A fault tolerant microarchitecture for safety-related automotive control

Additional Information:

- A Doctoral Thesis. Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University.

Metadata Record: [https://dspace.lboro.ac.uk/2134/14402](https://dspace.lboro.ac.uk/2134/14402)

Publisher: © Emmanuel Touloupis

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/
University Library

Author/Filing Title: Touloupsis, E.

Class Mark: T

Please note that fines are charged on ALL overdue items.

For Reference Only

0403271290
A Fault Tolerant Microarchitecture for Safety-Related Automotive Control

by

Emmanuel Touloupis, Dipl. El. Eng.

A Doctoral Thesis submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University

November 2005

© by Emmanuel Touloupis 2005
To my parents
ABSTRACT

THE successful use of fly-by-wire systems in aviation along with the positive experience of drive-by-wire systems with mechanical backup for braking and power steering have led to the development of complete drive-by-wire systems that reduce the cost of a vehicle, are lighter and provide better passive safety to the passenger. These systems have the form of a distributed, real-time embedded system. Similar architectures can be found in other safety-critical and mission-critical applications in avionics, as mentioned before, medical equipment, and the industrial sector.

The advances in embedded system technology has enabled designers to implement low-cost and small form factor electronics. However shrinking CMOS technologies are facing considerable reliability problems since they become more sensitive to transient faults.

This thesis investigates the application of traditional methods for the development of safety critical computer systems and their application on single-chip devices.

The contributions of this work are briefly summarised as follows:

- The development of a novel fault-tolerant architecture for protecting the processor core.
- Methods for performing fault-injection experiments on embedded processor architectures.
- Fault-models for multiple faults on digital systems with the use of statistical distributions.
- An extensive study of a processor's behaviour under the presence of faults within its pipelined execution unit.
I would like to thank my supervisors Dr. James A. Flint and Dr. Vassilios A. Chouliaras for the continuous advice and support throughout all the stages of this work. Their help has been invaluable.

I would like to express my gratitude to the management of the Motor Industry Research Association (MIRA Ltd.) for supporting financially the work presented in this thesis. In particular I would like to thank Dr. David D. Ward for his support and advice and for providing comments on the draft of this thesis. Thanks are also due to Prof. Geoff Callow and Dr. Anthony Baxendale for overseeing the project.

I acknowledge all my colleagues in Loughborough University for their support and companionship throughout these years.

All the wonderful people I met during my years in Loughborough and have become good friends have contributed even without knowing to this work and have made these years special. Special thanks go to the people I lived with: Ximo, Jose, Christian and Vaggelis.

Finally, I would like to express my deep gratitude and love to my family for always standing by me and always doing their best to provide me with everything necessary to follow my dreams.
# CONTENTS

List of abbreviations ................................................................. viii

List of figures ........................................................................ xi

List of tables ........................................................................... xiv

1. Introduction ........................................................................ 1
   1.1 The Use of Electronics in the Automotive Industry ............. 2
   1.2 Drive-by-Wire Systems .................................................... 4
   1.3 Reliability of Semiconductor Devices ................................. 10
   1.4 Contributions .................................................................. 11
   1.5 Thesis overview .............................................................. 12
   References ........................................................................... 14

2. Issues in the Design of Safety-Critical Computer Systems ......... 16
   2.1 Definitions and Concepts .................................................. 16
   2.2 Relevant Standards ......................................................... 35
   2.3 System Life-Cycle ........................................................... 36
   2.4 Fault Tolerance ............................................................... 40
   2.5 Conclusion ..................................................................... 50
   References ........................................................................... 51
3. The Origin and Mitigation of Faults in Microprocessors ........................................ 55
   3.1 Radiation Effects on Integrated Circuits ......................................................... 55
   3.2 EMC for Integrated Circuits ................................................................. 60
   3.3 Electromigration ............................................................................... 60
   3.4 Impact of New CMOS Technologies ....................................................... 61
   3.5 Proposed Solutions ............................................................. 63
   3.6 Conclusion .......... ................................. 69
   References ................................................................. 71

4. A Novel Pipeline-Protected Processor Core .................................................. 77
   4.1 A Method for the Protection of a RISC Pipeline ........................................... 77
   4.2 Choice of Processor .............................................................. 83
   4.3 Pipeline-Protected Processor Implementation with LEON2 ......................... 87
   4.4 Conclusion .............................................................. 103
   References ................................................................. 105

5. Processor Evaluation with Fault Injection Techniques ............................. 107
   5.1 Previous Work .............................................................. 107
   5.2 The Use of Foreign Language Interface (FLI) ............................................ 111
   5.3 Mixed Use of VHDL and FLI for Fault Injection ...................................... 115
   5.4 Hardware-Based Fault Injection ......................................................... 121
   5.5 Conclusion .............................................................. 124
   References ................................................................. 126

6. The Effect of Soft Errors on the Microprocessor Pipeline ....................... 130
   6.1 Previous Work .............................................................. 130
6.2 Fault and Error Propagation in the Pipeline .......................... 131
6.3 Modelling Single and Multiple Faults ................................. 132
6.4 Fault Injection Campaign Setup ........................................ 136
6.5 Results from Single Fault Injection ...................................... 142
6.6 Results from Multiple Fault Injection ................................. 150
6.7 Conclusion ................................................................. 154
References ................................................................. 158

7. Conclusions ........................................................................ 160
7.1 Contributions of the Thesis ................................................ 160
7.2 Suggestions for Further Research ....................................... 162

Appendix ................................................................................ 164

A. HAZOP Studies on Embedded Architectures .......................... 165

B. C Code of the Benchmark Programs ...................................... 171
   B.1 mtx4x4 ................................................................. 171
   B.2 bitcnt ................................................................. 172
   B.3 qsort ................................................................. 178

C. Author’s Publications ....................................................... 181
## LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Anti-lock Braking System</td>
</tr>
<tr>
<td>ACC</td>
<td>Adaptive Cruise Control</td>
</tr>
<tr>
<td>AHB</td>
<td>Advanced High-speed Bus</td>
</tr>
<tr>
<td>ALARP</td>
<td>As Low As Reasonably Practicable</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>AMBA</td>
<td>Advanced Microprocessor Bus Architecture</td>
</tr>
<tr>
<td>APB</td>
<td>Advanced Peripheral Bus</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>ASR</td>
<td>Acceleration Slip Regulation</td>
</tr>
<tr>
<td>BA</td>
<td>Brake Assist</td>
</tr>
<tr>
<td>BCH</td>
<td>Bose Chaudhuri Hochquenghem code</td>
</tr>
<tr>
<td>BIST</td>
<td>Built-In Self Test</td>
</tr>
<tr>
<td>CA</td>
<td>Collision Avoidance</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-The-Shelf</td>
</tr>
<tr>
<td>CP</td>
<td>Co-Processor</td>
</tr>
<tr>
<td>DE</td>
<td>Decode stage</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>DSU</td>
<td>Debug Support Unit</td>
</tr>
<tr>
<td>ECC</td>
<td>Error Correcting Code</td>
</tr>
<tr>
<td>EDAC</td>
<td>Error Detection And Correction</td>
</tr>
<tr>
<td>EHB</td>
<td>Electro-Hydraulic Braking</td>
</tr>
<tr>
<td>EMB</td>
<td>Electro-Mechanical Braking</td>
</tr>
<tr>
<td>EMC</td>
<td>Electromagnetic Compatibility</td>
</tr>
<tr>
<td>EMI</td>
<td>Electromagnetic Interference</td>
</tr>
<tr>
<td>EPS</td>
<td>Electro-Pneumatic Shifting</td>
</tr>
<tr>
<td>EPSi</td>
<td>Electrical Power Steering</td>
</tr>
<tr>
<td>ESP</td>
<td>Electronic Stability Program</td>
</tr>
<tr>
<td>ESS</td>
<td>Electronic Switching System</td>
</tr>
</tbody>
</table>
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX</td>
<td>Execution stage</td>
</tr>
<tr>
<td>FE</td>
<td>Fetch stage</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FIT</td>
<td>Failures In Time</td>
</tr>
<tr>
<td>FLI</td>
<td>Foreign Language Interface</td>
</tr>
<tr>
<td>FMEA</td>
<td>Fault Mode and Effects Analysis</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating Point Unit</td>
</tr>
<tr>
<td>FTA</td>
<td>Fault Tree Analysis</td>
</tr>
<tr>
<td>FTMP</td>
<td>Fault Tolerant Multi-Processor</td>
</tr>
<tr>
<td>HAZOP</td>
<td>Hazard and Operability study</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>IU</td>
<td>Integer Unit</td>
</tr>
<tr>
<td>LET</td>
<td>Linear Energy Transfer</td>
</tr>
<tr>
<td>LETth</td>
<td>Linear Energy Transfer threshold</td>
</tr>
<tr>
<td>LRR</td>
<td>Least Recently Replaced</td>
</tr>
<tr>
<td>LRU</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>ME</td>
<td>Memory Stage</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>MPC</td>
<td>Minimum Physical Constraints</td>
</tr>
<tr>
<td>NMR</td>
<td>N-Modular Redundancy</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Component Interface</td>
</tr>
<tr>
<td>PES</td>
<td>Programmable Electronic Systems</td>
</tr>
<tr>
<td>PLI</td>
<td>Programming Language Interface</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SBW</td>
<td>Steer-By-Wire</td>
</tr>
</tbody>
</table>
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEB</td>
<td>Single Event Burnout</td>
</tr>
<tr>
<td>SEFI</td>
<td>Single Event Functional Interrupt</td>
</tr>
<tr>
<td>SEL</td>
<td>Single Event Latchup</td>
</tr>
<tr>
<td>SER</td>
<td>Soft Error Rate</td>
</tr>
<tr>
<td>SEU</td>
<td>Single Event Upset</td>
</tr>
<tr>
<td>SIFT</td>
<td>Software Implemented Fault Tolerance</td>
</tr>
<tr>
<td>SIL</td>
<td>Safety Integrity Level</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
<tr>
<td>SOI</td>
<td>Silicon-On-Insulator</td>
</tr>
<tr>
<td>SPARC</td>
<td>Scalable Processor Architecture</td>
</tr>
<tr>
<td>SREGS</td>
<td>Special Registers</td>
</tr>
<tr>
<td>SWIFI</td>
<td>Software Implemented Fault Injection</td>
</tr>
<tr>
<td>TBW</td>
<td>Throttle-By-Wire</td>
</tr>
<tr>
<td>TCS</td>
<td>Traction Control System</td>
</tr>
<tr>
<td>TMR</td>
<td>Triple Modular Redundancy</td>
</tr>
<tr>
<td>TT-CAN</td>
<td>Time Triggered CAN</td>
</tr>
<tr>
<td>TTP</td>
<td>Time Triggered Protocol</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver Transmitter</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very high speed integrated circuit HDL</td>
</tr>
<tr>
<td>VHPI</td>
<td>VHDL Procedural Interface</td>
</tr>
<tr>
<td>VPI</td>
<td>Verilog Procedural Interface</td>
</tr>
<tr>
<td>WR</td>
<td>Write Stage</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1.1 Electronic systems introduced and their hazard severity [1.4] .......................... 3
1.2 Steer-by-wire system concept .............................................................................. 6
1.3 Centralised Steer-by-Wire structure, from [1.13] ................................................. 7
1.4 X-By-wire system architecture, from [1.5, 1.6] .................................................... 8
1.5 Brake-By-wire system architecture, from [1.4] .................................................... 9
1.6 SIRIUS system electrical architecture, from [1.16] ............................................ 10

2.1 Fault, errors, system failure .................................................................................. 18
2.2 The three-universe model for faults, errors and failures ..................................... 23
2.3 The propagation of faults, errors and failures within systems ............................ 23
2.4 Failure rate for electronic components .................................................................. 26
2.5 Reliability block diagram of a series system ....................................................... 30
2.6 Reliability block diagram of a parallel system .................................................... 31
2.7 Example of reliability block diagram series-parallel combination system ........ 31
2.8 Markov model of a system with no repair ............................................................ 32
2.9 Markov model of a system with repair ................................................................. 34
2.10 Simplified Markov model of a system with repair .............................................. 34
2.11 Example of a system life-cycle ............................................................................. 37
2.12 Levels of risk tolerability and the ALARP principle ........................................... 38
2.13 Risk reduction ...................................................................................................... 38
List of Figures

2.14 Recovery blocks program flow ........................................... 44
2.15 TMR system with (a) one voter and (b) with redundant voters ........ 46
2.16 Dynamic redundancy: (a) A standby spare arrangement (b) A self-checking pair 48
2.17 N-modular redundancy with spares ..................................... 49

3.1 Cosmic rays traversing the Earth's atmosphere (from [3.4]) ............. 56
3.2 Path of a charged particle through a CMOS transistor .................... 57
3.3 Example of a cross section vs LET curve for a microelectronic device ...... 59
3.4 Critical charge as a function of technology (from [3.8]) .................. 62
3.5 Bit and system error rates for: (a) DRAM and (b) SRAM (from [3.22]) .... 62
3.6 Different types of error mitigation techniques .................................. 64

4.1 Top-level diagram of the proposed redundant pipeline system .......... 80
4.2 Error-checking on each pipeline stage ..................................... 80
4.3 State-transfer mechanism ..................................................... 82
4.4 Top-level LEON2 diagram from [4.17] .................................... 86
4.5 Inputs and outputs of a comparator circuit used in the fault tolerant LEON architecture. Inputs are single bit for bit-comparators and multiple bits for word-comparators ........................................... 89
4.6 State transitions of the control module ..................................... 92
4.7 Timing diagram of an erroneous access to the cache memory .............. 95
4.8 Timing diagram of an erroneous access to the register file ................ 96
4.9 Markov model of the system ................................................... 98
4.10 System reliability: (a) single errors (b) single and double errors ...... 101
4.11 System reliability compared with IEC61508 requirements: (a) simple configuration (b) fault-tolerant configuration ......................... 102
List of Figures

4.12 Fault-tolerant configuration (a) floorplan and (b) VLSI microcell ............ 103

5.1 Fault injection platform based on the use of VHDL and FLI modules ......... 116
5.2 The mechanism for injecting non-destructive faults on pipeline registers.... 117
5.3 The three phases of the executed program ........................................... 118
5.4 Hardware based fault injection ............................................................. 123

6.1 Fault propagation in a pipeline .......................................................... 132
6.2 Pipeline state transition due to faults, after [6.12] ................................ 133
6.3 Normal distribution (a) pdf and (b) cdf .............................................. 134
6.4 By rejecting values outside some specified limits and replacing them with uniformly distributed random values, the (a) pdf and (b) cdf of the normal distribution change. In this example the limits are 0 and 10 ......... 136
6.5 Single pipeline architecture performance under single fault injection ....... 144
6.6 Pipeline stages behaviour in single pipeline architecture under single fault injection .......................................................... 146
6.7 Histogram of the delayed simulations in the single pipeline architecture (the percentage refer to the total amount of delayed simulations in each benchmark) .... 147
6.8 Redundant pipeline architecture performance under single fault injection ... 148
6.9 Pipeline stages behaviour in the redundant pipeline architecture under single fault injection .......................................................... 149
6.10 Probability of a fault propagating to the pipeline output ....................... 150
6.11 Single core results for double errors .................................................... 155
6.12 Redundant core results for double errors ............................................. 156

A.1 HAZOP is working in both direction to identify the causes and the consequences of hazards .......................................................... 165
LIST OF TABLES

2.1 Main fault categories ............................................. 21
2.2 IEC 61508 requirements ........................................... 39
3.1 IC technology characteristics (from [3.14]) ..................... 61
4.1 Output signals of the pipeline ................................... 91
4.2 Transition rates values ............................................ 100
4.3 Macrocell results .................................................. 102
5.1 AHB address allocation ............................................ 122
6.1 Levels of distribution of fault occurrence ....................... 137
6.2 Pipeline register grouping ......................................... 138
6.3 Pipeline register grouping (continued) ........................ 139
6.4 Execution times of the benchmark applications ................ 140
6.5 Comparison of results for the metx4x4 benchmark .............. 145
6.6 Number of delayed simulations in the single pipeline architecture ........................................... 145
6.7 Fault propagation characteristics (mtx4x4) .................... 150
6.8 Fault propagation characteristics (bitcnt) ..................... 150
6.9 Fault propagation characteristics (qsort) ....................... 151
6.10 Percentage of failure-free simulations with single and double fault injection for the redundant pipeline ........................................... 154
A.1 Guide words for VLSI embedded architectures ..................... 169
CHAPTER 1: INTRODUCTION

The most common perception of a computer system is a desktop PC that is used for a variety of complex applications. The range of applications covers a wide spectrum, from entertaining games to very specialised software for professionals. However, the majority of microprocessors are not used in traditional desktop computers. Instead they lie 'hidden' from the outside world as embedded systems inside various domestic appliances, aeroplanes and road vehicles, industrial control systems, portable gadgets, mobile phones, etc. The microprocessor architectures applied in these examples can be single-processor with the necessary peripherals, multi-processor when increased computational power is required, or even distributed networks of computer nodes, typically found in control applications.

One important factor that defines the whole design process in the applications mentioned above is the consequence of incorrect operation. An electronic flight control system that fails to operate according to its specification during a flight will put in danger the life of the passengers. If an adaptive cruise controller in a car fails to operate correctly, it may cause an accident that could potentially risk the life not only of the driver, but of other road users. Any system that has the capacity to do harm to humans or the environment is safety-related. The distinction of safety-related systems is not always obvious, but it can be claimed that all systems that control even modest amounts of power fall into that category. A case where the control logic of a microwave oven or a boiler fails in such way that it leaves the heater stuck on could result in a fire with lethal consequences. In many cases the term safety-critical system is applied to suggest a safety-related system of high criticality.

The motivation behind the work of this thesis is the introduction of safety-critical embedded systems in road vehicles and the need to satisfy the mass production constraints of the automotive industry. These systems require low-cost and highly reliable components. The main objective is to develop such components by exploiting advances in semiconductor technology.
that allow the implementation of large systems into one single chip. This thesis will present and analyse a novel, fault tolerant, multiprocessor system suitable for safety critical automotive applications.

1.1 The Use of Electronics in the Automotive Industry

The integration of electronics in automobiles began during the 1970's and became well established by the 1980's. The power of microprocessing was first utilised to control basic internal combustion engine parameters in order to increase efficiency and reduce pollution. With the rapid development of semiconductor technologies the use of microprocessors has expanded in many different areas from improved handling of passive and active safety to the integration of entertainment systems. Today's top-of-the-line vehicle uses over 80 microcontrollers, and an even greater number of power semiconductors and smart power ICs to perform functions in a variety of systems including powertrain control, antilock brakes, and air bag systems. The more recent developments include [1.4]:

- **Electronic Driver-Assisting Systems**: In these cases the existing mechanical systems are supported by electronics. Typical examples include antilock braking system (ABS), traction control system (TCS), electronic stability program (ESP) and brake assist (BA). In case the electronic control fails, the vehicle's braking system behaves like the conventional mechanical (or hydraulic) one. This is a fail safe design which means that at least a basic part of the system's functionality is provided in case the electronic system fails.

- **Drive-by-Wire Systems with Mechanical Backup**: Many drive-by-wire systems are already used in vehicles. Examples include electric power steering, electronic braking and throttle systems. While these systems are electronically controlled and operated, there is still a mechanical backup system if an electrical problem develops which makes the system fail safe.

- **Drive-by-Wire Systems without Mechanical Backup**: The successful use of fly-by-wire systems in aviation along with the positive experience of drive-by-wire systems with mechanical backup for braking and power steering have led to the development of complete drive-by-wire systems that reduce the cost of a vehicle, are lighter and are claimed

---

1 For a detailed review of the evolution of automotive electronics see [1.1, 1.2, 1.3]
to provide better passive safety to the passenger. There are however some acceptability issues with these systems from both consumers and legislative bodies. At the time of writing, steer-by-wire must have a mechanical linkage for legal reasons.

- **Higher-Level Automotive Control:** These are systems that may influence several other basic systems on the vehicle. As an example, adaptive cruise control (ACC) allows a vehicle to maintain a certain gap between itself and the vehicle in front, or to maintain a speed previously set by the driver by controlling both the engine and the brakes.

Figure 1.1 shows the hazard severity of different electronic and electrical driving systems that have been already introduced, or will be introduced in the near future. It is clear that the hazard severity of the drive-by-wire systems is considerably higher than systems found on vehicles currently. In order to facilitate the introduction of such systems, research is needed to provide high reliability for low cost, and it is in this context that the thesis presents its findings.

Figure 1.1: Electronic systems introduced and their hazard severity [1.4]
1.2 Drive-by-Wire Systems

There are many important issues to be taken into consideration when designing a drive-by-wire system, or an individual component thereof. These systems are still relatively new to the automotive sector and hence there is a great deal of research still to be done in this area. Many major vehicle manufacturers, suppliers and other institutes are involved in relevant projects in order to investigate the key points of the implementation.

The three basic by-wire systems in the automotive industry are throttle-by-wire, brake-by-wire and steer-by-wire. Throttle-by-wire systems that are already available in some conventional vehicles use a degree of redundancy for fault tolerance and have fail-safe operation. Brake-by-wire systems are the next to appear, since they are flexible enough to provide a degree of fault tolerance by the fact that the braking force can be applied on all four wheels and there is no single point of failure. The most technically challenging of the three seem to be the steer-by-wire systems, since they require correct operation in all of their major parts (as opposed to brake-by-wire that can maintain a correct operation even if for example one braking sensor in one wheel is not operating). It should be noted that there is a general trend towards electronically controlled systems in vehicles. The example where electronic systems are important are electronic powertrain controllers. These are essential components in modern vehicles as it would be extremely hard to produce an engine which would meet exhaust emission requirements without them. Electronic systems can also be used as driver assistance systems in order to improve safety, since it has been noted that the vast majority of vehicle accidents are caused by driver errors.

In the X-By-Wire project [1.5, 1.6] a Steer-By-Wire prototype system was built and tested. The report describes in detail all the stages of the design process and draws some important conclusions regarding the design of Drive-by-Wire systems. It is suggested that a Drive-by-Wire system in general must be designed as a distributed fault-tolerant system and some essential design features are identified:

Timing: Drive-by-Wire systems are hard real time. This means that not only the correctness of a calculated output value is vital, but also the point in time that this value is produced. This can be fulfilled by a systematically time-controlled architecture.

Fail-Silent Principle: In case a component of the system fails to produce correct results at
the specified time intervals, then there is a high risk of having the whole system malfunctioning. To reduce this risk the components act in a fail-silent manner. This means that when a component detects deviations from its predefined operation then it must switch into a state regarded as uncritical for the external world, e.g. by disconnecting from the rest of the system. The advantage of this concept is that it allows the system to fall into known states when a fault occurs and it allows fault tolerance with the use of redundant components. This feature is essential in order to build a *gracefully degradable* system, which is a system where the user does not see errors, except perhaps as a reduced level of system functionality.

**Distributed System:** This allows to break the system into modules that can be adjusted to the vehicle, depending on their operation. It also increases the availability of the system, reduces complexity and maintenance costs and provides scalability, flexibility and testability [1.7].

**Communication:** The flow of data among the subsystems plays a key role. It must provide data integrity, error detection, detection of hidden faults, consistency, fault recovery and fault tolerance [1.8]. Communication in Drive-by-Wire systems is performed through high-speed and usually time-triggered networks. Some typical examples include the Controller Area Network (CAN) [1.9], the Time Triggered Protocol (TTP/C) [1.10], the Time Triggered CAN (TT-CAN) [1.9], ByteFlight [1.11] and FlexRay [1.12].

The following sections present some proposed architectures of by-wire systems.

### 1.2.1 Steer-by-Wire

A steer-by-wire system replaces the traditional mechanical linkage between the steering wheel and the road wheel actuator (e.g. a rack and pinion steering system) with an electronic connection. The missing steering column's functions must be reproduced in both directions of action. In the forward direction the angle set by the driver in the steering wheel is measured by a steering angle sensor and is transferred (after the necessary processing to apply the steering algorithm) to the wheels. In the reverse direction, a displacement sensor measures the torque that occurs at the wheels and which is fed back to the driver as a counter torque on the steer-
ing wheel, closing the loop. Figure 1.2 shows the conceptual architecture of the steer-by-wire system.

![Figure 1.2: Steer-by-wire system concept](image)

An example of a steer-by-wire system with central control is presented by Kaufmann et al. [1.13]. The system is divided in three main subsystems as shown in Figure 1.3. The first is the hand wheel subsystem that provides torque feedback to the driver and position information on the driver's desired position for the road wheels. This subsystem contains position sensors, torque sensors and a mechanism to provide torque feedback to the driver. The second is the road wheel subsystem that positions the wheel according to the input provided by the driver. The road wheel subsystem is also responsible for providing the road force information. All the processing is performed by the control subsystem that calculates the desired hand-wheel torque and the road wheel position commands, as well as other advanced functions. These functions include:

- Torque Assist: The road forces on the wheels are summed to represent the equivalent force that would be present on a rack steering system. This value is then used to calculate the desired hand wheel torque.

- Return to Center: It provides a force to centralise the steering that is a function of the vehicle speed and the steering wheel angle away from center.

- Active Damping: It provides additional stability to the vehicle.

- Variable Ratio: Different relationship between the road and the steering wheel angles, depending on driving modes e.g. when parking.
1. Introduction

- Ackerman Correction: Adjustment of front wheels so Ackerman steering is achieved. The Ackerman concept is to have all four wheels rolling around a common point during a turn.

- Stability Correction: Spectral conditioning by filtering provides additional customisation of what the driver will feel.

Other functions can also be added and the steer-by-wire system can also be interconnect with other systems like an autonomous vehicle control or a collision avoidance system.

In [1.14] the control is divided into two parts, the steering controller and the steering wheel controller. The driver's steering command is detected by the angle sensor and is modified by a microprocessor system according to various vehicle dynamic variables such as the vehicle speed and the yaw rate. Then the driver's command becomes a reference value for the steering controller which in turn feeds it to an electric motor. This motor is mounted to the front axle and acts as a steering actuator. A second motor acts like a feedback actuator. It is coupled to the steering wheel and is controlled by the steering wheel controller. Some measures which have been proposed to comply with the safety requirements of such a system are:

- Redundant detection of the steering wheel angle by using multiple sensors.
1. Introduction

• Redundant detection of the steering angle of the front axle by using multiple sensors.

• Secure actuation of the steering wheel motor via the control signals by using an enable signal.

• Secure actuation of the steering motor via the control signals by using an enable signal.

• Monitoring of the sensor signals in terms of plausibility checks and analytical redundancy.

• Monitoring the micro-computer modules by means of a monitor module and vice versa.

The X-By-Wire project proposes a completely distributed system that is divided in three nodes: the steering actuator unit, the Steer-by-Wire control unit and the steering wheel unit. The three nodes are connected in a double channel TTP/C data bus and they consist of multiple micro-controllers to achieve fault tolerance. The architecture of that system is shown in Figure 1.4.

![Figure 1.4: X-By-wire system architecture, from [1.5, 1.6]](image)

1.2.2 Brake-by-Wire

The first approach towards a brake-by-wire system is the electrohydraulic brake system [1.15]. This system includes electric motor driven pumps to generate high pressure, that is then transformed to brake pressure by a high pressure accumulator. The pressure in each brake is controlled by a closed loop pressure control.
A brake-by-wire system without mechanical or other backup is described in [1.4]. This system consists of four electromechanical wheel brake modules with local microcomputers, an electromechanical brake pedal module, a duplex communication bus system and a central brake management computer as shown in Figure 1.5.

![Figure 1.5: Brake-By-wire system architecture, from [1.4]](image)

The real-time communication system and the power system have dynamic redundancy with hot standby. The pedal module must be fail-operational after one failure in the sensors, electronics or plug connections. Higher level functions such as ABS, TCS, ESP and the master supervision functionality of the brake-by-wire system are mainly implemented in the integrated software of the fail-silent central controller module.

1.2.3 Drive-by-Wire

An interesting implementation of a drive-by-wire system is SIRIUS 2001 from Chalmers University of Technology [1.16]. The design task of this project was to build a drive-by-wire car with four wheel steering and both left and right hand steering. All the basic functions of driv-

---

[2] Hot standby redundancy means that the redundant module is powered and in full operation even when the primary module operates correctly. This reduces the delay of switching to the redundant module when a fault occurs in the primary.
1. Introduction

Figure 1.6: SIRIUS system electrical architecture, from [1.16]

...ing (steering, braking, accelerating, decelerating) are handled by the same system following a global control strategy. The system was fitted in a Lotus Super 7 replica car.

The general control of the car is divided in two subsystems: one global control that controls the behavior of the vehicle as a whole and the individual local controls that control the actuators. The sensor adaptation part acts as an interface to the global control and may also check the sensor data for errors.

Figure 1.6 shows the electrical architecture of the system with the steering wheel, brake pedal, clutch and throttle. There are six computer nodes; the wheel nodes control the steering and braking system that exist in each wheel, while the central nodes control the system inputs. All the nodes communicate through a TTP/C bus.

1.3 Reliability of Semiconductor Devices

The systems briefly described in previous sections rely heavily on high performance microprocessors, however there are a number of known reliability issues associated with ICs. Semiconductor materials are by nature sensitive to ionising radiation. Radiation-induced faults in microelectronic circuits are caused when charged particles lose energy by ionising the medium through which they pass, leaving behind electron-hole pairs. The main sources of radiation that affect digital circuits are alpha particles from radioactive impurities contained in IC package materials, and high-energy neutrons from the cosmic rays which constantly bombard the earth.
These result in non-destructive soft errors (transient pulses in logic or bit-flips) known as single event upsets (SEUs), or permanent failures in the circuitry known as either single event latchups (SELs) or single event burnouts (SEBs). Permanent failures (or hard errors) have been successfully dealt with in current CMOS technologies. On the other hand, soft error rates resulting from SEUs are expected to increase as feature sizes continue to decrease [1.17]. Furthermore, the occurrence rate of on-chip electrical disturbances such as electromagnetic coupling between wires also increases due to the smaller distances between interconnects [1.18].

Soft errors were initially observed in DRAM memories in the late 70's [1.19], although SRAMs and memory elements in general are equally susceptible to radiation induced errors. This was a problem of electronic systems found mainly in aeroplanes and space applications, since the error rate at high altitudes is much higher than at sea level. This has been dealt with through the use of radiation hardened components and large scale redundancy. However, as the VLSI scaling trends have a negative impact on reliability on the sea level, other commercial sectors such as the automotive are directly affected, considering also the fact that modern vehicles are dominated by electronic systems. The solutions adapted in the aerospace industry are not in accordance with the cost and packaging constraints of the automotive industry. A very promising solution is the use of system-on-chip components that use fault-tolerant techniques on their internal architecture. Such a solution is presented in this thesis with the design of an embedded fault-tolerant microprocessor core.

