NELSON MANDELA
UNIVERSITY

# CHARACTERISATION OF SINGLE EVENT EFFECTS AND TOTAL IONISING DOSE EFFECTS OF AN INTEL ATOM MICROPROCESSOR

BY
MUEMA MALINDA

Submitted in fulfilment of the requirements for the degree of

Master of Engineering (MEng) in Mechatronics

In the Faculty of Engineering, the Built Environment and Information Technology at the

Nelson Mandela University

December, 2019

Supervisor: Prof Farouk Smith, PhD
Port Elizabeth, South Africa

*The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF*

NELSON MANDELA
UNIVERSITY

## DECLARATION BY CANDIDATE

**NAME:** MUEMA MALINDA

**STUDENT NUMBER:** 212213903

**QUALIFICATION:** MEng Mechatronics

**TITLE OF PROJECT:** CHARACTERISATION OF SINGLE EVENT EFFECTS AND TOTAL IONISING DOSE EFFECTS OF AN INTEL ATOM MICROPROCESSOR

**DECLARATION:**

In accordance with Rule G5.6.3, I hereby declare that the above-mentioned treatise/ dissertation/ thesis is my own work and that it has not previously been submitted for assessment to another University or for another qualification.

**SIGNATURE:** 

**DATE:** 1/11/2019

# Abstract

The rapid advancements of COTS microprocessors compared to radiation hardened microprocessors has attracted the interest of system designers within the aerospace sector. COTS microprocessors offer higher performance with lower energy requirements, both of which are desired characteristics for microprocessors used in spacecraft. COTS microprocessors, however, are much more susceptible to radiation damage therefore their SEE and TID responses needs to be evaluated before they can be incorporated into spacecraft. This thesis presents the process followed to evaluate said characteristics of a COTS Intel Atom E3815 microprocessor mounted on a DE3815TYBE single board PC.

Evaluation of the SEE response was carried out at NRF iThemba Labs in Cape Town, South Africa where the device was irradiated by a proton beam at 55.58 MeV and with varying beam currents. The device showed a higher sensitivity to functional interrupts when running with the onboard cache on compared to when running with the cache off, as would be expected. The cross-sections, respectively, are: $4.5x\ 10^{-10}\ cm^2$ and $2.8\ x\ 10^{-10}\ cm^2$.

TID testing on the other hand was carried out at the irradiation chamber of FruitFly Africa in Stellenbosch, South Africa. The test device was irradiated by gamma radiation from a Cobalt-60 source at a dose rate of 9.7kRad/h and to a total dose of 67.25kRad. Noticeable TID degradation, in the form of leakage currents, was observed once a total dose of about 20kRad was absorbed. The device then completely failed once a total dose of approximately 32kRad was absorbed.

These results suggest that the E3815 microprocessor would not be suitable for long term missions that require higher TID survivability. The processor could however be considered for short term missions launched into polar or high incline orbits where the dose rate is relatively low, and the mission is capable of tolerating functional interrupts.

# Key Words

Intel, Atom, Microprocessor, E3815, SEU, TID, COTS

# Acknowledgments


- I would like to thank my supervisor Prof. Farouk Smith for introducing me to this field of research and for his guidance and support throughout the duration of this project.
- I would also like to thank Mr. Arno Barnard for his assistance and invaluable advice and guidance in setting up and carrying out the experiments, as well as data acquisition.
- I am deeply indebted to the staff of iThemba Labs: Mr Jaime Nieto-Camero, Dr Retief Neveling, Dr Ricky Smit and all others involved. Without their willingness to offer their assistance, it would not be possible to carry out tests at the facility.
- Mr Victor Sciocatti for his helping hand during test setup and for logging events during the testing. This is appreciated.
- Acknowledgment to Mr. Jerome Johnson, Mr. Nathan Vermeulen and Mr. Shorn Fortuin of FruitFly Africa for quickly making arrangements to allow access to the gamma radiation facility on short notice.
- I am forever grateful to my parents and sister for their support and encouragement whenever I needed it most throughout this project.
- Gratitude is expressed to the NRF and NMU RCD for the financial support provided.

# Table of Contents

# List of Figures

# List of Tables



# List of Equations

# List of Acronyms

- ALU – Arithmetic and Logic Unit
- BJT – Bipolar Junction Transistor
- BLM – Beam Loss Monitor
- cDAQ – CompactDAQ
- CMOS – Complementary Metal Oxide Semiconductor
- COTS – Commercial Off-The-Shelf
- CPU – Central Processing Unit
- DRAM – Dynamic Random Access Memory
- DUT – Device Under Test
- ECC – Error-Correcting Code
- FinFET – Fin Field Effect Transistor
- FPGA – Field Programmable Gate Array
- FPU – Floating Point Unit
- FSM – Finite State Machine
- GCR – Galactic Cosmic Rays
- GPR – General-Purpose Register
- IC – Integrated Circuit
- LET – Linear Energy Transfer
- MBU – Multiple Bit Upsets
- MMX – MultiMedia eXtensions
- MOS – Metal Oxide Semiconductor
- MOSFET – Metal Oxide Semiconductor Field Effect Transistor
- NMOS – n-channel MOSFET
- OS – Operating System
- PMOS – p-channel MOSFET
- RAM – Random Access Memory
- RH – Radiation Hardened
- SATA – Serial AT Attachment
- SCR – Solar Cosmic Rays
- SEB – Single Event Burnout
- SEE – Single Event Effect
- SEFI – Single Event Functional Interrupt
- SEGR – Single Event Gate Rapture
- SEL – Single Event Latchup
- SET – Single Event Transient
- SEU – Single Event Upset
- SIMD – Single-Instruction, Multiple-Data
- SoC – System-on-a-Chip
- SO-DIMM – Small Outline Dual In-line Memory Module
- SOI – Silicon on Insulator
- SRAM – Static Random Access Memory

- SSE – Streaming SIMD Extensions
- SSD – Solid State Drive
- SSH – Secure Shell
- STI – Shallow Trench Isolation
- TID – Total Ionizing Dose
- TTL – Transistor-transistor Logic
- VI – Virtual Instrument

# 1. Introduction

With an ever-increasing demand for more computational power and reduced energy requirements by the aerospace sector, it is necessary to investigate the viability of solutions from different sectors and incorporate them if found to be advantageous. An example of this can be seen in the push to incorporate Commercial Off-The-Shelf (COTS) components such as microprocessors and Field Programmable Gate Arrays (FPGA) into equipment destined for missions in outer space [1].

This push to incorporate COTS components over specially designed hardware is motivated by a number of reasons, one of the main ones being that COTS components tend to cost less. This opens up space missions to developing countries and academic institutions that may lack the financial means to support entire part design programs [2, 3]. COTS components also tend to be more advanced in terms of generation and processing power. This is because there is higher demand and more pressure to innovate given the larger market.

Despite having these advantages, COTS components happen to be more susceptible to the negative effects brought about by exposure to radiation present in outer space. This is because it is not a requirement in their design for them to withstand it. Conversely, specially designed hardware will usually be radiation hardened (RH) and for this reason, the use of COTS components in space is limited to missions that do not require high reliability or long-term use [1].

With this in mind, information pertaining to the performance of specific COTS devices is of great value to a system designer. This is because it will facilitate decision making and lead to the design of missions that have higher probabilities of success. To this end, this thesis seeks to contribute by adding to the pool of knowledge of the performance of COTS microprocessors in high radiation environments. This shall be accomplished by characterising the Single Event Effects (SEE) and Total Ionizing Dose (TID)[1] effects on an Intel Atom E3815 microprocessor.

A processor from the Intel Atom family was chosen because the family consists of relatively cheap microprocessors that are mainly used in industrial computers and embedded systems. These processors have low power requirements which are in the range of several watts. This is advantageous in use cases such as CubeSats that have limited storage space for batteries and solar panels. Additionally, atom processors are capable of carrying out computationally complex and extensive tasks (the newest even having 64-bit instruction sets) and many of them are available as Systems-on-a-Chip with additional hardware included on the same chip as the processor. This allows for a large number of features to be packed into a satellite while keeping the internal volume that has been consumed by the device at a minimum, which is a desirable characteristic in satellite systems [4].

[1] SEE and TID are defined in Chapter 2

Atom processors are also compatible with many standard peripheral devices and communication protocols. This means that all it would take to interface such a processor with a peripheral device, say, a camera, would be a USB connection. This is advantageous since it allows for rapid system design. Despite this, there is limited literature on the performance of these processors in high radiation environments, possibly because testing has not been able to keep up with the rate of intergenerational improvements. This scarcity of literature is the main factor that contributed to the selection of an Intel Atom microprocessor for this research.

As a consequence of the choice of processor, it is important to state that the focus in this text will be on bulk substrate devices rather than Silicon on Insulator (SOI) which is known to be more resistant to SEEs [5]. This is because Intel manufactures processors on bulk substrate and not SOI [6].

## 1.1. Objectives

In order to achieve the main objective that is characterizing the SEE and TID effects of an Intel Atom E3815 microprocessor, a few secondary objectives will have to be completed. These are listed below:

- Literature Review. Before any experiments may be carried out it is necessary for there to be a clear understanding of the intricate details of:
    - Research that has already been carried out in order to avoid a repetition of the very same work.
    - The operation of microprocessors down to the logic gate level. This will prove useful in getting to understand how SEE and TID effects take place.
    - Radiation. This is a broad topic however focus should be given to the radiation environment in space.
    - SEE and TID. Building up on the just discussed sub-objectives (above this), a better understanding of how SEE and TID effects occur would be crucial.

- Learning how to program/interface the microprocessor. This will be necessary since experiments carried out to characterize SEE and TID effects rely on custom programs to run on the device and for the operational state (and current draw) of the device to be monitored.

- Designing the Experiments. The experiments will have to be compatible with the facilities and equipment available to carry them out. The custom programs to run on the device under test (DUT) will also have to be determined.

- Carrying out the Experiments and Collecting Data.

- Analysing and Interpretation of Data. At this stage the data collected will be used to characterize the device under test. Plots will also be created that show the operating current draw of the device in relation to the absorbed radiation dose (this is for TID effects).

- Completion of Thesis. Throughout the study period, a thesis detailing the events and findings of the study will be compiled.

## 1.2. Thesis Outline

This section provides a brief description of each subsequent chapter of the thesis.

**Chapter 2.** *Literature Background:* This chapter provides an overall view of the conditions in which spacecraft operate. A discussion on the general-purpose microprocessor is also given and the chapter ends with a description of the various ways in which radiation interacts with electronics.

**Chapter 3.** *Test Setup and Procedure:* This chapter provides a detailed description of the Intel Atom E3815 processor as well as the board used to run it. The experimental setups employed to investigate the SEE and TID characteristics of the processor are also described, together with considerations taken into account while designing the experiments. Algorithms of the test programs developed as well as photos of the test setups are provided.

**Chapter 4.** *Results:* This chapter presents and analyses the data obtained from the experiments described in Chapter 3.

**Chapter 5.** *Discussion and Conclusions:* Here, interpretations are provided for the results obtained and recommendations are given for future testing.

# 2. Literature Background

To understand the nature of the effects that radiation has on electronics, it is important to have a good understanding of certain fundamental definitions and concepts. These range from the operational basics of electronic devices to the characteristics of the space environment the devices are expected to be commissioned in.

This chapter gives an overview of the aforementioned and ends with a discussion on the interaction between radiation and electronic devices.

## 2.1. Radiation

Radiation is defined as energy, in the form of waves or particles, travelling in space from a given source [7]. Radiation can either be ionizing or non-ionizing. Ionizing radiation consists of electromagnetic waves or particles that carry enough energy to knock electrons out of their atomic orbitals or break molecular bonds thus creating ions [8]. Non-ionizing radiation on the other hand does not carry enough energy to cause ionization.

Ionizing radiation is of particular interest when it comes to the interaction between electronics and radiation since ionization of the electronic components does introduce implications to the performance of the device. Ionizing radiation comes in several forms which shall now be discussed.

### 2.1.1. Different types of Ionizing Radiation

#### 2.1.1.1. Alpha Particles (α)

These are helium nuclei consisting of 2 protons and 2 neutrons and carry a net positive charge of 2 units. Alpha particles typically have a range of a few centimetres in air and are heavily ionizing [9]. These particles may be produced as a by-product of nuclear fusion of hydrogen in the sun (and other stars) or by radioactive decay of heavier nuclides such as Americium [10].

#### 2.1.1.2. Beta Particles (β)

These consist of electrons and positrons and carry a net negative charge (in the case of electrons) or net positive charge (in the case of positrons) of 1. Beta particles have longer ranges in air than alpha particles due to the combined effect of their relatively lesser charge and relatively higher velocities. They are therefore lightly ionizing [9]. Radioactive decay of unstable nuclides is the main source of these particles.

#### 2.1.1.3. Protons

Also referred to as hydrogen nuclei, protons carry a net positive charge of 1 and have a range of a few centimetres in air [9]. There are no naturally occurring nuclides on earth that decompose to give proton radiation. However, in the terrestrial space environment, protons may come from cosmic rays and may be found trapped in the magnetic field of the earth [9, 11]. Protons are heavily ionizing [3].

### 2.1.1.4. Neutrons

Unlike the prior discussed types of radiation, neutrons carry no charge and therefore cause ionization indirectly. Such would take place where an atomic nucleus captures a neutron and enters an excited state, after which it would de-excite by emitting a gamma ray [9, 3].

There are no naturally occurring sources of neutrons on earth, however a method employed to obtain them is by combining a nuclide that emits alpha particles with a suitable target material [10].

### 2.1.1.5. Electromagnetic Radiation

In the context of *ionizing radiation*, these are high frequency, short-wavelength electromagnetic waves that are lightly ionizing and have very long ranges in air. They consist of X-Rays and Gamma (γ) Rays which are very similar but differ in how they are produced [9].

**X-Rays –** These are emitted when an electron shifts from a higher energy level to a lower energy level within an atom. The difference in energy between the two energy levels dictate the amount of energy emitted which in turn dictates the type of electromagnetic radiation that is emitted [12]. If this energy is high enough, the emitted electromagnetic wave is an X-Ray.

**Gamma (γ) Rays –** These can be produced in a number of ways, for instance, by nuclides undergoing radioactive decomposition or when matter and antimatter interact with and annihilate each other [13]. Gamma rays may also be produced as a by-product of nuclear fusion as is found in the core of the sun [14].

## 2.1.2. Units of Measurement

### 2.1.2.1. Radioactivity

The SI Unit of radioactivity is the **becquerel (Bq)** which is defined as the activity of a quantity of a radioactive material in which one nucleus disintegrates per second. A different unit for radioactivity that may be used is the **curie (Ci)** which is defined as the activity in 1g of Radium-226 which is equivalent to $3.7\ x\ 10^{10}$ disintegrations per second. Therefore 1 Ci = $3.7\ x\ 10^{10}$ Bq.

### 2.1.2.2. Energy

The **joule (J)** is the SI Unit for energy however when dealing with particles associated with radioactivity, the **electron volt (eV)** is commonly used. An Electron Volt is defined as the energy gained by an electron that is accelerated through a potential difference of 1 volt. The electron volt can be thought of as a measure of the kinetic energy possessed by a particle and 1eV = $1.602\ x\ 10^{-19}$J. Additionally, the **erg** (1 erg = $10^{-7}$J) is a unit that may be used.

#### 2.1.2.3. Linear Energy Transfer (LET)

This is defined as the energy deposited by an ionizing particle into the material that it is traversing through, per unit length. LET is usually expressed in MeV/cm. LET may also be referred to as "Stopping power" [15].

#### 2.1.2.4. Absorbed Dose

The **gray (Gy)** is the SI Unit of energy deposition into a material by radiation and is defined such that 1Gy is equal to the absorption of 1J of energy by 1kg of said material. Also, commonly used is the **rad** which is defined such that 1rad is equal to 100 ergs absorbed per gram of material. Using this relationship, it can be shown that 100 rad = 1Gy.

#### 2.1.2.5. Flux

This is simply defined as the number of particles passing through some defined cross-sectional area per unit time. This is usually expressed in the units $cm^{-2}s^{-1}$.

#### 2.1.2.6. Fluence

Defined as the time integrated flux of particles and is expressed in the units $cm^{-2}$.

### 2.1.3. Sources of Radiation in Terrestrial Space

The terrestrial space radiation environment consists of particles trapped in the magnetic field of the earth as well as cosmic rays. These particles vary in flux and energy at different altitudes and inclinations (or latitudes) which in turn dictate the effects that will be experienced by orbiting spacecraft.

#### 2.1.3.1. Cosmic Rays

Cosmic rays are an ever present low-flux component of the terrestrial space radiation environment and are the main component in interplanetary space. They consist of galactic cosmic rays (GCR), solar cosmic rays and terrestrial cosmic rays. The composition of galactic cosmic rays is approximately 85% protons, 14% alpha particles and 1% heavy nuclei, all of which come from outside the solar system [9]. During solar maximum (periods of high solar activity), solar wind reduces the flux of galactic cosmic rays and conversely, during solar minimum, galactic cosmic rays are at their maximum flux [16].

Solar cosmic rays (SCR) on the other hand originate from the sun and have slightly different compositions compared to their galactic counterparts. During large solar flares that make solar cosmic rays the dominant cosmic rays momentarily (in terms of total flux), SCR heavy nuclei flux will still be much smaller than GCR heavy nuclei flux [16]. Terrestrial cosmic rays are resultant secondary cosmic rays that arise from the interaction between the atmosphere and the prior discussed galactic and solar cosmic rays.

The magnetic field of the earth offers geomagnetic shielding from cosmic rays by action of deflecting incoming charged particles. The energy and momentum of a particle determines how much it will penetrate into the magnetic field and this is usually quantified as the magnetic rigidity of the particle calculated as:

$$magnetic\ rigidity = \frac{Momentum\ of\ the\ particle}{charge\ of\ the\ particle} \qquad \text{Eqn 2.1}$$

Each point in the magnetic field of the earth requires an incoming particle to have a minimum value of magnetic rigidity, referred to as "geomagnetic cut off", for the particle to be able to reach [9]. This cut off value tends to increase with reduction in altitude as can be seen in Figure 2.1.

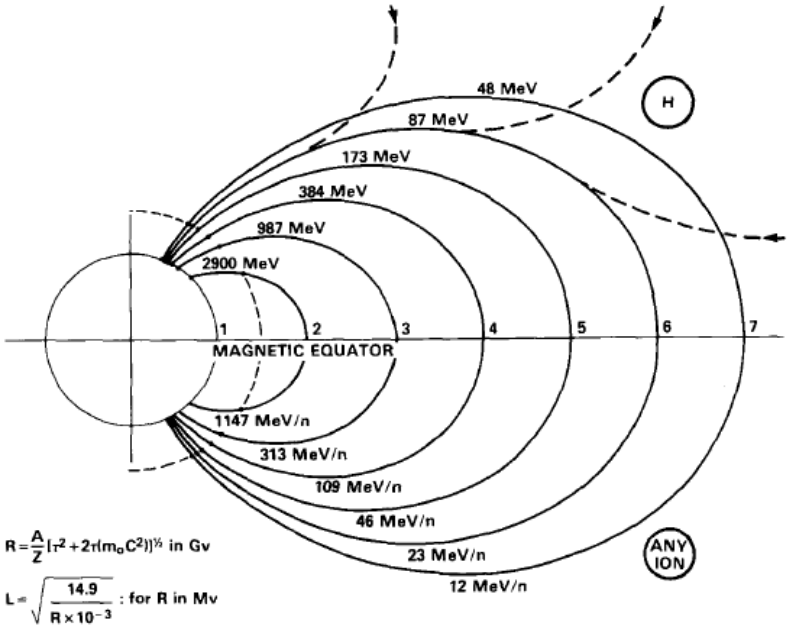



*Figure 2.1: Total energy required to penetrate the magnetosphere at different altitudes (measured in earth radii) [11]*

Additionally, the geomagnetic cut off value varies with inclination such that low incline orbits have higher geomagnetic cut-offs than high incline orbits. Consequently, geomagnetic cut off falls to zero at the edges of the magnetosphere (very high altitude) and at the magnetic poles of the earth (maximum inclination). This means that spacecraft in earth orbit are protected from cosmic rays, whose highest fluxes are of low energy particles, except spacecraft in polar orbits and geostationary orbits [9].

It is also important to remember that for high energy particles (>100GeV/nucleon) from galactic cosmic rays, both geomagnetic shielding and spacecraft shielding are relatively ineffective [16]. Cosmic rays in general contribute more towards Single Event Effects (SEE) than Total Ionizing Dose (TID) effects [17], both of which shall be discussed later.

### 2.1.3.2. Van Allen Belts

The magnetic field of the earth traps electrons, protons and some heavy ions that originate from solar wind. These particles spiral about the "closed loops" of the magnetic dipole of the earth and move back and forth between regions of maximum magnetic field strength. The electrons drift west to east while the protons drift east to west [11]. The motion of

these particles around the earth form domains which are referred to as the radiation belts or Van Allen belts [16] and is illustrated in Figure 2.2.

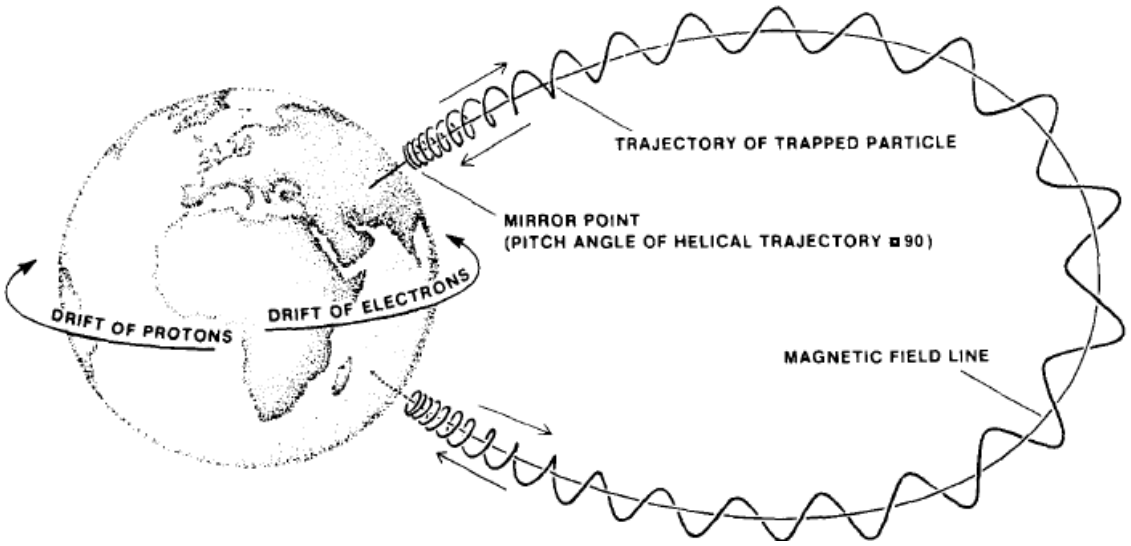



*Figure 2.2: Motion of particles trapped by the magnetic field of the earth. [11]*

The trapped electrons occupy two zones, namely the inner zone and the outer zone. The inner zone extends to an altitude of about 2.4 earth radii from the equator while the outer zone extends from an altitude of 2.8 to about 12 earth radii [18]. The region between the two zones (2.5 to 2.8 earth radii) is referred to as the slot. The electron density in the slot is usually low, however, may increase by a few orders of magnitude during magnetic storms [11].

The electron flux is lower in the inner zone compared to the outer zone and electron energies in this zone peak at around 5MeV. Comparatively, the electrons in the outer zone have energies that peak at around 7MeV [11].

Unlike electrons, protons cannot be assigned to inner and outer zones since they have energies that decrease monotonically with increase in altitude up to a trapping boundary at 3.8 earth radii [11]. Trapped protons may have energies as high as 500MeV with peak fluxes for the most energetic particles occurring at relatively low altitudes [16]. This variation is opposite to that of trapped electrons where the most energetic electrons are found at higher altitudes.

The South Atlantic Anomaly is also worth mentioning. It is as a result of the offset of earth's magnetic dipole to the axis of rotation by approximately 11°, with a displacement towards the western pacific. This causes a dip in the magnetic field which causes the radiation belts to reach lower altitudes over the coast of Brazil [3, 11]. The anomaly is responsible for most of the radiation absorbed by spacecraft in low earth orbit [11]. Figure 2.3 shows the effects of the anomaly where the orbiting spacecraft experiences radiation induced upsets over

the coast of Brazil at lower altitudes and only starts experiencing upsets at different longitudes with an increase in altitude. Refer to section 2.3 for a discussion on upsets.

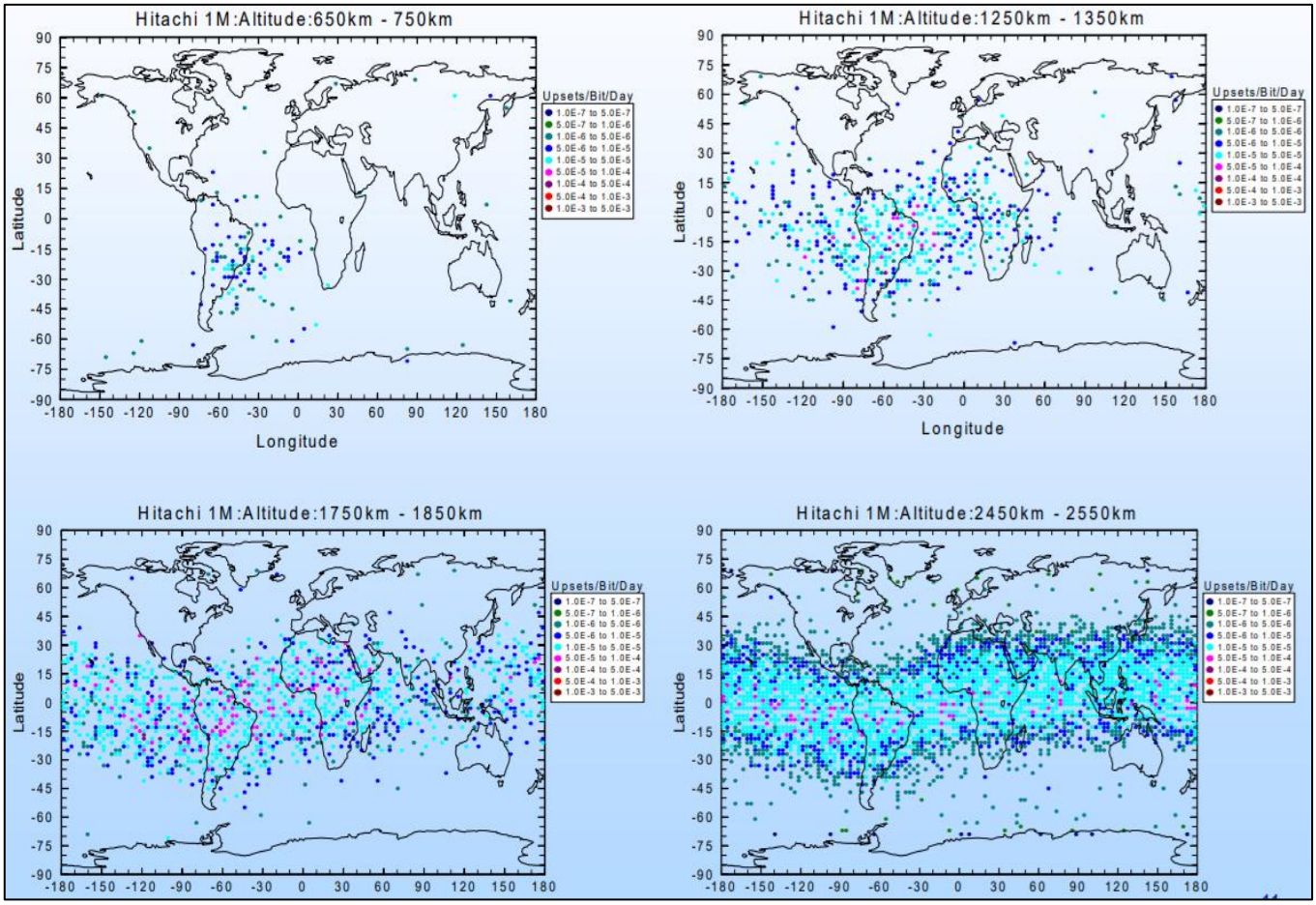



*Figure 2.3: Effects of the Asymmetry in the Proton Belts on SRAM Upset Rate at Varying Altitudes on CRUX/APEX [29]*

Cyclic variations in solar activity also have an effect on the fluxes of particles within the radiation belts. During periods of maximum solar activity, referred to as solar max, the flux of electrons is seen to increase while that of protons is seen to decrease. Conversely, during periods of minimum solar activity, the flux of protons is seen to increase while that of electrons is seen to decrease [16].

Two models have been created to provide estimates for the fluxes of each particle at different orbits. Namely, they are the AE8 and AP8 models which estimate electron and proton fluxes respectively. These models allow for long term averaged predications of particle fluxes meaning transient variations have been averaged out [11]. Figure 2.4 illustrates predictions made by both models at different altitudes in earth radii. Electron fluxes are estimated for electrons with energies higher than 1 MeV (right hand side of the figure) while proton fluxes have been estimated for protons with energies larger than 10 MeV (left hand side of the figure).

Exposure to protons and heavy ions on spacecraft is of concern since these are the main cause of single event effects. Both protons and electrons should however be considered when it comes to total dose absorbed by spacecraft in orbit [11].

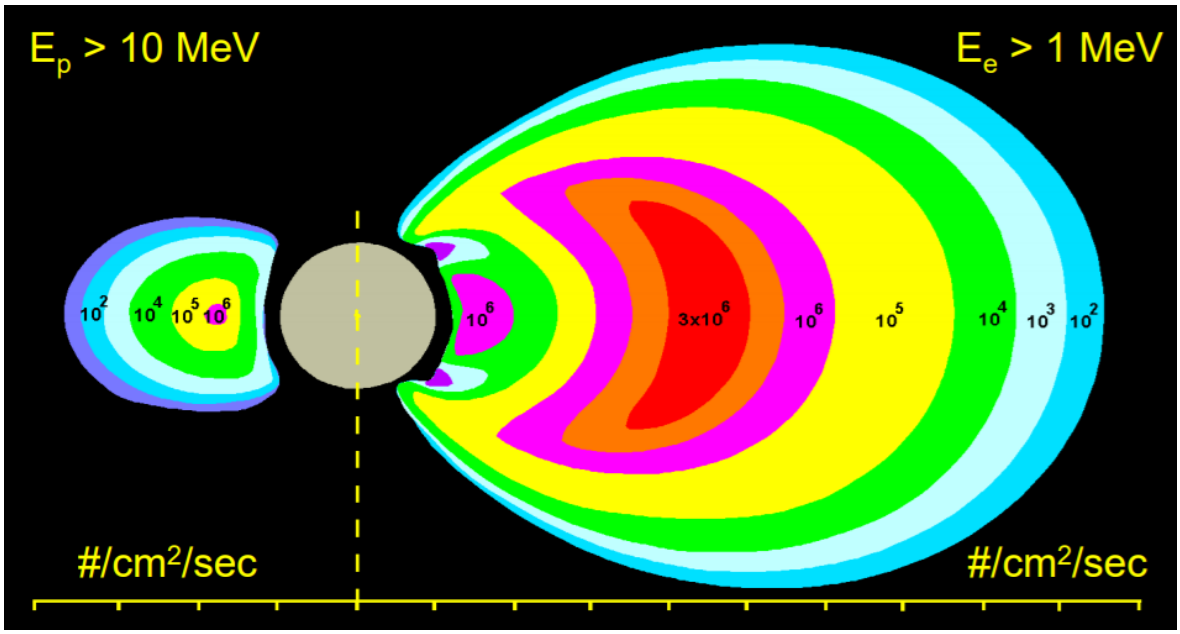



*Figure 2.4: Trapped Particles in the Earth's Magnetic Field: Proton & Electron Intensities [29]*

## 2.2. The Microprocessor

A microprocessor is a complex integrated circuit that executes instructions and performs various control tasks or calculations in computers and smart devices. The two main types of microprocessors are general purpose microprocessors and dedicated microprocessors. Dedicated microprocessors perform specific tasks and cannot perform any different types of tasks. In this text, focus will mainly be on general purpose microprocessors that can perform a multitude of tasks, each different from one another. General purpose microprocessors achieve this by operating under the control of software instructions.

As with microprocessors and the majority of modern integrated circuits, the MOSFET (Metal Oxide Semiconductor Field Effect Transistor) forms the fundamental electronic component with which these circuits are built [19]. These are advantageous over alternative technologies such as BJTs (Bipolar Junction Transistors) mainly because they are switched by voltage rather than current and can be shrunk to smaller sizes [20]. The combined effects of these lead to smaller sized electronics with lower power requirements. The structure and operation of MOSFETs shall now be discussed.

### 2.2.1. MOSFET

The MOSFET is a 3-terminal voltage-controlled switch. The 3 terminals are namely the source, gate and drain. The gate may be a metal layer (older technologies) or a high-conductivity polycrystalline silicon layer (newer technologies) [20, 21] and is deposited on an insulating layer of silicon oxide. In newer technologies, the silicon oxide is replaced by a dielectric material that has a higher relative permittivity (i.e. high-k dielectric) [15]. This insulating layer in turn separates the gate from the substrate which consists of doped silicon. Two regions on either end of and below the insulating oxide are doped with

impurities of opposite charge to the substrate doping. The surfaces of these two regions also have conductive material laid thus forming the source and drain terminals. Figure 2.5 illustrates the physical structure of a MOSFET.

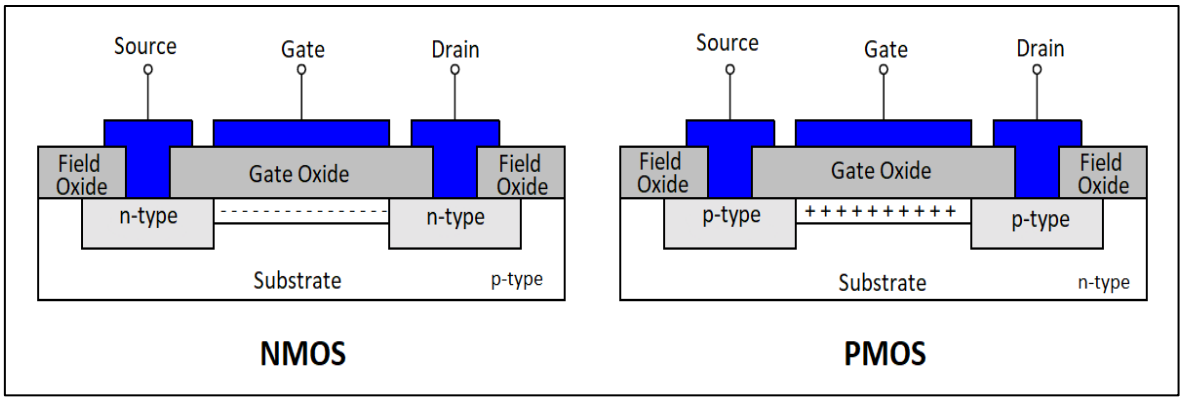



*Figure 2.5: Cross section showing the physical structure of NMOS and PMOS. Image adapted from [19]*

As can be seen in the figure, the charges of the doping impurities determine the type of MOSFET. An NMOS (n-type MOSFET) device has p-type substrate and embedded n-type doping at the drain and source. The PMOS (p-type MOSFET) is the exact opposite of this with a n-type substrate and p-type doping at the drain and source.

The operating principle of an ideal NMOS is as follows. If a positive voltage is applied at the gate, the electric field will pass through the gate oxide and repel the majority carriers (p-type) of the substrate. Consequently, a negative charge will begin to accumulate at the interface between the substrate and the gate oxide. If the applied gate voltage is large enough, minority carriers (electrons) will be attracted to the substrate-oxide interface and will form an inversion layer. This inversion layer is highlighted in Figure 2.5 by the minus sign (-) for NMOS. The voltage at which this inversion layer is formed is referred to as Threshold Voltage ($V_{TH}$).

At this point, the transistor is said to have been switched on. This is because current will flow between the source and drain, via the channel that is the inversion layer, given that a potential difference is applied between the two terminals. In the case of NMOS, the charge carriers for this current are electrons and will flow from the source terminal to the drain terminal.

If a negative voltage is applied at the gate, no inversion layer is formed and the source and drain terminals remain electrically isolated from each other. At this point the transistor is said to have been switched off. The process is identical for PMOS with the only difference being that of opposite charges. Therefore, a PMOS device is switched on by negative gate voltage and so on.

The MOSFETS that have been described are referred to as Enhancement mode which means that voltage has to be applied to their gates to increase conductivity. Depletion mode devices on the other hand have their conductivity decrease with voltage applied to their gates. Another way of thinking of the difference between the two is that

enhancement mode devices are normally off while depletion mode devices are normally on [21].

#### 2.2.1.1. FinFET

The Fin-Field Effect Transistor (FinFET) is a MOSFET with a 3-dimensional, multi-gate structure that is different to that of planar MOSFETS i.e. those MOSFETS described in the previous section. FinFETs may also be referred to as 3D or Tri-Gate transistors [22, 23].

