies the higher the amount of particles is. Soft Error Rate (SER) for CMOS devices operated at 2.5 V increased by about a factor of 2 as the altitude was increased from 9 km to 20. This continues until the amount stabilizes.

Soft Errors in Space
Cosmic Ray in space is comparable with the effects on high altitude from the earth surface, but in space there is no protective magnetic field. The main sources of energetic particles that are of concern to spacecraft designers are[33]:

1. protons and electrons trapped in the Van Allen belts,
2. heavy ions trapped in the Earths magnetosphere,
3. cosmic ray protons and heavy ions of multiple elements, and
4. protons and heavy ions from solar flares.

To protect against this radiation external shielding is applied, but the downside of this protection it will add to the weight and volume of the satellite which are limited. In space extra packaging of the chips is added as a shield, but most protection is coming from Error Correction Code (ECC) and other radiation hardening mechanisms. The methods and techniques are discussed in Section 2.3.

2.3 Fault Tolerant Techniques

As discussed in Section 2.2, there are many ways how a device can produce one or more errors. To take in account all these type of errors and especially SEUs in this section, a selection is made to describe the most common known fault tolerant techniques. The section is divided, with first the software based techniques, followed by the hardware techniques.

2.3.1 Software Fault Tolerance Techniques

While reliable systems typically employ hardware techniques to address soft-errors, software techniques can provide a lower-cost and more flexible alternative. Software techniques generally use the functionality to execute the same instruction multiple times and compare the results. This can be done in parallel or sequential. As an example we look at Error Detection by Duplicated Instructions (EDDI) demonstrated in [34], but the basic technique is also used in Software Implemented Fault Tolerance (SWIFT) in [35]. This software based fault tolerant technique uses the compiler to implement the method to execute the same instruction twice. In a VLIW this can be executed in parallel, but the check whether the outcome is the same is always sequential. When examining [36] a predecessor of EDDI the same principle is introduced not by use of the compiler, but by the programmer. Another type of software fault tolerance is called Error Detection and Correction (EDAC)[37]. This software implementation of the error control coding calculates a control word from the in- or output data. A codeword contains extra check bits that are used for error detection and correction. After reading data from external memory, the data is checked for possible errors and corrected. The decoded data is then ready to be used. This is generally implemented in hardware as shown in the next section, but a software implementation is also possible.

2.3.2 Fault Tolerant Techniques in Hardware

Hardware techniques to enable fault tolerance often include adding extra hardware and execute double or triple times. The most common fault tolerance techniques are:

1. ECC
   ECC is a method, by adding some extra bits to a signal which are used to detect is the original signal has changed. The simplest example is the parity bit calculation[38]. This is commonly used in memories, but can be implemented in all sorts of signals like the register or floating point units [39].

2. Duplication
   Duplication is a method similar to the software implementation, only in this case the instruction is not executed twice, but executed once in a unit that is duplicated. This does not require the compiler to adjust the program, but the hardware takes care of this. The hardware can also check if the outcome is the same, or this can be done in software

3. Triple Modular Redundancy (TMR)
   TMR is applied in many air-, spacecrafts and satellites. The technique dates back tot the 60s[40][41]. The technique is comparable to duplication, but now executed thrice. This has the benefit that with a binary outcome there is always majority in the outcome.

The automotive industry uses an other approach. It uses hardware from an older generation. This is similar as the use of an 1980s Central Processing Unit (CPU) in the Philae lander mentioned in Chapter 1. Older hardware has bigger transistors and bigger interconnect signal wires, because at that moment the technique to create smaller parts was not invented yet. The bigger electrical circuits have a longer lifespan which is needed in cars. The technology has proven to withstand the wear and tare during that lifetime. The downside of this method is that it lacks the speed of the current technology, but for the automotive industry this is not a bottleneck.

Fault Tolerance techniques boils down to two methods. We either execute the operation multiple times and compare the outcome, or we strengthen the current operation by shielding, with ECC or with older robuster technology. In software the copied operation is often displaced in time while in hardware this operation can be done in parallel.

2.4 Fault Injection

Fault injection is a method to validate Fault Tolerant Systems (FTSs) by injecting them artificially with wrong data. This can be done manually of automated or with automatic tools[42]. Fault injection is defined as follows:
Fault injection is the validation technique of the Dependability of FTSs which consists in the accomplishment of controlled experiments where the observation of the system's behavior in presence of faults is induced explicitly by the written introduction (injection) of faults in the system.
The techniques can be divided in three main categories:

- Hardware Implemented Fault Injection (HWIFI) This method is a physical disturbing. The hardware is put in an environment were there is heavy ion radiation for example (or electromagnetic interference, etc.) Or incorrect values are connected to the circuit pins[42][43].

- Software Implemented Fault Injection (SWIFI) This method is completely done in software as the name imposes. In software the errors are recreated that can occur in the hardware. For example the modification of memory data[43].

- Simulated fault injection: In this technique, the system under test is simulated in another computer system. Faults are induced altering the logical values of the model elements during the simulation.

HWIFI and SWIFI are not suitable because of the lack of the equipment or software tools. For this thesis the simulated fault injection techniques are further investigated.

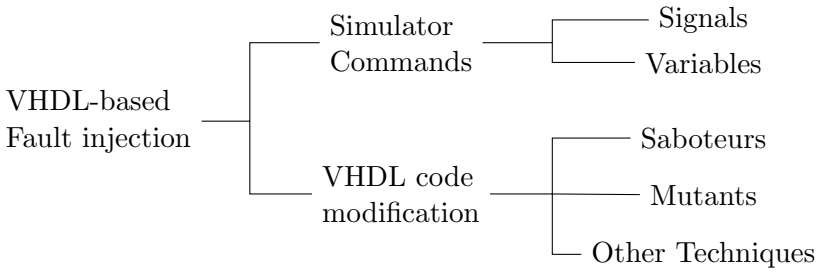


Figure 2.6: VHDL-based fault injection techniques (from [5])

The ρ-VEX processor is a VHDL based core and this technique is most suitable for the ρ-VEX processor.
Figure 2.6 gives a classification of different techniques that can be used to inject faults in a VHDL model. Simulator commands technique is based on the use of the simulator commands to modify the value of the model signals and variables without altering the VHDL code, for example in ModelSim you can force a signal from 0 to 1 in a simulation. VHDL code modification techniques change the model, adding saboteurs or using mutants of the model components. Other techniques include extending the VHDL language[44], but these are out of scope. Each method will be discussed briefly.

2.4.1 Simulator Commands

This fault injection technique is based on using the simulator to force the signals to another state. Moreover, as VHDL generic constants are managed as special variables, it is possible to inject some non-usual fault models, such as delay faults[45]. Using simulator commands it is possible to inject transient, permanent, and intermittent faults. Though, there exists one restriction: due to the special nature of variables in VHDL, it is not possible to inject permanent faults in variables. This technique is the easiest one to implement and its temporal cost (to perform the simulation) is by far the lowest. However, the number of fault models that can be injected is smaller than with the other techniques.

2.4.2 Saboteurs

A saboteur is a VHDL component that changes the value or timing characteristics of one or more signals when activated. It is usually inactive during normal system operation and activated only to inject a fault. A "serial saboteur" breaks up the signal path between a driver (output) and its corresponding receiver (input). A parallel saboteur is simply added as an extra driver to a signal[46]. According to [5] this distinction in saboteurs is expanded in eight different saboteurs.

2.4.3 Mutants

A mutant is a component which replaces another component. In [47] and [48] some examples of this technique is discussed. It works like the original component, but when

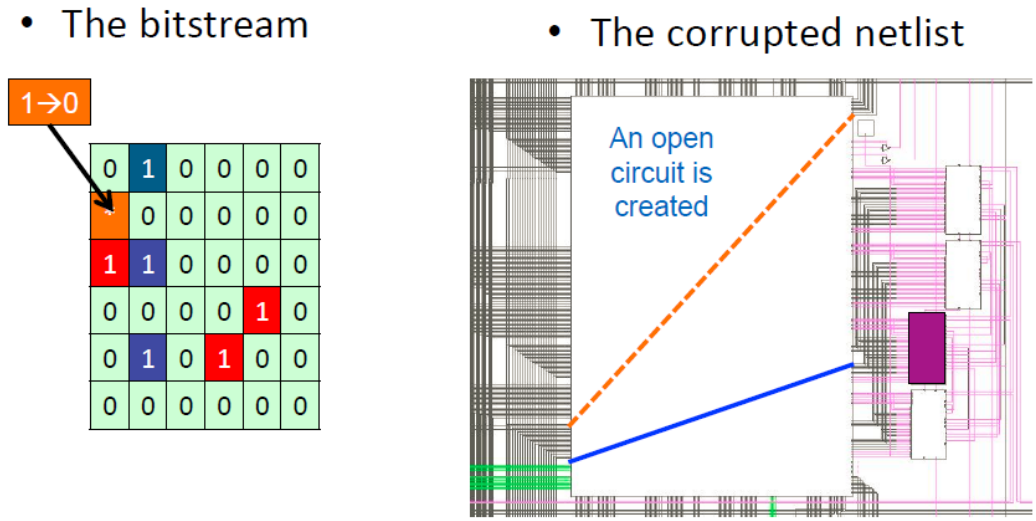


Figure 2.7: Adjustment op bit-stream to create an open (from [6]).

made active it generates faulty output. The mutation can be made in three ways:

- by adding saboteurs to structural model descriptions;
- by modifying structural descriptions replacing sub-components (i.e., a NAND gate can be replaced by a NOR gate)
- by automatically mutating statements in behavioral component descriptions, e.g., by generating wrong operators or exchanging variable identifiers; this is similar to the mutation techniques used by the software testing community.
- by manually mutating behavioral component descriptions to achieve complex and detailed fault models.

There can exist lots of possible mutations in a VHDL model, so representative subsets of faults at logical and RT levels must be considered.

These different types of fault injection can be automated with some tools like Multi-level Error/Fault Injection Simulation TOol (MEFISTO)[46] or VHDL-based Fault Injection Tool (VFIT) but these are all closed source and difficult to compare.

2.5 New Fault Injection

Because all of the tools to inject faults in the designs are closed source a manual adjustment of the VHDL looks promising, but still is a lot of manual labor. Therefore, a new state-of-the-art method is introduced: VERI-place[6][49]. The VERI-Place tool is able to perform the analysis and the placement of circuits on modern SRAM-based FPGAs.

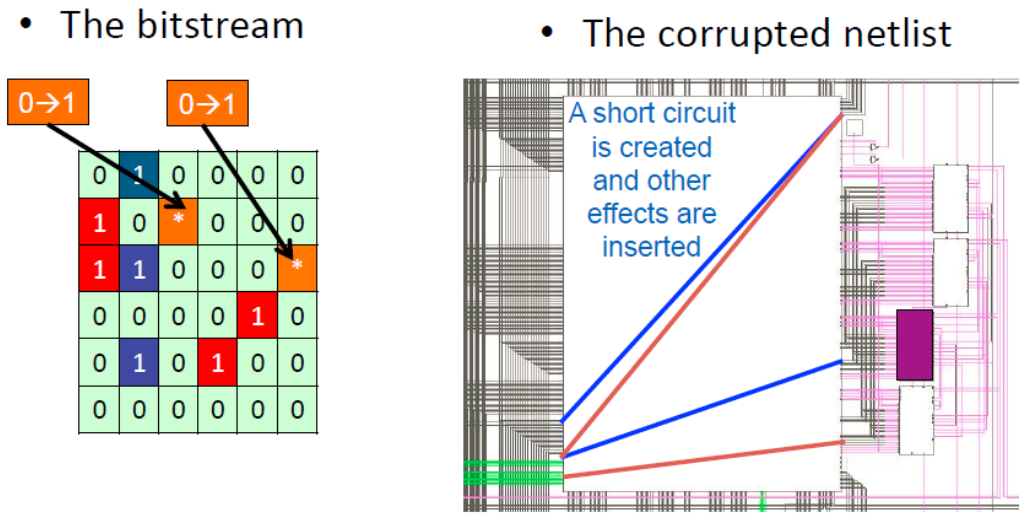


Figure 2.8: Adjustment op bitstream to create a short and other effects (from [6]).

The VERI-Place tool consists of an algorithm which loads the circuit description implemented on the SRAM-based FPGAs and performs the topological analysis. It parses the original bitstream file and generates a set of bitstream coordinates that correspond to the identification of the FPGA configuration memory cells possibly affected by SEUs.

Although this is a permanent fault and not a transient fault, after generating a lot of tests with different affected bit-streams this could generate an statistic sound image of the effects of SEUs. Compared to an automated tool with fault injection this method needs approximately 100 times more test to achieve viable results. The upside from this method is that it is suitable for use and not closed source.

2.6 Conclusions

This chapter presents the ρ-VEX platform as a version of a VLIW processor. The ρ-VEX processor is the only run-time reconfigurable VLIW softcore with a complete toolchain. It makes use of the benefits of reconfigurable computing and fills the gap between the general purpose core and the ASIC. With the use of FPGAs the softcore can dynamic change the issue-width if the running program and execute an other program on the free issues.

Subsequently, a classification is made from all the causes of fault in electronic equipment. This can be summarized in three main groups: Design- and manufacturing faults which are caused during the design an manufacturing period and before the real use. The second group are faults from degradations that is caused by the wear and tear of the transistors. Finally, the most important group is the faults caused by environmental issues. The most important is the error caused by charged particles or called the *soft error*.

