CHALMERS
UNIVERSITY OF TECHNOLOGY

SEU Mitigation Techniques for Advanced Reprogrammable FPGA in Space

Master's thesis in Embedded Electronic System Design

FREDRIK BROSSER
EMIL MILH

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2014

Fredrik Brosser, Emil Milh
© Fredrik Brosser, June 2014.
© Emil Milh, June 2014.
Examiner: Per Larsson-Edefors
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone + 46(0)31-772 1000
Department of Computer Science and Engineering
Gothenburg, Sweden June 2014

Abstract

FPGAs are becoming increasingly attractive for use in space applications due to their reconfiguration and signal processing capabilities, as well as their increasing speed and capacity. Traditional SRAM-based FPGAs, however, are highly sensitive to the ionising radiation environment in space, making them prone to radiation-induced memory upsets. In this thesis, design techniques for mitigating such upsets are implemented, tested and evaluated, with the purpose of enabling a replacement of conventional radiation-hardened or antifuse FPGAs with Xilinx commercial SRAM-based FPGAs.

A test framework using an exchangeable payload is developed for this purpose and run on a Xilinx Virtex-5 FPGA. A payload application is selected and used to test and compare the gains and costs related to different levels of redundancy and different FPGA configuration memory scrubbing methods. In comparing soft error mitigation methods, test results for availability, resource usage, mean time to failure and faults in time are considered. Realistic satellite orbit and radiation scenarios are considered, and a complete example application is presented.

The product of this work is a set of recommendations regarding the use of commercial SRAM-based FPGAs in space applications.

Table of Contents

Acronyms

AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
CLB	Configurable Logic Block
CMOS	Complementary Metal-Oxide Semiconductor
COTS	Commercial Off-The-Shelf
CRC	Cyclic Redundancy Check
CREME	Cosmic Ray Effects on Micro-Electronic Circuits
DDR	Double Data Rate
DFF	D-type Flip-Flop
DLP	Delay-Locked Loop
DPR	Dynamic Partial Reprogramming
DSP	Digital Signal Processing
DUT	Design Under Test
DWC	Duplication With Comparison
ECC	Error Correcting Code
EDAC	Error Detection and Correction
EEPROM	Electrically Erasable Programmable Read-Only Memory
FIT	Faults In Time
FPGA	Field-Programmable Gate Array
GEO	Geostationary Orbit
I/O	Input/Output
ICAP	Internal Configuration Access Port
JTAG	Joint Test Action Group (IEEE 1149.1 Standard)
LD+LR	Logic Decomposition, Logic Restructuring
LEO	Low Earth Orbit
LET	Linear Energy Transfer
LUT	Look-Up Table
MBU	Multiple Bit Upset
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
MTTR	Mean Time To Repair
NRE	Non-Recurring Engineering
PAR	Place And Route
PIP	Programmable Interconnect Point
PLL	Phase-Locked Loop
SBU	Single Bit Upset
SEB	Single Event Burnout
SECDED	Single Error Correction, Double Error Detection
SEE	Single Event Effect
SEFI	Single Event Functional Interrupt
SEGR	Single Event Gate Rupture
SEL	Single Event Latch-up
SET	Single Event Transient
SEU	Single Event Upset
SPENVIS	Space Environment Information System
SRAM	Static Random Access Memory
TID	Total Ionising Dose
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver/Transmitter
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

List of Figures

List of Tables

1 Introduction

FPGA devices, or *Field Programmable Gate Arrays*, have increased steadily in capacity and complexity over the last decade. The re-programmable and reconfigurable capabilities of FPGAs, their suitability for signal processing applications and the increasing capacity of such devices have made them increasingly attractive as alternatives to *Application Specific Integrated Circuits* (ASICs) in space applications. With recent *Static Random Access Memory* (SRAM) based FPGA devices comprising up to 2M logic cells and thousands of I/O pins [1], it is no longer feasible to disregard FPGA technology for space applications, especially when considering the high non-recurring engineering costs (NRE) involved in developing custom ASIC designs and the low production volumes typical for space applications.

Operating in a space environment raises a number of issues that need to be taken into consideration when designing a system, among them the effects on digital systems by the ionising radiation environment in space. Radiation can negatively impact the lifespan, performance and reliability of a digital system or device[2], [3]. While there are radiation resistant FPGA devices on the market, these are far behind in performance and capacity compared to standard, commercially available SRAM-based FPGAs, as well as being overly expensive for many projects.

While packaging and manufacturing techniques are important aspects of designing systems for space applications, it is often not the whole solution[2], and designing for fault-tolerance is becoming an increasingly important factor. The level of fault-tolerance required differs widely from application to application, as does the timing requirements. For simpler types of sensors and monitoring equipment such as cameras, real-time and availability requirements may be more relaxed. On the other side of the spectrum, high levels of reliability and hard timing constraints are necessary for mission critical systems such as on-board computers or communication. A wide range of applications exist in between these extremes, where SRAM-based FPGAs can prove useful.

1.1 FPGA technologies

There are three major types of FPGA technologies on the market: SRAM-, Flash- and Antifuse-based. SRAM-based FPGAs are reprogrammable, in theory an infinite number of times. Benefiting from SRAM and CMOS-process research in other parts of the semiconductor industry, SRAM-based FPGAs are at the forefront of FPGA technology in terms of integration level. An important factor is that SRAM-based FPGAs are manufactured in standard CMOS processes, giving the potential for high density devices. This allows for a large number and variation of resources to be available on-chip, such as memory, hard DSP blocks and embedded multipliers. SRAM-based FPGAs are the most common type for commercial applications and have the highest capacity and performance of the FPGA technologies discussed here. However, SRAM is a volatile storage format, and SRAM-based FPGAs need to be reprogrammed at power-up. This requires an off-chip storage solution for the programming bit stream, commonly in EEPROM or Flash on the board, adding to the number of components and complexity of the system. The two major manufacturers of SRAM-based FPGAs are Altera and Xilinx.

Compared to radiation-hardened processors conventionally used in space applications, FPGAs can constitute a highly versatile and high-performing alternative, especially considering the reprogrammability and capability for parallelisation. SRAM-based FPGAs also have the advantage over custom ASIC to be reprogrammable thus avoiding all development costs associated with ASIC development.

Flash-based FPGAs are a non-volatile alternative to SRAM-based FPGAs, based on so called *floating gates*. The non-volatile nature of flash enables live-on-startup FPGAs without the need for reprogramming, and the flash technology is intrinsically more resistant to radiation compared to SRAM. While flash technology has the advantage of smaller bit storage cells, requiring only one or two transistors to implement a configuration bit storage element compared to the five to six transistors used in SRAM, it lags behind SRAM in manufacturing process technology. Fig. 1 shows the principle structure of a memory cell in flash technology, using an isolated *floating gate*. Floating gates are programmed by *tunnel injection*, and then left in a *floating* state. This can be compared to the standard SRAM cell, illustrated by Fig. 2. One of the drawbacks of Flash-based FPGA technology is the gradual degradation of configuration memory cells due to charge build-up when reprogramming, limiting the number of times it is possible to reprogram the FPGA. This number, however, is in the order of hundreds of times, and is typically not an issue for space applications. Absorbed radiation over time also leads to charge build-up in the floating gate, eventually rendering the storage cell unusable. This means that flash-based FPGAs in general have a lower acceptable total accumulated radiation dose compared to SRAM-based FPGAs, which is a highly relevant factor for space applications. Also, charge leakage is a problem in flash-based FPGAs, where charge can leak from the floating gate through the insulating material surrounding it.

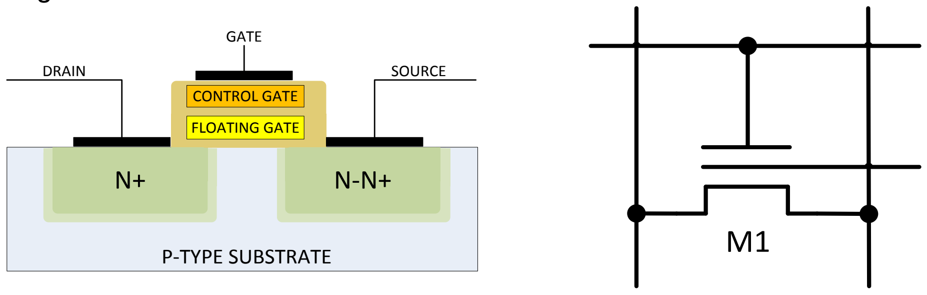


Fig. 1. Flash Memory Cell

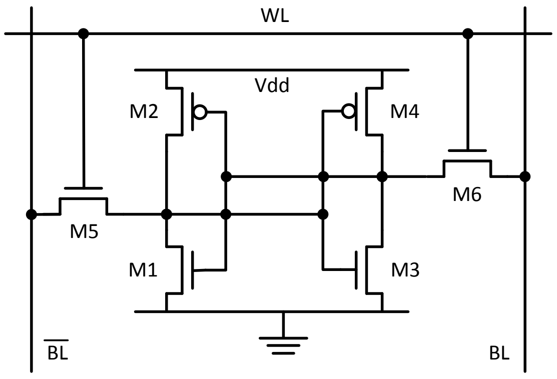


Fig. 2. SRAM Cell

SRAM FPGAs are sensitive to radiation-induced upsets in both their configuration and user memory [4],[5]). This requires a different approach to upset mitigation compared to ASICs, where the designer only needs to consider radiation-induced upsets in latches and user memory cells. Flash-based FPGAs are more resistant to radiation, as previously mentioned, but include SRAM-based components, mainly in user memory such as D-type Flip-Flops, which are sensitive to upsets. As a third alternative, *Antifuse-based FPGAs* have a distinct advantage in this area. Antifuse-based FPGAs have traditionally been used in space applications, and are based on one-time programmable antifuse connections. They are less susceptible to radiation-induced errors since the need for configuration bits for each individual interconnect point is eliminated, giving a sort of intrinsic radiation hardening for the configuration. This is also the antifuse technology's greatest disadvantage: once a fuse is "blown" by supplying a large current during programming, it cannot be reprogrammed. This makes antifuse-based FPGAs one-time programmable devices. Antifuse FPGAs are also expensive in relation to the performance they offer.

Table 1 gives an overview of the features of the different FPGA technologies discussed, and is meant as a quick comparison of the main features, advantages and drawbacks of the technologies. Here, capacity refers to the density and amount of logic that can be synthesised onto a single FPGA. It should also be noted that standard COTS SRAM-based FPGAs are generally cheaper and more available in comparison to their capacity. The work investigates the possibility of leveraging this cost-to-capacity ratio of SRAM-based FPGAs in space applications by using techniques for mitigating radiation-induced errors. Also, in Table 1, it might seem counter-intuitive that SRAM-based FPGAs have the largest memory cell size of the compared technologies, but at the same time the highest device capacity. This is mainly because of the difference in manufacturing process technology between the categories.

Table 1. Comparison of FPGA Technologies

Feature	SRAM	Flash	Antifuse
Reprogrammable	Yes	Yes	No
Volatile Configuration	Yes	No	No
Live On Startup	No	Yes	Yes
Memory Cell Size	Large	Small-Medium	Small
Radiation Sensitivity	**High**	Low-Medium	Low-None
Capacity	**High**	Medium	Low
Reprogramming Speed	Fast	Slow-Medium	N/A
Total Dose Tolerance	Medium-High	Low	High

1.2 Problem Statement

The core aim of the work described in this report is to investigate the feasibility of using Xilinx's commercial SRAM-based FPGAs in space applications, with respect to radiation-induced *soft-error* tolerance. Soft errors will be discussed in detail later in this report. The potential gains from using standard, commercial SRAM-based FPGAs rather than radiation hardened alternatives are higher capacity, better performance and lower cost. While these devices are used in space applications today, they are mostly part of non-critical systems, where temporary loss of data or reconfiguration downtime is acceptable. Radiation hardened versions of Xilinx SRAM-based FPGAs are available for Virtex-5 devices, but not for the Virtex-6 and Virtex-7 series. Previous generations of radiation hardened FPGAs from Xilinx have been proven in missions, notably in the NASA Mars Rover mission. However, the radiation hardened Virtex-5 devices are very expensive components.

1.3 Scope

The focus of this work is on Xilinx devices, mainly because Xilinx FPGAs are essentially free from latch-up effects up to a certain dose of radiation specified for the specific device. SRAM-based FPGAs are, as previously mentioned, manufactured using standard CMOS processes, and are typically one or more generations ahead of Flash and Antifuse FPGAs in terms of process node. Xilinx is also currently the market-leading vendor for SRAM-based FPGAs in general, and for aerospace grade SRAM-based FPGAs in particular. Altera currently has no radiation hardened products on the market. The suitability of Xilinx SRAM-based FPGAs will vary between applications, depending on the required level of fault-tolerance and availability, performance requirements and the power, size and complexity budget of the system as a whole.

In determining the feasibility of using these devices in space applications, no one single mission profile or target application will be specified. Instead, the aim is to evaluate different mitigation techniques, on their own or in combinations, to form a general recommendation which will then have to be adapted to the target application and mission. As mentioned earlier, different missions and (sub-) systems have different tolerances for downtime and error rates. It should be noted that some (most) of these techniques introduce hardware overhead. Overhead could be in the form of resource overhead on the FPGA, or as added overall system complexity. This may limit the gains in capacity and performance to be had from using SRAM-based FPGAs. This work will use Xilinx Virtex 5 as a starting point when discussing FPGA architecture and mitigation techniques for soft errors. [6] gives an excellent overview of the available mitigation techniques as well as an introduction to SEU related error modes in FPGA.

2 Background and Related Work

This section will give a background of the theory used in this report. Models and expressions used in this report will be explained as well.

2.1 Radiation

Providing radiation tolerance for microelectronics is a big challenge, and an active field of research. Knowledge from different engineering disciplines needs to be applied in order to solve the problem as efficiently as possible. An introduction to radiation is provided in this section to give an understanding of the terminology and methods.

2.1.1 Radiation types and measurements

Radiation is common in space environments. When the energy transferred by incoming radiation exceeds the energy of a particular electron in an atom, it is called *ionising radiation* [7]. The name comes from the fact that exposure to this type of radiation may ionise the electrons in an atom.

2.1.1.1 Common types of radiation

Radioactive elements comprise nucleus which are unstable. An unstable nucleus decays over time to a more stable state. When such a nucleus decays, it moves from a state of higher energy to a lower energy by the emission of energy. This energy is what is referred to as radioactivity. Radiation can be divided into three parts; α- (alpha), β- (beta) and γ- (gamma) radiation.

α -radiation is in essence the nucleus of helium. This type of radiation is charged and is therefore affected by magnetic fields, such as the magnetic field surrounding Earth. Heavier radioactive elements are often prone to emission of α-radiation. α -radiation loses energy rapidly when colliding with other materials. Therefore α-radiation has a range of a few centimeters in air and is easily shielded against even by thin shielding materials. α -particles can have energy levels of up to 7 MeV.

Electrons, or positrons, are what constitute β-radiation. β –radiation is also affected by magnetic fields since it is charged negatively in the case of electrons and positively in the case of positrons. The energy of a β-particle is often much smaller than that of an α-particle. β -particles can also have different energies depending on the neutrino particle, which is emitted in conjunction with the β-particle. Most β-particles have energy levels smaller than 1 MeV.

γ -radiation differs from the aforementioned types of radiation in that it can be perceived as radiation consisting of particles called photons or quanta but also as electromagnetic waves. In the case of γ-radiation being perceived as electromagnetic waves, it will have a wavelength λ shorter than 10pm. It should be noted that there are electromagnetic waves with higher wavelengths which are still considered to be radiation, such as ultra-violet radiation (λ < 120 nm) and x-rays (λ < 200 pm) [8].The frequency of electromagnetic radiation is dependent on wavelength according to (1) where λ is the wavelength, c is the speed of light in vacuum and f is the frequency. The energy of electromagnetic radiation can be calculated according to (2) where h is Planck's constant and E is the energy.

$$f = \frac{c}{\lambda} \tag{1}$$

$$E = hf \tag{2}$$

γ -radiation also differs in the way it is absorbed by different materials. When a γ-particle hits a material, it is slowed down due to the photoelectric effect, electron-hole pair generation and Compton scattering. The latter is the process where a photon collides with an electron bound in an atom releasing it from its bond. The release of said electron is what ionises the atom. γ -radiation has the ability to penetrate materials much deeper than the aforementioned types. The energy of γ-radiation is defined to be in the interval of 0.1-1.5 MeV [9].

Radiation originating from outer space is called *Galactic Cosmic Rays (GCR)*. Roughly 90% of GCR are protons, or hydrogen nuclei. Approximately 9% are α-particles and an additional 1% comprises electrons, β-particles and a small fraction of nuclei of heavier elements [2].

2.1.1.2 Measurements of radiation

One type of measurement used in conjunction with radiation is the *absorbed dose* or *total ionising dose (TID)* [10]. The absorbed dose is often measured in Gray (Gy), or less frequently, rad. 1 Gray of absorbed dose corresponds to an absorbed energy of 1 Joule per Kg. Where the dose of radiation has a biological significance, it is important to factor in the type of radiation in addition to the energy. This is done by the use of a *radiation weighting factor* w_R, which yields what is called the *equivalent dose*, as shown in (3), where H is the equivalent dose in *Sievert* and D is the absorbed dose in Gray. The equivalent dose has the same dimension as Gray which makes the weighting factor dimensionless. A higher weighting factor implies a higher biological hazard [9].

$$H = D \cdot w_R \tag{3}$$

When different types of radiation hit materials, energy is deposited from the radiation into the material. A common way of modelling this energy deposition is through *Linear Energy Transfer (LET)* [9]. LET is defined as the energy loss per unit length per density, as shown in (4). A common unit for LET is $\frac{\mathrm{MeVcm^2}}{\mathrm{mg}}$.

$$LET = \frac{dE}{dx}/\rho \tag{4}$$

LET is most commonly used to describe the energy deposition in different materials, but also in other contexts [10]. The LET for each particle has a unique dependence on its energy. There are a number of different particle effects which consume energy, but do not contribute to LET. Effects such as displacement damage, radiative losses, nuclear losses and bremsstrahlung will influence the amount of energy transferred through LET. Increasing the energy in particles might therefore not necessarily yield a higher deposited energy through LET [8]. Nuclei of heavier elements and α –particles usually have a higher LET [10], which is why they are often referred to as *High-LET particles* [11]. *Low-LET particles* comprise other types of radiation such as β- and γ-particles.

Effective linear energy transfer (LET_{eff}) is sometimes used to calculate the potential LET when the characteristics have been determined with a perpendicular particle beam. Since length of the travelled path increases with an increased angle of incident, there will also be a larger energy transfer. LET_{eff} can be resolved as a function of LET and the angle of incident θ according to (5).

$$LET_{eff} = \frac{LET}{\cos(\theta)} \tag{5}$$

Fluence is a flux integrated over a given time interval. The particle fluence defines the number of particles passing through a cross section of a sphere in a given amount of time [2]. Particle fluence as a function of LET is often used to describe the distribution of the LET values of particles in specified locations or paths in space. These graphs are often referred to as *LET spectra*.

2.1.2 Space environment

Earth's atmosphere is commonly divided into troposphere, stratosphere, mesosphere, thermosphere and exosphere, as shown in Fig. 3. The thermosphere is exposed to the full radiation spectra emitted by the sun. 35% of the radiation goes to increasing the heat of neutral particles in the thermosphere, and an additional 20% is consumed by oxygen as it gets dissociated. The remaining 45% of the radiation is reradiated as ultra-violet radiation. Because of this, there are elevated radiation levels beyond the thermosphere [12].

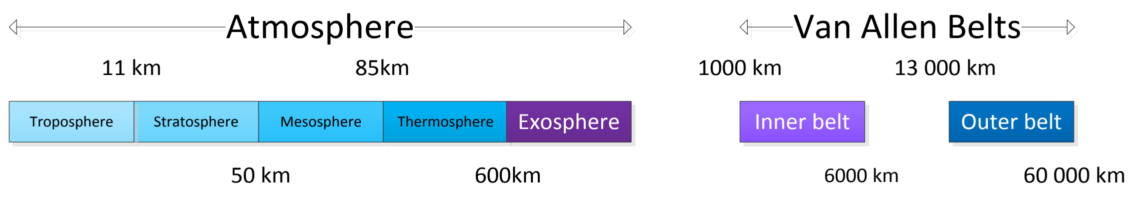


Fig. 3. The different atmospheric layers and Van Allen belts.

Due to the magnetic poles on Earth, a magnetic field spans around Earth from the magnetic North Pole to the magnetic South Pole. As most radiation particles are charged, many particles become trapped in the magnetic field surrounding Earth. At specific distances from Earth, the radiation particles gather more densely. These fields are called *Van Allen belts*. There is an outer and an inner Van Allen belt, as shown in Fig. 4. Due to the nature of a spherical magnetic dipole, there will be a stronger magnetic flux closer to Earth which decreases with the distance from Earth. The concentration of protons decreases with increasing distance from Earth. The inner Van Allen belt is therefore dominated by protons while the outer belt is dominated by electrons [13].

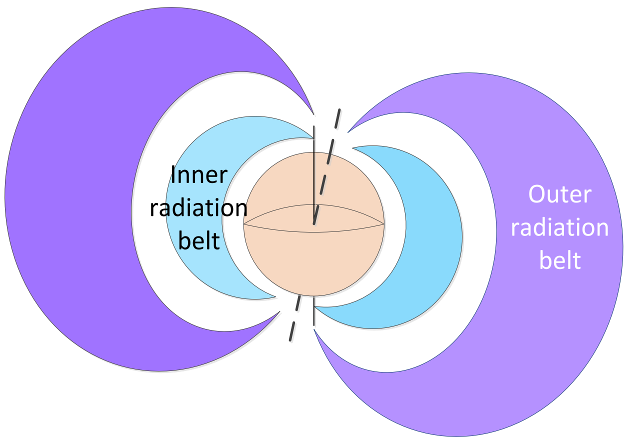


Fig. 4. The inner and outer Van Allen belts following Earth magnetic fields

As mentioned in Sec. 2.1.1.2, particle fluence as a function of LET is commonly used to describe particle density and particle composition at different distances and orbits. The LET spectrum for circular orbits at six different altitudes above Earth is given in Fig. 5. In Fig. 5 it is illustrated that the fluence of lower LET particles decreases with an increasing distance from Earth. This is due to the fact that the concentration of protons decreases significantly over the distances shown. Particles with a higher LET occur more frequently at higher altitudes, which is caused by the increasing GCR levels.

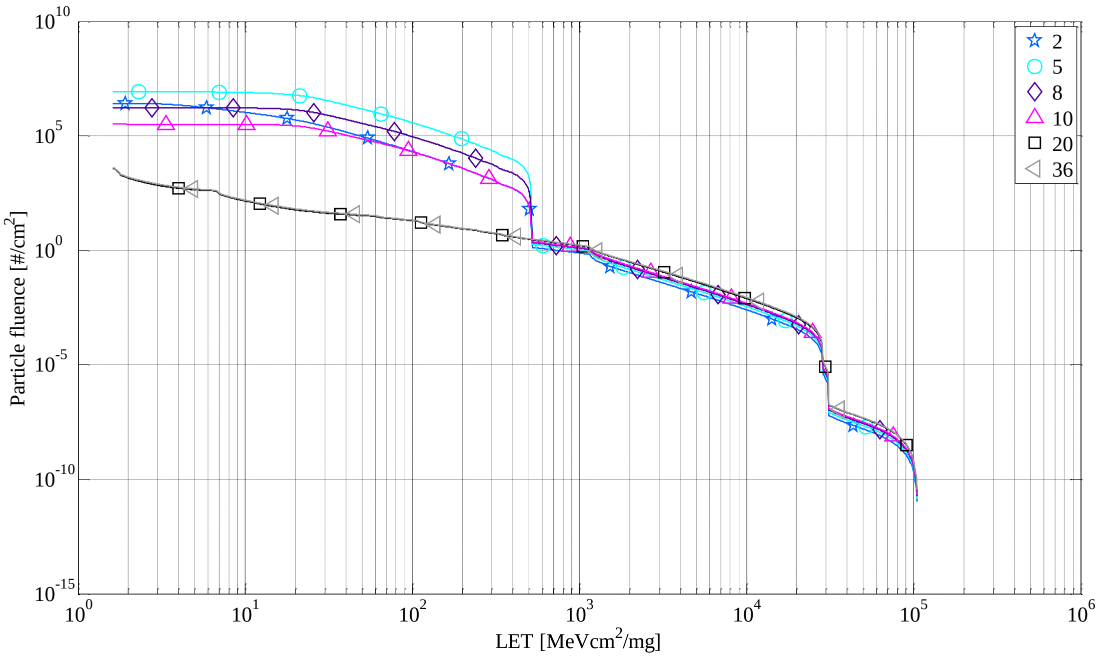


Fig. 5. LET spectrum for circular orbits at different altitudes.

The altitude unit is 10^6m.

2.1.3 Satellite Orbits

Depending on its orbit, a satellite will be subjected to varying levels of radiation. All spacecraft and man-made objects intentionally put into orbit are included in this category. A simplified definition of a satellite orbit is the path of a satellite around a point or body in space (here: Earth) naturally curved by the gravity of the body. Satellite orbits follow Kepler's laws of planetary motion, and are typically elliptical. An orbit is characterised by a number of *orbital elements*, including its semi-major axis (a), eccentricity, inclination (i), argument of perigee (ω), true anomaly and the right ascension of the ascending node (Ω). These are illustrated in Fig. 6.

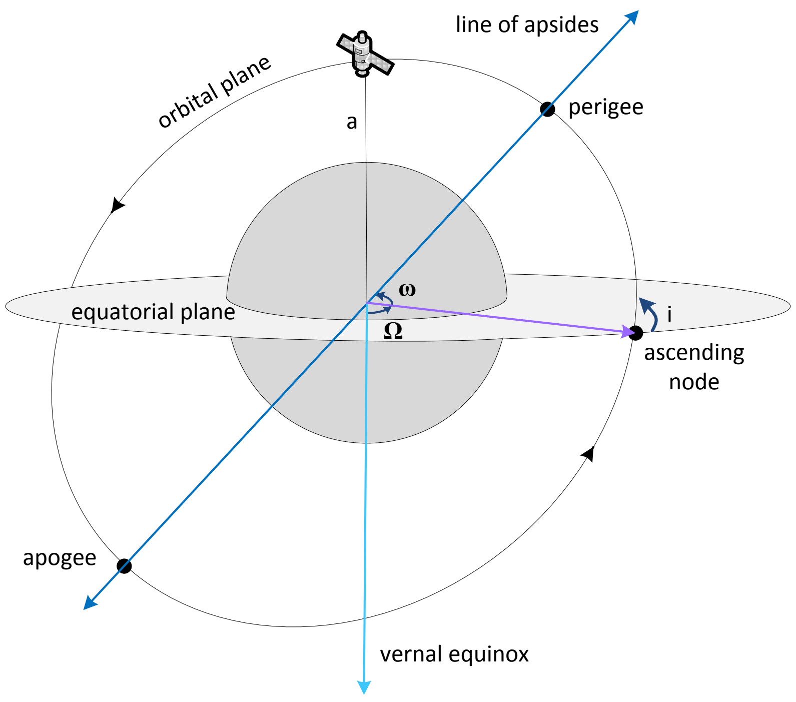


Fig. 6. Satellite orbit measurements.

Satellites in *Low-Earth Orbit* (LEO) typically operate at altitudes of a few hundred (400-800) km, but the range includes all satellites at altitudes from 160 to 2000 km over Earth's surface. A satellite in LEO experiences drag from the Earth's thermosphere, but is also to a certain degree protected from deep space radiation by Earth's magnetic field and the thermosphere. Satellites in LEO are subjected to the inner Van Allen belt radiation at points where the radiation belt is closer to Earth, such as when passing through the *South Atlantic Anomaly* (SAA) or during intense solar flares [14].

Satellites in *Geostationary Orbit* (GEO) have a constant altitude of 35,786 km above Earth's surface and appear to be stationary over a point on Earth. A satellite in GEO will not be affected by the trapped protons in the inner Van Allen belt, but is otherwise totally exposed to the space radiation environment.

2.1.4 Radiation Characteristics of Devices

The radiation in space can affect electronic systems negatively. When charge from radiation particles is deposited into a device, it has the potential of altering the internal state of, or damaging, the device. Such occurrences are referred to as *Single Event Effects (SEE)*.

2.1.4.1 Single event effects

Radiation particles with sufficient LET have the potential of introducing an SEE. When a particle with sufficient LET hits a device, it ionises the atoms along its propagation path, as shown in Fig. 7. This ionisation results in a deposited charge which has the potential to cause an SEE. It is usually nuclei that have sufficient energy and LET that cause direct ionisation in devices.

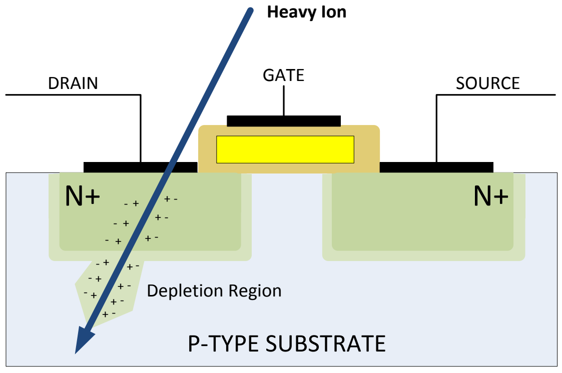


Fig. 7. A particle striking a transistor and creating an ionisation path

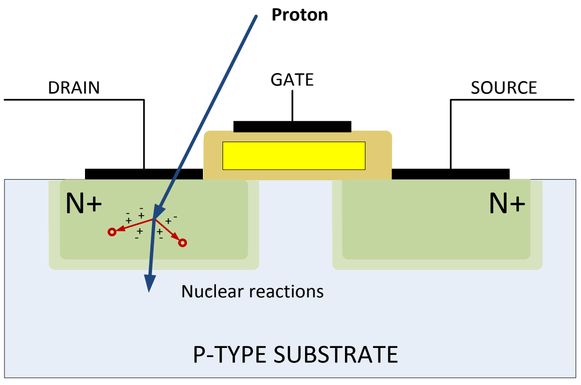


Fig. 8. A proton striking a transistor and inducing nuclear reactions

Protons, which in general have a lower energy and LET than nuclei, may also cause SEEs, although not by direct ionisation. Protons induce *nuclear reactions* which in turn have the potential of causing an SEE, as shown in Fig. 8. When protons collide with atoms, there is a probability of approximately 10^{-5} of a nuclear reaction occurring. Furthermore it is estimated that protons with energy levels of approximately 20 MeV deposit the largest amount of energy through indirect ionisation [2]. The proton fluence for circular orbits at different altitudes is shown in Fig. 9 where the altitude above Earth is given in km. As the proton flux is larger than CGR flux for the orbits at lower altitude, indirect ionisation through proton strikes are the dominating cause of SEEs for such orbits [2]. Fig. 9 shows that the fluence for 20 MeV protons is larger at an altitude of 4000 km compared to 2000 km and 8000 km.

Different types of SEEs can occur and they can be divided into *Soft errors* and *Hard errors.* Hard errors are permanently damaging effects and cannot be reversed by resetting or power cycling the system. *Single event induced burnout (SEB)* and *Single event gate rupture (SEGR)* are both examples of hard errors. These errors are likely to cause failures, either locally or for the whole device, as CMOS logic relies on complementary behaviour among transistors.

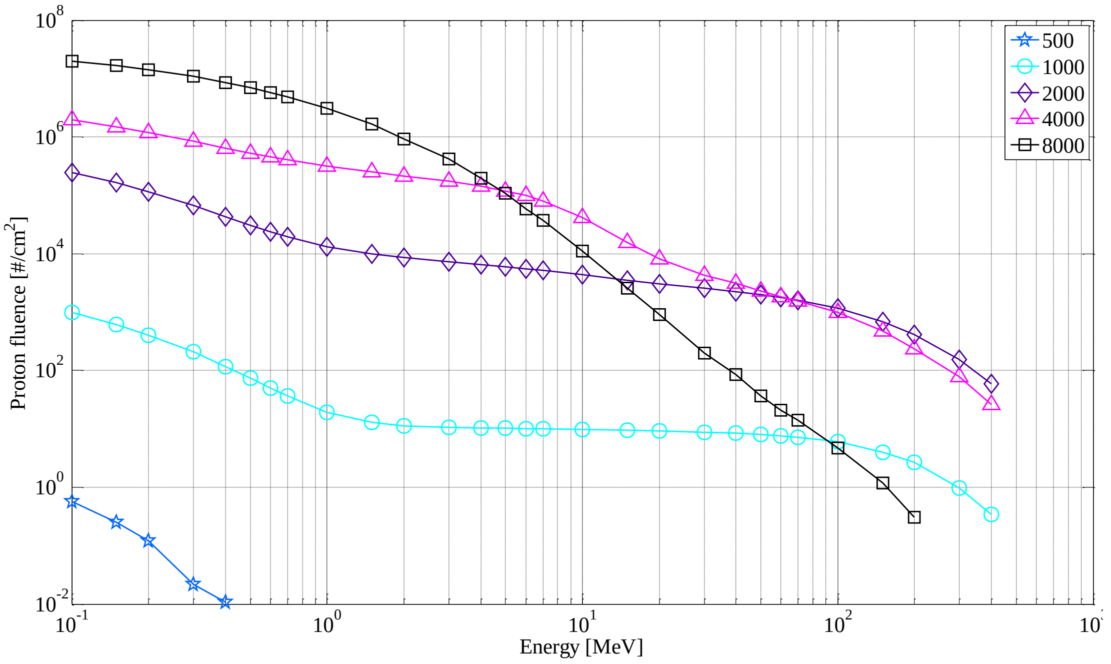


Fig. 9. Proton energy spectrum for different altitudes.

Soft errors, on the other hand, could indicate inverted data in storage elements or another reversible effect. *Single event upsets (SEU)* is an example of such an error where it indicates an inverted value in a storage element. An SEU implies that a memory element has got struck by a radiation particle after which the incident flips a bit. Vulnerable storage elements could be a variety of different kinds. It should, however, be noted that different types of memories have different sensitivities to SEU.

A special type of SEU is the *Single Event Functional Interrupt (SEFI)* which takes place when the basic functionality of the system is interrupted due to the upset. An example of an SEFI would be if an upset affected the clock tree, the communication interfaces or other essential parts. A *Single Event Transient (SET)* is an event where a particle deposits its energy into what becomes a time-limited pulse on a signal path or a wire. Depending on the instance of impact, an SET may be clocked into a memory element, or may be harmless if not stored or noted.

There is also a category of errors which are usually considered hard errors but can be corrected with a power cycle if current ratings are not exceeded. *Single Event Induced Latchup (SEL)* and *Single Event Induced Snapback (SES)* are examples of these types of errors [2]. The different errors mentioned are concluded in Table 2.

Table 2. SEEs divided into hard and soft errors

Hard Errors	Soft Errors
SEGR	SEFI
SEB	SEU
(SEL)	SET
(SES)	

2.1.4.2 Measurements in device characteristics

Cross section, often denoted σ, is a measurement of the probability of an event or impact often used in particle or nuclear physics. For the purposes of calculating SEU rates, cross section is the probability of getting an SEU. Cross section is defined according to (6) where N is the number of observable events and F is the particle fluence. As mentioned in [2], fluence is a flux integrated over a defined time interval. The unit for fluence is therefore per unit area, commonly $\frac{1}{\mathrm{cm}^2}$. The resulting unit of cross section is simply cm^2.

A configuration of transistors where the respective outputs are connected to the opposite input is called a latch. By charging or discharging a gate in a latch, data can be stored. This makes the latch a basic memory element. When either gate of such a memory element is struck by a charged particle, a charge is deposited into the gate. Provided that the deposited charge is large enough, it will be capable of switching the state of the memory element.

$$\sigma = \frac{N}{F} \tag{6}$$

The lower bound for the charge required to switch a memory element is referred to as a *critical charge* and is denoted Q_c. It has been suggested that the critical charge has a quadratic dependence on feature size, as shown in (7), where Q_c is resolved in pC and L is given in µm [2]. As the critical charge decreases, the SEU susceptibility increases.

$$Q_c = 0.023L^2 \tag{7}$$

A critical charge measurement can be used to calculate the corresponding LET. This particular LET is called *linear energy transfer threshold (LET_{th})* and can be calculated as shown in (8), where w_s is the *electron-hole pair generation energy constant* which is specific to each material. ρ is the material density, q is the *elementary charge* and d is the particle travel distance.

$$LET_{th} = \frac{Q_c w_s}{\rho q d} \tag{8}$$

LET_{th} can be used to make a first order approximation of the cross section. This type of approximation is known as the *critical charge method*. The principle of the critical charge method is to model the cross section as a step function according to (9) [15]. Alternatively, a more detailed cross-section data collection can be used in which all of the data points can be used. The latter method is often referred to as the *integral flux method* [2]. A comparison between the cross section of the two methods is shown in Fig. 10. It should be mentioned that similar methods exist which apply similar calculations but have different names. Among these are the methods which include the approximation of *sensitive volumes*. Sensitive volumes indicate what fraction of the device is susceptible to SEUs [2].

$$\begin{cases} LET < LET_{th} = 0 \\ LET \geq LET_{th} = k \end{cases} \tag{9}$$

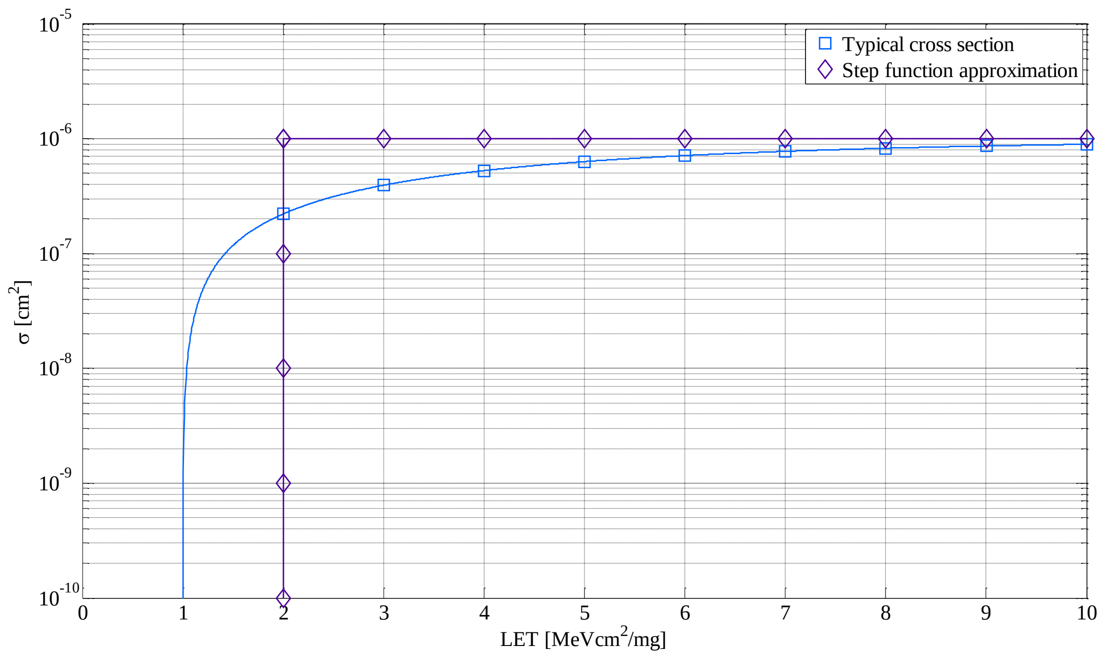


Fig. 10. Typical cross section characteristics used in approximations.

Regardless of which cross section model is used, the SEU rate can be calculated according to (10). M represents the number of SEU for the LET spectrum described by $F(x)$. $\sigma(x)$ represents the cross section of the device and the limits L_0 and L_{max} define the interval for which LET needs to be considered. There are several other formulations to calculate the SEU rate similar to (10), for example the Bradford, Pickel and Adams formulations [2].

$$M = \int_{L_0}^{L_{max}} \sigma(x)F(x)dx \tag{10}$$

2.1.5 Total Ionising Dose

The consequences of radiation are not solely limited to SEE. Over time an accumulative dose of radiation degrades the transistors of a CMOS circuit.

2.1.5.1 Positive oxide-trap charge

Standard MOSFETs are affected negatively by radiation. Radiation yields not only ionisation in transistors but also the creation of electron-hole pairs. Recombination of the electron-hole pairs occurs in parallel but a fraction remains nevertheless. This fraction is referred to as the *electron-hole charge yield*. Whenever electron-hole pairs emerge in an N-type MOSFET, the holes drift towards the channel at the Si/SiO_2 interface while the electrons are drawn to the gate. The accumulated amount of holes in combination with a positive gate bias forms a *positive oxide-trap charge*. Trapped charges will influence the transistor channel by biasing it to conduct, increasing the static leakage current as illustrated in Fig. 11.

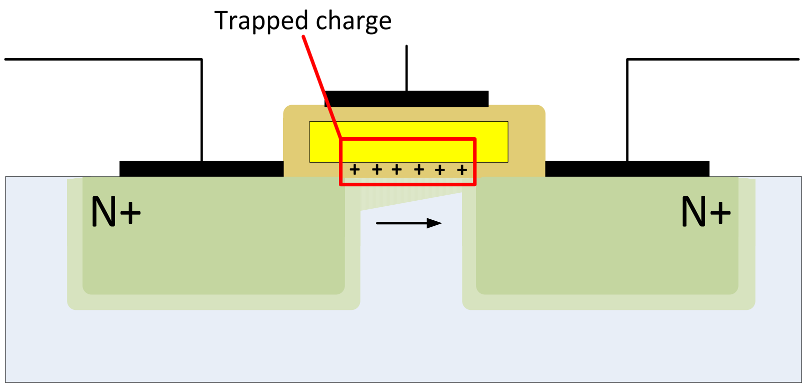


Fig. 11. Positive charge trapped in a positive oxide trap.