1.4 Contributions

The main objective of this thesis is the design of a fault tolerant system-on-chip architecture, suitable for a drive-by-wire system, and other systems with similar requirements. Useful information on fault tolerant microprocessors can be found in relevant published research. Typically the parts of the processor that are of major concern for researchers are on-chip memory arrays such as caches and register files because they occupy a substantial percentage of the chip's area and contain a large number of transistors. This thesis takes a different approach which complements current techniques by focusing on the control part of the processor and in particular on its pipeline unit. The pipeline is the “heart” of the microprocessor and is a vital part for its correct and continuous operation. It also contains a large number of memory elements, associated with both data and control operations. This thesis presents a novel multiple core system based on
applying redundancy at the pipeline level. The main advantages of the design are:

- Masking of all single errors and many cases of multiple errors.
- Fail-silent capability, due to the detection of all multiple errors.
- Error masking is performed without introducing any delay.
- Easy application of the concept to many different microprocessors.

Although the approach is completely general and scalable, the research has implemented a method in order to provide validation evidence. During the design stages, when no prototype is available, the validation is performed by fault injection during simulation of the system. The contributions of the thesis in this area are:

- Fault injection methods on VHDL models using VHDL entities and simulator tools.
- Fault injection support for a microprocessor architecture on an FPGA prototype.
- Fault modelling of multiple faults in a microprocessor architecture.
- Detailed analysis of the effects of single and multiple errors in scalar pipelined microprocessors. This study compares a normal microprocessor with its fault-tolerant version.

Finally, Appendix A provides some initial thoughts on the applications of a hazard and operability study on embedded architectures.

1.5 Thesis overview

Chapter 2 describes the design of safety-critical computer systems and gives the necessary theoretical background. Important terms are explained and various design methods are reviewed.

Chapter 3 provides a theoretical background on the reliability issues concerning semiconductor devices. It also reviews different fault tolerant computer systems.

Chapter 4 presents a novel fault tolerant architecture for the protection of the pipelined execution unit of a microprocessor.
Chapter 5 deals with fault injection techniques on VHDL models of microprocessors.

Chapter 6 presents a detailed study of the effects of errors on a pipelined microprocessor and validates the architecture that is presented in Chapter 4.

Chapter 7 concludes the thesis, and proposes areas for further study.
References


THE development of commercial electronic systems, whether they contain simple, single devices or complex multiprocessor systems passes through specific steps. The first step is to define the general concept of the system. This means that the desired operation is studied in order to define the detailed system specifications. This stage is very important for both the design cycle but also for the success of the final product. Incomplete specifications will increase the time to market as the design stage will often need clarifications and can potentially lead to a defective product. Once the system specifications are available the design stage will identify the optimal strategy to implement the system. This includes decisions on the system architecture, hardware and software partitioning, tools to be used, etc. The next step is to implement the system and finally test it to confirm that it meets the initial specifications. In the case of a safety-critical system, this procedure is not enough to guarantee the overall safety and reliability. A number of methods, rules and guidelines for the design cycle have been developed in order to make sure that the final system will meet its safety requirements.

This chapter gives the basic theoretical background needed for the design of safety-related computer systems. It describes all the stages of a design and presents the relevant standards and guidelines, following mostly examples from the automotive sector. Finally it presents the basic techniques that are applied in fault tolerant computer systems.

2.1 Definitions and Concepts

This section sets out some of the fundamental definitions involved in designing a safety critical system.
2. Issues in the Design of Safety-Critical Computer Systems

2.1.1 Safety and Dependability

Before continuing it is important to give a set of fundamental definitions that are required for a full understanding of this thesis. Since the subject is involved with safety-critical computer systems, a good start would be to present the definition of these terms as presented by Redmill et. al [2.1]:

*Safety is freedom from those conditions that can cause death, injury, occupational illness, or damage to or loss of equipment or property, or damage to the environment.*

*A system is an organised set of complementary, interacting parts, which achieves in its environment a defined objective as the result of the properties, capabilities and behaviours of both the parts and their interactions.*

Safety in a system can be achieved by minimising the emergence of risks and hazards [2.1]. Charette [2.2] defines risk as an event or action that has a loss associated with it, it involves uncertainty, chance and some element of choice. In terms of system safety, risk expresses the combination of the probability of a failure and the effect of such failure. A hazard is a physical situation, often following from some initiating event, that can lead to an accident. The notion of safety depends on the nature of the application under consideration and can take many different forms. For example safety in an automotive system is mainly associated with the avoidance of an accident while in an Internet application that involves electronic commerce it is associated with confidentiality and network security. The levels of safety are also decided according to the needs of each system, as it will be explained in the following sections. In terms of computer systems, several parameters exist in an attempt to measure the levels of safety. In general, computer systems that are used in safety-critical applications need to be dependable. Dependability [2.3] is defined as:

*...the property of a computing system which allows reliance to be justifiably placed on the service it delivers.*

There is a number of other attributes [2.4] that characterise dependability:

- **Availability**: The probability that the system will be functioning correctly at any given time.
2. Issues in the Design of Safety-Critical Computer Systems

- **Reliability**: The probability of a component, or a system, functioning correctly over a given period of time under a given set of operating conditions.

- **Integrity**: The absence of improper system alterations. This means that the system is able to detect any occurring faults and either correct them or inform the environment (e.g. interface with other system, human operator) about them. The term integrity is also used with a broader meaning in terms such as "safety integrity level" as it will be explained later. In this case it is used as a synonym for dependability.

- **Maintainability**: The ability to undergo modifications and repairs. Maintenance can be either corrective, in case of a reported system fault, or preventive, in case of discovery of a potential threat to the system’s operation. It also includes the scope for improvement of the system's functions and operations.

2.1.2 Faults, Errors, Failures

The overall dependability of a system, and thus its safety, are negatively affected in the presence of faults. A *fault* in general is defined as a defect within the system. This defect can remain hidden in the system without manifesting itself and without altering its operation. In this case the fault is characterised as *dormant*. Under certain conditions the fault may produce an *error*, which is a deviation from the required operation of a system or sub-system. The presence of an error can be detected by a system and further actions can be taken in order to remove it. An error that remains undetected in the system is called *latent*. Finally an error may lead to a system (or sub-system) failure, which occurs when the system fails to perform its required function. This is summarised in Fig. 2.1. In more general terms a system failure includes also the case where the system fails to provide the service expected (not just the one required in the system's specification), since usually human expectations are subjective and often poorly documented.

![Figure 2.1: Fault, errors, system failure](image-url)
Faults and failures can be classified in many different categories according to various viewpoints. A very detailed presentation of faults, errors and failures classification can be found in [2.4]. Here, the categories that are most relevant to the work of this thesis are reviewed.

Based on the cause of their occurrence, faults can be classified as:

- **Random faults**: They are associated with mechanical, electrical and electronic component failure. Such components, even while operating within their correct operating environment, fail randomly. These failures that usually occur because of aging, wear and tear and also the characteristics of the operating environment are normally well understood with the use of statistical data. This in turn allows the overall performance of the system to be predicted.

- **Systematic faults** cover many different forms, including system design faults (mechanical and/or electrical), software development faults and electromagnetic coupling. These faults are not random since they are designed into the system, making them difficult to model through statistical analysis. Systematic faults occur in the same manner in a given set of circumstances. It is very difficult to predict all possible systematic faults and embed in the system suitable mechanisms to mitigate their effects.

- **Systemic faults** are faults that pervade the whole system. They are caused by omissions or faults in the specification of the system, or by incorrect interpretation of the specification by the system designer. They are similar to systematic faults but are unlikely to be discovered by normal fault detection methods.

Faults can also be classified based on their duration:

- **Permanent faults**: These are faults that remain in existence indefinitely, or until some corrective action is taken. Most of the hardware component faults and design faults are permanent. Permanent faults are also caused by external causes that result in irreversible physical changes.

- **Transient faults**: They appear because of temporary environmental conditions for a period of time and then they disappear. Although they are transient, they may change the state of the system even after they disappear and some action must be taken to avoid a
failure. Examples of such faults are the effects of alpha particle strikes on a memory device, or a noise spike.

- **Intermittent faults**: These faults appear, disappear and then reappear at some later time. They can be triggered by specific environmental conditions such as high or low temperature, or they can occur because of unstable or marginal hardware.

Identifying the differences between permanent, transient and intermittent faults is sometimes difficult. Permanent faults often manifest their presence as transient, since although they remain dormant in the system, they are activated only when certain conditions are met. This makes it difficult for the designers to detect them, since very often it is necessary to reproduce these certain conditions. The distinction between transient and intermittent faults can also be difficult. The basic difference is that the cause of intermittent faults is situated within the system’s boundary and that all the appearances of the fault are identical. On the other hand transient faults can be caused by external factors and they are random in nature.

A final classification can be made based on the extent of faults. When a fault affects only a module of a system it is called a **localised fault**. If the effects of a fault can reflect to the whole system then we have a **global fault**. A typical design goal is to keep the faults localised on a specific module or sub-system where it is possible, or easier to control their effects. Potential global faults exist in every system and it is always essential to identify them during the design stage in order to develop strategies against them.

Table 2.1 summarises the fault categories that have been discussed and presents some examples for each category.

As mentioned before, an error is a deviation from the required operation of a system, a sub-system or a module. Errors that are not detected can lead to a failure. This depends mainly on two factors [2.4]:

1. The system’s structure: When any form of redundancy exists within the system it is possible to detect and mask an error before it can propagate and cause a failure in the system.

2. The system’s behaviour: The part of a system where the error occurs may not be used in any operation, or the error can also be eliminated (e.g. overwritten) before this part is
2. Issues in the Design of Safety-Critical Computer Systems

<table>
<thead>
<tr>
<th>CAUSE</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>e.g. a burned capacitor on the circuit board</td>
</tr>
<tr>
<td>Systematic</td>
<td>e.g. an incorrectly implemented software algorithm</td>
</tr>
<tr>
<td>Systemic</td>
<td>e.g. a selection of a correct but inappropriate software algorithm</td>
</tr>
<tr>
<td>Permanent</td>
<td>e.g. an on-chip short circuit</td>
</tr>
<tr>
<td>Transient</td>
<td>e.g. transient change of logic state due to a particle hit on a digital circuit</td>
</tr>
<tr>
<td>Intermittent</td>
<td>e.g. a fault resulting from a poor solder joint</td>
</tr>
<tr>
<td>Localised</td>
<td>e.g. faults occurring only on RAMs</td>
</tr>
<tr>
<td>Global</td>
<td>e.g. fault on the power supply of a circuit board</td>
</tr>
</tbody>
</table>

Table 2.1: Main fault categories

Similar to faults, failures are classified according to four different viewpoints [2.4]:

1. Failure Domain

   - **Content Failures:** The content of the system's output deviate from its specified function (e.g. the output result of an adder is wrong based on the input operands).
   - **Timing Failures:** The service provided by the system deviates in time from that specified. This can either mean that a result either appears later or earlier than it should appear according to the system's specifications.
   - **Halt Failures:** In this case both the content and the timing are incorrect and the system is halted. This means that for an external "observer" (e.g. human operator, different system with direct interface), the system's state does not change and no service is provided.
   - **Erratic Failures:** In this case the system is operating but the service it provides is erratic (e.g. wrong messages are randomly transmitted in a network).

2. Detectability

   - **Signaled Failures:** The failure in the system is detected by appropriate mechanisms that provide a warning signal to the system that receives the service.
   - **Unsignaled Failures:** Failures that are undetected.
3. Consistency

- **Consistent Failures**: The failure in the system is perceived by all system users in the same way.

- **Inconsistent Failures**: Some, or all of the system users perceive the failure differently (some may actually not perceive the failure). These failures are also referred as Byzantine [2.5].

4. Consequences

*From*

- **Minor Failures**: The cost of the consequences of the failure are not significant.

*to*

- **Catastrophic Failures**: The cost of the consequences of the failure is very high.

When examining computer systems, the nature of faults, errors and failures can be better perceived by using a three-universe model [2.6]. The first universe is the physical universe which is where the faults occur. The physical universe contains all the physical entities that constitute the system. These include semiconductor devices, circuit boards, simple circuit elements, power and network cables, displays, keyboards etc. The second universe is the informational universe. The informational universe is where errors resulting from the faults in the physical universe occur. Errors affect data structures, like bits, bytes or words and also any operation that uses digital logic such as control signals. The last universe is the external universe where the effects of faults and errors becomes apparent through a system failure. The external universe can consist of a human user, or other system(s) that use the services of the system under study. The three-universe concept is depicted in Figure 2.2 by extending Figure 2.1.

It must be emphasised, that the terms fault, error and failure are not absolute. This means that the same event can be correctly characterised as a fault, error or failure, depending on the boundaries that are set for the various sub-systems that form the system under study. For example, a short-circuit within the boundaries of a circuit is a fault. This fault, if activated by some specific input, will produce an incorrect output of the circuit, thus causing an error. If this error is latched into a register, then a failure is caused in the system that consists of the circuit and the register. By moving up one level of abstraction, the register then holds a fault within
2. Issues in the Design of Safety-Critical Computer Systems

2.1 Introduction

The design of safety-critical computer systems requires a comprehensive understanding of the potential issues that may arise during their operation. These issues can range from hardware failures to software errors and can have significant consequences in systems where safety is paramount. The three-universe model for faults, errors, and failures is a useful tool for understanding these issues. Figure 2.2 illustrates this model, where faults in the physical universe may lead to errors in the informational universe, which in turn may lead to failures in the external universe.

![Three-Universe Model](image)

**Figure 2.2**: The three-universe model for faults, errors, and failures

To design a dependable system, it is necessary to take certain measures to overcome the effects of faults. These include:

- **Fault avoidance**: Techniques applied in the design stage to prevent systematic and systemic faults. The most important methods are discussed in Section 2.2.

- **Fault removal**: Methods that involve validation and testing of the system after the design stage. Testing through fault injection is mainly discussed in Chapter 5.

- **Fault detection**: Techniques used during system operation to detect and avoid unsignaled failures.

In order to design a dependable system, there are some measures that have to be taken that will overcome the effects of faults [2.4, 2.7]:

- Fault avoidance: These techniques are applied in the design stage and focus on avoiding systematic and systemic faults. The most important methods are discussed in Section 2.2.

- Fault removal: These methods include validation and testing of the system after the design stage. Testing through fault injection is mainly discussed in Chapter 5.

- Fault detection: Fault detection techniques are used when the system is operating and their main purpose is to avoid unsignaled failures.
• Fault tolerance: Similar to fault detection, fault tolerance is designed within the system so that it can operate correctly with the presence of faults. Fault detection and fault tolerance are discussed in the following sections but they are also the main subject in Chapter 3 and Chapter 4.

None of the above methods can be perfectly applied, so a combination of all is always necessary.

2.1.3 Reliability

Reliability is an important parameter of a system because it can be quantified and give a clear picture of the overall dependability and safety. This is very important when evaluating a system since it is possible to generate figures for comparisons among different solutions. As mentioned before, reliability is the probability of a component, or a system, functioning correctly over a given period of time under a given set of operating conditions. This can be expressed in mathematical terms as follows: If we have a number of identical components $N$, operating under the same conditions and at a given time $t$ the number of correctly functioning components is $N_C(t)$ and the number of faulty components is $N_F(t)$ then the reliability of these components is:

$$ R(t) = \frac{N_C(t)}{N} $$

The unreliability is then defined as:

$$ F(t) = \frac{N_F(t)}{N} $$

Obviously, at any time $t$:

$$ R(t) + F(t) = 1 $$

The rate in which failures occur is expressed by the hazard function, (also known as hazard rate, or the failure rate function):

$$ z(t) = \frac{1}{N_C(t)} \frac{dN_F(t)}{dt} $$
With simple calculations, \( z(t) \) can be expressed in terms of reliability and unreliability as:

\[
z(t) = \frac{dR(t)}{dt} = \frac{dF(t)}{dt} - \frac{1}{R(t) - 1 - F(t)}
\]

(2.1)

It has been proven empirically that the hazard function for electronic components has the form of Figure 2.4, also known as the *bathtub curve*. Early in time the failure rate is high since a number of electronic components can be defective just after being manufactured. This is called *infant mortality period*. The middle phase is the *useful life period* where the failure rate is constant and at the end of the curve the failure rate increases as it enters the *wear-out period* where components start to experience failures due to wear and tear. During the useful life period, when the failure rate is constant we get:

\[
z(t) = \lambda
\]

and from (2.1),

\[
\frac{dR(t)}{dt} = \lambda
\]

The solution of this differential equation gives:

\[
R(t) = e^{-\lambda t}
\]

This is known as the *exponential failure law* and it is the most commonly used model, especially when calculating the reliability of systems that consist of several electronic components. However, there are cases where the failure rate is not constant during the system’s life. For example, the reliability of an embedded system does not depend only on its electronic components, but also on its software. Software may contain faults, but it can be upgraded with versions that correct them. This means that the software’s reliability in this case increases, thus the overall embedded system’s reliability is not constant. For this reason the failure rate function can be defined from various distribution functions (e.g. Weibull [2.8]) that can be suitable to model it [2.9].

A different approach to define the above parameters is to start from the *mortality function* \( f(t) \) which expresses the rate at which components are failing, referenced to the original population.
2. Issues in the Design of Safety-Critical Computer Systems

\[ f(t)dt \] is the probability of a component failing in the interval \((t, t + dt)\). In that case, the unreliability is the integral of \(f(t)\):

\[ F(t) = \int_0^t f(t)dt \]

and it expresses the probability that a component will have failed by time \(t\).

The hazard function can be expressed according to the mortality function as:

\[ z(t) = \frac{f(t)}{1 - F(t)} = \frac{f(t)}{R(t)} \]

### 2.1.4 Mean Time to Failure

Another parameter that is used to express the expected reliability of a category of components, or systems is the mean time to failure (MTTF). If we have a group of \(N\) identical component that each one fails for the first time in \(t_i\) then the mean time to failure for these components is defined as:

\[ MTTF = \frac{\sum_{i=1}^{N} t_i}{N} \]
2. Issues in the Design of Safety-Critical Computer Systems

It can be shown that:

\[
MTTF = \int_{0}^{\infty} R(t)dt
\]

This means that for a system with a constant failure rate \( \lambda \), the mean time to failure is:

\[
MTTF = \int_{0}^{\infty} e^{-\lambda t}dt = \frac{1}{\lambda}
\]

It must be noted that in this case, the probability of a component operating correctly for a time equal with its mean time to failure is:

\[
R(MTTF) = e^{-\lambda \frac{1}{\lambda}} = e^{-1} = 0.368
\]

This means that a system that follows the exponential failure law will operate without failure until a time equal to its mean time to failure with a probability of 36.8%, or in other words it will have failed by that time with a probability of 63.2%.

2.1.5 Mean Time to Repair

The mean time to repair (MTTR) is the average time taken to repair a system that has failed and get it operational. Similarly to the MTTF, the MTTR can be expressed as:

\[
MTTR = \frac{\sum_{i=1}^{N} t_i}{N}
\]

assuming that \( t_i \) is the time needed to repair the \( i^{th} \) of the \( N \) faults. The mean time to repair is generally difficult to calculate and it is usually expressed according to the repair rate \( \mu \). It is thus:

\[
MTTR = \frac{1}{\mu}
\]

2.1.6 Mean Time Between Failures

Assuming that after a repair the system operates as it was operating before the occurrence of the failure, the mean time between failures (MTBF) is given by:
2. Issues in the Design of Safety-Critical Computer Systems

\[ MTBF = MTTF + MTTR \]

Since in most of the cases the time to repair is many orders of magnitude smaller than the mean time to failure, MTBF and MTTF are practically equal. However the concept of these two parameters is different.

2.1.7 Failures in Time

Another commonly used unit used to express failure rates in computer systems is failures in time (FIT). FIT expresses the number of failures in a billion \((10^9)\) hours. In many cases FIT is used to express the failure rate of a computer system instead of MTTF or MTBF because it is additive.

2.1.8 Availability

As mentioned previously, the availability \((A)\) of a system is defined as the probability that the system will be functioning correctly at any given time. This can be expressed mathematically based on the previous terms as:

\[ A = \frac{MTTF}{MTTF + MTTR} \]

In many cases it is important for a system not only to operate correctly, but also to operate continuously without interruptions or performance degradation. Availability expresses the ability of a system to satisfy the above requirements. As an example, computer systems found in communication networks are mainly concerned with having very high availability.

2.1.9 Reliability Modelling

It is often required to calculate the overall reliability of a system that consists of several other components with known reliability (e.g. discrete electronic components). It is practically impossible to achieve this with experimental means. One reason is that the cost of producing a big number of system replicas is very high. The other important reason is that failures occur very
sporadically and with large intervals between them (large MTTF). There are analytical meth­ods to calculate reliability, the most important being the methods for combinatorial models and Markov modelling. Several examples of application of analytical methods for the calculation of a system’s reliability can be found in the literature (e.g. [2.10, 2.11, 2.12, 2.13, 2.14]). The following sections provide the basic knowledge for these methods.

**Series**

A system where all subsystem/components are connected in series is non-redundant. This means that in order for the system to maintain correct operation, each and every one of its components or subsystems must maintain correct operation. For example a multi-stage ampli­ifier will perform amplification only if the preamplifier and all subsequent gain stages are working correctly.

Figure 2.5 shows the block diagram of a series system consisting of \(N\) components. Assuming that the event that component \(i\) is operational at time \(t\) is represented by \(O_i(t)\), then the reliability of the overall system equals with the probability that all system components are operating correctly at time \(t\):

\[
R_{\text{series}}(t) = P(O_1(t) \cap O_2(t) \cap \cdots \cap O_N(t))
\]

If all the events \(O_i(t)\) are independent, then the reliability of a series system is:

\[
R_{\text{series}}(t) = R_1(t)R_2(t)\cdots R_N(t) = \prod_{i=1}^{N} R_i(t) \tag{2.2}
\]

For components that follow the exponential failure law, the equation 2.2 will give:

\[
e^{-\lambda_{\text{series}}t} = e^{-\lambda_1 t}e^{-\lambda_2 t} \cdots e^{-\lambda_N t} = e^{-(\lambda_1 + \lambda_2 + \cdots + \lambda_N)t}
\]

So, the failure rate of a series system can be calculated from the failure rates of the individual components as:

\[
\lambda_{\text{series}} = \lambda_1 + \lambda_2 + \cdots + \lambda_N = \sum_{i=1}^{N} \lambda_i
\]
2. Issues in the Design of Safety-Critical Computer Systems

Parallel

Contrary to a series system, parallel systems need at least one of their components operating correctly, in order for the whole system to operate correctly. The block diagram of a parallel system that consists of \( N \) components is shown in Figure 2.6. If the event that component \( i \) has failed at time \( t \) (as opposed to being operational in the previous case) is represented by \( C_i(t) \), then the unreliability of the overall system is:

\[
F_{\text{parallel}}(t) = P(C_1(t) \cap C_2(t) \cap \cdots \cap C_N(t))
\]

or

\[
F_{\text{parallel}}(t) = F_1(t)F_2(t)\cdots F_N(t) = \prod_{i=1}^{N} F_i(t)
\]

Since \( R(t) = 1 - F(t) \) for any system, the reliability of a parallel system is given by:

\[
R_{\text{parallel}}(t) = 1 - \prod_{i=1}^{N} (1 - R_i(t)) \tag{2.3}
\]

Complex Systems

In practice, systems are more complex than parallel or series systems. Often they can be represented as a combination of the above, as illustrated in the example of Figure 2.7. In this case the reliability can be calculated by systematically reducing the block diagram, starting from the simple sub-system structures that form a system in series or in parallel, with the use of Equations 2.2 and 2.3.

Other categories of complex systems are the various fault-tolerant redundant structures such as pairs and M-of-N systems. These structures and their reliability are reviewed in Section 2.4 that deals with fault tolerance.
Markov Modelling

In many cases it is very difficult to make a reliability block diagram of a system in order to determine its overall reliability. Furthermore, it is not possible to depict properties of the system such as repair, or fault-detection, which also play an important role in the calculation of the system reliability. In this case, a very popular method is the application of Markov modelling. The mathematical base of Markov modelling are the Markov random processes, a detailed analysis of which can be found in [2.15, 2.16].

A Markov model consists of a set of $N$ states that are mutually exclusive and each represents a specific combination of operating and non-operating system components, or sub-systems. Each state $i$ corresponds a state probability $P_i$ that is defined as:

$$P_i(t) = P[\text{System state } = i \text{ at time } t]$$
The second element of the Markov model is a set of transition probabilities, which simply define the probability of a transition from state \( i \) to state \( j \):

\[
 p_{ij} = P\{\text{System state}=j|\text{System state}=i\}
\]

The system's behaviour can be defined by using the following equation:

\[
P_i(t + \Delta t) = \sum_{j=1}^{N} p_{ji} P_j(t) \quad \text{for} \quad 1 \leq i \leq N
\]

or, by using a matrix notation:

\[
P(t + \Delta t) = P(t) \times P_t(t)
\]

where

\[
P_t = \begin{pmatrix}
p_{11} & p_{12} & \cdots \\
p_{21} & p_{22} & \cdots \\
\vdots & \vdots & \ddots
\end{pmatrix}
\]

Figure 2.8 shows the Markov model of a system with two states and a constant failure rate \( \lambda \). The state \( S_0 \) corresponds to an operational system, while state \( S_1 \) corresponds to a system that has experienced a failure. This means that the reliability \( R(t) \) and the unreliability \( F(t) \) are equal with the probability \( P_0(t) \) of the system being in state \( S_0 \), and the probability \( P_1(t) \) of the system being in state \( S_1 \) respectively.

![Markov model of a system with no repair](image)

Figure 2.8: Markov model of a system with no repair

From 2.4 we get:

\[
\begin{pmatrix}
P_0(t + \Delta t) \\
P_1(t + \Delta t)
\end{pmatrix} = \begin{pmatrix}
P_0(t) & P_1(t)
\end{pmatrix} \times \begin{pmatrix}
1 - \lambda \Delta t & \lambda \Delta t \\
0 & 1
\end{pmatrix}
\]
which gives

\[ \begin{align*}
P_0(t + \Delta t) &= (1 - \lambda \Delta t)P_0(t) \\
P_1(t + \Delta t) &= \lambda \Delta t P_0(t) + P_1(t)
\end{align*} \]

and leads to

\[ \begin{align*}
\frac{(P_0(t + \Delta t) - P_0(t))}{\Delta t} &= -\lambda P_0(t) \\
\frac{(P_1(t + \Delta t) - P_1(t))}{\Delta t} &= \lambda P_0(t)
\end{align*} \]

By taking the limit of \( \Delta t \) approaching to zero, then the above equations become differential:

\[ \begin{align*}
\frac{dP_0(t)}{dt} &= -\lambda P_0(t) \\
\frac{dP_1(t)}{dt} &= \lambda P_0(t)
\end{align*} \]

By solving the above equations we get:

\[ \begin{align*}
R(t) &= P_0(t) = e^{-\lambda t} \\
F(t) &= P_1(t) = 1 - e^{-\lambda t}
\end{align*} \]

Figure 2.9 shows the Markov model of a similar system that also has repair functionality. In this case the probability of being in the state \( S_0 \) is equal to the availability of the system \( A(t) \).

The state transition matrix of this model is:

\[ P_t = \begin{pmatrix}
1 - \lambda \Delta t & \lambda \Delta t \\
\mu \Delta t & 1 - \mu \Delta t
\end{pmatrix} \]

and the set of differential equations is:
2. Issues in the Design of Safety-Critical Computer Systems

Figure 2.9: Markov model of a system with repair

\[ \frac{dP_0(t)}{dt} = -\lambda P_0(t) + \mu P_1(t) \]
\[ \frac{dP_1(t)}{dt} = \lambda P_0(t) - \mu P_1(t) \]

After solving these equations we get:

\[ A(t) = P_0(t) = \frac{\mu}{\lambda + \mu} - \frac{\lambda}{\lambda + \mu} e^{-(\lambda+\mu)t} \]
\[ P_1(t) = \frac{\lambda}{\lambda + \mu} + \frac{\lambda}{\lambda + \mu} e^{-(\lambda+\mu)t} \]

It must be noted that it is possible to simplify the state diagram of a Markov model by omitting the \( \Delta t \) terms, as shown in Figure 2.10. In this case 2.4 takes the following form:

\[ \frac{d}{dt} P(t) = P_t \times P_t(t) \]  
(2.5)

Figure 2.10: Simplified Markov model of a system with repair
2. Issues in the Design of Safety-Critical Computer Systems

2.2 Relevant Standards

This section reviews the most important standards that deal with the design and development of safety-critical electronic systems. The majority of these standards originate from specific industry sectors, but in many cases they can be more broadly used.

One of the most influential standards regarding electronic components, issued in recent years by the International Electrotechnical Commission (IEC) is IEC61508: *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems* [2.17]. The importance of this standard arises from the fact that it is generic in that it applies to safety-related systems irrespective of their application, making it a basic safety publication. IEC61508 is a substantial document consisting of eight parts:

- **Part 0:** Functional Safety and IEC61508.
- **Part 1:** General Requirements.
- **Part 2:** Requirements for Electrical/Electronic/Programmable Electronic Safety-Related Systems.
- **Part 3:** Software Requirements.
- **Part 4:** Definitions and Abbreviations.
- **Part 5:** Examples of Methods for the Determination of Safety Integrity Levels.
- **Part 6:** Guidelines on the Application of Part 2 and 3.
- **Part 7:** Overview of Techniques and Measures.

It can be used as the basis for other industry standards, but it can also be used as a stand-alone standard.

Other relevant standards are issued by military authorities such as the UK Ministry of Defence with 00-55 [2.18] (safety-related software), 00-56 [2.19] (safety management) and 5200.28 [2.20] (trusted computer systems evaluation) from the US Department of Defense. Civil standards include the DO-178B [2.21] (commercial aircraft software) DO-254 [2.22] (commercial
aircraft electronic hardware) from civil aviation and EN 50128 [2.23] (software for railway control), EN 50129 [2.24] (electronic systems for signaling) from the railway industry. From the automotive sector there exists the MISRA guidelines for the development of vehicle based software [2.25, 2.26, 2.27] that have been produced by a UK consortium of automotive companies and has now been adopted as a standard [2.28].

It must be noted that in terms of hardware design, most of the above standards examine circuits on the level of discrete components. Only recent standards such as the DO-254 define rules for the use of ASICs (Application Specific Integrated Circuit) and PLDs (Programmable Electronic Device).

2.3 System Life-Cycle

Manufacturers of safety-related systems have a legal onus of using good practice in the design of their products. Standards and guidelines define a design process that will ensure that the final product is as safe as necessary. This is done by preventing or minimising the occurrence of hazards. Since this is practically impossible to achieve in many cases, the design methodologies also consider how to control any occurring hazards and mask or mitigate their impact on the system safety.

