Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Elaboration of a Fault-Tolerance Strategy for Space-borne Digital Signal Processing Applications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Bernhard Fuchs, BSc.
Matrikelnummer 0527603

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr. techn. Andreas Steininger
Mitwirkung: Dipl.-Ing. Dr. techn. Manfred Sust - RUAG Space GmbH Austria

Wien, 19.03.2012

(Unterschrift Verfasser)

(Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Elaboration of a Fault-Tolerance Strategy for Space-borne Digital Signal Processing Applications

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Bernhard Fuchs, BSc.
Registration Number 0527603

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.-Prof. Dipl.-Ing. Dr. techn. Andreas Steininger
Assistance: Dipl.-Ing. Dr. techn. Manfred Sust - RUAG Space GmbH Austria

Vienna, 19.03.2012

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Bernhard Fuchs, BSc.
Pfeilgasse 3/13, 1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasser)

Acknowledgements

It is a pleasure to thank the many people who made this thesis possible. First of all i want to thank my advisor Ao.Univ.-Prof. Dipl.-Ing. Dr. techn. Andreas Steininger. With his enthusiasm, his inspiration, and his great efforts to explain things clearly and simply, he helped to make computer science fun for me.

Secondly, i want to thank Dipl.-Ing. Dr. techn. Manfred Sust for his excellent support in all matters during the whole time. I would have been lost without him. I also want to thank all my colleagues at RUAG Space Austria for their kind support, reviewing and guidance. This has been of great value in this study.

I wish to thank Karin Larnhof for helping me get through the difficult times, and for all the emotional support, entertainment, and caring she provided.

Lastly, and most importantly, I wish to thank my parents, Christa Fuchs and Anton Fuchs. They bore me, raised me, supported me, taught me, and loved me. To them I dedicate this thesis.

Abstract

Programmable Digital Signal Processing (DSP) is of paramount importance for the success of contemporary space missions in support of earth observation, astro-sciences and telecommunications. The present thesis aims at contributing to solving the problem of combining intrinsically contradicting requirements such as high signal processing performance, data throughput and flexibility on the one hand with robustness and high availability in the hostile space environment on the other. For economic reasons, the problem is tackled on systematic rather than technological level giving preference to pure software solutions implemented on commercial off-the-shelf (COTS) processing platforms. The applicability of commercial components to space applications is very limited due to ionizing radiation which may cause permanent modifications of the used materials and consequently of the electrical characteristics, referred to as total-ionizing-dose (TID) effects, as well as single-event effects, experienced as soft errors in form of bit-flips or detrimentally as destructive latch-ups. Consequently, spaceborne signal processors are either fast but highly optimized for particular applications, thus, inflexible or programmable, slow and based on outdated semi-conductor technologies and processor architectures. The latter is due to the fact that for a commercial component to become a real space component it must have ample heritage, be screened or even modified for satisfying space-quality assurance requirements and it must be applied long enough to justify these investments. On the other hand, modern deep-sub-micron processes are not only superior with respect to low capacitances and, thus, high processing speed but also with respect to TID and latch-up insensitivity so that they can be easier qualified for space. However, the vulnerability with respect to single-event upsets remains, resulting in intolerably low system availability.

The objectives of the present thesis have been the selection of a modern programmable DSP as well as the identification, derivation, evaluation and experimental validation of fault-tolerance (FT) mechanisms such as software-based FT, an external FT-controller as well as a combination thereof, to be applied to this component in order to establish a spaceborne DSP-system satisfying payload dependability requirements. Typical applications of such a system are data and image processing, including filtering, decimation, coding and spectral analysis. Careful selection of FT-methods as well as optimal alloying of FT- and DSP-algorithms has been shown to be crucial for maintaining the full performance of either algorithm class and for ensuring software-product maintainability. Statistical measurements performed with the most promising candidate FT-mechanisms integrated along with typical DSP-algorithms have shown that software-only solutions, although economically attractive, fail in providing the required

availability in combination with the desired processing power, while these goals can be fully met, if software FT-techniques are combined with an external FT-controller.

Kurzfassung

Programmierbare digitale Signalverarbeitung (DSP) ist von außerordentlicher Bedeutung für den Erfolg von Weltraummissionen zu Zwecken der Erdbeobachtung, astrophysikalischen Forschung und Telekommunikation. Die vorliegende Arbeit liefert wesentliche Beiträge, bestehende Fortschrittsbarrieren, bewirkt durch die angesichts der Umweltbedingungen im Weltraum einander grundlegend widersprechenden Anforderungen von hoher Prozessorleistung, gepaart mit Flexibilität einerseits und hoher Zuverlässigkeit und Verfügbarkeit andererseits, zu überwinden. Aus wirtschaftlichen Gründen wird nicht auf neue Halbleitertechnologien gesetzt sondern auf Systemebene eingegriffen, wobei vorerst reine Software-Lösungen zur Anwendung auf Standard-Prozessorplattformen der Vorzug gegeben wird. Der Gebrauch von Standard-Elektronik im Weltraum ist aufgrund der dort herrschenden ionisierenden Strahlung nur sehr eingeschränkt möglich. Neben permanenten Veränderungen der mechanischen und elektrischen Eigenschaften durch Dosis-Effekte und Bauteilzerstörung durch Latch-Ups, sind so genannte Single-Event-Effekte, die in Form spontaner Bit-Fehler wahrgenommen werden, von Bedeutung. Aus diesen Gründen steht leistungsfähige weltraumtaugliche Signalverarbeitungselektronik fast ausschließlich nur in Form von hochspezialisierten integrierten Schaltungen zur Verfügung, während programmierbare, flexible Signalprozessoren auf veralteten Technologien beruhen und daher bei weitem nicht den Anforderungen gerecht werden können. Zum Einsatz veralteter Technologien kommt es wegen des in sie bestehenden Vertrauens und der langen Dauer einer Komponentenqualifikation, sowie wegen der wirtschaftliche Notwendigkeit, einmal für den Weltraum qualifizierte Bauelemente möglichst lange einzusetzen. Allerdings haben technologische Verbesserungen zur Erhöhung der Taktraten digitaler Bausteine auch zu einer Verringerung von Dosiseffekten und Latch-Ups geführt, sodass moderne Komponenten leichter für Weltraumanwendungen qualifiziert werden könnten, wären sie nicht nach wie vor empfindlich in Bezug auf Single-Event-Effekte (SEEs).

Ziel der vorliegenden Arbeit war es, moderne Signalprozessoren in Bezug auf ihre Weltraumtauglichkeit zu untersuchen, eine potentiell geeignete Komponente auszuwählen sowie nach Software-Algorithmen zur Erhöhung der Fehlertoleranz (FT) im Zusammenhang mit Single-Event-Effekten zu forschen. Dazu wurden sowohl reine Soft- und Hardware-Lösungen wie auch auch ein Hybridkonzept theoretisch untersucht sowie Messungen an einer Hardware-Realisierung durchgeführt und evaluiert. Das Hauptanwendungsgebiet einer derartigen DSP-Plattform liegt in den Bereichen der Datenkompression und Bildverarbeitung und umfasst unter anderem Filterung, Dezimation, Kodierung und Spektralanalyse. Da es bei diesen Anwendungen um größtmöglichen Datendurchsatz geht, FT-Algorithmen aber funktionsbedingt Prozessor-Ressourcen

stark beanspruchen, war einerseits der FT-Algorithmenwahl besondere Beachtung zu schenken, andererseits mussten innovative Methoden zur Verschränkung von FT- und DSP-Algorithmen gefunden werden, um deren ursprünglich individuell optimierte Eigenschaften auch in der Kombination zu erhalten und darüber hinaus die Möglichkeit zur Weiterentwicklung und Wartung nach diesen Konzepten entstandener Flug-Software zu gewährleisten. Auf experimentellem Weg konnte gezeigt werden, dass reine Software Lösungen zwar wirtschaftlich interessant sind, aber den gestellten Anforderungen in Bezug auf Verlässlichkeit nicht gerecht werden. Demonstriert wurde aber auch, dass alle zu Beginn der Arbeit gestellten Ziele erfüllt werden können, wenn die Softwaremethoden durch externe Hardware in Form eines FT-Controllers unterstützt werden.

Contents

CHAPTER 1

Introduction

In this chapter an introduction on motivation for the topic from a political perspective will be presented. Industry has to deal with the fact that radiation-hard components originated from the United States can only be used, if their usage is a-priori permitted by US-authorities. This background lays the foundation for the problem which is present when working with radiation-tolerant hardware.

1.1 Background

Space flight plays an important role in our daily life. The first European earth observation satellite named ERS-1 was launched in 1991. Its success paved the way for future space missions, especially for earth observation. ERS-2 followed while ERS-1 was still in orbit, which even allowed for tandem operation of the two spacecraft. At their time of launch the two ERS satellites were the most sophisticated earth observation spacecraft ever developed and launched by Europe. These highly successful satellites collected a wealth of valuable data on Earth's land surfaces, oceans, and polar caps and were called upon to monitor natural disasters such as severe flooding or earthquakes in remote parts of the world.

Because of the information gained from those missions, the European Space Agency (ESA) decided to extend the Earth Observation program to investigate our planet in more detail. The overall success of a mission depends on both quality and amount of science data generated during flight and, therefore, on the total time all devices forming the instrument suite are fully operational. Consequently, space electronics is built from highly reliable components, designed with apropriate margins (de-rating), incorporating redundancy concepts to further increase reliability. Peculiar environmental conditions require careful consideration of mechanical and thermal stress experienced during launch and cruise. In this context ionizing radiation is a particular threat for space systems so that the radiation tolerance of electrical components is of paramount importance. The malfunction of such an cost expensive spacecraft could result in a huge financial loss or even the end of a mission.

Radiation induced Single Event Effects (SEEs) cause serious problems. Depending on the mission profile, radiation effects may cause transient faults like a Single Event Transients (SETs) which could be experienced as temporary “bitflips”, referred to as Single Event Upsets (SEUs), or even as permanant faults such as Single Event Latchups (SELs), which usually result in the destruction and therefore loss of the affected device. To cope with effects like these, special radiation tolerant hardware can be used. Radiation tolerant devices are equipped with additional logic to provide fault tolerance, e.g. via error detection and correction (EDAC), Triple Modular Redundancy (TMR) or improved manufacturing processes. For historical, political and technological reasons, most radiation tolerant components are fabricated in the United States and the *International Traffic in Arms Regulation*[1] (ITAR) restricts the use of radiation hardened devices outside the US. Many organizations and manufactures try to use components which are not covered by ITAR. As a result the amount of hardware components actually available to the European space engineering community is very limited.

Apparently, this problem is most pronounced for applications with demanding requirements concerning processing speed, data throughput and storage as experienced in the context of Digital Signal Processing (DSP) desperately needed for on-board data reduction to guarantee sufficient scientific return in spite of limited data-down-link capacity[2]. Possible technical solutions are

Application Specific Integrated Circuits (ASICs) Respective state-of-the-art technology is available in Europe. However, as the channel widths have decreased, ASIC manufacturing costs have increased to the extent that full custom implementations for individual missions have become unaffordable. Due to the nature of space systems engineering, ASIC re-use (or even design re-use) is rarely possible for DSP-circuitry due to the fact that requirements are highly specific for each mission and exhibit (unnecessary) differences for diferent space agencies and customers.

Programmable Processors A European component with reasonable performance for control applications is available with the LEON-FT[3] micro-processor. A state-of-the-art European DSP is not available. Respective US components are under ITAR and Japanese devices are not sold as components but only as part of systems.

Field Programmable Gate Arrays (FPGAs) A good approach which is commonly chosen is the use of reprogrammable logic devices. However, most companies providing space-grade FPGAs are located in the United States which again leads to problems with ITAR. There also exists a European company which produces space-grade FPGAs but these devices can only be used at low clock frequencies, up to 20 MHz, and they can host only very small designs.

---

[1] `http://www.pmddtc.state.gov/regulations_laws/itar_official.html`

[2] `http://spacewire.esa.int/edp-page/presentations/ADCSS09_Trautner_NGDSP%20V1.0.pdf`

[3] `http://www.esa.int/TEC/Microelectronics/SEMUD70CYTE_0.html`

A viable option seems to be a processor platform based on a Texas Instruments (TI) digital signal processor. Although TI is an american company, the available space qualified DSPs are not covered by ITAR which enables its free use. However, the radiation tolerance attained by the rad-hard fabrication technology is not perfect and therefore the former mentioned radiation effects can still affect the device in many ways.

This thesis will present a firm concept to ensure fault-tolerance on a TI-DSP based radiation tolerant platform under the given constraints and demonstrate its capabilities by means of a running prototype system which is capable of processing a high amount of data with high performance, using optimized algorithms and fault olerance mechanisms.

1.2 Methodology

First of all an approach for the quantification and the rating of possible fault scenarios for the used platform will be established. This information is mainly based on a radiation report from Texas Instruments [30] and mission experience reported in other publications. Next, state of the art hardware and software fault-tolerance approaches will be evaluated and discussed.

Although the aim is a pure software solution, approaches using additional hardware will be discussed too. Because computation performance is constrained by the DSP and by the real-time environment, a theoretical evaluation is undertaken to eliminate all infeasible solutions at an early stage.

This is followed by the establishment of fundamental requirements which need to be satisfied by the system. Based on the fact that it is clearly impossible to implement a software solution without a cutback in performance, deductions which are necessary to ensure a given tolerance level will be shown. To give an insight on performance and fault-tolerance, all evaluations are done on typical signal processing algorithms, typical in the sense that they can be found in most spaceborne data processing applications. The assessment of actual performance and availability has been done via measurements on a physical prototype. To simulate SEU effects a fault injection tool has been implemented and the results gained from fault-injection experiments can be compared with theoretical predictions and simulations.

1.3 Structure of the Thesis

This thesis is structured into seven chapters. Following this introduction, Chapter two will recapitulate the basics of failure models which apply to space-grade components and in particular to the devices investigated within the frame of this thesis work. Chapter three will give an overview on commonly used hardware and software fault tolerant approaches in combat to radiation effects in space. Followed by a brief introduction to failure modeling and a listing of likely threat scenarios and their potential consequences, Chapter four will present the development platform supporting the present investigations. The implementation of a fault tolerant system for the selected demonstrator application forms the focus of Chapter five. It will be shown that it is possible to implement the application with adequate availability. Based on the knowledge gained in Chapter five, fault injection techniques will be used to evaluate the fault distribution, their effects and their outcome. The results will be presented in Chapter six. Chapter seven concludes the thesis and discusses open questions and future enhancements.

CHAPTER 2

The Problem of Radiation in Space

"Space weather is working its way into the national consciousness as we see an increasing number of problems with parts of our technological infrastructure such as satellite failures and widespread electrical power brownouts and blackouts"[1]

2.1 Introduction

The physical space environment is markedly different from circumstances to which terrestrial electronics is exposed. In space radiation based errors like *single event effects* (SEEs) have become a major issue. However, it is the presence of ionizing radiation and its effect on semiconductors, that creates a fault environment which is unique to space applications, threatening electronic circuit reliability.

Since their first observation by Guenzer and Wolicki in 1979 [19], the importance of *soft-error* effects, their causes and the methods for their mitigation, have rapidly increased. In the same year Ziegler and Lanford published an article [68] which presents a way to predict the number of faults induced by cosmic rays. They also predicted that SEEs are not limited to memories and that a similar upset phenomena could arise even at sea level.

Concentrating on effects in space two main radiation sources need to be considered:

- High energy protons, especially for low earth orbits (LEO)
- Cosmic rays, a heavy ion compunds of either solar or galactic origin.

---

[1] The National Space Weather Program, 1999

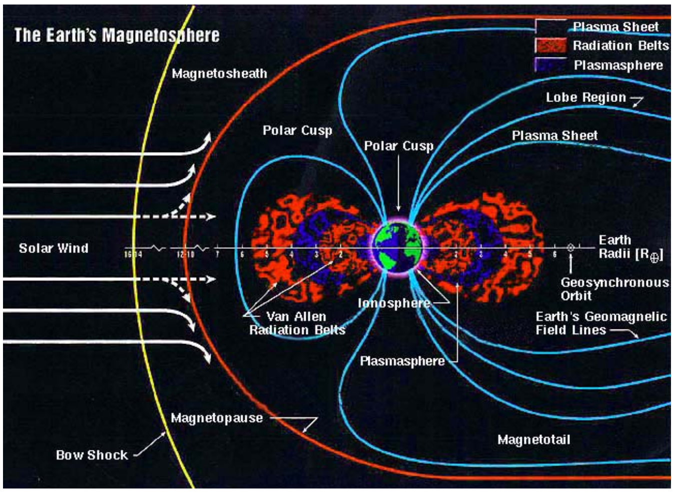


Figure 2.1: Geomagnetic field [Source: NASA Marshall Space Flight Center]

Plasma effects can be ignored in this context since plasma energies are in a range of less than 1 eV [2] to keV which is very low compared to the energy of ionizing radiation.
Due to the strong influence of the geomagnetic field, particle motion and location are also to be considered. It is known that the sun also has an impact on ionizing radiation levels and magnetic field characteristics, influenced by the eleven year solar cycle which is divided into four years of *solar min* followed by seven years of *solar max*, it is known that sun flares are a major contribution to the overall ionizing radiation level. Figure 2.1 shows particle motion and location influenced by the geomagnetic field.

The regions of interest for an earth observation mission are mainly the Van Allen Belts. As shown in Figure 2.2 the Van Allen Belts consist of the inner proton and the outer electron belt. The south atlantic anomaly (SAA), which is also shown, causes an increased flux of energetic particles at its location and therefore exposes a higher level of radiation to objects in this region. Particles trapped in the SAA as well as electrons belonging to the outer electron belt, which reaches down to the earth's surface close to the polar regions are the greatest threat for satellites in low-earth orbit (LEO). The trapped particles in the outer Van Allen Belt include electrons with an energy level of up to 7 MeV. Due to the low energy level shielding, against these particles is easy. On the other hand, there are mainly protons trapped inside the inner belt with an energy level of less than 500 MeV, which roughly varies inversely with altitude. Consequently, dose is affected by altitude and geomagnetic latitude.

Based on the information provided up to now, Table 2.1 aims at a classification of radiation according to root cause and affected orbits.

---

[2] Energy Unit: Electron Volt (eV) one eV is the energy gained by an electron by acceleration due to a potential difference of 1V. Energy in radiation is usually in the unit of MeV (10^6eV) or KeV (10^3eV). $1eV = 1.6 \cdot 10^{-19} J$, $1MeV = 1.6 \cdot 10^{-13} J$

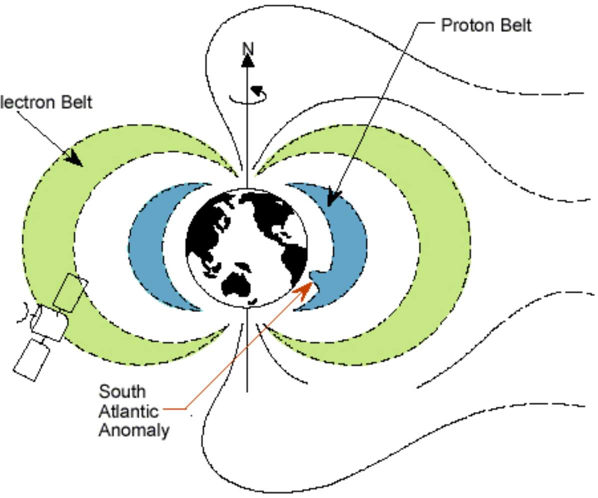


Figure 2.2: Trapped Radiation Belts Around Earth [Source: NASA Jet Propulsion Laboratory]

Radiation Source	Effects of Solar Cycle	Variations	Types of Orbits Affected
Trapped Protons	- Solar Min - Higher - Solar Max - Lower	-Geomagnetic Field - Solar Flares - Geomagnetic Storms	- LEO - HEO - Transfer Orbits
Galactic Cosmic Ray Ions	- Solar Min - Higher - Solar Max - Lower	Ionization Level	- LEO - GEO - HEO - Interplanetary
Solar Flare Protons	- Large Numbers During Solar Max - Few During Solar Min	- Distance from Sun Outside 1 AU - Orbit Attenuation - Location of Flare on Sun	- LEO $(I > 45°)$ - GEO - HEO - Interplanetary
Solar Flare Heavy Ions	- Large Numbers During Solar Max - Few During Solar Min	- Distance from Sun Outside 1 AU - Orbit Attenuation - Location of Flare on Sun	- LEO - GEO - HEO - Interplanetary

Table 2.1: Summary of Radiation Sources [Source: NASA - Single Event Effect Criticality Analysis]

2.2 Single Event Effects (SEE)

The former mentioned physical effects create the basis for the following description. If radiation exceeds a certain level their physical effects can cause electronic disturbances like SEEs, which vary depending on their energy level and their location. Practically, space radiation is described by the so called radiation spectrum, which is the density of particles of a particular radiation type as a function of particle energy.

SEEs can be described as a function of the charge deposit into a node of an integrated circuit in terms of linear energy transfer (LET) that denotes the energy imparted by the particle and is directly proportional to the response of the circuit. In other words, LET expresses the relevant characteristic of a particles passage through material. It gives up energy as a function of the distance it travels through the material and the density of the material.

$$L_{\Delta} = \frac{dE_{\Delta}}{dx} \tag{2.1}$$

LET is typically expressed in $MeV \cdot cm^2/mg$. The linear energy transfer threshold of a device – LET_{th} is by definition the minimum amount of energy required to cause an SEE in this device at a particle fluence of $10^7\ ions/cm^2$. A device having a $LET_{th} > 100 MeV \cdot cm^2/mg$ is considered practically SEE immune. Conversely a low LET_{th} implies high sensitivity.

The cross section CS, or σ, which is a function of the LET measures the probability for an SEE to occur. To define the upper limit the saturation cross section CS_{sat} or σ_{sat} is used. In other words σ is referred as the number of upsets observed divided by the number of ions per cm^2. σ_{sat} and LET_{th} are the key measures for SEEs. The following example shows how the cross section is calculated:

Cross Section Estimation Example – SEU-Testing of 8 Bit Memory

1. write 0000.0000 into memory
2. irradiate device with a known number of particles per cm^2 – (F)
3. stop irradiation
4. read out memory e.g. 0100.0010 → 2 upsets – (N)

The resulting cross section σ can be calculated using equation 2.2

$$\sigma_{SEU} = \frac{N}{F}[cm^2] \tag{2.2}$$

Because the cross section is measured for one particular LET, more experiments and calculations with different LET levels are necessary. Figure 2.3 shows the cross section as a function of LET for a particular device. Also, threshold and the saturation are visible.

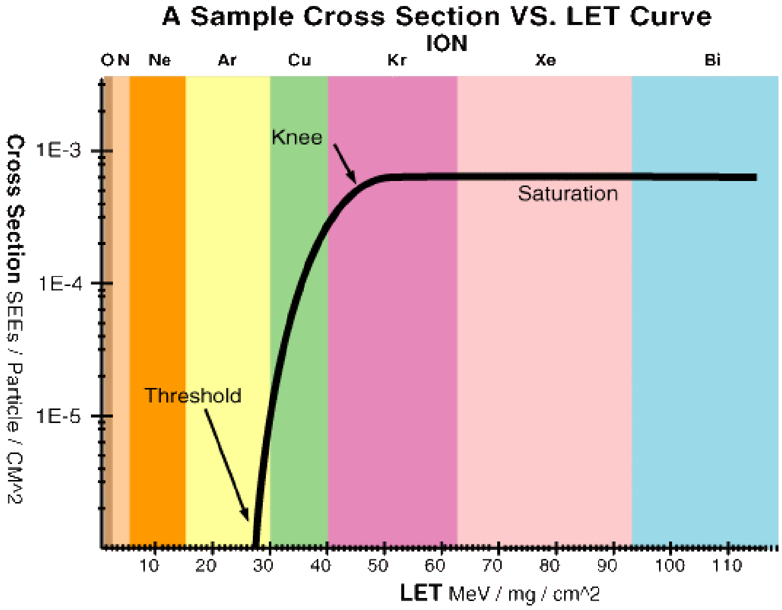


Figure 2.3: Cross section vs. LET

SEEs caused by energetic particles are divided into 2 categories:

- Permanent faults:
  - Single Event Latchup (SEL)
  - Single Event Burnout (SEB)
  - Single Event Gate Rupture (SEGR)
- Transient faults:
  - Single Event Upset (SEU)
  - Single Event Functional Interrupt (SEFI)

Since for many of those terms slightly different definitions can be found in literature, this work refers to the following space radiation definitions used by Ken LaBel, NASA Goddard Space Flight Center[3]:

- **Single Event Latchup (SEL)** is a potentially destructive condition involving parasitic circuit elements forming a silicon controlled rectifier (SCR). In traditional SEL, the current may destroy the device, if not limited and removed “in time”. A “microlatch” is a subset of SEL where the device current remains

---

[3] `http://klabs.org/richcontent/Tutorial/Radiation_Definitions.htm`

below the maximum specified for the device.
A removal of power to the device is required in all non-catastrophic SEL conditions in order to recover device operation.

- **Single Event Burnout (SEB)** is a highly localized burnout of the drain-source channel in power MOSFETs. SEB is a destructive condition.
- **Single Event Gate Rupture (SEGR)** is the burnout of a gate insulator in a power MOSFET. SEGR is a destructive condition.
- **Single Event Upset (SEU)** is a change of state or transient induced by an ionizing particle such as a cosmic ray or proton in a device. This may occur in digital, analog and optical components or may have effects in surrounding circuitry. These are "soft" bit errors in that a reset or rewriting of the device causes normal behaviour thereafter. A full SEU analysis considers the system effects of an upset.
- **Single Event Functional Interrupt (SEFI)** is a condition where the device stops operating in its normal mode, and usually requires a power reset or other special sequence to resume normal operations. It is a special case of SEU changing an internal control signal. One example would be a DRAM entering the test mode defined by JEDEC [4]. Another example is a microcircuit with IEEE 1149.1 JTAG circuitry leaving the TEST_LOGIC_RESET state and loading an unintended instruction into the instruction register (IR). Like other SEUs, the system effects must be properly analyzed. For example, a JTAG upset can cause the device to draw high currents or turn inputs into an outputs. The latter could, for example, drive a clock line to ground; thus, an independent clock signal should be used for the TCLK pin on devices without the optional TRST pin.

A complete summary of faults based on radiation can also be found in the the European standard E-ST-10-12C from the "European Cooperation for Space Standardization".[5]

2.3 Total Dose Effects

Total ionization dose (TID) is the non-reversible effect of ionizing radiation accumulated on a space mission over time. Ionizing radiation can generate electron-hole pairs in semiconductors and in insulators such as silicon dioxide. In principle, it is possible for the electrons and holes to recombine or to be transported away by an electric field. However, holes that have lower mobility than electrons are often trapped at interfaces between semiconductor and insulator or within the bulk material. The deposited charge changes the potential which can in turn cause an increase in the leakage currents which may increase power consumption, change the devices time constants, reduce gain, and change the threshold voltage of a metal-oxide semiconductor (MOS) transistor. In addition electron can get caught in non-conductive material. With increasing exposure, there

---

[4] JEDEC standards `http://www.jedec.org/`

[5] European Cooperation for Space Standardization `http://www.ecss.nl/`

is a continuing decrease in functionality until the device eventually fails. In other words, TID to an electronic device is the equivalent of a sunburn to the human skin. The measurement is done in terms of the absorbed dose which represents the absorbed energy. The unit for this measurement is *rad* (radiation absorbed dose) or the SI unit *gray* (Gy).

$$1\,Gy\ = 100\,rad = 1\ J/kg \tag{2.3}$$

Dose must always be referred to the absorbing material. TID effects on semiconductors can be summarized as follows:

- MOS transistors
  - Threshold voltage shift (ΔVt)
  - Leakage currents between source and drain, and between adjacent MOS-transistors
  - Transconductance decrease
  - Weak inversion slope decrease
- Bipolar transistors
  - Gain decrease
- JFET transistors
  - Decrease of P-JFET transconductance
  - No TID effect on N-JFET
- Silicon resistors
  - Resistance of P-Silicon resistor increases
  - No effect on N-silicon resistors
- MOS capacitors
  - no TID effect

2.4 Displacement Damage

Displacement damage is the result of non-ionizing radiation that causes atomic displacements when radiation interacts with atomic nuclei, displacing or removing them from their lattice sites. This upsets the periodicity of the lattice in the material, creating lattice defects. Displacement damage, also known as bulk damage, is caused by the cumulative effects of non-ionizing radiation that include protons and ions at all energies, electrons greater than about 150 keV, and neutrons from onboard radioactive power sources or secondary particles from the initial interactions. In short, displacement damage is the accumulation of crystal lattice defects caused by high energy radiation. The most important consequence of displacement damage in a semiconductor is a reduction in the lifetime of the minority carriers. Displacement damage is similar to

TID, which means that the effect is cumulative, although it is more complex to characterize than TID. A commonly used method to quantify displacement damage is non-ionizing energy loss (NIEL) [61].

2.5 Mission Orbits

There are extremely large variations in the level of radiation effects depending upon the trajectory through the radiation source. Satellites flying at Low Earth Orbits (LEOs), Highly Elliptical Orbits (HEOs), Geostationary Orbits (GEOs) and planetary and interplanetary missions experience very different environmental conditions [29].

At the beginning of this Chapter the susceptibility of electronic components to space radiation was explained. Also, a threshold for SEE immunity based on the energy level was derived. The following description shall help to understand how the level of radiation depends on a space vehicle's location in space. Although the presentation is qualitative, it gives a good indication for the probability of ocuurence of radiation induced effects.

2.5.1 Low Earth Orbit (LEO)

Satellites flying in LEO are passing regions of the Van-Allen Belts, filled with trapped protons and electrons, several times a day. The resulting flux varies largely depending on inclination and orbit. This is the most important characteristic of a LEO orbit with respect to radiation. The greatest inclination dependencies occur in the range of $0° < I < 30°$. The largest variation of the resulting flux is located between 200km up to 600km. At an altitude in excess of 600 km the flux only changes gradually. The location of the flux peaks depends on the energy of the particles. For trapped protons with an energy of $E > 10\text{MeV}$ it is found at about 4000km altitude.

The geomagnetic field works as a shield protecting satellites against cosmic rays and solar flare particles. This protection mechanism varies stronger with inclination than it does with altitude. Consequently, the exposure to radiation increases with increasing altitude as well as with increasing inclination. If the inclination reaches the pole regions, the spacecraft is outside the geomagnetic field lines and therefore fully exposed to cosmic rays and to solar flare particles for a significant portion of the orbit. Under normal space-weather conditions an inclination of $45°$ helps to shield the satellite completely against solar flare protons.

During strong solar events the geomagnetic field is distorted, resulting in cosmic ray and solar flare particles reaching previously unattainable altitudes and inclinations. The same effect applies to cosmic ray particles during strong magnetic storms.

2.5.2 Highly Elliptical Orbit (HEO)

Highly elliptical orbits are similar to LEO orbits, in that they pass through the Van Allen belts every day. However, because of their high apogee altitude (greater than about 30,000km), satellites in this kind of orbit also have to endure long exposure to the cosmic ray and solar flare environments, regardless of their inclination. The levels of trapped proton flux that HEOs encounter depend on the perigee position of the orbit including altitude, latitude, and longitude. If this position drifts during the course of the mission, the degree of drift must be taken into account when predicting proton flux levels [29]. Because of the orbit altitude ranging from 400 km (perigee) to 46,000 km (apogee) HEOs also accumulate high TID-levels due to both, the trapped proton exposure and the electrons in the outer belts where the spacecraft spends a significant amount of time during each apogee pass.