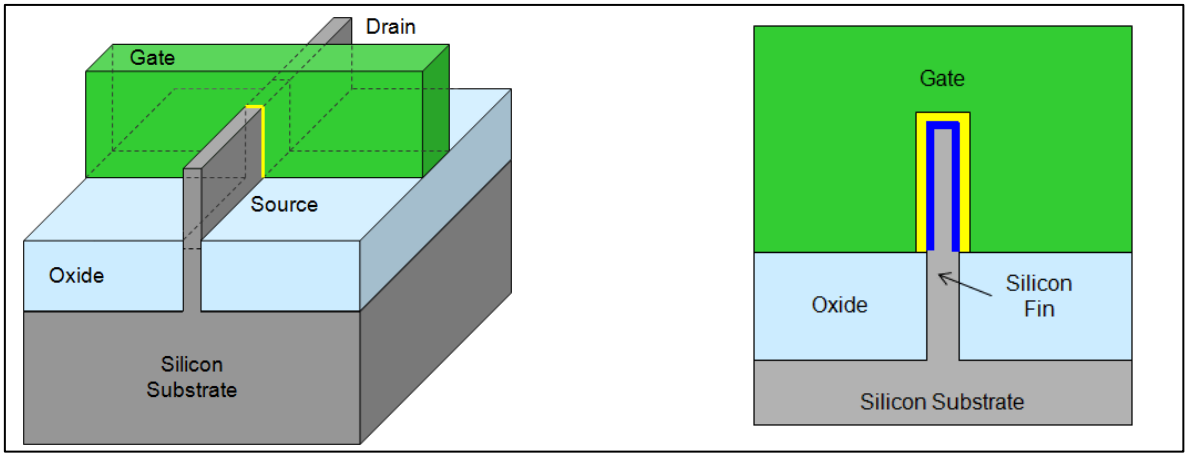



*Figure 2.6: Left – 3D structure of a FinFET. Right - Cross section view of a FinFET [6]*

Referring to Figure 2.6, the FinFET has a vertical channel that extends up from the silicon substrate. This channel forms a "fin" that gives the transistor its name and different variations of FinFETs may have more than one fin per transistor. Wrapped around the fin, on 3 faces, is the gate as well as a dielectric material that prevents direct contact between the two. The source and drain of the transistor are on either end of the fin. Trenches of oxide are present on the substrate to provide electrical isolation from adjacent transistors (i.e. field oxide). These oxide trenches may also be referred to as Shallow Trench Isolation (STI) [24].

The operation of a FinFET is similar to that of a planar MOSFET with the added advantage of having better control of the inversion layer within the channel. This stems from the fact that the gate now induces inversion from 3 directions rather than 1. Additional advantages that FinFETs have over planar MOSFETS are higher drive currents and lower leakage currents which allow for scaling of the transistors beyond 22nm [23, 25]. These, among other advantages, have motivated a large-scale shift from planar to 3-dimensional technologies within the electronics sector in recent times.

### 2.2.2. CMOS

In CMOS (Complementary Metal Oxide Semiconductor), NMOS and PMOS (either planar or FinFET) are combined on the same circuit and each type of transistor is used to represent different logic states [19]. For instance, the PMOS transistors may be used to output logic 1 while NMOS outputs logic 0. NMOS and PMOS therefore work in complement to each other in CMOS [19].

CMOS is advantageous over purely n-type or purely p-type circuitry for a number of reasons, some of which include: Lower power consumption, higher circuit density and the ability to combine analog and digital circuitry on the same chip [21].

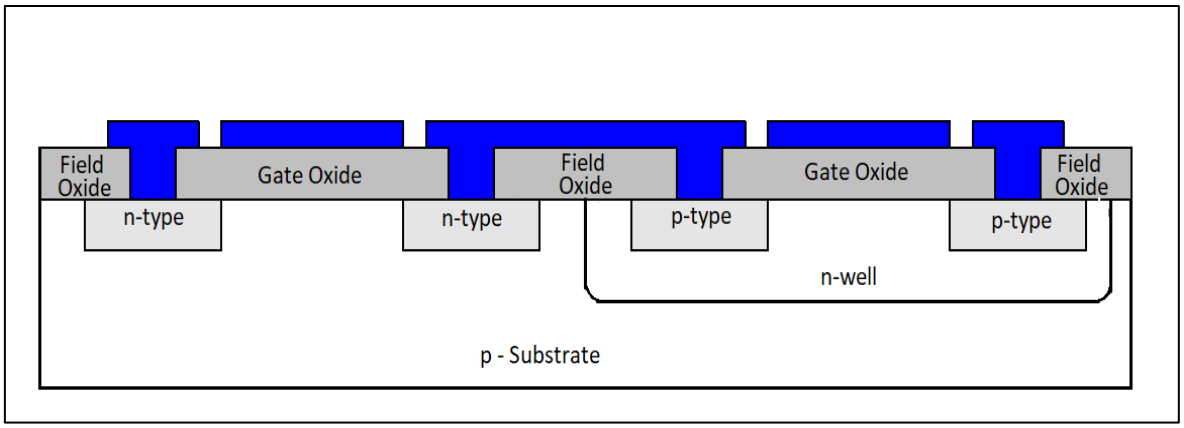



*Figure 2.7: Cross section of NMOS and PMOS fabricated with n-well CMOS technology. Image adapted from [19]*

## 2.2.3. Combinational and Sequential Circuits

Combinational circuits are those whose outputs depend solely on the current inputs to the circuit. Standard logic gates are combined to form the function of the circuit and a change to the input signal(s) will result in an immediate change to the output signal(s) [19]. Examples of combinational circuits include adders, multipliers, multiplexers, comparators, shifters etc.

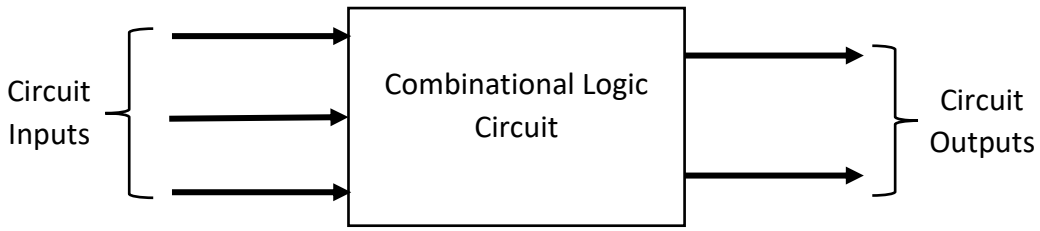
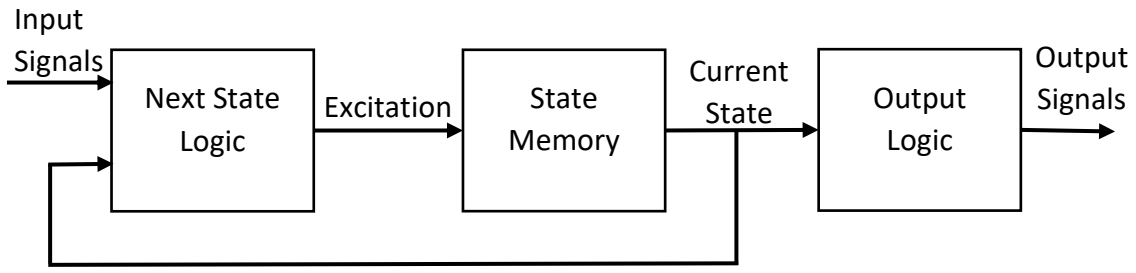



*Figure 2.8: Combinational Logic*

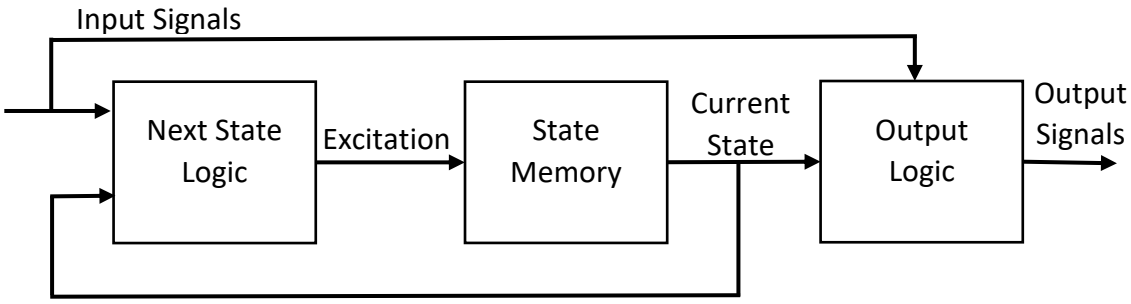
Sequential circuits, also known as Finite State Machines (FSM), differ from combinational circuits in several ways. The output signals of sequential circuits are determined by the input signals as well as all previous input signals to the circuit. Contrast this to combinational circuits where only the current input signals are relied upon. This means that sequential circuits need to remember previous inputs, and this is accomplished by implementing memory. This requirement for memory also sets sequential logic apart from combinational logic since the building blocks of the latter consist only of standard logic gates, while those of the former consist of bistable latches and flip-flops on top of the logic gates [26].

The structure of a sequential circuit/FSM starts with the state memory. This memory stores a bit combination that represents the history of all previous inputs to the circuit up to that point in time. This bit combination, at any one instance, is referred to as the state of the system [19]. The outputs of the system are determined from the current state by an output logic circuit which is a combinational circuit. Output logic may or may not be dependent on the current inputs to the system [19].

Operations that are to be performed by the FSM are usually assigned to a state. This is such that if the FSM is in a particular state, then operations assigned to that state will be carried out. The circuitry that determines what state the FSM shall move to next is called Next State Logic [19] and like the output logic circuit, it is a combinational circuit. The inputs to the next state logic are the current state of the FSM and the current inputs to the system [19] thus completing the definition of a sequential circuit since the current and previous inputs determine the outputs.




***Moore Finite State Machine***




***Mealy Finite State Machine***

*Figure 2.9: Finite State Machine Models*

As mentioned earlier, the output logic may or may not be dependent on the current inputs to the system. FSMs whose output logic circuits do not depend on the current inputs are classified as Moore Finite State Machines while those whose output logic include the current inputs are classified as Mealy Finite State Machines [19]. Figure 2.9 summarizes what has been discussed about finite state machines so far.

It is also important to note that changes in states in FSMs are usually triggered by an external signal. This may be done asynchronously based on events such as system reset, or

synchronously based on a global clock. There is also the option for pulse driven triggering where a state change occurs after an event is detected on the rising edge of the clock [26].

## 2.2.4. The General-Purpose Microprocessor

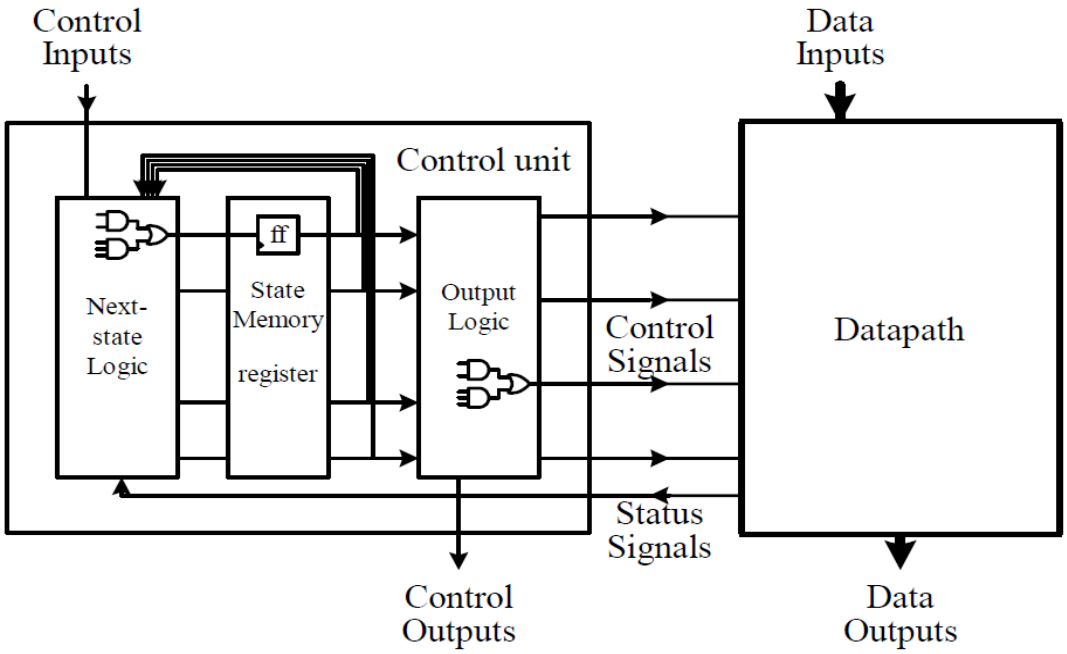



*Figure 2.10: Block Diagram of a Microprocessor. Image adapted from [19]*

The general-purpose microprocessor, also referred to as the central processing unit (CPU) consists of a control unit and a datapath as illustrated in Figure 2.10. The datapath contains all the circuitry required to carry out operations on data and is capable of performing all the operations defined in the instruction set of the CPU [19]. This circuitry includes functional units such as arithmetic and logic units, registers for temporary storage of data currently being operated on, multiplexers and buses that allow for data to be transferred to different parts of the datapath etc. Two important registers present in the datapath are the Instruction Register and the Program Counter. The Program Counter stores the address of the next instruction to be executed by the CPU while the Instruction register stores the actual instruction. It should be mentioned that the datapath is mostly composed of combinational circuitry [19].

The control unit on the other hand is responsible for controlling the datapath by way of asserting control signals as specified by an instruction. The control unit is a finite state machine and undergoes state changes in response to the clock cycle. Each state is assigned an operation in the datapath. Referring to Figure 2.10, the next state logic of the control unit has several inputs, namely, control inputs, the current state and the status signal from the datapath. The status signal is useful in cases where the next instruction to be executed is dependent on the result of the current computation (branch condition). This signal lets the control unit determine what state to move to next since it may be asserted or de-asserted with respect to the result of a branch condition [19].

The control unit typically cycles through 3 operations (fetch, decode, execute). This is referred to as the instruction cycle. For each of the three steps, the following happens:

1. **Fetch:** The control unit places the address stored in the program counter into the address bus. The external memory (where the program is stored) then loads data from this address into the data bus. This data is then loaded into the instruction register and the program counter is incremented. All this is accomplished in a single clock cycle [19].
2. **Decode:** The control unit reads the contents of the instruction register and jumps into the state assigned to carry out the task specified by the instruction. This is also accomplished in 1 clock cycle [19].
3. **Execute:** Now while in this state, the control unit generates control signals which in turn control what parts of the datapath are activated, depending on the task being performed. At this stage, the instruction is actually being carried out. Operations that require memory reading or writing may take more than a single clock cycle to complete which in turn would require additional states for the control unit to move to. This means that the execute step may take more than a single clock cycle to complete [19].

To increase processor throughput, pipelining may be implemented. Instruction pipelining is the technique in which different steps in the instruction cycle are performed simultaneously. For example, while the current instruction is being executed, the next instruction is being fetched from external memory. This has the advantage of allowing for faster execution of instructions however it does introduce some hazards. One such hazard is the data dependency hazard. If an instruction requires data from the preceding instruction, but the pipeline is such that the newest instruction is executed before the preceding one writes its result to memory, then a dependency hazard will have occurred [27]. Another hazard is the control hazard which occurs if a branch statement is reached. In this case, any newly fetched instruction may have to be flushed before it can be executed because the branch statement may cause the next instruction to be a different one form the anticipated [27].

## 2.3. Interaction of Radiation and Electronics

When energetic particles travel through materials, they deposit their energy through a number of mechanisms. Primarily, these particles lose energy through ionization and atomic displacement of the target material. The characteristics of such interactions are described by plots such as that in Figure 2.11. The plot is of the LET of a proton traversing through high density polyethylene plotted against the depth of penetration.

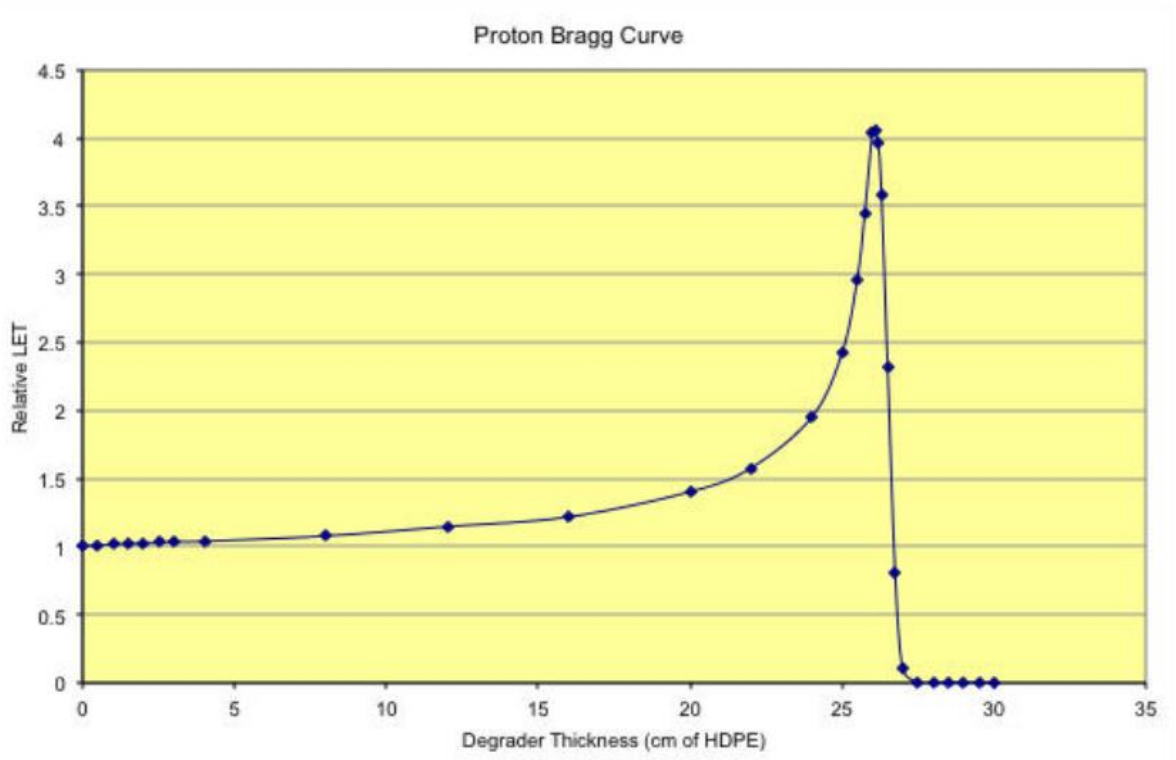



*Figure 2.11: Bragg curve for 205 MeV Protons in High density polyethylene (ρ = 0.97 g/cm3) [28]*

At first, the proton barely deposits any energy due to its velocity being in the relativistic range and this can be seen in the plot where the LET is almost flat [15]. This is the case because at such velocities, the proton barely has time to interact with the atoms of the target material and deposit energy. Additionally, at relativistic velocities, changes in energy barely have an effect on velocity [15]. Eventually, the proton begins to slow down and almost immediately loses all of its energy because at a lower velocity, it will deposit energy at a higher rate. Plots of this type are referred to as Bragg curves and the shape is similar even for different ions traversing through silicon in electronics. The peak LET is referred to as the Bragg peak [15], after which the proton will reach the end of its travel range.

From these interactions, one effect on electronics is displacement damage [9]. When an ion traverses through a semiconductor, it may collide with silicon nuclei, thus displacing them from the lattice and creating interstitials [9]. Where these nuclei use to occupy will now be vacancies. The vast majority of interstitials and vacancies usually recombine relatively quickly after irradiation however some do remain as defects. These defects may be of concern in bipolar transistor and optoelectronic devices [16]. Displacement damage however is not of major concern when it comes to CMOS technologies [16, 18, 29].

The other major effects that radiation has on electronics are single event effects (SEEs), that happen over a short span of time, and long-term effects that result from accumulated dose, also known as Total Ionizing Dose (TID) effects. Both SEE and TID effects arise from ionization caused by radiation. The remainder of this chapter discusses the mechanisms that give rise to SEEs and TID effects. With this regard, not much literature is available outlining these mechanisms on the FinFET that has only recently seen large scale adoption, however knowledge of the same on planar devices is well established.

## 2.3.1. Single Event Effects - SEE

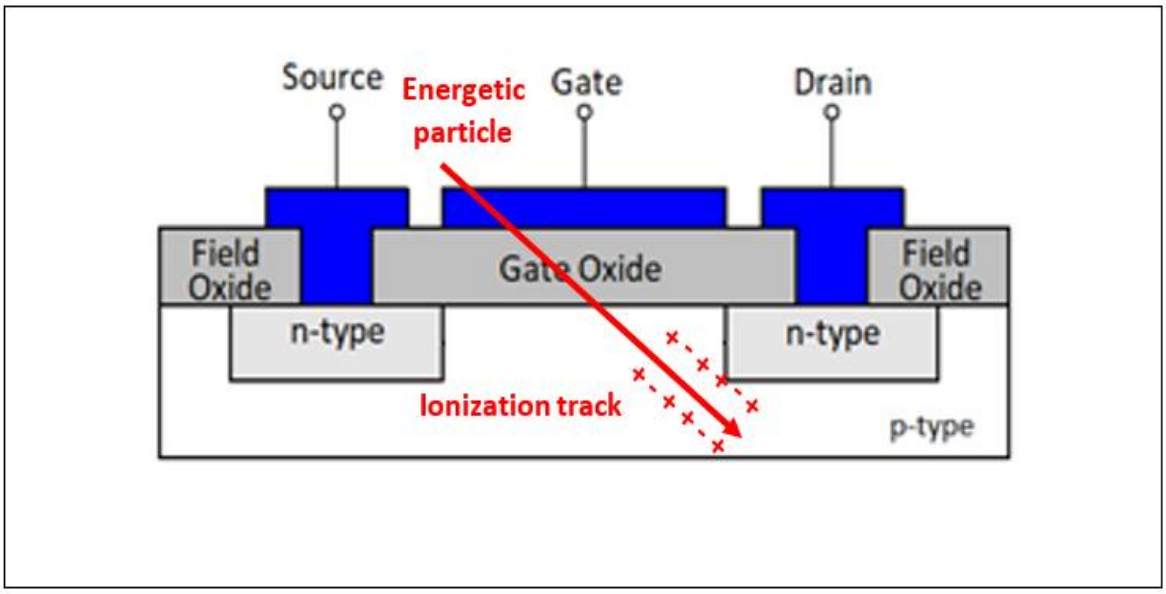



*Figure 2.12: Ionization track left behind by a charged particle. Image adapted from [19]*

Single Event Effects is a broad classification for different types of transient responses that electronics may have as a result of energetic particles striking sensitive areas. Some of these responses may manifest as temporary errors (soft errors) such as bit flips while others may take form as destructive events (hard errors) that cause permanent device failure. These effects are of major concern in the space community because they may have profound consequences on missions.

### 2.3.1.1. Single Event Transients - SET

Figure 2.12 illustrates the basic driving phenomenon that causes SEEs. The figure is of a planar MOSFET, however, the process is similar in FinFETs. An energetic particle striking a transistor may cause ionization where electron-hole pairs are formed. If this particle is a heavy ion, the ionization occurs directly (primarily) as well as indirectly. Indirect ionization is where the initial ion collides with other nuclei thus producing recoil ions. These recoil ions in turn cause further ionization. If the energetic particle is a proton, the ionization is primarily indirect, however, some direct ionization may occur in highly sensitive devices [15]. In the case of neutrons, the ionization is entirely indirect.

The electron hole pairs formed by the ionization cause a current pulse (whose width is in the picosecond scale [30]) that may propagate from the struck device to other devices in the circuit. This current pulse is referred to as a single event transient (SET). The struck device that experiences this SET will then recover by nature of its bias condition [15].

### 2.3.1.2. Single Event Upset - SEU

A single event upset (SEU) can be defined as a bitflip error that occurs in sequential logic as a result of direct or indirect interaction with charged particles. SEUs are soft errors because the affected circuitry will usually recover in the next memory write, set or reset operation.

One way in which SEUs occur is when an SET occurs in combinational logic and activates subsequent logic paths that would otherwise be inactive. Figure 2.13 shows such a situation. An SET is induced in the centre NAND gate and is propagated through to the D Flip-Flop. If this SET occurs for long enough to be met by the rising edge of the clock, an erroneous value may be latched into the Flip-Flop.

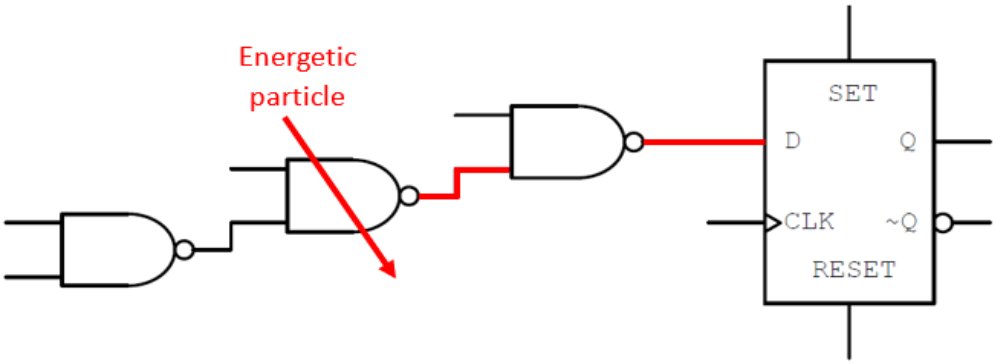



*Figure 2.13: SEU path in combinational logic*

An SEU may also occur as the result of a transient in the clock tree of the Flip-Flop, provided that the inputs to the Flip-Flop have changed within the clock cycle. Additionally, a transient in the SET or RESET lines would result in the same [15].

In a latch or Static Random Access Memory (SRAM) cell, an SEU may occur in the following way. Referring to Figure 2.14, a particle hit on the top inverter may induce an erroneous output. While this error propagates to the lower inverter as feedback, the lower inverter will be imposing a restorative signal to the top inverter. If the erroneous feedback exceeds the restorative signal, the logic state of the memory cell will change and an SEU will have occurred [18, 31].

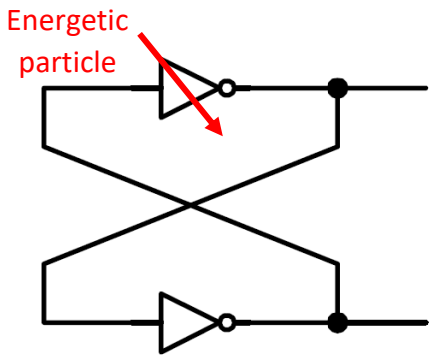



*Figure 2.14: SEU process in SRAM. Image adapted from [19]*

Single Event Upsets in Dynamic Random Access Memory (DRAM) occur through a completely different mechanism compared to SRAM. Figure 2.15 from [32] describes the different stages using single transistor-capacitor models for two DRAM cells. Logic “0” is stored in one cell where the potential well is filled with electrons and logic “1” in the other cell where the well is empty. Each cell is then struck by an α-particle that leaves behind an ionization track. Electrons induced from this ionization are then swept into the potential wells while the holes are repelled. For the cell that initially contained a logic “0”, there will be no change in the stored value, however, for the cell initially containing the logic “1”, the value will have changed to a logic “0”. From this simple model, it can be seen that for DRAM cells, only one electrical state is vulnerable to SEUs [31, 32].

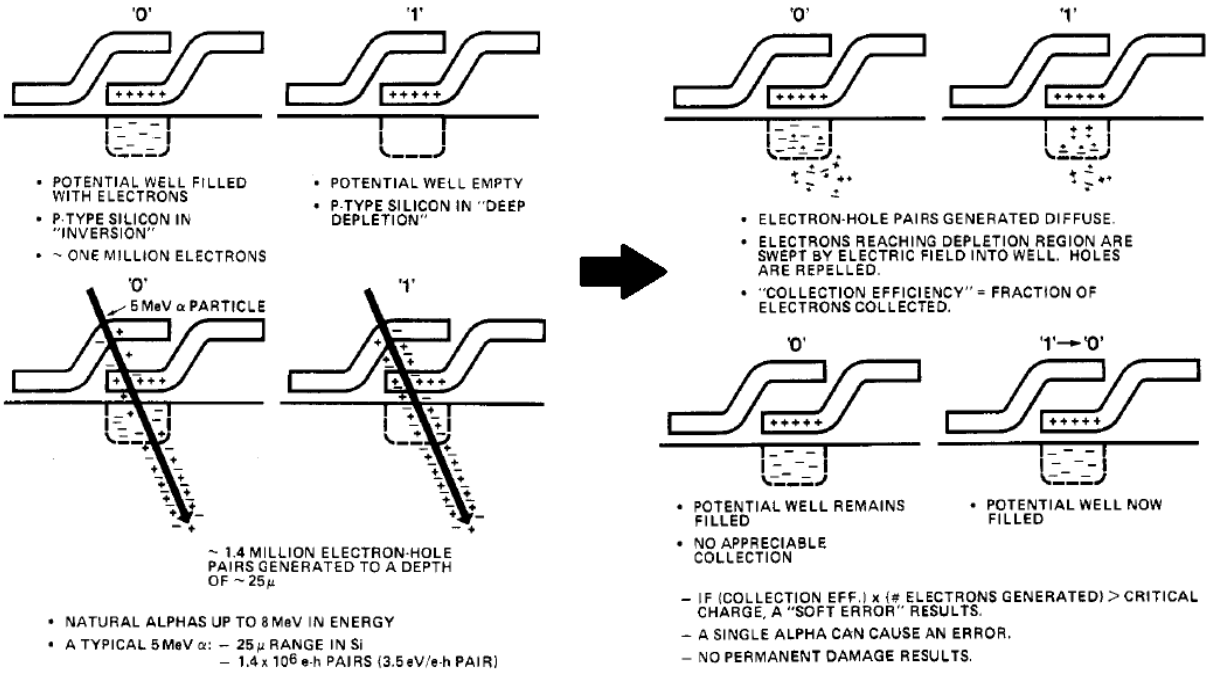



*Figure 2.15: Stages of SEU in DRAM. Image adapted from [32]*

### 2.3.1.3. Multiple Bit Upsets - MBU

The occurrence of multiple SEUs, within the same clock cycle, and induced by a single charged particle is referred to as multiple bit upsets (MBU). These may occur in one of two ways. The first is when a particle strike takes place on circuit nodes such as the clock tree which in turn would cause multiple sequential blocks to erroneously latch signals. The other is when a single charged particle carries with it enough energy to not only cause an SEU on the device it strikes, but also to those devices physically close to the struck device.

### 2.3.1.4. Single Event Functional Interrupt - SEFI

Single Event Functional Interrupts arise as a result of the prior discussed responses from electronics. They occur when errors brought about by SEUs and MBU propagate through the electronic device and alter the operation of the device as a whole. An example may be when SEUs occur in the control unit of a microprocessor forcing it to enter an undefined state. This way, the operation of the entire device may be disrupted. Devices sometimes

recover from SEFIs spontaneously however may require power cycling or system resets to restore operability in other instances.

### 2.3.1.5. Single Event Latch-up – SEL

Inherent to bulk CMOS technologies are parasitic vertical and lateral n-p-n and p-n-p BJTs [9] (refer to Figure 2.16). These parasitic transistors may form a silicon-controlled rectifier structure (p-n-p-n) which is usually biased in the "off" state by design of the CMOS and at normal operating conditions [18].

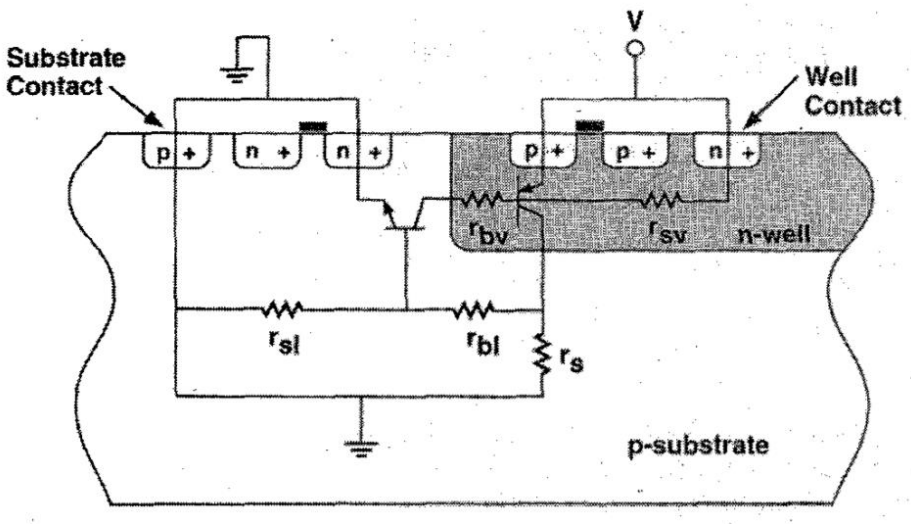



*Figure 2.16: Two-transistor model for latch-up in an n-well CMOS structure [33]*

A particle strike may bias this p-n-p-n structure into the "on" state which in turn provides a low impedance electrical path between the source and ground (i.e. creates a short-circuit). Consequently, a high current state is attained that the device can only recover from through a power cycle or exercising logic while the supply voltage is reduced [15]. This is referred to as a Single Event Latch-up (SEL) and it is potentially destructive if the latch-up current is large enough.

### 2.3.1.6. Single Event Burnout – SEB

SEBs are destructive events that occur in power MOSFETS. Their mechanism involves a parasitic BJT in the power MOSFET that is triggered into a regenerative forward bias by an energetic particle. This results in a destructive high current which causes permanent damage [15, 18].

### 2.3.1.7. Single Event Gate Rupture – SEGR

Like SEB, SEGR is a destructive response by power MOSFETS. It occurs when an energetic particle strikes the gate of said device causing it to rapture.

With all this mentioned, it should be kept in mind that as more advancements are made in the semiconductor industry, and with electronic feature sizes reducing, there are profound implications to the vulnerability of newer devices to SEEs. For instance, in planar devices, the critical charge ($Q_C$), defined as the absolute difference in charge content between HI and LO logic states [9], decreases linearly with device size. Conversely, SEU vulnerability increases with a decrease in critical charge [9].

The relationship between SEU vulnerability and critical charge however is not necessarily linear, due to factors like device thickness and operational frequency. As newer devices get thinner, there is less material for coincident energetic particles to traverse through, which in turn means that less energy is deposited into the devices [9]. The increase in operational frequencies also means that short SETs that would otherwise be harmless are more likely to get latched into memory.

To further demonstrate the impact that advancements have on SEE vulnerability, Karp *et al.* [24] found that advanced FinFET technologies have a higher SEL sensitivity compared to their advanced planar counterparts. Advanced FinFETs were also found to experience lower rates of SEUs by [25] and [34] compared to older planar technology nodes. Adding to this, scaling and physical structure of the FinFET are both factors that lead to shorter duration SET pulses [25, 35]. However, like planar devices, SEU vulnerability is dependent on the angle of incidence of the charged particle [36, 37].

The vulnerability of a device to SEEs is therefore experimentally determined on a case by case basis by plotting the cross section ($\sigma$) of the device against different particle LET values. In the case of protons, their initial energy is used instead of LET [31, 38]. The cross section at a given energy is calculated using the formula:

$$\sigma = \frac{n}{Ft\cos\theta} \qquad \textit{Eqn 2.2}$$

Where $n$ is the number of SEEs counted, $F$ is the flux of the particles, $t$ is the time of exposure and $\theta$ is the angle of incidence of the particles on the device. $\sigma$ is a measure of the probability of an incident particle to cause a SEE and is expressed in the units SEEs per particle/cm$^2$ or just as cm$^2$.

In order to determine the dependence of cross section on proton energy, 3 separate equations may be used to fit data obtained from tests into a graph similar to Figure 2.17. These are Bendel 1-parameter, Bendel 2-parameter and Weibull equations.

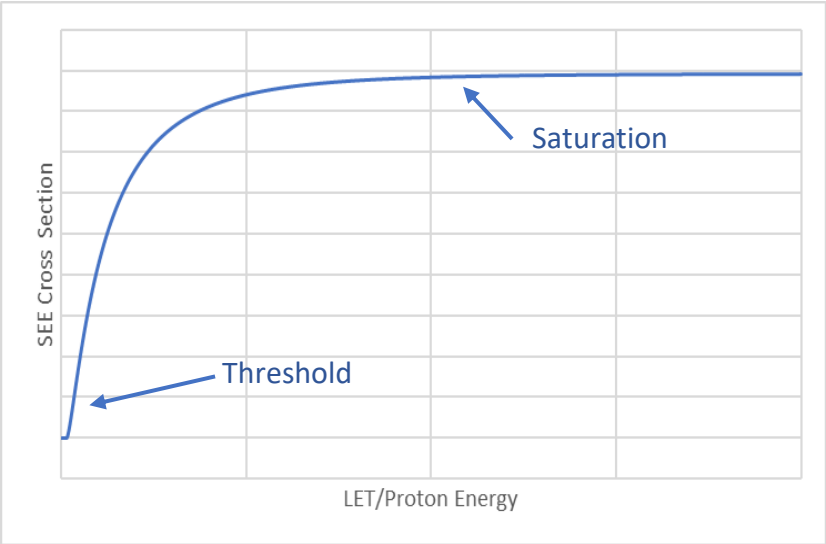



*Figure 2.17: Typical Shape of a Cross Section plot*

#### 2.3.1.7.1 Bendel 1-parameter

This single parameter equation may be used in cases where measurements were made at a single energy level [39]. The equation is given as [40]:

$$\sigma(E) = ({}^{24}/_{A})^{14}[1 - e^{-0.18\sqrt{Y}}]^4 \qquad \textit{Eqn 2.3}$$

Where

$$Y = (E - A)\sqrt{{}^{18}/_{A}}$$

Here, E is the proton energy in MeV and A is the upset sensitivity parameter, also in MeV. The function outputs are in units of $10^{-12}$ $cm^2$ [41].