The various stages of a project dealing with a safety-related system are described through lifecycle models. Several different models have been proposed and applied in different standards and different sectors over the years. An example of such a model is shown in Figure 2.11.

The required level of safety in every system depends on many factors, but in general the ALARP (as low as reasonably practicable) principle is followed, where the risks are classified as shown in Figure 2.12. The importance of a risk is defined by the probability of a failure and the effect of this failure. A system is usually designed to be safe as necessary and not safe as possible. This concept is shown in Figure 2.13.

The main phases of safety analysis for automotive systems based on the MISRA guidelines are [2.29]:

1. Modelling of the system: Before performing the analysis it is necessary to produce an ‘approximation’ of the system that highlights its features, its components, its intercon-
2. Issues in the Design of Safety-Critical Computer Systems

- Verification & Validation Planning
- Hardware Requirements
- Software Requirements
- Design and Implementation
- Verification & Reviews
- Validation
- Operation, Maintenance and Support

Figure 2.11: Example of a system life-cycle

- Identification and defines the system's boundaries.

2. Hazard identification: When the model is available, then all possible fault scenarios are considered and their consequences lead to the identification of possible hazards.

3. Hazard classification: When a list of hazards is available, their controllability category is assessed.

4. Assign a Safety Integrity Level (SIL) to the system: The system's SIL (levels 0 to 4) can be defined by using the categories of controllability that are defined as follows:

   Uncontrollable: This relates to failures whose effects are not controllable by the vehicle occupants and which are most likely to lead to extremely severe outcomes.
2. Issues in the Design of Safety-Critical Computer Systems

Levels of Risk Tolerability

- **Intolerable level**
  - Risk cannot be justified on any grounds

- **As Low as Reasonably Practicable (ALARP)**
  - Undesirable
    - Risk is justified only if risk reduction is impracticable or if its cost is grossly disproportionate to the improvement gained
  - Tolerable
    - Risk is justified only if cost of reduction would be disproportionate to the improvement gained

- **Broadly Acceptable**
  - Negligible Risk

*Figure 2.12: Levels of risk tolerability and the ALARP principle*

**Risk = Probability of Failure \times Effect of Failure**

*Figure 2.13: Risk reduction*

The outcome cannot be influenced by a human response.

**Difficult to control:** This relates to failures whose effects are not normally controllable by the vehicle occupants but could, under favourable circumstances, be influenced by a mature human response. They are likely to lead to very severe outcomes.

**Debilitating:** This relates to failures whose effects are usually controllable by a sensible human response and whilst there is a reduction in safety margin, can usually be expected to lead to outcomes which are at worst severe.
### Table 2.2: IEC 61508 requirements

<table>
<thead>
<tr>
<th>Level</th>
<th>Continuous Failures/Hour</th>
<th>Protection on Demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>$\geq 10^{-9}$ to $&lt;10^{-8}$</td>
<td>$10^{-5}$ to $&lt;10^{-4}$</td>
</tr>
<tr>
<td>3</td>
<td>$\geq 10^{-8}$ to $&lt;10^{-7}$</td>
<td>$10^{-4}$ to $&lt;10^{-3}$</td>
</tr>
<tr>
<td>2</td>
<td>$\geq 10^{-7}$ to $&lt;10^{-6}$</td>
<td>$10^{-3}$ to $&lt;10^{-2}$</td>
</tr>
<tr>
<td>1</td>
<td>$\geq 10^{-6}$ to $&lt;10^{-5}$</td>
<td>$10^{-2}$ to $&lt;10^{-1}$</td>
</tr>
</tbody>
</table>

**Distracting:** This relates to failures which produce operational limitations, but a normal human response will limit the outcome to no worse than minor.

**Nuisance only:** This relates to failures where safety is not normally considered to be affected and where the customer satisfaction is the main consideration.

The target failure rates for each SIL category are shown in Table 2.2. These probabilities are for hazards due to random faults. For systematic faults SIL indicates the level of rigour required in the design and validation.

5. Identify safety requirements: Based on the identified hazards, a list of possible causes is produced in order to identify the top-level safety requirements for the system.

6. Detailed safety analysis: When a SIL is assigned to the system and its sub-systems and functions then the safety requirements in terms of the physical implementation can be specified (e.g. need for redundancy, or diversity). When the general outline of the architecture of the system is defined, a more detailed safety analysis can be performed. The most popular methods are FMEA (Fault Mode and Effects Analysis), FTA (Fault Tree Analysis) or HAZOP (Hazard and operability study) [2.1].

FMEA aims to identify the ways in which a system can fail and the effects of the failures. It starts with a fault mode of a component and examines the effects of that fault.

FTA follows the opposite direction from FMEA. The analysis is done by breaking down the causes of a top-level failure by the use of ‘and’ and ‘or’ gates so that a tree is produced.

HAZOP is the application of a formal systematic technique to the identification of hazards. It examines the components and the interconnections between components so as...
to explore whether deviations from design intent are possible and if so, what might be their causes and their consequences. HAZOP is particularly powerful for exploring the interactions between parts of a system. HAZOP and its possible application on embedded architectures is presented in Appendix A.

2.4 Fault Tolerance

As has been mentioned, in most cases it is impossible to design a completely fault-free system. In this case faults, errors and even failures can be expected but must always be controlled and the necessary mechanisms to mask or mitigate their effects must be incorporated in the system. This is achieved by using fault tolerant methods. These methods can be on the hardware level, on the software level, or in both. This section reviews some basic fault tolerant solutions that are used on computer systems.

2.4.1 Fault Detection

Fault tolerance is often achieved by detecting faults. Faults can be detected as a result of an error they produce, so there are techniques that detect these errors and take the necessary action. As mentioned before, faults are associated with hardware and software, so it is possible to use hardware and software techniques to detect them. Some examples include:

*Functionality checking:* Software techniques can be used to detect hardware problems. This is achieved with routines that check to see if the hardware of the system is functioning correctly. Random-access memory (RAM) testing involves writing to and reading back from memory locations to verify its operation. This is not necessarily a proof that the device is working correctly, but over the years several very efficient algorithms have been developed for testing large memory spaces in a short period of time, with reasonable fault coverage. Read-only memories (ROM) may be tested by periodically calculating checksums and comparing these with known values. The processor or processors within a system may be checked by executing a sequence of calculations and comparing the results. In multiprocessor systems checks can be performed periodically to ensure that each processor can communicate with its neighbours.
2. Issues in the Design of Safety-Critical Computer Systems

Consistency checking: This is a purely software method. It involves the use of some knowledge of the nature of the information within the system to check its validity.

Signal comparison: In systems with redundant components it is possible to compare their signals to check their validity. A special case of signal comparison can be found in checking pairs.

Information redundancy: This includes techniques that add information that is used to check the validity of the data. Detected errors can be corrected on-the-fly. Examples include parity checking, cyclic redundancy codes and error correcting codes.

Instruction monitoring: Instruction monitoring is a feature that many processors include, which checks the validity of the operation code of an instruction and takes the necessary action if this code is corrupted.

Loopback testing: This technique is mostly used in communication lines. It involves the connections of the output of a module, as an input to the same module for verification.

Watchdog timers: Watchdog timers are used to reset a processor in case it 'crashes'. While the processor is in normal operation, the timer must be periodically reset before it reaches zero.

Bus monitoring: In this case the bus addresses are checked and compared with the allowable range of the application.

2.4.2 Redundancy

All forms of fault tolerance are achieved with some form of redundancy, which is the use of additional elements within the system that would not be required in a system completely free of faults. These additional elements may be in many different forms:

Hardware redundancy: The use of hardware components that are additional to those required to implement the system. The purpose of this is to detect and in some cases tolerate faults.

Software redundancy: The concept is similar to the hardware redundancy. Some examples are presented later in this chapter.
**2. Issues in the Design of Safety-Critical Computer Systems**

*Information redundancy:* As already mentioned, it involves the use of information in addition to that required to implement a given function, in order to detect and in some cases tolerate faults. Information redundancy can be implemented using hardware or software techniques.

*Temporal (time) redundancy:* The use of additional time to what is required to perform a given action, in order to detect and in some cases tolerate faults. It can be used to detect transient faults, for example by repeating calculations and comparing the results obtained.

*Design diversity:* This technique is used to protect the system against common-mode failures\(^1\). The main idea is to implement each redundant module in a different way. This technique can be used for both hardware and software modules.

### 2.4.3 Software fault-tolerance

*Software faults*

Even simple programs contain software faults that may take various forms including software specification faults, coding faults, logical errors with calculations, stack overflows or underflows, use of uninitialised variables, etc. Software does not fail randomly and does not degrade with age. Therefore all failures are systematic and are related to its design.

Software faults can be introduced in all the different phases of a software development project. The first of these stages is the specification and the definition of requirements for software. Where feasible, these can be minimised with the application of formal methods [2.30] (e.g. formal specification languages). The second stage is the coding process where the specifications are translated and implemented using one (or more) programming language. These faults are avoided with a number of techniques:

1. The use of structured programming techniques with high-level languages that require a specific programming discipline.

2. The use of static checking tools and similar automated design aids [2.26].

---

\(^1\) Failures usually, but not necessarily, due to design faults, that affect all redundant components in the same manner.
Finally, verification plays an important role in safety-critical software. In reality, exhaustive testing software is not an option for high integrity systems. Since the acceptable failure rate can be as low as $1 \times 10^{-8}$ there is not enough time to complete exhaustive tests. These systems must use formal verification techniques, that are however time consuming and require a high level of mathematical ability.

Since there is no way of avoiding completely the presence of software faults in the design, there are several techniques to tolerate them. The two most popular techniques are *N-version programming* and *Recovery blocks*.

It must be noted that the term "software fault-tolerance" very often implies software techniques that deal with hardware faults. Such techniques are discussed in Chapter 3.

### N-version programming

N-version programming [2.31, 2.32] involves the use of different software versions to implement the same specification. These versions have the same inputs and perform the same function using different algorithms, thus producing the same outputs. If the outputs are not the same then the action taken depends on the number of versions. If $N = 2$, the system cannot decide which result is correct, so it has to repeat the calculations hoping that the fault was produced by a transient error on the hardware. Alternatively, the system may try to perform some further diagnostics to decide which routine is in error. This problem is easily solved if more versions are used, for example if $N = 3$. In this case some form of voting can mask the effects of a fault. Although large values of $N$ have attractions from a functional point of view, the high costs involved usually make them impractical. If a single processor is used for the calculations, then the processing time increases by a factor greater than $N$, because of the complexity of the voting process. In case of $N$ different processors, then the cost is increased. The high development cost of this approach restricts its use to very critical applications where the cost can be tolerated, like the flight control system of the Airbus A330/340 aircraft.

The N-version programming effectiveness in terms of increasing software reliability has been questioned by Knight and Leveson [2.33] and a long series of discussions have followed [2.34].
Recovery blocks

Recovery blocks [2.31, 2.35] is a technique where software is divided in modules that are serially executed and afterwards some sort of acceptance test is used on the output. These tests may have several components and may for example include checks for runtime errors, reasonability, excessive execution time, or mathematical errors. For example a routine that calculates the square root of a number can have as acceptance test the squaring of its result to confirm that the original result is obtained. The structure of the recovery block mechanism is shown in Figure 2.14.

A recovery point is established at the beginning because a faulty module can cause damage to the system state, which must be repaired before correct operation can be resumed. The primary module is then executed, followed by its acceptance test. Failure of this test will result the execution of an alternative module, after which the acceptance test will be repeated. If the system fails the acceptance test for all the redundant modules, an overall software fault is detected and the system must take appropriate action.

2.4.4 Hardware fault-tolerance
Hardware fault models

The causes of a hardware fault may vary from physical failures of the components to external factors such as radiation and electromagnetic interference. In order to be able to analyse the system during its design, these faults are represented according to the effect they have on the system's behaviour. At the atomic level of a circuit the individual components such as resistors, transistors and their interconnections are considered. At this level most faults can be seen as connections which are either open-circuit or are incorrectly joined to some other line. This approach provides a good picture about the effects of faults, but becomes very complicated in large circuits. One method to reduce this complexity is to look at the circuit at a modular level. Typically the modules used would be gates or collection of gates. In order to represent faults in this level, several fault models exist [2.36]:

- **The single-stuck-at model:** This model assumes that a fault within a module will cause it to respond as if one of its inputs or outputs is stuck at a logic 1 or logic 0. It also assumes that the basic functionality of the circuit is otherwise unaffected and that the fault is permanent.

- **The bridging model:** In this model it is assumed that two or more nodes are accidentally joined together to form a permanent fault.

- **The stuck-open model:** This model is used for CMOS gates that cannot be modelled by a 'stuck-at' representation. When the output transistors of a CMOS gate are turned off, then the output tends to maintain its previous state for a time that depends on the capacitance of the gates connected to the output and the leakage circuits.

Another important category of hardware faults are radiation induced faults. These faults may produce permanent failures that fall into one of the previously mentioned categories, but more frequently they alter stored data in memory elements. This category of faults and the resulting errors and failures are the main concern of this thesis and they are thoroughly discussed in the following chapters.
2. Issues in the Design of Safety-Critical Computer Systems

Static redundancy

Static redundancy relies on the voting of the outputs of a number of modules to mask the effects of faults within these units. The simplest version of these arrangement is termed a Triple Modular Redundant system (TMR) [2.37, 2.38]. In a TMR system such as the one shown in Figure 2.15(a) three modules receive the same input signals and should under normal circumstances produce identical outputs. A voter compares the three outputs and produces the output of the system. The voter produces the output according to the majority view, which means that the system can tolerate a fault only in one module.

The TMR system of Figure 2.15(a) has two obvious single-point failures. One of them is the provision of the input signals to the system. These signals usually come from a module which could potentially exhibit a faulty behaviour and affect the whole system. To avoid this problem, modules (or inputs in general) can be duplicated or triplicated, or even use diverse modules to reduce the possibility of a systematic fault causing common failures. The other single point of failure is the voting arrangement. One approach is to design a dependable voter. Usually voters are simple, meaning that they can be carefully designed and their reliability can be calculated accurately. The other approach is to use more than one voter to generate multiple outputs. An example can be seen in Figure 2.15(b). Several such stages can be cascaded and generate a multistage TMR arrangement.

The TMR arrangement can be generalised using an arrangement of any number of modules. This is generally termed as N-modular redundancy (NMR) and in many cases an odd number of modules is used to enable a majority voting scheme to be used. In general an N-modular redundant system can tolerate the failure of $\frac{N-1}{2}$ modules without producing a system failure.
However, even in cases where fault tolerance is not possible, fault detection can still help the system to recover. For example in a TMR where two modules have failed, the voter will not be able to tolerate these faults, but will still be able to block a faulty output, unless both faulty modules produce the exact same result.

In the case of a TMR arrangement, the high reliability requirement is shifted to the voter. This is usually much simpler than the modules and therefore very high specification components can be used in this subsystem. Simple systems are more amenable to analytical formal methodology and so it is possible to reason about the safety with a high confidence level.

Assuming a perfect voter and components with equal reliability \( R \), the reliability of the TMR configuration equals with the probability of all modules operating correctly, plus the probability of two of the modules operating correctly. Hence,

\[
R_{TMR} = R^3 + 3R^2(1 - R) = 3R^2 - 2R^3
\]

If the voter has a reliability \( R_v < 1 \) then the system’s reliability becomes:

\[
R_{TMRv} = R_vR_{TMR}
\]

The above expressions can be generalised for a NMR system where \( M \) modules need to operate correctly to prevent a system failure as:

\[
R_{M-out-of-N} = \sum_{i=0}^{N-M} \left( \frac{N!}{(N-1)!i!} R^N - i (1 - R)^i \right)
\]

Dynamic redundancy

As seen above, static redundancy achieves fault tolerance by masking faults. The price for this approach is very high levels of redundancy, with at least three modules needed to tolerate one single fault. Dynamic systems follow a different approach using fault detection instead of fault masking. There is usually one main unit and a second stand-by unit is ready to take over when a fault in the primary module is detected. The success of this approach is very dependent on the fault detection process.

One of the most common dynamic redundancy systems is the standby spare arrangement that is shown in Figure 2.16(a). While no fault is detected, one single module drives the output of
2. Issues in the Design of Safety-Critical Computer Systems

Figure 2.16: Dynamic redundancy: (a) A standby spare arrangement (b) A self-checking pair

The system through a switch that is controlled by a fault detector. When a fault is detected in the operation of the first module, the system is reconfigured in order for the second module to drive the outputs of the system. This reconfiguration process causes a momentary disruption of the system. This disruption is shorter in case of a hot standby arrangement where at any time both modules are in full operation. In this case fault detection can be implemented with simple output comparisons. In a cold standby the second module is unpowered until called into service. This reduces power consumption and reduces wear and tear on the backup module, but increases the reconfiguration time and makes fault detection more difficult to implement.

Another example of dynamic redundancy is the self-checking pair shown in Figure 2.16(b). This arrangement is quite simple. The first module provides the system's output and a second module generates a reference result that is compared with the main one. When the comparator detects a fault then it activates its output. The comparator can be implemented either in hardware or software. If the modules are simple digital circuits with a few outputs, a two level XOR circuit can compare these signals. In the case where the modules are processors the comparison can be implemented in software, thus removing the need of additional hardware. To remove the possibility of a single point of failure, both processors can perform the comparison.

The reliability of a pair system depends a lot on the details of the system. Assuming that the main module has a fault coverage \( C_1 \), then the reliability of the system equals with the reliability of the main module, plus the probability of a fault occurring on the main module being detected and at the same time the second module operates correctly. This can be expressed

\[ R = R_1 + P_F R_2 \]

\( \text{where} \)

- \( R \) is the reliability of the system
- \( R_1 \) is the reliability of the main module
- \( P_F \) is the probability of a fault occurring on the main module
- \( R_2 \) is the reliability of the second module

Fault coverage is the percentage of detectable faults in a system. It is used to express the ability of a fault tolerant system in detecting faults.
as:

\[ R_{pair} = R_1 + (1 - R_1)C_1R_2 \]

**Hybrid redundancy**

Hybrid redundancy uses a combination of voting, fault detection and module switching, thus combining static and dynamic redundancy. This can be implemented with many different ways, but it can be generalised as some form of N-modular redundancy with spares as shown in Figure 2.17. When no fault is present, the system operates like a N-modular redundant system with the rest (M-N) modules remaining in standby. When a fault occurs, it is masked by the voter, but the disagreement detector identifies the module that generated the fault and replaces it with one of the standby modules.

![Figure 2.17: N-modular redundancy with spares](image)

The reliability of an NMR system with S spares is given by:

\[ R_{NMR-S} = \sum_i \left( \frac{(N+S)!}{(N+S-1)!i!} \right) R^{N+S-i}(1 - R)^i \]
2.5 Conclusion

In this chapter the main concepts of fault tolerant systems have been introduced in the context of automotive systems. In particular the chapter presents the fundamentals of fault tolerant computing in combination with the general practice followed in the design of fault tolerant automotive systems. In later chapters, hardware fault tolerance techniques and their application to protect the processor core will be discussed and be compared to a normal processor architecture. The focus will now change in the next chapters to analyse the main nature of faults and errors that affect integrated circuits.
References


CHAPTER 3: 
THE ORIGIN AND MITIGATION OF FAULTS IN 
MICROPROCESSORS

All types of electronic systems can malfunction due to external factors. The main ones causing faults within electronic IC components are radiation, electromigration and electromagnetic interference (EMI). The impact of radiation on electronics in particular has been a subject of great interest for both the semiconductor and aerospace research, that is now affecting also the automotive sector. This chapter describes the mechanisms that generate faults on electronic systems by giving a basic theoretical background and explaining the impact of the reducing dimensions and operating voltages combined with the increasing clock frequencies. It then continues by presenting several methods that have been applied in microprocessor based systems in order to avoid or tolerate the occurring faults. Finally, it summarises some important conclusions that are relevant to the work of this thesis.

3.1 Radiation Effects on Integrated Circuits

The first observations of malfunctioning electronics due to radiation were made in the 1950's, during nuclear bomb tests. However, the first report of errors caused by alpha particles was made by May and Woods in 1978 and was published in 1979 [3.1]. Alpha particles are the result of radioactive impurities found mainly in the package materials (mold compound, underfill, solder, etc.) and to a lesser extent in the materials used for the fabrication of the semiconductor device, with uranium and thorium having the highest radioactivities among them. Alpha particles are composed of two neutrons and two protons and are emitted with energies varying from 2 to 9MeV, so they are considered to be low-energy particles. With careful selection and further processing and decontamination of materials used for packaging, this effect can be minimised.
3. The Origin and Mitigation of Faults in Microprocessors

However, these techniques are expensive and considering the fact that they only address one part of the general problem, they are not suitable for commercial chip fabrication.

The second main source of radiation-induced upsets in electronics comes from extraterrestrial cosmic rays that bombard the earth’s surface [3.2]. The galactic cosmic rays that have a debatable origin mainly consist of protons, neutrons, pions and muons with different energies. Cosmic rays also include particles that originate from the sun, with relatively low energies. When they penetrate the Earth’s atmosphere, some are deflected and the remaining flux starts reacting with other particles of the atmosphere (Figure 3.1). The reactions are very complex, involving many short-lived, intermediate, unstable elements and resulting in a further decrease in the particle flux. Thus, the levels of radiation depend heavily on the altitude. A detailed analysis on this subject can be found in [3.3] where in addition it is shown that the radiation that reaches the Earth’s surface also depends on geographical position.

Since pions and muons are short-lived and protons along with electrons are detained in the atmosphere due to electric interactions, neutrons are the majority of particles that reach the earth surface and hit electronic components. Approximately 1% of the cosmic ray’s neutrons reach the Earth’s surface and they present a very wide energy spectrum [3.5]. As mentioned
already their flux depends on altitude (e.g. the incident radiation level is approximately 100 times higher in flying aircraft than it is at sea level) but also on the location (e.g. Latin America has higher radiation levels than North America or Europe).

When charged particles such as alpha particles enter a semiconductor material, they begin losing energy while at the same time ionising the medium (Figure 3.2). As a result this leads to the generation of electron-hole pairs which move due to the electric field inside the transistor. The resulting transient current may cause a drop or increase of a node’s voltage within the device and may change its logical state. The duration and amplitude of this transient phenomenon depends on the particle’s energy, the material, doping and several other factors. These are the so called “soft” errors or “soft” faults since they do not cause a permanent damage on the circuit, but only alter data. “Soft” errors are widely known as Single Event Upsets (SEUs) \(^1\). SEUs may appear in analog, digital and optical electronics, they may have the form of a transient pulse or a bit-flip\(^2\) in memories and they can also cause multiple-bit errors. A special case of SEUs are referred as Single Event Functional Interrupts (SEFIs) and they occur when specific parts of the circuitry are hit that put the device in stand-by, test or other undefined mode. Most semiconductor devices including memories contain such circuitry, that is used for testing after production. SEFIs are resolved by power reset.

Heavy particles such as neutrons or high-energy protons that can reach the Earth’s surface and have same effects with neutrons [3.6], have a different behaviour from alpha particles. When they hit the semiconductor and other chip material they cause nuclear reactions that generate

\(^1\) NASA defines SEUs as “radiation-induced errors in microelectronic circuits caused when charged particles (usually from the radiation belts or from cosmic rays) lose energy by ionizing the medium through which they pass, leaving behind a wake of electron-hole pairs”.

\(^2\) A change in the value of a memory cell from 0 to 1 or opposite.
a number of ionising particles that can subsequently generate SEUs as they travel through the medium [3.7]. High energy neutrons can generate about 10 times as many electron hole pairs as an alpha particle [3.8]. Low energy neutrons on the other hand interact with boron (B) [3.9], an element used extensively as a p-type dopant and also in the formation of dielectric layers. The result of this interaction is the breaking of the boron atom into an excited Li atom and an alpha particle which again may cause ionisation of the medium.

In space applications several other errors can occur due to the increased particle flux. Particles in this case may include heavy ions, protons, neutrons and electrons which are also affected by solar flares. Single EventLatchup (SEL) [3.10] is a category of permanent error that can potentially damage the device. Silicon CMOS circuits may contain parasitic bipolar transistors, which, under normal conditions, cannot be activated. Heavy charged particles can activate such a transistor and create a low resistance path from power to ground. This results in a large current that can overheat and destroy the device. SELs depend strongly on temperature and the CMOS technology (bulk CMOS devices are the most susceptible). A similar phenomenon which can cause destruction of a power MOSFET is called Single Event Burnout (SEB) [3.11]. The long term exposure to radiation for electronics used in space applications results in cumulative damages such as displacement damage dose which is due to the displacement of atoms hit by heavy particles and the total ionising dose due to the accumulation of charge in the material. All the above gradually change the characteristics (threshold, gain etc.) of transistors.

The energy that is deposited in a device when it is hit by a particle is expressed in a quantity known as the Linear Energy Transfer (LET) and is typically measured in units of MeV/mg/cm² (energy per density). As the particle loses energy while travelling through the material, it loses charge as it creates the electron-hole pairs. The charge loss per unit distance depends both on the type of particle and the density of the material. In general, heavy and energetic particles transfer more LET to the material than small particles. The transfer of LET is also higher if the material is dense. The total amount of charge deposited depends on more complex parameters. In order to create a SEU, a particle must deposit enough charge to change the state of the node. This is known as the critical charge \( Q_{\text{crit}} \). The critical charge is not constant and depends on the form of the radiation pulse (magnitude, duration, etc.) and the current state of the circuit.

One important metric that is used in measuring the effects of radiation on electronics is the cross section. The cross section expresses the device’s sensitivity to radiation as it measures the
3. The Origin and Mitigation of Faults in Microprocessors

The number of erroneous events observed on the device for different particle energies. The typical form for the cross section vs energy graph is shown in Figure 3.3. For very low LET no errors are observed. The energy for which errors start to occur is the LET threshold \((LET_{th})\). The curve usually reaches a saturation point after which no significant change is observed in error occurrence. The cross section can be used to measure any type of error: SEU, SEL, single bit-flip, multiple bit-flips, processor failures and it can even be extended to characterise a system, as a metric relating to the behaviour of a group of devices under radiation. The cross section is usually measured in errors per bit per particle per unit density \(((errors/bit)/(particle/cm^2))\) but since errors, bits and particles are unitless, the unit of cross section is \(cm^2\). In many cases in the literature it can be found as \(cm^2/device\) or \(cm^2/bit\). From the cross section it is possible to calculate the respective error rate if the characteristics of the operation environment such as the LET spectrum and the particle flux (particle density per time unit) are known. The most commonly used error rate, especially in terrestrial applications, is the Soft Error Rate (SER) [3.12]. SER measures the SEUs per bit per time unit, but can also be found expressing the SEU induced failures observed on a device per time unit. Typical units to report SER include \(errors/bit/day, errors/device/day, FIT/bit\) and \(FIT/device\).
3.2 EMC for Integrated Circuits

There are two main issues concerning electromagnetic compatibility (EMC) and ICs. The first is the electromagnetic energy emission of ICs while they are operating and the second is the susceptibility of ICs to electromagnetic waves from the operational environment. With the ever increasing use of electronic systems and also the extensive use of wireless applications (WiFi, mobile phones, etc.), the electromagnetic environment becomes more complex, making the EMC requirements of systems more challenging to meet. Research is focused mainly on modelling and measuring electromagnetic emissions in order to use the necessary techniques that minimise them. However, the susceptibility of ICs is also becoming an important topic since new generations of ICs not only cause more disruptions due to their increased complexity, but also become more susceptible due to their lower power supply. There is therefore an increased effort in standardised measurement procedures [3.13]. Detailed information on electromagnetic compatibility of integrated circuits can be found in [3.14].

An IC can be considered as a number of gates that switch state according to the clock frequency. During the switching activity of the gates, instantaneous high currents flow from and to the transistors that form them. This may result in a voltage drop of the $V_{CC}$ or a ground bounce that can affect all the components on the printed circuit and generate parasitic emissions. Electromagnetic interference can also be caused by an external source that is able to generate strong radiated energy that can affect lines on the circuit board and cause errors. Measurements have shown that electromagnetic interference can disrupt the communication between a microcontroller and its external RAM [3.15]. It can also affect the pads of input ports generating static and dynamic failures [3.16]. Crosstalk between the metal lines within the chip is also a significant source of errors, especially in multi-layered chips.

3.3 Electromigration

Over a period of time the flow of electrical current through metal tends to displace metal ions. In some places voids open up in the wires leading to open circuits and in other places ions are deposited causing shorts. In normal circuits this effect is quite small, but becomes important in the long term reliability of semiconductor IC devices where feature sizes are very small. This phenomenon is known as electromigration. The wires within the chip that are more
susceptible to electromigration are power-distribution lines due to the continuous flow of high current density. The effects of electromigration appear after a period of operation and in some cases may initially manifest as intermittent faults. Obviously in most cases electromigration causes permanent damage and the chip needs to be replaced. Specialised tools [3.17] can detect sensitive wires during the IC design stage so that preventative measures can be applied. The electromigration threshold can also be increased with the use of copper interconnects instead of aluminium ones.

3.4 Impact of New CMOS Technologies

Integrated circuit technology continues following steadily the famous Moore’s law [3.18]. Improving lithography processes allow the implementation of the same circuit in an ever decreasing silicon area. Consequently much more complex circuits can be integrated in the same area today than 5 or 10 years ago. This fact increases the need for efficient routing. As a result more metal layers are used to interconnect the various circuit elements. The increased complexity also increases dramatically the number of pins in new chip generation since more I/Os are possible, but also there is an increased need for power supply and ground nodes. Smaller feature sizes allow faster transistor switching, thus higher system clock frequencies. Furthermore, the need for low power consumption has decreased the power supply voltages for both core and I/O logic. Table 3.1 shows the development of IC technology between 1988 and 2004.

