# Radiation tolerant implementation of a soft-core processor for space applications

by

Johannes Gerhardus van der Horst

*Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Electronic Engineering and Computer Science at the University of Stellenbosch*

Department of Electrical and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, 7602, Matieland, South Africa.

Supervisors:

Prof. S. Mostert Mr. W. Smit

March 2007

Stellenbosch University http://scholar.sun.ac.za

Copyright © 2007 University of Stellenbosch
All rights reserved.

Stellenbosch University http://scholar.sun.ac.za

# Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
J.G. v.d. Horst

Date: . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

Stellenbosch University http://scholar.sun.ac.za

# Abstract

The availability of high density FPGAs has made the use of soft-core processors an attractive proposition for the low volume space market. Soft-core processors combine the power of programmable logic with the ease of use of a conventional processor to provide a highly customisable solution. However, the SRAM FPGAs used as implementation platform are especially susceptable to radiation induced single event upsets, due to the sensitivity of their configuration memory. To safely use these processors in a space environment requires the modification of the processor to safely mitigate these effects.

This thesis presents the process followed to develop and test a fault tolerant implementation of an 8-bit PicoBlaze soft-core processor on a Xilinx Spartan-3 SRAM FPGA.

A thorough investigation was made into the available methods that can be used to mitigate single event upsets, in order to identify the most suitable ones. Guidelines for the application of SEU mitigation techniques to SRAM FPGAs were proposed. A single event upset simulator was designed and constructed to compare the different techniques. It mimics SEUs by injecting errors into the configuration memory of an FPGA.

The results of error injection were used to develop a PicoBlaze implementation with limited overhead, while it still offers a high degree of error mitigation.

Three different designs were tested by proton irradiation to verify the protection afforded by the mitigation techniques. It was found that protected designs were more robust. The cross-section of the FPGA was also determined, which can be used with the SEU simulator to predict the dynamic cross-section of designs.

The work contained in this thesis demonstrates the use of open-source intellectual property with commercial-off-the-shelf components to develop a robust component for use in the miniature spacecraft market.

Stellenbosch University http://scholar.sun.ac.za

# Uittreksel

Die beskikbaarheid van hoë digtheid FPGAs maak programmeerbare logika verwerkers baie aantreklik vir gebruik in ruimtetoepassings. Programmeerbare logika verwerkers kombineer die krag van programmeerbare logika met die gebruiksgemak van tradisionele verwerkers om 'n hoogs aangepaste oplossing te bied. SRAM FPGAs, wat as die implementeringsplatform benut word, is egter sensitief vir korrupsie van geheuebisse deur bestraling. Die verwerker moet aangepas word om hierdie effekte veilig te kan hanteer.

Hierdie tesis beskryf die proses wat gevolg is om 'n fouttolerante weergawe van 'n 8-bis programmeerbare hardeware verwerker, die PicoBlaze, te ontwerp en te toets. Die verwerker is op 'n Spartan-3 FPGA van Xilinx implimenteer.

'n Deeglike studie van die metodes wat gebruik word om teen foute te beskerm is gemaak, om die mees geskiktes te identifiseer. 'n Stel riglyne vir die toepassing van fouttoleransie-tegnieke op SRAM FPGAs is voorgestel. Hardeware is spesifiek ontwerp om die effekte van bestraling in die konfigurasie geheue van FPGAs na te maak. Dit is benut om verskillende fouttoleransie-tegnieke te vergelyk.

Die resultate van die simulator is gebruik om 'n fouttolerante PicoBlaze implementasie te ontwikkel wat 'n beperkte hoeveelheid logika benodig.

Drie verskillende PicoBlaze ontwerpe is met protone bestraal om die verharding teen foute te bevestig. Die verharde ontwerpe was minder sensitief as die verwysingsontwerp. Die kruisdeursnit van die FPGA is ook gemeet. Dit kan gebruik word om die dinamiese kruisdeursnit van ontwerpe uit simulasie to bepaal.

Hierdie tesis illustreer hoe oopbron intellektuele eiendom en kommersiële komponente kombineer kan word om 'n robuuste produk vir die miniatuursatelliet-mark te ontwikkel.

Stellenbosch University http://scholar.sun.ac.za

# Acknowledgements

I would like to express my sincere gratitude to the following persons and institutions who made invaluable contributions to this project:

- My studyleaders, Prof Sias Mostert and Willem Smit, for their input, encouragement, ideas and feedback.
- iThemba LABS for allowing me the opportunity to perform radiation testing. I especially think of Mr. J. Symons and Mr. R. Mlambo for helping with the setup and execution of the experiments.
- Mr. J. Fabula and Mr. A. Lesea of Xilinx Corp, provided valuable feedback on the experimental results.
- Sunspace Satellite and Information Systems for their financial support.
- Lastly all the family and friends inside and outside the ESL, who supported, advised and encouraged.

Stellenbosch University http://scholar.sun.ac.za

# Contents

**Declaration** ii

**Abstract** iii

**Uittreksel** iv

**Acknowledgements** v

**Contents** vi

**List of Figures** xii

**List of Tables** xv

**Nomenclature** xvi

**1 Introduction** 1
- 1.1 Background . . . 1
- 1.2 Objectives . . . 2
- 1.3 Development tools . . . 3
- 1.4 Outline . . . 4

**2 Radiation in the Space Environment** 5
- 2.1 Types of radiation . . . 5
  - 2.1.1 Alpha ($\alpha$) radiation . . . 5
  - 2.1.2 Beta ($\beta$) radiation . . . 7
  - 2.1.3 Gamma ($\gamma$) radiation . . . 7
  - 2.1.4 Proton ($H^+$) radiation . . . 7
  - 2.1.5 Neutron radiation . . . 8
  - 2.1.6 X-rays . . . 9
- 2.2 Sources . . . 9
  - 2.2.1 The sun . . . 9
  - 2.2.2 Galactic cosmic rays (GCRs) . . . 9
  - 2.2.3 Van Allen belts . . . 10
  - 2.2.4 Secondary radiation . . . 10

Stellenbosch University http://scholar.sun.ac.za


2.3 Effects of radiation . . . . . 10
2.3.1 Single Event Effects (SEE) . . . . . 11
2.3.2 Displacement damage . . . . . 14
2.3.3 Total Ionising Dose Effects . . . . . 14
2.4 Conclusion . . . . . 14

**3 Soft-core Processor Selection 15**
3.1 GRLIB/LEON3 . . . . . 15
3.1.1 History . . . . . 16
3.1.2 Architecture . . . . . 16
3.1.3 Fault tolerant version . . . . . 17
3.1.4 License . . . . . 18
3.1.5 Performance . . . . . 19
3.1.6 Development Tools . . . . . 19
3.2 PicoBlaze . . . . . 21
3.2.1 History . . . . . 21
3.2.2 Architecture . . . . . 21
3.2.3 Performance . . . . . 22
3.2.4 License . . . . . 22
3.2.5 Development Tools . . . . . 22
3.3 Processor selection . . . . . 23
3.3.1 Application of error mitigation techniques . . . . . 23
3.3.2 Target FPGA . . . . . 23

**4 Fault Tolerance in FPGAs 26**
4.1 SEE effects on FPGAs . . . . . 26
4.1.1 Static cross-section . . . . . 26
4.1.2 Dynamic cross-section . . . . . 29
4.2 Physical tolerance . . . . . 30
4.2.1 Configuration memory . . . . . 30
4.2.2 Transistor level redundancy . . . . . 31
4.2.3 Manufacturing technology . . . . . 31
4.3 Error mitigation . . . . . 32
4.3.1 Spatial redundancy . . . . . 32
4.3.2 Temporal redundancy . . . . . 38
4.3.3 Data redundancy . . . . . 39
4.3.4 Error correction codes (ECC) . . . . . 39
4.3.5 Selective redundancy . . . . . 40
4.3.6 Granularity . . . . . 40
4.4 External protection . . . . . 41
4.4.1 Latch-up protection . . . . . 42
4.4.2 Configuration memory . . . . . 42

Stellenbosch University http://scholar.sun.ac.za


4.5 Soft-core processors . . . 42
4.5.1 Cache memory . . . 43
4.5.2 Checkpointing . . . 43
4.6 Implementation considerations . . . 43
4.6.1 FPGA resources . . . 44
4.6.2 Synthesis software . . . 44
4.7 Conclusion . . . 45

**5 Configuration Controller 48**
5.1 Target FPGA selection . . . 48
5.2 FPGA configuration . . . 49
5.2.1 Configuration modes . . . 49
5.2.2 Configuration speed . . . 50
5.2.3 Configuration file . . . 50
5.2.4 Readback verification . . . 53
5.3 High level design . . . 54
5.3.1 Requirements . . . 54
5.3.2 Possible designs . . . 54
5.4 Detailed design . . . 56
5.4.1 System voltage . . . 57
5.4.2 CPLD . . . 57
5.4.3 Microcontroller . . . 58
5.4.4 Non-volatile memory . . . 59
5.4.5 Interface to FPGA . . . 60
5.5 Programmable logic and software . . . 60
5.5.1 Erasing the flash memory . . . 61
5.5.2 Programming the flash memory . . . 62
5.5.3 CPLD registers . . . 63
5.5.4 FPGA configuration . . . 64
5.5.5 Readback verification . . . 66
5.5.6 Control software . . . 66
5.6 Conclusion . . . 67
5.6.1 Configuration controller . . . 67
5.6.2 Readback . . . 68
5.6.3 Limitations . . . 68

**6 Single Event Upset Simulator 69**
6.1 Background . . . 69
6.1.1 Device architecture . . . 70
6.1.2 Configuration memory . . . 70
6.1.3 Configuration bitmap . . . 71
6.2 Similar projects . . . 72

Stellenbosch University http://scholar.sun.ac.za


6.2.1 Los Alamos National Laboratory . . . 72
6.2.2 Brigham Young University . . . 74
6.2.3 Politechnica di Torino . . . 75
6.3 Error detection . . . 75
6.3.1 CRC monitoring . . . 75
6.3.2 Self-testing routine . . . 78
6.4 System description . . . 78
6.4.1 Control software . . . 79
6.4.2 PicoBlaze . . . 80
6.4.3 Post-processing . . . 83
6.5 Conclusion . . . 84
6.5.1 Speed . . . 84
6.5.2 Limitations . . . 84
6.5.3 Recommendations . . . 85

**7 Fault Tolerant PicoBlaze 86**
7.1 Designs . . . 86
7.1.1 Reference PicoBlaze (ISE) . . . 86
7.1.2 Reference PicoBlaze (Synplify Pro) . . . 90
7.1.3 High-level TMR (ISE) . . . 90
7.1.4 High-level TMR (Synplify Pro) . . . 92
7.1.5 High-level TMR (BUFTs) . . . 93
7.1.6 TMR flip-flops and protected program ROM . . . 94
7.1.7 Hamming-encoded memory and pins . . . 95
7.1.8 Final design . . . 99
7.2 Results . . . 101
7.3 Observations . . . 101
7.3.1 Upper bound on improvement . . . 102
7.3.2 Synthesis engine . . . 102
7.3.3 Majority voter implementation . . . 103
7.3.4 Redundant output pins . . . 103
7.3.5 Implementation considerations . . . 103
7.4 Conclusion . . . 104

**8 Radiation Testing 105**
8.1 iThemba LABS . . . 105
8.1.1 Accelerator operation . . . 105
8.2 Relevant previous tests . . . 106
8.2.1 Testing at iThemba LABS . . . 106
8.2.2 Rosetta experiment . . . 107
8.2.3 Brigham Young verification of SEU simulator . . . 107
8.3 Experimental design . . . 108

Stellenbosch University http://scholar.sun.ac.za


8.3.1 Tested designs . . . 108
8.3.2 Expected cross-section . . . 108
8.3.3 Experiment structure . . . 109
8.4 Experimental setup . . . 109
8.4.1 Test board . . . 110
8.4.2 Latch-up protection . . . 110
8.4.3 Communication link . . . 112
8.4.4 Readback software . . . 113
8.4.5 Configuration upset detection . . . 114
8.4.6 Control software . . . 114
8.4.7 Power . . . 115
8.4.8 Shielding . . . 115
8.4.9 Beam setup . . . 116
8.5 Results . . . 117
8.5.1 Initial cross-section measurements . . . 119
8.5.2 Latch-up . . . 120
8.5.3 Upset distribution . . . 120
8.5.4 Relative bit cross-sections . . . 121
8.5.5 Static cross-section . . . 122
8.5.6 Dynamic cross-section . . . 124
8.5.7 Comparison with SEU simulator . . . 127
8.5.8 Typical LEO failure rates . . . 128
8.6 Conclusion . . . 129

**9 Conclusions and recommendations 131**
9.1 Results . . . 131
9.1.1 Error mitigation . . . 131
9.1.2 SEU simulator . . . 132
9.1.3 Fault tolerant soft-core processor . . . 132
9.1.4 Radiation testing . . . 132
9.2 Recommendations . . . 133
9.2.1 Error mitigation . . . 133
9.2.2 Configuration controller . . . 133
9.2.3 SEU simulator . . . 133
9.2.4 Future radiation testing . . . 134
9.3 Final comments . . . 134

**Bibliography 135**

**A GNU General Public License 140**
A.1 The GNU General Public License . . . 140
A.2 The GNU Lesser General Public License . . . 140
A.3 Implications of using the GPL . . . 141

Stellenbosch University http://scholar.sun.ac.za


**B Selected source code excerpts** **142**
B.1 TMR using lookup tables . . . . . . . . . . 142
B.2 Tools . . . . . . . . . . 144
B.2.1 CRC calculator . . . . . . . . . . 144
B.2.2 Hamming memory generator . . . . . . . . . . 148
B.2.3 EDIF-level flip-flop insertion . . . . . . . . . . 151
B.2.4 Readback . . . . . . . . . . 153
B.2.5 Verification . . . . . . . . . . 155

**C Hardware schematics** **157**

**D Enclosed CD** **162**

Stellenbosch University http://scholar.sun.ac.za

# List of Figures

2.1 Typical orbits and radiation belts . . . . . 6
2.2 Dose-depth curves for the Explorer spacecraft. . . . . 6
2.3 High energy protons in the inner zone. . . . . 8
2.4 Mapping of errors showing the South Atlantic Anomaly . . . . . 13

3.1 LEON3 Integer unit pipeline. . . . . 17
3.2 LEON3 multiprocessor configuration . . . . . 18
3.3 PicoBlaze Embedded Microcontroller Block Diagram. . . . . 22

4.1 The set of possible upsets on Xilinx FPGAs . . . . . 27
4.2 Multiple Independent Redundant Transistors (MIRT) . . . . . 32
4.3 Triple modular redundancy with voting . . . . . 34
4.4 Redundant voters between sub-modules . . . . . 34
4.5 Voting inside the feedback loop. . . . . 35
4.6 Minority voters used to disable an output pin. . . . . 36
4.7 Sequential and combinatorial logic blocks in FTMR. . . . . 37
4.8 Time redundant D flip-flop . . . . . 38
4.9 8-bit word with parity bit. . . . . 39
4.10 Structure of a (12,8) Hamming code word. . . . . 40
4.11 TMR at different levels of granularity . . . . . 41

5.1 SelectMap configuration flow diagram . . . . . 51
5.2 Readback data verification flow. . . . . 52
5.3 Alignment of the readback data stream. . . . . 53
5.4 Microprocessor-centric configuration controller design. . . . . 55
5.5 Using a microprocessor and CPLD to configure a FPGA. . . . . 55
5.6 Block diagram of configuration controller. . . . . 56
5.7 Photo of S3Kit and configuration controller. . . . . 56
5.8 Usage of flash memory to store configuration and readback data. . . . . 60
5.9 3.3V configuration of a Spartan-3 device in slave-parallel mode. . . . . 61
5.10 Block diagram showing connections in the configuration controller. . . . . 62
5.11 Sequence diagram describing the flash erase operation. . . . . 63
5.12 Structure of packet used to program flash memory. . . . . 63
5.13 Sequence diagram for programming the flash memory. . . . . 64

Stellenbosch University http://scholar.sun.ac.za


5.14 Sequence diagram for configuring the FPGA. . . . . . . . . . . . . . . . . . . . 65
5.15 Diagram showing the connections for readback and verification on the CPLD 66

6.1 Spartan-3 family architecture. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
6.2 Annotated configuration memory bitmap for Spartan-3. . . . . . . . . . . . . 72
6.3 Annotated floor plan of design in Figure 6.2. . . . . . . . . . . . . . . . . . . . 73
6.4 SLAAC-1V reconfigurable computing board. . . . . . . . . . . . . . . . . . . . 74
6.5 16-bit CRC calculator with $(1 + x^2 + x^{15} + x^{16})$ polynomial. . . . . . . . . . . 76
6.6 Design file with added CRC calculator. . . . . . . . . . . . . . . . . . . . . . . 77
6.7 Block diagram of SEU simulator . . . . . . . . . . . . . . . . . . . . . . . . . . 79
6.8 Sequence diagram for a single error injection operation. . . . . . . . . . . . . 80

7.1 Block diagram of reference PicoBlaze design. . . . . . . . . . . . . . . . . . . . 88
7.2 Configuration bitmap for reference PicoBlaze with sensitive bits indicated. . 89
7.3 Majority voting on a bi-directional bus. . . . . . . . . . . . . . . . . . . . . . . 91
7.4 Configuration bitmap and sensitive bits for high-level TMR design. . . . . . 92
7.5 Majority voter using tri-state buffers . . . . . . . . . . . . . . . . . . . . . . . . 93
7.6 PicoBlaze with Hamming-encoded memory. . . . . . . . . . . . . . . . . . . . 98
7.7 Configuration bitmap and sensitive bits for design with Hamming-encoded memory. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
7.8 Block diagram of final PicoBlaze design. . . . . . . . . . . . . . . . . . . . . . . 99
7.9 Configuration bitmap and sensitive bits for final PicoBlaze design. . . . . . 100

8.1 Diagram showing solid pole cyclotron (SPC), with dee and dummy dee locations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
8.2 Radiation experiment setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
8.3 Latch-up protection circuit. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
8.4 Cross-section vs. effective LET curve for Virtex-II. . . . . . . . . . . . . . . . . 116
8.5 A photo of the setup in the proton treatment vault. . . . . . . . . . . . . . . . . 117
8.6 A photo of the FPGA in position in the proton treatment vault. . . . . . . . . 118
8.7 Close-up of the S3Kit and configuration controller. . . . . . . . . . . . . . . . . 118
8.8 Measured relationship between dose and number of upsets for static XC3S200.119
8.9 Distribution of upset locations for different designs. . . . . . . . . . . . . . . . 121
8.10 Configuration memory after reset,showing the BRAM initialised to 0x3FF . 122
8.11 Compensated cross-section measurements. . . . . . . . . . . . . . . . . . . . . 123
8.12 Measured cross-section for the tested device. . . . . . . . . . . . . . . . . . . . 123
8.13 Measured scaling factor for the reference design. . . . . . . . . . . . . . . . . . 125
8.14 Measured scaling factor for the design with high level TMR. . . . . . . . . . 126
8.15 Measured scaling factor for the final design. . . . . . . . . . . . . . . . . . . . . 126
8.16 Fluxes after shielding in 1000km orbit. . . . . . . . . . . . . . . . . . . . . . . . 129

C.1 Top level schematic of configuration controller . . . . . . . . . . . . . . . . . . 157
C.2 Microcontroller portion of configuration controller . . . . . . . . . . . . . . . . 158

Stellenbosch University http://scholar.sun.ac.za


C.3 CPLD and flash of configuration controller . . . . . . . . . . . . . . . . . . . . 159
C.4 Schematics for local transceiver. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 160
C.5 Schematics for remote transceiver. . . . . . . . . . . . . . . . . . . . . . . . . . . . 161

Stellenbosch University http://scholar.sun.ac.za

# List of Tables

3.1 PicoBlaze development tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

4.1 Relative size contributions to FPGA static cross-section. . . . . . . . . . . . . 27
4.2 Dual redundancy error detection and correction. . . . . . . . . . . . . . . . . . 33
4.3 Majority voter truth table . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

5.1 Spartan-3 configuration modes. . . . . . . . . . . . . . . . . . . . . . . . . . . 50

6.1 Spartan-3 XC3S200 Column Types. . . . . . . . . . . . . . . . . . . . . . . . . 71

7.1 (23,18) Hamming encoding matrix. . . . . . . . . . . . . . . . . . . . . . . . . 96
7.2 (23,18) Hamming error syndrome look-up table. . . . . . . . . . . . . . . . . . 97
7.3 Logic utilisation and performance of different PicoBlaze implementations. . 102
7.4 The scaling factor measured for different PicoBlaze implementations. . . . . 102

8.1 Measured pass rates and average number of upsets for tested PicoBlaze designs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
8.2 Measured scaling factors for tested PicoBlaze designs. . . . . . . . . . . . . . 126
8.3 Comparison of measured and simulated dynamic cross-sections. . . . . . . 127
8.4 Proton induced upset rates according to CREME96. . . . . . . . . . . . . . . 129
8.5 Expected failure rates for tested designs in space. . . . . . . . . . . . . . . . . 129
8.6 Static cross-section measurements for Spartan-3 XC3S200. . . . . . . . . . . . 130

Stellenbosch University http://scholar.sun.ac.za

# Nomenclature

### Symbols

| | |
|---|---|
| $\alpha$ | Static to dynamic cross-section scaling factor |
| $\Phi$ | Fluence (p/cm$^2$) |
| $N$ | Number of upsets |
| $\sigma$ | SEU Cross-section (cm$^2$) |
| $\sigma'$ | Dynamic SEU Cross-section (cm$^2$) |

### Acronyms and Abbreviations

| | |
|---|---|
| ADC | Analogue to Digital Converter |
| AHB | Advanced High-performance Bus |
| ALU | Arithmetic Logic Unit |
| AMBA | Advanced Microcontroller Bus Architecture |
| APB | Advanced Peripheral Bus |
| ASIC | Application Specific Integrated Circuit |
| BCH | Bose-Chaudhary-Hoquenghem |
| BJT | Bipolar Junction Transistor |
| BRAM | Block RAM |
| BUFT | Tri-state buffer |
| BYU | Brigham Young University |
| CCD | Charge-coupled device |
| CCSDS | Consultative Committee for Space Data Systems |
| CLB | Configurable Logic Block |
| CMOS | Complimentary Metal-Oxide Semiconductor |
| CPLD | Complex Programmable Logic Device |
| CREME96 | Cosmic Ray Effects on Micro-Electronics (1996 revision) |
| COTS | Commercial-Off-The-Shelf |
| CRC | Cyclic Redundancy Code |
| DLL | Delay Locked Loop |
| DCM | Digital Clock Manager |

Stellenbosch University http://scholar.sun.ac.za


| | |
|---|---|
| DR | Dual Redundancy |
| ECC | Error correction code |
| EEPROM | Electrically Eraseble Programmable Read-Only Memory |
| ESA | European Space Agency |
| FPGA | Field Programmable Gate Array |
| EDAC | Error Detection and Correction |
| EDIF | Electronic Design Interchange Format |
| FPU | Floating Point Unit |
| FTMR | Functional Triple Modular Redundancy |
| GCR | Galactic Cosmic Ray |
| GEO | Geostationary Orbit |
| GNU | GNU's Not Unix |
| GPL | GNU General Public License |
| HDL | Hardware Description Language |
| IDE | Integrated Development Environment |
| IOB | Input/Output Block |
| IP | Intellectual Property |
| ISR | Interrupt Service Routine |
| IU | Integer Unit |
| JPL | Jet Propulsion Laboratory |
| KCPSM | Konstant Coded Programmable State Machine |
| LANL | Los Alamos National Laboratory |
| LEO | Low Earth Orbit |
| LET | Linear Energy Tranfer |
| LFSR | Linear Feedback Shift Register |
| LGPL | GNU Lesser General Public License |
| LUT | Look-up Table |
| MBU | Multiple Bit Upset |
| MEO | Medium Earth Orbit |
| MIPS | Million Instructions Per Second |
| MIRT | Multiple Independent Redundant Transistors |
| MOS | Metal-Oxide Semiconductor |
| MOSFET | MOS Field Effect Transistor |
| NASA | National Aeronautics and Space Administration |
| NIEL | Non-ionising Energy Loss |
| PAR | Place And Route |

Stellenbosch University http://scholar.sun.ac.za


| | |
|---|---|
| PC | Program Counter |
| RAM | Random Access Memory |
| RF | Radio Frequency |
| RISC | Reduced Instruction Set Computing |
| ROM | Read-only Memory |
| SAA | South Atlantic Anomaly |
| SCP | Soft-Core Processor |
| SDRAM | Synchonous Dynamic RAM |
| SEB | Single Event Burnout |
| SED | Single Event Disturb |
| SEDR | Single Event Dielectric Rupture |
| SEE | Single Event Effect |
| SEFI | Single Event Functional Interrupt |
| SEGR | Single Event Gate Rupture |
| SEL | Single Event Latchup |
| SES | Single Event Snapback |
| SET | Single Event Transient |
| SEU | Single Event Upset |
| SHE | Single Hard Error |
| SMP | Symmetric Multi-processor |
| SoC | System on a Chip |
| SOI | Silicon on Insulator |
| SPARC | Scalable Processor Architecture |
| SPC | Solid-Pole Cyclotron |
| SSC | Seperated Sector Cyclotron |
| SRAM | Static RAM |
| S3Kit | Spartan-3 Starter Kit |
| TID | Total Ionising Dose |
| TMR | Triple Modular Redundancy |
| TQFP | Thin Quad Flat Pack |
| TTL | Transisitor Transistor Logic |
| UART | Universal Asynchronous Receiver/Transmitter |
| UML | Unified Modelling Language |
| VHDL | VHSIC Hardware Description Language |
| VGA | Video Graphics Array |
| VHSIC | Very High-Speed Integrated Circuit |
| XTMR | Xilinx TMR |

Stellenbosch University http://scholar.sun.ac.za

# Chapter 1

# Introduction

## 1.1 Background

The production of miniature spacecraft has become an increasingly important segment in the space market over the last few decades. The more modest on-orbit abilities of these spacecraft require a much simpler system architecture than conventional products, which lowers the cost of the miniature system significantly. These systems are characterised by the use of new technologies and commercial devices to improve performance and to keep costs down. Their lower cost makes these satellites much more accessible to developing countries and academic institutions.[1]

The University of Stellenbosch and South Africa entered this market with the launch of SUNSAT in 1999. The resulting commercialisation of the satellite technology has been accompanied by continous research into new technologies.

High-density field programmable gate arrays (FPGAs) are one example of a technology that holds much potential for application in the miniature satellite market. The devices allow users to implement custom logic functions at a very low cost. The availability of devices with millions of gates allows several components to be implemented on a single device, leading to higher integration and lower device count. FPGAs that use SRAM technology for their configuration memory (SRAM-based FPGAs) also offer further flexibility and adaptability by allowing on-orbit reconfiguration to take into account changing mission parameters, environmental effects and component failure. However, these devices are susceptible to radiation induced bit flips which decreases the reliability of the design. Radiation induced single event upsets pose a significant threat to the operation of soft-core processors, but can be mitigated to a large extent through specific design techniques.

The recent explosion in the popularity of open-source software has provided the means to quickly develop a high value product, based on existing intellectual property. The open-source concept has also migrated to the world of programmable logic where it

Stellenbosch University http://scholar.sun.ac.za


speeds up development and adds value to the hardware offerings of FPGA vendors. One interesting phenomenon has been the development of open-source soft-core processors: processors that are implemented in programmable logic.

These processors combine the ease of use of a conventional processor with the power of parallel execution on FPGAs. The use of a soft-core processor allows extensive modifications to suit specific applications. An additional benefit is that the processor architecture will not be discontinued by a manufacturer, because it can easily be migrated to another platform; recompiling is often enough. The use of such a processor also benefits from all the integration and adaptability features of the FPGA it is implemented on.

The convergence of these three elements forms the background to this study: utilising the power of open-source soft-core processors on high density FPGAs for space applications.

## 1.2 Objectives

The primary objective of this thesis was the development of a single event upset tolerant implementation of a soft-core processor.

This was achieved by evaluating various fault tolerance techniques to derive guidelines for their application to SRAM FPGAs. The protection afforded by the different techniques was measured by using a specially designed SEU simulator that mimics upset in the configuration memory by error injection. The best techniques were combined to developed a hardened PicoBlaze design. This design was tested by proton irradiation.

This entailed the following:

- A study of the radiation environment experience by low-earth orbit satellites. The manner in which the different types of radiation affects electronics had to be understood before mitigation techniques could be applied.
- An open source soft-core processor was selected. The LEON3 from Gaisler Research and the Xilinx PicoBlaze were investigated. The PicoBlaze was selected because it allowed easier modification for testing different mitigation strategies, hardware that supports it could easily be found. A small microcontroller was also deemed to be more suitable for the space environment than the larger LEON3. SRAM-based FPGAs were used as implementation platform, because the upset modes exhibited by flash and anti-fuse technologies can generally be seen as a subset of what SRAM FPGAs display.
- Viable fault tolerance techniques were identified so that guidelines for implementing error mitigation techniques on SRAM FPGAs could be proposed. A

Stellenbosch University http://scholar.sun.ac.za


description of the static and dynamic cross-sections was developed. This description was used throughout this thesis to compare the effectiveness of mitigation techniques.

- A configuration controller was designed and built. This allowed the evaluation of the advanced configuration and readback functions of Xilinx FPGAs. A hardware-based error injection tool-set that mimics the effect of SEUs in the configuration memory was implemented using the configuration controller. This SEU simulator uses a self-checking routine on the target FPGA to determine if corruption of the configuration memory influences a design.
- A radiation hardened version of the selected PicoBlaze soft-core processor was developed. Different error mitigation techniques were evaluated by simulating several mitigation techniques in the SEU simulator. The best of these techniques were combined to produce an SEU tolerant design.
- Selected designs were tested by proton radiation. This verified the developed fault tolerant design and the SEU simulator and provided information on the radiation characteristics of the Spartan-3 SRAM based FPGA.

Extensive programming was required for PC and embedded microcontroller targets. A large amount of programmable logic was also developed. Hardware was designed and constructed for the configuration controller. Additional hardware also had to be designed to make radiation testing possible.

## 1.3 Development tools

All development was done with `Fedora Core 3 Linux` as operating system. This environment was selected, because it is freely available, offers powerful tools for software development, and is very similar to Red Hat Enterprise Linux 3, which is officially supported by most electronic design tools.

Synthesis was performed using `Synplify Pro` from Synplicity – it is fast, powerful and delivers good results. The software is also available to universities at a fraction of the cost of a commercial license, which allowed the use of this industry standard application. `Xilinx Webpack`, a free version of the `Xilinx ISE` software, was used to place and route the results from synthesis. It also provided hardware configuration tools. HDL simulation was done using `Modeltech Modelsim`.

`Altium DXP` was used for printed circuit board layout.

Open source software was used where possible. The ATmega128 was programmed using `C` and the open source `avr-gcc` compiler. The device was configured using the open source `avrdude` software. Xilinx Spartan-3 programming software, `xc3sprog`, was modified to perform readback of the FPGA during radiation testing.

Stellenbosch University http://scholar.sun.ac.za


All other control, testing and data processing was done using the `Python` and `C` programming languages, with open source interpreters and compilers.

## 1.4 Outline

This document is structured as follows:

- Chapter 1 introduces the subject of this thesis and outlines the main objectives.
- Chapter 2 provides background on the sources and effects of radiation in the low earth orbit environment.
- Chapter 3 discusses the considered soft-core processors and the selection of the Xilinx PicoBlaze.
- Chapter 4 details the application of error mitigation techniques to SRAM FPGAs.
- Chapter 5 documents the design of the configuration controller hardware.
- Chapter 6 provides information about the functioning of the SEU injection toolset.
- Chapter 7 discusses the designs that were simulated on the SEU simulator to evaluate error mitigation techniques. The best techniques were combined to implement a fault tolerant PicoBlaze.
- Chapter 8 presents the radiation testing that was performed to verify the fault tolerant PicoBlaze and the SEU simulator. It also documents the cross-section measurements made during testing.
- The conclusion is presented in Chapter 9. The results of the various components of this thesis is summarised.

Stellenbosch University http://scholar.sun.ac.za

# Chapter 2

# Radiation in the Space Environment

A knowledge of the different forms of radiation and how they affect electronics is a prerequisite for understanding error mitigation strategies.

## 2.1 Types of radiation

"Radiation" is generally used to describe the emission of energy as electromagnetic waves or particles. A qualifier (eg. alpha radiation) is used to distinguish between the different types and sources.

A satellite will encounter different types of radiation, depending on its orbit. The level of interaction of the types varies depending on the particles involved, their ionisation ability and energy levels. Figure 2.1 relates the major radiation zones and typical spacecraft orbits, while 2.2 indications of the relative intensities of radiation typically found in a low earth orbit (below 1000km).

Radiation can be generally divided into two classes: *ionising* and *non-ionising* radiation. Ionising radiation tends to cause the formation of ions (by stripping electrons from or adding electrons to the atom). Non-ionising radiation causes structural damage by changing the crystal lattice.

The following are the main types of radiation in a space environment.[4][5][3]

### 2.1.1 Alpha ($\alpha$) radiation

These charged nuclei of Helium atoms ($He^{++}$) are highly ionising, but do not pose a major threat, as they rapidly lose energy due to ionisation, which limits their penetration ability.

When unstable heavy atomic nuclei decay, one of the by-products is $\alpha$ radiation. The primary source in the low earth orbit (LEO) environment is secondary radiation from

Stellenbosch University http://scholar.sun.ac.za

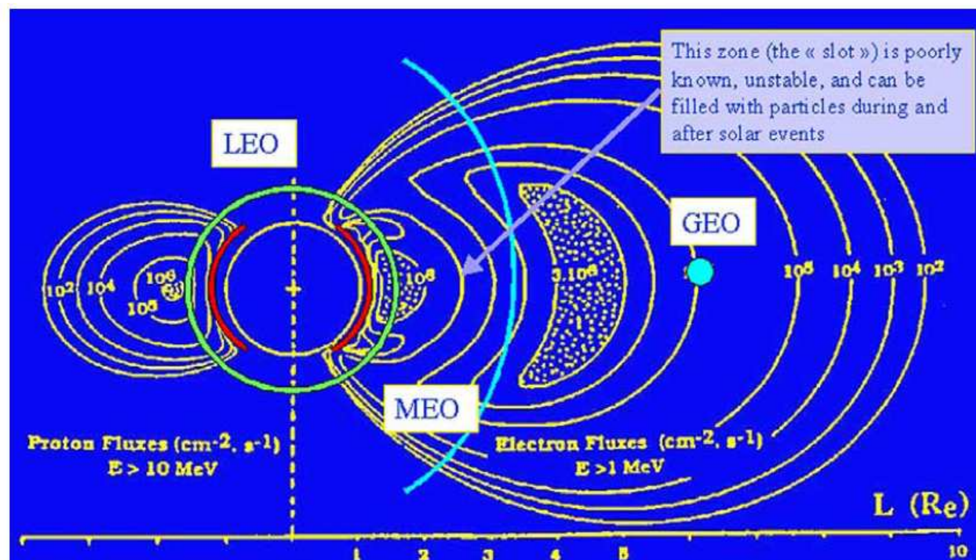




**Figure 2.1:** Typical orbits and radiation belts. Low earth orbit (LEO) satellites can be seen to pass through areas of high proton and electron flux. Spacecraft in medium earth orbit (MEO) or in a geostationary orbit (GEO) will primarily encounter a high electron flux due to the radiation belts.[2].

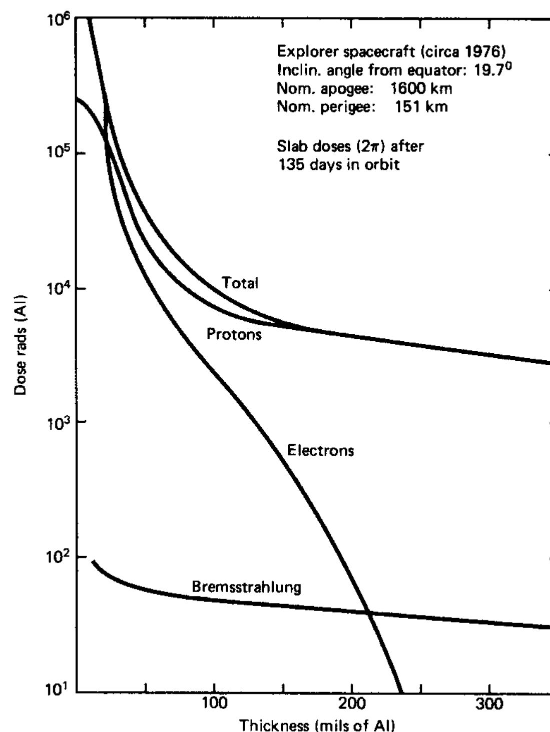



**Figure 2.2:** Dose-depth curves for the Explorer spacecraft, showing the penetration relationship between the various types of radiation. The graph shows the effect of shielding against the major radiation components. Even a thin layer of aluminium (<50 mils) causes a dramatic fall-off in the total ionising dose. The contribution made by proton radiation dominates the total dose.[3]

Stellenbosch University http://scholar.sun.ac.za


interactions with galactic cosmic rays, although trace amounts of heavy elements in chip packaging also contribute a small amount.

### 2.1.2 Beta ($\beta$) radiation

The number of protons in a nucleus is changed by one when a particle undergoes $\beta$ decay. As a result an electron and antineutrino ($\overline{\nu}$) or positron and neutrino ($\nu$) are emitted:

$$ {}^{N}_{Z}\mathrm{A} \rightarrow {}_{Z-1}^{\ \ \ N}\mathrm{B} + e^{-} + \overline{\nu} \quad (2.1) $$

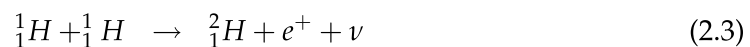
or

$$ {}^{N}_{Z}\mathrm{A} \rightarrow {}_{Z-1}^{\ \ \ N}\mathrm{B} + e^{+} + \nu \quad (2.2) $$

These high energy electrons are known as $\beta$ particles. They are relatively easily deflected because of their low mass and consequently have a low penetration ability. $\beta$ particles pose a negligible threat to electronics.

### 2.1.3 Gamma ($\gamma$) radiation

A particle undergoing radioactive decay is often left in an excited state. In order to reach a lower energy state a high energy electron (a $\gamma$ particle) is emitted. These particles have a very high energy (1 MeV to 1 GeV) in comparison to visible light (about 1 eV). The sun is a major source, with $\gamma$ rays produced by the fusion of hydrogen and deuterium, as shown below.



$$ {}^{1}_{1}H + {}^{1}_{1}H \rightarrow {}^{2}_{1}H + e^{+} + \nu \quad (2.3) $$

$$ {}^{1}_{1}H + {}^{2}_{1}H \rightarrow {}^{3}_{2}He + \gamma \quad (2.4) $$

Gamma rays primarily contribute to the total ionising dose (TID) accumulated over months and years in orbit. It results in the degradation in performance and the eventual failure of electronic components.

### 2.1.4 Proton ($H^{+}$) radiation

Proton radiation is simply hydrogen nuclei ($H^{+}$) moving at high speed. They are the primary concern in LEO systems due to their high mass, high ionising potential, as well as their relative abundance close to the earth.

Proton fluence can be predicted using models such as AP8MIN or AP8MAX which respectively provide the fluence for solar minimum and solar maximum periods. Pro-

Stellenbosch University http://scholar.sun.ac.za


ton densities decrease at solar maximum due to collisions of trapped protons with the atmosphere. Solar minimum models should therefore be used in mission planning.[1]

Figure 2.3 shows proton flux in the region of LEO satellites.

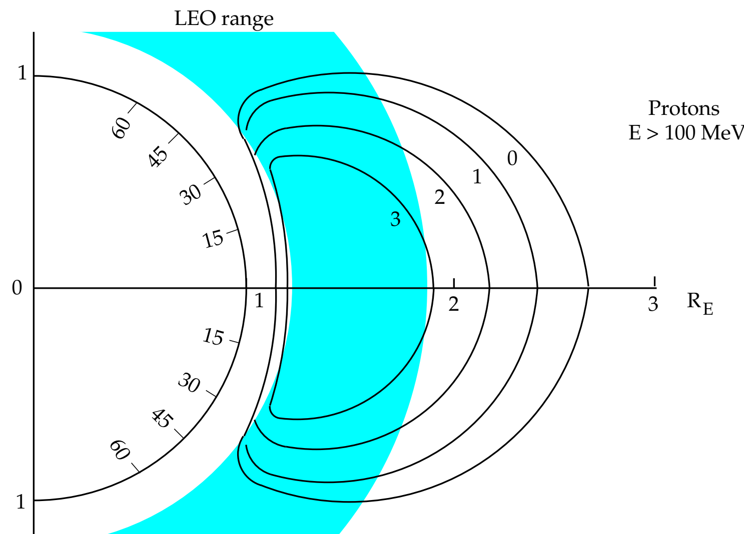



**Figure 2.3:** High energy protons in the inner zone as predicted by the AP8MIN model. The numbers on the contours represent the $\log_{10}$ of the integral flux in protons $cm^{-2}/s$. The horizontal axis is the magnetic equator marked in earth radii. Only protons with energies above 100MeV are considered. The approximate range of low-earth orbit satellites has been shaded. (Adapted from [1])

The linear energy transfer (LET) of protons is too low to directly induce single event effects (SEEs), unlike energetic ions. Instead, protons may undergo elastic or inelastic nuclear interactions with the target and transfer energy to the recoil atoms. Their greater mass raises their LET, causing SEEs. A proton-induced SEE is consequently an indirect radiation effect.

### 2.1.5 Neutron radiation

Neutrons are the primary source of non-ionising radiation. They are produced by radioactive decay and nuclear reactions and cause *displacement damage* and *secondary radiation*.

The collision of neutrons with a semiconductor can cause the atoms in the crystal lattice to be moved; this is referred to as displacement damage. It alters the minority carrier densities which causes the minority carrier lifetimes to decrease. The effect on the current gain of bipolar transistors is significant because they are minority carrier devices. MOS transistors, being majority carrier devices, are relatively unaffected. Therfore neutron radiation is a greater concern in bipolar circuits.

Stellenbosch University http://scholar.sun.ac.za


Although neutrons are non-ionising they can cause ionisation through secondary radiation. This can either be by producing ionising recoil atoms or ions, or by exciting atomic nuclei which will de-excite by emitting ionising gamma rays. Alternatively, the neutron can be absorbed by the target nucleus, which, in the case of silicon, decays by emitting an $\alpha$ particle or a proton.

### 2.1.6 X-rays

X-rays are photons with a particular energy and wavelength; similar to $\gamma$-rays.

*Characteristic x-rays* are emitted when an electron falls into a vacancy in the n=1 or n=2 levels of an atom. The vacancies themselves are normally created when electrons are knocked from these levels by high energy electrons. The energy of these particles relate to the difference in energy between the levels and are therefore discrete.

*Bremstralung* is caused when a target is bombarded by electrons. The deceleration of the moving charges causes radiation that has a continuous energy spectrum.

The particles are ionising but the ambient intensity in a LEO environment is low enough for it not to be considered a major threat. [3]

## 2.2 Sources

### 2.2.1 The sun

The sun is the dominant source of radiation in the solar system.

Under normal conditions fusion in the sun produces the solar wind, which is a relatively constant stream of low energy (keV) protons and high energy electrons. During solar flares a high energy (MeV) flux of electrons is emitted.

Solar flares are the result of periodic variations in the nuclear reactions on the chromosphere of the sun. Large quantities of particles are released, consisting mainly of protons ($> 90\%$), as well as alpha particles and heavy ions. The flux of heavy ions is usually far below the nominal galactic background radiation levels but it can be up to four orders of magnitude larger during some flares.

### 2.2.2 Galactic cosmic rays (GCRs)

GCRs are high speed, high energy particles primarily created by fusion reactions in distant stars. Measurements have shown that they consist of approximately 85% proton radiation, 14% alpha particles and 1% heavy ions. Particle energy can range from 0 to 10GeV. Hydrogen, carbon and oxygen form the bulk of the heavy ions, with energies

Stellenbosch University http://scholar.sun.ac.za


around 1GeV. GCRs can cause direct ionisation, similar to protons. Alternatively, a direct hit causes a shower of particles and secondary radiation.

GCRs are considered extremely disruptive and difficult to effectively shield against. Fortunately, the magnetic field of the earth provides a large measure of protection for LEO orbits, except for the areas over the poles.

They are the primary concern for satellites at higher altitudes (medium earth orbit and above).

### 2.2.3 Van Allen belts

The Van Allen belts consist of geomagnetically trapped electrons, protons and low quantities of heavy ions. They occupy a toroidal volume of space with an altitude of about 300km to 59000km.

The particle distribution can be divided into an inner zone and an outer zone at approximately 2.5 earth radii. The flux of electrons in the outer zone is about 10 times greater than in the inner zone. Electron energy levels in the outer zone are around 7 MeV, while it is less than 5 MeV in the inner zones.

Proton energies vary approximately inversely proportional to altitude. Energy levels greater than 400MeV is possible close to earth. Protons form the most significant radiation component in the inner belts and therefore also in LEO orbits.

The Earth's magnetic field is offset by approximitely 11° from the Earth's axis in the southern hemisphere. The displacement is towards the Western Pacific, which results in a dip in the magnetic field above the Atlantic Ocean. LEO spacecraft travelling through this area will pass through the Van Allen belts and consequently experience a dramatically higher fluence of trapped particles, as is shown in Figure 2.4.

### 2.2.4 Secondary radiation

Nuclear interaction, especially with neutrons and GRCs, can create unstable atomic isotopes. These isotopes decay over time and produce gamma rays.

This is more often observed on materials with a high atomic number - for instance traces of heavy elements in chip packaging and solder.

## 2.3 Effects of radiation

When considering the effects of radiation, a distinction is made between *hard errors*; that cause permanent damage to the device, *soft errors*; that cause a loss of state (ie.

Stellenbosch University http://scholar.sun.ac.za


information) without damaging the device and *firm errors*; that do not cause permanent damage but persist until the device is reset or reconfigured.

### 2.3.1 Single Event Effects (SEE)

Single event effects are caused by the impact of a single particle on a material depositing sufficient energy to cause an effect on the device. This can happen either through the prime strike (e.g., direct ionisation via GCRs) or by the secondary particles that are caused by the strike (e.g., indirect ionisation via protons).

Devices are characterised by their ability to withstand particles of different linear energy transfer (LET) level, which is measured in $\text{MeV/cm}^2\text{/mg}$. This is usually expressed for silicon, as it makes up the majority of a semiconductor device.

The effects of the SEE depends on the architecture of the device, as well as the location where the event occurrs.