Theses faults and errors are countered with fault tolerant techniques in software and in

hardware. This results in two most common approaches. Either, we execute the operation multiple times and compare the outcome, or we strengthen the current operation by shielding, use ECC or with older robuster technology.
Finally this chapter presents methods of how to inject faults in a design to verify if the system operates as intended. The most interesting method is VHDL-based fault injection, because this is best applicable in VHDL-based softcore. Also a new state-of-the-art method of testing by modifying the bit-stream in order to simulate SEUs called VERI-Place is discussed.

3 Design Considerations

The hardware platform used for this project is built around the reconfigurable ρ-VEX processor as discussed in the previous chapter. In this chapter, we use the knowledge acquired about the ρ-VEX processor, the error and the existing fault tolerance methods to find a custom method tailor-made for the platform. First, we will start with the vulnerable structures were the probability of the error to appear is the highest. After we acquired the location, possible methods to prevent the errors are considered and the best-suited option for the ρ-VEX processor is chosen to implement. The chosen implementation is then further investigated and a list of needed structures is put together with the required data. Finally, a list of control mechanisms is composed to integrate the design in the existing system so it can be called and used from the existing infrastructure.

3.1 Vulnerable Structures

Theory says memory structures are most vulnerable to be hit by a particle that causes a single-event upset (SEU), therefore we need to protect these structures [50][51][9]. In the ρ-VEX processor, the memory structures are located in the General Purpose Register (GPREG), the Instruction Memory (IMEM), the Data Memory (DMEM)and the normal registers in, for example the pipeline stages. All these memory structures are threatened by this event. This does not mean that it cannot occur in combinatorial logic as is stated in [9].

3.1.1 GPREG

The GPREG is a memory structure which can be hit with a charged particle. This charged particle can invert a bit, but it will also store this incorrect value and use this in future calculations. Multiple methods are to protect the memory against this event as discussed in Section 2.2. The ρ-VEX processor has a reconfigurable general purpose register and implements this in block Random Access Memory (RAM)s. This register is accessible for all pipelines when operation in the 8-issue width mode. When operating in a 4 times 2-issue width with each pair has their own context. The general purpose register is separated and only accessible by their context. Because of this reconfiguration option a lot of logic is needed. One method to protect this again single-event upsets (SEUs) is to implement this three times and use a majority voter[52]. This is not an option, because the amount of space needed for the logic of accessing GPREG is already a substantially part of the whole core and it will explode the amount of space needed or this.
A viable solution would be Error Correction Code (ECC) which has already been used in designs [39] and is also commercial available. Therefore we assume the GPREG as save and not subjected to SEUs.

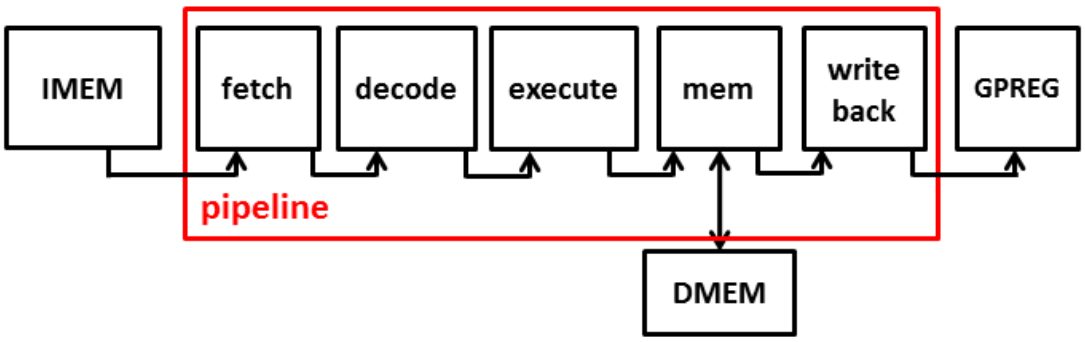


Figure 3.1: A simple representation of the pipeline

3.1.2 IMEM

The IMEM is also a memory structure that is vulnerable to SEUs. As the previous subsection suggests a triplication is an option at the cost of a lot of space. The ρ-VEX processor implements this in available block RAMs. Again this type of memory can be easily protected against these errors with existing technology like ECC or Error Detection and Correction (EDAC). This is already available, thus we assume that the information from the IMEM is correct.

3.1.3 DMEM

The DMEM is implemented on the ρ-VEX processor in block RAMs like the two memories described above. For the same reason as above we assume this information is correct, because implementing a method to protect this against SEUs is trivial.

3.1.4 Pipeline

The pipeline as shown in Figure 3.1 has memory structures between the stages implemented in registers. There are eight pipelines in the ρ-VEX processor implemented. In this situation the proposed method as suggested in [52] already proposes a triple register implementation with a voter. This could be a viable option. The option to use techniques like ECC or EDAC are not possible to implement in these registers.

3.1.5 Combinatorial logic

Although the main place for a SEU to occur is a memory element, combinatorial logic is also subject to SEUs. This means that an error can occur in the functional units. This small error will have a big impact, because it is used very often. Protecting against these errors is harder and more difficult. Techniques like ECC or EDAC are not applicable, thus this leaves only multiple execution as a viable option.

3.2 Implemented Protection

As is concluded in the above section the IMEM, DMEM and GPREG are implemented in block RAMs and a protection against SEUs is considered trivial, because multiple options already exists. Thus, we will consider these elements as protected. This leaves the registers in the pipeline and the functional units unprotected.

To implement protections in the ρ-VEX processor we want to make use of the special characteristics and implement a purely hardware-based fault tolerant execution. This will imply that the core becomes bigger, because we will add structures. We need to take in consideration to keep this as small as possible. Extra area means extra use of power and we want to prevent this as much as possible.

Area
: The basic principle in fault tolerance can be applied here. Make units and signals more robust, by adding ECC for example, or do the operation multiple times and compare the result. Making the units more robust has a disadvantage making great and structural adjustments to the existing core. As mentioned in Section 1.1 this does not meet the requirement to keep it modular.

Multiple execution
: Executing operations multiple times is very good possible because the ρ-VEX processor can operate in a 4 times 2-width issue. It already has the redundant units available in the other pipeline pairs. And it is possible to reconfigure during operation. So the basic principle of executing multiple times and compare the result can be applied.

Compare
: The protection of the functional units already suggests multiple execution. This can be double execution with a compare of the results, but a better option would be Triple Modular Redundancy (TMR) with a majority voter, because this eliminates the compare step. A majority voter selects from three binary inputs the majority as outcome. The advantage of operating in TMR mode is that if there is an SEU that is captured in a unit the operation will continue, without needing too rollback and execute again.

With the above argumentation in mind the option to operate in a 3 times 2-issue width configuration of the ρ-VEX processoris selected as best solution. With this configuration the TMR principle can be applied. This leaves one 2-issue width core available. The remaining core can be used as a spare pipeline. When one pipeline receives more errors than expected from SEU something more than soft errors is causing the errors. This could be one of the other causes of errors mentioned in Section 2.2.

3.2.1 Voting in the Pipeline

When operating in TMR there comes a point when the three signals need to be voted with the majority voter. When applying this to the pipeline more information about the pipeline is needed:

Stages
: A simple representation of the pipeline is shown in Figure 3.1. It has 5 stages (Fetch, Decode, Execute, Memory, Write-back). The first option that comes to mind is to implement a voter after every stage in this pipeline. This is a viable option, but this means adding a lot of logic. Also, the pipeline operations are not so separable as the theory suggests. The Arithmetic Logic Unit (ALU) and multiplier are so interconnected that adding a voter in between is impossible without changing the structure radically. The other option is to vote everything that comes in the pipeline and that leaves the pipeline. This will have the same effect as the other option, but is smaller and more modular, because it will leave the pipeline as it is.

Incoming Data
: Incoming data to the pipeline can be found in Figure3.1 and in Table 3.1. These are the instructions from the instruction memory, the data read from the registers and the read data from the memory.

Outgoing Data
: Outgoing data from the pipeline are the write to the register and the write to the memory

This means a voter for the instructions, memory, and the registers. Three units, compared to four in the previous proposal, and the latter leaves the pipeline intact. In Table 3.1 "Required Structure" is added this gives more information about the functionality. Read in the same data from the IMEM, DMEM or GPREG is extra work. A simple copy to the other lanes will suffice.

Table 3.1: Overview of in- and outgoing data from the pipeline with required structure

Type		Required Structure
Incoming data	IMEM	Copy
Incoming and outgoing	DMEM	Copy and Voter
Incoming and outgoing	GPREG	Voter

3.3 Control Structures

Only adding some voter or copy structures will not be sufficient. Some control structures need to be implemented. First the TMR of fault tolerance mode needs to be enabled from the code. Second, some information is required about the error in the active lanes. Finally, actions need to be taken with that information.

3.3.1 Protected Sections

The fault tolerance mode needs to protect important sections in the code. The code needs to tell the control system when to enable the fault tolerant mode and when the section is ended. For this purpose a pragma[1] when to start the TMR en when to disable the TMR is needed. When enabling the TMR the core needs to reconfigure from the current configuration to the configuration to a 4 time 2-issue width. This reconfiguration pragma already exist, so extending this reconfiguration with an TMR is needed to call, and end this from the code. This requirement is added to the list of functionalities in Table 3.2.

3.3.2 Detecting Errors

Operation in TMR leaves one 2-issue width pipeline available. This remaining pipeline can be used as a spare pipeline. When one pipeline receives more errors than we would expect from SEU something more than soft errors or SEUs is causing the errors. This could be one of the other causes of errors mentioned in Section 2.2.
Thus, we need to measure the amount of errors. When using a voter structure as mentioned in [52] the outcome gives a majority result but gives no information about the different input signals. To overcome this, the output signal needs to be compared with the input signal. To see if the input signal was different or the same. If this was not the case the input signal was faulty. This is added to Table 3.2 with required structures.

3.3.3 Counting Errors

When for example the ρ-VEX processor is exposed to a testing environment en bombarded with charged particles, the SEUs generate a lot of errors. These will be evenly spread among the other lanes. The other case is that due to some process variations the transistors in the first lane pair has some degradation issues. In the latter case the amount of errors in the core is not evenly spread, but concentrates in one location, or lane. With these two simple cases in mind the mechanism for counting errors is not a simple counter and see when a certain limit if errors is reached. If we use a simple counter the second example will give the right information and the first lane pair reaches the count limit first. This gives the indication that the lane is broken. But, when applying the same technique for the first example the information we get with a limit is meaningless. The amount of errors is evenly spread and it depends on coincidence which pair reaches the limit first. But this gives the indication that the lane pair is broken which is not true.
For this reason we do not use the absolute value of the fault, but we incorporate an other counter that will decrease the amount of fault from all the lanes. This way we keep a relative value of all the error and not an absolute value. Also, this will prevent that the counter should be capable of storing an infinite number when large programs are run. This functionality is added to Table 3.2.

[1] A pragma is for compiler directives that are machine-specific or operating-system-specific, i.e. it tells the compiler to do something, set some option, take some action, override some default, etc. that may or may not apply to all machines and operating systems.

Table 3.2: Overview of required structure

Function	Required Structure
Start and Stop TMR	Extend reconfiguration option
Detect Errors	Extend voter structure
Decide to switch lane	Fault count mechanism
Implement lane switch	Finite State Machine (FSM)

3.3.4 Switching Lanes

When established that a lane is broken a switch is initiated to the lane that is in spare at that moment. This needs to be done with care. Current operation in the pipeline need to be finished and adding extra delay is not desired. One stage at a time in the pipeline needs to be switched to the spare lane. The voter structures in stage 4 and 5 need to be signalled that a switch is to be executed. This timing sensitive switching needs to be handled with a FSM. FSMs can become very large and complex. To enable easy modifications in the future and avoid the complexity a choice is made to use a Hierarchical Finite State Machine (HFSM). This will keep the top-level simple and the second level of states, can be easily managed and extended if needed. This functionality is added to Table 3.2.

3.4 Conclusions

This chapter focuses in the vulnerable structures that are most likely to be hit by a SEU. These are often memory elements. Known countermeasures are found in similar designs which implement ECC or EDAC in the memory elements, therefore, we assume the IMEM, DMEM and the GPREG correct, because with a trivial existing solution these memory elements can be protected. The emphasis is on triplicating the pipeline and enable TMR. This method requires an element to copy the instruction three times and a majority voter on all the incoming and outgoing data from and to the DMEM and GPREG. Finally, the mechanism to count the errors is discussed to decide if the spare lane is used and a lane switch is initiated. These operations require a dedicated control mechanism to detect the errors and a FSM to handle the lane switch.