<table>
<thead>
<tr>
<th>Year</th>
<th>Technology</th>
<th>Metal Layers</th>
<th>Power Supply</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988</td>
<td>0.7μm</td>
<td>2</td>
<td>5V</td>
</tr>
<tr>
<td>1992</td>
<td>0.5μm</td>
<td>3</td>
<td>3.3V</td>
</tr>
<tr>
<td>1994</td>
<td>0.35μm</td>
<td>5</td>
<td>3.3V</td>
</tr>
<tr>
<td>1996</td>
<td>0.25μm</td>
<td>6</td>
<td>2.5V</td>
</tr>
<tr>
<td>1999</td>
<td>0.18μm</td>
<td>7</td>
<td>1.9V</td>
</tr>
<tr>
<td>2001</td>
<td>0.12μm</td>
<td>8</td>
<td>1.5V</td>
</tr>
<tr>
<td>2004</td>
<td>0.09μm</td>
<td>9</td>
<td>1V</td>
</tr>
</tbody>
</table>

The impact of scaling on the soft error rates of computer systems has been of concern since the 1970’s. However as the trend is towards deep sub-micron technologies, the concern is increasing
The stored charge that is used within digital circuits to represent data has decreased dramatically in recent years as a direct result of the decreased power supply. This means that the critical charge $Q_{crit}$ has also decreased (Figure 3.4) with a negative impact on the error susceptibility to particle hits. Improved manufacturing technology has managed to reduce bit SER of DRAMs. However, new systems require larger amounts of memory, so the overall system SER has not changed substantially (Figure 3.5(a)). In the case of SRAMs the bit SER has not reduced at a similar rate and as can be seen in Figure 3.5(b), the overall system SER has increased significantly.
Measurements have shown that for low clock frequencies, where the transient pulse induced by a particle hit is much shorter than the clock period, the error rate in combinational circuits is linearly dependent on frequency while for sequential circuits it is independent of frequency [3.23]. This fact suggests that as frequency increases, the errors observed will be dominated by transient faults originating from combinational logic rather than SEUs on sequential logic. Some relevant experiments on commercial microprocessors have confirmed the above expectation [3.24]. The increasing frequency will also tend to increase the occurrence of multiple-bit errors, since the duration of the transient pulse may overlap more than one clock edges. Until recently high-reliability systems were mostly concerned with the protection of memory systems. However, it is clear that errors in other parts of the logic such as ALU, core logic (e.g. instruction pipeline) and peripheral logic are becoming a significant proportion of the overall SER. In fact SER of flip-flop and latches is equal to the SER of SRAMs in 90nm technologies with a tendency to increase rapidly in future technologies [3.22].

As the complexity of on-chip circuits continues to increase, higher currents flow through the power supply lines, consequently increasing the susceptibility to electromigration. Furthermore, the increased number of metal layers makes crosstalk between the interconnection lines more probable as the distance between them decreases. The rapid switches in the output of each CMOS gate give rise to very short and sharp current peaks that are the sources of radiated or conducted emission [3.25]. The noise margin also decreases with lower supply voltages. These problems are usually successfully solved with techniques used in the fabrication process, however they may become an important threat to signal integrity in future CMOS technologies and will require further design considerations at the register transfer level.

3.5 Proposed Solutions

The most effective way to eliminate faults on ICs is to minimise the exposure to radiation through some form of shielding. However, this is not a realistic solution for most of the applications which usually have the demands of low cost and high mobility. It also does not eliminate electromagnetic interference problems, although clearly there are steps that can be taken externally to help protect the IC, but these too can be costly and complex to implement. Accepting that faults and errors are inevitable, it is necessary to use mitigation techniques.
There are three levels of mitigation techniques (Figure 3.6). The first, lowest level, involves improvements in the process and technology used. Careful decontamination of the materials, replacement of materials that increase error rates (such as boron based layers) and several different techniques for the CMOS technology structure can reduce SER. However these techniques can offer improvements only to a certain extent, increase significantly the final cost and are limited to special applications. One example is the use of silicon on insulator (SOI) [3.26] technologies.

The second level includes techniques that are applied on the circuit level. In large and complex systems (such as microprocessors) it is difficult to consider transistor-level techniques, so it is more common to work at the RTL. Redundancy, error detection and correction are the most typical methods implemented at this stage. The highest level includes software-based techniques (N-version programming, error correction algorithms etc.) that have been used extensively in the past but are now becoming inadequate since their efficiency depends on the correct operation of the hardware.

All of the mitigation techniques have to face various trade-offs. Lower SER comes usually at the expense of hardware overhead (on and off chip), performance degradation, increased design and fabrication complexity and higher cost. This thesis focuses on techniques of the second level because they can be cost effective and are technology independent. The following sections review the work that has been done on techniques at this level.
3. The Origin and Mitigation of Faults in Microprocessors

3.5.1 Large Scale Redundancy

The aerospace and communication industries have been one of the main driving forces in developing high reliability electronic systems since the early 1960's. These approaches are based on large scale redundancy, with component level replication. Given the requirement of high instantaneous reliability, aerospace systems typically achieve fault masking through massive redundancy. Communication systems on the other hand do not incorporate fault masking techniques and rely basically on fault detection and repair, since short disruptions in the operation are usually acceptable. The following examples are some of the most representative computer systems in this category but several more can be found in the literature. More examples can be found in [3.27].

FFMP (Fault Tolerant MultiProcessor) [3.28] is a computer system that was designed for safety-critical control application, mainly in civil aviation. It is designed to have a failure rate of $10^{-10}$ per hour and routinely operates on ten-hour flights where no airborne maintenance is available. It has the ability to mask errors without the need for program rollback and can also handle multiple faults. FTMP is a fault tolerant multiprocessor system. Compared to a normal multiprocessor system it incorporates a triplicated number of modules (processors, cache memories, memory modules and I/O access units) plus a number of spares, redundant interface buses (for memory and I/O access) and further supportive hardware modules (e.g. bus guardians). This means that the implementation of a FTMP version of a dual processor computer system with two spares has a redundancy factor on the discrete component level of more than $\times 8$. The program execution is considered as a number of "job steps". A triad of each type of module is allocated for each job-step, forming a TMR system. The system is operable as long as the number of failed modules is less than or equal to the number of spares.

A more software-oriented approach is used by SIFT (Software Implemented Fault Tolerance) [3.29] that is again designed for civil aviation systems. The system consists of a large number of redundant main processors that perform all the calculations and a large number of simple I/O processors that control the sensors and actuators of the control system. All processors communicate through high-bandwidth redundant buses. Each individual processor can have access to the other processor's modules. This allows the performance of a majority voting for the correct result and the detection of faulty modules that generate wrong results. Typically a 2-out-of-3 voting is used, but additional modules can be used for more critical tasks. The state
Another typical example of a large-scale redundant fault-tolerant computer is the Airbus A330 / A340 [3.30] primary flight control system. This system consists of three primary computers and two secondary computers. Each computer consists of two independent microprocessors. Hardware diversity is used with different microprocessors for primary and secondary computers and software redundancy is used on the computer level where each processor executes different program versions with the same specification. In addition there are two computers for data concentration and interfacing to displays.

A similar system is presented in [3.31] for a computer that is responsible for flight-critical operations of a space shuttle. Five computers communicate through serial buses with the avionic subsystems. Each computer consists of one central processor and one processor dedicated to input and output operations. Four of the computers are identically programmed to perform the flight-critical operations in a redundant configuration (quad/triplex or duplex operation, depending on the number of computers that have failed) and one computer is responsible for non-flight-critical avionic functions.

In cases such as communication systems, the reliability requirements are typically much lower. Availability is of higher importance than continuous fault-free operation. An example of this category is the electronic switching system (ESS) [3.32] which was designed to be out of service no more than a few minutes per year and can allow 1 in 10,000 calls to be processed incorrectly. For that reason the computer system (with improved versions that have evolved through the years) is based on duplex configurations for both the central processing and the peripheral units.

Several examples of smaller scale component-based redundancy also exist. In [3.33] a TMR system of Z80 microprocessors with shared global memory is presented. A similar 8085 based system can be found in [3.34]. Finally [3.35] presents a redundant microcontroller architecture for embedded real-time control in a spacecraft.

3.5.2 Fault Tolerant Processor Architectures

The first concern in fault tolerant processor systems is the protection of memory and storage elements. These include external memory modules, internal embedded memory (instruction
3. The Origin and Mitigation of Faults in Microprocessors

and data cache, scratchpad RAM, etc.) and register files. These are protected by parity bits or more complex error detecting and correcting (EDAC) codes [3.36]. Lately there has been an increasing concern in protecting other parts of the processor that perform arithmetic, control and logical functions. The techniques used in this case are more varied and depend on the specific architecture. They are considered difficult, time consuming and introduce performance penalties. Some representative examples are presented in the following paragraphs.

The IBM S/390 G5 processor [3.37] is a commercial solution for mission-critical business applications. The instruction decoding and executing elements (I-unit and E-unit) that constitute the pipeline are duplicated and the result of the last stage of the two pipelines are compared to detect possible errors. If an error is detected then the processor re-executes the last successfully executed instruction. This is achieved by recording the state of the system in special buffers and arrays. The registers and cache memories are protected by error correcting codes (ECC). Intel's Itanium processor [3.38] uses ECCs and parity techniques in order to protect all large memory structures. System "hangs" are prevented with the use of a watchdog timer.

Several approaches for fault-tolerance in similar high-performance superscalar processors exist [3.39, 3.40, 3.41, 3.42, 3.43, 3.44]. However, since superscalar processors target applications with high performance requirements, these techniques are based on duplication, mainly on the instruction level by duplicating instructions or by taking advantage of multi-threading capabilities, and roll back mechanisms. The fault-models considered in these cases are usually limited and as a result the actual fault coverage is not optimal.

The AE11 controller [3.45] was specifically designed for automotive applications. It includes built-in self-test (BIST) supporting circuits for preventive checks but also supports concurrent detection methods to detect errors at the time of their manifestation. However, this controller is based on a medium performance 8-bit core and its fault coverage does not satisfy high safety integrity level requirements.

The fault-tolerant version of the LEON microprocessor [3.46] is more relevant to the needs of an automotive application with high integrity level such as drive-by-wire. It was designed for space-flight applications but emphasis was given on performance, availability and low cost. The design is portable so it can be implemented with a wide range of commercial semiconductor

---

3 When a processor enters an unrecoverable execution loop.
4 LEON microprocessor is presented in detail in Chapter 4.
3. The Origin and Mitigation of Faults in Microprocessors

processes and it allows reuse in system-on-a-chip designs. The cache memories of the processor are protected with parity bits for error detection. If a data or tag line is corrupted, a cache miss is forced and the correct data is fetched from the external memory. The register can be protected again with parity bits or a (32,7) BCH checksum [3.36]. The codes are generated in the write stage of the pipelined execution unit and checking is done in the execution stage, after the register data has been read in decode stage. If a correctable error is detected, the erroneous data are corrected and written back to the register file. The pipeline then is flushed and resumes operation from the instruction where the error was detected. If the error is uncorrectable, then a trap is generated. The external memory is protected through an EDAC unit. For the remaining memory elements a TMR configuration at the flip-flop level is used. This strategy assumes that faults on combinational circuits are usually not latched into memory elements, which as pointed before is not valid for contemporary and future CMOS technologies.

An approach to protect the microprocessor control logic is presented in [3.47]. The control logic is divided in two parts. The first is dynamic control logic that depends on the program flow. An example of dynamic control logic is branch prediction. The second is static control logic with signal sequence that is constant for each instruction. Dynamic control logic is duplicated at the component level. When an error is detected, the pipeline repeats the same execution until the transient disappears. For static control, a signature generation mechanism is used. Before committing an instruction, the generated signature is checked from a set of precalculated signatures stored in a table. In case of a detected error, the pipeline re-fetches the failed instruction. This approach does not introduce high hardware overhead, but its complexity and inadequate fault coverage (reported 99% for a specific set of fault that were considered) makes it unsuitable for safety-critical applications. Similarly in [3.48, 3.49, 3.50] error detection through various codes is proposed. Emphasis is given on the recovery mechanism which is based on the use of programmable components (such as FPGAs) that act as a back-up processor with reduced functionality.

Time redundancy can also be used to protect control logic. In [3.51, 3.52] all the flip-flops are duplicated and separate clocks are used for each set. The second clock is shifted by a time $\delta$ so that the second set of flip-flops latches its inputs after the first. If a transient fault causes an error in the first set of flip-flops and lasts shorter than $\delta$ the second set will latch the correct value. A comparator is used to signal a disagreement between the two latches. This method
3. The Origin and Mitigation of Faults in Microprocessors

is technology-dependent as the time $\delta$ must be selected according to the average duration of transient faults which is not easy to determine. Furthermore, its application becomes even more difficult as the system clock frequency increases. Finally, it does not protect combinational circuits.

Recently a theoretical study on possible on-chip redundant processor configurations for safety-critical automotive applications was presented [3.53]. The study examines the possibility and the benefits of using redundant microprocessor structures that are embedded in a single chip. Several different architectures are compared in terms of hardware overhead and expected performance.

3.5.3 Software methods

Software methods can also be used for different levels of error-protection. They can for example be used to implement EDAC coding and decoding [3.54] for the protection of external memories. They can also be used to protect the integrity of both data and code by replicating every instruction and every variable [3.55, 3.56]. Another category of software techniques deals with tolerating program flow deviations [3.57, 3.58, 3.59].

In general, software methods alone cannot achieve the very high fault coverage required for a safety critical system, especially when considering multiple bit errors, increased memory requirements and decreased performance brought about by multiple code executions. Furthermore, the errors can often only be detected long after they occur, thus introducing hazards. They cannot therefore provide fast recovery mechanisms by fault masking.

3.6 Conclusion

The scaling trends in CMOS technology has a negative impact on the reliability of integrated circuits. Errors on core logic in particular are becoming an important percentage of the overall error rates of microprocessors, which renders the use of special measures on the RTL necessary. Various mitigation techniques have been presented over the years. However, the focus on these techniques is in keeping high performance and low hardware and area overhead, since they are usually applied on high-performance and complex microprocessors. The number of fault tolerant microprocessor solutions for safety-critical control applications is minimal and the existing
solutions are inefficient for future applications with very high reliability requirements such as drive-by-wire. There is a need for a highly reliable, high-performance microprocessor for control applications that is designed taking into account the fundamentals of safety-critical system design at all levels. This includes simple techniques that do not introduce any new hazards and where their effectiveness can be easily validated with standard methods, like those described in the previous chapter.

The pipelined execution unit is responsible for the continuous and error-free operation of a microprocessor. Such a vital unit needs to be adequately protected against errors. In the case of a scalar processor, redundancy in that level is not only achievable due to the advanced CMOS technology processes, but also provides extremely efficient protection while keeping the design principle very simple.
References


CHAPTER 4:
A NOVEL PIPELINE-PROTECTED PROCESSOR CORE

This chapter presents a method for protection of the pipeline core of a RISC processor. Previous work done on fault tolerant processors suggested that it is necessary to protect critical parts of the processor such as its pipeline. The application of traditional computer-system redundant techniques that have been used on a discrete component level, was considered since they offer great increase on the reliability and high fault coverage, while keeping the complexity low. A general redundant architecture based on the concept of sif-t-out redundancy is described, and it is followed by a description of a specific implementation.

4.1 A Method for the Protection of a RISC Pipeline

As it has been pointed in the previous chapter, the protection of the processor's control logic along with its internal datapath is essential for a correct and continuous operation. When considering a RISC processor, which is often used in embedded real-time control applications, the "heart" of the system is the pipeline that contains both the datapath and control signals. The general concept of an execution pipeline is that in each clock cycle, each stage takes as inputs the outputs of the previous stage and also signals from shared resources, and through combinatorial circuits it generates a number of outputs that are latched to a set of stage output registers on the following clock cycle. The data on these registers can be inputs for another stage, or for other shared resources. In every clock cycle, each pipeline stage is associated with a specific instruction, in the case of a scalar processor, or a multiplicity of instructions in the case of a superscalar processor. The instructions advance to the next stage on every clock cycle, although in some pipeline implementations specific instructions need to remain in a stage for more than

1 The shared resources may include (but not be limited to) a register file, an instruction cache memory system, a data cache memory system, any other type of external memory system (e.g. scratchpad RAM, different types external RAM or ROM memories etc.), a co-processor unit, a floating-point unit, or an on-chip debug unit.
one cycle. The whole pipeline operation can in general be stalled by external modules for a number of cycles using one or more stall signals, if deemed necessary, for correct execution of the software.

There are several different redundancy techniques that can be chosen to protect the pipeline, each with different advantages and disadvantages. The use of dynamic redundancy (pair of modules) offers low hardware overhead and good fault coverage. However, it cannot be chosen for a processor that targets hard real-time safety-critical applications because faults cannot be masked and external action such as system reset or roll-back is necessary which could lead to operational downtime. It must be noted that recovery time in this case has different importance among the various applications. For example a downtime even of several seconds can be acceptable for a server, but it could possibly have catastrophic results on a fly-by-wire system node.

The next type of redundancy to be considered is static redundancy. In this case three or more modules are needed, something that offers the possibility of immediate masking of faults. Static redundancy and the TMR configuration in particular is a very popular technique for the protection of digital circuits against SEUs [4.1]. There are several hazards associated with this type of redundancy. TMR can be applied on a low-level basis when considering the circuit in terms of interconnected combinational and sequential circuits, or on a higher-level basis when considering a group of combinational and sequential circuits as a module that performs a specific operation (e.g. the execution pipeline as a whole). In the first case the first solution is to triplicate the sequential circuits only. This solution is not satisfactory since the whole circuit needs protection. The other important hazard in this case is that the probability of a common fault in two out of three flip-flops that form a TMR group could be significant, especially in future technologies where feature size is small and a single particle strike could affect two adjacent flip-flops. The application of TMR on a module level with voting on the module’s outputs is simple to implement but in the case of systems with a complex operation such as a pipeline, when one of the modules deviates from its correct sequence, it loses synchronisation with the other two and operates in an erroneous state. Although the redundant system’s output will be correct due to the majority voting, a second fault on one of the remaining modules will produce erroneous results. Another category of hazards associated with TMR configurations deals with the voters. The voter is, as has been explained in an earlier chapter, a single point of failure
since it produces the output. This can be solved by triplicating the voters [4.2]. This solution cannot be adopted on the pipeline since this would require triplication of all the chip’s resources (pipelines, bus, register file etc.) which would increase dramatically the overall overhead. Furthermore, voting on multi-bit signals (also known as word-voting) presents some difficulties and requires more complex voters [4.3] in order to guarantee high reliability.

A hybrid redundancy solution is more appropriate since it can combine simple adaptation of its concept along with minimisation of single points of failure and can also give the possibility of reconfiguring the system when an error is detected. The technique proposed in this thesis is based on the concept of sift-out modular redundancy [4.4] with the addition of recovery mechanisms. The following paragraphs explain the general concept, while a more specific application and implementation is described in the next sections.

In this approach the execution pipeline is replicated at least three times (Figure 4.1). The multiple execution pipelines share all the common resources such as the instruction cache, data cache, register file etc. In addition there is a number of comparator circuits, normally one per pipeline stage (Figure 4.2) but there may be fewer if only the output signals are checked, and one control module that defines the configuration of the system. The comparator checks each common output register between the replicated pipelines. After system reset all pipelines and shared blocks are in a known state. The control module connects the default pipeline (pipeline 0) to the shared resources and execution commences in all pipelines, however, only the currently selected pipeline is allowed to drive the shared resources. If for any reason one or more pipelines produce a faulty output at any of their stages, the control mechanism will detect that through the comparator circuits. If the error(s) is (are) correctable then the control module will take actions to mask it (them). If the error(s) is (are) not correctable then the control module asserts a signal so that external action can take place (e.g. system reset, generation of an interrupt). It is preferable to detect the presence of an error as early as possible and not to wait until it manifests its presence in one of the outputs. This is because by the time the error propagates to the output, a second error may have occurred in a different pipeline, which will then be impossible to correct in case of a three pipeline configuration. Although in simple pipelines (such as in the case of scalar processors) an error on a register will typically propagate to an output in a very small number of clock cycles, there is a significant probability that it will remain dormant for several clock cycles.
4. A Novel Pipeline-Protected Processor Core

Figure 4.1: Top-level diagram of the proposed redundant pipeline system

Figure 4.2: Error-checking on each pipeline stage
Errors are detected by comparing the outputs of each pipeline and if necessary some of their internal signals, on a per-cycle basis. One or more errors are considered as non-correctable, when at the time of their occurrence all the pipelines that participate in the system disagree. This means that when there are only two remaining pipelines, errors occurring on them cannot be recovered, only detected. If an uncorrectable error occurs the system remains fail-silent. This is a very important and a required characteristic for components of a safety-critical system. It must be noted that it is assumed that more than one pipelines can fail at the same instant of time. However, since the pipeline is a complex system both in terms of its RTL description and in terms of its physical layout, it is extremely improbable that a single SEU (or other external factors such as EMC) would cause an identical error in two (or more) pipelines. External resources, including the clock signal, are assumed to have an error-free operation.

When correctable error(s) occur, the control module stalls processing for at least one cycle. During the stalled period it disconnects from the system the disagreeing ("faulty") pipelines. If one of the "faulty" pipelines is the default pipeline (taken initially to be Pipeline 0 in Figure 4.1), then the driving of the shared resources is shifted to the next available, non-"faulty" pipeline. After that reconfiguration period, the system resumes with the remaining pipelines. When non-correctable errors occur, the control module enters the "fault" state where an appropriate signal is asserted to trigger external action. When there are only two remaining non-"faulty" pipelines the system is in a "pair" state. Here it is only possible to detect an error in the two remaining pipelines, which will drive the system to the "fault" state.

When the number of the remaining pipelines is even, there is a possibility that errors occur in such way that there is no majority vote among the outputs of the pipelines, thus it is impossible to define which pipelines disagree. For example in a configuration with four pipelines there may be two pairs of agreeing pipelines, which makes it impossible to define the one that is correct. In that case the system enters the "fault" state. Although this scenario is highly improbable, it has to be considered in order to ensure the high dependability of the system.

When the system is in any state apart from the initial, or a fault state, it updates the contents of the registers of the disconnected pipelines with correct data, from one of the currently active "correct" pipelines, in order to overwrite the occurred error. When this operation is successfully complete, and no other error is detected, the controller reconfigures the system by reintroducing these pipelines. If more than one pipeline are disconnected they are all reintroduced concur-
4. A Novel Pipeline-Protected Processor Core

Figure 4.3: State-transfer mechanism

rently when all of them are in a correct state. The transfer of correct state from a “correct” pipeline to a “faulty” pipeline is managed by the control module and is shown in Figure 4.3. The inputs for every register can be selected between the normal, internally generated signals and the respective signals generated by another pipeline.

The comparator circuits must produce the following information through appropriate signals:

- Whether there is a disagreement among the pipelines.
- Which pipeline(s) disagree.
- Whether each possible pair of pipelines is in agreement.

The control module must produce the appropriate signals in order to:

- Define the pipeline that drives the shared resources.
- Stall the processing when necessary.
- Transfer register data between pipelines.

A significant advantage of this solution is that it allows high levels of reliability whilst keeping the controlling mechanism simple. It must also be noted that transient faults and errors on the controlling parts do not cause a general system failure, but will only lead to unnecessary reconfiguration. The only single point of failure is the output multiplexer, for which it is possible to reason about the safety and also there are known mechanisms for achieving high reliability. These issues are discussed in more detail in the section that describes the implementation of the above concept with a specific processor.
4.2 Choice of Processor

4.2.1 Processors in Safety-Critical Applications

Safety-critical systems typically use well understood scalar processors, where instructions are sequentially executed in their original order. However, in many cases, this generation of processor is incapable of meeting the performance requirements of some modern applications as seen in previous chapters and is outdated, legacy technology [4.5].

New generations of safety-critical systems with increased processor performance requirements can potentially follow two separate paths. The first is to utilise the available modern commercial off-the-shelf (COTS) processors and make use of the widely available COTS software tools. In the case of popular processors the design stage is greatly facilitated by the manufacturer and the existing experience and know-how. However, COTS processors are usually more complex than those needed in safety-critical systems and they are also designed to have good average-case performance in order to be suitable for a wide range of applications. In many cases the use of a processor in a real-time safety-critical system needs a worst case timing analysis to measure how long a software module needs to be executed and to make sure that the maximum value of this time is acceptable by the system’s specification. This analysis can be difficult in superscalar processors that can execute more than one instruction per cycle, thus increasing significantly the processing speed, but having as a result to include wait states when accessing memory modules that are typically much slower. Timing analysis is also difficult with multiple-level cache memories, since cache hits and misses cannot be easily predicted. Another important drawback of COTS processors is the design defects they contain, since their design principles are usually validated with informal arguments. Design bugs have been reported in many cases of commercial processors [4.6, 4.7] and in some cases they have been discovered after doing formal verification of existing cores (e.g. [4.8, 4.9, 4.10]). There are very few examples of commercial processors specifically designed for safety-critical applications that have undergone formal verification through all the design stages. One of these examples is VIPER (Verifiable Integrated Processor for Enhanced Reliability) [4.11], a very simple processor with limited capabilities. Formal verification is currently a large industry challenge since it hugely increases design cost and time.

The second solution is to design and manufacture a purpose-built processor. This solution has
the obvious advantage of tailoring the processor architecture and capabilities according to the needs of the specific safety-critical application. The advances in semiconductor technology and system-on-chip design tools favour such an approach. However a procedure like that would be cost effective only for applications that have high production volume. In that sense the automotive industry is extremely well placed to benefit from purpose-built processors for its new generation of safety-critical computer systems. Another drawback that should be noted is that such a processor would require the design of basic tools such as compilers, debuggers etc, from scratch.

In recent years there have been many important developments in software due to the advent of open-source codes, where developers effectively give away their source code to the wider community. A similar trend has started more recently in hardware design [4.12]. Several open-source IPs are being developed by the community that include a big variety of microprocessors, bus controllers, video controllers, co-processors, DSP cores, arithmetic cores, but also open-source software tools that enable the design of such IPs, since the commercially available ones are very expensive for individuals. Although the use of an open-source IP for a safety-critical application would be in general considered as bad practice, it gives many opportunities for research in the area of safety-related computer systems because unlike commercial IPs it gives the opportunity of having access to the detailed microarchitecture allowing modifications and experimentation. For that reason the work described in this thesis was based on LEON, an open-source microprocessor designed by the European Space Agency (ESA). Two different versions of LEON have been used by ESA to produce fault tolerant microprocessors aimed at critical space applications [4.13, 4.14]. LEON has also been used as the basis for a radiation-hardened processor [4.15] and several other space, military and consumer applications. These applications show that LEON and similar open-source designs are potentially well suited for use in safety-critical systems.

4.2.2 LEON processor

The LEON VHDL model describes a 32-bit processor that implements the SPARC V8 [4.16] (IEEE-1754) architecture. There are three different versions of this distribution: LEON1, LEON2 and LEON3. The main difference between the first two is that LEON2 incorporates an on-chip debug support unit, while LEON3 has an improved pipeline with more stages and
supports on-chip multiprocessor configurations. LEON2 [4.17] was selected for this work, since LEON3 was not available when the design process began. The main features of LEON2 are:

- SPARC V8 compliant integer unit with 5-stage pipeline.
- Hardware multiply, divide and MAC units.
- Interface to the Meiko FPU (Floating Point Unit) and custom co-processors.
- Separate instruction and data cache (Harvard architecture).
- Set-associative caches: 1 - 4 sets, 1 - 64 kbytes/set. Random, LRR (Least Recently Replaced) or LRU (Least Recently Used) replacement.
- Data cache snooping.
- AMBA-2.0 AHB and APB on-chip buses.
- 8/16/32-bits memory controller for external PROM and SRAM.
- 32-bits PC133 SDRAM controller.
- On-chip peripherals such as UARTs, timers, interrupt controller and 16-bit I/O port.
- Advanced on-chip debug support unit and trace buffer.
- Power-down mode.
- On-chip 0-waitstate scratch pad data RAM.
- Interface to high-performance IEEE-754 FPU.
- Low complexity 32-bit PCI target-only interface.
- 10/100 Ethernet MAC.
- SPARC V8 Reference Memory Management Unit (MMU).

The LEON2 synthesisable VHDL model is customisable through several parameters, allowing different configurations for various modules, exclusion of unnecessary parts and addition of custom IP blocks. A top-level block diagram of LEON2 that includes all possible peripherals is shown in Figure 4.4.
LEON2 Integer Unit (IU)

The LEON2 integer unit implements the full SPARC V8 standard, including all multiply and divide instructions. The number of register windows is configurable within the limit of the SPARC standard (2-32) with a default setting of 8. The LEON2 unit uses a single instruction issue pipeline with 5 stages:

**Instruction Fetch (FE)**: If the instruction cache is enabled, the instruction is fetched from the instruction cache. Otherwise the fetch is forwarded to the memory controller. The instruction is valid at the end of this stage and is latched inside the IU.

**Decode (DE)**: The instruction is decoded and the operands are read. Operands may come from the register file or from internal data bypasses. Call and branch target addresses are generated at this stage.

**Execute (EX)**: ALU, logical and shift operations are performed at this stage. The address is generated here for instructions involving memory references.
Memory (ME): The data cache memory is accessed in this stage. For cache reads the data will be valid at the end of this stage, at which point it is aligned as appropriate. Store data calculated in the previous stage is written to the data cache at this time.

Write-Back (WR): The result of any ALU, logical, shift or cache read operation are written back to the register file.

Cache Memories

The LEON2 processor implements a Harvard architecture with separate instruction and data memories connected to two independent cache controllers. Both instruction and data cache controllers can be separately configured to implement a direct-mapped or a multi-set associative cache with configurable set associativity of 2 to 4. The set size is also configurable from 1 to 64 kbyte divided into cache lines, each of 16 to 32 bytes of data. In a multi-set configuration there are three replacement policies available: least recently used (LRU), least recently replaced (LRR also known as FIFO since it replaces the "oldest" line) or (pseudo) random. The VHDL code generates the necessary hardware for the controllers according to the selections made.

On-chip Buses

The LEON2 processor uses two different types of the Advanced Microcontroller Bus Architecture (AMBA) [4.18]. The Advanced High-performance Bus (AHB) is used to connect the processor cache controllers to the memory controller and other optional high-speed units. In the default configuration the processor is the only master on the bus, while the memory controller and an APB bridge are provided as slaves. The Advanced Peripheral Bus (APB) is used to connect all the available peripherals.

4.3 Pipeline-Protected Processor Implementation with LEON2

The implementation of the concept with the LEON2 system-on-chip platform involved some preliminary decisions. The first involved the possible use of hardware diversity, since theoretically it would be possible to have a silt-out-and-repair modular redundancy with different pipelines, in order to mitigate possible design bugs in any of those. However, such a scheme
would increase dramatically the complexity of the control modules. In this case the redundant cores cannot be kept synchronised which means that the comparison of their outputs on a per-cycle basis becomes impossible. The differences in execution times would require some time-based protocol to perform comparisons. Furthermore, it would be necessary to modify the memory system since each pipeline would execute different software and access different variables. Similar problems would have to be faced in the case of N-version programming support. Keeping the control logic as simple as possible was considered essential for this design because it facilitates the generation of a safety case for the system.

The second decision involved the number of duplicate pipelines in the system. Although the general approach in this project regarding the nature of faults occurrence is pessimistic (for example multiple errors have been taken seriously under consideration although the usual practice in relevant research is considering only single errors), the use of three replicas in the system is considered satisfactory. However the implementation can easily be adapted by using additional pipelines if future CMOS technologies prove to be more susceptible to soft errors than the current predictions, or in the case of a very harsh environment such as space.