2.1.5.2 Displacement damage

Apart from the aforementioned possible effects, it is also possible for the crystalline silicon structure to take damage. *Displacement damage* is accumulated over time and means that atoms in the lattice structure are knocked out to leave vacancies and interstitial atoms behind. The consequence of displacement damage is the reduction in minority carrier mobility and lifetime.

Non-ionising energy loss (NIEL) is defined as the rate of energy loss due to displacements of atoms. NIEL is a common measurement when discussing displacements effects. Included in NIEL are the nuclear elastic collisions, the Coulomb elastic collisions and the inelastic collisions. *Displacement-damage dose* (D_d) is another metric with the unit of energy per weight, the same as Gray. The displacement-damage dose can be calculated as the product of NIEL and particle fluence [16].

2.2 FPGA Architecture and Sensitive Structures

In Xilinx FPGAs, the basic building blocks are CLBs, Configurable Logic Blocks. In Virtex 5 devices, the CLBs are made up of two logic slices which are independently connected to the general routing on the FPGA and to a carry chain structure [17]. There are two types of logic slices in Virtex 5, SLICEL and SLICEM. SLICEL can be seen as the basic logic slice type, and contains four 5-input look-up-tables, or LUTs, together with four D-type flip-flops(DFFs) and multiplexers for routing purposes. The LUTs can implement any 5-input logic function. SLICEM slices contain shift register functionality and provide the option of using the LUTs as distributed user RAM, as well as the basic resources described for SLICEL slices. When used as distributed RAM, LUTs are configured as memories for user data storage.

Other resources on the FPGA include Digital Clock Managers (DCM), Phase-Locked Loops (PLL), Block RAMs, DSP blocks, I/O blocks (IOBs) and buffers for connecting package pins. The FPGA resources are connected together by a configurable routing matrix. A common way of describing FPGAs is as configurable logic "islands" connected together by a "sea" of configurable routing paths.

When synthesising an FPGA design, the circuit function defined by the designer is mapped to these resources by synthesis tools. This mapping makes up the configuration of the device, and is stored in the SRAM-based configuration memory. The configuration memory defines the function and operation of all the described resources as well as the routing and connections on the FPGA, and can be seen as an underlying device definition layer. Fig. 12 gives an overview of a generic FPGA island-style architecture. The fold-out illustrates a simplified LUT-DFF pair inside a slide, inside a CLB. In this particular example the LUT implements an XOR function.

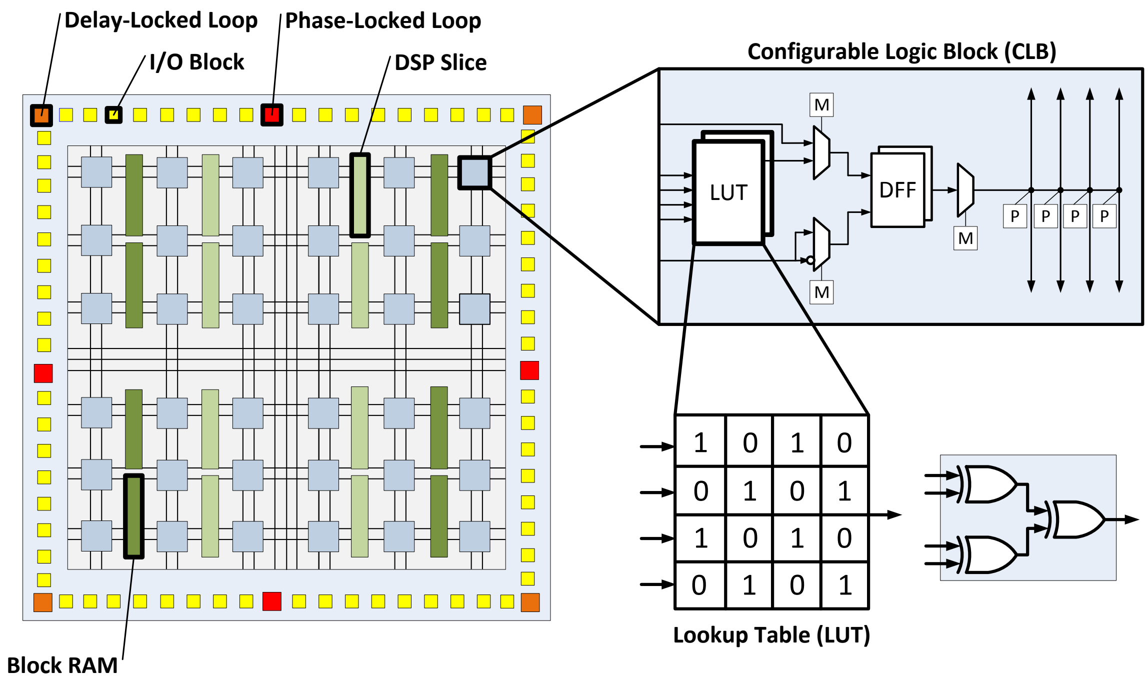


Fig. 12. FPGA Architecture Overview

Fig. 13 illustrates a LUT as a 4:16 decoder, and shows an illustration of the underlying configuration memory with each LUT bit stored in an SRAM cell.

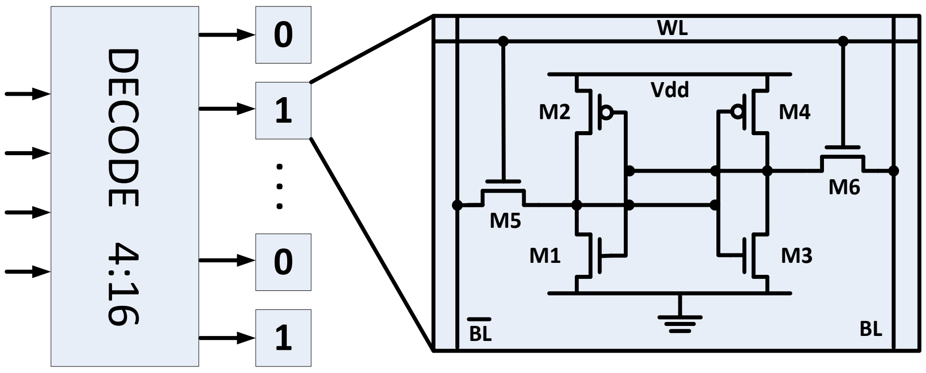


Fig. 13. LUT Configuration

The routing matrix and CLB-internal routing structures are made up of switchboxes, multiplexers, buffers and programmable interconnect points (PIP). All of these routing resources are configured by the corresponding bits in the configuration memory. Fig. 14 shows a section of the FPGA with four CLBs and their local interconnect matrix. The top fold-out shows an example of a connection box, with the crosses representing active, “on”, PIPs. Each PIP is configured to be active or inactive by a single bit in the configuration memory. The bottom fold-out shows a switchbox, with fully configurable connections between all vertical and horizontal connection lines.

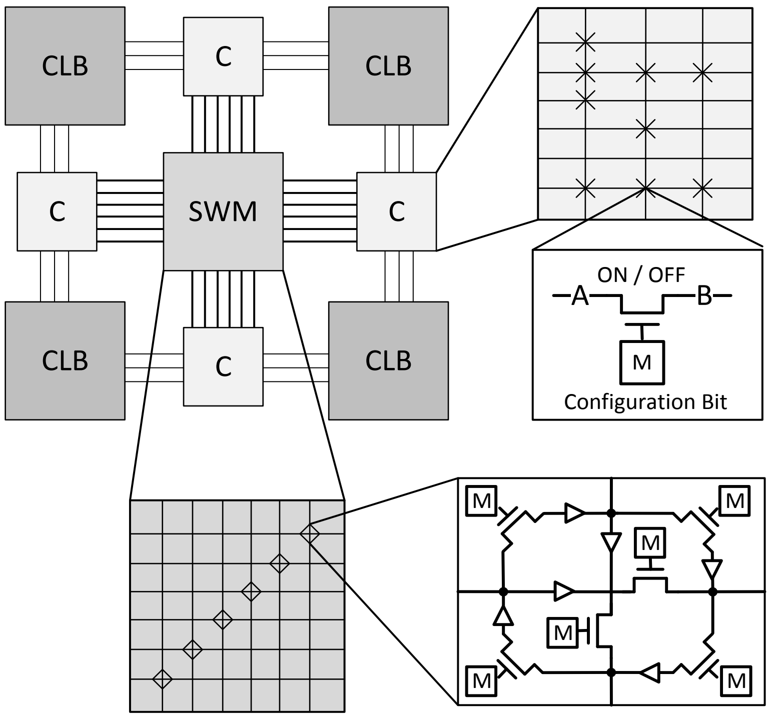


Fig. 14. Interconnect Matrix

SRAM-based FPGAs are programmed using a binary bit-stream, usually stored off-chip. For space applications, this off-chip configuration storage is usually in the form of a radiation-hardened EEPROM or Flash. Since the SRAM-based configuration memory is volatile, the bit stream has to be reprogrammed onto the FPGA on startup and power-cycling. The programming logic is responsible for writing the configuration memory via one of the configuration interfaces. The configuration interfaces allows programming, erasing, reading and verifying of the configuration memory, as well as performing functional and status tests on the FPGA. In Xilinx architectures, these configuration interfaces include JTAG, SelectMAP and ICAP [18].

JTAG is a serial, external interface available on almost all FPGA devices, commonly used for programming and debugging purposes. While JTAG is a comparatively slow, low bandwidth interface, it is easy to use and included in most IC debugging workflows. SelectMAP is a type of external parallel configuration port found on Xilinx FPGAs, and can provide a higher bandwidth compared to JTAG by using an 8- or 32-bit interface. Finally, ICAP, or Internal Configuration Access Port, is an internal configuration interface, similar to SelectMAP. ICAP can only be accessed by internal FPGA structures.

In order to program, modify or access the configuration memory, instructions are sent through a programming interface to configuration registers on the FPGA. All types of reconfigurations are hence being made through instructions sent to the configuration registers as shown in Fig. 15.

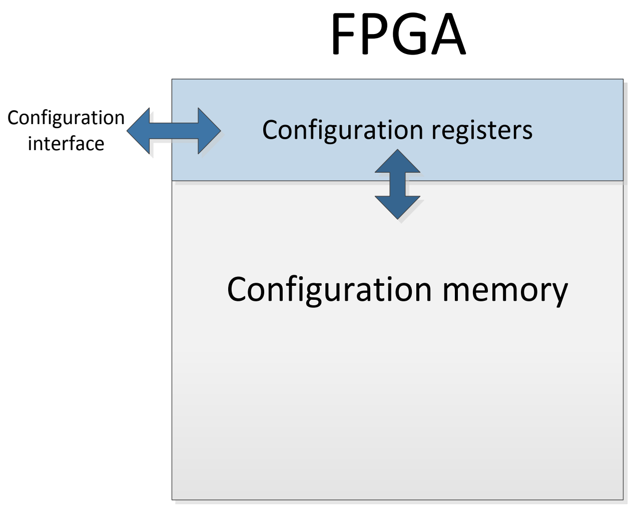


Fig. 15. Configuration Flow

There are 20 configuration registers in a Virtex 5 where each register has a unique purpose. The registers linked to the fundamental functions of the configuration interface comprise the Command Register (CMD), the Frame-Address Register (FAR) and the Frame-Data Input- and Output Registers (FDRI, FDRO). Whenever a request is made or when a command is sent, it is sent to the CMD. When reading or writing to the configuration memory, the frame address is specified in FAR. Data is written to FDRI and read from FDRO.

Every instruction sent to the configuration registers is made out of 32-bit words. Instructions can be sent as a *type-1* or *type-2 packet*. A type-1 packet comprises a 1-word header followed by a varying number of data words. A type-2 packet is only sent after a preceding type-1 packet, as shown in Fig. 16. Type-2 packets are used whenever a large number of words are sent.

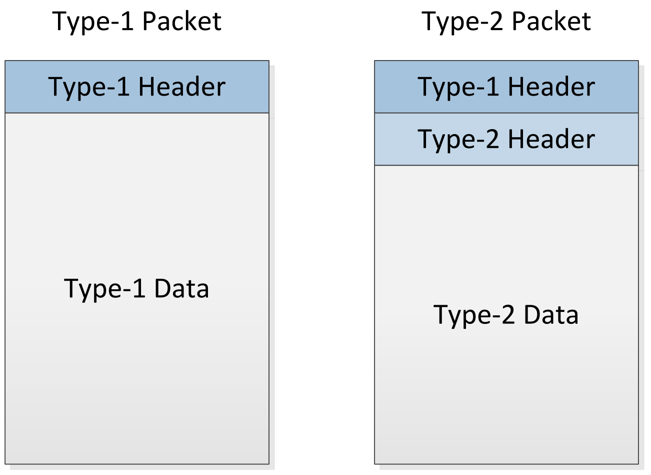


Fig. 16. Configuration Packet Types

In addition to the programming interface itself, it will be necessary to have a *bit file* and a PC compatible file downloader. After the design has been compiled, synthesised, placed and routed a bit file can be generated. The bit file contains all the data words, in sequence, required to program the FPGA through type-1 and type-2 packets. The largest packet in the bit file is a type-2 packet addressed to the FDRI containing all of the configuration frames.

ICAP_VIRTEX5 is a Virtex 5 primitive that makes it possible to access the configuration registers from inside the FPGA. The signals included in the ICAP interface are shown in Fig. 17. There are two ICAP ports available in the Virtex 5 FPGA. Such ports are available from the FPGA and can be instantiated to gain access to the configuration registers.

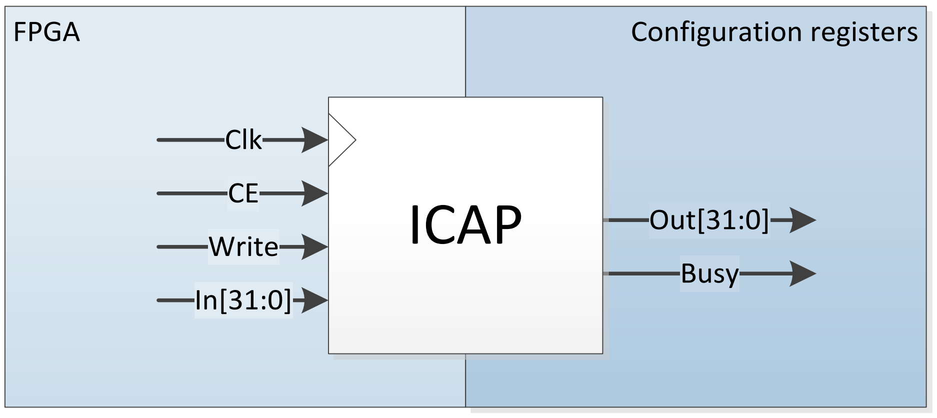


Fig. 17. ICAP Interface

The configuration memory in a Xilinx FPGA is divided into frames. Each frame corresponds to a portion of the programmable logic and routing, and is protected by a 12-bit error correcting code (ECC). A 32-bit Cyclic-Redundancy-Check value (CRC) is used to verify the integrity of the whole configuration memory. A single frame would typically correspond to a configurable logic slice, with surrounding routing resources. When programming, each frame can be individually addressed. In Xilinx Virtex 5, a configuration frame consists of 41 data words (a word is 32 bits).

Xilinx FPGAs can utilise Dynamic Partial Reprogramming, DPR, to reprogram a portion of the configuration memory during normal operation, without interrupting the operation of remaining parts of the system. DPR can be used to reprogram the device on frame level, using the frame-level addressing mentioned earlier. The ability to use DPR offers great flexibility and gives FPGAs a unique advantage over ASICs and traditional microprocessors. As the content of each frame can be read back and verified by Error Detection and Corection (EDAC) circuitry, it is possible to detect errors in the configuration memory by using the ECC fields. ECC is further elaborated in Sec. 2.4.4. It should be noted that only the subset of the configuration memory corresponding to actually utilised resources will be significant for the design, with the remainder essentially being treated as *don't care*. The device utilisation level is likely to be <100% for most applications. The used configuration bits and frames are referred to as sensitive bits and sensitive frames, respectively.

Xilinx Virtex 5 FPGAs contain dedicated DSP circuitry, in the form of DSP48E slices. Fig. 18 shows a simplified view of a DSP48E slice, featuring a 25x18 multiplier, internal pipelining registers and an arithmetic unit. DSP blocks are hard ASIC blocks embedded in the FPGAs array of programmable logic, and are much more area efficient compared to soft logic implementations of the same functionality [19]. As such, DSP blocks are not defined by an underlying configuration layer. The DSP48E is well suited for common DSP operations such as multiply-accumulate. An interesting feature of DSP48E is its run-time configurability, allowing the DSP slice functionality to be modified during operation, and even from cycle to cycle, through a set of control vectors (*OPMODE* and *ALUMODE*). These can not only be set at design time, but rather changed dynamically during run-time. The configuration vectors can be synthesised as constants or as signals originating from other parts of the system. DSP slices are arranged on the FPGA so that they can be cascaded through the use of fixed carry and shift lines to create wider operators than what would fit into a single DSP slice.

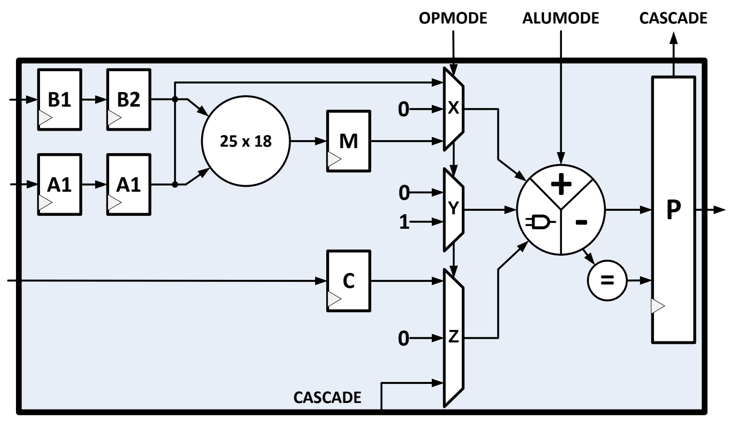


Fig. 18. DSP48E Slice

Block RAM, or BRAM, in Virtex 5 are made up of 36 kB SRAM memory blocks. These blocks can be cascaded and divided into a number of different configurations. For example, a single 36kB block can be used as a 36kx1 RAM, or as two functionally separate 18kx1 RAMs. It is also possible to create wider or larger RAM blocks by cascading BRAMs together. Fig. 19 shows a block diagram of a BRAM. An interesting feature is that the BRAM is dual-port, allowing access to both ports individually with each port having its own clock, address and enable signals.

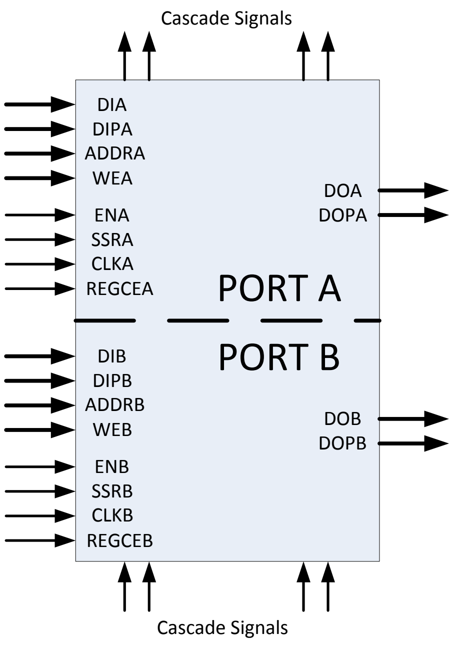


Fig. 19. 36kB BRAM

The FPGA resources of a design can be grouped into categories roughly according to the division discussed earlier. In this work, a distinction is made between Configuration, User Data and Architectural elements. The configuration group comprises all logic functions and routing controlled by bits in the configuration memory, such as LUT content, PIP connections and MUX control signals. This group determines the function of the FPGA as programmed by the designer.

User data is the dynamic memory content of storage elements, and is commonly read and written during normal operation. The content of these elements is user defined during operation, rather than programmed into the configuration memory. This includes DFF content, BRAM content and distributed RAM synthesised as LUTs.

Finally, architectural elements are the group of FPGA control elements mentioned earlier, including programming logic (JTAG, SelectMAP, ICAP), clock distribution and management, reset circuitry and PLLs. These functions are essential for the operation of the FPGA.

The majority of bits in an FPGA design mapping are configuration bits. The exact ratio of configuration to user data bits will depend on the application implemented in the FPGA. The amount of user data bits will depend on the utilisation of DFFs, BRAMs and distributed RAM in the application. A majority of the configuration bits will correspond to PIP and MUX control bits; in [20], [21]) this fraction is estimated to 80%, and in [22] to 90%.

Certain FPGA elements overlap between categories. For example, DSP48E slices in Virtex 5 would be considered here as partially belonging to configuration and partially to user data. This is because cascading and routing of DSP slices is defined by the configuration memory, while control signals and internal pipeline register contents are user memory.

Fig. 20 gives a graphical representation of the described FPGA element grouping. Specific SEU error modes for each of these categories will be discussed in Sec. 2.3.

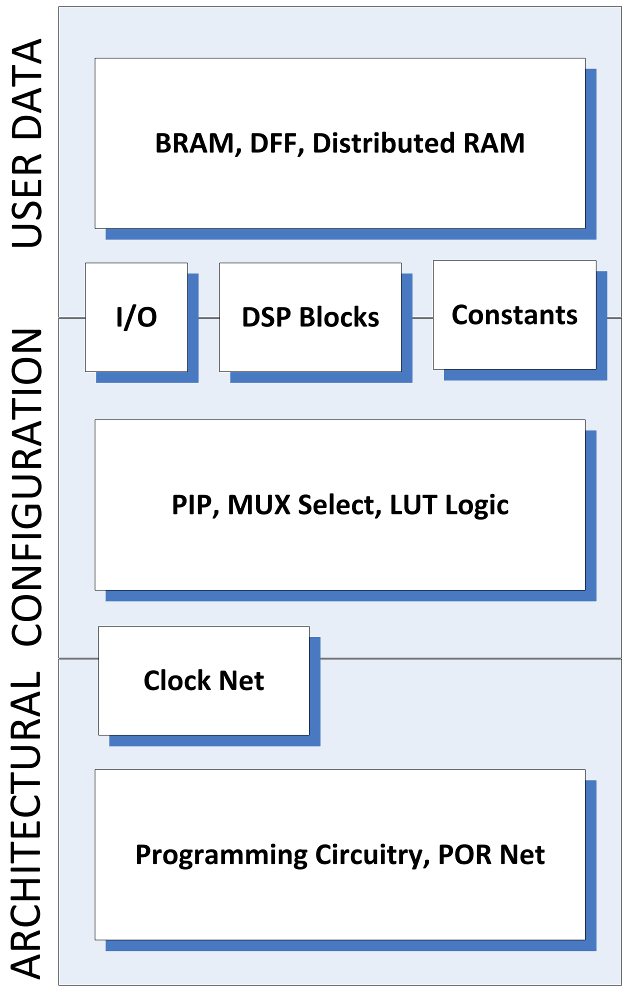

USER DATA
CONFIGURATION
ARCHITECTURAL
BRAM, DFF, Distributed RAM
I/O
DSP Blocks
Constants
PIP, MUX Select, LUT Logic
Clock Net
Programming Circuitry, POR Net

Fig. 20. FPGA Resource Groups

2.3 Single Event Upsets in SRAM-based FPGA

As with all SRAM-based electronics, SRAM-based FPGAs are susceptible to radiation-induced upsets [4]. The FPGA resources discussed in the previous section are all vulnerable to radiation-induced upsets. In this work, focus is put on soft errors, namely SEU and SET effects. For the purposes of this work, an SEU can be defined as a radiation-induced upset that causesx§ the state of a memory cell to change, from 1 to 0 or from 0 to 1. This is also informally known as a *bit-flip*. SETs are transient glitches on transmission lines or in combinatorial logic. Depending on the duration and amplitude of these glitches, they may lead to errors. SEUs are unpredictable and random by nature [2]. While one can estimate the approximate SEU rate, there is no way of predicting exactly when an SEU will occur. This section will discuss and categorise the different possible SEU error modes in Virtex 5 FPGAs, using the same notations and classifications as found in [6] and [4].

SEUs can result in a number of error modes in different parts of the FPGA. It should be noted, as discussed earlier, that not all SEUs will lead to errors, depending on the device utilisation level. Even in an application that uses 100% of the resources, not all configuration bits will be significant. In [23], the authors present a configuration memory sensitivity analysis for a typical FPGA application, comprising a soft-core processor, a bus structure and peripherals. The application uses 46% of the slices in the FPGA. It is found that for the example application discussed, only 14.16% of the configuration bits are sensitive bits with respect to SEUs, but that 84.93% of the configuration frames are used. This suggests that a majority of the configuration frames are under-utilised, which likely depends on the synthesis and *Place and Route* (PAR) tools optimising for performance. The authors also find that, for the particular application discussed, a majority of the sensitive configuration bits control interconnects and routing, as expected.

2.3.1 Configuration Memory Upsets

Configuration upsets occur when there is an SEU in an FPGA configuration memory bit, affecting the LUT content, I/O or routing. Upsets in the configuration memory are the dominant issue when discussing SEUs for FPGAs, with the majority of significant SEUs affecting the configuration memory [20]. This can be explained by the simple fact that there are a larger number of configuration bits compared to user data and architectural element bits.

Configuration upsets are static errors, as they will not disappear without repairing the configuration memory. Repairs are carried out by reconfiguring the frame containing the error. Three types of configuration memory upsets are discussed here: Routing, Logic and I/O errors. The sensitivity of the configuration memory is highly dependent on the application and the PAR policies applied. As the majority of configuration memory bits control routing, routing errors are likely to be the most common SEU effect [20]. The exact ratio will depend on the application implemented in the FPGA and the resources used.

2.3.1.1 Routing Upsets

SEUs can affect three categories of routing elements: PIPs, MUXes and Buffers [4].

2.3.1.1.1 PIP Errors

PIPs are simple on/off wire connections between two end-points. The on or off state of a PIP is controlled by a single configuration bit. An SEU affecting a PIP configuration can create an unwanted open or shortened circuit. An open circuit may disconnect two significant modules in the design, while a shortened circuit can create *bridging effects*, connecting together two modules that are designed to be logically separate. Fig. 21 illustrates an SEU that causes a shortened PIP connection.

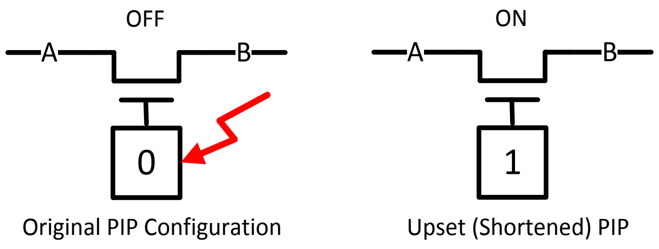


Fig. 21. Shortened PIP Error

2.3.1.1.2 MUX Select Errors

Multiplexers are widely used in FPGAs, for example to route signals within CLBs. Each MUX is controlled by select signals which are defined in the configuration memory. An SEU in a configuration memory cell defining a MUX control signal will cause a MUX routing error. Fig. 22 shows an example of an SEU in a MUX control signal.

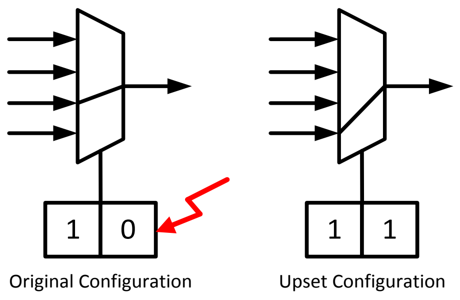


Fig. 22. MUX Select Error

2.3.1.1.3 Buffer Control Errors

Buffers can be seen as on/off switches to control if the input drives the output wire or not, and are often used for clock nets, I/O pads and bidirectional connections. An SEU in a buffer control configuration bit can lead to I/O direction errors or potentially driving two signals onto the same internal wire.

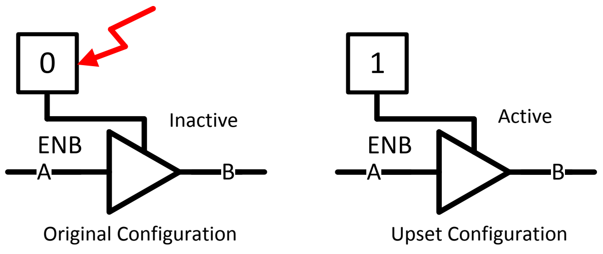


Fig. 23. Buffer Control Error

2.3.1.2 Logic Upsets

Logic upsets in the configuration memory are SEUs affecting the LUT content or control bits. This category also includes control bits for hard blocks such as DSP slices or BRAM.

2.3.1.2.1 LUT Content Errors

LUTs are used to implement common combinatorial logic. When implementing a logic function, a LUT functions as a mapping from the inputs to a single binary value stored in the configuration cell corresponding to the pointed-out value. This is the common use of LUTs. An upset in the configuration memory defining the LUT content will give an incorrect output when the inputs to the LUT are set to access the affected bit. This causes the logic function implemented in the LUT to be something different than the function specified by the configuration bit stream. Fig. 24 illustrates an SEU in a 4-input LUT implementing an XOR function. 4-input LUTs are used here for simplicity and illustrational purposes. After the upset, the particular input combination corresponding to the upset bit will no longer produce the correct output. However, the remaining unchanged bits will still produce the correct values. This can make LUT content errors difficult to detect based only on their output. In practice, this will manifest as an incorrect value produced by the combinatorial logic, which may later be clocked into a synchronous element as user data, be used as a control signal to another FPGA element or be used as an output signal from the FPGA.

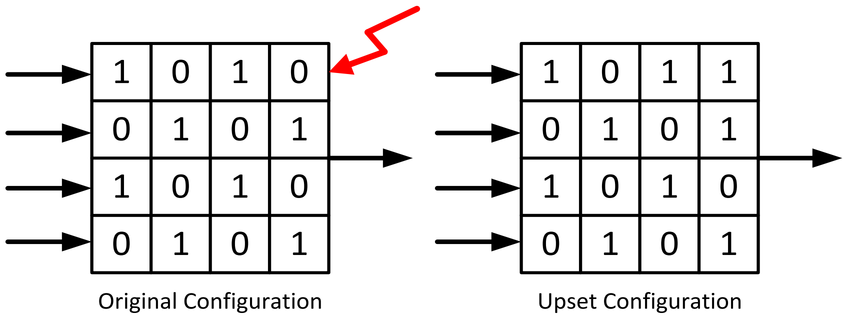


Fig. 24. LUT Content Error

2.3.1.2.2 Control Errors

Control bits are used to determine the usage of multi-functional blocks, such as LUTs and DSP slices, for cascading structures and internally in CLBs. For example, as mentioned earlier, LUTs can be used to implement combinatorial logic, shift registers or distributed RAM. The behaviour of a particular LUT is set by control bits in the configuration memory. For example, an SEU in a control bit can cause the LUT to be reconfigured from combinatorial logic to a distributed user RAM. It should be noted that not all LUTs can be configured as distributed RAM or shift registers [17]. In Xilinx Virtex 5, SLICEM logic slices can be configured in this way, while SLICEL lack this flexibility. BRAMs are also controlled by a number of control bits determining the behaviour of the BRAM, and can for example make the BRAM inaccessible, or reconfigure the BRAM to use a different access width.

In this work, we also consider upsets in fixed DSP block control signals and in carry chain structures to be control bit errors. The arithmetic carry chains are generally not accessible by the user from a high-level HDL description, but rather inferred by the synthesis tools. DSP control signals can be fixed by the configuration or, in the case of Xilinx DSP48, be set dynamically by the user.

2.3.1.3 I/O Upsets

As FPGA I/O blocks (IOB in Xilinx terminology) are configurable to work as inputs, outputs or bidirectional buffers, they make use of configuration bits that are sensitive to SEUs. A faulty I/O block configuration can lead to incorrect I/O behaviour as seen from outside the FPGA, or it can potentially have damaging effects on the system. In Xilinx IOBs, a single-bit error will not cause a damaging error.

2.3.2 User Data Upsets

User Data upsets are non-permanent (transient) errors in user data bits, such as SEUs in DFF, BRAM or distributed RAM content. As these errors occur in memory that is naturally read and written during normal operation, they may be overwritten before propagating further by introducing incorrect data into later stages. If the stored user data is used as control signals to other components in the FPGA, such as a DSP48E slice, it can also corrupt the soft configuration of those components. As user data is not defined by the configuration bit stream, upsets may be difficult to detect, as it may not be possible to distinguish between a computational error and a user data SEU.

2.3.2.1 DFF Upsets

The basic error mode for DFFs is a change in the state of the bit held by the DFF, later potentially propagating through the data path. As DFFs are used in basic design elements, this can also cause incorrect behaviour in for example clock transitions, shift registers or FIFO structures. SETs can cause user data errors if they occur on a transmission line or in combinatorial logic and later are clocked into a DFF. Fig. 25 shows an example of an SET affecting a transmission line in combinatorial logic, resulting in a glitch which is later clocked into a DFF. In this example, the output of the DFF is incorrectly changed to a logic 1.

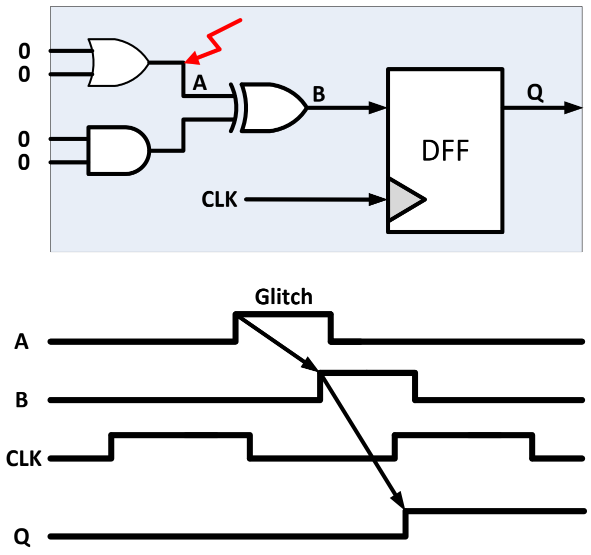


Fig. 25. Single Event Transient

There is a possibility of an SET input causing a glitch that violates the setup and hold times of the DFF. In a worst-case scenario, this results in the DFF being put in a meta-stable state. Not all SETs are propagated and clocked into synchronous elements, as logical, electrical and clock window masking effects apply. That is, a glitch on a transmission line may be logically masked by the combinatorial logic, it may be too low in amplitude or width to be significant, or it may not overlap with a positive clock edge on the synchronous element.

2.3.2.2 RAM Upsets

RAM upsets are SEUs in BRAM or distributed RAM, resulting in incorrect data being stored. There is also a possibility of SETs on enable or select lines which may result in corrupted data being clocked into user memory. No separation is made here between SEUs directly affecting a user data bit and SETs resulting in incorrect user data. User memory in RAM blocks can be protected by the use of ECC, as will be discussed in Sec. 2.4.4.3.

2.3.3 Architectural Upsets

The case of an SEU affecting configuration bits corresponding to system-critical FPGA control elements is called SEFI, as mentioned in Sec. 2.1.4.1. SEFIs are SEUs that affect the basic functionality of the FPGA, and include SEUs in programming logic (JTAG, SelectMAP and ICAP controllers), reset nets and clock resources, as well as their associated control registers. An SEFI can render the device unusable. SEFIs usually require a full reset and reconfiguration of the FPGA [4]. Since these system-critical structures are small in area and use few configuration bits compared to the rest of the sensitive bits in the FPGA, SEFIs are far less common than configuration and user data upsets.

If the programming functionality is affected by an SEFI, it can produce incorrect reads from or writes to the configuration memory, or prevent programming all together. SEFIs can occur in the control state machines or in routing of a programming interface. Since the programming interfaces have functionality to disable I/O blocks while reprogramming, this may affect outputs from the FPGA. SelectMap, JTAG and ICAP are all susceptible to SEUs. An SEU in a *Delay Locked Loop* (DLL) can make the clock signal unsynchronised with respect to output signals from the FPGA or lead to internal clock skew. The clock net in general is susceptible to SEUs. For example, an SET on the clock net or clock buffers may cause unwanted or incorrect clocking of the circuit. *Keeper circuits* are used to generate constant 1's or 0's used in the design, and are included as fixed circuitry in the FPGA architecture, utilising unused inputs to a logic block. An SEU can flip the value held by a keeper. An SEU (or SET) in the *reset net* of an FPGA can lead to a deconfiguration of the device, or an inadvertent or unexpected reset. In a worst-case scenario, it may affect the whole reset functionality of the FPGA.

The control elements of a Xilinx Virtex 5 FPGA use a number of control and status registers. All commands executed when programming, reading and checking the status of a device are made through these registers. These include among others general control registers, CRC registers for readback, Frame Address Register and Watchdog. An SEU in these registers may affect the ability to perform readback from the device, cause configuration bits to be written to an incorrect frame address, or cause a reset of the device. A full list of the available configuration registers and their function is provided in [18]. As these registers are architectural features whose implementation is not under the direct control of the user, errors may be difficult to detect other than via the observation of incorrect behaviour from outside the device.

2.4 SEU Mitigation Techniques

The concept of *fault tolerance* is a wide topic with many different combinations of implementations. In order to understand the concepts involved in fault tolerance, it is necessary to understand the chain of events that lead to a failure, as shown in Fig. 26.

2.4.1 Fault tolerance definitions and concepts

A *Fault* is the cause of an error. It could be an event, a bug, a faulty circuit connection etc. An *Error* is what is directly affected by the fault such as an erroneous output, a non-functioning module or a faulty state. A *Failure* is when a service is not delivered or when it does not comply with the specification. A program which fails to deliver a value or a circuit which fails to write an output are both examples of failures. There are a number of different failure modes which are commonly used:

- *Value failure* – An incorrect value is delivered by the service
- *Signalled failure* – A failure signal is provided by the service
- *Timing failure* – A result is delivered too early or too late by the service
- *Silent failure* – No result is delivered by the service

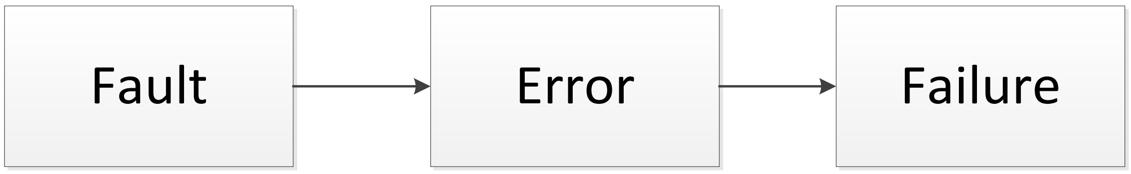


Fig. 26. Failure Chain

Two different concepts emerge when talking about fault tolerance. *Fault tolerance* is to avoid failures through the tolerance of errors. This means that the fault tolerance needs not to avoid faults itself but to stop errors from causing failures. *Error masking* is a common implementation used to provide fault tolerance. *Fault prevention*, however, is to prevent or limit the occurrence of faults. There is therefore no inherent tolerance when talking about fault prevention.

There are a number of different ways to implement fault tolerance. The most common method is to use some type of redundancy which can be further divided into three categories:

- Voting redundancy
- Standby redundancy
- Active redundancy

Voting redundancy is when errors are masked through majority voting. The most common type of voting redundancy is *Triple Modular Redundancy* (TMR). TMR for SRAM-based FPGAs will be discussed in detail in later sections. Majority voting requires $2f + 1$ units to tolerate f simulataneous faults. It is possible to run the system without an interrupt in the presence of a fault. Voting redundancy also has no requirements regarding failure mode, due to the fact that a differing result will be masked out regardless of failure cause. Voting redundancy leads to a relatively large added area overhead. Since the overhead area from added error mitigation circuitry in SRAM-based FPGAs itself is sensitive and has a failure rate, the overall gained reliability is reduced. This is an important aspect to consider when selecting a fault-tolerance scheme. In fact, some mitigation approaches for FPGA actually increase the overall SEU sensitivity, as discussed in [24].

Standby redundancy is the concept of having one primary unit and several spare units in standby. This type of redundancy works under the condition that the active unit becomes subject to a detectable value failure, a signalled failure or a silent failure. There is a delay during reconfiguration since all of the tasks need to be relayed to a spare unit. Standby redundancy requires $f + 1$ units to tolerate f simultaneous faults.

Active redundancy is the configuration in which two or more active units work in parallel to produce replicated results. This configuration will also only work for a detectable value failure, a signalled failure or a silent failure. Similarly to standby redundancy, active redundancy setups require $f + 1$ units to tolerate f simultaneous faults.

In the case of a fault induced error, the error may either propagate to adjacent parts or stay contained. Parts of a system which are capable of causing a failure for the whole system are called *single points of failure*. A typically desirable property of fault tolerant systems is a minimal number of such points of failure. A part of the system which is not a single point of failure can therefore be faulty without causing the system to fail. This means that the fault is limited to the node or component. The part of a system to which the fault is confined is called a *fault containment region*. It is of importance to have small fault containment regions in order to tolerate faults efficiently.

When looking at a system as a whole it is often useful to be able to model and simulate the reliability and behaviour. This is important when planning and designing a system to be able to implement the most efficient fault tolerant method. With known component failure distributions, it is possible to calculate metrics such as *mean time to failure (MTTF)* and the probability of the system working at a given time.

The lifetime of a component can often be modelled by the use of known distribution functions. More specifically, a *probability density function* describes the relative probability of an event. The probability density function is often denoted $f(x)$. A *distribution function*, also referred to as accumulative distribution function, is defined according to (11). The density function describes the probability that a stochastic variable X will have a value equal or smaller than a variable x, as shown in (11).

$$F(x) = \int_{-\infty}^{x} f(x)dx = P(X \leq x) \tag{11}$$

An application where the mentioned functions are used is when modelling lifetimes of components. Assume the probability density function describes the probability of a failure as a function of time with a known distribution. The distribution function can then be used to calculate the probability of a failure having occurred at a given time.