The following events are of prime importance:

#### Single event upsets (SEU)

SEUs are bit flips in registers, latches, the on-chip RAM or the configuration memory of SRAM FPGAs. Enough charge is deposited to change the stored logical value from 0 to 1, or 1 to 0.

SEUs are typically non destructive, they are therefore classified as *soft errors*. They can be corrected by simply updating the corrupted bit.

The relatively high proton flux makes SEUs the most prevalent SEE in SRAM FPGA and memory devices, in the LEO environment.

#### Multiple bit upsets (MBU)

MBUs are similar to SEUs, except that multiple memory elements are upset simultaneously. High density SRAM and SRAM based FPGAs are susceptible.

#### Single event functional interrupt (SEFI)

SEFI describes a sudden loss of normal operation, due to an SEE in a sensitive part of the device circuitry.

This is observed in complex devices with built-in state and control circuitry, such as FPGAs and microcontrollers. It is usually not fatal, but requires the device to be reset to recover.

Stellenbosch University http://scholar.sun.ac.za


#### Single event transients (SET)

SETs are caused by charged particles depositing (or removing) charge on a circuit element, which leads to an incorrect logic value.

In combinatorial logic, the charge will leak away (over several hundreds of picoseconds) and the system will return to a consistent state. The only danger therefore exists when synchronous logic is disturbed on a clock edge and the temporarily incorrect value is latched into a register. The incorrect value can then propagate through the rest of the system.

The sensitivity of a system to SETs increases in nanometer technologies, at high clock speeds and in low voltage systems.

#### Single event disturb (SED)

SED involves a momentary corruption in the value of a bit.

This has been observed in combinatorial logic and especially the latches in electronic devices.

#### Single hard error (SHE)

SHE is an unalterable change in the state of a memory element (a stuck bit).

It can occurs in memories and latches in logic devices.

#### Single event latch-up (SEL)

SEL is a hard error that occurs when interaction with a high energy particle triggers parasitic thyristors in CMOS devices.

This creates a short circuit, which is capable of damaging the device by thermal effect. Not all SELs are fatal - in SRAM FPGAs SEL can also lead to corruption of the configuration data, which requires reconfiguration of the device to restore the correct functionality.

SELs are an important concern in any space-borne system, because of the potentially fatal effects of latch-up.

#### Single event snapback (SES)

SES is similar to SEL, and results in high current in N-channel MOSFET and SOI devices. It is also potentially destructive.

Stellenbosch University http://scholar.sun.ac.za


### Single event gate rupture (SEGR)

SEGR has been observed when heavy ions hit power MOSFETs while a large bias voltage is applied to the gate. This leads to thermal breakdown and gate rupture.

Flash-based systems can be at risk during programming when a relatively large voltage is applied to the storage elements.

SEGR has a low probability of happening, but very little can be done to protect against it, other than considering the device characteristics during device selection.

### Single event dielectric rupture (SEDR)

SEDR is similar to SEGR, in that it results when the gate dielectric is ruptured. It can occur in non-volatile NMOS structures and non-volatile FPGAs.

### Single event burnout (SEB)

SEBs may occur in N-channel power MOSFETs and BJTs when the impact of a heavy ion forward biases the thin base region under the source of the device. If the drain-source voltage exceeds the local breakdown voltage of the transistor, the device can burn out due to large currents and high local power dissipation.

SEB, like SEGR, is unlikely, but should be taken into account during the component selection process.

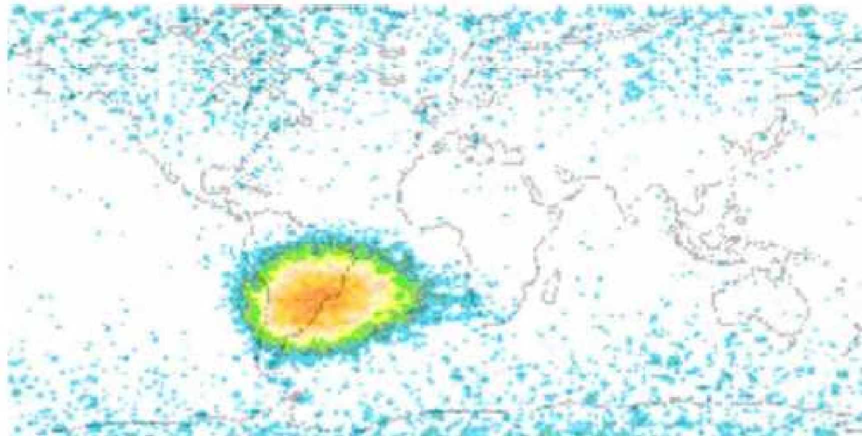


**Figure 2.4:** Mapping of data error density (in log scale) for 28 memories on-board ICARE experiment (Nov 2000 - Sept 2002). Dots show the locations were SEUs occurred; the high error rate in the South Atlantic Anomaly (SAA) is clearly visible. [2]

Stellenbosch University http://scholar.sun.ac.za


### 2.3.2 Displacement damage

The deposition of a non-ionising dose by protons and high energy $\alpha$ particles leads to lattice defects due to displacement of atoms in the structure.

This can be critical in devices where electrical parameters and background noise directly impacts on performance, such as CCDs, sensors and amplifiers.[2]

### 2.3.3 Total Ionising Dose Effects

TID is a measure of the cumulative effects of prolonged exposure to radiation, causing degradation in device performance.

The main sources are Solar Energetic Particle Events; which usually occur in association with solar flares, as well as trapped radiation in the South Atlantic Anomaly (SAA); where the Earth's magnetosphere dips closest to the earth, causing more trapped radiation. (See Figure 2.4.)

TID causes threshold shifts, increased device leakage (higher power consumption), timing changes, decreased functionality and ultimately device failure.[6]

Accelerated TID testing of devices can be done using a Cobalt 60 source. Although the source generates primarily $\gamma$-rays, the cumulative radiation effects are similar to those experienced in space.

Electronic devices can also display *annealing*. This refers to the total or partial self-healing that can be observed on a device after irradiation.

## 2.4 Conclusion

Of the various radiation effects discussed in this chapter, proton-induced SEUs form the most significant threat to LEO spacecraft.

The mitigation techniques discussed and implemented in later chapters are therefore aimed specifically at mitigating SEUs.

Stellenbosch University http://scholar.sun.ac.za

# Chapter 3

# Soft-core Processor Selection

With the advent of larger and faster FPGAs, the possibility of implementing a microprocessor in programmable logic has become more feasible. These soft-core processors (SCPs) combine the flexibility of software with the power of a parallelised design.

Their greatest strength lies in the high level of integration they offer, making true system-on-a-chip (SoC) design possible. By allowing customisation of the architecture, high performance can be achieved for specific tasks, while maintaining the ease of use that conventional processors offer. In addition, all the adaptive possibilities of reconfigurable FPGAs are applicable when an SCP is implemented on them. All these features make them highly attractive for the low-volume space market.[7]

Several high performance SCP cores are available. The use of this open-source intellectual property (IP) allows one to quickly and easily develop a high value product.

Two soft-core processors, the LEON3 from Gaisler Research and the Xilinx PicoBlaze, were investigated in detail to select a demonstration platform for SEU mitigation techniques.

## 3.1 GRLIB/LEON3

The LEON processor is an open source, soft-core processor developed and supported by Gaisler Research[1].

It is implemented as a high level VHDL model which is fully synthesisable with common synthesis tools. The model is extensively configurable through a graphical configuration package, which allows options such as cache size and organisation, arithmetic operation implementation, I/O modules and other IP cores to be selected.

[1]www.gaisler.com

Stellenbosch University http://scholar.sun.ac.za


### 3.1.1 History

The LEON processor was originally developed by Jiri Gaisler[8], while at the European Space Agency, in order to provide a high performance fault tolerant processor that can be implemented on non-radiation-hardened commercial-off-the-shelf (COTS) components. The source code of a version without fault tolerance was subsequently released under the GNU Public License[9] (GPL) to stimulate research and development using the processor.

LEON2 was released in 2002, with improved arithmetic support, and has since become the standard LEON implementation used in most applications[10].

LEON3 was initially released in 2004 and provides some extensions to the architecture of LEON2. It is provided as part of GRLIB[11], an open source library of IP cores.

### 3.1.2 Architecture

#### Features

All members of the LEON processor family are SPARC V8 (32 bit)[12] compliant processors, with separate instruction and data caches (Harvard architecture), an interrupt controller, on chip debugging support, timers and UARTS. Ethernet MAC and PCI interface IP is provided, as well as support for a separate co-processor.

LEON3 implements a 7 stage integer unit (IU) with hardware multiplication and division (see Figure 3.1). An optional interface to the Meiko floating point unit that supports serial access is also available. Alternatively, the high performance GRFPU[14] floating point unit, which can execute instructions in parallel with the IU can be used. If no floating point hardware is available, the calculations can be emulated at lower performance in software.

The SPARC V8E[15] extensions for embedded and real time systems are implemented and up to 4 multi-processors are supported. (A 4 processor system should fit into a 3000000 gate Xilinx XC2V3000 FPGA and operate at 80MHz.) LEON3 is distributed with a large library that is also released under the GPL. The libray includes IP cores for a CAN controller and CCSDS telemetry and telecommand functions.[13]

#### Structure

The processor is built around the ARM AMBA[16] bus specification and consists of configurable modules that can, if desired, be compiled into the system.

The core processor units are attached to the AHB (Advanced High-performance Bus), through which almost all communication takes place. Peripheral components are attached to the lower speed APB (Advanced Peripheral Bus), which connects to the AHB

Stellenbosch University http://scholar.sun.ac.za

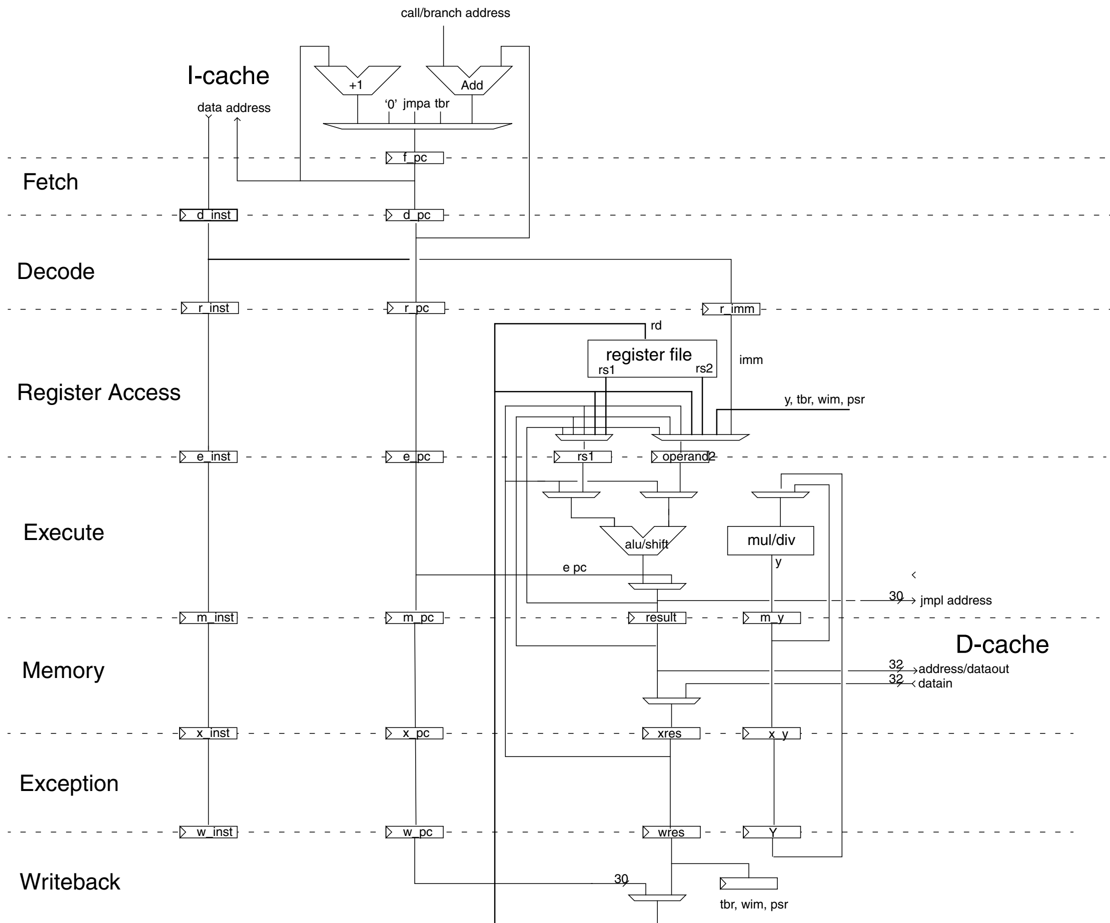




**Figure 3.1:** LEON3 Integer unit pipeline.[13]

through an AHB/APB bridge. Every module has a unique ID, which the AHB controller uses to arbitrate access to the bus.[17]

This architecture allows modules to be added or removed with minimum impact on the other modules. Multiprocessor LEON3 systems simply have more than one processor core on the AHB, as shown in Figure 3.2.

### 3.1.3 Fault tolerant version

All the LEON processors are also available in a fault-tolerant version, under a commercial IP license.

One of the major goals of these processors is to allow fault tolerant processors to be built using COTS hardware and software components, drastically reducing the system cost.[8] The system can be implemented in an FPGA with latch-up protection or resistance, like an Actel antifuse FPGA or a Xilinx XQVR device.

Stellenbosch University http://scholar.sun.ac.za

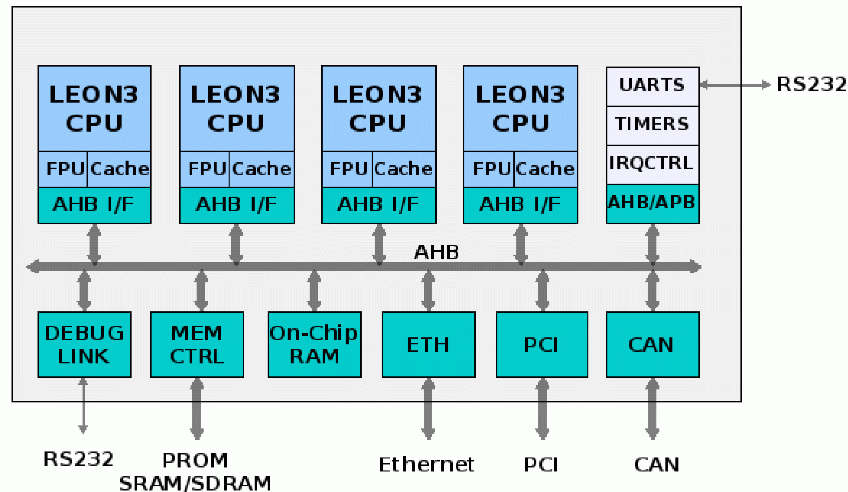




**Figure 3.2:** LEON3 multiprocessor configuration showing the bus-centric design, with all the components connected the AHB bus.[13]

#### LEON 1/2

These processors tolerate SEU errors by using triple redundant registers and on-chip error detection and correction codes (EDAC). Pipeline flushing and forced cache missing is used to clear SEUs without interrupting the processor.

The overhead of the fault tolerant implementation is about 100% of the original design.

The first prototypes were implemented on the Atmel ATC35 0.35$\mu$m process and performed successfully under heavy ions testing. All injected SEU errors were corrected without impacting the functioning of the system. The actual device threshold was measured at 6 MeV, while ion energies of up to 110 MeV were used in testing.

#### LEON 3

A fault tolerant version of the LEON3-FT was released in 2006. The application appears to be targeted at the rad-hard Actel RTAX antifuse FPGAs, with only the registers and memory being protected.

The overhead of the fault tolerant implementation is about 15% of the original design.

### 3.1.4 License

A brief discussion of the GPL is provided in Appendix A.

Stellenbosch University http://scholar.sun.ac.za


#### GRLIB/LEON3

The GRLIB source code is freely available, protected by the GNU GPL license. This means that designs based on GRLIB have to be distributed in full source under the same license. For commercial applications, where source distribution is not possible or not desirable, a commercial IP license can be obtained from Gaisler Research.[11]

Correspondence with Gaisler Research showed that the commercial licenses are normally granted on a project basis, which allows users to modify the source for a specific application. Other licensing schemes can also be negotiated.

#### LEON1 and LEON2

The LEON VHDL model is provided under two licenses: the GNU Public License (GPL)[9] and the Lesser GNU Public License (LGPL)[18]. The LGPL applies to the LEON model itself while remaining support files and test benches are provided under GPL.

This means that the LEON can be used as a core in a system-on-chip design without having to publish the source code of any additional IP-cores one might use. One must however publish any modifications that have been made to the LEON core itself, as described in LGPL. [19]

### 3.1.5 Performance

The core of the LEON processor requires approximately 30 000 gates (5 000 LUTs) plus RAM. This can go up to 100k-250k gates when other IP cores are included.

The LEON executes instructions at about 0.85 Dhrystone MIPS/MHz. [20][21]

Power consumption is largely dependent on the operating frequency, as well as the technology the design is implemented on.

### 3.1.6 Development Tools

GRLIB/LEON is designed to be used on a wide variety of hosts. As such, the provided scripts should work on any recent GNU/Linux or Solaris distribution. Microsoft Windows and Cygwin can also be used as a development platform.

#### Operating Systems

Several operating systems have been ported to the LEON processor.

Stellenbosch University http://scholar.sun.ac.za


The Real Time Executive for Multiprocessor Systems (RTEMS)[22] appears to be the most popular operating system, whether or not real-time scheduling is required. It was originally developed as a missile operating system by OAR Corporation, but has found applications in many other cases where a real-time system with multiprocessor support is needed.[23] RTEMS is released under a modified version of the GPL, with no restrictions on use.

A port of Snapgear Linux to the LEON architecture is provided by Gaisler Research and the LION development community[24]. Versions supporting memory management (MMU) and non-MMU systems are available, but no multiprocessor configurations are supported.

eCos is an open source, royalty-free, real-time operating system intended for embedded applications.[25] The highly configurable nature of eCos allows the operating system to be customised according to precise application requirements. This delivers the best possible run-time performance and an optimised hardware resource footprint. The symmetric multi-processor (SMP) functionality allows up to 16 LEON3 processors to run a single (multi-threaded) application. It has been tested with up to four processors, with good results.

VxWorks and Aonix ADA are the supported commercial operating systems.

#### Compilers

Gaisler Research provides the LEON Bare-C Cross Compiler System. It is based one the GNU compiler tools and the Newlib standalone C-library. The cross-compiler system allows compilation of sequential (non-tasking) C and C++ applications. It supports both hard and soft floating-point operations, as well as both V7 and V8 multiply and divide instructions.[26]

The RTEMS Cross-Compiler System is also available for systems running RTEMS.[27]

#### Related Software

TSIM is an instruction level simulator that offers high performance simulation of LEON systems.[28]

GRMON is a general purpose debugging monitor for the LEON debug support unit (DSU).[29] It improves upon and replaces the original DSUMON[30] debugging program. TSIM can be used as back-end, or on-chip debugging can be done using the Debug Support Unit (DSU) of the LEON.

Free evaluation versions of these programs are available.

Stellenbosch University http://scholar.sun.ac.za


## 3.2 PicoBlaze

The Xilinx PicoBlaze is a HDL defined microcontroller core that is optimised for Xilinx FPGAs.[31] It is extremely compact and offers a cost-effective control and simple data processing solution.

### 3.2.1 History

The PicoBlaze was originally written by Ken Chapman at Xilinx Corporation, as a simple, high performance IP core targeted at Xilinx FPGAs. The PicoBlaze complements the higher performance 32-bit MicroBlaze processor by providing a simple programmable state machine that requires very little logic resources.

The design was originally developed for Virtex, Virtex-E and Spartan-II devices, but versions supporting the CoolRunner-II, Spartan-3, Virtex-II and Virtex-4 have since been released. The KCPSM3, aimed at Spartan-3, Virtex-II and Virtex-4 devices, was used in this study.

### 3.2.2 Architecture

The PicoBlaze is an 8-bit RISC processor with 16 byte-wide general purpose registers. Instructions are executed from memory within the FPGA, for the Spartan-3 version this can be up to 1024 18 bit instructions. The instructions are compiled with the FPGA design and automatically loaded during the FPGA configuration process.

The arithmetic logic unit (ALU) performs all microcontroller calculations. There is no dedicated accumulator, therefore result of arithmetic instructions can be stored in any register. `ZERO` and `CARRY` flags may also be set.

An internal 64-byte scratchpad RAM provides more internal read/write storage. This increases the data that can be stored inside the processor and reserves the inputs and outputs ports for data that needs them.

The Input/Output ports allow the PicoBlaze to be connected to custom peripherals and to other FPGA logic.

The 10-bit program counter (PC) supports up to 1024 instructions. It normally points to the next instruction to be executed and increments automatically. Only `JUMP`, `CALL`, `RETURN` and `RETURNI` instructions can change this behaviour. On reaching the highest address; `0x3FF`, the PC rolls over to `0x000`.

A `CALL/RETURN` stack with 31 entries is maintained to support up to 31 nested `CALL` sequences.

An optional external interrupt input is available to handle asynchronous events. Five

Stellenbosch University http://scholar.sun.ac.za


clock cycles are required to respond to an interrupt.

After configuration of the FPGA, the PicoBlaze is automatically reset, which forces the processor into the initial state. The program counter is set to `0x000`, the flags are cleared, interrupts disabled and the stack is reset.

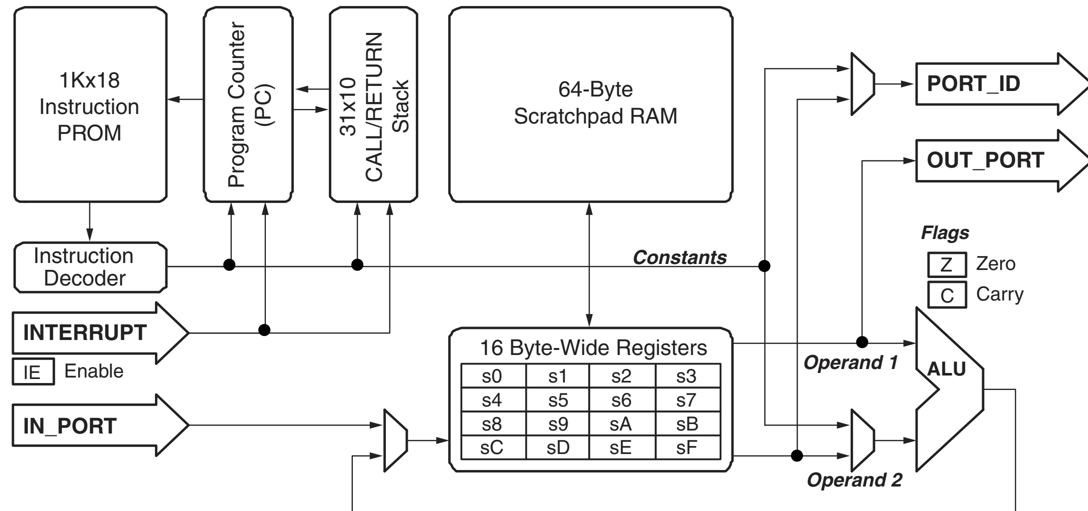



**Figure 3.3:** PicoBlaze Embedded Microcontroller Block Diagram.[31]

### 3.2.3 Performance

Despite the small size of the PicoBlaze it offers very respectable performance.

One instruction always executes in 2 clock cycles, which allows accurate predictions regarding performance. When implemented on a Spartan-3, up to 44 MIPS is possible, while a Virtex-II or Virtex-4 will reach 100MIPS.

### 3.2.4 License

The PicoBlaze is released in full source, without any restrictions on the use, modification or redistribution of the source code, as long as it is implemented on a Xilinx device.

The design is provided “as-is”, with no warranties or conditions, express, implied, statutory or otherwise. It is not supported as an official Xilinx product by Xilinx technical support, but the author can be contacted with issues or improvements to the design.

### 3.2.5 Development Tools

PicoBlaze programs are written in assembly code, which is translated into a VHDL file that contains the initialisation values of the on-chip program ROM. This file is instanti-

Stellenbosch University http://scholar.sun.ac.za


ated by the PicoBlaze core and is synthesised along with the rest of the design.

Several PicoBlaze assemblers and simulators are available, some offering integrated development environments (IDEs), as listed in Table 3.1. The IDEs are fairly similar in terms of the offered features; most of which simplify the debugging process.

The author of the PicoBlaze maintains that the use of an assembler instead of a compiler allows more flexibility in the architecture, because changes in architecture can easily be accommodated by small changes in the assembler. He believes that a task that requires a compiler for the PicoBlaze would probably be more suited to a more powerful SCP, such as the MicroBlaze. [32]

A small-C compiler, `PCCOMP`[2], is available for the standard PicoBlaze configuration.

Other tools that can speed development include a JTAG-based program loader and a function to update the program code without having to recompile the whole design.

## 3.3 Processor selection

For the purposes of this investigation, a SCP was needed that would allow the easy application of different mitigation techniques. Constraints were posed by the available hardware, especially the capacity of an FPGA.

Based on these factors, the PicoBlaze was selected to serve as platform for the evaluation of SEU mitigation techniques.

### 3.3.1 Application of error mitigation techniques

The highly modular and configurable design of the LEON makes it relatively straightforward to replace sub-modules with a fault tolerant equivalent. This makes it possible to apply different levels of protection, depending on the available resources. The relative complexity of the LEON would, however, make debugging fairly complex.

The PicoBlaze is a much smaller design than the LEON, with a simpler architecture. The low-level implementation is harder to modify directly, but the complete system is more manageable, even when major structural modifications are made.

### 3.3.2 Target FPGA

The LEON supports all the major FPGA vendors, while the PicoBlaze is only targeted at Xilinx FPGAs. The choice of FPGA was greatly influenced by the desire to simulate SEUs in the configuration memory, for which Xilinx FPGAs were regarded as the most suitable. The factors influencing FPGA selection are discussed in Chapter 6.

[2] http://www.poderico.co.uk/

Stellenbosch University http://scholar.sun.ac.za


**Table 3.1:** PicoBlaze development tools

| | **Xilinx KCPSM3** | **Mediatronix pBlazeIDE** | **Xilinx System Generator** | **kpicosim** |
|---|---|---|---|---|
| **Platform support** | Windows<br>Linux (using dosbox) | Windows 98, 2000,<br>NT, Me, XP | Windows 2000, XP | Linux |
| **Assmbler** | Command-line DOS | Graphical<br>within System Generator | Command-line | Graphical |
| **Simulator** | Support for<br>VHDL simulation | Graphical/Interactive | Graphical/Interactive | Graphical/Interactive |
| **Breakpoints** | N/A | Yes | Yes | Yes |
| **Register viewer** | N/A | Yes | Yes | Yes |
| **Memory Viewer** | N/A | Yes | Yes | Yes |
| **License** | Freeware | Freeware | Commercial | GNU GPL |

Stellenbosch University http://scholar.sun.ac.za


The capacity of the FPGA was a major concern: it needed to be at least 3.5 times the space required by the reference design, to allow triple redundant implementations. It was calculated that a triple redundant LEON would only fit into the largest device supported by the available software, a Xilinx Spartan-3 XC3S1500. A larger device would have required an additional software license for Xilinx ISE.

The availability of the target FPGA also played a role in the selection of a processor. A Spartan-3 XC3S200 development board that supports all the desired functionality could be purchased at low cost, while a circuit with an XC3S1500 would have had to be designed and manufactured.

Stellenbosch University http://scholar.sun.ac.za

# Chapter 4

# Fault Tolerance in FPGAs

Error detection and correction (EDAC) techniques, that are used to improve the fault tolerance of FPGA-based designs to SEE's, are discussed in this chapter. These techniques are presented in detail to serve as a reference for future error mitigation work. The proposed guidelines for applying these techniques are presented at the end of this chapter.

As with the rest of this thesis, the focus is primarily on the application of fault tolerance techniques to high-density SRAM devices, especially the Xilinx Virtex-II and Spartan-3 families.

## 4.1 SEE effects on FPGAs

The effect of an SEE on an SRAM FPGA can vary greatly depending on its location on the device. In this section the influence on FPGA primitives is discussed.

All devices have a static cross-section; representing the likelihood of an SEU, as well as a dynamic cross-section; which represents the likelihood of an SEU occurring that influences the functioning of the device.

### 4.1.1 Static cross-section

The static cross-section, $\sigma$, of a device is defined as:

$$\sigma = \frac{N}{\Phi} \sec\theta \qquad (4.1)$$

where $N$ is the number of upsets, $\Phi$ the fluence that caused $N$ upsets and $\theta$ the incidence angle of the particles relative to the surface normal.

This cross-section is composed of the sum of the cross-sections of the different elements

Stellenbosch University http://scholar.sun.ac.za


on a device (discussed below):

$$\sigma = \sigma_{config} + \sigma_{BRAM} + \sigma_{FF} + \sigma_{SET} + \sigma_{HL} + \sigma_{SEFI} \tag{4.2}$$

This relationship is expressed as a set in Figure 4.1. The sizes of the subsets are determined by the number of bits involved, as well as the susceptibility of the bits to upsets. Most categories are common in many digital devices (flip-flops, Block SelectRAM, transients, SEFI), while others are unique to SRAM FPGAs (configuration memory). Some of the subsets can be observed directly, which allows for measurement of the cross-sections, while cross-section of others can only be inferred.

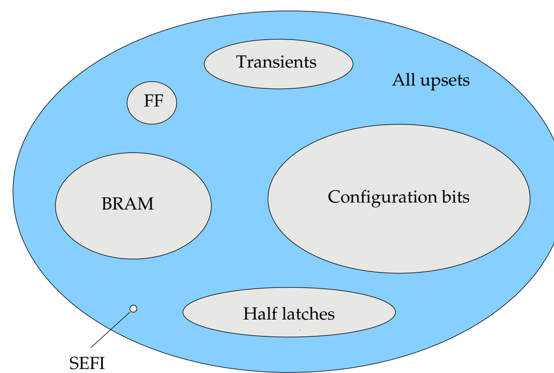



**Figure 4.1:** The set of possible upsets on Xilinx FPGAs

Table 4.1 lists the relative contributions of the different components for a Virtex QVR1000 and a Spartan-3 XC3S200. The contributions marked with a question mark are impossible to measure directly.

### Configuration bits ($\sigma_{config}$)

The largest contribution to the cross-section on SRAM FPGAs is made by the configuration memory. The configuration memory consists of SRAM ( also known as static-cell)

**Table 4.1:** Relative size contributions to FPGA static cross-section, using a uniform upset sensitivity. The contribution by SETs and half-latches is not taken into account. The "?" values are impossible to measure directly. Adapted from [33]

| | **Virtex XCVR1000** | | **Spartan XC3S200** | |
|---|---|---|---|---|
| | Bits | % | Bits | % |
| User Flip-Flops | 26,112 | 0.4% | 3840 | 0.37% |
| Block SelectRAM | 393,216 | 6.4% | 221184 | 21.2% |
| Configuration | 5,603,456 | 91.0% | 818016 | 78.4% |
| SEFI | ? | <.0021% | ? | ? |
| SET | ? | ? | ? | ? |
| Half-latches | ? | ? | ? | ? |

Stellenbosch University http://scholar.sun.ac.za


elements that determine the functionality of the programmed device by configuring routing bits and look-up tables (LUTs).

Upsets in this section may lead to erroneous processing, but because a large proportion of the elements are left unused on all designs, not all upsets will result in failure of the design.

The configuration memory of Xilinx FPGAs can be accessed through a mechanism called "readback". This allows inspection of the configuration while a device is running to detect SEUs. The configuration can be partially reconfigured in some devices to correct upsets in certain segments without influencing the design as a whole.

The look-up tables can also be configured as shift registers or as distributed RAM, with similar behaviour to the BRAM discussed below. For the purpose of this investigation, LUTs were considered as part of the configuration memory, except where otherwise stated, because they were primarily used to implement logic function.

#### Block RAM ($\sigma_{BRAM}$)

The Virtex and Spartan families of FPGAs offer an internal RAM element, Block RAM (BRAM), to store larger amounts of user data. These elements also make a significant contribution to the device cross-section.

BRAM is organised as configurable, synchronous 18-Kbit blocks in recent devices. It stores up to 18-Kbit more efficiently than flip-flops would.

The BRAM, like the configuration memory, can also be read back, if the device is shut down first. The structure of the BRAM makes it easy to implement parity codes to protect the stored data.

#### Flip-flops ($\sigma_{FF}$)

Flip-flops can be used as either registers or latches to store values in a design. The number of flip-flops used is usually small in comparison to the bits in the BRAM and configuration memory, so they only contribute a small part to the cross-section.

Although the state of a flip-flip can be captured, it is hard to predict correct values in a complex design. The easiest way to protect against SEUs is to compare redundant flip-flops.

#### Half-latches ($\sigma_{HL}$)

Half-latches are a convenient way of providing a constant 0 of 1 value in a design, without consuming LUTs or routing resources.

Stellenbosch University http://scholar.sun.ac.za


The element can be seen as a latch with no input; it is initialised at configuration. An upset in this element will only be indicated by the failure of the logic that depends on it.

It has been proposed that half-latches be replaced with a connection to a pin, to provide an external reference, but the cross-section of the routing elements used may lead to other problems. However, by moving the problem from the unobservable half-latch domain to the more accessible configuration memory, upsets can be detected.[34]

#### Transient effects ($\sigma_{SET}$)

Transient upsets are largely filtered by the clocking of registers, which limit the sensitivity to a very small period.

The duration of transient pulses is a couple of hundred picoseconds. In the future, SETs will probably become a greater problem at higher frequencies on faster FPGAs: very few FPGA based experiments have detected SETs to date.

#### SEFI ($\sigma_{SEFI}$)

FPGAs also contain various special elements to control the configuration and start-up of the device. Upsets in elements such as the JTAG controller, SelectMap controller, Power-on-reset logic and other unobservable logic can cause the device to reset, shutdown or to stop responding. The upset of readback logic can lead to an apparent large number of upsets in readback data.

The SEFI cross-section can be considered a lower bound on the dynamic cross-section, since it is determined by the physical properties of the device and can not be decreased.

#### Special elements

Most FPGAs offer special elements to improve device utilisation, such as delay-locked loops and hardware multipliers. These elements provide commonly-used, specialised logical functionality on the FPGA. They are configured by the configuration memory, but their internal states are usually hard to access.

Luckily, most of these elements also have a very small SEU cross-section.

### 4.1.2 Dynamic cross-section

The static cross-section presented above provides an upper bound on the sensitivity of any design on a particular device, because it is determined by the device itself. Another figure, the dynamic cross-section ($\sigma'$) gives a more accurate representation of the SEU sensitivity of a design.

Stellenbosch University http://scholar.sun.ac.za


The utilisation and extent to which SEUs are mitigated determine the difference between $\sigma$ and $\sigma'$. Even in dense designs only a part of the configuration memory is used. Upsets in unused areas rarely influence the functioning of the rest of the design. SEUs in the logic and memory may also be suppressed by logic, in effect mitigating the SEUs.

The dynamic cross-section can be expressed as:

$$\sigma' = \alpha\sigma \tag{4.3}$$

$$= \alpha_{config}\sigma_{config} + \alpha_{BRAM}\sigma_{BRAM} + \alpha_{FF}\sigma_{FF} + \alpha_{SET}\sigma_{SET} + \alpha_{HL}\sigma_{HL} + \sigma_{SEFI} \tag{4.4}$$

$$\approx \alpha_{config}\sigma_{config} + \alpha_{BRAM}\sigma_{BRAM} + \sigma_{SEFI} \tag{4.5}$$

where the $\alpha$ values represent the respective scaling between the static and dynamic cross-sections. No scaling is applied to the SEFI cross-section since it is device dependent, not design dependent. All the other cross-sections are influenced by the level of utilisation and mitigation in the design. The cross-sections of the configuration memory and BRAM areas are significantly larger than the others, making it possible to the approximation in Equation 4.5. The SEFI cross-section is also included in this approximation, since it will stay constant even if mitiagtin is applied.[35]

$\sigma_{BRAM}$ was initially included in $\sigma_{config}$ but radiation testing showed they have different levels of susceptibility to SEUs (ie. different bit cross-sections). This description distinguishes between the two cross-sections and is therefore more accurate.

## 4.2 Physical tolerance

The susceptibility of a device to radiation effects is primarily determined by the physical properties of the device. Devices with a reduced cross-section due to changes in the manufacturing process is referred to as *radiation hardend*. Chip manufacturers have altered the fabrication process to provide better radiation tolerance, in order to satisfy demand in the military/aerospace market. However, the stringent verification procedures required and low volumes involved mean that the price of these devices remain prohibitively high.

Physical tolerance has mostly provided an increase in the TID and SEL thresholds of devices. Memory elements (registers, latches and RAM) usually still need protection through other mechanisms.

### 4.2.1 Configuration memory

Although this thesis is primarily concerned with SRAM FPGAs, the significant contribution the configuration memory makes to the device cross-section makes it important

Stellenbosch University http://scholar.sun.ac.za


to note the alternatives.

Some FPGAs use flash memory to store the configuration, while antifuse devices are configured by burning physical fuses on the device. Flash devices allow reconfiguration, while antifuse devices are one-time programmable. In both cases the configuration memory is effectively immune to SEUs, with only memory elements requiring SEU protection. This makes these devices extremely attractive if the high capacity and reconfiguration offered by SRAM devices are not strong requirements.

It is also important to note that although the mask sizes of SRAM devices have been shrinking with every new generation of FPGAs, the devices have also become more robust to upsets. This can be attributed to improvements on the design of the elements used to implement the SRAM cells. [36]

### 4.2.2 Transistor level redundancy

Several SEU tolerant SRAM cells have been proposed. Most of these proposals add extra transistors to protect against SEUs. The greatest benefit of this approach is that it is completely transparent to the end user. The use of a commercial foundry is also a possibility.

One interesting mechanism, proposed by Rockwell Collins, is called "Multiple Independent Redundant Transistors" (MIRT). Gates are constructed by interleaving three redundant transistors that are physically separated. The separation and interleaving provides automatic triple voting. The physical structure is shown in Figure 4.2.

The physical separation ensures that only a single transistor is upset by an SEU. If a value in one of the devices is momentarily upset, the two complementary devices will dominate the output and result in variation of only 33% from the expected output voltage. The deviation should be sufficiently far from the threshold voltage for the correct value to be recognised.

The area overhead required to implement full MIRT protection is about an extra 125%. It can be decreased to only 25% if used in conjuction with other EDAC mechanisms.[37]

The only SEU tolerant FPGA released to date is the Atmel AT40KEL040. A configuration upset rate as low as $3\times10^{-6}$ upsets per device per day is claimed in the datasheet.[38]

### 4.2.3 Manufacturing technology

The radiation characteristics of devices can be improved at great expense by using techniques such as Silicon-on-Insulator (SOI) or Silicon-on-Sapphire (SOS). These techniques require high precision manufacturing, which is reflected in the cost. Improvements in the TID response and latch-up susceptibility have been the main gains, but the configuration memory of SRAM FPGAs still requires other mechanisms.

Stellenbosch University http://scholar.sun.ac.za

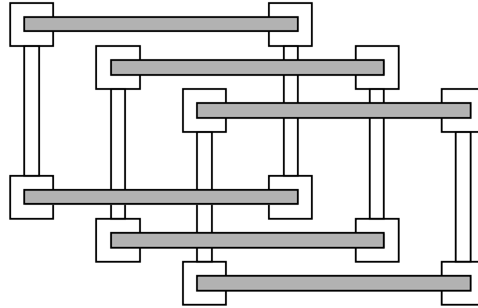



**Figure 4.2:** The physical structure used by Multiple Independent Redundant Transistors (MIRT). The spacing and interleaving provide automatic majority voting.[37]

Xilinx radiation hardened QPro devices are manufactured on a thin epitaxial wafer. This allows a guaranteed TID of 125krad (Si), while the device is also latch-up immune to a LET of 125MeV $cm^2/mg$. A redundant HDL implementation is still required for full SEU immunity.[39]

## 4.3 Error mitigation

Error mitigation is based on the philosophy that a device with adequate radiation error handling capabilities can provide a similar degree of dependability as a hardened device, but at a fraction of the cost. Instead of preventing errors, the propagation of errors is suppressed which limits further disruption to the system. If expressed in terms of cross-sections, error mitigation entails decreasing $\alpha$ (the scaling factor that relates the dynamic to the static cross-section).

All forms of error mitigation inherently depend on redundancy in one form or another. This meta data is used to reconstruct the correct behaviour. In some cases detecting an error offers enough protection – by flagging the data as incorrect, it can be handled in the rest of the system.

### 4.3.1 Spatial redundancy

Multiple physical copies of the same circuit can execute in parallel, so that outputs can be compared for error detecting and correction.

Stellenbosch University http://scholar.sun.ac.za


**Table 4.2:** Dual redundancy error detection and correction. When $x$ and $y$ do not match, an upset has occurred. The XOR only detects errors, the AND assumes that $1 \rightarrow 0$ cannot happen, while the OR assumes $0 \rightarrow 1$ upsets are impossible.

| $x$ | $y$ | XOR | AND | OR |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |

### Dual redundancy (DR)

Dual redundancy entails placing two identical copies of a logic block in parallel to detect errors. Any discrepancies in the outputs indicate an error.

DR has a significantly lower cost than triple redundancy, but usually cannot correct any errors. Correction would normally be handled by another mechanism. In this manner, the LEON-FT model uses dual redundancy in the integer unit pipeline. When an error is detected the pipeline is restarted to mitigate any effects.[8]

If SEUs on a device are biased; which means they display a different cross-section for $0 \rightarrow 1$ than for $1 \rightarrow 0$ upsets, DR can be used to implement an efficient probabilistic correction scheme. If $0 \rightarrow 1$ upsets are more frequently observed, performing a logical AND on the outputs of a module with that of a redundant copy would assume the 0 output to be correct. Devices displaying more $1 \rightarrow 0$ upsets can similarly be corrected using a logical OR, which favours a 1 output.

The truth table for the detection and correction possibilities is listed in Table 4.2.

### Triple Modular Redundancy

Triple modular redundancy uses three copies of the same module and a majority voter to correct any errors. If two or more inputs to the majority voter are the same, that value is regarded as correct and propagated. The most common example of TMR is the redundant D-flip-flop in Figure 4.3. The truth table for the voting circuit is given listed in Table 4.3.

Its relative simplicity and ease of implementation, as well as the excellent error mitigation properties, makes TMR the most most popular EDAC method for many applications. The greatest drawback is the high size overhead (a system with full TMR can be up to 350% the size of the original design).

### Multiple voters

When using TMR in the modules of a system, having a single voter presents a risk, because the voter itself is also sensitive to upsets. To mitigate this risk, redundant copies

Stellenbosch University http://scholar.sun.ac.za

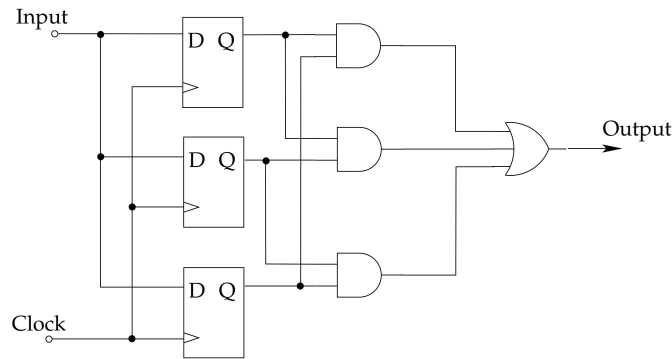
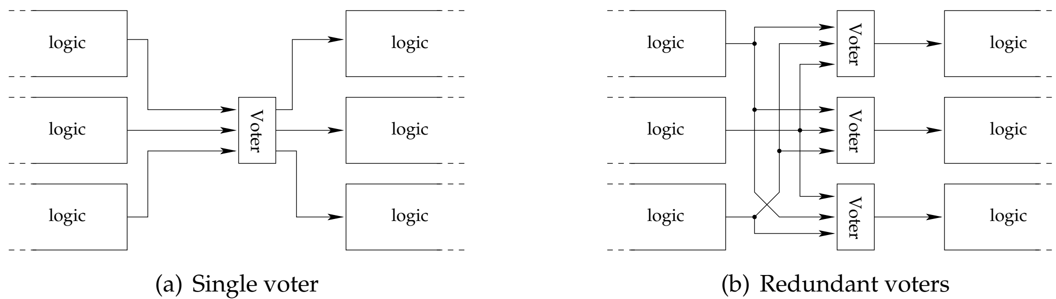




**Figure 4.3:** Triple modular redundancy with voting

**Table 4.3:** Majority voter truth table

| $x$ | $y$ | $z$ | Output |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

of the majority voters are used, each one driving its own module, as shown in Figure 4.4.




(a) Single voter (b) Redundant voters

**Figure 4.4:** Using a single voter between sub-modules (a) creates the risk of the voter itself being upset. By using redundant voters (b), even if a voter is influenced, the redundancy ensures that the system is not influenced.

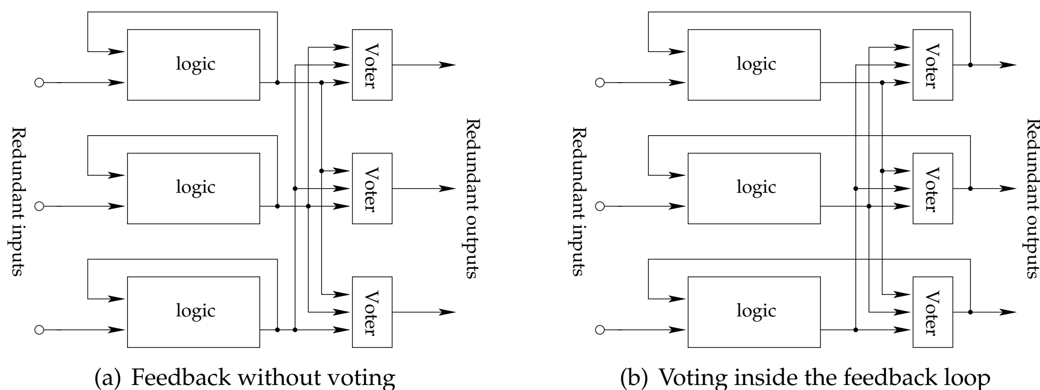
This is especially important in SRAM technologies, but when it is implemented in hard silicon, a single voting circuit is often enough.

Stellenbosch University http://scholar.sun.ac.za


### Feedback and memory

Systems containing feedback and memory also need special consideration when implementing TMR. Although the voting circuit corrects the outputs, an incorrect value may persist internally through feedback, state machine values or when stored in memory.