The implementation details are presented in the following sections:

4.3.1 LEON2 Configuration

The configuration of LEON2 was kept minimal in order to decrease simulation times as much as possible. For this reason the simulation model does not include any floating point unit or co-processor. Furthermore, the special I/O IP such as PCI and Ethernet interfaces have not been included for the same reason. The on-chip debug support unit is also unnecessary for simulation purposes so it was omitted. The instruction unit (pipeline) contains 8 register windows [4.16]. Both instruction and data cache have the same configuration: direct-mapped (1-set associative) with a size of 1 Kbyte. The size of each line is 32 bytes (or 8 words) and they follow a random replacement policy. The memory management unit (MMU) has not been included in the cache system.
4. A Novel Pipeline-Protected Processor Core

There are two types of comparator circuit: one for single-bit signals (bit-comparator) and one for multi-bit signals (word comparator). This happens because words are not checked on a bit-by-bit basis since such a scheme would not detect a disagreement if the same bit on two of the input words had been inverted [4.3]. Both the bit-comparator and the word-comparator produce the following outputs (Figure 4.5):

- **dis**: A signal that indicates whether one or more pipelines produces different outputs from the others.
- **dis\_vector**: A 3-bit vector where each bit corresponds to one pipeline and becomes ‘high’ when the pipeline in question disagrees with the other two.
- **match\_vector**: A 3-bit vector where each bit represents one pipeline pair and becomes ‘high’ when the outputs of the respective pair agree.

The execution pipeline of LEON2 consists of 5 stages. Each stage contains a set of output registers where it stores its results in every clock cycle. There is also another group of special state/status registers that hold system dependent values. Since LEON2 is a scalar processor it was chosen to place comparators only for the pipelines’ output signals and not check internal signals. As a result some latent faults\(^2\) will still exist, as error-injection experiments that are presented in the following chapters show. The percentage of latent faults is significantly high when errors are injected in the special registers. Depending on their exact location and the

\(^2\) A latent fault is an error that occurs in a register and remains within the system without being overwritten or causing a change at the system behaviour.
currently executing program, latent faults may cause a failure in a normal processor but not in the case of this fault-tolerant architecture, since if they propagate to an output they will be detected.

The output signal groups for the selected configuration are:

1. *iei*: Outputs to the instruction cache.
2. *dei*: Outputs to the data cache.

The above signals are presented in more detail in Table 4.1.

During the synthesis of the above model it was observed that some signals from the above list would cause combinational loops. This happens because there is a closed path which starts from the pipeline, passing through the comparators and the control module and then through the general stall signal back into the pipeline logic generating that same signal. In order to overcome this problem these signals are removed from the compare process and are passed through traditional majority voters. As error-injection results reveal, this has no significant implications in the fault-tolerant behaviour of the system, both for single errors (as it would be expected) but also for multiple errors. These signals are marked with an asterisk in Table 4.1.

4.3.3 Control Module

The control module defines the configuration of the architecture and synchronises the possible repair actions when an error is detected. The control module is implemented as a simple 5-state Mealy finite state machine (Figure 4.6). After system reset it starts in state ‘3’ where all three pipelines are operating concurrently and producing output. The outputs of the comparators are checked at every clock cycle. Since in this implementation the actions taken when a disagreement is detected do not depend on the location of the disagreement (outputs to the data cache, system outputs etc.), all of the comparator’s outputs are gated to generate global disagreement detection signals. This is achieved by passing all the `dis` and `dis.vector` signals through OR gates and the `match.vector` signals through an AND gate. When the control module detects a
### Table 4.1: Output signals of the pipeline

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Number of bits</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>iei.rpc</td>
<td>30</td>
<td>Raw address (next address to be latched)</td>
</tr>
<tr>
<td>iei.spe</td>
<td>30</td>
<td>Latched address (fetch stage)</td>
</tr>
<tr>
<td>iei.dpe</td>
<td>30</td>
<td>Latched address (decode stage)</td>
</tr>
<tr>
<td>iei.rbranch</td>
<td>1</td>
<td>Instruction branch (to be latched)</td>
</tr>
<tr>
<td>iei.bbranch</td>
<td>1</td>
<td>Instruction branch (fetch stage)</td>
</tr>
<tr>
<td>iei.nullify</td>
<td>1</td>
<td>Instruction nullify</td>
</tr>
<tr>
<td>iei.su</td>
<td>1</td>
<td>Supervisor mode [4.16]</td>
</tr>
<tr>
<td>iei.flush*</td>
<td>1</td>
<td>Flush instruction cache</td>
</tr>
<tr>
<td>dci.asi</td>
<td>8</td>
<td>ASI (address space identifier) for load/store instructions [4.16]</td>
</tr>
<tr>
<td>dci.maddress</td>
<td>32</td>
<td>Memory stage address</td>
</tr>
<tr>
<td>dci.eaddress*</td>
<td>32</td>
<td>Execute stage address</td>
</tr>
<tr>
<td>dci.edata</td>
<td>32</td>
<td>Execute stage data</td>
</tr>
<tr>
<td>dci.size</td>
<td>2</td>
<td>Size of load/store data (byte, half, word, double)</td>
</tr>
<tr>
<td>dci.signed</td>
<td>1</td>
<td>Sign extension of load data</td>
</tr>
<tr>
<td>dci.enaddr</td>
<td>1</td>
<td>Memory stage address flag</td>
</tr>
<tr>
<td>dci.enaddr*</td>
<td>1</td>
<td>Execute stage address flag</td>
</tr>
<tr>
<td>dci.nullify*</td>
<td>1</td>
<td>Data nullify</td>
</tr>
<tr>
<td>dci.lock</td>
<td>1</td>
<td>Lock data</td>
</tr>
<tr>
<td>dci.rend</td>
<td>1</td>
<td>Read data cache</td>
</tr>
<tr>
<td>dci.write</td>
<td>1</td>
<td>Write data cache</td>
</tr>
<tr>
<td>dci.flush*</td>
<td>1</td>
<td>Flush data cache</td>
</tr>
<tr>
<td>dci.dsuen</td>
<td>1</td>
<td>Debug support enabled</td>
</tr>
<tr>
<td>dci.msu</td>
<td>1</td>
<td>Memory stage supervisor</td>
</tr>
<tr>
<td>dci.eus</td>
<td>1</td>
<td>Execute stage supervisor</td>
</tr>
<tr>
<td>rfi.rd1addr</td>
<td>8</td>
<td>Register read address 1</td>
</tr>
<tr>
<td>rfi.rd2addr</td>
<td>8</td>
<td>Register read address 2</td>
</tr>
<tr>
<td>rfi.wraddr</td>
<td>8</td>
<td>Register write address</td>
</tr>
<tr>
<td>rfi.wrdata</td>
<td>32</td>
<td>Register write data</td>
</tr>
<tr>
<td>rfi.ren1</td>
<td>1</td>
<td>Read enable 1</td>
</tr>
<tr>
<td>rfi.ren2</td>
<td>1</td>
<td>Read enable 2</td>
</tr>
<tr>
<td>rfi.wren*</td>
<td>1</td>
<td>Write enable</td>
</tr>
<tr>
<td>iso.irqvec</td>
<td>4</td>
<td>Interrupt request vector</td>
</tr>
<tr>
<td>iso.error</td>
<td>1</td>
<td>Error mode [4.16]</td>
</tr>
<tr>
<td>iso.intack*</td>
<td>1</td>
<td>Interrupt acknowledge</td>
</tr>
<tr>
<td>iso.ipend</td>
<td>1</td>
<td>Interrupt pending</td>
</tr>
</tbody>
</table>
disagreement in any of the pipelines through these global signals, it de-asserts the \textit{holdn} signal for one cycle in all pipelines which results in a general stall for that period. In the next clock cycle it changes its state to the appropriate pair mode (‘01’, ‘02’ or ‘12’) where the “faulty” pipeline is removed from the system which now operates in a self-checking pair configuration. If the “faulty” pipeline is the default (pipeline 0), the select signal of the output multiplexors is changed so that pipeline 1 drives the outputs. During that state transfer the control module enables the select signal shown in Figure 4.3 in all the disabled pipeline’s registers in order to be updated with the correct data. The connection in the three pipelines is such that pipelines 0 and 1 update their registers from pipeline 2 while pipeline 2 updates its registers from pipeline 1. The system stays in this state for five clock cycles after which, if no other disagreement occurs between the remaining pair, it switches back to state ‘3’. If during self-checking pair configuration another error is detected in the remaining system the control module enters in state ‘fault’ where it asserts a signal that can be used to trigger external action, such as a system reset or a trap [4.16]. The control module can enter the ‘fault’ state straight from state ‘3’ if there is disagreement in more than one pipelines.

\textsuperscript{3} \textit{holdn} has negative logic.
The control module input and output signals are:

- \textit{rst}: Global system reset (input).
- \textit{clk}: System clock (input).
- \textit{disagr}: Global disagreement signal (input).
- \textit{dis.vector}: Global disagreement 3-bit vector (input).
- \textit{match.vector}: Global match 3-bit vector (input).
- \textit{holdn}: Pipeline stall signal from cache memories (input).
- \textit{tr.holdn}: Output pipeline stall signal.
- \textit{trmux.sel}: Output mux select signal.
- \textit{tr.tr.sel}: Output mux select signal for state transfer.
- \textit{fault}: Output signal to trigger external action.

Depending on the synthesis tool, the state machine can be implemented in several different ways. The two most popular are \textit{adjacent encoding} where the number of flip-flops used are the minimum required, and \textit{one-hot encoding} for which the flip-flop used are equal with the number of states. The latter of the two approaches generates faster state machines. Since the state machine required has five states it is clear that there will be a number of unused states, 3 for adjacent encoding and 27 for one-hot encoding. Good VHDL coding practice combined with special features of synthesis tools (e.g. [4.19]) can generate safe state machines that will switch back to a known state if they reach an unused state due to some external factor, while having predefined outputs in all unused states. This strategy guarantees 100\% fault free operation if we assume faults and errors on the state machine only.

In this implementation the control module has also a 3-bit counter in order to count the cycles that the processor remains in the self-checking pair mode. SEUs in this counter have no safety implication even if they cause an early reintroduction of the disabled pipeline because the state transfer can be achieved in a single clock cycle.
The system as a whole can tolerate errors in the controlling parts (comparators and control module) and can maintain a fault-free operation in most cases. Transients on the comparators' logic or an erroneous switch to a different (used or unused) state can potentially generate unnecessary reconfiguration actions or enter in 'fault' state. This may have minor safety implications as the system will temporarily operate at a reduced safety margin (in pair mode), but will not result in a system failure. There exist scenarios of multiple errors on both the control logic and the pipelines which can lead to a system failure, but these require very specific timing and locations for the errors which makes their probability of occurrence practically zero. Permanent faults would have more serious implications in many cases. They could result in continuous reconfiguration actions which again reduce the level of safety and also have an impact on performance due to the continuous short interruptions that are caused on the program execution. In some cases, permanent (stuck-at) faults could also make the system inoperable (e.g. if the system remains permanently in the 'fault' state) or even vitiate the error-masking capabilities of the system (e.g. a stuck-at-0 fault on a comparator's 'dis' output). However, permanent faults as discussed in earlier chapters are very rare in contemporary technologies, specially for terrestrial applications. Furthermore, a component replacement policy is expected for systems with very high integrity levels and long operation times.

4.3.4 Output Multiplexers

The output multiplexers generate the system's outputs. As it is shown in Figure 4.1 the multiplexers are single-points of failure which means that a fault within them may result in an erroneous output that can be the cause of a processor's failure. The multiplexers are combinational circuits with very small on-chip area, which means that the risk of such a transient fault is very small. However it is possible to protect these circuits with fault-detection mechanisms. The simplest and at the same time effective method for this type of circuits is the use of one or more parity bits [4.20].

4.3.5 Improvements

During the error-injection experiments which are reported in detail in subsequent chapters, it was observed that the behaviour of the system was not completely in accordance with the initial concept. In particular when injecting single errors during the execution of a program the system
should be able to mask the error successfully and maintain a fault-free operation. However, there was a small percentage of single error-injection simulations (around 0.16% in the benchmark program with the worse behaviour among the ones used) where wrong results were produced without being detected. This behaviour was also observed to a higher extent when multiple errors were injected. The vast majority of these cases resulted from injecting errors into the fetch stage and also to a smaller extent when errors were injected in the decode stage. The problem was caused by errors that resulted an erroneous instruction cache address generation, and specifically for bit errors on the least significant bits (LSB) of the address. This happens because when the (LSB) is changed, the resulting address is in close proximity to the correct address. Consequently there is a high likelihood that the erroneous address is already in the cache and will not cause a cache miss. As it can be seen in Figure 4.7, during cycle 2 the erroneous address (shown shaded) appears and causes the tr.holdn to de-assert during the same cycle. At cycle 3 the system has changed its configuration and resumes normal operation. If this error does not result in a cache miss the (erroneous) data caused by the erroneous address is available in cycle 3 because the cache memory is synchronous.

Although it can be argued that this problem does not require any corrective action since according to the error-injection results it causes a system failure only in a minority of cases, there are two simple solutions.

The first possible solution is to keep the tr.holdn signal de-asserted for two cycles instead of one. This solution requires an extra flip-flop within the control module and also increases the average delay time for fault masking.

\[4\] In both cases these errors were injected into the default pipeline.

\[5\] See ALARP principle in Chapter 2.
The second solution is the use of extra majority voters on the cache address outputs of the pipelines. This solution does not add any delay but it has a higher gate overhead compared to the previous one.

Another problem observed during error-injection simulations was the propagation of some errors into the register file. This was mainly observed by checking the contents of the register file at the end of the simulation and comparing them to the contents obtained after an error-free simulation. In some cases that varied between 0.03% of the single error-injection simulations and 0.4% of the multiple error-injection simulations, the register file contents were found different. This is also in disagreement with the general expectations for the system. Figure 4.8 shows an example of how this problem occurs. At clock cycle 2 we assume that the data to be written to one of the registers in the register file by the default pipeline is erroneous. The disagreement is detected and the system switches state according to the specified process. However, as it can be seen the erroneous data will be written to the register file on the rising edge of cycle 3. A similar problem will occur if the register address is corrupted instead. The solution to this problem is to simply certify the validity of the write signal with the use of the stall signal generated by the control module when a disagreement is detected. This can be done by gating the \texttt{wren} and \texttt{tr\_holdn} signals through an AND gate.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.8}
\caption{Timing diagram of an erroneous access to the register file}
\end{figure}

Results from error-injection experiments with and without the above improvements are presented and discussed in detail in Chapter 6.
4.3.6 Reliability Analysis

In order to estimate the reliability of the final system, a 3-state Markov model is considered (Figure 4.9). The system consists of the following states:

1. **N**: This is the normal initial state where the processor pipeline core operates continuously and correctly. Any errors occurring within the pipeline(s) are either overwritten before being detected by any mechanism, or they are detected and successfully masked in the case of the fault-tolerant system following the transitions shown in Figure 4.6.

2. **F**: This state represents a system failure. The microprocessor is considered to be in this state when one or more errors within the pipeline(s) result in the generation of an incorrect output, or the loss of sequence (e.g. entering an infinite loop).

3. **D**: The microprocessor enters this state when one or more errors within the pipeline(s) are detected but cannot be masked thus initiating a recovery mechanism (e.g. a trap).

The transition probabilities are derived from the following rates:

- **f**: The occurrence rate for errors that appear in the pipeline(s) and cause the processor to enter in state F.
- **d**: The occurrence rate for errors that appear in the pipeline(s) and cause the processor to enter in state D.
- **r**: The repair rate of the system. This can be calculated from the duration of the recovery mechanism.

The transition matrix for this model is:

\[
P_t = \begin{pmatrix}
-(f + d) & f & d \\
f & 0 & 0 \\
d & 0 & -r
\end{pmatrix}
\]

By calculating the following:

\[
\frac{d}{dt} P(t) = P_t \times P(t)
\]
we get a set of differential equations:

\[
\frac{dP_N(t)}{dt} = -(f + d)P_N(t) + rP_D(t) \\
\frac{dP_F(t)}{dt} = fP_N(t) \\
\frac{dP_D(t)}{dt} = dP_N(t) - rP_D(t)
\]

with initial conditions \(P_N(0) = 1, P_F(0) = 0\) and \(P_D(0) = 0\), where \(P_N(t), P_F(t)\) and \(P_D(t)\) are the probabilities of the system being in states N, F or D respectively at time \(t\).

By solving the above equations we get:

\[
P_N(t) = \frac{1}{2K} [d + f + k - r + (d - f + K + r)e^{Kt}]e^{-\frac{1}{2}(d+f+K+r)t}
\]
\[
P_F(t) = 1 - \frac{1}{2K}[-d + f + k - r + (d - f + K + r)e^{Kt}]e^{-\frac{1}{2}(d+f+K+r)t}
\]
\[
P_D(t) = \frac{d}{K}(e^{Kt} - 1)e^{-\frac{1}{2}(d+f+K+r)t}
\]

where

\[
K = \sqrt{(d + f + r)^2 - 4fr}
\]

The unreliability of the system is equivalent to the probability of being in state F. Thus, the reliability of the system is:
4. A Novel Pipeline-Protected Processor Core

\[ R(t) = 1 - P_F(t) \]

In order to present a graphical representation of the reliability, an example with two microprocessor configurations is presented. The first is a fault-tolerant configuration with a triplicated pipeline according to the methodology described in the previous sections and the second is a simple configuration with one pipeline. By combining the results obtained by error injection experiments with the following assumptions, specific values for the rates \( f \), \( d \) and \( r \) can be derived:

1. The time intervals \( \Delta t \) between consecutive observations of the system's state are large (e.g. one month).

2. It is assumed that the microprocessors operate in such an environment that one single or multiple error is observed within the pipeline(s) per month.

3. Once the system gets into state F it remains there indefinitely.

4. The recovery mechanism is a software routine that restores the operation of a system (e.g. through reset or through a roll-back algorithm). It is assumed that this mechanism does not fail and always brings the system back to its initial state. The duration of this operation can last no more than a few seconds.

From the second assumption, the rates \( f \) and \( d \) can be calculated by multiplying the pipeline error rate with the probability that this error causes the microprocessor to fail, or is detected respectively. The probability that a single error is detected \( p_{sd} \), the probability that a single error causes a failure \( p_{sf} \), the probability that a double error is detected \( p_{dd} \) and the probability that a double error causes a failure \( p_{df} \) are derived from the error injection experiments.

Considering single errors only, we get:

\[
\begin{align*}
  f_s & = p_{sf} \times \frac{error}{month} \\
  d_s & = p_{sd} \times \frac{error}{month}
\end{align*}
\]
Considering that an error can be either single or double with equal probability, we get:

\[
    f_{sd} = (0.5p_{sf} + 0.5p_{df}) \times \frac{\text{error}}{\text{month}}
\]

\[
    d_{sd} = (0.5p_{sd} + 0.5p_{dd}) \times \frac{\text{error}}{\text{month}}
\]

Table 4.2 presents the values obtained by using the results from error injection experiments performed on the two systems while executing a 4 × 4 matrix multiplication program. The time interval between the double errors ranges from 0 to 5 clock cycles and follows a normal distribution \(^6\).

\[
\begin{array}{|c|c|c|}
\hline
\text{Table 4.2: Transition rates values} \\
\text{Simple} & \text{Fault-tolerant} \\
\hline
f_{s(\text{error month})} & 0.133052 & 0.000759 \\
\hline
d_{s(\text{error month})} & 0.04241 & 0.000738 \\
\hline
f_{sd(\text{error month})} & 0.17356 & 0.0017785 \\
\hline
d_{sd(\text{error month})} & 0.0605435 & 0.0451725 \\
\hline
\end{array}
\]

According to the first and the fourth assumptions the duration of the recovery mechanism is negligible compared to the time \(\Delta t\). This means that if the system enters in state D, it returns back to state N with a probability of 1. Thus it can be assumed that \(r\Delta t = 1\). The detailed modelling of the transitions between states N and D is not important in estimating the system’s reliability, since both are considered safe.

Figure 4.10 shows the reliability graphs of the fault-tolerant and the simple configuration over a period of 24 months, based on the previous assumptions. It can be observed that under harsh conditions with one error per month, a single-core configuration has very low reliability, which drops below 50% after only 5 months when considering single errors only (Figure 4.10(a)) and after 4 months in the case of single and double errors (Figure 4.10(b)). On the other hand, the redundant configuration maintains a high reliability well above 95% during the same period, both for single and double errors.

Some interesting observations can be made when comparing the above reliability estimations with the reliability required from the IEC61508 standard for the various system integrity lev-

\(^6\) For more details refer to Chapter 6.
It must be noted that these requirements refer to a complete system and not sub-systems. However, they are used in this example to highlight the high reliability properties of the proposed architecture. Considering this time single errors only, that have a rate of one per year, the graphs of Figure 4.11 are obtained. As it can be seen in Figure 4.11(a), under such conditions a simple processor configuration is not suitable for any type of safety critical applications, since its reliability is below the limit that IEC61508 specifies for SIL 1 systems. The reliability of the redundant system is within the limits for SIL 3 systems and can be increased by using one of the improvements suggested earlier.

The overall processor reliability is defined by the reliability of each of its sub-systems (pipeline core, memories, buses, I/Os etc.) which means that this will generally be decreased compared to the graphs shown in Figure 4.11, since it is unrealistic to assume reliability equal to 1 in any such system. It thus cannot be claimed that the on-chip fault tolerant architecture that is being presented here can replace similar systems based on discrete components. However, it is highly suitable as a super-reliable component of a larger fault tolerant architecture, based either on component-level redundancy or on distributed systems.

It must be noted that the data that has been used for the fault-tolerant model, are taken from simulations of a system that do not incorporate any of the improvements discussed in the previous section.

Figure 4.10: System reliability: (a) single errors (b) single and double errors
4. A Novel Pipeline-Protected Processor Core

4.3.7 VLSI Implementation

Both the default, single pipeline configuration as well as the redundant configuration were implemented on a high performance, 8-copper layer silicon process from UMC. Both designs were read into Synopsys Design Compiler and an optimized logical netlist was obtained. The netlist was subsequently read into Synopsys Physical Compiler where the instruction cache, data cache and register file RAMs were placed and fixed into place in a Minimum-Physical-Constraints (MPC) flow. The optimized, placed netlist was finally read into Cadence SoC Encounter where power planning and detailed routing took place. Fig. 4.12 depict the MPC floorplan and the final routed design of the redundant configuration and Table 4.3 shows the statistics of both VLSI macrocells.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Single configuration</th>
<th>Redundant configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instances (Macros)</td>
<td>29132 (18)</td>
<td>71718 (18)</td>
</tr>
<tr>
<td>Area ($\mu m^2$)</td>
<td>3586685</td>
<td>4539518</td>
</tr>
<tr>
<td>Core utilisation</td>
<td>64.2%</td>
<td>67.4%</td>
</tr>
<tr>
<td>Clock Frequency (MHz)</td>
<td>229.3</td>
<td>174.8</td>
</tr>
</tbody>
</table>

As it can be observed in Table 4.3, the fault-tolerant configuration introduces an area overhead of around 26.6% and also a performance penalty in the maximum clock frequency compared to
the non-fault-tolerant configuration, for the chosen technology. These are expected trade-offs in the design of fault-tolerant systems and their impact must be considered according to the specifications of the target application. In terms of performance, the clock frequency is much higher than the one of typical high-performance processors used in contemporary automotive systems.

4.4 Conclusion

This chapter has presented a study of the applicability of classic redundant structures on internal on-chip modules, and in particular on the pipelined execution unit of a scalar RISC processor. Such redundant structures offer very high reliability with the expense of increased hardware overhead. Increased reliability of microprocessor components is essential for safety-critical applications in harsh radiation environments, but also for terrestrial safety-critical applications that will use components of near-future sub-micron CMOS technologies. On-chip redundancy proves to be very efficient in masking SEU induced errors without introducing delays and offers great flexibility in the deployment of repair and recovery mechanisms. Another great advantage is the simplicity of the concept which allows it to be applied in many different architectures, with very few implementation specific changes. The use of fault tolerant microprocessors with redundant pipeline cores can be beneficial for future safety-critical commercial applications.
such as drive-by-wire because apart from the obvious increase in safety, it can significantly ease the public acceptance of such systems and it is compatible with the ethics of safety-critical engineering [4.21].
References


CHAPTER 5:
PROCESSOR EVALUATION WITH FAULT INJECTION TECHNIQUES

Fault (or error)\(^1\) injection mechanisms are very important in the development of fault tolerant systems. They are mainly used to assess the effectiveness of fault and error detection mechanisms and to help predict the system’s error rate. They are also useful during the design stages since they can help in the detection of design errors, or in the identification of sensitive parts of an architecture that require the application of further protection techniques.

Fault injection is the deliberate change of the state of an element (wire, bit, byte in memory etc.) within a computer system. Fault injection systems must always incorporate a mechanism that monitors the behaviour of the computer system after the fault is injected in order to record its effects.

This chapter reviews the most popular methods for fault injection at different levels: software, simulation and physical. It then describes two simulation-based fault injection methods which are suitable for evaluating the pipeline-protected architecture. Finally it presents an approach for embedding a fault injection mechanism into a specific microprocessor architecture for hardware-based fault injection.

5.1 Previous Work

Fault injection is a relatively new area of research. There is no global approach to it since the fault models and the ways of injecting them are very application dependent. The following

\(^1\) According to theory SEU induced data corruption should be referred to as error since it is the result of transient faults. However, since it is still referred to as fault in most cases in the literature this term will be used in this and the following chapters to avoid confusion.
sections present examples of fault injection tools that target the evaluation of microprocessor
based systems.

5.1.1 Software Implemented Fault Injection (SWIFI)

An early example of a software tool used as a means of injecting faults into a microprocessor
system is FIAT [5.1]. FIAT allows the corruption of the memory image of a program by extract­
ing addresses from information provided by the compiler and the loader. The EFA tool [5.2]
focuses on injecting faults into the communication subsystem of a distributed system through
software. DOCTOR [5.3] is a flexible software-based tool that can inject microprocessor, mem­
ory and communication faults. The types of faults can be permanent, transient or intermittent.
However this tool is specific to a particular real-time distributed system. FERRARI [5.4] uses
special UNIX functions to corrupt the memory image of the program. The program under test
includes a software trap at the instruction to be corrupted. When the trap is reached, the tool
process changes the memory image and then the program resumes execution. A similar tool
based on the UNIX operating system is FINE [5.5] and later DEFINE [5.6]. These introduce
the possibility of injecting faults on the bus and on the CPU. Xception [5.7] uses a kernel mod­
ule that is statically linked to the kernel of the target system. This module provides the exception
handlers for the target application and the code that performs the fault injection. Fault injection
is possible on memory locations, on the bus, on the instruction unit (IU), on the floating-point
unit (FPU) and on the memory management unit (MMU). In [5.8], various memory locations
of a processor are corrupted by an interrupt handler. The interrupt is controlled externally. This
approach is referred as Code Emulating Upset (CEU).

The use of SWIFI tools has been very popular mainly because they are easy to implement and
adapt to a target system. They are also cost-effective since they do not require extra hardware.
Furthermore they are usually fast since they do not introduce significant delay in the execution
of the target applications. Although built around a specific system, their principles can also be
transferred relatively easily to other systems, making them quite portable. However, there are
many important limitations. The most important limitation in the context of this thesis, is the
fact that SWIFI cannot inject faults in non-programmer-visible locations. There are a significant
number of flip-flops, registers and signals that cannot be accessed through the instruction set,
but are equally sensitive to SEUs. Since the faults are injected through software which executes
5. Processor Evaluation with Fault Injection Techniques

on the target system, the results obtained are not realistic since the workload usually affects the fault-handling mechanisms. Furthermore, the time resolution in SWIFI is quite coarse. For example, it is not possible to inject a fault during the execution of an instruction, only between instructions.

5.1.2 Simulation Based Fault Injection

The extensive use of hardware description languages (HDLs) in the design of digital systems has enabled the use of simulation-based fault injection. Although this approach is time consuming since simulation time is considerably longer than real-time execution, it is favoured since it allows the testing of fault tolerant systems very early in the design stage. If a VHDL or Verilog description of the system is available, testing through simulation can be performed in great detail and it is potentially very accurate since it gives realistic emulation of faults and detailed monitoring of their consequences on the system. The HDL languages and the existing simulation environments provide a variety of tools in order to perform the fault injection and record the results. Most of the simulation based approaches use HDL, however there are some cases where other languages like C or C++ are used.

The FOCUS [5.9] tool allows fault sensitivity analysis of VLSI designs through simulation. To achieve that it needs a netlist of the hardware description of the system and converts it into a simulation model which is then used with the SPLICE simulation engine. DEPEND [5.10] is an integrated design and fault injection environment. It provides facilities to model fault tolerant architectures and perform fault injection studies. It uses a library of elementary and complex objects. Elementary objects provide basic functions for fault injection and data analysis, while complex objects simulate components found in computer systems such as CPUs and memories. The user writes the control program in C++ using the above libraries and uses it as an input to the simulation tool. MEFISTO [5.11] is a well known tool for fault injection into VHDL models. It uses some characteristics of the VHDL language in order to include in the design two types of components called saboteurs and mutants. A saboteur is a VHDL component that alters the value or timing characteristics of one or several signals when activated. A mutant is a specially modified component description. When inactive, it behaves as the original component and when activated it imitates the component's behaviour in the presence of faults. In order to control saboteurs and mutants it uses simulator specific commands to alter the control signals.
A method of using custom VHDL libraries to support fault injection in the VHDL description of a microprocessor is described in [5.12]. In [5.13] the concepts of MEFISTO are used and fault injection in VHDL models is performed with the use of sets of macros based on the command set of the simulator. The most important drawback of simulation based techniques is the long execution time of the fault injection campaigns. A solution is proposed in [5.14]. The system is simulated in a “golden run” where no faults are injected. During the “golden run”, various snapshots of the system’s state are taken. These snapshots are used in the fault injection campaigns as initial points of each simulation. The snapshot selected is the last available before the selected time for the injection of the fault.

5.1.3 Physical Level Fault Injection

Another popular method for injecting faults is the injection of physical faults on the actual target system hardware. This can be achieved through pin-level fault injection [5.15, 5.16, 5.17, 5.18], heavy-ion radiation [5.19, 5.20], electromagnetic interference [5.21] and laser fault injection [5.22]. The major advantage of these approaches is that the environment is realistic (although much harsher than the real world) and the results obtained can give accurate information on the behaviour of the system under such conditions. However, they require special hardware and instruments which are usually very expensive. Furthermore these experiments are complex to setup and control, and the internal signals can only be monitored in real-time if they are connected off the chip. The method is therefore much less useful as a diagnostic aid when developing a new architecture. It is also possible to permanently damage the system under test. These methods are therefore normally used only at the final stages of a design when a prototype is available, or even after production in order to measure the fault tolerant capabilities of the system.