#### 2.3.1.7.2 Bendel 2-parameter

This equation is a modification of the Bendel 1-parameter equation and has been shown to better describe device cross sections [42, 43]. There are two parameters: A – the upset sensitivity parameter and B (has no explicit physical interpretation). The equation is given as [42]:

$$\sigma(E) = ({}^{B}/_{A})^{14}[1 - e^{-0.18\sqrt{Y}}]^4 \qquad \textit{Eqn 2.4}$$

Where

$$Y = (E - A)\sqrt{{}^{18}/_{A}}$$

The Bendel 2-paremeter equation generally sees wider use than the Bendel 1-parameter equation.

#### 2.3.1.7.3 Weibull Equation

This equation is mainly used to fit cross sections from tests using heavy ions. However, for some devices, it has been shown to provide better fits than both the Bendel 1 and 2-parameter equations while using protons [39, 44]. The form of the equation is as follows:

$$\sigma(E) = A[1 - e^{-\left(\frac{E - E_0}{W}\right)^s}] \qquad \textit{Eqn 2.5}$$

Where A is the saturated cross section, E is the proton energy in MeV, $E_0$ is the threshold energy in MeV, W is the width of the rising portion of the graph and s is a dimensionless exponent that determines the shape of the graph (refer to Figure 2.17) [39, 44].

Since there are 4 variables in the Weibull equation, measurements need to be taken at no less than 4 different energy levels, preferably close to the threshold [39].

## 2.3.2. Dose Rate Effects

This encompasses a number of responses by electronic devices (upsets, latch-up, burnouts) [18] caused by exposure to a pulse of high amplitude ionizing radiation [31]. This radiation is usually in the form of X-rays or γ-rays that may result from the detonation of nuclear weaponry [9]. Unlike SEEs that are localised events, dose rate effects occur on the entire integrated circuit at once since they are a result of photocurrents that have been induced in the transistors by the radiation.

If the dose-rate is high enough, rail span collapse may be experienced. This is where the induced photocurrents cause a large voltage drop across the power supply and power distribution rails. This voltage drop in turn affects entire portions of the circuit inevitably leading to signal and/or data loss [45].

## 2.3.3. Total Ionizing Dose Effects

TID effects in CMOS arise from accumulated ionizing charge in electronics that leads to a degradation in performance of said devices. These effects are the dominant response electronics have to radiation [18] since ionizing dose will be absorbed by the device regardless of whether or not the radiation is capable of inducing SEEs.

The nature of the degradation, and to what degree, is reliant on a number of factors which include the dose rate, type of ionizing radiation, applied electrical field to the device, device geometry and physical structure among others [31]. The tolerance of a device to TID degradation will usually give an indication of the expected service life in space applications.

### 2.3.3.1. TID Effects at the Transistor Level

Models that explains the processes behind TID degradation on planar MOSFETs involve radiation induced charge getting trapped in the gate oxide of the transistors, effectively altering the threshold voltage ($\Delta V_{TH}$). Modern commercial CMOS technology however has very thin gate oxides (in the range of several nm and not necessarily $SiO_2$) and this has had the effect of diminishing $\Delta V_{TH}$ to negligible levels [15]. Even with this leading to $\Delta V_{TH}$

becoming a lesser concern, models developed to explain TID effects that still took $\Delta V_{TH}$ into consideration are useful because they still apply to field oxide isolation [15] or STI structures. It should also be noted that the TID response of bulk FinFETs is similar to that of planar bulk MOSFETS [46, 47]. The aforementioned models shall now be described.

In the first process shown in Figure 2.18, when ionizing radiation traverses through the gate oxide of a MOS transistor, it leaves behind a track of electron-hole pairs. Many of these recombine within the time scale of a picosecond, however a fraction of them do not. This is because some electrons, which are many times more mobile than holes, are swept towards the gate due to the applied positive bias. The holes left behind that failed to recombine create a net positive charge that causes a negative shift in threshold voltage in both PMOS and NMOS devices. This charge build-up however is less severe in PMOS. The type of the incident ionizing radiation as well as the applied bias influence the percentage of electrons that will initially recombine with holes [15, 18].

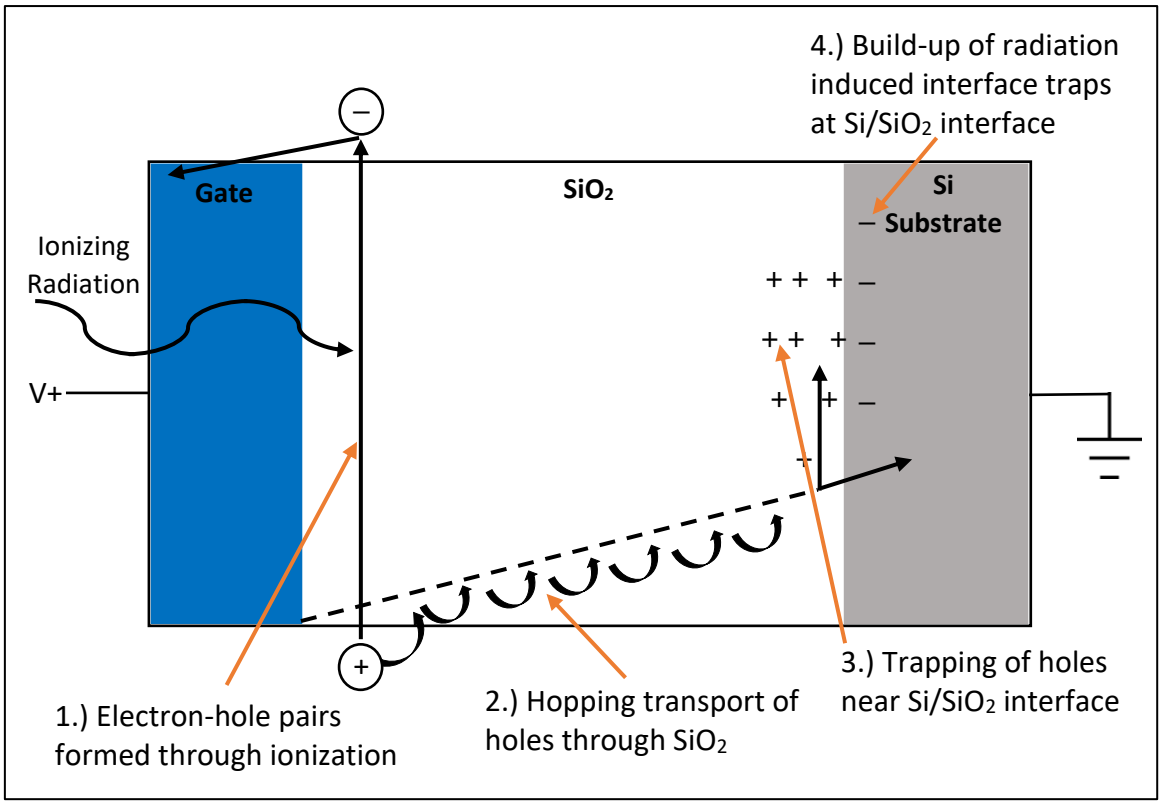



*Figure 2.18: Physical processes responsible for the radiation response of a MOS transistor*

The second process that takes place over a much larger time scale than the first involves the transport of the generated holes towards the $Si/SiO_2$ interface. This may occur in response to the electric field resultant from the applied bias [18]. The duration of this process is affected by temperature, thickness of the gate oxide and the magnitude of the electric field. At temperatures greater than 140K, hole transport is strongly temperature activated but below 140K it is not [15].

The third process occurs close to the $Si/SiO_2$ interface. A fraction of the holes that were getting transported across the $SiO_2$ may fall into **hole trapping sites** here. The exact fraction is strongly influenced by the electric field and temperature. A hole trapping site can be

described as a dangling bond in $SiO_2$ devoid of an oxygen atom. [15] describes this as a weak Si-Si bond where each Si atom is back bonded to 3 Oxygen atoms. These trapping sites are induced by radiation and others may have been introduced during manufacture [31]. The holes that got trapped may remain trapped for anywhere between hours and years, however, they do undergo gradual annealing.

The final process involves the build-up of radiation induced **interface traps** within the $Si/SiO_2$ interface. The occupancy of these amphoteric traps is determined by the electric field resultant from the applied bias and this has the consequence of introducing $\Delta V_{TH}$ that is reliant on the bias voltage [15]. Other than changes in the threshold voltage, other effects that charge trapping may have on the performance of a MOSFET are as follows:

**Switching speeds –** Given that the occupancy of radiation induced interface traps is strongly influenced by the electric field, a sweeping gate voltage will cause the traps to "fill up" or "empty". This phenomenon inevitably raises or lowers the amount of electrical charge required to bring the transistor into strong inversion. The negative effect of this manifests itself in the form of a reduction of switching speed of the transistor. This happens if the net charge of the interface traps makes it such that larger voltage swings are required to switch the device on and off [31].

**Carrier mobility –** To explain this, let us take the example of an NMOS device that has interface traps with a net negative charge. When the transistor is switched on, electrons travelling from the source to drain will experience coulomb scattering due to the repulsive negative charge of the interface traps. Effectively, this leads to an increase in "channel on resistance" [31].

**Leakage Currents –** Isolation structures may accumulate substantial amounts of charge that may lead to low resistance electrical paths forming. Parasitic leakage current may then use these paths to pass between neighbouring transistors or between source and drain of a single transistor. Figure 2.19 illustrates this for a 3-fin FinFET.

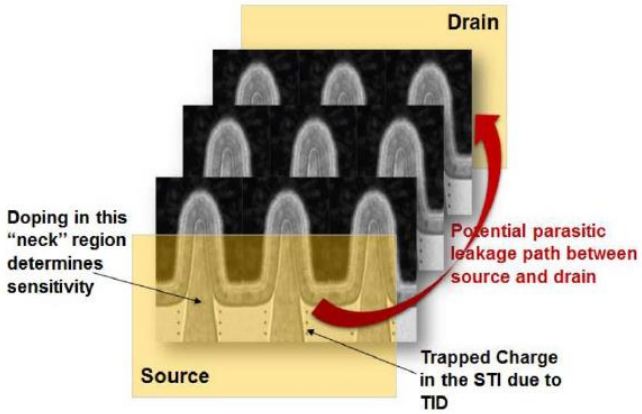



*Figure 2.19: Parasitic leakage currents in a 3-fin FinFET [47]*

## 2.3.3.2. TID Effects at the IC Level

Without needing mention, the top level TID response of a complex integrated circuit is governed by the underlying transistor level responses. For instance, TID induced transistor

source to drain leakage currents will lead to an increase in standby current of the IC. This is because the standby current of an IC is correlated to the OFF current of the transistors it is constituted of [31].

Adding on to this, an IC may experience Transistor-transistor Logic (TTL) compatibility issues. This may arise from TID induced changes in both carrier mobility and threshold voltage parameters at the transistor level. The combined effects of these on the I-V characteristics of the transistor may have a profound effect on signal propagation timing and levels [31]. Other IC level TID effects that arise directly from the transistor level are functionality implications, internal timing issues, changes in operating voltages and frequencies among others [31].

Worth mentioning, Zhang *et al.* [48] found that for 14-/16nm FinFET Flip-Flops, TID dose at first seemed to increase the SEE cross section but with further increase in dose, the cross section decreased. The doses at which the cross-section decreases were observed seemed to depend on the supply voltage and Flip-Flop design. Bacchini *et al.* [49] also reported on a significant decrease in retention time for data in DRAM as a consequence of absorbed dose.

Table 2.1 is presented to bring this chapter to a close. It highlights the different types of effects that different sources of radiation in space may have on electronics.

| **Effects** | **Sources** | | | |
|---|---|---|---|---|
| | *Van Allen Belts* | | *Solar Flares* | *Cosmic Rays* |
| | Electrons | Protons | Protons | Ions |
| **TID** | x | x | x | |
| | | | | |
| **SET** | | x | x | x |
| **SEU** | | x | x | x |
| **SEL** | | x | x | x |
| **SEB** | | x | x | x |
| **SEGR** | | x | x | x |
| **Displacement Damage** | x | x | x | |

*Table 2.1: Summary of radiation sources and their effects on electronics. Adapted from [50]*

# 3. Test Setup and Procedure

This chapter gives a brief overview of some considerations that should be made before testing the SEE and TID characteristics of a microprocessor. Further on, a description is given for the microprocessor that was tested as well as the test software and test setup used.

## 3.1. Testing Considerations

### 3.1.1. SEE Testing

#### 3.1.1.1. Testing methods

In the past, there were several approaches taken to testing older, simpler processors. Most of these involved using custom hardware and dedicated machine instructions that gave good visibility into the state of the processor at any given time during irradiation [38].

An example of these earlier approaches would be one where an external controller or computer would monitor the output pins of the processor under test. The DUT would execute a test program while the controller compared the DUT outputs with known values. The controller would then log and report any erroneous outputs detected. Alternatively, the controller would compare the DUT outputs to the outputs of a "golden chip", which was an identical processor executing the same program but not undergoing irradiation. The controller would then log and report any discrepancies in outputs between the two devices [38].

Another approach that was used was a single self-testing computer. The DUT would be installed as part of a full computer configuration. The DUT would then run test programs and compare the results to known values. If errors were detected, they would be reported by the DUT through connected peripherals [38].

Modern microprocessors on the other hand are very complex devices that operate at very high frequencies, have pin counts numbering in the thousands and require multiple supporting electronics just to function. This inevitably adds high complexity to the task of designing custom hardware and software (including operating systems) for testing purposes. For these reasons, tests carried out on newer processors typically use the single self-testing computer approach or testing the processors on development boards supplied by the respective manufacturer [38].

#### 3.1.1.2. Operating System

The choice of operating system (OS) plays a major role since the complexity and operational characteristics of the OS may heavily influence test results, sometimes even interfering with data acquisition. This latter point can be seen with Howard *et al.* [51] where they were testing Intel Pentium III and AMD K7 microprocessors. They experienced such a high OS crash rate that limited how much SEU data they could collect.

Generally, the more complex the OS is, the higher likelihood that it would crash during testing [38]. This is problematic since it is difficult to distinguish a crash caused by an error in the OS or by an error occurring in a critical location of the DUT [38]. It is therefore recommended that tests on microprocessors be carried out using primitive operating systems unless more complex operating systems are intended to be used for the mission [38].

Another implication is brought about by the use of pre-emptive operating systems. A pre-emptive OS is one which allows for tasks to be interrupted during their execution. During an interrupt, the processor typically switches context to service the cause of the interrupt. This context switch involves the processor pushing the values that were stored in its internal registers to special registers and memory. The processor will then populate its internal registers with new values that are required to service the interrupt and proceed to execute the interrupt service routine. Upon completion, the processor will switch context back to the task it was initially executing and rewrite the initial values to its internal registers once more. This is problematic if testing using the single self-testing computer approach because the "visibility" of the state of registers is limited to only when the test program is being executed. If an error occurs during an interrupt service routine, it will be missed.

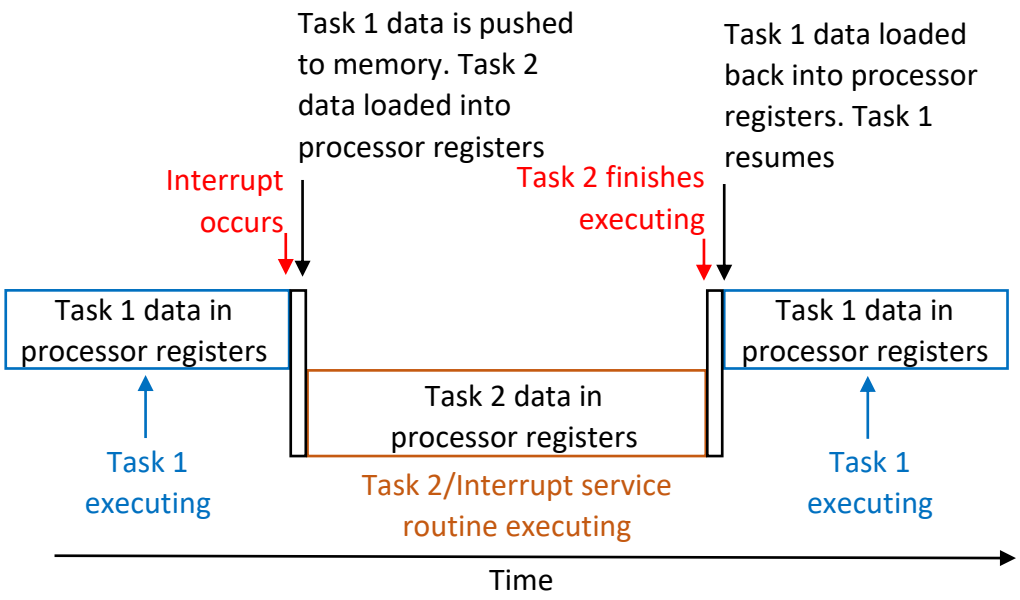



*Figure 3.1: Processor Context Switching as handled by a pre-emptive OS. Task 1, which checks for upsets, is initially executing. An interrupt causes a context switch to Task 2 (the interrupt service routine). If an upset occurs during this service routine, it will be missed by Task 1.*

To combat this, a real time operating system (these allow for deterministic task scheduling) could be used with the test program set to run as a high priority interrupt. This would minimize the probability of the test program getting interrupted and the processor registers having their values rewritten outside the control of the test program (i.e. the test program would run as Task 2 in Figure 3.1). This option however tends to be costly either financially or effort-wise due to complexity of implementation. In the financial sense, real time operating systems are typically intended for industrial and corporate use and are thus

priced accordingly. Conversely, free alternatives require a significant amount of system knowledge and time to implement. An example can be given of the RT Linux microkernel that once installed, will run the entire Linux kernel as a pre-emptive task (thus allowing different tasks to be run deterministically). To implement it, one has to first compile it then mount it beneath the Linux kernel [52], which in itself requires a significant amount of technical know-how.

Another possible solution would be to completely disable the pre-emption of the OS. This however does introduce problems such the system being unresponsive to user inputs (which are usually handled as interrupts) or the test program not even executing when required due to some other processes introducing indefinite holds to the task queue [53]. For these reasons, it would be best to only use this option for processors whose internal states can be monitored by external devices during testing.

#### 3.1.1.3. On-Board Cache

In most modern microprocessors, a significant portion of the die is dedicated to on-board cache memory (SRAM) [38]. Additionally, the total number of bits in cache is much higher than the number of registers in the processor. This means that when the processor is running and using the cache heavily, there is a higher probability of an SEU occurring in cache [38] and causing a system crash. This results in a higher device cross section [2, 51] compared to when the device is running without using cache memory.

It is therefore common practise to disable the on-board cache to allow for sufficient system up-time during testing. This also extends to devices on missions.

### 3.1.2. TID Testing

A consideration to be made for TID testing is whether to test while the processor is biased or unbiased. Results obtained from biased tests are more indicative of what the device will experience on mission since it is mostly going to be biased while in space [39].

Care should also be taken that the dose rate chosen for irradiation of the device is not between the hole trapping dominated and interface trap dominated response of the device. This prevents unrealistically high dose survivability figures from being obtained [15]. Other than this special case, the dose rate at which the test is carried out does not really matter. At the onset of testing, variations in results may be seen for devices tested at different dose rates (due to time dependence) however if the devices are given enough time to anneal, they will show similar performance and give similar test results provided that they have absorbed similar amounts of total dose [18, 31].

## 3.2. Device Tested

### 3.2.1. E3815 System-on-a-Chip

As mentioned earlier in Chapter 1, the device that this research focuses on is the Intel Atom E3815 microprocessor which comes packaged as part of a System-on-a-Chip (SoC). This SoC also includes integrated graphics, an integrated memory controller and an integrated platform controller hub [54]. The E3815 is a 64-bit processor with a feature size of 22nm and has 1 core that only supports 1 thread [55]. Additionally, it comes with 512 kB of L2 cache and runs at a base frequency of 1.46 GHz [55]. More specification details can be found in Appendix 1: Intel Atom E3815 Specifications .

Figure 3.2 is an illustration of the general architecture of a x64 Intel processor such as the E3815. x64 processors use the 64-bit instruction set architecture which is an extension of the previously used 32-bit x86 instruction set architecture [56] . There are 3 main register banks, namely the General-Purpose Registers (GPR), Floating Point Registers on which the MultiMedia eXtensions (MMX) registers [57] are overlaid and finally the XMM registers.

The General-Purpose Register bank consists of 16 registers that are each 64 bits wide. These registers are mainly used to store arguments that are passed to functions, store return values of functions and as temporary registers. Additionally, RBP (refer to Figure 3.2) is used to keep track of the base of the current stack frame and RSP is used as the stack pointer that points to the top of the stack [58]. Specific uses of each GPR are listed in Table 3.1.

The 8 floating point registers (FPR0-FPR7), together with status and control registers not shown in Figure 3.2 constitute the Floating-Point Unit (FPU). Each floating-point register is 80 bits wide. Overlaid on the FPU are the 8 MMX registers that are each 64 bits wide [56].

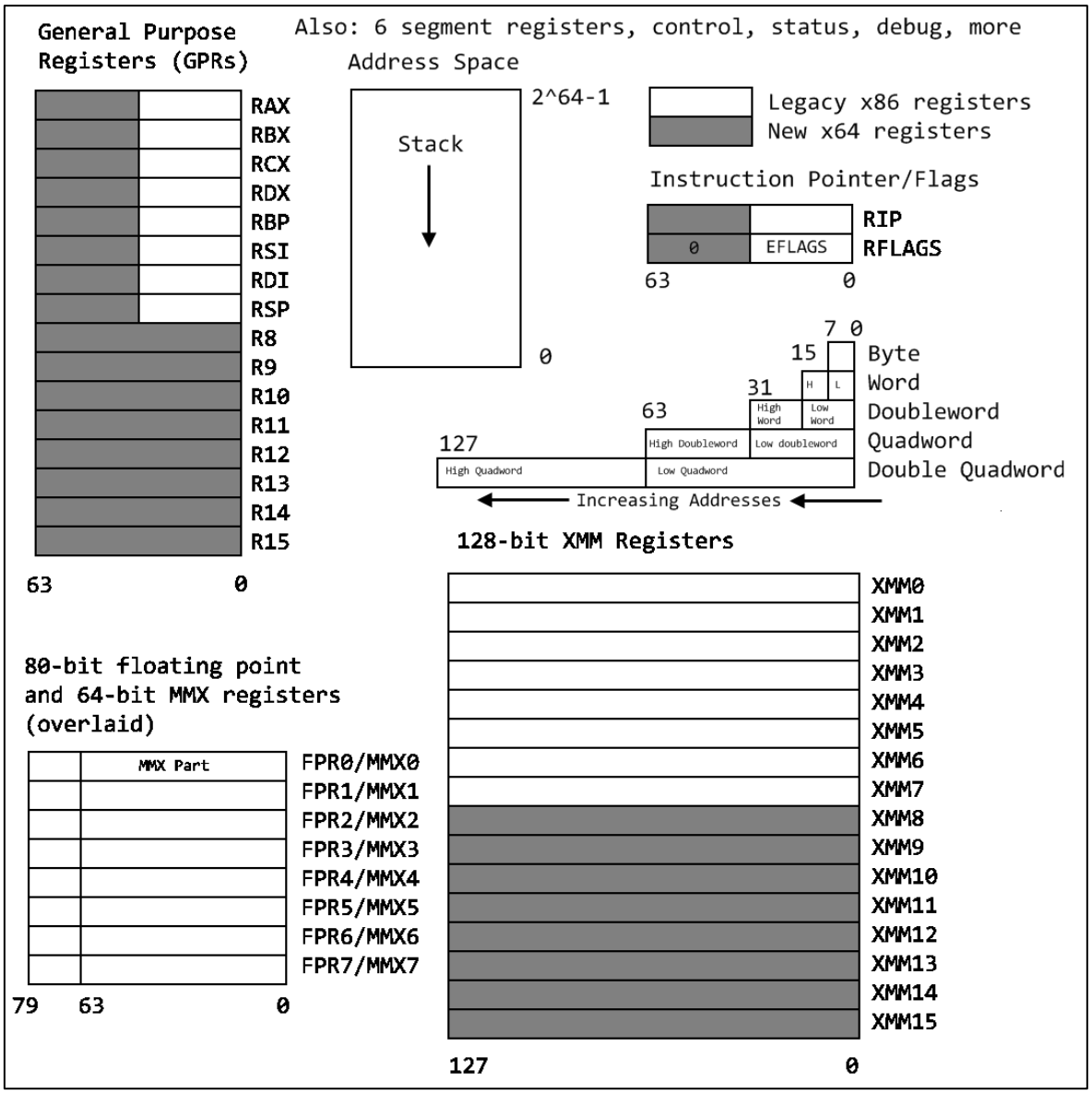



*Figure 3.2: General x64 Architecture [56]*

The FPU, as the name suggests, is used to carry out operations on floating point numbers. Unlike the GPRs, each FPU register is not individually addressable, instead, the entire FPU can only be accessed as a stack. FPR0 (see Figure 3.2) acts as the top of the stack and floating-point operations can only be performed on it and another register in the FPU (including itself) but not between any other 2 FPU registers [59]. The MMX registers are implemented to reduce the time that certain multimedia operations take to process, effectively increasing performance and speed [57]. They bring with them Single-Instruction, Multiple-Data (SIMD) instructions which allow for operations to be carried out on multiple integers simultaneously [60]. Since MMX and the FPU share the same hardware, MMX tasks cannot run while the FPU is in use and vice versa.

The XMM register bank consists of 16 registers that are each 128 bits wide. They are used by Streaming SIMD Extensions (SSE) instructions. SSE instructions are an extension of SIMD [60]. XMM registers can be used for the same operations as MMX and have additional

capabilities thanks to SSE instructions. This results in XMM seeing more common use than MMX in newer software. Each MMX and XMM register can be individually addressed.

| Register Name | Use(s) |
|---|---|
| **RAX** | Temporary register |
| | With variable arguments passes information about the number of vector registers used |
| | 1st return register |
| **RBX** | Callee-saved register |
| **RCX** | Used to pass 4th integer argument to functions |
| **RDX** | Used to pass 3rd argument to functions |
| | 2nd return register |
| **RBP** | Callee-saved register |
| | Optionally used as frame pointer |
| **RSI** | Used to pass 2nd argument to functions |
| **RDI** | Used to pass 1st argument to functions |
| **RSP** | Stack pointer |
| **R8** | Used to pass 5th argument to functions |
| **R9** | Used to pass 6th argument to functions |
| **R10** | Temporary register used for passing a function's static chain pointer |
| **R11** | Temporary register |
| **R12** | Callee-saved registers |
| **R13** | |
| **R14** | |
| **R15** | Callee-saved register |
| | Optionally used as Global Offsets Table (GOT) base pointer |

*Table 3.1: General Purpose Register Usage. Table adapted from [61]*

Other registers include control registers, virtualization registers, RIP (64-bit wide instruction pointer register), memory management registers, RFLAGS (stores flags used to keep track of some branch operations and control the processor), status registers and performance registers [56] among others. What has been presented here is by no means exhaustive of all the registers present in a x64 processor.

## 3.2.2. Intel NUC DE3815TYBE

Due to the complexity of the E3815 microprocessor, it was decided that the device would be tested on a commercially available single board industrial PC rather than on a custom designed board. This avoided the task of designing a compatible board and a custom operating system. To this end, the Intel NUC DE3815TYBE was selected. This board came with the E3815 SoC package already soldered on in a ball grid array configuration.

Figure 3.3 shows photos of the top side and bottom side of the board and Figure 3.4 is a block diagram of the major functional parts of the board.

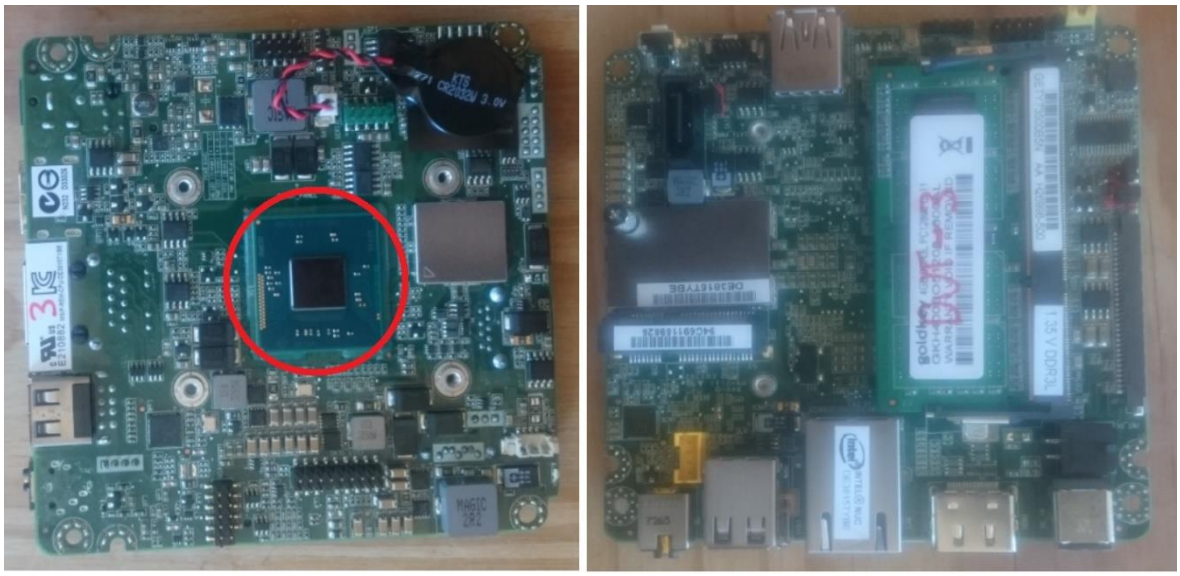


*Figure 3.3: Photos of the Intel NUC DE3815TYBE. Left -Top of the board with heatsink removed to expose the E3815 SoC (highlighted by red circle), Right -Bottom of the board with a 4 GB SO-DIMM RAM module installed*

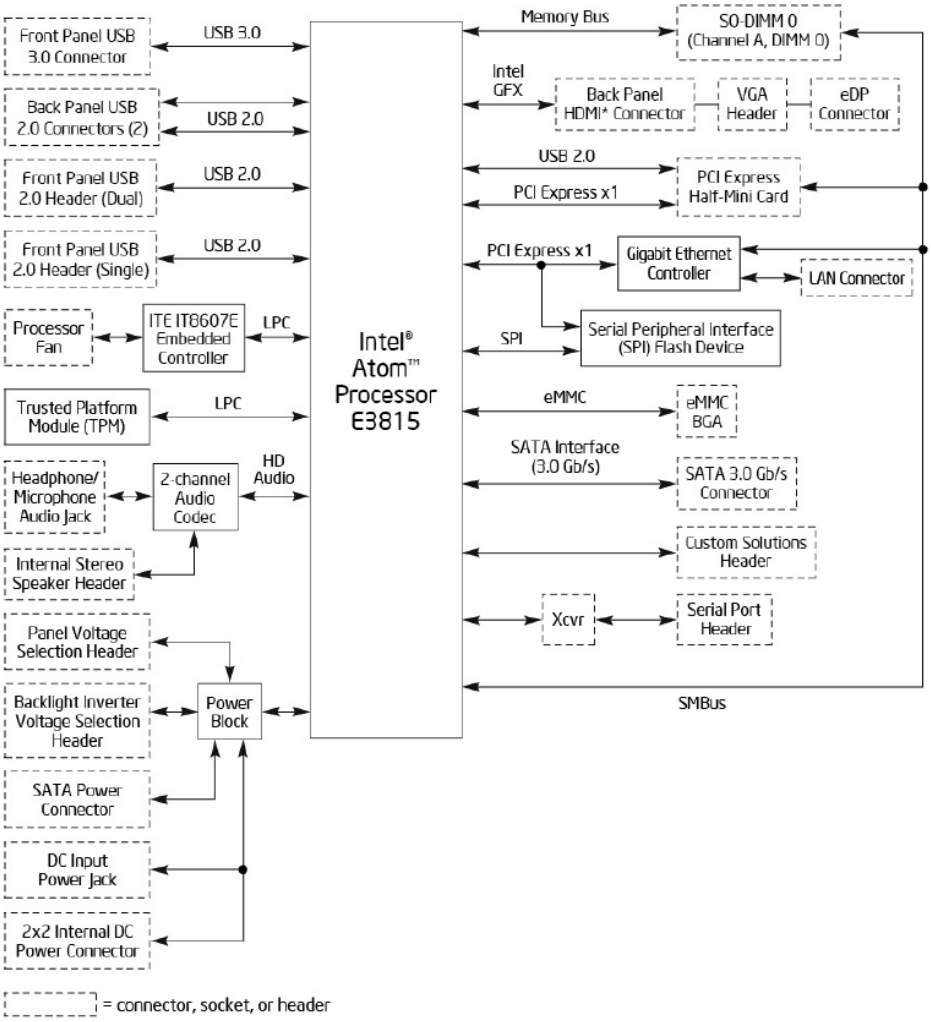



*Figure 3.4: Block Diagram of the major functional parts of the DE3815TYBE [54]*

As can be seen in the block diagram, the board is essentially an entire computer. It comes with 4 GB embedded eMMC memory and has USB, Ethernet and HDMI display ports that can be used to connect standard peripheral devices. The board can operate on DC input of 12V to 19V with a maximum current rating of 3A [54].

Additional hardware used includes 4 GB of non-ECC SO-DIMM memory (RAM) running at 1066 MHz (because the board did not support ECC RAM) and a 120 GB Solid State Drive (SSD) connected to the board via the SATA port. This SSD is where the operating system was installed.

As for software, the following was used:

- **BIOS version:** TYBYT20H.86A.0009.2017.0224.1346
- **OS installed:** Lubuntu 18:10 Desktop 64-bit

The Lubuntu operating system is a light weight build of the more popular Ubuntu operating system. Both are Linux distributions, however, Lubuntu was chosen because it is less demanding on the hardware, which is a preferable characteristic for SEE testing of microprocessors as discussed earlier.

# 3.3. Test Setup

## 3.3.1. SEE Test

SEE testing was carried out at the Neutron Therapy Vault of the NRF iThemba Labs in Cape Town, South Africa. The separated sector cyclotron at the facility is capable of accelerating proton beams to a maximum kinetic energy of 200MeV [62]. Testing was carried out in open air at a single beam energy of 55.58 MeV.

Referring to Figure 3.5 and Figure 3.6 , the board is mounted on an XY Table that was developed by Space Commercial Services (SCS). The XY Table, operated remotely, allows for vertical and horizontal positioning of the DUT with respect to the proton beam. The table also allows for the angle of incidence of the proton beam on the DUT to be varied, though, this feature was not utilized, and all tests were carried out with the DUT perpendicular to the proton beam. Additionally, a cooling fan was mounted on the table to blow air across the processor to keep it cool during testing. This is because the heatsink that came mounted on the processor was removed in order to expose the die to the proton beam unhindered.

The board received power from a 12V DC power supply unit. A shunt resistor and a relay were both connected in series with the board. The shunt resistor formed part of the current measurement instrumentation while the relay was used for remotely power cycling the board if necessary. A National Instruments CompactDAQ – 9184 Chassis (cDAQ-9184) with two modules installed also formed part of the setup. Of the two installed modules, one was the NI 9205 that is a voltage measurement device and the other was the NI 9403 that is a digital general-purpose input-output device. The NI 9205 was connected (in the Non-Referenced Single Ended configuration [63]) to measure the voltage drop across the shunt

resistor. The current draw of the board would then be determined from obtained voltage readings. The NI 9403 on the other hand was connected to the control pins of the relay. Figure 3.9 provides a schematic of this layout.