By placing a voter inside the feedback loop, as shown in Figure 4.5, one can ensure that the correct data is fed back, maintaining the correct state inside the module.



(a) Feedback without voting

(b) Voting inside the feedback loop

**Figure 4.5:** Voting inside the feedback loop helps to prevent persistent errors.

### Xilinx TMR (XTMR)

Xilinx has proposed a comprehensive TMR scheme for their Virtex FPGAs, in order to obtain full data retention and autonomous recovery.[40]

The different parts of a design are classified according to their function and structure:

- *Throughput logic* is a module of any size, synchronous or asynchronous, where no logic loops exist in the signal paths. Throughput logic is protected by simple duplication of the logic.
- *State machine logic* has a registered output that is fed back into a previous stage of the design. Replication and voting inside the feedback loop provide the required protection.
- *I/O logic* is protected by using redundant pins that are connected outside the FPGA. Input pins are merely connected to the same trace. Output signals require protection against conflict on the external bus, a minority voter is therefore used to disable a tri-state buffer on the pin of the misbehaving signal, as shown in Figure 4.6.

Stellenbosch University http://scholar.sun.ac.za


All pins are also triplicated, with the tracks connected outside the chip, to eliminate all single points of failure in the design. Signal conflict is prevented by minority voters which disable misbehaving pins from inside the FPGA. This scheme has been simulated and tested to provide 100% coverage of the configuration memory, leaving only the SEFIs in the control logic as a very small cross-section.

- *Special features* include dedicated hardware modules, such as Block RAM (BRAM) and the digital clock manager (DCM). The best way of protecting these elements is to wrap them in custom logic that provides EDAC functionality. This could be a parity based correction code for Block RAM, or a more advanced scheme that also refreshes data. Other elements can most easily be protected using TMR, with a reset circuit to correct potential state errors.

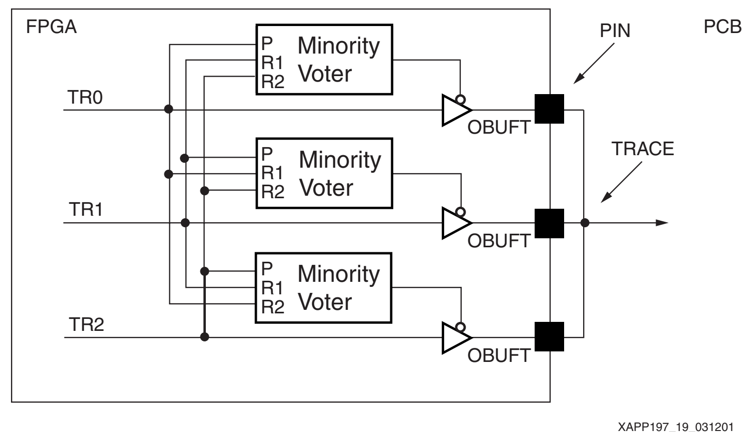



**Figure 4.6:** Minority voters used to disable an output pin.[40]

Applying these techniques can be quite laborious, so Xilinx has made templates available for several protected components. The complete mitiagation process has also been automated in software called `TMR Tool`.

### Functional Triple Modular Redundancy (FTMR)

Gaisler research has formulated a VHDL approach to error-mitigation that performs TMR at gate level. The goal is to free the designer from having to take TMR into consideration at all, allowing one to utilise the power of a high level description language, such as VHDL. Splitting the design into modules that conform to a specific behaviour and interface, allows various levels of TMR to be implemented, without the designer having to keep the details of the mitigation process in mind.[41]

Stellenbosch University http://scholar.sun.ac.za

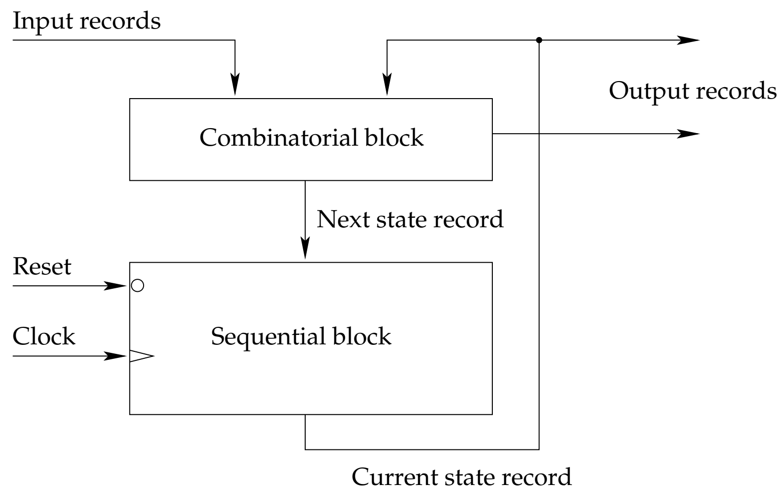




**Figure 4.7:** Sequential and combinatorial logic blocks in FTMR.[41]

A design is split into combinatorial and sequential blocks, as shown in Figure 4.7. The redundancy of the sequential block is achieved by implementing flip-flops with a special TMR-equivalent.

The redundancy in the combinatorial section is implemented by describing the combinatorial logic in a procedure that is instantiated multiple times.

VHDL records are used extensively to package redundant signals, for both voting and for passing between modules. The sequential block has one input record; containing the next state of all the elements in the block, one output record; that carries all the current values in the block, as well as clock and reset signals.

Only directly instantiated flip-flops are triplicated, as no way exists to triplicate inferred flip-flops.

Varying levels of redundancy is possible:

- The *structural* implementation uses only triplicated flip-flops.
- The *sequential* implementation uses redundancy only on the sequential logic
- The *combinatorial* implementation provides triplication of all the combinatorial logic and ports.For purely combinatorial logic, no voting occurs inside the module, since it is assumed that it will be voted when it reaches an output or a flip-flop.

Although FTMR seems to offer a way of separating the error mitigation process from the design procedure, the constraints for separating the sequential and combinatorial logic can often lead to a fairly low-level description of the system, in order to maintain control. This is contrary to the power of using a high level description language.

Stellenbosch University http://scholar.sun.ac.za


### 4.3.2 Temporal redundancy

Transient errors can be detected by repeating the execution sequence at another time.

A series of sequential operations can be repeated to confirm results. In software this involves executing all instructions more than once. This will more than halve the effective speed of the design, but will not increase the spatial requirements of the design.

A delay-based redundancy scheme can been used on sequential logic to correct transient errors. Unwanted transients can be effectively eliminated by replacing D flip-flops in a design with a equivalent, error-correcting flip-flip, as shown in Figure 4.8.

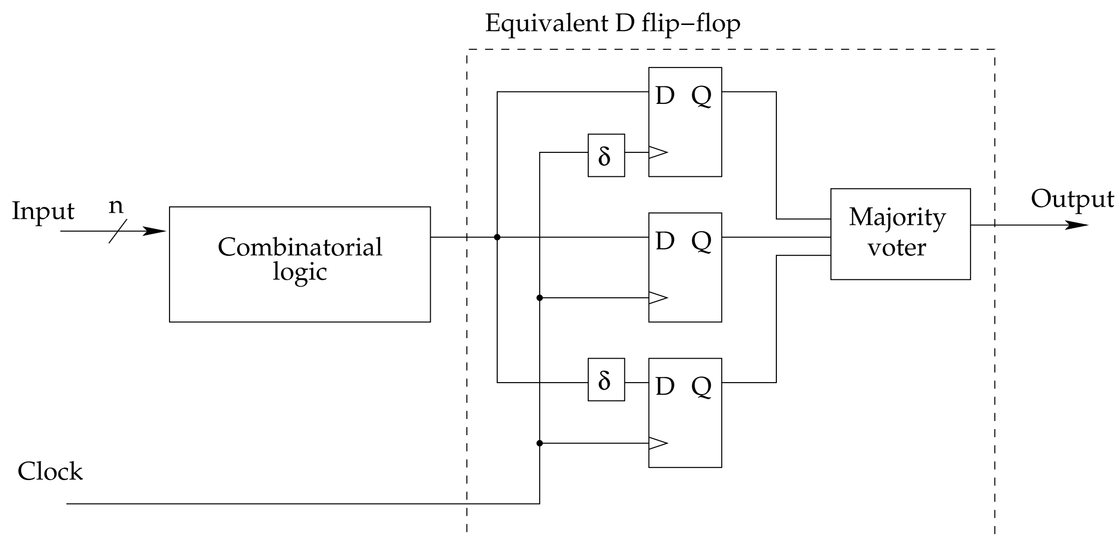



**Figure 4.8:** Time redundant D flip-flop. By using a delay $\delta$, the clock and data can be shifted to eliminate transient errors with a duration less than $\delta$.

In this equivalent flip-flop, a delay element ($\delta$) is used to shift the clock and data signals to sample the result at 3 different times. For an SET to influence the results, it would have to last long enough to influence two of the samples (longer than $\delta$). The performance penalty is largely determined by the the delay element.

This scheme is most suitable for ASIC applications, where an extra buffer can be inserted to provide the delay. In FPGAs, the delay can be implemented by routing the signals through another logic element. The best way to achieve it, would be to modify the netlist generated by the synthesis tool, because any high level description of the time redundant flip-flop would be optimised away.

A similar scheme was used for error detection in the ROC81, a fault tolerant version of the LEON-1 developed by iROC Technologies.[42]

Stellenbosch University http://scholar.sun.ac.za


### 4.3.3 Data redundancy

Data redundancy can also be seen as a subsection of spatial redundancy. Due to the fact that data is usually used in words or blocks it is possible to apply more sophisticated techniques than TMR.

In general, the more data needs to be protected, the more efficiently it can be done. However, advanced EDAC techniques also take longer to execute.

These techniques can be applied in hardware or software. Hardware can abstract the mitigation mechanisms from the programmer, but will decrease performance in high speed systems. Software techniques also decrease performance, but do not increase the hardware requirements.[43]

#### Single error detection

Errors can be detected by storing an extra parity bit with every word of data.

This bit is calculated to make the number of 1 bits in the word an odd number for odd parity, or an even number for even parity. The parity is checked again when the word is read. If it does not have the expected odd or even number of 1 bits, the word has been corrupted.

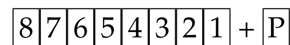



**Figure 4.9:** 8-bit word with parity bit (P). The parity bits are calculated to set the number of 1's in the word to an odd (odd parity) or equal (even parity) number. The parity is checked when the word is read: if it differs from the expected odd or even parity scheme, an upset has been detected.

The high demand for applications that use parity bits has encouraged Xilinx to provide extra bits in the internal RAM of their FPGAs to store these parity bits.

### 4.3.4 Error correction codes (ECC)

More meta data can be stored to allow error correction. Error correction can be applied to one dimensional data words (longitudinal codes), or to two dimensional blocks of data (block codes).

Block codes are usually applied to large blocks of data, while longitudinal codes are more useful for flow-through data, where EDAC needs to be performed on a single word.

Hamming codes are a popular example of ECC codes, due to their adaptability and easy implementation. Parity bits are generated by `XOR`'ing different combinations of bits in the data word, to check different sets of bits in the code word. In a Hamming

Stellenbosch University http://scholar.sun.ac.za


| Bit position: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Code word | P1 | P2 | 1 | P3 | 2 | 3 | 4 | P4 | 5 | 6 | 7 | 8 |

**Figure 4.10:** Structure of a (12,8) Hamming code word. The P$n$ bits are parity bits; the other store data.

code word, $r$ parity bits are added to the $m$-bit data word, forming an $(m + r)$-bit code word. The bits are numbered from 1, with bit 1 the highest order bit. The parity bits are placed at all positions that are a power of 2; data is stored in the other bits (Figure 4.10). Bit $b$ is checked by those parity bits $b_1, b_2, \ldots, b_j$ such that $b_1 + b_2 + \cdots + b_j = b$. For example, bit 7 is checked by bits 1, 2 and 4.

The unique set covered by the parity bits allows us to determine which bit was upset and correct it. The position of this bit is found by taking the intersection of the sets of bits covered by the incorrect parity bits. One will not be able to correct the data if two upsets occurred, but the corruption can still be detected.

The overhead decreases as more data is protected by Hamming ECC codes. In this way a (12,8) Hamming code uses a 12-bit word to store 8 bits data, a (23,16) Hamming code uses 23 bits to store a 16-bit word, a (38,32) Hamming uses only 6 parity bits to protect 32 data bits and so forth. Every word can only correct one or detect two errors. The decoding logic becomes progressively more complex as the word size increases.

The HDL implementation is done with `XOR`-operations, to obtain the encoded word. In software, code words can be generated by performing modulo-2 multiplication of the data with an encoding matrix.[44][45]

### 4.3.5 Selective redundancy

The high cost of TMR has prompted various techniques that exchange some protection for a lower system cost. "Gate level selective redundancy" involves evaluating relative frequency of input values at gates. Only those inputs that are deemed to be major contributors to possible errors are protected.[46]

This offers a good trade-off between cost and reliability, but requires beforehand knowledge of the relative signal frequencies in the design.

### 4.3.6 Granularity

Various levels of granularity are possible for all the above techniques. Generally, the error tolerance of a system will increase with a finer granularity, but the cost in terms of area and timing will increase. The error tolerance is only a consideration when multiple independent upsets are a threat, such in case where the reset rate is comparable to the dynamic upset rate.

Stellenbosch University http://scholar.sun.ac.za


The lowest cost TMR application has only top-level voting, ie. three copies of the complete design will be redundant. The overhead will only be one voter for every output of the design. However, this approach has very little internal protection. The large cross-section of the modules makes it quite possible for more than one module to be upset. If the errors persist, the majority voter will fail. The probability of failure decreases if redundancy is applied to modules inside the design with majority voting between the modules, as shown in Figure 4.11.

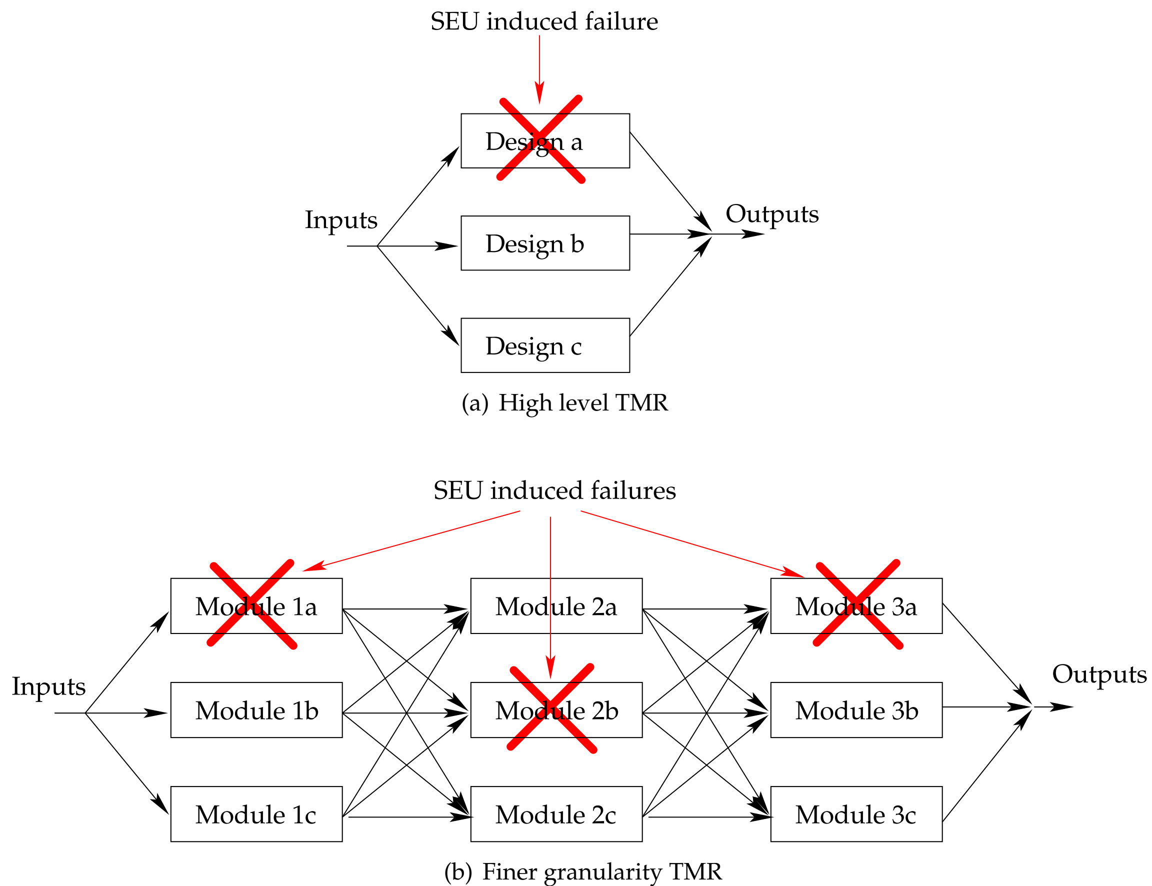



**Figure 4.11:** TMR at different levels of granularity. The high-level TMR in (a) can only mitigate a single SEU induced failure, while the implementation of finer granularity in (b) will still function correctly, even if one module in each of the three subsections fail.

## 4.4 External protection

External protection is used to protect the device against latch-up and to correct SEUs in the configuration.

Stellenbosch University http://scholar.sun.ac.za


### 4.4.1 Latch-up protection

When latch-up occurs, the FPGA draws an unexpectedly high current. To protect the device, the supply current must be monitored and interrupted if it exceeds a preset threshold. Doing this will cause a power off-on reset: if enough time is provided for the parasitic thyristors to switch off, permanent damage to the device can be prevented.

The power consumption of FPGA needs to be characterised before use in order to do this effectively, because different designs can draw drastically different currents. The current can be monitored with an analogue comparator which triggers a monostable ("one-shot" oscillator) to reset the device. Alternatively a microcontroller and an analogue-to-digital converter (ADC) can be used, either by the microcontroller polling the ADC or having the ADC raise an interrupt.

### 4.4.2 Configuration memory

The configuration memory, as discussed before, is a major contributor to the SEU cross-section of SRAM FPGAs. Although redundancy techniques such as TMR can mitigate the effects of upsets in the configuration, these errors need to be corrected to prevent accumulation.

"Scrubbing" involves reloading the entire configuration memory, thereby correcting all SEUs. This can be done at regular intervals (as determined by the desired reliability of the design) or triggered by error detection logic.

Xilinx FPGAs also allow readback and partial reconfiguration of the configuration memory. Readback can be used to verify the integrity of the configuration, and partial reconfiguration can be used to only correct the corrupted frame. It is possible to perform this readback and reconfiguration without interrupting the FPGA (on Virtex-II to Virtex-5 devices), which provides protection with almost 100% device availability.

The implementation and constraints of readback are discussed in Chapter 5.

## 4.5 Soft-core processors

Soft-core processors often have special features that require consideration, because of the complexity of the system. These features are mentioned here for the sake of completeness and because it illustrates some of the innovative ways that fault tolerance can be achieved in systems with higher complexity.

Stellenbosch University http://scholar.sun.ac.za


### 4.5.1 Cache memory

On-chip cache memory is used to speed up access to external memory by caching recently used items. This type of memory is also susceptible to upsets and needs protection.

SEUs need only be detected (for example by using a single bit parity code), since a correct copy of the original entry is still available in the external memory. Corrupt data will force a cache miss, refreshing the entry with the original data.

A write-through policy is normally used when writing to cache memory: dedicated hardware also writes the data to the external memory, without impacting on the performance of the rest of the system.

### 4.5.2 Checkpointing

The system needs to be capable of recovering from unexpected interruptions, whether from an SEU or an external reset. In the case of processors, a technique called "checkpointing" can be used to recover from these interruptions.

By storing a complete copy of the internal state of the processor in an off-chip location, the system can simply reload these values and continue executing as before. High performance can be obtained despite interruptions, if precautions are taken to ensure the data used by the processor is still valid. Such precautions can involve atomic operations – operations are performed in blocks, where enough data is stored to recover the previous state should anything go wrong. THis ensures that hte system is always in a consistent state. Alternatively all memory can be moved off-chip, so that the state of the design is not lost when the design is interrupted.

Checkpointing can be done in software, with a routine that is explicitly called at intervals, or is executed in such way that operations are performed atomically. The routine requires some processing overhead, so the frequency can also be changed to adapt to the threat of upsets. Alternatively, it is also possible to perform checkpointing in hardware, with dedicated logic taking a snapshot of the processor at intervals.

## 4.6 Implementation considerations

It is crucial to keep the final implementation on the FPGA in mind when considering the application of these techniques. Applying error-mitigation without realising the effect that implementation has, can nullify any benefits gained. The following provide some guidance on the issues encountered in implementation.

Stellenbosch University http://scholar.sun.ac.za


### 4.6.1 FPGA resources

All FPGAs use some form of configuration memory to combine the basic logic building blocks into the desired logic function.

Some elements, such as latches, registers, shift registers and RAM are more sensitive to SEUs than other elements. Most FPGAs also have some dedicated hardware to perform common tasks, such as integer multiplication or frequency scaling of clock signals. These elements, which use very few configuration bits, offer a very efficient way of improving performance and can decrease the upset cross-section.

In the case of Xilinx FPGAs, majority voters are implemented using 3 input look-up tables (LUTs). These LUTs store the output values in 8 registers, which are set during configuration. An upset in one of these registers will change the logic function implemented by the LUT. The LUTs are connected by the programming of routing bits in the configuration memory. These bits are also sensitive to upsets: an SEU can cause connections to be broken or unwanted connections to be created.[47][48]

The high level description of a multiplier in VHDL can lead to two widely different implementations. If the synthesis software recognises the structure and uses a dedicated hardware multiplier, it will have a much smaller cross-section than when the multiplier is implemented in the configurable logic. To encourage the software to recognise the logic it can translate to dedicated resources, it may be necessary to manually instantiate the hardware multiplier.

### 4.6.2 Synthesis software

A lot of time, effort and money has been spent on improving the logic synthesis tools, especially to make them more adept at recognising and optimising logic. The original motivation was to minimise the logic utilisation of a design, but the software also has the unfortunate tendency to remove the redundancy used to implement error mitigation. To prevent this unwanted optimisation, the synthesis software needs to be instructed to keep certain components and connections, even if they are obviously duplicated. Unfortunately, the method in which it is specified in HDL differs for different synthesis engines.

Most software use the "attribute" statement to specify which nets or modules should not be optimised. In `Synplify Pro`, this can be done by setting `syn_keep` on a wire, which prevents unwanted optimisation from removing combinatorial logic. Similarly, `syn_preserve` is used to protect sequential elements and `syn_noprune` stops unconnected instantiated elements from being removed.

A section of source code that illustrates the use `syn_preserve` with `Synplify Pro` is listed below (from the design discussed in Chapter 7.1.7).

Stellenbosch University http://scholar.sun.ac.za


```
library synplify;
  use synplify.attributes.all;

entity test_attr is
  port (
      ...
    );
end test_attr;

architecture behav of
  attribute syn_preserve:boolean;
  attribute syn_preserve of dec_loop:label is true;
    ...
begin
  ...
  dec_loop: hamming_dec_12_8
    port map(loop_in,loop_in_orig);
  ...
end architecture;
```

An alternative to using attributes would be to synthesise the unprotected design to produce an intermediate output file in an format such as EDIF. This file can then be processed to implement the required error mitigation techniques, before performing the final place-and-route (PAR) operation.

## 4.7 Conclusion

From the myriad of techniques available, a strategy must be selected to protect designs on SRAM FPGAs.

It is impossible to find a single solution that will provide optimal fault tolerance in all cases, since every case has different requirements. Instead, the following guidelines that should be taken into account, are proposed:

**Cross-section:** The relative cross-sections of the different sections on the FPGA need to be taken into account. The cross-section of the configuration memory and BRAM areas dominates, so these areas should be protected first. Mitigation techniques will decrease the scaling factor ($\alpha$) for both these sections, as well as the other smaller parts. $\sigma_{SEFI}$ remains unchanged, therefore it will become the dominant cross-section, if sufficient error protection techniques are applied to the other sections. This would be the maximum level of protection one can obtain on these devices. Luckily, $\sigma_{SEFI}$ is very small and should not pose a significant threat to most devices.

**Multi-level error mitigation:** SRAM FPGAs require a multi-level error mitigation approach. Errors can be mitigated in VHDL, but for reliable operation, external protection of the configuration memory is required. This system can further be

Stellenbosch University http://scholar.sun.ac.za


expanded to combine hardware, HDL and software based mitigation techniques to protect different areas of the system. By combining the strengths of the individual techniques, good protection can be obtained with a minimal overhead.

**Spatial redundancy:** Spatial redundancy is needed to protect against SEUs in the configuration memory, as it is the only way of providing protection against upsets in the configuration memory. While TMR is the most effective approach, many variations with different levels of protection exist.

Most soft-core processors also have elements, such as memory, that can be protected in other, more efficient ways.

**Cost:** The cost of mitigation can be measured in various ways. The most significant factors usually considered are the logic overhead, implementation effort, power and speed. Although is convenient to consider these factors separately, one should remember that they are all highly interdependent.

**Logic overhead:** A design with high-level TMR requires more than three times the logic required by an unprotected design. This overhead can be decreased by careful combination of TMR with other techniques.

Device utilisation is not the only way of defining cost: one can also look at the remaining resources on the FPGA. By analysing the design in this manner, extra logic could be traded off against scarcer elements, such as memory. This approach is more sensitive to the use of the remaining resources on the FPGA.

**Implementation effort:** The effort required to implement and debug a fault tolerant system is often a deciding factor when selecting techniques.

Ideally, the application of error mitigation mechanisms should be independent from the design process, in order to meet deadlines and keep costs down. This could be done by treating the design as a black box when applying the mitigation techniques, or automating the process in software.

Libraries of fault-tolerant components are also often used. This hides the fault tolerance mechanisms, but still requires cognisance on behalf of the designer.

Low-level fault-tolerance can be done, but is extremely labour-intensive and error-prone, and therefore not attractive.

**Power:** Power consumption is an important factor in satellites, where it is a limited resource.

Hardened designs are larger and require more power. Careful device and clock speed selection decrease power consumption significantly.

One could also use the mitigation techniques where the threat of SEUs are deemed large enough. By reconfiguring the FPGA with a protected design when it passes through the SAA (and the over poles, is desired), while using an unprotected

Stellenbosch University http://scholar.sun.ac.za


design in the rest of the orbit, high levels of SEU protection can be obtained, while limiting power consumption.

**Speed:** The speed of a design will be decreased by placing EDAC logic in the data path.

Majority voters and small EDAC circuits (which have a large logic overhead), are faster than larger, more space efficient EDAC circuits. Multiple voting in a logic path will also decrease speed.

Several of these mechanisms were evaluated by applying them to a PicoBlaze soft-core processor design. It is discussed in detail in Chapter 7.

Stellenbosch University http://scholar.sun.ac.za

# Chapter 5

# Configuration Controller

The configuration controller is used to perform high speed, advanced configuration functions on a target FPGA. It can also serve as the foundation of a configuration single event upset simulator and be used in radiation testing.

The controller responds to commands from a host PC by performing configuration or readback on the target FPGA. By locating all the necessary configuration hardware close to the FPGA, the system can autonomously maintain the integrity of the FPGA.

This chapter focuses on the design of the configuration controller; the SEU simulator and radiation testing are discussed in detail in the following chapters.

## 5.1 Target FPGA selection

The selection of a target FPGA was based on cost, configuration functionality and the availability of applicable literature.

The use of a Xilinx FPGA immediately seemed attractive. The devices are SRAM based, which makes reconfiguration and SEU simulation possible. The parallel SelectMap interface allows for very fast reconfiguration, as well as readback to verify the contents of the configuration memory. Xilinx also produces the only large, radiation-tolerant, SRAM-based FPGAs – working on a commercial Xilinx device makes possible migration to a hardened chip much easier. Finally, Xilinx provides a large amount of information about SEU protection on their devices.

The use of an Altera device was also considered. They offer equivalent high speed configuration, but because Altera does not offer any high reliability devices, very little information on the radiation characteristics of these devices is available.

The *Spartan-3 Starter Kit* (S3Kit), a low cost development board was purchased to provide the target FPGA. This board features an XC3S200-4C Spartan-3, 1 MB of external SRAM, an RS-232 transceiver, a PS-2 port, a VGA port, LEDs, buttons and switches.

Stellenbosch University http://scholar.sun.ac.za


Most importantly, it supports all the configuration modes supported by the Spartan-3. [49][50]

The XC3S200-4C has 200 000 usable system gates, 30Kbit distributed RAM and 216Kbit Block RAM. It also contains 12 dedicated multipliers and four digital clock managers. The package is a 256 pin ball-grid array. The device is specified for the commercial temperature range (0°C to 85°C) and has the speed grade "4". The package markings (AFQ0412) indicates that it was manufactured in December 2004, using 90nm technology on a 200mm UMC wafer. The mask revision is marked "A" and the lot code was D107348A. This information is necessary because the radiation test results in Chapter 8 are influenced by the device's lot number and mask revision.

Although no radiation-tolerant version of the Spartan-3 exists, it is similar in architecture and fabrication technology to Virtex-II, which can be obtained in a radiation-tolerant version. The cost of a Spartan-3 is also significantly lower than that of similar Virtex devices. Large Spartan-3 devices are also supported by the free `ISE WebPack`, so an additional software license was not required.

Plenty of information on the effects of SEUs on Xilinx based FPGAs can be obtained by using this FPGA as a development and testing platform. Although the architecture of the Altera devices differ, the techniques developed on the Spartan-3 would translate to these devices without requiring major modification. The effects of SEUs on flash and antifuse devices can be regarded as a subset of the effects displayed by SRAM devices.

## 5.2 FPGA configuration

The FPGA is configured by loading an application bitstream into the internal configuration memory of the device. The bitstream is organised into 32-bit words. These words carry instructions for the configuration logic, as well as data that will be stored in the configuration memory.

### 5.2.1 Configuration modes

The configuration memory can be programmed in different ways by using either serial or parallel data. In the "Master Serial" or "Master Parallel" modes, the FPGA clocks external non-volatile memory to obtain the configuration data. Alternatively, an external device can supply the clock signal along with the data to the FPGA, in what is known as "Slave Serial" or "Slave Parallel" modes. The mode is specified by setting values on the Spartan-3 mode pins (M0,M1 and M2).

The different configuration modes are listed in Table 5.1. The configuration is synchronised to a clock signal, which can either be an FPGA output (for Master Serial and Master Parallel), or an input (when using JTAG or the Slave modes). Depending on the

Stellenbosch University http://scholar.sun.ac.za


**Table 5.1:** Spartan-3 configuration modes.

| Configuration mode | Synchronising clock | Data Width |
|---|---|---|
| JTAG | TCK input | 1 |
| Master Serial | CCLK output | 1 |
| Slave Serial | CCLK input | 1 |
| Master Parallel | CCLK output | 8 |
| Slave Parallel | CCLK input | 8 |

configuration mode, several dedicated and dual-purpose pins can be used. The dual purpose pins can be used as general input/output pins after configuration, or can be configured to persist which allows the configuration to be read back at a later stage.[51]

### 5.2.2 Configuration speed

The highest possible speed is desired for the purposes of configuration and readback, to minimise downtime and maintenance overhead. Complete access to the bitstream is also required for SEU simulation purposes. The parallel configuration modes (SelectMap) provide the fastest configuration. Byte-wide data is written to the FPGA with a BUSY flag controlling the flow of data. To provide the required configuration and readback functionality, Slave Parallel mode with persistent configuration pins was used in the configuration controller. Configuration speeds of up to 50MHz/byte can be achieved in this mode, without the need for handshaking. The configuration flow is shown in Figure 5.1.

### 5.2.3 Configuration file

The place-and-route (PAR) software generates a binary file (`.bit`) that is used for configuration. This file contains header information, instructions to perform configuration, as well as the configuration data that is loaded into the configuration memory. The header contains information such as the name of the netlist used to create the file, the time of creation and the device it was fitted for. The configuration data is used to program logic and routing on the device, while the final part of the file starts the device.

The configuration file can also be generated without the header portion (`.bin`) to simplify in-system programming. The rest of the file can be used without further processing to program a device. The latter option is used in the configuration controller.

Stellenbosch University http://scholar.sun.ac.za

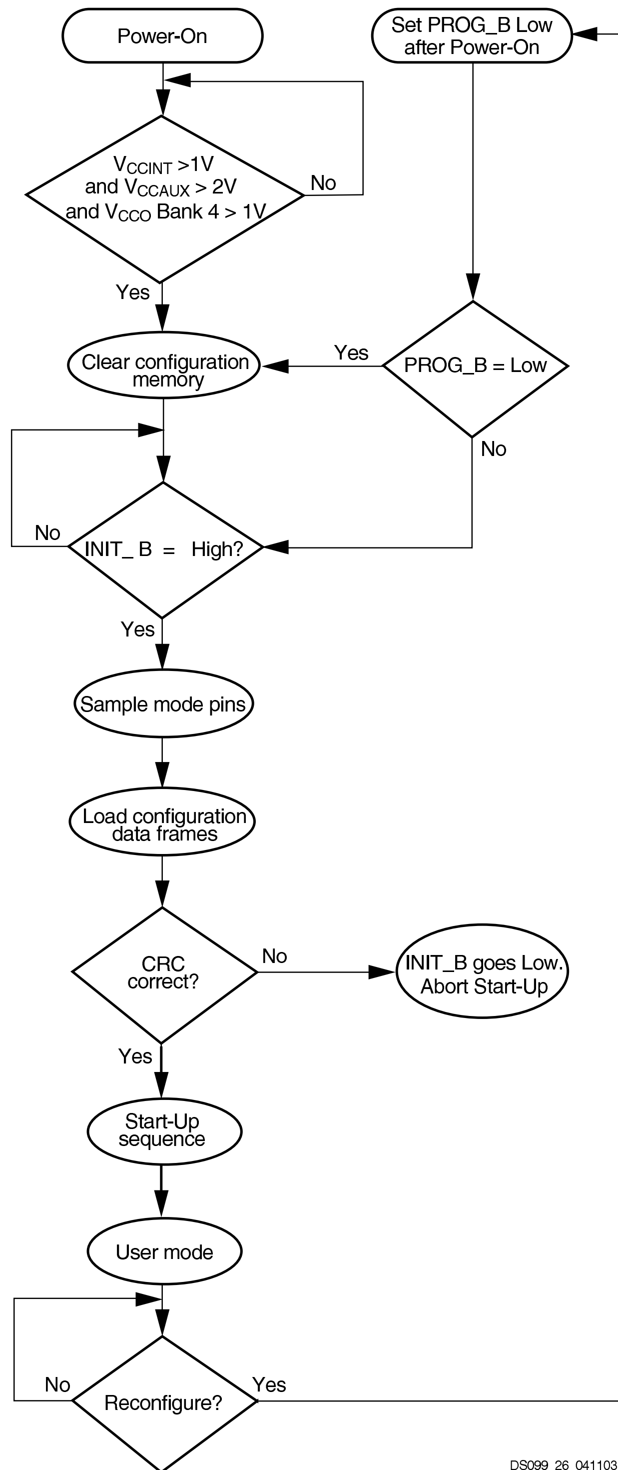




**Figure 5.1:** SelectMap (Parallel) configuration flow diagram[50]

Stellenbosch University http://scholar.sun.ac.za

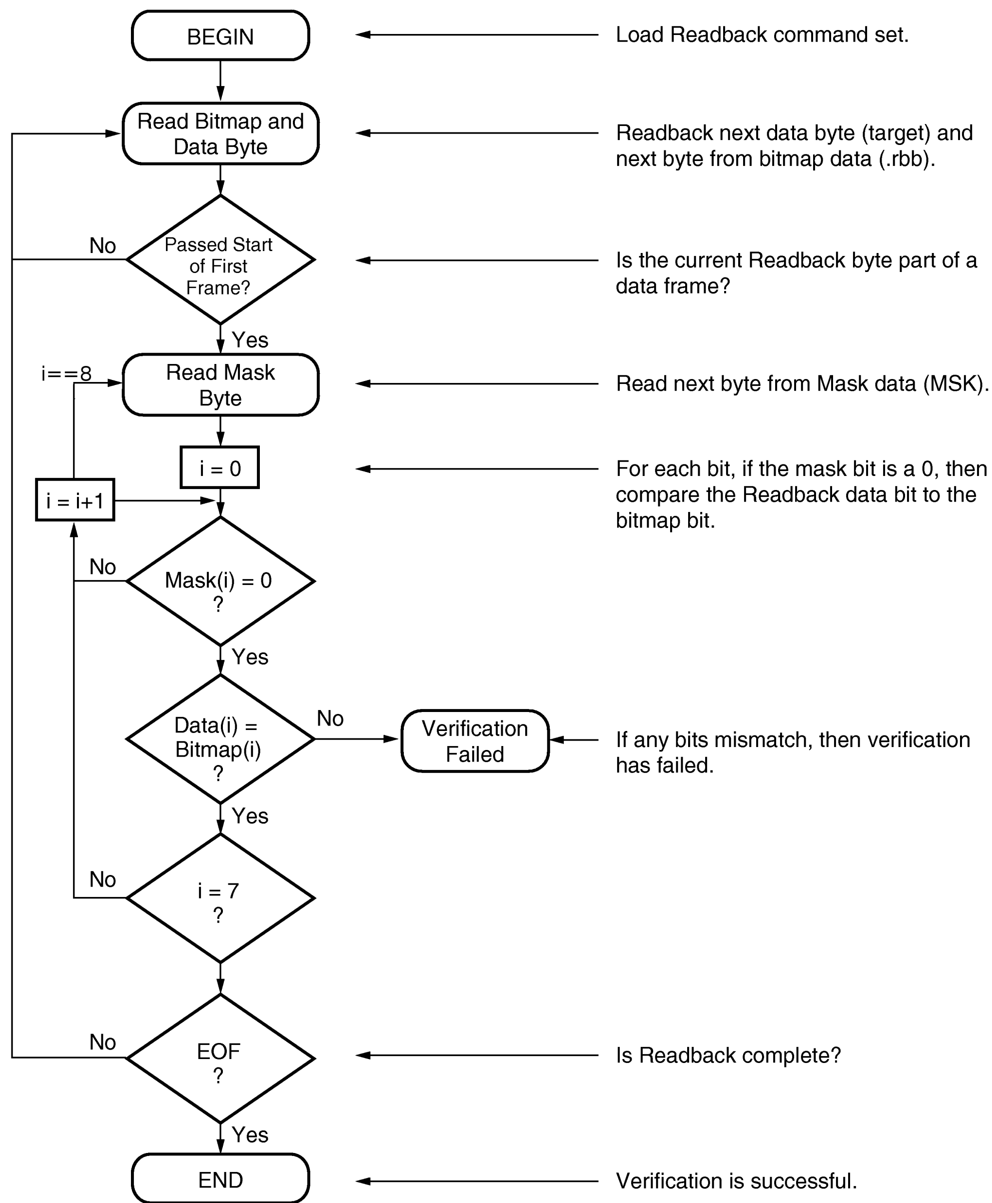




**Figure 5.2:** Readback data verification flow.[51]

Stellenbosch University http://scholar.sun.ac.za


### 5.2.4 Readback verification

The configuration memory is verified by:

1. Reading the contents (target data)

2. Masking the volatile bits

3. Performing a bit by bit comparison with the expected readback data. Any mismatches indicate upsets.

The configuration logic needs to be instructed to read the configuration memory. This is done by commands at the head of the readback file (`.rbb`). The target data is preceded by a block (one configuration frame) of padding data that should not be checked. The values of volatile data (registers, latches and RAM) are unknown and can therefore not be verified. These values are spread throughout the target data, therefore a mask file `.msk` is used to identify them. If a bit in the mask file is set to 0, then the corresponding bit in the readback data is checked. This process is shown in Figure 5.2, while Figure 5.3 shows the alignment of the different data streams.[51]

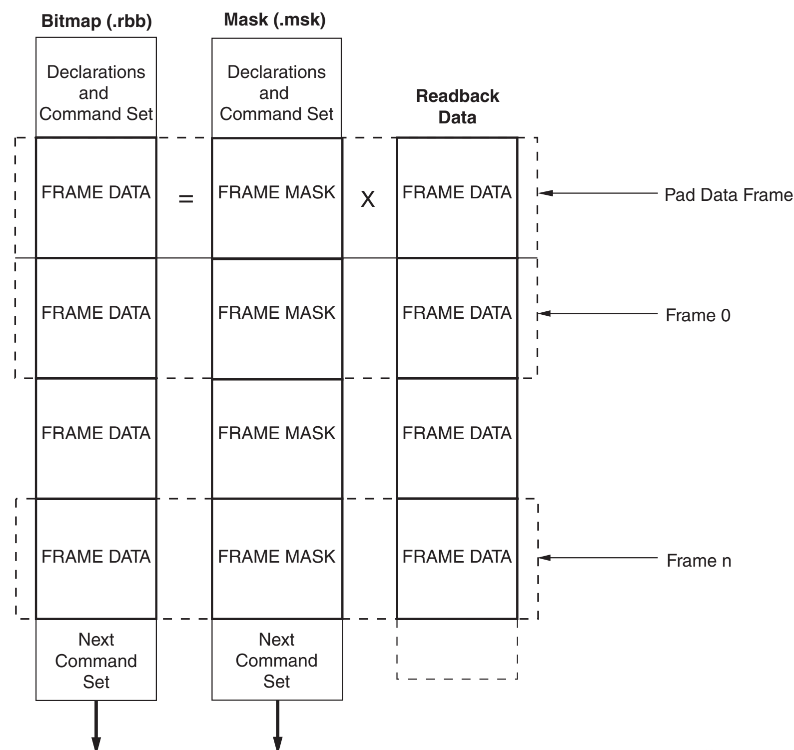



**Figure 5.3:** Alignment of the readback data stream.[51]

Stellenbosch University http://scholar.sun.ac.za


#### Constraints

The device needs to be shut down before performing readback. This prevents corruption of data in the internal RAM elements by access conflicts between the user and configuration logic. The FPGA does not lose the configuration data or register values when it is shut down, which allows it to be restarted without reconfiguration.

To overcome this constraint, a more complex verification method can be used. It entails reading only the parts of the configuration that do not contain volatile elements. This can be done without shutting the device down. The locations of the RAM elements are provided by the place and route software. This method allows the device to function while the configuration is checked.

## 5.3 High level design

The features required for the different applications were compared with possible designs, before one using a CPLD as the central component of the system was selected.

### 5.3.1 Requirements

The following requirements were identified:

- It is desirable for the configuration controller to be used with *different devices*.
- The configuration controller requires *non-volatile memory* to store the configuration data for the FPGA. The size of the configuration bitstream is 1047616 bits for an XC3S200, but can increase up to 13271936 bits for the largest devices.
- For normal configuration purposes, this data is loaded into the FPGA, but to simulate SEUs, any *single bit needs to be accessed and flipped*.
- The *speed* at which the FPGA is configured is of great importance, as this process is repeated for every bit in the bitstream. A parallel interface with the FPGA greatly improves the speed.

### 5.3.2 Possible designs

Various designs that could fulfill these requirements were considered.

A microcontroller that reads configuration data from flash and presents it to the FPGA, with the necessary control and clock signals, is one solution (Figure 5.4). This design has the benefit that all data handling is done inside the microprocessor, with the minimum of other components, thereby simplifying programming and control. However, the number of pins to implement a 20-bit address bus, two 8-bit data buses, plus an additional 8-bit bus for data, as well as the various control and clock signals required

Stellenbosch University http://scholar.sun.ac.za


by the FPGA and flash, made this configuration unrealistic. Even when an external address decoder is used, very few low cost devices exist that can provide the required pin-out. A high-performance processor is also needed to provide the desired performance.

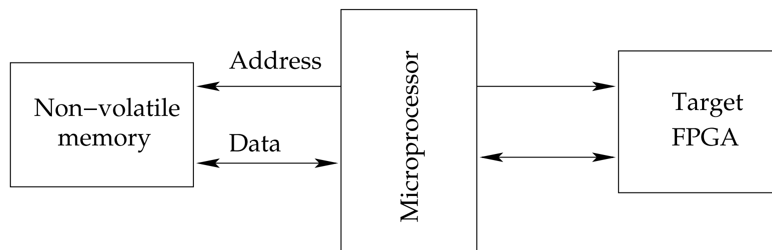



**Figure 5.4:** Microprocessor-centric configuration controller design. The required number of pins on the microprocessor makes a high performance version of this implementation unrealistic.

A Xilinx application note[52] describes how to use a Motorola microprocessor to upload the configuration bitstream to an FPGA, through a CPLD. The bitstream is stored on external flash memory, while the microprocessor, CPLD and flash share an address and data bus as shown in Figure 5.5. The speed of this design is again limited by the microprocessor, since configuration can only take place as quickly as the microprocessor can provide address and clock signals.

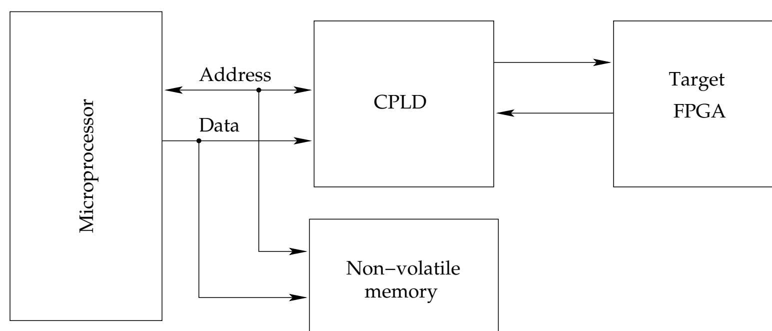



**Figure 5.5:** Using a microprocessor to configure a FPGA. The microprocessor provides control and address signals to the flash memory and CPLD to perform configuration (adapted from [52]).

The design that was eventually selected for implementation, improves performance by using the CPLD as the central component in the design. The CPLD receives commands from the microprocessor, reads and writes the non-volatile memory and performs the configuration functions on the FPGA. By providing the CPLD with a higher speed clock than the microprocessor, configuration can take at place at high speed. This design has the added advantage of easing the load on the microprocessor. This is achieved by encapsulating the configuration functionality in the CPLD, thereby allowing any

Stellenbosch University http://scholar.sun.ac.za


microprocessor to control the configuration sequence. The final block diagram is shown in Figure 5.6.