5.1.4 Hybrid Fault Injection

Several approaches have tried to mix the advantages that each of the above categories of fault injection has to offer. Extra hardware has been used to facilitate the fault injection process [5.23] or the monitoring of the fault’s propagation [5.24]. In [5.25] a mix of SWIFI and simulation methods were used in order to accelerate the process of fault injection. GOOFI [5.26] is a tool that supports software implemented fault injection and scan-chain implemented fault injection.
through built-in test logic. A similar approach is reported in [5.27, 5.28] where speeding up of the fault injection campaign is achieved by embedding synthesisable logic in the VHDL descriptions. This aids the fault injection and monitoring process by implementing the whole design in a FPGA.

5.2 The Use of Foreign Language Interface (FLI)

In the early years of FPGA design, devices had a very small number of gates compared to today’s standards. The digital circuits that were implemented on FPGAs were small enough to be modelled, simulated and verified at the gate level. As the density of the devices increased, schematic design entry became inadequate for large complex designs and hardware description languages, such as VHDL [5.29] and Verilog [5.30] were introduced. Nowadays with the use of very high density FPGAs and applications specific integrated circuits (ASIC), design validation has become an important issue. These designs usually require advanced verification algorithms. Some of those algorithms, even if they can be implemented in VHDL or Verilog, are not simulated efficiently in the HDL environment. In order to address this problem, modern simulators enable the interface with routines written in traditional programming languages such as C or C++. Typical applications of such an interface include encoding functions without native support in HDLs (e.g. trigonometric functions in Verilog), accessing functions of the operating system or accessing hardware devices (logic analyzers, data collection units, etc.).

Even in early versions, VHDL provided open access to programming language routines via foreign architectures and subprograms. This approach enables a very efficient connection between the simulator and user-written routines, but since it has not been standardised it depends on the simulator’s application program interface (API). This means that C code is not 100% portable between different simulator platforms and typically needs modification in order to be transferred to different simulators. The Verilog standard on the other hand, contains a description of the C language procedural interface, better known as programming language interface (PLI). PLI can therefore be treated as a standardised simulator API for routines written in C or C++. Most recent extensions to PLI are known as Verilog procedural interface (VPI). A solution enabling a standardised and similar interface between VHDL and C/C++ is in the final stage of development and is called VHPI (VHDL Procedural Interface) [5.31].
The use of the foreign language interface (FLI) [5.32] for VHDL provided by ModelSim [5.33] for fault injection simulation has been used extensively in the work in this thesis. FLI provides a wide variety of functions that are suitable for injecting faults in VHDL descriptions and for monitoring the fault injection campaign, thus forming a powerful and flexible fault injection tool.

FLI in VHDL is performed with the use of a foreign architecture. A foreign architecture is a VHDL entity that is instantiated in a design but does not (generally) contain any VHDL code. Instead it contains a link to a C model that communicates to the rest of the design through the ports of the foreign architecture. This C model behaves exactly like a VHDL process by reading and driving signal values, but can also take advantage of the power of C in order, for example, to ease the reading and writing of files, or communicate with other system processes.

The link to the C code is done through the FOREIGN attribute that is defined by the VHDL standard. The FOREIGN attribute is a string containing three parts and can be declared inside a package or the VHDL architecture itself as:

ATTRIBUTE foreign : string;
ATTRIBUTE foreign OF arch_name :
ARCHITECTURE IS "app_init app.so; parameter";

where,

app_init is the name of the initialisation function within the C code.
app.so is the path to the object file to be loaded.
parameter is an optional string parameter that can be passed to the initialisation function.

The C code is compiled and linked using platform specific compilers and linkers (e.g. Microsoft Visual C/C++ compiler for Windows based operating systems, gcc or cc for Linux based operating systems).

The entry point to the foreign C model is the initialisation function. This function typically allocates memory to hold variables for the instance, registers a callback function to free the memory when the simulator is restarted, saves the handles to the signals in the port list, creates
drivers on the ports that will be driven, creates one or more processes (a C function that can be
called when a signal changes, thus behaving as a VHDL process) and sensitises each process to
a list of signals. The initialisation function is declared as:

```c
app_init( mtiRegionIdT region, char *param, mtiInterfaceListT *generics, mtiInterfaceListT *ports )
```

The function is called during the initial phase of the simulation (elaboration) when the entire
design is loaded and all of its values are initialised. The first parameter is a region ID that can
be used to determine the location of this instance inside the design. The second parameter is
the last part of the string in the foreign attribute. The third parameter is a linked list of the
generic values for this instance. The list will be NULL if there are no generic parameters. The
last parameter is a linked list of the ports for this instance. These parameters are all passed
automatically through the FOREIGN attribute.

A fault injection module based on FLI can be implemented through a VHDL entity placed in
the top level file of the design under test. This entity has no input and output ports and is used
only to call the fault injection C functions. An example code for the fault injection initialization
function is:

```c
void fault_injector(
    mtiRegionIdT region,
    mtiInterfaceListT *generics,
    mtiInterfaceListT *ports,
    char *param
)

{
    mtiSignalIdT ip = CLOCK_ID;
    mtiProcessIdT proc;

    mti_AddLoadDoneCB( init, 0 );
    mti_AddSimStatusCB( fi_ended, 0 );
```
5. Processor Evaluation with Fault Injection Techniques

/* Create a process that injects the faults */
proc = mti_CreateProcess("i_f", inject_faults, ip);
mti_Sensitize(proc, ip, MTI_EVENT);

/* Create a process that monitors the fault injection campaign */
proc = mti_CreateProcess("monitor", monitor, ip);
mti_Sensitize(proc, ip, MTI_EVENT);
}

mti_AddLoadDoneCB is a library function that acts as a pointer to a user defined C function that is called whenever the elaboration phase of the simulation is finished. In this case the function is called init and is responsible for initialising global variables and calculating the parameters of the fault(s) to be injected during the simulation (type of fault, timing, location, etc.). These parameters are assigned to global variables in order to be accessible from the functions that are responsible for injecting the faults. Similarly, mti_AddSimStatusCB is a library function that acts as a pointer to a user defined C function that is called when the run status of the simulator changes. In this case the fi_ended function is called when the simulation finishes. It is responsible for gathering the data from the fault injection simulation and generating a report that is written to a file.

The next step is to generate the processes that are executed in parallel with the simulation as part of the VHDL design. Processes are created with the mti_CreateProcess library function. The inputs to this function are: the name that the created process will have within the VHDL structure, a pointer to the C function that includes the code of the created process and a signal ID (or a list of signal IDs) that will be inputs to the process (in this case the clock signal). The latter is optional but is used in order to have easy access to the signal IDs from within the C function. The next step is to define when the process will be called, through the function mti_Sensitize. In this case the processes are called whenever there is a change of value in the clock signal. Two processes are created in the above code, one responsible for injecting the faults (inject_faults) and one for monitoring the simulation (monitor). The monitoring depends on the design under test. In the case of a microprocessor for example, the monitoring process can gather information about the processor status during the fault injection simulation,
stop the simulation if the processor “hangs”, check the correctness of the results generated and record the number of clock cycles for the program execution in order to calculate the delay compared to a fault-free execution. Fault injection and monitoring can be performed through more than one independent processes and/or C functions as it will be seen in the next section.

All the library functions and support for type conversion between VHDL and C data types are included in the mt_i.h header file. A large number of functions are provided to assist the manipulation of VHDL regions, processes, signals, variables and types. There are also a number of simulator related functions for memory management, checkpoint and restore operations, time and event management, communication and command manipulation.

5.3 Mixed Use of VHDL and FLI for Fault Injection

Although the use of foreign language interface allows the development of a very flexible and powerful fault-injection tool, it cannot be used as a basis for developing a fault-injection strategy for a hardware-based system. FLI code cannot be synthesised and transferred into FPGA prototypes. Furthermore, the approaches for developing fault-injection algorithms in C cannot be easily mapped into hardware. A mixed FLI and VHDL system can be used instead, with VHDL modules that are based on the concept of saboteurs. This approach can be seen as an intermediate step between pure simulation-based fault injection and hardware-based fault injection. When introducing hardware-like fault injection, the methods and general strategy followed are more dependent to the design under test. The advantage of developing VHDL-based (and generally hardware-based) fault injection is that it gives better understanding of the possible fault modes of the design and helps to create better and more accurate fault models for it, especially in the case of very complex designs such as a microprocessor.

The work in this thesis has focused on the protection of the pipeline core of a RISC processor as described in the previous chapter. In order to examine the effects that faults in the pipeline have on the operation of the microprocessor, a mixed FLI and VHDL platform has been implemented. The general structure of the system is shown in Figure 5.1. The steps followed during a fault injection simulation are presented in detail in the following paragraphs.

When the design is loaded from the simulator, during the elaboration stage, the faults to be injected are selected by an FLI function. The function selects a number of faults from an existing
fault database and writes them in a file after sorting them according to their time of injection. An index that keeps the last selected fault from the database is also kept in order to select different faults if a series of fault injection simulations is performed. The fault injection database is created before the fault injection campaign begins, according to its requirements. For example, the faults database may include faults that are targeting a specific register or group of registers, pairs of concurrent faults, or completely random faults. In this case Microsoft Excel was used to generate the faults database, but an alternative approach is to embed the fault generation within the FLI function. The number of faults to be injected in each simulation run is configurable through a `#define` directive inside the C code.

After the faults file is created the simulation begins. A VHDL entity that is referred as the fault injector handles the fault injection process. Before continuing with the fault injection mechanism it is necessary to explain the types of faults that have been considered in these experiments. As it has been explained in Chapter 3, SEU induced faults may generate direct bit-flips on registers, or indirect bit-flips when a transient fault lasts long enough to be latched into a register. Only these cases have been considered, since transient faults that are not latched on registers cannot harm the system. Permanent faults have not been considered either, since their rate in new technologies is decreasing steadily, specifically in terrestrial applications. Furthermore, it is assumed that in a safety-critical application the critical ICs such as the microprocessor are
replaced before they reach their wear-out period, where permanent faults occur. This means that only non-destructive faults that affect the pipeline registers are considered.

The mechanism for injecting this type of faults is described in Figure 5.2. The registers within the pipeline unit are updated in every clock cycle. Their inputs come from combinational circuits and/or data from external peripherals. These inputs are also fed to the fault injector. The fault injector reads the first fault or faults to be injected from the faults file and update its internal structures accordingly when the simulation begins. The time of injection is expressed in clock cycles. When the cycle in which the fault is to be injected is reached, the fault injector "masks" the register inputs according to the information read from the faults file and creates a set of erroneous inputs for the registers. At the same time it changes the select signal of the multiplexer so that the erroneous inputs are latched into the registers. Most of the pipeline registers have an enable signal (usually referred to as the hold signal) that is used to stall the pipeline operation for some cycles, when data is expected from an external resource or in the case of multi-cycle instructions. The fault injector overrides this enable signal as shown in Figure 5.2 to guarantee that the selected fault is injected at the selected time of injection, independently of the current state of the pipeline. When the fault or faults are injected, the fault injector reads again the fault file to pick the next set of faults. Since the whole injection process is performed within one clock cycle, the fault injector is able to inject faults that occur in consecutive clock cycles. If there are no other faults to be injected, the fault injector does not interfere again in the operation of the pipeline until the end of the simulation run.

In the case of the redundant pipeline configuration, the fault injector has extra inputs and outputs
for the register inputs of the redundant pipelines. The fault injection process is the same as described above.

During the simulation there are a number of FLI processes that run in parallel and monitor the simulation while gathering useful data. The first set of FLI processes form the execution time monitor that counts the program execution time and stops the simulation when it exceeds a predefined execution time. It is implemented with two C functions:

\texttt{cnt\_time} : In every simulation the program that is executed by the microprocessor has three phases as shown in Figure 5.3. The first stage is the execution of a boot code that writes appropriate values to the configuration registers. This code is situated in the ROM address space which starts from the address 0x00000000. After the boot code the execution jumps in the RAM address space in address 0x40000000 where the actual application program is placed. When the execution of the program finishes some extra code is executed that forces the VHDL simulation model to write the required data to a file. This process is explained in more detail in the following paragraphs. \texttt{cnt\_time} counts the execution time of each phase by observing appropriate signals in the address bus, for the transition from the first stage to the second, while for the transition from the second stage to the third it observes a specific signal that is activated at that time.

\texttt{stop\_sim} : In some cases an error may force the microprocessor to lose program sequence by jumping into wrong sections of the program, possibly entering into an infinite loop, or causing a long delay. This function keeps track of the current execution time and compares it with the predefined expected execution time. If it exceeds a specific number of clock cycles (also predefined in the C code), it stops the simulation and assumes that the executed application has timed out.
The second set of FLI processes are referred to in Figure 5.1 as the System State Monitor and they consist of the following C functions:

**error_monitor**: This function is constantly monitoring the value of one specific signal within the processor that is asserted when the processor enters error mode. The information produced by this function is then written to the report file.

**vld**: This function monitors a signal within the VHDL architecture and indicates whether the program has entered the exit stage (Figure 5.3). This process is used because some faults generate illegal jumps which result in early termination of the simulation, without indication of an error. These cases are rare but need to be identified.

**state_monitor**: This function is used only in the fault injection simulations of the fault tolerant configuration. It records all the state transitions of the system (triple, pair01, pair02, pair12 and fault) in order for them to be written into the final report file.

All the above functions are included in a single C file that holds all the FLI related code. Before compiling the code, the user must assign values to the following macros:

**CLOCK_PERIOD**: This is the value of the clock period that is used in the simulation and is expressed in ps.

**EXECUTION_TIME**: This is the execution time of the application program. It can be obtained by performing an initial fault-free simulation before starting the fault injection campaign and it is expressed in clock cycles.

**MAX_OVERTIME**: This is the number of clock cycles that is allowed as a delay in the execution time before the simulation is characterised as having timed out.

**NOF**: This defines the number of faults that will be injected in one simulation run.

In some simulations faults may not generate a failure or activate a fault detection mechanism, but can however change the microarchitectural state of the system. This can be observed by checking the contents of the register file and the contents of all the pipeline registers and comparing them with the respective ones from a fault free simulation. In order to achieve that
outcome, some parts of the VHDL code have been modified. In particular the VHDL memory model that is used for the register file has an additional non-synthesisable feature that allows it to write the contents of all its internal arrays to a file, when triggered by an input signal. A similar modification has been made to the VHDL code that describes the operation of the pipeline unit in order to write all the contents of its registers to a file. This signal is controlled by software and it is activated when data is written to a specific register on the AMBA bus. This action is performed at the exit stage of the program (Figure 5.3).

A similar mechanism has been implemented for the data memory since it is necessary to check its contents to determine if the application program produced correct results, since no other I/O interface is possible in a simulation environment. However, in order to reduce the size of the output file two additional AMBA registers have been added that hold the initial address of the location of the output data and the length. These registers are again written from the software. This way, only the parts of data memory that contain useful data are written to the file.

The ModelSim simulator supports the use of Tcl script language. For that reason Tcl has been used to analyse all of the output files of the system along with the report generated by the FLI processes. The fault injection campaign is also controlled by the same Tcl script file. A loop is used that initiates a simulation, waits for its termination, collects all the data produced and writes it to a final report file.

The basic steps of a fault injection campaign are summarised as follows:

1. Generate a list of faults that define time (clock cycle) and location (register bit) of the injection.
2. Define the number of faults to be injected in each simulation and the total number of simulations for the campaign.
3. Define the various parameters in the FLI code that depend on the executed program with a golden run (a simulation without any error injection).
4. Run the simulations
5. Analyse the output data.
5.4 Hardware-Based Fault Injection

The use of hardware based fault injection methods is very rarely used. One aspect of hardware-based fault injection as mentioned before is the injection of faults on the physical level when the actual target system is available. This can give some interesting results for the assessment of a system under certain conditions, but is very expensive and time consuming to set up within the design stages of the SoC. The other form of hardware-based fault injection is the use of embedded logic that can be programmed or driven externally to inject the faults. This approach is however feasible with programmable devices (FPGAs mostly) which are often used for system testing and prototyping. The size and the complexity of the system is a limiting factor in this case, although contemporary FPGAs can fit multi-core embedded systems. This second approach can be very useful because it allows effective testing of the architecture in real-time. Once a hardware-based fault injection system is implemented, the number of fault injection campaigns can increase dramatically allowing more effective testing of the system, while maintaining the same detailed fault models that are used in simulation-based fault injection. The basic drawback of hardware-based fault injection in this case is that it is system dependent. The way of injecting faults and most importantly the means of communication and programming of the fault injection logic depends heavily on the architecture of the system under test. The design of the fault injection system can also become time consuming since it may also require the programming of software tools to support the fault injection and its monitoring and data collection parts, but previous experience can significantly expedite the design procedure.

The concept of the mechanism described in Figure 5.2 can be extended to hardware-based fault injection since it is directly synthesisable. However, several changes must be made in order to implement different means of communication with the fault injector and also monitor the fault injection process and collection of the necessary data (see also Figure 5.1). FLI routines cannot be synthesised and also the file I/O operations of the fault injector must be substituted by hardware logic. In order to overcome these problems, it is necessary to take advantage of some specific characteristics and features of the architecture and the available tools of the system under test.

The LEON processor uses the AMBA bus for all on-chip communication and data transfer. The VHDL distribution can be very easily configured in order to add peripherals on the AMBA
bus. A space of 256 MB is available for APB on-chip registers as it can be seen in Table 5.1. A small number of addresses are in use by default by various registers from the peripherals (memory controller, AHB bus configuration, cache controller, timers, UARTs, etc.) but the spaces between 0x800000D0 and 0x8FFFFFFC\(^2\) are free to be used by the fault injector.

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 - 0x1FFFFFFF</td>
<td>512 MB</td>
<td>PROM</td>
</tr>
<tr>
<td>0x20000000 - 0x3FFFFFFF</td>
<td>512 MB</td>
<td>Memory mapped I/O</td>
</tr>
<tr>
<td>0x40000000 - 0x7FFFFFFF</td>
<td>1 GB</td>
<td>SRAM - SDRAM</td>
</tr>
<tr>
<td>0x80000000 - 0x8FFFFFFF</td>
<td>256 MB</td>
<td>APB registers</td>
</tr>
<tr>
<td>0x90000000 - 0x9FFFFFFF</td>
<td>256 MB</td>
<td>Debug support unit</td>
</tr>
</tbody>
</table>

The LEON processor also includes a hardware debug support unit (DSU) to help software debugging on the target hardware. It consists of two sub-units, one responsible for external communication through a serial port or a PCI bus, and one sub-unit that implements the debug support related functions. The DSU controls the integer unit (pipeline) when the processor enters in debug mode. During that mode the DSU unit has access to a number of registers of the integer unit, the IU and FPU register file and the instruction and data cache memories, including their tags, through the address space shown in Table 5.1. Fault injection could be performed through the DSU by forcing the processor to enter debug mode, change the contents of the registers through the DSU interface and resume execution. However, only a small number of IU registers are accessible through the DSU (Y, PSR, WIM, TBR, PC, NPC, FSR and DSU trap registers). Furthermore the DSU cannot support the redundant core configuration without modifications. For that reason, an independent fault injector module is preferred that uses some of the features of the DSU.

The debugging of applications on LEON based target systems is performed through a debug monitor software tool called DSUMON that runs on a host PC. The PC can communicate with the target system through a standard serial port or through the PCI bus. DSUMON communicates with the DSU and has read/write access to some IU registers, register file and memory. It can also initialise LEON peripherals and memory settings. DSUMON can also download

\(^2\) APB register addresses must be a multiple of 4 since all of them are 32-bit wide.
applications to the LEON system and control their execution. This software tool can be used to synchronise the fault injection campaigns.

The system used to perform fault injection at the hardware level is shown in Figure 5.4. The fault injector module is attached to the APB AMBA bus and utilises a part of the available register address space. The registers inside the fault injector are mask registers for the internal IU registers and each of their bits corresponds to a bit of an IU register. IU registers have various sizes, so several different ones can be mapped onto a single fault injector register. The number of bits that are necessary and subsequently the number of registers and the address space utilised, depends on the configuration of the IU. With efficient use of VHDL constants, the VHDL description of the fault injector can take into account the IU configuration and generate only the necessary registers. The fault injector's internal registers can be written and read. When writing the registers, the values written are used as masks for the fault injection process. When a register is read, the value obtained is the current content of the IU register or registers that are mapped to it.

The fault injection process consists of the following steps:

First, the benchmark application must be loaded on the system's memory and a database of faults must be made available. With the use of DSUMON the program execution begins until the time of injection. This can be done with the following command:

```
step [count]
```

where `count` is the desired number of instructions to be executed before injecting the fault. When
the execution reaches this point the processor enters debug mode and the DSUMON has control of it through the DSU unit. At this time, the fault injector registers must be written to, according to the selected fault or faults. Writing a '0' leaves the corresponding IU bit unchanged, while writing a '1' changes the value of the corresponding IU bit. After setting all of the fault injector registers, the fault injection is performed by writing a random value on a special purpose register which activates the necessary signals (see Figure 5.2). The next step is to continue the execution of the program until the time of injection of the next fault, or until the end of the application program. The step command is again used accordingly.

The information concerning execution time of the application can be obtained through the use of a trace buffer included in the DSU which can store time-stamped executed instructions. The use of a watchpoint at any suitable point will give the necessary timing information.

The verification of the processor's status and the correctness of results at the end of the simulation is very straightforward since all memory locations and registers (register file and special purpose registers) can be accessed within DSUMON.

Although this solution can accelerate significantly the process of fault injection, thus allowing a large number of fault injection campaigns to be performed, with the use of many different fault models, it presents some important limitations and difficulties. The first and most important is that it requires a long design period for its implementation and involves a complex communication scheme between the added modules, the existing modules and also the host PC. Further difficulties with synthesis and timing can be faced if the density of the available FPGA is marginal or insufficient. The second problem has to do with the use of existing tools such as DSUMON since they were not created for fault injection and consequently lack some important functions (e.g. support of a script language). This makes the programming of a large number of successive fault injection experiments very difficult to automate. A solution could be the use of a custom software tool, which adds more time in the design of the fault injection system.

5.5 Conclusion

This chapter has investigated the implementation of fault injection mechanisms for digital processor-based designs and has reviewed their suitability for the work in this thesis. The need for fault injection is derived from the increasing use of such designs in safety or mission critical
Software implemented fault injection has been a popular method for assessing fault tolerant computer systems. Main advantages of SWIFI include the easiness of implementation and the fast execution of fault injection campaigns. However, it presents important limitations in accessing locations for injecting faults and low time resolution. Furthermore it can only be applied on processor based systems and not on simpler digital designs.

Simulation-based fault injection can favour detailed fault modelling and fault injection in digital systems. Since contemporary designs are available in a HDL form, fault injection modules can be used along with simulator tools to implement a fault injection platform. The main advantage of this method is that it allows very detailed fault injection on low level hardware models and also very efficient monitoring of the system’s behaviour. The main disadvantage on the other hand is that simulation of very complex system involves in many cases very long simulation runs.

A third approach for fault injection is the use of hardware methods. Apart from injecting physical faults through radiation or electromagnetic interference, it is possible to emulate faults on actual hardware with the use of embedded logic on FPGA prototypes. This method can significantly accelerate the fault injection campaign but its implementation depends heavily on the system under test and usually faces several difficulties during its design.

In the next chapter, fault injection studies on the protected pipeline processor will be presented along with some new theoretical work which models multiple faults.
References


CHAPTER 6:
THE EFFECT OF SOFT ERRORS ON THE
MICROPROCESSOR PIPELINE

THE study of the operation of digital systems in the presence of various types of fault and
error has been an important topic in the field of fault tolerant and safety-critical systems.
It is essential for designers to be aware of the fault tolerant capabilities of the components that
are used in their designs. Due to their variety, complexity of architectures, and also their im­
portance within a design, microprocessors have been thoroughly studied with fault injection
experiments. This chapter presents in detail many results that have been gathered after perform­
ing a number of fault injection experiments on LEON2 processor and its fault tolerant version
that has been the result of the work of this thesis. A fault model for the injection of multiple
non-concurrent faults is introduced and applied for the injection of double faults.

6.1 Previous Work

A number of results from fault injection experiments that use SWIFI or simulation-based tech­
niques on several different microprocessors have been reported in the literature. The main
objective of these experiments is to assess the suitability of each particular processor (or more
complex multi-processor architecture) to operate in radiation environments, and to characterise
the susceptibility to SEUs. The work presented in [6.1] is a detailed analysis of transient fault in­
jection experiments of a fault tolerant, dual-processor configuration. Results on fault detection,
error activation and fault latencies are discussed. The impact of SEUs on the data cache mem­
ory is presented in [6.2]. Analysis of the susceptibility of different parts (instruction unit, data
cache, instruction cache and register file) of a pipelined processor can be found in [6.3, 6.4].
In [6.5] fault injection experiments are performed on a deeply pipelined, out-of-order micro-
processor in order to detect its most vulnerable portions. The fault injection experiments are repeated on an improved configuration where these portions are protected using low-overhead fault tolerant techniques. The effects of transient and permanent faults on the program's control flow on a RISC processor is examined in [6.6]. In [6.7] results from fault injection experiments are reported to demonstrate the fault coverage of a proposed fault tolerant technique.

In many cases, fault injection experiments are used to demonstrate the efficiency in SER prediction of a fault injection platform or method. Such an example is the testing of an architecture based on a 80C51 microcontroller [6.8] by using the Code Emulated Upset (CEU) platform [6.9]. In [6.10] the effectiveness of a fault list reduction technique is demonstrated with results of fault injection on several combinational circuits.

In most of the forementioned work, the analysis of the effect of faults on the targeted system is not the main objective. For this reason the results presented do not cover in full detail many interesting subjects such as fault latencies or fault propagation. Furthermore, there is a gap in the knowledge with regards to the effects of multiple faults on microprocessor systems.

6.2 Fault and Error Propagation in the Pipeline

The relatively high susceptibility of memory chips to SEU soft errors is well known. Particles which enter the IC package can alter internal bits and may either corrupt stored data or affect the execution sequence. Such problems are solved with the use of parity bits or error detection and correction codes (EDAC). Control parts of the processor such as its pipelined execution unit are more rarely protected, although they contain significant amount of non-programmer visible storage elements. There are two different mechanisms by which soft faults are generated and propagate within microprocessor control circuits and data paths. These are shown in Figure 6.1. In the first mechanism, a particle strike causes a direct change of state in one or more bits of a register. In the second mechanism, a transient fault is produced by a particle strike or by electromagnetic interference on combinational logic. These single, discrete event transients change the voltage of one or more nodes for a small amount of time. The amplitude of the voltage drop (or increase) and its duration depend on the technology of the device [6.11]. As has been discussed in Chapter 3, the rate of the second type of faults is increasing in new technologies. When the duration of such a transient event is sufficient to overlap a clock edge,
it can propagate to the stage registers and cause errors in one or more bits, depending on the number of possible propagation paths. The effects of a fault on a pipeline register do not always lead to a failure. In many cases the affected register is not used by the current instruction, or the fault is overwritten before it propagates. Many processors include mechanisms that can detect the fault by generating special interrupts when the processor enters an incorrect state, or attempts a restricted operation. However, in some cases the fault may lead to a microprocessor failure, such as the generation of an incorrect result, or the loss of program sequence in which the processor enters an infinite loop and 'hangs'. The diagram shown in Figure 6.2 shows the effects of a fault in the pipeline and is based on a diagram presented in [6.12].

6.3 Modelling Single and Multiple Faults

The fault model used in almost all of the literature is the single bit flip of a state element. Multiple faults have only been considered when testing fault tolerant memories, but in these cases these faults are always assumed to have occurred concurrently. Non-concurrent fault pairs have been used in [6.13] purely for a theoretical analysis of the effectiveness of design diversity in redundant systems. However, the time of occurrence is not modelled in detail. As has been explained in previous chapters, radiation-induced errors may occur in several consecutive clock
cycles due to the characteristics of future CMOS sub-micron technologies. Furthermore, these errors may occur in different locations, depending on the different fault-propagating paths. Electromagnetic interference (EMI) with a periodic form can also be a source of consecutive errors. For that reason it is necessary to enhance the existing fault models with temporal parameters in order to efficiently validate the systems under test.

The easiest way to model non-concurrent multiple faults is to select their time of occurrence randomly in the specified time domain, and independently from each other. However, this model cannot represent accurately radiation induced multiple faults. For example, if two or three faults on one or more state elements occur due to a particle strike, they will all occur within a small number of clock cycles, just after the particle hits the device. Similarly, multiple faults that are caused by an EMI disturbance will occur within a time period that depends on the duration of the disturbance.
6. The Effect of Soft Errors on the Microprocessor Pipeline

6.3.1 A Novel Injection Method for Multiple Faults

Statistical distributions can be used to generate sets of random numbers in the time domain that are concentrated around a specific point in time. The normal distribution is a widely used and important distribution that has suitable properties for this application. It is symmetric and has a bell-shaped density function with a single peak. The probability density function (pdf) of the normal distribution with mean \( \mu \) and standard deviation \( \sigma \) is given by:

\[
f(t) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(t-\mu)^2}{2\sigma^2}}
\]  
(6.1)

and its cumulative distribution function (cdf) is defined by

\[
F(t) = \int_{-\infty}^{t} \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx
\]  
(6.2)

Figure 6.3 shows the pdf and cdf of the normal distribution for various values of \( \mu \) and \( \sigma \). Some very notable qualities of the normal distribution that are often referred to as the empirical rule are the following:

- 68% of the observations fall within one standard deviation of the mean, which is, between \((\mu - \sigma)\) and \((\mu + \sigma)\).
6. The Effect of Soft Errors on the Microprocessor Pipeline

- 95% of the observations fall within two standard deviations of the mean, which is, between \((\mu - 2\sigma)\) and \((\mu + 2\sigma)\).

- 99.7% of the observations fall within three standard deviations of the mean, which is, between \((\mu - 3\sigma)\) and \((\mu + 3\sigma)\).

The method of generating random numbers using the normal distribution is very simple and is based on the use of the cumulative distribution function (Equation 6.2 and Figure 6.3(b)). In order to generate \(N\) such numbers, the following two steps are followed:

1. Select a uniformly-distributed set of \(N\) numbers in the interval \(0 \leq F(t) \leq 1\).

2. Use the inverse of the cumulative distribution function using as inputs the numbers that have been obtained from step 1.