2.5.3 Geostationary Orbit (GEO)

At geostationary altitudes the only trapped protons that are present are below energy levels necessary to initiate nuclear events that could cause SEEs in materials surrounding the sensitive region of the device. However, GEOs are almost fully exposed to the galactic cosmic ray and solar flare particles. Protons below 40-50 MeV are normally geomagnetically attenuated, however, this attenuation breaks down during solar flare events and during geomagnetic storms. Field lines crossing the equator at about 7 earth radii during normal conditions can be compressed to about 4 earth radii during these events. As a result, particles previously deflected have access to much lower latitudes and altitudes.

2.5.4 Planetary and Interplanetary

The evaluation of the radiation environment for these missions can be extremely complex depending on the number of times the trajectory passes through the earth's radiation belts, how close the spacecraft gets to the sun, and how well known the environment of other planets is. Each of these factors must be very carefully taken into account along the exact mission trajectory.

CHAPTER 3

Hardware and Software Fault Tolerance

In 1980 a joint committee on "Fundamental Concepts and Terminology" was formed by the technical committee on Fault-Tolerant Computing of the IEEE Computer Society and the "International Federation for Information Processing" (IFIP), Working Group 10.4, "Dependable Computing and Fault Tolerance." This laid the foundation for many important actions on dependable systems.

Designing a reliable system requires finding a way to prevent failures caused by logical faults arising from various problems. Available electronic components provide little support for error detection and recovery and even less for the alleviation of strange behaviour caused by SEEs. Therefore, some kind of replication is required to attain acceptable fault-coverage.

The optimal redundant configuration for the implementation of a reliable computational component depends on the relative importance of operational correctness, life time and cost. Uninterrupted operational correctness generally requires multiple devices to be simultaneously on-line; long-life generally requires unpowered spares to be available. Both of these must be weighed against mass, power, real estate and the cost associated with the use of multiple devices. As described earlier, the primary goal of this thesis is to determine a DSP-platform architecture offering affordable protection against high transient error rates which have to be expected, if commercial or industry-standard electronic components are deployed in space.

This chapter summarizes the broad spectrum of techniques available to the designer of reliable digital systems. Techniques leading to increased reliability/availability can be divided into two groups of basic approaches.

- Fault intolerance (or fault-avoidance)
- Fault tolerance

3.1 Fault-Error-Chain

Before continuing some basic definitions are necessary. It is mandatory to understand the difference between faults, errors and failures:

- Fault: adjudged or hypothesized cause of an error
- Error: that part of state which may lead to a failure
- Failure: occurs when delivered services deviate from the specification

$$\boxed{fault \rightarrow error \rightarrow failure \rightarrow fault \rightarrow error \rightarrow failure \rightarrow \cdots}$$

In other words it can be described as follows: *An error is a manifestation of a fault in a system, which could lead to system failure*[1].

In general the occurrence of the first failure does not end in contained behaviour. Although a failure is the first deviation that can be observed from outside, it is necessary to consider that a failure can propagate onto the next level. A failure in one service may propagate into another service as a fault.

The next illustration shall increase the ability to recognize the importance of the fault-error-chain.

$$\boxed{\cdots \rightarrow event \rightarrow cause \rightarrow state \rightarrow event \rightarrow cause \rightarrow state \rightarrow \cdots}$$

The goal is to break this fault-error-chain and thereby increase the dependability of a system. The development of a dependable computing system calls for the combined utilization of a set of methods and techniques which can be classed in the following manner [31]:

- **Fault prevention:** how to prevent fault occurrence or introduction
- **Fault tolerance:** how to ensure a service up to fulfilling the system's function in the presence of faults
- **Fault removal:** how to reduce the presence (number, seriousness) of faults
- **Fault forecasting:** how to estimate the present number, the future incidence, and the consequences of faults

Depending on system and environment, not all of the above techniques may be applicable.

---

[1] Singhal/Shivaratri

3.2 Basics of Fault Tolerance

Fault intolerance results from conservative design practices such as the use of highly reliable components. The goal of fault intolerance is to reduce the possibility of a fault to occur in the first place. Techniques in this category are aimed at defining methodologies and standards that control the development process and prevent the introduction of faults. Some well known methodologies we would like to mention here as examples are shielding and the application of quality standards such as ISO9000[2], ECSS[3], DO178B[4], etc. are included here. However, even with the most careful avoidance techniques faults cannot be completely avoided and system failures will occasionally occur.

Fault Tolerance aims at with putting mechanisms in place allowing a system to still deliver the required service in the presence of faults, albeit some (graceful) degradation may have to be taken into account. Additional information provided and exploitet with this technique allows for interrupting the error chain at the transition from fault to error.

The necessary redundancy may be provided in two ways: time and space. Spatial redundancy, always requires additional hardware, viz. the addition of extra gates and may introduce additional information in form of coding. Hardware deals which the addition of extra gates, memory cells, bus lines, functional units and also the supply of extra information (e.g. coding) to guard against failures.

Time redundancy may be exclusively implemented in software and entails multiple or repeated execution of the same instruction and comparing the results in a straightforward or more sophisticated manner. Table 3.1 summarizes some techniques presented in [59].

The fundamental theory of reliable system design states that a system may go through as many as eight states in response to the occurrence of a fault [59]. Designing a reliable system involves a selection of a coordinated fault response that combines several reliability techniques. In the sequel, the following definitions will be used.

- **Fault confinement:** Limiting the spread of fault effects to one area of the system, thereby preventing contamination of other areas. Can be achieved through liberal use of fault-detection circuits, consistency checks before processing ("mutual suspicion"), multiple requests/confirmations. May be implemented in both hardware and software.
- **Fault/Error detection:** Recognizing unexpected behaviour in the system. Many techniques are available to detect faults, but an arbitrary period of time, called *error detection latency*, may pass before detection. Error detection techniques are divided into two major classes: off-line detection and on-line detection. During off-line detection the system is not able to perform useful operations until the detection process has finished. A

---

[2] International Organization for Standardization `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=42180`

[3] European Cooperation for Space Standardization `http://www.ecss.nl/`

[4] Radio Technical Commission for Aeronautics `http://www.rtca.org/`

typical example is the execution during an idle-phase. Thus, off-line detection assumes integrity before and possibly at intervals during operation, but not during the entire time of operation. On-line detection provides a real-time detection capability that is performed concurrent with application code.

- **Diagnosis:** This stage is necessary to gain information about the location and/or properties of errors. The information is sometimes provided already by the error detection mechanism.
- **Reconfiguration:** In case a fault is detected and a permanent fault identified, the system may be able to reconfigure its components either to replace the failed component or to isolate it from the rest of the system. Alternatively, it may simply be switched off and the system capability degraded in a process called *graceful degradation*.
- **Recovery:** Necessary to eliminate effects of faults. Two basic approaches of recovery are based on the techniques of fault masking and retry. *Fault masking* techniques hide the effects of faults by allowing redundant information to outweigh the incorrect information. In *retry* a second attempt of an operation is made and may well be successful because many faults are transient in nature, not causing any physical damage. One form of retry, often called *rollback*, makes use of the fact that the system operation is backed up to some point in its processing prior to fault detection and that operation can recommence from this point. Error detection latency becomes an important issue because the rollback must go far enough back to surpass all effects of undetected errors that may have occurred before the detected one.
- **Restart:** Occurs after the recovery of undamaged information. A “hot” restart, which is a resumption of all operations from the point of fault detection, is possible only if no damage has occurred. A “warm” restart implies that some of the processes can be resumed without loss. A “cold” restart corresponds to a complete reload of the system, without any processes surviving.
- **Repair:** A component diagnosed as having failed is replaced. As with detection, repair can be either on-line or off-line. In case of off-line repair the system must be shut down to perform the repair. In case of on-line repair the component my be replaced immediately by a backup spare in a procedure equivalent to reconfiguration. Alternatively operation may continue without the component, as in the case with masking redundancy or graceful degradation. In either case the failed component my be physically replaced or repaired without interrupting system operation.
- **Reintegration:** The repaired module must be reintegrated into the system. For on-line repair reintegration must be accomplished without interrupting system operation

Hardware Techniques		Software Techniques	
Class	Technique	Class	Technique
Fault avoidance	Environment modification Quality changes Component integration level	Fault avoidance (software engineering)	Modularity Object-oriented programming Capability-based programming Formal proofs
Fault detection	Duplication M-of-N codes, parity, checksums arithmetic codes cyclic codes self-checking and fail-safe logic Watch-dog timers and timeouts Consistency and capability checks Processor monitoring	Fault detection	Program monitoring
Masking redundancy	NMR/voting Error correcting codes Hamming SEC/DED, other codes Masking logic Interwoven logic, coded-state machines	Masking redundancy	Algorithm construction Diverse programming
Dynamic redundancy	Reconfigurable duplication Reconfigurable NMR Backup sparing Graceful degradation Reconfiguration Recovery	Dynamic redundancy	Forward error recovery Backward error recovery Retry, Checkpointing Journaling, Recovery blocks

Table 3.1: Classification of fault-tolerance improvement techniques [59]

3.3 Research Objectives and Success Metrics

Before considering particular fault tolerance approaches or evaluating their performance, the research objective and success criteria shall be defined.

Research Objectives:

- to determine fault tolerance techniques which are broadly applicable to the use of DSPs in space (especially the DSP presented later)
- to experimentally validate these techniques
- to evaluate the effectiveness of these techniques

The success metrics will be written as requirements since it is common in space business to specify the outcome by means of requirements.

Success Metrics:

- Candidate techniques shall be broadly applicable to existing DSPs for space applications.
- Techniques shall exhibit minimal intrusiveness with respect to single devices implementations
- System availability shall be at the level of typical contemporary (2011) space mission requirements.
- System resource utilization (mass, power, envelope, cost) shall be substantially less than for contemporary solutions achieving the same level of dependability.

3.4 Hardware Fault-Tolerance

The physical replication of hardware is perhaps the most common form of redundancy used. As semiconductor components have become smaller and less expensive, the concept of hardware redundancy has become more common and more practical. For terrestrial applications the cost associated with the replication of hardware within a system is decreasing, simply because the cost of hardware is decreasing. Hardware replication is also a convenient approach for the application developer. It allows concentrating on the software development processes and, thus, decreasing development costs. However, there is a price to pay in terms of additional hardware area, power consumption and speed, which is critical, especially when considering space applications. The latter may benefit from the aforementioned theory of ever falling hardware prices only, if commercial or at least industrial components can be used instead of space grade devices. One can distinguish between three basic forms of hardware redundancy, *passive*, *active* and *hybrid* redundancy.

The first two are often referred to as *static* and *dynamic* in literature. Passive redundancy uses the concept of fault-masking to hide the occurrence of errors and, therefore, prevents their manifestation. The passive approach is designed to achieve fault-tolerance without requiring any additional action. The most common form of passive redundancy is called *triple modular redundancy* (TMR) which uses three components in parallel. The fault masking is done via majority voting. Clearly, the single point of failure here is the voter. A more generalized approach of TMR is *N-modular redundancy* or NMR were N components work concurrently.

Active or dynamic redundancy achieves fault-tolerance by detecting the existence of faults and performing some action to isolate the effect. This can be described as

detection → isolation → reconfiguration or recovery

The last form is the hybrid approach. It combines the attractive features of both passive and active redundancy to prevent erroneous results from being generated. Hybrid approaches are often used in critical-computation applications were fault-masking and high reliability are required. Hardware fault-tolerance cannot only be achieved by just using additional hardware. *Information redundancy* allows fault-detection, fault-masking or even fault-recovery by adding redundant information to the data word. The most common approach is using error detection and error correction coding (EDAC).

3.4.1 Radiation-Hard Components

Space systems are more or less permanently exposed to radiation which causes a reduction of the lifetime of electronic components. Protection against radiation effects must be enforced by design. Main sources for radiation were already mentioned in chapter two. The primary goal of radiation hardening techniques is to increase the capability to handle specific radiation levels. This can be split in two main categories, *logical* and *physical* techniques:

Physical techniques deal with customizing layout and manufacturing processes. Silicon on insulator (SOI) is such a manufacturing technique. It adds a very thin insulating layer to the device which limits the sensitivity with respect to parasitic current. Because of this, layer capacity and switching time of transistors can be decreased at the cost of increased complexity of the manufacturing process.

A straightforward and inexpensive method for increasing radiation tolerance is shielding. However, depending on the thickness of the used shielding, Bremsstrahlung will result out of the deceleration of charged particles. In Chapter two the relation between material and LET was already shown. Therefore, shielding is effective against TID effects but may even worsen the situation with respect to SEEs.

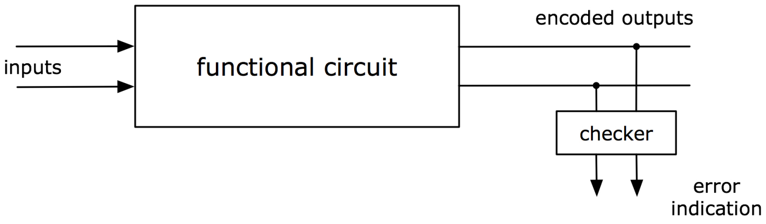


Figure 3.1: General structure of self-checking circuits

Other techniques which are often referred as *Radiation Hardening by Design* (RHBD) can be spilt in those dealing with error masking and those concentrating on error detection and correction. The latter will be treated in Section 3.4.3.3. Within the semiconductor design flow RHDB may be accomodated on chip-layout level or on transistor level:

- *RHBD on layout level*: The basic idea is to increase the minimal charge needed for SEUs to occur, which is referred to as critical charge (Q_{crit}). This can be achieved by increasing capacitance in sensitive nodes, typically by increasing the physical size of transistors. The bigger the capacitance the higher the resistance against SEUs. Increasing transistor size increases power consumption and used chip area [15]. In [2] and [60], edgeless transistors, referred to as Enclosed Layout Transistors (ELT), are used to eliminate radiation-induced leakage currents (between source and drain). As proposed in [37], [2], [60] to introduction of guard bands [3] around the devices is used to reduce the possibility of the latch-up probability.
- *RHBD on Transistor level*: The former mentioned techniques are based on spatial and temporal redundancy. On transistor level SEU tolerance can be improved by replicating state-holding nodes. This method is used in Heavy Ion Tolerant (HIT) cells [4] and in Dual Interlock Cells (DICE) [6].

3.4.2 Self-Checking Designs

Concurrent checking aims at verifying circuits during their normal operation. Self checking designs are implemented using concurrent error detection by means of hardware redundancy. A complex circuit is partitioned into functional blocks and block is implemented according to the structure shown in Figure 3.1 [40].

The most obvious way to achieve fault tolerance is the duplication of functional blocks. This method, referred to as duplication and comparison, benefits from the low engineering overhead required for simple replication. However, for the ultimate achievement of simple error detection the costs for this solution are more than 100% higher than for the unprotected design.

- Parity Code
- Dual-Rail Code
- Unordered Code

- Arithmetic Code

The parity code is the cheapest because it only adds one check bit to the information part. It can detect single errors (per information block) and even multiple errors, if the number of errors (per information block) is an odd number. A dual-rail code is a variety of duplication since check bits are the complements of the information bits. Dual-rail can detect any number of errors exclusively affecting the information part or the complement part.

By definition, codes are unordered, if the case that a codeword x covers another codeword z, denoted as $x > z$, meaning that x has a 1-bit at least in the 1-bit positions of z, does not occur. This property allows the code to detect multiple errors with some constrains. Errors affecting a single code-word must be unidirectional which means that only either ($0 \rightarrow 1$)- or ($1 \rightarrow 0$)-errors are allowed. The most important unordered codes are the m-out-of-n codes and the Berger codes. m-out-of-n code are non-separable codes which means that information and check bits are merged in a single code-word. Code-words are composed by generating patterns with exactly m 1-bits. For example, 1100, 1010 and 1001 are valid code-words for a 2-out-of-4-code.

Berger codes are separable unordered codes. The check part represents the number of 0 bits of the information part. For the information part *I* = 110011 coding would create check part *C* = 010. For n information bits the number of check bits is equal to $\lceil log2(n+1) \rceil$. Both m-out-of-n codes and Berger codes are arithmetic codes. They are interesting for checking arithmetic functions because they are preserved under such operations. The most commonly used arithmetic codes are separable. They are most often implemented as so called low cost arithmetic codes [49].

In [36] an improved approach for soft-error coverage using data-path parity was presented. The presented approach uses the "duplicate and compare" method on registers and logical elements connected directly with memory elements. Regions where soft-error effects can cause only little mutation to output registers were protected using coding techniques for cost reasons. Figure 3.2 shows an approach for a self-checking circuit based on partial duplication. The function of this approach can be summarized as follow: For intermediate results *IR* check bits $gen($*IR*$)$ are generated. In parallel, the expected check-word *pred(IR)* for the corresponding intermediate result is calculated. It depends on the operand's control *code(x)* and *code(y)* and the operand based correction information from the arithmetic unit.
The technique allows for the generation of self-checking arithmetic units with hardware overheads between 3% and 16%. Compared to TMR this is much lower, however, the technique is restricted to the detection of errors in registers and only used with distinct units rather than the complete designs.

3.4.3 Hardware EDAC

Error Detection and Correction (EDAC) [7] coding introduces information redundancy to allow for error correction in addition to pure error detection. The available methods can widely be

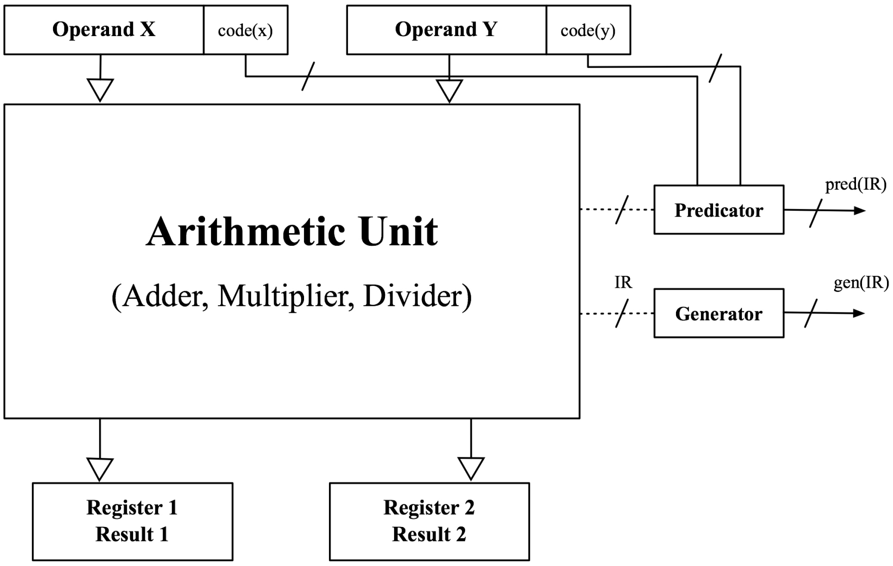


Figure 3.2: Self-checking through partial duplication [36]

used to improve the reliability of storage structures, like finite state machines (FSMs) as well as static and dynamic memories (SRAMs, DRAMs). Traditionally, a hamming code $(39, 32)$ is used for the protection of 32-bit data. In the theory of error control the designation (n, k) denotes a block code that takes a k-bit data word and maps it to an n-bit code word. The most popular coding techniques are introduced in the following section.

3.4.3.1 Parity Checking

The simplest but also weakest coding technique is parity checking by adding a single check-bit. Apparently, it requires a low level of redundancy to ensure detection of single bit errors. Due to the limited amount of redundant information, correction of false data is not possible. To generate the parity information a simple XOR operation can be used. The same simple operation can also be used to determine if an error has occurred. This makes the parity checking an attractive solution if only single bit error detection is required. However, it is not powerful enough to handle SEUs. To make it more attractive a modified version like hamming coding or rectangular coding can be used. Parity checking is not obsolete since it is commonly used in the hamming code $(39, 32)$ to generate the overall parity information (7th redundancy bit).

3.4.3.2 Rectangular Codes

Rectangular codes are used for the protection of larger amounts of information and if the data representation is in matrix form. An extra column and an additional row containing row and column parity information are inserted. To enable detection and correction the row-column parity intersection is used, which is demonstrated graphically in Table 3.2 to outline the basic idea

Block of data (32 bit)

D0	D1	D2	D3	D4	D5	D6	D7	PR0
D8	D9	D10	D11	D12	D13	D14	D15	PR1
D16	D17	D18	D19	D20	D21	*D22*	D23	PR2
D24	D25	D26	D27	D28	D29	D30	D31	PR3
PC0	PC1	PC2	PC3	PC4	PC5	PC6	PC7	P

Vertical parity

Horizontal parity

Table 3.2: Basic idea of rectangular codes

behind rectangular codes.

As shown in Table 3.2 the flip of a single bit (D22) will affect two parity bits from which the coordinates for locating the erroneous bit can be unambiguously derived. Triple errors can neither be detected nor corrected. In fact they can mimic a single error of a correct bit.

The amount of redundancy required to achieve a certain level of error-detection and correction performance decreases as the $(m \times n)$-matrix approaches a square matrix. Rectangular codes are capable of correcting single bit errors and of detecting double bit errors. But as the amount of data increases the performance decreases (with respect to the computational performance due to the computational overhead).

Rectangular codes are the basis for a method refered to as *Algorithm Based Fault Tolerance* (ABFT) [22], where the goal is to harden algorithms which operate on matrix structures like matrix multiplications.

3.4.3.3 Hamming Codes

Hamming codes, named after their developer Richard Hamming, employ c check bits to detect or correct erroneous information. Depending on the amount of chek bits, Hamming codes can detect and correct multiple errors. The case of $c = 1$ corresponds to parity checking and has already been discussed in subsection 3.4.3.1. For a better understanding some basic definitions need to be introduced at this point.

Hamming Distance

Consider a binary string of specific length referred to by one of the following synonymous terms: binary *block*, binary *vector*, binary *word* or just *codeword*. The hamming distance between to vectors of the same length is the number of bits in which they differ. The code *distance* is the minimum Hamming distance, which is the minimum amount of bits by which any code word differs from another.

Based on the Hamming distance of a code it is now possible to quantify its error-detection and error-correction capabilities. This sections only deals with *linear codes* where the difference and sum between two code words (in binary representation) results in a valid code word. The parameters of a Hamming code are:

d	...	the hamming distance of a code	(3.1)
D	...	the number of errors a code can detect	(3.2)
C	...	the number of errors a code can correct	(3.3)
n	...	the total number of bits in the code word	(3.4)
m	...	the number of information bits	(3.5)
c	...	the number of check (parity) bits	(3.6)

where $d, D, C, n, m,$ and c are all integers greater than or equal to zero. To detect D errors, the Hamming distance must exceed D at least by one, expressed as

$$d \geq D + 1. \tag{3.7}$$

The most popular codes are the so called parity code with $d = 3$ and $D = C = 1$, which is a single error-correction and single error-detection (SECSED) code as well as the single error-correction and double error detection (SECDED) code with $d = 4$, $D = 2$ and $C = 1$. A SECDED code is mainly used for implementing hardware EDAC because of both its determinism and its constant overhead.

General Hamming Coding Scheme

A coding scheme prodives a mapping of input-data to codewords. As mentioned earlier, the codeword contains extra check bits that are used for error-detection and error-correction. Consider a 32-bit data word represented by the row vector $D[d_0 d_1 \dots d_{31}]$. A SECDED Hamming Code, denoted as $(39, 32)$-code adds 7 check bits to these 32 bits and creates 39-bit codewords $(D[d_0 d_1 \dots d_{31} c_0 c_1 \dots c_6])$. Using this coding scheme the data bits are not changed and remain separable from the check bits so that this code type is referred to as systematic or separable code. An example for this coding scheme is used in the LEON2-FT processor which has built-in EDAC support for external memory interface, the integer-unit register file and the floating point unit register files. Table 3.3 shows the syndrome bits generation used in the $(39, 32)$-code.

For the particular case of the LEON2-FT the check bits are transported over a 40-bit external bus interface. Actually only 39-bits are required. The last bit is used for internal test procedures. Although it sounds very cheap to implement the coding, additional hardware add costs. Power and chip-area impact are almost as for triple modular redundancy (TMR) but coding is superior with respect to clock loading[5]. The logic needed for the generation for the decoding of check (or syndrome) bits adds further delay to the critical path.

---

[5] `http://klabs.org/richcontent/MAPLDCon00/Presentations/Session_A/A4_Barto_S.PDF`

$$
\begin{aligned}
c_0 = \quad & d_0 \oplus d_4 \oplus d_6 \oplus d_7 \oplus d_8 \oplus d_9 \oplus d_{11} \oplus d_{14} \oplus d_{17} \oplus d_{18} \oplus d_{19} \oplus d_{21} \oplus d_{26} \oplus d_{28} \oplus d_{29} \oplus d_{31} \\
c_1 = \quad & d_0 \oplus d_1 \oplus d_2 \oplus d_4 \oplus d_6 \oplus d_8 \oplus d_{10} \oplus d_{12} \oplus d_{16} \oplus d_{17} \oplus d_{18} \oplus d_{20} \oplus d_{22} \oplus d_{24} \oplus d_{26} \oplus d_{28} \\
c_2 = \quad & d_0 \oplus d_3 \oplus d_4 \oplus d_7 \oplus d_9 \oplus d_{10} \oplus d_{13} \oplus d_{15} \oplus d_{16} \oplus d_{19} \oplus d_{20} \oplus d_{23} \oplus d_{25} \oplus d_{26} \oplus d_{29} \oplus d_{31} \\
c_3 = \quad & d_0 \oplus d_1 \oplus d_5 \oplus d_6 \oplus d_7 \oplus d_{11} \oplus d_{12} \oplus d_{13} \oplus d_{16} \oplus d_{17} \oplus d_{21} \oplus d_{22} \oplus d_{23} \oplus d_{27} \oplus d_{28} \oplus d_{29} \\
c_4 = \quad & d_2 \oplus d_3 \oplus d_4 \oplus d_5 \oplus d_6 \oplus d_7 \oplus d_{14} \oplus d_{15} \oplus d_{18} \oplus d_{19} \oplus d_{20} \oplus d_{21} \oplus d_{22} \oplus d_{23} \oplus d_{30} \oplus d_{31} \\
c_5 = \quad & d_8 \oplus d_9 \oplus d_{10} \oplus d_{11} \oplus d_{12} \oplus d_{13} \oplus d_{14} \oplus d_{15} \oplus d_{24} \oplus d_{25} \oplus d_{26} \oplus d_{27} \oplus d_{28} \oplus d_{29} \oplus d_{30} \oplus d_{31} \\
c_6 = \quad & d_0 \oplus d_1 \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_5 \oplus d_6 \oplus d_7 \oplus d_{24} \oplus d_{25} \oplus d_{26} \oplus d_{27} \oplus d_{28} \oplus d_{29} \oplus d_{30} \oplus d_{31}
\end{aligned}
$$

Table 3.3: Hamming code used in LEON2-FT

3.4.3.4 Reed-Solomon Codes

So far, only binary codes, where information and code symbols are single bits, were considered. However, several bits can be summoned to form non-binary symbols and redundant non-binary symbols can be added for protection. This is referred to as non-binary coding. Non-binary coding appears naturally in systems with (naturally) non-binary data structures as, for example, in computers with data widths of bytes (8-bit), words (16-bit), double- or quad-words. The most famous non-binary codes are the Reed-Solomon codes, which are a special case of the BCH-Code (Bose-Chaudhuri-Hocquenghem-Codes) which were first presented in 1960 [51]. These codes are employed, if double or even multiple bit-error correction is required. Like the Hamming codes, the RS-code belongs to the group of systematic codes. Reed-Solomon codes are block-codes, which means that both information and its associated parity symbols form a block of n symbols. Individual blocks can be decoded without having to consider adjacent blocks.
The parameters of a Reed-Solomon code are:

$$m \qquad \ldots \text{number of bits per symbol} \tag{3.8}$$

$$n \qquad \ldots \text{block length} \tag{3.9}$$

$$k \qquad \ldots \text{uncoded message length in symbols} \tag{3.10}$$

$$(n-k) \qquad \ldots \text{number of check symbols} \tag{3.11}$$

$$t \qquad \ldots \text{number of correctable symbol errors} \tag{3.12}$$

$$(n-k) \qquad \ldots t = \frac{(n-k)}{2} \; for \; (n-k) \; even, \tag{3.13}$$

$$(n-k)-1 \qquad \ldots t = \frac{(n-k)-1}{2} \; for \; (n-k) \; odd. \tag{3.14}$$

Therefore, an RS code may be described as a (n, k)-code, where $n \leq 2^m - 1$ and $n - k \geq 2t$. 8-bit symbols ($m = 8$) coded in blocks of $n = 255$ symbols is a frequently used RS-code parameter set[6]. If $t = 10$ erroneous symbols are supposed to be correctable, the number of message

---

[6] `http://s.eeweb.com/articles/2011/08/14/tutorial-reed-solomon-1313383355.pdf`

symbols that can be used per block can be found from equation 3.13 to be 235.

An additional technique used to increase performance of block codes in general and of RS-codes in particular is *Interleaving*. During this process the encoded bits are rearranged over a span of several block lengths. This makes it very effective against burst errors. Clearly the receiver must know the bit arrangement to de-interleave the data before decoding. In contrast to Hamming codes, the implementation complexity shows a great imbalance between coding and decoding. The encoding process using a linear feedback shift register (LFSR) is easy to implement but the decoding requires about 10 times more resources.

Nowadays a variety of EDAC implementations exist since error control coding is a well-developed field. A modified version of the Hamming code was presented by Hsiao in [21] which allows faster generation of check bits. This was achieved by constraining the check bits generation. The ongoing development focuses on low-power implementations [9], better dependability [66] and adaptive coding [8]. Since the trend goes for multiple bit error-correction, especially in deep-space missions, Reed-Solomon or BCH codes are becoming inevitable in future space applications. Representative implementations can be found in [28], [18] or with the EDAC implemented in Maxwel's SCS750 spaceborne Power-PC based processor module[7]. For protecting memory against transient errors hardware EDAC mechanisms have always been the preferred choice, not only because of their attractive reliability and dependability but also for their "transparent" behaviour and and their high throughput. Although the use of Commercial Off-The-Shelf (COTS) components without hardware EDAC support grows rapidly, the need for hardware supported error detection and correction is inevitable if high reliability is needed in space.

3.4.4 Replication

A commonly known method for SEU mitigation is Triple Module Redundancy (TMR) with majority voting. TMR concepts can be applied at gate level and on higher levels such as the level of function blocks, also referred to as modules. Figure 3.3 presents the basic idea in form of a logical diagram.

Among all of the proposed techniques at gate level, TMR is the most effective one. Typically, all sequential elements are triplicated and majority voting is added. Since not all elements are replicated, this creates two single points of failures, viz. the voting circuitry and the combinatorial cloud. This can be solved using a different TMR approach. Figure 3.4 shows TMR for sequential, combinatorial and voter circuitry at gate level.

If TMR at module level is considered, an existing VHDL design can easily be transformed into a radiation-hardened one. One method proposed in [14] describes the automatic insertion of radiation-hard modules at Register-Transer Level (RTL). For this purpose, a tool named *Fault Insertion Tool* (FIT) was developed, which is able process an existing VHDL description and generate a hardened one based on the user input and the technique selected. As a pre-requisite,

---

[7] `http://about.maxwell.com/pdf/me/datasheets/sbc/scs750_rev7.pdf`

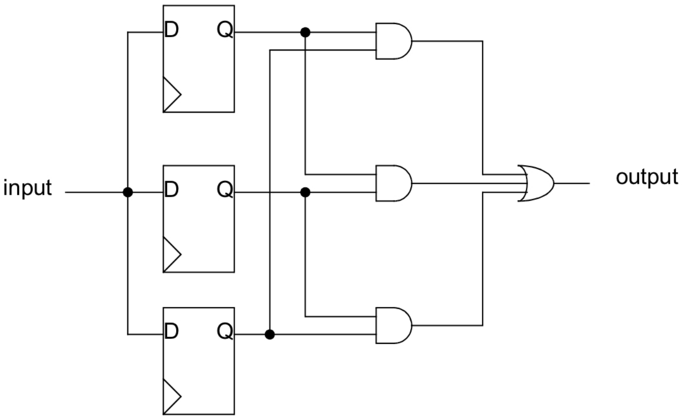


Figure 3.3: Gate-Level Triple Modular Redundancy with majority voting

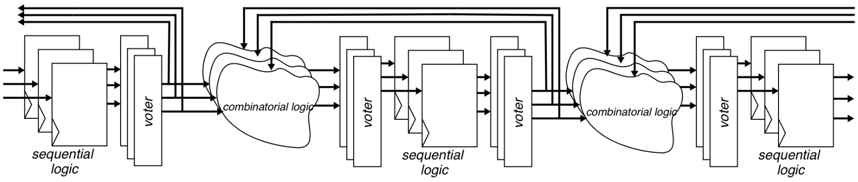


Figure 3.4: TMR for sequential, combinatorial and voter circuitry at gate level

the FT-insertion tool requires the VHDL description to be synthesizable. Figure 3.5 shows the schematic of the proposed tool.