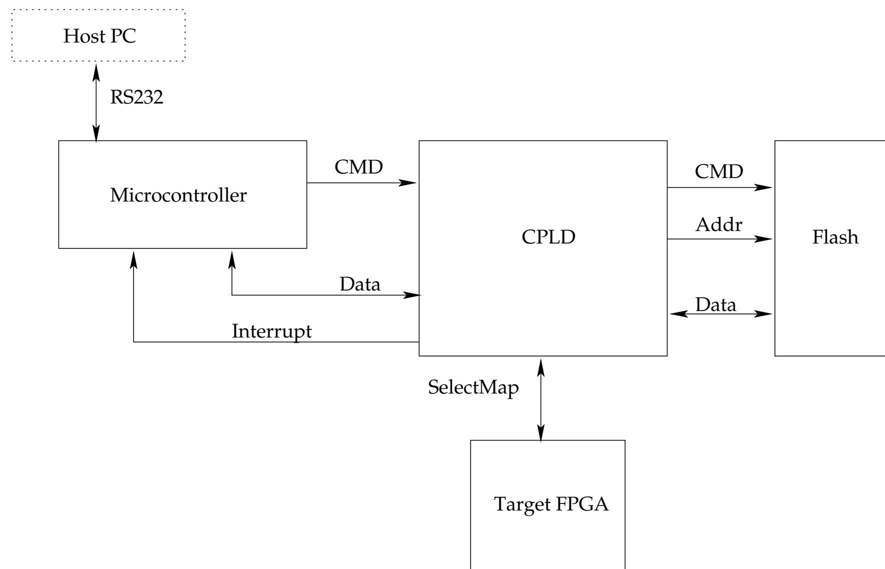



**Figure 5.6:** Block diagram of configuration controller. This design uses the CPLD to implement high speed configuration of the FPGA.

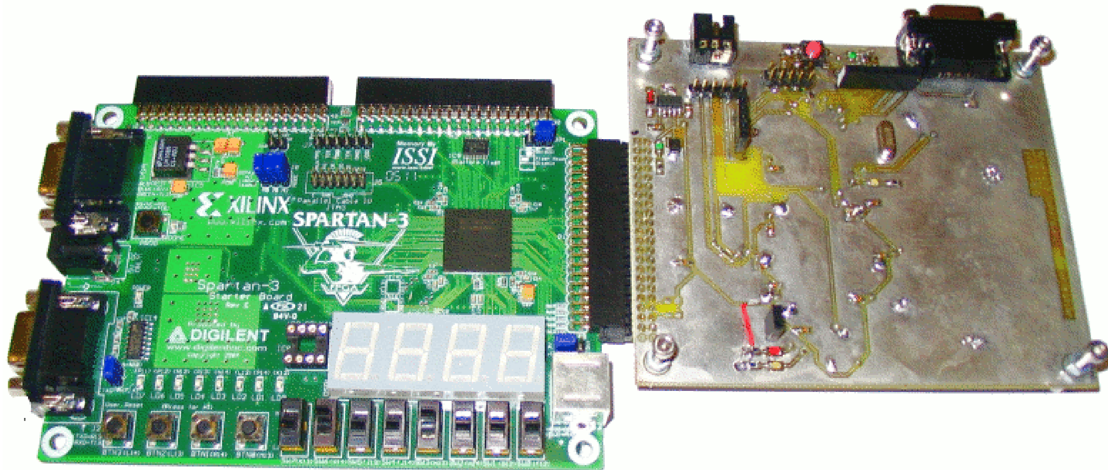



**Figure 5.7:** Docked S3Kit (left) and configuration controller (right).

## 5.4 Detailed design

In this section a detailed description of the major components and design considerations of the configuration controller is given. Although the components are discussed

Stellenbosch University http://scholar.sun.ac.za


independently, they are highly interdependent and therfor require interdependent selection and design.

The schematics for the configuration controller are given in Appendix C. A printed circuit board was laid out using `Altium DXP` and manufactured at University of Stellenbosch in-house facilities.

### 5.4.1 System voltage

The FPGA needs 2.5V signals on the configuration port. It is also possible, however, to use 3.3V signals if precautions are taken to prevent damage to the FPGA. Alternatively, a level translator can be used on the configuration bus to protect the FPGA.

It was decided that 3.3V should be used as the system voltage, since all the major components could be found in a compatible version.

### 5.4.2 CPLD

The CPLD forms the core of the configuration controller. Most of the high-speed configuration functionality is contained in its logic.

#### Device selection

A Xilinx Coolrunner XPLA3 was selected, because it requires a 3.3V and the target FPGA was from the same manufacturer. This allowed tool chain for the FPGA could be used “as-is“ for the CPLD, which saved a lot of development time.

To determine the required capacity of the CPLD, most of the design was implemented in VHDL, synthesised and fitted to a device. The XCR3384XL[53], with 384 macrocells, comfortably contained this preliminary design, with enough space for additions and modification. In addition, XCR3384XL is the largest Xilinx CPLD available in thin quad flat pack (TQFP) packaging. The Xilinx TQFP devices use a 10 mil pitch, which is the limit of what can be used on an in-house manufactured printed circuit board.

Further modifications and extensions to the logic eventually filled up the CPLD, showing that a smaller device would not have sufficed.

#### Connections

The CPLD has a 20-bit address bus connected to the flash memory as well as a 16-bit data bus and the necessary control signals.

The slave-parallel connections to the FPGA include an 8-bit data bus (`FPGA_DATA`), a configuration clock (`CCLK`) and control signals (`FPGA_DONE`, `PROGB`, `INITB` and `CSB`). The

Stellenbosch University http://scholar.sun.ac.za


physical connection to the FPGA is made through a 2×20 right angle header, as provided by the S3Kit.

The connections with the microcontroller are described below.

The design was supposed to use an external 20MHz oscillator as source for the high speed clock (CLK), but due to component availability, a 12 MHz oscillator was used instead.

### 5.4.3 Microcontroller

The microcontroller receives data and commands from a host PC and then commands the rest of the configuration controller correspondingly. Most of the processing is done by the CPLD, which leaves very few requirements for the microcontroller to satisfy.

#### Device selection

The MicroChip PIC architecture and the AVR from Atmel were considered as low-cost candidates.

The Atmel ATmega 128L[54] was selected. It has an 8-bit RISC architecture with 128KB of in-system programmable flash memory, 4KB RAM and 4KB EEPROM. It has enough input/output pins to provide the CPLD with 16-bits data, control signals and external interrupts. The microcontroller has enough RAM and flash memory that prototyping of the FPGA configuration algorithms can be done in software, before porting them to the CPLD.

Open-source software was used exclusively in the development and programming of the device. `avr-gcc`[1], which is part of the high quality GNU Compiler Collection, was used as `C` compiler. Several open-source downloading tools were evaluated, `avrdude`[2] was used extensively.

The opportunity to learn from using a new architecture (as the AVR was, unlike the PIC) was also attractive.

#### Connections

The microcontroller has a 5-bit command bus (`CPLD_CTL`), a 16-bit data bus (`MICRO_DATA`) and a clock/write signal (`SYS_CLK`) to the CPLD. Seven miscellaneous pins also connect the two devices; two of these can be used as external interrupts. The use of these connections is discussed in Section 5.5.

[1]http://gcc.gnu.org
[2]http://savannah.nongnu.org/projects/avrdude

Stellenbosch University http://scholar.sun.ac.za


The UART is connected to an RS-232 driver (MAX3232 from Texas Instruments). The analogue-to-digital converter pins are connected to a header, for use in current monitoring.

A 7.372 MHz crystal provides the clock signal.

### 5.4.4 Non-volatile memory

Flash memory is used as non-volatile storage for the configuration bitstream. The data is read from the flash in order to configure the FPGA.

The storage space requirement necessitated the use of flash over EEPROM. At least 4 time as much space as the size of the configuration file was required (see below). A 16-bit data bus to simplify readback and verification was also desirable. In addition, the device should preferably be byte or word addressable, so that a specific bit could be upset during SEU simulation. The programming time required for the device was not that important, but fast reading a priority.

Atmel NOR flash, AT49BV322D[55] satisfied all these requirements. It has a 32-megabit capacity that can be accessed as 2048 16-bit words or 4096 bytes. This is approximately 32 times more storage space than is required for the selected target FPGA (configuration file size for XC3S200 is 1047616 bits), but allows the configuration controller to be used with any device up to an XC3S2000 (2000000 system gates with a 7673024 bit configuration file), at very little extra cost.

The flash memory has a read access time of 70ns, which allows configuration to take place at up to 14.3 MHz, if a single byte is read per cycle.

The use of a NAND flash device was also considered because it has a high capacity and fast access times. NAND flash mostly supports reading in blocks. The overhead that would be required to keep track of byte addresses when reading small blocks of data was deemed to be too much to justify the performance increase.

#### Memory organisation

Due to the relatively limited resources on the CPLD, the non-volatile memory usage was arranged to minimise the logic and memory required to perform configuration and readback.

The memory was organised in such a way that the lower 8 bits can be connected to the FPGA (Figure 5.8). The configuration and readback data is stored in this section. The mask file is stored in the upper 8 bits, parallel to the readback data. This allows a single address access to retrieve all the required information to check the configuration memory. Although this approach does not utilise the flash completely, it simplifies the logic required to perform readback and verification of the FPGA.

Stellenbosch University http://scholar.sun.ac.za

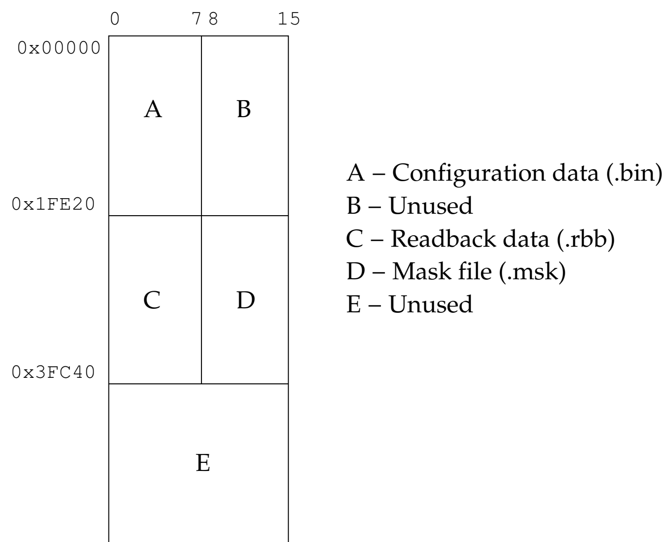




**Figure 5.8:** Usage of flash memory to store configuration and readback data. By storing the readback and mask data in parallel, upset detection can easily be performed.

### 5.4.5 Interface to FPGA

The connection of the configuration controller board to the FPGA required special care, since it connects a board with 2.5V logic to one with 3.3V logic.

A Xilinx application note describes this interface[56]. Series resistors have to be inserted on the dedicated configuration pins to account for the voltage drop, while pull-up resistors are used to increase the noise margin. In addition, the regulator on the Spartan side needs to be able to handle reverse current, or a parallel resistor needs to be inserted. Figure 5.9 shows the required connections for slave-parallel configuration.

Careful tracing of connections was required to ensure compatibility, because the target FPGA was already mounted on a populated board.

The standard configuration uses LVCMOS25 (2.5V swing, 12mA drive and fast slew-rate) for all dedicated configuration signals. Most of the resistors required for the connection was already on the S3Kit board; only the series resistors on `PROGB` and `CCLK` were needed.

## 5.5 Programmable logic and software

The CPLD is primarily responsible for the high speed configuration of the FPGA using data stored on the flash memory. It is also used to program the flash and perform readback on the FPGA.

It receives instructions that are synchronised to a clock signal from the microcontroller (`sysclock`). The instructions may initiate a state machine that is clocked by an external,

Stellenbosch University http://scholar.sun.ac.za


**Figure 5.9:** 3.3V configuration of a Spartan-3 device in slave-parallel mode, showing the additional connections and resistors required.

high speed oscillator (`clk`). This design allows the CPLD to operate much faster than the microcontroller. Consequently most functions do not require handshaking between the components.

The most important functions are discussed in this section. Figure 5.10 show the connections between the components.

### 5.5.1 Erasing the flash memory

The flash device needs to be erased before it can be programmed.

When the AVR receives the erase instruction ("`{`"), it commands the CPLD to reset the flash by pulling down the `flash_nrst` pin. This is done to place the flash device in a known state, which prevents the accidental storage of the erase instructions. The microcontroller then sends the erase command to the CPLD, which executes a series of write instructions that will initiate the erase sequence on the flash. These instructions are given by a state machine that is driven by the high speed clock (`clk`).

The `flash_nbusy` pin is pulled low by the flash for the duration of the erase process (approximately 25 seconds). The CPLD relays the `flash_nbusy` signal to the microcontroller through the `cpld_nbusy` pin. The microcontroller polls the busy-state until the pin goes high again, whereafter it notifies the host PC.

Figure 5.11 shows a sequence diagram of the erase sequence. The diagram shows the

Stellenbosch University http://scholar.sun.ac.za

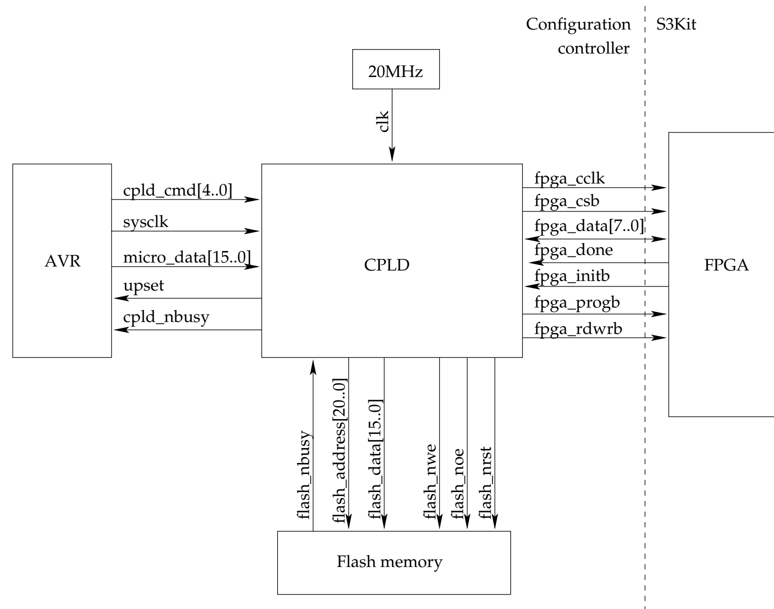




**Figure 5.10:** Block diagram showing connections in the configuration controller. The signals used in the text are shown.

activity of the system over time – it should be read from top to bottom. The host PC is an "actor": it is the device that initiates the sequence. Messages are passed between the components, as indicated by the arrows.

It was found that sequence diagrams provide a better description of the distributed processing and message passing that happens in the system. They are therefore used throughout this thesis to document the functioning of the configuration controller.

### 5.5.2 Programming the flash memory

Configuration data is downloaded to the configuration controller via an RS-232 link. The microcontroller interprets the data it receives according to the byte offset, as shown in the packet structure in Figure 5.12.

The "`f`" identifies the command as a configuration file download. The microprocessor responds by resetting the flash memory and preparing it for programming. A state machine on the CPLD writes instructions to the flash device to set it into "Single Pulse Programming mode". This mode allows data to be stored using a single clock pulse, which greatly simplifies the storage process.

The microcontroller stores five indices that are used during configuration and readback of the FPGA. The values of these indices depend on the configuration data; they are therefore set during the programming of the flash.

The configuration data is defined to start at address 0 in the flash memory, therefore no starting index is stored. The first index (A1 in Figure 5.12) provides the last address of

Stellenbosch University http://scholar.sun.ac.za

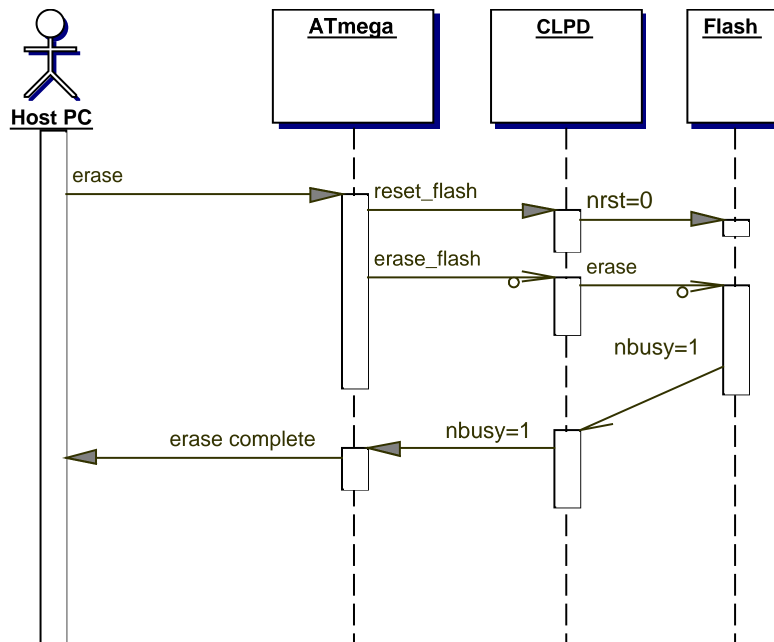




**Figure 5.11:** Sequence diagram describing the flash erase operation.

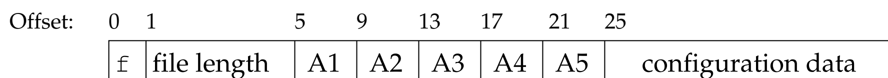



**Figure 5.12:** Structure of packet used to program flash memory. The "`f`" identifies the command as a configuration file download. The "file length" field specifies the length of teh configuration data. The indices A1 to A5 are used in configuration and readback.

the configuration bitstream. A2 is the start address that is used during readback. The first section of the readback data consists of instructions that are written to the FPGA. A3 indicates where these instructions end and the readback data starts. A4 points to the end of the readback data. The instructions required to restart the FPGA are in the range of A4 to A5. These indices are stored in the internal EEPROM of the microcontroller which allows the device to maintain these values, even if power is removed.

The final segment of the programming packet contains the actual data that is stored on the flash. The microcontroller waits until it has received two bytes and then the data is then written to the flash memory via the CPLD. The *file length* field in the packet tells the microcontroller how much data can be expected. The flash device is reset to exit the Single Pulse Programming mode when all the data has been received.

Figure 5.13 shows the sequence of events executed to program the flash memory.

### 5.5.3 CPLD registers

The CPLD maintains three registers that are used during configuration and readback.

Stellenbosch University http://scholar.sun.ac.za

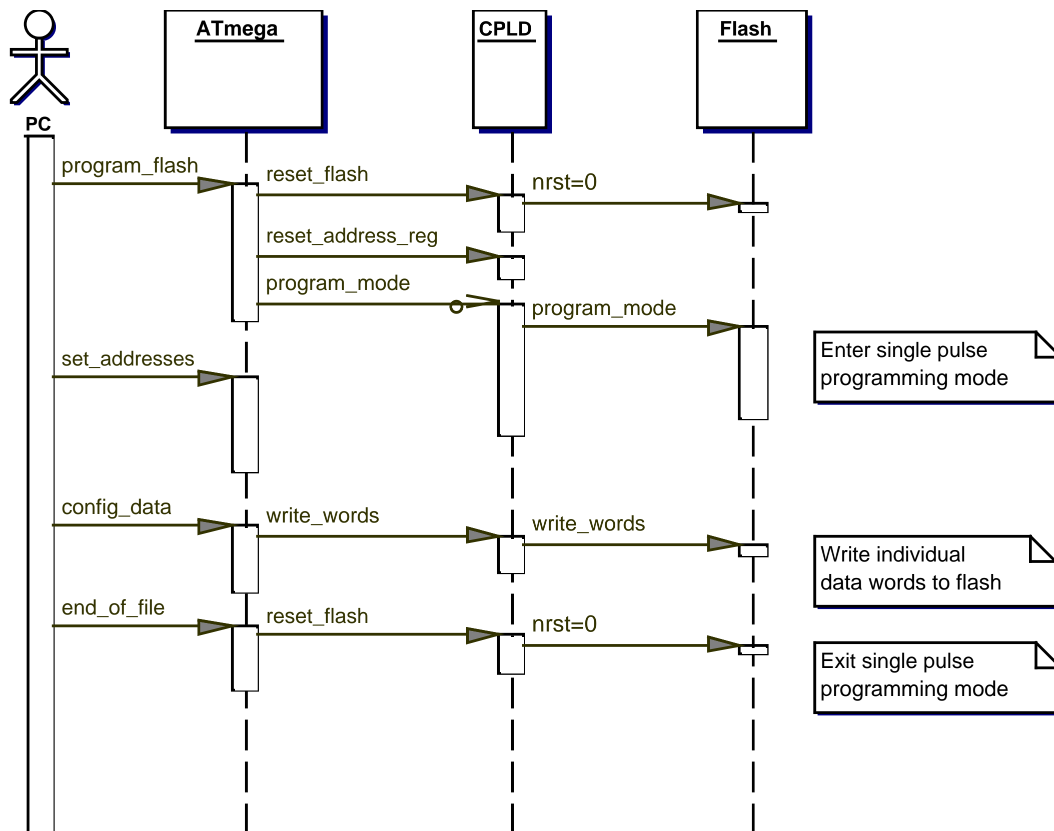




**Figure 5.13:** Sequence diagram for programming the flash memory.

- The *Address Counter* is used to provide a 20-bit address signal to the flash memory. It can be incremented at half the frequency of the `CLK` signal.
- The Address Counter is incremented during configuration until it equals the *Stop Address*, also a 20-bit value.
- A 23-bit *Upset Address* is used to specify a bit that needs to be upset during configuration error injection (Chapter 6). The upper 20 bits are used to match the address of a byte and the lower three bits are decoded to provide the index of the bit to flip.

The registers are set by writing two 16-bit words from the microcontroller to the CPLD. The first byte sets the lower 16 bits; the second the remaining bits. This requires a state machine that is synchronised to `sysclk`.

### 5.5.4 FPGA configuration

The FPGA is configured by loading the configuration data from the flash memory into the FPGA through the SelectMap port. The microcontroller intiates this sequence upon receipt of the "Configure" command ("`R`").

Stellenbosch University http://scholar.sun.ac.za


The fpga_progb pin of the FPGA is pulled low for one clock cycle to reset the device and prepare it for configuration.

The CPLD registers are initialised to specify the data range that is used for configuration. The Address Counter is set to 0 by resetting the CPLD and the Stop Address is set to the end of the configuration bitstream (A1 in 5.12).

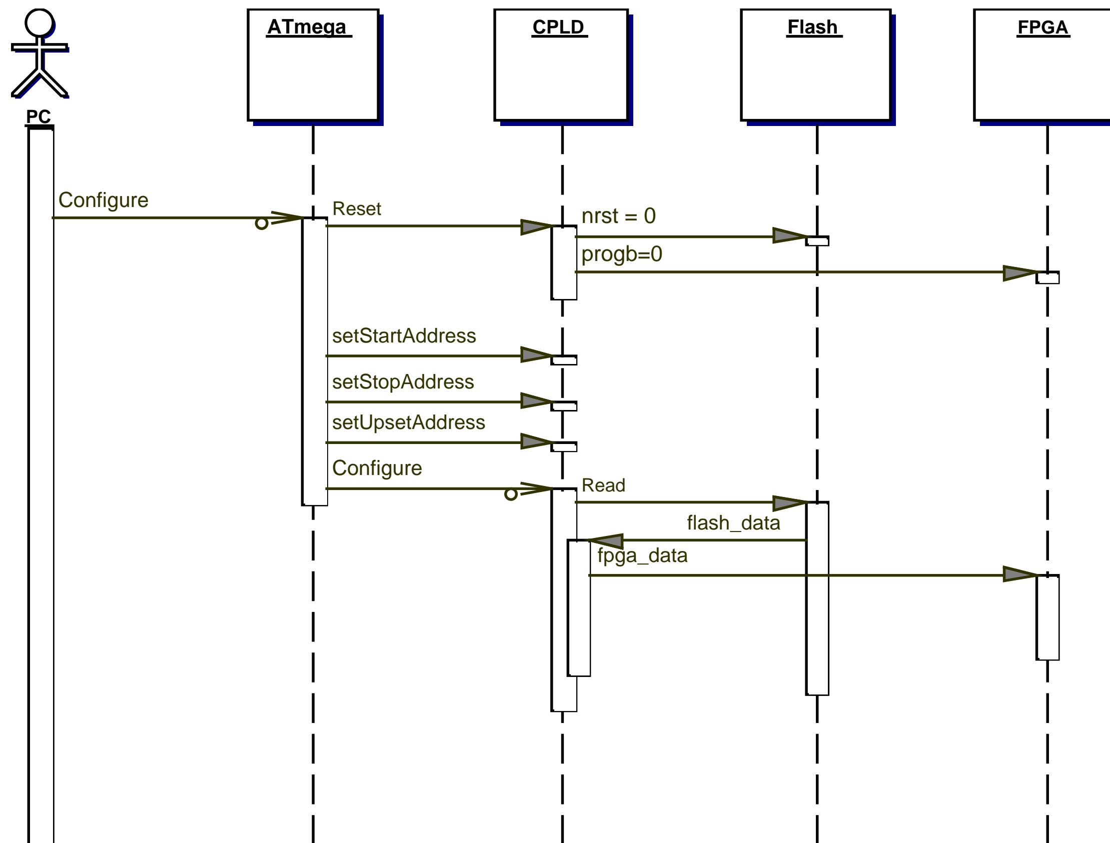



**Figure 5.14:** Sequence diagram for configuring the FPGA.

The CPLD then reads data from flash memory by incrementing the Address Counter until it equals the Stop Register. The lower 8 bits of the data is presented to the FPGA (syncronised to fpga_cclk). This transfer is performed by a high speed state machine, which allows data to be written to the FPGA at 6MHz (half clk frequency). The upset register is set to 0xFFFFF to perform configuration without injecting any upsets. SelectMap does not require handshaking if the configuration speed is below 50MHz, therefore it was not implemented.

Configuration takes approximately 21.8ms when all the overheads are taken into account.

Stellenbosch University http://scholar.sun.ac.za


### 5.5.5 Readback verification

Readback is performed by sending the microcontroller an "`N`". The readback flow, as shown in Figure 5.2 is controlled from the microcontroller by giving commands to the CPLD.

The readback instructions and data stored on the flash memory are composed of three sections. The first section consists of instructions that initiate the readback process. The address counter and stop address registers are set to A2 and A3 respectively to specify the instruction address range (Figure 5.12). These instructions are written to the FPGA by using the same writing logic as described in the previous section.

The microcontroller then sets the readback data range (from A3 to A4). The `fpga_rdnwr` pin is pulled high to allow data to be read from the FPGA. Data is then simultaneously read from the flash memory and the FPGA. The upper 8 bits of the flash data are used to mask the data read from the FPGA. The result is then compared with the lower 8 bits of the flash data to find an upset. If an upset is found, the `upset` interrupt pin on the microcontroller is driven high. The connections during verification are shown in Figure 5.15.

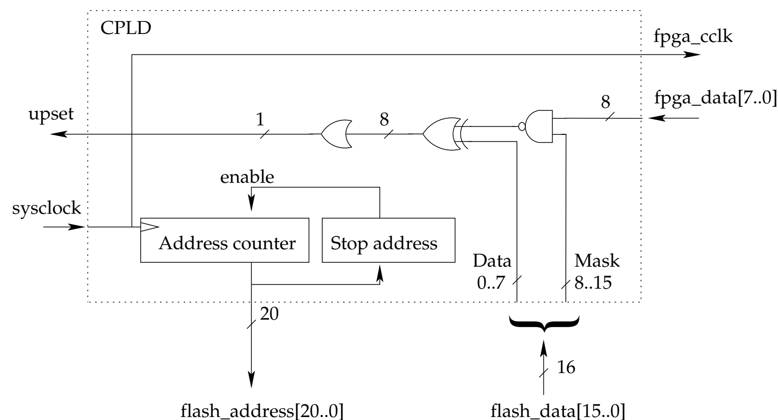



**Figure 5.15:** Diagram showing the connections for readback and verification on the CPLD

The final part of readback verification involves restarting the FPGA. The `fpga_rdnwr` pin is pulled low and instructions are read from A4 to A5 in the flash memory.

This upset detection scheme only indicates if an upset is found. To determine the location, the readback flow must be interrupted so that the current value of the address register can be read.

### 5.5.6 Control software

The host PC communicates with the configuration controller through the serial port. A program like `minicom` (Linux) or `telnet` (Windows) can be used to do most of the

Stellenbosch University http://scholar.sun.ac.za


communication with the device. A control program `confctl.py` was especially written to provide an accessible interface to the configuration controller.

The script takes command-line parameters, interprets them as instructions and data, in order to transmit the correctly formatted commands. This is especially important where the microcontroller expects binary data, as when programming the flash memory or setting configuration addresses. The script also performs byte-swapping on the configuration data.

Two options are available when downloading the configuration data to the flash memory. One can either download only the bitstream file (`.bin`), or the bitstream, readback and mask data need to be transmitted. In the latter case, it is necessary to generate an image file (`.img`) first that can be downloaded to the flash.

This image file is generated by `combine.py`, which arranges the configuration, readback and mask data as shown in Figure 5.8. A plain text file (`.idx`) with the addresses of the end of the configuration data, start of readback instructions, start of readback data, end of readback data and end of readback instructions indices is also generated. These files are used to construct the packet structure discussed in Section 5.5.2.

## 5.6 Conclusion

The configuration controller was sucessfully constructed. The availability of a configurable platform that can implement advanced configuration functionality greatly simplified the development of a radiation tolerant PicoBlaze.

The use of a similar design to perform operations on the configuration memory of SRAM devices is highly recommended in a space environment, because it allows the reconfiguration and reprogramming abilities of these devices to be utilised to their full.

### 5.6.1 Configuration controller

The platform is versatile enough to configure most Xilinx devices that support the SelectMap (parallel-slave) interface, without modification.

The load on the microcontroller is decreased substantially, by encapsulating most of the configuration and readback logic in the CPLD. This allows any processor that is already in the system to control the configuration functions.

The configuration controller provides one with considerable freedom in uploading different designs onto the same FPGA. The flash memory can be used to store an SEU protected and an unprotected design. If the spacecraft is in an area where upsets are likely (eg. the SAA or over the poles), then the protected design can be loaded onto the FPGA. The unprotected, lower power design can be used for the rest of the orbit.

Stellenbosch University http://scholar.sun.ac.za


The high reconfiguration speed keeps the interruption to the rest of the system at to minimum. The same principle of multiple designs on one device can be applied to completely different designs, where only the currently needed design is instantiated.

### 5.6.2 Readback

The benefits of using readback as part of an SEU mitigation scheme are limited on Spartan-3 FPGAs. Readback of the entire device takes as long as reconfiguration. The device needs to be shut down to prevent data corruption, which results in a similar downtime as simple scrubbing would.

It is therefore recommended that periodic scrubbing is used to correct upsets. This will required vital data on the FPGA to be checkpointed to minimise the disruption.

FPGAs in the Virtex family do not suffer from the same limitations as the Spartan-3 does: readback with partial reconfiguration can be used to protect the configuraton.

### 5.6.3 Limitations

The readback verification function was implemented, but not fully debugged due to time constraints.

The programmable logic interface to the FPGA has been verified by simulation with `ModelSim` and by tracing the signals in the hardware. The problem appears to lie in the configuration commands written to the FPGA. A very specific sequence of commands is required to initiate readback. This sequence also needs to be synchronised to the configuration logic in the FPGA.

Stellenbosch University http://scholar.sun.ac.za

# Chapter 6

# Single Event Upset Simulator

The configuration controller was used to construct an error injection tool-set that could simulate the effects of SEUs in the configuration memory of an FPGA.[1]

At the highest level of abstraction, the simulator consists of a reference design that exhibits the correct behaviour and a design with injected errors in the configuration memory. The operation of the two designs is compared to determine if the injected errors influenced the functionality of the design.

The main objective of the simulator was to provide a tool that could empirically help predict the dynamic cross-section of a given HDL design, without the cost and administration involved in having to resort to actual radiation testing.

Once sufficient data has been gathered about the properties of different techniques, a mitigation strategy can be selected that combines the most suitable techniques with the properties of the design.

## 6.1 Background

The SEU simulator exploits the ease by which SRAM FPGAs can be reprogrammed with a different configuration. By flipping individual bits in the configuration bit stream, SEUs in the configuration memory are be simulated. This is done by using the configuration controller, but instead of configuring the FPGA as usual, a single bit is flipped before it gets loaded into the device. Not all the configuration bits are used in a design. The SEU simulator can indicate the number and location of the bits that will cause the design to fail.

As stated in Chapter 4, the configuration memory and BRAM make up the largest segment of the SEU cross-section. Both can be corrupted by using the SEU simulator, providing a good approximation of the scaling factor ($\alpha$) for these components.

[1] As used in this document, "SEU simulator" refers specifically to this hardware and software error-injection tool-set.

Stellenbosch University http://scholar.sun.ac.za


A deeper knowledge of the configuration memory is required to interpret the SEU simulator results. The discussion here is specific to Xilinx Spartan-3 devices, but can apply (with minor modifications) to other Xilinx devices.

### 6.1.1 Device architecture

All FPGAs have non-programmable areas such as the configuration logic and boundary scan logic, as well as programmable areas that configure the routing and look-up tables used to implement logic.

The Spartan-3 is composed of "Input/Output Blocks" (IOBs), "Configurable Logic Blocks" (CLBs), a "Digital Clock Manager" (DCM) and an interconnecting routing and clock network. Specialised hardware resources, such as hardware multipliers and Block RAM, are also available. The organisation of these elements is shown in Figure 6.1. The IOBs are placed on the edge of the device, surrounding the CLBs. The CLBs are connected by a rich network of traces and programmable traces.

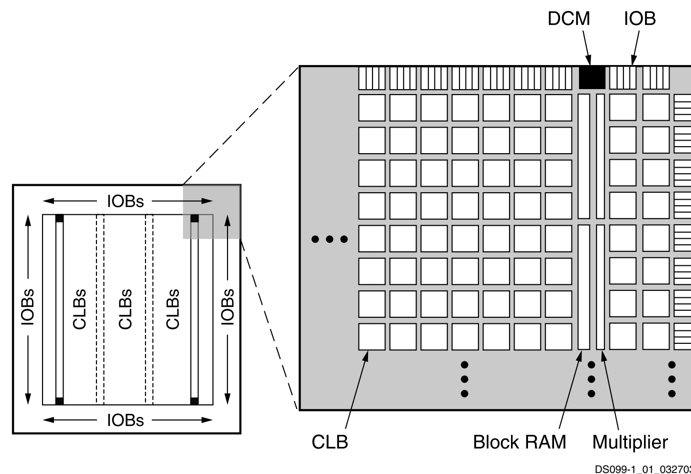



**Figure 6.1:** Spartan-3 family architecture.[50]

These elements are programmed by setting bits in the configuration memory. The configuration data needs to be reloaded on start-up, because the configuration memory is implemented using volatile SRAM cells.

### 6.1.2 Configuration memory

The configuration memory of the Spartan-3 can be visualised as a rectangle of bits. These bits are organised into frames that are one bit wide and extend from the top to

Stellenbosch University http://scholar.sun.ac.za


**Table 6.1:** Spartan-3 XC3S200 Column Types. The entries marked with * are device dependent.[51]

| Column type | Number of frames per column | Number of columns per device | Column address |
|---|---|---|---|
| TERM(L/R) | 2 | 2 | 00 |
| IOI(L/R) | 19 | 2 | 00 |
| CLB | 19 | 20* | 00 |
| BRAM | 76 | 2* | 01 |
| BRAM interconnect | 19 | 2* | 10 |
| GCLK | 3 | 1 | 00 |

the bottom of the configuration memory. A frame is the smallest portion of the configuration memory that can be read from or written to.

Frames are grouped in columns depending on the hardware they configure. The hardware configured by a frame is not limited to that implied by its name. For example, certain input/output blocks (IOBs) are configured in frames that primarily configure CLBs. In addition, frames do not map to a single piece of hardware, but rather configure part of several logic resources, as well as do some routing.

Table 6.1 lists the different types of columns. The TERM columns configure the output standards of pins on the left and right of the device. Those along the top and bottom are configured in the CLB frames. The IOI columns supply additional information pertaining to registers, multiplexers and buffers in the IOBs. The CLB columns provide configuration of the CLBs, as well as all routing and interconnects (other than the global clock trees). The Block RAM initialisation data is contained in the BRAM columns, while the BRAM routing information is in the BRAM interconnect columns. The GCLK column contains DCM attributes and global clock buffer information.

### 6.1.3 Configuration bitmap

A configuration bitmap; a graphical representation of the configuration bitstream, was found to be an extremely useful tool in the application of error mitigation techniques.

A program, `bin2im.py`, that can convert binary configuration files to images was written as a specialist tool. The program generates a rectangular bitmap, where black pixels correspond to bits that are set to 1, while 0 bits are grey. The XC3S200 configuration consists of 615 frames of 1696 bits each, which results in a $1696 \times 615$ pixel image.

The locations of the major column types in Table 6.1 were found by using this program to inspect the bitstream of various designs. In addition, the program also provides a means of visually inspecting the density and distribution of a design. Figure 6.2 shows a sample of the output for the PicoBlaze reference design discussed in the following chapter. The different column types are shaded and major features have been marked to aid in interpretation.

Stellenbosch University http://scholar.sun.ac.za

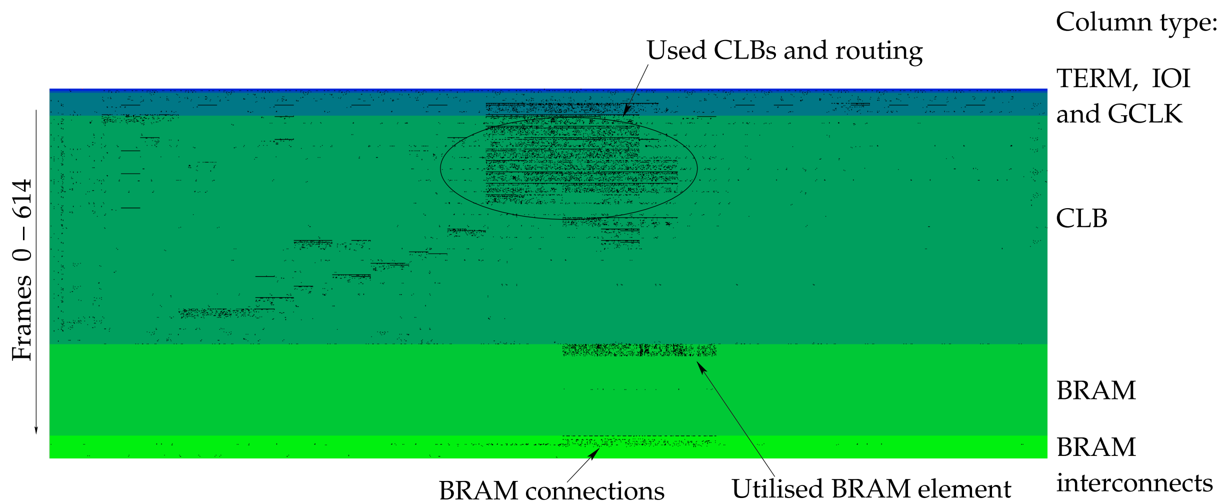




**Figure 6.2:** Annotated configuration memory bitmap for Spartan-3. Bits that are set to 1 in the configuration memory are represented by black pixels. The different types of columns are shaded and the major features marked. The image is rotated through 90° to save space: the frames are on horisontal lines from top (frame 0) to bottom (frame 615).

It is also interesting to note the correspondence between the physical layout of a design and the configuration memory. Figure 6.3 shows the layout, as produced by Xilinx `FPGA Editor`. The major features have again been pointed out. Note that the shape of the physical layout corresponds to the shape of set bits in the CLB section of the configuration bitmap, while the set bits in the BRAM and BRAM interconnects are located in the same area as the BRAM on the floor plan.

`bin2im.py` was also used to investigate the location of *sensitive bits*, ie. bits that influence the operation of the design when flipped. This is illustrated in Chapter 7.

Unfortunately, no way exists to identify the exact function of all the individual bits, as it is proprietary information. A bit level manipulation library, `JBits`, does exist, but only Virtex-II and Virtex-4 devices are supported.

## 6.2 Similar projects

Only a handful of SEU simulators have been built. A large part of the research in this area has been performed by Xilinx, to qualify SRAM FPGAs for use in radiation environments.

### 6.2.1 Los Alamos National Laboratory

An SEU insertion tool-set was developed at the Los Alamos National Laboratory (LANL), using a Xilinx Multilinx programming cable and a Virtex FPGA.[57] This system was used by Xilinx to verify that no single bit upset in a fully triple redundant system would

Stellenbosch University http://scholar.sun.ac.za

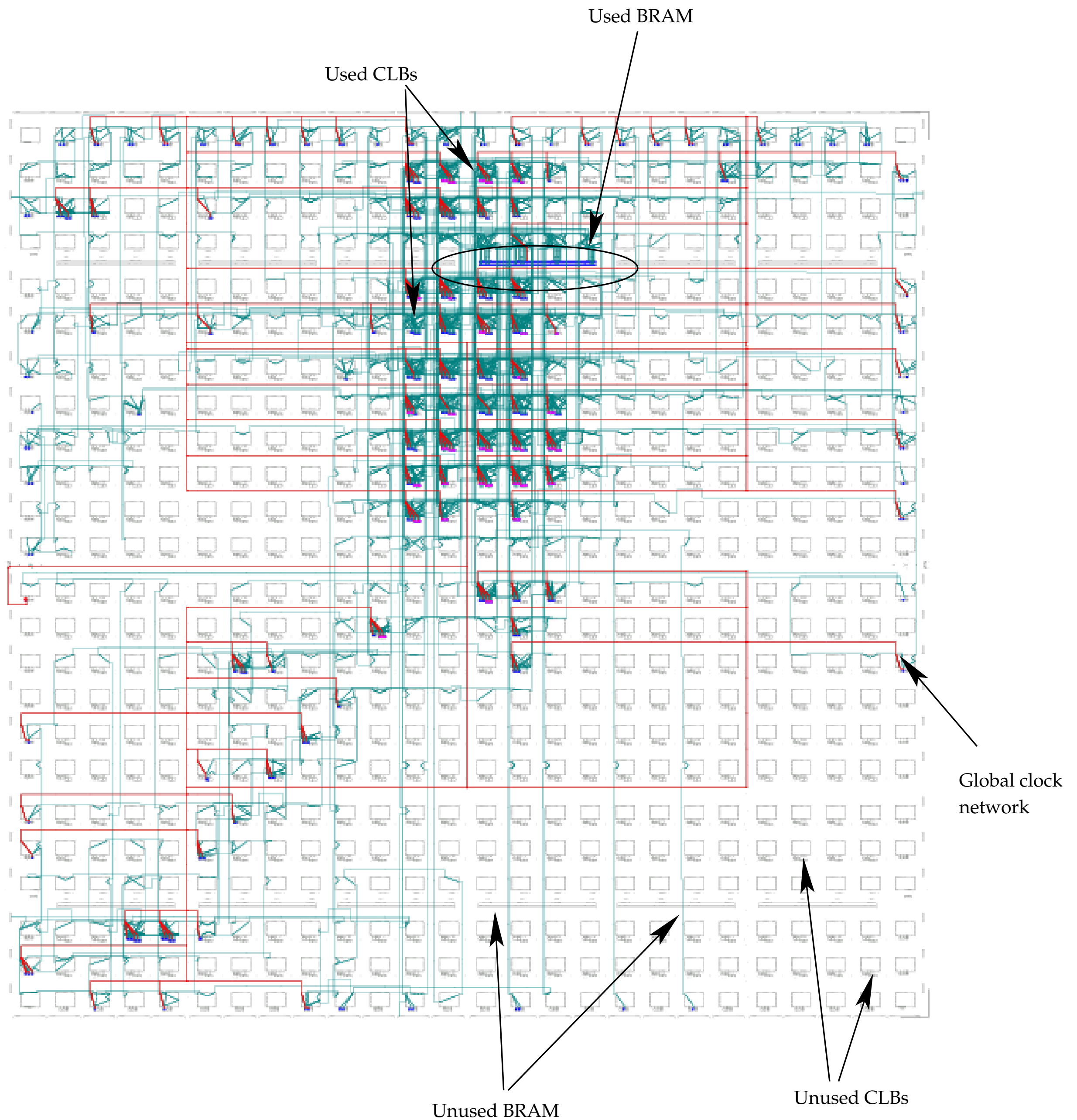




**Figure 6.3:** Annotated floor plan of design in Figure 6.2. The major features have been marked. Note the similarity between the used CLBs and the distribution of the set bits in the configuration bitmap. This image is rotate to correspond to the configuration bitmap.

Stellenbosch University http://scholar.sun.ac.za


influence the system behaviour.

### 6.2.2 Brigham Young University

The FPGA Reliability Studies group at Brigham Young University (BYU) has constructed an single event upset simulator to characterise the reliability of FPGA designs, in the hopes of developing efficient techniques to mitigate the effects of SEUs.[57]

The system is based on the SLAAC-1V reconfigurable computing board. As can be seen in Figure 6.4, the board has 3 Virtex XCV1000 FPGAs (X0, X1 and X2) connected via a three port crossbar, ZBT SRAM, and a PCI bus interface.

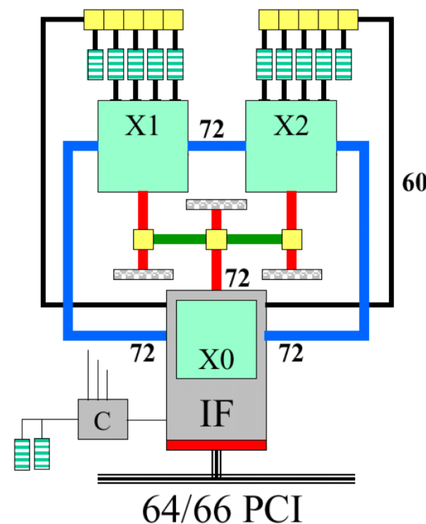



**Figure 6.4:** SLAAC-1V reconfigurable computing board.[57]

The C block in the figure is the configuration controller. It uses an Virtex XCV100 to provide dedicated high speed reconfiguration on any of the other FPGAs.

Testing involves loading correct designs into X1 and X2. The first bit in the configuration stream is flipped through partail reconfiguration, and the frame containing it is reloaded into X1. X0 is used to monitor the outputs of the other two FPGAs. Any discrepancies are reported to the host machine through the PCI bus. The bit is then repaired, in order to restore X1 to its original state. The test is repeated for every bit in the configuration stream.

The greatest advantage of using the SLAAC-1V board is the very high speed at which upsets can be tested. A single loop requires only 267us because only one frame is reconfigured at a time. This means that testing the entire configuration memory requires just under 27 minutes (for 5,962,944 configuration bits).