4 Implementation

This chapter describes the implementation of the different structures and functionalities determined in Chapter 3. The order in which the implementation is presented follows the chronological way the data flows. It begins with how the sections are marked as protected in the program and how the fault tolerance is turned on and off. Then, the system is explained in how the instructions flow through the three pipelines which form the Triple Modular Redundancy (TMR). At every stage were a majority voting is needed these are explained. An overview of the implemented system is shown in Figure 4.1. The different colors represents the different sections

4.1 Configuration Control

In Section 3.3 was argued that the fault tolerance need to protect sections of code by using a pragma. The pragma we use is an extension of the existing reconfiguration control. This is activated by writing to a register. The register is called the reconfiguration register, named after the function that it can reconfigure the core. Each active context can request a reconfiguration. The request is read by the decoder. This decoder interprets this register and translates this to a new configuration. The decoder needs to be extended in order to get the correct setup of the core. The desired configuration is a four times 2-issue width were all the cores want to read the same context and in the end configure

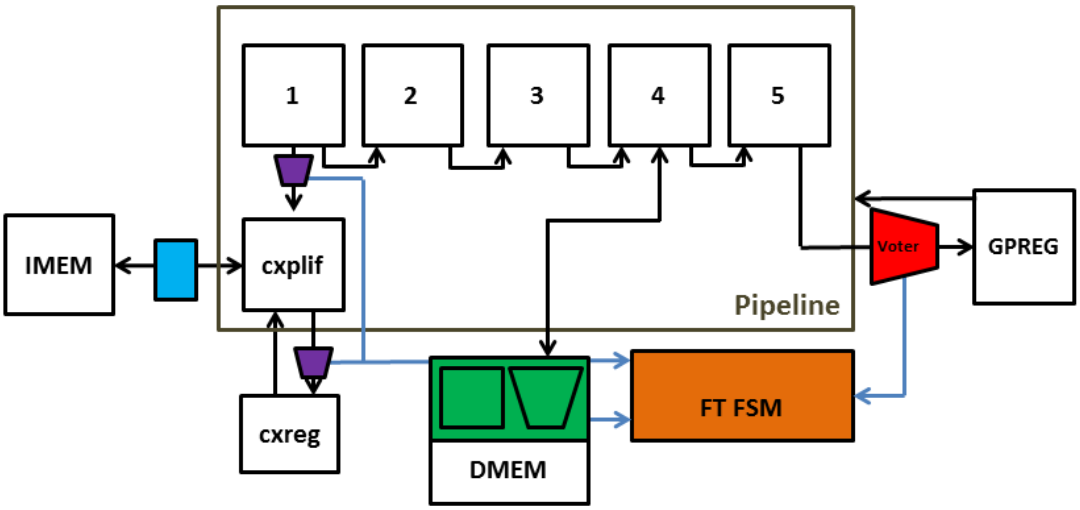


Figure 4.1: General overview of the implemented system

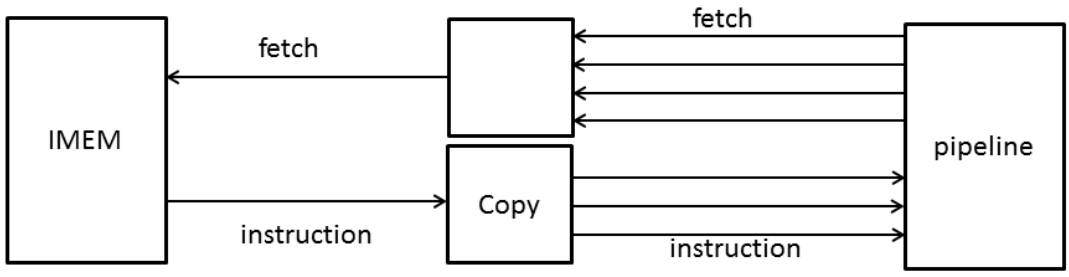


Figure 4.2: Instructions from the IMEM tot the pipeline

back to a normal configuration. After writing to the register in the code the core will wait five cycles to execute the last instructions and reconfigure to a four times 2-issue.

4.2 Instruction Multiplexer

In Section 3.2 the data that was coming from the Instruction Memory (IMEM) to the pipeline was declared save end needs to be copied to the other lanes. The function is schematically represented in Figure 4.2 and the position in the whole system is the blue square in Figure 4.1. First, this unit filters the four incoming fetch signal in to one fetch command from the core to the instruction memory. This signal is a signal from the pipeline to fetch the new instruction. Second function is dividing these instructions meant for only one lane over all active lanes in the core and feed the inactive lane NO Operations (NOPs). This could be incorporated with a voter, but there is no sense in this. All instructions come from the same place. There are no operations between the IMEM and the pipelines. When adding a voter we add extra logic which increases the chance of an error. Also we introduced a new configuration in Section 4.1 with 4 lanes who want to read the same data. These 4 read actions are now reduced to one single read request. This method can me compared to a read request to the data memory. In Section 4.5 this situation is solved in the same manner.

4.3 Voter Structure

In Section 3.3 detecting the error was needed for deciding if a lane switch was initiated. To enable this the design of a majority voter needs to be extended. Triplication of the execution will generate three signals. From these three signals a finite signal must be generated. As is already mentioned in Chapter 2, a previous version of the ρ-VEX processor incorporated TMR with a majority voter in the registers [52] [4]. This design compared every bit from the three signals and passed the most voted signal. This would be a zero or a one. This technique can be used to protect the complete pipeline and accommodate triple operation.
A small adaption to the structure is implemented as shown in Figure 4.3. The output

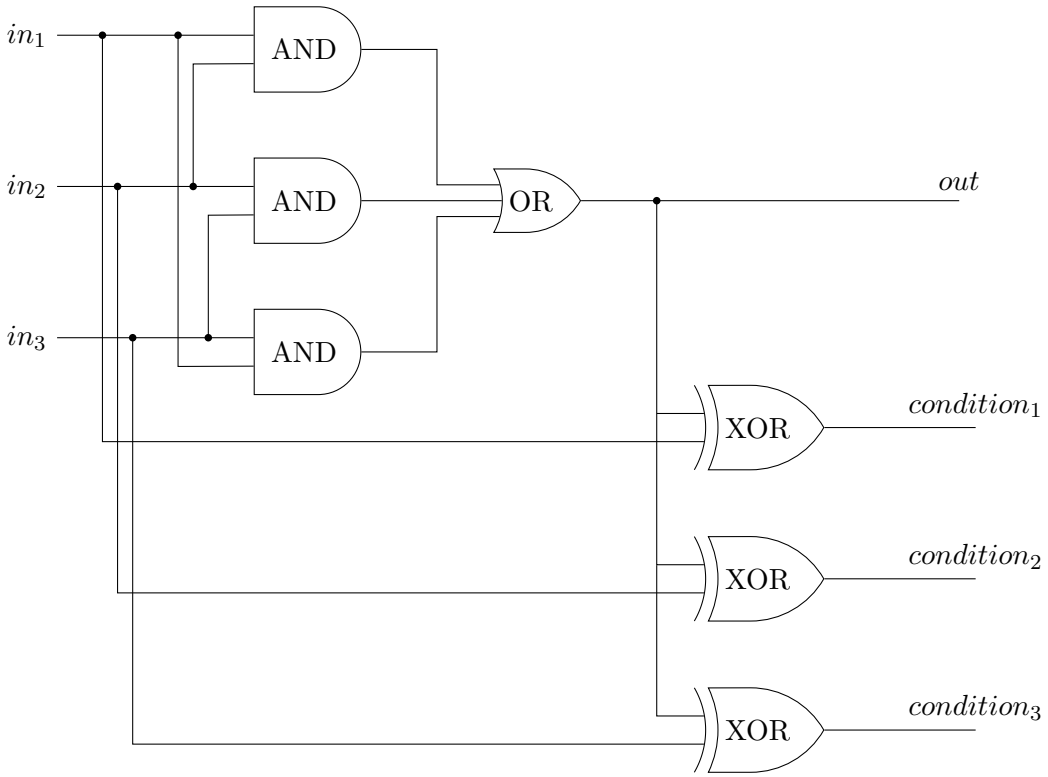


Figure 4.3: Voter structure

signal is compared with the three different input signals with an XOR-port. When there is a difference between the input and the output this will generate a '1' in the conditions. With this small adjustment the voter structure not only implements the standard majority voter, but generates information about the incoming signals. These incoming signals can be used to make assumptions about the state of the pipeline.
Figure 4.3 is implemented on bit-level, when zooming out and view this from a higher level to Figure 4.4 a complete voter mechanism is shown. We assume four incoming signals where three contain signals from the active lanes. The first case is when the fault tolerant mode is disabled. Then we connect the incoming ports to a signal and connect the signal to the outgoing ports. This way the signal only travels through the entity without modification.
The other case is when fault tolerant mode is enabled. The incoming signals are connected to the voter and only the three active signals are considered. The result of the voting is then used in the output of the entity. Depending on the function this result is copied back to the active lanes or not. The signals in Figure 4.4 marked as c_0, c_1 and c_2, represent the signal $condition_0$, $condition_1$ and $condition_2$ in Figure 4.3, but then for the whole signal and not the single bit. In figure 4.1 the structures are shown as a trapezium.

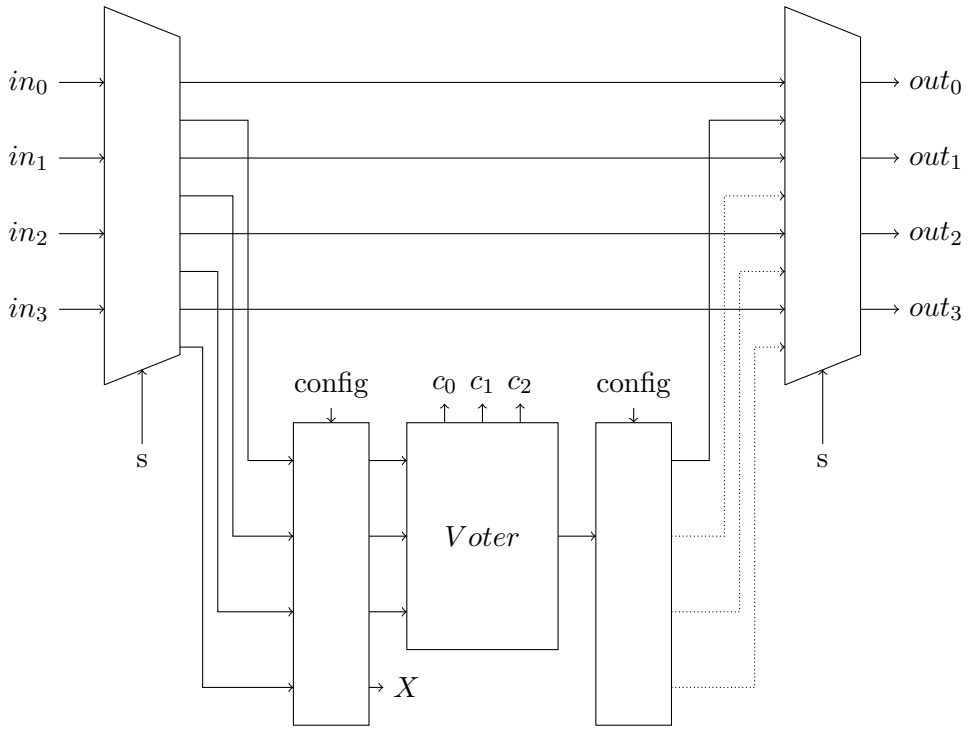


Figure 4.4: Voter mechanism

4.4 Program Counter Voter

In Section 3.2, a decision was made to only use all the in coming an outgoing signals of the pipeline. There is one exception being made to this. The branch unit is responsible for determining the next Program Counter (PC). This is based on the current PC. Every Branch unit determines the next PC on it's own and send the result to the ConteXt PipeLine InterFace (CXPLIF) as shown in Figure 4.5. In this unit the instructions are requested or fetched from the IMEM. This next PC is broad-casted to the context registers. The signal is fed through a voter, so all the next PCs are the same. This part is important because when for some reason the branch unit of a lane determines the wrong PC this has great consequences for the execution of the code. One lane can jump to a complete different part in the code that is executed. As a result the lane will execute the wrong instructions and produce the wrong outgoing data to the register and to the memory. This will never happen of course because the previous mentioned unit makes sure all the instructions to the pipelines are the same. The PC however is determined inside the pipeline. If a wrong branch is taken, the PC will stay wrong until a new branch occur and the lane will jump to the right place. To prevent this, an extra voter structure is implemented to check all the PC that are written from the branch units to the other lanes. This way if a branch unit takes the wrong branch the PC is restored to the correct value and not at the next point in the code where it needs to branch. The conditions from the two voters are connected to the other voter. Only one signal is used in the Finite State Machine (FSM).

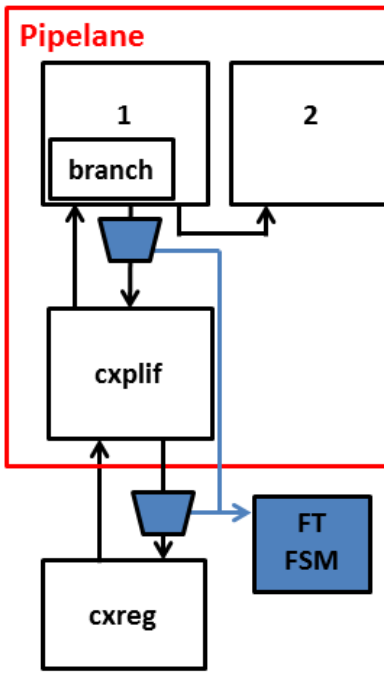


Figure 4.5: The two voters for the program counters

4.5 Memory Voter

In Table 3.1 we noticed that the data from and to the data memory needs two operations. A copy and a majority voting operation. The memory voter consists of two parts. First part is reading of memory and the second part is writing to memory. The writing part is voted and the reading operation is copied.

4.5.1 Memory Reading

Memory reading during fault tolerant mode is done with three lanes at the same time. If this read action was untouched all lanes want to read the same address at the same time. The memory will schedule this and serve the different lanes one at a time and stall the waiting lanes. This is unwanted because this will disrupt the lock-step state all lanes are in. Therefore, a memory read action is implemented with multiple voter structures as shown in 4.3. All four signals are voted to one signal to the Data Memory (DMEM). A read action is performed in two cycles. These two cycles are visualized in Figure 4.6. The action is initiated by the lane with enabling the *Read_Enable* signal together with an address of the data that needs to be read. This *Read_Enable* signal together with the address is "voted" and reduced to one read request to the memory from the first pipeline. This request is returned with the required data. As the request is made only from one lane, this needs to be distributed amongst the other lanes. No voter is needed, just a simple copy action amongst the active lanes.