The *reliability function*, or survival function, is defined according to (12). The reliability function is the probability that a stochastic variable X has a value greater than the variable x. As shown by (12), the reliability function can simply be resolved by calculating the remaining probability of a distribution function for the same variable. Using the earlier component lifetime application, the reliability function would return the probability of a failure not having occurred at a given time.

$$R(x) = 1 - F(x) = P(X > x) \tag{12}$$

In the case where the failure rate is described by a distribution, a probability density function can be used to resolve information as well. In this case, $MTTF$ is nothing but the expected value $E[X]$ of the same probability density function, which is defined according to (13). $MTTF$ is frequently used as a metric when describing fault-tolerant systems. *Mean time to repair (MTTR)* describes the rate at which a unit is repaired, replaced or reset. It can also be used when calculating the availability of a system.

$$MTTF = E[X] = \int_{-\infty}^{\infty} xf(x)dx \tag{13}$$

The *failure rate*, denoted $\lambda(x)$, is the frequency of failures per unit time. It is defined according to (14) as the ratio between the probability density function and reliability function. It is commonly expressed as failures per hours, which is another frequently occurring metric. *Mean time between failures (MTBF)* can be resolved as the inverted failure rate according to (14), but also as the sum of MTTF and MTTR as shown in (15).

$$\lambda(x) = \frac{f(x)}{R(x)} = \frac{1}{MTBF} \tag{14}$$

$$MTBF = MTTF + MTTR \tag{15}$$

The ratio between the downtime and running time of the system is the definition of *availability*. It can also be calculated using MTTF and MTTR as shown in (16). A common availability standard for critical systems is the “five-nines” standard , indicating an availability of 99.999% [25].

$$Availability = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF} \tag{16}$$

A common metric used when measuring fault tolerance is *Failures in Time (FIT).* FIT is the number of failures during one billion hours for one device and is calculated as shown in (18).

$$FIT = \frac{10^9}{MTBF} \tag{17}$$

2.4.2 Triple modular redundancy

TMR is the most common implementation of voting redundancy. Due to its masking properties it has become a popular method used to provide fault tolerance.

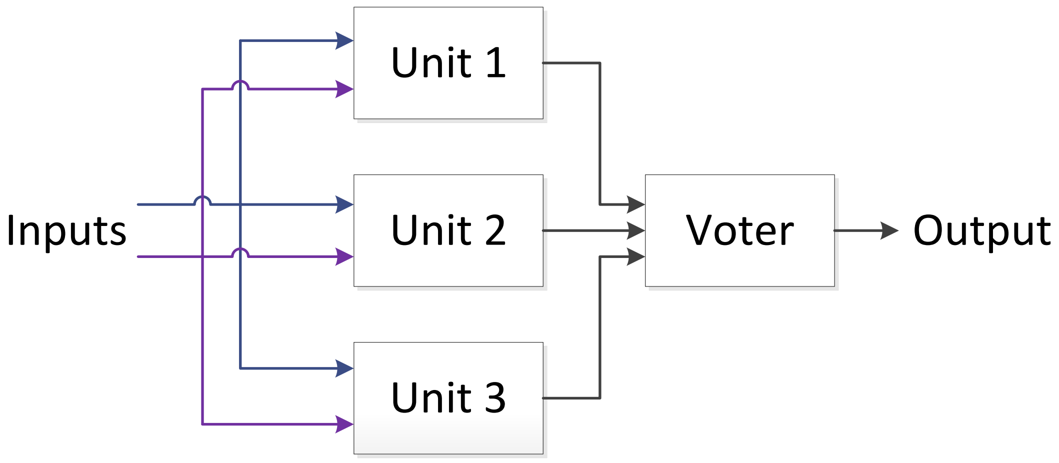


Fig. 27. Three parallel units in a TMR configuration.

The basic concept of TMR is to run three identical, redundant computation paths in parallel, running the same processes with the same inputs, as shown in Fig. 27. All units compute the same outputs, running the same processes. The outputs from each stage are compared with the parallel stages by the use of a majority voter. If the outputs of any unit deviate from the remaining two, the voter still produces the right output. There are a number of different approaches and versions of TMR implementations. Many attributes of the implementation can be varied in order to tailor the TMR implementation to available resources and the target application [25].

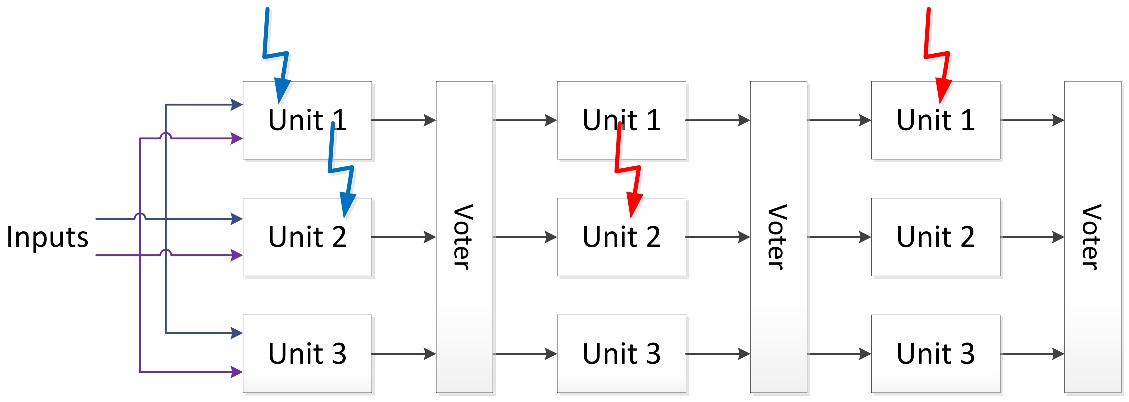


Fig. 28. Two different types of faults occurring in a TMR system.

A fundamental limitation of a TMR system is that one fault at the most can be tolerated per voter stage. Two types of multiple SEUs are shown in Fig. 28. The red disturbances indicate multiple errors taking place at two different stages. These errors will be masked out due to majority voting. The blue disturbances, however, indicate multiple errors taking place at the same stage. TMR will not be able to mask the fault since the majority is incorrect. This property makes voter stage partitioning and level of implementation an important part of the TMR design process. TMR is often used in conjunction with techniques that prevent errors from accumulating within one voter stage over time, such as *scrubbing*, as will be discussed in greater detail in Sec. 2.4.3.

SEUs can also occur in the configuration bits that define the interconnect matrix. In this case it is possible for a node of one redundant unit to become connected to another redundant unit in the same voter stage. This particular case is known as a *domain crossing event* and needs to be taken into consideration when modelling the overall availability of a system [26].

2.4.2.1 Level of implementation

TMR can be implemented on different system levels. A design can be triplicated within an FPGA, which is the common case. The design can also be triplicated on a higher level, for example by using redundant FPGAs on a device-level.

The most frequently occurring implementation is using a triplicated design within an FPGA[27], [28], [29], [30], [20]. The overall strategy in this case is to triplicate parts of the design or the whole design within a single FPGA. This is only possible if the single design fits within less than one third of the FPGA resources. A TMR system implemented within an FPGA will have several voter stages in the design. Such a system will therefore be able to tolerate at least one SEU in the triplicated parts of the design. In addition, multiple SEUs can be tolerated if they occur at different voter stages within a design. The way in which voter stage partitioning affects the fault tolerance will be discussed in Sec. 2.4.2.2.

Where the synthesised design equals or exceeds one third of the available FPGA resources, partial TMR may be a suitable implementation. Partial TMR implies that parts of the design will be implemented using TMR. A method of assigning TMR to selected parts of a design is presented in [27] where priority is given to sequential logic. The motivation for giving priority to sequential logic is due to the fact that sequential logic is harder to reconfigure and reset. Registers keeping track of internal states are implemented in sequential logic. Reconfiguring register content in the event of an SEU is difficult without resetting the system. Avoiding reconfiguration of sequential logic is therefore important. Second priority is given to nodes that output signals to a larger number of nodes. Signals which are connected to many nodes will have a greater significance and are therefore worth protecting.

Another presented suggestion is that TMR is implemented on FPGA level [31]. Such a configuration would consist of three FPGAs with an additional radiation hardened ASIC managing voting and configuration. A disadvantage with this implementation is that additional FPGAs result in additional overhead due to the peripherals associated with each additional FPGA. Furthermore, such a solution would increase the power dissipation and the required circuit-board area. An advantage with this solution is that the external scrubber and voter unit will be implemented on an independent radiation-hardened ASIC. The configuration and voter unit will also not require any allocation in any FPGA, which is sensitive to SEUs. When implementing scrubbing and voting on the FPGA, the fact that the scrubber and voters themselves can fail needs to be taken into account.

2.4.2.2 Voter partitioning

Voter partitioning is an important consideration in a TMR system. Depending on the density of voter units, different properties can be achieved. Increased partitioning is not necessarily beneficial. Increased power dissipation and area overhead will follow as more voter stages are included. Moreover it is important to take the increased logic into account, which itself will also have a failure rate.

In [20], a study is conducted for an FIR-filter implementation where partitioning is done to different degrees The different partitioning steps are evaluated by injecting faults in the configuration bits for all the designs respectively. The percentages of failures are then recorded together with the area overhead. The most interesting conclusion drawn by the authors in [20] is that the densest partitioning scheme does in fact not yield the best resilience to the faults inserted. Instead, a medium-density voter partitioning gives the best results. This suggests that the increase in configuration bits for the densest partitioning may result in a decrease in fault tolerance if the additional voters are not used efficiently enough. This is likely due to the fact that increasing the partitioning will also increase the number of SEU-sensitive configuration bits. The increase in number of slices varied from 217% for the least dense partitioning to 273% for the densest partitioning.

In [30], another similar study is conducted for a multiplier. A reference design with voters at the end is compared with the same design with additional voter stages. The number of slices occupied and the possible faults combinations covered are compared for all increments. An interesting observation is that the medium partition is the most efficient when calculating increase in area overhead per increase in fault coverage. Another interesting observation is that the increase in number of slices was more than 200% for all TMR implementations in this study. Furthermore, it is calculated in [30] that the densest voter partitioning would lead to a slice increase of 314%.

2.4.2.3 Voter structure

Voters can be implemented in a number of different ways. The simplest implementation is showed in Fig. 29. This type of voter could be implemented by simply using a majority voter for each bit, as shown in Fig. 30. This type of voter, however, is a single point of failure. If this particular voter fails, it will render all redundant stages useless.

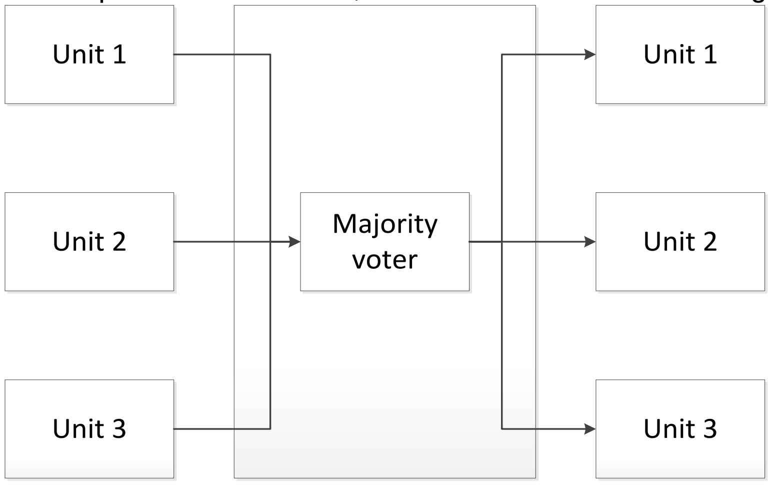


Fig. 29. Simple voter implementation comprising one majority voter.

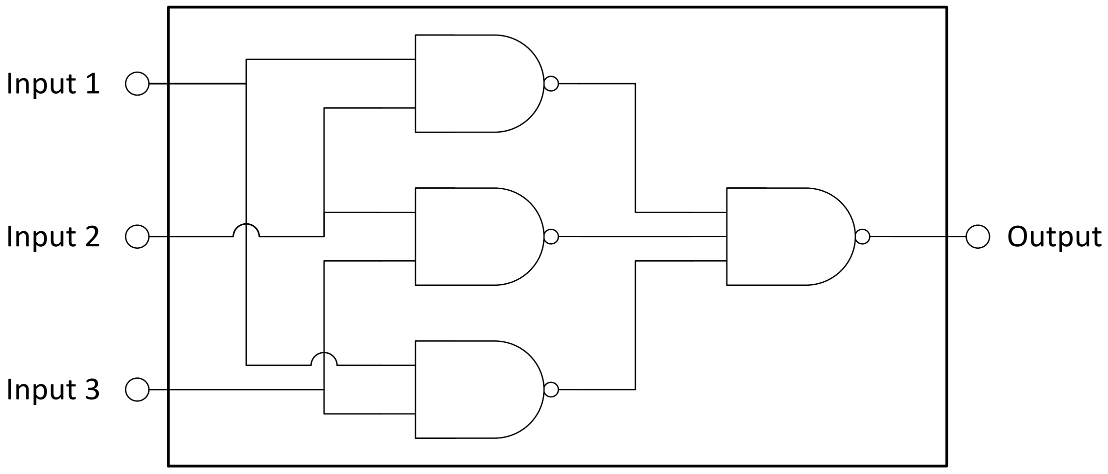


Fig. 30. Possible implementation of a three-input majority voter.

An improvement from the simpler voter is the setup shown in Fig. 31, using three parallel voter stages. Similarly, a bit-wise voter could be used to implement such a function. The triplicated voter stage requires more resources, but will not necessarily increase the critical timing path. This means that both voter types would have the same timing since the voters are parallel to each other.

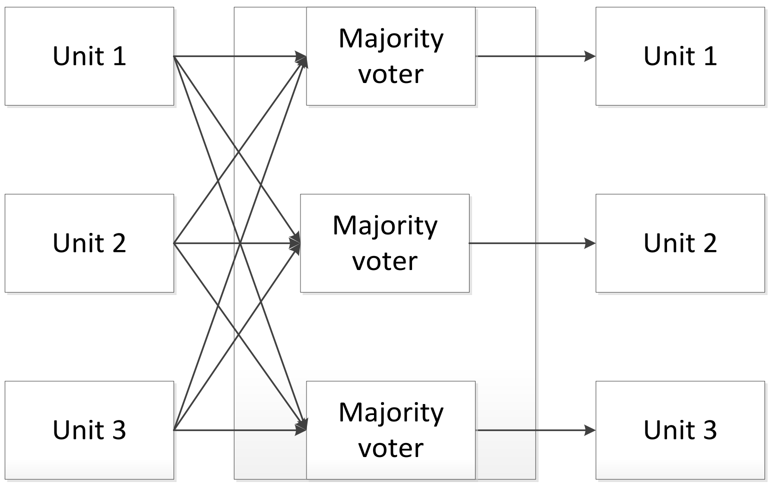


Fig. 31. Triplicated voter implementation comprising three majority voters.

2.4.2.4 Method of implementation

There are a number of different methods the designer can use in order to implement a TMR system. The designer can rely on synthesis tools to triplicate the design and to insert voters. Another option would be for the user to manually incorporate the redundancy in the HDL.

By relying on a synthesis tool, the designer will have to spend less time implementing the TMR. The designer might even be able to omit redundancy considerations completely when designing, instead treating the design as a normal design without the fault tolerance. A commonly used tool for TMR implementation is XTMR from Xilinx [32]. The XTMR tool applies TMR to the netlist output by the synthesis tool. Running the tool will triplicate all inputs including clock nets and combinatorial logic. Furthermore it will triplicate sequential logic and insert majority voters on feedback signals. Lastly it will triplicate output signals with minority voters to trigger and disable faulty outputs. This method of implementation requires minimal effort to make sure TMR is implemented. On the other hand, changing attributes of the TMR implementation becomes difficult; partial TMR, for instance, is not supported.

If full customisation is a requirement, it is also possible to implement TMR manually in the HDL description of a design. [28] presents a set of guidelines to be used in high-level HDL implementation of TMR. The approach is called *functional TMR (FTMR)*. Introducing redundancy in HDL can be troublesome since synthesis tools may optimise the design for performance or area, taking away the redundancy in the process. The FTMR method gives full customisation possibilities to the designer and the possibility to tailor the implementation by, for instance, limiting it to certain parts. FTMR will therefore be a viable method of implementation when partial TMR, for instance, is considered.

2.4.3 Configuration Memory Scrubbing

The term *scrubbing* refers to a category of error mitigation techniques that prevent error build-up by refreshing and restoring configuration memory cells to a known-good state. SRAM-based FPGAs are volatile, and therefore may require off-chip memory to store their configuration. By making use of a radiation-hardened off-chip memory (usually EEPROM or Flash) for this purpose, it can be used as a *golden reference*. Since SEUs in FPGA configuration memory bits are *persistent* errors, they will not disappear until repaired by reprogramming and overwriting the faulty bit with the correct configuration. Over time, configuration memory SEUs in a system without this repair capability will cause error build-up. Error build-up will eventually break other SEU mitigation techniques, such as TMR, by introducing multiple errors inside a single fault containment region [33]. Knowing the golden-reference bit stream, the configuration memory can be monitored and repaired by a configuration manager, or *scrubber*. Scrubbing is an important technique, as it repairs errors to prevent accumulation, rather than preventing, masking and tolerating errors like the other techniques discussed here. This makes the use of scrubbing practically mandatory in fault-tolerant FPGA systems. Error detection and error correction are two concepts important to scrubbing that will be discussed in later sections.

Scrubbers, or configuration managers, are in essence the same type of circuit that is responsible for the initial configuration of the FPGA, making use of the programming interfaces of the FPGA (JTAG, SelectMAP or ICAP). Through these interfaces, the configuration memory can be written or read back with a frame-level resolution. Scrubbing can be done by full or partial reconfiguration of the device. Using the dynamic partial reconfiguration features of Xilinx FPGAs, configuration memory repairs can be made without interrupting the operation of the whole device.

Scrubbing is a widely used technique, and considered in the literature as a vital part of implementing fault-tolerance for FPGAs. It is often used in conjunction with other mitigation techniques, as it only repairs existing errors rather than masking them when they arise. Scrubbing in conjunction with TMR is the most commonly applied combination for SEU mitigation, and gives an overall very effective solution [34]. There are a variety of techniques and implementation schemes for scrubbers, as will be discussed in detail in this section.

2.4.3.1 Scrubbing Techniques

Scrubbing can be done with device- or frame-level reconfiguration, corresponding to a full reconfiguration and dynamic partial reconfiguration, respectively. Device-level reconfiguration will invariably lead to some down-time, and may not be feasible for some applications. In Xilinx Virtex 5, scrubbing based on dynamic partial reconfiguration can be configured not to overwrite user data stored in shift registers and distributed RAM implemented as LUTs. Error detection is optional and varies in level between scrubbing schemes, as does the scrubbing rate. The scrubbing rate can be fixed or variable, depending on the scrubber implementation and complexity.

The most basic and least complex form of scrubbing is a simple periodic reconfiguration of the FPGA without error detection. This is known as *blind scrubbing*, *preventive scrubbing*, or in Xilinx terminology *Scheduled Maintenance* [35]. With blind scrubbing and frame-level dynamic reconfiguration, the FPGA is scrubbed from start to end, frame by frame. This is done periodically and without error detection. That is, frames are scrubbed regardless of whether an error has occurred in the frame or not. The scrubber will be in write mode during the time it is actively rewriting the configuration, and in idle (read) mode otherwise. The ratio between the scrubber time spent in read and write mode depends on the scrubbing rate. Blind scrubbing has the advantage of avoiding the extra complexity introduced by error detection, but can also be seen as an inefficient use of scrubber time as it will scrub uncorrupted frames as well as corrupted ones. Furthermore, the unnecessary time spent in write mode is a vulnerability as errors in the bit stream, frame addressing register or programming interface may cause a corrupted value to be written to a frame.

Readback scrubbing is an alternative to blind scrubbing. As the name suggests, readback scrubbers operate by reading back the configuration from the FPGA in order to determine if an error exists or not. In Xilinx terminology, this technique is known as *Running Repairs* [35]. The error detection is facilitated by the use of configuration frame-level ECC and CRC values calculated for the whole configuration memory. The scrubber can continuously read the configuration memory CRC value and compare it to that of the golden reference. If an error exists, frame ECC can be used to localise the corrupted frame. The corrupted frame is read back in whole to the scrubber and corrected by overwriting with the correct configuration from the golden reference. This *error detection and correction* (EDAC) mechanism works on single upsets within a frame, relying on the improbability that a multiple bit upset will occur in a single frame within the time it takes to correct an error. Readback can also be based on simple comparison with the reference memory, using a bit mask.

It is possible for a single SEU to affect two bits located in adjacent configuration memory frames. In this case, it will be seen by the readback scrubber as two distinct single-bit errors and corrected accordingly. Readback scrubbing has the advantage of being in read-mode most of the time, only ever going into write mode when an error is detected. It is also a more time-efficient use of active scrubbing by avoiding the reconfiguration of uncorrupted frames, but introduces complexity in terms of error detection and localisation logic. Fig. 32 provides an illustration of the operation and *Time To Repair* (TTR) of blind versus readback scrubbing. In this example, the blind scrubber does not make use of error detection, instead simply reconfiguring the FPGA periodically and frame by frame from start to end.

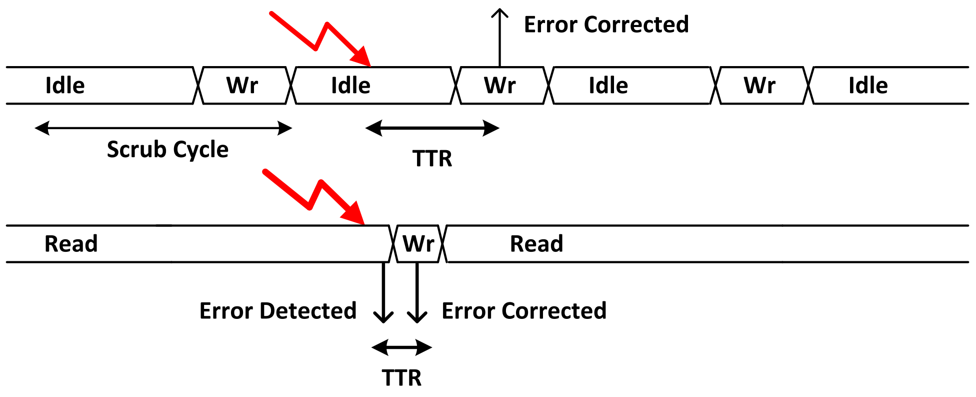


Fig. 32. Blind vs. Readback Scrubbing

A system may also benefit from employing a combination or compromise between these two distinct approaches, as discussed in [36]. For example, blind scrubbing may be supplemented by coarse-grain error detection based on the CRC value for the whole configuration memory. Upon detecting an error somewhere in the system, the next scheduled scrubbing round can be moved up to be performed as soon as possible. This reduces the time to repair the specific error. However, a blind scrubber still has no information about which frame to repair, meaning that a normal start to end scrubbing of the whole configuration memory has to be started. Readback scrubbing can also benefit from periodical scrubbing of the whole device, even if no error has been detected. This is to protect against the event of a multiple bit upset inside a single frame, which may otherwise have gone undetected by the EDAC mechanism.

More advanced scrubbing schemes can be implemented that take into account the sensitivity and criticality of individual modules or parts of the configuration memory, such as those discussed in [37]. The system might benefit from prioritising the scrubbing of certain critical modules. As described earlier, the basic blind scrubber will scrub the configuration memory from start to end. This is not necessarily the best approach. By modifying the frame sequence to be scrubbed, the critical modules can be set to be scrubbed more often. In [36] this approach is presented as *2D Scrubbing* or *Selective Scrubbing*. This method requires extra effort by the designer in implementing the scrubber, both in recognising the sensitive modules and in setting up the frame sequence as used by the scrubber.

If periodical scrubbing is employed, a scrubbing rate needs to be set according to the estimated SEU rate such that errors will be corrected before building up or causing multiple upsets within a module which may break fault-masking strategies. Commonly, a scrubbing rate of 10x the estimated SEU rate is used, as noted in [36]. Depending on how advanced the scrubber implementation is, the scrubber can use a fixed or variable scrubbing rate. A variable scrubbing rate can be useful when a mission experiences peaks in SEU rate when passing through specific sections of its orbit, for example the South Atlantic Anomaly in LEO. Reducing the scrubbing rate during low-SEU-rate periods can help save power.

2.4.3.2 Scrubbing Implementation

As previously mentioned, there are a variety of approaches to implementing the scrubbing techniques discussed in the previous section. Typically, a scrubber is placed in a rad-hard auxiliary device external to the FPGA, and interfaced with the FPGA through JTAG or SelectMAP. The auxiliary device, or *external scrubber*, is then responsible for error detection and correction, as well as handling the non-volatile reference memory. The external scrubber can be placed in an auxiliary rad-hard FPGA or microprocessor, or an ASIC. Using an external scrubber setup has the advantage of separating scrubber and target FPGA, allowing the scrubber to be implemented in a device that is radiation hardened by process [38]. The purpose of this is to make the scrubber itself immune, or at least much less susceptible, to SEUs. However, the programming interfaces on the target FPGA, for example SelectMAP, are still susceptible to SEUs. By using a dedicated scrubber and by storing pre-calculated CRC and ECC values in its internal memory, scrubbing performance can be increased.

Scrubbers can also be *internal* to the FPGA, meaning that the scrubber is implemented in logic and placed in the target FPGA itself, and is therefore *self-hosting*. While reducing overall system complexity, this has the obvious disadvantage of being susceptible to SEUs in the scrubber logic. Also, there is still a need for an off-chip, non-volatile memory and memory controller for the initial programming of the FPGA. Some internal scrubber implementations feature the possibility of *self-scrubbing*, where the scrubber can scrub portions of itself using dynamic partial reprogramming. A comparison between an internal and an external scrubber implementation is made in [39]. The authors come to the conclusion that the internal scrubber implementation examined is less reliable and efficient compared to the external implementation. Fig. 33 shows simplified block diagrams of example internal and external scrubber setups.

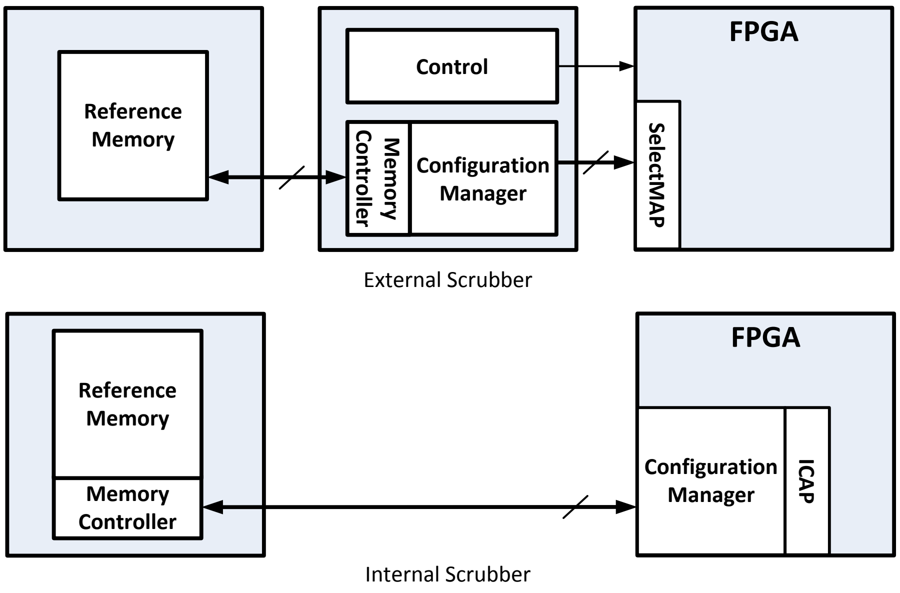


Fig. 33. External and Internal Scrubbers

The configuration manager (the scrubber itself) can be primarily *software-based* or completely *hardware-based*. A pure hardware approach, either internally in the FPGA or implemented on an external host module, can be efficient both in terms of performance and interfaces to the reference memory and target FPGA. This is the common approach used in the literature [40]. With a hardware-based scrubber, the scrubbing algorithm is fixed, typically based on a state machine.

Using a software-based scrubber running on a rad-hard microprocessor is an alternative that allows for greater flexibility in the scrubbing algorithm which can be based on more sophisticated schemes. However, a software-based scrubber this is typically significantly slower than a pure hardware approach, as it requires the execution of the scrubbing algorithm program in the microprocessor, as well as interfacing the microprocessor with the reference memory and the target FPGA. Employing a compromise between the two alternatives can be an efficient approach, for example by implementing the scrubber in an auxiliary FPGA with a soft-core processor. This allows fast interfacing and data processing by the FPGA logic (e.g. CRC calculations), while the soft-core handles the high-level algorithm.

In systems where one (external) scrubber is responsible for scrubbing multiple devices, the hardware-software-hybrid approach may be suitable due to the number of interface pins required, and the usually more sophisticated scrubbing schemes. The scrubber can also be used in device-redundant systems to detect errors manifested as discrepancies between the outputs of the redundant devices.

Since scrubbers require continuous use of the programming interfaces of the FPGA, special techniques need to be implemented in order to update the reference configuration during operation and make use of dynamic partial reconfiguration. In [41], the authors suggest a scheme for circumventing this problem, making use of a partially reprogrammable memory to store the reference configuration.

Xilinx provides a macro for soft error detection and correction. The SEU Controller, or *SEU_cntrl,* is self-hosted by the target FPGA. SEU_cntrl can access the Internal Configuration Access Port (ICAP). It uses the frame ECC and the Configuration Memory CRC to detect errors and automatically correct them in a *Single Error Correction, Double Error Detection* (SECDED) scheme, which can be seen as a form of scrubbing. It can also access the SelectMAP port [42]. Fig. 34 shows a block diagram of the SEU_cntrl macro.

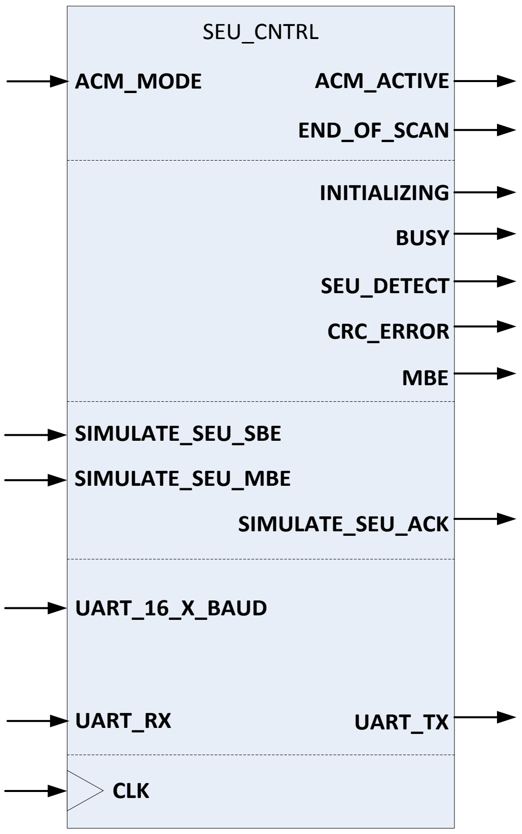


Fig. 34. Xilinx SEU Controller Macro

A useful feature of the SEU_cntrl macro is the ability to simulate SEUs for the purpose of testing mitigation schemes. The macro can inject errors (bit-flips) in the configuration memory, either randomly or in locations defined by the user. This allows for a predictable and controlled testing method when evaluating SEU mitigation schemes, and will be used later on in this work. Featuring a standard UART connection, the macro can interface with a PC for error injection and logging. Internally, the macro uses a Pico-blaze core, an 18kB BRAM block and an ICAP controller [42].

The SEU controller macro is itself susceptible to SEUs. Specifically, there are two types of errors that can cause the SEU_cntrl to fail: Multiple Bit Upsets (MBUs) in the same frame, and SEUs in the configuration memory or user memory of the SEU Controller itself. Some of these (less critical ones) may be corrected, as the SEU_cntrl has the ability to scrub itself. Errors in the programming interface (ICAP) or frame ECC errors may lead to the SEU Controller writing incorrect frames to the FPGA configuration memory. In Virtex 6 and newer Virtex devices, the SEU Controller is known as the SEM Core (Soft Error Mitigation Core). The SEM is an IP Core that can be generated through Xilinx CoreGen.

2.4.3.3 System Context and Limitations

In selecting a scrubber implementation scheme, the scrubber has to be put into context and in relation to the system as a whole. This includes an assessment of the SEU rate and potential risks, as well as considering the fault-tolerant techniques that are used in conjunction with the scrubber, in order to achieve the desired reliability levels. Overall power consumption, system complexity and by extension cost, are other factors in selecting the scrubber implementation scheme. The added system complexity by an external scrubber may be infeasible for some applications. Watchdog and configuration register scrubbing can be implemented by the configuration manager to protect against cases in which the target device experiences an SEFI that disrupts critical control elements. This may require a full reconfiguration of the FPGA, and assumes that the scrubber is not itself affected by the SEFI. Control registers in the FPGA can be polled periodically by an external circuit or external scrubber. As a minimum, a system for space applications employing scrubbing needs to implement an off-chip memory, a configuration manager, a scrubber (here the two are separated to highlight the difference between the initial programming circuitry and the continuous scrubbing) and a watchdog timer [38]. Table 3 gives an overview of the scrubbing classifications and implementation options discussed in this section.

Table 3. Scrubber Variations

Implementation Aspect	**Alternatives**
Scrubbing Approach	*Readback* *Blind* Blind with basic *Error Detection*
Implementation	*Hardware* (FSM) *Software* (Soft-core) *Hybrid*
Placement	*External* *Internal*
Multiple Devices	*One-device* scrubber *N-device* scrubber
Scrubbing Frequency	*Periodical* *Variable* *Watchdog Timeout* On Error Detection
Error Detection	*Frame ECC* *Memory CRC* *Direct Compare*
Error Correction	*Full* *Partial* Frame-level *SECDED*

During scrubbing, power is dissipated in the scrubber circuitry and in the programming interface of the FPGA. A more complex scrubber implementation with readback and compare naturally leads to a higher power consumption, which has to be put in relation to the overall system power consumption. Using the variable scrubbing rate briefly discussed earlier, power consumption can be reduced [36].

Scrubbing can be seen as complementary technique to fault-masking or fault-tolerance techniques such as TMR. It is also important to understand the limitations of the selected scrubbing scheme, as the scrubber may not be able to scrub all parts of the system (although depending on the application this may not be necessary), and may itself be susceptible to SEUs.

2.4.4 Error Correcting Codes

Error Correcting Codes, ECC, are additional bits added to a data sequence in order to detect and potentially correct errors in that data sequence, according to some algorithm. ECC can be used to verify data integrity in fault-tolerant FPGA designs. In this section, the use of ECC for protection of user data, configuration memory and state machine states is discussed.

2.4.4.1 State Machine Encoding

State machines are common design elements in both ASIC and FPGA designs. A state machine is defined by its inputs, outputs, state, state transitions and initial state, and can be written as $F = (I, O, S, \partial, S_0)$. The current state, S, is stored in DFFs and encoded as a binary value. Selecting the state encoding scheme with respect to SEUs is important to avoid risking the state machine going into an undefined, potentially unrecoverable state. As a basic rule, all states possible with the selected encoding scheme need to be defined, and there must be no potential *hang state* where the state machine is stuck indefinitely. Different encoding schemes exist for state machines, including *Gray coding* (using a *hamming distance* of 1), *One-Hot* and general Hamming codes (Hamming distance of >1). One-Hot encoding uses as many flip-flops as it has states, which is a resource-costly approach. However, it also has the advantage of simple combinatorial logic for state transitions. In [43], the authors provide a set of criteria for fault-tolerance encoding schemes for state machines. The authors go on to compare Gray coding, One-hot, *Hamming-2* and *Hamming-3* encoding. It is found that Hamming-3 provides the best fault tolerance with respect to SEUs. However, it is also the slowest and most resource-demanding encoding scheme.

2.4.4.2 Configuration Memory ECC

As mentioned in Sec. 2.2, the configuration memory of a Virtex 5 FPGA is protected using ECC on a frame-level and a CRC value for the whole configuration memory. The error correction scheme and details described here are specific for Virtex devices, but the principles are general. In Virtex 5, each frame is protected by a 12-bit ECC value using Hamming code, calculated by the FPGA design tools when generating the bit-stream file. An ECC macro, named FRAME_ECC_VIRTEX5, can be instantiated by the user for error detection as part of a SECDED scheme. A configuration frame is read back through one of the available programming interfaces and stored by the SECDED circuitry. The ECC and the frame content are used to calculate a *syndrome* value [18], which indicates the existence of an error as well as the location within the frame. The identified bit can be inverted, and the whole frame can be written back, correcting the error. This applies for a single-bit error within a frame. Multiple bit errors can be detected, but not located, making a reconfiguration of the entire frame necessary.

The syndrome value is a 12-bit vector, where the MSB indicates the presence of an error, and the remaining 11 bits are used to locate the error within the frame. As a configuration frame in Virtex 5 consists of 1,312 bits, 11 bits are sufficient to address any bit in the frame. If no error exists, the syndrome value should consist of all zeroes. In case the MSB is 0, but the 11 location bits are not all zeroes, this indicates the presence of a multiple bit error [18].

The whole configuration memory is protected by a 32-bit CRC value. This value is calculated for the original configuration and stored by a configuration manager. When reading back the configuration memory, the configuration manager can calculate the CRC value and compare it with the original. If there is a discrepancy between the calculated and the original CRC values, an error is present somewhere in the configuration memory [18].

2.4.4.3 User Data Protection

Given the variable nature of *user data* in FPGA, there is no golden reference with which to compare, as discussed in Sec. 2.2. Using the two user data error modes defined in Sec. 2.2 as a starting point, we discuss the use of error correcting codes to protect the contents of DFFs, distributed RAM and BRAM. Single DFFs need to be protected by redundancy, for example using the approach in [44]. For registers or arrays of DFFs, it is possible to implement error correcting code in logic, although this is not commonly done.

The 32kB Block RAMs in Xilinx Virtex-5 can be protected by enabling the built-in 64-bit ECC, which is able to detect and correct single-bit errors, and to detect double bit errors. This requires the (dual-port) BRAM to be configured as a 512x64 bit memory block, where the two ports work as dedicated read and write ports, respectively [17]. The ECC is based on Hamming code and uses eight parity bits to implement *SECDED*. If a single-bit error is detected, it will be automatically corrected. This will be signalled by an output signal. Double bit errors cannot be automatically corrected, but they can be detected, which is also signalled by an output. Internally, the BRAM ECC module consists of an encoder and a decoder, as well as the required MUX structure. This is illustrated by Fig. 35. The module interfaces with the BRAM block, working as an intermediate between the BRAM block and the user. The addition of an ECC module will typically not be directly noticeable from the BRAM user's point of view, but it will affect timing. The encoder and decoder blocks can be configured to be used individually. In [45], the authors present an EDAC setup and calculations of the EDAC Word Error rate for BRAM blocks in a radiation-hardened version of Xilinx Virtex 5.

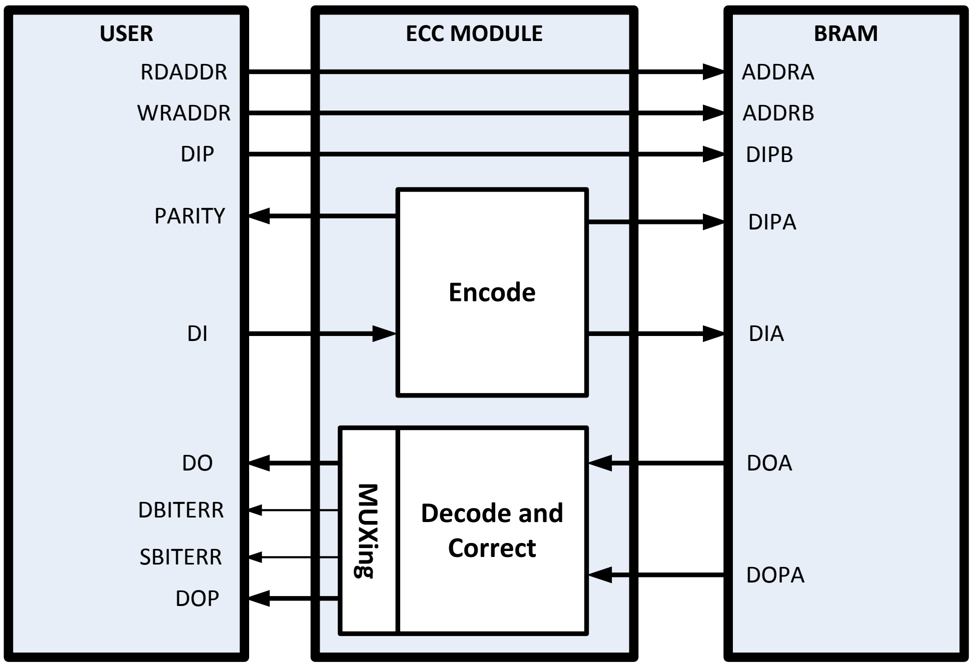


Fig. 35. ECC Protected BRAM

2.4.5 Checkpointing and Rollback

Checkpointing is the technique of periodically saving the (known-good) state of a circuit, in what is commonly known as a *snapshot*. This allows the system to revert (*Rollback*) to the last checkpoint when detecting an error, to recover from the error and continue execution from a known-good state. Checkpointing has been commonly implemented in software in fault-tolerant processor systems[23], [46], [47], but can also be applied to general logic or to soft-core processors in programmable logic. Applying hardware checkpointing is aimed at reducing the overhead effects of more expensive fault-tolerant techniques such as TMR. The concept of checkpointing requires mechanisms to store system states, detect errors through scrubbing readback, and to perform rollback to restore the system state. This inevitably introduces some time losses, as the error is not masked in the same way as for TMR implementations, but this may be feasible for some applications. Checkpointing can be implemented on-chip or in external circuitry. Fig. 36 illustrates the checkpointing principle.