The simulator has been thoroughly verified by radiation testing.

Stellenbosch University http://scholar.sun.ac.za


### 6.2.3 Politechnica di Torino

A fault injection environment was developed at the Politechnica di Torino by using `JBits`, a Java library for manipulating the bitstream of Xilinx Virtex devices.[58]

`JBits` allows the flipping of bits associated with specific resources (CLBs and routing bits) on an FPGA. This allows the user to obtain information on the influence of bit-flips on specific components.

Bits used by a design are identified with `JBits` and individually flipped. The corrupted configuration file is simulated in the `VirtexDS` simulator. It is a program that uses the actual configuration file to simulate the behaviour of the device. This approach offers a very simple way of testing corrupted designs, but since the `VirtexDS` software is not optimised for speed, simulation is time-consuming, taking from hours to days. It would be possible to replace the software with hardware, to overcome this limitation.

## 6.3 Error detection

A critical part of the simulator, is the detection of errors. Several mechanisms that can perform the comparison between a reference design and a corrupted design exist.

The BYU simulator uses the "Golden Chip" method, where two identical devices are executing synchronously and in parallel. All the output are checked for any mismatch, therefore another FPGA or CPLD is required. This method detects all errors and offers high performance, but greatly increases the complexity of the system.

Two other methods were implemented and tested for this SEU simulator. The first uses a cyclic-redundancy check to perform a "virtual golden chip" test. The method that was selected for use in the simulator is based on a self-testing software routine.

### 6.3.1 CRC monitoring

One can also log the outputs of the reference test and compare this with the outputs of the design under test, instead of having a Golden Chip running in parallel to the design under test. This simplifies the system design, but still requires a way in which a large volume of data can be measured and logged.

A novel variation on this method was investigated. Instead of storing all the data, the outputs were used to compute a checksum by using a cyclic redundancy check (CRC). Although it was not used in the final simulator design the method is described here, as it is an important demonstration of automated VHDL generation and it is promising enough to warrant further investigation.

Stellenbosch University http://scholar.sun.ac.za


### Cyclic redundancy check

A CRC is a type of hash function that is calculated by using a linear feedback shift register (LFSR), as shown in Figure 6.5. Input values (D) are shifted into the register, while an XOR loop is used to calculate new values. As a result, a unique checksum is generated for a particular input series. If a bit changes between the reference design and the design under test, a different checksum is calculated. CRC codes are used extensively in communication systems, where they are used to detect burst errors.

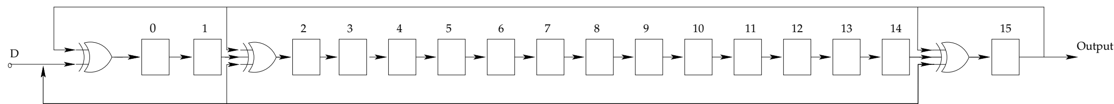



**Figure 6.5:** 16-bit CRC calculator with $(1 + x^2 + x^{15} + x^{16})$ polynomial for 1-bit data. The data bit ("D") is fed into the shift register to calculate a new chekcsum.

### Implementation

A CRC calculator was written in VHDL in order to apply CRC checking to the outputs of a design. All the outputs pins are sampled on every clock cycle, to obtain the inputs to the LFSR. After a predetermined number of clock cycles, the CRC calculator interrupts execution and raises an external interrupt. The checksum is then be read from the CRC calculator.

The only way to sample all the used output pins was to place the CRC calculator inside the FPGA, therefore it contributes to the dynamic cross-section. The CRC calculator is implemented with dual redundancy to allow the mitigation of upsets in the CRC calculator. By reading two checksums from the FPGA and comparing with the golden checksum to detect upsets, the cross-section of the CRC calculator was minimised. If the two checksums were not the same, an upset must have occurred in one of the redundant CRC calculators. If any of the two checksums matched the golden checksum, the design was assumed to have functioned correctly.

The CRC caluculator was controlled from CPLD. Upon receiving a command from the microcontroller, the CPLD would read a checksum from the FPGA and store it as a reference value. After every run, when the CRC calculator raises an external interrupt, the CPLD would again read both checksums and check for differences. The result could then be reported to the microcontroller, which would pass it along to the host PC.

A CRC with a 16-bit checksum and a 256-bit data width was implemented, based on open source code.[2] It uses the (0,2,15,16) polynomial, similar to the one in Figure 6.5. A 16-bit parity code is calculated from the data, where every parity bit checks a unique

[2]http://www.easics.com

Stellenbosch University http://scholar.sun.ac.za


set of data bits. This is XORed with the checksum and transformed according to the polynomial to produce the new checksum. Synthesis was performed using Synplify Pro. Unused data inputs were removed during the synthesis process.

The modules were implemented and tested on the S3Kit and configuration controller boards.

#### Automated CRC calculator insertion

A program called addcrc.py was developed to automate the process of adding the CRC calculator to a design. This program also provided an opportunity to evaluate the possibilities of modifying VHDL through software to insert EDAC mechanisms.

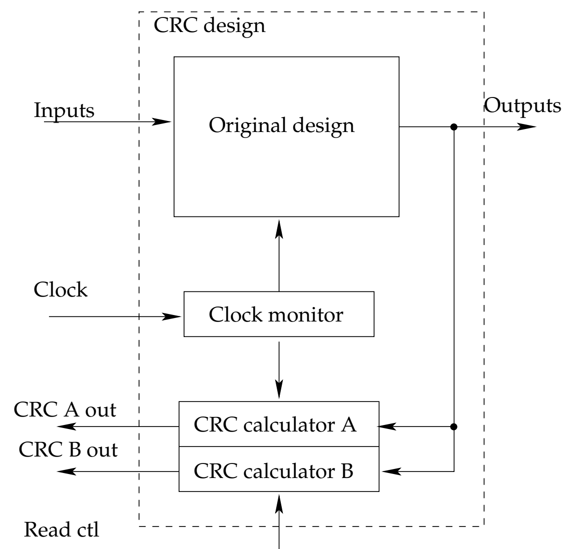



**Figure 6.6:** Design file with added CRC calculator. The clock monitor is responsible for interrupting execution after a predetermined number of clock cycles.

The program takes a top-level VHDL design file as input and generates a wrapper file to connect the design to a CRC calculator. It scans the entity declaration part for output ports, which it connects to the input signals of the CRC calculator in a new top level module. The name of the clock signal is specified as a parameter. The resultant file structure is shown in Figure 6.6.

Bidirectional ports are currently not supported, but including them will be a fairly trivial task.

The program is completely separated from the design of the rest of the system and only needs to be run right before a design is compiled.

The source for addcrc.py is listed in Appendix B.

Stellenbosch University http://scholar.sun.ac.za


#### Drawbacks

Although this method is promising, some of its drawbacks led to it it being rejected in favour of the self-test routine described below.

The greatest problem involved running tests of long duration, especially during radiation testing. The CRC monitor works well for a single upset test, but multiple upsets over a longer periods of time can lead to failed tests appearing correct.

It is very difficult to determine the cycle time of the CRC monitor, ie. the time before values in the sequence starts to repeat. Apart from computer simulation, one has no way of predicting the possibility of the checksum values matching if the inputs differed. Even a 5ms PicoBlaze test would generate 250000 samples for every output pin. It is quite possible for the hash collisions to occur, which would obscure test failures.

### 6.3.2 Self-testing routine

Another alternative is to check the system response to a set of predetermined input vectors. If the vectors are selected with care, very good coverage of the system can be achieved at very little extra complexity. This has been used to good effect on systems containing finite impulse response filters, where a given input will always result in a given output.

The instructions for a self-testing routine of a soft-core processor can be seen as such a specific input vector. Very good coverage can be obtained if the test is constructed to cover the instruction set and to utilise all the elements in the design.

A self-testing routine (described below), was designed to use all the elements in the PicoBlaze design, as well as cover most of the instruction set. The test results are reported to the host PC, which checks whether a run failed or not.

The self-testing routine can also be used during radiation testing, by repeating the test in an infinite loop.

## 6.4 System description

The SEU simulator uses the configuration controller, described in the previous chapter, to rapidly inject errors into every configuration bit. The reference design is a PicoBlaze soft-core processor that is programmed to perform a number of integrity checks. The results are monitored on a PC and the data is logged. If a design behaves differently than expected for a flipped bit, that bit is recorded as a “sensitive bit”, ie. a bit that will cause the design to malfunction when upset.

Runs are classified as having passed or failed, no distinction is made between the different types of failures.

Stellenbosch University http://scholar.sun.ac.za

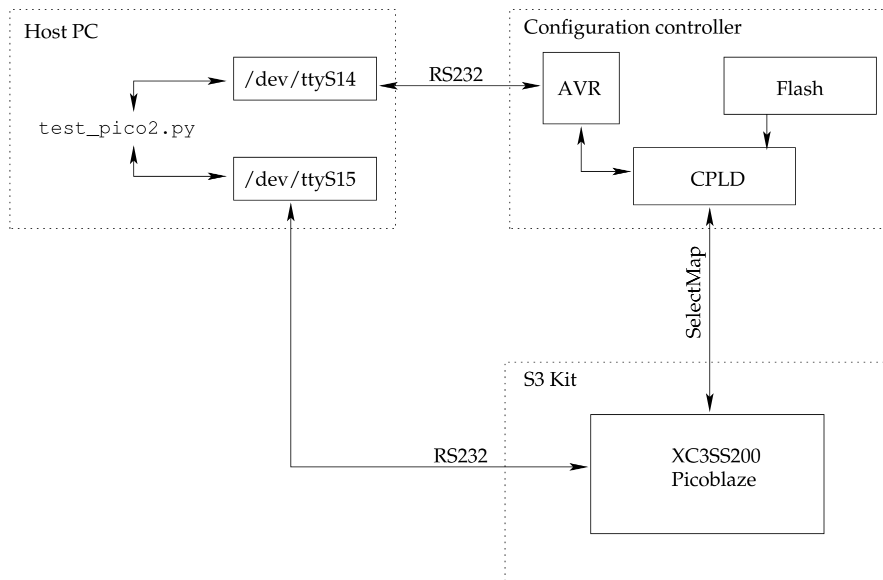




**Figure 6.7:** Block diagram of SEU simulator

A sequence diagram of a single upset sequence is shown in Figure 6.8. This is similar to the configuration sequence discussed in the previous chapter, except the "Upset Address" register is set to the bit address that is flipped.

### 6.4.1 Control software

The simulation run is controlled from the host PC, by running the `test_pico2.py` program.

This program uses `confctl.py` to configure the FPGA with a flipped bit in the first position in the configuration memory, while it monitors the PicoBlaze through the serial port. The PicoBlaze returns the results of the self-checking routine, which the host software compares to a string of expected characters. Any deviation from the expected values is logged as a failure, as the function of the PicoBlaze has been compromised. When a result for the first test has been obtained, the device is reset and the process repeated for the second bit, then the third and so on. All 1043040 bits in the configuration memory are tested in this manner.

The injected upsets may cause unexpected behaviour by the PicoBlaze. The software has a time-out function that terminates the test is the PicoBlaze fails to respond. Similarly, if PicoBlaze gets stuck in an infinite loop and keep transmitting data, the program has to reset the FPGA and clear the serial buffer on the PC.

A log file is used to store the address, result and received string for every test. To cope with the frequent power failures that were experienced during the time the simulator was used, the logged data was frequently written to disk. The simulation was split into sections of approximately 100000 bits each, with separate log files for added security.

Stellenbosch University http://scholar.sun.ac.za

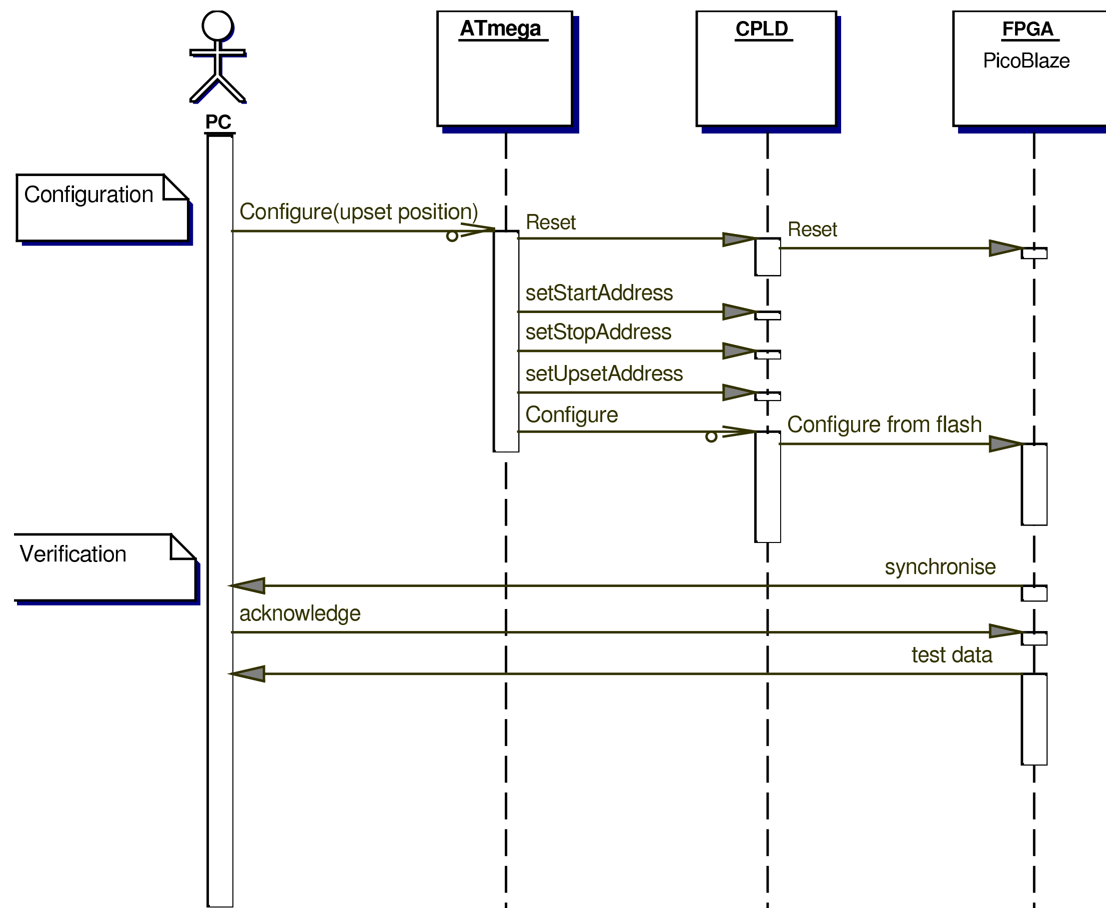




**Figure 6.8:** Sequence diagram for a single error injection operation. This sequence is repeated for every bit in the configuration memory.

### 6.4.2 PicoBlaze

A self testing program is used to verify the functionality of the implemented PicoBlaze designs.

The goal of the test routine was to mimic an actual embedded system by using components that are frequently used in such systems, such as a large number of output pins, UART and external memory. This is not an exhaustive test, in the same way that such a system would rarely be using all the components. Instead, the routine tries to detect upsets that would influence the system under normal execution conditions.

The different PicoBlaze configurations used are discussed in the next chapter.

#### Start-up

The PicoBlaze automatically starts after configuration of the FPGA. The first instructions are used to disable the external SRAM in order to prevent possible bus conflict. A value of `0xCC` is written to the LED output port to give a visual indication that the

Stellenbosch University http://scholar.sun.ac.za


system has been configured and is running.

#### Synchronisation

The PicoBlaze then sends a 32-bit synchronisation string (0xFF669955) to the host computer – this is used to set up the state machine in the control software.

#### External SRAM

The external SRAM is tested by calculating and writing values to eight different addresses in both the lower and upper SRAM chip. The values are then recalculated and compared with the read back data.

The primary interest lies in verifying the access to the external RAM, since upsets are injected into the FPGA and not the RAM chips. Although this test is not exhaustive, is does provide enough coverage of the possible failure modes for the purposes of this study, without taking up too much time.

It was therefore decided that the lower two address registers should be loaded with 1, this value is shifted to the left to obtain the next address. The data written to the RAM starts at 0x5A, this is decremented with 0x02 for every new address.

#### Timer interrupt

The timer interrupt is tested by enabling an interrupt service routine that increments a register every $1\mu$s. The program waits in a loop, until the value of the register is 0xFF, before sending a character to the host.

The test fails if the host computer does not receive the correct character before it times out.

#### UART

Although the UART has already been used in the previous tests, the PicoBlaze also transmits 0xA55A to the host.

Then the PicoBlaze blocks execution until a character has been received from the host. This character is incremented by 1, before transmitting it back.

#### Scratch pad RAM

The on-chip scratch pad RAM is tested by writing 0x5A to all the memory locations. This data is then read back and compared with the original data. The process is repeated with 0xA5.

Stellenbosch University http://scholar.sun.ac.za


As with the external SRAM, this test can be more exhaustive, firstly by checking for upsets in the memory before rewriting it and secondly by writing different bit patterns. Once again, the objective is to find upsets that influence the normal operation of the design, rather than to find all upsets.

#### CALL/RETURN

The CALL/RETURN stack of the PicoBlaze is tested using a recursive function.

The function increments a counter until the stack is full and then decrements another for every `RETURN` call it makes. Both counters are returned to the host for verification.

#### Registers

Access to the registers is tested by propagating data `0xA55A` from register `s0` to `s1`, then `s1` to `s2`, up to `s15`. If the final value differs from the initial value, an error is reported.

#### External input and output

The use of input and output pins is tested by reading and writing data to external components on the S3 Kit board.

A loop-back port was constructed by connecting 8 output pins to 8 input pins. A value is written to the output port, and the result read on the input port. The value of the 8 sliding switches on the board is also read.

Both values are reported to the host computer.

#### ALU

The arithmetic logic unit is verified by executing a sequence of ALU instructions and comparing the result with an expected value.

Logical instructions (`and`, `or`, `xor` and `test`) are tested first, then the shift instructions (`rl`, `rr`, `sla` and `sra`).

The addition and subtraction instructions are used extensively in the rest of the test program and as such did not warrant individual testing.

#### Hardware multiplication

The hardware multiplier is used to multiply two values. The result spans the width of the output port. A weakness in this test is that only one multiplication is performed,

Stellenbosch University http://scholar.sun.ac.za


but this is in keeping with the philosophy of running a test that is representative of an actual application.

#### Program memory

The code occupies about 30% of the program memory. Upsets in the unused part would not interrupt the functioning of the design, similar to a real-world application.

#### Instructions

All instructions, with the exception of the more esoteric types of shift and rotate instructions were used in the test program. Any error in the instruction decoding logic would cause the device to deviate from its expected behaviour.

### 6.4.3 Post-processing

When a simulation is complete, the log files are processed to construct a list of upsets that caused the design to fail.

By running `bin2im.py`, the locations of the set bits in the configuration memory can be obtained, as well as a mapping of the locations of the sensitive bits.

The sensitive bits give us a good indication of the dynamic cross-section ($\sigma'$) of the different designs. Since no physical cross-section measurements are available, the $\alpha$ values of different designs are calculated to allow comparison.

The cross-sections of bits in the BRAM columns differ from the cross-section of bits in the rest of the configuration, therefore these areas are considered separately.

$$\begin{aligned} \alpha &= \frac{\sigma'}{\sigma} \\ \alpha_{BRAM} &= \frac{n_{BRAM}}{257792} \\ \alpha_{config} &= \frac{n_{config}}{785248} \end{aligned}$$

Note that $\alpha_{BRAM}$ is calculated by dividing with the total number of bits in the BRAM columns, which is 15% more than the number of bits in the configuration memory. By making this approximation, all further measurements are greatly simplified without losing much accuracy.

Stellenbosch University http://scholar.sun.ac.za


## 6.5 Conclusion

The SEU simulator provides a low-cost tool that can aid in developing SEU tolerant designs for SRAM FPGAs. It can be used to validate SEU tolerant designs, helping to find the optimal trade-off between cost and hardness, without having to resort to radiation testing. The simulator can measure the dynamic cross-section of the configuration and BRAM areas of the target FPGA.

The greatest difference between this SEU simulator and other implementations, is that a very good trade-off between speed and cost has been obtained.

### 6.5.1 Speed

Every upset iteration of the simulator requires, on average, 27.5ms. To test an entire XC3S200 configuration takes just under 8 hours.

Of this 27.5ms, about 5 ms is used for the execution of the self-checking routine, 21.8ms is used for reconfiguration and the remaining 700us is the overhead required for control and verification.

Although this slower than the BYU simulator, it is still quite respectable, given the much lower cost of the system. The simulator also provides faster results than exhaustive radiation testing does.

### 6.5.2 Limitations

It is important to keep in mind that the SEU simulator can only observe (and test) the configuration memory and BRAM areas on the devices. The effects of upsets to half-latches, registers and upsets resulting in SEFI, can not be simulated. These sections have a relatively small cross-section when compared to the tested area, therefore the simulation gives a good approximation of the dynamic cross-section.

The current implementation of the SEU simulator can only inject errors before start-up. A more accurate model can be obtained by using partial reconfiguration to inject upsets into an already running device. This will add a time dimension to upsets, resulting in a more realistic model of actual SEUs.

The current method of upset testing is "blind", because the exact element influenced by a bit-flip is unknown. By using Virtex devices, one could use the `JBits`-library to overcome this limitation.

Stellenbosch University http://scholar.sun.ac.za


### 6.5.3 Recommendations

Although already an extremely useful tool, future implementations of the simulator can still be improved in various ways.

The system can be sped up significantly. The SelectMap interface allows configuration at up to 50MHz. The current system performs configuration at 6MHz. The access speed to the flash memory limits the configuration speed to 14MHz, but this can be improved by using faster flash. From the PAR results, the current implementation should support configuration speeds up to 16MHz, if the high frequency characteristics of the board is improved. Even higher speeds should be possible by improving the speed of the HDL design, or using a faster device.

Alternatively, the low cost of the system can be exploited to build multiple hardware copies and perform simulations in parallel.

The system can be stream-lined by laying out a single board. A socket can be used to accommodate different FPGAs, making the testing of different devices possible.

Stellenbosch University http://scholar.sun.ac.za

# Chapter 7

# Fault Tolerant PicoBlaze

A solid background knowledge of the effects of radiation and the techniques used to mitigate them have been established in preceding chapters. A mechanism to measure the improvement in fault tolerance has also been constructed. All these components are now combined to create a fault tolerant PicoBlaze implementation.

The power of the SEU simulator lies in its ability to measure the improvement offered by the applied mitigation techniques. However, it was interesting to also evaluate other aspects of the different mitigation techniques, such as the impact on performance and the ease with which it could be implemented.

Several variations of the PicoBlaze were implemented and tested on the SEU simulator, to evaluate various error mitigation techniques. The best features of the different designs were combined to produce an SEU tolerant PicoBlaze implementation.

## 7.1 Designs

A PicoBlaze design was implemented to serve as reference design. This design was modified to test different fault tolerance mechanisms. The different designs all executed the same self-checking routine, as described in the previous chapter. This allows us to compare the hardness of the different designs.

In all cases, except where noted otherwise, the designs were synthesised to operate at 50MHz. No other constraints, except for routing, were applied.

### 7.1.1 Reference PicoBlaze (ISE)

The PicoBlaze soft-core processor was extended to implement various features commonly used in an onboard 8-bit microcontroller.

Stellenbosch University http://scholar.sun.ac.za


### Implementation

The connections made to the KCPSM3 core to implement the reference design are shown in Figure 7.1.

The Spartan-3 Starter Kit has two external 512kB SRAM modules. These were combined by using the chip enable (CE) signal to form continuous memory. Only the lower 8-bits of the 16-bit data bus were utilised, which resulted in the memory being controlled as a single 512kB module. The upper bits were later used in some of the hardened designs.

The LEDs on the board were connected to an output port to serve as visual indication that the system was running. The sliding switches were used as an 8-bit input signal.

To increase the number of pins used, an 8-bit loop-back port was instantiated. The output pins were connected to the inputs pins by an external connection. By writing data to this port, upsets in the IOBs can be detected.

One of the hardware multipliers on the Spartan-3 was used to provide asynchronous integer multiplication. Two output ports supplied the input values, while the result could be read on an input port.

The bidirectional UART supplied with the core was used for communication (115200 baud, 8N1). The design was configured to run at 50MHz. A counter was set up to generate a $1\mu$s timer interrupt.

The program code was stored in a single BRAM block.

Synthesis, mapping and PAR were performed with Xilinx ISE 8.1, which mapped the design to 208 flip-flops and 326 4-input look-up tables. The complete design used 91 pins.

### Error injection results

The configuration memory bitmap is shown in Figure 7.2(a). Grey pixels represent 0 bits in the configuration memory, while 1 bits are black. The locations of bits that caused the device to fail have been marked in black in Figure 7.2(b).

It can be seen that the locations of sensitive bits correspond to utilised resources on the devices. It is very important to note that not only 1 bits in the configuration memory are used to implement the logic of the device, but that upsets in 0 bits can also cause the design to fail.

The utilised area of BRAM, where the processor instructions are stored, is very sensitive to upsets.

Another important observation is that a large number of bits that do not influence the design. These bits correspond to unused resources on the device.

Stellenbosch University http://scholar.sun.ac.za

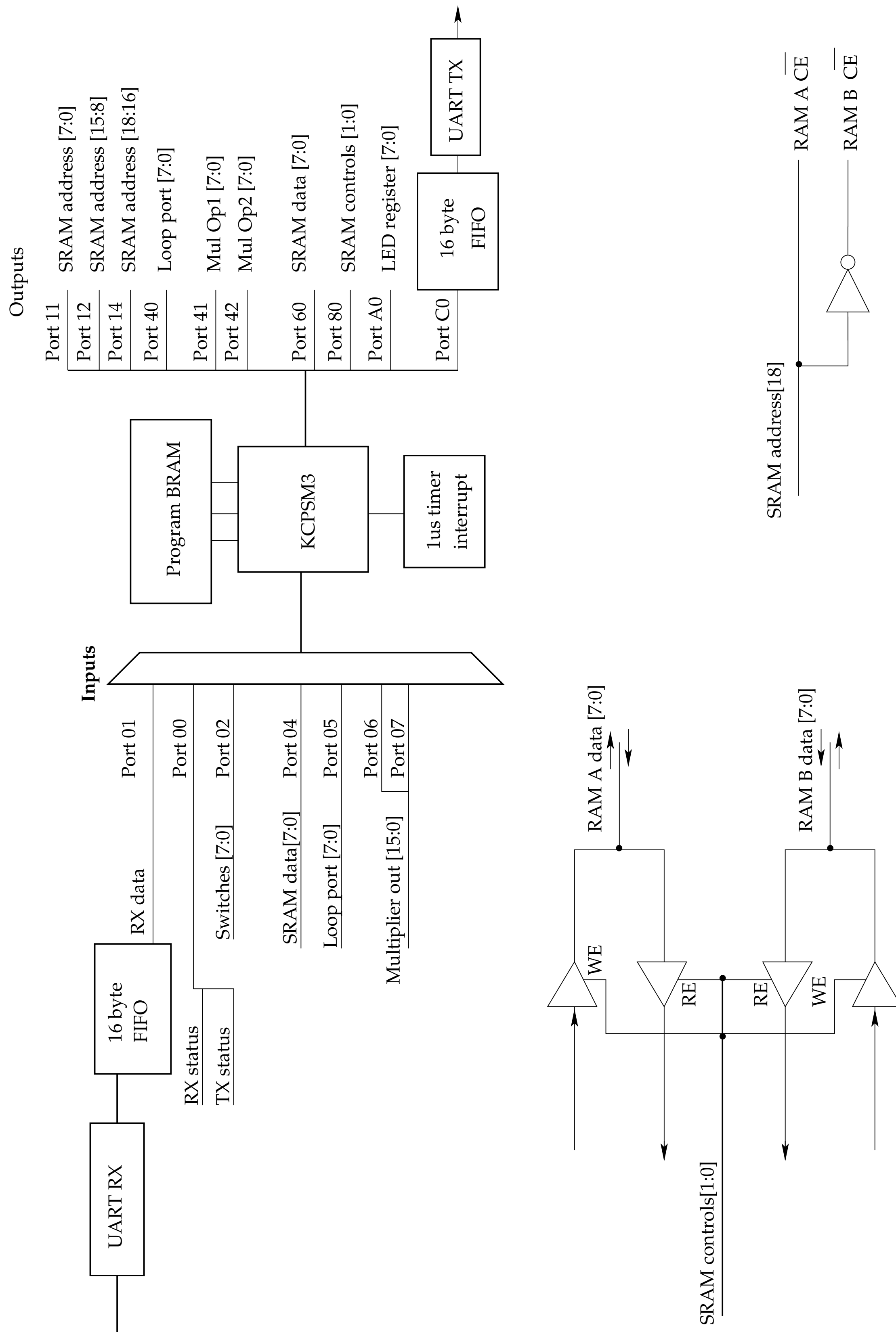
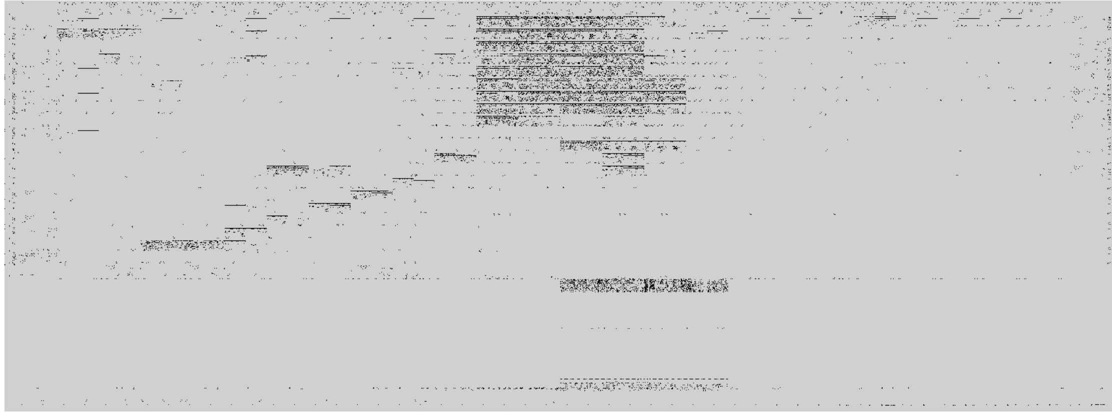




**Figure 7.1:** Block diagram of reference PicoBlaze design, showing decoding used on external SRAM

Stellenbosch University http://scholar.sun.ac.za

(a) Configuration bitmap for reference PicoBlaze

(b) Sensitive bits in configuration memory

**Figure 7.2:** Configuration bitmap for reference PicoBlaze with sensitive bits indicated. For the configuration bitmap, a 1 bit in the configuration memory is represented by a black pixel, while 0 bits are grey. The bits that caused the device to fail (sensitive bits) are indicated by black pixels in (b). Note the correspondence between the used bits in (a) and the sensitive bits in (b).

2678 sensitive bits are located in the BRAM area, with another 13642 bits in the rest of the configuration memory. This allows one to calculate the scaling factors as follows:

$$\alpha_{config} = \frac{13642}{785248} = 0.01737$$

$$\alpha_{BRAM} = \frac{2678}{257792} = 0.01309$$

It is important to realise that these $\alpha$ values are small, because the reference PicoBlaze design uses very little logic (only about 6% of the FPGA is utilised). A larger design without mitigation will have larger $\alpha$ values.

ISE reported that the design could run at up to 76.64 MHz.

Stellenbosch University http://scholar.sun.ac.za


### 7.1.2 Reference PicoBlaze (Synplify Pro)

The reference PicoBlaze described above was synthesised using Synplify Pro and ISE was used to perform PAR. This was done to test the performance of other software and to verify the results obtained with ISE.

Most of the design has been specified with low level VHDL, so the difference in implementation between the two software engines is very small. Synplify Pro used a few more registers (241), but fewer LUTs (316). Timing analysis in ISE predicted a maximum operation frequency of 71.15MHz.

#### Error injection results

SEU simulation supported the measurements made with the ISE implementation. The configuration bitmap and sensitive bit distribution were similar to the other implementation, and are therefore not shown.

13345 sensitive bits were found in the configuration memory and 2673 in the BRAM columns. This gave the scaling factors as:

$$\alpha_{config} = 0.01699$$
$$\alpha_{BRAM} = 0.010377$$

### 7.1.3 High-level TMR (ISE)

A high-level TMR implementation was obtained by instantiation of three redundant copies of the reference PicoBlaze design and using majority voters to check the outputs.

The PicoBlaze was treated as a black box for this implementation. No protection or correction mechanisms were applied internally.

#### Implementation

Implementation took place in four phases, which are also reflected in the top-level VHDL file. The Picoblaze was modified to allow a TMR instantiation, arrays were declared to simplify the implementation process, redundant copies of the PicoBlaze were instantiated and the signals were connected using voters.

The only modification to the PicoBlaze, was to replace the bi-directional ports (`INOUT`) with uni-directional ports (both `IN` and `OUT`). The signal that controls reading and writing was changed to an output signal.

By declaring arrays with three elements for all the ports on the PicoBlaze, a `GENERATE` statement could be used to easily instantiate three redundant copies of the entity.

Stellenbosch University http://scholar.sun.ac.za


All output signals, as well as the control signals for bi-directional buses, were voted using LUT-based voters. The voters were implemented by using two overloaded functions, one for single bits (type `STD_LOGIC`) and one for bit vectors (`STD_LOGIC_VECTOR`). Input signals were simply connected to the input signals of the different instantiated entities. The connections for a bi-directional port are shown in Figure 7.3.

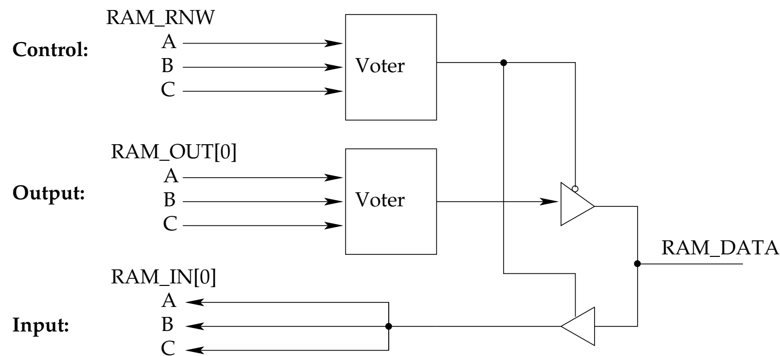



**Figure 7.3:** Majority voting on a bi-directional bus, as implemented on the external SRAM data bus. The output signal and the control signal are both voted, while the input signal is directly connected to the inputs of the redundant modules. The letters on the individual traces correspond to the redundant instance of the entity they are connected to.

The design was implemented with Xilinx ISE 8.1, which reported using 1026 4-input LUTs (3.14 times as many as the original design) and 648 registers. One would expect the number of registers to be exactly three times more than in the reference design ($3 \times 208 = 624$), but 24 latches were inferred to implement the voting logic, increasing the number slightly.

### Error injection results

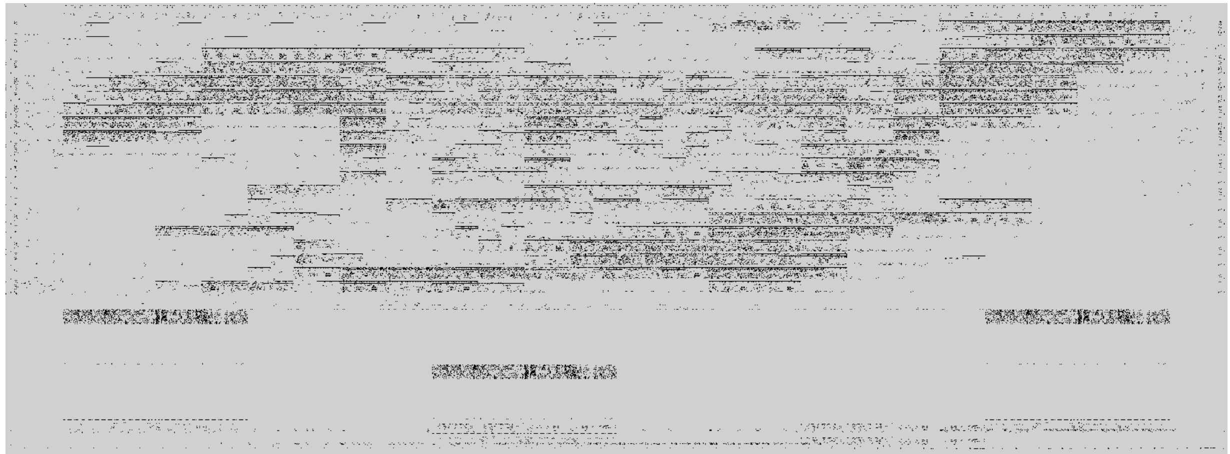
The configuration bitmap and sensitive bit locations are shown in Figure 7.4. The increased number of used configuration bits can clearly be seen, as well as the three BRAM elements now being used. Despite the increase in device utilisation, the number of sensitive bits decreased dramatically.

Only 19 bits in the BRAM area cause the design to fail when upset. The rest of the configuration memory has only 1175 sensitive bits. This shows that $\alpha$ has decreased to

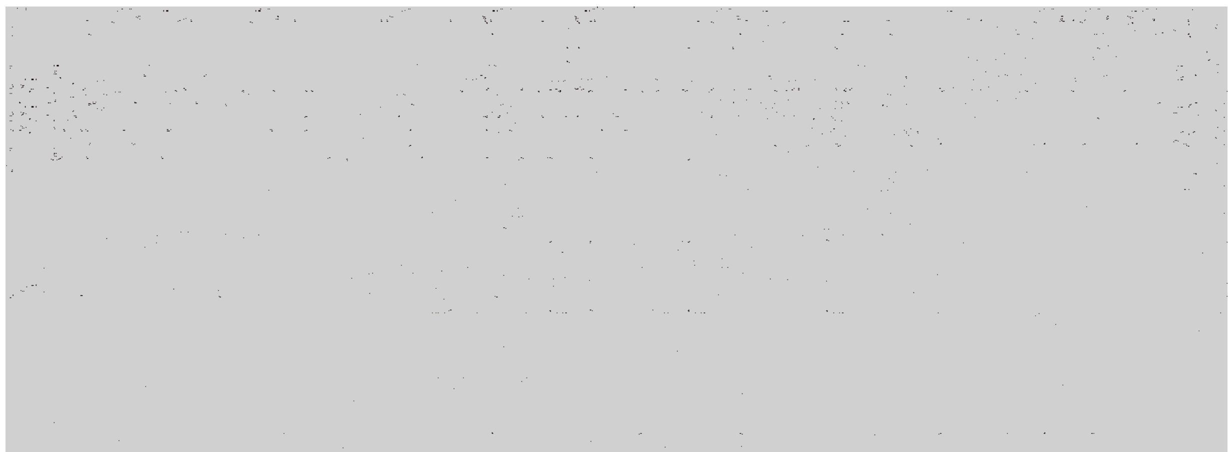
$$\begin{aligned} \alpha_{config} &= 0.00150 \\ \alpha_{BRAM} &= 0.00007 \end{aligned}$$

The dynamic cross-section of the design has therefore been substantially decreased by the TMR implementation.

Stellenbosch University http://scholar.sun.ac.za

(a) Configuration bitmap



(b) Sensitive bits

**Figure 7.4:** Configuration bitmap and sensitive bits for high-level TMR design. Note the increase in used bits in the configuration memory in (a), due to the design now being more than 300% the size of the original. Majority voting manages to significantly decrease the number of sensitive bits, as can be seen in (b).

### 7.1.4 High-level TMR (Synplify Pro)

The same TMR design was also implemented with `Synplify Pro`. As before, the results correlated well with those of the ISE implementation.

The design was mapped to 648 registers (24 were used as latches) and 997 LUTs (3.16 times as many as the reference design synthesised with Synplify Pro). The maximum clock frequency was reported as 68.71MHz.

#### Error injection results

The configuration bitmap and sensitive upsets are not shown, since they are very similar to that of the ISE implementation.

Stellenbosch University http://scholar.sun.ac.za


$$\alpha_{config} = 0.00156$$
$$\alpha_{BRAM} = 0.00010$$

### 7.1.5 High-level TMR (BUFTs)

The majority voters used in the previous two designs, can also be constructed using active-low tri-state buffers (BUFTs).

According to a Xilinx application note[40], the circuit in Figure 7.5 can be used as a majority voter. The active low tri-state buffers with a pull-up resistor on the output will transmit the correct values if all the inputs match. If the input signals do not match, simulation will show contention on the output, but on hardware it will act as a majority voter.

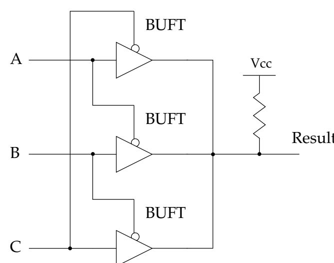



**Figure 7.5:** Majority voter using tri-state buffers

This happens because the architecture of Virtex and Spartan devices allows the circuit to be implemented in the bus logic of the FPGA. The circuit, after mapping to the BUFTs that connects CLBs to the buses, is equivalent to a majority voter.

A benefit of this technique is that the voter circuits do not require extra logic, it is therefore useful in designs where limited logic resources are available.

#### Implementation

This design could only be synthesised with `Synplify Pro`, as `ISE 8.1` predicted contention on the voter outputs.

The voting function used in the LUT-based TMR implementations, was replaced with a voting module (`TRV_BUFT.vhd`). Single voters were also wrapped in a file that would generate an arbitrary number of voters, to be used with vector signals.

The source for the voting circuits is listed in Appendix B.

Stellenbosch University http://scholar.sun.ac.za


The circuit was synthesised using `Synplify Pro` and was mapped to 648 registers and 1007 4-input LUTs. The maximum clock frequency was reported as 77.27 MHz.

#### Error injection results

The configuration bitmap and upset distribution are similar to the previous TMR implementations, and are therefore not shown.

17 BRAM bits and 1165 configuration bits were found to be sensitive. This gives $\alpha$ as:

$$\begin{aligned}\alpha_{config} &= 0.00148\\ \alpha_{BRAM} &= 0.00007\end{aligned}$$

### 7.1.6 TMR flip-flops and protected program ROM

A design with automatically inserted TMR flip-flops and protected program ROM was also tested.

This implementation had two objectives:

- Illustrate that protecting only the memory elements in a design does not offer sufficient protection in designs implemented on SRAM FPGAs (unlike flash or antifuse devices).
- Evaluate the manipulation of a design on EDIF level, after `Synplify Pro` had been used to synthesise and map a design.

#### Implementation

A TMR version of the program ROM was implemented with LUT-based voters. Since the focus was on only protecting the memory, the interface to the TMR version was the same as the original, allowing a drop-in replacement in the original file.

The design was then compiled with `Synplify Pro` to produce an EDIF netlist as output. The netlist would normally be used as an input file by the Xilinx software, but in this case software was used to replace all the instantiated flip-flops and latches with a TMR equivalent. The program `tmrff.py` was specially developed for this purpose.

The EDIF files generated by Synplify contain all the inferred logic and connections, but describes it in terms of primitive components. These components are defined in libraries, which are included in the file. The file is in plain text and the structure is extremely modular, making parsing and manipulation very easy.

By studying the generated EDIF file for a manually instantiated TMR flip-flop, an general template for TMR flip-flops could be constructed. This template was used to gen-

Stellenbosch University http://scholar.sun.ac.za


erate an EDIF library of redundant flip-flops that have a three input look-up table as majority voter. By keeping the interface to the equivalent TMR flip-flops the same as that of the originals, the references to the original components could be updated by a simple string substitution.

The resulting netlist could then be translated by the Xilinx tools to generate the configuration file. The final implementation uses 723 flip-flops and 571 4-input LUTs. The maximum operating frequency was reported as 53.61 MHz. This is lower than the other designs, mainly because of the voting logic that was inserted in the path between the program ROM and the instruction execution unit. The timing optimisations made by `Synplify Pro` were also invalidated by modifying the design after synthesis and mapping.

The source code of `tmrff.py` is listed in Appendix B

#### Error injection results

SEU simulation proved that only protecting the flip-flops does not offer much improvement in SRAM FPGAs.

13770 sensitive bits were found in the configuration section, while the TMR BRAM implementation showed only 30 sensitive bits. The scaling factors were calculated as:

$$\begin{aligned} \alpha_{config} &= 0.01754 \\ \alpha_{BRAM} &= 0.00012 \end{aligned}$$

### 7.1.7 Hamming-encoded memory and pins

To evaluate the use of an error-correction code, the reference design was modified to use Hamming-encoded instructions. The Hamming encoder was also applied to some of the output pins in the design, to implement pin-redundancy.

The main concern with this implementation was to keep the decoding fast, as it takes place in the instruction path. A Hamming code was selected, because it offers single bit correction and double bit detection, while being relatively simple to implement in both VHDL and (for generation and verification) on a PC. Once implemented, it is also easy to scale to other word widths. Other encodings, such as Bose-Chaudhary-Hoquenghem (BCH) offer higer levels of protection, but are more complicated to calculate.[45]

#### Implementation

The 18-bit instructions in the program memory were protected with a (23,18) Hamming code. This encoding uses 5 parity bits to protect 18 data bits.

Stellenbosch University http://scholar.sun.ac.za


**Table 7.1:** (23,18) Hamming encoding matrix. The bits marked with an asterisk are `XOR`'ed together to calculate the check bits.[44]

| Parity bit | Bit address in data word | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| P1 | * | * | | * | * | | * | | * | | * | * | | * | | * | | * |
| P2 | * | | * | * | | * | * | | | * | * | | * | * | | | * | * |
| P3 | | * | * | * | | | | * | * | * | * | | | | * | * | * | * |
| P4 | | | | | * | * | * | * | * | * | * | | | | | | | |
| P5 | | | | | | | | | | | | * | * | * | * | * | * | * |

One of the output files of the KCPSM3 (PicoBlaze) assembler is a text file with hexadecimal instructions, generated from the program code. The assembler generates a program memory VHDL file that contains a single BRAM block, initialised with the instructions.

The hexadecimal instructions were processed by a program, `hamming_mem.py`, that calculated the parity bits by performing modulo 2 multiplication with the encoding matrix in Table 7.1. The code word was constructed by placing the partity bits in the positions that are a power of 2 to form a 23-bit code word. Two BRAM primitives were initialised with the encoded instructions by using a VHDL template file. This encoded memory module replaced the original program memory that was generated by the assembler.