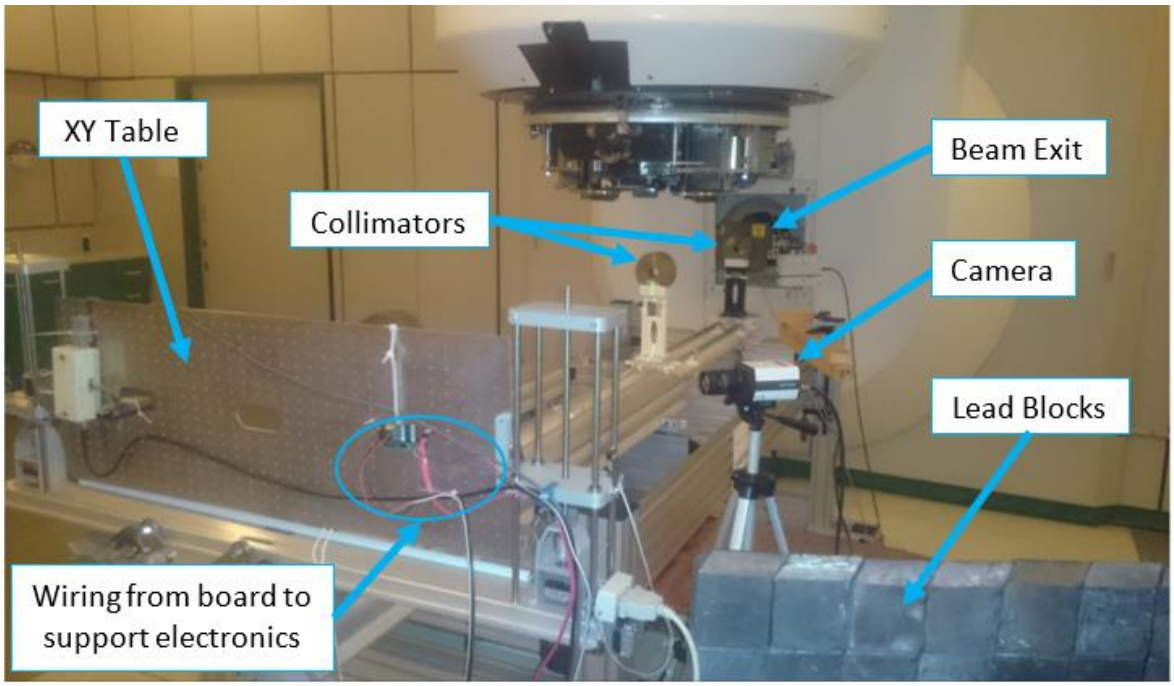



*Figure 3.5: SEE Setup in the Neutron Therapy Vault at NRF iThemba Labs*

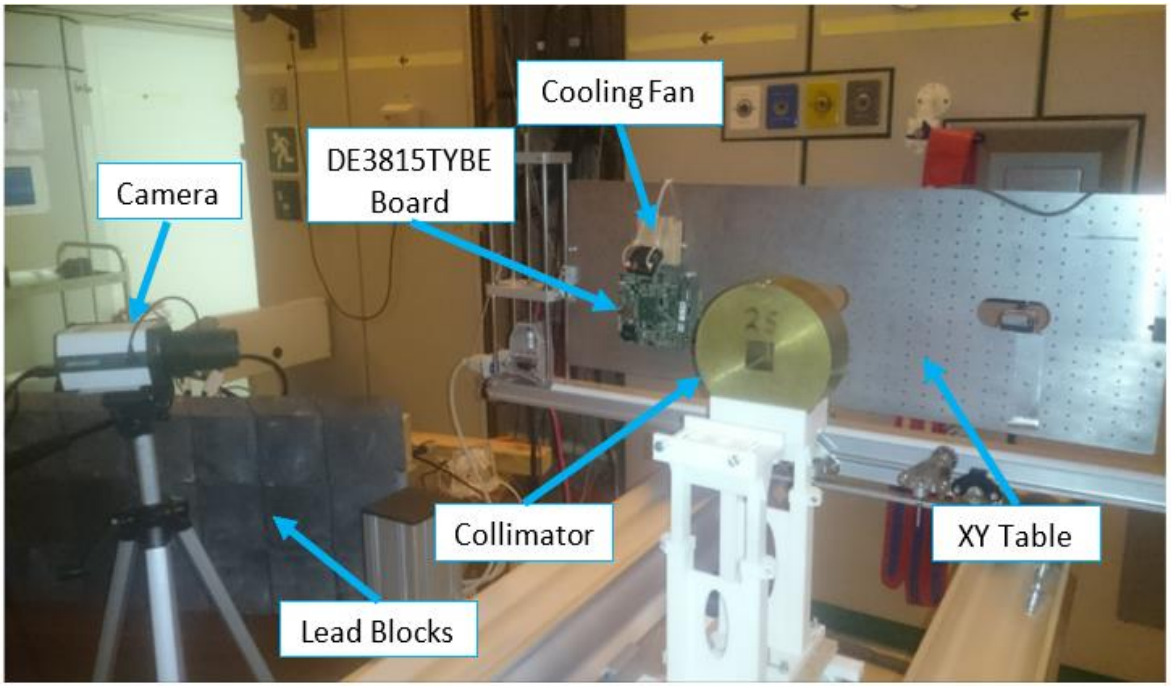



*Figure 3.6: SEE Setup in the Neutron Therapy Vault at NRF iThemba Labs (different angle)*

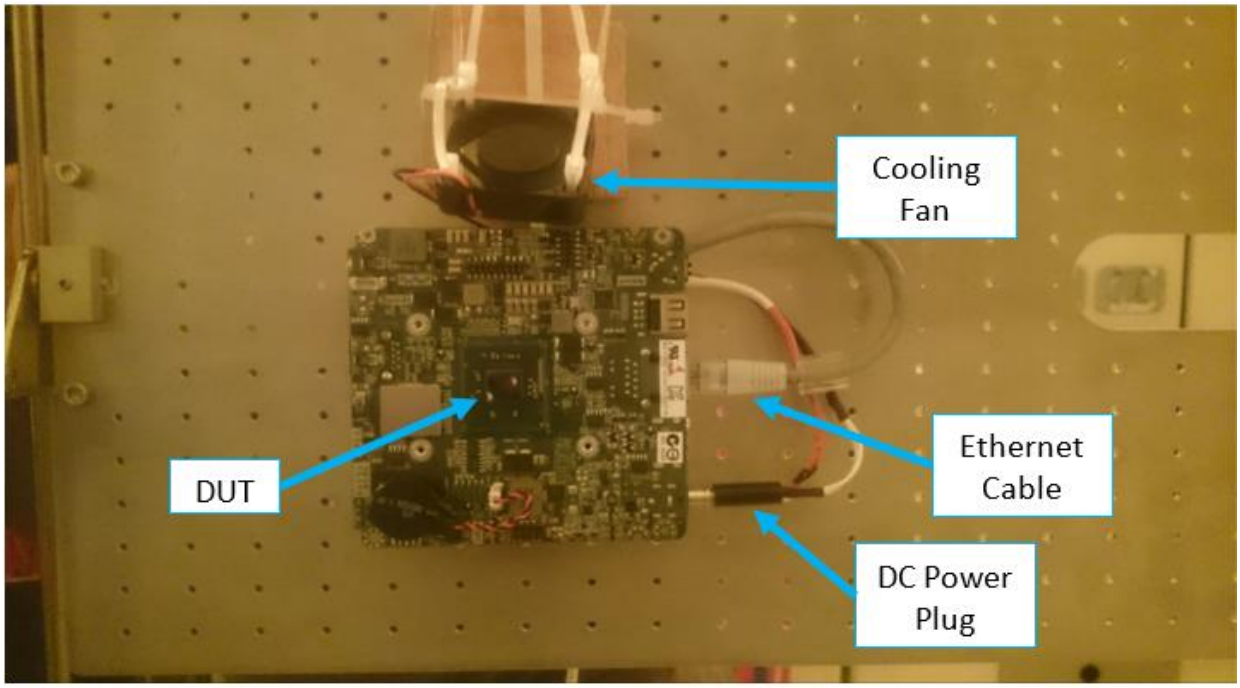



*Figure 3.7: Closeup photo of the board mounted on the XY Table*

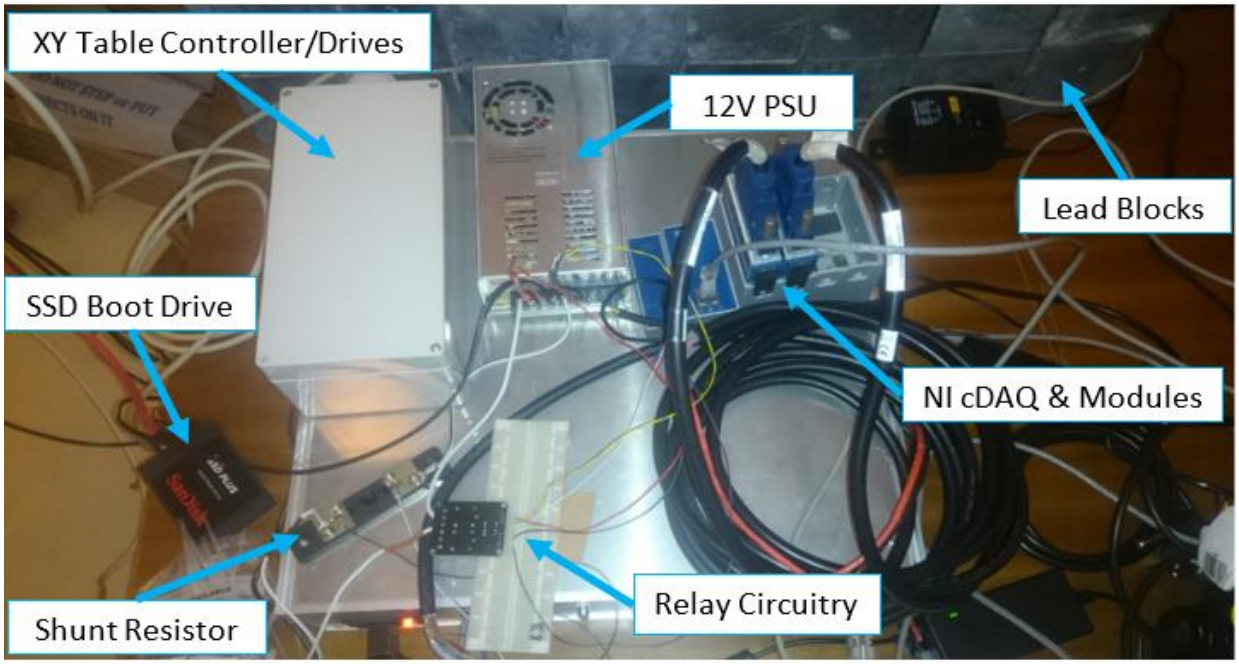



*Figure 3.8: Support Electronics (behind lead blocks). Network switch not shown.*

A barrier consisting of interlocking blocks of lead was used to shield the support electronics from any scattered protons and/or secondary radiation produced during the test. Inclusive of these electronics was the SATA SSD boot drive of the board in which the OS was installed. By design, the test software would record any errors detected during testing to the drive (the single self-testing computer approach described in section 3.1.1.1 was used).

The board, as well as the cDAQ and control circuitry for the XY table were all connected to a network switch via Ethernet cables. This network switch was also connected to other network switches in the facility that subsequently reached the control room.

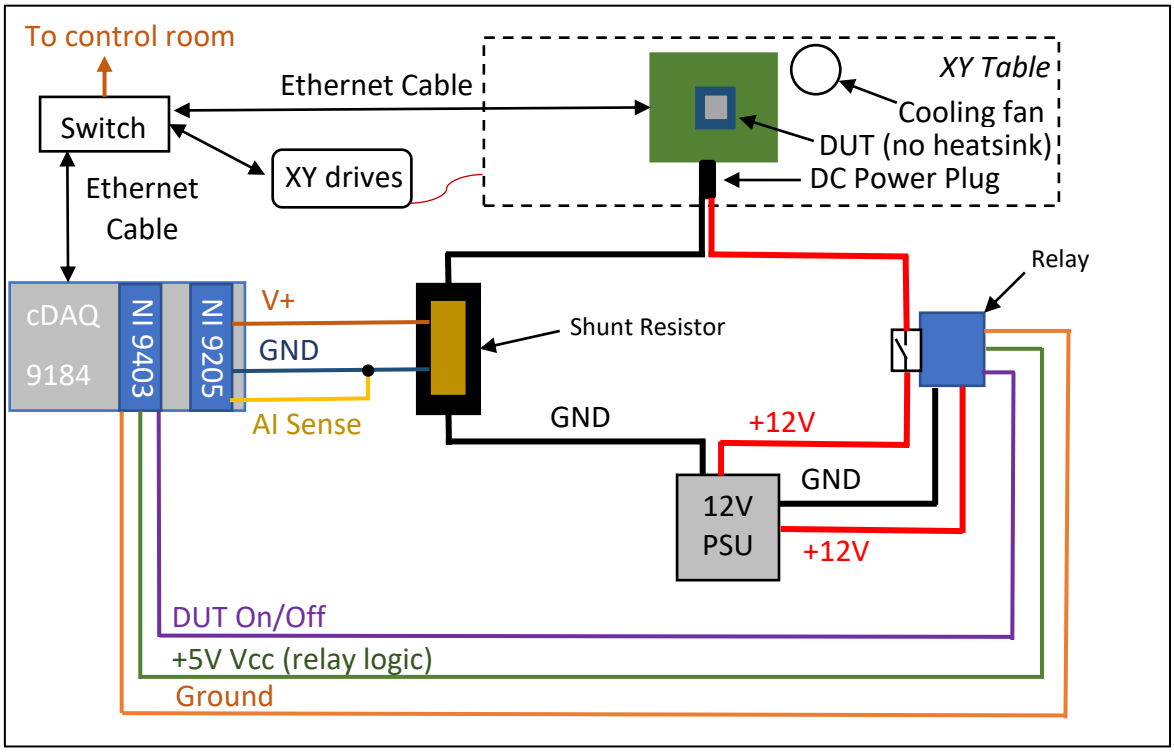



*Figure 3.9: Wiring Diagram for SEE Setup (SSD Boot Drive not shown)*

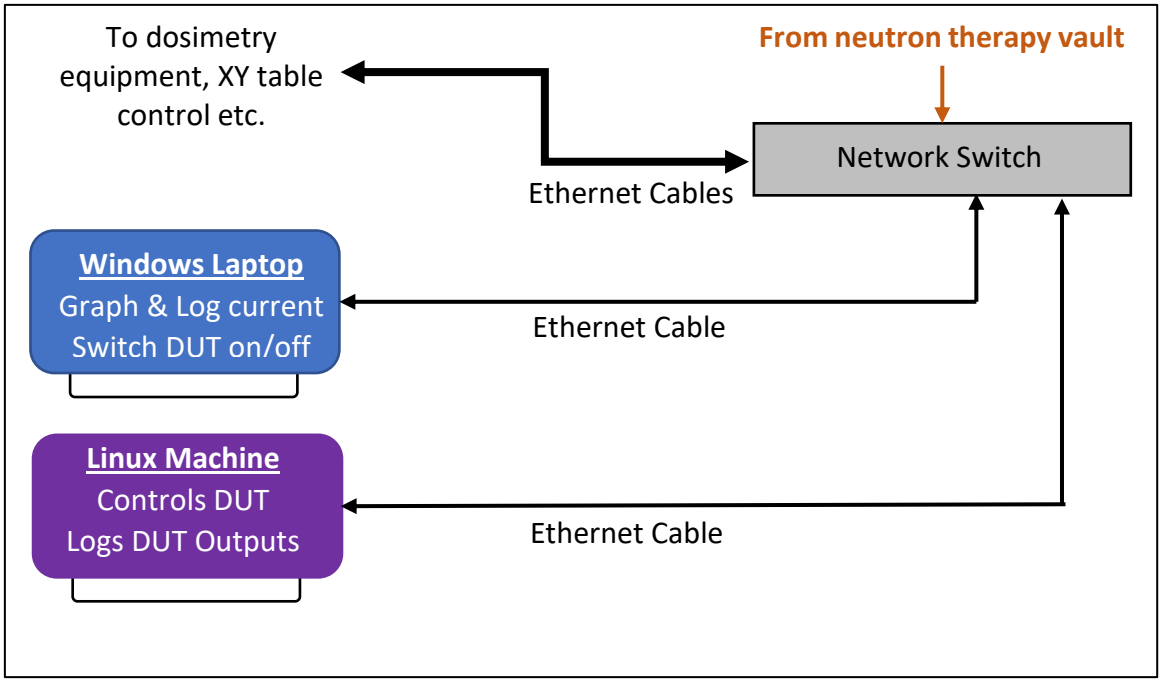



*Figure 3.10: Network Device map in control room for SEE Setup*

In the control room was a windows laptop running a LabVIEW Virtual Instrument (VI) that communicated to the cDAQ through the network. From this machine, the current draw of the board was monitored and manual power cycling. Additionally, there was a DE3815TYBE

NUC board running the same version of Lubuntu as the test board. This board was used to control the test-board via a Secure Shell (SSH) terminal through the network as well as displaying the outputs of the test software on a screen. Dosimetry and control of the XY table were also done through the network.

### 3.3.1.1. Data Acquisition

As discussed, the NI 9205 module installed on the cDAQ-9184 was used for the current measurements while the NI 9403 was used to control the relay that power cycled the test board on user command. Figure 3.11 and Figure 3.12 are from the LabVIEW VI created to accomplish these tasks.

The centre of the front panel of the VI is dominated by a waveform chart that would display current measurements in real time while running (100 samples taken at 1kHz every 200ms). The plotted value would be the quotient of the voltage measured across the shunt resistor and the resistance of the shunt resistor (0.005Ω). Additionally, all measured values would be appended to a log file if the user chose to do so. The VI also has indicator LEDs to show if the DUT is currently receiving power and whether or not measurements are being written to file. Finally, there were counters that the user could manually increment to keep track of the number of SEFI or SEL encountered.

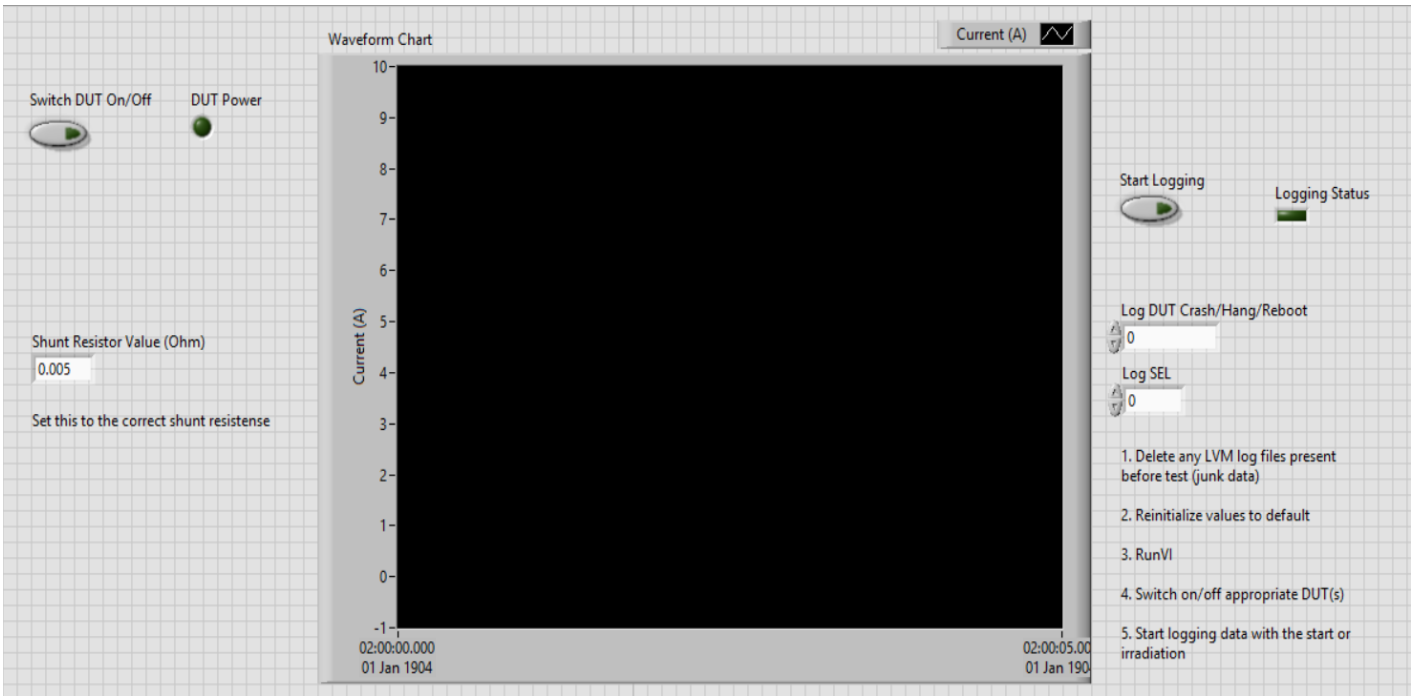



*Figure 3.11: Front panel of the LabVIEW VI used in SEE testing*

As for the DUT and the test software it was running, data was recorded in a number of ways. The first, mentioned earlier, was by the test software logging errors to a file on the boot drive. The second was in the control room. The Linux machine used to control the DUT via SSH would display the same outputs from the test software on the on-screen terminal. These outputs would be recorded locally by screen capture software. The screen recordings

would serve as backups in case the boot drive failed. Also, a manually handwritten log of events was kept for events like program crashes and SEFI that the DUT could not record.

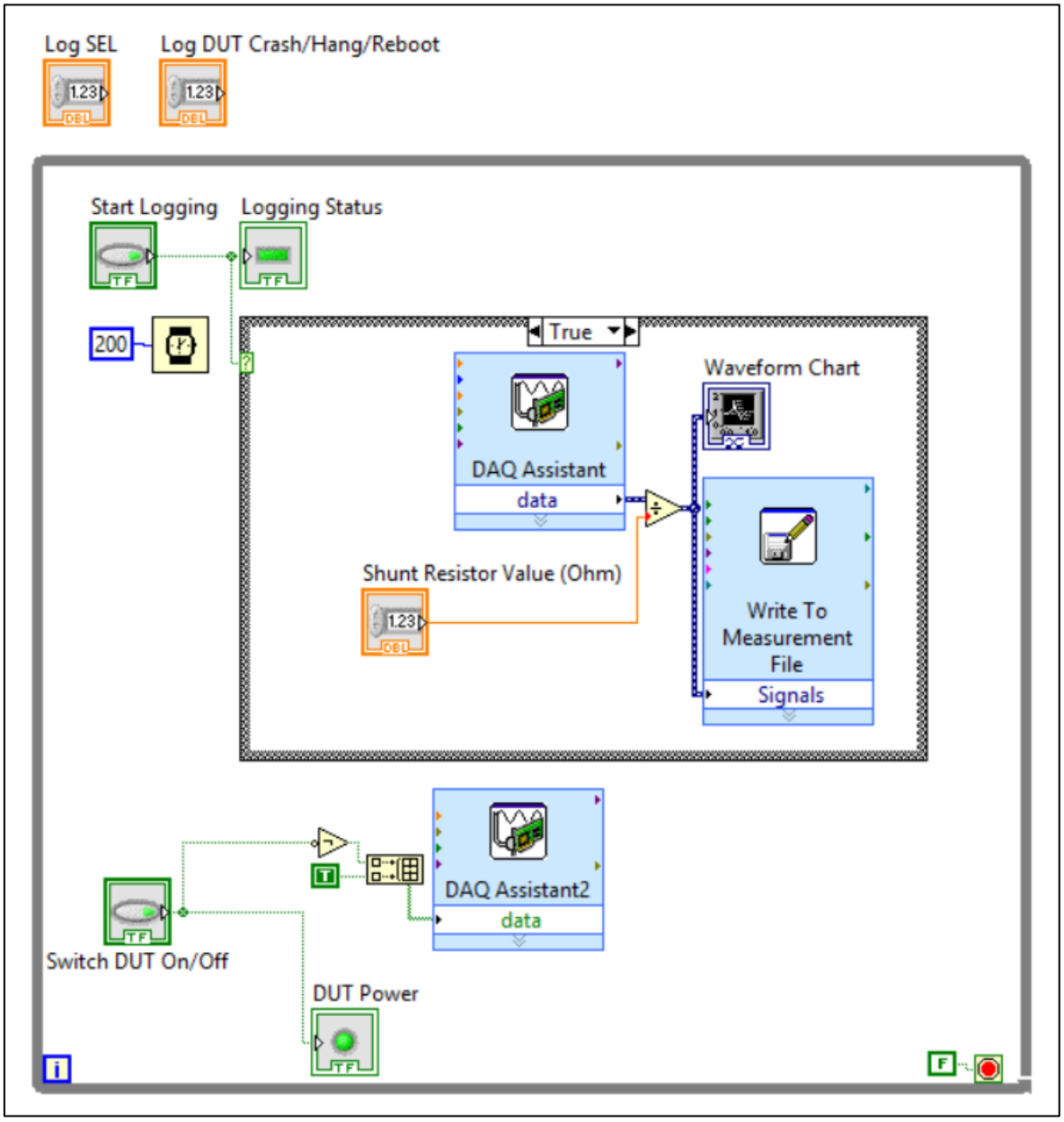



*Figure 3.12: LabVIEW code for the VI used in SEE testing*

### 3.3.1.2. Test Software

If knowledge of the fundamental response of registers to radiation is known, it is possible to predict the expected behaviour of different types of applications running on a particular device [38]. With this in mind and given that different types of registers may have different sensitivities to upsets [38], several test programs were developed, each focusing on testing a specific bank of registers. The register banks tested were the GPRs, MMX and XMM (refer to Figure 3.2). The 64-bit wide MMX bank was chosen over the 80-bit wide FPU since each MMX register is individually addressable, while the FPU is only accessible through the top of the stack (FPR0) [59]. This means that any FPU operations carried out during testing would consistently rewrite the values in the FPU registers, potentially correcting any errors caused by radiation before they were detected. It was expected that a sensitivity of MMX registers alone would be indicative of the performance to expect from applications that are FPU intensive and/or MMX intensive since MMX and FPU share the same physical register space.

Additional programs used that utilized the Arithmetic and Logic Unit (ALU) of the processor included a generic CPU workload benchmark (Sysbench) installed from the Ubuntu repository and a "math test" that was written to carry out the following calculation and compare it with the known result:

$$\cos(\sin\left(\sin\left(\sqrt{2}(\sin(\cos(\sin(\cos(16032001e^{\pi})))))\right)\right)) \qquad \textit{Eqn 3.1}$$

Any correlation between the results obtained from the "math test" program and the sensitivity of the MMX registers was also to be investigated.

Finally, a Linux kernel module was written to allow for enabling or disabling of the on-board cache in runtime. These programs were meant to run in scenarios where the on-board cache was disabled, as well as scenarios where it was enabled. Results obtained from both scenarios would then be compared to determine the influence that the on-board cache had on the device sensitivity.

Ideally, one would want to test most, if not all of the processor circuitry. However, factoring in the complexity of the device, manpower available to create test software and beam time available, this would not be feasible. Therefore, the test programs mentioned would test circuitry that was understood to be of relatively high importance, among other reasons, such as, addressability and feasibility (for instance modifying the control registers would interfere with the self-testing capabilities of the device therefore control register testing was excluded).

The test programs were written using a combination of the C programming language and extended inline assembly calls (i.e. assembly code embedded in the C code). The "volatile" modifier was used whenever making an inline assembly call so as to instruct the compiler not to optimize the code [64], potentially changing how it behaves. The algorithms implemented share similarities with those presented in the papers [38, 51, 65] but also differ in a number of ways. They shall now be described.

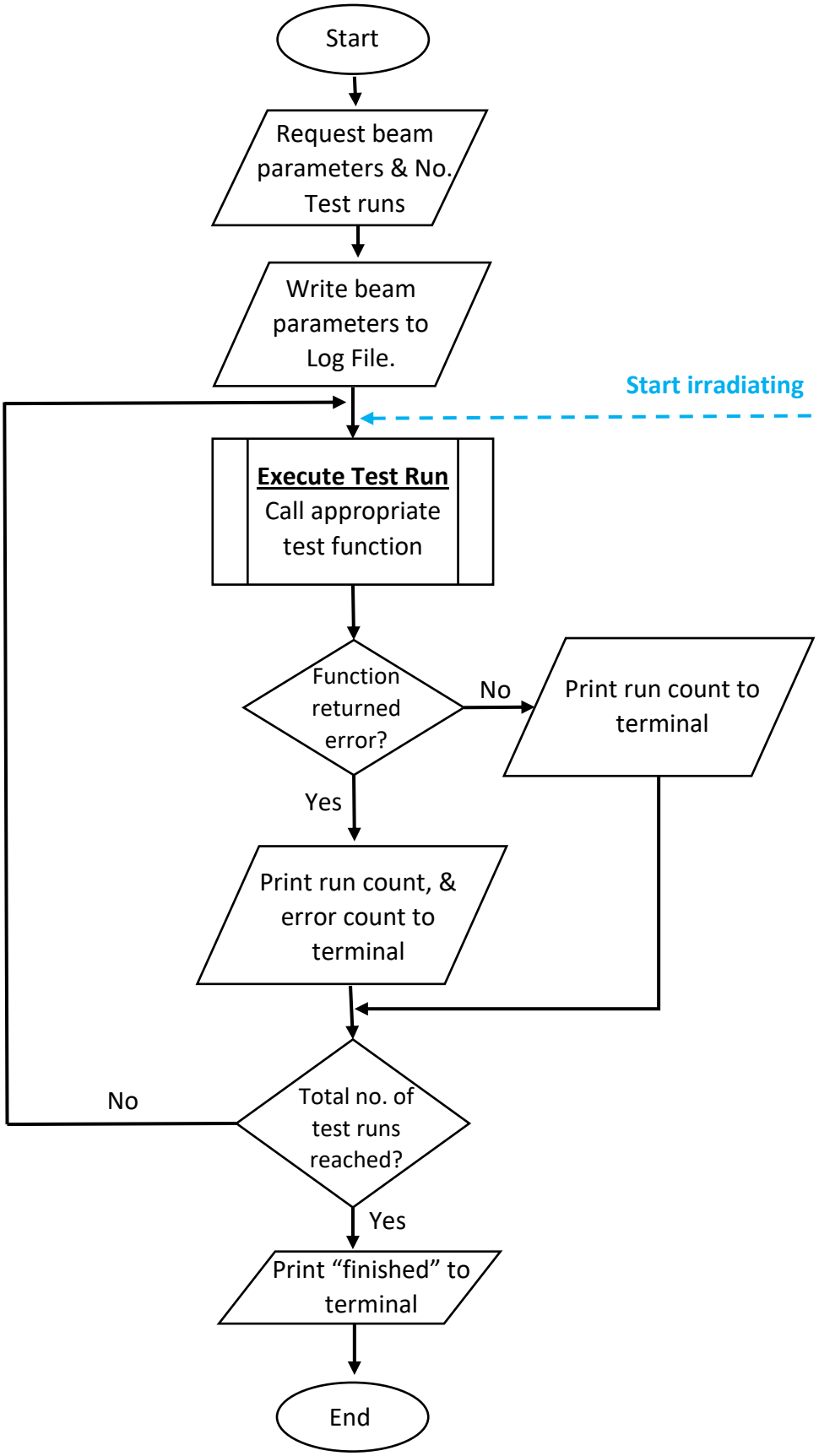



*Figure 3.13: General steps taken by the main function of the developed test programs*

Figure 3.13 shows the form of the main function of each test program created. When initiated, the program would prompt the user to input the beam characteristics and number of test-runs to perform. The program would then proceed to append a log file stored in the SSD with a new heading indicating the beam parameters.

Within the program, a test run is defined as the successful execution of the appropriate test function. For example, if the function for testing the GPRs is called and it executes to completion, a single test run will have elapsed. Through each test run, the run count and details of what the test function returned are printed to terminal. This is important since it allows the operator to easily recognize a system hang (and other anomalous behaviour) as well as providing a redundant log of events.

Each test function returned a value to indicate whether or not an error had been detected, and in the case of the GPR test function, the type of error would also be specified (e.g. whether it was an upset or a write fail). On top of this, each test function would create an error report that would both be written to the log file as well as displayed on terminal. For instance, the GPR test error report, intermediately stored in program memory (RAM), consisted of the following:

- Location of the register where the error was detected
- The value stored in the register used as the baseline for comparisons
- 3 copies of the value stored in the register that is being reported

The decision to output 3 copies of the erroneous value to RAM was a soft implementation of triple modular redundancy. This way, during analysis of the results, it would be easier to determine whether the reported value was the actual value stored in the reported register, or an erroneous value due to an upset occurring in RAM. The “true” value would be the majority voted one.

#### 3.3.1.2.1 GPR Test Function (Pseudocode)

Refer to Figure 3.2 on page 32 for register names. Registers shall be prefixed by a % in the pseudocode:

```
1.  RESET the upset-detected flag and failed-to-write flag in RAM
2.  WRITE 0xf0f0f0f0f0f0f0f0 to %rax from RAM //This is the baseline register
3.  WRITE [NumberOfScanCycles] to %rbx
4.  COPY %rax to %rcx
5.  IF %rax not equal to %rcx          //If the value failed to write
6.      COPY %rax to %rcx
7.      IF %rax not equal to %rcx
8.          COPY %rax to %rcx
9.          IF %rax not equal to %rcx
10.             GENERATE error report.
11.             SET failed-to-write flag in RAM
12.             PRINT report to terminal
13.             APPEND report to logfile
14.             RETURN to main with write-fail signal
```

15. END IF
16. END IF
17. END IF
18. REPEAT step 4 - 17 for each subsequent GPR in place of %rcx excluding %rbp & %rsp
19. BEGIN LOOP to scan registers for changes. Counter in %rbx, decrementing
20. IF %rax not equal to %rcx
21. GENERATE error report
22. SET upset-detected flag in RAM
23. PRINT report to terminal
24. APPEND report to logfile
25. RETURN to main with upset-detected signal
26. END IF
27. REPEAT step 20-26 for each subsequent GPR in place of %rcx excluding %rbp & %rsp
28. END LOOP
29. PRINT "no error detected" to terminal
30. RETURN to main with no-error signal

The source code for the GPR test program can be found in Appendix 2: "GPR Test Source Code". The registers RBP and RSP are excluded from the test since modifications to them by the program result in the program crashing with a segmentation fault. The RAX register is used to store the baseline value of 0xf0f0f0f0f0f0f0f0 and for comparisons with subsequent registers. The baseline value was chosen since it fills the entire 64-bit wide register with an even alternating pattern of 1111 and 0000 in binary. This would be useful in investigating upset asymmetry (i.e. 1 → 0 upsets versus 0 → 1 upsets).

In tests with the cache on, the value written to RBX was 150,000,000 and with cache off, the number was 3,000,000. These numbers represent how many times the entire register bank would be scanned for errors and were chosen such that each test run (defined in the flowchart) would take about 5s to execute. Reason for this was to ensure a roughly predictable update rate on the terminal which made it easier for the operator to spot anomalies such as system hangs. Keep in mind that with cache off, the system operates sluggishly and without this being done, the user could improperly assume that the system had hanged.

In the register initialization phase of the function, if a register fails to have the correct value written to it on the first attempt, a second attempt will be made. If the second attempt fails, a third will be made. If this still fails, an error report will be generated. If the same register fails the same way for subsequent test runs, it would be indicative of a stuck bit(s), in which case the program would be modified to exclude this register from the test. This is because the sequential order of execution will result in error reports being dominated by that one failed register, especially if it is one of the first registers to be initialized.

#### 3.3.1.2.2 MMX Test Function (Pseudocode)

The source code for the MMX test program can be found in Appendix 3: "MMX Test Source Code". The pseudocode for the function follows:

1. WRITE [NumberOfScanCycles] to %rdx
2. WRITE 0xf0f0f0f0f0f0f0f0 to %r8, %r9 and %r10 from RAM //These are the baselines
3. WRITE 0xf0f0f0f0f0f0f0f0 to %mm0 from RAM
4. COPY %mm0 to %mm1 – %mm7
5. BEGIN LOOP to scan registers for changes. Counter in %rdx, decrementing
6. COPY %mm0 to %rax, %rbx and %rcx
7. IF (%r8 not equal to %rax) AND (%r9 not equal to %rbx) AND (%r10 not equal to %rcx)
8. GENERATE Error report
9. PRINT report to terminal
10. APPEND report to logfile
11. RETURN to main with error-detected signal
12. ELSE
13. Repeat step 6 – 12 for each subsequent MMX register in place of %mm0
14. END IF
15. END LOOP
16. PRINT "no error detected" to terminal
17. RETURN to main with no-error signal

There are no compare opcodes/instructions that operate on MMX that also allow for branching statements like in GPRs. For this reason, if any decision is to be made depending on the contents of a given MMX register, the value in the MMX register must first be copied to a GPR. The comparison and branch instructions are then executed using this GPR.

In the MMX test function, 3 GPRs (R8, R9 and R10) store 3 copies of the baseline value. Another 3 GPRs (RAX, RBX and RCX) are used as temporary storage to hold 3 copies of the value within the MMX register currently undergoing inspection. An error in the MMX register is deemed to have occurred if the values of all three of the following register pairs differ from each other: RAX and R8, RBX and R9, RCX and R10, in which case an error report is generated. The report in this case returns the value in the current MMX register and the value in the three GPRs used as temporary storage, as well as the location of the MMX register.

The use of 3 GPR pairs in the comparison rather than 1 reduces the probability of a false positive error report in case an upset occurs in a GPR. All 3 pairs are guaranteed to mismatch if the value copied from the MMX register is different from the baseline, however, the same cannot be assumed if a mismatch is caused by upsets in only a few of the GPRs.

There, however, still is a possibility for false positives to occur if the GPRs are experiencing a high enough upset rate. For these cases, false positives would have to be distinguished from real MMX upsets during the analysis of the error reports obtained from testing.

Finally, for similar reasons as those given in the GPR test function, the number of scan cycles with cache on was 150,000,000 and 1,000,000 with cache off.

#### 3.3.1.2.3 XMM Test Function (Pseudocode)

Source code for this can be found in Appendix 4: "XMM Test Source Code". The XMM registers, like the MMX registers, do not have any compare instructions that allow for branching/decisions. Consequently, the same approach used in the MMX test function will be used for XMM. Also, since XMM registers are 128-bits wide, double the number of GPRs will be required to store the contents of a single XMM register. The "L" and "H" suffixes are used to specify the lower 64-bits and the higher 64-bits of the specified XMM register (respectively) in the pseudocode that follows:

1. WRITE 0xf0f0f0f0f0f0f0f0 to %rax, %rbx and %rcx from RAM //These are the baselines
2. WRITE [NumberOfScanCycles] to %rdx
3. WRITE 0xf0f0f0f0f0f0f0f0 to %xmm0L from RAM
4. WRITE 0xf0f0f0f0f0f0f0f0 to %xmm0H from RAM
5. COPY %xmm0 to %xmm1 - %xmm14 //Notice we haven't included %xmm15
6. BEGIN LOOP to scan registers for changes. Counter in %rdx, decrementing
7. COPY %xmm0L to %r10, %r11 and %r12
8. COPY %xmm0H to %xmm15L //xmm15 is used as swapping space
9. COPY %xmm15L to %r13, %r14 and %r15
10. IF (%rax not equal to %r10) AND (%rbx not equal to %r11) AND (%rcx not equal to %r12)
11. GENERATE Error report
12. PRINT report to terminal
13. APPEND report to logfile
14. RETURN to main with error-detected signal
15. ELSE IF (%rax not equal to %r13) AND (%rbx not equal to %r14) AND (%rcx not equal to %r15)
16. GENERATE Error report
17. PRINT report to terminal
18. APPEND report to logfile
19. RETURN to main with error-detected signal
20. END IF
21. END IF
22. Repeat step 7 – 21 for each subsequent XMM register in place of %xmm0 excluding %xmm15
23. END LOOP
24. PRINT "no error detected" to terminal
25. RETURN to main with no-error signal

The register XMM15 was used differently in the function compared to XMM0-XMM14. This is because XMM registers can only copy the lower 64-bits to a GPR but not the higher 64-bits. XMM15 (or rather XMM15L) was therefore used as a swapping register that would keep getting overwritten by the higher 64-bits of the register currently being inspected. This way, the number of rewrites occurring on XMM0 - XMM14 are minimized.