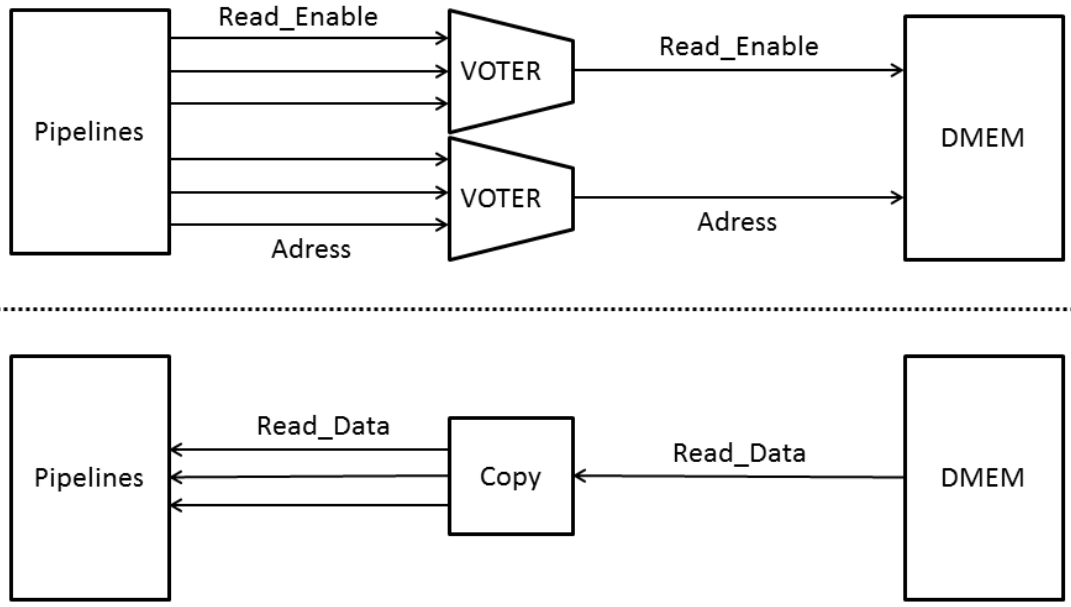


Figure 4.6: Memory read operation in two cycles

4.5.2 Memory Writing

Memory writing during fault tolerant mode is also done with three lanes. This looks the same as the read action. All three lanes want to write the same data to the same address at the same time. If this is left untouched the DMEM will stall the other cores and will write the data one per cycle. The solution in this case looks the same as the writing solution and is visualized in Figure 4.7. Four voter structures are used to check if the three lanes want to write the same Write_Enable, Address, Write_data, and Write_Mask. These voter structures are implemented with multiple voter structures as shown in Figure 4.3. After voting the data is written only by the first lane with the correct data to the memory.

4.6 Register Voter

The register of the ρ-VEX processor, as presented in Section 2.1, is special in the case that it is reconfigurable to a way it can be shared or not depending on the configuration. In this case we use the possibility that it can be shared. All lanes can read a value and all lanes can write a value to the registers if they are set to the same context. For reading from the register no special adaptations are needed. All lanes can read because of the special construction, so no stalls occur or other disruption operations that can trigger the lanes to be forced out of the lock-step.
For writing the same technique with a voter structure as shown in 4.3 is used. This element is shown in the red color in Figure 4.1 as part of the whole system. Three lanes try to write a value. All lanes raise their write enable signal and present the data end

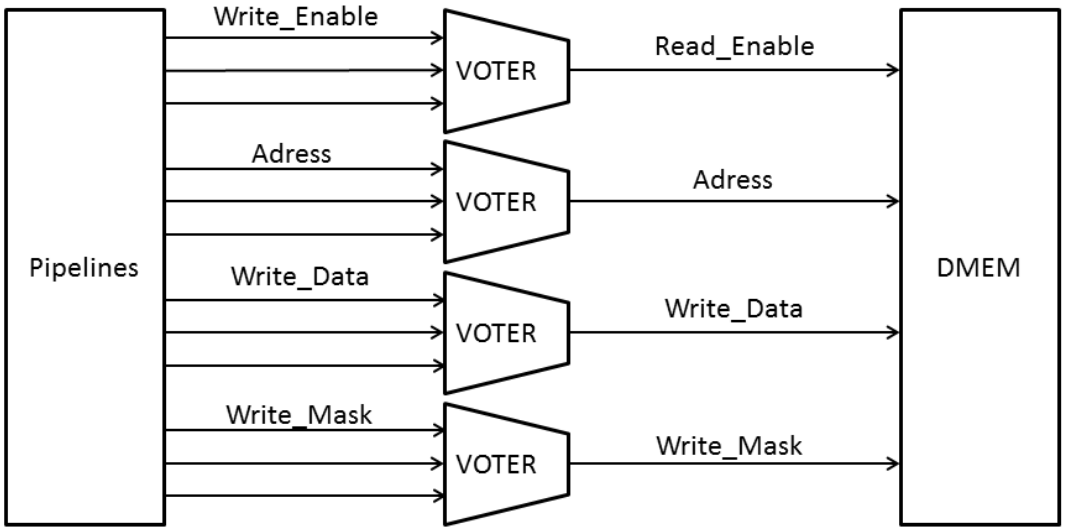


Figure 4.7: Memory write operation in two cycles

the address in which register the data needs to be placed. All signals are vote and in the end only one lane will have the *writeEnable* raised as is shown in Figure 4.8. This data is distributed in the General Purpose Register (GPREG) so others can read the value.

Forwarding Structure
When comparing this tot the Memory Write actions as shown in 4.7, the signals are still copied to the other lanes and not only one, except for the writeEnable. This is done because, the forward mechanism must not be disturbed, or we will loose significant performance. For this reason we will write the back the correct value in the pipeline to

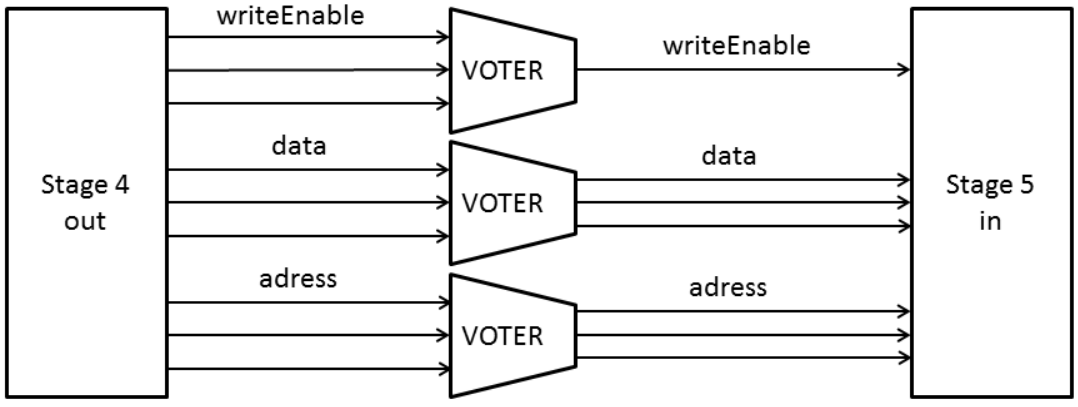


Figure 4.8: Register write operation

the active lanes, but the write enable is only kept in one lane. With this modification we ensure that all lanes keep their forward logic and ensure the correct value is distributed among the lanes.

This is critical for the algorithm where we detect errors. For example, if a lane wants to write the wrong value to the register, this is corrected in the voter. When this is not corrected in the specific lane, the forward logic may propagate this incorrect value to other stages and cause more errors. In this situation one error may produce one or more recognized errors. To compensate for this the correct value is written to all the lanes. In this situation one error will only be recognized as one error.

4.6.1 Fault-Count Mechanism

In Section 3.3 we argued to implement a smart mechanism to count the errors during use. To implement this we create a buffer or a counter. One error does not cause a failure due to the TMR structure. In other words, the system can accept an error without further actions. Of course, if the number of errors gets to much we must assume something is wrong and the lane must be marked as broken.
The algorithm is straight forward. The active lanes are monitored and the amount of instructions is monitored with a simple counter. If the amount of errors within a certain amount of instructions is to high a signal is generated to the FSM to mark it as broken. For example, if more than 3 errors are detected within 3000 instructions a signal is generated. But if it is less than those 3 the buffer will decrease after 3000 instructions with one. After a good amount of instructions without errors the buffer will remain empty. The reason for using a decrease mechanism is to make no difference between long programs ans short programs. If looking at an absolute value, longer programs run longer and will endure more single-event upsets (SEUs). With this mechanism this effect is gone. As an addition to this the four counters are routed through to the control register so this can be easily read from software.

4.7 Hierarchical Finite State Machine

In Section 3.3 the control of the TMR and the lane switching was entrusted in a FSM. During implementation a better solution presented itself in using a Hierarchical Finite State Machine (HFSM). The reason for choosing an HFSM is that is relatively simpler to expand and adapt[53]. The top level is kept very simple in only four basic states:

1. State Idle
   This state is active after a reset, when the fault tolerant mode is not active and normal operation is enabled. All units are set on pass-through. This pass-through is one of the experiments to ensure the added functionality has no influence on the existing functionality.

2. State TMR
   This state is designed for the fault tolerance mode in TMR mode. This has multiple sub-states to enable the switch to the spare pipeline that is not used during

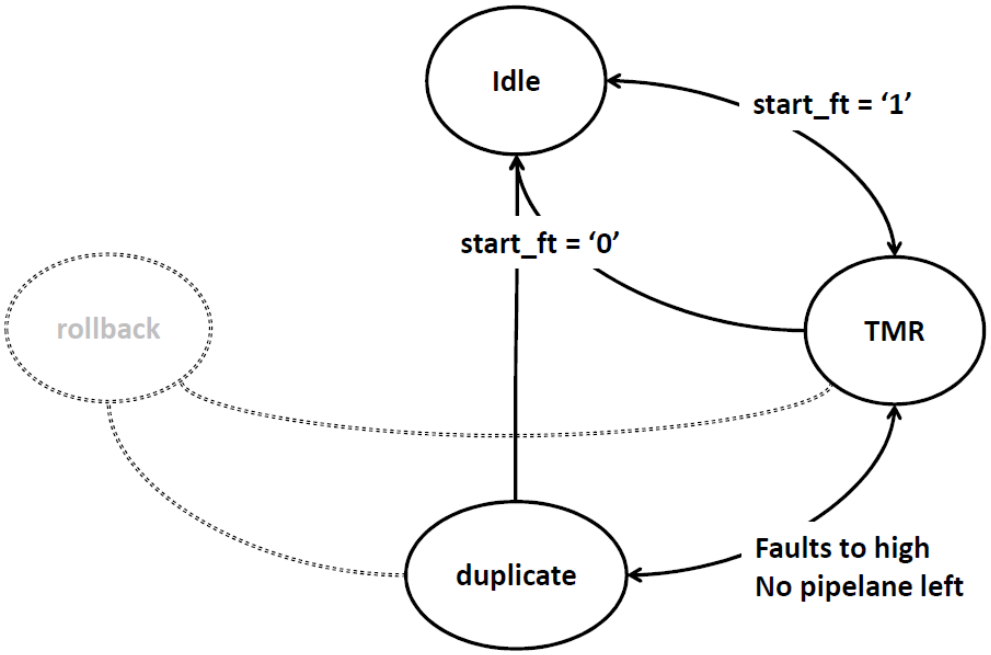


Figure 4.9: State diagram of the top-level FSM.

operation. The functionality of the state is further explained in the paragraphs below.

3. State Duplicate
   This state is designed as a last resort when it is clear that 2 lane pairs are broken and only two can be trusted. During this state TMR still continues to function, but in the signal from the voter only the two active signals are used. When this comparison also fails a trap can be triggered to stop the core, but this last part is not yet implemented.

4. State Rollback
   This state is designed as an expansion unit. When State TMR and State Duplicate have passed this is the only option left. When there is an error in the comparison of the two remaining lanes there is no majority in voting. A rollback is needed to execute this again an compare the results for a second time. This rollback has much implications and is implemented as Future Work to enhance the current work.

Sub-state TMR

The sub-state TMR handles delicate part in this FSM is to enable dynamic control over the spare lane and the broken lane. It is easy to make an FSM and create all the cases, but that would mean that the amount of states will explode. When using dynamic saving of the spare lanes, the active lanes and the broken states this can become smaller in code and in logic.
To achieve this a simple basic *exclusive or* and some *and* operations are executed on

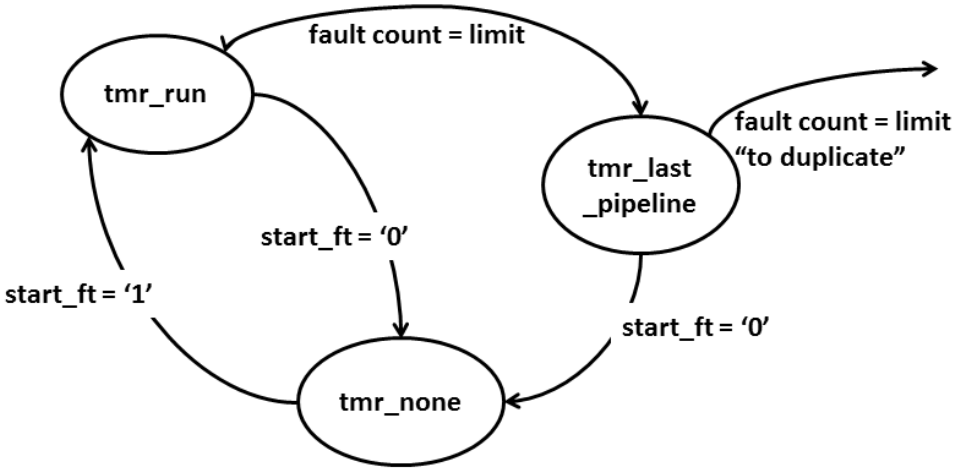


Figure 4.10: State diagram of the substates in TMR state

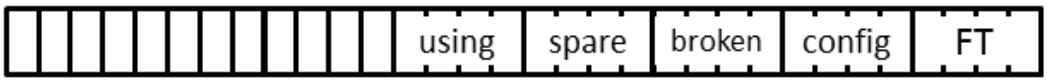


Figure 4.11: Organization of the statusvector

some arrays and this is stored to determine the next state. The input of this FSM depends of course if fault tolerance is enabled or not. Furthermore, the next state depends on the current using lanes, the spare lanes and the amount of errors that are measured. Based on these values is decided if the current configuration is maintained or a switch is needed to use the spare lane, or if only two lanes are used.