To initialise both RAM blocks, it was required that the encoded instructions were split into a lower 16-bit portion, a 2-bit central portion and an upper 5-bit portion. The separation of the instructions was required because of the way the BRAM primitives are initialised (data (0:15) and parity (16:17) bits are separated). The half of the memory that is not used by the program memory can still be used for other purposes by instantiating a 9x2048 BRAM element (by changing the address width-to-depth ratio).

The template file also instantiated a (23,18) Hamming decoder, to provide the decrypted instructions to the rest of the system. Decoding involves recalculating the check bits and `XOR`'ing them with the original check bits to generate an *error syndrome*. The syndrome is compared with a look-up table to determine if any have errors occurred. If only one error occurred, the position is given by the integer value of the syndrome and the bit corrected. Table 7.2 lists the error syndrome values.

The encoded memory module was tested by simulating it in `Modelsim` and comparing the decoded instructions with the original ones.

The data buses to the SRAM and loop-back port were protected with a (12,8) Hamming code. This implemented a form of pin redundancy.[59] Only the buses where extra pins were available were protected with the Hamming codes, because the S3Kit did not have external traces connected to allow internal voting.

The SRAM data bus and loop-back port were widened to 12 bits to accommodate the 4 extra parity bits. By sending protected data over these ports, the buses were protected against any one pin or a connection to a pin being upset.

Stellenbosch University http://scholar.sun.ac.za


**Table 7.2:** (23,18) Hamming error syndrome look-up table. This table provides the error type and location, which allows single upsets to be detected.

| Inverted bit | Syndrome error code | | | | |
|---|---|---|---|---|---|
| | P5 | P4 | P3 | P2 | P1 |
| No error | 0 | 0 | 0 | 0 | 0 |
| Bit 1 | 0 | 0 | 0 | 0 | 1 |
| Bit 2 | 0 | 0 | 0 | 1 | 0 |
| Bit 3 | 0 | 0 | 0 | 1 | 1 |
| Bit 4 | 0 | 0 | 1 | 0 | 0 |
| Bit 5 | 0 | 0 | 1 | 0 | 1 |
| Bit 6 | 0 | 0 | 1 | 1 | 0 |
| Bit 7 | 0 | 0 | 1 | 1 | 1 |
| Bit 8 | 0 | 1 | 0 | 0 | 0 |
| Bit 9 | 0 | 1 | 0 | 0 | 1 |
| Bit 10 | 0 | 1 | 0 | 1 | 0 |
| Bit 11 | 0 | 1 | 0 | 1 | 1 |
| Bit 12 | 0 | 1 | 1 | 0 | 0 |
| Bit 13 | 0 | 1 | 1 | 0 | 1 |
| Bit 14 | 0 | 1 | 1 | 1 | 0 |
| Bit 15 | 0 | 1 | 1 | 1 | 1 |
| Bit 16 | 1 | 0 | 0 | 0 | 0 |
| Bit 17 | 1 | 0 | 0 | 1 | 1 |
| Bit 18 | 1 | 0 | 0 | 1 | 0 |
| Uncorrectable | All other syndromes | | | | |

Note that, although encoded data was stored on the SRAM, it was not tested for upsets like the FPGA was. In a real application the use of Hamming protection on external memory is be highly recommended.

By implementing the Hamming decoders and encoder in separate modules they can be inserted into the design with very little modification.

Figure 7.6 shows the changes made to add ECC protection.

The design was synthesised with `Synplify Pro` and PAR was performed with `ISE 8.1`. It uses 419 4-input LUTs, 241 flip-flops and has a maximum clock frequency of 52.27 MHz.

### Error injection results

The configuration bitmap and sensitive bit locations are shown in Figure 7.7. The one and half BRAM primitives used by the Hamming encoded memory show very few sensitive bits in that area. It should also be noted that the number of sensitive bits in the IOB columns is dramatically less than before, indicating that the Hamming-encoded buses are effective in mitigating single flipped bits in that area.

Error injection found 13803 sensitive bits in the configuration section and 14 in the

Stellenbosch University http://scholar.sun.ac.za

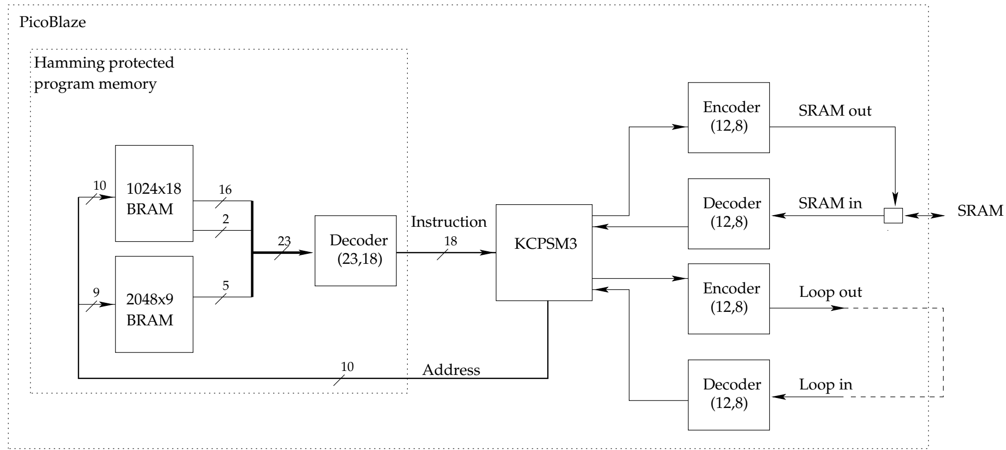
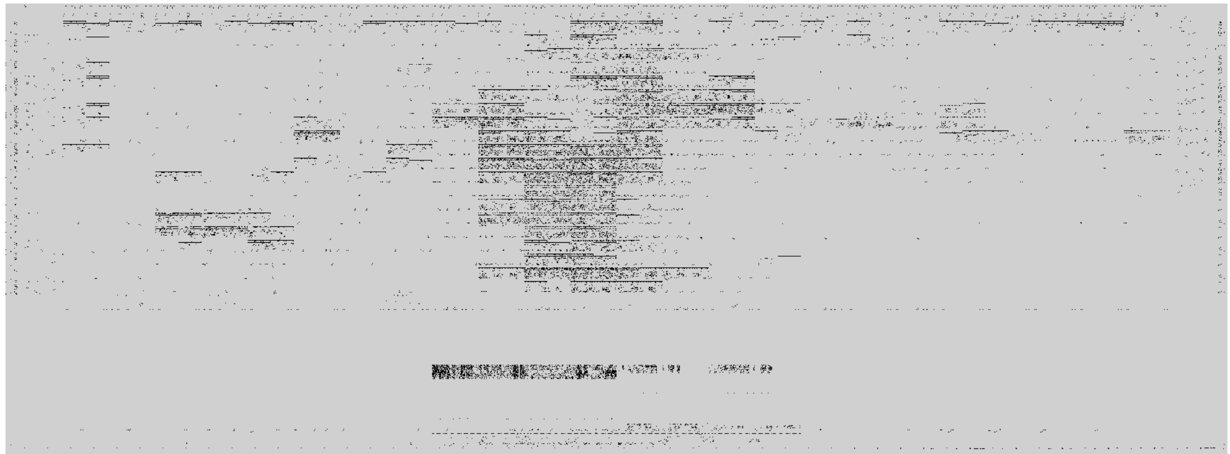
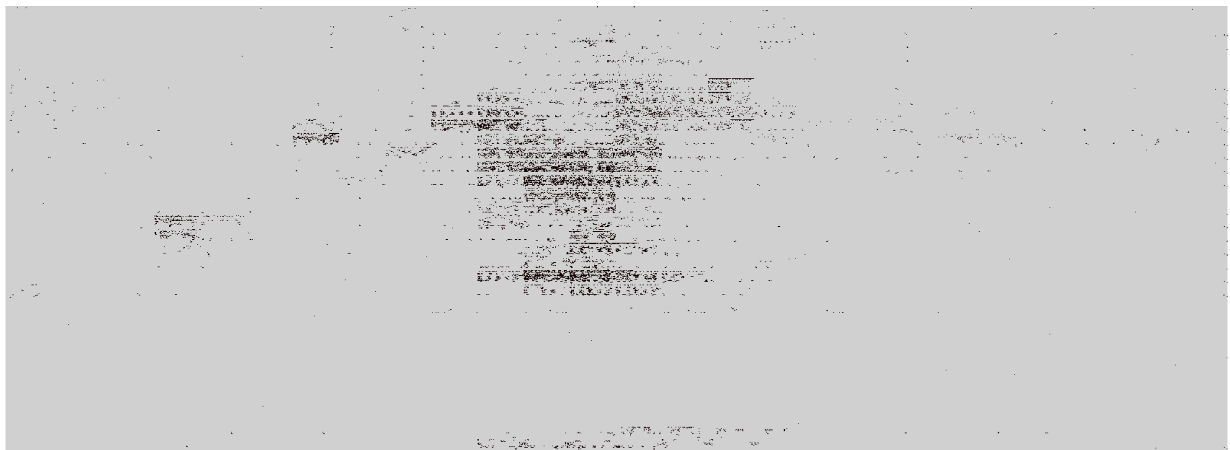




**Figure 7.6:** Block diagram of PicoBlaze with Hamming-encoded memory and data ports. The decoder for the memory is encapsulated in the program memory files.



(a) Configuration bitmap



(b) Sensitive bits

**Figure 7.7:** Configuration bitmap and sensitive bits for design with Hamming encoded memory and output pins. The one and a half BRAM blocks that are utilised by the design can be seen in (a). The decrease in sensitivity in the IOB columns is also visible in (b).

BRAM area, giving the scaling factors as:

$$\alpha_{config} = 0.01758$$
$$\alpha_{BRAM} = 0.00005$$

### 7.1.8 Final design

A final design was composed from the results of the above experiments. The objective was to find a good trade-off between protection, resource usage and implementation effort.

#### Implementation

The high-level TMR design using BUFT voters was modified to use less BRAM and only one multiplier. In addition, the SRAM data bus and loop-back port were protected by using redundant pins.

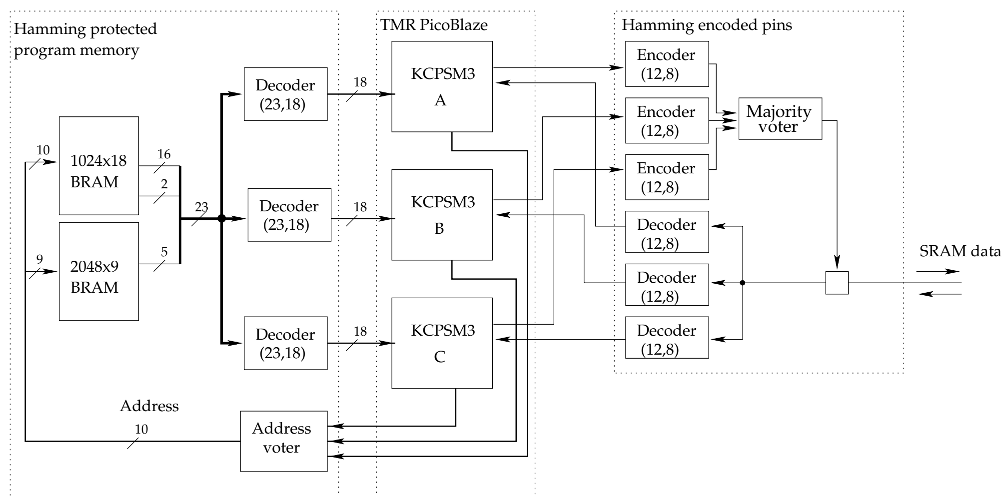



**Figure 7.8:** Block diagram of final Picoblaze design, showing major error mitigation elements. TMR is used to protect the internal KCPSM3 logic, while Hamming codes protect program memory and pins. The loop-back port is not shown, but it uses a similar TMR/Hamming encoding structure as the SRAM data port.

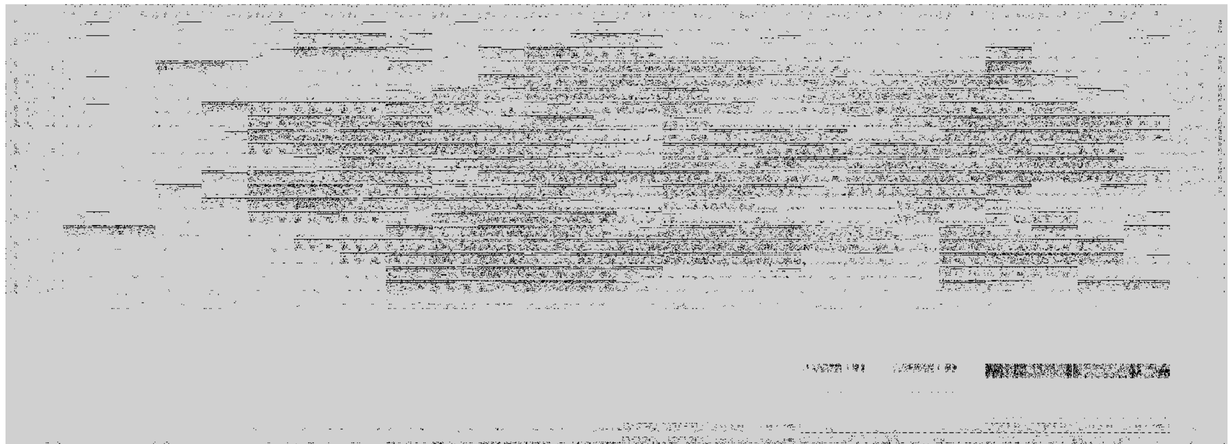
The XC3S200 has twelve 18×18 multipliers, as well as twelve 1024×18 bit BRAMs. As these are resources that are frequently used in designs, it was considered worthwhile to use fewer of these resources, even if it increased the dynamic cross-section slightly.

The program memory was again protected using a Hamming (23,18) encoding, as discussed in the previous section. To protect against upsets in the decoding logic, the decoders were implemented in triplicate, one for every redundant PicoBlaze core. The address buses were voted using a single majority voter.

The outputs to the SRAM data bus and the loop-back port were encoded first, then voted, as shown in Figure 7.8. This configuration required more logic than first voting and then encoding would, but the resulting SEU cross-section is significantly smaller.

The single hardware multiplier was connected to the three KCPSM3 instances. Majority voting was used to protect the input signals.

The design was synthesised with `Synplify Pro` and PAR performed with Xilinx `ISE 8.1`, resulting in 1316 used 4-input LUTs, 660 registers. Timing analysis calculated the maximum clock speed at 55.633MHz.



(a) Configuration bitmap

(b) Sensitive bits

**Figure 7.9:** Configuration bitmap and sensitive bits for design the final PicoBlaze design.

#### Error injection results

The configuration bitmap and sensitive bit locations are shown in Figure 7.9.

Error injection found 40 sensitive bits in the BRAM columns and 1529 in the rest of the configuration memory. The scaling factors were calculated as:

$$\alpha_{config} = 0.00195$$
$$\alpha_{BRAM} = 0.00016$$

## 7.2 Results

The tested PicoBlaze designs vary greatly in performance and logic utilisation. Table 7.3 summarises the characteristics of the different implementations in terms of logic utilisation and performance, while 7.4 lists the measured scaling factors.

The logic utilisation can be expressed in terms of the number of used LUTs and registers, which give a good indication of the size of the user logic. However, this does not take the specialised resources such as BRAM and multipliers into account. The equivalent number of gates gives us another a measure that includes these components. This figure is produced by the PAR software to give an indication of the number of gates required to implement a given design in and ASIC. The gate count for the final design is lower because the design requires much less BRAM than the TMR design.

It can be seen that the TMR implementations are the most expensive in terms of logic, but offer the smallest performance penalty. On average, these TMR implementations require about 310% of the LUTs used by the reference design. All the memory and multipliers are also triplicated. TMR was found to be fairly straight-forward to implement, making it an attractive, if expensive, mitigation techinique.

The use of Hamming ECC to protect the memory decreased the BRAM use by half, but also slowed the design down to 52.3MHz. This trade-off is seen as worthwhile, since BRAM is a relatively scarce resource. The performance penalty is mainly due to the (32,18) Hamming decoder being on the path between the program memory and instruction decoder. Software was used to generate an encoded memory module, consequently the designer did not have to take memory protection into account. This allowed the mitigation to be another step in the compile cycle of the software, which worked very well.

The final design required more care in the preparation of the source code, as the design had to be modified for the different connections. This effort was rewarded with a design that offers protection comparable to that of TMR, but with a lower overhead.

## 7.3 Observations

Other observations, that are not bound to a specific design, were also made.

**Table 7.3:** Logic utilisation and performance of different PicoBlaze implementations.

| Design | 4-input LUTs | Registers | Equivalent Gates | Speed (MHz) |
|---|---|---|---|---|
| Reference (ISE) | 326 | 208 | 83107 | 76.64 |
| Reference (Synplify) | 316 | 241 | 83308 | 71.15 |
| High-level TMR (ISE) | 1026 | 648 | 249531 | 75.48 |
| High-level TMR (Synplify) | 997 | 648 | 249348 | 68.71 |
| High-level TMR using BUFTs | 1007 | 648 | 249408 | 77.26 |
| TMR flip-flops | 571 | 723 | 219784 | 53.61 |
| Hamming-encoded memory | 419 | 241 | 149414 | 52.27 |
| Final design | 1316 | 660 | 185660 | 55.633 |

**Table 7.4:** The scaling factor measured for different PicoBlaze implementations.

| Design | $\alpha_{config}$ | $\alpha_{BRAM}$ |
|---|---|---|
| Reference (ISE) | 0.01737 | 0.01039 |
| Reference (Synplify) | 0.01699 | 0.01037 |
| High-level TMR (ISE) | 0.00150 | 0.00007 |
| High-level TMR (Synplify) | 0.00156 | 0.00010 |
| High-level TMR using BUFTs | 0.00148 | 0.00007 |
| TMR flip-flops | 0.01754 | 0.00012 |
| Hamming-encoded memory | 0.01758 | 0.00005 |
| Final design | 0.00195 | 0.00016 |

### 7.3.1 Upper bound on improvement

The $\alpha$ values obtained by the high-level TMR designs should be interpreted as being close to the upper bound on the improvement that can be obtained. The remaining sensitive bits are due to the unprotected voters and routing bits. Using redundant pins will eliminate even these bits.

It is important to note that any design of arbitrary size would obtain the same level of protection from TMR. TMR does not decrease the dynamic cross-section by a constant scaling factor, but rather makes the cross-section of the configuration and BRAM areas approach 0 $cm^2$.

### 7.3.2 Synthesis engine

The two synthesis engines used (Xilinx `ISE 8.1` and Synplicity `Synplify Pro`) provided approximately the same level of SEU sensitivity. Differences in their synthesis and mapping strategies lead to differences in the implementation, which can explain the observed discrepancies.

The greatest benefit of using `Synplify Pro` was that more mitigation techniques could be applied when using it.

### 7.3.3 Majority voter implementation

The TMR implementation that uses TRV_BUFTs appears to be slightly more resilient to injected errors, when compared to the LUT implementations. The expected cross-section of both components is expected to be the same, which may indicate that the placement of the design on the device made the difference.

Functions were generally used to implement LUT-based voters, while the tri-state buffer voters were instantiated as entities. The latter approach was found to be more flexible, since the voter implementation was kept separate from the design source code. It allowed the voter to to modified without running into any language or synthesis problems.

### 7.3.4 Redundant output pins

The designs using Hamming ECC to protect the output pins showed a marked decrease in the IOB sensitivity.

In many cases it is not feasible to use triple redundant output pins, due to device or system constraints. In these cases it can still be beneficial to implement an error-correction code, such as Hamming ECC, to provide protection.

### 7.3.5 Implementation considerations

It was observed throughout the design process that the application of these error mitigation techniques could be automated, or at least simplified, to a large extent by using software and fault tolerant libraries.

#### Software

The design and implementation of a HDL design involves several distinct stages. Error mitigation code can be added at different stages in the design process. In general, adding the code earlier, allows one to use the power of the HDL to easily describe the mitigation process. By inserting the code at a later stage in the process, the synthesis engine can be used without fear of optimisation removing redundant components, but a detailed knowledge of the target FPGA is required.

The use of software to manipulate both VHDL source files and EDIF netlists was evaluated. The extremely modular structure of an EDIF netlist and the low-level description it provides, makes it the more attractive option for automated error mitigation. [1]

[1] `Synplify Pro` can also generate a mapped VHDL design that provides a similar low-level netlist description. This description is similar to the EDIF description and should require similar effort to manipulate.

VHDL that has been written by a person uses a larger syntax and it often more complex. In this case the only attractive use for automated mitigation would be to implement high-level TMR, which treats the design as a black box.

#### Fault-tolerant libraries

A library of fault-tolerant components was created during the implementation of the different designs. It includes majority voters, Hamming encoders and decoders of various widths, as well as wrappers around the different ECC-encoded BRAM elements.

This library greatly simplifies the application of mitigation techniques, as the designer can instantiate components, without having to consider the intricacies of the implementation. Some consideration is still required when doing the high-level design, as one needs to decide how and where the different elements of the design connect to the error mitigation modules.

The files are provided on the included CD.

## 7.4 Conclusion

A fault-tolerant PicoBlaze implementation was developed by measuring the benefits offered by different mitigation techniques and applying the most suitable ones to the reference soft-core processor design. The implementation is not the most resistant to SEUs, but it offers a high level of protection at a lower cost than full TMR.

The HDL implementation hardens the logic, but the device still requires external protection. An external configuration controller is required to correct upsets in the configuration memory and latch-up protection is also needed.

# Chapter 8

# Radiation Testing

Proton irradiation of selected designs was performed to test the actual hardness gained from the fault tolerance mechanisms. The results from radiation testing also serve to measure the validity of the SEU simulator. In addition, valuable information about the physical SEU properties of the device is gained.

Protons were selected because they are the most significant source of SEUs in LEO satellites. SEUs caused by other high energy particles are similar to proton induced upsets, therefore testing with protons gives a good indication of how a design will respond in the LEO environment.

## 8.1 iThemba LABS

The iThemba Laboratory for Accelerator Based Sciences is the only facility in South Africa where proton irradiation can be performed.

The facilities are used for radiation medicine, including proton and neutron treatment, isotope manufacture, as well as particle science research.[60]

### 8.1.1 Accelerator operation

Particles can be accelerated by a cyclotron or a Van der Graaff accelerator. It works on the principal that a charged particle, moving in a magnetic field, experiences a force perpendicular to the plane formed by the particle velocity vector and the magnetic field

vector, as described by the Lorentz force law.

$$
\begin{aligned}
\mathbf{f}(\mathbf{r},t) &= q\left(\mathbf{E}(\mathbf{r},t) + \mathbf{v} \times \mu_0 \mathbf{H}(\mathbf{r},t)\right) \\
\text{with} & \\
q &= \text{Particle charge} \\
\mathbf{f} &= \text{Force vector} \\
\mathbf{E} &= \text{Electric field vector} \\
\mathbf{v} &= \text{Velocity vector} \\
\mu_0 &= \text{Permeability of free space} \\
\mathbf{H} &= \text{Magnetic field vector} \\
\mathbf{r} &= \text{Position} \\
t &= \text{Time}
\end{aligned}
\tag{8.1}
$$

The electric field is used to accelerate the particle, while the magnetic field is used for steering.

iThemba has 3 cyclotrons: a large Separated Sector Cyclotron (SSC) and two smaller Solid Pole Cyclotrons (SPCs). The SSC can accelerate protons to a maximum energy of 200 MeV, while the SPCs are used as injector cyclotrons.

In a cyclotron particles are accelerated by a potential difference between a so-called “dee” and “dummy dee”. Four acceleration gaps are created by two dees. Ions are produced by an internal ion source and are accelerated towards the first dee, which has an opposite polarity to the particle. The polarity to the dee is changed by a radio frequency generator, which causes the particle to be accelerated to the ground polarity dummy dee and then towards the next dee. This process is repeated while the particle gains energy and spirals outwards. At a high enough energy, the trajectory of the particle intersects the electrostatic extraction channel, where the particle is redirected from the cyclotron onto the beam line, using a different magnetic field.[61][60]

## 8.2 Relevant previous tests

### 8.2.1 Testing at iThemba LABS

H. Berner previously tested the SEU characteristics of an ADSP processor at iThemba LABS, using both a vacuum chamber and the proton therapy station. In the final report it was recommended that future tests be carried out at the therapy station, as it made setup much easier and provided much better results.[61]

Only the therapy station was therefore considered when designing this experiment.

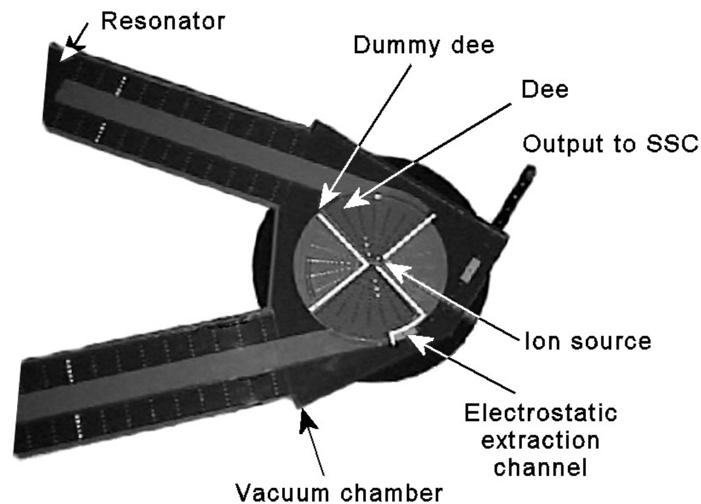



**Figure 8.1:** Diagram showing solid pole cyclotron (SPC), with dee and dummy dee locations.

### 8.2.2 Rosetta experiment

The Rosetta experiment is a long running experiment by Xilinx to determine the effects of atmospheric radiation on SRAM FPGAs at different altitudes.[36]

Banks of Virtex family and Spartan family FPGAs are run continuously, with the configuration of each being checked sequentially through readback. If an upset is detected, the configuration is corrected and the event logged. By running similar tests over a long period of time, at different altitudes, useful information about the effects of atmospheric upsets on SRAM FPGAs has been collected.

### 8.2.3 Brigham Young verification of SEU simulator

The Brigham Young SEU simulator that was discussed in a previous chapter was verified by proton testing at the Los Alamos National Laboratories.

A previously simulated FPGA design was exposed to the beam, and the configuration checked at intervals. The interval was adjusted to usually result in a single memory upset. Any upset bits were corrected through partial reconfiguration, while the FPGA output was compared to a golden chip running concurrently. By using the high performance SLAAC-1V board as base, a large number of samples could be taken in a fairly short period of time.

By comparing the simulated and experimental results, it was shown that the simulator accurately predicted the effect of 98% of the configuration memory upsets.[62]

## 8.3 Experimental design

In planning the experiment, a testing methodology had to be decided on to ensure useful results.

The main objective was to determine the scaling factor ($\alpha$) that relates the static and dynamic cross-sections. To achieve this one requires enough samples to draw statistically sound conclusions. Factors such as the observation speed and the dose rate of the beam also had to be taken into account in the design of the experiment.

A secondary objective was to measure the physical cross-section of the device, so that predictions of the upset rates in space can be made.

### 8.3.1 Tested designs

Three designs were tested:

- The PicoBlaze reference design, synthesised with `Synplify Pro` (Chapter 7.1.2)
- The high-level TMR design that uses tri-state buffer majority voters (Chapter 7.1.5)
- The final design that protected memory and output pins with Hamming codes, with the rest of the design covered by TMR, also with tri-state buffer voters (Chapter 7.1.8).

The selected designs allowed comparison between the reference design and two hardened designs. SEU simulation reported the TMR design as being the most robust. The final design is also good at mitigation, but uses less logic than the TMR implementation.

### 8.3.2 Expected cross-section

No proton SEU data for Spartan-3 devices was available. Xilinx only tests the sensitivity of the commercial devices to atmospheric neutrons[36]. For Xilinx to provide reliable proton data would require strict and expensive quality control and testing, because the sensitivity varies with the manufacturing technology and mask set.

As no information for an equivalent 90nm device was available, the cross-section from neutron testing was used to obtain an approximate cross-section. The non-ionising energy loss (NIEL) can be used to relate the upset cross-sections of different particles, if one assumes the amount of fragmentation in the silicon nuclei to be proportional to the NIEL cross-sections. For protons and neutrons, the NIEL cross-sections are approximately similar above a few tens of MeV, thus the SEU cross-sections should also be similar.[63].

The bit cross-section for the configuration and BRAM areas ($\sigma_{config(bit)_n}$ and $\sigma_{BRAM(bit)_n}$) from Xilinx testing[36] was multiplied by the number bits to obtain an expected device cross-section to neutrons ($\sigma_n$).

$$\begin{aligned}\sigma_n &= n_{config}\sigma_{config(bit)_n} + n_{BRAM}\sigma_{BRAM(bit)_n} \\ &= 821856 \times 2.4 \times 10^{-14} + 221184 \times 3.48 \times 10^{-14} \\ \sigma_n &= 2.742 \times 10^{-8}\text{cm}^2\end{aligned}$$

The SEU simulator predicts that the dynamic cross-section for the designs is much smaller than the static cross-section. It is consequently not feasible to test the device using single upset doses (as was done in the verification of the BYU simulator), because the expected failure rate is so small. To achieve high confidence in the measurements would require thousands of runs, which would take much longer than the available beam time.

It was decided to expose the device to a large enough dose that multiple upsets are caused, which increases the probability of failure. The experiment can then be modelled as a set of Bernoulli trails, which allows one to obtain the actual cross-section. Three different doses were used to obtain different failure rates.

### 8.3.3 Experiment structure

The static cross-section of the device had to be determined first. The FPGA was reset and irradiated before the upsets in the configuration was checked.

The dynamic part of the experiment consisted of a series of runs where one design was repeatedly irradiated with a specific fluence. This provided an average failure rate for the design at that dose. The runs were then repeated with two higher doses on the same design. The process was repeated for the two other designs.

A constant proton energy (100MeV) was used for all the runs.

## 8.4 Experimental setup

The experimental setup was kept as similar as possible to the simulation setup, because the simulator hardware had already been thoroughly tested. It also eliminated the possibility of a design change influencing the results.

Minor changes had to be made to the program running on the PicoBlaze to repeat the self-test sequence. A latch-up protection circuit was also added. A differential trans-

mission cable, with transceivers at both ends, had to be designed and constructed to allow communication between the control station and the testing chamber.

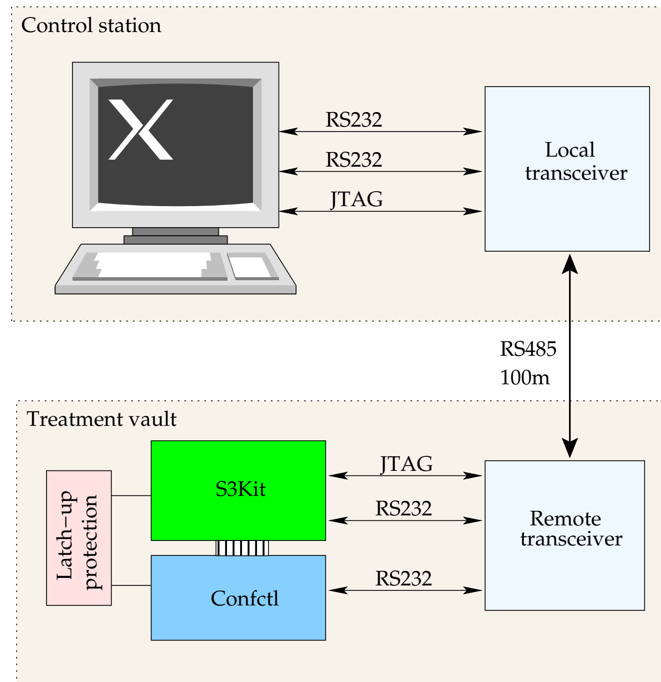



**Figure 8.2:** Block diagram of setup for radiation experiment

The system consisted of the test board, with the Spartan 3 Starter Kit and configuration controller, a latch-up protection circuit, two transceivers to communicate over a 100m cable, a control computer and a 100m power cable.

### 8.4.1 Test board

The Spartan 3 Starter Kit board and configuration controller were used in the radiation vault. The FPGA was regarded as the device under test. No other active components were exposed to the beam. The rest of the system acted as supporting hardware.

This allowed the use of exactly the same HDL code as had been simulated, only the code of PicoBlaze had been modified to loop repeatedly over the self test code. The PicoBlaze blocked execution while waiting for a character from the control station – this was used the control frequency of self testing.

### 8.4.2 Latch-up protection

An external power control circuit was adapted from [61] to allow the configuration controller to do a power off/on reset of the FPGA board, in case of latch-up. The original

SEU simulator design had the configuration controller providing the S3Kit with an unregulated 5V input. This connection was cut and the board was instead powered by the controlled output of the latch-up protection circuit.

The controlled power supply consists of an IRF9350N n-channel power MOSFET which could be switched on and off by the AVR. An LM358 operational amplifier is used as buffer between the 3.3V AVR and the 5V supply voltage.

The current powering the FPGA board was monitored by measuring the voltage drop over a 0.5Ω resistor with a differential amplifier. The measured voltages have a 5V nominal value. The voltages are first divided down to 2.5V, to prevent saturation of the op-amp, before amplifying the voltage difference. The differential amplifier was designed that the output voltage will be small enough that the AVR will not experience more than 3.3V on the ADC pin.

The output of the differential amplifier is also divided down to the voltage range of the configuration controller (0-3.3V) and measured using the built-in 10-bit ADC of the AVR. The resistors used in the different stages are chosen to differ by orders of magnitude, allowing the division and amplification stages to be designed separately.

A current measuring function was driven by the timer interrupt to monitor the current every 125ms. If the current exceeds a preset level (≈ 200mA at 5V), the AVR will remove all power from the FPGA circuit and notify the control computer. The computer has to switch it on again for the test to continue. The control computer can also query the configuration controller for the last read current value. One unit measured by the ADC translated to 0.556 mA.

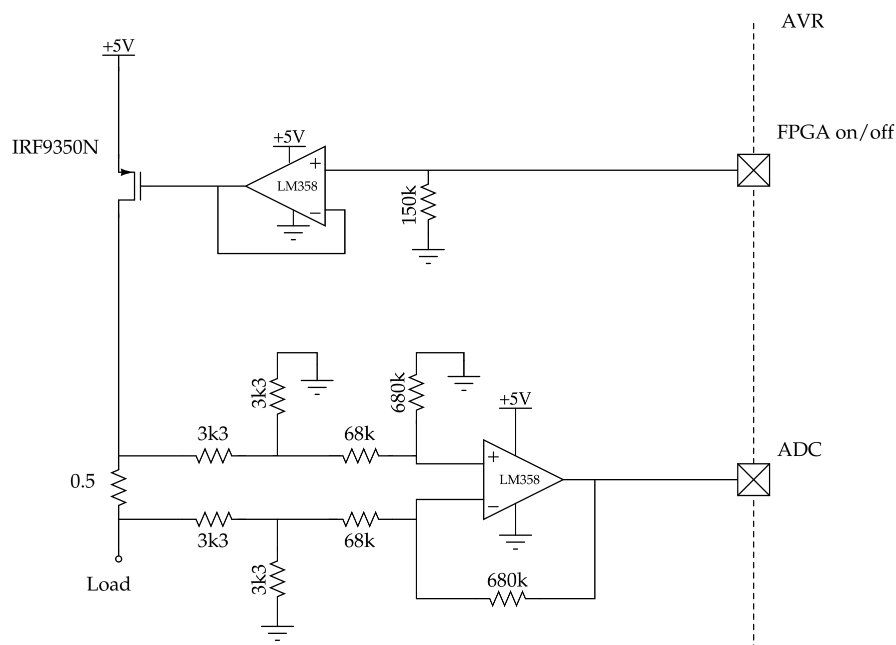



**Figure 8.3:** Latch-up protection circuit.

Provision had to be made for power leaking from the connecting pins between the FPGA and configuration controller's CPLD. The 1.25 V internal voltage requirement of the FPGA was easily satisfied by having a high output on one of the pins of the CPLD. It was consequently found that the configuration memory would not reset, because the FPGA still had too much power. This problem was remedied by forcing the interface pins to ground and disabling the internal pull-up resistors in the CPLD. Alternatively, a CPLD with an output-disable pin could have been used.

### 8.4.3 Communication link

The control room is located approximately 80m from the therapy station due to safety reasons. The experiment was controlled from the control room, because the room is also used to control the beam. Apart from the normal monitoring and control systems used during proton therapy, no communication connection to the therapy station exists. A communications link had to be designed to connect the test board with the control software.

The experimental setup required that two RS-232 serial links as well as a parallel JTAG cable was patched to a significantly greater range than specified by the original communication standards. The serial links was used to communicate with the AVR and PicoBlaze, while the JTAG connection was used for readback of the FPGA configuration memory.

Various options were available for extending the range. A radio frequency link was initially considered, but the reinforced concrete shielding of the vault made it unlikely to work reliably.

When considering wired transmission, a differential protocol was immediately attractive, due to the noise immunity and greater distance it would provide. To minimise disruption in the existing SEU simulation system, the output from the PC was converted to TTL levels. This was performed by RS-232 transceivers on the serial links and a TTL buffer for the JTAG cable. The TTL signal was fed into RS-485 drivers (Texas Instruments SN65HVD5x full-duplex RS-485 drivers and receivers) which transmitted the data across a 100m 20-core telephone cable. The remote transceiver converted the differential RS-485 signal to TTL levels and used RS-232 drivers to generate the original signal protocol for transmission to the test board. Exactly the same procedure was followed for transmitting signals back to the computer.

This cable was laid down the hallway from the control room to the proton vault. By only using one communication cable and one power cable, the experiment could be set up quickly and easily.

The JTAG output of the parallel port on the host PC proved to be very noisy. This noise was enough to reach the threshold voltage for the TTL-buffers, resulting in a corrupted signal being supplied to the RS-485 converters. Small capacitors were placed on the

traces to clean the signals before the buffers. The use of a shielded cable between the transceiver and the PC would probably also have solved the problem.

This system provided a high quality signal, only the transmission delay (see below) caused some trouble. The RS-232 links were tested with a loop-back plug and performed faultlessly at 230k baud over 200m; far exceeding the requirements for the radiation test.

The schematics are provided in Appendix C.

### 8.4.4 Readback software

The JTAG link has a clock line for synchronised data transfer. It was found that the Xilinx configuration and readback software, `Impact`, failed when communicating over the cable. This can be ascribed to `Impact` attempting to communicate at 200kHz, but the transmission delay over the 100m cable is about 800ns. The buffers and transceivers caused approximately 300ns of this delay, the rest was due to the propagation delay on the cable itself:

$$\begin{aligned} v_{cable} &= 0.66 \times c \\ &= 0.66 \times 3 \times 10^8 \\ &= 1.98 \times 10^8 m/s \\ \text{With } l = 100m: & \\ t_{cable} &= l/v_{cable} \\ &= 505ns \end{aligned}$$

The problem lay in the high communication speed, but `Impact` offered no setting to decrease it for the programmer in use, so other software was investigated.

Andrew Roger's `xc3sprog`[1] is open source Spartan-3 programming software for `GNU/Linux`. It could successfully identify and program the FPGA over the 100m cable, but no readback capability was available. The source code was extended to allow this.

To perform readback on a on Xilinx FPGA, one needs to give a series of commands to the JTAG logic, before one can access the readback functionality of the FPGA. The necessary JTAG commands were added to the source of `xc3sprog` by studying the information provided in Spartan-3 configuration application notes, along with the output of `Impact`.

The procedure reads data at about 100kHz, but the total readback time is still similar to that provided by `Impact`.

The main readback function is listed in Appendix B.

[1] http://xc3sprog.sourceforge.net/

### 8.4.5 Configuration upset detection

The upsets are detected as described in Chapter 6, but using the PC.

The readback function writes the read configuration data into a file. This file contains the original configuration information, current RAM and (optionally) register values. The register and RAM values are expected to change, so a mask file (`.msk`) is XOR'ed with the readback data to prevent checking of these values. The result is then compared with the reference readback bitmap (`.rbb`), as generated by the ISE tools. Any discrepancy is classified as an SEU.

A verification method was implemented in `Python` to detect SEUs and their positions in the configuration memory. It was tested by manually corrupting bits in the configuration file before uploading it to the device and then checking that only the unmasked upsets were found by the verification algorithm. This also verified the location of the detected upsets.

The source code for the verification program is listed in Appendix B.

### 8.4.6 Control software

A control and data logging program was written in `Python`. The program provides a command line interface for controlling the test, while all communication with the experimental hardware is logged. The power consumption of the FPGA is also monitored for latch-up behaviour. Interfacing between the software and the beam control system was not possible, so all beam settings and measurements had to be entered by the operator.

#### Initialisation

The program starts the test by checking communication with the AVR. If this succeeds, it attempts to establish contact with the PicoBlaze, warning if it is not found. A detection scan is also run on the JTAG chain, to verify that the FPGA is accessible.

Readback is then verified by twice reading the configuration memory of the static device. Both readback files are compared with a reference file, an upset may indicate problems in the communication setup, or permanent damage to the chip due to a stuck bit.

#### Static testing

For static testing the device is reset and exposed to the beam for a preset dose. When the dose has been delivered, the operator initiates readback of the configuration memory. The number of upsets, dose, beam on-time and current are logged.

Static testing requires the binary comparison of two files (the readback data with a reference). This was done by calling the utility `cmp` from the software and parsing the output to find the upset locations.

At the end of static testing, the program generates a graph of the dose-upsets relationship, which is used to set the dose for dynamic testing.

#### Dynamic testing

Dynamic testing is similar to the testing done during simulation, in that the PicoBlaze repeatedly executes the same self-checking program and reports the results to the host PC. The run is stopped by the user when the required dose has been delivered, and readback is performed to determine the number of upsets. All communication between the PicoBlaze and PC is logged, as well as current and beam information.

### 8.4.7 Power

The system requires 200mA at 12V. 100m flex cable was used to provide power to the remote equipment. Linear regulators were used to clean and regulate the voltage provided to the test boards. The voltage drop across the cable was measured as less than 1.5V under normal current conditions.

### 8.4.8 Shielding

The filter on the beam was regarded as adequate protection for the supporting hardware, no other shielding was used. The expected out of beam fluence is less than 1% of that in the beam, with background neutron radiation being the primary concern.

When considering SEUs the microcontroller on the configuration controller board is the most sensitive component, but the memory cross-section was regarded as small enough to be at low risk. Recovery from an SEU would involve a manual power off/on reset of the microcontroller. This could easily and quickly be performed from the control station, causing a only a minor disruption in the test sequence.

Total dose effects would probably cause the power supply components to fail, but the short duration of the tests made any TID effects negligible. TID induced failure is also normally preceded by a gradual increase in current, which would serve as warning to terminate the test and allow the devices time to anneal.

No radiation effects were observed on the supporting hardware, therefore confirming the validity of this approach.

### 8.4.9 Beam setup

The measurement and control equipment in the therapy station is geared towards the treatment of patients. The accuracy of the delivered dose and position of the beam epicentre are therefore the most important considerations.

The required dose is set before switching the beam on. The beam is switched on by removing blocks in the beam line, which allow protons to pass. It takes approximately 3 to 5 seconds from switching on the beam to when the first particles hit the target. The beam is stopped by lowering the blocks into the beam. This is much faster than switching the beam on, because it is also used as an emergency shutdown mechanism.

The cross-section of a semiconductor device is dependent on the energy of the particles. Very few upsets are observed below a certain threshold energy (<20MeV). Above the threshold, the cross-section increases sharply before reaching a saturation value. This can be seen in Figure 8.4, which shows the cross-section for different energy levels on a Xilinx Vitex-II.[64]

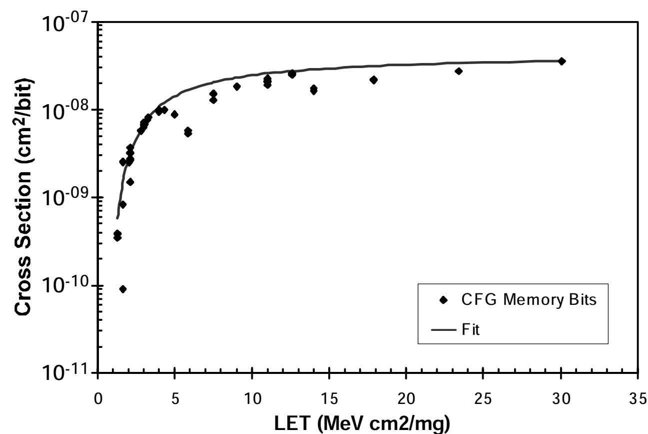



**Figure 8.4:** Cross-section vs. effective LET curve for configuration memory bits on a Xilinx Virtex-II.[64]

The measurement equipment functions reliably at energy levels greater than 80MeV. High energy levels increase the risk of permanent damage to the device due to stuck bits and SEL. 100MeV protons were therefore used in all runs; the energy level is also high enough that the saturation cross-section can be measured.

The beam is calibrated by measuring the stopping distance of protons in a water tank. According the ICRU Report 49[65], the stopping power of protons at 100MeV is $7.29 \times 10^6$ eV.cm$^2$/g. The tank is then removed and the FPGA is placed in the beam. Lasers are used to align the FPGA in the beam, relative to the room. This allows the setup to be exactly recreated if tests need to be continued on another occasion.

A 30mm×30mm square collimator is used to shield the rest of the board from the beam. The collimator is large enough to allow the entire 18mm×18mm FPGA package to be irradiated. No other active components were irradiated.

The Spartan-3 Starter Kit and configuration controller boards were mounted on a plywood board, which was held in a vice that was placed on the therapy chair in the proton treatment vault. The chair can be moved to position the FPGA correctly.

Figures 8.5 and 8.6 show the setup in the proton treatment vault. A close-up of the FPGA board is shown in Figure 8.7.

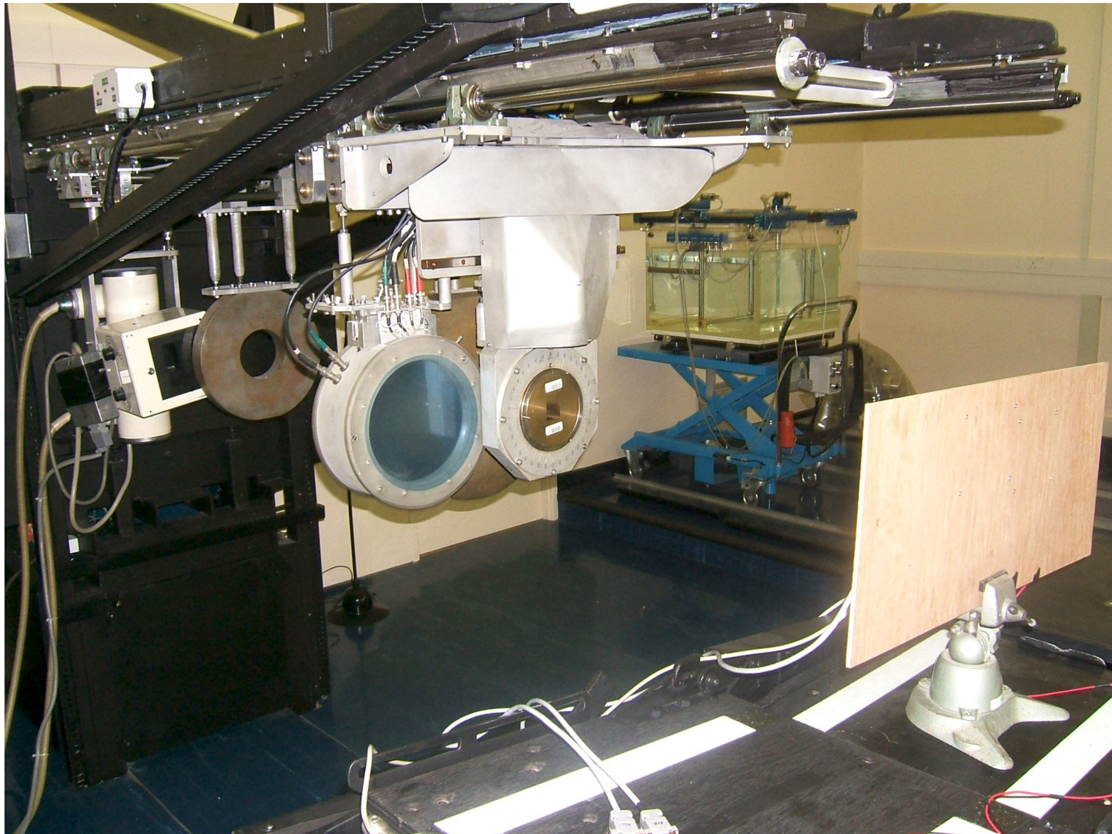


**Figure 8.5:** A photo of the setup in the proton treatment vault. The beam comes in from through the structure on the left, which measures and collimates the beam. The board stands on the chair used for therapy. The water tank used for calibration is visible in the background.

## 8.5 Results

The experiment was performed in three sessions over two months. Every session took approximately four hours.

The first session was used to determine an approximate cross-section for the device by static testing, as well as testing the performance of the reference design. During

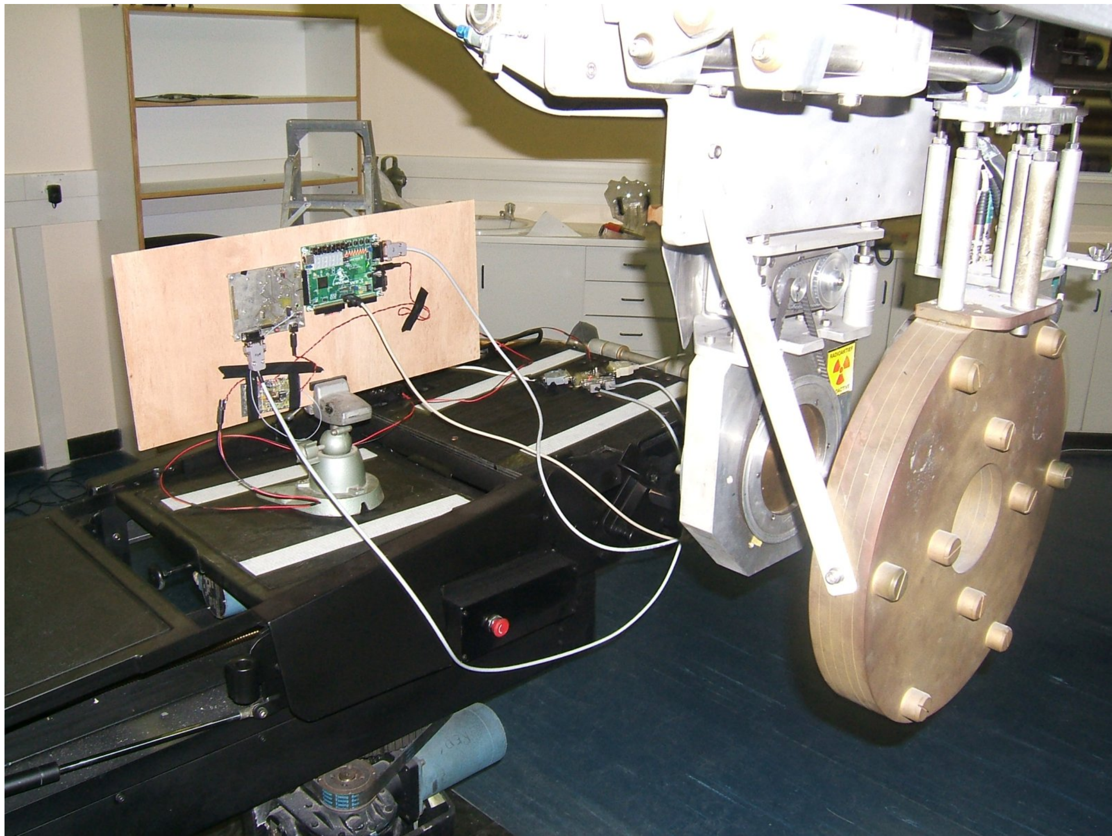


**Figure 8.6:** A photo of the FPGA in position in the proton treatment vault. The collimator is on the right, the S3Kit and configuration controller boards are standing on the treatment chair. The transceiver is visible in the centre of the image, lying on the chair.

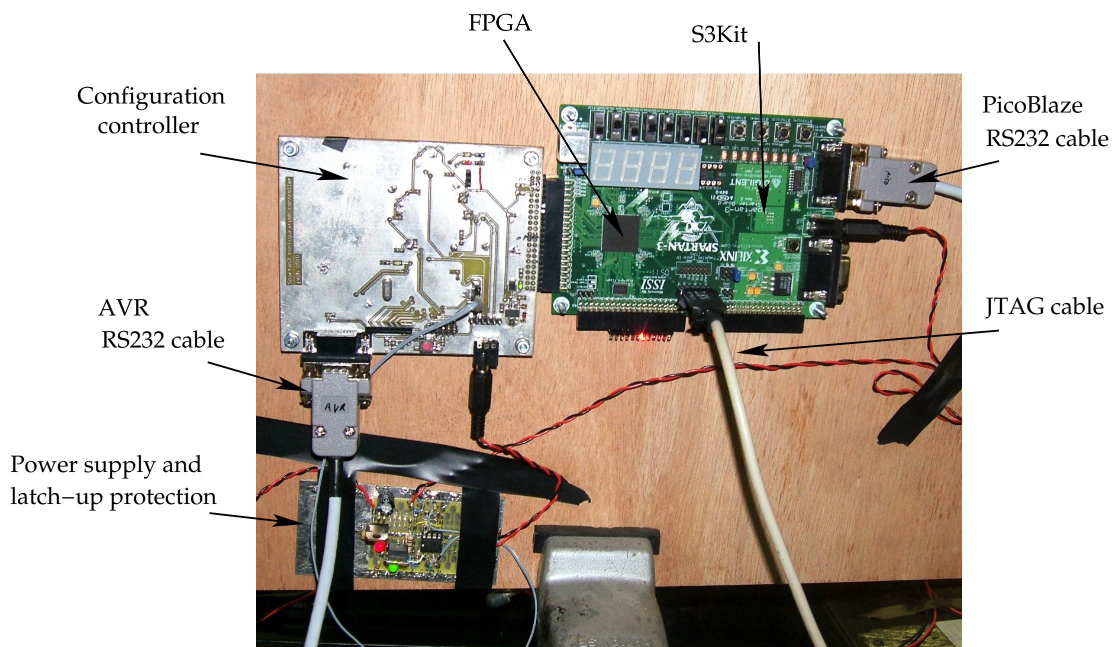



**Figure 8.7:** Close-up of the S3Kit and configuration controller in the treatment vault.

the second session the static cross-section was again measured, and the two fault tolerant designs were tested. When the results of the two sessions were compared, vastly different cross-sections and upset rates were found. A third session was scheduled to identify which of the previous sessions was correct and to obtain better measurements. The measurements of the second session were repeated in the third session, and the measured cross-section and upset rates correlated well with the first session.

The measurements from the second session were disregarded in the following calculations. The discrepancy with the other measurements can most likely be ascribed to an error in the experimental setup.

### 8.5.1 Initial cross-section measurements

The FPGA was reset and exposed to incremental doses of protons to determine the relationship between the dose and the number of upsets. After every dose, the configuration was read back through the JTAG port and compared with the original configuration memory.

During the first session, doses of $3.85 \times 10^8$ p/cm$^2$and $3.85 \times 10^9$ p/cm$^2$were used. Attempts to measure the number of upsets at higher doses were prevented by high current conditions, so it was decided to continue with dynamic testing, which was expected to reinforce the already measured values.

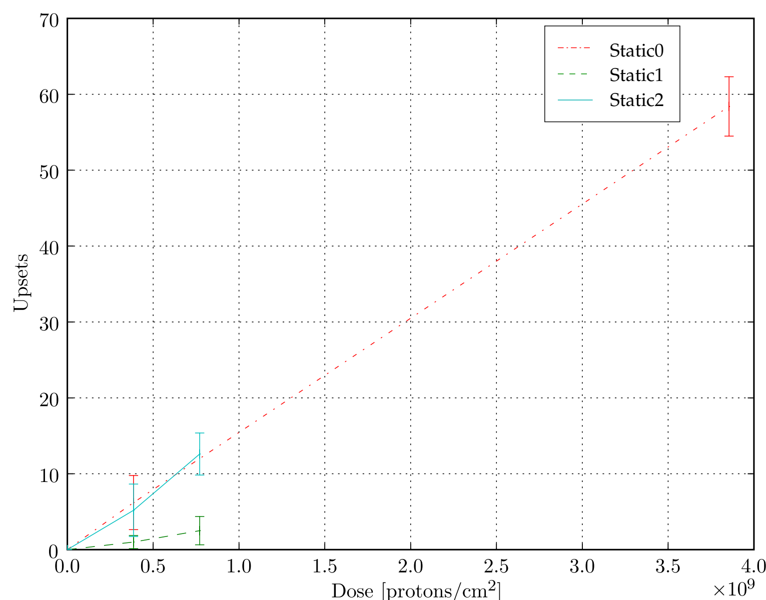



**Figure 8.8:** Measured relationship between dose and number of upsets for static XC3S200, with error bars showing the standard deviation. The individual sessions are plotted separately. The inconsistent values measured during the second session are clearly visible.

The procedure was repeated at the second and third sessions, using $3.85\times10^8$ p/cm$^2$and $7.7\times10^8$ p/cm$^2$doses. The second session delivered much lower values than expected, the third session corresponded closely with the first test, as can be seen in Figure 8.8.

The graphs were extended to pass through the origin, as can be expected for the upset mechanism. The graphs all display the expected linear relationship between dose and the number of upsets. The much lower values measured in the second session are clearly visible.

It was repeatedly verified that resetting the device successfully clears flipped bits in the configuration: no persistent errors were observed.

This dose-to-upset ratio was used to determine the doses used during dynamic testing. The values were compensated to determine the static cross-section, as discussed later.

### 8.5.2 Latch-up

High current conditions, indicative of latch-up, were observed on several occasions, especially during tests of long duration.

The latch-up threshold was initially set too high which resulted in the board drawing a large current for a several minutes. The voltage regulator on the S3kit seemed to saturate at 280mA (at 5V), which prevented more current from being drawn. The latch-up protection circuitry functioned correctly after the threshold was lowered. Switching the FPGA off and on successfully cleared the condition.

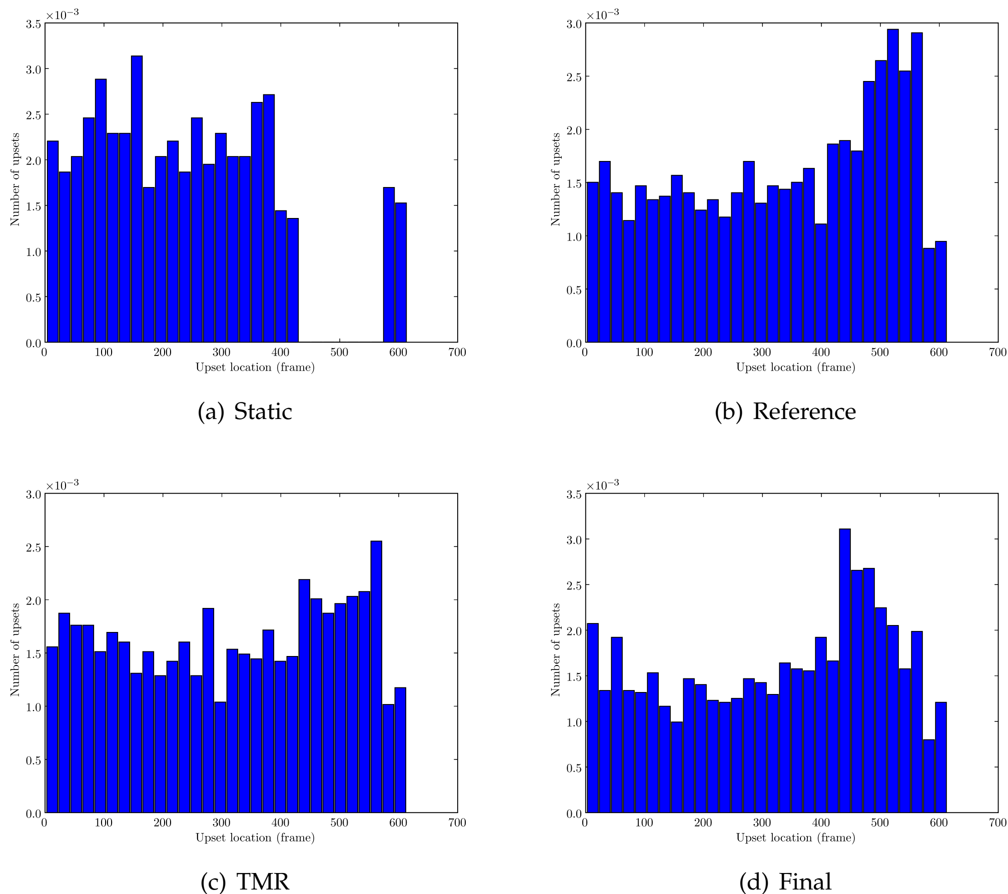
The Spartan-3 seems quite resilient to high-current, as no permanent damage was observed in the device. All other designs tested on the board functioned perfectly.

### 8.5.3 Upset distribution

The location of the upsets was investigated to check whether the upset distribution across the configuration memory is uniform.

However, it was found that the device displays two distinct zones with different sensitivities to SEUs. If one considers the histograms in Figure 8.9, that display the number of upsets in the different regions of the configuration memory, a noticeable difference exists between the zone from frame 430 to frame 570 and the rest of the device. Referring back to Chapter 6.1.3, where the mapping of the configuration memory to hardware was discussed, this area can be seen to relate to the Block Select RAM (BRAM) of the FPGA (frames 425 to 577).

The BRAM area appears to be more sensitive to upsets than the rest of the configuration memory, except for the case of the static design, where no upsets were recorded. Further investigation revealed that the BRAM is initialised to `0x3FF` when the device is reset, as can be seen in the image in Figure 8.10. Although this appears to indicate that




(a) Static (b) Reference

(c) TMR (d) Final

**Figure 8.9:** Distribution of upset locations in the configuration memory for different designs. Every bin corresponds to approximately 20 configuration frames. The BRAM can be seen to have a different sensitivity.

0 to 1 bit-flips are more frequent, this only holds true for the static design. No biasing in the flip direction could be discerned in the dynamic designs, which correlates with the tests performed by Xilinx.

It is therefore presumed that the BRAM data that is read back from the static design is not the true contents of the BRAM. Upsets in the BRAM area were consequently not observable in the static design.

### 8.5.4 Relative bit cross-sections

The dynamic upset distribution was used to determine the relative bit cross-section of the configuration memory ($\sigma_{config(bit)}$) and BRAM ($\sigma_{BRAM(bit)}$) areas.

All the dynamic designs have a significant proportion of the BRAM that is masked during readback, which prevents the detection of upsets in that area. The number of measured upsets was increased by the area covered by the mask for the calculations, because the distribution of SEUs in the BRAM area is uniform.

**Figure 8.10:** Configuration memory after reset,showing the BRAM initialised to `0x3FF`. 0 bits are grey, 1 bits are black.

The density of upsets, as given by the histograms in Figure 8.9, can now be used to determine the relative bit cross-sections of the two areas:

$$\frac{\sigma_{BRAM(bit)}}{\sigma_{config(bit)}} = \frac{\text{(Upset density in BRAM)} \times \text{Mask area}}{\text{(Upset density in config)}} \tag{8.2}$$

The relative cross-section used in all further calculations was obtained by taking the average of the relative cross-sections of the three dynamic designs:

$$\sigma_{BRAM(bit)} = 1.75\sigma_{config(bit)} \tag{8.3}$$

### 8.5.5 Static cross-section

The static upset measurements were scaled according to Equation 8.3 to compensate for the lack of upsets in the BRAM section. The dynamic measurements were scaled to compensate for the number of masked bits in the readback verification file (10% to 20% depending on the design).

The relationship between the number of upsets after compensation and the dose is shown in Figure 8.11. The static cross-section is calculated as follows:

$$\begin{aligned} \sigma &= \frac{N}{\Phi} \\ &= 2.327 \times 10^{-8}\text{cm}^2 \end{aligned}$$

The standard deviation on this cross-section is $6.171 \times 10^{-9}\text{cm}^2$.

The bit cross-sections of the configuration memory and BRAM bits can be calculated

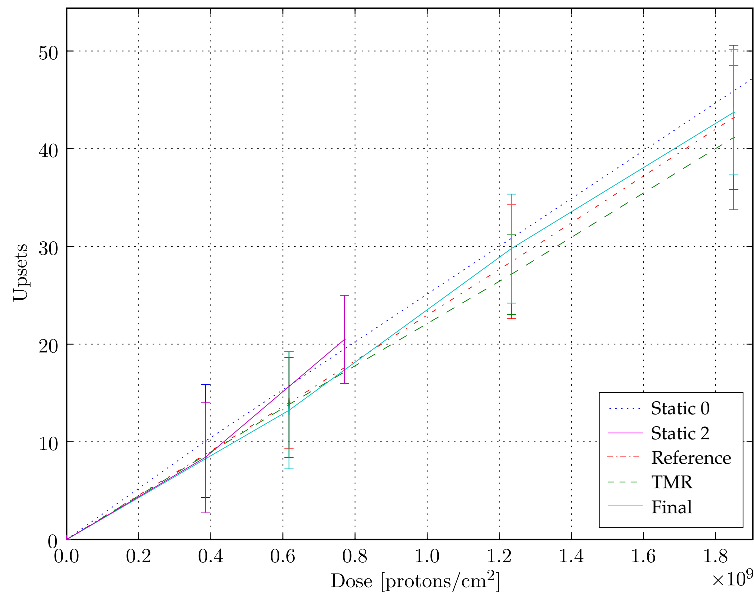



**Figure 8.11:** The compensated cross-section measurements for the different tests and designs match well. The error bars indicate the standard deviation. To improve clarity, only the dose range used for dynamic testing is shown. Static 0 and 2 are the static measurements of the first and third sessions.

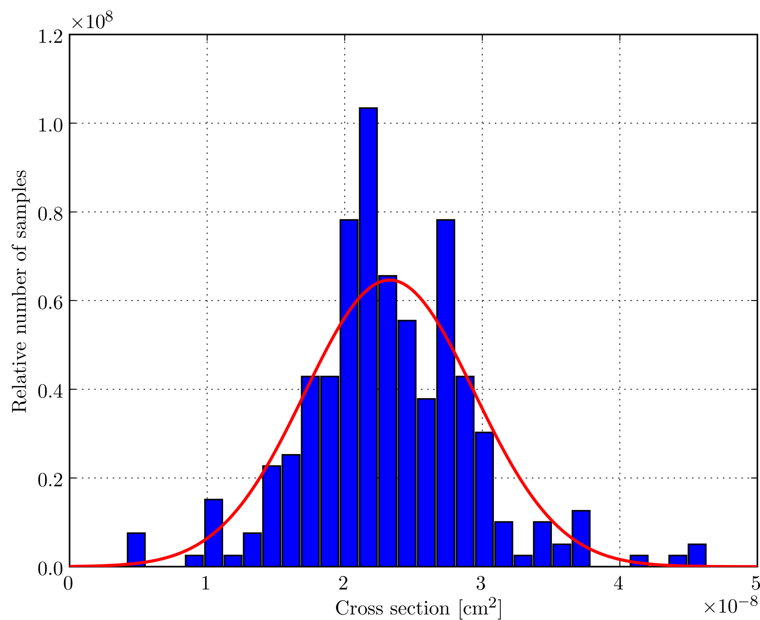



**Figure 8.12:** The measured cross-section for the tested device is $2.327 \times 10^{-8}\text{cm}^2$, with a standard deviation of $6.171 \times 10^{-9}\text{cm}^2$. This histogram shows the distribution of values obtained from the individual runs. (282 runs were used to compile this graph.)

using Equation 8.3 and the number of bits on the device:

$$\begin{aligned}
\sigma &= n_{config}\sigma_{config(bit)} + n_{BRAM}\sigma_{BRAM(bit)} \\
&= n_{config}\sigma_{config(bit)} + 1.75 \times n_{BRAM}\sigma_{config(bit)} \\
\Rightarrow \sigma_{config(bit)} &= \frac{\sigma}{n_{config} + 1.75 \times n_{BRAM}} \\
&= \frac{2.327 \times 10^{-8}}{821856 + 1.75 \times 221184} \\
&= 1.925 \times 10^{-14}\text{cm}^2 \\
\Rightarrow \sigma_{BRAM(bit)} &= 1.75\sigma_{config(bit)} \\
&= 3.368 \times 10^{-14}\text{cm}^2
\end{aligned}$$

These bit cross-sections can be used to calculate the static cross-sections of other Spartan-3 devices. Only one device was measured, so it should be kept in mind that commercial devices show variations in these values, depending on the mask set and manufacturing line.

### 8.5.6 Dynamic cross-section

The very small cross-section predicted by the SEU simulator for the fault tolerant designs makes it difficult to accurately measure the scaling factor, without performing thousands of tests. This difficulty was overcome by determining the failure rate for a relatively large number of upsets, then using that to determine the failure rate for a single upset.

Testing of a design can be modelled as a set of Bernoulli trails (where "Pass" and "Fail" are the only possible outcomes).[66] The probability that a single event upset will occur in one of the sensitive bits, causing the design to fail, is given by:

$$P\{\text{sensitive bit upset}\} = \binom{N}{k} p^k (1-p)^{N-k} \tag{8.4}$$

with $k$ the number of upsets in sensitive bits, $N$ the total number of upsets and $p$ the probability that an upset will cause the design to fail.

If the design is still functioning after a run, one assumes that no upsets were experienced in any of the sensitive bits. Therefore $k = 0$ and $p = \alpha$, which gives:

$$P\{pass\} = (1-\alpha)^N \tag{8.5}$$

$$\Rightarrow \alpha = 1 - \sqrt[N]{P\{pass\}} \tag{8.6}$$

The pass rates measured for the different designs are listed in Table 8.1. The average

**Table 8.1:** Measured pass rates and average number of upsets for tested PicoBlaze designs. The columns give the pass rate and average number of upsets for the listed dose. The pass rate gives the measured probability that a design will still be functioning after receiving the specified dose. The number of upsets has been adjusted to compensate for the masking of upsets by the mask file.

| **Dose (p/cm$^2$)** | **$6.168 \times 10^8$** | | **$1.234 \times 10^9$** | | **$1.85 \times 10^9$** | |
|---|---|---|---|---|---|---|
| | Pass rate | Upsets | Pass rate | Upsets | Pass rate | Upsets |
| **Reference** | 0.559 | 13.987 | 0.500 | 27.891 | 0.286 | 42.384 |
| **TMR** | 0.774 | 14.293 | 0.613 | 27.642 | 0.481 | 41.245 |
| **Final** | 0.788 | 14.108 | 0.600 | 30.374 | 0.484 | 44.902 |

number of upsets for the individual designs was used to determine the scaling factors from Equation 8.6.

The results are plotted in Figures 8.13, 8.14 and 8.15. A straight, horizontal line is expected in all cases, although some statistical uncertainty is also present. The values for the reference design are fairly scattered due to it being composed of fewer samples than the other two designs. The mean of these values was used in further calculations. Table 8.2 summarises the scaling factors at the different doses.

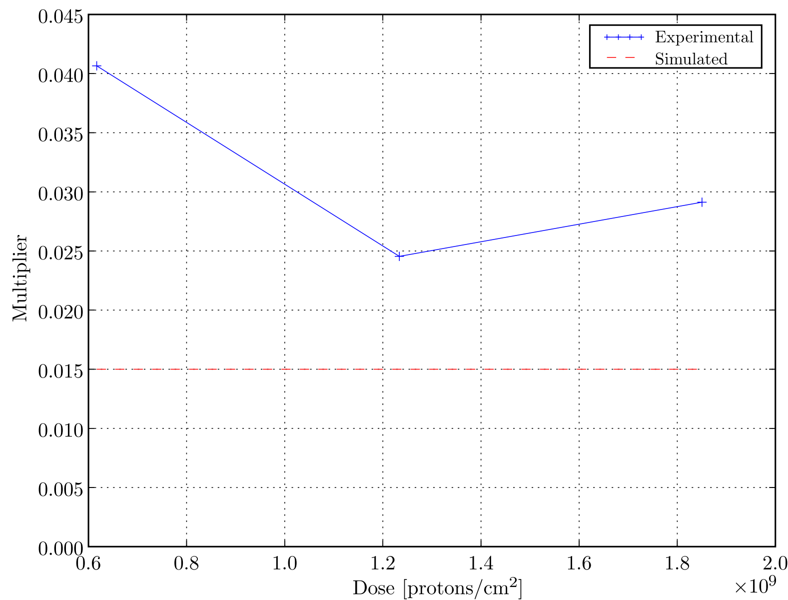



**Figure 8.13:** Measured scaling factor for the reference design (Design A). The scaling factor as predicted by the SEU simulator is also shown.

It is evident that the scaling factor for the reference design is larger than those of the two mitigated designs. The mitigated designs show approximately the same scaling factor. This is similar to what was predicted by error injection, but the measured results are all significantly greater than the simulator predicted.

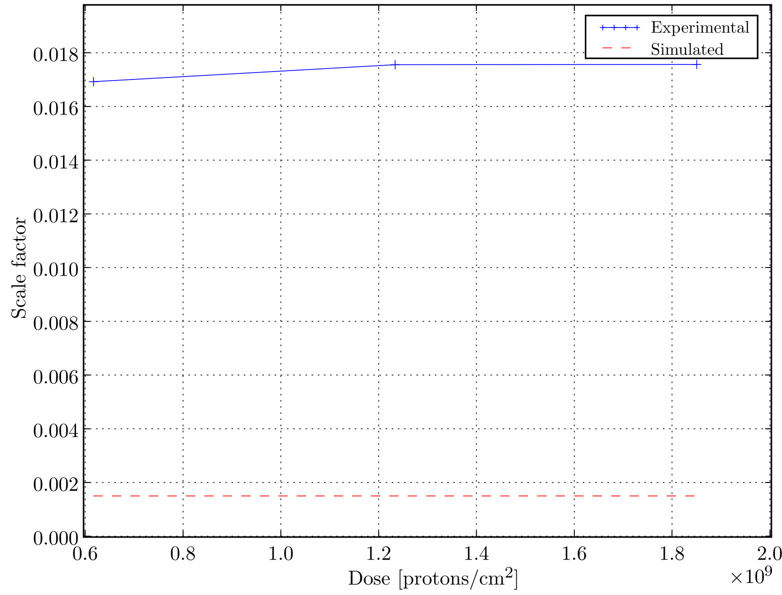



**Figure 8.14:** Measured scaling factor for the design with high level TMR.

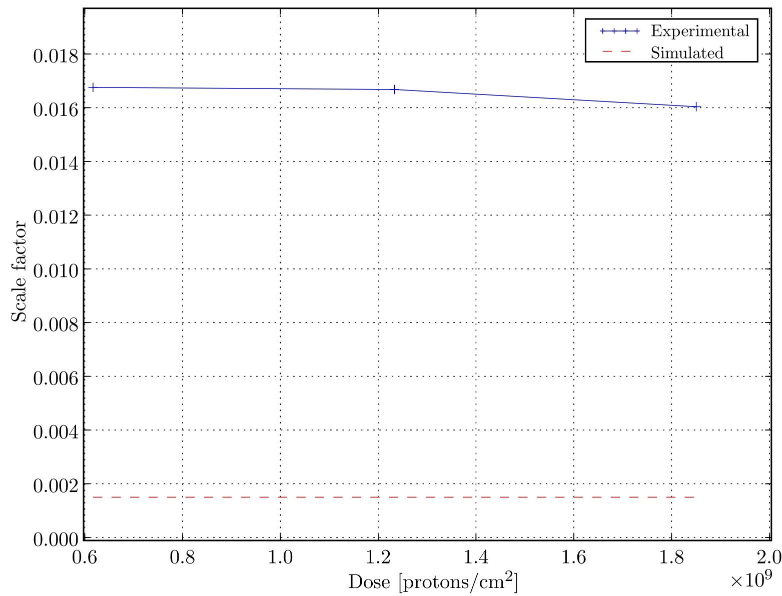



**Figure 8.15:** Measured scaling factor for the final design.

**Table 8.2:** Measured scaling factors ($\alpha$) for tested PicoBlaze designs at different doses. The mitigated designs have smaller scaling factors, which means the dynamic cross-section is smaller.

| **Dose (p/cm$^2$)** | $6.168 \times 10^8$ | $1.234 \times 10^9$ | $1.85 \times 10^9$ |
|---|---|---|---|
| **Reference** | 0.0407 | 0.0245 | 0.0291 |
| **TMR** | 0.0169 | 0.01753 | 0.01756 |
| **Final** | 0.0167 | 0.0166 | 0.01601 |

**Table 8.3:** Comparison of measured and simulated dynamic cross-sections. The discrepancy between the values measured through radiation testing $\sigma'_{(rad)}$ and error injection $\sigma'_{(sim)}$ is mostly likely due to the SEFI cross-section, $\sigma_{SEFI}$. Although this term makes a significant contribution to the cross-section for the tested designs, this is only due to the small size of the reference design and the effectiveness of the error mitigation techniques in the other designs. All values are in $cm^2$.

| | $\sigma'_{(rad)}$ | $\sigma'_{(sim)}$ | $\epsilon$ |
|---|---|---|---|
| Reference | $7.687\times10^{-10}$ | $3.460\times10^{-10}$ | $4.227\times10^{-10}$ |
| TMR | $4.035\times10^{-10}$ | $2.393\times10^{-11}$ | $3.796\times10^{-10}$ |
| Final | $3.832\times10^{-10}$ | $3.204\times10^{-11}$ | $3.512\times10^{-10}$ |

### 8.5.7 Comparison with SEU simulator

The radiation testing results have to be compared with the SEU simulator measurements to evaluate the simulator.

The simulator determined the $\alpha_{BRAM}$ and $\alpha_{config}$ values, while the radiation test provides the different $\sigma$ values and an $\alpha$ for the complete design.

The simulated and measured dynamic cross-sections are compared as follows:

$$\sigma'_{(rad)} = \sigma'_{(sim)} + \epsilon \tag{8.7}$$

$$\alpha_{(rad)}\sigma_{(rad)} = \alpha_{config(sim)}\sigma_{config(rad)} + \alpha_{BRAM(sim)}\sigma_{BRAM(rad)} + \epsilon \tag{8.8}$$

The error term ($\epsilon$) is used to measure the mismatch between left- and right-hand side of the equation. It contains all the dynamic cross-sections that could not be measured by the simulator, as well as a small measurement uncertainty.

The dynamic cross-sections for the different designs, as well as the resulting $\epsilon$ values, are listed in Table 8.3. The measured cross-section and mean $\alpha$ values were used in the calculations.

From Equation 4.4:

$$\epsilon = \sigma_{SEFI} + \alpha_{FF}\sigma_{FF} + \alpha_{SET}\sigma_{SET} + \alpha_{HL}\sigma_{HL} \tag{8.9}$$

The similarity in $\epsilon$ values indicate that the observed discrepancy is most likely due to the measurent of the SEFI cross-section ($\sigma_{SEFI}$). The size of $\sigma_{SEFI}$ is comparable to that measured in Virtex devives in testing at JPL. Further experiments to confirm this conclusion can still be performed.

The significant influence of $\sigma_{SEFI}$ on the cross-section of the tested designs is due to the small size of the reference design (it uses approximately 6% of the configuration memory) and the protection afforded by the mitigation techniques in the other two designs. The configuration and BRAM dynamic cross-sections will increase with larger, unprotected designs, which will make their contribution to the dynamic cross-section dominate. The effective application of error mitigation techniques in the TMR and

reference designs has decreased these cross-sections to well below $\sigma_{SEFI}$.

It can be assumed that the techniques that protect against upsets in the configuration will also protect against SETs, upsets in flip-flops and upsets in half-latches. Therefore $\epsilon$ measured in the protected designs must be approximately equal to each other and also to the SEFI cross-section:

$$\sigma_{SEFI} \approx 3.65 \times 10^{-10}\text{cm}^2 \tag{8.10}$$

A similar SEFI cross-section can be expected on other Spartan-3 devices. This cross-section should therefore be added to future error injection results to obtain more accurate dynamic cross-section estimates.

$\epsilon$ for the reference design is larger, because the other SEU cross-sections are not mitigated.

### 8.5.8 Typical LEO failure rates

The radiation data was used to determine approximate failure rates for the tested designs in a space environment.

The 1996 Cosmic Ray Effects on Micro-Electronics (CREME96) is a widely-used suite of programs for creating numerical models of the radiation environment near the earth. It is used to evaluate the radation effects on electronic devices in orbit and provides estimates of the high LET radiation environment for manned spacecraft. Comparison to on-orbit data has demonstrated the accuracy of the model. The Naval Research Laboratory hosts a website that allows users to perform calculations with the CREME96 program suite.[67]

CREME96 was used to calculate the expected upset rates for a Spartan-3 XC3S200 in low earth orbit. The calculations were performed for a spacecraft in a 1000km sun-synchronous orbit with an inclination of 99.48°. Average flux conditions were used, ie. solar minimum activity without flares. 3mm aluminium was used as shielding.

The Spartan-3 cross-section was approximated using a Weibull curve, similar to the curves measured in JPL testing of Virtex-II devices.[68] The onset energy (level below which no upsets occur) was set to 1MeV and the measured cross-sections of the different components were used as plateau.

Figure 8.16 shows the calculated fluxes experienced by the device (after shielding). Note that protons are the greatest radiation source. This validates the use of protons for radiation testing and SEU rate calculations.

The results for proton induced upset calculations are listed in Table 8.4. This can be used to determine the expected failure rates for the irradiated designs. The configuration and BRAM upset rates are scaled with their repsective $\alpha$ values and added to

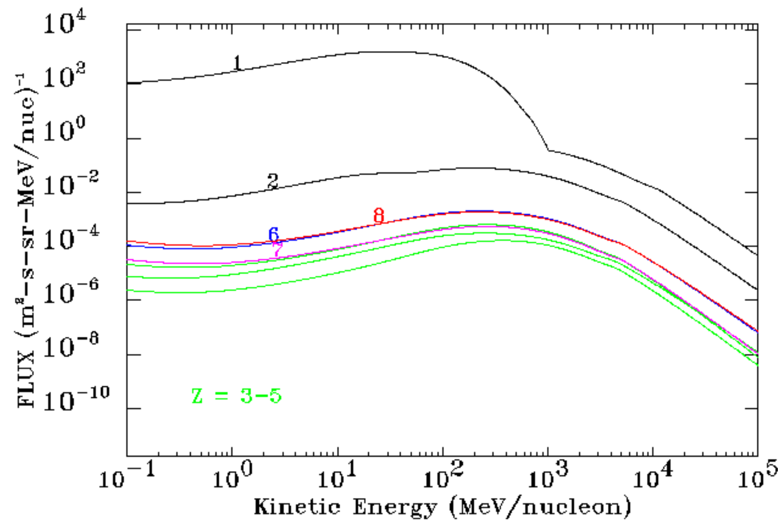



**Figure 8.16:** Fluxes after shielding in 1000km sun-synchronous orbit for eight most prevalent elements. The majority of the radiation is caused by protons (Z=1).

**Table 8.4:** Proton induced upset rates according to CREME96 for a Spartan-3 XC3S200. A 1000km sun-synchronous orbit in average solar conditions was used in the calculations.

| | SEUs/bit/sec | SEUs/bit/day | SEUs/device/sec | SEUs/device/day |
|---|---|---|---|---|
| Configuration | $4.94 \times 10^{-12}$ | $4.27 \times 10^{-7}$ | $4.02 \times 10^{-6}$ | $3.47 \times 10^{-1}$ |
| BRAM | $8.65 \times 10^{-12}$ | $7.47 \times 10^{-7}$ | $1.91 \times 10^{-6}$ | $1.65 \times 10^{-1}$ |
| SEFI | - | - | $9.37 \times 10^{-8}$ | $8.07 \times 10^{-3}$ |

**Table 8.5:** Expected failure rates for tested designs in space.

| Design | Failures/device/day |
|---|---|
| Reference | 0.01617 |
| High-level TMR | 0.00870 |
| Final | 0.00885 |

the SEFI upset rate. The failure rates for the tested designs are listed in Table 8.5. As expected, the hardened designs are more robust to SEU induced failures.

## 8.6 Conclusion

The radiation test showed that iThemba LABS can be used to test the response of electronic devices to SEUs. The test setup was significantly simpler and more manageable than the previous test at iThemba.

Three designs were irradiated to compare the SEU simulator results to actual radiation testing. The designs with error mitigation performed better than the one without mitigation, as predicted by the simulator. The SEU simulator performs well in measuring the effect of error mitigation on an HDL level.

**Table 8.6:** Static cross-section measurements for Spartan-3 XC3S200.

| Component | Cross-section ($cm^2$) | Percentage |
|---|---|---|
| $\sigma_{config}$ | $1.58\times10^{-8}$ | 66.94% |
| $\sigma_{BRAM}$ | $7.45\times10^{-9}$ | 31.52% |
| $\sigma_{SEFI}$ | $3.65\times10^{-10}$ | 1.54% |
| Total | $2.36\times10^{-8}$ | 100% |

A constant offset was observed when the radiation and simulator results were compared. This can be satisfactorily explained by the SEFI cross-section, which can not be measured by the SEU simulator. This cross-section stays fairly constant for different Spartan-3 devices and should therefore be added to future fault injection results to obtain a more accurate estimate of the dynamic cross-section. Further radiation experments to confirm this is recommended.

The SEFI cross-section made a significant contribution to the dynamic cross-section for all the tested designs. It is important to note that this is only the case with small designs (such as the reference PicoBlaze design) and designs with good SEU mitigation.

The static cross-section measurements are summarised in Table 8.6. The SEFI cross-section is expected to be similar for other Spartan-3 devices. The BRAM and configuration cross-sections will vary, depending on the capacity of the device. These figures form an upper bound on the dynamic cross-section. The dominant contributions by the configuration and BRAM makes the application of error mitigation techniques necessary for most designs.

$\sigma_{SEFI}$ can be considered a lower limit on the dynamic cross-section. This means that mitigation techniques are only truly productive until the configuration and BRAM dynamic cross-sections have been decreased to below 10% of this value. Further mitigation would not make any significant improvements to the SEU hardness of the device, because $\sigma_{SEFI}$ would dominate.

Latch-up was observed during testing but no permanent damage was detected, which is good.

The measured cross-sections were used to calculate (with CREME96) the expected on-orbit failure rates of the tested designs.

# Chapter 9

# Conclusions and recommendations

This document presented the development and testing of a single-event upset resistant PicoBlaze soft-core processor on an SRAM FPGA.

The application of error mitigation techniques to SRAM FPGAs was investigated to derive guidelines for achieving fault tolerance on SRAM FPGAs.

A fault tolerant version of the Xilinx PicoBlaze soft-core processor was implemented by using a developed error injection tool-set. This tool-set was used to emperically evaluate the improvement offered by different error mitigation techniques.

Three designs were irradiated with protons to test their SEU susceptibility, as well as measure the physical radiation characteristics of the Spartan-3 device. The designs with error mitigation performed better than the unprotected reference design, demonstrating the effectiveness of HDL mitigation. The measurements were compared with the simulator results to measure the SEFI cross-section, which can not be changed by HDL mitigation.

## 9.1 Results

### 9.1.1 Error mitigation

Guidelines for achieving fault tolerance in SRAM FPGAs were derived after an extensive literature study.

Comprehensive protection of a design on an SRAM FPGA requires mitigation on HDL level, as well as external protection against latch-up and corruption of the configuration memory.

SRAM FPGAs are particularly sensitive to upsets in the configuration memory. Therefore spatial redundancy, such as TMR, is recommended on HDL level to minimise the disruption caused.

The recommended error mitigation strategy entails applying high-level TMR to a design and then optimising areas where sufficient protection can be achieved through more efficient means. Simulation of the resulting design can be performed to obtain an approximate dynamic cross-section.

### 9.1.2 SEU simulator

A set of tools, consisting of an SEU simulator and support software, has been developed.

The simulator can be used to measure dynamic cross-section of a design efficiently by injecting errors into the configuration memory of a Xilinx FPGA. This allows the effectiveness of different mitigation strategies to be compared, without having to resort to the cost and administration involved in radiation tests. The software tools include the control software for simulation and the configuration visualisation program.

The SEU simulator can be used to compare functionally equivalent designs with different error mitigation strategies to obtain the best design. It can be used to determine an approximate dynamic cross-section, when combined with radiation measurements for the configuration bit cross-section ($\sigma'_{config(bit)}$), BRAM bit cross-section ($\sigma'_{BRAM(bit)}$) and the SEFI cross-section ($\sigma_{SEFI}$).

### 9.1.3 Fault tolerant soft-core processor

The PicoBlaze and LEON3 soft-core processors were investigated as candidates for a radiation tolerant implementation. The PicoBlaze was selected based on available hardware and because its smaller size allowed several fault tolerant versions to be implemented and compared.

A final implemetation that offers a good trade-off between the mitigation overhead and the hardness was derived. This design uses Hamming encoding to protect the program memory and some output pins. TMR is used to protect the rest of the design. The radiation tolerance is similar to that afforded by a full TMR implementation, but usage of scarce resources such as BRAM and hardware multipliers is lower.

### 9.1.4 Radiation testing

Three designs were irradiated at iThemba LABS to verify the protection afforded by the mitigation techniques and to evaluate the accuracy of the SEU simulator. The physical radiation characteristics of the Spartan-3 device were also measured.

The hardended designs performed better than the unprotected design, indicating that the HDL mitigation was successful. However, the dynamic cross-section differed with a

constant offset from the predictions of the SEU simulator. This difference was identified as the SEFI cross-section ($\sigma_{SEFI}$).

## 9.2 Recommendations

### 9.2.1 Error mitigation

The use of active readback and partial reconfiguration to improve the dependability of designs on SRAM FPGAs can still be investigated.

The methods discussed and implemented were considered with Xilinx SRAM-based FPGAs as target. Altera devices have architectural differences that may need to be taken into account. The protection of registers and memory were regarded as sufficient for flash and antifuse devices, actual radiation testing of these devices with different EDAC strategies can still be done. Flash devices offer especially interesting posibilities, with the low SEU cross-section, low power consumption and the ability to be reconfigured.

The automated application of error mitigation techniques with software can also be investigated further. Once an appropriate model of a design has been constructed from the HDL or EDIF description, it should be fairly easy to apply selected hardening techniques. This will make the hardening process one step in the design cycle, freeing the developer from having to keep it in mind.

### 9.2.2 Configuration controller

If the configuration controller is implemented on a larger FPGA or CPLD, a soft-core processor can be adapted to perform the duties of the configuration controller. This will decrease the component count and possibily lead to an increase in performance.

A further extension would be to store instructions with the configuration data in the flash memory. In this design, the data on the flash will become a program that the configuration controller executes to perform configuration and selective readback. A post-processing routine can format the configuration data before it is downloaded to the flash memory. Only a small instruction set is required, but it will greatly simplify advanced configuration functions.

### 9.2.3 SEU simulator

The SEU simulator has been verified to provide useful information and can now be used for development of fault tolerant designs. Only minor modifications would be necessary to accommodate other Xilinx devices. To tests designs on Altera devices, the differences in configuration flow and lack of readback on these devices would be need to be taken into account.

The relative sparseness of sensitive configuration bits in most designs can be used to significantly speed up testing. Multiple bits can be flipped at the same time, when the design fails, the bits can be upset individually to find the sensitive bit(s). This will be especially useful on large devices where tens of millions of bits need to be tested.

Further investigation of the discrepancy between the simulated and measured dynamic cross-sections should also be conducted.

### 9.2.4 Future radiation testing

If a large number of future tests are planned, it will definitely be worth the investment to develop a system that can be integrated with the existing setup at iThemba, to provide rapid and automated testing of a design.

## 9.3 Final comments

The process documented in this thesis demonstrates how a *open-source IP* and *COTS FPGAs* can be combined to develop a *robust low cost component* for use in *miniature spacecraft*.

# Bibliography


[1] J. R. Wertz and W. J. Larson, Eds., *Space mission analysis and design*, 4th ed. Kluwer Academic Publishers, 1999.

[2] S. Duzellier, "Radiation effects of electronic devices in space," *Aerospace and Technology*, no. 9, pp. 93–99, November 2004.

[3] H. Grobler, "Aspects affecting the design of a low earth orbit satellite on-board computer," Master's thesis, University of Stellenbosch, December 2000.

[4] (2005) NASA Office of Logic Design. [Online]. Available: http://www.klabs.org

[5] R. A. Serway and R. J. Beichner, *Physics for Scientists and Engineers with modern physics*, 5th ed. Saunders College Publishing, 2000.

[6] (2005, March) NASA/GSFC Radiation Effects & Analysis. [Online]. Available: http://radhome.gsfc.nasa.gov/

[7] S. Habinc, "Suitability of reprogrammable FPGAs in space applications," Gaisler Research," Feasibility Report, Sept 2002.