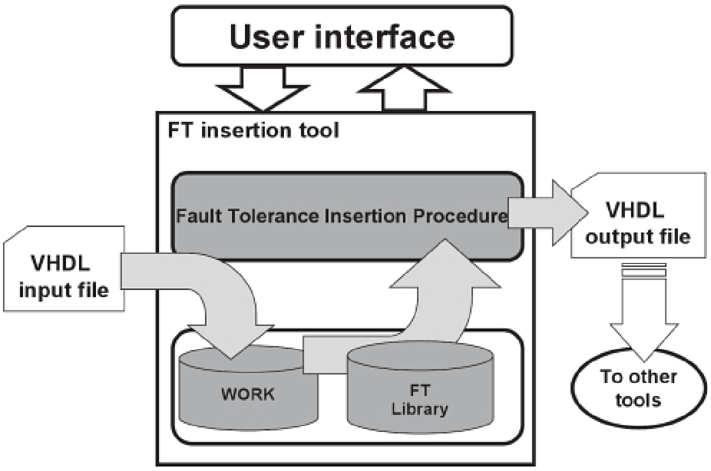


Figure 3.5: Fault Insertion Tool Principle [14]

Despite the good error mitigation capability of TMR, the main disadvantages of TMR are threefold area- and power-consumption. In literature, several ideas for improving the efficiency of TMR on implementation as well as on conceptional level have been presented. In [38], data integrity was increased by changing the voting scheme to word-voting. A self checking TMR circuit was presented in [16]. Also, partial or selective TMR was proposed as an alternative [47] [56]. The key idea is to mitigate effects only in critical sections.

3.4.5 Watchdog

A watchdog is a dedicated hardware component or a mixed solution using a hardware-timer and software to monitor a system by checking, if particular actions occur according to an expected schedule. It is used to trigger a system reset in case of an error such as an infinite loop. System monitoring using watchdog timers has always been a reliable solution to deal with timing anomalies or deadlocks arising from software or hardware failures. Two types of watchdog can be distinguished: (i) built-in processor modules called watchdog or watchdog-timer which decrements internal registers and asynchronously resetting the CPU in case of an underflow of the counter, (ii) self-made watchdogs using a built-in timer. It is called software timer because the reset of the CPU is done manually in software. To service the "watchdog" the CPU has to reload the watchdog counting register periodically using software routines. In case the execution gets stuck in an infinite loop, the watchdog counting register cannot be reloaded, thereby causing the CPU to pull the internal reset. In case of a self-built watchdog a self-written software interrupt routine can be executed. Therefore, it is under full control of the user how execution is continued. Depending on the used processor a watchdog can be found inside a device or as an external component itself.

In addition to the fact that embedded designs for space cannot be easily restarted by human interaction, deadlock of the device could result in permanent failure. Even if possible, a manual restart by humans could be too slow to meet the availability requirements of the system. In principle, such severe problems can be overcome by using a simple watchdog timer. However, unpredictable effects resulting from transient faults may reset a standard watchdog unexpectedly. To combat such problems, a more robust watchdog approache have been proposed in literature. One approach aims at reducing the probability of occurrence of such an event by limiting the time during which the system can be reset to a limited time window [67]. Nevertheless, windowed watchdog timers are unable to detect undue resets within their safe window. To overcome this problem, the windowed watchdog concept was extended to the sequenced watchdog timer concept presented in [12]. Despite their simplicity, watchdog timers have become essential elements even of the most complex self-reparable systems. It is also worth mentioning, that they play an important role in the context of *Control Flow Checking* which will be discussed in more detail in section 3.5.3. So called Hardened Cores (H-Cores) which are related to watchdog timers are used in support of TMR to ensure safe and timely recovery from SEFIs.

3.4.6 Hardware Based Scrubbing

Although EDAC helps increasing the tolerance with respect to single or multiple upsets, SEUs can accumulate over time in high capacity memories and the number of errors, therefore, exceed the EDAC detection and correction capabilities. Memory scrubbing aims at counteracting by periodic reading of data blocks. During this read process the decoder checks the information bits against the syndrome bits, corrects them, if necessary, and writes them back to the same memory location. Depending on the hardware support, this can either be part of the EDAC circuitry or needs to be done using a simple software read-back function.

Periodic scrubbing can be used to avoid the accumulation of different SEEs and, thus, reduce the probability of unrecoverable multiple errors. The efficiency does not exclusively depend on the EDAC circuitry, but on the used scrubbing interval. A well-tuned scrubbing interval renders the multiple failure probability almost to zero. Scrubbing mechanisms employ additional hardware and and they introduce processing overhead, so that less consuming methods have been devised [53].

Although scrubbing seems to be the solution for all memory related SEEs, there are still some issues to be concerned of. Depending on the scrubbing strategy some errors may remain undetected or false detections may occur. In case of autonomous scrubbing, where an dedicated component (inside or outside the memory) performs the refresh process faults occurring between read and write cycles may cause the generation of incorrect syndrome information. The consequences are obvious: false alarms or data corruption enforced by the error-correction mechanism during the next scrubbing iteration.

Therefore, scrubbing does not guarantee failure-free memory operation and should, therefore, be used in combination with other SEU mitigation techniques.

3.5 Software Implemented Fault-Tolerance

Software implemented fault-tolerance (SWIFT) plays an important role in the design of systems for critical applications. Despite the fact that there may not be any alternative to SWIFT due to lacking hardware support, in particular if COTS components are used, SWIFT solutions are attractive because of their modularity and flexibility. In contrast to dedicated hardware solutions which may be embedded as radiation hardened nuclei in the susceptible environment they are supposed to protect, software modules for fault protection are fully affected by any vulnerabilities of the taget hardware. In principle, during the execution of a software fault-tolerance module a bit-flip inside the register file of the processor could well cause modification of the control flow. Incorrect processing of the module without being noticed were the result. Aparently, a mixed approach using different software techniques needs to be used concurrently to minimize the effects caused by SEEs.

This section briefly summarizes the basic techniques used for SWIFT. Most key concepts were already introduced in Section 3.4. In this section the main differences, advantages and limitations will be considered from a software point of view.

3.5.1 Software EDAC

The first approach for using software implemented EDAC for a space mission was attempted in [58] by McCluskey, Saxena and Shirvani. Their goal was to evaluate performance and reliability for software-only solutions within the frame of the Stanford ARGOS project. Motivated by the fear of additional computation overhead, researchers studied the feasibility of using general purpose microprocessors for EDAC software-only implementation [45] [46]. The simplest approach is to use parity codes which, however, offer far less error detection and correction capability than more complex types of coding. As already mentioned in Section 3.4.3.3, more powerful error correction codes, add more check bits and tend to require more complex encoding and decoding algorithms. Consequently, system design needs to be based on a trade-off between performance overhead and error-performance.

3.5.2 Software Based Scrubbing

Similar to hardware based scrubbing its software equivalent allows minimizing the probability of accumulated faults ensuring that error recovery based on the EDAC can be successful. The crucial difference is that the software approach needs to access the memory and store the accessed information inside the register file of the processor. Consequently, it is indeed possible to encounter SEE effects during scrubbing. While this problem is also encountered for the hardware implementation, the software solution is more prone to faults since more components must be used to implement the same function.

Consider the loading of consistent information from memory into the register file. This implies correct decoding. If after correct loading the information gets modified inside the register, a subsequent write instruction will write a different memory word into the memory. This would result in a modification of the syndrome information and will thus create an undetectable loss of information.

The overhead depends on the memory distribution and on the processor used and may be extremely high if compared to a hardware implementation. Depending on the used memory and the access time needed for loading data into registers, this method creates an enormous overhead compared to a hardware solution. Consider these two solutions:

Memory	Access Time	CPU Tasks	Data Transferred
no HW ECC protection	fast	Load, Check/Correct, (Syndrome gen.), Write	(Data)
HW ECC protection	slow	Load, (Write)	Data, Syndrome

Table 3.4: Scrubbing Solutions

In Table 3.4 enclosure in parentheses denotes CPU tasks that may be omitted for specific implementations. For unprotected memory the syndrome generation may be omitted in case anomalies are not detected during the check phase. For the case of ECC protected memory the error coding is done outside the processor. Therefore, a read instruction is enough for triggering the ECC check routines of the external memory controller.

3.5.3 Control Flow Checking

Transient or permanent faults can cause an incorrect sequence of instruction execution and, thus, result in the creation of a control flow error (CFE). In the introduction the requirement for operational correctness was introduced. Since not all internal failures propagate to the system boundaries where they could become observable, internal system state monitoring is mandatory. Apparently external observers can only check deterministic behaviour visible from outside the system.

Control Flow Checking using Software Signatures (CFCSS) [43] as proposed by McCluskey, Shirvani and Oh in 2002 is such a technique for inherent run-time checking. Since the key concepts are mostly the same for all control flow checking techniques and since primarily Control Flow checking using Software Signature (CFCSS) has been evaluated within the frame of the present thesis work, the main focus will be on this technique. For the sake of completeness the other approaches will be listed as well.

3.5.3.1 Control Flow Checking Using Software Signatures

The information provided in this section is based on the terminology introduced in [43], which is summarized in Table 3.5.

V	$\{v_i : i = 1, 2 \ldots, n\}$ set of vertices denoting basic blocks
E	set of edges denoting possible control flow between basic blocks
P	program graph $\{V, E\}$
s_i	signature of v_i
d_i	signature difference in v_i
G	run-time signature
G_i	value of G in v_i
D	run-time adjusting signature
$br_{i,j}$	a branch from v_i to v_j
$suc(v_i)$	set of sucessors of v_i
$pred(v_i)$	set of predecessors of v_i

Table 3.5: Notation for Control Flow Checking based on [43]

- *Basic Block*: A maximal set of ordered instructions whose execution begins at the first instruction and terminates at the last instruction. There is no branch instruction in a basic block except possibly the last one. A basic block terminates at either an instruction branching to another basic block or an instruction receiving transfer of control flow from two or more places in the program [65].

- *Program Graph*: From the definitions of V and E, a program can be represented by a program-graph (or control flow graph - CFG) P. The $br_{i,j}$ are not necessarily explicit branch instructions; they also represent fall-through execution paths, jumps, subroutine calls and returns.
- *Illegal Branch*: v_j is in the $suc(v_i)$ if and only if $br_{i,j}$ is included in E. Similarly, v_i is in $pred(v_j)$ if and only if $br_{i,j}$ is included in E. If a program is represented by its $P = \{V, E\}$, then $br_{i,j}$ (during the execution of P) is illegal if $br_{i,j}$ is not included in E. This illegal branch indicates a control flow error which can be caused by transient or permanent faults in hardware such as the program counter, address circuits, or memory system.
- *Branch-Fan-in Node*: If a node receives more than 2 transfers of CF it is a branch-fan-in node, i.e., the number of nodes in $pred(v) > 1$.
- *Branch Insertion*: Branch-insertion occurs when one of the instructions in the node is changed to a branch-instruction as the result of an error.
- *Branch Deletion*: Branch-deletion occurs when an error causes the branch-instruction of a node to change to a nonbranch instruction. As a result, the node without the branch-instruction merges with the node that is adjacent to it in the memory address space.
- Node-Difference of v_i and v_j: the result of performing the bitwise XOR operation of v_i and v_j, i.e., xor-difference $v_i \oplus v_j$ where v_i and v_j are binary numbers.

CFCSS is able to check the control flow of programs using existing instructions based on the given architecture but without using any special hardware. To do so, the program flow has to be mapped to a control flow graph (CFG) representing the execution flow. Figure 3.6 shows the mapping of instructions to a control flow graph.

Every node inside this graph represents a basic block. Every node in this graph receives a unique number called signature. This signature can be embedded into the program code using preprocessing or at compile time. One general purpose register is used as general signature-register (GSR). During run-time, every time a transfer to a new node is done, the run-time signature G stored inside the GSR is updated with the value of the signature of the target and compared against the signature attached to the node currently processed and stored in the GSR. This allows for checking the control flow associated with deterministic and, thus, unique transitions from one node to another.

To allow for merging multiple branches into a single node, the so called runtime adjusting signature D is introduced. If combined (XORED) with G, a correct signature can be computed even for multi fan-in vertices. Due to the imperfect coverage of control-flow checking some control flow errors sill arise, even for CFEs, viz.

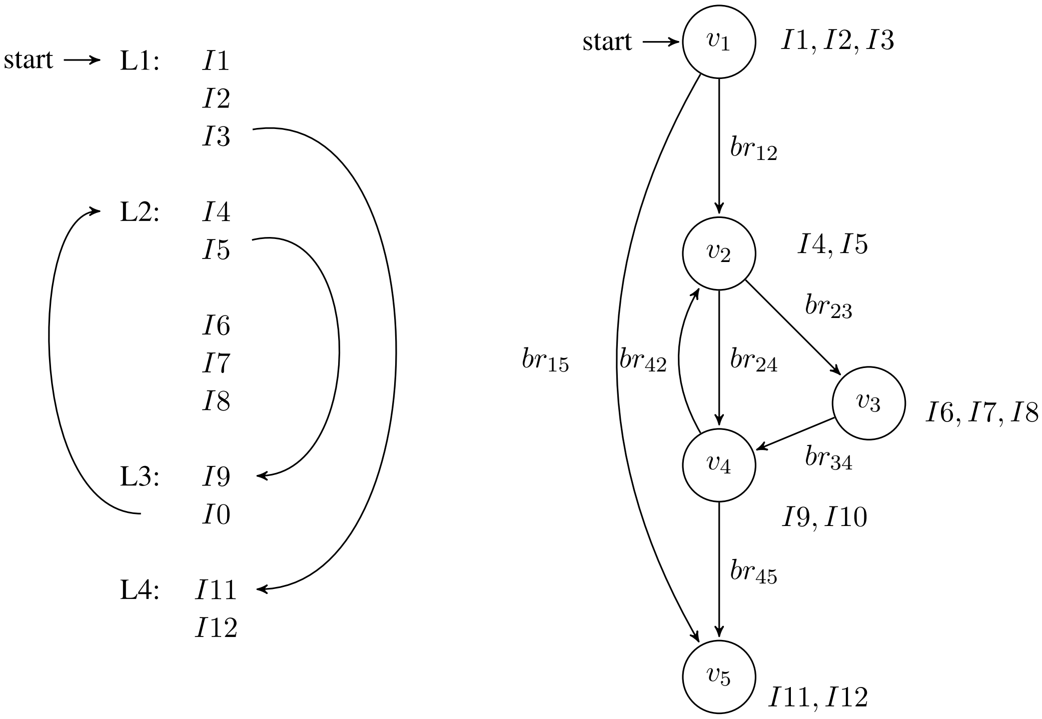


Figure 3.6: Sequence of instructions and its control flow graph [43]

- Aliasing
- Endless loops

If multiple nodes share multiple branch-fan-in nodes as their destination nodes, aliasing between legal and illegal branches may occur and cause undetectable control flow errors. The threat of aliasing can be reduced by increasing the signature size and by increasing the hamming distances between the addresses pointing to the storage locations of the first instructions of the basic blocks.

For the endless loop case consider a valid endless loop which is indeed not an erroneous control flow but also a unwanted (in many cases critical) situation. CFCSS behaviour in the context of endless loops will be discussed in more detail as part of the experimental evaluation.

Although CFCSS was introduced for intra-procedure checking it can also be used for inter-procedure checking. Assuming a standard signal processing algorithm following a deterministic control flow, every function may be mapped on a basic block. In contrast to intra-procedure

Hardware Techniques		Software Techniques	
Path Signature Analysis	[39]	assertions	[35], [1]
Signature Instruction Streams (SIS)	[57]	watchdog task	[35]
Asynchronous SIS	[11]	Block Signature Self-Checking	[34]
Continuous Signature Monitoring (CSM)	[63], [64]	Error Capturing Instructions (ECI)	[34]
extended-precision checksum method	[54]	timers to check the behaviour of the program	[20]
On-line Signature Learning and Checking (OSLC)	[34]	Available Resource-driven Control-flow monitoring (ARC)	[55]
Implicit Signature Checking (ISC)	[44]	temporal redundancy methods	[23]
Signature Checking on Instruction Level	[17]	Hardware Assested Pre-emptive CFC	[48]

Table 3.6: Classification of control flow techniques

checking the resulting graph represents the behaviour on functional level. Compared to intra-procedure checking the overhead introduced is lower.

Comparing both methodologies, it becomes noticeable that both approaches do have their advantages. Intra procedure checking allows for faster detection but introduces high overhead. Inter procedure checking benefits from low overhead and easy implementation. Consequently, a mixture of intra-procedure and inter-procedure CFCSS seems worth further investigation.

Algorithm details for CFCSS can be found in [43]. Since CFCSS has evolved from structural integrity checking (SIC) [32], the underlying concepts are the same, with the main difference that SIC needs a watchdog snooping on the data bus to perform signature checking. In another technique, referred to as Block Signature Self-Checking [33] the need for a watchdog was removed by replacing it with a subroutine. A drawback of this solution is the need for a memory location dependent signature.

Many control flow checking techniques, –based on extra hardware or without–, can be found in literature and the most importing methods are listed in Table 3.6 along with the references where further information can be found.

Most of the techniques assign signatures on block level rather than on instruction level. In particular, the ones presented in [17] and [57] allow for intra-block checking by calculating the signature from the instruction. All techniques mentioned in Table 3.6 have in common that they create additional overhead. In case the signature check is performed by the CPU additional instructions need to be scheduled during runtime. Many hardware solutions allow concurrent checking without additional instructions running in the CPU. Implementation feasibility hinges on the granularity of the control flow graph. Beside the given terminology which states that a basic block is branch free, it is also possible to apply more coarse-grained granularity in case of hard deadlines or performance bottlenecks.

3.5.4 Time Triple Modular Redundancy

Time Triple Modular Redundancy or (TTMR) [10] is a patented hybrid method combining two different fault-tolerance approaches. First Dual Modular Redundancy (DMR) is used to introduce spacial redundancy for all logic cells. As a second step, time redundancy is introduced: two identical operations are performed by the same hardware at different times. After storing

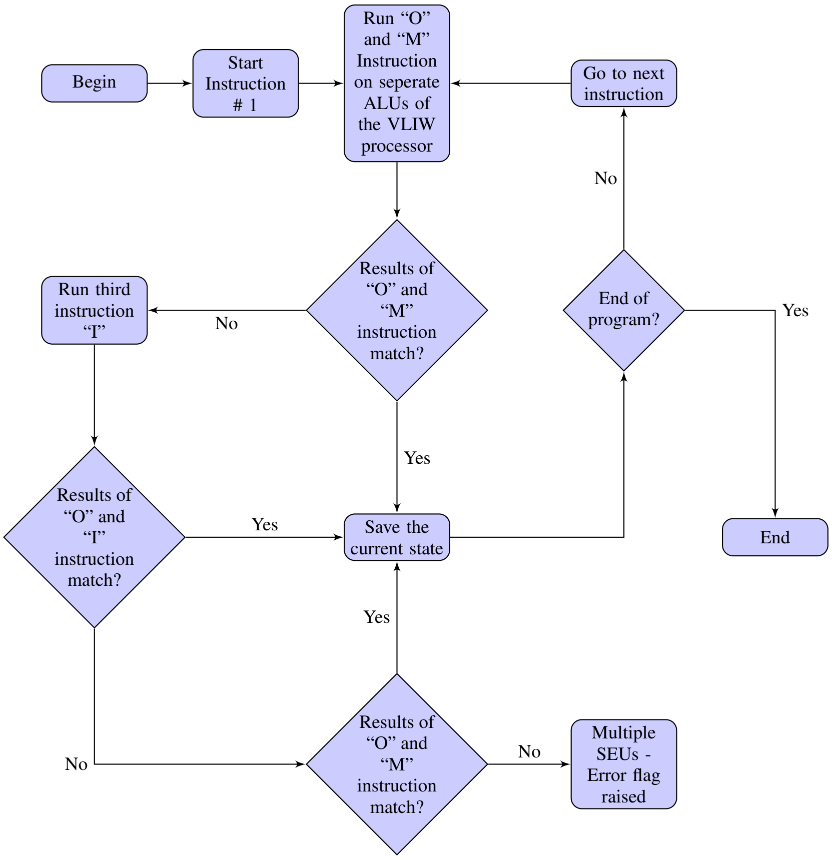


Figure 3.7: TTMR Algorithm executing three copies of programs/instructions [10]

those results gained from the DMR execution, a hardened voter resolves them.

TTMR is able to exploit the parallel executing units of a Very Long Instruction Word (VLIW) processor as redundant units by executing identical instructions in parallel. A flow chart of the algorithm is shown in Figure 3.7. To increase the overall performance DMR is used at the beginning ("`O`" for the original instruction and "`M`" for the mirrored instruction). After comparing both results, depending on the outcome of the test concerning the spatial redundancy, time redundancy may be used to execute a third instruction ("`I`") and to finally vote over all three results. It is important to mention that space micro [8], who developed TTMR and H-Core, does not licence its TTMR compiler suite for evaluation and testing. Therefore a detailed analysis was not possible.

---

[8] `http://www.spacemicro.com/`

3.5.5 Error Detection by inserting Duplicate Instructions - EDDI

Oh et al. proposed in [42] a novel software redundancy technique called EDDI wherein all instructions are duplicated and check instructions comparing the results are inserted. Because this technique works on instruction level (Assembler level) a special compiler is used to transform the “normal” code into a hardened one. Different registers and different memory regions are used so that the “copy program” does not interfere with the original outcome. Synchronization points are added at certain locations to guarantee consistency between original and redundant output values. In principle, synchronization points can be placed before store instructions since the correctness of a program is in relation to the memory output. However, this is insufficient since branch instructions could skip a relevant store instruction. Therefore, jump or branch instructions are also synchronization points. Table 3.7 and Table 3.8 show an example of mapping original code to EDDI code.

```
ld r12=[GLOBAL]

add r11=r12,r13




st m[r11]=r12
```

Table 3.7: Original Code

```
    ld r12=[GLOBAL]
1:  ld r22=[GLOBAL+offset]
    add r11=r12,r13
2:  add r21=r22,r23
3:  cmp.neq.unc p1,p0=r11,r21
4:  cmp.neq.or p1,p0=r12,r22
5:  (p1) br faultDetected
    st m[r11]=r12
6:  st m[r21+offset]=r22
```

Table 3.8: EDDI Code

Although EDDI is able to detect transient faults effectively, it is expensive in terms of computational and memory overhead necessary for saving the instrumented program copy. The overall performance of EDDI can be improved if memory is protected by some error-detection and correction mechanism. As a consequence, the amount of duplicate memory instructions can be reduced by reducing the number of stores, thus, reducing overhead and increasing efficiency.

Oh et al. also proposed a slightly adapted version of EDDI, referred to as Error Detection by Diverse Data and Duplicated Instructions (ED^4I). However, ED^4I is merely used to detect software faults rather than hardware faults –both permanent and transient. Since permanent hardware faults are not the scope of this thesis, this method will not be described in detail. However, the interested reader can find additional information in [41].

Although these approaches seem to be both simple and effective, great care has to be taken with a final judgement. The effectiveness of instruction replication techniques highly depends on the instruction set architecture used in the particular platform. The schemes described above assume that all computations are finished during one clock cycle. If this is not the case, overhead appears multiplied by the number of needed clock-cycles. Most of today’s processors for space use architectures inevitably demanding delay slots. The handling of delay slots is, thus, mandatory since simple code insertion would create non-conformal and, therefore, incorrect code.

3.5.6 Undetected Faults

As already mentioned, some faults may always go undetected. For example, due to the fact that redundancy is achieved solely through software instructions, one has to consider the delay problem due to the time interval between the availability of the validation information and the instant in time when the validated data are actually used. Potential problems could be the corruption of program code, the change of a random instruction into a store instruction or, even worse, a change into a branching instruction. Apparently, the probability of a valid instruction to be transformed into another valid instruction word depends on the Hamming distance of the instruction set. It must be clear that not all sources and causes of fault-detection failure can be listed here. The following examples shall give insight into typical undetected errors.

So called *Multi-bit Architectural State Errors* could arise through accumulation since the state of the processor is not exposed to the software application. Orocessor control logic, for example, is clearly not protectable with the exclusive use of software techniques. Consider such faults causing a deadlock. An instruction ready for execution inside the pipeline could be marked as stalled. The program execution could not continue and the program would never terminate or move into the idle state [5].

In Section 3.5.3 the problem of infinite loops was mentioned. *Control Flow Errors* such as infinite loops cannot be completely eliminated if the control flow is valid between the basic blocks. The only solution to this would be temporal checking using timers or a watchdog. This requires fully deterministic algorithm execution to have full knowledge of the execution time. Nevertheless, it is still possible to encounter infinite loops, even with watchdog checking, for example, if the infinite loop continues resetting the watchdog.

In Section 3.5.5, an improved version of EDDI using only one load instruction was presented. Because of this improvement the value inside the register gets duplicated into another register. Since there is no replication of this value, a fault occurring between the end of the load instruction and start of the copy could allow the fault to propagate to the redundant register. Obviously such a fault cannot be detected.

3.6 Combined Hardware and Software Fault-Tolerance

Due to the imperfect coverage of techniques entirely relying on software and the fact that the possibilities for adding hardware support may be very limited, it is clear that one method alone is not the solution to all problems. In Chapter 6 it will be shown that software only techniques require substantial overhead for reasonable coverage. Considering the postulated needs of next generation Space-borne instruments for science and earth observation, the capacity of potential candidate processors is just sufficient to fulfil the pure processing tasks without leaving margins for accommodating fault-tolerance overhead. Therefore, processor load due to fault protection must be at least partially off-loaded to dedicted hardware. This is the point where hybrid fault tolerance becomes interesting.

Just one example for an efficient implementation of this paradigm is TTMR in combination with H-Core. Other examples are low-cost hybrid hardware/software redundancy techniques proposed by Reis et al, referred to as CompileR Assisted Fault Tolerance (CRAFT) [52] and CompileR Assisted Fault Tolerance with Recovery (CRAFTR) [24].

Three inexpensive concepts derived from CRAFT and CRAFTR are briefly presented in the following sections.

Checking Store Buffer

In order to protect data that is written into memory the Checking Store Buffer (CSB) technique can be used to duplicate store instructions in the same way it duplicates all other instructions, except that store instructions are tagged with a single-bit version identifier, indicating whether a store is an original or a duplicate. The modified code is then run on hardware incorporating an augmented store buffer, which does not commit data to be written to memory until it is validated. An entry becomes validated once the original and the duplicate version of the store have been sent to the store buffer and the addresses and values of the two stores match perfectly. Although this technique duplicates all stores extra memory traffic is not created, since there is only one memory transaction for each pair of stores. Another benefit is that each pair of instructions can now be scheduled independently, whereas in for other concepts, e.g. EDDI, store instructions are synchronization points.

Load Value Queue

In traditional SWIFT techniques load values need to be duplicated to enable redundant computation. The principle is the same as before. Instead of multiple loads from memory a dedicated buffer called Load Value Queue (LVQ) is used. The LVQ only accesses memory for the original load instruction and bypasses the load value for the duplicate load from the LVQ. An LVQ entry is deallocated if and only if both original and duplicate versions of a load have executed successfully. A duplicate load can successfully bypass the load value from the LVQ if and only if its address matches that of the original load buffered in the LVQ.

Checking Store Buffer and Load Value Queue

This technique duplicates both store and load instructions and adds both the checking store buffer and the load value queue enhancements simultaneously. This allows to reduce the performance degradation and to increase the level of fault coverage, if compared to traditional SWIFT methods.

Since modifications of off-the-shelf Integrated Circuits (ICs) are almost impossible, the addition of external hardware, regardless if programmable or not, seems to be inevitable. This issue will be retrieved in Section 5.7.

CHAPTER 4

Digital Signal Processing Platform

After having reviewed SEEs as well as their origins and possible mitigation techniques, it is possible to select the processing core to form a platform for evaluating the most promising candidates. Clearly, this choice is supposed to be ITAR-free and sustainable. Since the present work is performed in a space industrial context, the freedom we have in this selection process is very limited. On the one hand, the processing platform is supposed to be programmable in the strict sense, asking for the possibility to use different programming languages – at least assembly language and C – as well as for tool support in form of compilers, debugger, in-circuit-emulator, evaluation board, etc. This excludes the implementation as *configurable* processing platform such as an Application Specific Integrated Circuit (ASIC) containing data paths, optimized for particular algorithms, e.g. multiply-accumulate, FIR-filtering, functional transforms, etc., controlled by means of a simple state-machine-controller, but also Field Programmable Gate Arrays (FPGAs), regardless, if re-programable in space or not. Nevertheless, in spite of the required flexibility with respect to processing needs, the processing platform is supposed to directly target classical DSP applications and, thus, not to be based on a Von-Neumann architecture. Finally, mission targets are near future and operational, meaning that the selected platform will eventually have to be accepted by risk-aware mission primes and space agencies. This forbids selecting the latest technologies, which, although very fast, are lacking heritage on the one hand and are still prone to rapid obsolescence on the other.

The aforementioned near-future space missions also define the desired level of processing performance. Based on mission analyses performed by ESA and on roadmapping done by Austrian space industry, the following mission-types, listed along with their data processing requirements, may be taken as guidelines:

Science: Spectrometers (EUCLID, PLATO, SPICA) - up to a few 100 MFLOPS

Earth Observation - Optical: Infra-Red Interferometers (Infrared Atmospheric Sounder (IASI), METEOSAT 3rd Generation) - 2.2 Gbps and 10 GFLOPS

Earth Observation - Microwave: Synthetic Aperture Radars (SARs) - several Gbps

In ESA terminology *science missions* are directed towards astrophysics and fundamental physics and are strictly distinguished from *earth-observation missions* which, however, may as well be very scientific in nature. EUCLID, PLATO and SPICA are science missions forming part of ESA's *Cosmic Vision* program. EUCLID aims at mapping the geometry of the dark universe, which accounts for the vast majority (76%) of the energy density of the universe, PLATO will study PLAnetary Transits and Oscillations of stars and SPICA hopes to discover the origins of galaxies, stars and planets.

The current state-of-the art in spaceborne digital signal processing is (still) represented by the TSC21020 DSP developed by ESA almost two decades ago. At a clock-frequency of 20 MHz it offers 40 MFLOPS sustained and 60 MFLOPS peak performance. The TSC 21020 was developed by licencing the design files of the commercial DSP ADSP 21020 by Analog Devices and implementing this design in a European radiation tolerant ASIC process. A similar approach for a successor to the TSC 21020 is currently under discussion but far from even being initiated because Europen focus is on driving the evlution of SPARC-architecture based spaceborne general purpose processors of the LEON type, which has been explicitely excluded from becoming a candidate within the present context.