Control Signals
When the core is implemented on a Field-Programmable Gate Array (FPGA) it is impossible to track the signals inside. For this purpose some signals are lead to the outside so they can be read from the software. The fault that are registered in the faultcounter are wired to control registers in the core. Another signal is created to show the status of the fault tolerance control mechanism. This vector is shown in Figure 4.11.

1. The first four bits are reserved for the fault tolerance enable signal. This is current only one bit, but there is room reserved for more options.
2. The config-signal which represents the lanes that are active.
3. The broken-signal represents the lanes that generate to much faults and are marked as broken
4. The spare signal represents the lane that is in spare when no lane switch has been performed.

5. The using signal is the signal that is currently being used.

This signal can be read form the software and is used to see the status of the system. At the moment only 20 of the available 32 bits are used. The remainder can be used in future work.

4.8 Saboteur

To perform a decent test of the system a manual adaptation of the code is made. To simulate a single-event upset (SEU) a saboteur is created as mentioned in Subsection 2.4.2. The setup is kept as simple as possible and is visualized in Figure 4.12. The complete VHDL-code is shown in Appendix A. The unit need a clock-signal and a reset to enable the counter inside the unit. In the program code a register can be set as limit. When the counter reaches this limit the saboteur will sent a signal. This signal can be used to invert some bits for example. When executing the code with different limits set, it is easy to adjust the amount of errors injected in the design.

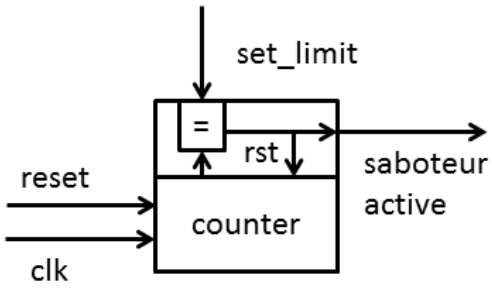


Figure 4.12: Schematic structure of saboteur

4.9 Challenges

During the implementation some challenges appeared which are hard to categorize but are worth mentioning. These seemingly small issues are hard to find but are essential for a proper function of the whole system.

- **Configuration Control.**
  To avoid generating all the logic needed to determine all the decoded signals combinatorially, this unit contains a state machine to do it in multiple cycles. The challenge is to write this down in a structure that generates less logic and is fast.

- **PC Distribution.**
  The PC determination is a delicate thing in this process. Because the branch units act on the same context, they all read from the same register. The register that is read is determined by the lane that is the active one and not by the ones

that are added as fault tolerant. In the example we use $CR\_CRR = 0x9990$; the active lane is indicated by the zero. The remainder is added for extra execution. If this lane is deactivated for some reason, like the amount of faults is to high. The branch registers become also inactive. This will cause the branch units not to function properly anymore. To circumvent this the lane is kept active, but is only performing NOPs. This could be done nicer, but a complete redesign of the branch unit was needed. This is against the principle to keep the fault tolerant layer as modular as possible.

- **Timing** The *memvoter* and *regvoter* require delicate delay in timing. The simple solution to make a chain of registers was eventually the best solution, but if the timing is one off, this might introduce extra faults

4.10 Conclusion

This chapter describes how the elements are implemented in the design. First, a global overview is created and then the complete mechanism is chronological explained. Firstly, the configuration control enables the ρ-VEX processor to change in an other setup, then the instructions are triplicated to three planes. The PC voter is responsible that the correct PC is broad-casted to the other pipelines. The memory voter and register voter make sure the write and read to the DMEM and GPREG is voted and copied. The HFSM is explained and the custom made saboteur is introduced. Finally. the chapter is concluded with some challenges during implantation.

5 Experiments

This chapter is where the implemented design as decided in Chapter 3 and implemented in Chapter 4 is tested and verified. To verify and prove that the system we produced does what it is supposed to do, the following actions are taken. First, we verify that every component individually works as expected. Each component will be put to the test and the outcome is discussed. After this individual test the whole system will be tested. This means that it should execute the programs correctly. Then the system is injected with faulty data and measured how the system handles these faults. Finally, an analysis is made about the added hardware and what influence the added hardware implies. All tests are performed on the Vertex 6 ML605 Evaluation Kit with the standalone core without cache memory and without stop-bit implementation.

5.1 Testing the Components

The testing follows the structure of Chapter 4. The order in which the components are implemented is used as order in which the components are tested. As reference Figure 5.1 is re-added. Testing of the components is done using a custom made control register. It is just a simple 32 bit long vector, but if we lead it to the different entities, we can use it to manually enable these entities. The different tests are organized by first mentioning the name of the component, then a theoretical way to test is proposed. The actual test is presented and the outcome is shown. An overview of the tests is shown in Table 5.1.

5.1.1 Configuration Control

The configuration control is the first unit to test. To test this properly all units are disabled and the fault tolerance pragma is written to the reconfiguration register. Then the core should reconfigure and execute the program four times.

Test

The test is executed by stating this in the c-code with the command $CR\_CRR = 0x9990;$. Because all other units are disabled the core will no longer function in lock-step. The result from test is that the uart[1] produces 4 times *success* as output. This test is done in simulation as well as implemented on an Field-Programmable Gate Array (FPGA). Due to the behavior of the uart the word *success* can be mixed with the other word, but a simple count of all the letters gives us each letter 4 times.

Outcome: Successful

[1] A UART (Universal Asynchronous Receiver/Transmitter) is the microchip with programming that controls a computer's interface to its attached serial devices. Specifically, it provides the computer with the RS-232C Data Terminal Equipment (DTE) interface so that it can "talk" to and exchange data with modems and other serial devices[54].

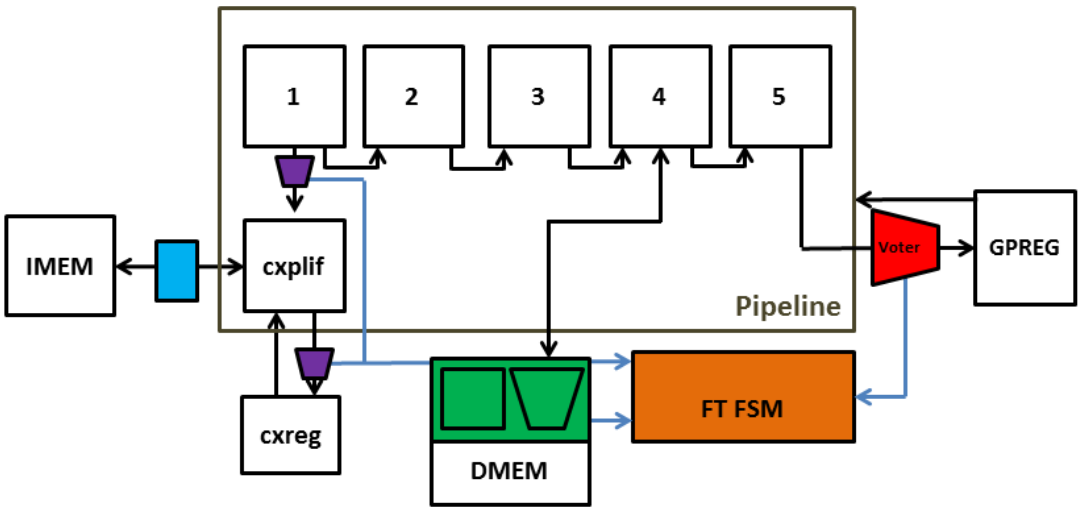


Figure 5.1: General overview of the implemented system

5.1.2 Instruction Multiplexer

The entity to get the instructions from the Instruction Memory (IMEM) can be tested with a short piece of assembly code. We will write the one set of instructions and see if the instructions are copied to the other lanes.

Test

The multiplexer is enabled with a control signal that is added specific for these tests. When it is enabled we only write instructions to the first set off lane. Part of this assembler code is shown in Listing 5.1.

Listing 5.1: Part of the assembler for testing the instruction multiplexer

```
# after the start enable the instruction multiplexer manually
c0 mov $r0.2 = 0xF
;;
c0 stw CR_FTT_ADDR[$r0.0] = $r0.2
;;
# wait some cycles to for the voters are actually on
.....
# start with filling some numbers in the register
c0 mov $r0.2 = 1
c0 mov $r0.3 = 2
;;
c0 mov $r0.4 = 3
c0 mov $r0.5 = 4
;;
c0 mov $r0.6 = 5
c0 mov $r0.7 = 6
```

```
;;
# perform some operations
c0 stw -4 [$r0.1] = $r0.2
;;
# disable the plexer manually
c0 stw CR_FTT_ADDR[$r0.0] = $r0.0
;;
```

The test showed the exact same instruction in the other lanes.
Outcome: Successful

5.1.3 Program Counter Voter

To get the correct Program Counter (PC) to all the pairs we use a voter structure to distribute the PC. An observation is made of the counter when all lanes operate in lock-step. We test this unit in normal operation and see if the unit corrects the PC.
Test
Again some assembler code as shown in Listing 5.2 is used. All units are disabled and only the PC voter is enabled. We write only instructions to the first pair. The PC should be overwritten to zero, because the majority of the counters is zero.

Listing 5.2: Part of the assembler for testing the program counter voter

```
# after the start enable the instruction multiplexer manually
c0 mov $r0.2 = 0xF0000
;;
c0 stw CR_FTT_ADDR[$r0.0] = $r0.2
;;
# wait one cycles to for the voters are
.....
# start with filling some numbers in the register
c0 mov $r0.2 = 1
c0 mov $r0.3 = 2
;;
c0 mov $r0.4 = 3
c0 mov $r0.5 = 4
;;
c0 mov $r0.6 = 5
c0 mov $r0.7 = 6
;;
;;
# disable the multiplexer manually
c0 stw CR_FTT_ADDR[$r0.0] = $r0.0
;;
```

After the execution we see the voter overwrite the PC to the majority of inputs.
Outcome: Successful

5.1.4 Memory Voter

The functionality of this component has two parts as explained in Section 4.5. This means the test is also divided in these two parts. First, we start with a memory write, followed with the memory read operation.

Memory Write
We test the functionality by enabling the voter and see if three the same write operations are merged in one write operation. We also test the voter by adding one fault value.
Test
To test the Memory Voter in writing a small piece of assembly is written to test only this part of the core. A part is shown in Listing 5.3. The assembly first write to a control register. This control register is fed to the entity to manually configure the entity. Then two times three inputs are generated.

Listing 5.3: Part of the assembler for testing the memory write function

```
# after the start enable the memvoter manually
c0 mov $r0.2 =  0xF000000
;;
c0 stw CR_FTT_ADDR[$r0.0] = $r0.2
;;
....
;;
# perform some operations
c0 stw -4 [$r0.1] = $r0.2
c0 nop
c0 stw -4 [$r0.1] = $r0.3
c0 nop
c0 stw -4 [$r0.1] = $r0.2 # this is wrong
;;
# perform some store operations to the memory
c0 stw -8 [$r0.1] = $r0.3
c0 nop
c0 stw -8 [$r0.1] = $r0.3
c0 nop
c0 stw -8 [$r0.1] = $r0.2 # this is wrong
;;
# disable the voters manually three times
c0 stw CR_FTT_ADDR[$r0.0] = $r0.0
c0 nop
c0 stw CR_FTT_ADDR[$r0.0] = $r0.0
c0 nop
c0 stw CR_FTT_ADDR[$r0.0] = $r0.0
c0 nop
;;
```

The resulting write signal to the Data Memory (DMEM) was correct and only one single write. Next to this result the voter registered the wrong inputs in the faultcounter.
Outcome: Successful

Memory Read
The purpose of this test is to see if three memory read requests are merged in only one request. Furthermore, we want to see if the read data is copied from the lane that executed the merged read to the other lanes.
Test
As expected this part is done in almost the same way. Again we use some simple assembly code to enable only the memory voter. Some values are then written to the memory and after enabling the fault tolerant mode this is read from the memory. Part of this code is shown in Listing 5.4.

Listing 5.4: Part of the assembler for testing the memory read function

```
# start with filling some numbers in the register
    c0 mov $r0.2 = 1
    c0 mov $r0.3 = 2
    ;;
    # perform some operations
    c0 stw -4 [$r0.1] = $r0.2
    ;;
    c0 stw -8 [$r0.1] = $r0.3
    ;;
    # enable the memvoter manually
    c0 mov $r0.2 =  0xF000000
    ;;
    c0 stw CR_FTT_ADDR[$r0.0] = $r0.2
    ;;
    # wait some cycles to for the voters are actually on
    ....
    # read from the memory in fault tolerant mode
    c0 ldw $r0.12 = -4[$r0.1]
    c0 nop
    c0 ldw $r0.12 = -4[$r0.1]
    c0 nop
    c0 ldw $r0.12 = -4[$r0.1]
    ;;
    # read from the memory in fault tolerant mode
    c0 ldw $r0.13 = -8[$r0.1]
    c0 nop
    c0 ldw $r0.13 = -4[$r0.1] # wrong
    c0 nop
    c0 ldw $r0.13 = -8[$r0.1]
    ;;
```

As is shown in the code first, we write some value to the register. We need to verify that the values are correct and comparing only zero's is impractical. For this test the general purpose binary functions is disabled using the $DYNAMIC = false$ argument. An observation was made that the correct data was read by all lanes and the induced error was filtered and stored in the faultcounter.
Outcome: Successful

5.1.5 Register Voter

In accordance with the Memory Voter Write we can test the register in the same way. we want to see if the write is voted and copied to the next stage for the forward structure.
Test
A different assembler code is created to test the voter. Part of the code is shown in Listing 5.5. As before, the assembly first write to a control register. This control register is fed to the entity to manually configure the entity. This means enable only the register voter and the faultcounter.

Listing 5.5: Part of the assembler for testing the register voter

```
# write the value in the register
c0 mov $r0.2 =  0xF0000
;;
# store the value to the control register
c0 stw CR_FTT_ADDR[$r0.0] = $r0.2
;;
.....
;;
# perform some operations to the register
c0 add $r0.8 = $r0.2, $r0.3 # =3
c0 add $r0.9 = $r0.2, $r0.4 # =4
c0 add $r0.8 = $r0.2, $r0.3 # =3
c0 add $r0.9 = $r0.3, $r0.4 # =5 this is wrong
c0 add $r0.8 = $r0.3, $r0.3 # =4 this is wrong
c0 add $r0.9 = $r0.2, $r0.4 # =4
;;
# disable the voters manually three times
c0 stw CR_FTT_ADDR[$r0.0] = $r0.0
c0 nop
c0 stw CR_FTT_ADDR[$r0.0] = $r0.0
c0 nop
c0 stw CR_FTT_ADDR[$r0.0] = $r0.0
c0 nop
;;
```

As is listed in Listing 5.5 after the value is written to the control register some values are written to the same general purpose register. This is of course only possible when

the voter is active. In the fault counter a fault is shown and the wrong error is corrected by the voter. This value is forwarded in the pipeline.
Outcome: Successful

5.1.6 Fault-Counter

Testing the faultcounter is mainly covered in the previous section. When manually feeding the faults to the Memory Voter and to the Register Voter the counter registers the right amount of errors on the correct lane pairs.
Outcome: Successful

5.1.7 Lane switching

The lane switching is triggered by the fault counter. If a certain level is reached the faultcounter will signal the Finite State Machine (FSM) that the lane is wrong. The FSM handles the switch to the next lane. To test this we need to enable one of the voters and feed them with wrong information.
Test
The same code used in Listing 5.5 and 5.3 is used. We induce errors and the FSM is observed using the statusvector mentioned in Section 4.7.
Outcome: Successful

Table 5.1: Overview of executed tests

Test	Outcome	Remarks
Configuration control	Successful	
Multiplexer	Successful	With faultcounter
Pcvoter	Successful	With faultcounter
Memory write	Successful	With faultcounter
Memory read	Successful	Disable generic binary With faultcounter
Regvoter	Successful	With faultcounter
Lane Switch	Successful	

5.2 Testing the whole system

After the individual components are tested and verified, the system is put to the test in normal operation environment. The whole system is compared with the original core as reference.

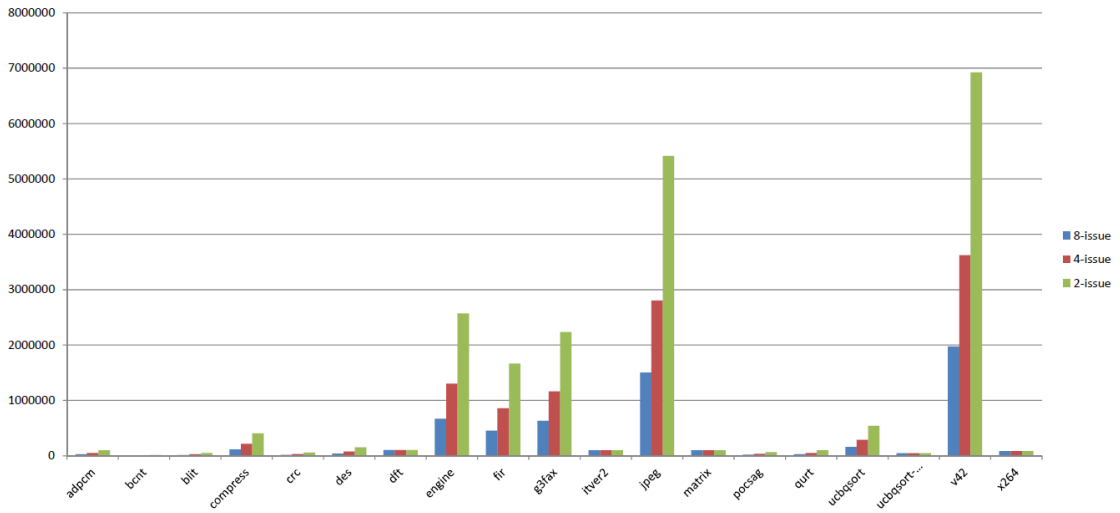


Figure 5.2: Benchmark of original ρ-VEX processor.

5.2.1 Original Core

The original core has been rewritten at the end of 2014. This is the starting-point of the thesis. Small adjustment are made in 2015 and the latest version is taken for the final testing. For the benchmarking the stand-alone version without cache is tested. The reason for taking this version is that all the changes have no effect on the cache or other components. And synthesizing this version takes less time. The Powerstone benchmark suite [55] is used to test the core. The results are shown in Figure 5.2 and show the score in a bar graph with each test executed in the three possible issue-widths, namely the 8-issue width, the 4-issue width and the 2 issue-width. The complete results are shown in Appendix B. As the image shows, programs with high Instruction Level Parallelism (ILP) benefits from an 8-issue width with a roughly four times speedup compared to a 2-issue width. For example *jpeg*, *v42*, *engine*. On the other hand some tests hardly benefit from the higher issue-width and do not improve at all. For example *matrix* or *dft*. This is expected behavior according to Section 2.1.

5.2.2 Compare FT with original core

We compare the fault tolerance core with the original core in two setups. First, a test with the fault tolerant mode off to fulfill one of the goals mentioned in Chapter 1. "To implement a low-cost, modular, and easy to use method in a working and running environment". When the fault tolerance is not used or disabled this should not influence the operation of the existing core.
The second setup is to compare the execution with an 2-issue width. The reason for choosing the 2-issue width is simple. This architecture tells us the most about the performance. When we use an 8-issue with and we enable fault tolerance we essentially reconfigure back to an 2-issue width. An increase in the amount of active cycles is

Table 5.2: Active cycles from benchmarks as shown in Figure 5.2

Benchmark	adpcm	bcnt	blit	compress	crc	des	dft
2-issue	26510	3863	13656	116865	15670	39256	101158
4-issue	50410	6610	25706	212925	29504	76854	101181
8-issue	98210	12104	49806	405084	57172	152050	101197

Benchmark	engine	fir	g3fax	itver2	jpeg	matrix	pocsag
2-issue	670090	455080	630548	97699	1504057	97697	18693
4-issue	1303860	858227	1165061	97722	2807143	97720	34886
8-issue	2571400	1664521	2234087	100351	5413315	97772	67292

Benchmark	qurt	ucbqsort	ucbqsort -fast	v42	x264
2-issue	27051	158925	45609	1975738	84675
4-issue	51005	285999	45634	3625023	84701
8-issue	98913	540147	45691	6923593	84757

expected. The only difference in performance we see is the bss clearing[2] before the execution of *main*() that is performed before reconfiguration when comparing 8-issue of 2-issue width in fault tolerance. But when we compare to an 2-issue width the amount of cycles should be comparable.
After verification that the individual components operate as expected the FT-version is compared with the original core. The test consists of two parts:

1. Compare the core with fault tolerance disabled.

2. Compare the core with fault tolerance enabled in 2-issue width.

Fault Tolerance Disabled
The result of the tests are positive. The benchmarks show no difference in the amount of active cycles when we use a fault tolerant capable core with no fault tolerance enabled.
Fault Tolerance enabled in 2-issue width compared to a 2-issue with
As shown in Table 5.3 the amount of active cycles is comparable with the amount of cycles from the original core.

The difference we would expect is exactly 5 cycles that is needed for the reconfiguration to enable the fault tolerance. As is shown in Table 5.3 this is not perfectly true. The amount is sometimes 4 and sometimes 6. This difference is due to uart communication. The lower baud rate can cause a small misalignment which causes the one cycle offset.

5.3 Fault Injection

To test if the core operates as expected we use the method described in Section 2.4.2. This method is a known way to test VHDL softcore implementations. We use the implemented saboteur discussed in Section 4.8. The signal inverting by the saboteur

[2]bss is used by many compilers and linkers for a part of the data segment containing statically-allocated variables represented solely by zero-valued bits initially [56]

Table 5.3: Amount of active cycles from a 2-issue width ρ-VEX processor with and without FT enabled

	adpcm	**bcnt**	**blit**	**compress**	**crc**	**des**
2-issue	98210	12104	49806	405084	57172	152050
2-issue tmr	98214	12109	49811	405136	57177	152056

	dft	**engine**	**fir**	**g3fax**	**itver2**	**jpeg**
2-issue	101197	2571400	1664521	2234087	100351	5413315
2-issue tmr	101198	2571405	1664527	2234092	100331	5413321

	matrix	**pocsag**	**qurt**	**ucbqsort**	**v42**	**x264**
2-issue	97772	67292	98614	540147	6923593	84757
2-issue tmr	97777	67296	98920	540152	6923598	84762

must be done sensible. Changing the value read in the register proves nothing, because the system assumes this is correct and it will not be detected. The other side is when one of the three signals that is written to the register is changed. This will again show no interesting information because the design is based on this and all the error are detected. We want to test the weaknesses of the system.
In Figure 5.4 a test is performed that the data read section in the pipeline is changed. This should cause a read of wrong data if a read action took place at that moment. For this test the benchmark program v42 is used, because this is the longest one and the ρ-VEX processor is put in 2-issue width. All the tests were preformed correct and the errors are read from the counters in the custom designed registers. For this purpose the lane switch was disabled.

As is shown the detected errors show a straight line, but only a part of the errors are detected. This is only five percent of the injected faults which are shown in the green line. This is what we expect because not every inversions by the saboteur coincide with a data read action. Further investigation shows that the amount of detected errors in accordance with what we see in Figure 5.3. The amount is data read actions is very low in the benchmarks. This is a good representation of the reality when a single-event upset (SEU) is not coincide with an active signal in the core.

In Figure 5.4 another inject is performed in the decode of the instruction. The amount of detected errors is much higher than in the previous example. The reason some errors are not detected can be found in the decode. This is found when further inspecting the injection in ModelSim. When for example a NO Operation (NOP) is decoded not all the values of the instruction are inspected. Only a few are necessary to decide if it is a NOP and the inverted bit is not inspected in the decode.

5.3.1 Statistic Analysis using VERI-Place

This experiment is postponed, because the program to test the system has not arrived in time.

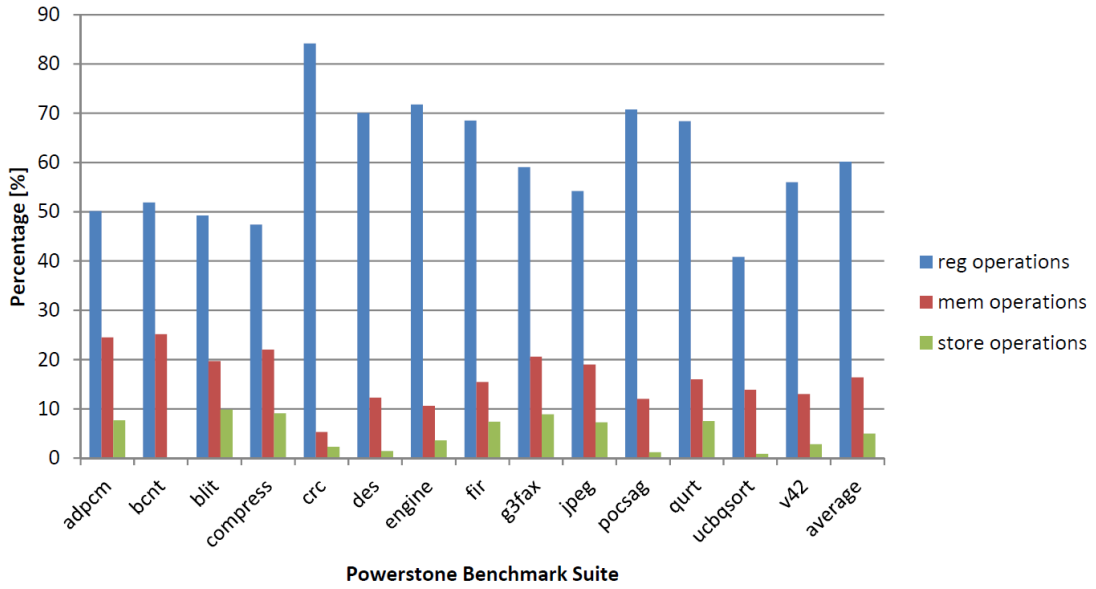


Figure 5.3: Memory vs register operations in Powerstone for ρ-VEX processor

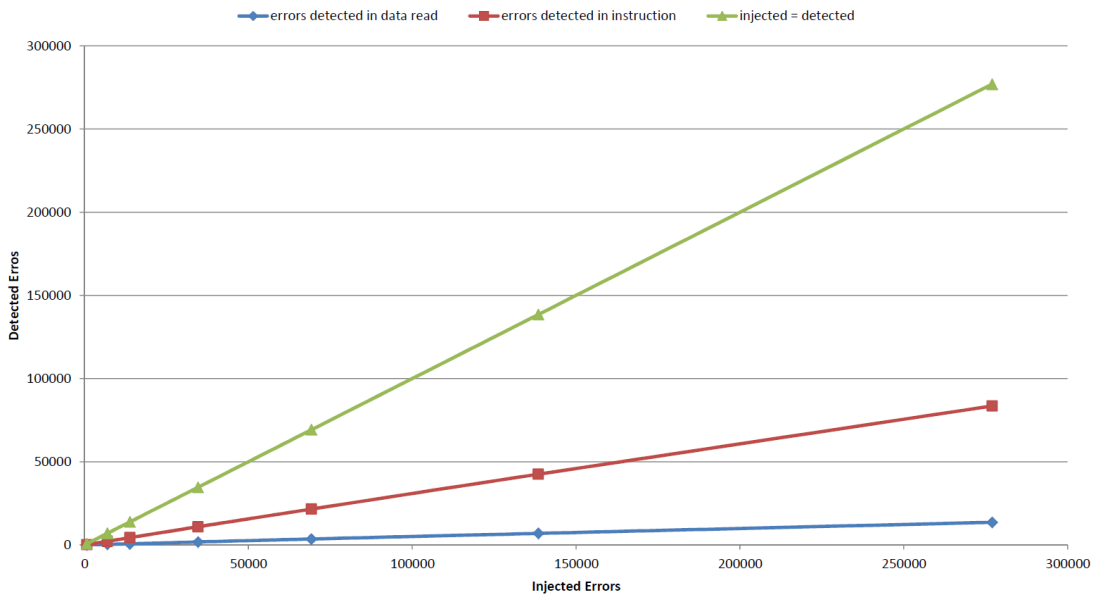


Figure 5.4: Inserting errors in the read data in the pipeline.

5.4 Added Hardware

After verifying that the hardware operates an analysis of the added hardware is made. We start with the clock speed and continue with the added hardware per unit.

Clock Speed
After synthesizing the design with an extended synthesis report we can conclude that the design does not interfere with the longest path. This means the clock speed can remain the same.

Resources Added
After generating an extended synthesis report the mapping of the different components is analysed from the FPGA the result is shown in Table 5.4. This is generated from the FPGA implementation of the standalone core without cache, without stop-bit implementation. Thus this is the smallest possible ρ-VEX processor with eight issues. All the other implementations are bigger with more peripherals. As Table 5.4 shows the added hardware is roughly 6%. This means that with the addition of 6%, in the worst case, a complete new functionality is added to the design which can be extended with little effort or new hardware. Therefore, it is safe to conclude that this added feature is worth the resources.

Table 5.4: Added hardware per component

Module	Slices	Slice Reg	Lut	Lutram	bram/fifo	dsp48E1	bufg
Total ML605	20419	15168	51849	1163	347	16	2
FSM	281	11	634	0	0	0	0
plex	62	0	195	0	0	0	0
pcvtr	243	2	450	0	0	0	0
memvtr	218	11	634	0	0	0	0
regvtr	498	16	977	0	0	0	0
Total FT	1302	40	3090	0	0	0	0
Percentage	6,38	0,26	5,96	0	0	0	0

5.5 Discussion

When evaluating the core we see that even with an extreme amount of errors the core still functions properly. One could argue that when injecting more than 100.000 errors per second in the core the problem lies not in the fault tolerant mechanism anymore. But outside the core. With this implementation at the cost of little added hardware a complete new functionality is added. With this functionality the reconfigurability of the ρ-VEX processor is extended to add new functionality with existing hardware. With the new mechanism of counting the errors one could even say it can not even prevent against single-event upsets (SEUs), but it might be an idea to use it against aging problems. The core uses the first lane always and it might be possible that this will suffer from aging problem sooner than other lanes. With the introduction of this capability one could counter this and use other lanes.

5.6 Conclusions

This chapter verifies step by step if the design works as intended. First, each component is tested while operating in the ρ-VEX processor when all other components of the fault-tolerant design, when possible, are disabled. All components are individually tested using custom made assembler code. With extra control mechanisms, individual units can be enabled through the software. Furthermore, extra status can be read in the software to ensure the correct outcome. When all individual components work as intended the whole system is tested and compared with the original core without fault tolerant capabilities. This was validated on the FPGA and gave a five cycle increase compared to a 2-issue width execution. These five are exactly the five extra cycles for reconfiguring. The third step is testing how the system performs when faults are injected. This showed that injection of errors are not always detected. This depends on the program and if the saboteur hits an active or used signal. Finally, the added hardware is compared to the existing hardware and concluded that the added hardware is small compared to the added functionality.

6 Conclusion

6.1 Summary

Chapter 2 presents the ρ-VEX platform as a version of a Very Long Instruction Word (VLIW) processor. The ρ-VEX processor is the only run-time reconfigurable VLIW softcore with a complete toolchain. It makes use of the benefits of reconfigurable computing and fills the gap between the general purpose core and the Application Specific Integrated Circuit (ASIC). With the use of Field-Programmable Gate Arrays (FPGAs) the softcore can dynamic change the issue-width if the running program and execute an other program on the free issues. Subsequently, a classification is made from all the causes of fault in electronic equipment. This can be summarized in three main groups: Design- and manufacturing faults which are caused during the design an manufacturing period and before the real use. The second group are faults from degradations that is caused by the wear and tear of the transistors. Finally, the most important group is the faults caused by environmental issues. The most important is the error caused by charged particles or called the *soft error*. Theses faults and errors are countered with fault tolerant techniques in software and in hardware. This results in two most common approaches. Either, we execute the operation multiple times and compare the outcome, or we strengthen the current operation by shielding, use Error Correction Code (ECC) or with older robuster technology. Finally, Chapter 2 presents methods of how to inject faults in a design to verify if the system operates as intended. The most interesting method is VHDL-based fault injection, because this is best applicable in VHDL-based softcore. Also a new state-of-the-art method of testing by modifying the bit-stream in order to simulate single-event upsets (SEUs) called VERI-Place is discussed.

Chapter 3 focuses in the vulnerable structures that are most likely to be hit by a single-event upset (SEU). These are often memory elements. Known countermeasures are found in similar designs which implement ECC or Error Detection and Correction (EDAC) in the memory elements, therefore, we assume the Instruction Memory (IMEM), Data Memory (DMEM) and the General Purpose Register (GPREG) correct, because with a trivial existing solution these memory elements can be protected. The emphasis is on triplicating the pipeline and enable Triple Modular Redundancy (TMR). This method requires an element to copy the instruction three times and a majority voter on all the incoming and outgoing data from and to the DMEM and GPREG. Finally, the mechanism to count the errors is discussed to decide if the spare lane is used and a lane switch is initiated. These operations require a dedicated control mechanism to detect the errors and a Finite State Machine (FSM) to handle the lane switch.

Chapter 4 describes how the elements are implemented in the design. First, a global overview is created and then the complete mechanism is chronological explained. Firstly, the configuration control enables the ρ-VEX processor to change in an other setup, then the instructions are triplicated to three planes. The PC voter is responsible that the correct Program Counter (PC) is broad-casted to the other pipelines. The memory voter and register voter make sure the write and read to the DMEM and GPREG is voted and copied. The Hierarchical Finite State Machine (HFSM) is explained and the custom made saboteur is introduced. Finally. the chapter is concluded with some challenges during implantation.

Chapter 5 verifies step by step if the design works as intended. First, each component is tested while operating in the ρ-VEX processor when all other components of the fault-tolerant design, when possible, are disabled. All components are individually tested using custom made assembler code. With extra control mechanisms, individual units can be enabled through the software. Furthermore, extra status can be read in the software to ensure the correct outcome. When all individual components work as intended the whole system is tested and compared with the original core without fault tolerant capabilities. This was validated on the Field-Programmable Gate Array (FPGA) and gave a five cycle increase compared to a 2-issue width execution. These five are exactly the five extra cycles for reconfiguring. The third step is testing how the system performs when faults are injected. This showed that injection of errors are not always detected. This depends on the program and if the saboteur hits an active or used signal. Finally, the added hardware is compared to the existing hardware and concluded that the added hardware is small compared to the added functionality.

6.2 Main contributions

In this thesis we present the implementation of a dynamic implementation of a fault tolerant execution of a critical section on the ρ-VEX processor. This section presents the main contributions of this project. The ρ-VEX processor will be tested in space in the near future. This fault tolerant execution will contribute heavily to the success of that test. In Figure 5.1 the overview of the added hardware in the core is shown. All the goals defined in Section 1.1 were achieved in the following ways;

1. The dynamic execution is achieved by using a special reconfiguration in run-time. Writing to the reconfiguration register triggers a special TMR execution of the code section.

2. The system has no major changes to the core and adds no extra cycles when inactive. During experiments no added cycles were measured when not enabling the fault tolerant method.

3. All operations are executed correct in fault tolerant mode. During experiments all operations execute correct from every possible configuration

The main contributions of this thesis are:

1. Created a Configuration in the ρ-VEX processor that executes the same program in lock-step mode.
2. Created a small bit-wise voter that signals which incoming signal was wrong
3. Created a HFSM that is prepared for expansion with other fault tolerant modes.
4. Created a dynamic method to create a section that can be executed in TMR during run-time.
5. Created a connection to the control registers to control the system from the software.

6.3 Future Work

The ρ-VEX processor is a project built by students and can always be improved. There is still a lot of work to be done to enable a full fault tolerant core. This thesis only takes care of a small part. This section will list a number of these tasks related to the complete en fully functional fault tolerant core

1. **Implement ECC in the IMEM, DMEM and GPREG.**
   As discussed in Section 3.1, the protection of the memory structures could be implemented relative simple. But to make a complete fault tolerant system this must be executed.
2. **Implement Duplicate execution in a two time 4-issue width setup.**
   The TMR setup losses quite some performance, but can continue operation when errors are detected. With a two times 4-issue width setup, the operation needs to roll back in case of errors, but it has bigger performance. At the cost of fault tolerance more speed is gained.
3. **Integrate the control logic more in the design.**
   The integration with the control registers can be improved. The registers should be writable from the debug bus so more tests and setups can be integrated in the whole system
4. **Adaptation of Branch unit to enable better voting.**
   During integration the branch unit prove to be a bit of a problem. With some modifications to the architecture a better setup could be created. At this point all lanes need to be active. Th best solution would be deactivating the spare lane.
5. **Use spare lane for other context**
   At this moment the fourth lane is used as spare and executes NO Operations (NOPs). This resource could be used to executed an extra program.
6. **Integrate the system with an interrupt to ensure correct operation of OS calls.**
   When operation in a real system an option could be to integrate interrupts from the

operation system. These interrupts should very important that a correct execution is needed and fault tolerance is activated. Other operations could be executed normal mode.

Bibliography

[1] Sebastian Anthony. (2014) Comet lander Philae has entered sleep mode, and may never wake up again (updated). [Online]. Available: http://tinyurl.com/p9hzkp7/

[2] A. Y. Waza, R. N. Mir *et al.*, "Reconfigurable Architectures," *Journal of Advanced Computer Science & Technology*, vol. 1, no. 4, pp. 337–346, 2012.

[3] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys (csuR)*, vol. 34, no. 2, pp. 171–210, 2002.

[4] F. Anjam, *Run-time Adaptable VLIW Processors: Resources, Performance, Power Consumption, and Reliability Trade-offs*. TU Delft, Delft University of Technology, 2013.

[5] J. Gracia, J. C. Baraza, D. Gil, and P. J. Gil, "Comparison and application of different VHDL-based fault injection techniques," in *Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on*. IEEE, 2001, pp. 233–241.

[6] L. Sterpone. (2014) VERI-Place Algorithm. [Online]. Available: http://www.cad.polito.it/~sterpone/tools.html

[7] Memebridge. (1999) Moore's Law. [Online]. Available: http://www.mooreslaw.org

[8] D. Borodin, B. B. Juurlink, S. Hamdioui, and S. Vassiliadis, "Instruction-level fault tolerance configurability," *Journal of Signal Processing Systems*, vol. 57, no. 1, pp. 89–105, 2009.

[9] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 389–398.

[10] The CPUshack museum. (2014) Welcome Back Rosetta: The Dynex MAS31750 Awakens. [Online]. Available: http://tinyurl.com/nvu2ty5/

[11] ——. (2014) Here comes Philae! Powered by an RTX2010. [Online]. Available: http://www.cpushack.com/2014/11/12/here-comes-philae-powered-by-an-rtx2010/

[12] S. Wong, F. Anjam, and F. Nadeem, "Dynamically reconfigurable register file for a softcore VLIW processor," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 969–972.

[13] J. J. Hoozemans, "Porting Linux to the rVEX reconfigurable VLIW softcore," 2014.

[14] F. Anjam, M. Nadeem, and S. Wong, "A VLIW softcore processor with dynamically adjustable issue-slots," in *Field-Programmable Technology (FPT), 2010 International Conference on*. IEEE, 2010, pp. 393–398.