Error reports generated by this function would include all 128-bits of the XMM register that is being reported, the contents of the temporary registers (R10, R11, R12 for the lower 64-bits and R13, R14, R15 for the higher 64-bits) and the location of the register being reported. With cache on, the number of scan cycles was 150,000,000 and with cache off it was 35,000.

#### 3.3.1.2.4 Math Test Function (Pseudocode)

Source code for this can be found in Appendix 5: "Math Test Source Code". Pseudocode follows:

```
1. CALCULATE equation result
2. IF result is correct
3.     PRINT to terminal "ok"
4. ELSE
5.     PRINT to terminal "Mismatch"
6.     APPEND to logfile "Mismatch"
7. END IF
```

This was the simplest function implementation to check for errors in the ALU. The error report would simply be a count of the number of incorrect/unsuccessful calculations.

#### 3.3.1.2.5 Sysbench Benchmark

This is a simple benchmark tool that benchmarks the processor by validating prime numbers. At the end of a run it prints to terminal the performance of the processor.

The terminal command entered to run the benchmark is as follows:

*sysbench --test=cpu --cpu-max-prime=20000 run*

#### 3.3.1.2.6 Cache Disable/Enable Kernel Module

Disabling the processor cache would be achieved by setting bits 29 and 30 of the control register (cr0) to 1 [66]. Since modification of the control registers of the processor is only allowed to be done at user level 0, a Linux kernel module had to be created.

The source code, that uses the same procedure as [67], can be found in Appendix 6: "Cache Disable/Enable Kernel Module" and the Makefile in Appendix 7: "Makefile for Cache Disable/Enable Kernel Module". Once compiled, all one had to do to disable the cache was insert the module into the kernel (as root) using the "insmod" command. To re-enable the cache, the module would have to be unloaded using the "rmmod" command.

## 3.3.1.3. Testing Procedure

The first thing to do was to measure and verify the beam spot size and uniformity. This was done by Mr. Arno Barnard of Stellenbosch University (a collaborator) together with members of iThemba staff. All the while, the board was in the beam shadow cast by the collimators.

On completion, the beam was shut off and the XY table was commanded to move the DUT into position. The operator on the controller Linux machine then initiated the execution of test programs on the DUT. Beam at 2nA current was then switched on and as the DUT was

undergoing irradiation, the terminal outputs as well as the current draw were monitored on screen.

If a test program completed without detecting a significant number of errors, it would be re-invoked as quickly as possible and made to execute for a longer period of time. If a large current spike was noted or the system hanged or rebooted, the beam would be switched off and the board power cycled and given enough time to resume program execution before the beam was switched on again. This was to minimize accumulation of fluence without results. This procedure was repeated for beam currents of 5nA, 15nA, 20nA and momentarily for 30nA, though this latter case was preceded by irradiation of a different device while the DUT was in the beam shadow once more.

All testing was done with the DUT operating at stock frequency since the BIOS did not allow for modification of the CPU multiplier.

### 3.3.2. TID Test

TID testing was carried out at FruitFly Africa in Stellenbosch, South Africa. The facility utilizes a cylindrical cobalt-60 gamma radiation source to sterilize fruit fly male pupae before releasing them back to the wild. This is done for population control of the international quarantine pest [68].

Typically, when in use, containers that contain the pupae are placed on the motorized turntable highlighted in Figure 3.14. When switched on, this turntable rotates about the centre cylinder that is oriented vertically. It is within this vertical cylinder that the cobalt-60 source is mechanically raised from or lowered to its underground storage location. Control of the movement of the source is done remotely from outside the irradiation chamber.

The gamma radiation emanating from the source follows the inverse square law, which means that the rate of dose delivered by the radiation to a target varies inversely with the square of the distance between the target and the cobalt-60 source.

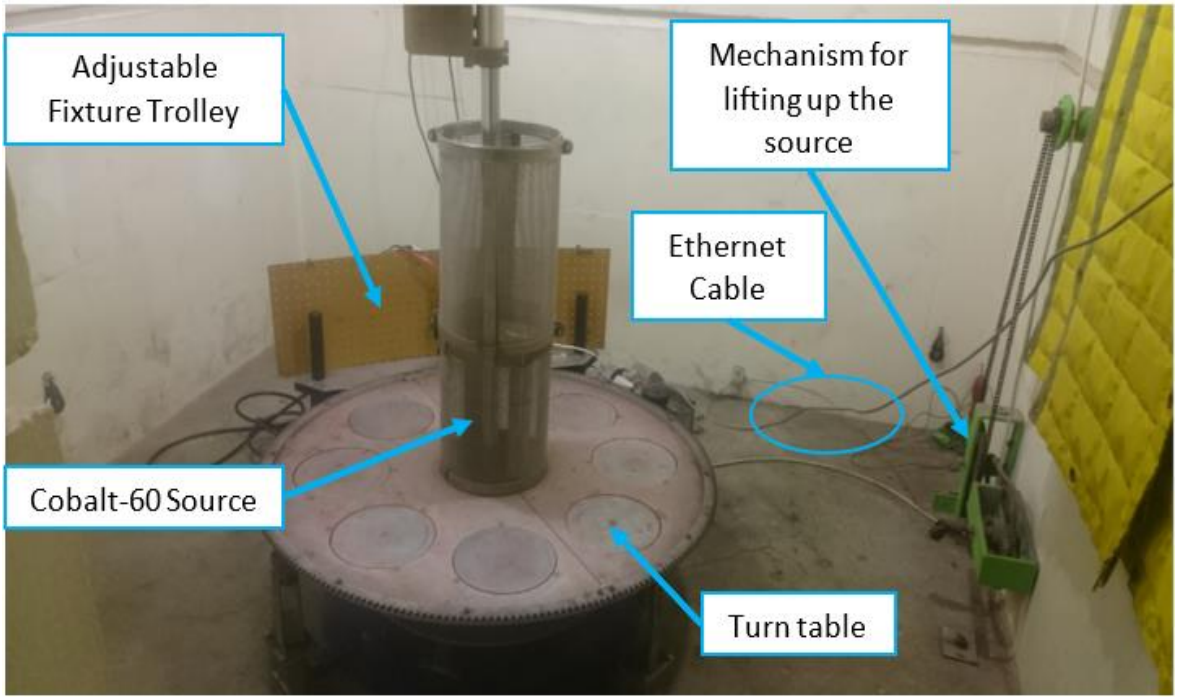



*Figure 3.14: Irradiation chamber at FruitFly – Stellenbosch. In this photo, the Cobalt-60 source is still underground*

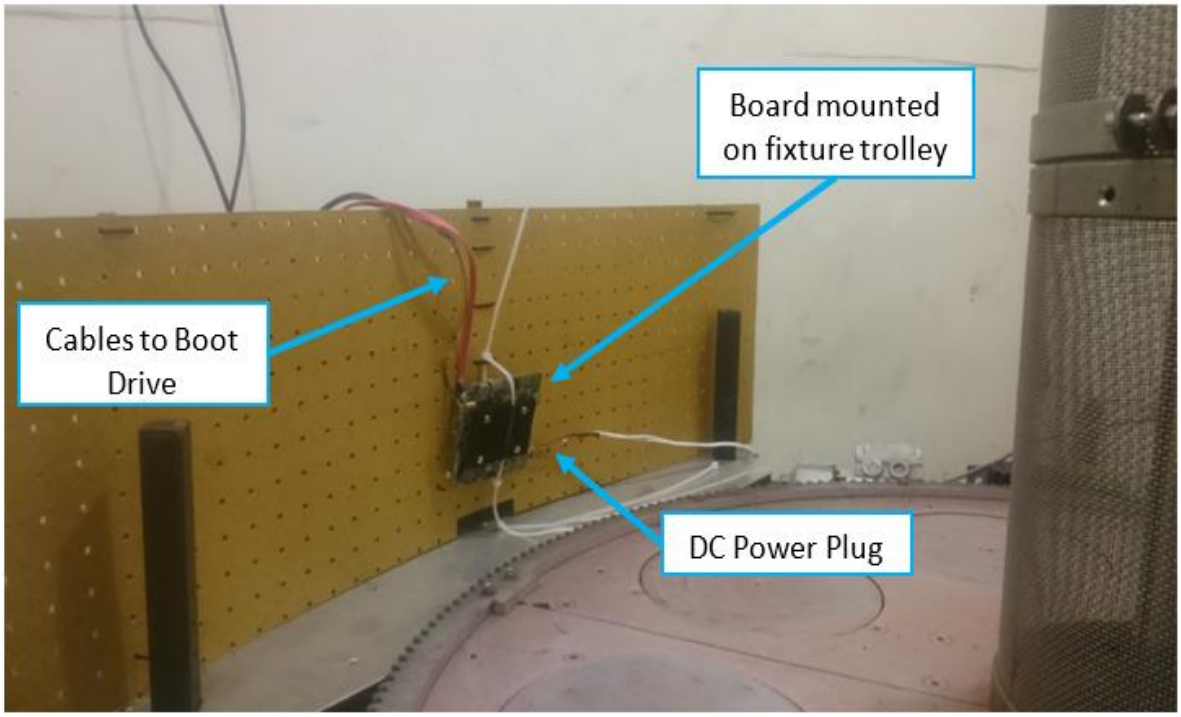



*Figure 3.15: TID test setup in the irradiation chamber at FruitFly*

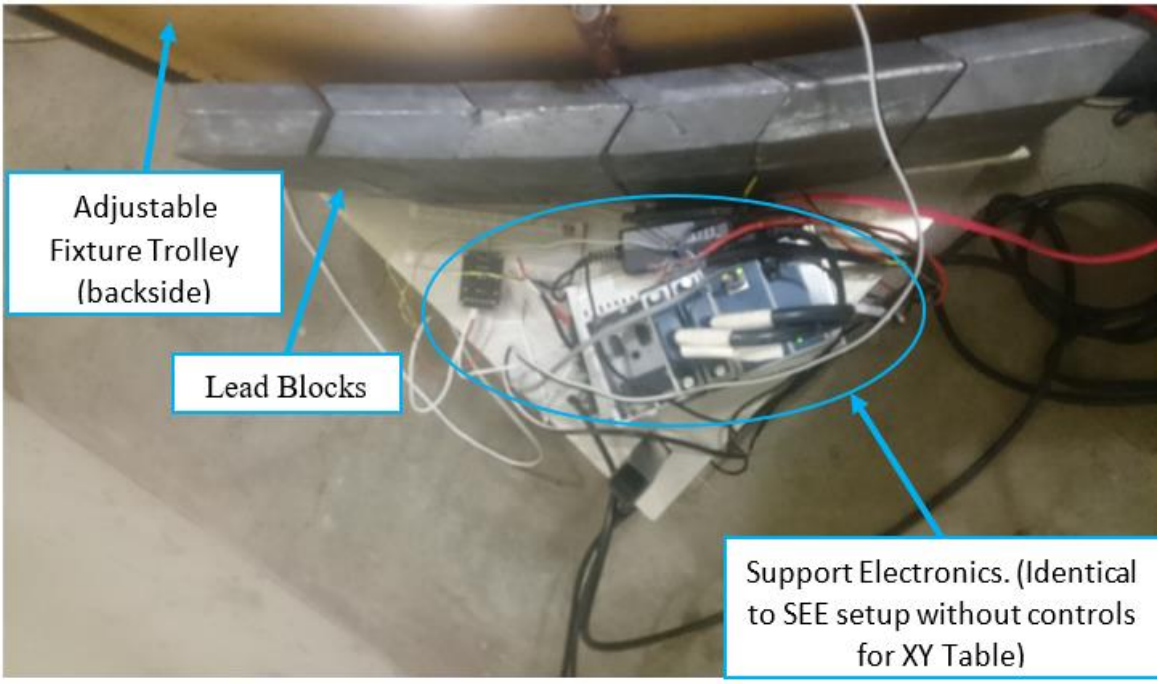



*Figure 3.16: TID test setup in the irradiation chamber at FruitFly (different angle showing support electronics)*

The setup used for TID testing is shown in Figure 3.15 and Figure 3.16. The new board undergoing testing (not the same one used for SEE testing) was mounted on a fixture trolley that was placed beyond the turntable. This trolley is adjustable such that it allowed for the distance between the board and the cobalt-60 source to be varied, effectively allowing for the dose rate to be chosen. It was desired for the DUT to receive a total dose of at least 100 kRad (many missions have been required to survive this dose [29]). This as well as the time available to carry out the test meant that the required dose rate was approximately 10 kRad/h. Utilizing a spreadsheet provided by a collaborator [69], it was determined that the DUT should be placed 57.95 cm from the source to give a dose rate of 9.7 kRad/h.

At the back of the trolley was a protective barrier formed by a series of interlocking lead blocks. These served the same purpose as in the SEE setup, they protected the support electronics from getting damaged by the radiation. The support electronics were also set up in a similar manner to the SEE setup however the test board was not connected to an ethernet network and the cDAQ was directly connected to a windows laptop that was located outside the irradiation chamber. The ethernet cable that connected the cDAQ to the windows laptop left the irradiation chamber through a cable duct that ran through the 1m thick concrete wall of the chamber. Lead cylinders with small cut outs for cable runs were inserted on either end of this cable duct so as to minimize the amount of radiation and ionized air that could travel to the operator side. Other differences with the SEE setup are the absence of the XY table and cooling fan, and the presence of a heatsink on the DUT. The heatsink was not removed since it would not hinder the gamma radiation from reaching the DUT [70]. A schematic of the setup is given in Figure 3.17.

During the TID test, the DUT was set to automatically be running the Sysbench benchmark at intervals of 2 minutes whenever the machine booted up. This would ensure a consistent

load on the processor throughout the test. The board BIOS had also been set to auto-reboot if for whatever reason the board lost power and then the power came back on again.

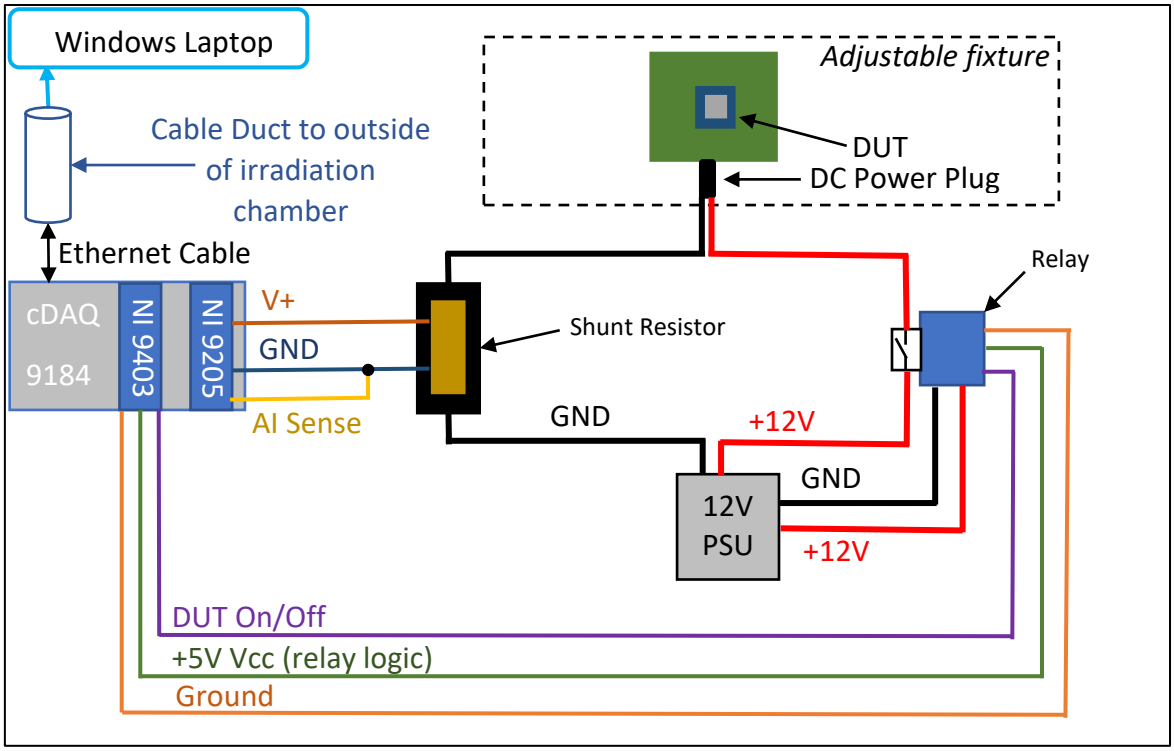



*Figure 3.17: Wiring Diagram for TID Setup (SSD Boot Drive not shown)*

### 3.3.2.1. Data Acquisition

Data acquisition for the TID tests was done using the NI 9205 installed on the cDAQ chassis. The module, like in the previous test, was measuring the voltage across the shunt resistor. This value would be divided by the resistance of the shunt thus yielding the current draw of the board which would both be graphed and appended to a log file.

The LabVIEW VI that had been used for the SEE test was reused however it had been modified to record similar data for 4 additional devices undergoing separate unrelated tests. Measurements for these devices were made using different channels of the NI 9205. Another modification made to the VI was the frequency at which measurements were made. Since the test was intended to run for approximately 10 hours, the VI was set to read 100 samples at 1kHz every 1 minute, opposed to every 200ms as in the SEE test.


*Figure 3.18: Front panel of LabVIEW VI used for TID testing*

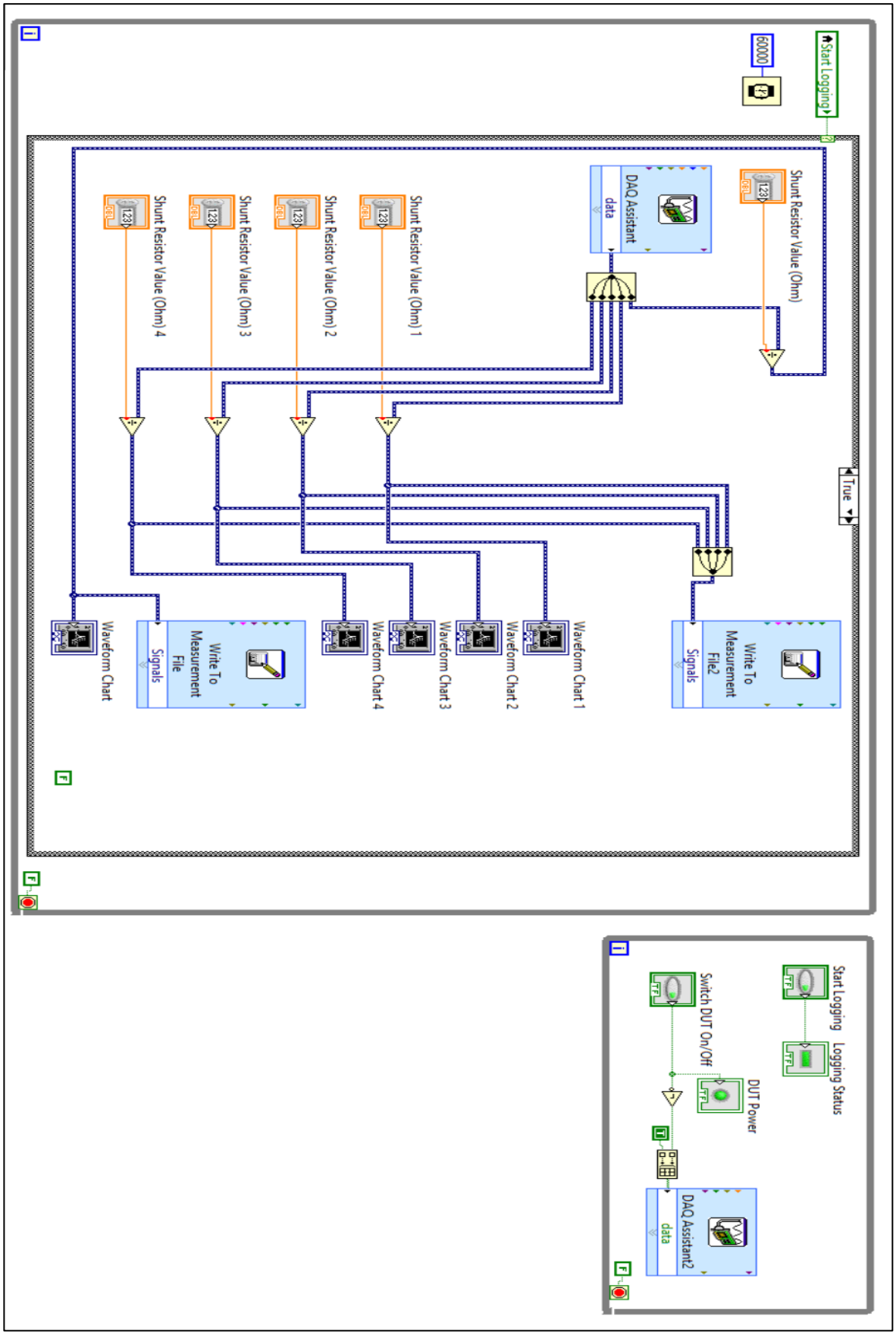



*Figure 3.19: LabVIEW code for the VI used in TID test*

### 3.3.2.2. Testing Procedure

The testing procedure for TID was rather straight forward, especially when compared to the procedure used for SEE testing. Once the setup was complete and it was verified that everything was operational, the entrance to the irradiation chamber was sealed. The user then initiated logging of the DUT current draw on the LabVIEW VI. At this point, the cobalt-60 source was raised from its underground storage and irradiation of the DUT begun. The test was then left to run for the predetermined amount of time.

After the test period elapsed, the cobalt-60 source was lowered back underground and the air in the irradiation chamber (now mostly composed of ozone) was pumped out. Logging of the current draw of the DUT was also stopped at this point.

The test board was then removed from the chamber and allowed to anneal at room temperature for no less than 196 hours, after which performance was checked.

# 4. Results

This chapter gives a description of the observations made during experimentation and presents data that was obtained. Methods adopted to analyse said data are also described, with examples given.

## 4.1. SEE Results

Characterization of the beam by use of several BLMs (Beam Loss Monitors) showed a spot size of 20mm diameter and a flux uniformity that varied by less than 10% of the average value [71].This flux variation should be considered to be a systematic error present throughout the results presented in this section.

On initial test runs irradiating the DUT, it was noted that the supply current to the board would spike from an average value of about 0.5A to 1A – 1.5A. Initially, these were interpreted to be SEL events which would lead to the beam getting shut off and the board power cycled. It was only later determined that these were transient current spikes that went away on their own and did not seem to affect system operation.

As for the test programs developed, all kept getting interrupted by a system hang or system auto restart before any error information could be detected and reported. As a result, no register specific or ALU specific data could be obtained. Program hangs and system restarts (counted as instances of SEFI) were therefore used to characterize the device.

Once data was collected from a significant number of SEFI events, a high flux saturation test was carried out on one of the BLMs (at 30nA). In this time, the test-board had been moved to the beam shadow cast by the collimators. After this saturation test was completed, it was observed that the DUT failed to recognize some Linux command line commands (such as "*ls*", "*clear*", "*dmesg*") and was also incapable of running two of the test programs (GPR test and MMX test). A power cycle seemed to resolve the problem with the Linux commands but did nothing to remedy the two test programs.

Two final test runs were carried out on the DUT at a beam current of 30nA. The first one saw the device hang and the second one saw the device experience a current spike of about 3A, promptly followed by the device powering off and failing to start up again, even after a power cycle. Additionally, an LED on the board atypically switched on and stayed on. No specific error codes were found to match this on the specification manuals for the board. With these final observations, the board was declared "dead" and the experiment was brought to an end.

Table 4.1 is a log of events compiled from the screen recordings made, hand written notes and data recorded and made available by collaborators such as [71]. A plot of the supply current to the board for the entirety of the test is also provided in Figure 4.2. While interpreting this plot, take note that the point at which irradiation of the board begun does not necessarily coincide with time = 0 and neither does it with the point at which the current spikes begin to occur.

| Run No. | Cache State | Event | Cummilative time (minutes) | Run duration (seconds) | Measured Fluence (particles/cm$^2$) |
|---|---|---|---|---|---|
| | | **Current = 2nA** | | | |
| 10 | Cache OFF | Current spike | 1.05 | 63.0 | 526316096.2 |
| 11 | Cache OFF | Current spike | 2.87 | 109.2 | 913600996.8 |
| 12 | Cache OFF | Current spike | 3.04 | 10.2 | 63742496.88 |
| 13 | Cache OFF | Slow GPR test, nearly intepreted as hang. No power cycle | 6.38 | 200.4 | 1718808655 |
| 14 | Cache OFF | Current spike(s), hang | 8.87 | 149.4 | 1256400890 |
| 15 | Cache OFF | Current spike(s), hang | 11.06 | 131.4 | 1152443614 |
| | | **Current = 5nA** | | | |
| 16 | Cache OFF | Current spike(s), reboot | 14.00 | 176.4 | 4164281774 |
| 17 | Cache ON | Current spike(s), hang | 16.59 | 155.4 | 3748100275 |
| 18 | Cache ON | Crash | 18.45 | 111.6 | 2798803480 |
| 19 | Cache ON | Crash | 19.83 | 82.8 | 2048942966 |
| 20 | Cache ON | Crash | 21.57 | 104.4 | 2590598720 |
| 21 | Cache ON | Operator error in terminal. No power Cycle | 22.22 | 39.0 | 892643707.6 |
| 22 | Cache ON | Crash | 23.25 | 61.8 | 1509505243 |
| 23 | Cache ON | Crash | 24.31 | 63.6 | 1529280964 |
| 24 | Cache ON | Crash | 25.37 | 63.6 | 1538381113 |
| 25 | Cache ON | Crash | 27.05 | 100.8 | 2505567265 |
| 26 | Cache ON | Crash | 28.47 | 85.2 | 2121868528 |
| 27 | Cache ON | Crash | 28.87 | 24.0 | 566282123.5 |
| 28 | Cache ON | Crash | 30.54 | 100.2 | 2461849241 |
| | | **Current = 10nA** | | | |
| 29 | Cache ON | Crash | 31.38 | 50.4 | 2220228901 |
| | | **Current = 15nA** | | | |
| 30 | Cache ON | Crash | 32.12 | 44.4 | 2958771236 |
| 31 | Cache ON | Crash | 32.39 | 16.2 | 1012686894 |
| 32 | Cache ON | Crash | 32.81 | 25.2 | 1666840382 |
| 33 | Cache ON | [No beam] | 33.33 | 31.2 | 0 |
| 34 | Cache ON | Crash | 33.85 | 31.2 | 1754711746 |
| 35 | Cache ON | Crash | 34.78 | 55.8 | 3807303063 |
| 36 | Cache OFF | Crash | 35.23 | 27.0 | 1770694011 |
| 37 | Cache OFF | Crash | 35.88 | 39.0 | 2584379940 |
| 38 | Cache OFF | [No beam] | 37.00 | 67.2 | 0 |
| 39 | Cache OFF | RF Trip (cut off the beam) No power cycle | 37.35 | 21.0 | 1224249796 |
| 40 | Cache OFF | Crash | 38.53 | 70.8 | 4995255912 |
| 41 | Cache OFF | Crash | 39.91 | 82.8 | 5852224550 |
| 42 | Cache OFF | Crash | 40.76 | 51.0 | 3532412249 |
| | | **Current = 30nA** | | | |
| **43 - 45** | **BLM Saturation test** | **(test board in beam shadow still powered on)** | **45.86** | **306.0** | |
| | | *Some bash commands not recognized and GPR & MMX programs not running after BLM saturation test. Power cycle fixed bash only* | | | |
| 46 | Cache ON | Crash | 46.90 | 62.4 | 7884604832 |
| 47 | Cache ON | 3A current spike, board dead, LED stuck ON | 48.27 | 82.2 | 10606689294 |

*Table 4.1: Combined log of events from SEE testing at iThemba Labs (17th January 2019). Log data is included from [71]. "Crash" can be interpreted as either a system hang or auto reboot. All test runs ended with the test-board getting power cycled, unless otherwise indicated.*

| Key | |
|---|---|
| **Cache OFF runs:** | |
| **Cache ON runs:** | |
| **Beam stopped for non SEFI related reason:** | Red Text |

*Figure 4.1: Key for Table 4.1*

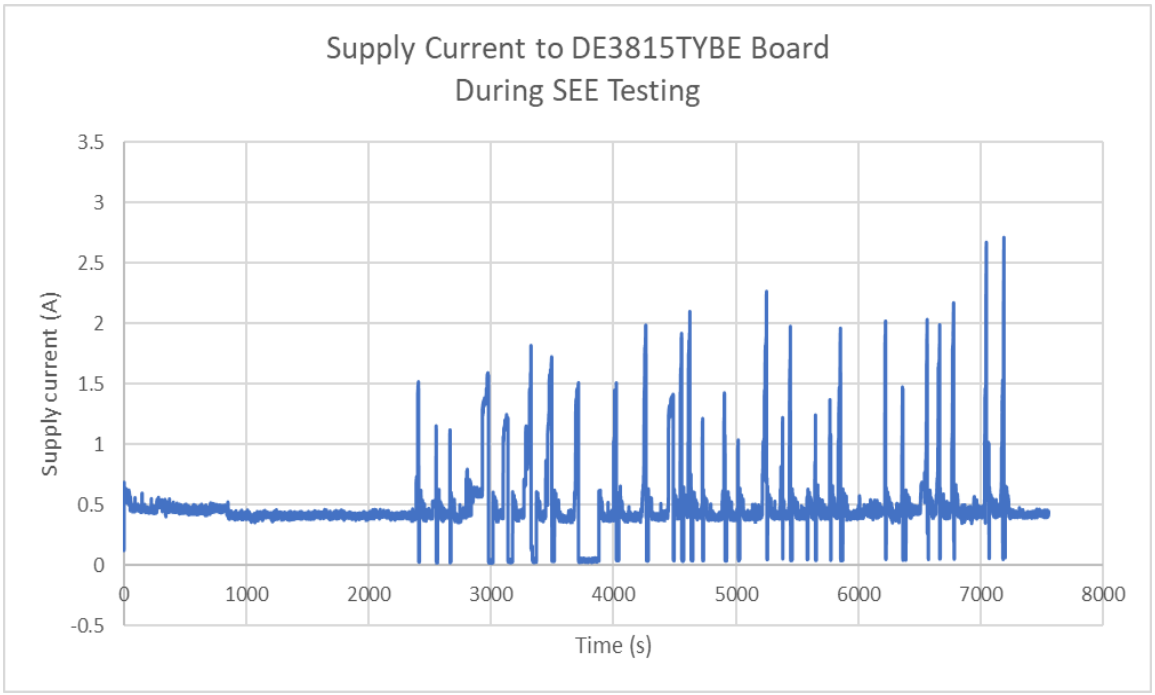



*Figure 4.2: Supply Current to DE3815TYBE during SEE Testing.*

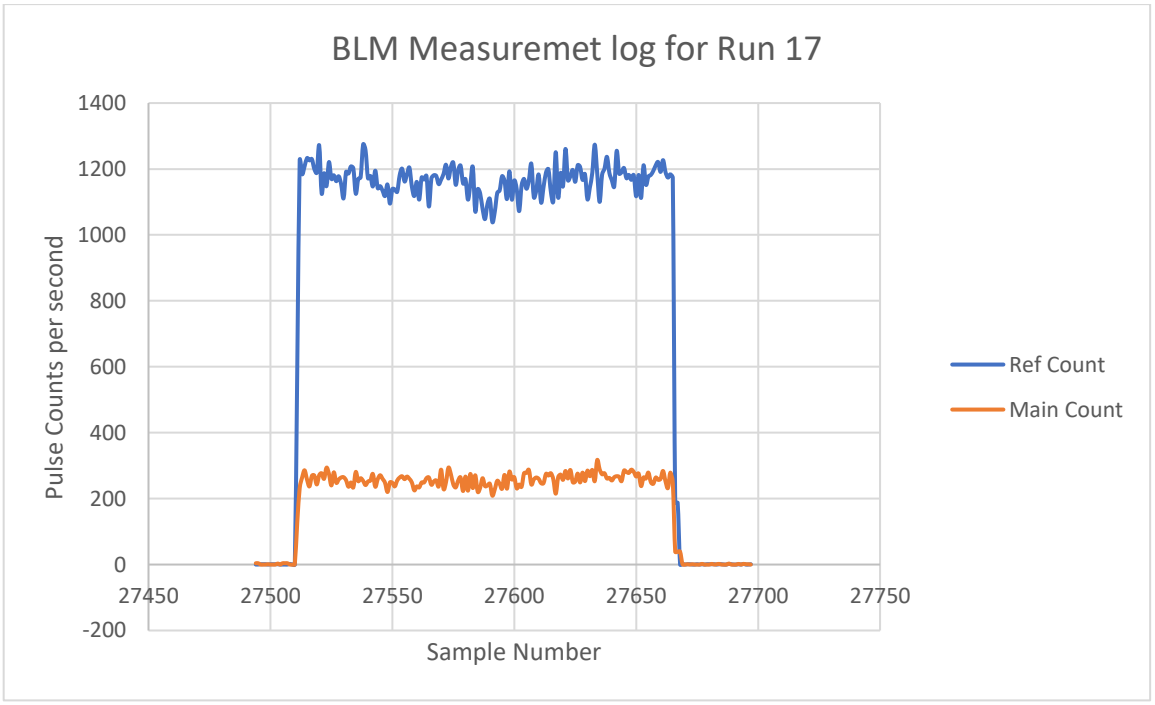



*Figure 4.3: An example plot that summarizes measurements made by the BLMs. The data presented here was measured from test run 17. Log files were provided by [71].*

The total fluence delivered to the DUT per test run was determined from log files of BLM measurements. Each BLM measured the instantaneous proton flux in its physical location at intervals of 1 second. These log files provided data such as what is summarized in Figure 4.3.

Using a mathematical relationship provided by [71], a scaling factor was calculated that would scale the readings of the Reference BLM ("Ref Count" in Figure 4.3) to the instantaneous flux that was at the location of the DUT within the beam. This flux measurement would be in the units protons per BLM area per second, where the BLM area was determined from the BLM sensor dimensions:

$$BLM\ area = 2.712mm\ x\ 2.712mm = \ 73.55\ x\ 10^{-3}\ cm^2$$

As an example, in the log file summarized by Figure 4.3, the Ref Count value at sample number 27550 is 1139 pulse counts. Using this, and the scaling factor that had been determined:

$$Scaling\ factor = 1524.63$$

$$Scaled\ Ref\ Count = Scaling\ factor\ x\ Ref\ Count = 1524.63\ x\ 1139$$
$$= \ 1.74\ x\ 10^6\ protons\ per\ BLM\ area\ per\ second\ (instantanious\ flux\ at\ DUT)$$

To convert this to the units of protons/cm$^2$/s, the value would be divided by the BLM area. Following through:

$$\frac{1.74\ x\ 10^6}{73.55\ x\ 10^{-3}} = \ 2.36\ x\ 10^7\ protons/cm^2/s$$

This means that when sample number 27550 was recorded by the reference BLM, the instantaneous flux at the location of the DUT was $2.36\ x\ 10^7$ protons/cm$^2$/s. The total fluence delivered by the end of the test run (per cm$^2$) would then be determined by adding up the instantaneous flux at the DUT location for each sample taken throughout the duration of the test run (i.e. integrating the instantaneous flux).

The DUT die was measured to be a square of 1cm by 1cm (see Figure 3.3), which gives an area of 1cm$^2$. Therefore, the total fluence delivered to the DUT by a test run would simply be the value of the prior determined fluence. This process was repeated for each logfile provided, where each file corresponded to a test run. The total calculated fluence to have been delivered to the DUT for each test run is presented in the rightmost column of Table 4.1 (Page 56).

| | Cache ON | | | Cache OFF | | |
|---|---|---|---|---|---|---|
| **Beam Current (nA)** | SEFI Count | Fluence (protons/cm$^2$) | **$\sigma_{DEVICE}$** | SEFI Count | Fluence (protons/cm$^2$) | **$\sigma_{DEVICE}$** |
| **2** | X | X | **X** | 2 | 4127653159 | **4.84537E-10** |
| **5** | 11 | 2.43E+10 | **4.52E-10** | 1 | 4164281774 | **2.40137E-10** |
| **10** | 1 | 2.22E+09 | **4.50E-10** | X | X | **X** |
| **15** | 5 | 1.12E+10 | **4.46E-10** | 5 | 19959216458 | **2.50511E-10** |
| **30** | 2 | 1.85E+10 | **1.08E-10** | X | X | **X** |

*Table 4.2: Calculated cross sections at different beam currents (all at 55.58 MeV). X indicates no data available.*

As discussed in Section 2.3.1, the cross-section is determined using Eqn 2.2 repeated below:

$$\sigma = \frac{n}{Ft\cos\theta}$$

The numerator is the number of SEE events counted while the denominator is essentially an expression that calculates the total fluence delivered to the DUT. Table 4.2 shows cross sections calculated using this equation at each beam current that testing was carried out. For example, at 5nA with the Cache On, the cross section is calculated as follows:

$$\sigma_{DEVICE} = \frac{No.SEFI\ counted}{Sum\ of\ fluence\ delivered\ by\ each\ test\ run} = \frac{11}{2.43\ x\ 10^{10}}$$

$$\sigma_{DEVICE} = 4.52\ x\ 10^{-10}\ SEFI/proton/cm^2$$

The cross sections in this case can only be presented on a per device basis rather than per-bit. This is because it is impossible to isolate the root cause of the SEFI with the equipment that was available.