Among the limited number of programmable DSPs suitable for space from a reliability and quality assurance point of view as well as exhibiting a minimum of radiation tolerance to the extent that radiation-effect mitigation techniques can in principle be successful, only devices from Texas Instruments (TI) seem to satisfy our requirements at least partially. Although TI is a US company, their devices are not explicitely produced for space applications and, thus, ITAR free. The TI-component best suited for our applications is the SMV 320C6701, which has already attained flight heritage during several space missions, e.g. GEZGIN [27] or the image-processing subsystems of BILSAT-1 or SPHERES [13]. Since this component has the best chances to qualify for upcoming space missions and since there is already a code-compatible successor component, viz. the SM 320C6727, under development, it has been selected as platform for the practical part of the present thesis work dedicated to the evaluation of the capabilities of candidate software enabled radiation mitigation techniques. In the following sections architecture and relevant features are reviewed and references to detailed sources are provided.

4.1 SMV320C6701 Digital Signal Processor - DSP

The basis of this component is the TMS320C6701 DSP which was introduced in 1998 as the world's highest performance floating-point DSP at that time. The DSP consists of eight independent functional units, each of which can execute a 32 bit instruction every clock cycle. The instruction set architecture is a slightly modified Verly Long Instruction Word (VLIW) architecture referred to as VelociTI^{TM}.

Although the TMS320C6701 can operate at up to 167MHz the space version[1] only achieves 140MHz clock rate, which gives a total throughput of 1.120 million instructions per second. The instruction set is RISC-like and register based, which means that nearly all instructions operate on registers located in the register file. Six out of the eight function units are capable of performing floating point operations each clock cycle. Four of these independent function units allow the processor to compute two single-point-precision operations and two multiply and accumulate (MAC) operations every clock cycle. Memory space is byte addressable using load and store instructions allowing movement of 8-, 16-, 32- and 64-bit data. Two out of the eight functional units support simultaneous memory access every cycle allowing, 128-bit information transfers.

The on-chip memory consists of one megabit of Static Random Access Memory (SRAM) organized as Harvard-architecture using different address ranges for program and data memory. The memory is split equally, so that 64KB are available for each address range. The instruction memory is able to hold 16K 32-bit instructions or 2K 256-bit VLIW instructions. Alternatively, the program memory can be configured as program cache to improve external memory latency.

To ensure independent transfer of data during computation a Direct Memory Access (DMA) controller consisting of four channels plus an additional auxiliary channel for the Host Processor Interface (HPI) is available. Due to the internal bus structure, the core is able to perform only one DMA-channel operation at time.

External memory interfaces are supported via the External Memory Interface (EMIF) logic allowing glueless connection to different asynchronous and synchronous memory types. The aforementioned fifth DMA channel is hard-wired to the HPI-port which allows internal access to nearly the complete memory map of the DSP without CPU intervention. Through HPI a general purpose processor or external fault tolerance controller is able to perform various actions on the core such as bootloading, data-transfer, configuration and even fault-tolerance. Additional information about the core, the instruction set and peripheral devices can be found on the Texas Instruments product web-site [2]. Figure 4.1 shows the functional diagram of the DSP.

4.1.1 Radiation Tolerance

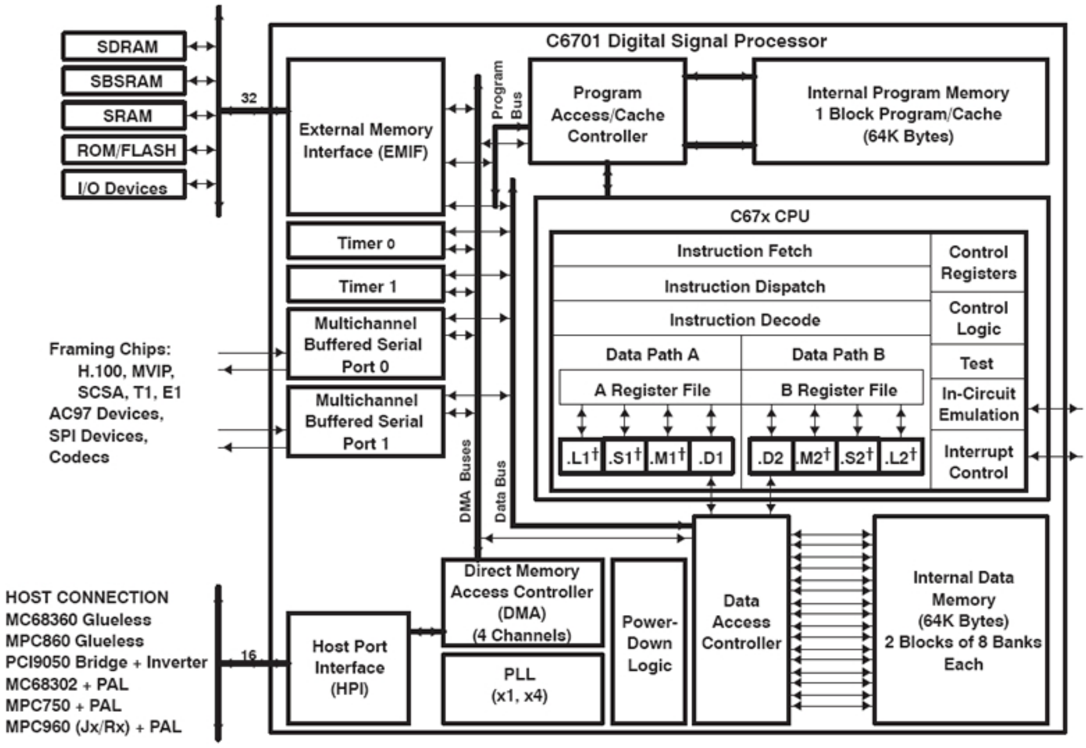
SMV320C6701 properties concerning radiation, as provided in the data sheet [25] are:

- Rad-Tolerant: 100-kRad (Si) TID
- SEL Immune at 89MeV-cm^2/mg LET Ions
- QML-V Qualified, SMD 5962-98661
- QML Processing according to MIL-PRF-38535

---

[1] `http://www.ti.com/lit/ds/sgus030e/sgus030e.pdf`

[2] `http://www.ti.com/product/smj320c6701-sp`



† These functional units execute floating-point instructions.

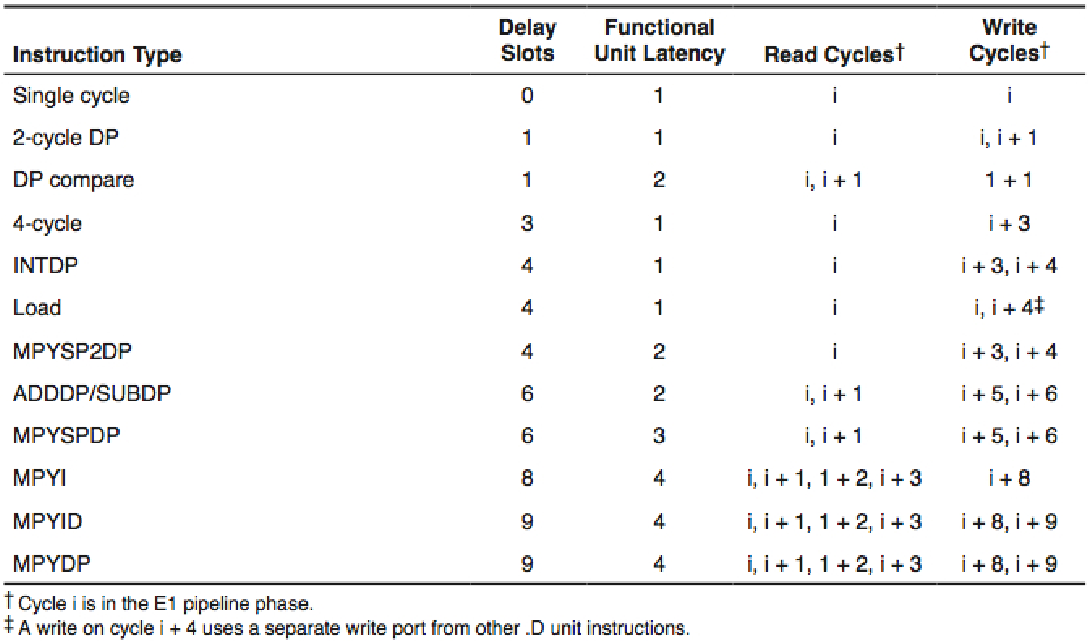
Figure 4.1: Functional diagram of the C6701 core [25]

- Temperature range from $-55°C$ to $115°C$

To reach these values, the radiation tolerance of the commercial version was increased on semiconductor-technology level, without the addition of extra fault tolerance logic. The QML-V design is based on an epitaxial layer process, process which enhances SEL-tolerance. An evaluation of TID and SEE behaviour of this DSP can be found in [30]. Figure A.1, A.2, A.3 and A.4 show the results of SEU estimation using a CREME96 model [62]. A summary of estimated SEU-rates estimation published by TI can be found in Table A.1. The proton upset rates for various LEO orbits are listed in Tables A.2, A.3, A.4.

4.1.2 Radiation Relevant Processor Behaviour

In the following sections some processor specific behaviour will be described. This is necessary since the mapping of SWIFT-methods depends on the internal behaviour of the DSP.



Instruction Type	Delay Slots	Functional Unit Latency	Read Cycles†	Write Cycles†
Single cycle	0	1	i	i
2-cycle DP	1	1	i	i, i + 1
DP compare	1	2	i, i + 1	1 + 1
4-cycle	3	1	i	i + 3
INTDP	4	1	i	i + 3, i + 4
Load	4	1	i	i, i + 4‡
MPYSP2DP	4	2	i	i + 3, i + 4
ADDDP/SUBDP	6	2	i, i + 1	i + 5, i + 6
MPYSPDP	6	3	i, i + 1	i + 5, i + 6
MPYI	8	4	i, i + 1, 1 + 2, i + 3	i + 8
MPYID	9	4	i, i + 1, 1 + 2, i + 3	i + 8, i + 9
MPYDP	9	4	i, i + 1, 1 + 2, i + 3	i + 8, i + 9

† Cycle i is in the E1 pipeline phase.
‡ A write on cycle i + 4 uses a separate write port from other .D unit instructions.

Figure 4.2: Delay Slots and Functional Unit Latencies [26]

4.1.2.1 Delay Slots and Functional Unit Latency

This property is limited to floating point and memory operations. The term delay slot refers to additional cycles needed until results become available for reading after operands have been fetched. A single cycle or single delay-slot instruction executed in cycle i reads all operands in this cycle and stores the result so that it is available for reading in cycle $i + 1$. A four delay slot instruction would read operands in cycle i as well and produce the result ready for reading in cycle $i + 4$.

Functional Unit Latency (FUL) on the other hand corresponds to the amount of cycles the unit is busy and, therefore, unable to accept new instructions. Notice that due to pipelining FUL can be one, even for multi-cycle instructions. This knowledge is essential since many software fault tolerance techniques deal with atomic replication and insertion of instructions. Figure 4.2 shows the number of delay slots associated with each type of instruction.

4.1.2.2 Instruction Fetching and Parallelism

An instruction fetch operation always fetches eight instructions at a time, coded in a single VLIW. Since eight function units can operate in parallel, the basic format of a *Fetch Packet* defines the execution of each instruction individually. The execution is partially controlled by the p-bit (bit 0), which decides whether an instruction is executed in parallel or after another instruction. If p-bit of instruction i is 1, then instruction $i + 1$ is executed in parallel within the

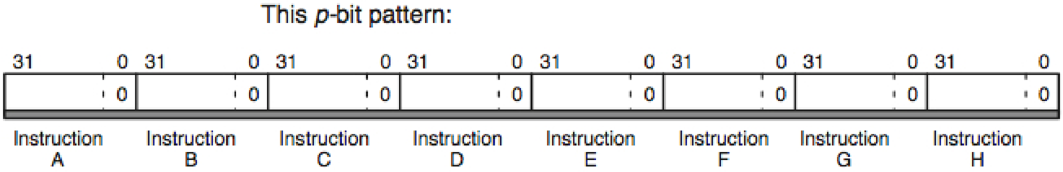


Figure 4.3: Fetch packet p-bit pattern for fully sequential processing [26]

same cycle as i. On the other hand, if the p-bit of instruction i is 0, then instruction $i + 1$ is executed after instruction i. All instructions executing in parallel constitute an *execute packet*. Each instruction of an execute packet is assigned to a different functional unit. The combination of different p-bit patterns allows for forming three different execution sequences, viz.

- Fully sequential
- Fully parallel
- Partially sequential

Figure 4.3 illustrates the meaning of the p-bit inside the VLIW execute packet.

4.1.3 Conditional Operations

Controlled by a 3-bit opcode field referred to as *creg*, most instructions can be conditional. This enables the compiler to generate efficient code since conditional instructions can be scheduled without prior checking of conditions. Not all registers are able to support the checking of conditions. Only five registers inside the register file can be used for this purpose viz. B0, B1, B2, A1 and A2.

	[B0]	ADD	.L1	A1,A2,A3
||		ADD	.L2x	A1,B2,B3

Table 4.1: Conditional code execution example

Table 4.1 gives an example of two parallel instructions, where one of the instructions is only executed if the value stored in register B0 is larger than zero. The other instruction is executed in any case, regardless of the value stored in B0.

4.2 SEU Threat Scenarios

Although permanent faults are dangerous for electronic components, they will be treated as negligible in this evaluation since they are relatively rare. The main focus will lie on transient faults or *soft errors* and their possible location and effects:

- Program Memory:

  - Modification of program code without immediate detection: erroneous instruction remains syntactically valid after impact
  - Modification of program code with immediate detection: erroneous instruction becomes invalid opcode. The SMV320C6701 does not provide a dedicated exception for ‘ìllegal opcode” as most processors do. The outcome of such opcode is classified as undefined behaviour by the manufacturer.
  - Result:
    - Data related errors
    - Control flow errors
  - Code statements can be divided into two types regardless of their location:

    IN1: Statements affecting data (assignments, arithmetic expressions, computations, etc.)

    IN2: Statements affecting the execution flow (e.g. tests, loops, procedure calls and returns)

- Data Memory:
  - Change of constant values (coefficients, state information)
  - Change of data before, during or after processing (ready to transfer to data handling unit (DHU))
  - Result:
    - Data related errors
    - Control flow errors
- Register File:
  - Basic registers: A,B 0-16 - Core registers including control service register and interrupt control register
  - Peripheral Registers: Timers mapped to Interrupts, GPIO, EMIF
  - DMA: Input, Output, Internal state
  - McBSP: Communication, Commanding and Telemetry
  - Result:
    - Data related errors
    - Control flow errors

There are still additional fault locations that were not mentioned explicitly, although they can be responsible for soft-errors. Such locations are combinatorial logic, internal registers, clock tree, etc.

A classification proposed in [50] permits good characterization of experienced behaviour. To address the **immediate** effects caused by errors a distinction of two types of errors is proposed, viz.[3]

ER1: Errors changing the operation to be performed by the statement, without changing the code execution flow, e.g. changing an *add* into a *sub*

ER2: Errors changing the execution flow immediately, e.g. by transforming an *add* operation into a *branch* or vice versa

This classification allows to characterize SEEs on a higher abstraction level, without loss of information and with practical granularity. Through linking of the given statements and errors all possible fault classes can emerge. To address error types ER1 and ER2 a short analysis of the used instruction syntax and opcode is required. The reference for this can be found in the *CPU and Instruction Set Reference Guide*[4].

4.2.1 Opcode Hamming Distance

The Hamming-distance between two instructions is of interest because it gives an indication on how many bit-flips are necessary to create another valid code word, for example to turn an arithmetic operation into branching. The hamming distance between two instructions of the same type is one. Consider two identical *ADD* instructions where the only difference is that the operand register address of operand A is different. Within the frame of a DSP evaluation, a complete analysis of all instructions with all possible parameters would be too complex. Therefore, a test program using various signal processing algorithms to statistically evaluate the hamming distance between different branch instructions and all other instruction inside the program was used. The mean Hamming-distance found with this approach for the C6701 is 2.

4.3 Failure Model

In Section 4.2 the causes and consequences of failures in data and program memory were addressed. The main focus of the failure model presented in this section will be put on the well known *Single Event Upset* described by the *bit-flip* fault model. This model is based on the assumption that for a particular storage cell only a single bit-flip occurs during the circuit's execution. To evaluate the availability of the system, injected faults are classified according to the effect the have on the program behaviour. Distinguished effects are:

- No error (unnecessary for correct execution)
- Error detected and correctly handled
- Error detected but incorrectly handled
- Error undetected

---

[3] delayed change in execution flow still possible by corrupting e.g. the counter!

[4] `http://www.ti.com/lit/ug/spru733a/spru733a.pdf`

The model in this thesis is directed towards the description of SEU effects modifying the program behaviour and on data errors. Although data errors were also analysed, the main focus is on control-flow errors. For most data processing platforms used in space the main task is number crunching so that sporadic failures are tolerated up to a given level as long as the platform continues processing data and remains open for receiving commands.

In Section 4.1.2.1,4.1.2.2 ,4.1.3 and 4.2.1 processor architecture dependent properties were presented. These properties allow for a compact construction of higher level statements but also introduce the threat of transforming low-level errors up to a higher level. Important examples are:

- p-bit mutation,
- creg mutation and
- Opcode illegalisation.

4.3.1 p-bit mutation

In a radiation environment, the processor's use of the p-bit pattern introduces considerable risk. Imagine a highly optimized code. The compiler generated code is optimized such that during each cycle new output values are generated. Concurrently new input operands are loaded. A p-bit flip during execution would completely disturb instruction scheduling. While some part of the fetch-packet had already been executed the other were still in the pipeline (many situations create undefined behaviour inside the core and are also not documented).

These effects are unpredictable due to incomplete documentation. The possible outcome could affect both, control flow and data consistency.

4.3.2 creg mutation

Similar to the p-bit pattern problem, conditional statements are both a blessing and a curse. The curse is associated with a single hit on either the conditional register or the instruction itself, which could result in an obviation of a conditional branch instruction or in the simultaneous occurrence of two mutually exclusive instructions. The "||" prefix indicates parallel execution in the C6X assembler. The .L1 and .L2x postfixes indicate the data-path ("1" for data-path A and "2" for data-path B, the additional x indicates the cross path between two sides) and, thus, reference the executing functional unit. This specific case is illustrated by means of program code in Tables 4.2 and 4.3.

```
     [B0]   ADD   .L1    A1,A2,A3
||   [!B0]  ADD   .L2x   A1,B2,B3
```

Table 4.2: Original Code

```
     [B0]   ADD   .L1    A1,A2,A3
||          ADD   .L2x   A1,B2,B3
```

Table 4.3: Modified Code

4.3.3 Opcode Illegalisation

It has been mentioned already that illegal opcodes do not cause exceptions in the CPU and pipeline conflicts are not 0-delay resolved. For such cases the behaviour is undocumented and must, hence, be assumed undefined. Since even the insertion or modification of the simple *NOP* instruction could result in undefined behaviour, possible outcomes and their consequences need to be analyzed.

As already mentioned in Section 4.2

- an erroneous instruction may remain syntactically valid after impact or
- an erroneous instruction may become invalid opcode

In flight software the entire program memory space is hardly ever used and the unused space is filled with jump instructions into the reset vector.

Consider a multi cycle *NOP* instruction in parallel with an *ADD*. The results of both operations will only be available after completion of the NOP cycle. In case a branch is scheduled in parallel with a multi cycle *NOP* this can become a problem.

The former statements assume that the program code is modified before the execute packet is fetched from memory. Although an unintended branch instruction may be created due to SEEs it may never be executed. The decoding stage of the pipeline is split in two parts. First instruction dispatching is done and instructions are assigned to appropriate functional units. Second, instruction decoding is performed. Source registers, destination registers and associated paths are decoded in preparation of the execution of the instructions in the functional units. In case a fetch packet contains two branches, a third branch resulting from an SEE would imply that one of these three branches will not be serviced. Unfortunately, there is no information inside the documentation available.

To sum up, it is not possible to estimate “typical” outcomes without resorting to statistical analysis using fault injection. With fault injection experiments it is possible to assess the probability of illegal opcode and to determine how often illegal opcode will on average corrupt the program flow or even destroy data consistency.

4.3.4 Architectural Barriers

The DSP’s architecture, optimized for high speed signal processing, complicates the implementation of SWIFT methods. A list of such barriers, found during the course of the present thesis work, is provided as Table 4.4.

Access to L1P

Because of the DSP’s Harvard architecture, there is no possibility to access program code in real-time. This prohibits checking for errors inside the program code directly via instructions.

Issue	Origin	Proposed solution
no access to L1P	Harvard architecture	access L1P via DMA
no illegal opcode detection	architecture weakness	periodic check/re-programming (reduce probability)
missing register locking	compiler	software workaround

Table 4.4: Architectural implications complicating SWIFT implementation

Consequently, it is not possible to detect corrupted instructions. DMA can be used to access the program memory and to transfer selected regions into the data memory for further inspection. As a result of the inevitable transfer delay, the drawback of this solution is a consistency gap introduced between information source and destination.

Detection of Illegal Opcode

Since the hardware does not provide exception mechanisms for illegal opcode, the programmer must take into account that the execution code may not be free of errors. Software mechanisms can be used for detecting a limited number of illegal instructions inside the program memory before execution. The only one method for precise detection is to compare the code with a representative copy which should be protected against SEEs.

The underlying idea is to store and access a protected representative copy of critical code sections, which can be used to compare portions of the program memory before their execution. Since the program memory is not directly accessible, a DMA transfer task can be used to copy fragments of critical code sections into the data memory L1D. DMA transfers can be performed current to CPU execution and, thus, the overhead is due to the latency introduced by the DMA. Comparison of the code fragment retrieved by DMA and the protected copy must be completed before the actual execution.

In spite of the existence of other ways to detect illegal opcode, the aforementioned solution is the only one capable to locate the corruption. Other techniques rely on the fact that anomalies during execution would cause collateral anomalies with higher detection probability.

A more probabilistic approach to reduce the sensitivity with respect to illegal opcode uses the the program cache controller to permanently fetch instructions from an external ECC protected source. However, this only works if the size of the code running on the target exceeds the internal program cache area, which is often not the case due to the simplicity of DSP algorithms. Another solution could be the use a timed DMA transfer task to re-write portions of the program memory.

Integrity checking or re-programming, whichever technique is used, can only reduce the probability of executing illegal opcodes, but it does not completely solve the problem.

Missing register locking

For many SWIFT methods a dedicated register to hold state information is needed. Since it is not possible to lock a register for special purposes, the programmer must take care that the required information is available during execution and stored safely if not needed. The created overhead must not be underestimated, since the compiler must schedule additional load and store operations.

CHAPTER 5

Concept and Experimental Implementation

5.1 Introduction

When implementing SWIFT methods, developers of applications or algorithms do not want to rewrite their code. This is not unjustified due to immense costs for validating qualified flight-code. The upcoming need for computational performance and the complexity of modern digital signal processors demand huge creativity when it comes to the design of timing critical applications. Additionally one must think of the time needed to modify the existing code and to ensure availability of the overall application. To support those actions automated solutions based on pre- or post-compilation or automated code transformation are preferred solutions. Unfortunately the availability of such commercial tools is rare. Those companies which provide automated solutions only offer them combined with their platform design, which is unwanted in many cases.

The second unpleasant problem arises if the performance penalty gained from the additional overhead created by SWIFT methods is to large. SWIFT methods were originally developed for the use with COTS components. COTS components are obviously not designed to tolerate radiation nor have been considered especially for space usage. The reason for the usage of COTS components was that their performance was often a multiple of today's available space components, which outweighs the performance reduction created by SWIFT methods through their immense performance advantage.

However, since there is a huge encroachment to establish an ITAR free design, and their is no comparably other DSP available, the Texas Instruments based DSP platform was selected. Technically important however is that the platform used in this evaluation is neither COTS nor radiation-hard. As already mentioned in Section 4.1.1, the DSP used has some advantages in the field of radiation performance, but cannot compete with state-of-the-art COTS components when it comes to performance or computational throughput.

This Chapter demonstrates how some of the earlier theoretically shown methods can be implemented for evaluation on the TMS320C6701 EVM (evaluation module) platform. It will be shown how efficient different SWIFT methods can be implemented and what problems can arise based on the used architecture. Additionally, the gained results will show that the fault coverage can be improved but only with significant costs. Therefore hybrid approach using hardware and software will be presented at the end.

Since the TMS320C6701 is the design base of the former mentioned SMV320C6701, the behaviour and the instruction set is identical with the space-grade version and therefore allows it to use it for evaluation. The used eclipse based development environment named *Code Composer Studio*[1] was provided by TI. A complete compiler suite [2] is provided within this IDE.

5.2 Memory Layout

To ensure transparency during the evaluation process, a uniform memory layout has been developed, allowing the developer to evaluate and internally track the effects caused by SWIFT and SEU simulations. Table 5.1 shows the memory layout provided by the evaluation module. Fot this, a custom memory section layout was superimposed. Table 5.2 depicts the memory layout defined in the linker file. The main disadvantage of using internal tracking is the reduction of operational memory, resulting in a smaller cross-section for SEU simulations. This side effect has to be taken into account since a non-invasive bus system, allowing to transfer information to the DSP in a transparent manner, is not available.

Start Address	End Address	Size (Bytes)	Description
0x00000000	0x0000FFFF	64K	Internal program memory (IPM)
0x00400000	0x0043FFFF	256K	SBSRAM
0x01780000	0x0178001F	32	DSP control/status registers
0x01800000	0x01BFFFFF	4M	Internal peripherals
0x02000000	0x023FFFFF	4M	SDRAM (bank 0)
0x03000000	0x03000000	4M	SDRAM (bank 1)
0x80000000	0x8000FFFF	64K	Internal data memory (IDM)

Table 5.1: C6701 EValuation Module memory map

Fault tolerance of COTS components is often achieved by adding a memory protection unit between the memory interface and the external memory. If the cache controller is configured to pre-fetch all instructions from this hardened memory, this allows to protect external memory using dedicated logic and, thus, maintain consistency of the program code inside the internal memory. With the Evaluation Module this approach is not possible.

---

[1] `http://www.ti.com/tool/ccstudio`

[2] code generation tools ver. 7.0.5

Nr.	Section Name	Base Address	Length	Filler	Usage
1	VECS	org: 0x0000 0000	len = 0x0000 0400	none	Interrupt vectors
2	PMEM	org: 0x0000 0400	len = 0x0000 FC00	none	Program code
3	DMEM	org: 0x8000 0000	len = 0x0000 F3F0	none	variables, coefficients, constants, . . .
4	DMEM_CFC_SPACE	org: 0x8000 F3F0	len = 0x0000 000F	none	CFC check registers G,D + 8 Byte margin
5	DMEM_MIR	org: 0x8000 F400	len = 0x0000 0500	fill: 0x10101010	$(CodeMirror)^{-1}\ \forall\ critical\_functions$
6	DMEM_CODE_SCRATCH	org: 0x8000 F900	len = 0x0000 0500	fill: 0xDEADBEAF	$program\ memory \xrightarrow{\text{DMA}} data\ memory$
7	DMEM_FI_SPACE	org: 0x8000 FE00	len = 0x0000 0200	fill: 0xA0A0A0A0	Fault detection counter, fault information
8	SBSRAM	org: 0x0040 0000	len = 0x0004 0000	none	unused
9	SDRAM2_SIM_EXT	org: 0x0200 0000	len = 0x0040 0000	none	external EDAC source (SIM)
10	SDRAM3	org: 0x0300 0000	len = 0x0040 0000	none	unused

Table 5.2: Standard memory layout as used during evaluation

```
void fir(short x[], short h[], short y[])
{
        int i,j, sum;
        for (j = 0; j < 100; j++)
        {
            sum = 0;
            for (i = 0; i < 32; i++)
            {
                sum += x[i+j] * h[i];
            }
            y[j] = sum >> 15;
        }
}
```

Listing 5.1: FIR Filter - C Code

5.3 Programming Language

As listed in 5.3, three different programming languages are readily supported by the DSP toolchain. They are listed in Table 5.3. Texas Instruments classifies those techniques based on the efficiency and the effort for programming in them. The information presented here is used later in Section 5.6.

Source	Optimization	Efficiency	Effort
Assembly	By hand	100%	High
Linear Assembly	Assembly optimizer	$95-100\%$	Medium
C/C++	Optimizing compiler	$80-100\%$	Low

Table 5.3: Optimization efficiency (relative to hand optimization)

Appendix A.17 shows the recommended design flow for developing application software for the Texas Instruments C6000 Architecture. It can be seen in Figure A.17 Phase 3, that the final step is to create linear assembly code, which is the best trade off between coding efficiency and the effort associated with code development and maintenance. Linear assembly is similar to hand assembly except that there is no need for inserting *NOP* instructions to fill empty delay slots, that the functional unit does not need to be specified, that the allocation of registers is done automatically, that the grouping of instructions in parallel is performed automatically and that symbolic variable names are accepted.

To give an example for hand optimized assembly vs. linear assembly, a finite impulse response (FIR) filter implementation written in C is shown in Listing 5.1. The equivalent code written in linear assembly is shown in Figure 5.1, while the use of hand assembly is demonstrated in Figure 5.2.

As shown before, all SWIFT techniques are based on replication, modification or the insertion of machine instructions. However, Listing 5.1 shows a C level implementation which has become abstracted from the underlying machine instructions. Finer granularity is obtained by using linear assembly and conclusions concerning its efficiency can be drawn from Figure 5.1.

Although the code is linear and without general selection of functional units the programmer can assign specific instructions to certain functional units. The code is still readable, due to the non-parallel syntax. The assembly optimizer demands a high knowledge of the language primitives. If code optimization is enabled, instructions introduced for the purpose of improving fault tolerance but irrelevant from a functional point of view may be rescheduled or even removed by the optimizer. Apparently, this complicates the implementation of software based fault tolerance on this level.

The effort associated with both establishment and maintenance of hand assembly increases exponentially with the instruction count. Figure 5.2 shows a part of the software for the same filter written in hand assembly. In contrast to Figure 5.1, Figure 5.2 does not show the whole function but only the internal loop filter - the difference is impressive.

Compared to linear assembly, partitioning and scheduling is entirely to the programmer. Compared to C or linear assembly, the effort needed for programming in hand assembly is huge. Simple modifications of existing code become a nightmare, in particular, if flight-software design rules have to be applied. Otherwise, if it is required to instrument hand assembly code it would be best doing it with automated tools instead of doing it by hand.

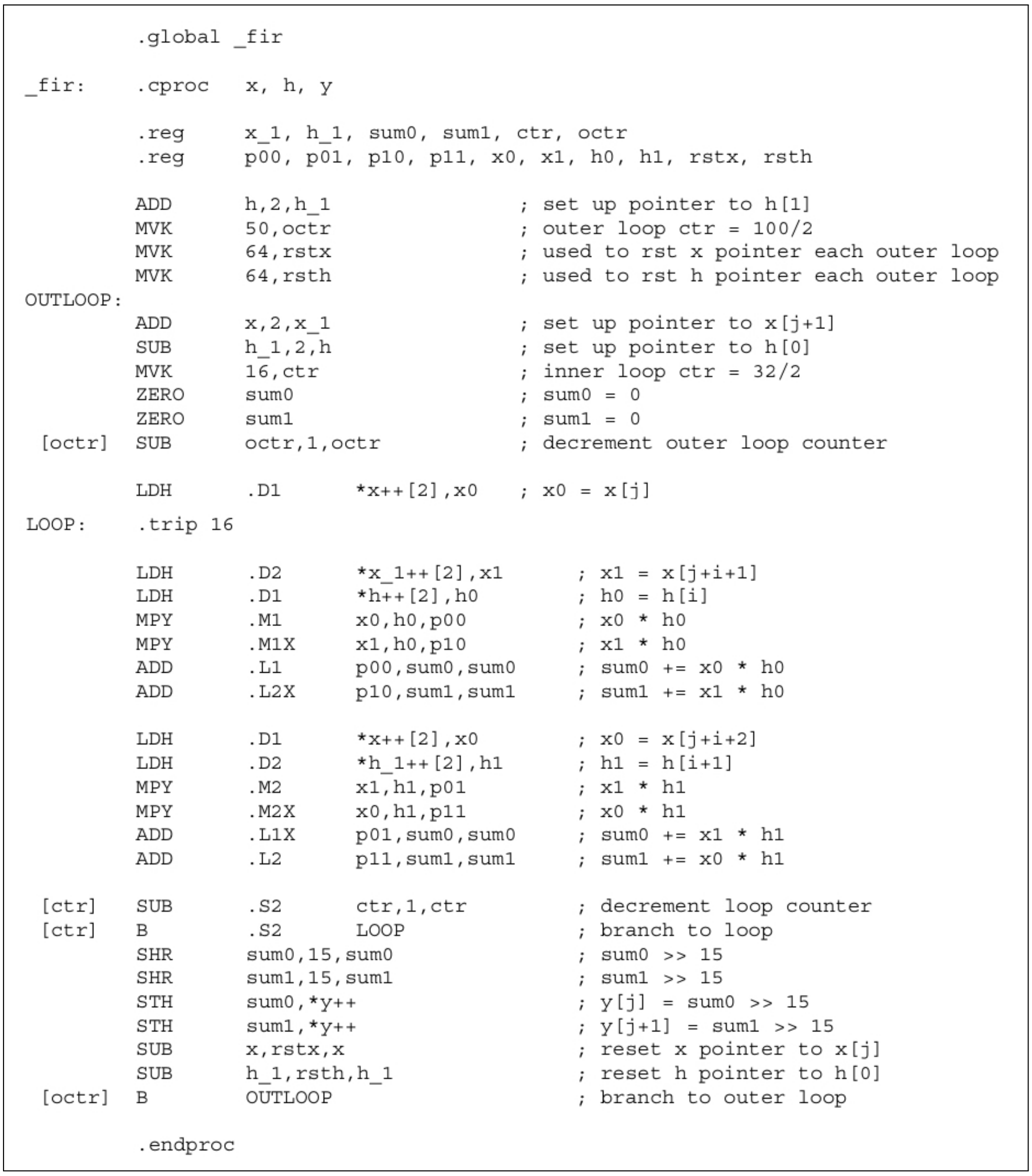
5.4 Software EDAC

Software error detection and correction was realised by implementing a SECDEC Hamming Code (39, 32). As mentioned in Section 3.4.3.3, this code is able to detect up to two error and to correct one. Since the overhead grows linearly with the amount of data to be protected, the performance overhead can be calculated easily. The code was implemented in two different versions.

First collecting redundancy information in 32-bit words, because it was believed to be faster as the compiler is optimized for 32-bit data transfers, then in bytes, leaving 1 bit unused, since the code has only 7-bit redundancy. The implementation consists of three separate functions performing the following tasks:

- syndrome information calculation
- checking against syndrome information
- error correction (in case of a correctable error)

The first function takes address and the amount of data as input and stores the parity information, starting at the address passed as third parameter during the function call. Meaning and purpose of the second function obviously is search for bit-flips and, if any, their correction. In case of an SEU, correctable or not, information regarding the fault is written into a dedicated memory section labelled *DMEM_FI_SPACE* for post processing. Table 5.4 and 5.5 show the results in terms of cycles for the two implementations.

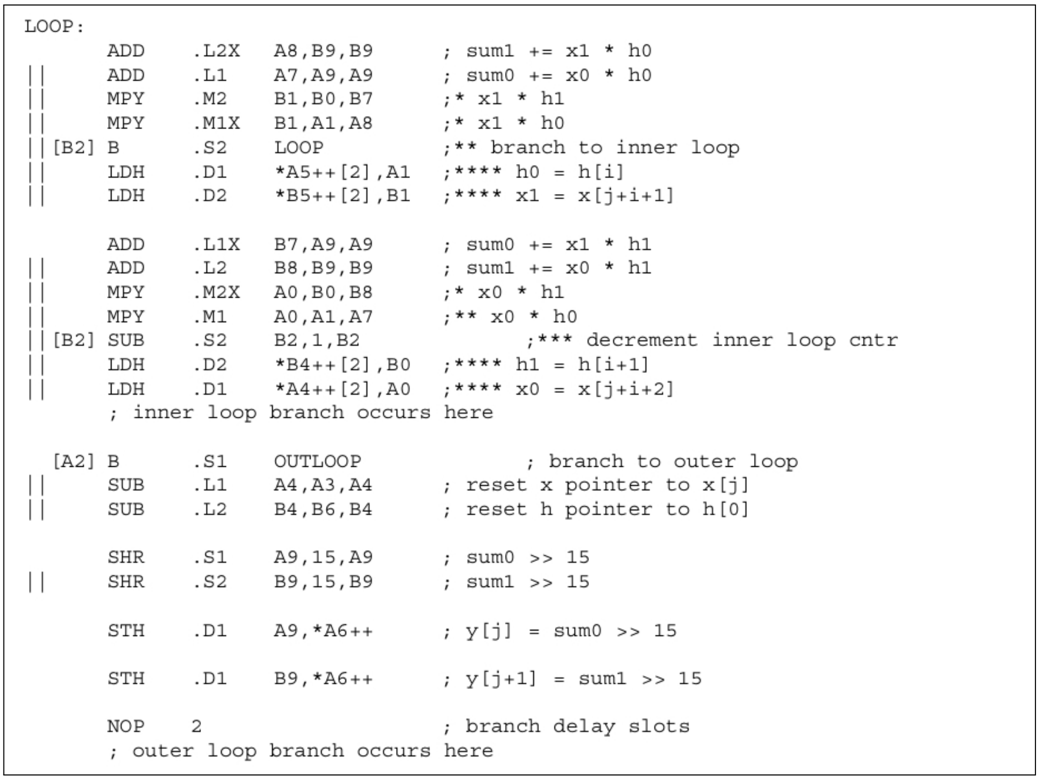


```
        .global _fir

_fir:   .cproc   x, h, y

        .reg     x_1, h_1, sum0, sum1, ctr, octr
        .reg     p00, p01, p10, p11, x0, x1, h0, h1, rstx, rsth

        ADD      h,2,h_1                 ; set up pointer to h[1]
        MVK      50,octr                 ; outer loop ctr = 100/2
        MVK      64,rstx                 ; used to rst x pointer each outer loop
        MVK      64,rsth                 ; used to rst h pointer each outer loop
OUTLOOP:
        ADD      x,2,x_1                 ; set up pointer to x[j+1]
        SUB      h_1,2,h                 ; set up pointer to h[0]
        MVK      16,ctr                  ; inner loop ctr = 32/2
        ZERO     sum0                    ; sum0 = 0
        ZERO     sum1                    ; sum1 = 0
 [octr] SUB      octr,1,octr             ; decrement outer loop counter

        LDH      .D1      *x++[2],x0     ; x0 = x[j]
LOOP:   .trip 16

        LDH      .D2      *x_1++[2],x1      ; x1 = x[j+i+1]
        LDH      .D1      *h++[2],h0        ; h0 = h[i]
        MPY      .M1      x0,h0,p00         ; x0 * h0
        MPY      .M1X     x1,h0,p10         ; x1 * h0
        ADD      .L1      p00,sum0,sum0     ; sum0 += x0 * h0
        ADD      .L2X     p10,sum1,sum1     ; sum1 += x1 * h0

        LDH      .D1      *x++[2],x0        ; x0 = x[j+i+2]
        LDH      .D2      *h_1++[2],h1      ; h1 = h[i+1]
        MPY      .M2      x1,h1,p01         ; x1 * h1
        MPY      .M2X     x0,h1,p11         ; x0 * h1
        ADD      .L1X     p01,sum0,sum0     ; sum0 += x1 * h1
        ADD      .L2      p11,sum1,sum1     ; sum1 += x0 * h1

 [ctr]  SUB      .S2      ctr,1,ctr         ; decrement loop counter
 [ctr]  B        .S2      LOOP              ; branch to loop
        SHR      sum0,15,sum0               ; sum0 >> 15
        SHR      sum1,15,sum1               ; sum1 >> 15
        STH      sum0,*y++                  ; y[j] = sum0 >> 15
        STH      sum1,*y++                  ; y[j+1] = sum1 >> 15
        SUB      x,rstx,x                   ; reset x pointer to x[j]
        SUB      h_1,rsth,h_1               ; reset h pointer to h[0]
 [octr] B        OUTLOOP                    ; branch to outer loop

        .endproc
```

Figure 5.1: Equivalent FIR Filter - Linear Assembly Code [26]

Function	total cycles	cyles per 32-bit	costs per 32-bit word
Calc Syndrome	8748	17.1	18
Check and Correct	17956	35.1	36
Conditional Check	21028	41.9	42

Table 5.4: Software EDAC - 512 x 32-bit with 32-bit redundancy



```
LOOP:
        ADD     .L2X    A8,B9,B9        ; sum1 += x1 * h0
||      ADD     .L1     A7,A9,A9        ; sum0 += x0 * h0
||      MPY     .M2     B1,B0,B7        ;* x1 * h1
||      MPY     .M1X    B1,A1,A8        ;* x1 * h0
||[B2]  B       .S2     LOOP            ;** branch to inner loop
||      LDH     .D1     *A5++[2],A1     ;**** h0 = h[i]
||      LDH     .D2     *B5++[2],B1     ;**** x1 = x[j+i+1]

        ADD     .L1X    B7,A9,A9        ; sum0 += x1 * h1
||      ADD     .L2     B8,B9,B9        ; sum1 += x0 * h1
||      MPY     .M2X    A0,B0,B8        ;* x0 * h1
||      MPY     .M1     A0,A1,A7        ;** x0 * h0
||[B2]  SUB     .S2     B2,1,B2                 ;*** decrement inner loop cntr
||      LDH     .D2     *B4++[2],B0     ;**** h1 = h[i+1]
||      LDH     .D1     *A4++[2],A0     ;**** x0 = x[j+i+2]
        ; inner loop branch occurs here

  [A2]  B       .S1     OUTLOOP                 ; branch to outer loop
||      SUB     .L1     A4,A3,A4        ; reset x pointer to x[j]
||      SUB     .L2     B4,B6,B4        ; reset h pointer to h[0]

        SHR     .S1     A9,15,A9        ; sum0 >> 15
||      SHR     .S2     B9,15,B9        ; sum1 >> 15

        STH     .D1     A9,*A6++        ; y[j] = sum0 >> 15

        STH     .D1     B9,*A6++        ; y[j+1] = sum1 >> 15

        NOP     2                       ; branch delay slots
        ; outer loop branch occurs here
```

Figure 5.2: Equivalent FIR Filter - Hand Assembly Code [26]

Function	total cycles	cyles per 32-bit	costs per 32-bit word
Calc Syndrome	8776	17.1	18
Check and Correct	17956	35.1	36
Conditional Check	21540	42.7	43

Table 5.5: Software EDAC - 512 x 32-bit with 8-bit redundancy

Based on the information provided in Table 5.5 it is obvious that protecting the whole memory may not be practicable, but, of course, this depends on the application and on the timing. Before considering software EDAC routines as ultimate solution, it must be carefully checked which amount of data needs to be protected and how much time can be alloted to this function during algorithm execution.

For classical FIR-filter, for example, it might be appropriate to protect the filter coefficients. Of course, constant coefficients can be reloaded during idle phases, but practical algorithms used in space instruments are adaptive, requiring on-line coefficient calculation. Calibration of non-linearities, caused by variable environmental conditions, is just one example.

The minimum number of clock cycles c required to perform N-tap FIR-filtering on n input samples is

$$c = \frac{N \cdot n}{2} \tag{5.1}$$

because the SMV 320C6701 can perform 2 multiply-and-accumulates (MACs) in a single cycle. Equation (5.1) is a lower bound because it does not take into account any overhead for branching and stack operations and it is valid, if both input samples and coefficients are real-valued and all calcualtions are performed in single-precision floating point arithmetics.

Table 5.5 may be used to decide upon the coefficient check-and-correct rate in relation to the introduced overhead. Per input data sample 256 cycles are required to perform filtering for each clock cycle associated with a signal-sample. 48272 cycles are required for applying EDAC to 512 coefficients, according to Table 5.5. Assuming that an overhead of 100% - halving the throughput - were acceptable, the coefficients could be updated after the processing of every $n = 188$ signal samples. In this example, input- and output data are not protected at all.

Also, the EDAC function is not protected in the example above. This is possible by, for example, control-flow checks, but will complicate software development and maintenance and introduce further overhead.

In contrast to above example, EDAC code resides in the program memory, which implies that overhead created by transferring data from program memory to into the data memory, where it can be accessed for inspection, must be taken into account. This issue is dealt with in the next section.

5.5 Mirror Checking

Mirror Checking or data diversity is one of the simplest methods to check consistency between two sets of data. Two identical or complementary sources are compared to detect differences. Apart from the fact that this technique doubles the processing system's memory requirements, program-memory access for checking purposes can be difficult in a modified Harvard-architecture as used in TI-DSPs. As mentioned earlier this can only be done by a-priori copying via DMA. Although this method allows for detecting and overwriting multiple errors within the frame of a single 32-bit read/write access, the method may be disturbed by the fact that SEEs may affect both original and mirrored data, which requires additional measures to be taken.

In case a mismatch between two sets of information is detected one of the following solutions may be used to enhance the global state view:

- checking both sets against a checksum like: parity, CRC, ...
- checking both sets against redundant information like EDAC, which also allows for error correction.
- checking both sets against a third set, which is protected and therefore slower in terms of access time

The resulting corrective action clearly depends on the situation. For example, if a parity mismatch is detected on one set it is recommended to replace both sets since the check between two sets cannot be faster than a reload via DMA. The only reason for checking rather than periodically overwriting data is the interest in actually gaining knowledge of error events so that they can be thoroughly reported and that potentially affected data can be marked invalid.

The approach implemented for evaluation purposes does not just copy the memory contents from one location to another, but for protection the copied version is modified by inverting each bit of every data word. Consequently, the XORing on 32-bit words can be used on both data-sets to detect anomalies. The reason why XORing is superior to comparing instructions with respect to thoughtful resource utilization, because XORing is available in four different functional units while only two calculation units support the compare function. It is not necessary to invert every word to perform a check using *XOR* instructions, but it has been shown as helpful to distinguish the two sets. In case of a mismatch and depending on the size of the set to be checked, it is up to the programmer to decide how to continue. An example wout be to simply trigger a replace routine. The programmer has the choice to replace sets upon the first uncorrectable mismatch to minimize overhead or to trigger warm start routine to ensure a clean environment.

The proposed method allows for detecting and overwriting multiple errors, but has the disadvantage that at least one set needs to be replaced. If this method is applied to the protection of larger memory areas, the associated overhead can be minimized, if the re-load process is triggered by the first error event.

Mirror checking and software EDAC have in common that their overhead grows linearly with the amount of data to protect. In great contrast to EDAC resorting to partial correction to reduce the effort associated with the protective measure is not possible. Reloading of mirrored sections accomplished by means of DMA. As mentioned earlier, the protection of program code is only possible after a-priori transfer. Because the program memory is of the same size as the data memory, a complete check is only possible piece by piece. Alternatively, program code could in principle be executed directly from external memory which could be well protected. However, this is practically unreasonable due to the considerable overhead introduced by the EMIF. However, from Table 5.6, the overhead introduced by transferring data via DMA, the EMIF disadvantage is not visible, since the average latency is very similar to the latency experienced for DMA to internal data memory.

The size column presented in Table 5.6 represents the amount of Bytes transferred via 32-bit block transfers. The quoted number of cycles per Byte is the average calculated from the total number of cycles. The costs quotes the number of clock cycles to be needed for each 32-bit word.

The Table 5.6 also reveals an important behaviour of this platform. Comparison of the latencies with and without involvement of the program memory brings forth that the experienced access bandwidth to the program memory is lower than for other memories. This behaviour is due to the fact that the program access controller fetches a VLIW every cycle. If the DMA controller is configured to attain CPU priority mode, it is unable to service the program memory and

Memory	Size (Bytes)	total cycles	cyles per Byte	costs per 32-bit word
PRAM → DRAM	512	796	1.55	7
	1024	1440	1.40	6
	2048	2716	1.32	6
PRAM → EMIF:SDRAM	1024	1460	1.42	6
DRAM → EMIF:SDRAM	1024	696	0.68	3
EMIF:SDRAM → DRAM	1024	708	0.69	3

Table 5.6: DMA Transfer Latency

therefore, stalls. TI calls this *Memory hit effect*. Two modules want to access the same location and since one of the modules has to give way, this severely effects transfer time.

The data memory is subdivided into two equally sized memory data banks. Because of this bank concept, it is possible to perform two 64-bit loads into bank-0 and a 64-bit DMA transfer into bank-1 at the same time. This knowledge is essential and needs to be taken into account when planning software fault tolerance and performance overhead.

Memory	Size (Bytes)	total cycles	cyles each Byte	costs per 32-bit word
DRAM	512	172	0.36	2
	1024	304	0.29	2
EMIF:SDRAM	512	3888	7.59	31
EMIF:SDRAM	1024	7744	7.56	31

Table 5.7: Memory-type dependent latency associated with data inversion

Data transfer is followed by data inversion, which is not effortless and adds to the overhead. As Table 5.7 shows, the average effort for inverting data inside internal data memory is 2 cycles, while it amounts to 32 cycles if working with external. EMIF encounters huge latency every time an access is scheduled as CPU load/store instruction. Due to the poor data inversion performance when working directly from external memory the preferred memory destination for this technique is obviously internal memory. Table 5.8 provides a feeling for the effort needed to compare two mirrors residing in the DSP's internal data memory.

Memory	Size (Bytes)	total cycles	cyles each Byte	costs per 32-bit word
DRAM	512	1172	2.28	10
DRAM	1024	2296	2.24	10

Table 5.8: Internal data comparison latency

5.5.1 Comparison of Software EDAC and Mirror Checking

Software EDAC and Mirror Checking may at least in principle be used for the same purpose. Based upon the latency tables presented in previous sections a comparison of the two techniques can be attempted by accumulating the latencies associated with all tasks necessary for error correction. Initial overheads, as compared in Table 5.9, are associated with the establishment of redundancy, viz. calculation and storage of check bits for EDAC and the establishment of the mirror for mirror checking.

Method	Cycle overhead	Data overhead
Mirror checking	2716 for transferring + 608 for inverting	3×512 words
Software EDAC	8776 for calculating syndrome bits	128 words

Table 5.9: Initial overheads for protecting 512 filter taps

Mirror checking requires 2.716 cycles for transferring data from “block-a” to “block-b”, if this is not already done by multiple linking in case of static coefficients, plus 608 cycles for inverting the copied set. Compared to EDAC, this method is much faster although it has to be stated that in case of constant data it is also possible to generate and store the check bits during compile time.
Since both methods can tolerate different types of errors, the single bit flip case is assumed for calculating the latencies associated with error detection and correction.

Method	Cycle overhead for detection	Cycle overhead for correction
Mirror checking	1024 cycles for comparing	1416 for reloading both sets
Software EDAC	17956 for checking and correcting	none

Table 5.10: Cycle overhead for protecting 512 filter taps against single bit failures

Compared to EDAC the approach using mirror checking seems to be more efficient if the data overhead can be neglected. However, mirror checking is inferior to EDAC if the data to be protected is changing frequently. In addition, mirror checking inevitably requires DMA, which is not the case for EDAC. The internal DMA controller is only able to service one DMA channel per clock cycle. Consequently, periodic DMA transfer creates a bottleneck slowing down the system. Frequently, DSP is implemented as streaming application fetching data by means of DMA, processing and streaming the data, which may be spoilt by lacking DMA availability.

Preferably, both methods should be combined. Mirror checking can be used to protect constant values like coefficients, while software EDAC can be used to protect processing results which need to be held inside the internal memory for a longer period of time.

5.6 Control Flow Checking by Software Signatures

This approach was already introduced in section 3.5.3. Having selected a particular platform, some open questions may now be answered. The question what is best for which case will be answered later. Also open is the question how the control graph gets mapped to code and why it is still necessary to use a watchdog timer. To ease the implementation and the evaluation interruptible code is not considered. Before continuing, it is necessary to analyze how the theoretical concept can be mapped to the instruction set. As mentioned often before the VLIW architecture and the function unit delay averts limiting the impact of individual instruction to one cycle. This complicates the mapping of the methods to hardware instructions. The following questions will be clarified in this section.

- Protection of loop kernels
- Intra or inter procedure checking or both?
- How to deal with delay slots?
- How to deal with endless loops?
- What can be done in case of a CFE?
- At which level should CFCSS be implemented (trade-off between performance and complexity?)
- How can CFCSS be optimized?

Loop kernel protection is not straight-forward. Due to the nature of signal processing algorithms, data is mostly processed in loops, simplifying the code and increasing code readability. Assuming DMA tasks like data streaming, the underlying loops consume the greatest part of the overall execution time. Consequently, optimization of loop performance is key in a DSP system and several solutions have been proposed. The one straightforwardly provided by the selected DSP is *instruction level parallelism* facilitated by means of VLIW instruction handling, enabling the computation of multiple statements during a single clock cycle and, thus, boosting computational efficiency – not only during loop executions.

The problem is to introduce SWIFT functions for protecting these kernels without interfering with the optimizations built into the processing platform. Since a branch delay slot is five cycles long, the minimum number of instructions inside the loop kernel is four. This can be confirmed by inspecting optimized example code provided by Texas Instruments within a frame of assembly-optimized general-purpose signal processing routines within the *TMS320C67X DSP Library*[3]. These library functions can be used as a lower-bound on execution performance since they are optimized with respect to execution time for the given platform.

Since most computations are done within a loop kernel it is clear that during these cycles most or all of the available functional units are utilized. If there are not enough free and appropriate “slots” inside a VLIW execute packet it is not possible to simply add additional instructions.

---

[3] `http://www.ti.com/tool/sprc121`

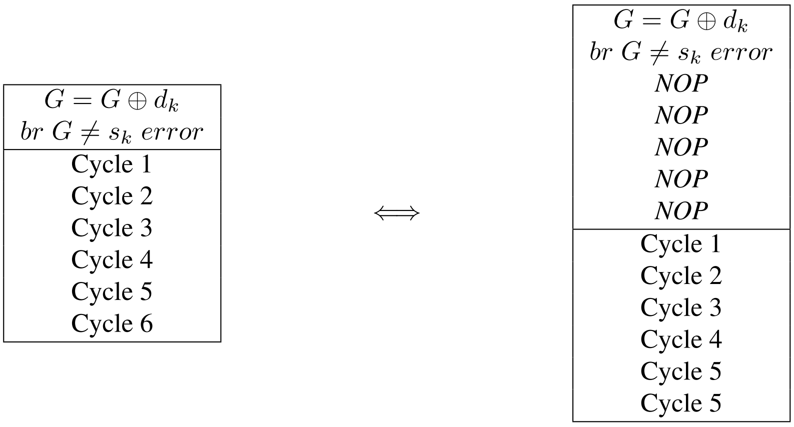


Figure 5.3: introductory checking instruction; theoretical implementaion (left) vs. practical implementation on a pipelined processor with a branch delay slot of $\delta_{br} = 5$ (right).

Therefore, the only choice is to increase the loop size and, therefore, complicate both scheduling and partitioning.

According to the *basic block* definition forming part of the CFCSS terminology, stating that branching instructions must not occur in a basic block with the exception of its last instruction, it is impossible to simply separate between branching- and standard-instructions without performance loss. Consequently, a clean implementation would have to utilize all functional units with *NOP* instructions during a branch delay. In the context of highly optimized loop kernels this concept would severly cut on the computational performance. For optimal performance up to eight functional units need to be utilized during each clock cycle. This leads to two problems, viz. branch delay slot handling and functional unit delay handling.

5.6.1 Branch Delay Slot Handling

CFCSS assumes that instructions are executed sequentially and a strict cut between distinct application instructions and instructions dedicated to control-flow checking is in principle possible. However, as the example depicted in Figure 5.3 reveals, strict separation of application and error protection introduces considerable overhead and is, thus, impractical due to the absence of single-cycle branch instructions in pipelined processors.

If each basic block starts with a check sequence which is supposed to be strictly separated from application instructions and if the processor architecture imposes on the one hand a branch delay of δ_{br} cycles and offers N_u parallel instruction units on the other, the number of wasted instructions is $ins_{lost} = \delta_{br} \cdot N_u$. For the SMV 320C6701 processor $\delta_{br} = 5$ cycles and up to $N_u = 8$ instructions could be executed in parallel, resulting in a loss of 40 instructions.

To take advantage of the instruction level parallelism ofered by the DSP, the instructions necessary for control flow checking must be scheduled in parallel with application instructions. On the other hand, it must be guaranteed that all instructions in support of control flow checking are executed in the way and the order intended by the programmer, which limits the possibility to use higher level languages and optimization tools. As explained in section 5.3, DSP software for the SMV 320C6701 can be developed on (hand) assembly, linear-assembly or C-code level. To ensure a simpler modification of existing algorithms a macro library written in linear assembly was developed. In case that no automated tool support is available the effort needed for manually rescheduling and repartitioning of a code section increases depending on the programming level. The costs would involve expensive re-coding since the algorithm is assumed to be optimal and therefore written in hand-assembly. In case of linear assembly, an application developer could insert macro instructions that fulfil the operations necessary. Because the assembly optimizer does the scheduling and partitioning automatically code modifications tend to be easier. For better understanding, refer to Figure 5.1 and Figure 5.2.

5.6.2 Loop Kernels

To utilize all functional units of the DSP in the best possible way, software pipelining techniques are used to schedule instructions of a loop so that multiple iterations of the loop can execute in parallel. This technique, illustrated in Figure 5.4 is often called loop-unrolling and it is essential if high throughput is of interest. However, as the most efficient implementations are based on 4 cycle loop kernels, the major computation is done during the inner loop of the algorithm (loop kernel). Since branching introduces a delay of 5 cycles, multiple branches need to be placed inside the loop kernel, creating three problems to be tackled.

As explained earlier, the theory of control flow checking assumes a strict cut between application and control flow checking code. The implications of such a strict separation, discussed in detail for branch delay slot handling in section 5.6.1, are far more severe for loop kernels due to the fact that the latter contain multiple branch instructions. Additionally, it has to be taken into account that there are at least 3 more instructions necessary to calculate the checksum. In total, an overhead of at least 75% for all control-flow checking instructions would have to be accepted, provided appropriate loop-scheduling can be at all accomplished by the compiler. More detailed overhead estimations will be provided later.

The second problem is related to the actual placement of control flow instructions within the program code. Loop kernels are written in a way that as many functional units as possible are used concurrently, which reduces the options for adding additional instructions in parallel without re-partitioning and re-scheduling.

The third problem is related to the register usage. During execution of kernel routines it is often the case that all available registers are engaged, forcing the compiler to allocate stack space for temporarily storing register information outside the register file. The resulting overhead is large. Assuming that at least 2 more registers are needed to store the global signature register GSR and the runtime signature register D, 2 load and 2 store instructions[4] have to be added to

---

[4] requires that the pointer to this stack address is stored inside a register

the overhead budget.

The loop pattern show in Figure 5.4 can be found in almost every signal processing algorithm so that finding a solution for the proper implementation of loop kernels, as presented in the next section, is of paramount importance. A way to handle this situation will be presented in the next section.

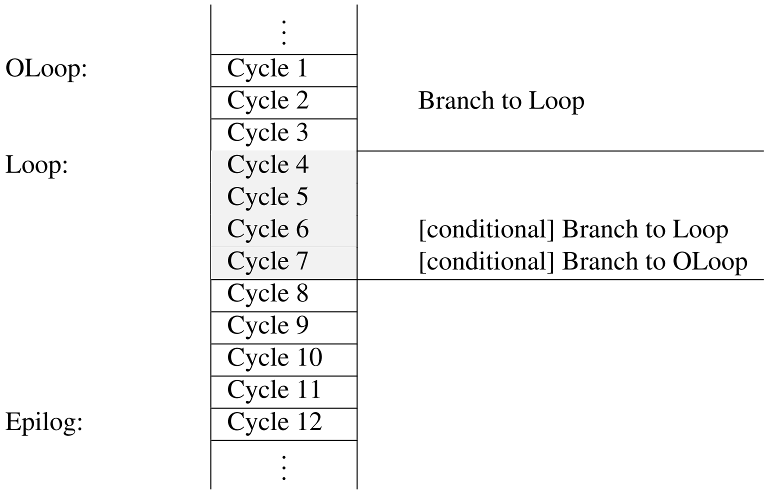


Figure 5.4: Branch pipeline effects

5.6.3 CFCSS Software Coding

The present section aims at investigating how efficient two CFCSS checking algorithms proposed in [43] can be implemented with the SMV 320C6701 DSP. The algorithms, referred to as A and B, are described by means of the terminology explained by Table 3.5, where N denotes the total number of nodes in the program.

5.6.3.1 Algorithm A

Algorithm A assigns signatures and check instructions to each node in a program.

1. Identify all basic blocks, build program flow-graph and number all nodes in the program flow-graph
2. Assign s_i to v_i in which $s_i \neq s_j$ if $i \neq j,\ i, j = 1, 2, \ldots N$
3. For each v_j, $j = 1, 2, \ldots N$
    - 3.1 For v_j, whose $pred(v_j)$ is only one node v_i, then $d_j = s_i \oplus s_j$
    - 3.2 For v_j, whore $pred(v_j)$ is a set of nodes $v_{i,1}, v_{i,2}, \ldots, v_{i,M}$ - therefore, v_j is a branch-fan-in node - the signature difference is determined by one of the nodes (picked arbitrarily) as $d_j = s_{i,1} \oplus s_j$. For $v_{i,m},\ m = 1, 2, \ldots M$, insert an instruction $D_{i,m} = s_{i,1} \oplus s_{i,m}$ into $v_{i,m}$ This instruction should be located after the "br $(G \neq s_j)$ error" instruction in $v_{i,m}$.
    - 3.3 Insert instruction $G = G \oplus d_j$ at the beginning of v_j
    - 3.4 If v_j is a branch-fan-in node, then insert an instruction $G = G \oplus D$ after $G = G \oplus d_j$ in node v_j
    - 3.5 Insert an instruction "br $(G \neq s_j)$ error" after the instructions placed in steps 3.3 or 3.4

Every node needs to fulfill the following synthetic instructions:

1. $d_j = s_i \oplus s_j$
2. $G = G \oplus d_j$
3. "br $(G \neq s_j)$ error"

Since not every node is a branch-fan-in node this is valid for each node. The first operation calculates the signature difference between two different nodes. Since the control flow is known this may be implemented at compile time, saving this instruction at run time. The second instruction requires two clock cycles. First, the signature difference must be loaded as a constant value into a general purpose register. Then an *XOR* instruction using the global signature register and the signature difference is executed. The last two cycles contain the check sequence. First the global signature register is compared with the node signature to determine its validity. In case of a mismatch a branch to an error handler is executed. The signature difference d_j is loaded as constant value into a register to minimize the load delay, which amounted to four clock cycles, if the value were retrieved from memory. The *XOR* operation does not have an immediate field so that the fastest solution is to load it as a constant using a Move-Signed-Constant-into-Register (MVK) instruction. Due to the 16-bit address field of the opcode of this instruction 65536 nodes can be addressed. Assuming that the branch instruction delay slots are negligible, the minimum cycle count associated with the processing of one node is 4. If the node is also a branch-fan-in node two additional instructions are necessary, resulting in a total cycle count of 6.

It can be concluded that adding 4 cycles of overhead to a 4-cycle internal loop is not efficient at all.

According to algorithm A every node contains one (synthetic) instruction responsible for comparing the run-time signature with the signature of the node. In the actual implementation this synthetic instruction requires two code instructions. If immediate error detection were not necessary, signature comparison could be postponed. Once an illegal branch is taken run-time signature and node signature will be different. All nodes have distinct node signatures so that the divergence will sustain. Therefore, signature comparison does not have to be accomplished for all but only for selected nodes. This leads to algorithm B which reduces the cycle delay to 2 cycles at nodes without check procedure.

5.6.3.2 Algorithm B

1. Identify all basic blocks, build program flow-graph, and number all nodes in the program flow-graph
2. Assign an s_i to v_i, in which $s_i \neq s_j$ if $i \neq j,\ i, j = 1, 2, \dots N$
3. For each v_j, $j = 1, 2, \dots N$
   - 3.1 For v_j, whose $pred(v_j)$ is only one node v_i, then $d_j = s_i \oplus s_j$
   - 3.2 For v_j, whore $pred(v_j)$ is a set of nodes $v_{i,1}, v_{i,2}, \dots, v_{i,M}$ - therefore, v_j is a branch-fan-in node - the signature difference is determined by one of the nodes (picked arbitrarily) as $d_j = s_{i,1} \oplus s_j$. For $v_{i,m},\ m = 1, 2, \dots M$, insert an instruction $D_{i,m} = s_{i,1} \oplus s_{i,m}$ into $v_{i,m}$ This instruction should be located after the "br $(G \neq s_j)$ error" instruction in $v_{i,m}$.
   - 3.3 Insert instruction $G = G \oplus d_j$ at the beginning of v_j
   - 3.4 If v_j is a branch-fan-in node, then insert an instruction $G = G \oplus D$ after $G = G \oplus d_j$ in node v_j
   - 3.5 Insert an instruction "br $(G \neq s_j)$ error" only into v_j where to comparison between the run-time signature $G = G_i$ and the signature s_i is wanted

The difference between algorithm A and B is mostly due to step 3.5. Instead of comparing the signatures for every node, algorithm B limits this task at nodes where immediate control-flow error detection is absolutely necessary. However, this also increases the fault detection delay.

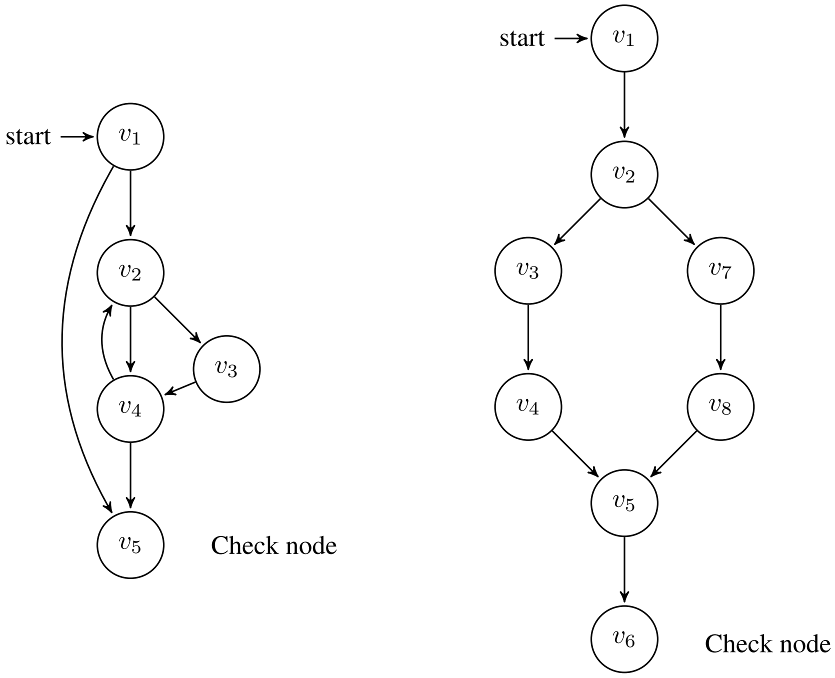


Figure 5.5: Two examples of algorithm B, only two nodes v_5 (left) and v_6 (right) are performing the check sequence

5.6.4 Infinite loops

Infinite loops, regardless if wanted or not desired, cannot be detected using the before mentioned techniques. Endless transition between nodes $v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_2 \rightarrow v_3 \rightarrow \ldots$, as shown in the left example of Figure 5.5, could be the result of an SEU inside the loop counter. To handle infinite loops additional information must be embedded into the application code. One approach presented in [54], originally intended to make use of hardware assistance could be purely implemented in software by exploiting the instruction level parallelism supported by the hardware.

Checksum	
Cycle 1	→ susbtract
Cycle 2	→ susbtract
⋮	
Cycle n	→ susbtract
Zero Check	

Figure 5.6: Checksum based control-flow checking

The basic idea is that the number of clock cycles required to execute a block consiting of several nodes is known so that it can be loaded as a type of checksum into a register before the first clock cycle of the block gets executed. Every cycle the checksum is decremented using a parallel *SUB* instruction so that the checksum should become zero at the end of the block. Control-flow error detection can then be based on checking if this condition is satisfied. The disadvantage of this technique resides in implementation constraints. Since the method depends on exact knowledge of the program execution schedule it can only be implemented in hand assembly. In addition, considerable overhead is created due to the necessity of

- one *MKV* instruction,
- one *SUB* instruction in parallel with the application code and
- one branch instruction, directing towards an error handler (assuming that the checksum is located inside a conditional register, which would allow a branch without a previous compare instruction)

Resource requirements for this approach are a free functional unit and a free conditional register throughout the entire block processing time. Since only 5 out of 32 registers are conditional and these registers are essential for conditional branching used during loop execution and other conditional statements, e.g. if-else, free register availability is implausible.

Therefore, a transparent handling using a watchdog timer, still seems to be the best solution. However, this requires fast handshaking between the CPU and the external watchdog.

5.6.5 Intra- versus Inter-Procedure Checking

Until now, the basic block definition was used to represent parts of the code at instruction or, in our case, at cycle level. As seen before, the overhead introduced at this level is high. Some performance improvement can be expected, if the basic block definition is used to model function call rather than instructions. Due to the coarser granularity the introduced overhead is reduced by decreasing the instructions-to-nodes ratio (INR). Integrating the control-flow checking into DSP-algorithms is simplified to the extent that programming in C becomes possible. Besides the loss of granularity, the main disadvantage is the increased fault-detection delay. The severity of this clearly depends on the application and how important it is to detect a fault as early as possible. Just imagine the output generated by a processing stage of a spaceborne instrument. In case the internal data processing for a specific block has finished delayed detection would cause a serious amount of additional communication between different parts of the instrument control units. Such high-level behaviour had to be incorporated in the instrument controller in form of an error-handling routine which in turn would require additional testing on instrument level. This example shows that cost considerably increases, if problems are deferred from a lower to a higher level in a system due to the fact that test costs grow with the number of system components involved.

To choose the right method one should consider that most signal processing applications have to deal with huge amounts of data and that this data is always processed in the same way.

	Inter-procedure	Intra-procedure	Intra/Inter-procedure
Granularity	low	high	medium
Overhead	low	high	medium
Availability	low	high	high
Efficiency	high	low	high
Fault detection latency	high	low	low
Complexity	low	high	medium
Application specific	no	yes	yes

Table 5.11: Intra procedure vs. Inter procedure checking

If most of the time is spent with the processing of a specific function, it is best to consider intra procedure checking. On the other hand, a more complex state-machine processing the input samples differently at each stage can be optimized by means of inter-procedure checking. In conclusion we can state:

The overall time spent within a specific region of the code determines the vulnerability to control flow errors during execution of this segment.

As an example, we may assume an FIR-filter processing image-frames of 1 ms duration. In case the filter processing takes about $600\mu s$ and the processing is done without interruption, this implies that 60% of the computation time are spent repeatedly executing 640 Bytes of code[5]. This illustrates why it is important to clearly distinguish between intra and inter procedure checking. Apparently, mixing both approaches leads to an interesting and important extension.

Table 5.11 compares the costs and benefits of the three approaches. Inter-procedure checking benefits from low overhead, low complexity and the fact that it is application independent because existing signal processing routines do not need modifications. Nevertheless, although inter-procedure checking enables algorithm independent mapping it remains application with respect to the systematic effects resulting from control-flow error detection delays.

Intra-procedure checking on the other hand benefits from finer granularity, resulting in lower fault detection delay and, thus, higher availability.

If a clear conclusion concerning the overall computational effort of different routines cannot be made or if the selection of just one granularity level is insufficient, the mixed approach can be used. Candidates for the application of the mixed approach are cases were input block size is a-priori not known so that simple decisions about the execution time cannot be taken. Other examples go beyond just protecting the signal processing routines, in particular, if it is essential that the signal processing procedures are called in the correct sequence. The combination of both approaches reveals a relationship between efficiency and total overhead which offers plenty of

---

[5] DSP_sp_fir_gen taken from the TI DSPLIB

opportunities for customization and optimization.

One property not covered by Table 5.11 is the effort needed for generating the control flow graph. Besides the fact that there may exist some proprietary tools for various platforms, all control flow graphs found in this thesis were constructed by hand. To keep the complexity of the mixed approach low both techniques can be implemented in a self-contained manner, just sharing a common error routine. This avoids intransparent and non-serviceable control flow graphs. It is of paramount importance that intermediate signature results or register states are not overwritten by other control flow functions.

5.6.6 The Error Case

The handling of detected control flow errors is done by calling a common error handling function. This function is the same for inter and intra procedure checking. Control flow checking does not store any information concerning older states or events which makes it impossible to determine the transition causing the error and to recover without loss of information. Because the source of the experienced CFE could be in any of three different regions viz. program memory, data memory or core registers, the only solution is to reload all memory sections, reinitialise the CPU and begin with a clean boot preceeded by a small test routine. Since this is the only possible solution without adding any checkpoint mechanism, the time required for restarting directly impacts system availability. The faster this procedure can be exercised the better the resulting availability.

Error handling requires calling the corrective function within a potentially unstable stack environment. Unstable because, the control flow error could have been triggered by a previously corrupted stack. Under such conditions safe function calls are not possible. In order to be able to call functions, regardless if coded in C or assembly language, a non-corrupted stack and a correct stack pointer are essential. This would imply a complex error handling routine and is thus far away from efficient. The problem can be overcome by means of a simple trick based on non-maskable interrupts (NMI). During an NMI the program counter address of the last executed instruction gets stored in the non-maskable-interrupt-return-pointer register (NRP). Because the CPU is allowed to write to this register it can be configured to point to an error handling routine. Using the built-in *B NRP*[6] routine the jump can be performed without even resorting to a stack or frame pointer.

5.6.7 Undetectable Faults

Even if considerable effort is taken some faults will never be detected. Considering the internal program memory size of 64KB along with the DSP's addressing scheme a 17 bit program counter implemented in hardware would be a reasonable expectation. However, various tests have shown that all 32 bits can be written and read via JTAG, indicating that the program counter actually features 32 bits. For bit-flips affecting the upper 15 bits of the register the re-

---

[6] which means branch to address stored in NRP

sulting address would exceed the internal memory space. Even worse the program fetch address is incremented every cycle regardless, if data can get fetched or not due to the non-existence of addresses. While overflows of the program counter were not observed during testing a complete halt of the processor was encountered for various addresses. Upon the occurrence of such events, resetting the device was the only solution for recovery.

Another undetectable fault would be a jump within a basic block. Although this is contradicting the definition of a basic block it is nevertheless possible. If a basic block is longer than the pipeline size, a jump within the basic block and therefore, before the next check function will pass undetected with all CFCSS algorithms considered within the frame of the present work. The effect was even observed for an optimized CFCSS implementation based on algorithm B, where not all nodes include a check function.

5.6.8 Macro Library

To enable faster code instrumentation, a macro library written in linear assembly was developed. This library allows modifying existing linear assembly code by introducing various CFCSS macros between existing code parts. The macros are written at a granularity such that the assembly optimizer is not able to remove the statements needed for control flow checking. However, the creation of the control flow graph has still to be done by hand. Depending on the basic block granularity, the selected assembly optimization level and the selected algorithm, this approach allows for inexpensive protection of code execution.

5.6.9 Intra-Procedure Checking Timing Performance Simulations

To evaluate the overhead produced by different approaches different signal processing algorithms were taken and modified to support control flow checking. The outcome of these simulations shall give insight into the efficiency of control flow checking implementations with the selected platform. The following algorithms were modified to incorporate intra-procedure control flow checking:

- single precision matrix multiplication
- single precision matrix transpose
- single precision maximum of vector
- single precision radix-2 FFT with complex input samples

These algorithms were selected because they are essential for almost every signal processing chain and allow due to their structure straightforward reasoning concerning the induced overhead, which may be leveraged to other - more complex - algorithms.

For the aforementioned the following information, relevant with respect to the implementation of control-flow checking, will be provided:

- equivalent C code

Implementation level	Matrix size 3x3	 4x4	 7x7
Hand assembly without FT	180	420	712
Linear assembly without FT	248	856	1516
Linear assembly with FT macros	372	832	1380

Table 5.12: CFCSS clock cycle measurement for the given matrix multiplication algorithm

- constructed control flow graph containing signatures for every node
- signature differences between nodes
- real-time signature, in case of a necessary update

Although the code of each function is shown as C code for the sake of clarity, the implementation for all intra-procedure test cases was done at linear assembly level using the macro library mentioned earlier. Furthermore, the presented control-flow graphs do not directly match the C code but were derived at linear assembly level where most loops were unrolled for performance purposes.

Single Precision Matrix Multiplication

The original matrix multiplication algorithm was taken from the DSPLIB provided by Texas Instruments. The function written in linear assembly was modified by adding the necessary control flow checking instructions using a macro library. Due to the register count loop unrolling was not performed. The resulting control graph consists of three loops where each label node represents a multiple-branch-fan-in node. Since Algorithm B was implemented only the most outer node was selected as check node. Detailed information about the loop structure and the signatures can be found in Appendix A.2.1. The resulting performance measures are shown in Table 5.12

It can be seen that compared to linear assembly the fault tolerant version is at most 1.5 times slower than the non-tolerant one when multiplying two 3x3-matrices. However, when multiplying larger matrices, e.g. 4x4 or 7x7, the fault tolerant version is faster than the non-tolerant one. This effect is related to the assembly optimizer. During the compilation process the assembly optimizer cannot assume a distinct matrix size and, therefore, schedules the algorithm in a more generic manner. Compared to the unprotected hand assembly version without FT the performance loss is about twice the time assuming the given input size.

Single Precision Matrix Transpose

The original matrix transpose algorithm was taken from the DSPLIB provided by Texas Instruments. Like before, the function written in linear assembly was modified by adding the necessary control flow checking instructions using a macro library. Algorithm B was implemented. Due to the small frame window and the resulting smaller register count, loop unrolling was attempted. The resulting graph consists of a single loop. The node containing the loop instructions was implemented as a multiple-branch-fan-in node. Detailed information about the loop structure and the signatures can be found in Appendix A.2.2. The resulting performance measures are shown in Table 5.13.

	Matrix size		
Implementation level	3x3	4x4	7x7
Hand assembly without FT	48	104	132
Linear assembly without FT	60	140	184
Linear assembly with FT macros	152	476	636

Table 5.13: CFCSS clock cycle measurement for the examined matrix transpose algorithm

It can be seen in Table 5.13 that the linear assembly version with fault tolerance is at least 2.5 times slower than the linear assembly version without FT-support. This results from the small number of internal loop operations during each iteration. In case fewer instructions are executed inside each iteration the resulting overhead by inserting additional instructions is huge. Compared to hand assembly without FT the performance loss is between 3 and 4.8, growing along with the matrix size.

Single Precision Maximum Value of a Vector

Like before the “maximum of a vector” algorithm was taken from the DSPLIB provided by Texas Instruments. The function written in linear assembly was modified by adding necessary control flow checking instructions using a macro library. As before, it was possible to unroll the loop to increase performance. Algorithm B was implemented. The node containing the loop instructions was implemented as multiple-branch-fan-in node. Detailed information about the loop structure and the signatures can be found in Appendix A.2.3. The resulting performance measures are shown in Table 5.14.

	Vector length		
Implementation level	39	99	198
Hand assembly without FT	76	104	156
Linear assembly without FT	68	108	176
Linear assembly with FT macros	232	512	976

Table 5.14: CFCSS clock cycle measurement for the “maximum value of vector” algorithm

Table 5.14 shows that the linear assembly version with fault tolereance is at least 3.4 times slower than the unmodified version. This can also be explained by the small number of instructions inside the actual loop. It can also be seen that the induced overhead increases with the input size so that compared to hand assembly without FT the resulting loss factors range from 3 to 6.2 for the considered vector lengths.

Single Precision Floating-Point Radix-2 FFT With Complex Input

The original Fast Fourier Transform algorithm was taken from the DSPLIB provided by Texas Instruments. The function written in linear assembly was modified by adding control flow checking instructions, using a macro library. Due to the register count loop unrolling was applied. The resulting control graph consists of two loops where each node represents a multiple-branch-fan-in node. Algorithm B was implemented and the two loop label nodes were selected as check-nodes. Detailed information about the loop structure and the signatures can be found in Appendix A.2.4. The measured performance is shown in Table 5.15.

	FFT size		
Implementation level	128	256	512
Hand assembly without FT	1852	4160	9284
Linear assembly without FT	6304	14260	30912
Linear assembly with FT macros	13448	30612	68704

Table 5.15: CFCSS clock cycle measurements for the FFT algorithm

Table 5.15 shows that the linear assembly version with fault tolerance is at least twice as slow as the unmodified version. Relative efficiency increases with growing input-vector size. Compared to the hand-assembly version without FT the fault tolerant version is about 7.2 times slower almost regardless of the FFT-length.

5.6.10 Inter-Procedure Checking Timing Performance Simulations

As mentioned earlier, inter-procedure checking is a technique for improving system reliability, modelling function calls as basic blocks of a control-flow graph. This method depends very little on the underlying algorithms and in combination with hand optimized assembly it can be made very efficient. Due to the coarse granularity the execution time between consecutive checks (or function calls) contributes to the fault detection delay. Unlike for Intra-Procedure Checking (IAPC), the temporal behaviour of Inter-Procedure Checking (IEPC) can be estimated without having to perform simulations involving the potential DSP-algorithms. From simulation runs the mean overhead introduced by IEPC was found to be about 52 cycles per checked block. In a first approximation, the fault-detection delay is equal to the interval between two checks. If checks are performed for all basic blocks, the fault-detection time is determined by the execution time of the individual procedures and its maximum corresponds to the procedure with the longest execution time.

5.7 Fault Tolerance Controller

Although SWIFT methods have been shown to be very effective with respect to mitigating the effects of radiation induced upsets, their implementation creates considerable overhead, reducing processing performance and data throughput. In addition, the investigations have brought forth that certain error-types cannot be detected at all, viz.

- infinite loops
- Deadlock
- Consistency of large data sets and/or over longer periods of time
- Program memory mutations not resulting in detectable control-flow errors
- Register modifications not leading to control-flow errors

As reasoned earlier, the problem of infinite loops cannot be solved without external hardware. The same applies to procssor hang-up. For both cases neither error detection nor taking an appropriate action is possible, leaving an externally triggered reset as only solution. Further, memory contents and registers need to be protected, taking into consideration that the implementation of error correction coding is much more efficient in hardware than in software as long as delays and latencies introduced by off-chip access stay within reasonable limits.

As a consequence of the gained experience, the present section introduces a combination of hardware- and software fault tolerance techniques aiming at the minimization of computational overhead in the DSP and at filling the residual fault-coverage gap. For this purpose an external logic function, the so called *Fault Tolerance Controller* (FTC), is introduced. The main tasks of this controller are the monitoring of the DSP's current state, taking care of its configuration and the handling error cases. In a first step, this function is specified independent of a particular processor selection and the anticipated work sharing between FTC and micro-processor is summarized in Table 5.16.

Fault Tolerance Controller Task	Processor Task
Scrubbing	Register Dump into Memory
Interfacing/Data streaming	Watchdog reset
Bootloader	CFC at inter-procedure level
Watchdog	Periodic dump of critical variables
EDAC/Memory Controller	

Table 5.16: Task allocation for a combined HW/SW-Platform approach

To overcome the problem of data inconsistency or program memory mutations the FTC can perform scrubbing. This decreases the probability of uncorrectable multi-bit errors and, therefore, the amount of uncorrectable faults. Further, the FTC can also be used to periodically check the program memory consistency to eliminate the occurrence of CFEs.

The block diagram depicting both function sharing and interaction between FTC and the target DSP is provided in Figure 5.7.

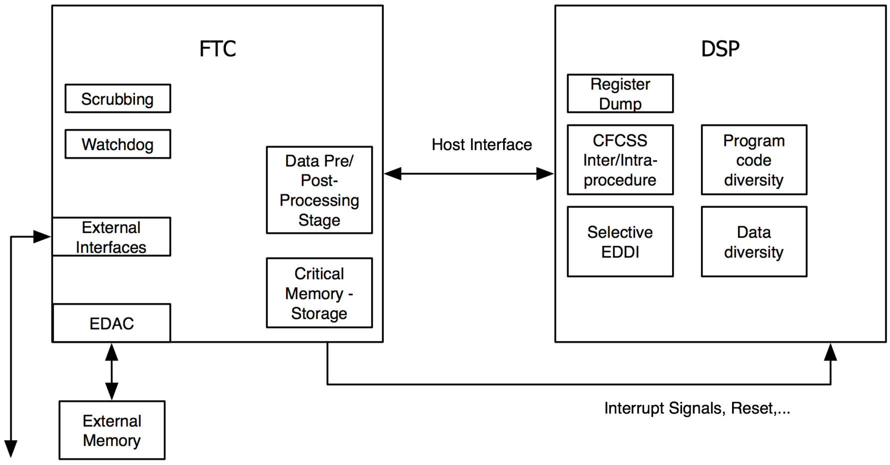


Figure 5.7: Implementation of the combined HW/SW-approach for the SMV 320C6701

TI-DSPs feature a *host processor interface* (HPI), providing access to DSP's internal memory. This allows the FTC to download portions of the internal memory and to compare them with protected copies during run-time. HPI transfers are transparent to the CPU, can be done in parallel to the signal processing and do not create any overhead in the DSP. In case of simultaneous arbitration of program- or data-memory locations by HPI and CPU, the latter is given higher priority and HPI-data output will be delayed for one cycle. In this way, the FTC is able to monitor and reconfigure various parts of the DSP as long as they reside in the DSP's memory map. DSP areas containing valuable information but not directly reachable via the memory map may still be monitored by means of a software workaround using a timer controlled dump routine. Based on information retrieved this way, critical configuration registers, like the interrupt control register or the external memory interface control register, can be re-loaded. However, this can only be done at a limited rate and not continuously so that part of the responsibility is left with the DSP-software, inevitably performing control-flow checking to further reduce the probability of control-flow errors to the desired level.

Any code executing on the DSP is susceptible to SEEs. Therefore, it must be taken into account that routines autonomously providing information to an external observer may fool the FTC. Such a situation could be caused by register faults which can only be covered by means of EDDI. From a practical point of view SEEs affecting program execution can only be detected in case of a manifestation being an error consequence and prior to manifestation in case of unsuccessful comparison or successful pre-checking. To cover these cases a two step approach is proposed. The first step is to prevent errors by means of code replication and successive checking, which is frequently referred to as code replication or code diversity. The second step foresees transformation of critical code into equivalent hardened code, using EDDI methods.

With respect to the practical FTC implementation in space hardware it is important to note that spaceborne instrument data processing often requires pre- and post-processing for assembling and disassembling packetized, multiplexed, staggered or concatenated data - typical fixed-point arithmetics tasks. The same applies to coarse filtering, scaling and look-up-table based non-linearity correction. Typically, high data volumes are processed in that way prior to being transferred to a decimation stage. With the DSP's internal memory being limited, external memory must be used, shared with the DSP and protected by means of feed-through EDAC, which in a state-of-the-art spaceborne FPGA can be implemented with as little as one wait state. Apparently, it is beneficial to combine these inevitable hardware functions with the FTC in a single FPGA.

One may be tempted to compare the FTC approach with other techniques such as *H-Core* and *TTMR*. However, in contrast to H-Core, the FTC is able to directly derive information concerning both availability and functionality of the DSP without having to draw general conclusions on a high level of abstraction. Compared to TTMR, the FTC approach offers the same level of availability with much lower latency and superior efficiency.

CHAPTER 6

Fault Injection

So far, candidate failure mitigation concepts were evaluated by means of reasoning and their suitability in terms of overhead, latencies and delays was assessed by experiments. To obtain an estimate of the overall performance of the proposed techniques, in particular on their practical soft-error mitigation capability, fault injection experiments were performed as well, using the test environment described in Section 6.1. Lacking the possibility to directly inject physical effects like sudden SEEs, the key idea was to inject faults as bit flips into storage elements. This restricts the simulated failure cases to SEUs inside three different regions, viz. program memory, data memory and the register file.

One aim of the experiments was to quantify the effectiveness of the proposed software-only solutions. Another goal was to attain a better understanding of the effects of SEEs acting upon the selected DSP-platform potentially unveiling particularly unfavourable operating conditions and providing indications for the improvement of the platform with respect to SEE-robustness.

The experiments were supposed to answer the following questions:

- How many CFEs are detected using the given fault tolerance?
- How many SEUs create errors in the output data (Data Errors - DEs)?
- How is the relation between CFEs and DEs?
- How many faults do create no measurable effect?
- How often is illegal opcode encountered and what is its outcome?
- How is the distribution of faults locations that were activated?

6.1 Fault Injection Environment

The focus of this injection experiment was split in two parts. First a functionality validation was done for all implemented SWIFT methods and secondly, a statistical measurement of failure probabilities was attempted. Since, the development of such tools can be time consuming, a lightweight but nevertheless performant solution was developed. Based on *Debug Server Scripting* (DSS), which is provided by the Texas Instruments IDE, referred to as Code Composer Studio, a scripting environment for injecting different kind of faults was implemented. DSS allows JAVA-scripting or the use of other third party tools such as Javascript, Phyton or TCL. Because Javascript is the default scripting language as a well understand tool, it was chosen for the present application.

The connection between the Fault-Injection-Toolchain and the DSP was established via JTAG. To gain access to the DSP the target has to be halted. Consequently, a run-time injection of faults is not possible, which is a limitation that is caused by the JTAG interface. Newer interfaces like the Nexus standard would allow run-time access. Unfortunately, a nexus power space DSP is not available at the moment. In principle, there are two possibilities for error insertion under this constraint:

1. *Timing based processing:* After storing the fault list in an internal database structure the target is being started as normal. At a configurable time the target is halted and the first fault of the database is injected. Both original and modified value are written into an XML log file for later analysis. Thereafter the execution of the target is resumed for a given interval and effects caused by the fault are searched for and monitored. This process is repeated until the last fault in the database is encountered.
2. *Breakpoint based processing:* Along with the preparation of the fault database, breakpoint locations for faults to be injected have to be set. The target is started and after halting at the first breakpoint the corresponding fault is retrieved from the database. The actual injection process is the same as in the timing based version. A log file entry is created for further analysis. The approach is, however, limited by the amount of memory available for storing hardware breakpoints in the JTAG debugger. If larger sets of faults need to be processed, breakpoints have to be grouped for piecewise loading into the target. This requires exact knowledge of the control flow. Breakpoint based processing requires a preparatory analysis of the application code and, therefore, closely couples the fault set with the application code. The main advantage, however, is the reproducibility injection experiments.

The method used in the present project is based on timing based processing. With this choice the injection process has become independent of the executed code and it was not necessary to analyze the code before the fault injection campaign. Due to the asynchronous behaviour of this method, the injection process is more random since program execution is fully asynchronous to the IDE. Injection experiments of this type cannot be reproduced due to the asynchronous execution.

Both methods have their benefits. Further, a mixed approach could also be of service.

The following types of faults can be injected:

- Type: Single and double bit-flips
- Program memory:
  - Fixed address
  - Random address
- Data memory:
  - Fixed address
  - Random address
- Registers:
  - Fixed register
  - Random register

Run-time Fault Injector

As discussed before, the hardware environment available for the present activity does not allow for injecting faults without interrupting the DSP. This limitation is not due to the DSP but due to the off-the-shelf evaluation board, which does not allow external memory access other than by the JTAG interface.

For follow-on activities we propose another approach based on additional hardware to provide such features. The capabilities of the host processor interface (HPI), allowing for transparently reading and writing internal memory allows development of a lightweight fault injector which is able to inject faults without causing interference to the program execution. Inducing faults during run time will reduce test time and simplify timing measurements, e.g. error detection latencies, although it has to be noted that realistic and fully representative statistical error simulation is in fact possible with the JTAG based approach, albeit with higher post-processing effort. It is proposed to base such a lightweight fault injector on commercial off-the-shelf USB modules, such as the QuickUSB[1] module, featuring general purpose pins that can be used to create waveforms for direct communication with the HPI interface.

6.2 Experiment Setup

Several experiments were conducted using arbitrary faults as well as different application loads. All experiments share a similar flow as shown in Figure 6.1. All workloads have common that they were all derived from a simple test application containing the same initialisation routines as well as a small state machine switching through its states in a fully deterministic manner. They are, however, different in the way the control flow checking mechanisms are deployed. Depending on the workload this can either be inside a particular function called within one of the states

---

[1] `http://www.bitwisesys.com/`

in case of intra-procedure checking, or every time a transition from one state to another is made in case of inter-procedure checking. Based on the same approach also workloads for a mixed approach, using both intra- and inter-procedure checking, were generated.

Basic Experiment Flow

For each experiment iteration a reset of the device is performed first. Then the object file containing the memory information is loaded into the target. Next, the target is configured to run until an arbitrarily chosen instant in time and to stop thereafter. The execution-time interval was split in two parts. First, enough time to execute all required initialisation routines is provided to guarantee that faults were not injected during this phase.

Upon expiration of the fault-injection delay d the injection process was activated and after completion of the injection process the target execution was continued until time t.

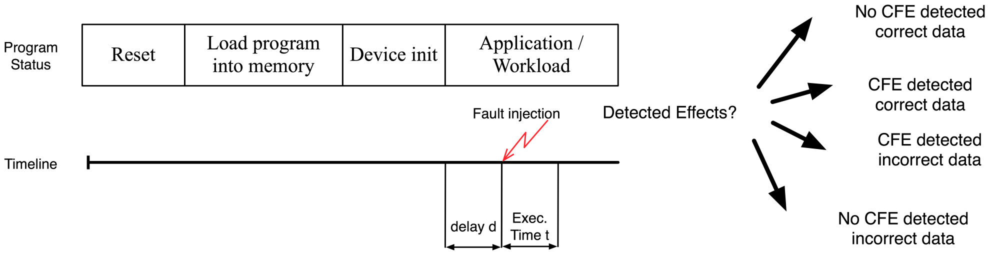


Figure 6.1: Experiment flow

Output Classification

After execution over time t the reaction of the target is observed and written into an XML-log file. In particular, the following issues are evaluated: (i) A check is performed whether the application workload running on the target has produced correct results, and (ii) error-detection information is logged. The available error detection mechanisms are divided into two classes, i.e. internal and external. Internal mechanisms describe the detection capabilities of the implemented error detection mechanism. External methods are used as counter measures to evaluate the functionality and the correctness. Additionally, external methods are used to detect failures resulting from undetectable conditions. Based on this information four different outcomes of an experiment can be distinguished:

1. **No error:** The result of the application is correct. This means that the fault injection has not been effective, which may be due to injection into an unused location or caused by logical masking of the injected fault.
2. **Error detected and correctly handled:** The error was successfully detected by at least one of the available error detection routines. After detection the intended action was taken.

3. **Error detected but incorrectly handled:** The error was successfully detected by at least one of the available error detection routines. After detection incorrect data was still generated.
4. **Error undetected:** The application produces incorrect output but none of the error detection routines was triggered.

The no-error case does not provide any insight in the target's inherent robustness against SEUs. It only shows that either the fault location was inappropriate or that the fault was overwritten shortly after injection. However, due to the data generated by the fault injector, this case can be further analyzed.

Anticipated Results

Two types of faults are expected, viz. errors related to the generated output and control flow variations. Which of these two error-types prevails strongly depends on the fault location. Given that there is a huge amount of general purpose registers and only small portions of executable code mainly register induced faults will be experienced. Since the application code processes within an infinite loop, latent faults inside looping segments are impossible.

Experiment Parameter Space

Based on the above considerations suitable values for fault injection delay d and execution window t have to be determined. In order to inject the fault during application execution, the minimal fault injection delay is set to the amount of time needed for initialisation. Since the application code executes inside an infinite loop the maximum value of d can be chosen arbitrarily. In order to cover a complete application execution cycle, the time t must be twice the time needed for one application cycle. This ensures that faults injected into critical regions are in any case activated.

6.3 Experiment Flow

The basic experiment flow was already mentioned earlier. Now, more details and the reasoning concerning fault effects are provided. This is applicable for all taken experiments. Although it seems that different experiments were made most of their outcome was gathered during a workload-specific experiment.

After initialisation, which is of little importance for the overall analysis and after a delay d the fault is injected. A fault location is randomly selected from the following options:

- Program memory
- Data memory
- Register file

In case of a register induced fault, an element vector of possible registers is built. A particular register is then selected via the element number, using random number generation. Memory locations are selected in the same way. Memory locations required for post-processing purposes (see section 5.2) were excluded from the random-number generation.Exclusively single-bit flip faults were considered.

Every run of each experiment was conducted according to the following plan:

- For each run:
    1. Reset CPU
    2. Load Code into memory
    3. Run until an arbitrary time d
    4. Halt the target
    5. Generate a random SBF containing a random location and a random bit
    6. Resume execution for time t
    7. Read out diagnosis database
    8. Read out the results processed during execution
    9. Write information into XML log-file

The list above shows the experiment flow sufficient for the evaluation of the results delivered by the internal detection mechanisms. However, above steps are insufficient for the evaluation of these mechanisms with respect to their capabilities in terms of counter measures. Therefore, a mixing approach using both, “timing based processing” and “breakpoint based processing” was used. The addition of breakpoints allows the fault-injection environment to act as an “external observer”. This helps to attain a better understanding of fault effects and shall provide an indication which faults could have been detected if external logic would have been present.

According to the detailed experiment flow, the target is stopped after time t. Due to the asynchronous execution of the target it is impossible to determine, if the actual program counter location is correct. However, with the additional breakpoints the virtual external observer is able to detect the existence of faults invisible to the fault-tolerance mechanisms implemented in software. In this way it is possible to determine how often CFEs are not detected.

The analysis of the internal mechanisms contains two steps:

- The dump of the diagnosis database, which is used to track all errors that were detected by internal mechanisms. Every time a CFE is detected the fault is documented in the database and the reset vector is executed.
- The dump of the generated data, which provides insight into the availability of the target. This dump depends on the specific workload executed during the experiment. The application flow is deterministic and the input data is constant during the whole experiment so

that the generated output can be compared easily with a failure free data-set. Since it is possible that after calling the reset vector the falsified output is overwritten, the target is halted immediately after jumping into the initialisation routine. This guarantees that false information is not overwritten during the upcoming application cycle.

6.4 Experimental Results

6.4.1 Experiment 1: Detected CFEs

A first series of fault injections was carried out to investigate how many activated faults, can be detected using the earlier mentioned control flow checking technique and how the detection probability depends on the fault location. For this purpose, different workloads based on the test-applications as shown in Section 5.6.9 were used. The number of injected faults for each workload is denoted by n. Table 6.1 shows the results gained from the experiment.

		Data Integrity		Fault Location			
Workload	Detected CFEs	OK	NOK	Reg	L1D	L1P	Total DEs
Matrix-Multiplication with Inter Check	11	10	1	7	1	3	1.472
Matrix-Multiplication with Intra Check	15	15	0	15	0	0	94
Matrix-Transpose with Intra Check	0	0	0	0	0	0	159
Max of Vector with Intra Check	74	74	0	74	0	0	141
Complex FFT with Intra Check	207	207	0	207	0	0	1.475

Table 6.1: Detected CFEs, $n = 12000$

Before drawing any conclusions it is necessary to explain how the values presented in Table 6.1 were gathered. Each workload derived from the earlier mentioned base program represents a distinct application and can be considered independent since the workloads were executed sequentially, each one being afflicted with n injected faults.

The “Detected CFEs” column represents the amount of CFEs that were detected during execution using internal software mechanisms. In particular, every time CFC routines detected an anomaly it was logged inside the fault database. Since the dump of this database is scheduled at the end of each iteration and since one error cannot be detected twice (stop inside reset vector), it is not necessary to correct errors if still remanent. This value may be different from the number of CFEs that have really occurred during execution due to the possibility of undetected CFEs.

The “Data Integrity” columns represent how often false data was generated in combination with a detected CFE. This can only happen in two cases. First, an incorrect handling of CFEs due to a software error inside the CFC routines or, secondly, due to a late detection of the error. The first case can be caused by a fault injected directly into parts of the CFC code. Because of the non-atomicity of the CFC instructions the resulting CFE may be detected in the second iteration. The second case can originate from an inappropriate injection time. Although all routines protected by CFC contain at least 3 “nodes” it is not necessary that the last one performs the check routine. In case that the fault was injected during the execution of the last node, it may corrupt the output data shortly before exiting. The check if data was processed correctly is done by dumping the generated data into the fault injection environment and comparing it with a known error-free result.

The “Fault Location“ column represents the fault distribution of all detected CFEs. This information was derived by automated post-processing of the log-files containing an excessive amount of manifold information. The “Total DEs” column represents the number of data errors detected during the whole workload. There is no specific assignment to any action or event inside the system. Although this information has to be interpreted with care it is shown here to provide insight on how often false data is generated.

Not surprisingly, the number of detected CFEs which are shown in the “Detected CFEs” columns is quite small, which reflects the fact that one or two check-nodes distributed over the whole code are not enough. Also, the number of detected faults corresponding to the data integrity meets the expectations. Nearly every time a CFE was detected the output was uncorrupted and, therefore, falsified data was not generated. Although the column “Detected CFEs” constitutes the main outcome of this experiment, faults that are detected as data errors (DEs) without detected CFE cannot simply be ignored. Without further knowledge about their manifestation they must be considered as data errors which cannot be detected using control flow checking.

By looking at the fault distributions shown in Appendix A.4 it can be seen that the fault location has not much impact on the error detection ratio. This can either be related to the small memory utilization or to a lack of observability. An important conclusion of experiment 1 is that the distribution of check-nodes has a significant impact on the CFE detection ratio. By doubling the amount of check nodes, as it is the case in the FFT workload, the resulting detection ratio becomes a multiple compared to all other workloads. Another result is that the ratio between detected CFEs and correct data output shows that using this technique the overall system availability can be increased. It is also important to note that nearly all of the detected CFEs originate from faults injected into the register file. This somehow unexpected result is an indication that the implemented technique is more powerful than expected.

6.4.2 Experiment 2: Errors in the Output data (Data Errors - DEs)

The aim was to evaluate the correlation between injected faults and resulting data errors. The experiment was conducted in order to attain a detailed insight on how different fault locations are related to DEs. DEs were expected to mainly occur in case of faults injected to either the data memory or the register file. Another goal was to illustrate how many DEs could have been covered using error detection and error correction techniques. The number of faults for each workload is denoted by n. Table 6.2 shows the results gained from the experiment, which were gathered in the same way as for Experiment 1.

The “Total DEs” column represents the amount of errors detected by dumping the generated data and comparing it with an error-free data-set. Recalling experiment 1, it has been shown

Workload	Total DEs	Fault Location		
		Reg	L1D	L1P
General Purpose with Inter Check	1.472	751	364	347
Matrix-Multiplication with Intra Check	94	58	14	22
Matrix-Transpose with Intra Check	159	150	4	5
Max of Vector with Intra Check	141	128	9	4
Complex FFT with Intra Check	1475	1445	20	10

Table 6.2: Measured DEs, $n = 12000$

earlier that nearly all detected CFEs also prevented the generation of false data. In this particular case the total amount of DEs represents all faults that can either not be related to internally detected CFEs nor can they be classified using external mechanisms[2].

The "Fault location" column represents the distribution of the origin of the measured DEs. Due to the vast amount of logged data, this information again had to be derived using post-processing of the injector log-files.

The number of DEs related to register based faults is very high. This confirms that due to the large amount of registers the usage of ECC would not suffice. Apparently, compared to all other intra-procedure tests, the FFT workload experiences the highest amount of data errors related to the fault count. There are two reasons: first, the total amount of data produced during execution is higher and second, the greater amount of time needed for processing expands the fault injection window compared to all other intra-procedure workloads.

The result illustrates the vulnerability with respect to SEUs occurring in registers. In search of a counter measure recall the FFT workload of experiment 1. Due to the increased amount of check nodes it was possible to increase the detection ratio and, therefore, to decrease the amount of false data generated. Applying this knowledge to the current experiment and looking at the particular fault distribution of the FFT workload, it appears as if further increasing the check node ratio would decrease false data generation. Apparently, this is not really a solution. By neglecting the loss of external observability, an appropriate solution would be the use of *EDDI*, *TTMR* or an embedded checksum based on rectangular coding.

6.4.3 Experiment 3: No Effect-Faults

For every iteration of each experiment, the injection location and the outcome after a given period of time was analyzed. This allows to determine the amount of faults not activated or overwritten due to a recent write to the fault location. The number of faults for each workload is denoted by n. Table 6.3 shows the results gained from the experiment.

---

[2] problem of external observability

Workload	No Error Faults	Fault Location Reg	 L1D	 L1P
General Purpose with Inter Check	10796	5465	2604	2722
Matrix-Multiplication with Intra Check	11609	5938	2993	2974
Matrix-Transpose with Intra Check	11841	5956	2928	2957
Max of Vector with Intra Check	11859	5843	2999	3017
Complex FFT with Intra Check	10525	4615	2984	2926

Table 6.3: No-Effect faults, $n = 12000$

The "No Error" column represents the amount of faults that did not create any measurable effect. Testing of this hypothesis was accomplished by comparing the generated data with a clean reference and by checking the fault database inside the target. However, this does not guarantee that the executional trace was exactly as intended. It only illustrates how many faults were not creating any measurable effect. Two cases cannot be covered: (i) Errors inside other regions of the memory and (ii) control flow errors without the scope of the implemented techniques. The first case can only be covered by comparing the whole memory of the target against an error-free one. Due to the unstable debug interface of the development tool-chain this is only possible with extreme effort. The second case originates from the lack of observability. It could be covered by implementing an extensive breakpoint strategy. Since this is workload dependent and highly time consuming (in terms of engineering time) this was not pursued.

The fault location definition is the same as for all other experiments. Again, it was gathered by means of post-processing.

As expected, the number of faults without measurable effect is dominant, which partly reflects that the application only exercises parts of the available resources and, hence, faults injected into unused resources remain inactive. The bigger part, however, is believed to be the result of limited observability both external and internal. Nearly half of the no-effect-faults are related to registers. Of course, this result does not prove that all faults were really inactive. Their inactivity can only be proven by dumping the whole memory and by comparing it with a representative copy after every iteration.

6.4.4 Experiment 4: Illegal Opcode

The aim was to find out how often a modification of a single bit within an instruction word results in a syntactically incorrect instruction. Therefore, the internal opcode detection unit of the IDE was used. Every time syntactically invalid opcode was executed an error message was generated. However, this does not give any insight into the overall count of illegal instructions since a measurable effect can only be in case the instruction is executed. To evaluate the amount of total illegal instructions the entire memory must be dumped. Due to the unstable behaviour of the debugger, this was omitted. The conducted experiment was supposed to give an insight into the appearance of executed illegal opcode constrained by the size of periodically executed code. It is expected that the execution of illegal instructions mainly creates data errors. The reason for this is the distribution of conditional bits (creg) over various instructions, e.g. branch instructions. The number of faults for each workload is denoted by n. Table 6.4 shows the results gained from the experiment.

Workload	Illegal opcode	CFEs	DEs
General Purpose with Inter Check	59	0	9
Matrix-Multiplication with Intra Check	78	0	13
Matrix-Transpose with Intra Check	110	0	36
Max of Vector with Intra Check	100	0	38
Complex FFT with Intra Check	29	29	0

Table 6.4: Illegal Opcode and their effects, $n = 12000$

The experiment results shown in Table 6.4 have been obtained using a slightly different fault injection environment. Due to fact that illegal opcode is not detected by the DSP and that the outcome is not easily observable, the experiment setup was changed from "injection in target" into "injection into a simulator" based execution using Texas-Instrument's cycle accurate DSP simulator. This does not show what is really going on inside the target but can be used to determine how often illegal opcode is encountered.

The column labelled "Illegal opcode" represents how often illegal opcode was executed using the simulator. It does not show how often illegal opcode was produced, but how often this problematic situation was encountered. The "CFEs" and "DEs" columns connect observed warnings and their outcome.

The results shown in Table 6.4 confirm our expectations concerning the occurrence of DEs. It is surprising that the outcome of illegal opcode is either dedicated to a CFE or to a DE but never to both. But this can also be related to the lack of observation. The FFT workload was the only one detecting CFEs. The reason for this can be the larger code size or the longer time necessary time needed for executing the function.

6.4.5 Experiment 5: Mixed Intra/Inter Procedure Checking

The aim of this experiment was to evaluate the benefit created by an mix of intra and inter procedure control flow checking. For this purpose, a workload containing the aforementioned base program was created. The experiment contains parts of the "General Purpose with Inter check" and the "Matrix-Multiplication with Intra check" workload. The environment setup was rearranged from simulation to target evaluation. It was presumed that the CFE detection ratio would increase due to the more frequent occurrence of CFC instructions. Additionally, several breakpoints were inserted to enhance the "perception" of the external observer. These breakpoints were supposed to give insight on how often undetected CFEs were encountered and how often they could have been detected using external logic. As before, it is necessary to explain how the values presented in Table 6.5 were gathered.

	Detected CFEs			Undetected CFEs			
Workload	Reg	L1D	L1P	Reg	L1D	L1P	Total CFEs
Mat-Mult with Inter/Intra Check	598	1	12	374	0	48	1038

Table 6.5: CFE detection ratio, $n = 23258$

The column “Detected CFEs” represents the amount of control flow errors that were detected using the internal detection mechanisms. The first breakpoint was inserted at the reset vector. This breakpoint is only triggered in case of a detected CFE. As already mentioned, every detected CFE is followed by a jump into the reset routine. Next, two breakpoints were added to ensure that the calculation routines were only executed twice. These two breakpoints ensure that in the absence of CFEs the target stops within one of these two.

The presented numbers were gathered in the following manner: (i) After injection of a fault the target is continuing its asynchronous execution. (ii) after time t the execution is stopped and the current program counter is analyzed. If the program counter matches the reset vector, the internal mechanisms have successfully detected the control flow anomaly. The location of the error origin was determined using post processing of the fault injector logfile.

The column “Undetected CFEs” represents the number of CFEs that were not detected using internal software mechanisms. It was derived by performing an analysis of the current program counter value. If the program counter is not matching any of the breakpoints mentioned before, an undetected CFE can be reported.

The results shown in Table 6.5 illustrate the effectiveness of the mixed approach. Due to the fact that the overhead caused by adding the inter-checking routines is constant and moderate, the amount of detected CFEs was increased several times without adding substantially to the execution-time budget. Compared to all other workload examples, this approach seems to be most effective in terms cost and benefit. One might think that the fairly high number of undetected CFEs is still a show stopper, but by looking at the workload in more detail it can be seen that relative to the small number of CFC instructions (small compared to all instructions executed) the outcome is indeed reasonable.

Although there is still room for improvement using internal mechanisms it has also been shown that external logic like the FTC presented in Section 5.7 can be used to substantially improve the overall detection ratio. A detailed analysis of the injected faults can be found in Appendix A.22.

6.5 Test Evaluation

The tests have shown that by using software only techniques it is not possible to achieve a reasonable availability level. As expected, the CFCSS techniques are able to detect control flow errors and can, thus, prevent the generation of data faults if the fault was introduced in a region modifying the control flow. On the other hand it has become apparent that the main cause for data errors based on register faults cannot be reduced, in spite of the great effort taken. As presumed, this effect cannot be covered using CFC methodologies. However, it has been shown that the use of an external fault tolerance controller can significantly decrease the gap left by the software-only routines.

A more differentiated view has revealed that it is indeed possible to detect a respectable amount of SEE related faults using software-only techniques but that they do not have enough power to be used in a stand alone manner. The following conclusions can be drawn from the experiments:

- Control Flow Checking can be used to reduce the occurrence of CFEs and consequently also reduce the number of data related faults
- Finer CFCSS granularity helps increasing the detection ratio
- The amount of externally detected data errors is directly related to the induced register faults
- Illegal opcode mainly causes data errors
- A CFCSS-only solution is not adequate to provide the high availability required for use in space
- Resource intensive solutions like *TTMR* or *EDDI* do have their justification in spite of their impractical appearance

Recalling the success metrics shown in Section 3.3 it can be stated that nearly all goals have been reached. None of the presented techniques is bound to the platform selected for the present work, but may be applied to a broad spectrum of present and future space-grade DSPs. Conclusions concerning about the intrusiveness of the implemented techniques must be considered ambiguous. On the one hand, the induced overhead varies strongly depending on the selected granularity. On the other hand it has been shown that more or less good SEU performance can be achieved with finer granularity.

CHAPTER 7

Conclusions and Outlook

The proposed spaceborne DSP platform in form of Texas Instrument's SMV 320C6701 DSP represents an economically efficient way of implementing fault tolerance without being affected by ITAR, which is not the case for all other platforms equivalent with respect to provided services and performances. To estimate the risk of radiation induced SEEs their outcome and their consequences were analyzed. In a first step the quantification process was decomposed into two steps, namely occurrence and effects. Furthermore, existing application code for typical signal processing algorithms was modified and extended, as its original form would have been prone to faults.

Three mechanisms based on control flow checking were distinguished, i.e. inter-procedure-, intra-procedure- and inter/intra-procedure-checking. With respect to data integrity, two mechanisms viz. software EDAC and Mirror Checking, were distinguished. Since the overhead introduced by both data-integrity methods was enormous, the techniques were soon removed from the list of potential candidates to be further examined for practical use in space systems.

Control Flow Errors were studied in more detail and evaluated using different fault injection experiments. Interestingly enough, it was found that data related errors tend to have register based faults as origin. On the other hand, it has been shown that it is indeed possible to detect a reasonable count of CFEs using a software-only solution. Also, the threat of executed illegal opcode was eased.

SWIFT methods can be seen as a first step towards fault tolerance using the given architecture. It has been shown that without external observation it is implausible to provide an adequate level of availability. The risk of implementing a software-only architecture on a radiation tolerant DSP seems to be higher than might be initially anticipated, and it seems reasonable to consider such solution with the appropriate care. The usage of the proposed fault tolerance controller increases error detection capabilities and reduces fault detection delays to the extent rendering the approach attractive for space-mission deployment.

Apparently, due to the limited scope of the present undertaking this thesis leaves some open issues as well as points which require further analysis. A more detailed look has to be taken at the effects related to register faults. Also, fault-detection delay or fault-activation delay should be investigated in more detail. Moreover an example for a combined approach using *EDDI* or *TTMR* should be analyzed. Furthermore a physical prototype using the FTC can be used to experimentally analyze the overall sensitivity to SEEs.

With respect to mass, power consumption and total costs, the presented solution can easily keep up with current state-of-the art spaceborne technology offering the same level of radiation tolerance but considerably less processing performance and data throughput.

Direct comparison of the presented solution with other state-of-the art DSPs both spaceborne and terrestrial was out of the scope of the present activity and would have been difficult even on paper-level, due to the lack of comparable information.

To allow for a more practicable conclusion recall the information presented in Section A.1 and [30]. It shows that the DSP has an average background SEU rate of less than 7.710^{-3} upsets per day, corresponding to approximately 130 days of upset free operation in geo-stationary orbit (GEO. Augmenting this information by the knowledge gained through the experiments, the DSP in combination with an FTC can be considered virtually fault tolerant and very attractive for most practical space applications.

APPENDIX A

Appendix

A.1 SMV320C6701 - Radiation performance

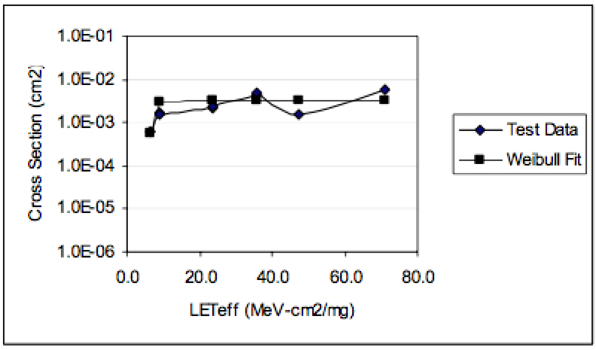


Figure A.1: SEU Characteristics - Data Memory Verification using BIST

Environment	EMIF, McBSP, DMA, Power Down Logic, Data Access Controller, Program/Cache Memory, Data Memory, Boot Modes	Program/Cache and Data, Memory Program Access/Cache Controller and Data Access Controller	Data Memory Verification using BIST	CPU
solar min	4.11E-05	3.32E-04	6.39E-03	7.94E-04
solar max	9.17E-06	6.83E-05	1.12E-03	1.13E-04
worst 5-min	3.78E-02	2.44E-01	1.80E-02	5.83E-01
worst day	3.59E-04	2.43E-03	1.44E-00	5.40E-03
worst week	1.85E-04	1.42E-03	3.4E-00	3.79E-03

Table A.1: Summary of the SEU Rate (Upsets/day) for the SVM320C6701

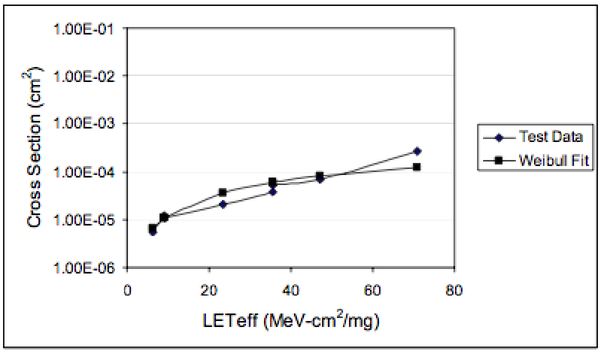


Figure A.2: SEU Characteristics - EMIF, McBSP, DMA, Power Down Logic, Data Access Controller, Program/Cache Memory, Data Memory, Boot Modes

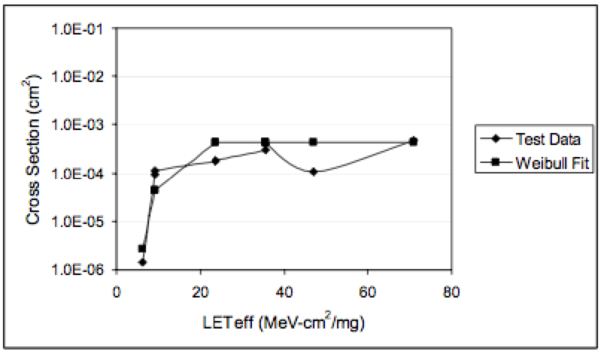


Figure A.3: SEU Characteristics - CPU

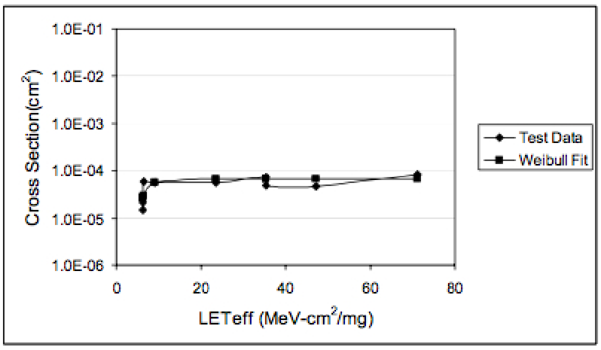


Figure A.4: SEU Characteristics - Program/Cache and Data Memory, Program Access/Cache Controller and Data Access Controller

Functional Block	1,000 km	5,000 km	10,000 km	15,000 km
Data Memory Verification using BIST	1.6E-04	1.7E-01	2.0E-02	1.9E-05
CPU	1.03E-07	8.04E-05	3.97E-06	7.90E-11
Program/Cache and Data, Memory Program Access/Cache Controller and Data Access Controller	1.77E-06	3.54E-03	2.15E-04	2.07E-07
EMIF, McBSP, DMA, Power Down Logic, Data Access Controller, Program/Cache Memory, Data Memory, Boot Modes	2.75E-06	6.67E-03	4.11E-04	3.95E-07

Table A.2: Proton upset rates (Upsets/day) at various altitudes and 0 deg inclination angle

Functional Block	1,000 km	5,000 km	10,000 km	15,000 km
Data Memory Verification using BIST	7.9E-04	8.4E-01	4.8E-02	5.8E-05
CPU	2.38E-06	4.05E-05	1.54E-06	2.91E-11
Program/Cache and Data, Memory Program Access/Cache Controller and Data Access Controller	8.85E-06	9.14E-03	5.28E-04	6.33E-07
EMIF, McBSP, DMA, Power Down Logic, Data Access Controller, Program/Cache Memory, Data Memory, Boot Modes	6.51E-05	1.05E-03	8.15E-05	1.46E-07

Table A.3: Proton upset rates (Upsets/day) at various altitudes and 28.5 deg inclination angle

Functional Block	1,000 km	5,000 km	10,000 km	15,000 km
Data Memory Verification using BIST	9.0E-04	4.6E-01	4.4E-03	4.1E-06
CPU	1.39E-06	2.20E-05	8.71E-07	1.70E-11
Program/Cache and Data, Memory Program Access/Cache Controller and Data Access Controller	9.85E-06	5.06E-03	4.77E-05	4.45E-08
EMIF, McBSP, DMA, Power Down Logic, Data Access Controller, Program/Cache Memory, Data Memory, Boot Modes	3.91E-05	5.90E-04	9.11E-05	8.50E-08

Table A.4: Proton upset rates (Upsets/day) at various altitudes and 51.6 deg inclination angle

A.2 Control Flow Checking using Software Signatures - Intra Procedure

The following section shall give a detailed insight of how the different function were modelled and implement using control flow checking techniques. The including Tables and Figure are self-explanatory.

A.2.1 Single Precision Matrix Multiplication - CFCSS sheet

```
void DSPF_sp_mat_mul(float *x, int r1, int c1, float *y, int c2, float *r)
{
        int i, j, k;
        float sum;
        // Multiply each row in x by each column in y.
        // The product of row m in x and column n in y is placed
        // in position (m,n) in the result.
        for (i = 0; i < r1; i++)
        {
                for (j = 0; j < c2; j++)
                {
                        sum = 0;
                        for (k = 0; k < c1; k++)
                        {
                                sum += x[k + i*c1] * y[j + k*c2];
                        }
                        r[j + i*c2] = sum;
                }
        }
}
```

Listing A.1: Matrix multiplication in C taken from the DSPLIB

Transition			Signature d_k
From	→	To	
1		2	0x3333
2		3	0x1111
3		4	0x7777
4		5	0x1111

Figure A.5: Signature difference

RT-Signature update function

Function	D_i	$s_{i,1} \oplus s_{i,m}$
RT-1	0x5555	$(4 \oplus 1)$
RT-2	0x6666	$(4 \oplus 2)$
RT-3	0x7777	$(4 \oplus 3)$

Figure A.6: Real-time signature

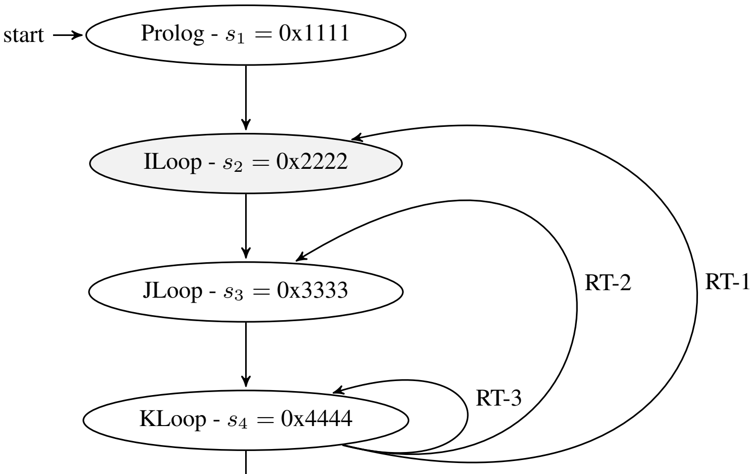


Figure A.7: Control Flow Graph

A.2.2 Single Precision Matrix Transpose - CFCSS sheet

```c
void DSPF_sp_mat_trans(const float ∗restrict x, int rows, int cols, float ∗restrict r)
{
	int i,j;
	for(i=0; i<cols; i++)
	{
		for(j=0; j<rows; j++)
		{
			r[i ∗ rows + j] = x[i + cols ∗ j];
		}
	}
}
```

Listing A.2: Matrix transpose in C taken from the DSPLIB

Transition			Signature d_k
From	$\rightarrow$	To	
1		2	0x3331
2		3	0x1117

Figure A.8: Signature difference

RT-Signature update function

Function	D_i	$s_{i,1} \oplus s_{i,m}$
RT-1	0x3331	$(1 \oplus 2)$

Figure A.9: Real-time signature

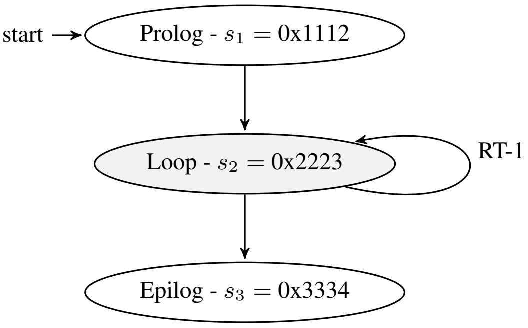


Figure A.10: Control Flow Graph

A.2.3 Single precision maximum value of vector - CFCSS sheet

```
float DSPF_sp_maxval(const float* x, int nx)
{
        int i,index;
        float max;
        *((int *)&max) = 0xff800000;
        for (i = 0; i < nx; i++)
        {
                if (x[i] > max)
                {
                        max = x[i];
                        index = i;
                }
                return max;
        }
}
```

Listing A.3: Maximum value in C taken from the DSPLIB

Transition			Signature d_k
From	$\rightarrow$	To	
1		2	0x3337
2		3	0x1111

Figure A.11: Signature difference

RT-Signature update function

Function	D_i	$s_{i,1} \oplus s_{i,m}$
RT-1	0x3337	$(1 \oplus 2)$

Figure A.12: Real-time signature

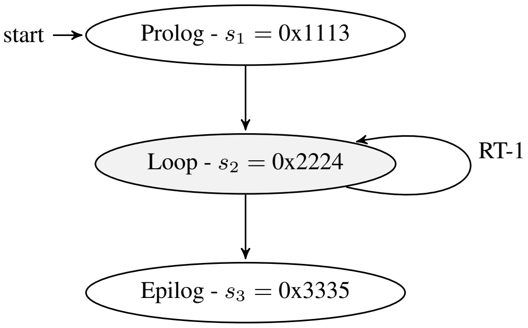


Figure A.13: Control Flow Graph

A.2.4 Single precision floating-point radix-2 FFT with complex input - CFCSS sheet

```
void DSPF_sp_cfftr2_dit(float∗ x, float∗ w, short n)
{
        short n2, ie, ia, i, j, k, m;
        float rtemp, itemp, c, s;
        n2 = n;
        ie = 1;
        for(k=n; k > 1; k >>= 1)
        {
                n2 >>= 1; ia = 0;
                for(j=0; j < ie; j++)
                {
                        c = w[2∗j];
                        s = w[2∗j+1];
                        for(i=0; i < n2; i++)
                        {
                                m = ia + n2;
                                rtemp = c∗x[2∗m] + +s ∗ x[2∗m+1];
                                itemp = c∗x[2∗m+1] − s∗ x[2∗m];
                                x[2∗m] = x[2∗ia] − rtemp;
                                x[2∗m+1]= x[2∗ia+1] − itemp;
                                x[2∗ia]= x[2∗ia] + rtemp;
                                x[2∗ia+1] = = x[2∗ia+1] + itemp;
                                ia++;
                        }
                        ia += n2;
                }
                ie <<= 1;
        }
}
```

Listing A.4: Single precision floating-point radix-2 FFT with complex input taken from the DSPLIB

Transition			Signature d_k
From	$\rightarrow$	To	
1		2	0x3331
2		3	0x1113
3		4	0x7771

Figure A.14: Signature difference

RT-Signature update function

Function	D_i	$s_{i,1} \oplus s_{i,m}$
RT-1	0x1113	$(3 \oplus 3)$
RT-2	0x2222	$(3 \oplus 1)$

Figure A.15: Real-time signature

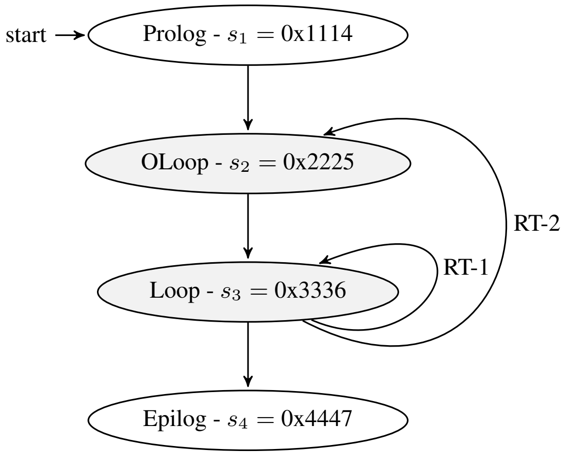


Figure A.16: Control Flow Graph

A.3 Texas Instruments Software Design Flow

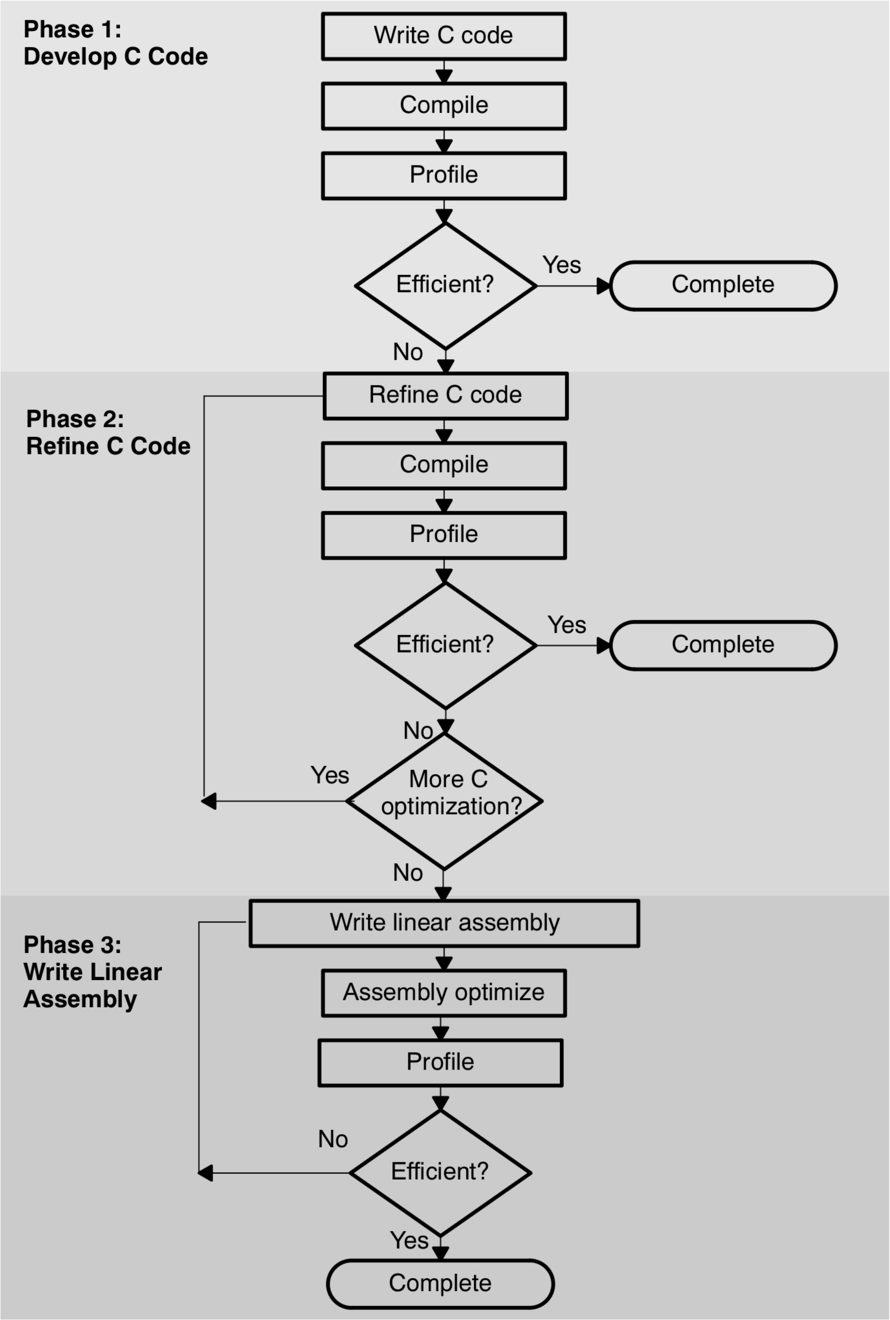


Figure A.17: Texas Instruments Software Design Flow [26]

A.4 Fault Injection Experiments

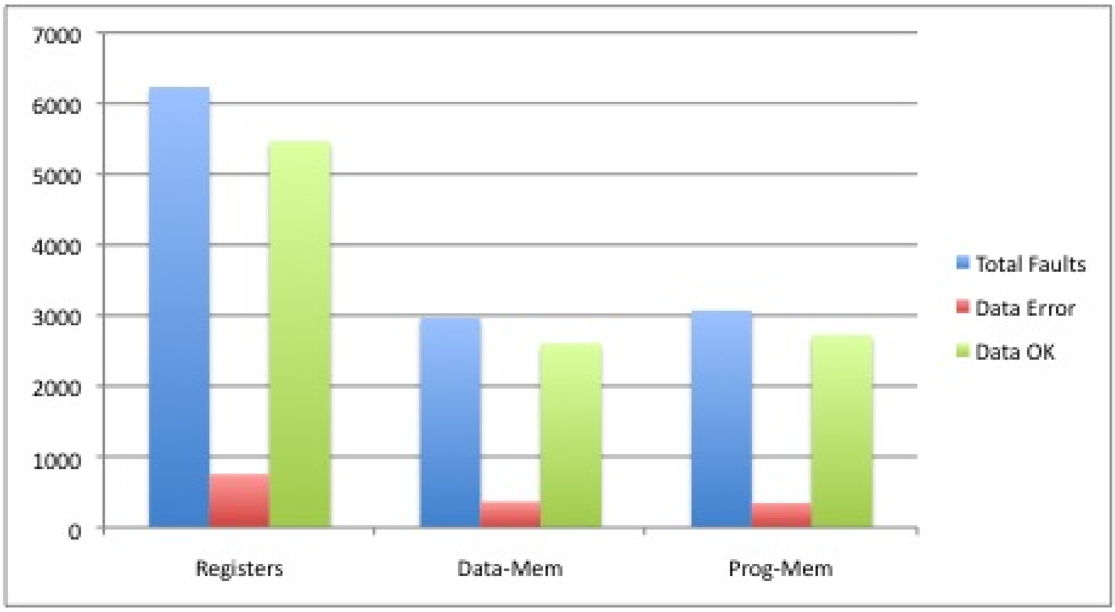


Figure A.18: Fault Distribution for Inter-Procedure Workload

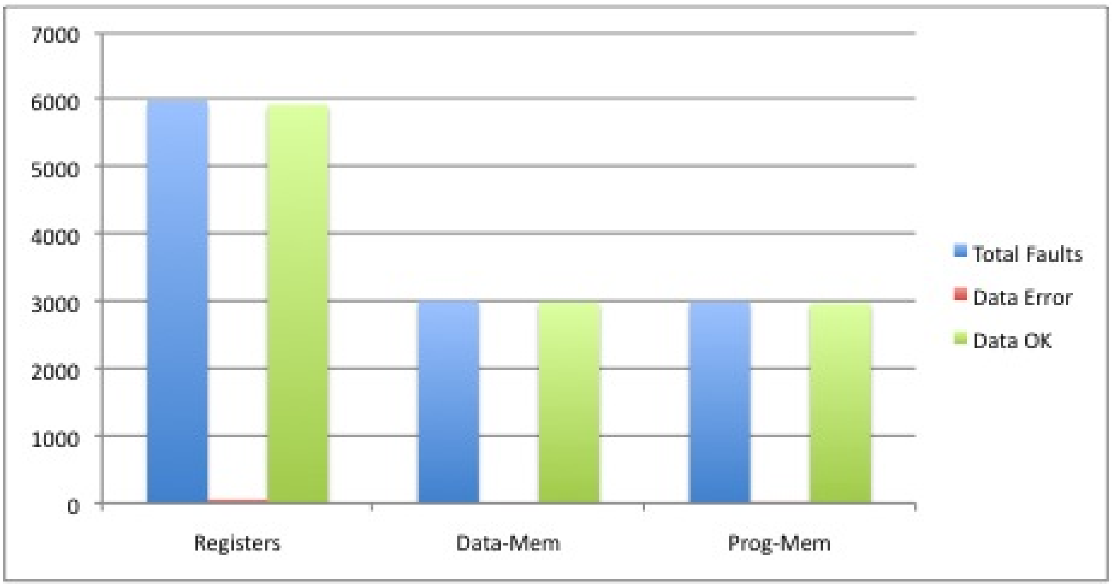


Figure A.19: Fault Distribution for Intra-Procedure Matrix Multiplication Workload

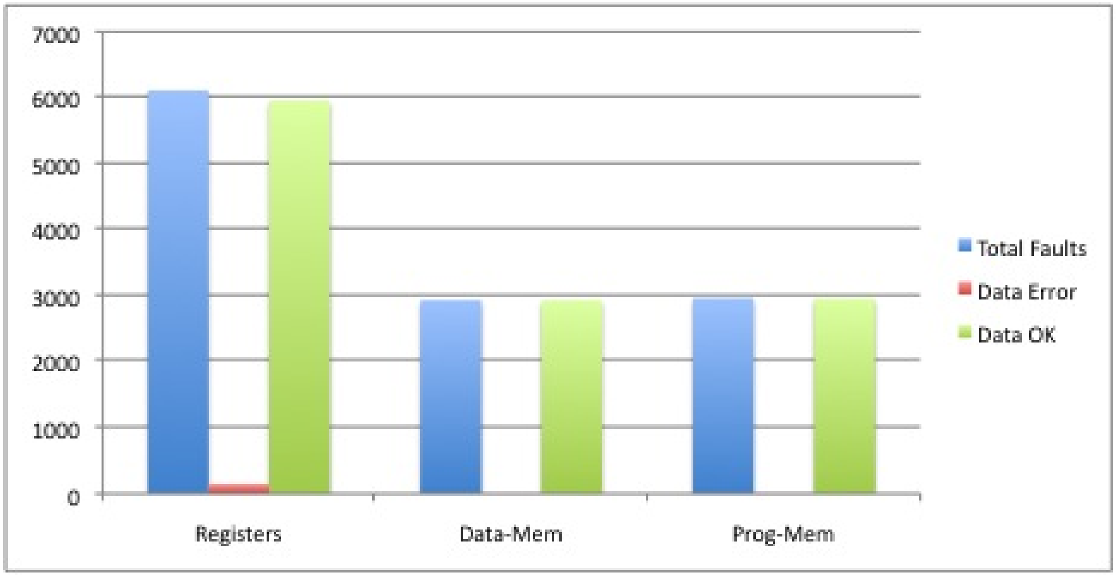


Figure A.20: Fault Distribution for Intra-Procedure Matrix Transpose Workload

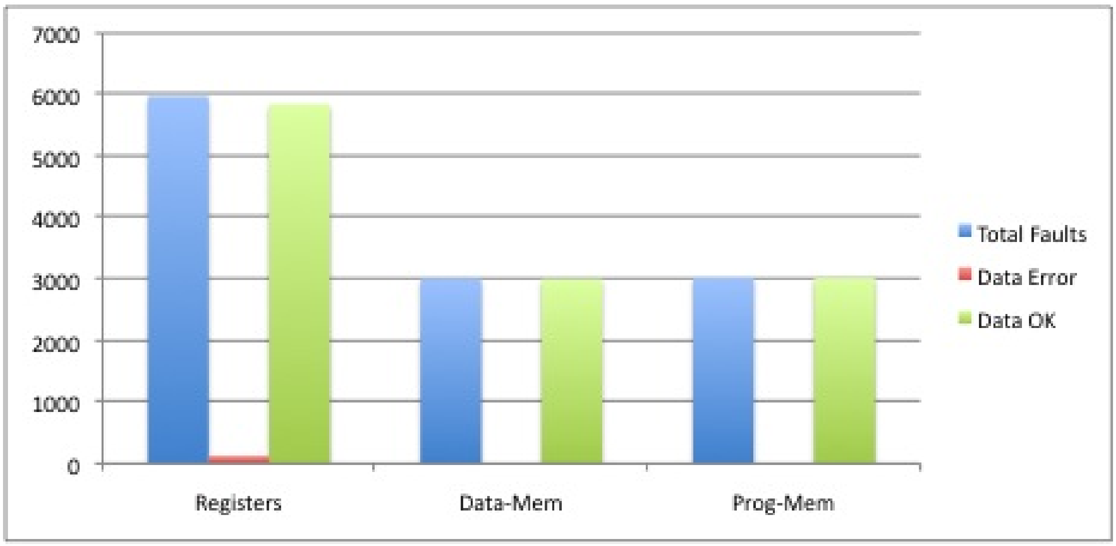


Figure A.21: Fault Distribution for Intra-Procedure Maximum Value of Vector Workload

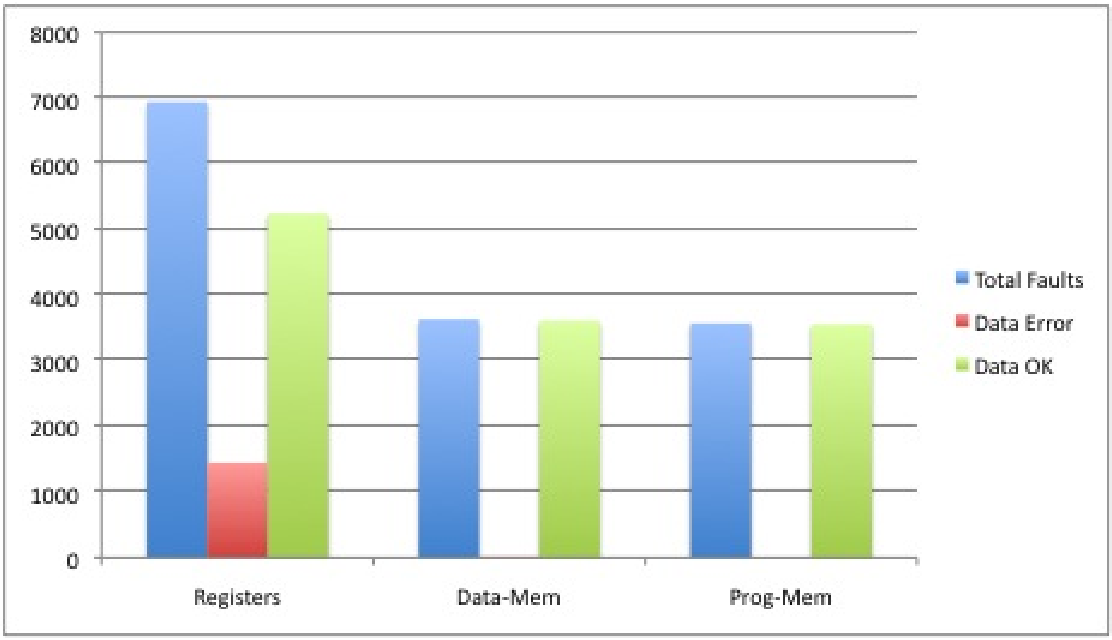


Figure A.22: Fault Distribution for Intra-Procedure Complex FFT Workload

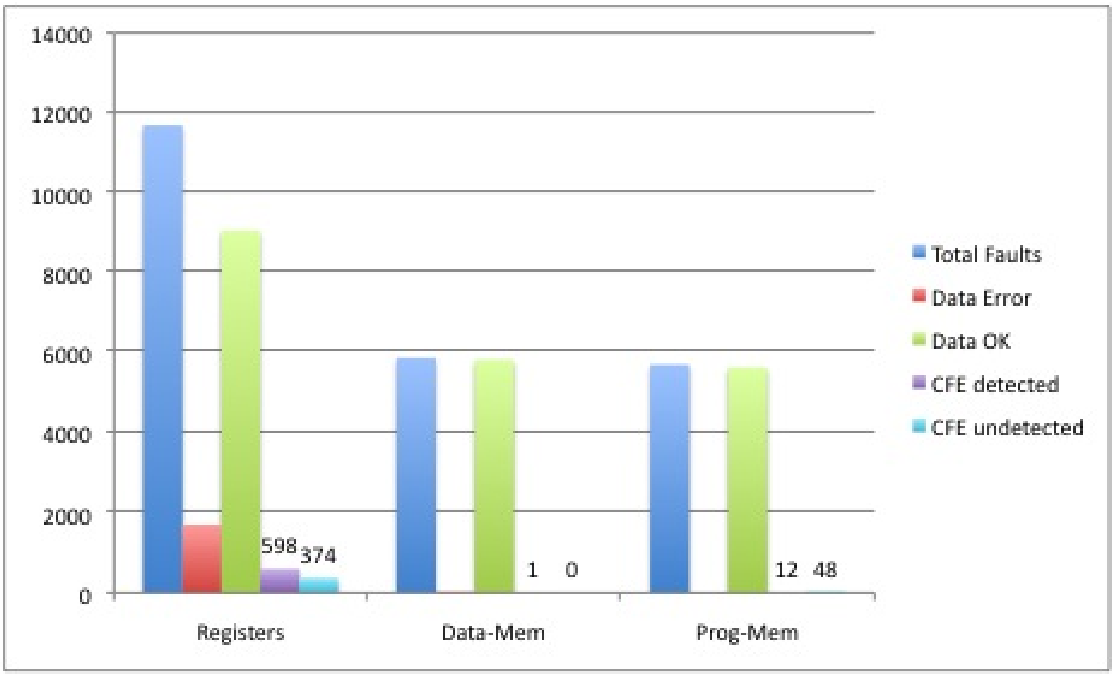


Figure A.23: Fault Distribution for Intra/Inter-Procedure Workload

Bibliography

[1] D. Andrews. Using executable assertions for testing and fault tolerance. *9th Fault-Tolerance Computing Symp.*, pages 20–22, 1979.

[2] G. Anelli, M. Campbell, M. Delmastro, F. Faccio, S. Floria, A. Giraldo, E. Heijne, P. Jarron, K. Kloukinas, A. Marchioro, P. Moreira, and W. Snoeys. Radiation tolerant vlsi circuits in standard deep submicron cmos technologies for the lhc experiments: practical design aspects. *Nuclear Science, IEEE Transactions on*, 46(6):1690 –1696, dec. 1999.

[3] A. Balasubramanian, B.L. Bhuva, J.D. Black, and L.W. Massengill. Rhbd techniques for mitigating effects of single-event hits using guard-gates. *Nuclear Science, IEEE Transactions on*, 52(6):2531 – 2535, dec. 2005.

[4] D. Bessot and R. Velazco. Design of seu-hardened cmos memory cells: the hit cell. In *Radiation and its Effects on Components and Systems, 1993.,RADECS 93., Second European Conference on*, pages 563 –570, sep 1993.

[5] Jason A. Blome, Shantanu Gupta, Shuguang Feng, Scott Mahlke, and Daryl Bradley. Cost-efficient soft error protection for embedded microprocessors, 2006.

[6] T. Calin, M. Nicolaidis, and R. Velazco. Upset hardened memory design for submicron cmos technology. *Nuclear Science, IEEE Transactions on*, 43(6):2874 –2878, dec 1996.

[7] C. L. Chen and M. Y. Hsiao. Error-correcting codes for semiconductor memory applications: a state-of-the-art review. *IBM J. Res. Dev.*, 28:124–134, March 1984.

[8] Ching-Yi Chen and Cheng-Wen Wu. An adaptive code rate edac scheme for random access memory. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 735 –740, march 2010.

[9] Po-Yuan Chen, Yi-Ting Yeh, Chao-Hsun Chen, Jen-Chieh Yeh, Cheng-Wen Wu, Jeng-Shen Lee, and Yu-Chang Lin. An enhanced edac methodology for low power psram. In *Test Conference, 2006. ITC '06. IEEE International*, pages 1 –10, oct. 2006.

[10] D.R. Czajkowski, P.K. Samudrala, and M.P. Pagey. Seu mitigation for reconfigurable fpgas. In *Aerospace Conference, 2006 IEEE*, page 7 pp., 0-0 2006.

[11] J.B. Eifert and J.P. Shen. Processor monitoring using asynchronous signatured instruction streams. In *Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on*, page 106, jun 1995.

[12] A.M. El-Attar and G. Fahmy. An improved watchdog timer to enhance imaging system reliability in the presence of soft errors. In *Signal Processing and Information Technology, 2007 IEEE International Symposium on*, pages 1100 –1104, dec. 2007.

[13] J. Enright, M. Hilstad, A. Saenz-Otero, and D. Miller. The spheres guest scientist program: collaborative science on the iss. In *Aerospace Conference, 2004. Proceedings. 2004 IEEE*, volume 1, pages 6 vol. (xvi+4192), march 2004.

[14] L. Entrena, C. Lopez, and E. Olias. Automatic insertion of fault-tolerant structures at the rt level. In *On-Line Testing Workshop, 2001. Proceedings. Seventh International*, pages 48 –50, 2001.

[15] Cmos Technology For, F. Faccio, K. Kloukinas, G. Magazzù, and A. Marchioro. Seu effects in registers and in a dual-ported static ram designed in a 0.25 m.

[16] N. Gaitanis. The design of totally self-checking tmr fault-tolerant systems. *Computers, IEEE Transactions on*, 37(11):1450 –1454, nov 1988.

[17] Thomas M. Galla, Michael Sprachmann, Andreas Steininger, and Christopher Temple. Control flow monitoring for a time-triggered communication controller, 1999.

[18] J. Gambles, L. Miles, J. Hass, W. Smith, S. Whitaker, and B. Smith. An ultra-low-power, radiation-tolerant reed solomon encoder for space applications. In *Custom Integrated Circuits Conference, 2003. Proceedings of the IEEE 2003*, pages 631 – 634, sept. 2003.

[19] C. S. Guenzer, E. A. Wolicki, and R. G. Allas. Single event upset of dynamic rams by neutrons and protons. *Nuclear Science, IEEE Transactions on*, 26(6):5048 –5052, dec. 1979.

[20] M. Rela H. Madeira and J. G. Silvia. Time behavior monitoring as an error detection mechanism, 1993.

[21] M. Y. Hsiao. A class of optimal minimum odd-weight-column sec-ded codes. *IBM J. Res. Dev.*, 14:395–401, July 1970.

[22] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Computers*, 33(6):518–528, 1984.

[23] V V Ignatushchenko. et al., effectiveness of temporal redundancy of parallel computational processes. *Automation and Remote Control*, (55):900–911, 1994.

[24] George A. Reis III. *SOFTWARE MODULATED FAULT TOLERANCE*. PhD thesis, Princeton University, 2008.

[25] Texas Instruments. Smj320c6701 data sheet - sgus030e.

[26] Texas Instruments. Tms320c6000 programmer's guide - spru198k, 2011.

[27] N. Ismailoglu, O. Benderli, I. Korkmaz, S. Yesil, R. Sever, H. Sunay, T. Kolcak, and Y.C. Tekmen. Gezgin: a case study of a real-time image processing subsystem for micro-satellites. In *Recent Advances in Space Technologies, 2003. RAST '03. International Conference on. Proceedings of*, pages 302 – 307, nov. 2003.

[28] Roland Weigand Jan Andersson, Jiri Gaisler. NEXT GENERATION MULTIPURPOSE MICROPROCESSOR. *DASIA - Data Systems In Aerospace*, 2010.

[29] NASA/Goddard Space Flight Center Janet Barth. IEEE NSREC Short Course Session I - Radiation Environments. *IEEE*, pages 1–127, 1997.

[30] R. Joshi, R. Daniels, M. Shoga, and M. Gauthier. Radiation hardness evaluation of a class v 32-bit floating-point digital signal processor. In *Radiation Effects Data Workshop, 2005. IEEE*, pages 70 – 78, july 2005.

[31] J. C. Laprie. Dependability of computer systems: concepts, limits, improvements. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 2–11, 1995.

[32] D.J. Lu. Watchdog processors and structural integrity checking. *Computers, IEEE Transactions on*, C-31(7):681 –685, july 1982.

[33] H. Madeira and J.G. Silva. On-line signature learning and checking: experimental evaluation. In *CompEuro '91. Advanced Computer Technology, Reliable Systems and Applications. 5th Annual European Computer Conference. Proceedings.*, pages 642 –646, may 1991.

[34] H Madeira and J G Silvia. On-line signature learning and checking, dependable computing for critical applications 2, 1992.

[35] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Trans. Comput.*, 37:160–174, February 1988.

[36] D. Marienfeld. *Effiziente Fehlererkennung für arithmetische Einheiten*. Potsdam, 2007.

[37] D.G. Mavis and D.R. Alexander. Employing radiation hardness by design techniques with commercial integrated circuit processes. In *Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE*, volume 1, pages 2.1 –15–22 vol.1, oct 1997.

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

[39] Masood Namjoo. Techniques for concurrent testing of vlsi processor operation. In *ITC*, pages 461–468, 1982.

[40] M. Nicolaidis and Y. Zorian. On-line testing for vlsi—a compendium of approaches. *Journal of Electronic Testing*, 12:7–20, 1998. 10.1023/A:1008244815697.

[41] N. Oh, S. Mitra, and E.J. McCluskey. Ed4i: error detection by diverse data and duplicated instructions. *Computers, IEEE Transactions on*, 51(2):180 –199, feb 2002.

[42] N. Oh, P.P. Shirvani, and E.J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63 –75, mar 2002.

[43] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-flow checking by software signatures. *IEEE TRANSACTIONS ON RELIABILITY*, 51:111–122, 2002.

[44] J. Ohlsson and M. Rimen. Implicit signature checking. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 218 –227, jun 1995.

[45] R.H. Paschburg. Software Implementation of Error-Correcting Codes. *Univ. Illinois, Urbana*, 1974.

[46] R.H. Paschburg. Error correction with a microprocessor. *Proc. IEEE National Aerospace and Electronics Conf*, 1977.

[47] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin. Improving fpga design robustness with partial tmr. In *Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International*, pages 226 –232, march 2006.

[48] Roshan G. Ragel and Sri Parameswaran. Hardware assisted pre-emptive control flow checking for embedded processors to improve reliability. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, pages 100–105, New York, NY, USA, 2006. ACM.

[49] Thammavarapu R. N. Rao. *Error Coding for Arithmetic Processors*. Academic Press, Inc., Orlando, FL, USA, 1974.

[50] Maurizio Rebaudengo, Matteo Sonza Reorda, Marco Torchiano, and Massimo Violante. Soft-error detection through software fault-tolerance techniques. In *Proceedings of the 14th International Symposium on Defect and Fault-Tolerance in VLSI Systems*, DFT '99, pages 210–218, Washington, DC, USA, 1999. IEEE Computer Society.

[51] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[52] G.A. Reis, J. Chang, N. Vachharajani, S.S. Mukherjee, R. Rangan, and D.I. August. Design and evaluation of hybrid fault-detection systems. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 148 – 159, june 2005.

[53] A.M. Saleh, J.J. Serrano, and J.H. Patel. Reliability of scrubbing recovery-techniques for memory systems. *Reliability, IEEE Transactions on*, 39(1):114 –122, apr 1990.

[54] N.R. Saxena and E.J. McCluskey. Control-flow checking using watchdog assists and extended-precision checksums. *Computers, IEEE Transactions on*, 39(4):554 –559, apr 1990.

[55] M.A. Schuette and J.P. Shen. Exploiting instruction-level parallelism for integrated control-flow monitoring. *Computers, IEEE Transactions on*, 43(2):129 –140, feb 1994.

[56] Xiaoxuan She and P.K. Samudrala. Selective triple modular redundancy for single event upset (seu) mitigation. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 344 –350, 29 2009-aug. 1 2009.

[57] J P Shen and M A Schuette. On-line selfmonitoring using signatured instruction streams. In *In ITC*, pages 28–2, 1983.

[58] Philip P. Shirvani and Edward J. McCluskey. Fault-tolerant systems in a space environment: The crc argos project. pages 98–2. A, 1998.

[59] Daniel P. Siewiorek and Robert S. Swarz. *Reliable computer systems (2nd ed.): design and evaluation*. Digital Press, Newton, MA, USA, 2. edition, 1992.

[60] W. Snoeys, F. Faccio, M. Burns, M. Campbell, E. Cantatore, N. Carrer, L. Casagrande, A. Cavagnoli, C. Dachs, S. Di Liberto, F. Formenti, A. Giraldo, E.H.M. Heijne, P. Jarron, M. Letheren, A. Marchioro, P. Martinengo, F. Meddi, B. Mikulec, M. Morando, M. Morel, E. Noah, A. Paccagnella, I. Ropotar, S. Saladino, W. Sansen, F. Santopietro, F. Scarlassara, G.F. Segato, P.M. Signe, F. Soramel, L. Vannucci, and K. Vleugels. Layout techniques to enhance the radiation tolerance of standard cmos technologies demonstrated on a pixel detector readout chip. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 439(2-3):349 – 360, 2000.

[61] J.R. Srour, C.J. Marshall, and P.W. Marshall. Review of displacement damage effects in silicon devices. *Nuclear Science, IEEE Transactions on*, 50(3):653 – 670, june 2003.

[62] A.J. Tylka, Jr. Adams, J.H., P.R. Boberg, B. Brownstein, W.F. Dietrich, E.O. Flueckiger, E.L. Petersen, M.A. Shea, D.F. Smart, and E.C. Smith. Creme96: A revision of the cosmic ray effects on micro-electronics code. *Nuclear Science, IEEE Transactions on*, 44(6):2150 –2160, dec 1997.

[63] K. Wilken and J.P. Shen. Continuous signature monitoring: efficient concurrent-detection of processor control errors. In *Test Conference, 1988. Proceedings. New Frontiers in Testing, International*, pages 914 –925, sep 1988.

[64] K. Wilken and J.P. Shen. Continuous signature monitoring: low-cost concurrent detection of processor control errors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 9(6):629 –641, jun 1990.

[65] S.S. Yau and Fu-Chung Chen. An approach to concurrent control flow checking. *Software Engineering, IEEE Transactions on*, SE-6(2):126 – 137, march 1980.

[66] Q. Zhao, Y. Ichinomiya, M. Amagasaki, M. Iida, and T. Sueyoshi. A novel soft error detection and correction circuit for embedded reconfigurable systems. *Embedded Systems Letters, IEEE*, 3(3):89 –92, sept. 2011.

[67] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Y. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. J. O'Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus. Ibm experiments in soft fails in computer electronics (1978;1994). *IBM Journal of Research and Development*, 40(1):3 –18, jan. 1996.

[68] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. 206(4420):776–788, 1979.