Referring to equations 6.1 and 6.2, the variable \(t\) represents continuous time. However, in the fault injection experiments that have been performed time is expressed in system clock cycles which is a variable with discrete integer values. For that reason all the values obtained using the above method are rounded to the closest integer.

The empirical rule can be used to calculate the value of \(\sigma\) according to the needs of the fault injection experiment. For example, for a preselected value of \(\mu\), 99.7%\(^1\) of the selected numbers will be in the region between \(\mu - 3\) and \(\mu + 3\) clock cycles, if \(\sigma = 1\).

For fault injection experiments concerning faults occurring during the execution of a program, all values must be constrained within the program execution time. However, by using the above method there is a finite probability of obtaining a time of fault occurrence that is either less than zero, or greater than the program execution time. This probability increases for large values of \(\sigma\), or when the preselected value of \(\mu\) is close to zero, or close to the end of the program execution time. In the event that this happens, the value is rejected and another integer random number is selected according to the uniform distribution function:

\[
F(t) = \frac{t}{t_{\text{max}}} \quad \text{(6.3)}
\]

\(^1\) This value approximates 100% when the obtained values are rounded to integers.
6. The Effect of Soft Errors on the Microprocessor Pipeline

Figure 6.4: By rejecting values outside some specified limits and replacing them with uniformly distributed random values, the (a) pdf and (b) cdf of the normal distribution change. In this example the limits are 0 and 10

where $t_{\text{max}}$ is the last execution cycle number being considered in the fault injection campaign. By using this scheme it is guaranteed that the chosen fault or faults will be injected only during the execution of the program, and in addition it provides a method for injecting concentrated or uniformly distributed individual or multiple faults. For small values of $\sigma$, the fault profile takes the form of the cumulative distribution function of 6.1 and as $\sigma$ increases it tends towards the uniform distribution of 6.3. Figures 6.4(a) and 6.4(b) depict the graphical representation of the effects of the above scheme.

6.4 Fault Injection Campaign Setup

Fault injection experiments have been performed on two microprocessor architectures. The first was the LEON2 microprocessor without any modifications (hereafter known as the single pipeline architecture) and the second was a fault-tolerant version of LEON2 with redundancy on the pipelined execution unit as described on Chapter 4 (hereafter known as the redundant pipeline architecture).

There were two main categories of fault injection campaigns. One in which single faults were injected in each simulation and one where double faults were injected following the fault model described in the previous section to define the time of injection for each pair of faults. For each
pair, a random value for $\mu$ was selected and the time of injection for each fault was defined using a preselected value of $\sigma$. Due to the long duration of VHDL simulations, only a small number of values for $\sigma$ has been used (Table 6.1). Since the processor being investigated has a 5-stage pipeline, it is interesting to observe its behaviour when both faults occur concurrently or within a small time difference. This is a realistic approach to model the effects of a single particle strike. By setting $\sigma = 0.833$, 99.7% of the faults occur within 5 clock cycles. Another set of simulations was performed with concurrent faults. The value of 10 is selected to increase the level of distribution by about one order of magnitude. In this case, 99.7% of faults occur within 60 clock cycles. The next value of $\sigma$ was chosen so that the level of distribution approximately covers the total execution time of the shortest benchmark program. As a result, 99.7% of faults occur within 3060 clock cycles. Furthermore, a set of simulations was performed where the time of occurrence for each fault was selected by using a uniform distribution. The latter two simulation campaigns can be used to model sets of fault that are caused by different events (e.g. two independent particle strikes, or intermittent electromagnetic interference).

Table 6.1: Levels of distribution of fault occurrence

<table>
<thead>
<tr>
<th>Concurrent</th>
<th>The two faults are concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma = 0.833$</td>
<td>The two faults occur within 5 clock cycles</td>
</tr>
<tr>
<td>$\sigma = 10$</td>
<td>The two faults occur within 10 clock cycles</td>
</tr>
<tr>
<td>$\sigma = 510$</td>
<td>The two faults occur within 3060 clock cycles</td>
</tr>
<tr>
<td>Uniform</td>
<td>The time of the two faults is chosen by using a uniform distribution</td>
</tr>
</tbody>
</table>

The register to be affected by each fault is selected randomly. The probability of one register being selected depends on the number of bits it consists of. The registers of the pipeline have been grouped according to the stage they belong to. As a result there are 5 groups for each stage (fetch, decode, execute, memory, write back) and one group of special purpose registers. These groups are shown in detail in Tables 6.2 and 6.3.

Since the behaviour of the processor after injecting faults depends on the workload, several benchmark applications have been used in these fault injection experiments:

- *mtx4x4*: A program that multiplies two 4x4 integer matrices and stores the result at a specified memory location.
6. The Effect of Soft Errors on the Microprocessor Pipeline

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Number of bits</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>fe.pc</td>
<td>30</td>
<td>Program counter</td>
</tr>
<tr>
<td>fe.branch</td>
<td>1</td>
<td>Branch indicator</td>
</tr>
<tr>
<td>de.inst</td>
<td>32</td>
<td>Instruction</td>
</tr>
<tr>
<td>de.pc</td>
<td>30</td>
<td>Program counter</td>
</tr>
<tr>
<td>de.mexc</td>
<td>1</td>
<td>Memory exception</td>
</tr>
<tr>
<td>de.annul</td>
<td>1</td>
<td>Instruction annul bit</td>
</tr>
<tr>
<td>de.cnt</td>
<td>2</td>
<td>Cycle number (multi-cycle instruction)</td>
</tr>
<tr>
<td>de.mulcnt</td>
<td>5</td>
<td>Cycle number (multiplier)</td>
</tr>
<tr>
<td>de.pv</td>
<td>1</td>
<td>Program counter valid flag</td>
</tr>
<tr>
<td>de.cwp</td>
<td>3</td>
<td>Current window pointer</td>
</tr>
<tr>
<td>de.step</td>
<td>1</td>
<td>Single step</td>
</tr>
<tr>
<td>ex.lbpl</td>
<td>1</td>
<td>Load bypass enable</td>
</tr>
<tr>
<td>ex.lbpl2</td>
<td>1</td>
<td>Load bypass enable</td>
</tr>
<tr>
<td>ex.ctrl</td>
<td>82</td>
<td>Pipeline control registers</td>
</tr>
<tr>
<td>ex.write.cwp</td>
<td>1</td>
<td>Write current window pointer</td>
</tr>
<tr>
<td>ex.write.icc</td>
<td>1</td>
<td>Write integer condition codes</td>
</tr>
<tr>
<td>ex.write.reg</td>
<td>1</td>
<td>Write in register file</td>
</tr>
<tr>
<td>ex.write.y</td>
<td>1</td>
<td>Write Y register</td>
</tr>
<tr>
<td>ex.rs1data</td>
<td>32</td>
<td>Source operand 1</td>
</tr>
<tr>
<td>ex.rs2data</td>
<td>32</td>
<td>Source operand 2</td>
</tr>
<tr>
<td>ex.alu.cin</td>
<td>1</td>
<td>ALU carry-in</td>
</tr>
<tr>
<td>ex.aluop</td>
<td>3</td>
<td>ALU operation</td>
</tr>
<tr>
<td>ex.alusel</td>
<td>2</td>
<td>ALU result select</td>
</tr>
<tr>
<td>ex.aluadd</td>
<td>1</td>
<td>Add/substract select</td>
</tr>
<tr>
<td>ex.mulstep</td>
<td>1</td>
<td>For MULSCC instruction</td>
</tr>
<tr>
<td>ex.mulinsn</td>
<td>1</td>
<td>For SMUL/UMUL instructions</td>
</tr>
<tr>
<td>ex.ymsb</td>
<td>1</td>
<td>Y register most significant bit</td>
</tr>
<tr>
<td>dci.write</td>
<td>1</td>
<td>Write data cache</td>
</tr>
<tr>
<td>dci.asi</td>
<td>8</td>
<td>Address space identifier for load/store instructions</td>
</tr>
<tr>
<td>dci.enaddr</td>
<td>1</td>
<td>Memory stage address flag</td>
</tr>
<tr>
<td>dci.read</td>
<td>1</td>
<td>Read data cache</td>
</tr>
<tr>
<td>dci.lock</td>
<td>1</td>
<td>Lock data</td>
</tr>
</tbody>
</table>

Table 6.2: Pipeline register grouping
### Table 6.3: Pipeline register grouping (continued)

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Number of bits</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>me.result</td>
<td>32</td>
<td>Result from execute stage</td>
</tr>
<tr>
<td>me.y</td>
<td>32</td>
<td>Pipeline Y register</td>
</tr>
<tr>
<td>me.ctrl</td>
<td>82</td>
<td>Pipeline control registers</td>
</tr>
<tr>
<td>me.memory.load</td>
<td>1</td>
<td>Memory load</td>
</tr>
<tr>
<td>me.mulinsn</td>
<td>1</td>
<td>For SMUL/UMUL instructions</td>
</tr>
<tr>
<td>me.write.cwp</td>
<td>1</td>
<td>Write current window pointer</td>
</tr>
<tr>
<td>me.write.icc</td>
<td>1</td>
<td>Write integer condition codes</td>
</tr>
<tr>
<td>me.write_reg</td>
<td>1</td>
<td>Write in register file</td>
</tr>
<tr>
<td>me.write.y</td>
<td>1</td>
<td>Write Y register</td>
</tr>
<tr>
<td>me.cwp</td>
<td>3</td>
<td>Current window pointer</td>
</tr>
<tr>
<td>me.icc</td>
<td>4</td>
<td>Integer condition codes</td>
</tr>
<tr>
<td>me.addr.misal</td>
<td>1</td>
<td>Misaligned address</td>
</tr>
<tr>
<td>me.irqen</td>
<td>1</td>
<td>Interrupt request enable</td>
</tr>
<tr>
<td>me.ipend</td>
<td>1</td>
<td>Interrupt pending</td>
</tr>
<tr>
<td>me.werr</td>
<td>1</td>
<td>Store error</td>
</tr>
<tr>
<td>me.su</td>
<td>1</td>
<td>Supervisor mode</td>
</tr>
<tr>
<td>dcLsize</td>
<td>2</td>
<td>Size of load/store data (byte, half, word, double)</td>
</tr>
<tr>
<td>dcLsigned</td>
<td>1</td>
<td>Sign extension of load data</td>
</tr>
<tr>
<td>wr.y</td>
<td>32</td>
<td>Pipeline Y register</td>
</tr>
<tr>
<td>wr.ctrl</td>
<td>82</td>
<td>Pipeline control registers</td>
</tr>
<tr>
<td>wr.write.cwp</td>
<td>1</td>
<td>Write current window pointer</td>
</tr>
<tr>
<td>wr.write.icc</td>
<td>1</td>
<td>Write integer condition codes</td>
</tr>
<tr>
<td>wr.write_reg</td>
<td>1</td>
<td>Write in register file</td>
</tr>
<tr>
<td>wr.cwp</td>
<td>3</td>
<td>Current window pointer</td>
</tr>
<tr>
<td>wr.icc</td>
<td>4</td>
<td>Integer condition codes</td>
</tr>
<tr>
<td>wr.result</td>
<td>32</td>
<td>Result from memory stage</td>
</tr>
<tr>
<td>wr.ipcsel</td>
<td>2</td>
<td>Trap program counter select</td>
</tr>
<tr>
<td>wr.trapping</td>
<td>1</td>
<td>Trap generation</td>
</tr>
<tr>
<td>wr.error</td>
<td>1</td>
<td>Error mode</td>
</tr>
<tr>
<td>wr.mexc</td>
<td>1</td>
<td>Memory exception</td>
</tr>
<tr>
<td>wr.intack</td>
<td>1</td>
<td>Interrupt acknowledge</td>
</tr>
<tr>
<td>sregs.cwp</td>
<td>3</td>
<td>Current window pointer</td>
</tr>
<tr>
<td>sregs.icc</td>
<td>4</td>
<td>Integer condition codes</td>
</tr>
<tr>
<td>sregs.tt</td>
<td>8</td>
<td>Trap type</td>
</tr>
<tr>
<td>sregs.tba</td>
<td>20</td>
<td>Trap base address</td>
</tr>
<tr>
<td>sregs.wim</td>
<td>8</td>
<td>Window invalid mask</td>
</tr>
<tr>
<td>sregs.pil</td>
<td>4</td>
<td>Processor interrupt level</td>
</tr>
<tr>
<td>sregs.cc</td>
<td>1</td>
<td>Enable coprocessor</td>
</tr>
<tr>
<td>sregs.ef</td>
<td>1</td>
<td>Enable floating point unit</td>
</tr>
<tr>
<td>sregs.ps</td>
<td>1</td>
<td>Previous supervisor flag</td>
</tr>
<tr>
<td>sregs.s</td>
<td>1</td>
<td>Supervisor flag</td>
</tr>
<tr>
<td>sregs.et</td>
<td>1</td>
<td>Enable traps</td>
</tr>
</tbody>
</table>
6. The Effect of Soft Errors on the Microprocessor Pipeline

- **bitcnt**: An algorithm for testing the bit manipulation abilities of the processor by counting the number of bits in an array of integers using five different methods. This is part of the automotive and industrial control category of the MiBench embedded benchmark suite [6.14] and a few minor modifications have been made in order to adopt its I/O operations to the VHDL simulation environment.

- **qsort**: This program sorts a number of strings using a well known quick sort algorithm. It is also a part of the automotive and industrial control category of the MiBench benchmark suite, but it has been modified to reduce its run-time and to adapt its I/O operations to the VHDL simulation environment.

The execution times of each application are shown in Table 6.4 and their C code can be found in Appendix B.

<table>
<thead>
<tr>
<th>Program</th>
<th>Duration (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtx4x4</td>
<td>3150</td>
</tr>
<tr>
<td>bitcnt</td>
<td>14740</td>
</tr>
<tr>
<td>qsort</td>
<td>46800</td>
</tr>
</tbody>
</table>

6.4.1 Fault Injection Procedure

The basic steps of a fault injection campaign are the following:

1. Generate a list of faults that define time (clock cycle) and location (register bit) of the injection.

2. Define the number of faults to be injected in each simulation and the total number of simulations for the campaign.

3. Define various parameters that depend on the executed program with a golden run (a simulation without any fault injection).

4. Run the simulations.

5. Analyse the output data.
The output data for such a scheme consists of information about the correctness of results, the presence of latent faults inside the microarchitecture after the end of the program execution, the status of the processor at the end of the simulation and the execution time. The simulation categories are therefore defined as follows:

- **No Effect**: The program terminates normally, the results are correct and the contents of the pipeline registers and the register file are identical to those in the golden run.

- **Latent**: The program terminates normally, the results are correct but the contents of the pipeline registers and/or the register file are not the same as those in the golden run.

- **Wrong Result**: The program terminates normally but the results are incorrect.

- **Timed Out**: The program failed to terminate within a predefined time limit and the simulation was halted externally. Since the work in this thesis mainly targets hard real-time applications, this time limit has been kept short (400 clock cycles).

- **Exception**: The processor detected an erroneous condition and created a trap forcing it into error mode.

For each benchmark program two types of fault injection experiments have been performed:

1. **Single Fault**: One set of 6000 simulations with one fault injected in each. The simulations are divided equally with 1000 simulations for each of the 6 register groups, fe, de, ex, etc.

2. **Double Fault**: One set of 2000 simulations for each register group and for every different level of distribution chosen (see Table 6.1), resulting a total number of 60,000 simulations. In every simulation a fault pair is injected.

### 6.4.2 Calculating the Overall Performance

The data obtained from each campaign gives an indication of the behaviour of each pipeline stage. In order to get an overall view of the pipeline behaviour, the total probability theorem can be used. This means that each simulation result category is calculated by combining the figures obtained for each stage and the probability of each stage being hit by a fault. The latter
depends on the number of bits each stage (register group) contains. The percentage for each category is thus calculated as:

\[ P_C(t) = \frac{N_{fe}}{N_{total}} P_{fe} + \frac{N_{de}}{N_{total}} P_{de} + \frac{N_{ex}}{N_{total}} P_{ex} + \frac{N_{me}}{N_{total}} P_{me} + \frac{N_{wr}}{N_{total}} P_{wr} + \frac{N_{sregs}}{N_{total}} P_{sregs} \]  
(6.4)

Where:

- \( P_C \) is the percentage for a category C.
- \( N_{fe}, N_{de}, N_{ex}, N_{me}, N_{wr}, N_{sregs} \) are the number of bits of each register group.
- \( N_{total} \) is the number of bits of the pipeline unit.
- \( P_{fe}, P_{de}, P_{ex}, P_{me}, P_{wr}, P_{sregs} \) are the percentages in each register group for the category C.

6.5 Results from Single Fault Injection

6.5.1 Single Pipeline Top-Level Fault Classifications

The results from the fault injection experiments performed on the single pipeline architecture are shown in Figure 6.5. The dependence of the performance on the processor's workload in this architecture is very clear. It can be seen that mtx4x4 has the worst performance in producing wrong results. This is due to the fact that this application benchmark is computationally intensive with extensive use of the ALU and very few references to the memory. This means that the vast majority of the ALU operations in this benchmark are directly associated with the actual data due to the lack of loops within the code, having as a result higher rates in producing wrong results. This explains the improved rate observed in bitcnt which is also ALU intensive. The qsort benchmark is based on many comparisons in order to sort an array of strings. The percentage of ALU related instructions is smaller and the percentage of load/store instructions is higher, resulting in a generally better performance.

The generation of exceptions (e.g. illegal memory reference, illegal opcode) is also dependent on the workload, as it can be observed from the graph. However, the percentage of generated
6. The Effect of Soft Errors on the Microprocessor Pipeline

exceptions is quite low in an architecture without any additional fault detection mechanism, such as the one that has been used for these simulations. The mechanisms that generate exceptions are the only means of soft fault detection in this case and the obtained results show that they are inadequate when high data integrity is a requirement. Exceptions are mainly generated when faults occur on the fetch stage or on the special registers. As it is explained later, the latter depends on the workload.

The probability of having a simulation characterised as “Timed Out” increases when the executed program uses recursive algorithms and many loops. A fault may increase the number of loops performed, thus delaying the termination of the program. This is consistent with the obtained results where it is seen that qsort which uses a recursive algorithm, has an increased percentage of “Timed Out” simulations while mtx4x4 has the lowest.

The probability of a fault remaining latent within the microarchitecture can be affected by many different factors. A fault may propagate to the register file and remain there unused. In other cases, a fault in the program counter or on loop controlling data may alter the execution sequence, thus changing the microarchitectural state at the end, without however causing implications on the correctness of results. Another class of latent faults are those that occur on unused special registers. Each of these factors depend on the workload, however the overall rate of latent faults seems to be the same for the three benchmarks used in these experiments. This is made more clear when examining the contribution of each stage in the generation of latent faults, for each different benchmark program (see Figure 6.6).

6.5.2 Single Pipeline Detailed Fault Behaviour

Figure 6.6 depicts the contribution of each pipeline stage in the overall pipeline behaviour for the different benchmarks. The first observation, as already mentioned, concerns the high rate of latent faults in special registers. This is expected since the majority of special registers are written during the boot sequence and they are not changed or in some cases even accessed during the execution of the application. The second observation concerns the fetch stage where there is a high rate of generated exceptions. This can be explained by the fact that the main register of the fetch stage is the program counter and a fault in one of its bits can create an illegal memory reference. The decode stage seems to have a similar behaviour but to a lesser extent. The special registers contribute an important percentage to the generated exceptions in
6. The Effect of Soft Errors on the Microprocessor Pipeline

The Effect of Soft Errors on the Microprocessor Pipeline

The bitcnt benchmark. This is due to the fact that with bitcnt, faults on special registers have a higher probability of propagating to the pipeline's output. The reason for that is explained in the next section, where propagation probabilities are presented. Faults in the execution stage are mainly responsible for the generation of wrong results. This is more apparent in the two ALU intensive benchmarks (mtx4x4 and bitcnt). Finally, faults in the fetch stage seem to be the main cause of "Timed Out" simulations. As explained before, faults on the program counter have a direct effect on the program executing sequence and can significantly delay termination.

It is difficult to compare these results with other reported work, mainly because of the lack of experiments on the same microprocessor. The only relevant analysis made on LEON2 processor can be found in [6.3] however this used a hardware-based fault injection platform which allow more experiments to be performed. The detailed fault injection mechanism and the exact processor configuration are not known. Furthermore, in this work registers are grouped according to the pipeline stage they belong to, without information about the special registers. Comparisons between the different categories are difficult, also due to the fact that different benchmark programs have been used. The only common benchmark is mtx4x4 (however not the exact same code has been used) results from which are presented for comparison in Table 6.5. As it can be seen the results from Rebaudengo et al. show that generation of wrong results occurs less frequently, but there is a slightly increased number of timed out executions. As already mentioned, these differences cannot be easily explained without knowing important details on the fault injection mechanism, the processor configuration and the group of registers that have been
targeted.

Table 6.5: Comparison of results for the *mtx4x4* benchmark

<table>
<thead>
<tr>
<th></th>
<th>Results (%)</th>
<th>Results from [6.3] (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fe</td>
<td>44.4</td>
<td>8.5</td>
</tr>
<tr>
<td>de</td>
<td>70.9</td>
<td>2.5</td>
</tr>
<tr>
<td>ex</td>
<td>72.5</td>
<td>2</td>
</tr>
<tr>
<td>me</td>
<td>84.5</td>
<td>1.9</td>
</tr>
<tr>
<td>wr</td>
<td>89.4</td>
<td>2.8</td>
</tr>
<tr>
<td>sregs</td>
<td>27.7</td>
<td>65</td>
</tr>
</tbody>
</table>

An important number of fault injection simulations showed a delayed termination in the execution of the program. A delay of even a few clock cycles can be very important in hard real-time applications and their study is of great importance. Some statistics related to this issue are shown in Figure 6.7 and Table 6.6. Since simulations that delayed more than 400 clock cycles were characterised as “Timed Out”, the numbers shown in this graph are smaller than 400. It can be seen that the majority of the delayed executions come from faults in the fetch stage, in a similar manner with the “Timed Out” category.

Table 6.6: Number of delayed simulations in the single pipeline architecture

<table>
<thead>
<tr>
<th></th>
<th>fe</th>
<th>de</th>
<th>ex</th>
<th>me</th>
<th>wr</th>
<th>sregs</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtx4x4</td>
<td>489</td>
<td>97</td>
<td>73</td>
<td>21</td>
<td>15</td>
<td>0</td>
<td>695</td>
</tr>
<tr>
<td>bitcnt</td>
<td>396</td>
<td>60</td>
<td>65</td>
<td>40</td>
<td>23</td>
<td>21</td>
<td>605</td>
</tr>
<tr>
<td>qsort</td>
<td>315</td>
<td>41</td>
<td>23</td>
<td>9</td>
<td>6</td>
<td>33</td>
<td>427</td>
</tr>
</tbody>
</table>

6.5.3 Redundant Pipeline Architecture

The results from the fault injection experiments performed on the redundant pipeline architecture are shown in Figure 6.8. As it can be seen in this graph there is a substantial improvement in fault tolerance compared to the single pipeline architecture. Although it is not visible in this graph, there were some simulations that produced wrong results with the *mtx4x4* and *bitcnt* benchmarks. This issue is associated with the interface to the cache memories and has been
Figure 6.6: Pipeline stages behaviour in single pipeline architecture under single fault injection
6. The Effect of Soft Errors on the Microprocessor Pipeline

Figure 6.7: Histogram of the delayed simulations in the single pipeline architecture (the percentage refer to the total amount of delayed simulations in each benchmark)

thoroughly discussed in Chapter 4. The configuration used in the simulations with the \textit{qsort} benchmark program implemented the improvements suggested in Chapter 4\textsuperscript{2} to address the above problem. For that reason the performance in this case is better and no wrong results are observed. Overall, a very important characteristic of the redundant architecture apart from its obvious fault tolerant properties is the very small dependency on the application that is executed.

The main reason that has a negative effect on the “No Effect” category is the presence of latent faults. As it can be seen in Figure 6.9 the special registers are the most susceptible to maintain latent faults. The rate of latent faults is however reduced compared to the single pipeline architecture due to the fact that latent faults in this case are only observed on the pipeline registers, since no single fault can propagate to the register file without being detected and masked. It can be argued that latent faults in this architecture cannot cause a system failure since even if at any point they propagate to the pipeline output they will be detected. By adding the “No Effect” and “Latent” categories it is concluded that the system achieves a fault coverage that reaches 99.9\%, 99.8\% and 100\% for \textit{mtx4x4}, \textit{bitcnt} and \textit{qsort} respectively.

By tracking the details of the system’s reconfiguration actions (from triple to pair mode) it is also possible to comment on the probability of a fault propagating to one of the pipeline’s output signals. It is also possible to obtain the latency of the propagation by the difference between the time of fault injection and the time of the beginning of the reconfiguration. Figure 6.10

\textsuperscript{2} The cache interface signals are passed through majority voters.
6. The Effect of Soft Errors on the Microprocessor Pipeline

Figure 6.8: Redundant pipeline architecture performance under single fault injection

shows the probability of a fault propagating to the output for the whole pipeline unit and for each stage separately. All faults in the fetch stage immediately affect the outputs since both the program counter and the branch flag are connected to the instruction cache. It can be noted that the propagation probability is similar for every benchmark program in each stage except for the special registers. Faults in a special register can remain dormant for a long period of time and manifest their presence to an output only when special conditions are met. For example a fault in the trap base address (TBA) register will only be activated if a trap is caused during the execution of a program. This example is highlighted in these fault injection experiments. As it can be seen, the propagation probability in the special registers is much higher in the bitcnt benchmark compared to the other two benchmarks. This is due to the fact that during the execution of bitcnt, "window_overflow" and "window_underflow" traps [6.15] are generated several times. Since the TBA register is the largest among the special registers, faults of this type are frequent in the fault injection experiments, thus increasing the propagation rate in this case. The difference in the behaviour of faults on special registers compared to the other groups of registers is also shown in Tables 6.7, 6.8 and 6.9 where some statistics regarding fault propagation are shown. It must be noted that since a small number of signals are removed from the comparator signals and are passed only through majority signals\(^3\), a few propagated faults do not cause any reconfiguration actions. This means that the figures regarding fault propagation are slightly underestimated, mainly in the execute stage.

\(^3\) See section 4.3.2 and Table 4.1 in Chapter 4
Figure 6.9: Pipeline stages behaviour in the redundant pipeline architecture under single fault injection
6. The Effect of Soft Errors on the Microprocessor Pipeline

Figure 6.10: Probability of a fault propagating to the pipeline output

Table 6.7: Fault propagation characteristics (mtx4x4)

<table>
<thead>
<tr>
<th></th>
<th>fe</th>
<th>de</th>
<th>ex</th>
<th>me</th>
<th>wr</th>
<th>sregs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. value (clock cycles)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Max. value (clock cycles)</td>
<td>1</td>
<td>19</td>
<td>22</td>
<td>33</td>
<td>29</td>
<td>18</td>
</tr>
<tr>
<td>Mean value (clock cycles)</td>
<td>1</td>
<td>1.9</td>
<td>3</td>
<td>3.3</td>
<td>1.9</td>
<td>3.3</td>
</tr>
<tr>
<td>Propagated faults (%)</td>
<td>100</td>
<td>73.5</td>
<td>47.1</td>
<td>32.7</td>
<td>24.5</td>
<td>9.2</td>
</tr>
</tbody>
</table>

Table 6.8: Fault propagation characteristics (bitset)

<table>
<thead>
<tr>
<th></th>
<th>fe</th>
<th>de</th>
<th>ex</th>
<th>me</th>
<th>wr</th>
<th>sregs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. value (clock cycles)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Max. value (clock cycles)</td>
<td>1</td>
<td>29</td>
<td>53</td>
<td>110</td>
<td>22</td>
<td>6382</td>
</tr>
<tr>
<td>Mean value (clock cycles)</td>
<td>1</td>
<td>1.9</td>
<td>2.2</td>
<td>3.5</td>
<td>1.3</td>
<td>1265.9</td>
</tr>
<tr>
<td>Propagated faults (%)</td>
<td>100</td>
<td>79.3</td>
<td>48.3</td>
<td>31.2</td>
<td>26.1</td>
<td>76.6</td>
</tr>
</tbody>
</table>

6.6 Results from Multiple Fault Injection

Figures 6.11 and 6.12 present the detailed results from the simulations with pairs of faults injected in each run. These graphs present the overall processor behaviour (for the two architectures) in every category, as a function of the fault timing distribution through different values of $\sigma$. 
Table 6.9: Fault propagation characteristics (qsort)

<table>
<thead>
<tr>
<th></th>
<th>fe</th>
<th>de</th>
<th>ex</th>
<th>me</th>
<th>wr</th>
<th>sregs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. value (clock cycles)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Max. value (clock cycles)</td>
<td>1</td>
<td>22</td>
<td>21</td>
<td>23</td>
<td>53</td>
<td>41288</td>
</tr>
<tr>
<td>Mean value (clock cycles)</td>
<td>1</td>
<td>1.8</td>
<td>2.2</td>
<td>1.5</td>
<td>1.4</td>
<td>3148.6</td>
</tr>
<tr>
<td>Propagated faults (%)</td>
<td>100</td>
<td>78.7</td>
<td>50.7</td>
<td>29.9</td>
<td>26.4</td>
<td>16.2</td>
</tr>
</tbody>
</table>

6.6.1 Faults without Effect

As expected, the percentage of simulations that finish without any observed deviation from the fault-free behaviour decreased compared to the single fault injection campaign in the single pipeline architecture (Figure 6.11(a)). This observed decrease is approximately the same for each benchmark and varies between 13% to 15% for mtx4x4, 9.1% to 13% for bitcnt and 11% to 13% for qsort. The dependence on the workload for this category is again observed with qsort having a much better performance compared to the other two benchmark programs. The effect of inter-fault occurrence time is not significant but a general trend is apparent, showing that the performance deteriorates as the time between faults increases. The level of deterioration is the same for all the benchmarks.

The reduction in the “No Effect” category (Figure 6.12(a)) for the redundant pipeline architecture varies from 1.2% to 10.4% for mtx4x4, 1% to 11.4% for bitcnt and from 2.7% to 13.4% for qsort. In this case the behaviour of the system seems to be much more dependent on inter-fault occurrence time. For concurrent faults the rate of effectless fault pairs is the highest, since in many cases they manifest as single faults (e.g. in most cases, two concurrent faults on the same register have the same effect as a single fault on that register). However, when the two faults occur at different times, the probability of disrupting the normal processor operation increases. As it can be observed, small inter-fault occurrence times tend to have the most severe impact on this architecture. This is expected, because of the critical reconfiguration period that starts after the detection of the first fault and lasts for a few clock cycles. A second fault occurring during that period, may disrupt the normal execution by either triggering a fault detection mechanism or causing a failure. As the inter-fault occurrence time increases, the effects of the fault pairs are less intense. Another important observation in this graph is that the behaviour of this architecture does not depend heavily on the workload, since all benchmarks have approximately the
same percentages in all of the categories.