The data presented in Table 4.2 is visualized in Figure 4.4.

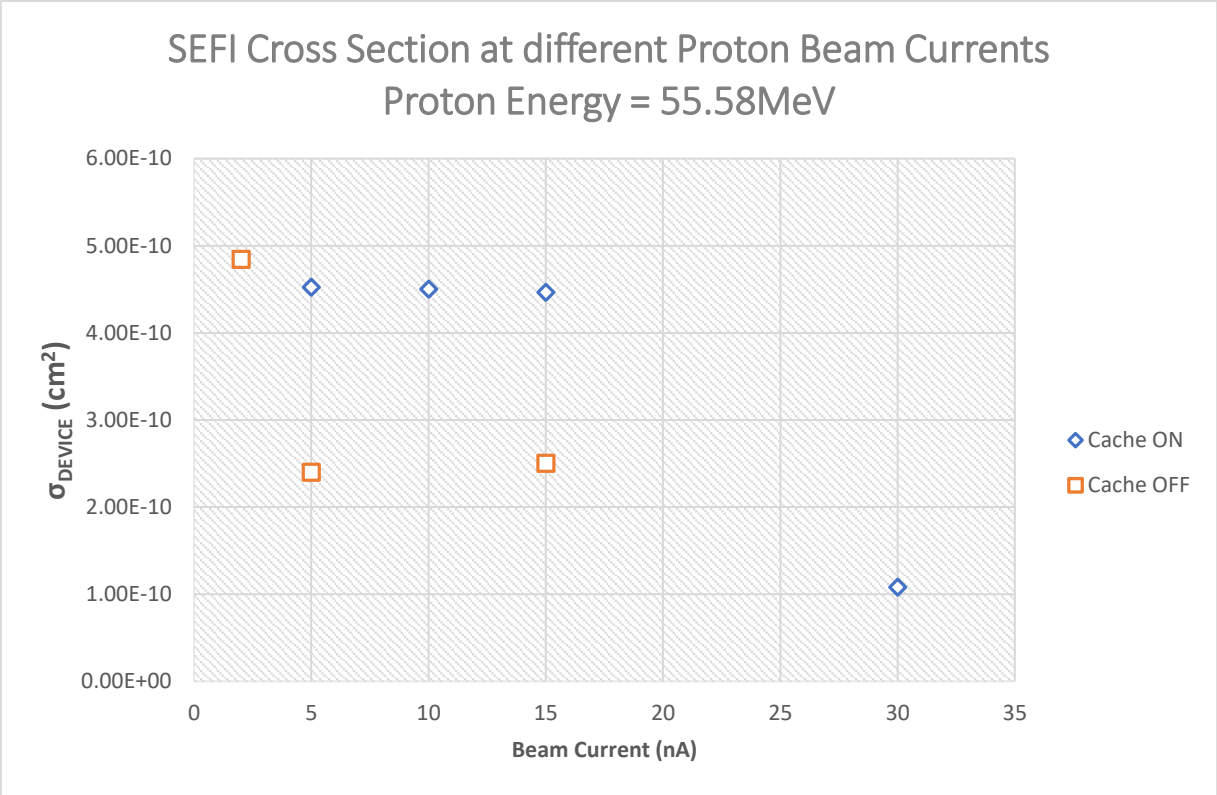



*Figure 4.4: Calculated device cross-sections at different beam currents.*

It will immediately be noticed that for each test case (Cache On/Cache Off), the majority of the data points sit around the same value for cross section. This is about $2.5\ x\ 10^{-10}$ cm$^2$ and $4.5\ x\ 10^{-10}$ cm$^2$ for Cache Off and Cache On respectively. This was somewhat expected since cross section is dependent on the fluence rather than flux (or beam current), within reasonable limits. Another observation is that the majority of Cache Off data points are

lower than the majority of Cache On data points, implying that Cache Off is less sensitive to SEFI. This too was to be expected as was discussed in section 3.1.1.3.

The cross section determined from the 30nA (Cache On) test is considered to be an outlier. This is not only because of the large deviation from the majority of Cache On data points, but also because the test was carried out only after the DUT started showing signs of succumbing to TID damage. It is believed that during the BLM saturation test (carried out at large fluxes), stray deflected protons and secondary particles (i.e. neutrons) found their way to the test-board that was in the beam shadow. The combined total dose from these likely slower particles, as well as from the prior test runs, induced TID damage that inadvertently influenced SEFI sensitivity. For this reason, subsequent analysis ignores this data point.

As for the Cache Off cross section at 2nA, it is believed that the deviation from the majority of cache off data points was due to a methodical error made while carrying out one of the test runs. Referring to Table 4.1, Run 13 was prematurely ended and the beam switched off after slow execution of a test program was accidentally interpreted to be a SEFI. The board was not power cycled before Run 14 commenced. The combined fluence from both runs was however taken into account while calculating the cross section.

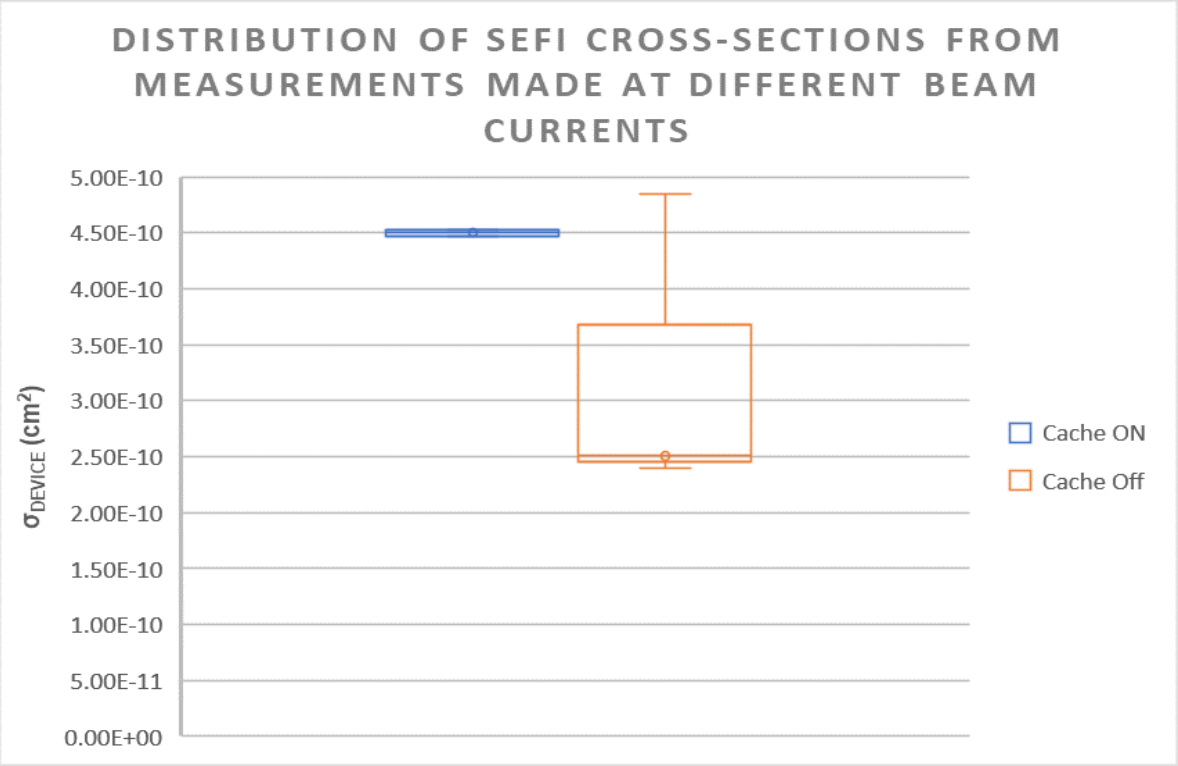



*Figure 4.5: Distribution of cross sections determined (55.58MeV)*

Figure 4.5 shows the distribution of the calculated device cross sections at different beam currents. Additionally, the overall SEE response of the DUT for the entire test is summarized in Table 4.3 and visualized in Figure 4.6. As mentioned, the cross section determined at 30nA (Cache On) is not included.

| | SEFI Count | Total Fluence (protons/cm$^2$) | $\sigma_{DEVICE}$ |
|---|---|---|---|
| **Cache Off:** | 8 | 28251151392 | 2.83174E-10 |
| **Cache On:** | 17 | 37732365848 | 4.50542E-10 |

*Table 4.3: Overall device cross sections.*

Although the amount of data collected was not statistically significant enough to fit an accurate and reliable cross section curve, one was still generated to provide ballpark figures of the responses to expect at different proton energies. The Bendel 1-parameter equation was used since all testing was carried out at a single beam energy. Numerical methods were used to determine the sensitivity parameter (A) for each case of the overall device cross section. Table 4.4 and Figure 4.7 summarize this information.

| Cache State | $\sigma_{DEVICE}$ at 55.58 MeV (cm$^2$) | A at 55.58 MeV |
|---|---|---|
| **OFF** | 2.83174E-10 | 14.50519 |
| **ON** | 4.50542E-10 | 14.05895 |

*Table 4.4: Bendel 1-prameter "A" parameter values for Cache On and Cache Off cross sections.*

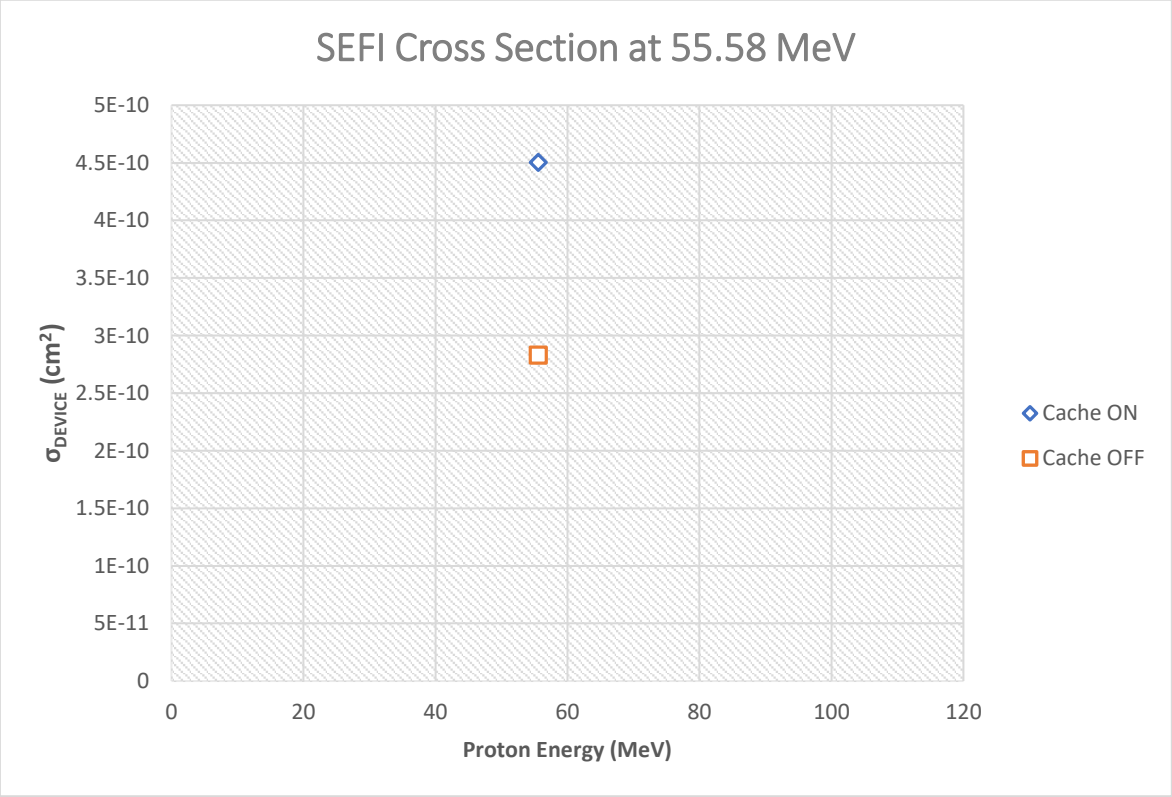



*Figure 4.6: Overall device cross sections*

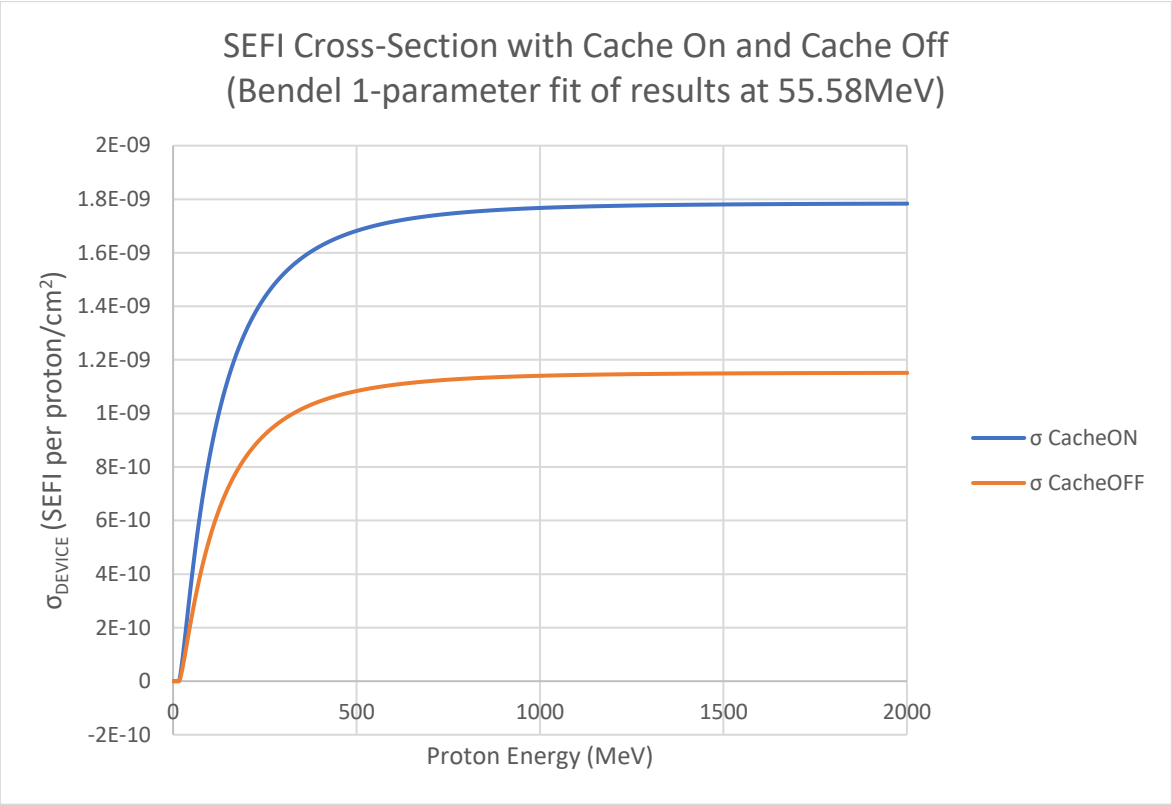



*Figure 4.7: Bendel 1-parameter curve fitted to the overall SEFI cross sections*

## 4.2. TID Results

Before any results are interpreted, it should be noted that the distance between the DUT and the Cobalt-60 source had some uncertainty in it. This inevitably extended to the final dose rate value of 9.7kRad/h. The tape measure used to determine the distance to place the DUT introduced an uncertainty of about 1cm while the method of mounting the board to the fixture introduced an additional 1cm (board was not perfectly vertical). With both of these considered, it was determined that the lower bound for the dose rate was 9.064kRad/h and the upper bound was 10.406kRad/h [69].

The DUT was irradiated for 6.933 hours to a total dose of 67.25kRad (± uncertainty). By the end of this, it was noted that the device was no longer powered on and attempts to power cycle it failed. The device was then left to anneal at room temperature for a total of 210.28 hours post irradiation. Despite this, it still failed to power on when connected to power.

Figure 4.8 is a plot of all the data collected during the test, plotted against the duration of the test. Figure 4.9 on the other hand is a plot of averaged data (i.e. average value of all samples made per 1-minute interval) plotted against absorbed dose.

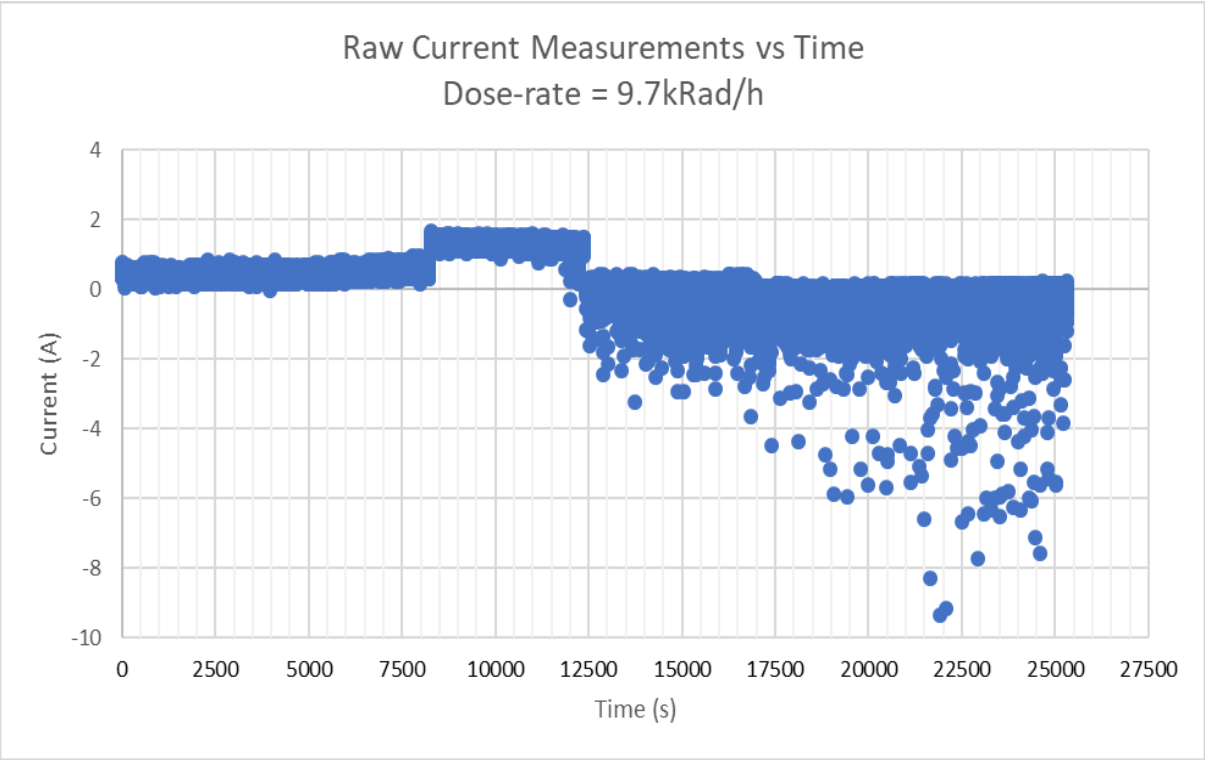



*Figure 4.8: Raw data obtained from TID test.*

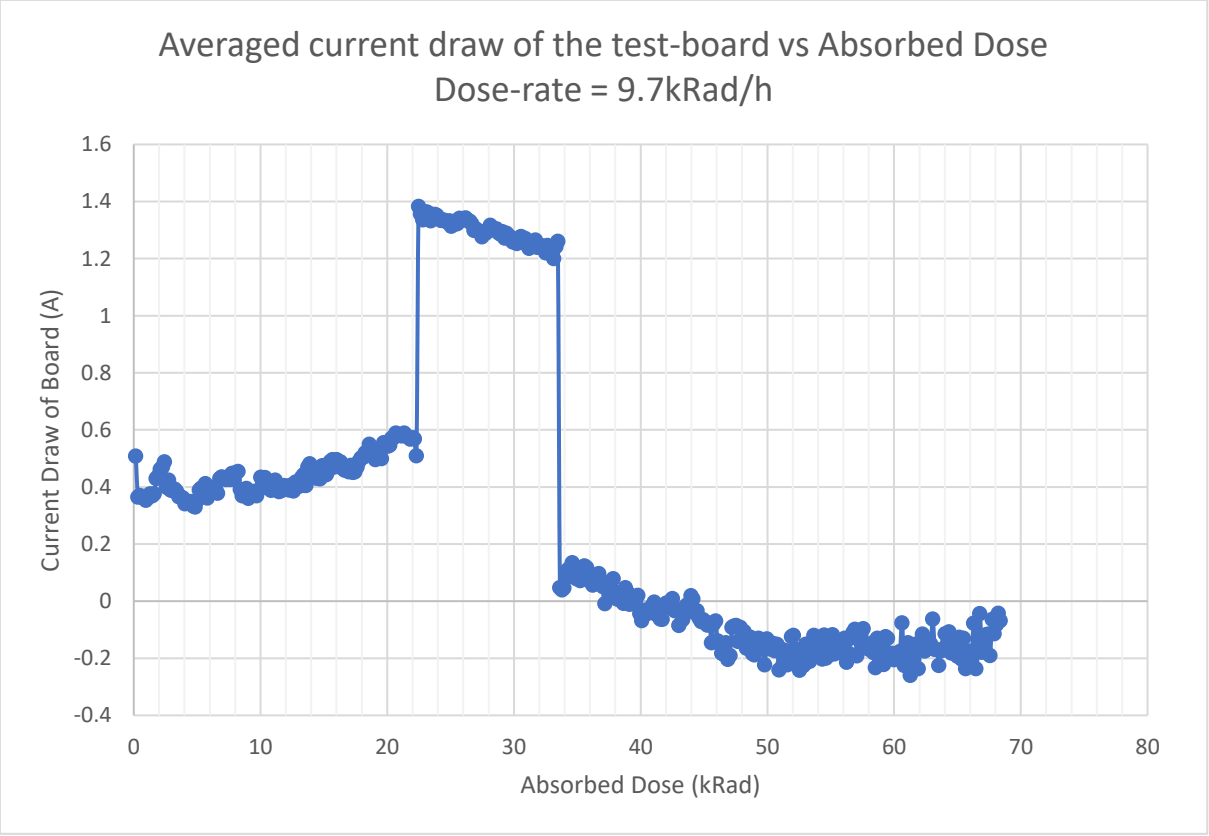



*Figure 4.9: Averaged current draw vs TID absorbed during the test*

From Figure 4.9, it can be seen that the supply current to the board is fairly constant at the beginning of the test but steadily starts increasing at a total dose of around 15kRad. The supply current then sharply increases after about 22kRad is absorbed. This is believed to be a direct result of parasitic leakage currents being induced at the transistor level. It is assumed that this is the point where the device will start to display operability problems.

At about 34kRad of total dose, the supply current sharply decreases to well below normal operational levels. This is assumed to correspond to a system failure and that this is the point where the board switched off and was unable to switch itself back on again. The supply current then continues to gradually decrease with absorbed dose until it reverses direction and appears to settle at an average value of -0.2A with random fluctuations of up to -10A.

# 5. Discussion and Conclusions

Data obtained from the SEE test points to the device having a relatively high SEL tolerance since no SEL events were observed for the entirety of the experiment. However, transient current spikes were observed, though they appeared to be harmless to system operation. None of these transients ever exceeded the rated current of the board and there was no discernible effect observed on test program execution. It is possible that they were a result of other components on the test board getting affected by stray protons and/or secondary radiation. These transients however caused an increase to the average supply current which in turn increased the average power draw of the board. In a satellite where energy is limited, this may be a serious concern.

No data was obtained with regards to the susceptibility of specific register banks of the processor. It was already mentioned that the SEFI rate was high enough such that the register specific test programs kept on getting interrupted mid execution, however, each program did manage to execute for some amount of time, about a minute or more on average. Despite this, none reported upsets within that time. Several reasons may contribute to this behaviour, individually and collectively.

1. The upset rate on the registers may have not been high enough for upsets to be detected in the time that the programs were running.
2. The sequential order in which the programs checked for errors in a given register bank significantly narrowed the upset detection window. It is possible that upsets occurred in multiple registers but were missed because they did not occur in the register that the program was inspecting at the time. This is a fundamental limitation imposed by the architecture of the processor.
3. The registers tested occupied a very small portion of the processor which in turn made it less probable for them to be struck by protons during irradiation.
4. The pre-emptive nature of the operating system may have been limiting the number of detectable upsets by interrupting the test program in order to service some other process. This would result in less scan cycles happening per unit time and upsets getting corrected by processor context switching.

The overall SEE response was dominated by SEFI. This should come as no surprise given the level of complexity of the processor. This complexity adds to the likelihood of SEFIs occurring since the protons would not have to strike only the most crucial areas of the processor to cause a SEFI. The protons could also strike less crucial areas of the device and induce errors here that could then propagate to more crucial areas, eventually causing a SEFI. For instance, errors induced in the output logic of the control unit could propagate through the datapath and back to the state memory registers thus forcing the control unit (and processor as a whole) to enter an undefined state. This makes it difficult to pinpoint where exactly the original upset(s) occurred that led to the observed SEFI. Adding on to this, support electronics for the processor that are located on the board could also have been affected by stray protons and/or secondary radiation. Depending on the severity of

the effect caused by the radiation, the entire system could momentarily lose functionality as the OS tries to correct the perceived error. This last point demonstrates why the SEFI rate observed cannot be completely attributed to just the processor.

With that said, a clear relationship can be seen between SEFI sensitivity and the state of the on-board cache of the processor. With the cache on, the SEFI sensitivity is about 1.5 times the SEFI sensitivity with the cache off at 55.58 MeV. It is clear that the on-board cache, when active, provides a large number of registers that could potentially experience upsets and propagate said upsets to the rest of the processor and cause SEFI.

TID test results imply that the device was operable up until a total dose of about 22kRad was absorbed. After this point, a significant increase to the supply current is observed, likely as a direct result of leakage currents in the processor. However, it is unclear as to why the current steadily decreased right before the device appeared to completely fail. It may be possible that the isolation structures (or field oxide) of affected transistors may have been undergoing short term annealing which in turn led to a decrease in leakage currents, however this is in conflict with the fact that the device was still undergoing irradiation and receiving ionizing dose. Further investigation into this observation is required.

The relatively low TID survivability of the device that has been observed would mean that the device would not be suitable for use in long term missions, especially those with orbits within the Van Allen belts. Longer survivability would likely be observed in high altitude (>7 earth radii) polar and high incline orbits where the spacecraft would spend the least amount of time within the radiation belts. Unfortunately, these orbits offer little geomagnetic shielding for the spacecraft, meaning that the craft would be more vulnerable to solar events and cosmic ray induced SEEs. Regardless, the SEE rate experienced here (ignoring solar and galactic events) is expected to be lower than that experienced in lower altitude orbits since these SEEs would predominantly be induced by low flux cosmic rays. Even though cosmic ray particles typically possess energy that is orders of magnitude higher than that of protons present in lower altitudes of the radiation belts, their lower flux would mean that the spacecraft, and by extension, the processor, would be less frequently struck by a cosmic ray than if the same spacecraft was in a lower altitude orbit where high fluxes of energetic protons are present.

The SEFI cross section observed cannot be used to specify a suitable orbit for a mission utilizing the processor within this text since the maximum tolerable SEE rate heavily relies on the application and mission requirements. A cost-benefit analysis would have to be carried out by a system designer should they desire to incorporate the E3815 into their system. If the processor is used, it would be recommended that it be operated at a lower frequency than that used during the test since it is likely that a lower SEFI sensitivity would be experienced. Also, shielding the spacecraft would add to the TID survivability.

With all of this said, the data presented here paints only a broad picture of the response of the E3815 processor to radiation. The processor is a complicated device that would require more time and resources in order to more accurately characterize.

## 5.1. Recommendations for further research

- The OS used plays a large role in the behaviour of the device during testing. A recommendation would be to use simpler operating systems that give more control of the hardware to the user.
- Use of a development board. Development boards for the E3800 processors come with the processor already soldered on and therefore would not offer direct access to the processor pins. However, these boards allow for custom BIOS to be installed. This may prove beneficial if a custom operating system is intended to be used.
- Monitoring of support electronics during testing. This would be beneficial since it would add a layer of transparency as to whether observed behaviour is due to errors within the processor or due to the support electronics failing.
- Inspection of a larger number of registers. The experiments presented here inspected 3 of the registers that applications mostly use. Other important registers such as the instruction pointer, flags register etc. that were not investigated also play a role in the overall device response, therefore determining their sensitivities to radiation would be useful.
- Inclusion of timing tests during TID testing. Although shifts in the threshold voltage of transistors present in modern devices is negligible [15], including timing tests may provide some insight into the state of the processor during irradiation.

# Bibliography


[1] R. Ginosar, “Survey of Processors for Space,” in *Proceedings of DASIA 2012 DAta Systems In Aerospace*, Drubrovnik, Croatia, 2012.

[2] S. M. Guertin, M. Amrbar and S. Vartanian, “Radiation Test Results for Common CubeSat Microcontrollers and Microprocessors,” in *2015 IEEE Radiation Effects Data Workshop (REDW)*, Boston, MA, USA, 2015.

[3] J. G. van der Horst, “Radiation tolerant implementation of a soft-core processor for space applications,” M.S. thesis, University of Stellenbosch, Stellenbosch, 2007.

[4] European Space Agency, “Onboard Computers,” European Space Agency, 18 February 2019. [Online]. Available: https://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Onboard_Computers. [Accessed 14 March 2019].

[5] M. Gaillardin, M. Raine, P. Paillet, M. Martinez, C. Marcandella, S. Girard, O. Duhamel, N. Richard, F. Andrieu, S. Barraud and O. Faynot, “Radiation effects in advanced SOI devices: New insights into Total Ionizing Dose and Single-Event Effects,” in *2013 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, Monterey, CA, USA, 2013.

[6] Intel, “Intel Announces New 22nm 3D Tri-gate Transistors,” 2011. [Online]. Available: https://www.intel.co.za/content/www/za/en/silicon-innovations/standards-22nm-3d-tri-gate-transistors-presentation.html. [Accessed 26 August 2018].

[7] Australian Nuclear Science and Technology Organisation, “What is radiation?,” Australian Nuclear Science and Technology Organisation, [Online]. Available: http://www.ansto.gov.au/NuclearFacts/Whatisradiation/index.htm. [Accessed 4 June 2018].

[8] World Health Organization, “What is Ionizing Radiation?,” World Health Organization, [Online]. Available: http://www.who.int/ionizing_radiation/about/what_is_ir/en/. [Accessed 4 June 2018].

[9] A. Holmes-Siedle and L. Adams, Handbook of Radiation Effects, New York: Oxford University Press, 2000.

[10] Harvard University, “α, β, γ, n Sources and Detection,” Harvard University, [Online]. Available: https://sciencedemonstrations.fas.harvard.edu/presentations/α-β-γ-n-sources-and-detection. [Accessed 4 June 2018].

[11] E. G. STASSINOPOULOS and J. P. RAYMOND, “The Space Radiation Environment for Electronics,” *PROCEEDINGS OF THE IEEE,* vol. 76, no. 11, pp. 1423-1442, 1988.

[12] KSU Physics Education Group, “Hydrogen Spectroscopy - Emission,” Kansas State University, [Online]. Available: https://web.phys.ksu.edu/vqm/tutorials/hydrogen/. [Accessed 17 March 2019].

[13] HEASARC and D. A. Smale, “Processes that Create Cosmic Gamma Rays,” National Aeronautics and Space Administration, October 2010. [Online]. Available: https://imagine.gsfc.nasa.gov/science/toolbox/gamma_generation.html. [Accessed 17 March 2019].

[14] J. Hanania, K. Stenhouse and J. Donev, “Nuclear fusion in the Sun,” Energy Education, 26 August 2015. [Online]. Available: https://energyeducation.ca/encyclopedia/Nuclear_fusion_in_the_Sun. [Accessed 17 March 2019].

[15] H. Garrett, I. Jun, T. Oldham, M. Baze and R. Ecoffet, Space Radiation Environments and Their Effects on Devices and Systems: Back to the Basics, Las Vegas: IEEE, 2011.

[16] J. R. Schwank, “Basic mechanisms of radiation effects in the natural space radiation environment,” in *Conference: 31. annual international nuclear and space radiation effects conference*, Tucson, 1994.

[17] S. v. Aardt, “TOTAL IONIZING DOSE AND SINGLE EVENT UPSET TESTING OF FLASH BASED FIELD PROGRAMMABLE GATE ARRAYS,” M.Eng. thesis, Nelson Mandela Metropolitan University, Port Elizabeth, 2014.

[18] F. Smith, “Total Ionizing Dose Mitigation by means of Reconfigurable FPGA Computing,” Ph.D. dissertation, University of Stellenbosch, Stellenbosch, 2007.

[19] E. O. Hwang, Digital Logic and Microprocessor Design with VHDL, Toronto, Ontario: Thomson/Nelson, 2006.

[20] D. A. Neamen, Microelectronics Circuit Analysis and Design, New York: McGraw-Hill, 2010.

[21] L. Harrison, “An introduction to Depletion-mode MOSFETs,” [Online]. Available: http://www.aldinc.com/pdf/IntroDepletionModeMOSFET.pdf. [Accessed 20 July 2018].

[22] Intel, “3D, 22 nm: New Technology Delivers An Unprecedented Combination of Performance and Power Efficiency,” Intel, [Online]. Available: https://www.intel.co.za/content/www/za/en/silicon-innovations/intel-22nm-technology.html. [Accessed 26 August 2018].

[23] P. H. Vora and R. Lad, “A Review Paper on CMOS, SOI and FinFET Technology,” [Online]. Available: https://www.design-reuse.com/articles/41330/cmos-soi-finfet-technology-review-paper.html. [Accessed 26 August 2018].

[24] J. Karp, M. J. Hart, P. Maillard, G. Hellings and D. Linten, “Single-Event Latch-Up: Increased Sensitivity From Planar to FinFET,” *IEEE TRANSACTIONS ON NUCLEAR SCIENCE,* vol. 65, no. 1, pp. 217-222, 2018.

[25] P. Nsengiyumva, D. R. Ball, J. S. Kauppila, N. Tam, M. McCurdy, W. T. Holman, M. L. Alles, B. L. Bhuva and L. W. Massengill, “A Comparison of the SEU Response of Planar and FinFET D Flip-Flops at Advanced Technology Nodes,” *IEEE TRANSACTIONS ON NUCLEAR SCIENCE,* vol. 63, no. 1, pp. 266-272, 2016.

[26] M. Anwar, “Sequential Logic Circuits,” Electronic & Electrical Engineer's Guide, 25 September 2014. [Online]. Available: http://eeeguide1.blogspot.com/2014/09/sequential-logic-circuits.html. [Accessed 24 July 2018].

[27] H.-W. Huang, PIC Microcontroller: An Introduction to Software and Hardware Interfacing, Mankato: DELMAR CENGAGE Learning , 2007.

[28] NASA Space Radiation Laboratory, “NASA Space Radiation Laboratory User Guide III. Technical Data: Bragg Curves and Peaks,” NASA Space Radiation Laboratory, [Online]. Available: https://www.bnl.gov/nsrl/userguide/bragg-curves-and-peaks.php. [Accessed 13 August 2018].

[29] K. A. LaBel, “Radiation Effects on Electronics 101: Simple Concepts and New Challenges,” 21 April 2004. [Online]. Available: https://nepp.nasa.gov/DocUploads/392333B0-7A48-4A04-A3A72B0B1DD73343/Rad_Effects_101_WebEx.pdf. [Accessed 08 June 2018].

[30] P. Nsengiyumva, “CHARACTERIZATION OF THE CMOS FINFET STRUCTURE ON SINGLE-EVENT EFFECTS - BASIC CHARGE COLLECTION MECHANISMS AND SOFT ERROR MODES,” Ph.D. dissertation, Vanderbilt University, Nashville, Tennessee, 2018.

[31] T. P. Ma and P. V. Dressendorfer, Ionizing Radiation Effects in MOS Devices & Circuits, New York: John Wiley & Sons, 1989.

[32] T. C. May and M. H. Woods, “Alpha-Particle-Induced Soft Errors in Dynamic Memories,” *IEEE Transactions on Electron Devices,* vol. 26, no. 1, pp. 2-9, 1979.

[33] A. H. Johnston, “The Influence of VLSI Technology Evolution on Radiation-Induced Latchup in Space Systems,” *IEEE Transactions on Nuclear Science,* vol. 43, no. 2, pp. 505 - 521, 1996.

[34] S. Lee, I. Kim, S. Ha, C.-s. Yu, J. Noh, S. Pae and J. Park, “Radiation-Induced Soft Error Rate Analyses for 14 nm FinFET SRAM Devices,” in *2015 IEEE International Reliability Physics Symposium*, Monterey, CA, USA, 2015.

[35] H. Zhang, H. Jiang, B. L. Bhuva, J. S. Kauppila, W. T. Holman and L. W. Massengill, “Frequency Dependence of Heavy-Ion-Induced Single-Event Responses of Flip-Flops in a 16-nm Bulk FinFET Technology,” *IEEE TRANSACTIONS ON NUCLEAR SCIENCE,* vol. 65, no. 1, pp. 413-417, 2018.

[36] H. Zhang, H. Jiang, T. R. Assis, D. R. Ball, B. Narasimham, A. Anvar, L. W. Massengill and B. L. Bhuva, “Angular Effects of Heavy-Ion Strikes on Single-Event Upset Response of Flip-Flop Designs in 16-nm Bulk FinFET Technology,” *IEEE TRANSACTIONS ON NUCLEAR SCIENCE,* vol. 64, no. 1, pp. 491-496, 2017.