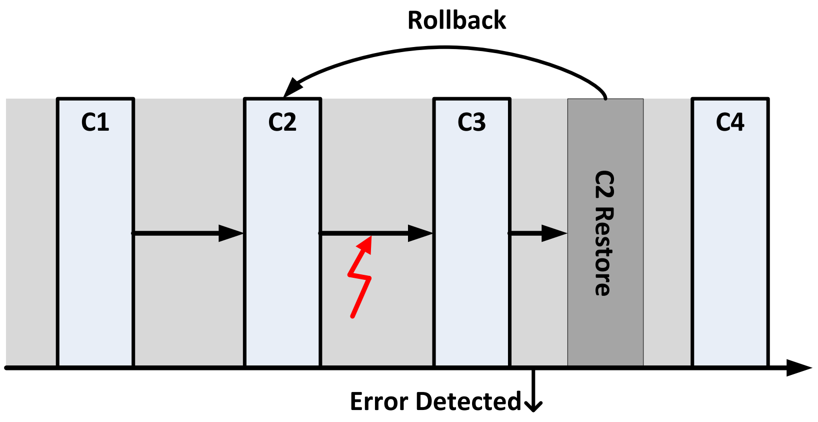


Fig. 36. Checkpointing and Rollback Principle

In FPGA, internal checkpoint saving can be made very efficient by leveraging the high on-chip bandwidth capabilities. However, the checkpointing structures are themselves susceptible to SEUs, and checkpointing may not be trivial or suitable to implement for all types of applications. When implementing a soft-core processor based system on FPGA, checkpointing can be a low-cost alternative to TMR [48]. As checkpointing only restores the system state, it should be combined with error correction and build-up prevention techniques (Configuration Scrubbing). The frequency of checkpoint storing has to be weighed against scrubbing frequency and timing requirements. The scrubber has to be able to scrub the entire memory between checkpoints, setting an upper limit to the frequency. Checkpointing with a low frequency, on the other hand, leads to potentially long recovery latencies due to the required re-execution [47]. As checkpointing is well suited for soft-core processor systems, it may be a good idea to combine checkpointing for suitable modules in the system with TMR for more critical parts, as well as normal configuration memory scrubbing being performed in parallel. Checkpointing must, however, be synchronised with the scrubber's readback and repair activities to be meaningful [48].

2.4.6 Temporal Redundancy

Temporal redundancy differs from *spatial redundancy* techniques such as TMR, in that calculations are repeated in time instead of being performed in parallel redundant paths. While temporal redundancy has the potential for lower area overhead compared to TMR, there are some obvious disadvantages [24]. Firstly, since all computations have to be repeated (commonly three times), there is obviously an increase in the execution time for a task. Also, if there is a permanent error in the configuration memory, all the time-redundant computations will exhibit the same error in their outputs. This makes temporal redundancy mainly useful for mitigating transient errors only.

Another type of temporal redundancy utilises clock skew on sequential elements to mitigate SETs. By using, for instance, three separate clock signals skewed by 90°, combined with voting circuitry, transient glitches on input signals from combinatorial logic can be masked. As for the technique of repeating calculations in time, *clock skewing* does not offer protection against configuration errors, and provides a rather weak alternative for SEU mitigation for space applications.

2.4.7 Tool-Level Techniques

Certain SEU mitigation and prevention techniques for FPGA can be implemented on a design tool level, and are focused on generating an inherently more robust configuration bit stream. This can be done in several different ways. *SEU-aware Place And Route* (PAR) techniques incorporate metrics and weight functions into the PAR algorithm to optimise the placement and routing of resources on the FPGA with respect to error avoidance [49]. Traditional FPGA design tools often try to optimise for speed or area, potentially removing intentional redundancy and sabotaging the fault-tolerant design techniques employed by the designer. There are also vendor-supplied redundancy tools, such as the previously mentioned XTMR tool from Xilinx, which are specifically targeted at automatically adding redundancy to the design. The efficiency and customisation opportunities of such designs may be limited, or at least not in line with what the designer wants, so there may be a trade-off between engineering effort and redundancy efficiency.

When focusing on SEU mitigation, it is typically of interest to try to optimise the placement of resources for scrubbing purposes. This can mean packing logic into fewer configuration frames and utilising each frame more to reduce the number of sensitive frames, even if it makes less sense when optimising for performance alone. Having fewer sensitive frames to scrub can help reduce the MTTR, provided that the *sensitive frame map* is available to the scrubber, or incorporated into the scrubbing sequence.

When using voting redundancy, such as TMR, it may be beneficial to place the redundant paths physically apart on the chip to reduce the possibility of a single fault affecting multiple redundant paths. This, however, introduces a degree of routing overhead and potential timing issues when routing signals from the redundant paths to connect them in voter stages.

For SET prevention, retiming algorithms can be applied on the tool-level to prevent the propagation of SETs, as discussed in [50]. A low-overhead method was presented in [51], which utilises restructuring of the logic expressions and LUT contents in combination with spare logic in the CLBs in Xilinx Virtex FPGAs to provide some degree of error masking. This method will here be referred to as *Logic Decomposition, Logic Restructuring* (LD+LR).

2.4.7.1 SEU-Aware PAR

The basic goals of FPGA PAR tools are to provide a physical allocation of the synthesised design modules onto the FPGA resources, to provide routing paths in an efficient and minimal way, and to if possible meet the timing constraints set by the designer. Traditional PAR tools are based on the concept of *simulated annealing*, where design blocks are placed randomly in the initial placement stage, then iteratively improving the placement through the swapping of blocks. In determining whether two blocks should swap places or not, a *cost function* is used. The placement cost function weighs the wiring and timing costs, and determines the improvement value of the swap by computing a delta value between the old and the new placement. As the PAR algorithm progresses, in simulated annealing referred to as *lowering the temperature*, the algorithm accepts less and less *bad* swaps. The idea of starting at a *high temperature* is to allow some bad swaps to take place in order to overcome local minima, where less sophisticated greedy PAR algorithms risk getting stuck. A simplified version of a cost function is given by (18), as defined in [52] and used in [49]. Here, λ is a user-defined weight constant, and the *Previous Cost* refers to the previous temperature step.

$$\Delta Cost = \lambda \frac{\Delta Wiring\ Cost}{Previous\ Wiring\ Cost} + (1-\lambda)\frac{\Delta Timing\ Cost}{Previous\ Timing\ Cost} \tag{18}$$

In [49], the authors suggest adding an SEU-sensitivity component to the cost function, using the VPR tool [53]. The modified placement cost function as presented in [49] is given by (19). Here, σ is a user-set constant similar to λ.

$$\begin{aligned}\Delta Cost = \lambda \frac{\Delta Wiring\ Cost}{Previous\ Wiring\ Cost} + (1-\lambda)\frac{\Delta Timing\ Cost}{Previous\ Timing\ Cost}(1-\sigma) \\ + \sigma \frac{\Delta SEU\ Cost}{Previous\ SEU\ Cost}\end{aligned} \tag{19}$$

The authors go on to define three wiring fault categories, similar to what was discussed in Sec. 2.3.1.1: *Shortened*, *Open* or *Bridging* faults. When employing a TMR scheme, *bridging effects* can effectively break the redundancy by bridging together two of the redundant paths. The basic principle for avoiding short and open wiring faults in PAR is simple: shorter wires mean fewer PIPs to upset. Bridging faults are more complex to avoid, and generally involve placing sensitive modules so that they do not share connection points more than necessary. For routing, the authors of [49] include an SEU cost component similarly to what is done for placement.

A PAR algorithm specifically targeted at keeping TMR implementations robust is presented in [22]. The algorithm, named *RoRA* for *Reliability-Oriented place and Route Algorithm* by the authors, attempts to route the signals between the redundant paths so that upsets cannot affect both paths.

While SEU-Aware PAR algorithms on their own do not mask or prevent errors, they assist to a degree in avoiding routing errors and errors affecting multiple modules, without inferring much overhead in terms of area and power. For instance, the authors in [49] present a 22% improvement (reduction) in SEU-susceptibility at a cost of 5% critical path delay and 8% increased power consumption when using SEU-aware placement and routing.

2.4.7.2 LD+LR

In[54], [21], [51], low-overhead fault masking techniques that are discussed utilises LUT content restructuring combined with spare logic in the CLBs of a Xilinx Virtex device. The proposed methods are quite similar in nature, and offer a degree of fault masking where the area overhead will depend on LUT utilisation. This is an attempt to reduce the massive area and power penalties imposed by TMR. The basic idea is to utilise the fact that in (Xilinx and Altera) FPGAs, the *n-input* LUTs are made up of two *(n-1) input* LUTs (for example, a 6-input LUT in recent Xilinx Virtex FPGAs consists internally of two 5-input LUTs, here denoted as sub-LUTs for simplicity). If less than all inputs to a LUT are used, there will be a free sub-LUT, giving the possibility to duplicate the LUT content into the two sub-LUTs. The technique presented in [51], called *LD+LR*, is a post-PAR design step that extracts the LUT information from the bit stream and attempts to perform *logical restructuring* on the LUT content to maximise the number of 0's or 1's in a LUT. To facilitate this, logic expressions may need to be broken down into components implemented in different LUTs.

Fault masking is achieved by using spare logic gates in the CLB on the outputs of the internally duplicated LUTs. A $0 \rightarrow 1$ error can be masked by an AND operation on the outputs of the LUTs, and similarly a $1 \rightarrow 0$ error is masked by an OR operation. Fig. 37 shows an example where a 5-input logic function is duplicated inside a 6-LUT, the number of 0's maximised in the LUT content, and the outputs AND-ed to mask $0 \rightarrow 1$ faults. In [51], the authors achieve an 85% fault masking level with only a 7% increase in slice usage on a Xilinx Virtex 6. This requires additional steps in the design flow, as the LD+LR algorithm is applied post-PAR and then resynthesised.

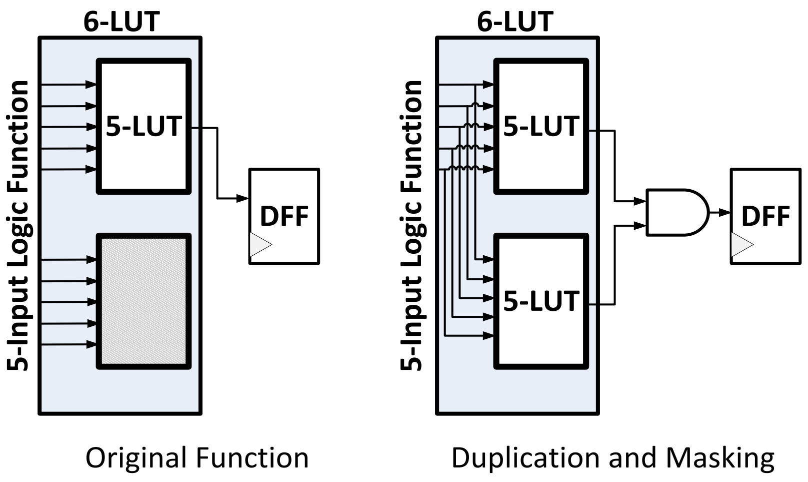


Fig. 37. LD+LR Masking

3 Method

Testing SEU mitigation techniques requires some form of testing method that allows the introduction of SEUs in the design and observation of the behaviour. The obvious approach is to subject an FPGA with the implemented design to actual radiation, using a particle beam with known characteristics. This is known as *in-beam testing*. While in-beam testing may offer the most realistic testing environment, it also requires access to a radiation source and advanced equipment. The test process is both expensive and time-consuming. For certain important numbers, such as TID tolerance, vendors commonly supply in-beam testing results for their devices.

Gate-level simulation may be used as normal, with the addition of introducing so called *stuck-at errors* to fix a bit to a logical 1 or 0. While it is not particularly useful for testing the overall susceptibility of a system, it can allow detailed analysis of fault propagation, as the designer can trace signals through the system in a predictable way. Analysing the configuration bit stream can give an idea of what stuck-at errors to simulate and trace.

Another alternative, *fault injection*, refers to the method of injecting faults into the configuration bit stream or user data. This can be achieved in a number of ways, for example by intentionally programming an FPGA with an incorrect bit stream, or to use dynamic partial reconfiguration to introduce faults. Fault injection can be implemented in hardware or software, and there are fault-injection software suites available for FPGA, most notably *FLIPPER,* discussed in [55]. The Xilinx SEU Controller Macro discussed earlier includes functionality to inject single or multiple bit errors into the design, at a user specified or random frame address [42]. How fault injection is implemented is an important aspect in designing an evaluation method for fault-tolerant techniques, and will be discussed further in Sec. 3.3.3.

While frame-level fault injection might not be quite as realistic as in-beam testing, it can certainly be very close when used in conjunction with a good SEU-rate estimate. Fault-injection techniques also offer several advantages. It is far less time consuming and costly compared to in-beam testing, and offers good error-logging possibilities, as well as a predictable behaviour for studying specific parts of the design or specific error modes. One drawback is that it is not possible to simulate some SET effects. A combination of using fault injection and gate-level simulation can prove very efficient for evaluating SEU mitigation techniques, and will be used in this work. This section describes the implementation and use of an FPGA-based test platform for evaluating SEU mitigation techniques, with focus on TMR and Scrubbing implementations.

3.1 Test Platform

In order to test and evaluate the mitigation techniques introduced in the previous chapters, a test platform has been implemented. The purpose and aim of the FPGA test platform is to provide a common base structure on which different test payloads can be implemented and tested. In designing the test platform, a few basic functions were identified: a platform needs to provide fault injection, fault detection and logging, and an exchangeable payload. To facilitate this, and to be practically implementable on an FPGA board, the test platform also needs to provide interfaces to off-chip resources, such as memory.

Since the test platform is implemented to test and evaluate mitigation techniques on the FPGA, test data and results needs to be collected and stored. In order to provide easy controllability and to send back test data, communication with a PC is required. Also, it is desirable to move complexity from hardware (on the FPGA) to software (on a PC) in order to improve data logging capability and to reduce the time and effort required in developing. This subsection describes the test platform and the board it is implemented on.

3.1.1 Board

The platform is implemented on a Virtex-5 LX50 Evaluation Board from AvNet [56]. The board is based around a Xilinx Virtex-5 FPGA (XC5VLX50) and includes two DDR2 memory modules (totalling 64MB), a 16 MB flash memory, a 32 MB xcf32p Xilinx Platform Flash and connectors for RS232, Ethernet and USB. The board is also equipped with a JTAG connector for programming, a small LCD display, generic I/O pins, a clock synthesiser, as well as push buttons and LEDs for user interaction. A 100 MHz oscillator provides the base clock to the FPGA. Fig. 38 shows the evaluation board. A block diagram of the most important components is presented in Fig. 39.

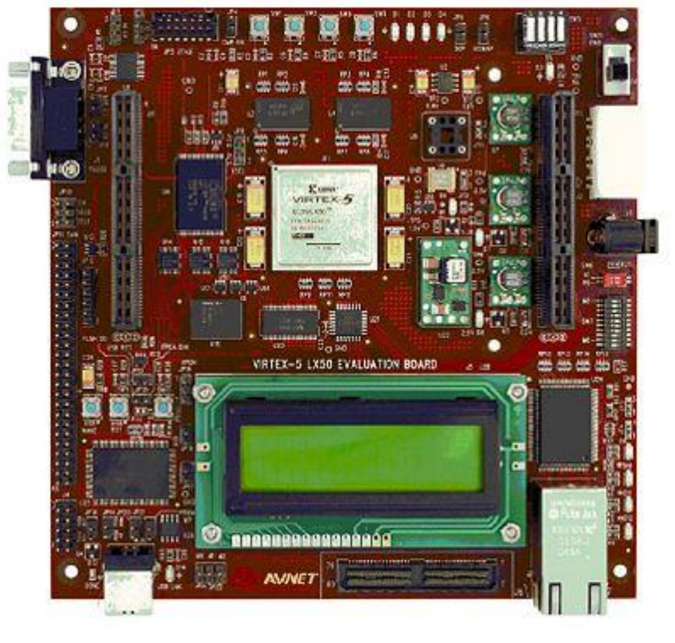


Fig. 38. Virtex 5 Evaluation Board

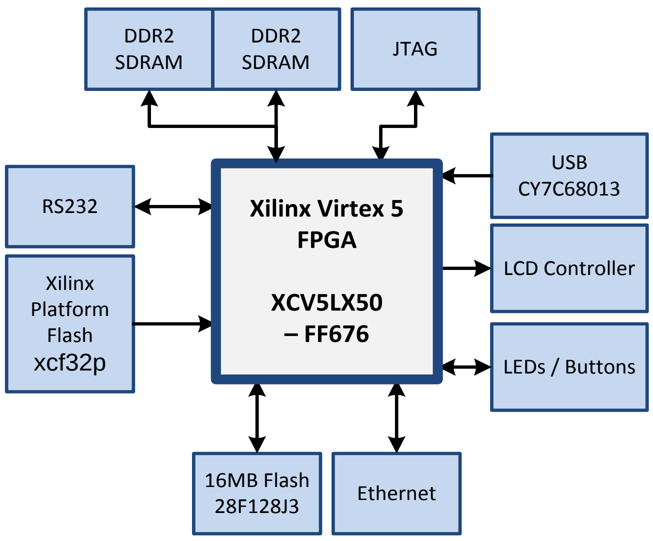


Fig. 39. Virtex 5 Evaluation Board, Block Diagram

The Virtex-5 LX50 has a fixed speed grade of -1 and is implemented in an FF676 package. Table 4 gives an indication of the number of the most important resource types available on the FPGA.

Table 4. XC5VLX50 Resources

Resource	Number / Amount
Logic Slices	7,200
Block RAM	1,728 kB
Xtreme DSP Slices	128
DSP48E Slices	48
DCMs	12

3.1.2 FPGA Test Platform

The FPGA Test Platform architecturally consists of three main parts: A bus structure and system framework, a test structure, and the payload itself. By keeping the payload separate from the rest of the test framework, it can easily be replaced and acts as a Device under Test, DUT. Fig. 40 gives an overview of the FPGA platform, its off-chip memory interfaces and the PC communication.

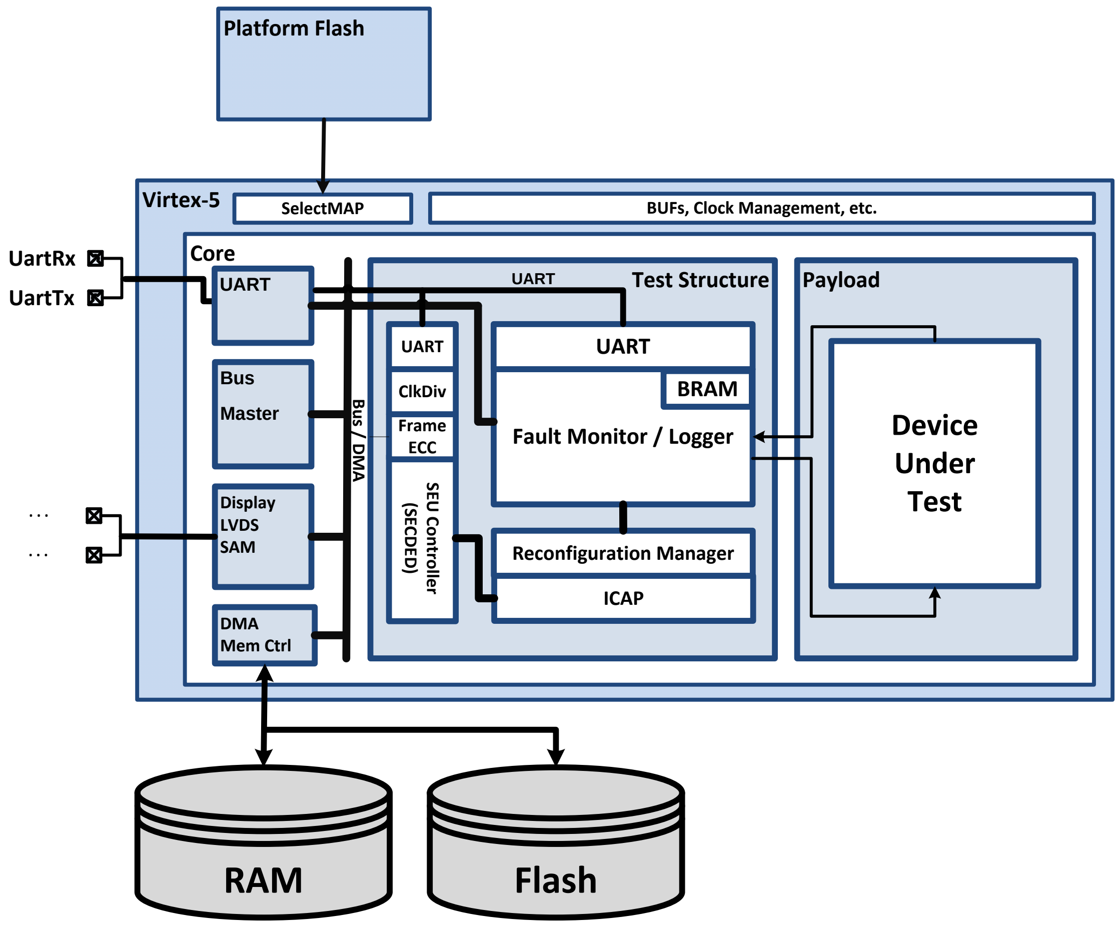


Fig. 40. FPGA Test Platform

3.1.2.1 Bus Structure and System Framework

The bus structure includes a bus-master, DMA handling and several peripheral units connected to the central bus network. The bus network consists of a memory access bus (for DMA) and a register bus. Peripheral units include a UART receiver/transmitter, an LCD interface, a DDR2-SDRAM interface and a Flash memory interface. Units connected to the π-bus can access configuration registers and DMA channels through access requests to the bus master. When a unit on the bus makes a DMA request or register access, the bus master acts as an arbiter, handling all mapping and interfacing to the off-chip memories. The system framework handles all the underlying pin mappings and clock management. The system and bus structure are run on a 40 MHz clock derived from the on-board 100 MHz oscillator.

3.1.2.2 Test Structure

The test structure handles fault injection, result checking, error logging and scrubbing/reconfiguration. The test structure consists of three parts: SEU Controller, SEU Monitor and Reconfiguration Manager.

3.1.2.2.1 SEU Controller

The SEU Controller is an instantiation of the Xilinx SEU Controller Macro described earlier, and implements SECDED functionality and fault injection capability through the use of Frame ECC and ICAP primitives. An UART interface enables communication with a PC through a set of commands and status send-backs. The controller is able to detect and correct (single-bit) configuration-memory errors, and can detect and indicate multiple bit errors.

3.1.2.2.2 SEU Monitor

The SEU Monitor is a module responsible for testing the DUT and storing the results in the memory. Test vectors are kept by the monitor and sent as inputs to the DUT, which then sends the computed results back to the monitor to be compared with the expected result. The monitor then stores a status vector to memory, indicating any errors that may have occurred in the DUT. Similarly to the controller, the monitor implements an UART interface for receiving commands from a PC and sending back the logs when requested. The monitor’s set of commands does not overlap with those of the controller, allowing both units to listen to the same UART receive channel at the same time. Commands that are not recognised are simply ignored. The monitor can store test log data to an internal BRAM, or to the 16MB off-chip flash memory through DMA requests on the bus. The flash memory option is included for tests that require reconfiguration of the whole FPGA, in which case the test logs need to be kept in a non-volatile off-chip location. BRAM is the default storage location for the test logs. Fig. 41 shows an example of a test run.

The communication between the SEU Monitor and the DUT consists of the input and output data, as well as *run* and *done* signals. These allow the Monitor to control when a test is started through the DUT. Having as few signals to and from the DUT as possible is important to maximise the separation between test framework and payload.

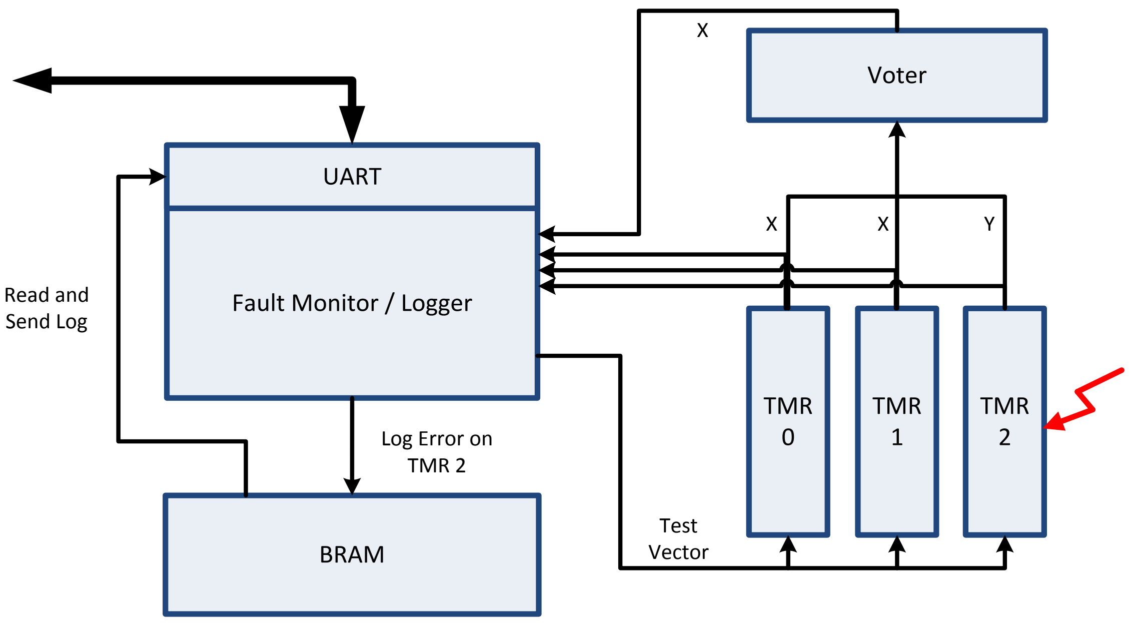


Fig. 41. Example Test Run

In this example, a TMR implementation of the DUT is tested. A fault is inserted in the DUT through the SEU Controller, which in this example has hit one of the redundant branches, TMR 2. The monitor then receives a command from the PC telling it to run the test, and proceeds with sending the test vector inputs to each of the three TMR branches. The injected error causes TMR2 to produce an incorrect output. The monitor logs all outputs from the redundant branches as well as from the voter stage, and stores a log word to BRAM, indicating that one of the redundant branches has produced an incorrect output, but that the voter stage has managed to mask the error. Upon receiving another command from the PC telling the monitor to send the log data, the entire log stored in BRAM is sent over UART to the PC in a raw hex format. The log data is received and saved by software on the PC.

The raw log format stored in memory by the monitor uses 16 bits for each test run. The meaning of each of these bits will depend on the test points used in the DUT and will have to be interpreted accordingly. As an example, when testing a TMR design with tripled voters, one would typically wish to log the status (correct or incorrect) of the outputs of each of the TMR branches and of each of the voters. An example is shown in Table 5.

Table 5. Log Line Example

	Unused	R_2	V_2	R_1	V_1	R_0	V_0	TMR_2	TMR_1	TMR_0
Bit	**[16:9]**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	-	1	1	1	1	1	1	0	1	1

In this example, TMR0-2 indicates correct output on the redundant branches, R_n indicates correct output from Voter n, and V_n indicates that Voter n correctly identified if it has a majority result or not. This particular log line, corresponding to a single test run, would indicate that an incorrect output has been detected from TMR branch 2 as a result from the injected fault, but that the error was masked by the voters. This log line is logged by the SEU Monitor as *0xFFFB*.

3.1.2.2.3 Reconfiguration Manager

Interfacing to the configuration memory is done through an ICAP interface instantiated in the reconfiguration manager. The reconfiguration manager is responsible for scrubbing, either in the form of dynamic partial reconfiguration, or as a simple full reconfiguration that can be triggered by the monitor module. A full reconfiguration is triggered by setting a single trigger signal from the test framework, which triggers a state machine sending a sequence of commands to the ICAP interface. The reconfiguration manager, or the *Scrubber*, can be modelled as internal or external. This is done by including or excluding the configuration frames corresponding to the scrubber in the fault insertion frame list, respectively.

3.1.2.3 Payload

The payload represents the actual device under test. The payload can easily be exchanged, depending on what needs to be tested. A payload has a set of test points which depends on the application. In the monitor example given in 3.1.2.2.2, the output from each of the TMR branches is a separate test point, as well as the output from the voter stage.

3.1.2.4 Selecting a Payload/DUT Application

For evaluating mitigation techniques, a 128-bit AES application has been selected. The AES, or Advanced Encryption Standard[57], block takes as input a 128-bit key and a 128-bit plaintext to be encrypted. Output consists of a 128-bit encrypted text. The application was chosen partly because of its extensive documentation and practical applications, partly because of the computational properties: because each step in the computation depends on the previous steps, an error somewhere in the computation chain is likely to propagate through to the output. Using an AES block implementation for testing is also a good match for testing both gate-level and module-level redundancy techniques, as well as being suitable for dynamic partial reconfiguration. Furthermore, output data is easy to verify, and there are a large number of test vectors available. For each test that is run by the monitor, a total of 283 test vectors are run and compared with an expected, pre-computed result. These test vectors are specified in the AES standard.

The AES-128 block used here do not make use of BRAM or DSP slices on the FPGA, instead using only logic and sequential elements. A set of constant values are used for the so called *sbox* in the AES implementation. The AES algorithm will not be discussed in detail here. Instead, readers are referred to the standard, given by [57].

An FIR filter application has also been implemented and tested. However, due to time-consuming test sequences and lack of time, this report puts focus on the AES application. Two other DUT applications were used in preliminary testing stages: a simple shift register chain, and a telemetry decoding module. These were abandoned in favour of the AES block. The shift register chain offers simple observability, for example by shifting an XOR-pattern, but does not use the logic resources in an efficient manner with respect to fault injection. Furthermore, the huge shift register needed to make up a feasible target for fault injections makes up a very unrealistic application. The telemetry decoder module on the other hand is a highly realistic and usable application. However, it lacks obvious testing points, making it difficult to set up a fault model for analysis, and it is too big to be conveniently contained and monitored by the test framework.

3.1.2.5 VHDL Implementation

The test platform is implemented as a configurable VHDL model, following the RUAG Space coding and design standards. The design follows (although not strictly) a flat hierarchy approach. Records, code blocks and local signals are used where appropriate. The implementation follows the same module composition as shown in Fig. 40, and is largely built around the bus structure. Table 6 shows the resource usage of the test framework, not including the DUT instance.

Table 6. Test Framework Resource Usage

Slice	Slice Reg	LUT	LUTRAM	BRAM	DSP48E	DCM
2,089	2,789	3,844	108	15	3	2

The 40 MHz internal bus clock is used to clock all modules attached to the bus. The SEU Controller is clocked at 50 MHz, deriving its clock signal directly from the 100 MHz clock. The DDR2 memory makes use of a 200 MHz clock signal to communicate with the DDR2 on the board. Fig. 42 illustrates the bus structure. The blocks shown in the figure correspond to VHDL entities. As the SEU controller operates in a different clock region compared to the rest of the system, asynchronous interfaces are implemented where signals cross the clock domains. The clock management and buffers are defined in an underlying layer, together with pin assignments.

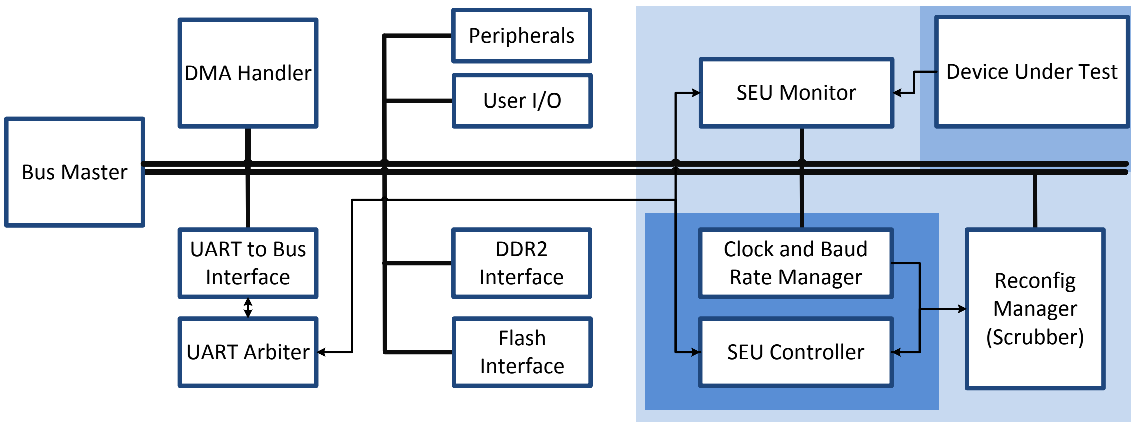


Fig. 42. Test Framework Bus Structure

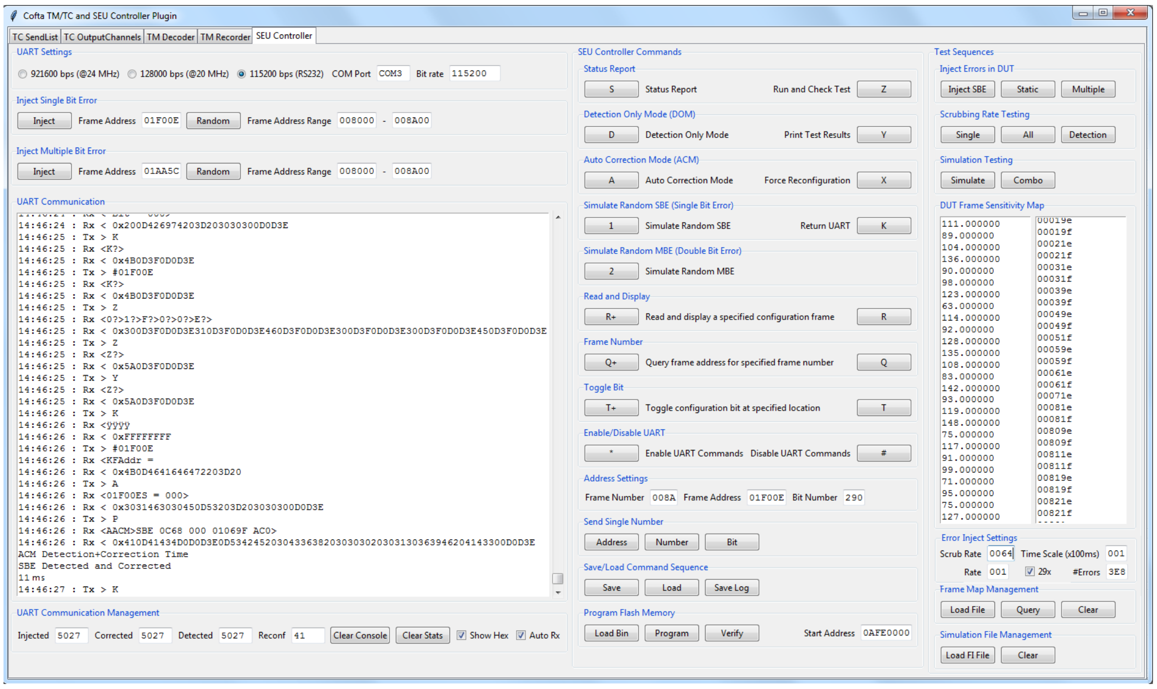
The UART arbiter controls which UART module has access to the transmission line (Tx) on the board. The three modules that implement UART interfaces (UART-to-Bus interface, SEU Monitor, SEU Controller) can request the outgoing Tx line, and the arbiter assigns it according to what mode the test platform is in. The peripherals box included in Fig. 42 represents the Ethernet, SAM, LVDS and LCD Display modules.

For Synthesis, a normal Synplify flow is used. The netlist (.edf) file created by Synplify is then imported to Xilinx PlanAhead for PAR. Sec. 3.3.1 mentions more on the PAR approach used. A user constraints (.ucf) file is used in PlanAhead for pin mappings and clock signal constraints.

3.1.3 PC Communication and Software

Communication with a PC is achieved through the on-board RS232 interface, using a fixed baud rate of 115,200 kbps. In total, the test platform implements three separate UART receivers/transmitters. The first UART interface connects the PC to the bus structure, allowing the PC to read and write data to registers and memories through the bus master. The second and third UART interfaces belong to the SEU Controller and Monitor, respectively. An UART arbiter decides which UART receiver/transmitter is allowed to write to the outgoing UART transmission pin. This setup allows the PC to send commands that switch between register/memory mode and test mode. Furthermore, it gives the possibility to move some of the complexity of the test procedures to software on the PC side.

The host PC runs a Tcl/Tk application to monitor and control the test platform and test runs. By using the Tcl application, tests can be automated and scaled easily. Fig. 43 shows a screenshot of the application during running tests.



921600 bps (@24 MHz) 128000 bps (@20 MHz) [✓ 115200 bps (RS232)] COM Port COM3 Bit rate 115200

Inject Single Bit Error: Frame Address 01F00E Frame Address Range 008000 - 008A00

Inject Multiple Bit Error: Frame Address 01AA5C Frame Address Range 008000 - 008A00

```
14:46:24 : Rx < 0x200D426974203D203030300D0D3E
14:46:25 : Tx > K
14:46:25 : Rx <K?>
14:46:25 : Rx < 0x4B0D3F0D0D3E
14:46:25 : Tx > #01F00E
14:46:25 : Rx <K?>
14:46:25 : Rx < 0x4B0D3F0D0D3E
14:46:25 : Tx > Z
14:46:25 : Rx <0?>1?>F?>0?>0?>E?>
14:46:25 : Rx < 0x300D3F0D0D3E310D3F0D0D3E460D3F0D0D3E300D3F0D0D3E300D3F0D0D3E450D3F0D0D3E
14:46:25 : Tx > Z
14:46:25 : Rx <Z?>
14:46:25 : Rx < 0x5A0D3F0D0D3E
14:46:25 : Tx > Y
14:46:25 : Rx <Z?>
14:46:25 : Rx < 0x5A0D3F0D0D3E
14:46:26 : Tx > K
14:46:26 : Rx <ÿÿÿÿ
14:46:26 : Rx < 0xFFFFFFFF
14:46:26 : Tx > #01F00E
14:46:26 : Rx <KFAddr =
14:46:26 : Rx < 0x4B0D4641646472203D20
14:46:26 : Tx > A
14:46:26 : Rx <01F00ES = 000>
14:46:26 : Rx < 0x3031463030450D53203D203030300D0D3E
14:46:26 : Tx > P
14:46:26 : Rx <AACM>SBE 0C68 000 01069F AC0>
14:46:26 : Rx < 0x410D41434D0D0D3E0D534245203043363820303030203031303639462041433000D0D3E
ACM Detection+Correction Time
SBE Detected and Corrected
11 ms
14:46:27 : Tx > K
```

Injected 5027 Corrected 5027 Detected 5027 Reconf 41 [✓ Show Hex] [✓ Auto Rx]

Address Settings: Frame Number 008A Frame Address 01F00E Bit Number 290

Program Flash Memory: Start Address 0AFE0000

DUT Frame Sensitivity Map:

```
111.000000   00019e
89.000000    00019f
104.000000   00021e
136.000000   00021f
90.000000    00031e
98.000000    00031f
123.000000   00039e
63.000000    00039f
114.000000   00049e
92.000000    00049f
128.000000   00051f
135.000000   00059e
108.000000   00059f
83.000000    00061e
142.000000   00061f
93.000000    00071e
119.000000   00081e
148.000000   00081f
75.000000    00809e
117.000000   00809f
91.000000    00811e
99.000000    00811f
71.000000    00819e
95.000000    00819f
75.000000    00821e
127.000000   00821f
```

Error Inject Settings: Scrub Rate 0064 Time Scale (x100ms) 001 Rate 001 [✓ 29x] #Errors 3E8

Fig. 43. Monitor Application

When in register/memory mode, the application can access and modify the internal FPGA registers as well as any Flash or DDR content, through the π-bus structure.
In test mode, the PC interfaces with the test framework on the FPGA. The SEU Controller and SEU Monitor listen to the same receiver channel from the PC, and the transmission channel (to the host PC) can be set by sending certain commands from the PC. A list of commands is given below. The list of commands includes all the SEU Controller commands as specified in Table 7 as well as the SEU Monitor commands. Some of the commands require additional arguments, as indicated in the table.

Table 7. Test Platform UART Commands

Command	Target	Description
> S	Controller	SEU Controller Status Report
> D	Controller	Set SEU Controller in Detection Only Mode
> A	Controller	Set SEU Controller in Auto Correction Mode
> 1	Controller	Simulate Random SBU
> 2	Controller	Simulate Random MBU
> R **> #FrameAddress**	Controller	Read and display specified configuration frame
> Q **> #FrameNumber**	Controller	Query frame address for specified frame number
> T **> #FrameAddress** **> #BitNumber**	Controller	Toggle configuration bit at specified location
> Z	Monitor	Run all test vectors and log result
> Y	Monitor	Print entire log from memory to UART channel
> X	Monitor	Trigger reconfiguration sequence
> K	Monitor	Give UART control to Controller

The Tcl application has functionality to load a set of frame addresses corresponding to the DUT, in order to be able to randomly insert configuration bit upsets in the DUT only. It also includes functionality to load a Matlab-generated fault-injection pattern, storing logs, saving command sequences and programming the flash memory. How these Matlab fault injection patterns and DUT frame mappings are generated, is discussed in detail in Sec. 3.3.3. The Tcl application also supports running fully automated test sequences. These sequences are discussed in Sec. 3.3.10.

3.2 Implementing Mitigation Technique Candidates

The focus of this work has been put on evaluating TMR and Scrubbing techniques. This section describes the different versions of TMR and Scrubbing implemented. These implementations are then used as payloads in the test platform.

3.2.1 TMR Implementations

Four TMR variants have been implemented, based on the AES block. By setting generics in the VHDL code, the same AES payload entity can be configured for different levels of TMR. The AES payloads are made up of AES-128 encryption blocks and voter stages where applicable. The available TMR implementations are listed below.

Reference: A reference design that consists of a single AES block without any TMR protection. Output consists of a single 128-bit vector containing the encrypted result. This design is used as a baseline reference with which to compare the mitigated designs. All injected faults leading to errors will propagate through the reference design and cause an incorrect output. Only one test point is logged for the reference design, namely the output correctness.

Single Voter TMR: A module-level TMR design with a single voter stage. Output consists of the majority result and a majority indicator from the voter stage. A single voter design can mask errors on a single TMR branch (one of the AES modules), but will produce an incorrect output if an error occurs in multiple branches or in the voter itself. Five test points are used in the single voter design. These are the individual outputs from the redundant encryption blocks, as well as the result and majority indicator from the voter stage.

Triple Voter TMR: A module-level TMR design where the voter stage is itself tripled. The output consists of three sets of result and majority indicator signals, one pair from each of the voter stages. The output from each of the TMR branches is fed to each of the voter stages. This design is able to mask errors in the voter stage, as opposed to the single voter version. However, multiple TMR branch errors (the so called *Bridging Errors*) will still cause an error on the outputs. The test points used in the triple voter design are similar to the single voter ones, with the addition of two voter result and majority pairs. In total, nine test points are used.

Synplify TMR: A tool-level TMR design, where the reference design is synthesised using Synplify Premier. The synthesis tool applies TMR protection at a lower level than the single and triple voter TMR implementations. The interfaces look the same as the reference design from the outside. This is achieved by setting the Synplify attribute *radhardlevel = distributed_tmr* in the Synplify constraints file. The single test point taken from the Synplify TMR implementation is the correctness of the output. Letting the design tools apply TMR to a design removes the need for a designer to change the RTL code, thereby saving design time and effort.

To prevent the synthesis tool from removing redundant modules and paths, the inputs to the redundant branches are kept as separate ports in the VHDL entity. Synplify attributes *syn_keep* and *syn_preserve* are used on all signals and instances, respectively, in the AES payload. Not applying these will cause the synthesis tool to remove all redundancy. By passing a VHDL generic to the SEU Monitor instance, it will check for and log errors based on the test points for the specified design.

Fig. 44 shows block diagrams of the TMR implementations. Clockwise from the upper left diagram, these are: Single Voter TMR, Reference design, Synplify-applied TMR, Triple Voter TMR. The test points of each design are indicated in the diagrams.

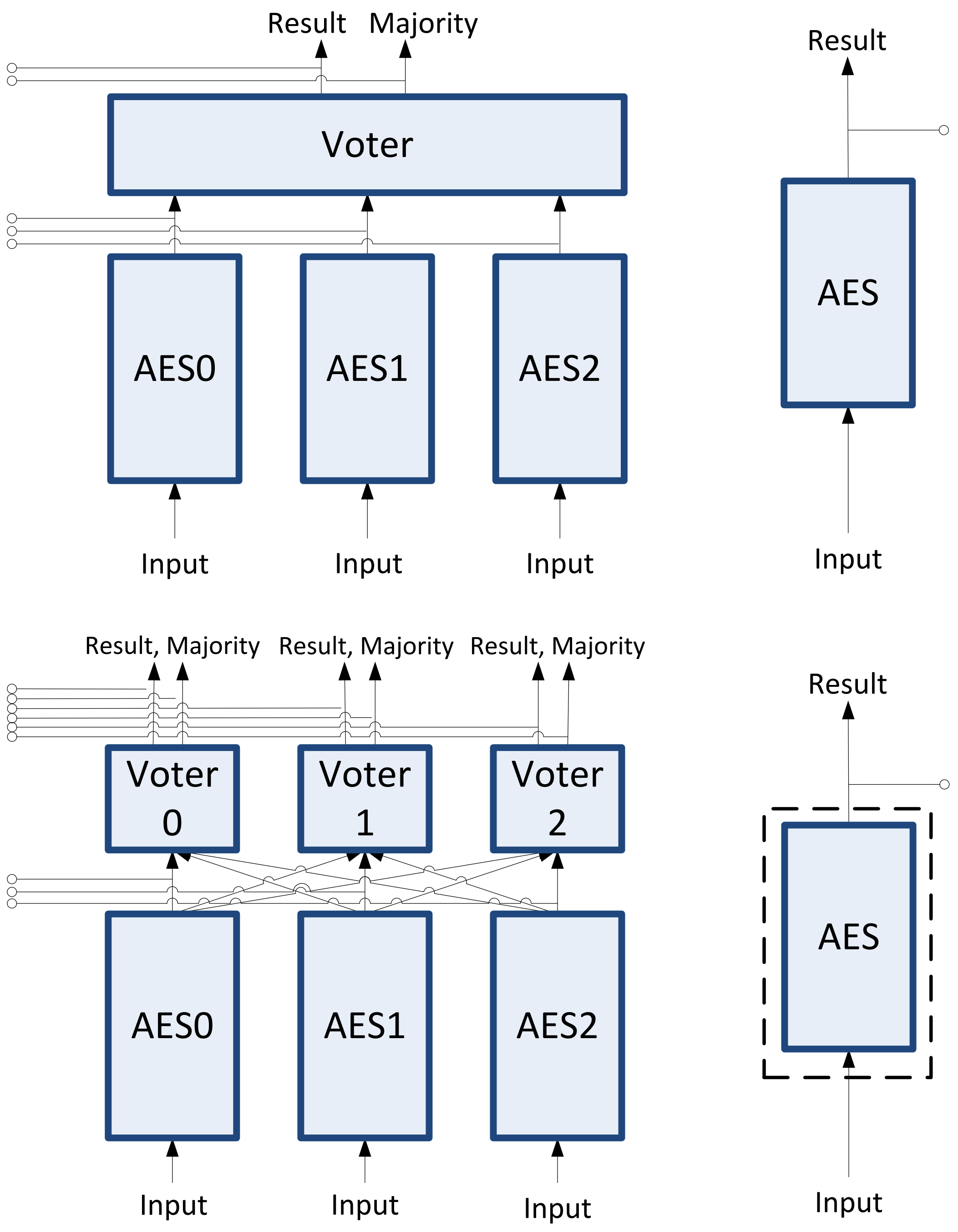


Fig. 44. TMR Implementations

3.2.2 Scrubber Implementations

This section describes the different scrubbing approaches implemented. These scrubbing implementations make up the Reconfiguration Manager module. Five different scrubbing approaches have been implemented, as listed below. Each of these scrubbing implementations can be combined with any of the TMR implementations. Also, by keeping the scrubber inside or outside the payload of the test platform, it can represent an internal or external scrubber implementation, respectively. This is illustrated in Fig. 45.

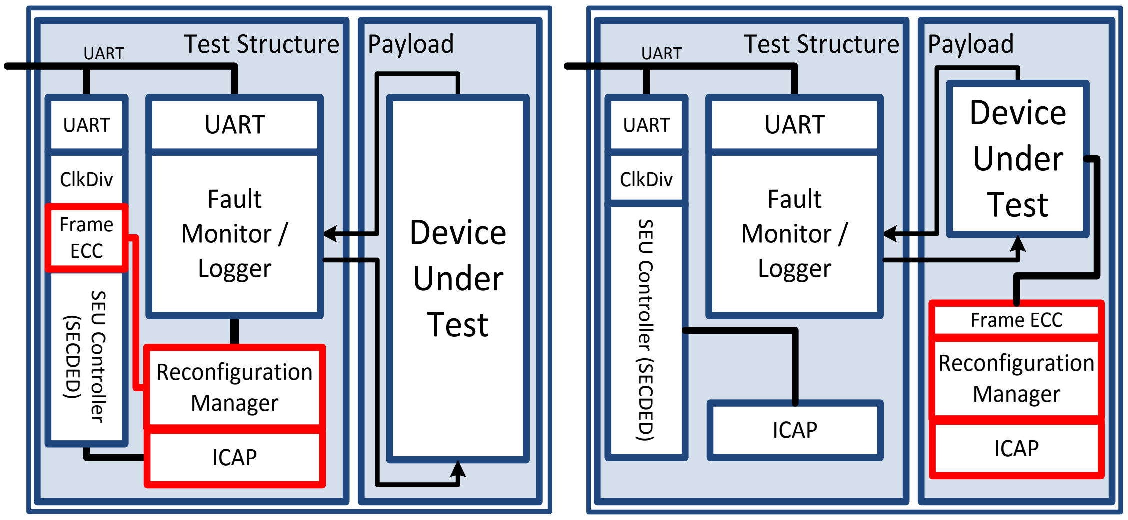


Fig. 45. External vs. Internal Scrubber

<u>Blind Scrubbing</u>: Using the Blind Scrubbing approach, scrubbing is done by fully or partially reconfiguring the FPGA periodically according to a fixed rate. In this report, full reconfiguration is used. This approach eliminates the need for error detection circuitry, but is not responsive to errors in the same way as scrubbing on error detection. Blind Scrubbing is referred to as *Scheduled Maintenance* in [35]. In terms of additional hardware required when implementing an internal scrubber on the FPGA, blind scrubbing is relatively cheap. A cycle counter and scrubbing-configuration register is required on-chip, as well as an ICAP controller. The counter can periodically trigger an FSM, sending a reprogramming command to the FPGAs configuration registers and triggering a read from an off-chip PROM/Platform Flash.

<u>CRC-Based Error Detection</u>: This type of scrubber performs a full reconfiguration upon detecting an error in the configuration memory CRC. CRC error detection is fast, but the scrubber is unable to pinpoint in which frame the error has occurred. In the design presented here, the CRC error signal originates from the Xilinx Virtex-5 FrameECC primitive. This is known as *Emergency Maintenance* in [35]. Apart from the FrameECC primitive with some supporting logic, an ICAP controller and a reprogramming command FSM are needed.

Frame ECC-Based Error Detection: Frame ECC-based scrubbing makes use of the same primitive as CRC-based scrubbing, but instead of triggering on the CRC for the whole configuration memory, the Frame-level ECC is used. This allows the scrubber to trigger a partial reconfiguration for that particular frame when an error is detected. Using partial reconfiguration allows scrubbing individual modules without interrupting, which may be beneficial in conjunction with some TMR implementations. For example, a Triple Voter design has no single point of failure (on a module level), and any one of the TMR branches or tripled voters can be down for scrubbing without interrupting the operation of the TMR'ed section. The downside to dynamic partial reconfiguration (DPR), however, is the overhead from a DPR manager block, and the buffering required.

Combination Scrubbing: The combination scrubbing option represents a mix of partial and full reconfiguration. It makes use of the SEU Controller macro for correcting single bits, using the macro's SECDED bit-toggling functionality. If a multi-bit error occurs, or if the SEU Controller is not able to correct an error, a full reconfiguration is triggered. This combines the method described in [58] and adds MBE correction by full reconfiguration. Fig. 45 shows a block diagram comparison of an external (left) versus an internal (right) scrubber implementation using the Frame ECC-based error detection method. The hardware overhead when using this approach consists of the SEU Controller macro, together with an ICAP controller and reprogramming FSM for full reconfigurations when an MBE is detected.

3.3 Test Method

3.3.1 FPGA Partitioning

As the DUT and the required framework will be implemented on the same FPGA it will be necessary to distinguish the test framework from the actual DUT. As faults will be inserted in a random manner, it is important that the DUT is isolated at a known interval of frame addresses. By having a known frame-address interval, containing solely the DUT, it will be possible to insert faults at random into the DUT only. Partitioning the FPGA in this way allows for realistic SEU simulations in the DUT, while keeping the test framework outside of the fault injection range. The partitioning starts at the VHDL description where it is important that the DUT can be contained within one or more component instances from the same level in the hierarchy. Declaring the DUT instances from the same layer will make it possible to easily partition the complete design into two p-blocks; one for the DUT and one for the framework. The modules belonging to the DUT and the test framework are placed inside two separate VHDL block structures. Once the design has been structured in this way it is synthesised, producing a netlist in the Xilinx .edf format.

With the netlist available, the actual physical partitioning on the FPGA can be made. PlanAhead is used for this task. PlanAhead facilitates Physical Block (*p-block*) partitioning by the use of a GUI and a blueprint of the target FPGA, documented in [59]. To separate the DUT, all of the involved instances are partitioned into a new p-block. This p-block is then placed on the device by picking an area on the device. Next, the remaining framework is put into a second p-block which is placed at a different location on the FPGA. Regard is taken to what resources are connected to which addresses to make sure that the design fits into the least number of frames possible. Fig. 46 illustrates the described separation of DUT and test framework. As can be seen from the figure, the DUT partition (highlighted upper right) is tightly packed and constrained, while the rest of the design is more relaxed with respect to resource allocation. The borders of the p-blocks are marked with purple lines.

Selecting the number of slices for the p-blocks is an important detail in the partitioning process. In order to maximise the probability of an SEU actually inducing a fault within a given frame interval (the DUT), p-blocks have to be as small as possible. By making the p-blocks as small as possible, a worst case scenario is tested. The smallest number of frames can be found by iteratively making the p-block smaller until the tool is no longer able to successfully partition it. Once the design is mapped and placed-and-routed a bit-file can be generated through the bitgen function in PlanAhead. Identifying the frame addresses that correspond to the DUT is done through a set of Python scripts. From the bitgen process in PlanAhead, a logic allocation (.ll) is generated. Table 8 shows an example excerpt from an .ll file.

Table 8. Logic Allocation File Excerpt

Bit	Frame	Bit	Slice	Latch	Net
. . .					
1067778	**0x00000a9f**	126	X33Y61	DQ	AesI_B.Aes1I/..
1067815	**0x00000b1f**	1182	X35Y77	CQ	AesI_B.Aes2I/..
1067838	**0x00000b1f**	1187	X35Y77	DQ	AesI_B.Aes2I/..
1067843	**0x00000b1f**	1309	X34Y78	AQ	AesI_B.Aes2I/..
1067902	**0x00000b1f**	1310	X34Y79	DQ	AesI_B.Aes2I/..
1067932	**0x00000b9e**	66	X37Y61	BQ	AesI_B.Aes2I/..

The logic allocation file contains information about what configuration frame addresses are mapped to each of the VHDL design entities. A Python script is used to parse out and list all the frame addresses corresponding to the DUT. The resulting list of frames makes up a sensitive frame map. This mapping is then loaded into the Tcl application on the PC, as described earlier. Being able to target the test platform to only inject faults into the part of the FPGAs configuration memory corresponding to the DUT is a basic requisite for this type of testing to work. As the DUT communicates with the SEU Monitor, there is a possibility of an injected error affecting the *run* and *done* signals of the DUT, or the input/output data signals to/from the DUT. This is however an unlikely scenario, and tests that are affected by this can simply be re-run.

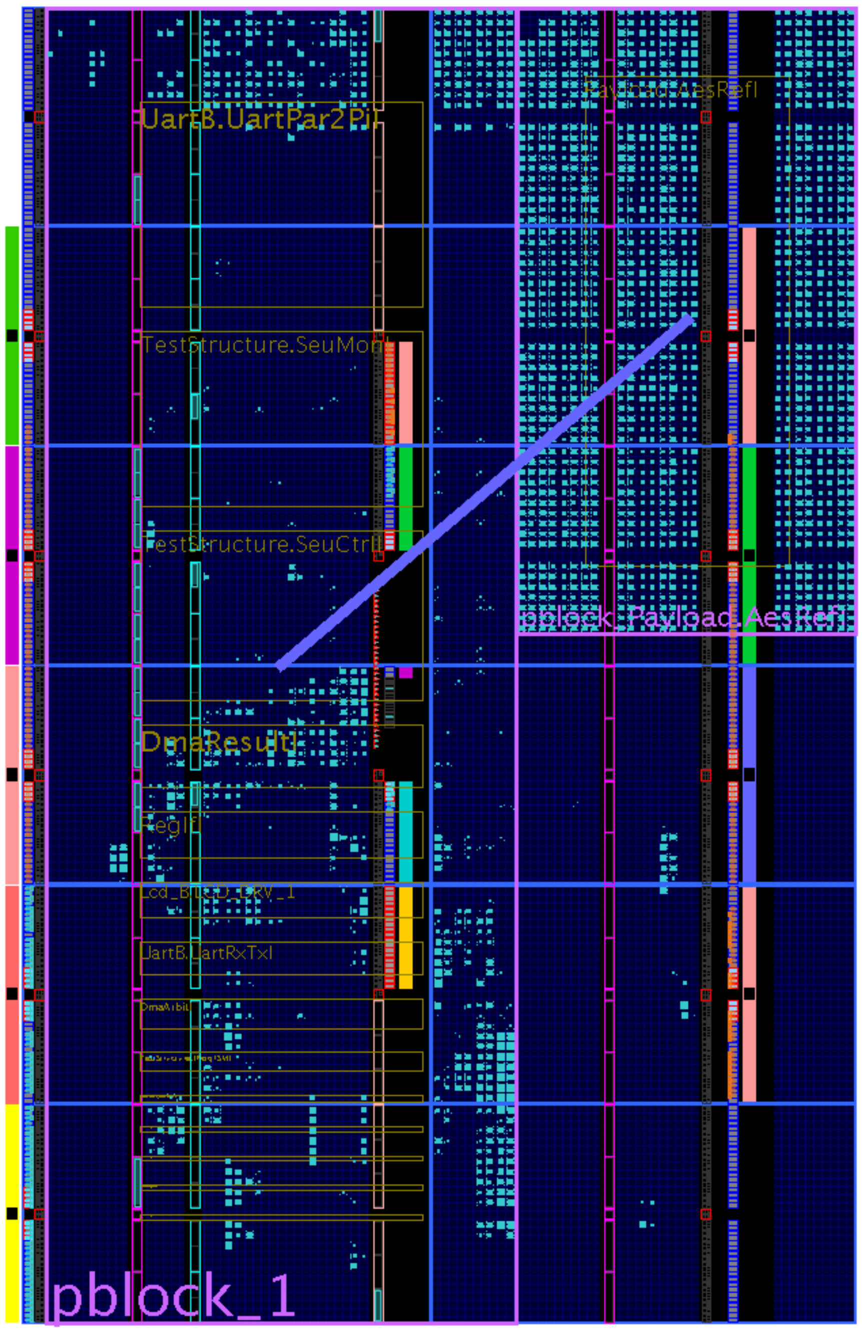


Fig. 46. FPGA Partitioning

A frame number to frame address mapping can be obtained by sending queries to the SEU Controller using the “Q” command and iterating over the whole range of frame numbers. This has to be done once only, since the relationship between frame number and frame address is fixed for a given FPGA.

Other scripts that are used include a Python script that can introduce errors into the binary .bit file, creating an incorrect bit stream. For this purpose, the method for obtaining frame addresses from frame numbers, as described above, is used. This can be used to test the impact of upsets in specific configuration bits on the DUT.

3.3.2 Timing Constraints

For a design to work after the place-and-route step it is important that essential timing constraints are met. Often the designer is responsible for identifying and setting these timing constraints. Some of the constraints are not set by default and will need to be defined manually. Unfulfilled timing constraints, however, might become apparent first once the margins in the PAR step become smaller.

An example of a scenario where the place-and-route-step margins become small is when a design is partitioned into p-blocks. In the specific case described in this report, the DUT is compressed as much as possible in a p-block. During this process, many of the timing problems emerge. The first step of the solution is to identify the parts of the DUT that stop working. The second step is to introduce sufficient timing constraints for identified instances. For this specific test application, timing constraints were introduced for all signals involved with the ICAP interface. Furthermore signals connected to the UART interface were constrained as well. The constraints were set using the timing constraint tool available in PlanAhead. The timing constraint used was *maximum path delay*. Each timing constraint was gradually decreased until the implementation was working as intended. A set of default timing constraints, used in each consecutive implementation, was compiled in this manner.

3.3.3 Fault Injection

Being able to inject faults in an efficient, controllable and consistent manner is a key factor in obtaining good test results and performance estimates for SEU mitigation techniques. By having good random number generation and an accurate SEU distribution, the tests can also be representative for real SEUs.

Fault injection is done through the SEU Controller. The controller is able to insert single or multi-bit faults at random locations in the configuration memory using the commands “1” and “2”, as described in Table 7. Using the “T” command, it is also possible to specify exactly which frame address and bit number within that frame to upset. Using the sensitive frame mapping described in Sec. 3.3.1, the Tcl application on the PC can request single bit upsets in those specific frames corresponding to the DUT. The exact bit number within the selected frame to be upset is generated by the Tcl application. Fig. 47 shows a graphical illustration of 1000 random fault insertions (coloured bars) into a DUT comprising 64 configuration frames (each frame consists of 1312 bits). The colour scale is for visualisation purposes only, and bears no further meaning. The figure shows that the fault insertion is acceptably random and evenly distributed over the range of frames and bits in the DUT.

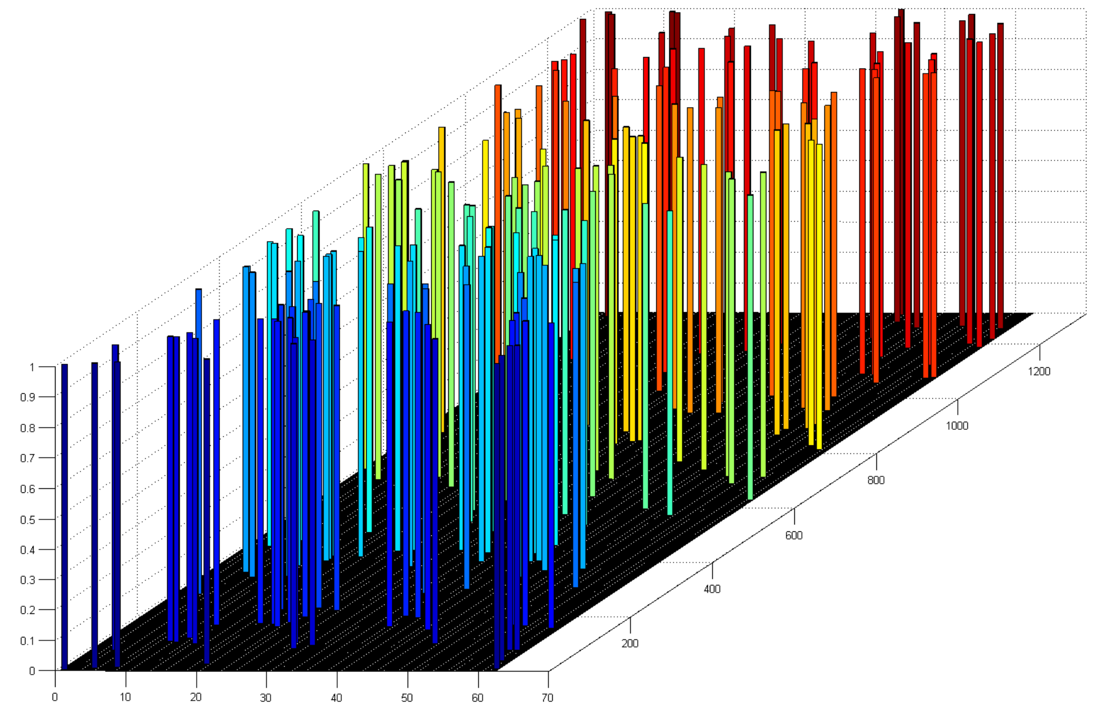


Fig. 47. Distribution of inserted faults where the x-axis represents frame and the y axis represents the bit.

To simulate randomly incoming SEUs, Matlab is used to generate sequences of SEU injections. The time between two consecutive SEUs in the sequence follows a Gaussian distribution where the mean, *μ,* is decided by the expected SEU rate. The time between two fault injections in the generated sequence is scaled with a time factor, depending on the SEU rate, later on. This allows the scaling of results according to SEU rate. Generating fault injection sequences in this manner ensures both repeatability and consistency with current SEU rate parameters.

Fig. 48 shows the fault-insertion mechanism. All faults are injected through the SEU Controller macro using ICAP. The separation between DUT and test structure is made by partitioning using p-blocks in PlanAhead as mentioned in the previous section.

Several other fault-injection approaches have been suggested in the literature: In [60], the authors suggest a *non-intrusive* fault-injection approach. The fault-injection method described in this report shares some similarities with the fault-injection proposed in [60], with some key differences. In this report, faults are injected by sending commands to the SEU Controller, which uses ICAP, whereas the system described in [60] makes use of JTAG.

In the test platform described in this report, the test framework and fault-injection mechanism are synthesised, placed and routed together with the DUT. This makes fault injection through the SEU Controller an *intrusive* fault injection method, as its presence in the system affects the way the DUT is implemented on the FPGA. Using the partitioning of the FPGA as described in the previous section, however, an assessment has been made that the intrusiveness of the fault-injection mechanism should not affect results in a significant way.

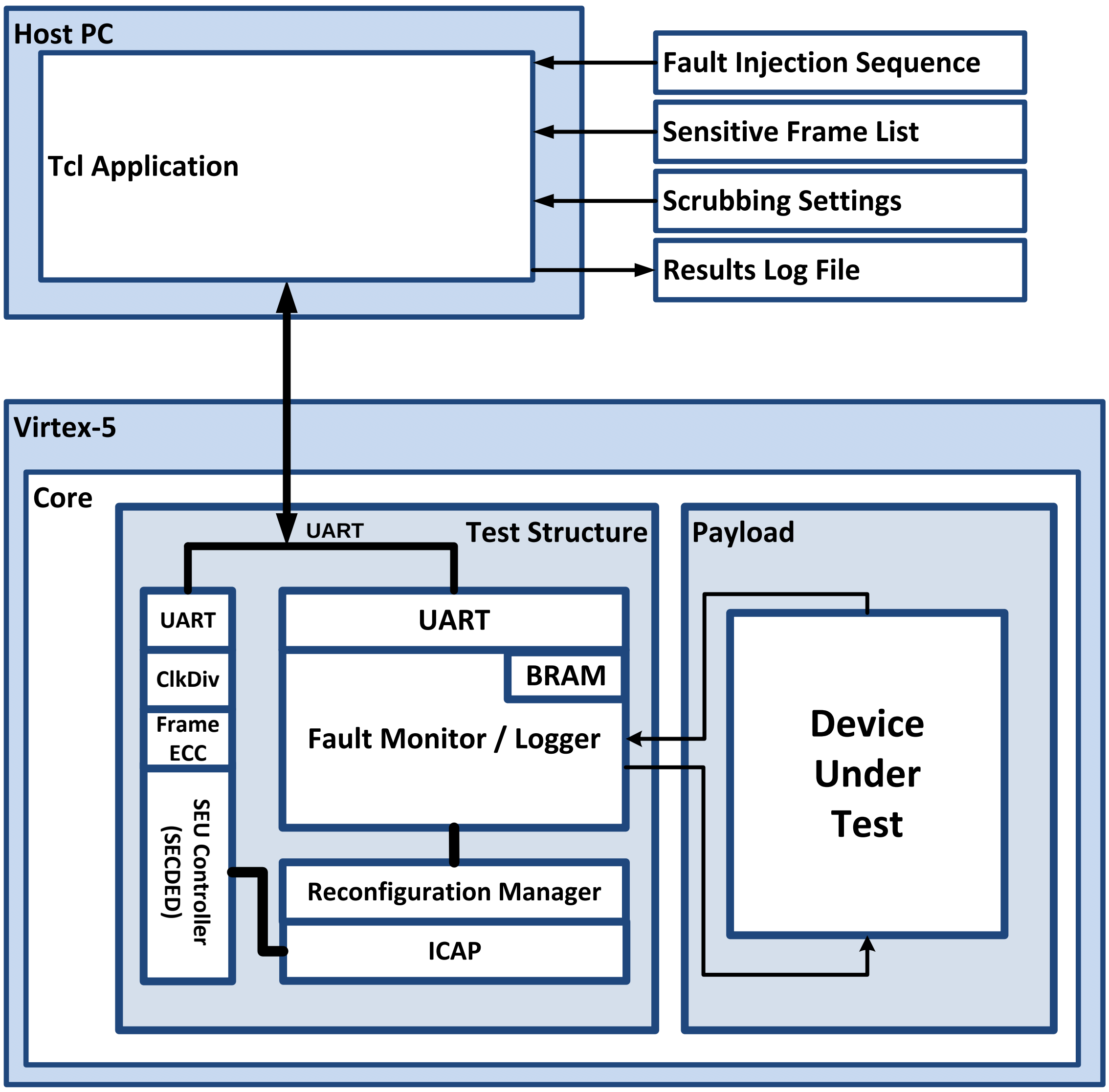


Fig. 48. Fault-Insertion Mechanism

In [61], the authors list a few criteria for fault-injection systems used to evaluate fault tolerant techniques. The authors go on to propose an external SEU generator approach, which similarly to [60] makes use of JTAG to inject errors.

The performance bottleneck in the fault-injection system proposed in this report is the RS-232 communication between the test framework and the host application on the PC.

3.3.4 Static Tests

In this report, a separation is made between static and dynamic testing. This section describes the static testing method. In static testing, single bit upsets are tested in isolation to see their impact on the function of the circuit. A fault is randomly injected in the DUT. After the fault injection, the DUT is tested by running a large number of test vectors through the DUT, with the SEU Monitor checking the result for each test vector. The reason for using a large number of test vectors is the possibility that certain test vectors mask certain faults in the DUT. By using a set of test vectors that together offer a good coverage, the risk of a fault going undetected is minimised.

An error is considered to have occurred when one or more of the test vectors cause the DUT to produce an incorrect result. It should again be noted that even when inserting faults (bit-flips) only in the configuration frames corresponding to the DUT, not all inserted fault will be noticeable as errors on the output (not all bits are sensitive, as discussed earlier). After running the test vectors, any errors are logged and stored to memory by the monitor. The fault is then corrected by the controller, regardless of whether or not the fault caused an observable error. Each one of these steps is run by sending a command from the Tcl application on the PC. These steps together are considered as one test run for a single fault injection. The log line stored for each such test run gives an indication of whether or not the injected fault had an effect on the DUT. A test set, or test campaign, is made up of a number of test runs. In this report, 4,000 fault injections make up a test campaign. This number was chosen to provide a reasonable statistical confidence. By analysing the logs from an entire test campaign, statistics are obtained for the static testing of the specific DUT being tested. As mentioned earlier, this approach tests the effects of single fault injections in isolation, giving a sense of how a single bit-flip can affect the DUT.

Static testing is useful for evaluating the masking capabilities of a mitigation technique, for example TMR. The test process is managed by the Tcl application running on the PC. A fault injection campaign comprising X test runs can be visualised as in the flowchart in Fig. 49, seen from the PC's point of view.

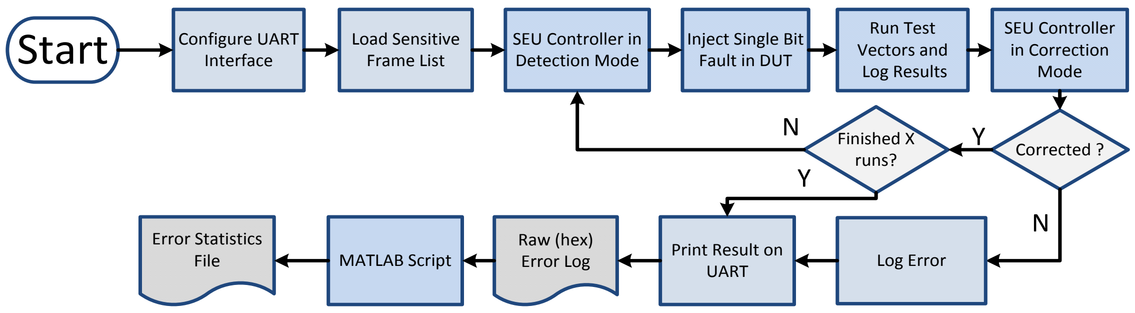

Start
Configure UART Interface
Load Sensitive Frame List
SEU Controller in Detection Mode
Inject Single Bit Fault in DUT
Run Test Vectors and Log Results
SEU Controller in Correction Mode
Corrected ?
Y
N
Finished X runs?
N
Y
Log Error
Print Result on UART
Raw (hex) Error Log
MATLAB Script
Error Statistics File

Fig. 49. Static Testing Flow

3.3.5 Dynamic Tests

Dynamic tests differ from static tests in that errors are not corrected between each fault insertion. This allows the analysis of dynamic processes, such as error build-up, and the effect of scrubbing. With this type of test, the system is running normally rather than being paused in between each isolated test as it is in static testing. In dynamic testing, the DUT is continuously run and fed with data. The Tcl application on the PC controls all commands sent to the FPGA for error logging, fault insertion, reconfiguration and UART control. By keeping the test sequence completely in software on the PC, tests can be easily changed, and more advanced logging capability is possible.

By using the fault injection technique described in Sec. 3.3.3, the Tcl Application can simulate a specific SEU rate with the appropriate level of randomness. A dynamic test could for example be as simple as injecting faults according to such a distribution into a TMR enabled DUT, in order to see how many errors, on average, that particular TMR implementation can handle before being broken by error build-up. Dynamic testing is useful for seeing how a system or technique performs over time, or for testing scrubbing techniques. From dynamic testing, numbers for Availability, MTTF, MTTR and performance over time can be obtained. Therefore, dynamic testing is more of a simulation of a real scenario, rather than simple isolated, single error testing as in static tests. With dynamic testing, different scrubber implementations and scrubbing rates can be tested for effectiveness, as well as testing the robustness of different TMR implementations. The dynamic testing procedure is based on events in the Tcl Application. Each event can be seen as a slot, where a single event can take place. This event can be a fault insertion, a reconfiguration or an idle slot.

All event slots are considered to take the same amount of time, referred to here as a '*tick'*. When running the actual tests, different types of commands sent to the FPGA from the Tcl Application will take different amounts of time, mainly due to the RS232 communication. As the RS232 is the bottleneck in the test system with respect to delay, it is desirable to design a test approach that eliminates or masks the communications related real-time delays. In a real FPGA system operating in space, an SEU can be seen as an instantaneous event.

Fig. 50 illustrates a simple dynamic test simulating a periodical scrubbing approach combined with random SEU insertion, given a certain average SEU rate. The control sequence, managed by the Tcl Application, uses the (Matlab-generated) fault-injection sequence to determine at which ticks to inject faults. The DUT frame mapping file contains information on where in the FPGA to inject errors. Scrubbing, or reconfiguration, commands are sent periodically according to the programmed scrubbing rate. The R1-R3 slots in the figure represent reconfiguration events, and F1-F3 represent fault injections. Empty slots are idle. Each fault injection event consists of an error injection, running all the test vectors for that error, logging and printing the result, then giving control of the UART back to the SEU Controller. The triangle shapes in the figure represent testing of the DUT and printing back results to the PC. In this particular example, the DUT was able to mask F1, but produced an error (purple triangle in the log line) after F2 due to error build-up. In this example, TTR would be 2 ticks (half the time between two scheduled scrubbing events, measured in discrete time points).

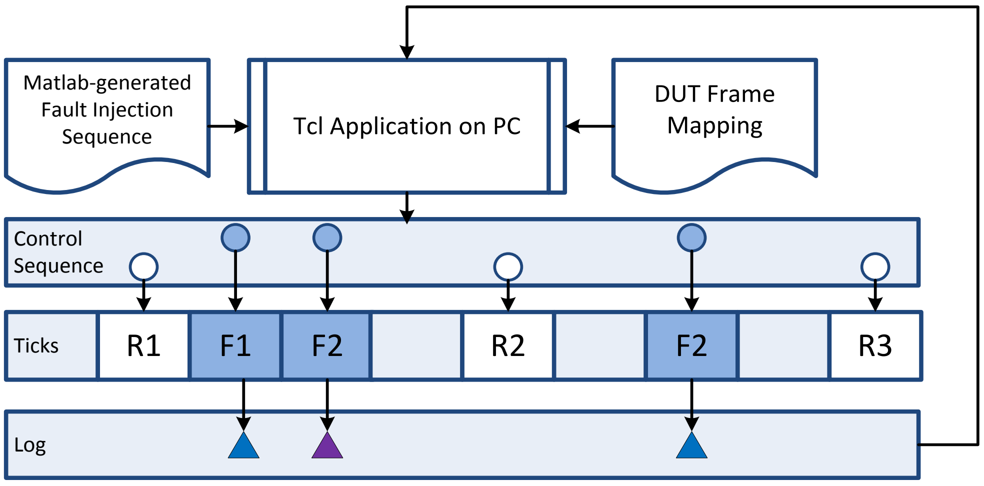


Fig. 50. Basic Dynamic Testing

By using this method, tests are easily scalable to whichever SEU and Scrubbing rates need to be tested. The fault injection sequence generated by Matlab only needs to provide a good-enough resolution, as the actual fault injection time stamps used by the Tcl Application can be scaled. There are two time measurements that cannot be scaled using this method. The first one is the time it takes to fully reconfigure the FPGA (for the Virtex-5 test platform used here, this time is about 500ms). The other one is the time it takes to detect a configuration error, when using a scrubber based on error detection. These times need to be measured and converted to the Tick-timescale. The time of one tick needs to be scaled according to the SEU rate.

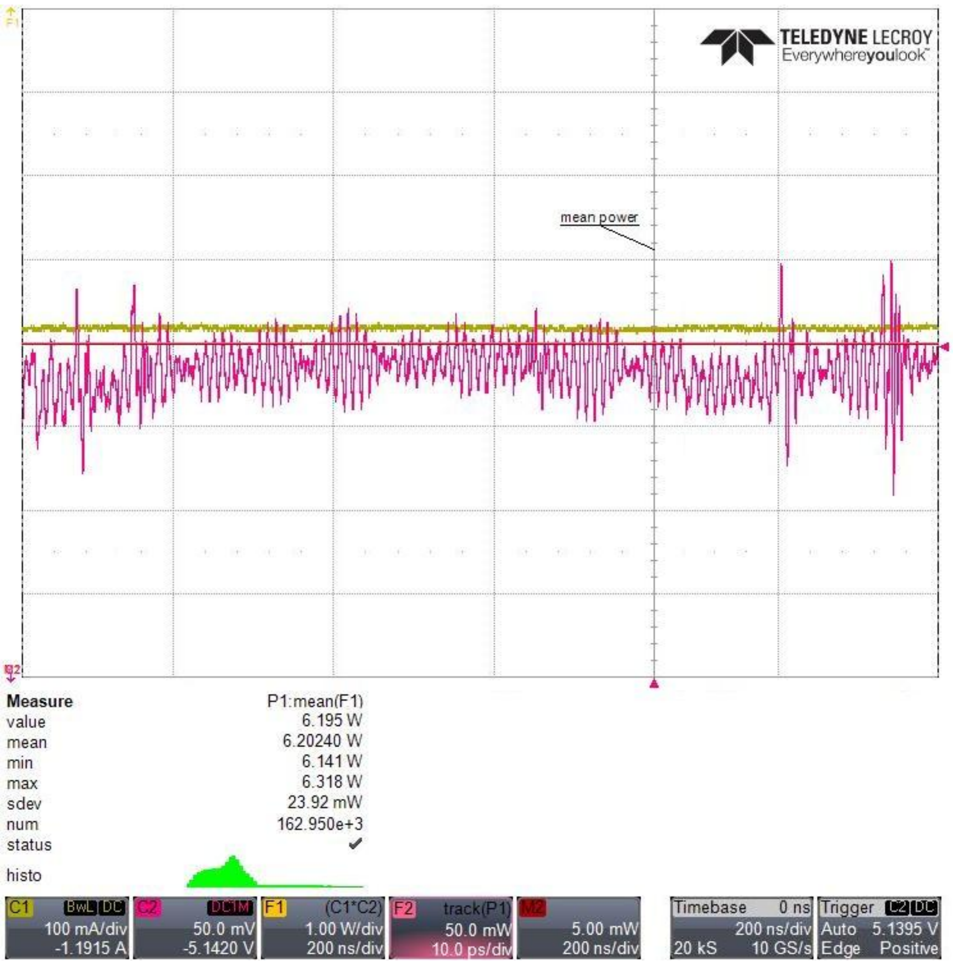
As an example, given a resolution of 100 for the fault injection sequence, and an expected SEU rate of $1\frac{\text{SEU}}{\text{minute}}$, the tick duration would be $T_{tick} = \frac{1}{100 * SEU\ Rate} = \frac{SEU\ Period}{100} = \frac{3}{5}\text{seconds.}$

Using this tick time, a reconfiguration event would cause the circuit to be unavailable for 5 ticks. This type of measurement is used later when calculating availability, which is the primary measurement for comparing SEU mitigation techniques.

3.3.6 Measuring Power

As the evaluation board used lacks current sense resistors and circuitry for measuring power consumption, measurements have to be made at the board master power supply. An alternative would be to physically modify the board by replacing the DC-DC regulators for *VccInt*, *VccAux* and *VccO* with regulators equipped with power measurement circuitry. However, besides from being difficult, this not be done without damaging the board and so will not be considered here.

Measuring power consumption at the external power supply is not an entirely accurate method, as measurements are made on the whole board and not only the FPGA itself. This method was indeed applied in this work, but the differences in power consumption between mitigation technique implementations proved to be too small to measure. This is mainly because the power dissipated by the board as a whole is large compared to the power drawn by the FPGA, and losses in voltage regulators are likely to dominate. During power measurements, variations were recorded depending on a number of factors such as temperature. Because of this, power measurements were abandoned. Fig. 51 shows a screenshot from the oscilloscope used for the attempted power measurements.



```
Measure     P1:mean(F1)
value           6.195 W
mean          6.20240 W
min             6.141 W
max             6.318 W
sdev          23.92 mW
num         162.950e+3
status              ✓
histo
```

```
C1  BwL DC    | C2  DC1M   | F1   (C1*C2) | F2  track(P1) | M2         | Timebase    0 ns | Trigger  C2 DC
100 mA/div    | 50.0 mV    | 1.00 W/div   | 50.0 mW       | 5.00 mW    | 200 ns/div       | Auto  5.1395 V
-1.1915 A     | -5.1420 V  | 200 ns/div   | 10.0 ps/div   | 200 ns/div | 20 kS   10 GS/s  | Edge  Positive
```

Fig. 51. Power Measurements

3.3.7 Measuring Availability

Availability in a context of fault tolerance represents how often the system is functional and delivering its service. The availability can therefore be calculated as described in Eq. 16. In our case, however, there will be another contribution to the down time of the system apart from the system producing the wrong result. Whenever the system is being scrubbed it is also unavailable and therefore unable to fulfil its purpose. Availability, as used hereafter in this report, is therefore calculated according to Eq. 19.

$$Availability = \frac{ticks_{all} - (ticks_{incorrect} + ticks_{scrubbing})}{ticks_{all}} = \frac{ticks_{correct}}{ticks_{all}} \qquad (19)$$

From Eq.19 it is apparent that availability represents the ratio of correct ticks to all ticks. Correct ticks comprise ticks where the right end result is produced by the implementation and no scrubbing is in progress. Number of correct ticks is calculated by deducting the ticks where the DUT produces the wrong result and the number of ticks where the DUT is occupied by scrubbing, as shown in Eq.19.

3.3.8 Measuring Area

Area measurements are read from the Map Report File generated by PlanAhead. The *-details* command is supplied to the MAP stage in PlanAhead, generating a detailed resource usage report in Section 13 of the *.mrp* file, as documented in [62].

Table 9 presents an example of an excerpt from an .mrp file generated during mapping of a triple voter TMR implementation. The instances Aes0-2 represent the redundant AES modules, and Voter0-2 the tripled voter stages. Using this method it is possible to compare the areas of different payloads, without the underlying test framework.

Table 9. Map Report File Excerpt

Module	Slices	Slice Reg	LUTs	LUTRAM
. . .				
+Payload.AesTmr	3/1298	4/1603	3/4224	0/0
++AesI_B.Aes0	349/349	404/404	1143/1143	0/0
++AesI_B.Aes1	367/367	404/404	1143/1143	0/0
++AesI_B.Aes2	366/366	404/404	1141/1143	0/0
++Voter_B.Voter0	71/71	129/129	264/264	0/0
++Voter_B.Voter1	71/71	129/129	264/264	0/0
++Voter_B.Voter2	71/71	129/129	264/264	0/0
+TestStructure.BaudDivider	12/12	10/10	24/24	0/0
+TestStructure.ReconfMgr	15/15	16/16	31/31	0/0
+TestStructure.SeuCtrl	77/183	194/314	96/328	0/86
+TestStructure.SeuMon	323/380	89/159	1049/1174	0/0
+TestStructure.UartArb	12/12	0/0	24/24	0/0

While the AES blocks are all equal in size, there may be a slight difference in the number of slices used. This will vary with the PAR approach used by PlanAhead. Looking at the number of Slice Registers first, and the number of LUTs second, gives a good approximation of how big the module is. Other resources such as DSP blocks, DCMs and BRAMs are not shown here. Note that in the example shown in Table 9, the Reconfiguration Manager (*TestStructure.ReconfMgr*) is not a part of the Payload. This means that the scrubber is not included in area measurements, and no faults are inserted in the scrubber. Keeping the scrubber outside of the payload in this manner represents having an external scrubber implementation.

3.3.9 Tool and Language Versions

Table 10 gives a summary of the tools and languages used in this project, as well as the version used. Note that editors, supporting applications and individual scripts are not listed here.

Table 10. Tools and Languages

Tool / Language	Version	Vendor	Used For
Synplify Pro / Premier	H-2013.03	Synopsys	Synthesis
PlanAhead	14.4	Xilinx	Floor-planning, PAR
Tcl	8.6		In-tool scripting, test manager
Python	3.3.2		General scripting, text parsing
Perl	5.18.2		General scripting, text parsing
Matlab	R2012a	Mathworks	Data analysis, statistics
iMPACT	14.5	Xilinx	FPGA / PROM Programming
VHDL	-93		HDL
OMERE	3.6.3.0	TRad	Orbit and SEU calculations
SPENVIS	4.6.7		SEU calculations (CREME'96)
ModelSim	10.2a	Mentor	Simulations

3.3.10 Test Toolchain and Plan

Fig. 52 shows the complete test toolchain from VHDL code to test results file. The input to the toolchain is a VHDL model of the DUT and an estimated SEU rate. As output from the entire chain, complete test run statistics are generated.

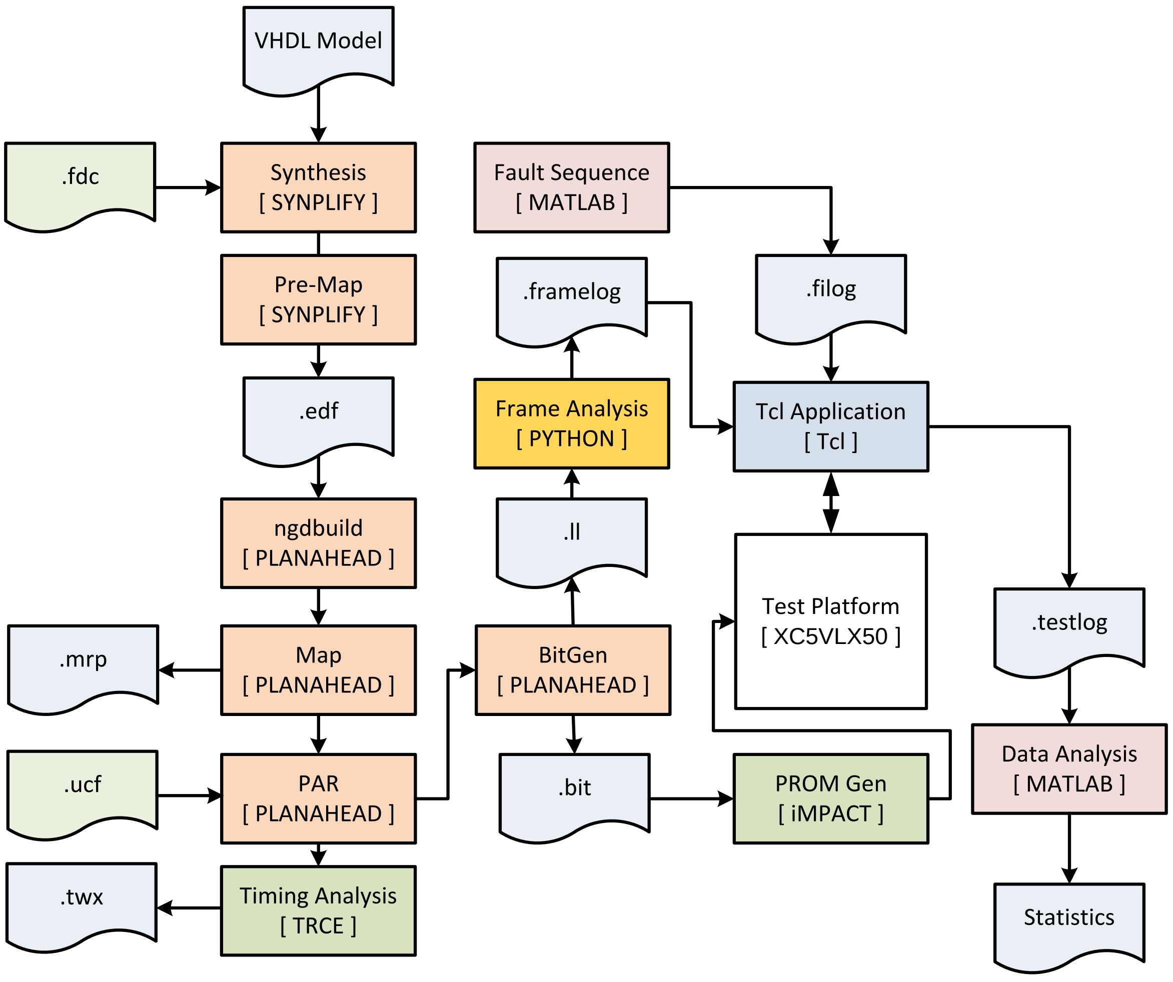


Fig. 52. Complete Test Toolchain

The Tcl Application will run different test sequences and log different log data on the FPGA depending on the type of test and the type of DUT used. As mentioned earlier, a test campaign in this report consists of 4,000 fault injections. These test sequences consist of a series of commands sent via RS232 to the SEU Controller and SEU Monitor. Fig. 53 and Fig. 54 contain pseudo-code representations of the test sequences for blind scrubbing implementations and static testing, respectively.

The sensitive frame list used is generated by the Python script as seen in Fig. 52 (.*framelog* file). The fault insertion log (.*filog* file) is generated by Matlab as described earlier. When the tick count matches a fault insertion point as stated in the fault insertion list, a fault is inserted. In the blind scrubbing test sequence, reconfigurations are requested by sending the command “X” periodically with the scrubbing rate that is to be tested. Using a μ-value of 100 ticks for the fault insertion, a scrubbing rate of 5 would, for example, correspond to sending a reconfiguration command every 20 ticks. Output consists of a .*testlog* file. The Xilinx tool TRCE is run for timing analysis [63].

```
SFL  := Read Sensitive Frame List
RFIL := Read Fault Insertion List

while i<4000 loop
     if (nextReconfiguration) then
              Command "X"                               # Reconfigure
              Command "*"
              Command "D"
     elseif (nextFaultInjection) then
              FrameAddr  := $SFL(random)
              BitNumber  := random(0,1312)
              Command "T"                               # Inject Error in...
              Send $FrameAddr                           # Frame Address
              Send $BitNumber                           # Bit Number
              Command "Z"                               # Run Test
              Command "Y"                               # Print Test Result
              Log                                       # Print to Log File
              Command "K"                               # Return UART
     end if
     Tick $delay                                        # Tick forward
end loop
```

Fig. 53. Blind Scrubbing Test Sequence

```
SFL  := Read Sensitive Frame List

while i<4000 loop
     Command "D"                                        # Detection Mode
     FrameAddr  := $SFL(random)
     BitNumber  := random(0,1312)
     Command "T"                                        # Inject Error in...
     Send $FrameAddr                                    # Frame Address
     Send $BitNumber                                    # Bit Number
     Command "Z"                                        # Run Test
     Command "Y"                                        # Print Test Result
     Log                                                # Print to Log File
     Command "A"                                        # Correction Mode
     Await Correction                                   # Wait for Correction
     Tick $delay                                        # Tick forward
end loop
```

Fig. 54. Static Testing Sequence

Using the test tools and mitigation techniques available, a test plan has been created in order to catch as many relevant trends as possible. In particular, the test plan is designed to investigate scrubbing rate versus availability.
Fig. 55 shows a summary of the dynamic tests (scrubbing tests).

Scrubbing Technique	TMR Implementation	Scrubber Location
Blind Scrubbing	Reference	Internal
		External
	Synplify TMR	Internal
		External
	Single Voter	Internal
		External
	Triple Voter	Internal
		External
Scrubbing on CRC Error	Reference	Internal
		External
	Synplify TMR	Internal
		External
	Single Voter	Internal
		External
	Triple Voter	Internal
		External
Scrubbing on Frame ECC Error	Reference	Internal
		External
	Synplify TMR	Internal
		External
	Single Voter	Internal
		External
	Triple Voter	Internal
		External
SECDED + scrubbing on MBE	Reference	Internal
		External
	Synplify TMR	Internal
		External
	Single Voter	Internal
		External
	Triple Voter	Internal
		External

Fig. 55. Scrubbing Test Plan

For blind scrubbing, different scrubbing rates are tested. The scrubbing rate is always stated in relation to the expected SEU rate. In the test plan presented here, the rates tested are: 0.01, 0.1, 0.5, 1, 2, 5, and 10 times the SEU rate. This means that for the lowest scrubbing rate, an average of 100 faults are injected between scrubbing events. For the fastest scrubbing rate, the FPGA is scrubbed on average 10 times between each fault injection.

Detection-based scrubbing techniques, on the other hand, perform scrubbing only upon detecting an error. This means that a ‘*scrubbing rate*’ has no meaning for detection-based scrubbers, and therefore only one test is conduced per combination of detection method, TMR implementation and scrubber placement.

For the static testing, only the different TMR implementations are tested (it would not make sense to test a scrubbing implementation with the static testing method as described in this report). Fig. 56 shows an overview of the static tests conducted.

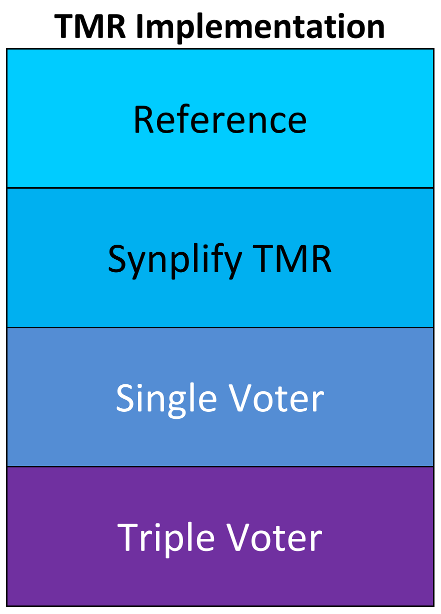


Fig. 56. Static Testing Plan

The purpose of the dynamic (scrubbing) tests is to provide information on MTTF, MTTR and Availability, as well as to simulate a “real” scenario to see how TMR implementations hold up to errors over time. Static tests are performed with the purpose of testing the masking capabilities of different TMR implementations only.

3.3.11 Test Data Analysis

As mentioned in Sec. 3.1.2.2.2, the SEU monitor is used to test and monitor the DUT to make sure that it is working properly. Apart from knowing that a failure has occurred, it is also interesting to see in what part of the DUT failed. For this purpose multiple signals, as shown in Table 5, are logged by the SEU monitor. For each combination of logged status bits it is possible to categorise a fault. Since this task can be done afterwards, it is done in a post-processing script.

The constructed MATLAB script starts by importing the log for the constructed Tcl log. It then proceeds to identify the status bits by searching for the request command. Depending on what implementation mode is used (Reference, Single Voter, Triple Voter or Synplify TMR) the script interprets the status bits differently. Depending on the combination of status bits, the script calculates if the result is correct, what category the error belongs to and if the scrub can be done in parallel.

The available result categories are: *Correct result*, *Voter error*, *Single TMR Error*, *Error in 2 TMR units (Bridge Effect)* and *Multiple Errors*. A Voter error is any combination of errors where each AES unit outputs a correct value while one or more voters output an incorrect value. A Single TMR Error is an error where one AES unit outputs an incorrect value while the remaining DUT functions as intended. A Bridge Effect error is an occurrence where two of the AES outputs are incorrect while the rest of the DUT is working correctly. Finally, Multiple Errors indicate that at least one AES unit and one voter are erroneous. Once all errors have been categorised, statistics are concluded and metrics are calculated.

The MATLAB script also calculates the scrubbing metrics such as the availability, MTTF and MTTR. For blind scrubbing, all the instances for scrubbing and fault insertion can be calculated from the fault injection file and based on the specified scrubbing rate. For the blind scrubbing mode the script therefore only extracts the status bits for each fault insertion.

For CRC triggered scrubbing, frame ECC triggered scrubbing and SECDED the MATLAB script also extracts the detection and scrubbing durations. In this case availability, MTTR and MTTF is based on the extracted detection duration and the scrubbing duration. These are calculated by the PC application and saved in the log. All compiled metrics for each run are saved in summary reports.

4 Results

In this section, the results from the static and dynamic testing campaigns are presented. Sec. 5 provides a discussion of the results.

4.1 Evaluating TMR Implementations

The static tests are performed as described in Sec. 3.3.4, and are used to give an indication of how well the different TMR implementations can mask errors. Table 11 gives a summary of the static testing results.

Table 11. Static Testing Results

TMR	Injected Errors	Observable Errors	Failures	Failures(%)
Reference	4000	1084	1084	27.1
Single Voter	4000	697	34	0.85
Triple Voter	4000	714	14	0.35
Synplify TMR	4000	10	10	0.25

In the reference AES design, all observable errors will also be failures, as they lead to an incorrect output. This gives a failure rate of 27.1%, meaning that 27.1% of SEUs affecting the configuration memory will result in an incorrect output. Observable errors in this report are defined as errors that produce an incorrect output from an AES block or a voter stage

Table 12 shows a breakdown of the observed errors in each of the tested TMR implementations.

Table 12. Observable Errors, Static Testing

TMR	Single Errors	Bridge Errors	Voter Errors	Multiple	Total
Reference	1084 (100.0%)	N/A	N/A	N/A	1084
Single Voter	663 (95.1%)	10 (1.4%)	20 (2.9%)	4 (0.6%)	697
Triple Voter	674 (94.4%)	6 (0.8%)	32 (4.5%)	2 (0.3%)	714
Synplify TMR	10 (100.0%)	N/A	N/A	N/A	10

The categories bridge errors, voter errors and multiple errors are not applicable to the reference design or to the Synplify TMR design. A single error is defined as an error affecting the output of a single TMR branch. In the reference design, this means an error has occurred in the (one and only) AES encryption block, and therefore an incorrect output is noted.
Looking at the resources used for each of the TMR implementations, Table 13 shows the relative resource usage on an XC5VLX50 FPGA. The designs are synthesised using Synplify Premier (H-2013.03).

Table 13. TMR Implementations, Resources

TMR	Slices	Slice Regs	LUTs	LUT-DFF-Pairs	Relative
Reference	291	408	1143	1143	1
Single Voter	1354	1345	3698	3698	3.24 x
Triple Voter	1472	1603	4224	4224	3.87 x
Synplify TMR	2275	1224	5436	5436	4.76 x

4.2 Evaluating Scrubbing Methods

By running dynamic, continuous tests on TMR systems and employing different scrubbing methods, statistics can be obtained for the efficiency of the scrubbing methods. These methods also have to be put into relation with the area overhead and added complexity they bring to the system.

4.2.1 Availability

In this section, results for availability are depicted as a function of SEU rate, and in the case of blind scrubbing also Scrubbing rate. The scrubbing rate is defined as the frequency of scrubbing divided by the frequency of SEUs. Sweeping the SEU rate in this manner corresponds to comparing the performance of a scrubber at different levels of radiation.

4.2.1.1 Blind Scrubbing

Fig. 57 shows the availability of four different systems, all using blind scrubbing, but with different TMR implementations. In this plot, the mean time between SEUs is fixed at 1 SEU per minute. The scrubbing rate is varied to illustrate how scrubbing more or less often affects the system's availability. The x-axis (scrubbing frequency) is varied from 0.01 to 10. These values correspond to a periodical scrubbing performed 100x slower to 10x faster than the expected SEU rate.

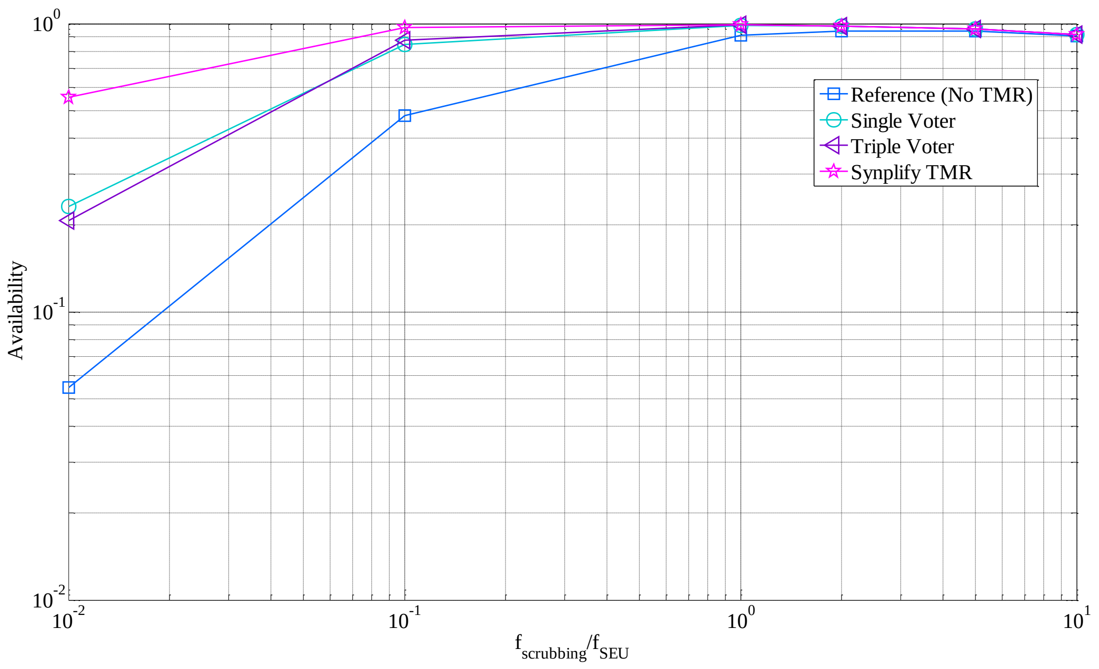


Fig. 57. Availability, Blind Scrubbing (at SEU Rate 1/min)

In
Fig. 58, results for blind scrubbing availability are presented in four different graphs for the different TMR implementations and reference case. In this figure, the scrubbing rate as well as the mean time between SEUs is varied. The black line represents the maximum availability value at each SEU rate. The colour coding represents the level of availability.

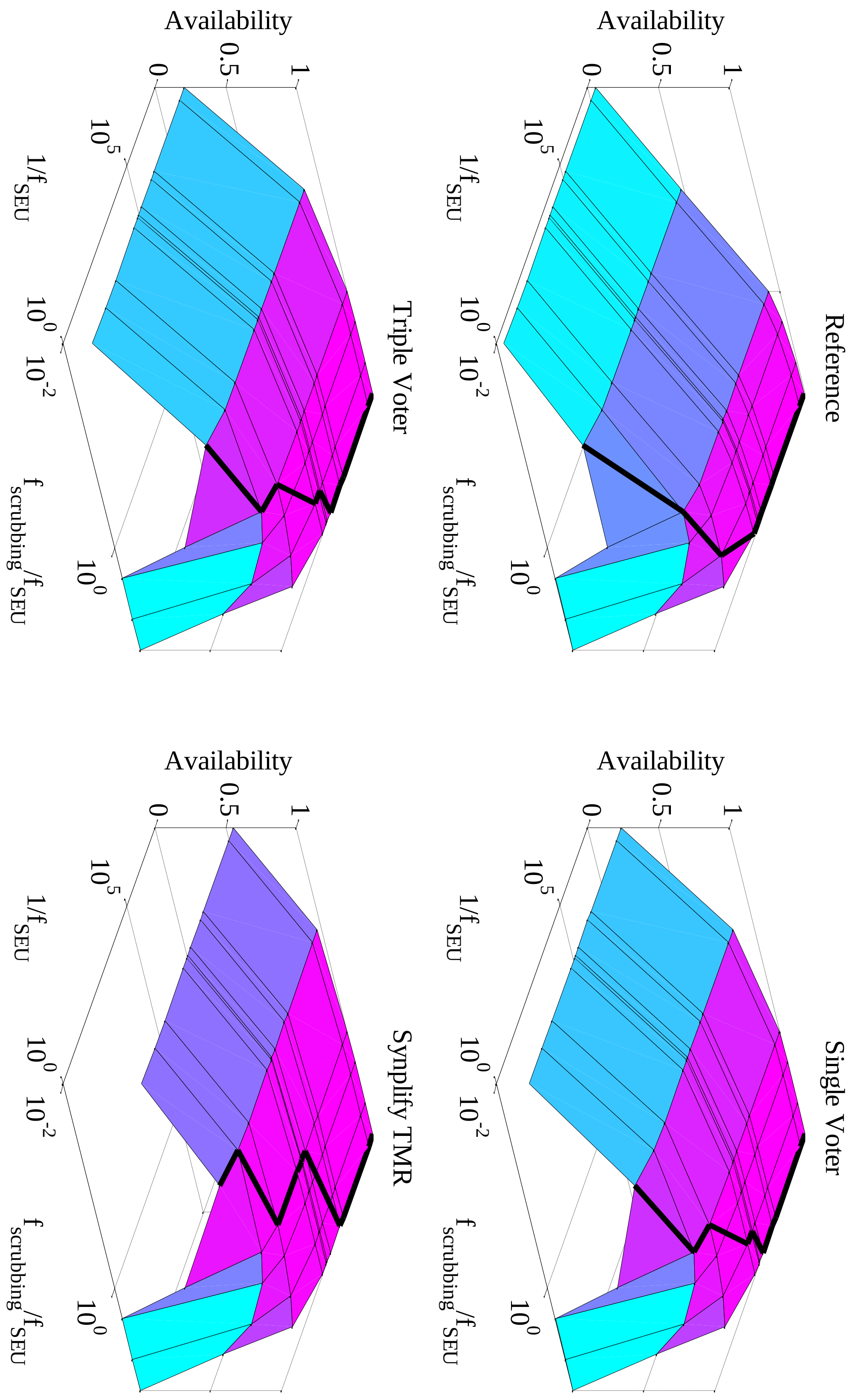


Fig. 58. Availability, Blind Scrubbing

Fig. 59 illustrates this by showing the percentage of downtime (when the system is not available) when the system is busy with reconfiguration. The inverse of the plotted values correspond to the percentage of the total downtime the system is producing an incorrect output.

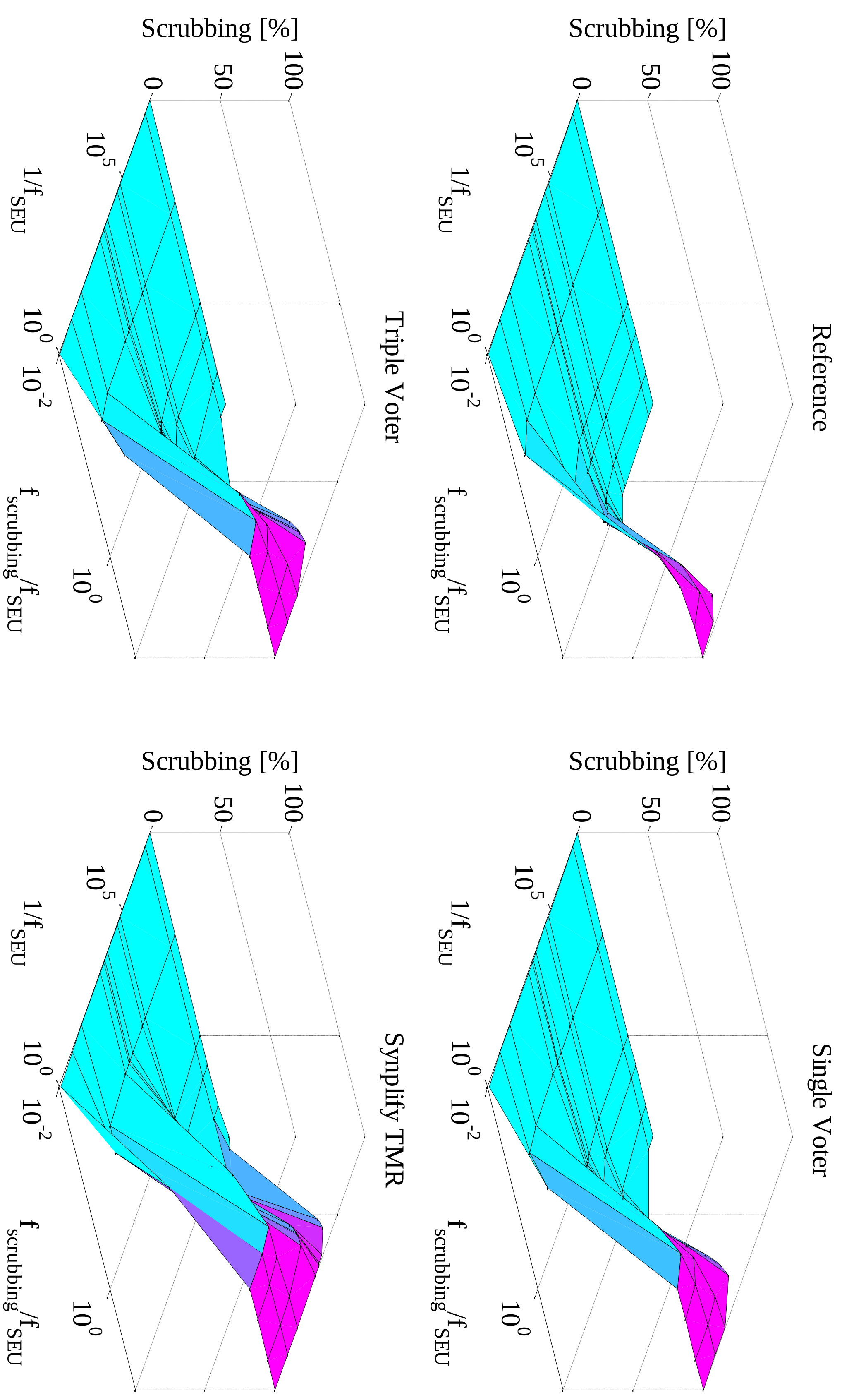


Fig. 59. Cause of Downtime, Blind Scrubbing

Fig. 60 shows the cause of downtime for blind scrubbing, similarly to Fig. 59, but fixed at an SEU rate of 1 SEU per minute.

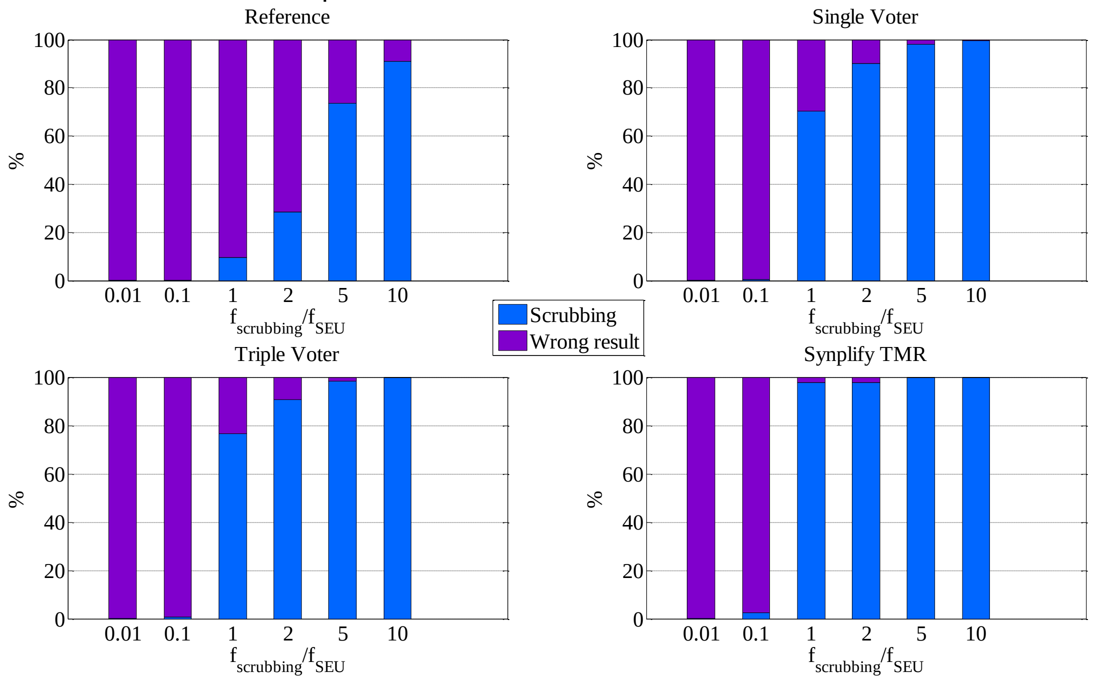


Fig. 60. Cause of Downtime, Blind Scrubbing (at SEU rate 1/min)

Fig. 61 shows a plot of Availability/LUT as a function of scrubbing rate for the different TMR implementations and the reference case.

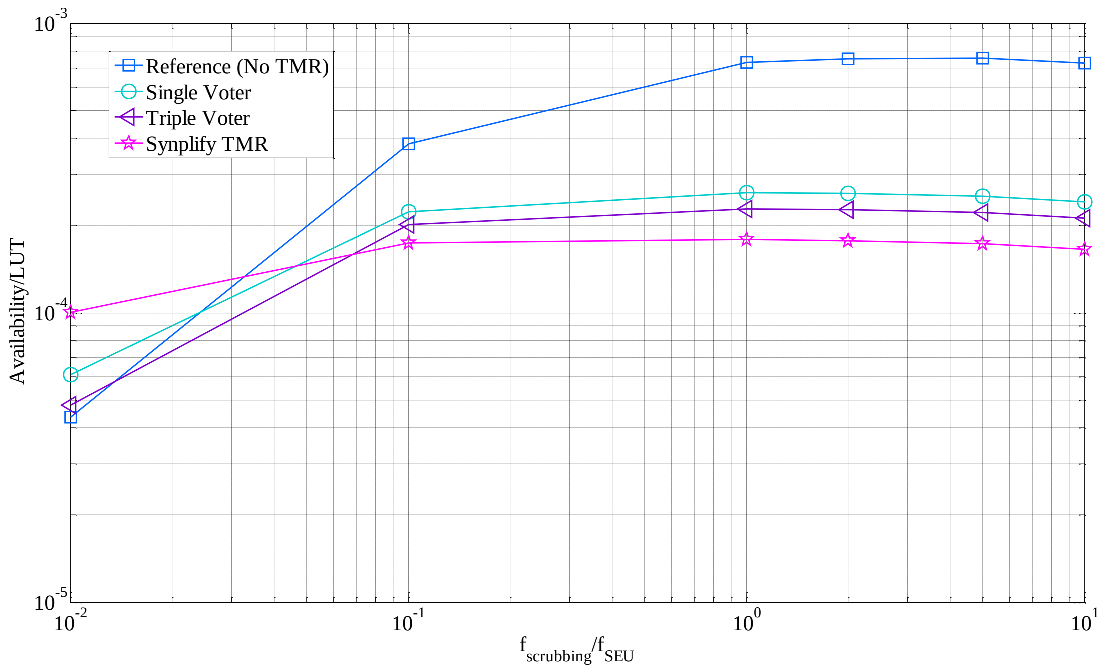


Fig. 61. Availability per LUT, Blind Scrubbing (at SEU Rate 1/min)

Fig. 62 shows Availability/LUT for the three TMR implementations in combination with blind scrubbing, again as a function of the mean time between SEUs and the scrubbing rate. The reference design is not included here.

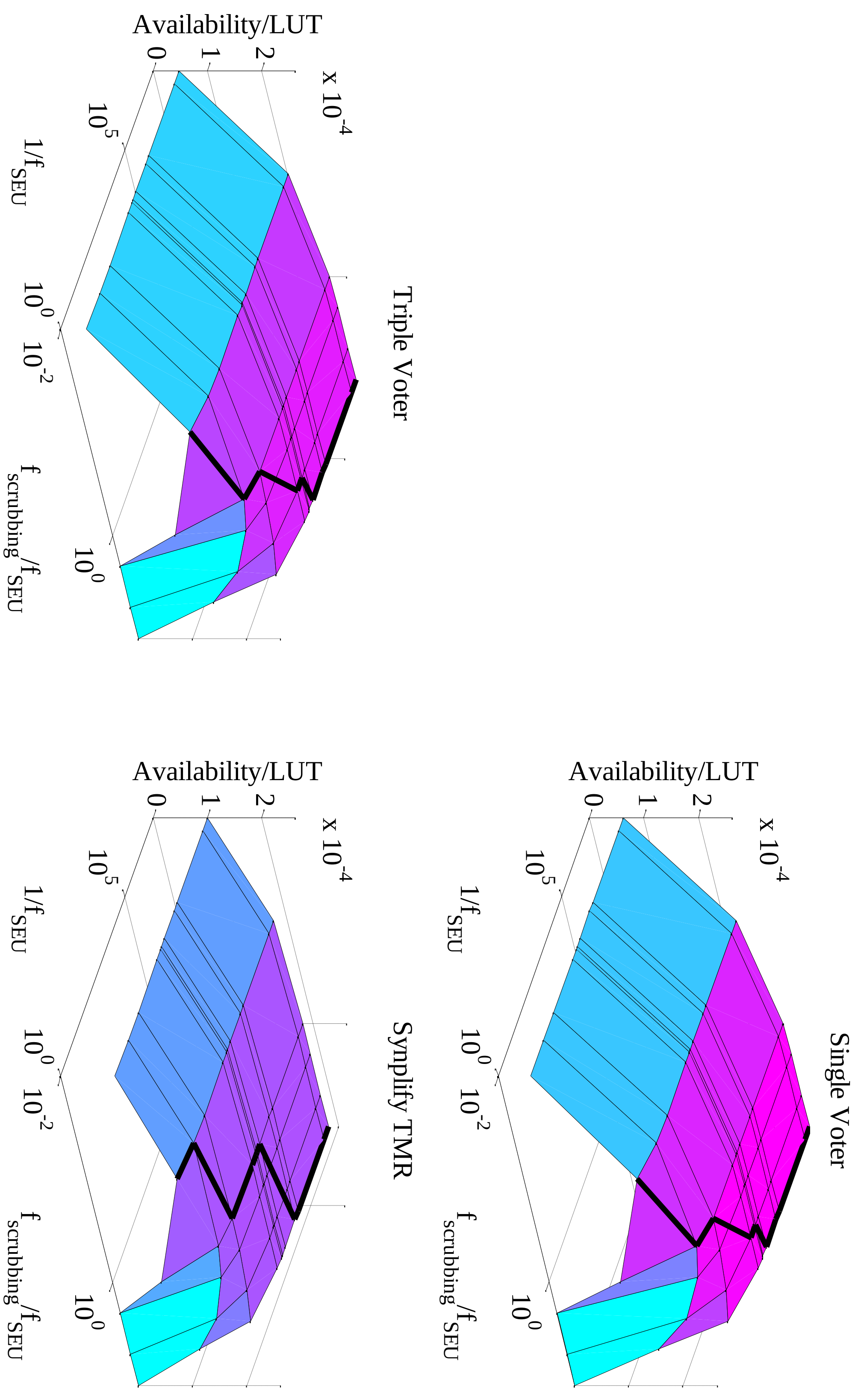


Fig. 62. Availability per LUT, Blind Scrubbing

4.2.1.2 Error Detection-based Scrubbers

For the implemented scrubbers that make use of error detection methods (CRC, ECC, SECDED), the concept of scrubbing rate has no meaning, as scrubbing is done only after detection, and not periodically as for blind scrubbing.. Table 14 shows availability for the three detection-based scrubbing methods when varying the SEU rate from 1 per second to 1 per day.

Table 14. Detection-based Scrubbers

SEU Rate	Detection Mode	Reference	Single Voter	Triple Voter	Synplify TMR
1 / Sec	CRC	0,509659089	0,516125647	0,505838768	0,507322385
	ECC	0,923216257	0,994693569	0,999696190	0,998800744
	SECDED	0,991487829	0,995001109	0,998250712	0,998424918
1 / Min	CRC	0,991827651	0,991935427	0,991763979	0,991788706
	ECC	0,998720270	0,999911559	0,999994936	0,999980012
	SECDED	0,992158090	0,999916685	0,999970845	0,999967535
1 / Hr	CRC	0,999863794	0,999865590	0,999862732	0,999863145
	ECC	0,999978671	0,999998525	0,999999915	0,999999666
	SECDED	0,992169261	0,999998611	0,999999514	0,998671579
1 / Day	CRC	0,999994324	0,999994399	0,999994280	0,999994297
	ECC	0,999999111	0,999999938	0,999999996	0,999999986
	SECDED	0,992169443	0,999999942	0,999999979	0,999999981

4.2.2 Resource Usage

Table 15 presents a comparison of the resource usage for the different implemented scrubbing methods. The numbers presented for the case with no scrubber implementation can be seen as not having a scrubber at all, or keeping the scrubber externally. The numbers for CRC, ECC and SECDED scrubbers represent the case of having the scrubber internally on the FPGA. Relative overhead percentages are presented for each combination of TMR and Scrubbing, with the no-scrubber case as a reference case for each TMR implementation.

Table 15. Scrubber Implementations, Resources

Scrubber	TMR	Slices	Slice Regs	LUTs	Relative
None	Reference	291	408	1143	100%
	Single Voter	1354	1345	3698	100%
	Triple Voter	1472	1603	4224	100%
	Synplify TMR	2275	1224	5436	100%
CRC	Reference	315	424	1179	104%
	Single Voter	1373	1361	3634	101%
	Triple Voter	1491	1619	4241	101%
	Synplify TMR	2294	1240	5472	101%
ECC	Reference	746	1042	2051	255%
	Single Voter	1809	1979	4606	147%
	Triple Voter	1843	2237	5125	140%
	Synplify TMR	2730	1858	6344	152%
SECDED	Reference	505	734	1526	180%
	Single Voter	1596	1671	3607	124%
	Triple Voter	1687	1933	4598	120%
	Synplify TMR	2490	1550	5819	127%

Fig. 63 shows a graphical representation of the relative resource usage for each of the implemented scrubbing methods, normalised to reference (the resource usage of reference is 1.0). In Fig. 63, the TMR implementations' names are shortened as *SV* (Single Voter), *TV* (Triple Voter) and *Synp* (Synplify TMR).

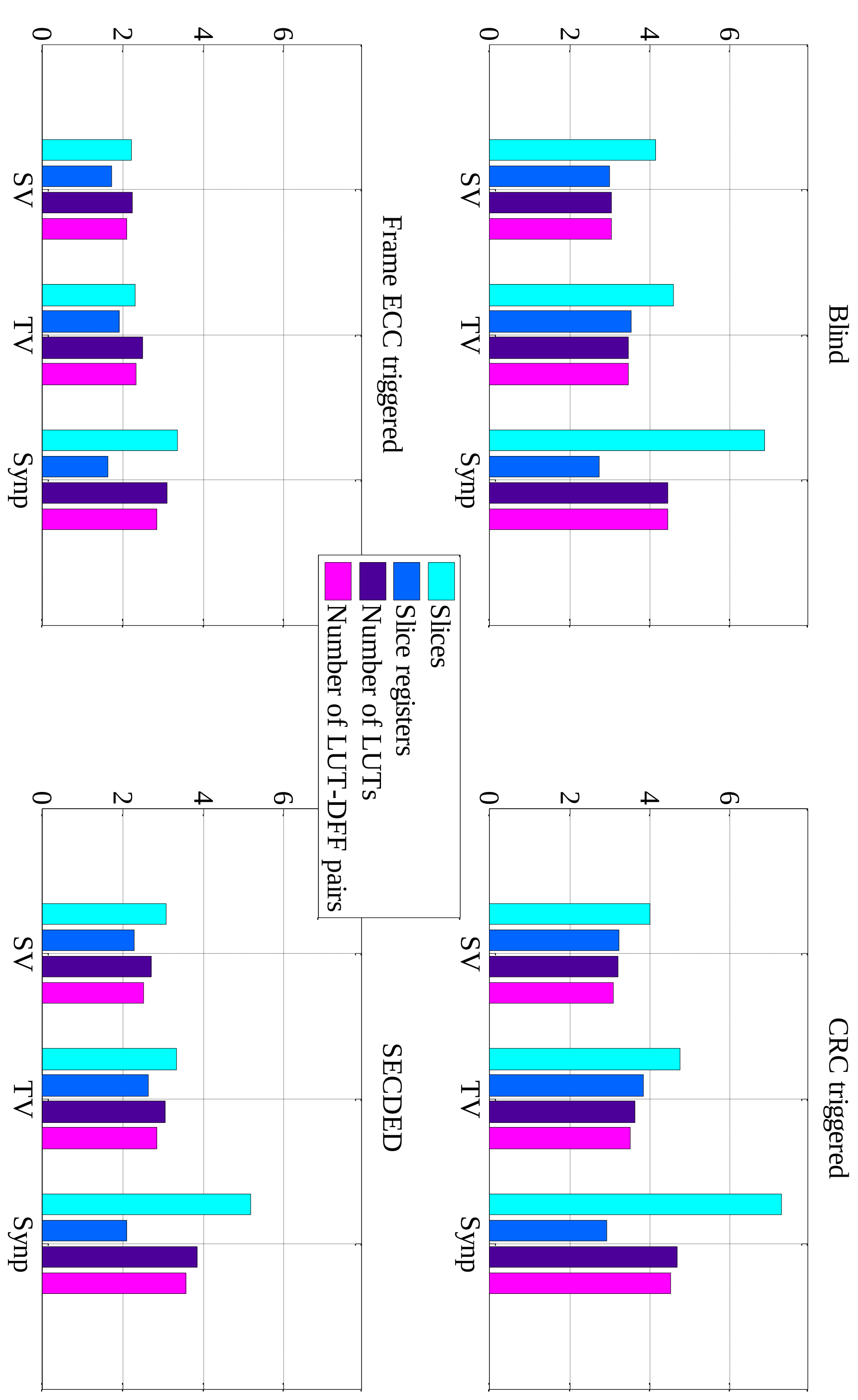


Fig. 63. Relative Resource Usage for Different Scrubbing Methods

4.2.3 Mean Time to Failure

For a lot of applications, Mean Time to Failure (MTTF) numbers can be more relevant and give more information than availability. Fig. 64 shows MTTF as a function of scrubbing rate for blind scrubbing systems, again using a fixed mean time between SEUs of 1 min.

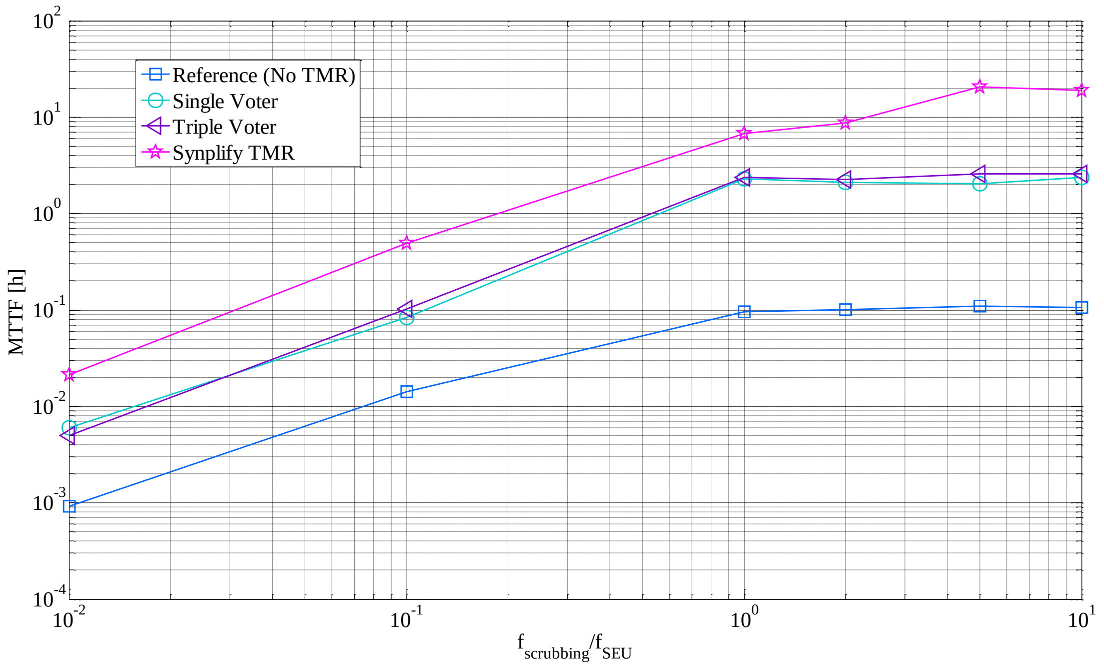


Fig. 64. MTTF, Blind Scrubbing (at SEU Rate 1/min)

Note that the y-axis in Fig. 64 is given in hours and on a logarithmic scale. Note the asymptotic tendency for each TMR implementation.

Fig. 65 and Fig. 66 show MTTF and MTTF per LUT as functions of the SEU rate for blind scrubbing. It should be noted that the z-axis is not normalised in Fig. 65.

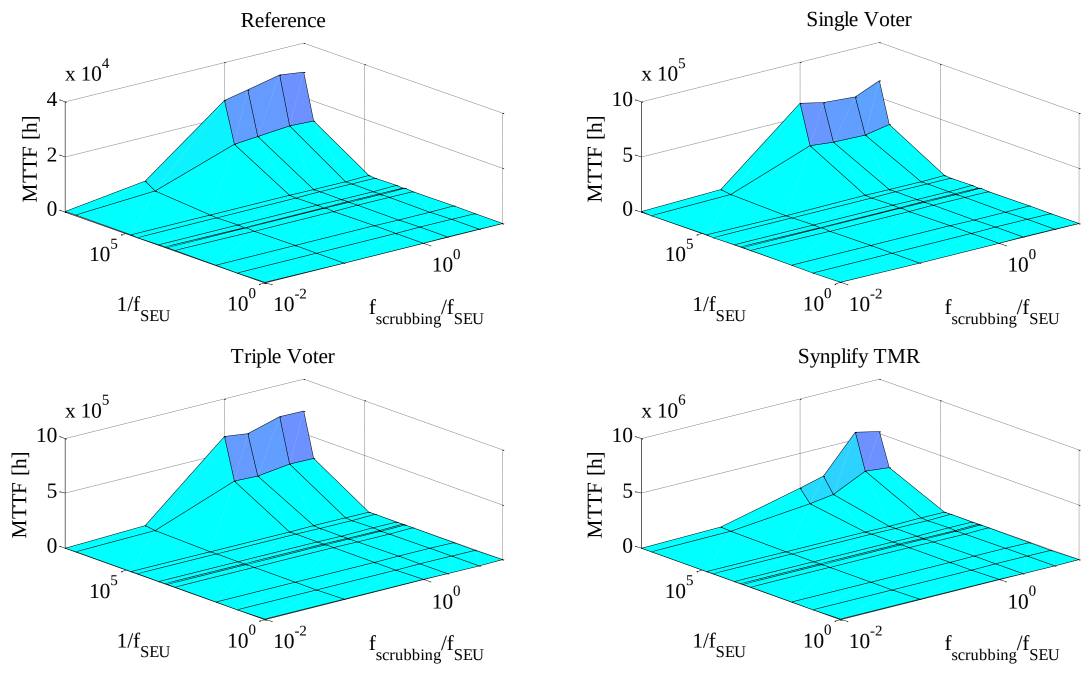


Fig. 65. MTTF, Blind Scrubbing

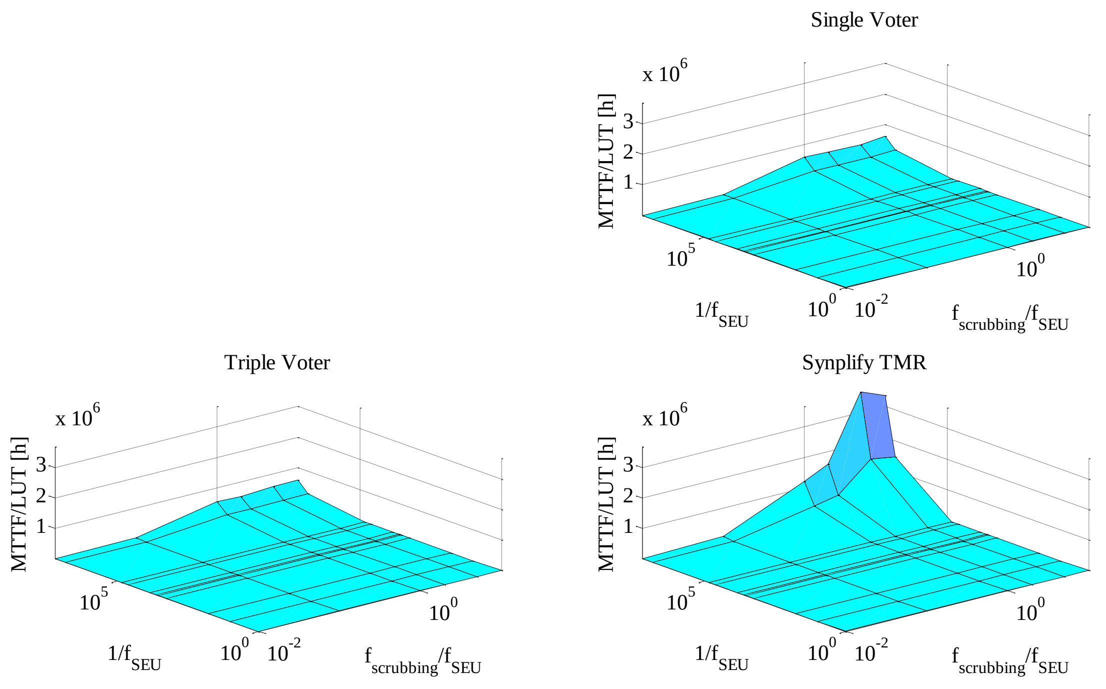


Fig. 66. MTTF per LUT, Blind Scrubbing

Fig. 67 shows MTTF as a function of SEU rate for the three error detection-based scrubbing methods: CRC, ECC and SECDED, given in hours.

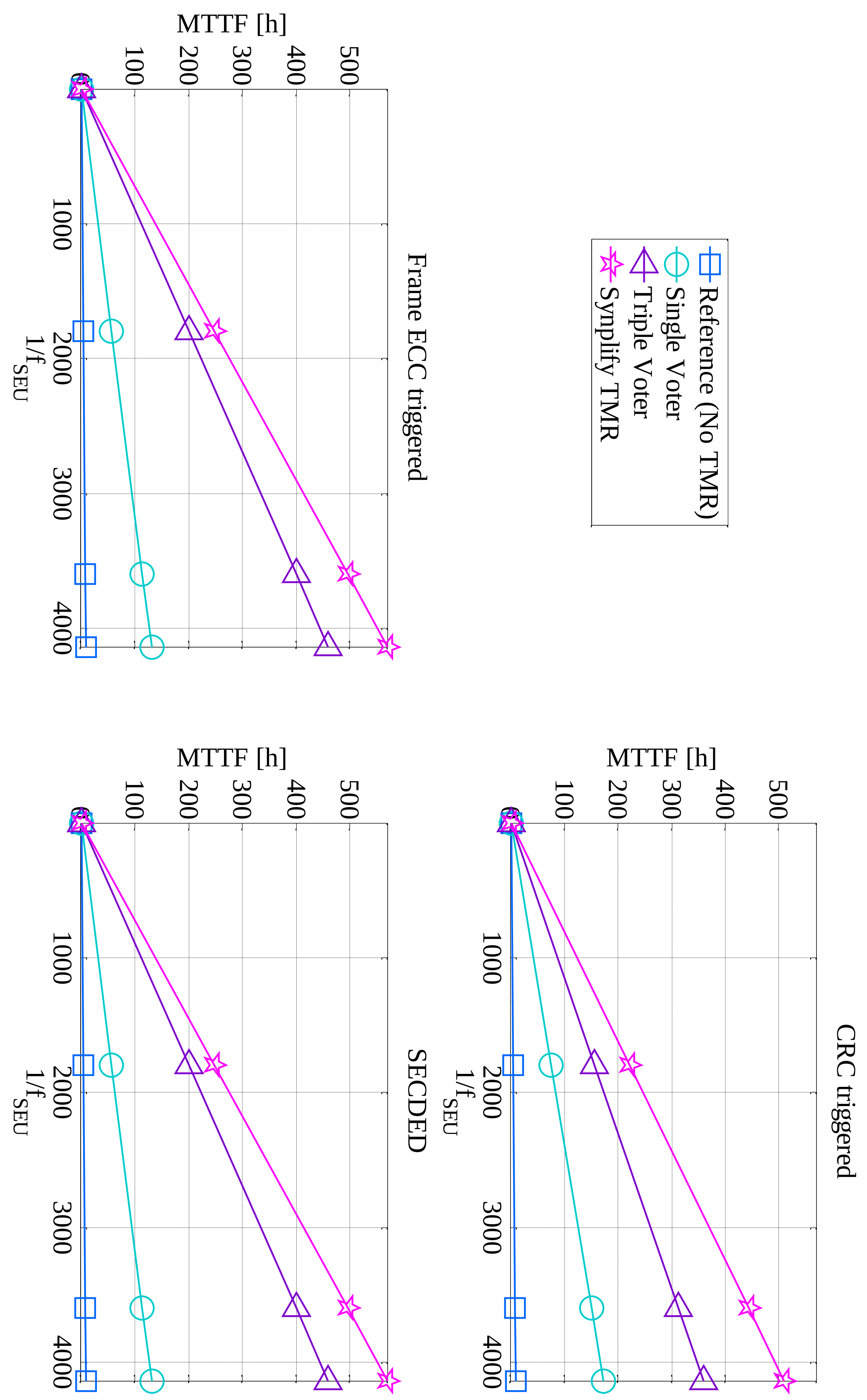


Fig. 67. MTTF, Error Detection-based Scrubbing

Fig. 68 shows MTTF per LUT as a function of SEU rate for each of the TMR implementations for the detection-based scrubbing methods.

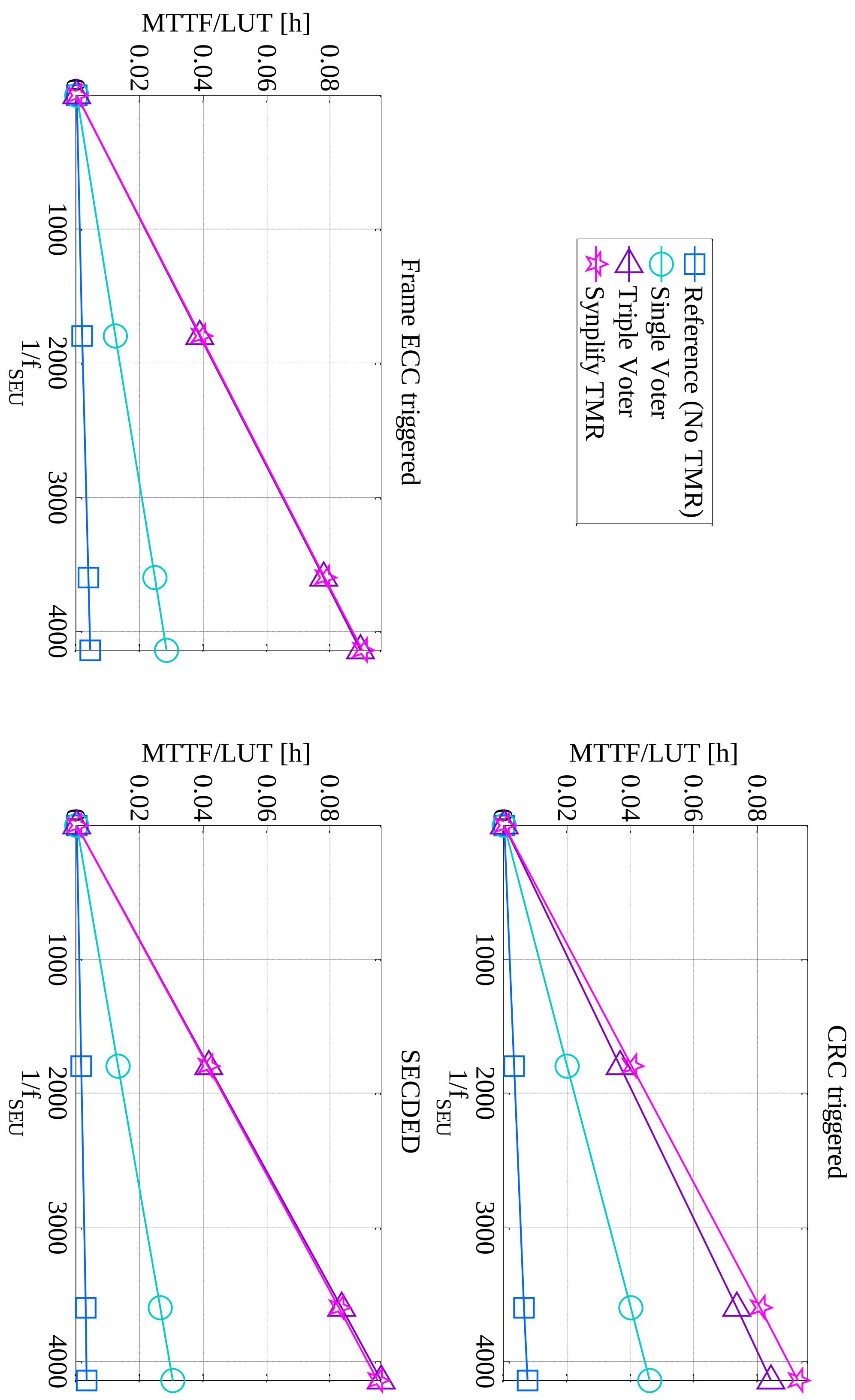


Fig. 68. MTTF per LUT, Error Detection-based Scrubbing

4.3 Faults In Time

Depending on implementation there will be difference in availability, MTTF and FIT. The relation between these metrics and how they are derived is described in 2.4.1. Systems with the CRC, ECC and SECDED implementation will yield a high availability. In these specific cases, the availability comes close to 1 and the difference between the implementations will be negligible. A plot showing the different availabilities will therefore not be a fair comparison. MTTF and FIT will be shown for all combinations instead. Fig. 69 shows MTTF for the error-detection based scrubbing methods, similar to Fig. 67, but over an extended x-axis.

Using these numbers, by a simple rescaling of MTTF according to Eq. 17, a FIT value is obtained. Fig. 70 show FIT values for the error detection-based scrubbing methods.

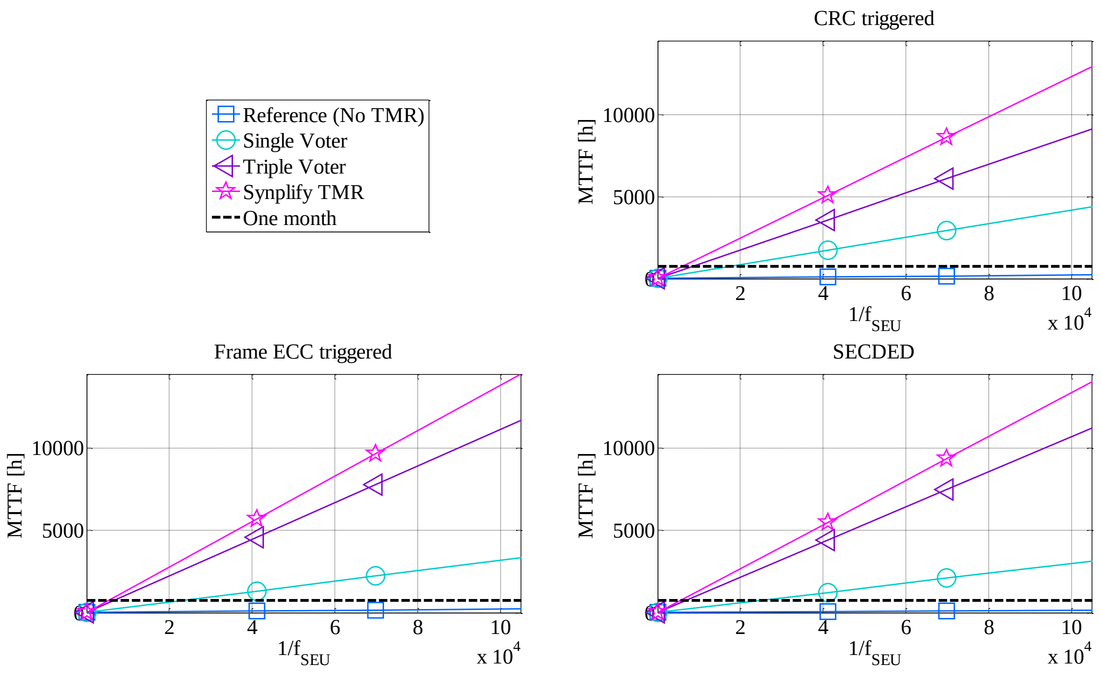


Fig. 69. MTTF, Error Detection-based Scrubbing (extended x-axis)

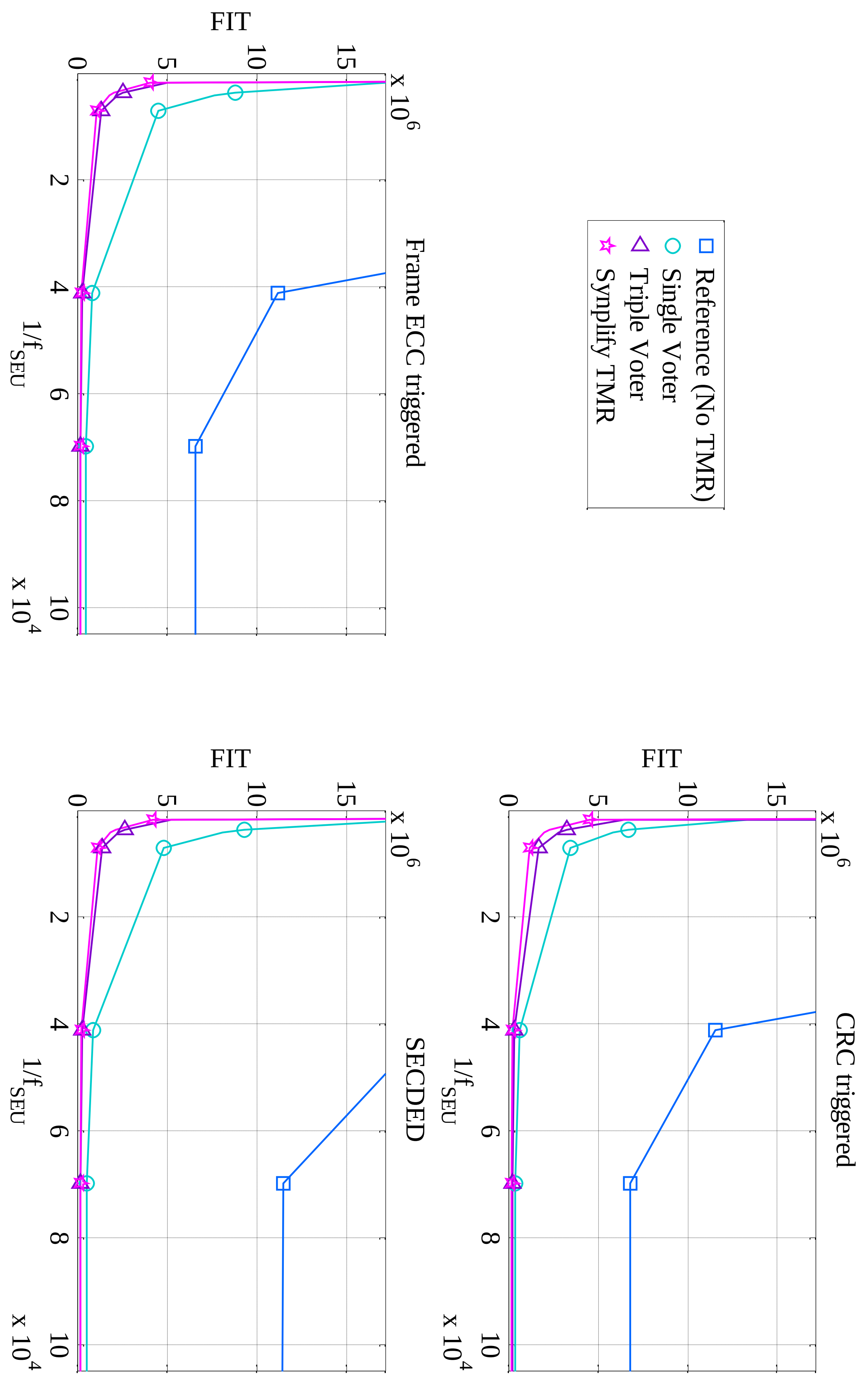


Fig. 70. FIT, Error Detection-based Scrubbing

4.4 Example LEO and GEO Scenarios

In order to provide realistic and complete examples of the feasibility of SRAM-based FPGAs in space, two common satellite orbits have been selected. One Low Earth Orbit (LEO) and one Geosynchronous Orbit (GEO) are used, as they represent the most common cases for commercial satellites. This section will present a complete example, weighing in device characteristics, satellite orbit parameters and radiation models, finally arriving at a FIT value for each orbit, for the particular FPGA type.

4.4.1 Device Characteristics and Test Application

The FPGA used throughout the experiments detailed in this report is the XC5VLX50. However, the XC5VLX50 is a commercial, non-rad-hard FPGA, and so no radiation testing results are available from the manufacturer. Because of this, calculations will be made using the device characteristics published for another Virtex-5 series FPGA, the Virtex-5QV FX130T (XQR5VFX130T). The Virtex-5QV FPGA is a radiation-hardened version for space applications [64], for which radiation testing results are available. The FX130T has a larger capacity than the VLX50 FPGA. In [65], extensive radiation test results for the Virtex-5QV FX130T are presented.

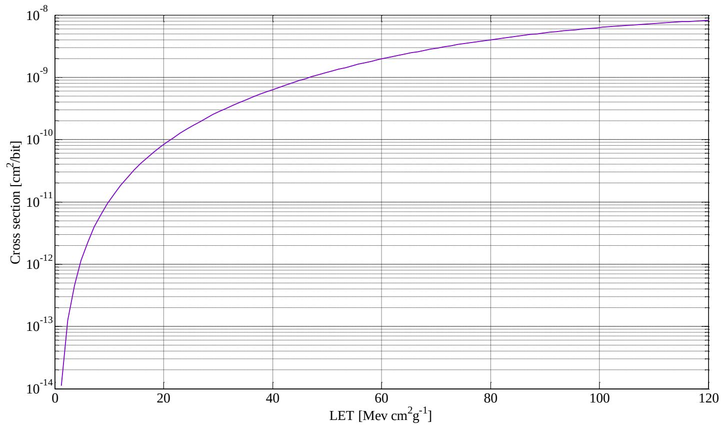


Fig. 71. Cross Section, Heavy Ions

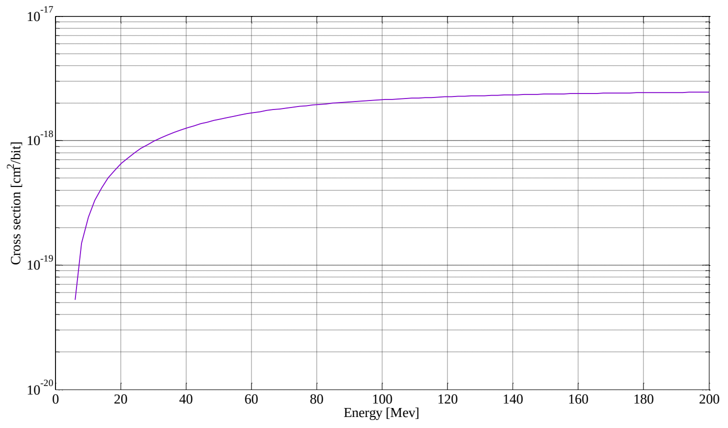


Fig. 72. Cross Section, Protons

Fig. 71 and Fig. 72 show the cross section of the FX130T device for heavy ions and protons, respectively. The Weibull parameters used are from [65].

The test application is the same 128-bit AES encryption application used in the previously described experiments, but scaled up to fill the entire FPGA. For the purpose of calculating SEU rates, all the configuration cells are considered as being used, along with all user flip-flops. No DSP48E slices are used. Two different versions of the test application for each orbit will be used: a version with 5% BRAM usage, and another version where 50% of the available BRAM is used. Using the resource usage report from the *.mrp* files generated by PlanAhead post-PAR and the resources available on the FX130T as stated in [64], the numbers for *functional flip-flops* and device usage percentage are calculated.

Available on the FX130T are 20,480 slices, 320 DSP48E slices, 596x18kB BRAM and 298x36kB BRAM.

In Table 16, the FX130T FPGA has been filled with as many AES blocks as will fit. The level of BRAM used is fixed at 5% of the total available memory. As a measure of the size of the application, and for easy comparison with ASIC designs, the measurement *Functional D-Flip-Flops* is used. This measurement is the number of DFFs that add to the functionality to the system, and does not include redundant flip-flops. For example, a reference design AES block and a TMR AES block has the same number of functional DFFs, as they perform the same function.

A characteristic of the AES block is that it is logic-dominated, meaning that LUT resources will be the bottleneck when scaling the design to fill the whole FPGA. These numbers are highly application specific. The total number of DFFs available is 81,920. Theoretically, all of these could be used in an application (see Table 17).

Table 16. Functional DFFs, AES Application in FX130T FPGA

TMR	Functional DFFs	BRAM	Device Usage	AES Blocks
Reference	28,968	512 kB	99.1%	71
Single Voter	8,976	512 kB	99.3%	22
Triple Voter	7,752	512 kB	98.0%	19
Synplify TMR	6,120	512 kB	99.5%	15

Table 17. Functional DFFs, Theoretical Maximum in FX130T FPGA

TMR	Functional DFFs
Reference	81,920
Single Voter	25,380
Triple Voter	21,920
Synplify TMR	17,320

4.4.2 Satellite Orbits and Radiation Profiles

The example Low-Earth Orbit is specified by the parameters in Table 18, and its ground track is illustrated in Fig. 73.

Table 18. LEO Parameters

Altitude	800km
Inclination	98°
RAAN	20°
Argument of Perigee	0°
True Anomaly	0°

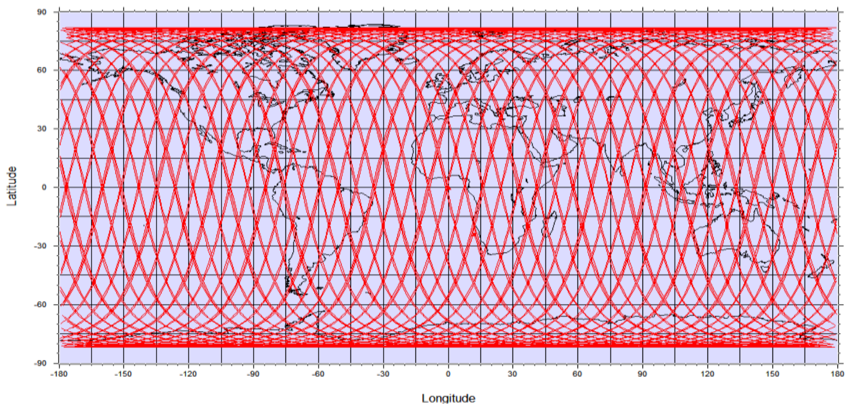


Fig. 73. LEO Satellite Groundtrack

This represents a standard *sun-synchronous* orbit with a retrograde inclination, typical for example for a remote sensing / Earth observation satellite. Aluminium shielding of thickness 0.100" is included in the calculations in this chapter. Table 19 presents the orbital parameters for the Geostationary (GEO) orbit used, illustrated in Fig. 74.

Table 19. GEO Parameters

Altitude	35,786 km
Longitude	-55°

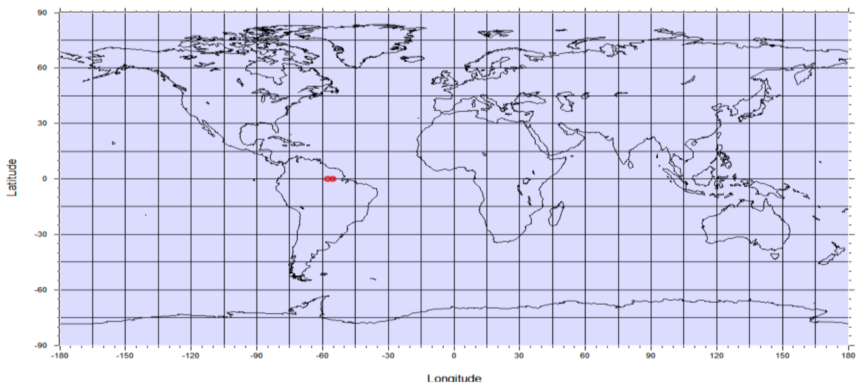


Fig. 74. GEO Satellite Groundtrack

The LET spectrum for the LEO is shown in Fig. 75 and the LET spectrum for GEO is shown in Fig. 76.

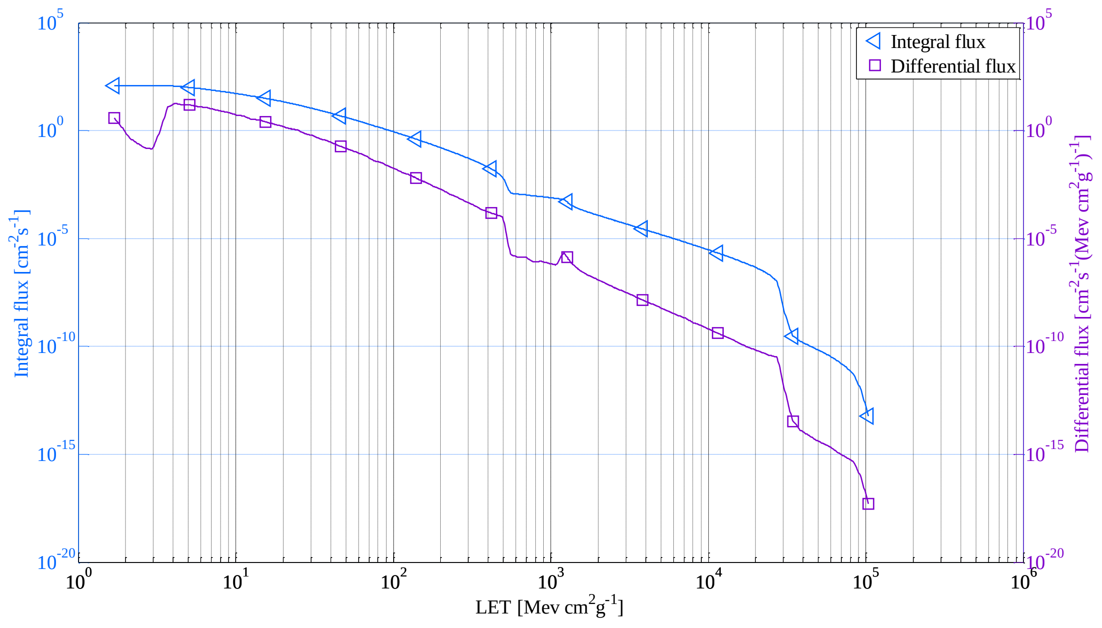


Fig. 75. LET Spectrum, LEO

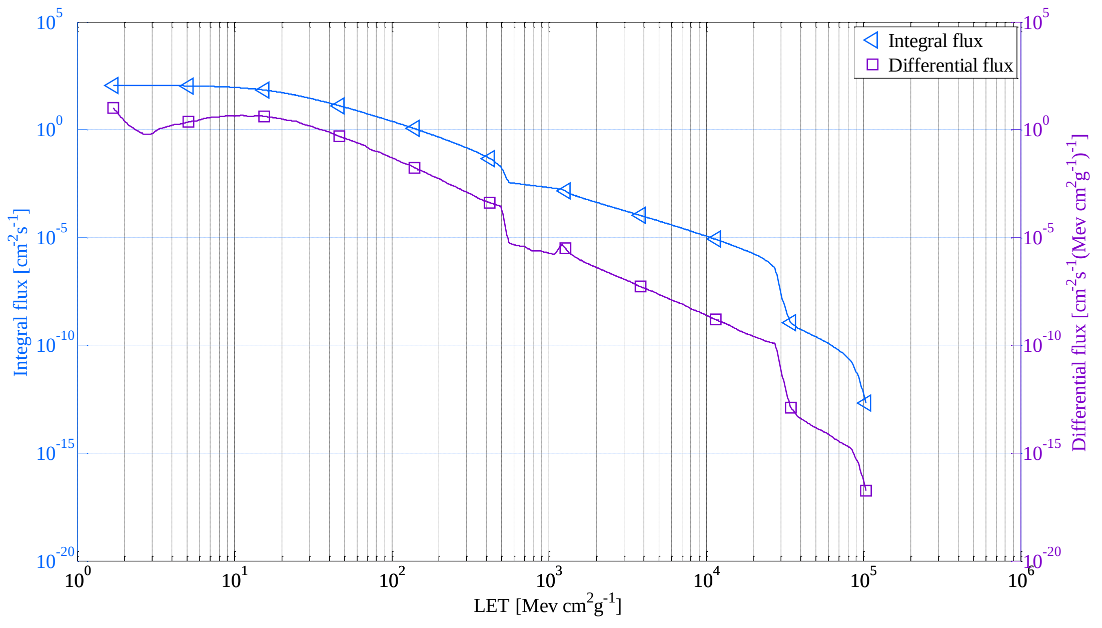


Fig. 76. LET Spectrum, GEO

4.4.3 FITs

The expected SEU periods for the two examples orbits are derived from Fig. 75 and Fig. 76. The corresponding SEU rates for each of these cases are shown as vertical lines in Fig. 77 and Fig. 78. Furthermore, FIT is calculated for LEO and GEO for the SECDED implementation. These values are presented in Table 20.

MTTF values for the different implementations are presented in Fig. 77. The x axis shows the SEU period in seconds. The FIT for all detection based-scrubbing combinations as a function of SEU period is shown in Fig. 78.