[15] IBM. (2015) Power enterprise servers. [Online]. Available: http://www-03.ibm.com/systems/power/hardware/enterprise.html

[16] C. Iseli and E. Sanchez, "Spyder: A reconfigurable VLIW processor using FPGAs," in *FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on*. IEEE, 1993, pp. 17–24.

[17] I. Xilinx, "Microblaze processor reference guide," *reference manual*, vol. 23, 2006.

[18] N. I. Altera, "Processor Reference Handbook," *San Jose*, 2009.

[19] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri, "A VLIW processor with reconfigurable instruction set for embedded applications," *Solid-State Circuits, IEEE Journal of*, vol. 38, no. 11, pp. 1876–1886, 2003.

[20] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An FPGA-based VLIW processor with custom hardware execution," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. ACM, 2005, pp. 107–117.

[21] S. Wong, T. Van As, and G. Brown, "ρ-VEX: A reconfigurable and extensible softcore VLIW processor," in *ICECE Technology, 2008. FPT 2008. International Conference on*. IEEE, 2008, pp. 369–372.

[22] A. Brandon and S. Wong, "Support for dynamic issue width in VLIW processors using generic binaries," in *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, pp. 827–832.

[23] A. B. Gebregiorgis, "Aging Mitigation Schemes for Embedded Memories," Ph.D. dissertation, TU Delft, Delft University of Technology, 2014.

[24] M. Bushnell and V. D. Agrawal, *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*. Springer Science & Business Media, 2000, vol. 17.

[25] E. H. Ibe, *Terrestrial Radiation Effects in ULSI Devices and Electronic Systems*. John Wiley & Sons, 2015.

[26] S. Bhunia, S. Mukhopadhyay, and K. Roy, "Process variations and process-tolerant design," in *VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on*. IEEE, 2007, pp. 699–704.

[27] R. D. Younger, K. A. McIntosh, J. W. Chludzinski, D. C. Oakley, L. J. Mahoney, J. E. Funk, J. P. Donnelly, and S. Verghese, "Crosstalk analysis of integrated Geiger-mode avalanche photodiode focal plane arrays," in *Proc. SPIE*, vol. 7320, 2009, pp. 73 200Q–1.

[28] S. Borkar, "Design challenges of technology scaling," *Micro, IEEE*, vol. 19, no. 4, pp. 23–29, 1999.

[29] M. Hargrove, S. Voldman, R. Gauthier, J. Brown, K. Duncan, and W. Craig, "Latchup in CMOS technology," in *1998 IEEE International Reliability Physics Symposium Proceedings. 36th Annual (Cat. No. 98CH36173)*, 1998.

[30] J. F. Ziegler, "Terrestrial cosmic ray intensities," *IBM Journal of Research and Development*, vol. 42, no. 1, pp. 117–140, 1998.

[31] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on.* IEEE, 2005, pp. 243–247.

[32] E. Normand, "Single-event effects in avionics," *Nuclear Science, IEEE Transactions on*, vol. 43, no. 2, pp. 461–474, 1996.

[33] NASA. (2011) Result from on-orbit testing of the FRAM memory test Experiment on the FASTSAT micro-satellite. [Online]. Available: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20110015720.pdf

[34] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 63–75, 2002.

[35] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization.* IEEE Computer Society, 2005, pp. 243–254.

[36] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante, "Soft-error detection through software fault-tolerance techniques," in *Defect and Fault Tolerance in VLSI Systems, 1999. DFT'99. International Symposium on.* IEEE, 1999, pp. 210–218.

[37] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, "Software-implemented EDAC protection against SEUs," *Reliability, IEEE Transactions on*, vol. 49, no. 3, pp. 273–284, 2000.

[38] S. E. Anderson. (1997-2005) Computing parity. [Online]. Available: http://graphics.stanford.edu/~seander/bithacks.html

[39] AEROFLEX GAISLER. (2013) Leon3-ft sparc v8 processor leon3ft-rtax gaisler data sheet and users manual.

[40] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata studies*, vol. 34, pp. 43–98, 1956.

[41] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[42] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[43] J. Arlat, "Validation de la sûreté de fonctionnement par injection de fautes, méthode- mise en oeuvre- application," Ph.D. dissertation, 1990.

[44] T. A. Delong, B. W. Johnson, and J. A. P. Iii, "A fault injection technique for VHDL behavioral-level models," *IEEE Design & Test of Computers*, no. 4, pp. 24–33, 1996.

[45] J. R. Armstrong, F.-S. Lam, and P. C. Ward, "Test generation and fault simulation for behavioral models," *Performance and Fault Modeling with VHDL*, pp. 240–303, 1992.

[46] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: the MEFISTO tool," in *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on.* IEEE, 1994, pp. 66–75.

[47] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil, "Improvement of fault injection techniques based on VHDL code modification," in *High-Level Design Validation and Test Workshop, 2005. Tenth IEEE International.* IEEE, 2005, pp. 19–26.

[48] J.-C. Baraza, J. Gracia, S. Blanc, D. Gil, and P.-J. Gil, "Enhancement of fault injection techniques based on the modification of VHDL code," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 6, pp. 693–706, 2008.

[49] L. Sterpone and M. Violante, "A new analytical approach to estimate the effects of SEUs in TMR architectures implemented through SRAM-based FPGAs," *Nuclear Science, IEEE Transactions on*, vol. 52, no. 6, pp. 2217–2223, 2005.

[50] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson, "On latching probability of particle induced transients in combinational networks," in *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on.* IEEE, 1994, pp. 340–349.

[51] J. S. Kim, C. Nicopoulos, N. Vijaykrishnan, Y. Xie, and E. Lattanzi, "A probabilistic model for soft-error rate estimation in combinational logic," in *Proc. of the Intl Workshop on Probabilistic Analysis Techniques for Real-time and Embedded Systems*, 2004.

[52] F. Anjam and S. Wong, "Configurable fault-tolerance for a configurable VLIW processor," in *Reconfigurable Computing: Architectures, Tools and Applications.* Springer, 2013, pp. 167–178.

[53] V. Sklyarov, "Hierarchical finite-state machines and their use for digital control," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 7, no. 2, pp. 222–228, 1999.

[54] Margaret Rouse. (2011) Uart (universal asynchronous receiver/-transmitter). [Online]. Available: http://whatis.techtarget.com/definition/UART-Universal-Asynchronous-Receiver-Transmitter

[55] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the Low-Power M CORE TM Architecture," in *Power Driven Microarchitecture Workshop*. Citeseer, 1998, pp. 145–150.

[56] C. M. Kormanyos, *Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming*. Springer Science & Business Media, 2013.

List of definitions

ILP Instruction level parallelism refers to the existence of independent operations in a program which can be executed together in a single clock cycle. Finding some independent operations in a program or a stream of operations is the job of a compiler in case of a VLIW processor or run-time control hardware in case of a superscalar processor. Instruction Level Parallelism (ILP) can be combined with any other type of parallelism to further enhance the performance.

Fault Injection Fault injection is the validation technique of the Dependability of Fault Tolerant Systems (FTSs) which consists in the accomplishment of controlled experiments where the observation of the systems behavior in presence of faults is induced explicitly by the written introduction (injection) of faults in the system

Saboteur

A

Listing A.1: Part of the assembler for testing the register voter

```
-- Creation of a saboteur

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use ieee.std_logic_unsigned.all;
use rvex.common_pkg.all;

entity ft_saboteur is
    port(
        clk                         : in  std_logic;
        clk_enable                  : in  std_logic;
        reset                       : in  std_logic;
        start                       : in  std_logic;
        scrp4_limit                 : in  rvex_address_type;
        saboteur                    : out std_logic
    );
end ft_saboteur;

architecture structural of ft_saboteur is
--signals for the saboteur
signal count        : unsigned (31 downto 0):=(others=>'0');
signal new_count    : unsigned (31 downto 0):=(others=>'0');
signal cntEn        : std_logic:='0';
signal saboteur     : std_logic:='0';
signal clkreset     : std_logic:='0';
signal limit        : integer:=0;
signal set_limit    : integer:=0;

-- Simple counter
counterregister:process (clk, reset, clkreset)
    begin
        if (rising_edge(clk)) then
            if (reset = '1' or clkreset = '1') then
                count <= (others => '0'); --reset to zero
```

```
            else
                count <= new_count;
            end if;
        end if;
    end process;

    -- Simple counter
        process (count, cntEn)
    begin
        if (cntEn = '1') then
                        new_count       <= count + 1;
        else
                        new_count       <= count;
        end if;
        end process;

    limit       <= to_integer(count);
    set_limit   <= to_integer(unsigned(scrp4_limit));

    process(start, limit, set_limit)
    begin
        if (start = '1' and limit = set_limit) then
            cntEn           <= '1';
            saboteur        <= '1';
            clkreset        <= '1';
        elsif (start = '1') then
            saboteur        <= '0';
            cntEn           <= '1';
            clkreset        <= '0';
        else
            saboteur        <= '0';
            cntEn           <= '0';
            clkreset        <= '0';
        end if;
    end process;
end structural;
```

Benchmark of the rewritten ρ-VEX processor

B

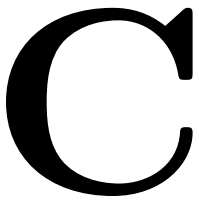
Table B.1: Complete benchmark numbers part 1

	adpcm	**bcnt**	**blit**	**compress**	**crc**	**des**	**dft**	**engine**	**fir**	**g3fax**
8-issue										
active cycles	26510	3863	13656	116865	15670	39256	101158	670090	455080	630548
stall	0	0	0	0	0	0	0	0	0	0
commited bundle count	23900	2747	12050	99133	13834	37598	93415	633770	403147	534513
commitedsyllable count	191200	21976	96400	793064	110672	300784	186920	5070160	3225176	4276104
commited NOP count	144217	16158	74387	578233	81006	168617	145331	3956929	2320466	3181607
4-issue										
active cycles	50410	6610	25706	212925	29504	76854	101181	1303860	858227	1165061
stall	0	0	0	0	0	0	0	0	0	0
commited bundle count	47800	5494	24100	195264	27668	75196	93437	1267540	806294	1069026
commitedsyllable count	191200	21976	96400	781056	110672	300784	186920	5070160	3225176	4276104
commited NOP count	144217	16158	74387	567726	81006	168617	145331	3956929	2320466	3181607
2-issue										
active cycles	98210	12104	49806	405084	57172	152050	101197	2571400	1664521	2234087
stall	0	0	0	0	0	0	0	0	0	0
commited bundle count	95600	10988	482000	387460	55336	150392	93460	2535080	1612588	2138052
commitedsyllable count	191200	21976	96400	774920	110672	300784	186920	5070160	3225176	4276104
commited NOP count	144217	16158	74387	562357	81006	168617	145331	3956929	2320466	3181607

Table B.2: Complete benchmark numbers part 2

	itver2	**jpeg**	**matrix**	**pocsag**	**qurt**	**ucbqsort**	**ucbqsort- fast**	**v42**	**x264**
8-issue									
active cycles	97699	1504057	97697	18693	27051	158925	45609	1975738	84675
stall	0	0	0	0	0	0	0	0	0
commited bundle count	93119	1303086	92805	16203	23954	127074	43081	1649285	76914
commitedsyllable count	744952	10424688	742440	129624	191632	1016592	344648	13194280	615312
commited NOP count	611886	7534912	642371	82453	138638	798735	299732	10038099	532058
4-issue									
active cycles	97722	2807143	97720	34886	51005	285999	45634	3625023	84701
stall	0	0	0	0	0	0	0	0	0
commited bundle count	95304	2606172	94982	32406	47908	254148	44212	3298570	78726
commitedsyllable count	381216	10424688	379928	129624	191632	1016592	176848	13194280	314904
commited NOP count	293617	7534912	325173	82453	138638	798735	153365	10038099	269201
2-issue									
active cycles	100351	5413315	97772	67292	98913	540147	45691	6923593	84757
stall	0	0	0	0	0	0	0	0	0
commited bundle count	99024	5212344	96160	64812	95816	508296	44792	6597140	79708
commitedsyllable count	198048	10424688	192320	129624	191632	1016592	89584	13194280	159416
commited NOP count	133345	7534912	161016	82453	138638	798735	77009	10038099	133149

Reflection



When starting this thesis the current design of the ρ-VEX processor was still a work in progress. This was a good thing, because it gave me time to have a good thought about how to solve the question.
I started with a basic design based on the old version of the ρ-VEX processor. During my development I spotted a special trend in the amount of hardware I added. I started of small with some simple functionality which gradually grew bigger and bigger. To elaborate I started to triplicate the instructions and continued to implement the voters in the register writes and the memory writes. After that I connected it to a basic FSM which grew to a fully functional FSM. I saw a trend here that with increasing time the amount of code of hardware increased exponentially.
Just before the summer holiday I was at a standstill. I used the original 2-issue width setup and secretly enabled the other cores to execute the instruction two times more. This created a loop of endless patching and ugly fives. After a tip from a fellow student I implemented the "Configuration Control". From this moment on I was a the peak of the amount of code and hardware. From that moment on I could clean the code and make it smaller and nicer.
Eventually I could reduce the complete setup to a lean and small addition. When looking back it seams so little for such a long period, but I guess the path getting there is part of the work that can't be captured on a piece of paper.

Klaas

Biography

Klaas Meun began his study at the Royal Military Academy in Breda. After a few years exercising his profession as an officer for the Royal Netherlands Army he was selected for a new position and was required to add a Master of Science degree to his resume. To achieve this, he started at TU Delft University of Technology in 2010. After his Bachelor's degree in Electrical Engineering, he started in 2013 with his Master in Computer Engineering.