[8] J. Gaisler, "A Portable and Fault Tolerant Microprocessor based on the SPARC V8 Architecture," 2002.

[9] "GNU General Public License," Free Software Foundation, 1991.

[10] *LEON2 Processor User's Manual*, XST ed., Gaisler Research, January 2005.

[11] *GRLIB IP Library User's Manual, ver 0.15 beta*, Gaisler Research, 2004.

[12] *The SPARC Architecture Manual Ver 8*, SPARC International, 1992.

[13] *LEON3 Processor User's Manual*, Gaisler Research, Oct 2004.

[14] *GRFPU User's Manual*, Gaisler Research.

[15] *SPARC-V8 Embedded (V8E) Architecture Specification*, SPARC International, 1996.

[16] *AMBA bus specification (rev 2.0)*, ARM, 1999.

[17] A. Sule. (2003, Jul) LEON enhancement tutorial. [Online]. Available: http://www.ece.ncsu.edu/muse/

[18] "GNU Lesser General Public License," Free Software Foundation, 1991.

[19] *LEON Processor User's Manual*, Gaisler Research, Nov 2001.

[20] J. Gaisler, "LEON Open Source Processor," Presentation.

[21] D. Driessens and T. Tierens, "Overview of Embedded Processors: Excalibur, LEON, MicroBlaze, NIOS, OpenRISC, Virtex II Pro," De Nayer Instituut, Tech. Rep., 2003.

[22] *RTEMS 4.6.1 documentation*, On-Line Applications Research Corporation, 2004. [Online]. Available: http://www.rtems.com

[23] S. Allison, C. Wermelskirchen, H. Rarback, and T. Straumann, "Experiences With RTEMS in Production at SPEAR," Presentation, SSRL/SLAC.

[24] C. Buchacher. (2004, 10) Lion Project: Linux for the LEON SPARC Processor. [Online]. Available: http://www.hsse.fh-hagenberg.at/Studierende/hse02006/lion/

[25] (2005, March) eCos home page. [Online]. Available: http://sourceware.org/ecos

[26] *LEON Bare-C cross-sompiler system v1.0.8*, Gaisler Research.

[27] *RCC User's Manual, ver 1.0.6*, Gaisler Research, December 2004.

[28] *TSIM Simulator User's Manual (ver 1.3)*, Gaisler Research, March 2004.

[29] *GRMON User's Manual (ver 1.0.8)*, Gaisler Research, May 2005.

[30] J. Gaisler, *LEON DSU Monitor User's Manual (ver 1.0.8)*, Gaisler Research, May 2003.

[31] *UG129 (v1.1.1): PicoBlaze 8-bit Embedded Microcontroller User Guide for Spartan-3, Virtex-II and Virtex-II Pro FPGAs*, November 2005.

[32] K. Chapman. (2002, March) Xilinx Texclusive:Creating Embedded Microcontrollers (Programmable State Machines). [Online]. Available: http://www.xilinx.com

[33] M. Wirtlin, N. Rollins, M. Caffrey, and P. Graham, "Hardness by design techniques for field programmable gate arrays," 2003.

[34] P. Graham, M. Caffrey, E. D. J. N. Rollins, and M. Withlin, "SEU mitigation for half-latches in Xilinx Virtex FPGAs," *IEEE Transactions on Nuclear Science*, vol. 50, no. 6, pp. 2139–2146, 2003.

[35] M. Caffrey, P. Graham, E. Johnson, M. Wirthlin, N. Rollins, and C. Carmichael, "Single Event Upsets in SRAM based FPGAs," *Military and Aerospace Applications of Programmable Logic (MAPLD)*, 2002.

[36] A. Lesea, S. Drimer, J. J. Fabula, C. Carmichael, and P. Alfke, "The rosetta experiment: Atmospheric soft error rate testing in differing technology fpgas," *IEEE Transactions on Device and Materials Reliabiltiy*, vol. 5, no. 3, pp. 317–328, September 2005.

[37] D. W. Jensen, S. E. Koenck, and A. C. Tribble, "A radiation hardened by design approach to solve single event upsets," Presentation, Advanced Technology Center Rockwell Collins.

[38] *AT40KAL Family 5K-50K Coprocessor with FreeRAM*, ATMEL, 2004.

[39] "QPRO High-Reliability QML Certified and Radiation Hardened Products for Aerospace and Defense Applications," Xilinx," Data book, January 2000.

[40] C. Carmichael, "XAPP197 (v1.0): Triple Module Redundancy Design Techniques for Virtex FPGAs," Xilinx," Application Note, Nov 2001.

[41] S. Habinc, "Functional triple modular redundancy," Gaisler Research," Design and Assessment Report, December 2002.

[42] M. N. Damien Chardonnerau, Raijmond Keulen *et al.*, "Fault tolerant 32bit RISC processor implementation and radiation test results," Sept 2001.

[43] P. P. Shirvanii, N. Saxena, and E. J. McCluskey, "Software-Implemented EDAC Protection Against SEUs," Center for Reliable Computing, Departments of Electrical Engineering and Computer Science, Stanford University, Tech. Rep.

[44] A. Tanenbaum, *Structured Computer Organization*, 4th ed. Prentice Hall International, 1999.

[45] W. C. Huffman and V. Pless, *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003.

[46] P. K. Samudrala, J. Ramos, and S. Katkoori, "Selective Triple Modular Redundancy (STMR) based on Single-Event Upset (SEU) tolerant synthesis for FPGAs," *IEEE Transactions on Nuclear Science*, vol. 51, no. 5, pp. 2957–2969, 2002.

[47] "XAPP151 (v1.7): Virtex series configuration architecture user guide," Xilinx," Application note, October 2004.

[48] "VirtexII Platform FPGAs: Complete datasheets," Xilinx," Datasheet, March 2005, DS031 (v3.4). [Online]. Available: http://www.xilinx.com

[49] *UG130 (v1.1): Spartan-3 Starter Kit User Guide*, May 2005.

[50] "Spartan3 FPGA: Complete datasheets," Xilinx," Datasheet, January 2005, DS099 (v1.4). [Online]. Available: http://www.xilinx.com

[51] C. W. Tseng, "XAPP452 (v1.0): Spartan-3 Advanced Configuration Architecture," Xilinx," application note, December 2004.

[52] M. Ng and M. Peattie, "XAPP502 (v1.4): Using a microprocessor to configuare Xilinx FPGAs via Slave Serial or SelectMap mode," Xilinx," Application note, November 2002.

[53] "XCR3384XL: 382 Marcocell CPLD," Xilinx," Datasheet, April 2005, DS024 (v1.9). [Online]. Available: http://www.xilinx.com

[54] "ATmega128(L): 8-bit AVR microcontroller with 128K bytes in-system programmable flash," Atmel," Datasheet, November 2004. [Online]. Available: http://www.atmel.com

[55] "AT49BV322D(T): 32-megabit 3-volt only flash memory," Atmel," Datasheet, September 2005. [Online]. Available: http://www.atmel.com

[56] K. Goldblatt, "XAPP453 (v1.0): The 3.3V configuration of Spartan-3 FPGAs," Xilinx," Application note, February 2005.

[57] M. Caffrey, E. Johnson, and M. Wirthlin, "Single-Event Upset Simulations on an FPGA," *Engineering of reconfigurable systems and algorithms (ERSA)*, June 2002.

[58] M. Rebaudengo and M. Sonza Reorda and M. Violante, "Simulation-based analysis of SEU effects on SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 51, no. 6, Dec. 2004.

[59] N. Rollins, M. J. Wirthlin, M. Caffrey, and P. Graham, "Reliability of programmable input/output pins in the presence of configuration upsets," *Military and Aerospace Applications of Programmable Logic (MAPLD)*, September 2004.

[60] "iThemba Laboratory for Accelerator Based Sciences," November 2006. [Online]. Available: www.tlabs.ac.za

[61] H. Berner, "The Selection and Single Upset Event Testing of a DSP Processor for a LEO Satellite," Master's thesis, University of Stellenbosch, March 2002.

[62] E. Johnson, N. Rollins, M. J. Wirthlin, M. Caffrey, and P. Graham, "Accelerator validation of an FPGA SEU simulator," *IEEE Transactions on Nuclear Science*, vol. 50, no. 6, pp. 2147–2157, December 2003.

[63] N. J. Buchanan and D. M. Gingrich, "Proton induced radiation effects on a Xilinx FPGA and estimates of SEE in the ATLAS environment."

[64] C. Yui, G. Swift, and C. Carmichael, "Single Event Upset Susceptibility testing of the Xilinx Virtex-II FPGA," *Military and Aerospace Applications of Programmable Logic (MAPLD)*, 2002.

[65] "Stopping powers and ranges for protons and alpha particles," International Commission on Radiation Units and Measurements," ICRU Report 49, 1993.

[66] P. Z. Peebles, *Probability, Random Signals and Random Signal Principles*. McGraw Hill, Inc, 2001.

[67] (2006, December) CREME96. [Online]. Available: http://creme96.nrl.navy.mil

[68] *Virtex-II static SEU Charatersation*, Xilinx SEE Consortium, January 2004.

[69] (2005, March) Free software foundation. [Online]. Available: http://www.fsf.org

# Appendix A

# GNU General Public License

The GNU GPL has become a popular license for releasing open source software and HDL intellectual property.

This chapter provides a brief introduction to the GPL, and the implications of using software/IP covered by it.

## A.1 The GNU General Public License

The goal of the GNU General Public License is to protect the freedom of users of free software (with "free" used like freedom, not price). The GPL grants users the right to access the source code of software, to allow modification and distribution. To achieve this, the GPL restricts anyone to deny their users these rights.

In practise, this translates into granting the users of software the same rights as developers, most notably the right modify the sources and to redistribute (for a fee if so desired) the software.

The GPL allows unconstrained further development and use for academic, research and commercial purposes, however, all work derived from GPL-ed work will also be covered by the GPL. The developer may require a fee to be paid for distribution, but royalties from further use or distribution cannot be claimed, as this will limit the freedom of the user.

The Free Software Foundation[69] provides the most information on the use and implications of using the GPL.

## A.2 The GNU Lesser General Public License

The LGPL is also aimed at protecting the freedom of software users, but has less strict constraints on linking and using LGPLed software.

Unlike the GPL, the LGPL allows code to be linked with a library under another license, without requiring that the sources of the other library be made public without restriction. However, if an executable created during linking is distributed, it needs to be distributed under the LGPL, which requires all the sources to be made public.

## A.3 Implications of using the GPL

The GPL allows unlimited, royalty free use and development of software, as long as the resulting product is again distributed under the GPL. This means that a derived product has to be distributed with the full sources of the final design.

One is allowed to distribute binary libraries under a propriety license, along with a GPL-ed library (with its sources). A person or company which merges to two may use it for his own purposes, but may not distribute the derived product since he can not fulfil the GPL.

# Appendix B

# Selected source code excerpts

Due to the sheer volume of source code used in this project, only the academically significant portions are presented here.

## B.1 TMR using lookup tables

### TRV_BUFT.vhd

```
--  Adapted from xapp197
--  Johannes van der Horst
--
library IEEE;
use IEEE.Std_Logic_1164.all;

entity TRV_BUFT is
port (
   TR0 : in std_logic;
   TR1 : in std_logic;
   TR2 : in std_logic;
   V : out std_logic);
end TRV_BUFT;

Architecture RTL of TRV_BUFT is

component BUFT
port (  I : in std_logic;
        T : in std_logic;
        O : out std_logic);
end component;

component PULLUP
port (  O : out std_logic);
end component;

Begin


BUFT0: BUFT
port map (
   I => TR0,
```

```
    T => TR2,
    O => V);

BUFT1: BUFT
port map (
    I => TR1,
    T => TR0,
    O => V);

BUFT2: BUFT
port map (
    I => TR2,
    T => TR1,
    O => V);

PLLP: PULLUP
port map (O => V);

end RTL;
```

### voter.vhd

```
--! An array of triple redundant voters.
--!
--! Instantiates an array of n TRV_LUTs or TRV_BUFTs, depending on the
--! architecture specified, as provided in Xilinx xapp197
--!
--! TRV_LUTs are faster, but consume more logical resources in the FPGA.
--!
--! TRV_BUFTs are recommended for systems with constraints on the available
--! logic resources, at a slight speed penalty.
--!
--! Johannes van der Horst
--! jvdh@sun.ac.za
--! 2005-7-26

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.Std_logic_arith.all;
library unisim;
use unisim.vcomponents.all;


entity voter is
generic (
  n : integer := 32
  );
port (
    TR0 : in std_logic_vector;
    TR1 : in std_logic_vector;
    TR2 : in std_logic_vector;
    V : out std_logic_vector);
end voter;

architecture buft of voter is
  component TRV_BUFT
  port (
      TR0 : in std_logic;
      TR1 : in std_logic;
      TR2 : in std_logic;
```

```
    V : out std_logic);
  end component;
begin
  vector: for i in 0 to n-1 generate
    sv: TRV_BUFT
      port map(TR0(i),TR1(i),TR2(i),V(i));
  end generate;
end architecture buft;
```

## B.2 Tools

### B.2.1 CRC calculator

#### addcrc.py

```
#!/usr/bin/python
#
# Adds a CRC calculator to the in/outputs of an arbitrary design.
# Assumes that the source file is valid VHDL.
#
# Johannes van der Horst
# jvdh@sun.ac.za
# 2006-03-26


import sys
import tokenize
import re
import math
import time

crcwidth = 255

newentity = "--\n--␣Generated␣by␣addcrc.py␣on␣"+str(time.ctime())+"\n--␣\n--
␣␣␣␣Author:␣Johannes␣van␣der␣Horst\n--
␣␣␣␣Email:␣jvdh@sun.ac.za\n--
␣␣␣␣2006-03-28\n--\n\n"


newentity += "\nlibrary␣ieee;\nuse␣ieee.std_logic_1164.all;\n\nentity␣"

newarch = "architecture␣behav␣of␣"
newsignal = "";
newcode = "begin\n\t"
dwidth = 0;
  # dictionary of signals: d[name]:(dir,width,bit assigned)
sig_dict = {};


def error(s):
  print s

def signal(i):
  global newentity
  global newarch
  global newsignal;
  global en
  global dwidth
```

```
  global sig_dict

  n = 1;

  sig = "\tsignal␣";
  while en[i-1]!=":" :
    sig += en[i];
    newentity += en[i]+"␣";
    if (en[i]!=",") and (en[i]!=":"):
      sig += "_crc"
      sig_dict[en[i]+"_crc"] = -1
    if (en[i]==","):
      n+=1
    sig += "␣";
    i += 1

  dir = en[i]
  newentity += dir+"␣";
  i += 1

  w = 0;
  while en[i-1]!=";":
    newentity += en[i]+"␣";
    if en[i].lower()=="std_logic":
      w = 1;
      for c in sig_dict:
        if sig_dict[c]== -1:
          sig_dict[c] = (dir, 1,0);

    elif en[i].lower()=="std_logic_vector":
      w = abs(int(en[i+2])-int(en[i+4])+1)
      for c in sig_dict:
        if sig_dict[c]== -1:
          sig_dict[c] = (dir, abs(int(en[i+2])-int(en[i+4])+1),0)

    sig +=  en[i] + "␣";
    i+=1;
  if dir!="in":
    newsignal += sig+"\n";
    dwidth += w*n;

  if en[i]!="end":
    newentity += "\n\t\t";
  return i;
#####################################################
## Main



if (len(sys.argv)<3):
  print "CRC␣wrapper␣file␣generator"
  print "Usage:␣"
  print "./addcrc.py␣<file>␣<clk>␣<max_cycle>␣:␣Generate␣CRC␣calculator␣for␣<file>
␣␣␣␣␣␣using␣<clk>␣as␣main␣clock,␣run␣for␣<max_cycle>␣clock␣cycles"
else:
  clkname = sys.argv[2]
  max_cycle = sys.argv[3]
  int(max_cycle)
```

```
src = open(sys.argv[1]).read();


start = src.lower().find("entity");
stop = src.lower().find("architecture");
#print "slice",start,":", stop
en = filter(None, re.split("--.*\r\n|[\r\n\t␣]|(;)|([_a-zA-Z0-9]*)",src[start:stop]))
#print en

  #entity(0);
i = 0;
if en[i].lower() != "entity":
  error("Entity␣expected");
i +=1;
  # name
name = en[i];
newentity += en[i]+"_crc␣is\n\tport␣(\n\t\t"
newarch += en[i]+"_crc␣is␣\n\t\t--␣orignal␣component\n\tcomponent␣"
for j in range(1,len(en)-3):
  newarch += en[j]+"␣"
  if ((en[j]==';') and (en[j+1].lower()!="end")) or ((en[j-1].lower()=="port")):
    newarch += "\n\t\t"
newarch += "\n\tend␣component;\n\n\t\t"
while en[i]!='(':
  i+=1;   # is Port lbrac
i+=1;
  # read all signals
newentity += "rdnwr_crc0:␣in␣std_logic;\n
␣␣␣␣\t\t␣rclk_crc:␣in␣std_logic;\n
␣␣␣␣\t\t␣crc_dout:␣out␣std_logic_vector(1␣downto␣0);\n
␣␣␣␣\t\t␣nstop␣:␣out␣std_logic;\n\t\t";
while i < len(en)-4:
  i = signal(i)
newentity += "\nend␣"+name+"_crc;\n"

newsignal += "\tsignal␣din_crc␣:␣std_logic_vector(255␣downto␣0);\n
␣␣␣␣\tsignal␣mclk_new␣:␣std_logic;\n
␣␣␣␣\tsignal␣crc_nstop␣:␣std_logic;\n";



  # component instatiation
newcode += "\n\t\t--␣Original␣file\n\tcomp0␣:␣"+name+"␣port␣map("
comp0 = filter(None, re.split("--.*\r\n|:.*;|,|[\r\n\t␣]|([_a-zA-Z0-9]*)",
    src[start:stop]))[5:-3]
for i in comp0[:-1]:
  (sd, sw, su) =  sig_dict[i+"_crc"];
  if (sd != "in"):
    newcode +=i+"_crc,"
  elif (i == clkname):
    newcode +=i+"_new,"
  else:
    newcode+= i+","

newcode+= comp0[-1]+");\n"

  # crc calculator
newarch += "\n\t\t--␣CRC␣calculator\n
␣␣␣␣\tcomponent␣crc16_"+str(crcwidth)+"␣is\n
␣␣␣␣\tgeneric␣(\n\t\tmax_cycle␣:␣integer␣:=␣100000\n
```

```
␣␣␣␣\t);\n
␣␣␣␣\tport␣(\n
␣␣␣␣\t\tclk:␣in␣std_logic;\
␣␣␣␣n\t\trclk:in␣std_logic;\n
␣␣␣␣\t\trdnwr:in␣std_logic;\n
␣␣␣␣\t\tdin:in␣std_logic_vector("+str(crcwidth)+"␣downto␣0);\n
␣␣␣␣\t\tdout␣:␣out␣std_logic_vector(1␣downto␣0);\n
␣␣␣␣\t\tnstop:␣out␣std_logic);\n
␣␣␣␣\tend␣component;\n";

print sig_dict
sigiter = sig_dict.iterkeys();
t = 0;
sd = "in";
sw = 0;
su = 0;
sn = "mclk_crc";
done = 0;
newcode += "\n\t\t--␣CRC␣calculator";
newcode += "\n\tdin_crc("+str(crcwidth)+"␣downto␣"+str(crcwidth-dwidth+1)+")␣<=␣";

print math.ceil(float(dwidth)/crcwidth)
for n in range(0,round(math.ceil(float(dwidth)/crcwidth))):
  if n == 1:
    print "\t--␣WARNING:␣rdnwr_crc␣ports␣need␣to␣be␣added␣manually\n"
    newcode += "\n\t\t--␣WARNING:␣rdnwr_crc␣ports␣need␣to␣be␣added␣manually:"
  print n
  print sn ,":",sig_dict[sn]
  used = 0;
  while ((used < crcwidth) and (done==0)) :
    if (sw-su) == 0:
      while ((sd == "in") or (sw-su == 0)) and (done==0):
        try:
          sn = sigiter.next()
        except StopIteration:
          done = 1
        [sd,sw,su] = sig_dict[sn]

    if (used > 0) and (sw-su>0):
      newcode += "␣&\n\t\t"
    if (sw-su>0) and (sw-su <= (crcwidth-used)):
      newcode += sn
      if (sw-su>0) and (sw>1):     #part of bus has already been used
        newcode += "("+str(sw-su-1)+"␣downto␣0)"
      used += sw-su
      su = sw

    elif sw-su > crcwidth-used:
      print "sw-su␣>="
      newcode += sn+"("+str(sw-su-1)+"␣downto␣"+str(sw-su-(crcwidth-used))+")"
      su = crcwidth - used;
      used += crcwidth - used

    sig_dict.pop(sn)
    sig_dict[sn] = (sd, sw, su)

newcode += ";\n"

if done!=0:   # din pins unconnected
  newcode += "\n\tdin_crc("+str(crcwidth - dwidth)+"␣downto␣0)␣<=␣(others␣=>␣'0');\n"
```

```
newcode += "\n\tcrc_"+str(n)+":␣crc16_"+str(crcwidth)+"
␣␣␣␣generic␣map␣("+max_cycle+")␣port␣map("+clkname+"_new,rclk_crc,rdnwr_crc"
    +str(n)+",din_crc,crc_dout,crc_nstop);"
newcode += "\n\n\t\t--␣Connect␣output␣signals␣to␣inputs␣of␣crc␣calculator\n"

for sn in sig_dict.keys():
  (sd,sw,su) = sig_dict[sn];
  if sd!="in":
    newcode += "\t"+sn[0:len(sn)-4]+"␣<=␣"+sn+";\n"
  print sn,":",sig_dict[sn]

newcode += "\n\tnstop␣<=␣crc_nstop;\n\tmclk_new␣<=␣mclk␣and␣crc_nstop;\n";
newcode += "\nend␣behav;\n\n";
  # save result

fout = open(name+"_crc.vhd",'w')
fout.write(newentity+"\n")
fout.write(newarch+"\n")
fout.write(newsignal+"\n")
fout.write(newcode);
print dwidth
print sig_dict
```

### B.2.2 Hamming memory generator

#### hamming_mem.py

```
#!/usr/bin/python2.3
#
# Converts hex output of picoblaze assembler to
# (23,18) hamming encoded equivalents.
#
# Requires numpy, string

import time
import string
from numpy import *

  # encoding matrix
He = array([
    [1,1,0,1,1,0,1,0,1,0,1,1,0,1,0,1,0,1],  #p1
    [1,0,1,1,0,1,1,0,0,1,1,0,1,1,0,0,1,1],  #p2
    [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,1,1,1,0,0,0,1,1,1,1,0,0,0,1,1,1,1],    #p3
    [0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,0,0],    #p4
    [0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1],    #p5
    [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0],
```

```
    [0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1]])


def word2arr(c):
  s = [];
  for i in range (0,18):
    s.append((c>>i)%2)
  return array(s)

def arr2data(a):
    # lower 16 bits
  l = 0;
    # middle bits (stored in parity fields of bram)
  p = 0;
    # upper 5 bits
  u = 0;
  for i in range (0,16):
    l = (l<<1)+a[15-i]
  for i in range (0, 2):
    p = (p<<1)+a[17-i]
  for i in range (0, 5):
    u = (u<<1)+a[22-i]

# print int(l) + int(p<<16) + int(u<<18)

  return (u, p, l)

  # encodes a data word,
def encode(i):
  return arr2data(matrixmultiply(He,word2arr(i).transpose())%2)
  #return arr2data(concatenate([word2arr(i),[0,0,0,0,0]]))

def main():
  init_val = dict();
  src = open("BIST.HEX",'r').readlines()

##for i in range(0,56):
##  print encode(i);
##  b = concatenate([word2arr(i),[0,0,0,0,0]])
##  print b
##  print arr2data(b)
##
##print "----------------"
##
##for i in range(0xffff,0x1001f):
##  print encode(i);
##  b = concatenate([word2arr(i),[0,0,0,0,0]])
##  print b
##  print arr2data(b)
##
##print "----------------"

##for i in range(0x2ffff,0x3001f):
##  print encode(i);
##  b = concatenate([word2arr(i),[0,0,0,0,1]])
##  print b
```

```
##  print arr2data(b)

  par = [];
    # generate hex with parity
  for i in src:
#    print i, "%x"%(int(i,16) / 0x10000)
    par.append(encode(int(i,16)))
#    par.append((0x10,(int(i,16)/0x10000),int(i,16)%0x10000))



  cb = 0;
  cp = 0;
  cu = 0;

  init = 0;
  init_u = 0;
  init_p = 0;

  b = ""
  p = ""
  p0 = 0;
  u = ""

  for (uppr,par,lwr) in par:
#    print "     %x"%par,par<<((cp%2)*2), p0

      #1024x5 contents
    u = hex(uppr)[2:].zfill(2) + u
    cu+=1
    if cu >= 32:
      l = "INITU_%s"%(hex(init_u)[2:].zfill(2))
      init_val[l] = u
#      print l, u
      cu = 0;
      u = "";
      init_u += 1;


      # parity contents
    if (cp%2)==0:
      p0 = par
    else :
      p0 = (par<<2)+p0

    if cp%2 != 0:
      p = hex(p0)[2:].zfill(1) + p
#      print hex(p0)[2:].zfill(1)
      p0 = 0;

    cp+=1
    if cp >= 128:
      l = "INITP_%s"%(hex(init_p)[2:].zfill(2))
      init_val[l] = p
        #parity bits of upper bram
#      l = "INITPU_%s"%(hex(init_p)[2:].zfill(2))
#      init_val[l] = "0000000000000000000000000000000000000000000000000000000000000000"
      init_p+=1
      cp = 0
      p0 = 0
```

```
      p = ""

      # 1024x18 contents
    b = hex(lwr)[2:].zfill(4) + b
    cb+=1
    if cb >= 16:
      l = "INIT_%s"%(hex(init)[2:].zfill(2))
      init_val[l] = b
      #print l,b
#     bram.append("INIT_%s => X\"%s\",\n"%(hex(init)[2:].zfill(2),b))
      cb = 0;
      b = "";
      init += 1;



  return init_val

#------------------------------
#print He


repl = main()

repl["timestamp"] = time.asctime()

out = open("out.vhd",'w')
hdl = open("ROM_form.vhd",'r').read()


template =  hdl[hdl.find("{begin template}")+17:]

for s in repl.iterkeys():
  template = string.replace(template,"{"+s.upper()+"}",repl[s])

out.write(template)
out.close()
```

### B.2.3 EDIF-level flip-flop insertion

#### tmrff.py

```
#!/usr/bin/python
#
# Replaces flips flops with tmr flip flops
# 1) Insert TMR_FF library into EDF file
# 2) Replace known ff references with tmr instances
# 3) Update VIRTEX
# 4) Update UNILIB with other components in TMF_FF library
#
# Usage: python tmrff.py <input.edf> <output.edf>
#
# uses virtex.py and unilib.py
#
# Author: Johannes van der Horst
# jvdh@sun.ac.za
# 2006-05-27

import sys
```

```
import re

from virtex import virtex_dict
from unilib import unilib_dict

# regexp strings
re_LB = re.compile("\(",re.MULTILINE)
re_RB = re.compile("\)",re.MULTILINE)

def matchingParenthesis(str, start):
  str += "()" # something to keep the iteraters happy
  n = 1;
  lbiter = re_LB.finditer(str[start:]);
  rbiter = re_RB.finditer(str[start:]);
  rb = rbiter.next()
  lb = lbiter.next()
  while n!=0:
    if (lb.start() < rb.start()) :
      n += 1
      lb = lbiter.next()
    else:
      rb = rbiter.next()
      n -= 1
  return start+lb.start()

def splitLibs(src):
    #library virtex
  start = re.compile("\(library␣virtex",re.IGNORECASE+re.MULTILINE).search(src).start()
  virtex_lib = src[start:matchingParenthesis(src,start+1)]
    #initial code
  initial = src[:start-1]
    #library unilib
  start = re.compile("\(library␣unilib",re.IGNORECASE+re.MULTILINE).search(src).start()
  end = matchingParenthesis(src,start+1)
  unilib_lib = src[start:end]

  work = src[end:]

  return initial, virtex_lib ,unilib_lib, work

# -------------  main -------------
src = open(sys.argv[1]).read()
tmrff_lib = open("tmr_ff.edf").read()

  # 1) split src into inividual libraries
head, virtex_lib, unilib_lib, work = splitLibs(src)

  # 2) replace references to known flip flops with new ones
for comp in unilib_dict:
  old = "(viewRef␣PRIM␣(cellRef␣"+comp+"␣(libraryRef␣UNILIB)))"
  new = "(viewRef␣implement␣(cellRef␣TMR_"+comp+"␣(libraryRef␣TMR_FF)))"
  work = work.replace(old,new)

  # 3) update virtex_lib, if needed
new_virtex = virtex_lib[:len(virtex_lib)-4];
for comp in virtex_dict:
  if (virtex_lib.find(comp) == -1):
    new_virtex += virtex_dict[comp]
new_virtex += ')'
```

```
  # 4) update unilib, if needed
new_unilib = unilib_lib[:len(unilib_lib)-4];
for comp in unilib_dict:
  if (unilib_lib.find("cell␣"+comp+"␣") == -1):
    new_unilib += unilib_dict[comp]
new_unilib += ')'


f = open(sys.argv[2],"w")
f.write(head + new_virtex + new_unilib +tmrff_lib + work);
```

### B.2.4 Readback

#### progalgxc3s.cpp

```
// Extract from progalgxc3s.cpp

#include "progalgxc3s.h"

const byte ProgAlgXC3S::JPROGRAM=0x0b;
const byte ProgAlgXC3S::CFG_IN=0x05;
const byte ProgAlgXC3S::JSHUTDOWN=0x0d;
const byte ProgAlgXC3S::JSTART=0x0c;
const byte ProgAlgXC3S::BYPASS=0x3f;
const byte ProgAlgXC3S::HIGHZ=0x0a;
const byte ProgAlgXC3S::CFG_OUT=0x04;
const byte ProgAlgXC3S::EXTEST=0x00;
const byte ProgAlgXC3S::SAMPLE=0x01;

  /**
   * Read configuration memory and write to rb_data
   *
   * Johannes van der Horst
   * 2006-08-21
   */
int ProgAlgXC3S::readback(BitFile &file)
{
  long len = 1044736+64;
  unsigned char tdo[len];

  jtag->shiftIR(&JSHUTDOWN);
  io->cycleTCK(12);
  jtag->shiftIR(&CFG_IN);

  byte init[64];
  jtag->longToByteArray(0xffffffff,&init[0]);   //dummy
  jtag->longToByteArray(0x66aa9955,&init[4]);   //sync
  jtag->longToByteArray(0x8004000c,&init[8]);   // FAR
  jtag->longToByteArray(0xffffffff,&init[12]);  // lots
  jtag->longToByteArray(0x8006800c,&init[16]);  //FLR
  jtag->longToByteArray(0x2c000000,&init[20]);  //FLR (0x34 for xc3s200)
  jtag->longToByteArray(0x8001000c,&init[24]);  //CMD
  jtag->longToByteArray(0x20000000,&init[28]);  //read config data
  jtag->longToByteArray(0x00060014,&init[32]);  //FDRO read
  jtag->longToByteArray(0xac000012,&init[36]);  // word count (one frame)
  jtag->longToByteArray(0x00000000,&init[40]);
  jtag->longToByteArray(0x00000000,&init[44]);
  jtag->shiftDR(init,0,384,32,false); // Align to 32 bits.
```

```
jtag->shiftIR(&CFG_OUT);
byte hdr[4];
byte padding[1696/8];
jtag->shiftDR(0,padding,1696/8,false);


io->tapTestLogicReset();
io->setTapState(IOBase::RUN_TEST_IDLE);
jtag->shiftIR(&CFG_IN);


byte hdr2[60];
jtag->shiftIR(&CFG_IN);
jtag->longToByteArray(0xffffffff,&hdr2[0]);   //dummy
jtag->longToByteArray(0x66aa9955,&hdr2[4]);   //sync
jtag->longToByteArray(0x8006800c,&hdr2[8]);   //FLR
jtag->longToByteArray(0x2c000000,&hdr2[12]);  // 0x34 for xc3s200
jtag->longToByteArray(0x8001000c,&hdr2[16]);  // CMD
jtag->longToByteArray(0xe0000000,&hdr2[20]);  // clear CRC
jtag->longToByteArray(0x8001000c,&hdr2[24]);  // CMD
jtag->longToByteArray(0x20000000,&hdr2[28]);  // read config data
jtag->longToByteArray(0x8004000c,&hdr2[32]);  // FAR
jtag->longToByteArray(0x00000000,&hdr2[36]);  // 0
jtag->longToByteArray(0x00060014,&hdr2[40]);  // FDRO read
jtag->longToByteArray(0x11fe0012,&hdr2[44]);  //
jtag->longToByteArray(0x00000000,&hdr2[48]);  // flush

jtag->shiftDR(hdr2,0,480,32);
    // read data
jtag->shiftIR(&CFG_OUT);
jtag->shiftDR(0,tdo,len,false);

  // store tdo
char* fname = "rb_data";

FILE *fptr=fopen(fname,"wb");
if(fptr==0){
  fprintf(stderr,"Cannot open file");
  return 0;
}

  // write to file (ignore first few bytes until the testing is done)
for(int i=8; i<len/8; i++){
  byte b=file.bitRevTable[tdo[i]]; // Reverse bit order
  fwrite(&b,1,1,fptr);
}
fclose(fptr);

  // end readback and restart the device
byte tail[32];
jtag->shiftIR(&CFG_IN);
jtag->longToByteArray(0xffffffff,&tail[0]);    // dummy
jtag->longToByteArray(0x66aa9955,&tail[4]);    // sync
jtag->longToByteArray(0x8001000c,&tail[8]);    // CMD
jtag->longToByteArray(0xe0000000,&tail[12]);   // clear crc
jtag->longToByteArray(0x00000000,&tail[16]);   // flush
jtag->longToByteArray(0x00000000,&tail[20]);   // flush
jtag->longToByteArray(0x00000000,&tail[24]);   // flush
jtag->longToByteArray(0x00000000,&tail[28]);   // flush
jtag->shiftDR(tail,0,192,32);
```

```
  jtag->shiftIR(&JSTART);
  io->cycleTCK(12);

  jtag->shiftIR(&BYPASS);

  fprintf(stderr,"done\n\n");

}
```

### B.2.5 Verification

#### verify.py

```
#!/usr/bin/python
#
# Verifies readback data against ISE output files, count number of mismatching bits.
#
# The extension of the input file is stripped and replaced with .msk for
# the mask file and .rbb for the readback verification file.  An optional offset value can
# be given to correct alignment of different files
#
# Usage:
# python verify.py readback.data [offset]
#
# Author: Johannes van der Horst
# 2006-08-04
#

import sys
import time

def verify(argv):
  if (len(argv)==1):
    print "python␣verify.py␣readback.data␣rbb␣msk␣[offset]\n"
  else:
    print "verifying..."
    try:
      file  = argv[1]
      basename = (file.split("."))[0]
      offset = 0


    # if (len(argv)>2):
    #   offset = (int)(argv[4])

        # config frame length (bytes)
      fl = 212  # for XC3S200

        #reference data (generated by ISE)
      #ref = open(basename+".rbb",'r').read()
      ref = open(argv[2],'r').read()
        #strip header (search for \x7F\x88 - only valid with CRC disabled?)
      hdr = ref.find("\x7F\x88")+2
  #   print hdr,fl,hdr+fl

        # rbb data starts from
      rbb = ref[hdr+fl:]
  #   print "%d"%(ord(rbb[0]))
```

```
      data_in = open(file,'r').read()[fl-offset:]
  #   print len(data_in)
      f = open("data_in",'w')
      f.write(data_in)
      f.close()

    # msk = open(basename+".msk",'r').read()
      msk = open(argv[3],'r').read()
      msk = msk[msk.find("\x7F\x88")+2:]

      f = open("msk",'w')
      f.write(msk)
      f.close()

      #print len(msk)
      #print len(rbb)
      #print "\n\n"

      for o in range(0,1):
        count = 0;
        i = 8;
        p = [];
        for c in data_in[:len(data_in)-8]:
          m = msk[i+o]
          ch = ((ord(c)) ^ ord(rbb[i])) & ~ord(m)
          if ch!=0 :
              # where in byte did this happen?
            for sh in range(0,8):
              if (ch >> sh) & 0x01 == 1:
                p.append(((i-1)*8+sh,ord(rbb[i]),(ord(c)),ord(m)));
                count +=1
          i+=1;

        return (count , p)
    except IOError:
      print "IOError"
      return (-1,-1)


verify(sys.argv)
```

# Appendix C

# Hardware schematics

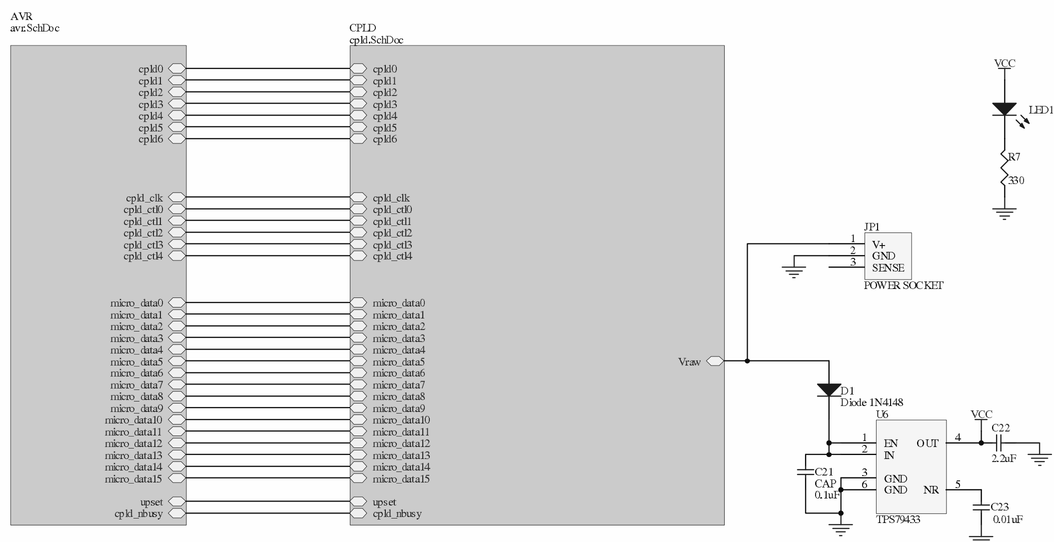



**Figure C.1:** Top level schematic of configuration controller

**Figure C.2:** Microcontroller portion of configuration controller

**Figure C.3:** CPLD and flash of configuration controller

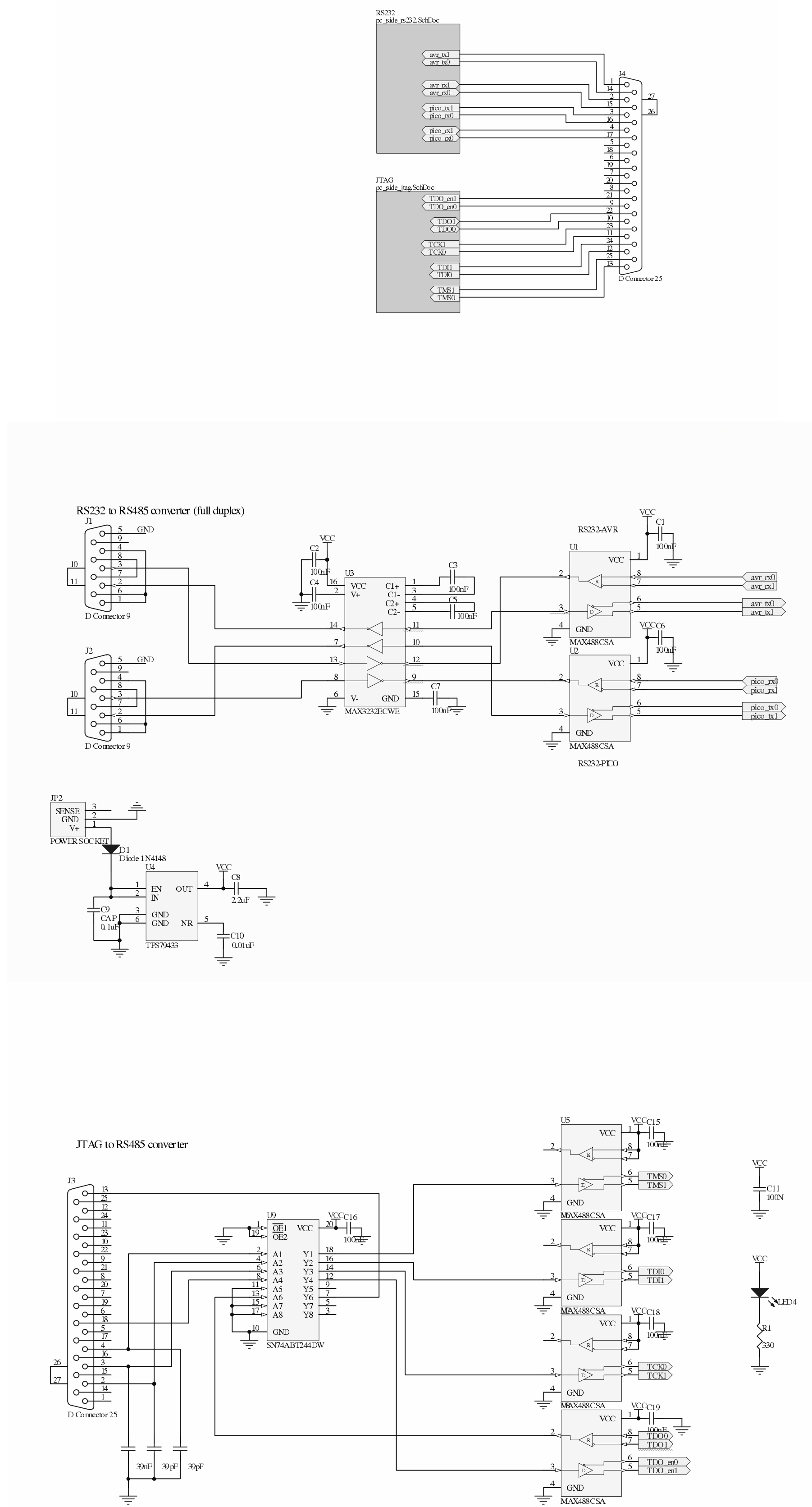



**Figure C.4:** Schematics for local transceiver.

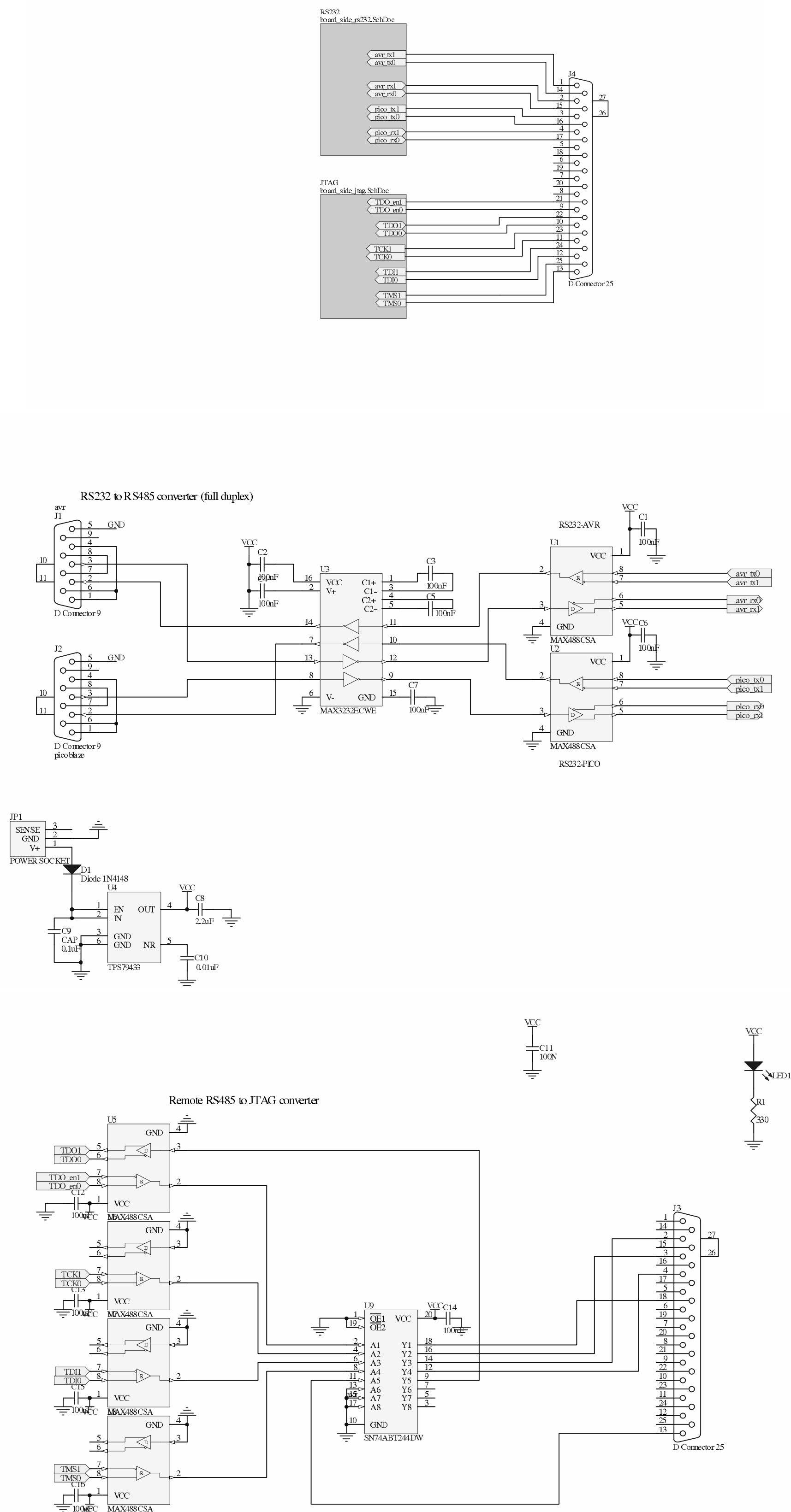



**Figure C.5:** Schematics for remote transceiver.

# Appendix D

# Enclosed CD

A CD with the source code is included with this document.

It provides the source files for the configuration controller, PicoBlaze designs, as well as the control software developed for simulation and radiation testing.

The `Protel` schematics for the developed hardware are also included.

It also contains additional photos documenting the hardware and radiation testing setup.