Table 20. FIT, Example Orbits using SECDED Scrubbing

Orbit	BRAM Usage	Reference	Single Voter	Triple Voter	Synplify TMR
LEO	0%	4.835E+04	2.016E+03	5.635E+02	4.515E+02
	5%	1.146E+07	4.778E+05	1.336E+05	1.070E+05
	50%	1.142E+08	4.760E+06	1.331E+06	1.066E+06
GEO	0%	1.108E+05	4.618E+03	1.291E+03	1.034E+03
	5%	1.948E+07	8.122E+05	2.270E+05	1.819E+05
	50%	1.938E+08	8.081E+06	2.259E+06	1.810E+06

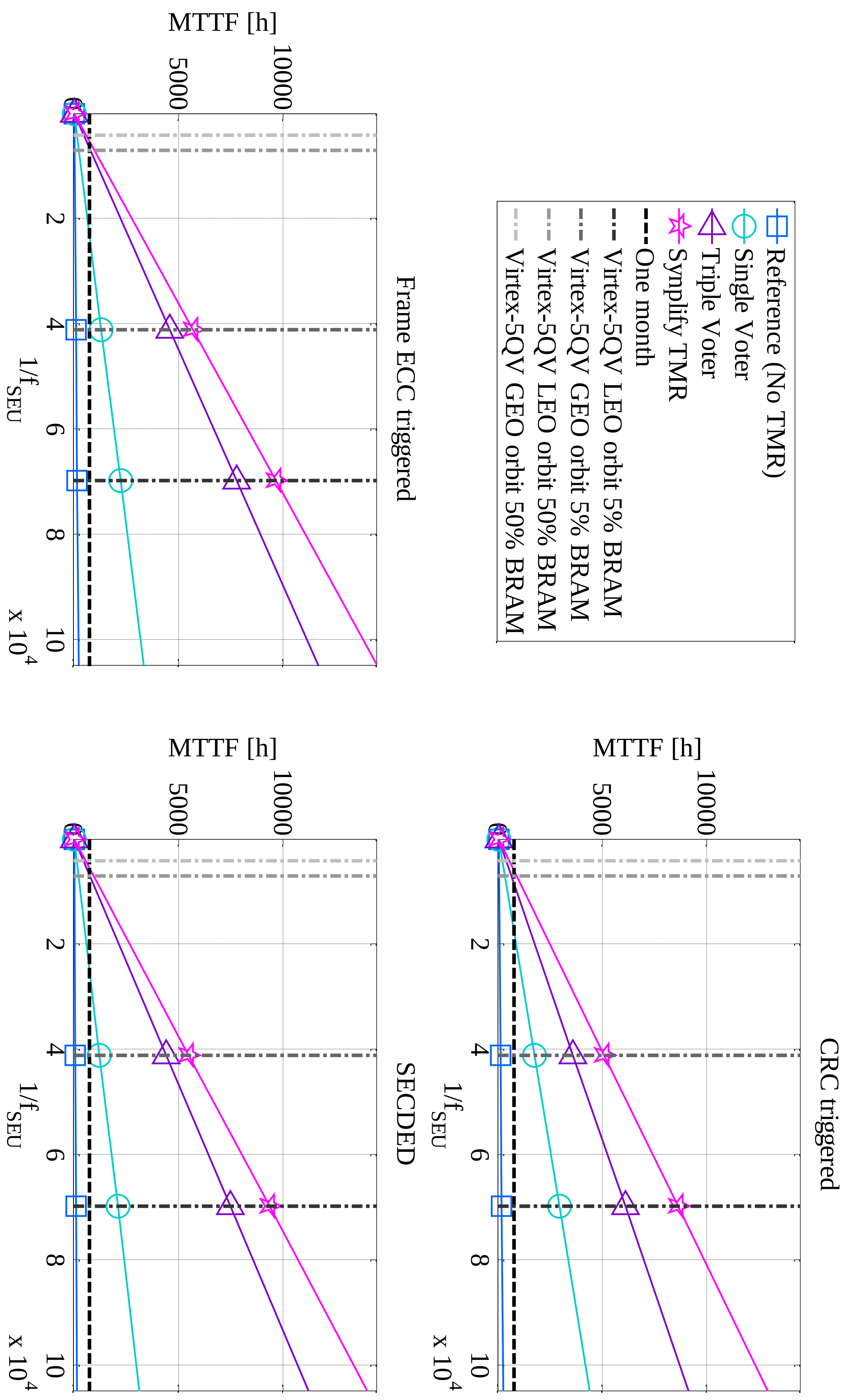


Fig. 77. MTTF, Error Detection-based Scrubbing

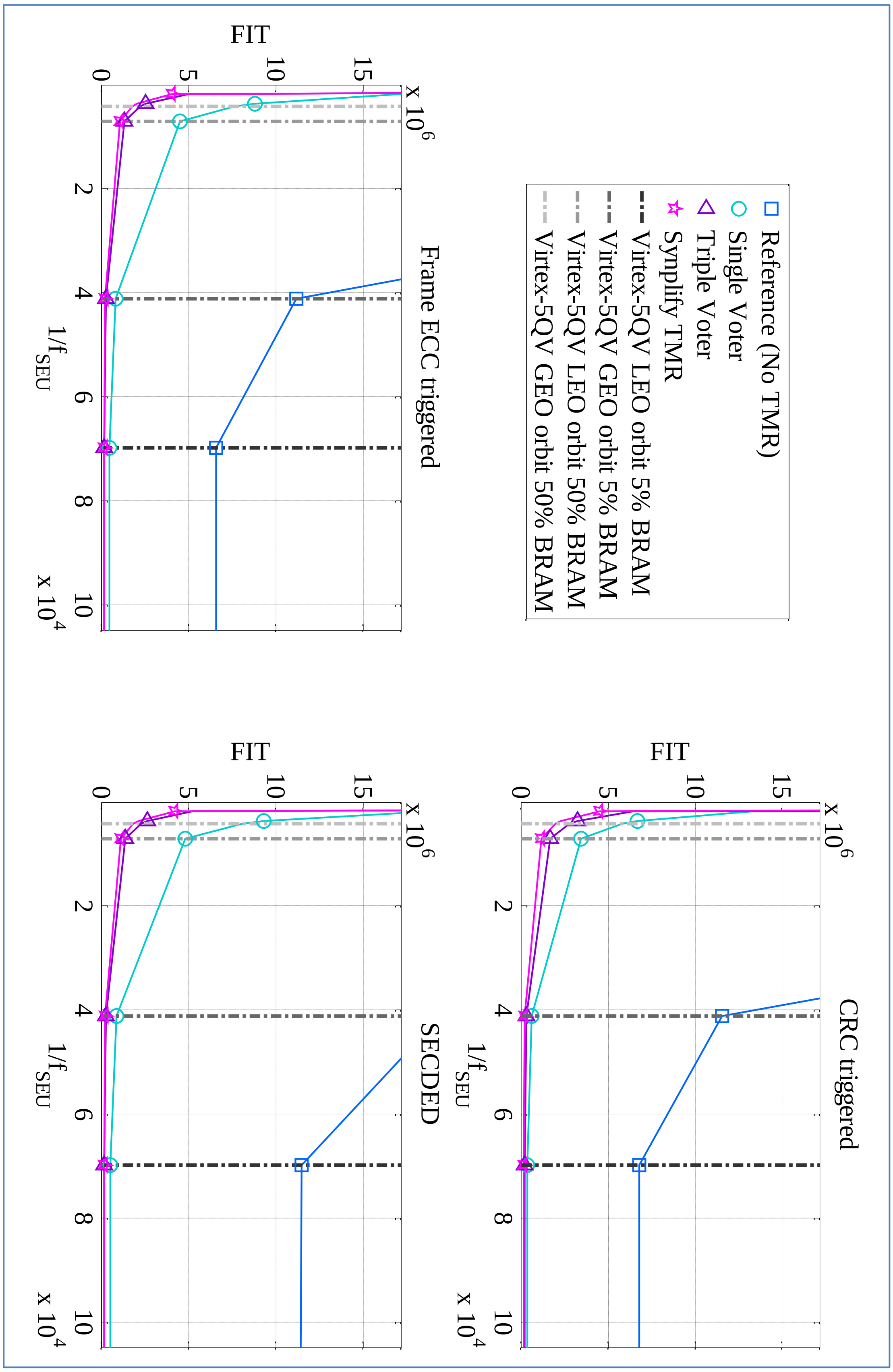


Fig. 78. FIT, Error Detection-based Scrubbing

5 Discussion and Recommendations

This section provides a discussion of the method and results, and also gives a set of general recommendations for the use of Xilinx commercial SRAM-based FPGAs in space applications.

5.1 Relevance and Limitations of Test Method and Results

The results presented in this report apply to the selected test application, the 128-bit AES encryption block. While the techniques discussed are general, the results, and therefore the optimal mitigation strategy, will vary widely depending on the application. Using an AES application gives the tests a high level of observability, since an error anywhere in the encryption chain is likely to propagate through to the output. The AES application is rather logic-heavy, in that it uses a lot of logic rather than sequential elements. The AES application has no feedback or dependency on earlier encryptions, so the results presented in this report will not take feedback effects into considerations. In many real applications, there is feedback or strong state dependencies that may exhibit different error patterns when incorrect state or data is introduced into the system. The test method only tests upsets in the configuration memory. It does not test user data upsets or single event transients (SETs).

The AES application used here does not make use of BRAM or DSP blocks. It is a conscious choice to make the application as scalable and as placement-independent as possible, by using only fabric resources. Also, it allows for tighter packing into a p-block than might otherwise have been possible. This means that the AES application can be duplicated to fill up the entire FPGA with a high degree of resource utilisation, with respect to logic slices. A scaled-up version of the system, using multiple instances of the AES application, shows a similar hit percentage of SEUs. Overall, the application, and therefore the results, can be scaled to fit any size of FPGA, while retaining its behaviour.

As this report aims to investigate the feasibility of using non-rad-hard, COTS SRAM-based FPGAs in space, the AES block represents a realistic application. This type of FPGA would typically not be used in critical applications such as On Board Computers. Therefore, a microprocessor application is not selected. Applications such as the AES block would typically be found in communication modules or payload instruments, where commercial SRAM-based FPGAs could be considered.

While the test platform and method have some limitations and error sources, overall they produce reliable results that are representative of a real FPGA application in a space scenario. By running tests on an actual FPGA and injecting errors in a predictable and controlled manner in the configuration memory, the test method covers all the possible error modes, given a large enough number of fault injections.

The fault injection method and error logging used in this project are intrusive, in the sense that they are implemented, synthesised, placed and routed on the FPGA side-by-side with the DUT. Because of this, the test framework will unavoidably affect the DUT. While communications and interactions between the DUT and the test framework are kept to a minimum, there are some shared connection points. This may affect the statistics and results slightly. However, these effects are unlikely to have any significant impact on the trends shown by the results. The tests do show some variations from test run to test run. By using a large number of injected faults (tens of thousands), these variations are averaged out.

Looking at the test platform itself, the major bottleneck in terms of test performance is the serial UART communication between the host PC and the test FPGA. Also, the

Xilinx SEU Controller macro used for fault injection has an upper limit on how often faults can be injected. The “tick” system that the Tcl application on the host PC uses allows for the UART communication time to be excluded from the ticks and therefore scaling of the SEU period without re-running tests. This is because all times are measured in ticks, which is a unit-less measurement that can then be scaled by a time factor according to the SEU period. Only reconfiguration events and detection times are measured in absolute time, and have to be converted to ticks. Using this method, the calculations do not have to take the communication latency between the host PC and the test FPGA into consideration, as all communication events are considered to take place inside of a single tick slot.

Some possible sources of errors during test runs are single event upsets that affect the communication between the SEU Monitor and the DUT. It has been noted during some tests that this may cause the test system to time-out while waiting for a response from the DUT. In case this happens, it is logged by the SEU Monitor as an error. This however was corrected by modifying the monitor to exclude a non-responding AES unit and log the test vector anyway.

5.2 Trends in Result Data

Some important trends and tendencies have been observed during the test data analysis. First of all, it should be clear to designers that an availability of 100% cannot be achieved and, in fact, should not be the goal of a design process. The higher the availability is, the costlier it will be to improve it further, in terms of resources and system complexity. Where the threshold is between what is feasible, in terms of availability per resource, will depend on the application.

Overall, cost is an important aspect of fault-tolerance. Fault-tolerance carries a cost either in terms of resources or in terms of (down-) time. Triple Modular Redundancy introduces a considerable resource overhead. Depending on the implementation, this overhead is somewhere in the range of 200% - 400% of the original implementation. This can be clearly seen in Table 13, and is supported by the literature ([30], [20]). Depending on how the application is placed and routed on the FPGA, and on what special resources it uses, the effective overhead could be even greater. If the resource cost of applying TMR to the whole application is too great, it can be useful to consider partial TMR, applying redundancy only to critical parts of the application (as discussed in 2.4.2), such as feedback loops or the circuitry in finite state machines keeping track of the state.

Scrubbing circuitry can be placed internally on the FPGA, or externally. Internal scrubbing logic uses resources on the FPGA, introducing overhead, as seen in Table 14. Depending on which scrubbing method is used and the device utilisation, resource overhead from using an internal scrubber may be significant or not, or may even be the dominant part of the design resource-wise.
Again looking at Table 14, it is shown that different scrubbing methods vary widely in their implementation size. The numbers in Table 14 are for single AES blocks: scaling the application to fill the entire FPGA would make the scrubber implementation overhead much smaller in comparison.

It is important to note that scrubbing circuitry itself is susceptible to radiation-induced upsets. A malfunctioning scrubber can cause a lot of damage by writing incorrect frame data, writing to the wrong frames, or scrubbing unexpectedly. Scrubbers and fault-tolerance monitoring circuitry can be protected by TMR just as the payload application. The advantage of keeping a scrubber internally on the FPGA is that overall system complexity is reduced, and that the scrubber can easily access error detection signals with after a shorter delay. However, internal scrubbers also have the disadvantages of being susceptible to SEUs, as well as taking up resources on the FPGA. External scrubbers can be implemented as radiation-hardened ASICs. This adds to the overall system complexity, but may prove to be a better solution since the scrubber's target system and the scrubber are completely separated.

5.2.1 TMR implementation

In implementing TMR, it can be useful to consider the observability of errors. As an example, the Synplify TMR implementation used in this project offers no way of observing partial errors, as only errors on the output from a module can be seen. In the Triple Voter TMR design on a module-level, on the other hand, errors on separate TMR branches and voters can be observed. This can be used in error detection, if the application allows it.

An important and very generally applicable concept is that adding redundancy also adds area to the application, thereby adding more resources susceptible to SEUs. That is, the redundancy overhead also has a failure rate. This can be seen as making the target for SEUs bigger, but at the same time more robust. The same concept also applies when scaling an application. Going from a design comprising 1,000 functional DFFs to the same design scaled to 10,000 functional DFFs will cause MTTF to scale linearly, and FIT to scale accordingly, with the area increase, when considering the whole system. Simply put, more functionality means more SEU-susceptible flip-flops.

The static testing results in Table 11 give a good sense of how good the different TMR implementations are at masking errors. The reference design is provided as a baseline, and gives an estimate of how large a percentage of injected errors will cause actual observable errors. Throughout the experiments conducted and described in this report, about 25% of the injected errors give an observable error somewhere in the system. For the Single Voter and Triple Voter TMR designs in Table 11, there are about 700 observable errors per 4000 injected errors. However, most of these errors affect single TMR branches, and are masked by the voter stage. For the Synplify TMR implementation, only failures are observable using the method described in this report. As Synplify TMR applies TMR on a much lower level than the single and triple voter module-level TMR designs, only the actual output from the whole AES block can be observed.

Comparing the different TMR implementations, it can be seen that the Triple Voter and Synplify TMR designs offer a similar level of masking, while the Single Voter design has a slightly higher failure rate. This is because the voter stage in the Single Voter design represents a single point of failure. An SEU affecting the voter output itself cannot be masked by the Single Voter design. Triple Voter and Synplify TMR on the other hand, can deal with such errors. None of the designs, however, can protect against the case where a single bit upset affects two separate TMR branches (a so called bridge error). These bridge errors are the primary cause of failure in the Triple Voter design.

Different TMR implementations will yield different numbers and types of errors, as can be seen in Table 12. For the Single Voter, bridge errors, voter errors and multiple errors cause incorrect output results, while single errors are masked. As single errors make up the majority of observed errors, the single voter design is indeed able to mask a large portion of the observed errors. The reason why the amount of single errors is so much greater than bridge, voter and multiple errors is quite simply explained by the fact that the AES modules occupy more resources compared to the voter, whose size is almost negligible in comparison. Bridge errors are rare, because they can only occur when two separate TMR branches erroneously become connected and therefore share an error. As noted earlier, the single voter is susceptible to voter errors, as it is a single point of failure for the design. The increased amount of voter errors for the Triple Voter TMR design compared to the Single Voter design can again be explained by the simple fact that three voters make a bigger target than one.

Overall, the Triple Voter design sees only a slightly higher number of total observable errors compared to the Single Voter design, which is roughly proportional to the size difference. It should be noted that there is an increased number of voter errors observed due to the larger amount of resources occupied by voters, but that these can in most cases be masked out by the triplicated voters. In the case of the Synplify TMR design, it is not possible to observe errors other than correct or incorrect output.

5.2.2 Scrubbing implementation

The relative-area-column of Table 14 shows some interesting results. Out of the implemented scrubbing methods, Frame ECC-based detection with dynamic partial reconfiguration is the most expensive in terms of resources. A SECDED scrubber making use of the Xilinx SEU Controller is a slightly more economical option. CRC-based full reconfiguration presents little overhead. Together, these observations give an idea of the area overhead for each scrubbing method, to be kept in mind while looking at MTTF and FIT numbers.

The performance of blind scrubbing has been extensively researched in this work. In particular, the relation between scrubbing frequency and SEU rate has been studied.
Looking at Fig. 57, the Synplify TMR system consistently performs better than the other TMR implementations, up to a scrubbing rate of 1 (scrubbing at the same speed as expected SEUs). The plot in Fig. 57 is fixed at an SEU Rate of 1 SEU per minute. An interesting thing to note in this plot is that availability seems to drop after scrubbing rate 1. The scrubbing rate, at which this happens, varies widely with the mean time between SEUs. For longer mean time between SEUs, a higher scrubbing frequency to SEU frequency ratio will result in a better availability.

Looking at the same trend, but also varying the SEU rate, gives Fig. 58. In Fig. 58, a sharp drop is seen for all TMR implementations when the SEU and scrubbing frequencies are both high. This is because the system is trying to scrub faster than what is possible. A full reconfiguration takes a certain amount of time (~500 ms).
Trying to perform a full reconfiguration (scrubbing) at 10x the expected SEU frequency, if the mean time between SEUs is 1 second, will result in the system being down for scrubbing 100% of the time. Fig. 59 shows this behaviour by plotting the scrubbing rate and SEU rate versus the portion of the total system downtime that is caused by scrubbing. A reasonable conclusion from these figures is that scrubbing more often does not always lead to better availability, and can in fact be directly harmful to the availability of a system, if done too often. In addition to this, power and system complexity concerns should be added.

The black line drawn for each TMR implementation in Fig. 58 gives the optimal scrubbing rate for a given SEU rate. Comparing the four implementations, it can be seen that the reference design can be scrubbed more often in relation to the SEU rate, without losing in availability. This is because the time spent in scrubbing mode is quite small compared to the time the circuit is producing an incorrect output, as the reference design is so sensitive to errors in the first place. The Synplify TMR implementation is much more fault-tolerant.

In Fig.60, the SEU rate (mean time between SEUs) has been fixed at 1 SEU / minute to visualise more clearly the cost of scrubbing in terms of downtime. These graphs show that there is a point where the dominating contributor to system downtime shifts from being wrong result on output to scrubbing. The higher the masking rate of the TMR implementation, the most sudden this shift will be. The main thing to note in Fig. 60 is the difference between the reference design and the TMR designs. The reference design gradually shifts over, while the TMR designs shift more suddenly. In the reference design, a much larger proportion of the downtime is caused by having an incorrect circuit, producing an incorrect output. As mentioned earlier, the scrubbing rate at which the switch of dominant factor from wrong output to scrubbing occurs depends on the mean time between SEUs.

For a sense of how area-efficient the tested SEU mitigation techniques are, Fig. 61 shows used resources per availability, at a fixed SEU rate of 1 SEU / minute. As for the previous pairs of graphs, this is accompanied by Fig. 62, where the SEU rate is varied. An interesting thing to note here is that redundancy implementations with high levels of availability are (un-proportionally) expensive in terms of resources. The reference design is not included in Fig. 62, as it would show much higher values than the TMR implementations.

The first row of Table 14, CRC scrubbing with an SEU rate of 1/second, is interesting. Given that a full reconfiguration takes about 500ms, the system will be performing reconfigurations half of its total operational time. This gives a low availability, of around 0.5. For the reference, Frame ECC-based detection scrubber with SEU rate 1/second, availability is much higher. This is due to the fact that even if Frame ECC is slower than CRC to detect errors, repairs are made much faster, as only a partial reconfiguration of a single frame has to be made, rather than a full reconfiguration.

Looking at resource usage for the TMR implementations, as illustrated by Fig. 63, three distinct area profiles emerge. The Triple Voter design uses the most slice registers, but comparing overall resource usage in LUT-DFF-Pairs used, Synplify TMR is the most expensive implementation in terms of area. The LUTs are the bottleneck in these TMR implementations. The Single Voter, Triple Voter and Synplify TMR use roughly a factor 3, 4 and 5 times more area than the unprotected reference design, respectively.

A combination of a low SEU rate and a high scrubbing-rate-to-SEU-rate ratio results in high MTTF. The first thing to note is that Synplify TMR is consistently the best-performing TMR implementation. Secondly, it can be seen from Fig. 67 that the Triple Voter and Synplify TMR designs benefit more from using partial reconfiguration than the Single Voter TMR design does. As mentioned earlier, this is due to the fact that the Single Voter design has a single point of failure, the voter. The green line is placed at 730 hours, which is equal to one month.

Looking at Fig. 68, an interesting tendency to note in this figure is that the Triple Voter and Synplify TMR designs are almost exactly as area-efficient. Also, comparing Fig. 68 to Fig. 62 shows that while the unprotected reference design may give the highest availability-per-LUT ratio, it is much worse when looking at the MTTF-to-LUT ratio.

Ticks, which are used in calculations by MATLAB, are scaled with SEU period. However, some duration are counted as absolute values, such as scrubbing duration and error detection times, although these absolute times are negligible in comparison to the ticks which are scaled. This gives MTTF a linear relation to the SEU period, which is illustrated by Fig. 77.

When MTTR<<MTTF, which is true in the presented example, then MTTF ~ MTBF. From this information it is expected that FIT as a function of SEU period resembles an f(x) = 1/x curve. Synplify TMR has the highest MTTF for all SEU periods which yields the lowest FIT for each SEU period, as shown in Fig. 76. The FIT is calculated as a function of MTTF as shown in Eq. (17).

5.3 Orbit example calculations

As the two different orbits differ severely in altitude, it is expected that the LET spectrum for the two respective orbits are different. An observable difference between the two spectrums is that the particle fluence is higher for all LET levels in the GEO spectrum. This is expected as LEO is on a much lower altitude and therefore under the influence of Earth’s protective magnetic field.

FIT is a common figure of merit used in conjunction with fault tolerance. When a system designer specifies a required fault tolerance value, this number is often given in FIT. The estimated requirements can be used in two different ways when assessing fault tolerance. The first approach is to start at a given FIT value. For this required FIT value, each of the detection scrubbing implementations will give a value for the maximum tolerable SEU period. This can be seen graphically where a FIT corresponds to a horizontal line, as is seen in Fig. 77 and Fig. 78. This horizontal line will intersect with the different implementations at different SEU rates. The shorter the SEU period, the smaller the radiation constraints have to be on the target FPGA.

Another approach is to pick a target FPGA, run radiation tests for this particular FPGA and resolve an SEU rate by integrating it with particle fluence for the orbit of interest. With these parameters, the SEU period can be used to find which implementation yields the lowest number of FIT. The SEU period can be represented graphically as a vertical line where the resolved number of FIT will be where the respective curves intersect with said vertical line, as is done in Fig. 77 and Fig. 78.

5.4 Protecting User Data

In this report, focus has been on mitigation techniques for SEUs in the configuration memory. Protecting user data is also an important aspect in providing fault tolerance. This problem, however, exists in ASIC development as well, so coding standards and design considerations are well established. Using the fault-tolerance mechanisms described in this report will help guarantee the function of the implemented logic, which is a requisite for user data protection schemes. Deploying triple modular redundancy schemes for all user data flip-flops is an efficient mitigation strategy for user data upsets, but drastically reduces the number of functional flip-flops available to the designer. As for configuration memory upset mitigation techniques, when designing systems employing redundancy to protect user data, special care has to be taken to making sure that the synthesis and design tools do not optimise away or inadvertently breaks the redundancy.

There are a few basic rules when designing robust applications. State Machines should be designed without deadlock states, and using dummy states where necessary. This, of course, depends on the designer being able to trust the underlying logic configuration. State machine encodings can be chosen for robustness, for example employing Hamming-3 coding as discussed in Sec. 2.4.4.1.

Errors in user data can be difficult to detect, if the data is not protected by CRC. The only other way to detect such errors is by comparing and detecting errors in output data. If this type of error detection mechanism is implemented in a system, it can for example trigger a reconfiguration of the affected module. However, reconfiguring and resetting a module will cause it to lose its state, and resynchronising with other parts of the system may take a long time.

Synthesis and PAR tools play a role in minimising the susceptibility to user data upsets. In designing redundant, fault-tolerant systems, it is often important to keep track of what is allowed for the tools to optimise and what is to remain untouched to achieve the desired level of redundancy. Considering constant-value flip-flops, these must be synthesised as constant signals or un-clocked flip-flops. During the experiments described in this report, unexpected errors were encountered due to incorrect values being clocked into flip-flops which were intended to hold constant values. Constant values can be considered as user data or as configuration memory, following the division presented in Fig. 20. In [66], the authors provide an interesting discussion on how to approach SEU susceptibility for constant values in FPGAs.

DSP slices pose a challenge in that their internal pipelining registers and configuration vectors cannot be protected by TMR, yet are still susceptible to upsets. Possible solutions are temporal redundancy, or tripling DSP slices on a module level.

In conclusion, user data protection is very much up to the designer to implement and keep in mind. Normal, robust coding and design work for state machine designs should be applied for FPGAs just as for ASIC designs. Larger memory or register blocks such as BRAM can be protected by error correcting codes. Indeed, in space applications, RAM is almost always protected by EDAC. In [63], the authors present results for ECC protected BRAM for the radiation-hardened Xilinx *XQR5VFX130* FPGA. Flip-flops may also be tripled and voted. The role of configuration memory SEU protection is to guarantee the integrity of the logic function implemented in the circuit.

6 Conclusion

The results presented in this report suggest that standard, commercial SRAM-based FPGAs from Xilinx can indeed be used in space applications for standard LEO and GEO missions. The suitability of such devices, however, depends fully on the mission profile, the target device and on the application itself. For suitable applications, the type of FPGA discussed can provide a very cost-to-area-efficient alternative to conventional ASICs, radiation hardened FPGAs and antifuse FPGAs.

It is recommended to implement TMR for most applications. TMR needs to be combined with some type of scrubbing in order to be efficient. For certain applications, duplication with comparison or some other error detection method may be feasible, given that the application can afford the downtime while repairing and rerunning.

When implementing redundancy, tool-inferred TMR constitutes a convenient and safe method of implementation, as no changes have to be made in the RTL code. In Xilinx TMR tool and in Synplify Pro, TMR can be applied simply by setting a flag or setting synthesis attributes.

Error detection-based scrubbing is most often superior in performance and response-time compared to blind scrubbing. In the case of blind scrubbing, the scrubbing rate needs to be adapted to the expected SEU rate. Scrubbing more often does not always lead to better availability for the system. Blind Scrubbing, however, still serves a purpose if done as maintenance in an adaptive manner for instance whenever the system is idle.

The Xilinx SEU Controller Macro / SEM (SECDED) represent an area-efficient and effective single error correction method. Combined with full or partial reconfiguration upon double error detection, the method is an efficient scrubbing method. Partial reconfiguration is in general better than full reconfiguration when combined with TMR, as it allows module-level reconfiguration without interrupting the function of the circuit. However, partial reconfiguration requires extra hardware, and is significantly more complicated than simply triggering a full reconfiguration from an external PROM.

6.1 Other Recommendations

It should be noted that the use of on-chip RAM and DSP slices greatly affects the overall susceptibility to radiation, which in turn increases the SEU rate experienced by the device. BRAM and DSP slices.

It is recommended that each application where a non rad-hard device is considered is evaluated thoroughly. All normal considerations regarding protection of user data, FSM states and block RAM that apply to ASIC and rad-hard FPGA designs should be applied to designs in SRAM-based FPGAs as well. This type of FPGA should be considered for non-critical systems only, such as instruments, image processing applications and non-time-critical communication links. Mission-critical, safety-critical or real-time systems should not be implemented in standard commercial SRAM-based FPGAs. Mission length will also need to be factored in the decision of whether or not to use an SRAM-based commercial FPGA. For longer missions, system designers need to make sure the selected FPGA has a high enough tolerated TID.

Certain applications are more suitable for implementation in SRAM-based FPGAs than others. Some applications can afford to rerun computations or transmissions, or may have a natural downtime, such as communication links that are only active for short periods. In these applications, scrubbing can be performed by taking advantage of the natural window of downtime. This type of application load-based scrubbing relies on the implemented fault masking to mask any errors during heavy workloads, and scrubbing to correct errors when the application is free.

If redundancy is implemented in RTL code (represented by the module-level single and triple voter TMR designs in this report), special case has to be taken to ensure that the synthesis tool does not remove the intentionally designed redundancy by optimising. In general, the designer needs to pay attention to what the design tools are doing to the code, in order to make sure that the intended fault-tolerant techniques are kept. Synthesis attributes to preserve redundancy are available in XST and Synplify, and should be used. It is usually a good idea to keep a critical stance towards the design tools. Even if no errors or warnings are reported, the tools may not produce what the designer expects. Also, the designer should make sure that the synthesis tool does not remove unreachable FSM states, as they may be important for robustness. In Synplify Pro, it is recommended that the designer specifies the attributes *syn_keep*, *syn_preserve* and *syn_encoding* for signals, components and state machines, respectively.

If observability can be implemented for an application, at a feasible hardware cost, it is often beneficial. Errors can be observed at module-level internally in the FPGA, or at a device-level in the system. On a system level, it may be efficient to implement a single, radiation-hardened scrubbing ASIC tasked with scrubbing multiple FPGAs. In case one of the FPGAs in a system encounters an error, a system alarm type of signal can be triggered, allowing the other FPGAs to adapt their workload while the faulty FPGA is being repaired.

7 Future Work

Building upon the work presented in this report, several extensions are possible that would add to the quality of the results. Using an application with feedback or strong state dependence as a payload would allow the study of fault propagation and persistence. It would also be a good platform for studying the efficiency of partial TMR, for example by protecting only vital feedback loops with redundancy. An FIR filter application was implemented during this project, but was never fully tested due to a lack of time.

The test platform and host PC application show some unexpected behaviour from time to time, most likely due to bugs. Adapting the platform to other FPGA architectures than Virtex-5, and running tests to compare different FPGA families, would be a highly interesting experiment.

8 Acknowledgements

We would like to express our gratitude to our supervisors, *Peter Spjuth* and *Vilhelm Geijer* at RUAG Space, for their support, suggestions and patience with technical questions, in spite of their busy schedules. Also we would like to thank our examiner and academic supervisor, Professor *Per Larsson-Edefors* at Chalmers University of Technology, for guiding and supporting us during this project, even when it meant having to sacrifice his lunch breaks. We thank *Stanley Mattsson* and *Sture Larsson* at RUAG Space for their support and input in radiation-related questions.

A thank you is also in order to *Martin Lindskog* at RUAG Space for helping out with hardware and equipment. Furthermore, we thank our manager, *Andreas Karlsson*, for his support and interest in our work, and for the opportunity for us to undertake this thesis project at RUAG Space in the first place. Last but not least, big ups to the whole Digital Research and Development team at RUAG Space for making it a great place for us to do our thesis work.

9 Bibliography

[1] Xilinx, “UG470: 7 Series FPGAs Configuration User Guide,” 2013.

[2] E. Petersen, Single Events Effects in Aerospace, John Wiley & Sons, 2011.

[3] F. Sturesson, S. Mattsson, C. Carmichael and R. Harboe-Sorensen, “Heavy ion characterization of SEU mitigation methods for the Virtex FPGA,” in *European Conference on Radiation and Its Effects on Components and Systems*, 2001.

[4] H. Quinn, P. Graham, K. Morgan, J. Krone, M. Caffrey and M. Wirthlin, “An Introduction to Radiation-Induced Failure Modes and Related Mitigation Methods For Xilinx SRAM FPGAs,” in *International Conference on Engineering of Reconfigurable Systems & Algorithms*, 2008.

[5] C. Yui, G. Swift, C. Carmichael and R. Koga, “SEU mitigation testing of Xilinx Virtex II FPGAs,” in *IEEE Radiation Effects Data Workshop*, 2003.

[6] S. Habinc, Gaisler Research, “Suitability of reprogrammable FPGAs in space applications,” 2002.

[7] J. G. Drobny, Ionizing Radiation and Polymers, Elsevie, 2012.

[8] “Stopping of Ions Heavier than Helium,” *Journal of the ICRU,* vol. 5, no. 1, 2005.

[9] T. Henriksen and D. H. Maillie, Radiation and Health, Taylor & Francis, 2003.

[10] J. H. Trainor, “Instrument and spacecraft faults associated with nuclear radiation in space,” *Advances in Space Research,* vol. 14, no. 10, p. 685–693, 1994.

[11] B. M. Rabin, K. L. Carrihill-Knoll and B. Shukitt-Hale, “Operant responding following exposure to HZE particles and its relationship to particle energy and linear energy transfer,” *Advances in Space Research,* vol. 48, no. 2, pp. 370-377, 2011.

[12] D. G. Lesins, “Atmosphere,” AccessScience, McGraw Hill Education, [Online]. Available: http://www.accessscience.com/content/atmosphere/058800. [Accessed 31 01 2014].

[13] D. P. Riley and D. M. Walt, “Van Allen radiation,” AccessScience, McGraw Hill Education, [Online]. Available: http://www.accessscience.com/content/van-allen-radiation/727110. [Accessed 01 02 2014].

[14] P. Fortescue, G. Swinerd, J. Stark, A. Tatnall, J. Farrow, M. Bandecchi and C. Francis, Spacecraft System Engineering, John Wiley & Sons, 2011.

[15] “CREME96, Critical Charge Method,” [Online]. Available: https://creme.isde.vanderbilt.edu/CREME-MC/help/critical-charge-method. [Accessed 02 02 2014].

[16] C. C. Foster, “Total Ionizing Dose and Displacement-Damage Effects in Microelectronics,” *MRS Bulletin,* vol. 28, no. 2, pp. 136-140, 2003.

[17] Xilinx, “UG190: Virtex 5 User Guide,” 2012.

[18] Xilinx, “UG191: Virtex 5 FPGA Configuration User Guide,” 2012.

[19] Xilinx, “UG193: Virtex-5 FPGA Xtreme DSP Design Considerations,” 2012.

[20] F. Lima Karstensmidt, L. Sterpone, L. Carro and M. Sonza Reorda, “On the Optimal Design of Triple Modular Redundancy Logic for SRAM-based FPGAs,” in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2005.

[21] K. Huang, Y. Hu, X. Li, G. Hua, H. Liu and B. Liu, “Exploiting Free LUT Entries to Mitigate Soft Errors in SRAM-based FPGAs,” in *Asian Test Symposium (ATS)*, 2011 .

[22] L. Sterpone and M. Violante, “A new reliability-oriented place and route algorithm for SRAM-based FPGAs,” *IEEE Transactions on Computers,* vol. 55, no. 6, pp.

732 - 744 , 2006.

[23] A. Sari and M. Psarakis, “Scrubbing-based SEU mitigation approach for Systems-on-Programmable-Chips,” in *International Conference on Field-Programmable Technology (FPT)*, 2011.

[24] K. Morgan, D. McMurtrey, B. Pratt and M. Wirthlin, “A Comparison of TMR With Alternative Fault-Tolerant Design Techniques for FPGAs,” *IEEE Transactions on Nuclear Science,* vol. 54, no. 6, pp. 2065 - 2072 , 2007.

[25] N. Storey, Safety Critical Computer Systems, Harlow: Addison-Wesley, 1996.

[26] H. Quinn, K. Morgan, P. Graham and J. Krone, “Domain Crossing Errors: Limitations on Single Device Triple-Modular Redundancy Circuits in Xilinx FPGAs,” *IEEE Transactions on Nuclear Science,* vol. 54, no. 6, pp. 2037 - 2043 , 2007.

[27] B. Pratt, M. Caffrey, J. F. Carroll, P. Graham, K. Morgan and M. Wirthlin, “Fine-Grain SEU Mitigation for FPGAs Using Partial TMR,” *Nuclear Science,* vol. 55, no. 4, pp. 2274-2280, 2008.

[28] Gaisler Research, “Functional Triple Modular Redundancy (FTMR),” Göteborg, 2002.

[29] E. Kamanu, P. Reddy, K. Hsu and M. Lukowaik, “A new architecture for single-event detection & reconfiguration of SRAM-based FPGAs,” in *High Assurance Systems Engineering Symposium*, Plano, TX, 2007.

[30] X. Wang, “Partitioning Triple Modular Redundancy for Single Event Upset Mitigation in FPGA,” in *E-Product E-Service and E-Entertainment (ICEEE)*, Henan, 2012.

[31] G. L. Smith and L. de la Torre, “Techniques to Enable FPGA Based Reconfigurable Fault Tolerant Space Computing,” in *Aerospace Conference*, Big Sky, MT , 2006.

[32] A. Martin-Ortega, M. Alvarez, S. Esteve, S. Rodriguez and S. Lopez-Buedo, “Radiation Hardening of FPGA-Based SoCs through Self-Reconfiguration and XTMR Techniques,” in *Southern Conference on Programmable Logic*, San Carlos de Bariloche , 2008.

[33] G.-H. Asadi and M. B. Tahoori, “Soft error mitigation for SRAM-based FPGAs,” in *IEEE VLSI Test Symposium*, 2005.

[34] Xilinx, “XAPP987: Single Event Upset Mitigation Selection Guide,” 2008.

[35] Xilinx, “XAPP864: SEU Strategies for Virtex-5 Devices,” 2010.

[36] I. Herrera-Alzu and M. López-Vallejo, “Design Techniques for Xilinx Virtex FPGA Configuration Memory Scrubbers,” *IEEE Transactions on Nuclear Science,* vol. 60, no. 1, pp. 376 - 385, 2013.

[37] J.-Y. Lee, C.-R. Chang, N. Jing, J. Su, S. Wen, R. Wong and L. He, “Heterogeneous Configuration Memory Scrubbing for Soft Error Mitigation in FPGAs,” in *International Conference on Field-Programmable Technology (FPT)*, 2012.

[38] NASA REAG, Melanie Berg, “Xilinx Virtex-Family Scrubbing Methodologies,” 2012.

[39] M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, K. LaBel, M. Friendlich, H. Kim and A. Phan, “Effectiveness of internal vs. external SEU scrubbing mitigation strategies in a Xilinx FPGA: Design, test, and analysis,” in *European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, 2007.

[40] P. Adell, G. Allen, G. Swift and S. McClure, “Assessing and mitigating radiation effects in Xilinx SRAM FPGAs,” in *European Conference on Radiation and Its Effects on Components and Systems (RADECS)* , 2008.

[41] J. Heiner, B. Sellers, M. Wirthlin and J. Kalb, “FPGA Partial Reconfiguration via Configuration Scrubbing,” in *International Conference on Field Programmable*

Logic and Applications (FPL), 2009.

[42] Xilinx, "New Generation Virtex-5 SEU Controller," 2010.

[43] Taft Naegle, S., Burke, G., Newell, M, NASA Jet Propulsion Laboratory, "Fault-Tolerant Coding for State Machines," 2008.

[44] S. Habinc, Gaisler Research, "Functional Triple Modular Redundancy (FTMR)," 2002.

[45] G. Allen, L. Edmonds, T. C.W., G. Swift and C. Carmichael, "Single-Event Upset (SEU) Results of Embedded Error Detect and Correct Enabled Block Random Access Memory (Block RAM) Within the Xilinx XQR5VFX130," *IEEE Transactions on Nuclear Science,* vol. 57, no. 6, 2010.

[46] M. Sonza Reorda, M. Violante, C. Meinhardt and R. Reis, "A low-cost SEE mitigation solution for soft-processors embedded in systems on programmable chips," in *IEEE/ACM Design Automation & Test in Europe (DATE)*, 2009.

[47] S. Punnekkat, A. Burns and R. Davis, "Analysis of Checkpointing for Real-Time Systems," *Real-Time Systems,* vol. 20, no. 1, pp. 83-102, 2001.

[48] A. Sari, M. Psarakis and D. Gizopoulos, "Combining Checkpointing and Scrubbing in FPGA-based Real-Time Systems," in *IEEE VLSI Test Symposium (VTS)*, 2013.

[49] H. Zarandi, S. Miremadi, D. Pradhan and J. Mathew, "SEU-Mitigation Placement and Routing Algorithms and Their Impact in SRAM-based FPGAs," in *International Symposium on Quality Electronics Design (ISQED)*, 2007.

[50] W. Xu, J. Wang, Y. Hu and J.-Y. Lee, "In-Place FPGA Retiming for Mitigation of Variational Single-Event Transient Faults," *IEEE Transactions on Circuits and Systems,* vol. 58, no. 6, pp. 1372 - 1381 , 2011.

[51] A. Das, S. Venkataraman and A. Kumar, "Improving autonomous soft-error tolerance of FPGA through LUT configuration bit manipulation," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2013.

[52] V. Betz, J. Rose and A. Marquardt, Architecture and CAD for Deep-Submicron FPGAs, Kluwer Academic Publishers, 1999.

[53] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," in *International Workshop on Field Programmable Logic and Applications*, 1997.

[54] J.-Y. Lee, Y. Hu, R. Majumdar, L. He and M. Li, "Fault-tolerant resynthesis with dual-output LUTs," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010.

[55] M. Alderighi, F. Casini, S. d'Angelo and M. Mancini, "Evaluation of Single Event Upset Mitigation Schemes for SRAM based FPGAs using the FLIPPER Fault Injection Platform," in *IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, 2007 .

[56] AvNet, "Xilinx Virtex-5 LX Evaluation Kit User Guide (Rev 1.5)," 2009.

[57] National Institute of Standards and Technology, "Specification of the Advanced Encryption Standard. FIPS PUB 197," 2001.

[58] J. Heiner, N. Collins and M. Wirthlin, "Fault Tolerant ICAP Controller for High-Reliable Internal Scrubbing," *IEEE Aerospace Conference,* pp. 1-10, 2008.

[59] Xilinx, "UG632: PlanAhead User Guide (v14.1)," 2012.

[60] E. Abolhassani Ghazaani, Z. Ghaderi and S. Ghassem Miremadi, "A Non-instrusive Portable Fault Injection Framework to Assess Reliability of FPGA-based Designs," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013.

[61] M. Straka, J. Kastil and Z. Kotasek, "SEU Simulation Framework for Xilinx FPGA: First Step Towards Testing Fault Tolerant Systems," in *14th Euromicro Conference*

on Digital System Design, 2011.

[62] Xilinx, “UG628: Command Line Tools User Guide (v13.1),” 2011.

[63] Xilinx, “UG612: Timing Closure User Guide (v13.4),” 2012.

[64] Xilinx, “DS192: Radiation-Hardened, Space-Grade Virtex-5QV Family Overview,” 2012.

[65] G. Swift (Xilinx Inc.), G. Allen (Jet Propulsion Laboratory), “Virtex-5QV Static SEU Characterization Summary,” 2013.

[66] G. Allen, L. Edmonds, C. W. Tseng, G. Swift and C. Carmichael, “Single-Event Upset (SEU) Results of Embedded Error Detect and Correct Enabled Block Random Access Memory (Block RAM) Within the Xilinx XQR5VFX130,” *IEEE Transactions on Nuclear Science,* vol. 57, no. 6, pp. 3426 - 3431 , 2012.

[67] H. Quinn, G. Allen, G. Swift, C. W. Tseng, P. Graham, K. Morgan and P. Ostler, “SEU-Susceptibility of Logical Constants in Xilinx FPGA Designs,” *IEEE Transactions on Nuclear Science,* vol. 56, no. 6, pp. 3527-3533, 2009.