[37] P. Nsengiyumva, L. W. Massengill, J. S. Kauppila, J. A. Maharrey, R. C. Harrington, T. D. Haeffner, D. R. Ball, M. L. Alles, B. L. Bhuva, W. T. Holman, E. X. Zhang, J. D. Rowe and A. L. Sternberg, “Angular Effects on Single-Event Mechanisms in Bulk FinFET Technologies,” *IEEE TRANSACTIONS ON NUCLEAR SCIENCE,* vol. 65, no. 1, pp. 223-230, 2018.

[38] F. Irom, “Guideline for Ground Radiation Testing of Microprocessors in the Space Radiation Environment,” Jet Propulsion Laboratory, National Aeronautics and Space Administration, Pasadena, California, 2008.

[39] S. Buchner, P. Marshall, S. Kniffin and K. LaBel, “Proton Test Guideline Development – Lessons Learned,” 22 August 2002. [Online]. Available: https://radhome.gsfc.nasa.gov/radhome/papers/proton_testing_guidelines_2002.pdf. [Accessed 2 February 2019].

[40] B. Sierawski and M. Mendenhall, “Bendel 1-parameter function,” Vanderbilt University, November 2010. [Online]. Available: https://creme.isde.vanderbilt.edu/CREME-MC/help/bendel-1-parameter-function. [Accessed 2 February 2019].

[41] W. Bendel and E. Petersen, “Proton Upsets in Orbit,” *IEEE Transactions on Nuclear Science,* Vols. NS-30, no. 6, pp. 4481-4485, 1983.

[42] B. Sierawski and M. Mendenhall, “Bendel 2-parameter function,” Vanderbilt University, November 2010. [Online]. Available: https://creme.isde.vanderbilt.edu/CREME-MC/help/bendel-2-parameter-function. [Accessed 2 February 2019].

[43] W. Stapor, J. Meyers, J. Langworthy and E. Petersen, “TWO PARAMETER BENDEL MODEL CALCULATIONS FOR PREDICTING PROTON INDUCED UPSET,” *IEEE TRANSACTIONS ON NUCLEAR SCIENCE,* vol. 37, no. 6, pp. 1966-1973, 1990.

[44] B. Sierawski and M. Mendenhall, “Weibull,” Vanderbilt University, November 2010. [Online]. Available: https://creme.isde.vanderbilt.edu/CREME-MC/help/weibull. [Accessed 2 February 2019].

[45] S. R. D and F. D. M, Radiation Effects And Soft Errors In Integrated Circuits And Electronic Devices, Singapore: World Scientific Publishing Co. Pte. Ltd., 2004.

[46] I. Chatterjee, E. X. Zhang, B. L.Bhuva, M. A. Alles, R. Schrimpf, D. M. Fleetwood, Y.-P. Fang and A. Oates, “Bias Dependence of Total-Dose Effects in Bulk FinFETs,” *IEEE TRANSACTIONS ON NUCLEAR SCIENCE,* vol. 60, no. 6, pp. 4476-4482, 2013.

[47] I. Chatterjee, E. X. Zhang, B. L. Bhuva, R. A. Reed, M. L. Alles, N. N. Mahatme, D. R. Ball, R. D.Schrimpf, D. Fleetwood, D. Linten, E. Simôen, J. Mitard and C. Claeys, “Geometry Dependence of Total-Dose Effects in Bulk FinFETs,” *IEEE TRANSACTIONS ON NUCLEAR SCIENCE,* vol. 61, no. 6, pp. 2951-2958, 2014.

[48] H. Zhang, H. Jiang, X. Fan, J. S. Kauppila, I. Chatterjee, B. L. Bhuva and L. W. Massengill, “Effects of Total-Ionizing-Dose Irradiation on Single-Event Response for Flip-Flop Designs at a 14-/16-nm Bulk FinFET Technology Node,” *IEEE TRANSACTIONS ON NUCLEAR SCIENCE,* vol. 65, no. 8, pp. 1928-1934, 2018.

[49] A. Bacchini, G. Furano, M. Rovatti and M. Ottavi, “Total Ionizing Dose Effects on DRAM Data Retention Time,” *IEEE TRANSACTIONS ON NUCLEAR SCIENCE,* vol. 61, no. 6, pp. 3690-3693, 2014.

[50] M. Davies, Standard Handbook for Aeronautical and Astronautical Engineers, New York, Chicago, San Francisco, Lisbon, London, Madrid, Mexico City, Milan, New Delhi, San Juan, Seoul, Singapore, Sydney, Toronto: McGRAW-HILL, 2003.

[51] J. W. H. Jr, M. A. Carts, R. Stattel, C. E. Rogers, T. L. Irwin, C. Dunsmore, J. A. Sciarini and K. A. LaBel, “Total Dose and Single Event Effects Testing of the Intel Pentium III (P3) and AMD K7 Microprocessors,” in *IEEE Radiation Effects Data Workshop*, Vancouver, BC, Canada, Canada, 2001.

[52] M. Barabanov, “Open RTLinux Installation Instructions,” FSM Labs, Inc., 26 July 2001. [Online]. Available: http://cs.uccs.edu/~cchow/pub/master/dsknoop/doc/html/Installation/. [Accessed 8 February 2019].

[53] KernelNewbies, “Preemption under Linux,” KernelNewbies, 30 December 2017. [Online]. Available: https://kernelnewbies.org/FAQ/Preemption. [Accessed 6 April 2019].

[54] Intel, “Intel Thin Canyon NUC Atom E3815 1.46GHz,” June 2015. [Online]. Available: https://za.rs-online.com/web/p/single-board-computers/8829694/. [Accessed 14 September 2018].

[55] Intel, “Intel Atom® Processor E3815 512K Cache, 1.46 GHz,” [Online]. Available: https://ark.intel.com/products/78476/Intel-Atom-Processor-E3815-512K-Cache-1_46-GHz. [Accessed 15 March 2018].

[56] C. Lomont, “Introduction to x64 Assembly,” 19 March 2012. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-x64-assembly. [Accessed 16 October 2018].

[57] J. Pawasauskas, “CS563 - Advanced Topics in Computer Graphics,” 22 April 1997. [Online]. Available: https://web.cs.wpi.edu/~matt/courses/cs563/talks/powwie/p3/mmx.htm. [Accessed November 2018].

[58] T. Doeppner, “x64 Cheat Sheet,” 2018. [Online]. Available: https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf. [Accessed November 2018].

[59] The FreeBSD Documentation Project, “11.13. Using the FPU,” FreeBSD Foundation, 26 August 2018. [Online]. Available: https://www.freebsd.org/doc/en/books/developers-handbook/x86-fpu.html. [Accessed November 2018].

[60] Oracle, “x86 Assembly Language Reference Manual,” October 2017. [Online]. Available: https://docs.oracle.com/cd/E53394_01/html/E54851/index.html. [Accessed November 2018].

[61] M. Matz, J. Hubicka, A. Jaeger and M. Mitchell, “System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.7,” 17 November 2014. [Online]. Available: https://www.uclibc.org/docs/psABI-x86_64.pdf. [Accessed November 2018].

[62] NRF iThemba, “Accelerators – Overview,” iThemba Laboratory for Accelerator Based Sciences, 2019. [Online]. Available: https://tlabs.ac.za/accelerators/. [Accessed 14 February 2019].

[63] National Instruments, “OPERATING INSTRUCTIONS AND SPECIFICATIONS NI 9205,” 2008. [Online]. Available: ni.com/manuals. [Accessed 2019 January 2019].

[64] Free Software Foundation, Inc., “6.46.2 Extended Asm - Assembler Instructions with C Expression Operands,” 2018. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm. [Accessed November 2018].

[65] D. M. Hiemstra, S. Yu and M. Pop, “Single Event Upset Characterization of the Pentium® 4, Pentium® III and Low Power Pentium® MMX Microprocessors using

Proton Irradiation," in *IEEE Radiation Effects Data Workshop*, Phoenix, AZ, USA, USA, 2002.

[66] J. Seporaitis, "CPU Registers CR0," GitHub, Inc., 14 April 2017. [Online]. Available: https://github.com/seporaitis/xv6-public/wiki/CPU-Registers-CR0. [Accessed 17 January 2019].

[67] ulmo, "Disabling CPU caches," LinuxQuestions.org, 23 March 2012. [Online]. Available: https://www.linuxquestions.org/questions/linux-kernel-70/disabling-cpu-caches-936077/. [Accessed June 2018].

[68] Fruit Fly Africa, "Welcome to FruitFly Africa," Fruit Fly Africa, 2019. [Online]. Available: http://www.fruitfly.co.za/. [Accessed 15 February 2019].

[69] iThemba and A. Barnard, "Radiation Dose Calculation for ARC Co-60 source," Stellenbosch, 2019.

[70] schoolscience, "Properties of ionising radiations," schoolscience, [Online]. Available: http://resources.schoolscience.co.uk/stfc/14-16/partch5pg3.html. [Accessed 06 April 2019].

[71] A. Barnard, "Report on BLM log data for SEE test at iTL on 2019/01/19," Cape Town, South Africa, 2019.

# Appendices

## Appendix 1: Intel Atom E3815 Specifications [55]


### Intel Atom® Processor E3815

512K Cache, 1.46 GHz

Specifications

Export specifications

**Essentials**

| | |
|---|---|
| Product Collection | Intel® Atom™ Processor E Series |
| Code Name | Products formerly Bay Trail |
| Vertical Segment | Embedded |
| Processor Number | E3815 |
| Status | Launched |
| Launch Date | Q4'13 |
| Lithography | 22 nm |
| Recommended Customer Price | $31.00 |

**Performance**

| | |
|---|---|
| # of Cores | 1 |
| # of Threads | 1 |
| Processor Base Frequency | 1.46 GHz |
| Cache | 512 KB L2 |
| TDP | 5 W |

**Supplemental Information**

| | |
|---|---|
| Embedded Options Available | Yes |
| Datasheet | View now |

**Memory Specifications**

| | |
|---|---|
| Max Memory Size (dependent on memory type) | 8 GB |
| Memory Types | DDR3L 1067 |
| Max # of Memory Channels | 1 |

| | |
|---|---|
| ECC Memory Supported ‡ | Yes |

## Processor Graphics

| | |
|---|---|
| Processor Graphics ‡ | Intel® HD Graphics for Intel Atom® Processor Z3700 Series |
| Graphics Base Frequency | 400 MHz |
| Graphics Burst Frequency | 400 MHz |
| Intel® Quick Sync Video | Yes |
| # of Displays Supported ‡ | 2 |

## Expansion Options

| | |
|---|---|
| PCI Express Revision | 2.0 |
| PCI Express Configurations ‡ | x4, x2, x1 |
| Max # of PCI Express Lanes | 4 |

## I/O Specifications

| | |
|---|---|
| USB Revision | 2.0, 3.0 |
| Total # of SATA Ports | 2 |
| Integrated LAN | No |
| Integrated IDE | No |
| UART | Yes |

## Package Specifications

| | |
|---|---|
| Sockets Supported | FCBGA1170 |
| $T_{JUNCTION}$ | -40°C to 110°C |
| Package Size | 25mm x 27mm |
| Low Halogen Options Available | See MDDS |

## Advanced Technologies

| | |
|---|---|
| Intel® vPro™ Technology ‡ | No |
| Intel® Hyper-Threading Technology ‡ | No |
| Intel® Virtualization Technology (VT-x) ‡ | Yes |
| Intel® Virtualization Technology for Directed I/O (VT-d) ‡ | No |
| Intel® VT-x with Extended Page Tables (EPT) ‡ | Yes |
| Intel® 64 ‡ | Yes |

| | |
|---|---|
| Instruction Set | 64-bit |
| Enhanced Intel SpeedStep® Technology | Yes |
| Intel® HD Audio Technology | Yes |

## Security & Reliability

| | |
|---|---|
| Intel® AES New Instructions | Yes |
| Intel® Trusted Execution Technology ‡ | No |
| Execute Disable Bit ‡ | Yes |

## Ordering and Compliance

## Downloads and Software

C

# Appendix 2: GPR Test Source Code

```
#include <stdio.h>
#include <stdlib.h>

int GPRTest()
{
    /*Value below shall be written to each GPR except %RBP and %RSP*/
    unsigned long long int inputVar = 0xf0f0f0f0f0f0f0f0;

    /*Variables below shall store outputs from the asm program*/
    unsigned long long int readBackInput = 0, outError1 = 0, outError2 = 0,
    outError3 = 0;
    unsigned int stuckBit = 0;      /*If the asm outputs a non-zero value to this
    variable, possibly a stuck bit has been detected, a register has failed to be
    written to or an error has occured in the register that the variable is
    stored in*/
    unsigned int seuOccured = 0;    /*If a non-zero value is returned here, an
    error has occured. It will not be updated if a stuck bit is detected*/
    unsigned int position = 0;      /*This will store the position of the
    register that the error occured in*/

    /*Below is code that tests the GPRs*/
    asm volatile ("Begin:       movq    $0,             %[seuTrue]      \n\t"
                  "             movq    $0,             %[stuck]        \n\t"
                  "             movq    %[inputVal],    %%rax           \n\t"
                  "             movq    $150000000,     %%rbx           \n\t"
                  "LoadRCX:     movq    %%rax,          %%rcx           \n\t"
                  "             cmp     %%rax,          %%rcx           \n\t"
                  "             jne     StuckBitRCX                     \n\t"
                  "LoadRDX:     movq    %%rax,          %%rdx           \n\t"
                  "             cmp     %%rax,          %%rdx           \n\t"
                  "             jne     StuckBitRDX                     \n\t"
                  "LoadRSI:     movq    %%rax,          %%rsi           \n\t"
                  "             cmp     %%rax,          %%rsi           \n\t"
                  "             jne     StuckBitRSI                     \n\t"
                  "LoadRDI:     movq    %%rax,          %%rdi           \n\t"
                  "             cmp     %%rax,          %%rdi           \n\t"
                  "             jne     StuckBitRDI                     \n\t"
                  "LoadR8:      movq    %%rax,          %%r8            \n\t"
                  "             cmp     %%rax,          %%r8            \n\t"
                  "             jne     StuckBitR8                      \n\t"
                  "LoadR9:      movq    %%rax,          %%r9            \n\t"
                  "             cmp     %%rax,          %%r9            \n\t"
                  "             jne     StuckBitR9                      \n\t"
                  "LoadR10:     movq    %%rax,          %%r10           \n\t"
                  "             cmp     %%rax,          %%r10           \n\t"
                  "             jne     StuckBitR10                     \n\t"
                  "LoadR11:     movq    %%rax,          %%r11           \n\t"
                  "             cmp     %%rax,          %%r11           \n\t"
                  "             jne     StuckBitR11                     \n\t"
                  "LoadR12:     movq    %%rax,          %%r12           \n\t"
                  "             cmp     %%rax,          %%r12           \n\t"
                  "             jne     StuckBitR12                     \n\t"
                  "LoadR13:     movq    %%rax,          %%r13           \n\t"
                  "             cmp     %%rax,          %%r13           \n\t"
                  "             jne     StuckBitR13                     \n\t"
                  "LoadR14:     movq    %%rax,          %%r14           \n\t"
                  "             cmp     %%rax,          %%r14           \n\t"
                  "             jne     StuckBitR14                     \n\t"
                  "LoadR15:     movq    %%rax,          %%r15           \n\t"
                  "             cmp     %%rax,          %%r15           \n\t"
                  "             jne     StuckBitR15                     \n\t"
                  "             jmp     Iterate                         \n\t"

                  /*The code below retries writing to registers that have failed
                  to have the correct value written to them. After 2 failed
```

```
attempts,the asm exits and updates the "stuck bit" flag*/

"StuckBitRCX: movq    %%rax,          %%rcx           \n\t"
"             cmp     %%rax,          %%rcx           \n\t"
"             je      LoadRDX                         \n\t"
"             movq    %%rax,          %%rcx           \n\t"
"             cmp     %%rax,          %%rcx           \n\t"
"             je      LoadRDX                         \n\t"
"             movq    %%rax,          %[readInput]    \n\t"
"             movq    %%rcx,          %[error1]       \n\t"
"             movq    %%rcx,          %[error2]       \n\t"
"             movq    %%rcx,          %[error3]       \n\t"
"             movb    $0xf0,          %[stuck]        \n\t"
"             movb    $2,             %[location]     \n\t"
"             jmp     End                             \n\t"

"StuckBitRDX: movq    %%rax,          %%rdx           \n\t"
"             cmp     %%rax,          %%rdx           \n\t"
"             je      LoadRSI                         \n\t"
"             movq    %%rax,          %%rdx           \n\t"
"             cmp     %%rax,          %%rdx           \n\t"
"             je      LoadRSI                         \n\t"
"             movq    %%rax,          %[readInput]    \n\t"
"             movq    %%rdx,          %[error1]       \n\t"
"             movq    %%rdx,          %[error2]       \n\t"
"             movq    %%rdx,          %[error3]       \n\t"
"             movb    $0xf0,          %[stuck]        \n\t"
"             movb    $3,             %[location]     \n\t"
"             jmp     End                             \n\t"

"StuckBitRSI: movq    %%rax,          %%rsi           \n\t"
"             cmp     %%rax,          %%rsi           \n\t"
"             je      LoadRDI                         \n\t"
"             movq    %%rax,          %%rsi           \n\t"
"             cmp     %%rax,          %%rsi           \n\t"
"             je      LoadRDI                         \n\t"
"             movq    %%rax,          %[readInput]    \n\t"
"             movq    %%rsi,          %[error1]       \n\t"
"             movq    %%rsi,          %[error2]       \n\t"
"             movq    %%rsi,          %[error3]       \n\t"
"             movb    $0xf0,          %[stuck]        \n\t"
"             movb    $5,             %[location]     \n\t"
"             jmp     End                             \n\t"

"StuckBitRDI: movq    %%rax,          %%rdi           \n\t"
"             cmp     %%rax,          %%rdi           \n\t"
"             je      LoadR8                          \n\t"
"             movq    %%rax,          %%rdi           \n\t"
"             cmp     %%rax,          %%rdi           \n\t"
"             je      LoadR8                          \n\t"
"             movq    %%rax,          %[readInput]    \n\t"
"             movq    %%rdi,          %[error1]       \n\t"
"             movq    %%rdi,          %[error2]       \n\t"
"             movq    %%rdi,          %[error3]       \n\t"
"             movb    $0xf0,          %[stuck]        \n\t"
"             movb    $6,             %[location]     \n\t"
"             jmp     End                             \n\t"


"StuckBitR8:  movq    %%rax,          %%r8            \n\t"
"             cmp     %%rax,          %%r8            \n\t"
"             je      LoadR9                          \n\t"
"             movq    %%rax,          %%r8            \n\t"
"             cmp     %%rax,          %%r8            \n\t"
"             je      LoadR9                          \n\t"
"             movq    %%rax,          %[readInput]    \n\t"
"             movq    %%r8,           %[error1]       \n\t"
"             movq    %%r8,           %[error2]       \n\t"
```

```
"             movq    %%r8,           %[error3]       \n\t"
"             movb    $0xf0,          %[stuck]        \n\t"
"             movb    $8,             %[location]     \n\t"
"             jmp     End                             \n\t"

"StuckBitR9:  movq    %%rax,          %%r9            \n\t"
"             cmp     %%rax,          %%r9            \n\t"
"             je      LoadR10                         \n\t"
"             movq    %%rax,          %%r9            \n\t"
"             cmp     %%rax,          %%r9            \n\t"
"             je      LoadR10                         \n\t"
"             movq    %%rax,          %[readInput]    \n\t"
"             movq    %%r9,           %[error1]       \n\t"
"             movq    %%r9,           %[error2]       \n\t"
"             movq    %%r9,           %[error3]       \n\t"
"             movb    $0xf0,          %[stuck]        \n\t"
"             movb    $9,             %[location]     \n\t"
"             jmp     End                             \n\t"

"StuckBitR10: movq    %%rax,          %%r10           \n\t"
"             cmp     %%rax,          %%r10           \n\t"
"             je      LoadR11                         \n\t"
"             movq    %%rax,          %%r10           \n\t"
"             cmp     %%rax,          %%r10           \n\t"
"             je      LoadR11                         \n\t"
"             movq    %%rax,          %[readInput]    \n\t"
"             movq    %%r10,          %[error1]       \n\t"
"             movq    %%r10,          %[error2]       \n\t"
"             movq    %%r10,          %[error3]       \n\t"
"             movb    $0xf0,          %[stuck]        \n\t"
"             movb    $10,             %[location]    \n\t"
"             jmp     End                             \n\t"

"StuckBitR11: movq    %%rax,          %%r11           \n\t"
"             cmp     %%rax,          %%r11           \n\t"
"             je      LoadR12                         \n\t"
"             movq    %%rax,          %%r11           \n\t"
"             cmp     %%rax,          %%r11           \n\t"
"             je      LoadR12                         \n\t"
"             movq    %%rax,          %[readInput]    \n\t"
"             movq    %%r11,          %[error1]       \n\t"
"             movq    %%r11,          %[error2]       \n\t"
"             movq    %%r11,          %[error3]       \n\t"
"             movb    $0xf0,          %[stuck]        \n\t"
"             movb    $11,            %[location]     \n\t"
"             jmp     End                             \n\t"

"StuckBitR12: movq    %%rax,          %%r12           \n\t"
"             cmp     %%rax,          %%r12           \n\t"
"             je      LoadR13                         \n\t"
"             movq    %%rax,          %%r12           \n\t"
"             cmp     %%rax,          %%r12           \n\t"
"             je      LoadR13                         \n\t"
"             movq    %%rax,          %[readInput]    \n\t"
"             movq    %%r12,          %[error1]       \n\t"
"             movq    %%r12,          %[error2]       \n\t"
"             movq    %%r12,          %[error3]       \n\t"
"             movb    $0xf0,          %[stuck]        \n\t"
"             movb    $12,            %[location]     \n\t"
"             jmp     End                             \n\t"

"StuckBitR13: movq    %%rax,          %%r13           \n\t"
"             cmp     %%rax,          %%r13           \n\t"
"             je      LoadR14                         \n\t"
"             movq    %%rax,          %%r13           \n\t"
"             cmp     %%rax,          %%r13           \n\t"
"             je      LoadR14                         \n\t"
"             movq    %%rax,          %[readInput]    \n\t"
```

```
 "             movq    %%r13,          %[error1]       \n\t"
 "             movq    %%r13,          %[error2]       \n\t"
 "             movq    %%r13,          %[error3]       \n\t"
 "             movb    $0xf0,          %[stuck]        \n\t"
 "             movb    $13,            %[location]     \n\t"
 "             jmp     End                             \n\t"

 "StuckBitR14: movq    %%rax,          %%r14           \n\t"
 "             cmp     %%rax,          %%r14           \n\t"
 "             je      LoadR15                         \n\t"
 "             movq    %%rax,          %%r14           \n\t"
 "             cmp     %%rax,          %%r14           \n\t"
 "             je      LoadR15                         \n\t"
 "             movq    %%rax,          %[readInput]    \n\t"
 "             movq    %%r14,          %[error1]       \n\t"
 "             movq    %%r14,          %[error2]       \n\t"
 "             movq    %%r14,          %[error3]       \n\t"
 "             movb    $0xf0,          %[stuck]        \n\t"
 "             movb    $14,            %[location]     \n\t"
 "             jmp     End                             \n\t"

 "StuckBitR15: movq    %%rax,          %%r15           \n\t"
 /*retry writing the value*/
 "             cmp     %%rax,          %%r15           \n\t"
 /*Check if it worked*/
 "             je      Iterate                         \n\t"
 /*Resume with program if succesful*/
 "             movq    %%rax,          %%r15           \n\t"
/*retry writing the value one more time*/
 "             cmp     %%rax,          %%r15           \n\t"
 /*Check if it worked*/
 "             je      Iterate                         \n\t"
/*Resume with program if succesful*/
 "             movq    %%rax,          %[readInput]    \n\t"
/*Output the value that was to be written to the register*/
 "             movq    %%r15,          %[error1]       \n\t"
/*Unsuccesful, writing erronous values to output variables*/
 "             movq    %%r15,          %[error2]       \n\t"
 "             movq    %%r15,          %[error3]       \n\t"
 "             movb    $0xf0,          %[stuck]        \n\t"
 /*Set the stuck bit flag*/
 "             movb    $15,            %[location]     \n\t"
 /*Save the locaion of the register*/
 "             jmp     End                             \n\t"
 /*End the asm function*/

 /*Code segment below checks for any errors that occur during
 execution*/

 "RCX_Error:   movq    %%rax,          %[readInput]    \n\t"
 "             movq    %%rcx,          %[error1]       \n\t"
 "             movq    %%rcx,          %[error2]       \n\t"
 "             movq    %%rcx,          %[error3]       \n\t"
 "             movb    $2,             %[location]     \n\t"
 "             movb    $0xf0,          %[seuTrue]      \n\t"
 "             jmp     End                             \n\t"

 "RDX_Error:   movq    %%rax,          %[readInput]    \n\t"
 "             movq    %%rdx,          %[error1]       \n\t"
 "             movq    %%rdx,          %[error2]       \n\t"
 "             movq    %%rdx,          %[error3]       \n\t"
 "             movb    $3,             %[location]     \n\t"
 "             movb    $0xf0,          %[seuTrue]      \n\t"
 "             jmp     End                             \n\t"

 "RSI_Error:   movq    %%rax,          %[readInput]    \n\t"
 "             movq    %%rsi,          %[error1]       \n\t"
 "             movq    %%rsi,          %[error2]       \n\t"
```

```
"             movq    %%rsi,          %[error3]       \n\t"
"             movb    $5,             %[location]     \n\t"
"             movb    $0xf0,          %[seuTrue]      \n\t"
"             jmp     End                             \n\t"

"RDI_Error:   movq    %%rax,          %[readInput]    \n\t"
"             movq    %%rdi,          %[error1]       \n\t"
"             movq    %%rdi,          %[error2]       \n\t"
"             movq    %%rdi,          %[error3]       \n\t"
"             movb    $6,             %[location]     \n\t"
"             movb    $0xf0,          %[seuTrue]      \n\t"
"             jmp     End                             \n\t"

"R8_Error:    movq    %%rax,          %[readInput]    \n\t"
"             movq    %%r8,           %[error1]       \n\t"
"             movq    %%r8,           %[error2]       \n\t"
"             movq    %%r8,           %[error3]       \n\t"
"             movb    $8,             %[location]     \n\t"
"             movb    $0xf0,          %[seuTrue]      \n\t"
"             jmp     End                             \n\t"

"R9_Error:    movq    %%rax,          %[readInput]    \n\t"
"             movq    %%r9,           %[error1]       \n\t"
"             movq    %%r9,           %[error2]       \n\t"
"             movq    %%r9,           %[error3]       \n\t"
"             movb    $9,             %[location]     \n\t"
"             movb    $0xf0,          %[seuTrue]      \n\t"
"             jmp     End                             \n\t"

"R10_Error:   movq    %%rax,          %[readInput]    \n\t"
"             movq    %%r10,          %[error1]       \n\t"
"             movq    %%r10,          %[error2]       \n\t"
"             movq    %%r10,          %[error3]       \n\t"
"             movb    $10,            %[location]     \n\t"
"             movb    $0xf0,          %[seuTrue]      \n\t"
"             jmp     End                             \n\t"

"R11_Error:   movq    %%rax,          %[readInput]    \n\t"
"             movq    %%r11,          %[error1]       \n\t"
"             movq    %%r11,          %[error2]       \n\t"
"             movq    %%r11,          %[error3]       \n\t"
"             movb    $11,            %[location]     \n\t"
"             movb    $0xf0,          %[seuTrue]      \n\t"
"             jmp     End                             \n\t"

"R12_Error:   movq    %%rax,          %[readInput]    \n\t"
"             movq    %%r12,          %[error1]       \n\t"
"             movq    %%r12,          %[error2]       \n\t"
"             movq    %%r12,          %[error3]       \n\t"
"             movb    $12,            %[location]     \n\t"
"             movb    $0xf0,          %[seuTrue]      \n\t"
"             jmp     End                             \n\t"

"R13_Error:   movq    %%rax,          %[readInput]    \n\t"
"             movq    %%r13,          %[error1]       \n\t"
"             movq    %%r13,          %[error2]       \n\t"
"             movq    %%r13,          %[error3]       \n\t"
"             movb    $13,            %[location]     \n\t"
"             movb    $0xf0,          %[seuTrue]      \n\t"
"             jmp     End                             \n\t"

"R14_Error:   movq    %%rax,          %[readInput]    \n\t"
"             movq    %%r14,          %[error1]       \n\t"
"             movq    %%r14,          %[error2]       \n\t"
"             movq    %%r14,          %[error3]       \n\t"
"             movb    $14,            %[location]     \n\t"
"             movb    $0xf0,          %[seuTrue]      \n\t"
"             jmp     End                             \n\t"
```

```
                "R15_Error:   movq    %%rax,          %[readInput]    \n\t"
                "             movq    %%r15,          %[error1]       \n\t"
                "             movq    %%r15,          %[error2]       \n\t"
                "             movq    %%r15,          %[error3]       \n\t"
                "             movb    $15,            %[location]     \n\t"
                "             movb    $0xf0,          %[seuTrue]      \n\t"
                "             jmp     End                             \n\t"

                  /*The code segment belowis a loop that checks the registers
                  to see if any value has changed since loaded*/

                "Iterate:                                             \n\t"
                "             cmp     %%rax,          %%rcx           \n\t"
                "             jne     RCX_Error                       \n\t"
                "             cmp     %%rax,          %%rdx           \n\t"
                "             jne     RDX_Error                       \n\t"
                "             cmp     %%rax,          %%rsi           \n\t"
                "             jne     RSI_Error                       \n\t"
                "             cmp     %%rax,          %%rdi           \n\t"
                "             jne     RDI_Error                       \n\t"
                "             cmp     %%rax,          %%r8            \n\t"
                "             jne     R8_Error                        \n\t"
                "             cmp     %%rax,          %%r9            \n\t"
                "             jne     R9_Error                        \n\t"
                "             cmp     %%rax,          %%r10           \n\t"
                "             jne     R10_Error                       \n\t"
                "             cmp     %%rax,          %%r11           \n\t"
                "             jne     R11_Error                       \n\t"
                "             cmp     %%rax,          %%r12           \n\t"
                "             jne     R12_Error                       \n\t"
                "             cmp     %%rax,          %%r13           \n\t"
                "             jne     R13_Error                       \n\t"
                "             cmp     %%rax,          %%r14           \n\t"
                "             jne     R14_Error                       \n\t"
                "             cmp     %%rax,          %%r15           \n\t"
                "             jne     R15_Error                       \n\t"

                "             dec     %%rbx                           \n\t"
                /*Decrementing the loop counter*/
                "             cmpq    $0,             %%rbx           \n\t"
                "             jne     Iterate                         \n\t"
                "End:         nop                                     \n\t"


                : [readInput]    "=m" (readBackInput),
                  [error1]       "=m" (outError1),
                  [error2]       "=m" (outError2),
                  [error3]       "=m" (outError3),
                  [stuck]        "=m" (stuckBit),
                  [seuTrue]      "=m" (seuOccured),
                  [location]     "=m" (position)

                : [inputVal]  "m" (inputVar)

                :"%rax","%rbx","%rcx","%rdx","%rsi","%rdi","%r8","%r9","%r10",
                 "%r11","%r12","%r13","%r14","%r15");

                /*Data logging follows*/
if (stuckBit == 0xf0)
   {
    /*First we print the mssage on the console*/
    printf("Failed to write %d\t Input: %llx\t Error1: %llx\t Error2: %llx\t
    Error3: %llx\t", position, readBackInput, outError1,outError2,outError3);

    /*Now we write to file*/
    FILE * myFilePointer = fopen("GPR_Failed_Writes.csv", "a");
    fprintf(myFilePointer, "%d, %llx, %llx, %llx, %llx\n", position,
```

```
            readBackInput, outError1,outError2,outError3);
            fclose(myFilePointer);
            return 1;
           }
        else if (seuOccured == 0xf0)
           {
            /*First we print the message on the console*/
            printf("Error at %d\t Input: %llx\t Error1: %llx\t Error2: %llx\t
             Error3: %llx\t", position,readBackInput,outError1,outError2,outError3);

            /*Now we write to file*/
            FILE * myFilePointer = fopen("GPR_Errors.csv", "a");
            fprintf(myFilePointer, "%d, %llx, %llx, %llx, %llx\n", position,
            readBackInput, outError1,outError2,outError3);
            fclose(myFilePointer);
            return 2;
           }
        else
           {
            printf("No error detected");
            return 0;
           }
}


int main()
{
    int numTestRuns;
    float energy,flux;

    printf("Enter Test Energy in MeV\n");
    scanf("%f", &energy);
    printf("Enter test flux\n");
    scanf("%f", &flux);
    printf("Enter the number of tests to run (for loop counter)\n");
    scanf("%d", &numTestRuns);

    /*We beginby creating the files that will be used to log errors or failed
    writes, as well as test parameters*/
    FILE * fp1 = fopen("GPR_Failed_Writes.csv", "a");
    fprintf(fp1, "\n");
    fprintf(fp1, "New test run\n");
    fprintf(fp1, "============\n");
    fprintf(fp1,"Energy: %f, Flux: %f\n", energy, flux);
    fprintf(fp1, "REG Locaton, Read Back Value, Error 1, Error 2, Error 3\n");
    fclose(fp1);

    FILE * fp2 = fopen("GPR_Errors.csv", "a");
    fprintf(fp2, "\n");
    fprintf(fp2, "New test run\n");
    fprintf(fp2, "============\n");
    fprintf(fp2,"Energy: %f, Flux: %f\n", energy, flux);
    fprintf(fp2, "REG Locaton, Read Back Value, Error 1, Error 2, Error 3\n");
    fclose(fp2);

    printf("Log file Appended. Beginning test\n");

    /*Now we run the actual test*/
    int errCount = 0, writeFailCount = 0;
    for (int i = 0; i < numTestRuns; i++)
    {
        printf("Run %d: ",i);
        int errType = GPRTest();
        if (errType == 1)
        {
            writeFailCount++;
            printf("WF No: %d\n",writeFailCount);
        }
```

J

```
        else if (errType == 2)
        {
            errCount++;
            printf("E No: %d\n",errCount);
        }
        else
            printf("\n");
    }
    printf("\n\nGPR Test Program Completed\n");

    return 0;
}
```

# Appendix 3: MMX Test Source Code

```
#include <stdio.h>
#include <stdlib.h>

int MMXTest ()
{
    /*Below are vriables declared to be input to the asm*/
    /*Do not change the writeval during execution. It is hardcoded in the asm*/
    long long unsigned int writeVal = 0xf0f0f0f0f0f0f0f0, numLoops = 150000000;

    /*Below are variables to hold data output by the asm*/
    long long unsigned int error1, error2, error3, currentMMXVal;
    unsigned int location = 0xf0;

    asm volatile ("Begin:       movq        %[loopCount],   %%rdx           \n\t"
                  "             movq        %[toWrite],     %%r8            \n\t"
                  "             movq        %[toWrite],     %%r9            \n\t"
                  "             movq        %[toWrite],     %%r10           \n\t"
                  "             movq        %[toWrite],     %%mm0           \n\t"
                  "             movq        %%mm0,          %%mm1           \n\t"
                  "             movq        %%mm0,          %%mm2           \n\t"
                  "             movq        %%mm0,          %%mm3           \n\t"
                  "             movq        %%mm0,          %%mm4           \n\t"
                  "             movq        %%mm0,          %%mm5           \n\t"
                  "             movq        %%mm0,          %%mm6           \n\t"
                  "             movq        %%mm0,          %%mm7           \n\t"

                  /*Now that the registers are loaded, we check for any errors.
                  Unfortunately due to the inability of MMX registers to be used
                  as operands for branch commands, we will not be able to
                  differenciate a bitflip(s) error from a write fail error*/

                  "CheckMM0:    movq        %%mm0,          %%rax           \n\t"
                  "             movq        %%mm0,          %%rbx           \n\t"
                  "             movq        %%mm0,          %%rcx           \n\t"
                  "             cmp         %%r8,           %%rax           \n\t"
                  "             je          CheckMM1                        \n\t"
                  "             cmp         %%r9,           %%rbx           \n\t"
                  "             je          CheckMM1                        \n\t"
                  "             cmp         %%r10,          %%rcx           \n\t"
                  "             je          CheckMM1                        \n\t"
                  "             movq        %%mm0,          %[readBackVal]  \n\t"
                  "             movq        %%rax,          %[err1]         \n\t"
                  "             movq        %%rbx,          %[err2]         \n\t"
                  "             movq        %%rcx,          %[err3]         \n\t"
                  "             movb        $0,             %[position]     \n\t"
                  "             jmp         End                             \n\t"

                  "CheckMM1:    movq        %%mm1,          %%rax           \n\t"
                  "             movq        %%mm1,          %%rbx           \n\t"
                  "             movq        %%mm1,          %%rcx           \n\t"
                  "             cmp         %%r8,           %%rax           \n\t"
                  "             je          CheckMM2                        \n\t"
                  "             cmp         %%r9,           %%rbx           \n\t"
                  "             je          CheckMM2                        \n\t"
```

```
"             cmp         %%r10,          %%rcx           \n\t"
"             je          CheckMM2                        \n\t"
"             movq        %%mm1,          %[readBackVal]  \n\t"
"             movq        %%rax,          %[err1]         \n\t"
"             movq        %%rbx,          %[err2]         \n\t"
"             movq        %%rcx,          %[err3]         \n\t"
"             movb        $1,             %[position]     \n\t"
"             jmp         End                             \n\t"

"CheckMM2:    movq        %%mm2,          %%rax           \n\t"
"             movq        %%mm2,          %%rbx           \n\t"
"             movq        %%mm2,          %%rcx           \n\t"
"             cmp         %%r8,           %%rax           \n\t"
"             je          CheckMM3                        \n\t"
"             cmp         %%r9,           %%rbx           \n\t"
"             je          CheckMM3                        \n\t"
"             cmp         %%r10,          %%rcx           \n\t"
"             je          CheckMM3                        \n\t"
"             movq        %%mm2,          %[readBackVal]  \n\t"
"             movq        %%rax,          %[err1]         \n\t"
"             movq        %%rbx,          %[err2]         \n\t"
"             movq        %%rcx,          %[err3]         \n\t"
"             movb        $2,             %[position]     \n\t"
"             jmp         End                             \n\t"

"CheckMM3:    movq        %%mm3,          %%rax           \n\t"
"             movq        %%mm3,          %%rbx           \n\t"
"             movq        %%mm3,          %%rcx           \n\t"
"             cmp         %%r8,           %%rax           \n\t"
"             je          CheckMM4                        \n\t"
"             cmp         %%r9,           %%rbx           \n\t"
"             je          CheckMM4                        \n\t"
"             cmp         %%r10,          %%rcx           \n\t"
"             je          CheckMM4                        \n\t"
"             movq        %%mm3,          %[readBackVal]  \n\t"
"             movq        %%rax,          %[err1]         \n\t"
"             movq        %%rbx,          %[err2]         \n\t"
"             movq        %%rcx,          %[err3]         \n\t"
"             movb        $3,             %[position]     \n\t"
"             jmp         End                             \n\t"

"CheckMM4:    movq        %%mm4,          %%rax           \n\t"
"             movq        %%mm4,          %%rbx           \n\t"
"             movq        %%mm4,          %%rcx           \n\t"
"             cmp         %%r8,           %%rax           \n\t"
"             je          CheckMM5                        \n\t"
"             cmp         %%r9,           %%rbx           \n\t"
"             je          CheckMM5                        \n\t"
"             cmp         %%r10,          %%rcx           \n\t"
"             je          CheckMM5                        \n\t"
"             movq        %%mm4,          %[readBackVal]  \n\t"
"             movq        %%rax,          %[err1]         \n\t"
"             movq        %%rbx,          %[err2]         \n\t"
"             movq        %%rcx,          %[err3]         \n\t"
"             movb        $4,             %[position]     \n\t"
"             jmp         End                             \n\t"
```

```
"CheckMM5:    movq        %%mm5,          %%rax           \n\t"
"             movq        %%mm5,          %%rbx           \n\t"
"             movq        %%mm5,          %%rcx           \n\t"
"             cmp         %%r8,           %%rax           \n\t"
"             je          CheckMM6                        \n\t"
"             cmp         %%r9,           %%rbx           \n\t"
"             je          CheckMM6                        \n\t"
"             cmp         %%r10,          %%rcx           \n\t"
"             je          CheckMM6                        \n\t"
"             movq        %%mm5,          %[readBackVal]  \n\t"
"             movq        %%rax,          %[err1]         \n\t"
"             movq        %%rbx,          %[err2]         \n\t"
"             movq        %%rcx,          %[err3]         \n\t"
"             movb        $5,             %[position]     \n\t"
"             jmp         End                             \n\t"

"CheckMM6:    movq        %%mm6,          %%rax           \n\t"
"             movq        %%mm6,          %%rbx           \n\t"
"             movq        %%mm6,          %%rcx           \n\t"
"             cmp         %%r8,           %%rax           \n\t"
"             je          CheckMM7                        \n\t"
"             cmp         %%r9,           %%rbx           \n\t"
"             je          CheckMM7                        \n\t"
"             cmp         %%r10,          %%rcx           \n\t"
"             je          CheckMM7                        \n\t"
"             movq        %%mm6,          %[readBackVal]  \n\t"
"             movq        %%rax,          %[err1]         \n\t"
"             movq        %%rbx,          %[err2]         \n\t"
"             movq        %%rcx,          %[err3]         \n\t"
"             movb        $6,             %[position]     \n\t"
"             jmp         End                             \n\t"

"CheckMM7:    movq        %%mm7,          %%rax           \n\t"
"             movq        %%mm7,          %%rbx           \n\t"
"             movq        %%mm7,          %%rcx           \n\t"
"             cmp         %%r8,           %%rax           \n\t"
"             je          Loop                            \n\t"
"             cmp         %%r9,           %%rbx           \n\t"
"             je          Loop                            \n\t"
"             cmp         %%r10,          %%rcx           \n\t"
"             je          Loop                            \n\t"
"             movq        %%mm7,          %[readBackVal]  \n\t"
"             movq        %%rax,          %[err1]         \n\t"
"             movq        %%rbx,          %[err2]         \n\t"
"             movq        %%rcx,          %[err3]         \n\t"
"             movb        $7,             %[position]     \n\t"
"             jmp         End                             \n\t"


"Loop:        dec         %%rdx                           \n\t"
"             cmpq        $0,             %%rdx           \n\t"
"             jne         CheckMM0                        \n\t"

"End:         emms                                        \n\t"
/*Thos opcode releases the FPU to be used for other functions*/
```

N

```
                    : [readBackVal]   "=m" (currentMMXVal),
                      [err1]          "=m" (error1),
                      [err2]          "=m" (error2),
                      [err3]          "=m" (error3),
                      [position]      "=m" (location)

                    : [toWrite]   "m" (writeVal),
                      [loopCount] "m" (numLoops)

                    :"%rax","%rbx","%rcx","%rdx","%r8","%r9","%r10","%mm0","%mm1",
                     "%mm2","%mm3","%mm4","%mm5","%mm6","%mm7");

    if(location != 0xf0)
       {
        /*First we prompt the user*/
        printf("Error at %d\t Input: %llx\t Error1: %llx\t Error2: %llx\t Error3:
        %llx\t", location, currentMMXVal, error1,error2,error3);
        /*Now we write to file*/
        FILE * myFilePointer = fopen("MMX_Errors.csv", "a");
        fprintf(myFilePointer, "%d, %llx, %llx, %llx, %llx\n", location,
        currentMMXVal, error1,error2,error3);
        fclose(myFilePointer);
        return 1;
       }
    else
       {
         printf("No error detected");
         return 0;
       }
}

int main()
{
    int numTestRuns;
    float energy,flux;

    printf("Enter Test Energy in MeV\n");
    scanf("%f", &energy);
    printf("Enter test flux\n");
    scanf("%f", &flux);
    printf("Enter the number of tests to run (for loop counter)\n");
    scanf("%d", &numTestRuns);

    /*We beginby creating the files that will be used to log errors as well as
    test parameters*/
    FILE * fp = fopen("MMX_Errors.csv", "a");
    fprintf(fp, "\n");
    fprintf(fp, "New test run\n");
    fprintf(fp, "============\n");
    fprintf(fp,"Energy: %f, Flux: %f\n", energy, flux);
    fprintf(fp, "REG Locaton, Read Back Value, Error 1, Error 2, Error 3\n");
    fclose(fp);

    printf("Log file Appended. Beginning test\n");

    /*Now we run the actual test*/
    int errCount = 0;
```

```
    for (int i = 0; i < numTestRuns; i++)
    {
        printf("Run %d: ",i);
        int errType = MMXTest();
        if (errType == 1)
        {
            errCount++;
            printf("Error No: %d\n",errCount);
        }
        else
            printf("\n");
    }
    printf("\n\nGPR Test Program Completed\n");
    return 0;
}
```

# Appendix 4: XMM Test Source Code

```
#include <stdio.h>
#include <stdlib.h>

int XMMTest()
{
    /*Below are inputs to the asm*/
    unsigned int loopCount = 1500000;
    unsigned long long int inputVal = 0xf0f0f0f0f0f0f0f0;

    /*outputs from the asm*/
    unsigned long long int error1H, error2H, error3H, error1L, error2L, error3L;
    unsigned long long int xmmValH, xmmValL;
    unsigned int errorFlag = 0, location = 0xf0;

    /*Begin asm*/

    asm volatile("Begin:        movq    %[toWrite],     %%rax       \n\t"
                 "              movq    %[toWrite],     %%rbx       \n\t"
                 "              movq    %[toWrite],     %%rcx       \n\t"
                 "              movq    %[numLoops],    %%rdx       \n\t"

                 "              movlps  %[toWrite],     %%xmm0      \n\t"
                 "              movhps  %[toWrite],     %%xmm0      \n\t"
                 /*Loading the xmm registers with known values, excluding xmm15*/
                 "              movaps  %%xmm0,         %%xmm1      \n\t"
                 "              movaps  %%xmm0,         %%xmm2      \n\t"
                 "              movaps  %%xmm0,         %%xmm3      \n\t"
                 "              movaps  %%xmm0,         %%xmm4      \n\t"
                 "              movaps  %%xmm0,         %%xmm5      \n\t"
                 "              movaps  %%xmm0,         %%xmm6      \n\t"
                 "              movaps  %%xmm0,         %%xmm7      \n\t"
                 "              movaps  %%xmm0,         %%xmm8      \n\t"
                 "              movaps  %%xmm0,         %%xmm9      \n\t"
                 "              movaps  %%xmm0,         %%xmm10     \n\t"
                 "              movaps  %%xmm0,         %%xmm11     \n\t"
                 "              movaps  %%xmm0,         %%xmm12     \n\t"
                 "              movaps  %%xmm0,         %%xmm13     \n\t"
                 "              movaps  %%xmm0,         %%xmm14     \n\t"


                 /*Now we begin checking for errors that may occur during
                 irradiation. XMM15 shall be used as a placeholder register
                 for swapping data between registers*/

                 "CheckXMM0L:   movq    %%xmm0,         %%r10       \n\t"
                 "              movq    %%xmm0,         %%r11       \n\t"
                 "              movq    %%xmm0,         %%r12       \n\t"
                 "              movhlps %%xmm0,         %%xmm15     \n\t"
                 /*Only way to copy from XMM to GPR is through lower quadword*/
                 "              movq    %%xmm15,        %%r13       \n\t"
                 "              movq    %%xmm15,        %%r14       \n\t"
                 "              movq    %%xmm15,        %%r15       \n\t"
                 "              cmp     %%r10,          %%rax       \n\t"
                 "              je      CheckXMM0H                  \n\t"
```

Q

```
"               cmp     %%r11,          %%rbx       \n\t"
"               je      CheckXMM0H                  \n\t"
"               cmp     %%r12,          %%rcx       \n\t"
"               je      CheckXMM0H                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm0,         %[readBackL]\n\t"
"               movhps  %%xmm0,         %[readBackH]\n\t"
"               movb    $0,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM0H:    cmp     %%r13,          %%rax       \n\t"
"               je      CheckXMM1L                  \n\t"
"               cmp     %%r14,          %%rbx       \n\t"
"               je      CheckXMM1L                  \n\t"
"               cmp     %%r15,          %%rcx       \n\t"
"               je      CheckXMM1L                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm0,         %[readBackL]\n\t"
"               movhps  %%xmm0,         %[readBackH]\n\t"
"               movb    $0,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM1L:    movq    %%xmm1,         %%r10       \n\t"
"               movq    %%xmm1,         %%r11       \n\t"
"               movq    %%xmm1,         %%r12       \n\t"
"               movhlps %%xmm1,         %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"               movq    %%xmm15,        %%r13       \n\t"
"               movq    %%xmm15,        %%r14       \n\t"
"               movq    %%xmm15,        %%r15       \n\t"
"               cmp     %%r10,          %%rax       \n\t"
"               je      CheckXMM1H                  \n\t"
"               cmp     %%r11,          %%rbx       \n\t"
"               je      CheckXMM1H                  \n\t"
"               cmp     %%r12,          %%rcx       \n\t"
"               je      CheckXMM1H                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm1,         %[readBackL]\n\t"
"               movhps  %%xmm1,         %[readBackH]\n\t"
"               movb    $1,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
```

R

```
"CheckXMM1H:   cmp     %%r13,          %%rax       \n\t"
"              je      CheckXMM2L                  \n\t"
"              cmp     %%r14,          %%rbx       \n\t"
"              je      CheckXMM2L                  \n\t"
"              cmp     %%r15,          %%rcx       \n\t"
"              je      CheckXMM2L                  \n\t"
"              movq    %%r10,          %[err1L]    \n\t"
"              movq    %%r11,          %[err2L]    \n\t"
"              movq    %%r12,          %[err3L]    \n\t"
"              movq    %%r13,          %[err1H]    \n\t"
"              movq    %%r14,          %[err2H]    \n\t"
"              movq    %%r15,          %[err3H]    \n\t"
"              movlps  %%xmm1,         %[readBackL]\n\t"
"              movhps  %%xmm1,         %[readBackH]\n\t"
"              movb    $1,             %[position] \n\t"
"              movb    $0xf0,          %[errFlag]  \n\t"
"              jmp     End                         \n\t"
"CheckXMM2L:   movq    %%xmm2,         %%r10       \n\t"
"              movq    %%xmm2,         %%r11       \n\t"
"              movq    %%xmm2,         %%r12       \n\t"
"              movhlps %%xmm2,         %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"              movq    %%xmm15,        %%r13       \n\t"
"              movq    %%xmm15,        %%r14       \n\t"
"              movq    %%xmm15,        %%r15       \n\t"
"              cmp     %%r10,          %%rax       \n\t"
"              je      CheckXMM2H                  \n\t"
"              cmp     %%r11,          %%rbx       \n\t"
"              je      CheckXMM2H                  \n\t"
"              cmp     %%r12,          %%rcx       \n\t"
"              je      CheckXMM2H                  \n\t"
"              movq    %%r10,          %[err1L]    \n\t"
"              movq    %%r11,          %[err2L]    \n\t"
"              movq    %%r12,          %[err3L]    \n\t"
"              movq    %%r13,          %[err1H]    \n\t"
"              movq    %%r14,          %[err2H]    \n\t"
"              movq    %%r15,          %[err3H]    \n\t"
"              movlps  %%xmm2,         %[readBackL]\n\t"
"              movhps  %%xmm2,         %[readBackH]\n\t"
"              movb    $2,             %[position] \n\t"
"              movb    $0xf0,          %[errFlag]  \n\t"
"              jmp     End                         \n\t"
"CheckXMM2H:   cmp     %%r13,          %%rax       \n\t"
"              je      CheckXMM3L                  \n\t"
"              cmp     %%r14,          %%rbx       \n\t"
"              je      CheckXMM3L                  \n\t"
"              cmp     %%r15,          %%rcx       \n\t"
"              je      CheckXMM3L                  \n\t"
"              movq    %%r10,          %[err1L]    \n\t"
"              movq    %%r11,          %[err2L]    \n\t"
"              movq    %%r12,          %[err3L]    \n\t"
"              movq    %%r13,          %[err1H]    \n\t"
"              movq    %%r14,          %[err2H]    \n\t"
"              movq    %%r15,          %[err3H]    \n\t"
"              movlps  %%xmm2,         %[readBackL]\n\t"
"              movhps  %%xmm2,         %[readBackH]\n\t"
"              movb    $2,             %[position] \n\t"
```

S

```
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM3L:    movq    %%xmm3,         %%r10       \n\t"
"               movq    %%xmm3,         %%r11       \n\t"
"               movq    %%xmm3,         %%r12       \n\t"
"               movhlps %%xmm3,         %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"               movq    %%xmm15,        %%r13       \n\t"
"               movq    %%xmm15,        %%r14       \n\t"
"               movq    %%xmm15,        %%r15       \n\t"
"               cmp     %%r10,          %%rax       \n\t"
"               je      CheckXMM3H                  \n\t"
"               cmp     %%r11,          %%rbx       \n\t"
"               je      CheckXMM3H                  \n\t"
"               cmp     %%r12,          %%rcx       \n\t"
"               je      CheckXMM3H                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm3,         %[readBackL]\n\t"
"               movhps  %%xmm3,         %[readBackH]\n\t"
"               movb    $3,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM3H:    cmp     %%r13,          %%rax       \n\t"
"               je      CheckXMM4L                  \n\t"
"               cmp     %%r14,          %%rbx       \n\t"
"               je      CheckXMM4L                  \n\t"
"               cmp     %%r15,          %%rcx       \n\t"
"               je      CheckXMM4L                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm3,         %[readBackL]\n\t"
"               movhps  %%xmm3,         %[readBackH]\n\t"
"               movb    $3,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"

"CheckXMM4L:    movq    %%xmm4,         %%r10       \n\t"
"               movq    %%xmm4,         %%r11       \n\t"
"               movq    %%xmm4,         %%r12       \n\t"
"               movhlps %%xmm4,         %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"               movq    %%xmm15,        %%r13       \n\t"
"               movq    %%xmm15,        %%r14       \n\t"
"               movq    %%xmm15,        %%r15       \n\t"
"               cmp     %%r10,          %%rax       \n\t"
"               je      CheckXMM4H                  \n\t"
"               cmp     %%r11,          %%rbx       \n\t"
"               je      CheckXMM4H                  \n\t"
```

T

```
"               cmp     %%r12,          %%rcx       \n\t"
"               je      CheckXMM4H                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm4,         %[readBackL]\n\t"
"               movhps  %%xmm4,         %[readBackH]\n\t"
"               movb    $4,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM4H:    cmp     %%r13,          %%rax       \n\t"
"               je      CheckXMM5L                  \n\t"
"               cmp     %%r14,          %%rbx       \n\t"
"               je      CheckXMM5L                  \n\t"
"               cmp     %%r15,          %%rcx       \n\t"
"               je      CheckXMM5L                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm4,         %[readBackL]\n\t"
"               movhps  %%xmm4,         %[readBackH]\n\t"
"               movb    $4,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"

"CheckXMM5L:    movq    %%xmm5,         %%r10       \n\t"
"               movq    %%xmm5,         %%r11       \n\t"
"               movq    %%xmm5,         %%r12       \n\t"
"               movhlps %%xmm5,         %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"               movq    %%xmm15,        %%r13       \n\t"
"               movq    %%xmm15,        %%r14       \n\t"
"               movq    %%xmm15,        %%r15       \n\t"
"               cmp     %%r10,          %%rax       \n\t"
"               je      CheckXMM5H                  \n\t"
"               cmp     %%r11,          %%rbx       \n\t"
"               je      CheckXMM5H                  \n\t"
"               cmp     %%r12,          %%rcx       \n\t"
"               je      CheckXMM5H                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm5,         %[readBackL]\n\t"
"               movhps  %%xmm5,         %[readBackH]\n\t"
"               movb    $5,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM5H:    cmp     %%r13,          %%rax       \n\t"
```

U

```
"               je      CheckXMM6L                  \n\t"
"               cmp     %%r14,          %%rbx       \n\t"
"               je      CheckXMM6L                  \n\t"
"               cmp     %%r15,          %%rcx       \n\t"
"               je      CheckXMM6L                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm5,         %[readBackL]\n\t"
"               movhps  %%xmm5,         %[readBackH]\n\t"
"               movb    $5,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM6L:    movq    %%xmm6,         %%r10       \n\t"
"               movq    %%xmm6,         %%r11       \n\t"
"               movq    %%xmm6,         %%r12       \n\t"
"               movhlps %%xmm6,         %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"               movq    %%xmm15,        %%r13       \n\t"
"               movq    %%xmm15,        %%r14       \n\t"
"               movq    %%xmm15,        %%r15       \n\t"
"               cmp     %%r10,          %%rax       \n\t"
"               je      CheckXMM6H                  \n\t"
"               cmp     %%r11,          %%rbx       \n\t"
"               je      CheckXMM6H                  \n\t"
"               cmp     %%r12,          %%rcx       \n\t"
"               je      CheckXMM6H                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm6,         %[readBackL]\n\t"
"               movhps  %%xmm6,         %[readBackH]\n\t"
"               movb    $6,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM6H:    cmp     %%r13,          %%rax       \n\t"
"               je      CheckXMM7L                  \n\t"
"               cmp     %%r14,          %%rbx       \n\t"
"               je      CheckXMM7L                  \n\t"
"               cmp     %%r15,          %%rcx       \n\t"
"               je      CheckXMM7L                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm6,         %[readBackL]\n\t"
"               movhps  %%xmm6,         %[readBackH]\n\t"
"               movb    $6,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
```

```
"               jmp     End                         \n\t"
"CheckXMM7L:    movq    %%xmm7,         %%r10       \n\t"
"               movq    %%xmm7,         %%r11       \n\t"
"               movq    %%xmm7,         %%r12       \n\t"
"               movhlps %%xmm7,         %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"               movq    %%xmm15,        %%r13       \n\t"
"               movq    %%xmm15,        %%r14       \n\t"
"               movq    %%xmm15,        %%r15       \n\t"
"               cmp     %%r10,          %%rax       \n\t"
"               je      CheckXMM7H                  \n\t"
"               cmp     %%r11,          %%rbx       \n\t"
"               je      CheckXMM7H                  \n\t"
"               cmp     %%r12,          %%rcx       \n\t"
"               je      CheckXMM7H                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm7,         %[readBackL]\n\t"
"               movhps  %%xmm7,         %[readBackH]\n\t"
"               movb    $7,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM7H:    cmp     %%r13,          %%rax       \n\t"
"               je      CheckXMM8L                  \n\t"
"               cmp     %%r14,          %%rbx       \n\t"
"               je      CheckXMM8L                  \n\t"
"               cmp     %%r15,          %%rcx       \n\t"
"               je      CheckXMM8L                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm7,         %[readBackL]\n\t"
"               movhps  %%xmm7,         %[readBackH]\n\t"
"               movb    $7,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"

"CheckXMM8L:    movq    %%xmm8,         %%r10       \n\t"
"               movq    %%xmm8,         %%r11       \n\t"
"               movq    %%xmm8,         %%r12       \n\t"
"               movhlps %%xmm8,         %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"               movq    %%xmm15,        %%r13       \n\t"
"               movq    %%xmm15,        %%r14       \n\t"
"               movq    %%xmm15,        %%r15       \n\t"
"               cmp     %%r10,          %%rax       \n\t"
"               je      CheckXMM8H                  \n\t"
"               cmp     %%r11,          %%rbx       \n\t"
"               je      CheckXMM8H                  \n\t"
"               cmp     %%r12,          %%rcx       \n\t"
```

```
"               je      CheckXMM8H                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm8,         %[readBackL]\n\t"
"               movhps  %%xmm8,         %[readBackH]\n\t"
"               movb    $8,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM8H:    cmp     %%r13,          %%rax       \n\t"
"               je      CheckXMM9L                  \n\t"
"               cmp     %%r14,          %%rbx       \n\t"
"               je      CheckXMM9L                  \n\t"
"               cmp     %%r15,          %%rcx       \n\t"
"               je      CheckXMM9L                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm8,         %[readBackL]\n\t"
"               movhps  %%xmm8,         %[readBackH]\n\t"
"               movb    $8,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"

"CheckXMM9L:    movq    %%xmm9,         %%r10       \n\t"
"               movq    %%xmm9,         %%r11       \n\t"
"               movq    %%xmm9,         %%r12       \n\t"
"               movhlps %%xmm9,         %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"               movq    %%xmm15,        %%r13       \n\t"
"               movq    %%xmm15,        %%r14       \n\t"
"               movq    %%xmm15,        %%r15       \n\t"
"               cmp     %%r10,          %%rax       \n\t"
"               je      CheckXMM9H                  \n\t"
"               cmp     %%r11,          %%rbx       \n\t"
"               je      CheckXMM9H                  \n\t"
"               cmp     %%r12,          %%rcx       \n\t"
"               je      CheckXMM9H                  \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm9,         %[readBackL]\n\t"
"               movhps  %%xmm9,         %[readBackH]\n\t"
"               movb    $9,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM9H:    cmp     %%r13,          %%rax       \n\t"
"               je      CheckXMM10L                 \n\t"
```

```
"               cmp     %%r14,          %%rbx       \n\t"
"               je      CheckXMM10L                 \n\t"
"               cmp     %%r15,          %%rcx       \n\t"
"               je      CheckXMM10L                 \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm9,         %[readBackL]\n\t"
"               movhps  %%xmm9,         %[readBackH]\n\t"
"               movb    $9,             %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"

"CheckXMM10L:   movq    %%xmm10,        %%r10       \n\t"
"               movq    %%xmm10,        %%r11       \n\t"
"               movq    %%xmm10,        %%r12       \n\t"
"               movhlps %%xmm10,        %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"               movq    %%xmm15,        %%r13       \n\t"
"               movq    %%xmm15,        %%r14       \n\t"
"               movq    %%xmm15,        %%r15       \n\t"
"               cmp     %%r10,          %%rax       \n\t"
"               je      CheckXMM10H                 \n\t"
"               cmp     %%r11,          %%rbx       \n\t"
"               je      CheckXMM10H                 \n\t"
"               cmp     %%r12,          %%rcx       \n\t"
"               je      CheckXMM10H                 \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm10,        %[readBackL]\n\t"
"               movhps  %%xmm10,        %[readBackH]\n\t"
"               movb    $10,            %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM10H:   cmp     %%r13,          %%rax       \n\t"
"               je      CheckXMM11L                 \n\t"
"               cmp     %%r14,          %%rbx       \n\t"
"               je      CheckXMM11L                 \n\t"
"               cmp     %%r15,          %%rcx       \n\t"
"               je      CheckXMM11L                 \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm10,        %[readBackL]\n\t"
"               movhps  %%xmm10,        %[readBackH]\n\t"
"               movb    $10,            %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
```

Y

```
"              jmp     End                         \n\t"

"CheckXMM11L:  movq    %%xmm11,        %%r10       \n\t"
"              movq    %%xmm11,        %%r11       \n\t"
"              movq    %%xmm11,        %%r12       \n\t"
"              movhlps %%xmm11,        %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"              movq    %%xmm15,        %%r13       \n\t"
"              movq    %%xmm15,        %%r14       \n\t"
"              movq    %%xmm15,        %%r15       \n\t"
"              cmp     %%r10,          %%rax       \n\t"
"              je      CheckXMM11H                 \n\t"
"              cmp     %%r11,          %%rbx       \n\t"
"              je      CheckXMM11H                 \n\t"
"              cmp     %%r12,          %%rcx       \n\t"
"              je      CheckXMM11H                 \n\t"
"              movq    %%r10,          %[err1L]    \n\t"
"              movq    %%r11,          %[err2L]    \n\t"
"              movq    %%r12,          %[err3L]    \n\t"
"              movq    %%r13,          %[err1H]    \n\t"
"              movq    %%r14,          %[err2H]    \n\t"
"              movq    %%r15,          %[err3H]    \n\t"
"              movlps  %%xmm11,        %[readBackL]\n\t"
"              movhps  %%xmm11,        %[readBackH]\n\t"
"              movb    $11,            %[position] \n\t"
"              movb    $0xf0,          %[errFlag]  \n\t"
"              jmp     End                         \n\t"
"CheckXMM11H:  cmp     %%r13,          %%rax       \n\t"
"              je      CheckXMM12L                 \n\t"
"              cmp     %%r14,          %%rbx       \n\t"
"              je      CheckXMM12L                 \n\t"
"              cmp     %%r15,          %%rcx       \n\t"
"              je      CheckXMM12L                 \n\t"
"              movq    %%r10,          %[err1L]    \n\t"
"              movq    %%r11,          %[err2L]    \n\t"
"              movq    %%r12,          %[err3L]    \n\t"
"              movq    %%r13,          %[err1H]    \n\t"
"              movq    %%r14,          %[err2H]    \n\t"
"              movq    %%r15,          %[err3H]    \n\t"
"              movlps  %%xmm11,        %[readBackL]\n\t"
"              movhps  %%xmm11,        %[readBackH]\n\t"
"              movb    $11,            %[position] \n\t"
"              movb    $0xf0,          %[errFlag]  \n\t"
"              jmp     End                         \n\t"

"CheckXMM12L:  movq    %%xmm12,        %%r10       \n\t"
"              movq    %%xmm12,        %%r11       \n\t"
"              movq    %%xmm12,        %%r12       \n\t"
"              movhlps %%xmm12,        %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"              movq    %%xmm15,        %%r13       \n\t"
"              movq    %%xmm15,        %%r14       \n\t"
"              movq    %%xmm15,        %%r15       \n\t"
"              cmp     %%r10,          %%rax       \n\t"
"              je      CheckXMM12H                 \n\t"
"              cmp     %%r11,          %%rbx       \n\t"
"              je      CheckXMM12H                 \n\t"
```

Z

```
"               cmp     %%r12,          %%rcx       \n\t"
"               je      CheckXMM12H                 \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm12,        %[readBackL]\n\t"
"               movhps  %%xmm12,        %[readBackH]\n\t"
"               movb    $12,            %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM12H:   cmp     %%r13,          %%rax       \n\t"
"               je      CheckXMM13L                 \n\t"
"               cmp     %%r14,          %%rbx       \n\t"
"               je      CheckXMM13L                 \n\t"
"               cmp     %%r15,          %%rcx       \n\t"
"               je      CheckXMM13L                 \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm12,        %[readBackL]\n\t"
"               movhps  %%xmm12,        %[readBackH]\n\t"
"               movb    $12,            %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"

"CheckXMM13L:   movq    %%xmm13,        %%r10       \n\t"
"               movq    %%xmm13,        %%r11       \n\t"
"               movq    %%xmm13,        %%r12       \n\t"
"               movhlps %%xmm13,        %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"               movq    %%xmm15,        %%r13       \n\t"
"               movq    %%xmm15,        %%r14       \n\t"
"               movq    %%xmm15,        %%r15       \n\t"
"               cmp     %%r10,          %%rax       \n\t"
"               je      CheckXMM13H                 \n\t"
"               cmp     %%r11,          %%rbx       \n\t"
"               je      CheckXMM13H                 \n\t"
"               cmp     %%r12,          %%rcx       \n\t"
"               je      CheckXMM13H                 \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm13,        %[readBackL]\n\t"
"               movhps  %%xmm13,        %[readBackH]\n\t"
"               movb    $13,            %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM13H:   cmp     %%r13,          %%rax       \n\t"
```

```
"               je      CheckXMM14L                 \n\t"
"               cmp     %%r14,          %%rbx       \n\t"
"               je      CheckXMM14L                 \n\t"
"               cmp     %%r15,          %%rcx       \n\t"
"               je      CheckXMM14L                 \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm13,        %[readBackL]\n\t"
"               movhps  %%xmm13,        %[readBackH]\n\t"
"               movb    $13,            %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"

"CheckXMM14L:   movq    %%xmm14,        %%r10       \n\t"
"               movq    %%xmm14,        %%r11       \n\t"
"               movq    %%xmm14,        %%r12       \n\t"
"               movhlps %%xmm14,        %%xmm15     \n\t"
/*Only way to copy from XMM to GPR is through lower quadword*/
"               movq    %%xmm15,        %%r13       \n\t"
"               movq    %%xmm15,        %%r14       \n\t"
"               movq    %%xmm15,        %%r15       \n\t"
"               cmp     %%r10,          %%rax       \n\t"
"               je      CheckXMM14H                 \n\t"
"               cmp     %%r11,          %%rbx       \n\t"
"               je      CheckXMM14H                 \n\t"
"               cmp     %%r12,          %%rcx       \n\t"
"               je      CheckXMM14H                 \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm14,        %[readBackL]\n\t"
"               movhps  %%xmm14,        %[readBackH]\n\t"
"               movb    $14,            %[position] \n\t"
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"
"CheckXMM14H:   cmp     %%r13,          %%rax       \n\t"
"               je      Loop                        \n\t"
"               cmp     %%r14,          %%rbx       \n\t"
"               je      Loop                        \n\t"
"               cmp     %%r15,          %%rcx       \n\t"
"               je      Loop                        \n\t"
"               movq    %%r10,          %[err1L]    \n\t"
"               movq    %%r11,          %[err2L]    \n\t"
"               movq    %%r12,          %[err3L]    \n\t"
"               movq    %%r13,          %[err1H]    \n\t"
"               movq    %%r14,          %[err2H]    \n\t"
"               movq    %%r15,          %[err3H]    \n\t"
"               movlps  %%xmm14,        %[readBackL]\n\t"
"               movhps  %%xmm14,        %[readBackH]\n\t"
"               movb    $14,            %[position] \n\t"
```

```
"               movb    $0xf0,          %[errFlag]  \n\t"
"               jmp     End                         \n\t"

"Loop:          dec         %%rdx                   \n\t"
"               cmpq        $0,             %%rdx   \n\t"
"               jne         CheckXMM0L              \n\t"
"End:           nop                                 \n\t"


: [err1H] "=m" (error1H),
  [err1L] "=m" (error1L),
  [err2H] "=m" (error2H),
  [err2L] "=m" (error2L),
  [err3H] "=m" (error3H),
  [err3L] "=m" (error3L),
  [readBackH] "=m" (xmmValH),
  [readBackL] "=m" (xmmValL),
  [errFlag] "=m" (errorFlag),
  [position] "=m" (location)

: [toWrite] "m" (inputVal),
  [numLoops] "m" (loopCount)


:"%rax","%rbx","%rcx","%rdx","%r10","%r11","%r12","%r13","%r14",
 "%r15","%xmm0","%xmm1","%xmm2","%xmm3","%xmm4","%xmm5","%xmm6",
 "%xmm7","%xmm8","%xmm9","%xmm10","%xmm11","%xmm12","%xmm13",
 "%xmm14","%xmm15");

  if (errorFlag == 0xf0)
  {
      /*First we prompt the user*/
      printf("XMM%d: %llx %llx\n",location, xmmValH, xmmValL);
      printf("Error 1: %llx %llx\n",error1H, error1L);
      printf("Error 2: %llx %llx\n",error2H, error2L);
      printf("Error 3: %llx %llx\n",error3H, error3L);
      /*Now we write to file*/
      FILE * myFilePointer = fopen("XMM_Errors.csv", "a");
     fprintf(myFilePointer, "%d, %llx, %llx, %llx,
    %llx,%llx,%llx,%llx,%llx\n", location, xmmValH, xmmValL,
     error1H, error1L, error2H, error2L, error3H, error3L);
      fclose(myFilePointer);
      return 1;
  }
  else
  {
     printf("No error detected");
     return 0;
  }
}
```

CC

```c
int main()
{
    int numTestRuns;
    float energy,flux;

    printf("Enter Test Energy in MeV\n");
    scanf("%f", &energy);
    printf("Enter test flux\n");
    scanf("%f", &flux);
    printf("Enter the number of tests to run (for loop counter)\n");
    scanf("%d", &numTestRuns);

    /*We beginby creating the files that will be used to log errors as well
    as test parameters*/
    FILE * fp = fopen("XMM_Errors.csv", "a");
    fprintf(fp, "\n");
    fprintf(fp, "New test run\n");
    fprintf(fp, "============\n");
    fprintf(fp,"Energy: %f, Flux: %f\n", energy, flux);
    fprintf(fp, "REG Locaton, xmmH, xmmL, Error1H, Error1L, Error2H, Error2L,
    Error3H, Error3L\n");
    fclose(fp);

    printf("Log file Appended. Beginning test\n");

    /*Now we run the actual test*/
    int errCount = 0;
    for (int i = 0; i < numTestRuns; i++)
    {
        printf("Run %d: ",i);
        int errType = XMMTest();
        if (errType == 1)
        {
            errCount++;
            printf("Error Count: %d\n",errCount);
        }
        else
            printf("\n");
    }
    printf("\n\nGPR Test Program Completed\n");
    return 0;
}
```

DD

# Appendix 5: Math Test Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>


int main()
{
    long double result = 0, correctResult = 0.7126148039773066;
    float energy, flux;
    int numLoops;

    printf("Enter Energy in MeV\n");
    scanf("%f", &energy);
    printf("Enter flux\n");
    scanf("%f", &flux);
    printf("Enter the number of calculations to perform\n");
    scanf("%d", &numLoops);

    FILE * fp = fopen("Cache Test.csv", "a");
    fprintf(fp, "Energy:\t %f\n", energy);
    fprintf(fp, "Flux:\t %f\n", flux);
    fclose(fp);
    printf("File Appended\n");


    int errCount = 0;
    for (int i = 0; i < numLoops; i++)
    {
        result=cos(sin(sin(M_SQRT2*sin(cos(sin(cos(pow(M_E,M_PI)*16032001)))))));
        if (result == correctResult)
            printf("Run: %d\t Ok\n", i);
        else
        {
            errCount++;
            printf("Run: %d\t Mismatch\t Error Count: %d\n", i, errCount);

            FILE * tempFp = fopen("Cache Test.csv", "a");
            fprintf(tempFp,"Run No,%d,Error Count,%d\n", i, errCount);
            fclose(tempFp);
        }
    }
    return 0;
}
```

# Appendix 6: Cache Disable/Enable Kernel Module

```
#include <linux/init.h>
#include <linux/module.h>

static int switchOffCache_init(void)
{
   printk(KERN_ALERT "Cache is switching OFF\n");
   asm volatile("mov    %%cr0,                                          %%eax   \n\t"
                "add     $0b01100000000000000000000000000000,     %%eax   \n\t"
                "mov     %%eax,                                          %%cr0   \n\t"
                "wbinvd                                                          \n\t"
                          :
                          :
                          :"%eax");
   printk(KERN_ALERT "Cache has been switched OFF\n");
   return 0;
}


static void switchOffCache_exit(void)

{
   printk(KERN_ALERT "Cache is switching ON\n");
   asm volatile("mov    %%cr0,                                          %%eax   \n\t"
                 "sub    $0b01100000000000000000000000000000,     %%eax   \n\t"
                 "mov    %%eax,                                          %%cr0   \n\t"
                 "wbinvd                                                          \n\t"
                          :
                          :
                          :"%eax");
   printk(KERN_ALERT "Cache has been switched ON\n");
}

module_init(switchOffCache_init);
module_exit(switchOffCache_exit);
```

# Appendix 7: Makefile for Cache Disable/Enable Kernel Module

```
obj-m += CacheDisableEnable.o

KDIR = /usr/src/linux-headers-4.18.0-13-generic


all:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
	rm -rf *.o *.ko *.mod.* *.symvers *.order
```