6.6.2 Wrong Results

Significant increase is observed in the generation of incorrect results in the single pipeline architecture (Figure 6.11(b)). The percentages for different values of $\sigma$ do not present important variation, although once again a trend of increase can be observed as the inter-fault occurrence time increases. Compared to the results obtained from the single fault injection campaigns, the percentages of incorrect result generation is observed to be slightly less than doubled for each benchmark.

The percentages of incorrect result are very different for the redundant pipeline architecture (Figure 6.12(b)). For the two benchmarks ($\text{mtx4x4}$ and $\text{bitcnt}$) the percentage increases with a peak at $\sigma = 10$, but it remains reasonably low. For $\text{qsort}$ where the improved architecture has been used, the percentage is even lower and remains unchanged for different levels of inter-fault timing distribution. An apparently paradoxical situation can be observed for $\text{mtx4x4}$ and $\text{bitcnt}$ in the category of concurrent faults. The percentage obtained in this case is lower than the respective percentage obtained for single fault injection. This can be explained by the fact that the presence of two concurrent faults increase the probability of at least one being detected before any of the two create any type of failure. Hence, in some cases the two concurrent faults are equivalent to a more "easily detectable" single fault. This is not observed with the $\text{qsort}$ benchmark where the wrong results generated from the single fault injection campaign were zero.

6.6.3 Timed Out Simulations

The percentage of timed out simulations does not present any dependency to the inter-fault occurrence time in the single pipeline architecture as it can be seen in Figure 6.11(c). Similarly to the single fault injection campaign, the $\text{qsort}$ benchmark has a higher rate of timed out simulations due to its longer execution time and the strict time out limit that has been set in these experiments. This is also observed in the redundant pipeline architecture (Figure 6.12(c)), where the timed out percentage for $\sigma = 10$ is as high as the average "Wrong Result" percentage for this benchmark.
The vast majority of failures originate from fault pairs on specific critical registers that are concurrent or the second occurs shortly after the first. This is explained in the following example: Assuming that pipeline 0 has been disabled because of the first fault, data from the registers of pipeline 1 are transferred to pipeline 0 to overwrite the fault. If a second fault occurs on a register of pipeline 1 that does not instantly propagate to the output, this fault will be transferred to pipeline 0 without being detected. When pipeline 0 is reintroduced to the system, pipeline 0 and 1 will contain an identical fault which may lead to a system failure. As it can be seen from the results, this scenario has a very low probability of happening.

6.6.4 Latent Faults

The percentages of simulations that finish normally with correct results but latent faults are detected, are shown in Figure 6.11(d) for the single pipeline architecture and in Figure 6.12(d) for the redundant pipeline architecture. A small increase is observed in all cases compared to the results obtained from the single fault injection. There is not a general trend that characterises the behaviour of the two architectures in relation to the inter-fault occurrence time, since no significant variation is observed for different values of $\sigma$.

6.6.5 Detected Faults

The number of simulations where the faults are detected tend to increase in the single pipeline architecture (Figure 6.11(e)) as $\sigma$ increases. The relative increase is kept the same among the three different benchmarks. Compared to the results obtained from the single fault injection campaign, the increase in the detection of fault conditions by the mechanisms of the LEON2 processor varies between the benchmark programs. In particular it is almost doubled with mtx4x4, is significantly increased but not doubled with qsort and is remained at the same levels with bitcnt. It must be noted that the detection in the single pipeline architecture is performed by LEON's own fault detection mechanisms. On the other hand, fault detection is performed solely by the proposed fault tolerant mechanism in the case of the redundant pipeline architecture. In this case, only faults that are not correctable are detected. The difference between the two cases can be seen when comparing Figures 6.11(e) and 6.12(e). When the fault pairs are concurrent, very few cases are uncorrectable. However, when the second fault occurs a few
cycles after the other it is more difficult for both faults to be masked. Hence, the detection percentages rise to 9% – 10% for $\sigma = 0.833$. As $\sigma$ increases, less fault pairs become uncorrectable and the percentage drops. As shown in the relevant graph, this behaviour is not dependent to the benchmark program, since the percentages in all three are very similar.

### 6.6.6 Overall Reliability

By adding the “No Effect” and “Exception” categories we can obtain a broader “Failure Free” category that gives a general image of the reliability of the system. As mentioned earlier, the “Latent” category can also be seen as fail safe for the redundant architecture, since even when the latent faults propagate to an output signal, they will be detected and when possible corrected. Table 6.10 summarises the level of reliability achieved by the proposed redundant architecture by presenting the figures of the “Failure-Free” category for single and double faults. As it can be seen, the level of reliability does not change in the initial configuration, while it is slightly reduced in the improved which was used with the qsort benchmark.

Table 6.10: Percentage of failure-free simulations with single and double fault injection for the redundant pipeline

<table>
<thead>
<tr>
<th></th>
<th>Single Faults</th>
<th>Double Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtx4x4</td>
<td>99.93%</td>
<td>99.93%</td>
</tr>
<tr>
<td>bitcnt</td>
<td>99.83%</td>
<td>99.83%</td>
</tr>
<tr>
<td>qsort</td>
<td>100%</td>
<td>99.94%</td>
</tr>
</tbody>
</table>

### 6.7 Conclusion

This chapter has presented a very detailed analysis of fault effects on two pipeline processor configurations. The results have revealed very subtle features of both the LEON2 processor itself and its behaviour in the presence of faults in its pipeline unit. A substantial improvement in reliability has been proven for the proposed redundant architecture by exploring the effects of injection of both single and double faults. A novel fault model for the injection of multiple faults has been proposed and applied for the double fault injection. This new model aims to represent radiation induced soft fault that may be non concurrent. The validity and usefulness of this
Figure 6.11: Single core results for double errors
6. The Effect of Soft Errors on the Microprocessor Pipeline

Figure 6.12: Redundant core results for double errors
model has been proven since different behaviour has been observed when changing the time between the occurrence of the two faults. Although this was more apparent in the redundant architecture due to its nature, a dependency has also been observed in many cases in the single pipeline architecture. This model is completely general and could be used for the injection of multiple faults to many different digital systems.
References


THE aim of this thesis was to study and implement system-on-chip computer architectures suitable for near-future safety-critical automotive applications. The outcome of this work was the design and implementation of a novel fault tolerant architecture for the control parts of microprocessors. This architecture was fully validated by means of a fault injection method based on a simple statistical technique. This chapter summarises the contribution of the thesis on this subject and proposes some areas for further research.

7.1 Contributions of the Thesis

The safety requirements of computer systems in drive-by-wire systems that are currently under development in the automotive sector have been studied. In these systems, which are mainly based on microcontrollers, increased reliability and availability through fault tolerance is necessary at the component and the system level. Strategies that have been traditionally followed in space applications and in fly-by-wire systems found in aeroplanes cannot be directly applied on the automotive sector due to its strict cost and packaging constraints. Advances in VLSI technology allow the implementation of single-chip solutions to this problem that can help in reducing the required level of component redundancy.

The reliability of integrated circuits has been explained. ICs have been traditionally susceptible to soft errors caused mainly by cosmic radiation and radioactive impurities that are contained in their packaging materials. The ever-increasing number of transistors integrated in advancing CMOS technologies along with higher system clock speeds have a negative impact on the error rate of ICs. Faults in combinational circuits are becoming a significant proportion of the overall soft error rate in microprocessor technologies.

The protection of memories against soft errors has been a subject of research for several decades
since the first observation of data corruption due to radiation. However, few methods have been proposed for complex logic within processors such as its core unit. This thesis proposes a fault tolerant architecture based on hybrid redundancy and applies it on a popular SPARC V8 microprocessor to protect its pipeline unit. The architecture can mask single bit faults without interrupting or delaying the microprocessor’s operation. Multiple faults can also be successfully handled through fault masking, or in the cases where this is not possible, through fault detection. This allows the system to remain fail silent. Redundancy is not only applied on the memory elements of the pipeline unit, but also on the combinational parts of the circuit. One main advantage of this concept is its simplicity which allows easy adaption to other blocks within an embedded system, and also on different types of microprocessors.

Validation is an important part of the development of a fault tolerant system. This thesis proposes a fault model for multiple faults on microprocessor architectures based on a simple statistical distribution. This model extends the single bit-flip that is traditionally used to test the susceptibility of microprocessor architectures to radiation induced errors.

Fault injection on simulation models is the most popular method of testing the behaviour of microprocessors in radiation environments. This thesis proposes three different approaches to implement fault injection platforms and highlights their strengths and weaknesses. Two of these methods are based on simulation using VHDL models and simulator tools, while the third is based on an FPGA prototype with embedded fault injection logic.

Finally, this thesis presents a detailed analysis of the effects of faults that occur on pipeline memory elements. The results obtained from extensive fault injection experiments are used to carefully examine the behaviour of a non fault tolerant SPARC V8 processor and compare it with a version of the same processor that uses redundant pipeline units according to the proposed scheme. The results provide useful information on the processors’ behaviour under the presence of faults, demonstrate the dependence of the failure rate on the workload in a non-fault tolerant configuration and also highlight the substantial improvement in reliability of the fault tolerant architecture. The proposed general model for multiple faults is also used in these fault injection experiments and the importance of timing in fault occurrence is proven.
7. Conclusions

7.2 Suggestions for Further Research

The work in this thesis has instigated some different areas of related research.

7.2.1 Design of Embedded Architectures for Safety-Critical Systems

The development of computer systems that target a wide variety of safety-critical control applications has become a subject of interest mainly because of their low cost and high performance. In particular, the main approach that can be found in relevant work involves the design and hardware-software setup of distributed embedded systems. These are networks of microprocessor based nodes (e.g. main control units, sensors, actuators) that can target control systems in aircrafts, vehicles and a variety of industrial applications. These systems rely on off-the-shelf components that have usually a limited reliability level, requiring thus considerable levels of redundancy. A study on the impact of the use of very high-reliable components on the architectures and the cost of such systems for various integrity levels would be very beneficial for the fault tolerant computing community.

The use of custom components (ASIC, FPGA, etc.) on safety-critical computer systems is also relatively recent. Although several well established methods for safety analysis of safety-critical systems exist and are applied in industry, the same does not hold for the design of safety-critical system-on-chip architectures. It would be useful to define one or more different safety analysis processes on the chip level for embedded architectures in order to facilitate the safety assessment of the final system-on-chip and the broader system of which it may be part of. Appendix A has provided some initial comments on HAZOP analysis of VLSI embedded architectures. It was clear that the methods for analysing the safety of embedded systems is inadequate and the definition of a related safety framework would be beneficial.

7.2.2 Ultra-Reliable Embedded Processor

Fault tolerant techniques have been traditionally applied mostly on very complex, high performance microprocessors that are mainly used as servers and need to guarantee very low probabilities of failures and minimal downtime. Due to the high complexity and high performance requirements only those fault tolerant techniques are used that minimise the overhead and per-
formance penalty, thus narrowing the fault coverage. Microprocessors used in safety-critical control applications are generally simpler with average performance requirements. This allows the application of large scale redundancy on their critical parts. Research needs to be performed on the design of an ultra-reliable microprocessor with high levels of fault tolerance on all its sub-blocks (main core, bus, memories, I/Os etc.) for embedded applications.

7.2.3 Fault Tolerant, High-Performance Architectures

State-of-the-art VLSI technology allow the fabrication of single chip systems that can integrate a substantial number of processor cores and several other resources. Based on this fact, it would be possible to develop a large single-chip system with many cores that can operate independently but also as groups, forming this way dynamic, static or hybrid redundant structures. The exact configuration will ideally be defined dynamically by an operating system that will allocate the necessary resources to all the processes according to their level of criticality. For instance, a non-critical process can be assigned to a single core while a highly critical process can be assigned to a group of three cores that are configured to form a hybrid redundant system. A similar concept can be applied also to the other on-chip resources. Such a scheme would also support software fault tolerant methods, with the possibility of running different versions of a program in parallel. This flexible system can combine very high performance with its multi-processing capabilities, and also different levels of hardware and software fault tolerance.

7.2.4 Fault Injection Tools

Fault injection is a very useful method of testing and validating digital systems. Several different methods and approaches have been proposed that have in general different characteristics. This is due to the fact that each tool was built based on one particular target system, thus having limited portability. All the different approaches that can be found in the literature should be unified to one powerful and flexible tool. A very useful study would involve the careful and detailed analysis of the needs of a fault injection tool for simple and complex digital systems, followed by its implementation. Support for different languages (e.g. VHDL, Verilog, SystemC), the possibility of generating synthesizable models for hardware fault injection and a wide variety of fault models are among the main requirements for this work.
HAZOP (Hazard and Operability study) is a technique for identifying and analysing the hazards and operational concerns of a system. Although universally applicable, it has until recently remained confined to the process industries. HAZOP starts from a particular deviation from design intent and works backwards to identify its possible causes and also forwards to explore its consequences (Figure A.1). The identification of deviations is based on the use of predetermined 'guide words', each of which focuses attention on a particular type of deviation. A HAZOP team is formed that applies these guide words on all components and interactions between them that are shown on a system's representation.

A HAZOP is carried out on a representation of the system under study. The representation of the system can use symbols, text or a mixture of two. It can be either physical, logical and in some cases both can be examined. The only restrictions on the representations used are that they must in some form capture the intent of the designer and the HAZOP team must understand them. The representation must enable the team to identify the elements of the system that they need to study. It is essential that the team leader understands in detail the representation. Training may be necessary for team members. If the representations do not cover all the attributes of the system that could affect the system's safety, the HAZOP must be clearly identified as a partial study. Multiple representations can present problems and must be examined in a sequential manner.
manner, unless parallel examination does not create any confusion. Functional programming languages and state transition diagrams present the most significant initial problems. This is because they present the greatest difficulty to the identification of the attributes of the design. It is also difficult to develop clear definitions of the interpretations of the deviations from design intent.

Components and interactions between them have certain properties that reflect the way in which they function. The principle of HAZOP is to study what would happen if the attributes deviate from design intent in order to identify the hazards of the system. The study leader must identify the implicit and explicit attributes in advance of the study and make the implicit ones clear.

Examples of what might typically be considered in determining the scope of a HAZOP are:

- The stage of the system’s life cycle at which the study is to be carried out (preliminary or detailed).
- The context of the study within the overall safety and hazard analyses of the system. This helps to identify previous relevant studies and to define the need to plan subsequent studies.
- The extent of the threats to safety which the system could pose.
- The boundary of the system being studied.
- The boundary of the study.
- The number and the nature of design representations to be studied.
- Safety requirements of the system that may be available. Knowing for example the safety integrity level of the system may be useful in defining which types of hazards are of particular interest.
- Legislation that is relevant to the system under development.
- Any other systems with which the system is to be integrated or connected.
- The purpose to which the results of the study will be put. The results of a HAZOP are typically used for informing system design, but they may also be used for determining the questions to ask a supplier, or for improving the safety of an existing operational system.
The objectives should define the purpose of the HAZOP and what the results of the study should provide. Some issues that should be considered are:

- Whether the study is an overview study to provide initial information or a detailed study to provide definitive information on hazards.
- The types of hazards that the study is intended to identify.
- Whether it is required in this study to check if recommendations in previous studies have been implemented or if measures have been taken to eliminate or mitigate previously identified hazards.
- The context of this study within the overall hazard and safety analyses of the system.

The generic set of guide words that is used in a HAZOP study is:

- **No**: This is the complete negation of the design intention; no part of the intention is achieved, but nothing else happens.
- **More**: This is a quantitative increase.
- **Less**: This is a quantitative decrease.
- **As well as**: This is a qualitative increase, where all the design intention is achieved together with additional activity.
- **Part of**: This is a qualitative decrease, where only part of the design intention is achieved.
- **Reverse**: This is the logical opposite of the intention.
- **Other than**: This is a complete substitution, where no part of the original intention is achieved but something quite different happens.

When it is relevant to examine timing as part of a HAZOP, additional guidewords are needed. The following can be added to the generic set:

- **Early**: Something happens earlier in time than intended.
- **Late**: Something happens later in time than intended.
A. HAZOP Studies on Embedded Architectures

- **Before:** Something happens earlier in a sequence than intended.

- **After:** Something happens later in a sequence than intended.

The generic set of guide words is always applicable (and complete) to describe deviations of a system, although it may need interpreting in a particular application. It is the responsibility of the study leader, who is usually working with one of the designers or someone who knows the design representation well, to define the interpretations of each of the guide words for each attribute of the entities on the representation. An effective way to derive appropriate interpretations is to take the generic meaning of each guide word and to consider what interpretation best expresses its meaning in the given context (with respect to the attribute in question and the nature of the system). The use of HAZOP in programmable electronic systems (PES) is fairly new and best practice is still evolving.

There is no proposed methodology to apply HAZOP on VLSI embedded architectures. However, such an effort would be very useful for custom chips that target safety-critical systems. During the design of such a chip the details of the whole system of which it will be part of could be unknown. Nevertheless, a report of a HAZOP study on its architecture that identifies possible hazards for this specific component would be beneficial for the designers of the safety critical system. Furthermore, it could prove valuable in identifying critical parts of the architecture, or design mistakes and omissions. This is due to the fact that HAZOP studies can be performed many times and during different phases of a design.

The design of VLSI embedded architectures is nowadays very flexible and highly configurable. A top-level representation of the system can be the base for a HAZOP study. By carefully examining the various blocks and their interconnections between them and outside the system’s boundary, it is possible to identify hazards which will lead to design suggestions or corrections. For example, it is possible to set a list of requirements for a memory block such as its number of ports and its speed. The guide words can be tailored appropriately. A generic set of guide words is presented in Table A.1, but these can be changed or more can be added, depending on the details of the system under study. When low-level analysis is necessary (e.g. gate level), other methods are recommended such as FMEA.

The general information about HAZOP that are presented in this Appendix are taken from the book of Redmill, Chudleigh and Catmur [1].
<table>
<thead>
<tr>
<th>No part of</th>
<th>More</th>
<th>Less</th>
<th>As well as</th>
<th>Part of</th>
<th>Reverse</th>
<th>Other than</th>
<th>Early</th>
<th>Late</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>intention is achieved</td>
<td>A</td>
<td>A</td>
<td>All design but with additional results</td>
<td>Only some of the intention is achieved</td>
<td>The logical opposite of the intention</td>
<td>Result other than original intention is achieved</td>
<td>Related to clock time</td>
<td>Related to clock time</td>
<td>Related to order or sequence</td>
<td>Related to order or sequence</td>
</tr>
</tbody>
</table>

**Attribute of multi-bit data and control signals**

| No signal passed | Data value bigger than expected | Data value smaller than expected | Bits are missing but not credible | Information complete but not credible | Signals are not set within one clock cycle |

**Attribute of single-bit data and control signals**

| No signal passed | Reversed logic | Not credible | Signal is not set within one clock cycle |

**Attribute of clock signal**

| No clock signal | Higher frequency than intended | Lower frequency than intended | |

Table A.1: Guide words for VLSI embedded architectures
CHAPTER B:
C CODE OF THE BENCHMARK PROGRAMS

B.1 mtx4x4

#include "leon.h"
#include "test.h"

#define ARRJ.LEN 30
#define TESTJ. END 19

unsigned char * msg = (unsigned char *) IOAREA;
volatile struct regs *lr = (struct regs *) PREGS;

report(test_case) int test_case; { msg[test_case] = 0; } 

int main(void){

  static unsigned long int c[4][4];
  static unsigned long int a[4][4] = { 45, 198, 12, 0,
                                    321, 99, 529, 3,
                                    100, 32, 72, 91,
                                    794, 978, 12, 890};

  static unsigned long int b[4][4] = { 633, 1, 12, 257,
                                    9, 88, 999, 256,
                                    101, 789, 114, 9,
                                    71, 56, 52, 582};

  c[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0] + a[0][2]*b[2][0] + a[0][3]*b[3][0];
  c[0][1] = a[0][0]*b[0][1] + a[0][1]*b[1][1] + a[0][2]*b[2][1] + a[0][3]*b[3][1];
  c[0][2] = a[0][0]*b[0][2] + a[0][1]*b[1][2] + a[0][2]*b[2][2] + a[0][3]*b[3][2];
  c[0][3] = a[0][0]*b[0][3] + a[0][1]*b[1][3] + a[0][2]*b[2][3] + a[0][3]*b[3][3];
  c[1][0] = a[1][0]*b[0][0] + a[1][1]*b[1][0] + a[1][2]*b[2][0] + a[1][3]*b[3][0];
  c[1][1] = a[1][0]*b[0][1] + a[1][1]*b[1][1] + a[1][2]*b[2][1] + a[1][3]*b[3][1];
  c[1][2] = a[1][0]*b[0][2] + a[1][1]*b[1][2] + a[1][2]*b[2][2] + a[1][3]*b[3][2];
  c[1][3] = a[1][0]*b[0][3] + a[1][1]*b[1][3] + a[1][2]*b[2][3] + a[1][3]*b[3][3];
  c[2][0] = a[2][0]*b[0][0] + a[2][1]*b[1][0] + a[2][2]*b[2][0] + a[2][3]*b[3][0];

  ...
B. C Code of the Benchmark Programs

\[
c[3][0] = a[3][0] * b[0][0] + a[3][1] * b[1][0] + a[3][2] * b[2][0] + a[3][3] * b[3][0];
\]

// Write the results contained in the simulated memory on a file
 lr = (struct regs *) PREGS;
 lr->dmpstartadd = c;
 lr->dmplength = 17;

// Finish simulation
 report(TEST_END);
 return 0;
}

B.2 bitcnt

B.2.1 bitcnts.c

/* +++Date last modified: 05-Jul-1997 */
/*
 ** BITCNTS.C - Test program for bit counting functions
 **
 ** public domain by Bob Stout & Auke Reitsma
 **
 ** 21/1/2005 (elet): Modified to be used in LEON simulations in ModelSim */

#include <stdio.h>
#include <stdlib.h>
#include "conio.h"
#include <limits.h>
#include <float.h>
#include "bitops.h"
#include "leon.h"

#define FUNCS 7
#define TEST_END 19

static int CDECL bit_shifter(long int x);
{
    int i, n;
B. C Code of the Benchmark Programs

```c
for (i = n = 0; x && (i < (sizeof(long) * CHAR_BIT)); ++i, x >>= 1)
    n += (int)(x & 1L);
return n;
```

```c
unsigned char *msg = (unsigned char *) IOAREA;
volatile struct regs *Ir = (struct regs *) PREGS;

report(test_case) int test_case; { msg[test_case] = 0; }

int main(void)
{
    int i;
    long j, n, seed;
    int iterations;
    static long results[FUNCS];
    static int (*cdecl pBitCntFunc[FUNCS])(long) = {
        bit.count,
        bitcount,
        ntbl.bitcnt ,
        ntbl.bitcount ,
        /* bit.bitcnt, DOESNT WORK*/
        BW.btbl.bitcount ,
        AR.btbl.bitcount ,
        bit.shifter
    };

    iterations = 10;

    for (i = 0; i < FUNCS; i++) {
        for (j = n = 0, seed = rand(); j < iterations; j++, seed += 13)
            n += pBitCntFunc[i](seed);

        results[i] = n;
    }

    // Write the results contained in the simulated memory on a file
    Ir = (struct regs *) PREGS;
    Ir->dmpstartadd = results;
    Ir->dmplength = 8;

    // Finish simulation
    report(TEST.END);
```
B. C Code of the Benchmark Programs

B.2.2  bitcnt.1.c

/* +++Date last modified: 05-Jul-1997 */
/*
** Bit counter by Ratko Tomic
*/
#include "bitops.h"

int CDECL bitcount(long x)
{
    int n = 0;

    /*
    ** The loop will execute once for each bit of x set, this is in average
    ** twice as fast as the shift-test method.
    */
    if (x)
        do
           n++;
        while (0 != (x = x&(x-1)));
    return(n);
}

B.2.3  bitcnt.2.c

/* +++Date last modified: 05-Jul-1997 */
/*
** Bit counter by Ratko Tomic
*/
#include "bitops.h"

int CDECL bitcount2(long i)
{
    i = ((i & 0xffffffffL) >> 1) + (i & 0x55555555L);
    i = ((i & 0xCCCCCCCCL) >> 2) + (i & 0x33333333L);
    i = ((i & 0xF0F0F0F0L) >> 4) + (i & 0x0F0F0F0FL);
    i = ((i & 0xFF00FF00L) >> 8) + (i & 0x00FF00FFL);
    i = ((i & 0xFFFF0000L) >> 16) + (i & 0x0000FFFFL);
    return (int)i;
}
B. C Code of the Benchmark Programs

B.2.4 bitcnt_3.c

/* +++Date last modified: 05-Jul-1997 */

/**
BITCNT_3.C — Bit counting functions using table lookup
**
** public domain by Auke Reitsma and Bruce Wedding
*/

#include "bitops.h" /* from Snippets */

/**
Bits table
*/

static char bits[256] =
{
    0, 1, 1, 1, 2, 2, 3, 1, 2, 2, 3, 3, 4, 1/* 0 - 15 */
   1, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 4, 5, 1/* 16 - 31 */
   1, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 4, 5, 1/* 32 - 47 */
   2, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 5, 6, 1/* 48 - 63 */
   1, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 4, 5, 1/* 64 - 79 */
   2, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 5, 6, 1/* 80 - 95 */
   2, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 5, 6, 1/* 96 - 111 */
   3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 1/* 112 - 127 */
   1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 4, 5, 1/* 128 - 143 */
   2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 5, 6, 1/* 144 - 159 */
   2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 5, 6, 1/* 160 - 175 */
   3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 1/* 176 - 191 */
   2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 5, 6, 1/* 192 - 207 */
   3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 1/* 208 - 223 */
   3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, 1/* 224 - 239 */
   4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8 /* 240 - 255 */
};

/**
Count bits in each nybble
**
** Note: Only the first 16 table entries are used, the rest could be
** omitted.
*/

int CDECL ntbl_bitcount(long int x)
{
    return
bits[ (int) (x & 0x0000000FUL) ] +
bits[ (int)((x & 0x000000F00UL) >> 4) ] +
bits[ (int)((x & 0x00000F000UL) >> 8) ] +
bits[ (int)((x & 0x000F0000UL) >> 12) ] +
bits[ (int)((x & 0xF0000000UL) >> 16) ] +
bits[ (int)((x & 0x0F0000000UL) >> 20) ] +
bits[ (int)((x & 0xF00000000UL) >> 24) ] +
bits[ (int)((x & 0x0F000000000UL) >> 28) ];

/
** Count bits in each byte
**
** by Bruce Wedding, works best on Watcom & Borland */

int CDECL BW_bitcount(long int x)
{
    union
    {
        unsigned char ch[4];
        long y;
    } U;

    U.y = x;

}

/*
** Count bits in each byte
**
** by Auke Reitsma, works best on Microsoft, Symantec, and others */

int CDECL AR_bitcount(long int x)
{
    unsigned char *Ptr = (unsigned char *)x;
    int Accu;

    Accu = bits[ *Ptr ++ ];
    Accu += bits[ *Ptr ++ ];
    Accu += bits[ *Ptr ++ ];
    Accu += bits[ *Ptr ++ ];
B. C Code of the Benchmark Programs

B.2.5  bitcnt.4.c

/* +++Date last modified: 05-Jul-1997 */

/*
 ** BITCNT.4.C - Recursive bit counting functions using table lookup
 **
 ** public domain by Bob Stout
 */

#include "bitops.h" /* from Snippets */

static char bits[256] =
{
    0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, /* 0 - 15 */
    1, 2, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, /* 16 - 31 */
    1, 2, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, /* 32 - 47 */
    2, 3, 4, 3, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, /* 48 - 63 */
    1, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, /* 64 - 79 */
    2, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, /* 80 - 95 */
    2, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, /* 96 - 111 */
    3, 4, 4, 5, 4, 5, 4, 5, 5, 6, 5, 6, 6, 7, /* 112 - 127 */
    1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5, /* 128 - 143 */
    2, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, /* 144 - 159 */
    2, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, /* 160 - 175 */
    3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 5, 7, /* 176 - 191 */
    2, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6, /* 192 - 207 */
    3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7, /* 208 - 223 */
    3, 4, 4, 5, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 6, 7, /* 224 - 239 */
    4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8 /* 240 - 255 */
};

/*
 ** Count bits in each nybble
 **
 ** Note: Only the first 16 table entries are used. The rest could be
 ** omitted.
 */

int CDECL ntbl_bitcnt(long x)
{
    int cnt = bits[(int)(x & 0x000000FLL)];
B. C Code of the Benchmark Programs

```c
if (OL != (x >>= 4))
    cnt += ntohlbitcnt(x);

return cnt;
}

/*
** Count bits in each byte
*/

int CDECL btbLbitcnt(long x)
{
    int cnt = bits[ ((char *)&x)[0] & 0xFF ];

    if (OL != (x >>= 8))
        cnt += btbLbitcnt(x);

    return cnt;
}

B.3 qsort

/* 10-2-2005(elet): Modified to be used in LEON simulations in ModelSim */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "leon.h"

#define ARR.LEN 30
#define TEST.END 19

unsigned char *msg = (unsigned char *) IOAREA;
volatile struct regs *lr = (struct regs *) PREGS;

report(test_case) int test_case; { msg[test_case] = 0; }

struct myStringStruct {
    char qstring[128];
    // int id;
};

static int compare(const void *e1, const void *e2) {
    int result;
```
result = strcmp((*(struct myStringStruct *)elem1)).qstring,
        (*(struct myStringStruct *)elem2)).qstring);

    return (result < 0) ? 1 : ((result == 0) ? 0 : -1);
}

int main(void
    static int i;

    static struct myStringStruct array[ARRLEN] = {
        "Kurt",
        "Vonneguts",
        "Address",
        "MIT",
        "class",
        "sunscreen",
        "If",
        "could",
        "you",
        "tip",
        "the",
        "future",
        "be",
        "longterm",
        "sunscreen",
        "proved",
        "whereas",
        "the",
        "of",
        "no",
        "experience",
        "this",
        "beauty",
        "of",
        "mind",
        "beauty",
        "faded",
        "20",
        "at",
        "of",
    };

    qsort(array, 30, sizeof(struct myStringStruct), compare);
// Write the results contained in the simulated memory on a file
lr = (struct lregs *) PREGS;
lr->dmpstartadd = array;
lr->dmplength = 30*(sizeof(struct myStringStruct)/4)+1;

// Finish simulation
report(TEST_END);
return 0;
}
CHAPTER C:
AUTHOR’S PUBLICATIONS

A number of publications have resulted from the work in this thesis. These are as follows:


