UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

Total Ionizing Dose Mitigation by means of Reconfigurable FPGA Computing

Farouk Smith

Dissertation presented for the degree of Doctor of Philosophy in Engineering at the University of Stellenbosch

Supervisor: Prof. S. Mostert

December 2007

Declaration

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

SIGNATURE	DATE

Synopsis

There is increasing use of commercial components in space technology and it is important to recognize that the space radiation environment poses the risk of permanent malfunction due to radiation. Therefore, the integrated circuits used for spacecraft electronics must be resistant to radiation.

The effect of using the MOSFET device in a radiation environment is that the gate oxide becomes ionized by the dose it absorbs due to the radiation induced trapped charges in the gate-oxide. The trapped charges in the gate-oxide generate additional space charge fields at the oxide-substrate interface. After a sufficient dose, a large positive charge builds up, having the same effect as if a positive voltage was applied to the gate terminal. Therefore, the transistor source to drain current can no longer be controlled by the gate terminal and the device remains on permanently resulting in device failure.

There are four processes involved in the radiation response of MOS devices. First, the ionizing radiation acts with the gate oxide layer to produce electron-hole pairs. Some fraction of the electron-hole pairs recombine depending on the type of incident particle and the applied gate to substrate voltage, i.e. **the electric field**. The mobility of the electron is orders of magnitude larger than that of the holes in the gate oxide, and is swept away very quickly in the direction of the gate terminal. The time for the electrons to be swept away is on the order of 1ps. The holes that escape recombination remain near their point of origin. The number of these surviving holes determines the initial response of the device after a short pulse of radiation. **The cause of the first process, i.e. the presence of the electric field, is the main motivation for design method described in this dissertation.**

The second process is the slow transport of holes toward the oxide-silicon interface due to the presence of **the electric field**. When the holes reach the interface, process 3, they become captured in long term trapping sites and this is the main cause of the permanent threshold voltage shift in MOS devices. The fourth process is the buildup of interface states in the substrate near the interface

The main contribution of this dissertation is the development of the novel Switched Modular Redundancy (SMR) method for mitigating the effects of space radiation on satellite electronics. The overall idea of the SMR method is as follows: A charged particle is accelerated in the presence of an electric field. However, in a solid, electrons will move around randomly in the absence of an applied electric field. Therefore if one averages the movement over time there will be no overall motion of charge carriers in any particular direction. On applying an electric field charge carriers will on average move in a direction aligned with the electric field, with positive charge carriers such as holes moving in the direction of field, and negative charge carriers moving in the opposite direction. As is the case with process one and two above.

It is proposed in this dissertation that if we apply the flatband voltage (normaly a zero bias for the ideal NMOS transistor) to the gate terminal of a MOS transistor in the presence of ionizing radiation, i.e. no electric field across the gate oxide, both the free electrons and holes will on average remain near their point of origin, and therefore have a greater probability of recombination. Thus, the threshold voltage shift in MOS devices will be less severe for the gate terminal in an unbiased condition. The flatband conditions for the real MOS transistor is discussed in appendix E.

It was further proposed that by adding redundancy and applying a resting policy, one can significantly prolong the useful life of MOS components in space. The fact that the rate of the threshold voltage shift in MOS devices is dependant on the bias voltage applied to the gate terminal is a very important phenomenon that can be exploited, since we have direct control and access to the voltage applied to the gate terminal. If for example, two identical gates were under the influence of radiation and the gate voltage is alternated between the two, then the two gates should be able to withstand more total dose radiation than using only one gate. This redundancy could be used in a circuit to mitigate for total ionizing dose.

The SMR methodology would be to duplicate each gate in a circuit, then selectively only activating one gate at a time allowing the other to anneal during its off cycle. The SMR algorithm was code in the “C” language. In the proposed design methodology, the design engineer need not be concerned about radiation effects when describing the hardware implementation in a hardware description language. Instead, the design engineer makes use of conventional design techniques. When the design is complete, it is synthesized to obtain the gate level netlist in edif format. The edif netlist is converted to structural VHDL code during synthesis. The structural VHDL netlist is fed into the SMR “C” algorithm to obtain the identical redundant circuit components. The resultant file is also a structural VHDL netlist. The generated VHDL netlist or SMR circuit can then be mapped to a Field Programmable Gate Array (FPGA).

Spacecraft electronic designers increasingly demand high performance microprocessors and FPGAs, because of their high performance and flexibility. Because FPGAs are reprogrammable, they offer the additional benefits of allowing on-orbit design changes. Data can be sent after launch to correct errors or to improve system performance. System including FPGAs covers a wide range of space applications, and consequently, they are the object of this study in order to implement and test the SMR algorithm.

We apply the principles of reconfigurable computing to implement the Switched Modular Redundancy Algorithm in order to mitigate for Total Ionizing Dose (TID) effects in FPGA’s. It is shown by means of experimentation that this new design technique provides greatly improved TID tolerance for FPGAs.

This study was necessary in order to make the cost of satellite manufacturing as low as possible by making use of Commercial off-the-shelf (COTS) components. However, these COTS components are very susceptible to the hazards of the space environment.

One could also make use of Radiation Hard components for the purpose of satellite manufacturing, however, this will defeat the purpose of making the satellite manufacturing cost as low as possible as the cost of the radiation hard electronic

components are significantly higher than their commercial counterparts. Added to this is the undesirable fact that the radiation hard components are a few generations behind as far as speed and performance is concerned, thus providing even greater motivation for making use of Commercial components.

Radiation hardened components are obtained by making use of special processing methods in order to improve the components radiation tolerance. Modifying the process steps is one of the three ways to improve the radiation tolerance of an integrated circuit. The two other possibilities are to use special layout techniques or special circuit and system architectures.

Another method, in which to make Complementary Metal Oxide Silicon (CMOS) circuits tolerant to ionizing radiation is to distribute the workload among redundant modules (called Switched Modular Redundancy above) in the circuit. This new method will be described in detail in this thesis.

Opsomming

Daar is ‘n verhoogde gebruik van kommersieële komponente in die ruimte en dit is belangrik om die risiko van bestaling in die ruimte omgewing in erken agv die risiko van permanente beskadiging te wyte aan bestraling. Dit is vir hierdie rede, dat die geintegreerde stroombane wat gebruik word vir ruimte elektronika bestand teen bestraling moet wees.

Die effek van die gebruik van die MOSFET toerusting in ‘n bestralings omgewing is dat die hek oksied ge-ioniseer word deur die gevangde bestralings ge-induseerde gevange ladings in die hek-oksied. Die vaste lading in die hek-oksied produseer ’n addisioneele spannings veld by die oxide-substraat intervlak. Na 'n voldoende dosis, vorm ‘n groot positieve lading, en dit het dieselfde effek as 'n positief spanning wat oor die hek aangelê word. Om hierdie rede, kan die transistor die stroom tussen die drein en “source” nie meer afskakel nie en die transistor bly permanent aan geskakel wat die stroombaan laat faal.

Daar is vier prosesse betrokke in die bestralings effek op MOS komponente. Eerstens, die ioniseering bestraling se impak op die hek oksied laag produseer elektron-holte pare. ‘n Fraksie van die elektroon-holte pare herkombineer afhangend op die soort van partikel en die hek tot substraat spanning, agv byvoorbeeld die elektries veld op die hek. Die mobilisasie van die elektrone is veel groter as die van die holtes in die hek oksied, en die elektrone beweeg baie gou in die rigting van die hek terminaal. Die tyd vir die elektrone om te weg te beweeg is in die orde van 1ps. Die holtes wat herkombinasie ontsnap bly naby hul punt van oorsprong. Die aantal van die oorlewende holtes bepaal die aanvanklike effek van die komponent na 'n kort pols van bestraling. Die oorsaak van die eerste proses en die teenwoordigheid van die elektriese veld is die hoof motivering vir die ontwerp metode wat beskryf word in hierdie verhandeling.

Die tweede proses is die stadige vervoer van holtes na die oksied-silikon koppelvlak as gevolg van die teenwoordigheid van die elektriese veld. Wanneer die holtes die koppelvlak bereik, proses 3, word hulle gevang in lank termyn areas en die positiewe lading is die hoof oorsaak van die permanente drumpel spanning verskuiwing in MOS komponente. Die vierde proses is die opbou van koppelvlak toestande in die substraat naby die koppelvlak.

Die hoof bydrae van hierdie verhandeling is die ontwikkeling van die Skakel Modulêr Oortolligheid (SMR) metode om die effekte van die ruimte bestraling op sateliet elektronika te verminder. Die uitgangspunt van die SMR metode is as volg: 'n elektron word versnel in die teenwoordigheid van 'n elektriese veld. Maar in ’n vaste stof, sal die elektrone na willekeur rond beweeg in die afwesigheid van 'n aangelegde elektriese veld. Om hierdie rede, as ons die beweging gemiddelt oor tyd meet sal daar geen algehele beweging van lading draers in enige presiese rigting wees nie. Met die aanwending van 'n elektries veld sal lading draers op gemiddeld beweeg in 'n rigting gerig met die elektries veld, met positief lading draers soos holtes, beweeg in die rigting van die veld, en negatiewe ladind draers beweeg in die teenoorgestelde rigting.

Dit word voorgestel in hierdie verhandeling dat indien ons 'n nul spanning aan wend oor die hek terminaal van 'n MOS transistor in die teenwoordigheid van ioniseerende bestraling, deur byvoorbeeld geen elektriese veld oor die hek aan te lê nie, dan sal beide die vry elektrone en holtes gemiddeld naby hul punt van oorsprong bly, en om hierdie rede het hulle ‘n groter waarskynlikheid van herkombinasie. Dus, die drempel spanning aanpassing in MOS komponente sal kleiner wees vir die hek terminaal in 'n nul spannings kondisie.

Dit word verder voorgestel dat deur oortolligheid by te voeg en die aanwending van 'n rus beleid, dat die leeftyd van MOS komponente beduidend verleng kan word in die ruimte. Die feit dat die tempo van die drumpel spanning verandering in MOS komponente afhanklik is van die spanning wat aangewend word tot die hek terminal, is 'n baie belangrike verskynsel wat uitgebuit kan word. Met direkte beheer en toegang tot die

spanning wat aangelê word op die hek terminaal. Indien byvoorbeeld, twee identiese hekke onder die invloed van bestraling was, en die hek spanning word gewissel tussen die twee, dan behoort die twee hekke instaat te wees om meer bestand te wees teen totale dosis bestraling, in vergelyking wanneer net een hek gebruik word. Hierdie oortolligheid kan gebruik word in elektronika om die totale ioniseerings dosis te verminder.

Die SMR metode behels die dupliseering van elke hek in 'n elektroniese stroombaan. Daarna word selektief een hek op 'n tyd ge-aktifeer om toe te laat dat die ander herstel gedurende sy “af” kringloop. Die SMR algoritme is gekode in die “C” taal. In die voorgestelde ontwerp metode, hoef die ontwerp ingeneur nie bekommerd te wees omtrent bestraling effekte wanneer die hardeware implementasie in 'n hardeware beskrywing taal beskryf word nie. In plaas daarvan, maak die ontwerp ingeneur gebruik van konvensionele ontwerp tegnieke. Wanneer die ontwerp voltooi is, word dit gesintetiseer om die hek vlak netlist in edif formaat te verkry. Die edif netlist word dan omgekeer na strukturele VHDL kode gedurende sintese. Die strukturele VHDL netlist word dan gevoed binne-in die SMR “C” algoritme om die identiese oorbodige stroombaan komponente te verkry. Die resultaat is ook 'n strukturele VHDL netlist. Die ontwikkelde VHDL netlist of SMR stroombaan kan dan oorgedra word tot 'n veld programmeerbare hek struktuur (FPGA).

Ruimtetuig elektroniese ontwerpers benodig meer en meer hoë werkverrigting mikroprosesseerders en FPGAs weens hul hoë werkverrigting en buigsaamheid. Omdat FPGAs herprogrameerbaar is, bied hulle die addisioneel voordele om dit moontlik te maak om ruimte wentelbaan ontwerp veranderings te doen. Data kan gestuur word na lansering om foute te korigeer of om stelsel werkverrigting te verbeter. Stelsels wat FPGAs insluit word gebruik in 'n wye reeks van die ruimte applikasies, en gevolglik, is FPGAs die objek van hierdie studie ten einde die SMR algoritme te implementeer en te toets.

Ons gebruik die beginsels van her-konfigureerbare logika om die SMR algoritme te implementeer ten einde ’n groter toleransie van elektronika te kry in die teenwoordigheid

van totaal ioniseering dosis (TID) effekte in FPGAs. Daar is eksperimenteel aangetoon dat hierdie nuwe ontwerp tegniek voorsien ’n groot verbetering in TID toleransie vir FPGAs.

Hierdie studie was nodig ten einde die koste van sateliet vervaardiging so laag as moontlik te maak deur gebruik te maak van kommersieële “af-die-rak” (COTS) komponente. Die problem is dat hierdie COTS komponente is nooit ontwerp om bestralings bestand te wees nie en hulle is baie vatbaar vir bestraling in ’n ruimte omgewing.

Ons kan ook gebruik maak van bestralings verharde komponente vir die doel van sateliet vervaardiging,   Maar dit sal die doel om sateliet vervaardiging koste so laag as moontlik te hou ondermyn, omdat die koste van die bestralings verharde elektroniese komponente beduidend hoër is as hul kommersieële alternatiewe. Voorts is bestralings verharde komponent gewoonlik ’n generasie of twee agter sover as spoed en werkverrigting aangaan en daarom is daar dus is ’n groter motivering vir die gebruik van kommersieële komponente.

Bestralings verharde komponente word verkry deur gebruik te maak van spesiaal prosessering metodes ten einde die komponente bestralings toleransie te verbeter. Modifisering van die proses stappe is een van die drie maniere om die bestralings toleransie van 'n geintegreer stroombaan te verbeter. Die twee ander moontlikhede is om gebruik te maak van spesiale uitleg tegnieke of spesiale stroombaan en stelsel argitekture.

'n Ander metode, om aanvullende metaal oksied silicium (CMOS) stroombane meer bestand te maak teen ioniseering uitstraling is om die werk-las tussen oorbodig modules (genoem SMR) in die stroombaan te versprei. Hierdie nuwe metode word in detail in hierdie tesis beskryf.

To my father and in loving memory of my mother.

Acknowledgements

I wish to thank the following people and organizations for their support in this research.

- Telkom, for their generous support during the duration of this project.
- A special thank-you to my supervisor Prof. Sias Mostert, for your advice and helping me to stay motivated. Your efforts and time are well appreciated.
- Mr. Arno Barnard, for his advice and assistance with setting up the radiation test bed.
- Mr Johan Arendse for the quick turn around time of the PCB soldering.
- Mr Quintis Brandt, also for his quick response in ordering the electronic components
- Praveen Samudrala for his assistance and making his STMR code available
- My other friends and fellow students in the ESL Lab group for making the ESL lab a friendly center for learning
- My parents, for always believing in me and for their constant encouragement throughout my academic career.
- To my wife and son, for their understanding, love and support during the four years that I have spent on this project. And to my beautiful new born daughter, for making me smile the past few weeks.

Table of Contents

List of Figures

List of Tables

Glossary of Acronyms

ARC	Agriculture Research Council
ASIC	Application Specific Integrated Circuit
CMOS	Complimentary metal-oxide semiconductor field effect transistor
CTRW	continuous-time random walk
DHP	Dynamically loadable Hardware Plugin
DUT	device under test
EAB	embedded array blocks
FLEX	flexible logic element matrix
FPGA	Field Programmable Gate Arrays
GEO	Geostationary Orbits
HEO	Highly Elliptical Orbits
IC	integrated circuit
Icc	Power supply current
ICR	In-circuit reconfigurability
ICT	Information Communications and Technology
IOE	Input-output elements
LAB	logic array block
LE	logic element
LEO	Low Earth Orbit
LET	linear energy transfer
LETth	LET threshold
LUT	look-up table
MOS	Metal Oxide Silicon
MOSFET MOS	metal-oxide semiconductor field effect transistor
NMOS	n-channel MOS

ONO	oxide-nitrideoxide
PROM	programmable read only memory
RHBD	Radiation Hardened By Design
SAA	South Atlantic Anomaly
SATNAC	South African Telecommunication and Network Application Conference
SEB	Single Event Burnout
SEE	Single Event Effects
SEFI	single event functional interrupt
SEGR	Single Event Gate Rupture
SEL	Single Event Latch-up
SES	Single Event Snapback
SET	single event transient
SEU	Single Event Upsets
SMR	Switched Modular Redundancy
SOI	Silicon-On-Insulator
SPI	Serial Port Interface
SRAM	static random access memory
TID	Total Ionizing Dose
TMR	Triple Module Redundancy
VFB	flatband voltage
VT	threshold voltage

Chapter 1

Introduction

1.1 Introduction

The study of radiation effects on semiconductor devices started about forty years ago, when the first satellites experienced serious problems due to the detrimental effect on their electronic circuits as a result of space radiation. Since then, there has been an increasing interest in the study of circuits which can work in a radiation environment, driven by all the possible applications of these kinds of circuits, such as advanced weaponry, instrumentation for nuclear power plants, high-energy physics experiments and, last but not least space missions and satellites [ANEL00].

In the 1970s the view was widely held that designing radiation-hardened spacecraft and systems would become “redundant” with the development of radiation-hardened electronic components. Unfortunately that is not the reality of today. In fact, reducing radiation effects on spacecraft systems to manageable levels is more complex than ever. There are currently no completely radiation hardened devices in existence. The need for a system with high levels of performance has exceeded the capabilities of available radiation hardened components and technology. At the same time, the demand for electronics goods in commercial markets has greatly decreased the manufacturer’s interest in developing radiation hardened components, driving up the cost of radiation hardened parts and making them increasingly unavailable [BAR97].

The design of digital circuits for space application needs first to consider the space radiation environment and the satellite orbits. It is essential to study the radiation effects in digital circuits and the ways to qualify these digital circuits for space applications. In space, integrated circuits are subjected to hostile environments composed of a variety of particles including photons, charged particles, neutrons and others. The charged particles can hit the ICs resulting in non-destructive or destructive effects according to the charge intensity and to the hit location.

The analysis of radiation effects on integrated circuits and the development of mitigation techniques are strongly associated to the target device architecture. For each different circuit, there is a most suitable mitigation solution to be applied [LIMA02]. **Consequently, in order to suggest a mitigation solution, first it is necessary to investigate the architecture**. In the past years the integrated circuit industry has designed complex architectures in order to improve performance and logic density and to reduce cost. Examples of this development are Application Specific Integrated Circuits (ASICs) such as microprocessors and the high-density Field Programmable Gate Arrays (FPGAs). Microprocessors have made a dramatic impact in the way systems are designed, providing a high information process in a single chip. FPGAs have also made a major improvement in system designs by adding the reconfigurability feature. More complex structures are constantly being added in the FPGA architecture, supported by substantial increase in logic density and performance in the last few years. **Both architectures contain million of transistors located in many distinct logic blocks, making the modeling, test and understanding of such complex architecture very difficult.**

Due to the constant advances in technology in the last few years, the gap between FPGAs and ASICs in terms of performance has been reduced to a negligible level, for the majority of applications. Consequently, next generation architectures do not claim to reduce that gap anymore, but to merge microprocessors and reconfigurability features in the same device in order to improve performance and flexibility. However, it is not defined yet which architecture will be embedded in the other one. While some traditional ASIC companies are adding embedded programmable logic cores in their devices, FPGA companies are adding ASIC IP cores in the FPGA programmable matrix.

In the paper "(When) Will FPGA kill ASICs?" [RUTE01], many ASIC and FPGA companies discussed next generations trends. Which solution is going to be more attractive, the ASIC with embedded programmable logic or FPGAs with embedded soft and hard IP cores? None has given a final answer, but all have agreed that different markets need different solutions, and there is always a price to pay when performance, high density and flexibility are all required together. Both choices force changes in the software design flow. However, it seems that fewer changes must be done in the case of FPGA with IP cores and more advantages can be easily achieved with that.

What is important to note for the purposes of this study is that the space and military market, just as the commercial market, requires high performance devices with low power, low cost, high flexibility and time to market. Added to that, the space and military applications also request an extra feature: the radiation tolerance.

Spacecraft electronic designers increasingly demand high performance microprocessors and FPGAs, because of their high performance and flexibility. Because FPGAs are reprogrammable, they offer the additional benefits of allowing on-orbit design changes. Data can be sent after launch to correct errors or to improve system performance. System including FPGAs covers a wide range of space applications, and consequently, they are the object of this study.

Fundamentally the radiation effects of FPGA's are not different from any other CMOS-based digital IC [WANG04]. Each FPGA is unique in its architecture, and each has its unique response to radiation. However, the challenge is to correlate the radiation induced response to the basic mechanisms of the Metal Oxide Silicon (MOS) transistor radiation response.

The main contribution of this dissertation is the development of the novel Switched Modular Redundancy (SMR) method for mitigating the effects of space radiation on satellite electronics. Once the SMR principle was developed for the MOS transistor, it was used to provide TID mitigation for the FPGA. This is possible, because at its lowest level, the FPGA is constructed from MOS transistors.

For implementation purposes the principles of reconfigurable computing was applied to implement the Modular Redundancy Algorithm in order to mitigate for Total Ionizing Dose (TID) effects in FPGA's, and thereby increasing the FPGA's functional lifetime and performance in the presence of ionizing radiation.

1.2 Outline of this Dissertation

This introduction forms the first chapter of this thesis. The remainder of the document is organized as follows:

Chapter 2: Background and Related Work for TID effects. Chapter 2 presents an overall view of the important issues regarding TID effects on electronic components as well as related research. The space radiation environment is first presented where various important factors are identified that is important to consider as far as satellite electronics are concerned. The chapter then introduces total ionizing dose effects as far as electronics are concerned followed by an in depth discussion of the radiation response of the MOS transistor. Various approaches used toward radiation hardened integrated circuits are then discussed followed by a discussion of the radiation effects observed in FPGA's which concludes this chapter.

Chapter 3: Switched Modular Redundancy. In chapter 3 we present the proposed Switched Modular Redundancy (SMR) method. The effect that the gate bias or electric field across the MOS capacitor, has on the radiation response of the MOS oxide, is a very import matter which will be considered more closely in this chapter and forms the basis for the novel Total Ionizing Dose mitigation technique, called Switched Modular Redundancy. The chapter includes an in depth discussion on how the SMR methodology can be used in FPGA's by means of reconfigurable computing in order to mitigate for TID effects.

Chapter 4: Experimental Setup and Methodology. The response of MOS devices to radiation is very variable and it is thus impossible at present to use theory alone to predict the device response. Therefore, testing integrated circuits in a severe radiation environment in advance to their use in operational systems is very important and it will help to reduce the probability of failures in future space applications. Fault injection is normally used to perform Single Event Upset (SEU) testing on electronic circuits, while actual ground tests are performed in order to test the ionizing dose performance of electronic circuits. In this chapter we describe the experimental setup for the ground testing as well as the radiation facility. We also describe the architecture of the FPGA used in the testing as well as the layout of the radiation PCB test boards.

Chapter 5: Experimental Results. This chapter presents the results of the radiation testing response of the FPGA's with various configurations. The chapter starts of with the experimental results of the resting policy applied to FPGA's in a radiation environment followed by the results of the effect of the clock and configuration memory on the FPGA radiation response. The chapter concludes with the results of testing the SMR algorithm by means of reconfigurable FPGA computing in a radiation environment.

Chapter 6: Conclusions and Recommendations. This chapter presents a set of conclusions that were drawn from this study as well as recommendations that may be used for future work regarding radiation effects on FPGA's.

Appendices: Sets of appendices are presented to provide further background information.

1.3 Published Work

Original work that we present in this thesis has been published at various conferences including some IEEE conferences and IEEE journals. These publications are:

Local Conference Publications

1) **F. Smith, S. Mostert, "Low Cost Satellite Communication – Space Weather and Commercial Electronic Components", SATNAC, September 2004.**

2) **F. Smith, S. Mostert, "Low Cost Satellite Communication – Designing Integrated Circuits to withstand Space Radiation", SATNAC, September 2005.**
3) **F. Smith, S. Mostert, "Switched Modular Redundancy for TID Mitigation in Digital Circuits", SATNAC, September 2006.**
4) **F. Smith, S. Mostert, "Reconfigurable computing for TID Mitigation in Digital Satellite Circuits", Accepted for publication at SATNAC 2007.**

The South African Telecommunication and Network Application Conference (SATNAC) is the event for Industry, Academia and Operators to publish on matters concerning progress achieved in applied research in the Information Communications and Technology (ICT) sector.

International IEEE Conference Publications

5) **F. Smith, S. Mostert, "Reconfigurable FPGA Computing to mitigate for Total Ionizing Dose Effects", 2007 IEEE Aerospace Conference, Big Sky, Montana, USA, March 3 - 10, 2007.**

The 2008 Aero Space conference will be the twenty-ninth in a series of annual weeklong winter conferences designed for aerospace experts, academia, military personnel, and industry leaders in a stimulating and thought-provoking environment. The Conference promotes interdisciplinary understanding of aerospace systems, their underlying science and technology, and their applications to government and commercial endeavors. Papers are peer reviewed and typically provide the technical depth characteristic of journal articles.

6) **F. Smith, S. Mostert, "Total Ionizing Dose Mitigation by means of Reconfigurable Computing", Accepted for publication at IEEE RADECS 2007 conference, September 10-14, 2007, Deauville, France.**

9th European Conference Radiation and Its Effects on Components and Systems (RADECS). Organized by the Commissariat à l'Energie Atomique, September 10-14, 2007 - Deauville, France. Since 1989, the goal of the European RADECS Conferences and Workshops has been to serve the various international industrial and research communities interested in radiation effects in electronics and optoelectronics. The 2007 theme is “Radiation Effects, from Materials to Systems: a Multi-Scale Approach”.

International IEEE Journal Publications

7) **F. Smith, S. Mostert, “Total Ionizing Dose Mitigation by means of Reconfigurable FPGA Computing” IEEE Transactions on Nuclear Science, Vol. 54, No. 4, pp. 1343 – 1349, August 2007.**

IEEE Transactions on Nuclear Science (TNS) was the number-four journal in nuclear science and technology in 2002, according to the annual Journal Citation Report (2002 edition) published by the Institute for Scientific Information. The TNS journal devote itself to publication or other dissemination of original contributions to the theory, experiments, educational methods and applications of these fields, and to the development of standards. Areas of technical activity shall include but not be limited to the following: Nuclear science and engineering including: instrumentation for research; detection and measurement of radiation; nuclear biomedical applications; radiation monitoring and safety equipment; particle accelerators; nuclear instrumentation development for reactor systems; effects of radiation on materials, components, and systems; and applications of radiation and nuclear energy to other than utility power generation.

Chapter 2

Background and Related Work for TID effects

The interaction of radiation with matter is a very broad and complex topic. In this chapter we try to analyze the problem with the aim of explaining the more important aspects, which are essential for a physical comprehension of the degradation observed in electronic devices and circuits under radiation. The manner in which radiation interacts with solid materials depends on the type, kinetic energy, mass and charge of the incident particle and the mass, atomic number and density of the target material. The effects can be classified in the following three ways: 1) Total dose as a result of ionization damage, 2) Bulk effects as a result of displacement damage and 3) Single Event Effects (SEE) as a result of an energetic particle strike [SMIT94, LABE96]. In this study we concentrate on the design techniques to mitigate for TID effects, and therefore only concentrate the effects due to ionizing radiation. SEE effects and Displacement damage is merely mentioned for completeness.

This chapter begins with a brief overview of the space radiation environment.

2.1 The Space Radiation Environment

The space radiation environment can have serious effects on satellite electronics [STAS88]. Satellites in low Earth orbits are known to have detrimental radiation effects when they are over the South Atlantic Anomaly due to the decrease in magnetic field strength.

Space weather has been blamed for satellite failures that have cost the commercial satellite industry millions. Solar conditions drive the space weather environment near Earth. Explosions on the Sun send giga-watts of energy hurtling towards Earth via the solar wind, causing space storms around Earth.

The solar wind is a gusty ever-present stream of ions and electrons emitted by the Sun's hot atmosphere. When solar wind conditions change sharply, for example during huge solar events called coronal mass ejections, the Earth's magnetic environment is affected and cause large fluxes of 'killer' electrons that encircle the Earth with a potentially deadly effect on satellites. Also, satellite surfaces can charge to thousands of volts, ground-based compass needles can shift by 10 degrees, communications can be affected and power distribution systems can have problems [ANTA00].

Space is part of everybody's daily lives. Satellites transmit television, telephone and other information around the world, and watch over our changing environment. As it tracks destructive hurricanes on Earth, a satellite might itself be damaged by another kind of storm - one that occurs in space. These magnetic storms disrupt radio communications and have caused electricity blackouts.

Since the start of the space age we have known that space is far from being empty. Near-Earth, high-energy radiation, which is hazardous to spacecraft and astronauts, is trapped in the Van Allen belts. What we don't know is precisely how and why the particles are accelerated to dangerous million-volt energies, it is believed that changes in the solar wind are to blame.

It is currently impossible and uneconomic to design a satellite that is entirely immune to variations in the space environment. It is vital for the satellite operator to be aware of these conditions. There will be future satellite problems, and even failures. But the gains and profits to be had are too great and have to be investigated in order to ensure the safeguarding of the satellite operator’s investment.

It is important to define the radiation environment of a satellite orbit in order to quantify the amount of radiation exposure the satellite will experience during its design lifetime. This provides for an element of trade-off for orbit and component selection during the design phase of the satellite. When a satellite mission is defined, the knowledge of the radiation environment will allow an estimation of the suitability of a range of potential components for successful mission completion [LABE98].

A number of areas exist near earth that have unique radiation characteristics important to satellite electronics. The sections that follow will examine these regions and the type of radiation found in each. A Satellite orbit may intersect more than one of these regions.

2.1.1 Units

A radiation environment is defined when one knows the kind of particles, their energies and their fluxes [HOLM02]. Further, if we are interested in the interaction with matter a very common way of describing the radiation is to indicate the energy absorbed by a specimen per mass unit. The most common particles are electrons, protons, neutrons, heavy ions and photons ranging from UV to gamma energies. All these particles except neutrons produce ionization effects in materials, so for them it is easy to find the energy released passing through the matter (for instance with a photodiode) and we can use correlations between the fluxes and the doses absorbed. On the other side neutron interactions are just nuclear-like, and it is more common to account their effect by using the flux (or the time-integrated flux, called fluence). The flux is the number of particles passing during 1 s through a 1 cm^2 area [$cm^{-2}s^{-1}$]; integrating over the time we get the fluence, which is [cm^{-2}]. The energy deposited is measured in rad, being 1rad = 10^{-2} Js^{-1}. The advantage in using the rad as radiation unit is evident if we consider that for instance in 1 cm^3 of Silicon 1 rad corresponds roughly to the generation of $4x10^{13}$ electron-hole pairs [HOLM02]. This way one gets an immediate idea of the damage induced in Silicon-based devices, in which the number of majority carriers is about 10^{16} carriers/cm^3.

2.1.2 The Radiation Belts

The Earth's radiation belts, also know as the Van Allen belts, consist mainly of electrons of energy up to a few MeV and protons up to several hundred MeV which are trapped in the Earth's magnetic field. The field is basically that of a magnetic dipole and in those regions where the field lines are 'closed', charged particles become trapped in the magnetosphere [ADAM02]. The trapped electrons are classified into inner and outer zones, Fig 2.1. The inner zone extends to about 2.4 Earth radii and the outer zone from 2.8 to 12 Earth radii. The gap between the two zones is referred to as the 'slot'. The electron energies extend up to 7 MeV with the most energetic electrons being in the outer zones. The proton environment varies with distance from earth inversely and peaks at approximately 3.8 earth radii [STAS88]. Proton energies can reach values over 400 MeV in this region. In 1998, there were a series of large, solar disturbances that caused a new radiation belt to form in the so-called “slot region” between the inner and outer van Allen belts. The new belt eventually disappeared once the solar activity subsided. The earth’s magnetic field is not geographically symmetrical; local distortions are caused by an offset and tilt of the magnetic axis and by geological influences; in the Southern Hemisphere, one important distortion is known as the ‘South Atlantic Anomaly' (SAA). In this region, field lines containing significant energetic-particle fluxes, approach the earth’s surface giving flux enhancement at low altitudes in the region of South America. The SAA is responsible for most of the radiation received by satellites in the Low Earth Orbit (LEO) altitude.

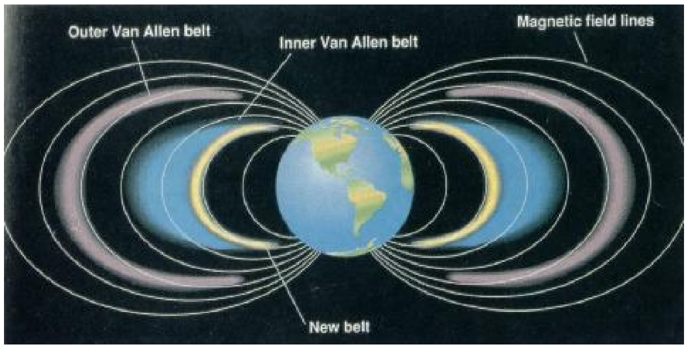


Fig 2.1 The Van Allen radiation belts [MONR03].

2.1.3 Cosmic Rays

Cosmic Rays comprise 85% protons, 14 % alpha particles, and 1% heavier ions covering the full range of elements, some of the more abundant being, for example, carbon and iron nuclei. They are partly kept out by the earth’s magnetic field and have easier access at the poles compared with the equator. From the point of view of space systems it is particles in the energy range 1-20 GeV per nucleon that have most influence [STAS88]. For cosmic rays to reach a spacecraft in Earth orbit, they must penetrate the Earth’s magnetic field. Since they are moving, charged particles, they will tend to be deflected by the magnetic field. However, this tendency is opposed by the energy of the particles as they move at high velocity towards the Earth. Its momentum divided by its charge determines a particle's penetrating ability, and this quotient is referred to as its 'magnetic rigidity'. A cosmic ray will require a minimum magnetic rigidity to reach each point within the Earth's magnetic field [ADAM02]. Particles below the minimum will be deflected and this minimum is called the geomagnetic cutoff value. The cutoff value falls to zero at the edges of the magnetosphere and at the Earth's magnetic poles. Since the cosmic ray flux is highest at low energies, a satellite in Earth orbit will be protected to some extent from cosmic rays by the magnetic field. The degree of protection will depend on the altitude and inclination of the orbit. **The geostationary orbit at an altitude of 35860 km and of crucial importance to communication satellites is afforded virtually no magnetic shielding against cosmic rays; polar orbits, important for Earth observation satellites, are also significantly exposed. Low Earth orbits used by commercial satellite constellations will have variable protection; the most exposed being those in high inclination orbits.**

2.1.4 Solar Flares

In the years around solar maximum the sun is an additional sporadic source of lower energy particles accelerated during certain solar flares and in the subsequent coronal mass ejections [DYER01]. These solar particle events last for several days at a time and comprise both protons and heavier ions with variable composition from event to event. Energies typically range up to several hundred MeV and have most influence on high inclination or high altitude systems. Occasional events produce particles of several GeV in energy and these can reach equatorial latitudes.

2.1.5 Satellite Orbits Environments

The main sources of energetic particles that are of concern to spacecraft designers are:
1) protons and electrons trapped in the Van Allen belts,
2) heavy ions trapped in the magnetosphere,
3) cosmic ray protons and heavy ions, and

4) protons and heavy ions from solar flares.

The levels of all of these sources are affected by the activity of the sun. The solar cycle is divided into two activity phases: the solar minimum and the solar maximum. The cycle lasts about eleven years, with approximately four years of solar minimum and seven years of solar maximum.

There are also extremely large variations in the levels of SEE inducing flux that a given spacecraft encounters, depending on its trajectory through the radiation sources. Missions flying at Low Earth Orbits, Highly Elliptical Orbits (HEOs), and Geostationary Orbits (GEOs), and Planetary and Interplanetary missions have vastly different environmental concerns [DYER01].

2.1.5.1 Low Earth Orbits

Satellites in LEOs pass through the particles trapped in the Van Allen belts several times each day. The level of fluxes seen during these passes varies greatly with orbit inclination and altitude. The location of the peak fluxes depends on the energy of the particle. For protons with $E > 10$ MeV, the peak is at about 3000 km. For normal geomagnetic and solar activity conditions, the flux level drops rapidly at altitudes over 3000 km. However, high-energy protons have been detected in the regions above 3000 km after large geomagnetic storms and solar flare events.

2.1.5.2 Highly Elliptical Orbits

Highly elliptical orbits are similar to LEO orbits, they pass through the Van Allen belts each day. However, because of their high altitude, they also have long exposures to the cosmic ray and solar flare environments regardless of their inclination. The levels of trapped proton fluxes that HEOs encounter depends on the perigee position of the orbit including altitude, latitude, and longitude. If this position drifts during the course of the mission, the degree of drift must be taken into account when predicting proton flux levels. HEOs also accumulate high total ionization dose levels due to both the trapped proton exposure and the electrons in the outer belts where the spacecraft spends a significant amount of time during each apogee pass.

2.1.5.3 Geostationary Orbits

At geostationary altitudes, the only trapped protons that are present are below energy levels necessary to initiate the nuclear events in materials surrounding the sensitive region of the device that cause SEE. However, GEOs are almost fully exposed to the galactic cosmic ray and solar flare particles. Protons below about 40-50 MeV are normally geomagnetically attenuated, however, this attenuation breaks down during solar flare events and geomagnetic storms. Field lines that cross the equator at about 7 earth radii during normal conditions can be compressed down to about 4 earth radii during these events. As a result, particles that were previously deflected have access to much lower latitudes and altitudes. Further, GEO satellites are continuously exposed to trapped electrons, hence, the total dose ionization accumulated in GEO orbits can be severe for locations on the satellite with little shielding.

Table 2.1 Summary of Radiation Sources

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

The following rules must be observed for estimating total dose environments [SPAC96]:

a) For satellites in low inclination (< 28 degrees) LEO, (< 500 km) in both northern and southern hemispheres, typical dose rates due to trapped Van Allen electrons and protons are **100-1000 rad(Si)/year**.
b) For satellites in higher inclinations (between 20 and 85 degrees) LEO orbits in both northern and southern hemispheres, typical dose rates due to increased number of trapped electrons are **1000-10,000 rad(Si)/year**.

2.1.6 Total Ionizing Dose

Total ionizing dose is a cumulative effect which causes degradation of microelectronics and materials. As TID accumulates, parametric degradation occurs, degrading performance, and components can eventually fail to function. TID is caused by exposure to electrons and protons. As mentioned in the previous chapter, some technologies are hardened to TID effects through specialized processing. However, shielding is often used to mitigate the effects of TID on unhardened components. Fig. 2.2 is a plot of total ionizing dose in krads of silicon as a function of aluminum shield thickness for various orbits around the Earth per annum. The two curves in the lower half of the graph are for LEOs that pass through the SAA. The curves that are higher on the graph are orbits that pass through more intense regions of radiation that are at higher altitudes in the belts. At > 300 mils (7.6 mm) of shielding, highly energetic trapped protons dominate the dose levels [BART97].

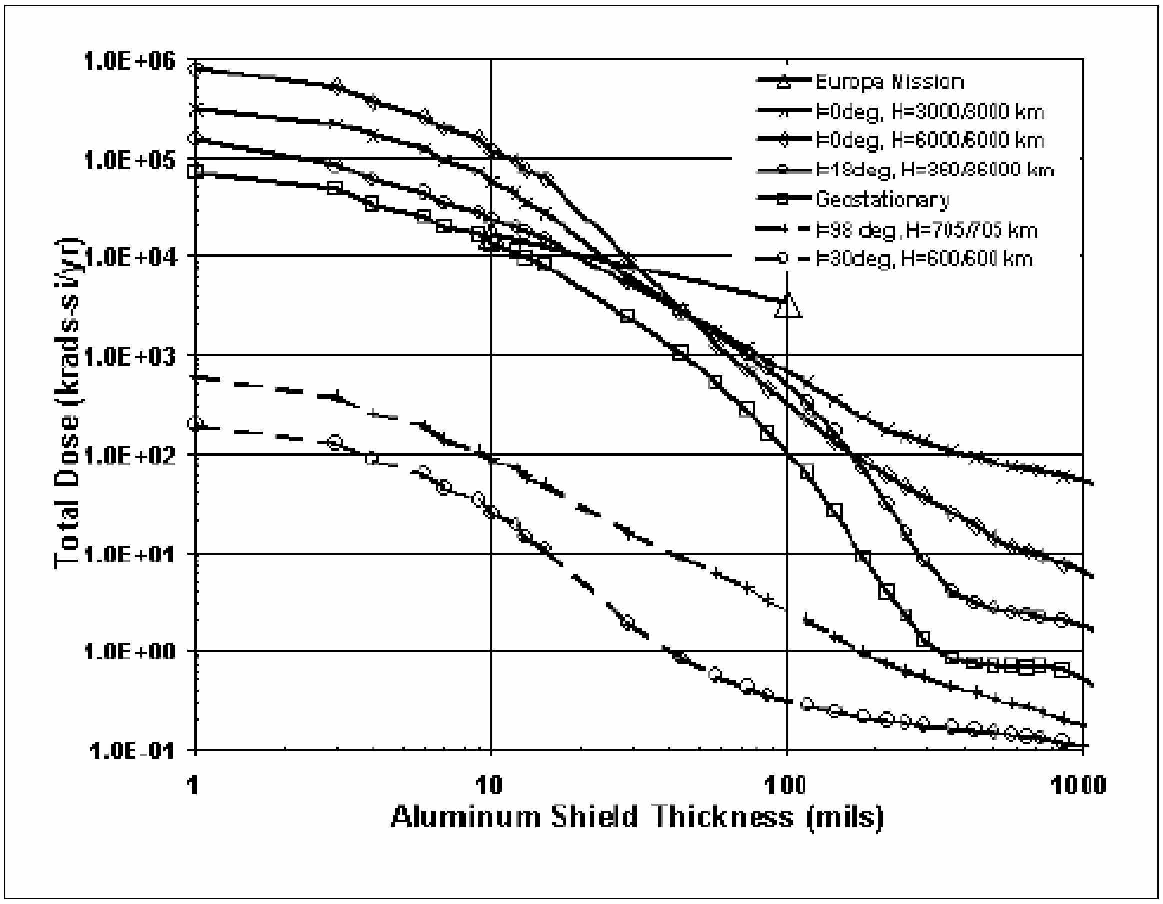


Fig 2.2 Total ionizing dose-depth curves for various orbits around the Earth [BART97]

2.2 Total Ionizing Dose Effects

Ionizing radiation dose is defined as the amount of energy deposited by ionization per unit mass of material. SI Units are J/Kg (Gray). The majority of radiation effects depend on rate of delivery and so dose-rate information is required. In particular, TID radiation induced charge buildup in MOS devices depends on: dose, dose rate, type of ionizing radiation, applied and internal electric fields, device geometry, operating temperature, post-irradiation conditions (e.g. time and temperature), dielectric material properties, fabrication processing, oxide impurities, nitrogen and sodium, final processing packaging, burn-in reliability screens, and aging [HUGE03]. As stated before, issues of IC architecture also impact survivability against radiation effects.

Accumulated dose leads to threshold voltage shifts in CMOS devices due to trapped holes in the oxide and the formation of interface states. In addition increased leakage currents and gain degradation in bipolar devices can occur [LABE96]. **It has been shown that the dominant radiation effects in MOS devices are due to TID effects, and not due to displacement damage, the usual cause of radiation-induced degradation in bipolar devices [HUGE03].**

To understand the operation of the metal-oxide semiconductor field effect transistor (MOSFET or MOS), which is the basic building block of modern digital circuits, refer to Fig. 2.3. The diagram illustrates the cross-section of an n-channel using a p-type substrate. The normal operation of the n-channel MOS (NMOS) transistor is as follows: When a positive voltage is applied to the gate terminal, an electric field is created between the gate and the silicon substrate. In effect, this behaviour is very much the same as a parallel plate capacitor. Due to the presence of the electric field, the majority carriers in the substrate (holes in p-type) will be repelled from the gate-oxide substrate interface and minority carriers (electrons) will be attracted, forming what is termed an inversion layer. When a potential difference is applied between the source and drain terminals, the inversion layer provides a low resistance path for electrons to flow. The device is said to be turned on, and the gate voltage at which the inversion layer just begin to transmit current is termed the threshold voltage of the device.

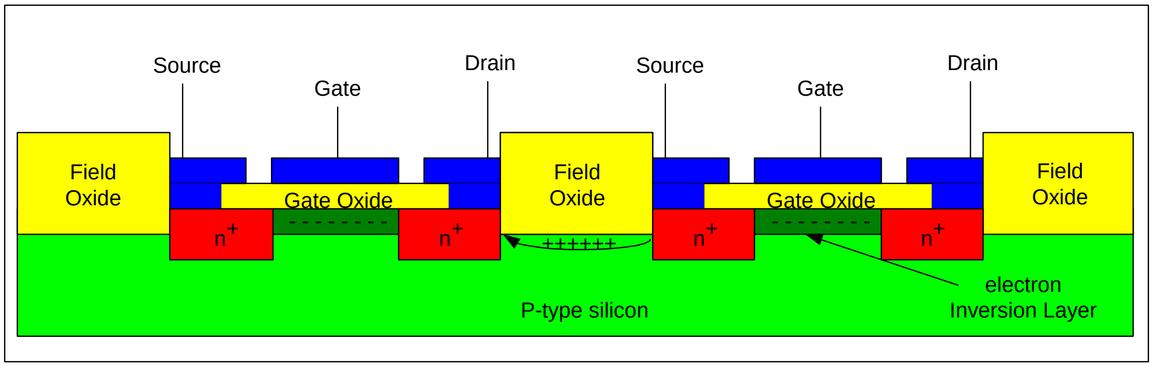


Fig 2.3 Cross section of NMOS device with trapped charge in the oxide

The effect of using the MOSFET device in a radiation environment is that the gate oxide becomes ionized by the dose it absorbs due to the radiation induced trapped charges in the gate-oxide. The trapped charges in the gate-oxide generate additional space charge fields at the oxide-substrate interface. After a sufficient dose, a large positive charge builds up, having the same effect as if a positive voltage was applied to the gate terminal. Therefore, the transistor source to drain current can no longer be controlled by the gate terminal and the device remains on permanently resulting in device failure.

The radiation response of the PMOS transistor exhibits the same pattern, but the effect is opposite. The normal operation of the PMOS transistor is as follows: When a negative voltage is applied to the gate terminal with respect to the substrate, an electric field is also created between the gate and substrate. However, this field is in the opposite direction as in the case of NMOS. When exposed to ionizing radiation, the free electrons move in the direction of the silicon substrate, whereas the positive holes move in the direction of the gate oxide interface where they become trapped. This means that positive charge buildup in PMOS devices is less severe than in NMOS, because the charges get trapped at the gate oxide interface. Thus, the charge buildup in PMOS devices is less effective in shifting the threshold voltage of the device [SROU82]. **Already it should be clear that the electric field in the gate oxide (and in this case its direction) has a major effect on the radiation response of the device.**

The trapped oxide charge distribution can depend on time, and more specifically, on how the electric field in the oxide changes with time. Ionization effects depend not only on the dose, but the response is also time dependent. Fig. 2.3 also shows a thick field oxide, which serves to control the silicon surface charge adjacent to the MOS device and prevent parasitic channels to adjacent devices. Positive charge build-up also occurs here.

Both bulk and Silicon-On-Insulator (SOI) CMOS structures, are subject to the effects described above. SOI is often cited as a specifically radiation-hard technology because of its resistance to transient radiation effects, primarily single-event effects caused by heavy ions or photo-currents produced by high dose-rate ionizing radiation. Although SOI can provide superior device speed because of reduced parasitic capacitance, this technology is not inherently more resistant to total-dose radiation. If anything, the additional oxide interfaces tend to complicate matters.

Of most concern in the total dose effects is the creation of hole-electron pairs in silicon dioxide, an insulator used to: 1) isolate neighbouring transistors (field oxide), 2) provide gate isolation in silicon MOSFET technology, and 3) provide surface passivation in silicon bipolar technology. In any silicon technology in which silicon dioxide is in contact with low acceptor doping level (p-type) silicon, total-dose effects must be considered. The dominant effects are due to holes being trapped at the interface of the Si-SiO2, causing free electrons to be attracted to the interface, and resulting in inversion in the silicon near the interface. If the low doped p-region isolates two n-doped regions, then isolation is compromised and leakage currents may flow between the two n-regions.

In addition to hole trapping, interface traps, which may be charged positively or negatively depending upon bias condition, are also generated at the Si-SiO2 interface. Two effects are associated with interface traps. In n-channel MOS transistors under positive bias conditions, electrons are trapped in these states. This increases the threshold voltage. Electrons transporting through n-channels, or holes transporting through p-channels, undergo Coulomb scattering from the charged interface states. This results in reduction in carrier channel mobility and hence increased channel "on" resistance.

In sub-micron devices, the hole trapping near the Si/SiO2 interface is the primary effect. Charge inversion in the silicon at the interface and as a consequence parasitic leakage paths can be created due to the hole trapping near the Si/SiO2 interface. There are two kinds of leakage paths. The first is the edge-leakage path between the drain and source at the edge of a NMOSFET. The other kind, called field-leakage path is between any two n+-junctions separated by a field oxide. The edge leakage is usually more serious than the field leakage because the shorter path length.

In sub-micron devices studied in this thesis, the radiation effects due to the gate oxide is negligible because the oxide thickness is too thin to trap net charges and the interface quality is too good to be activated by radiations. For a PMOSFET, the leakages are reduced because the silicon surface is induced to favor the accumulation [WANG04, OLDH99].

Dose Rate Effects that are related to the rate at which radiation is absorbed in circuits include upset, latch-up, snap back in integrated circuits, and burn-out in bipolar and n-channel power transistors. All these effects are a consequence of radiation generated photocurrents in p-n junctions. Electron-hole pairs generated in the depletion region of a p-n junction by ionizing radiation are swept out by the high electric field present in this region. This promptly collected charge is termed the prompt photocurrent. Carriers generated within a diffusion length of the depletion region will diffuse to the depletion region where they are collected. These photocurrents sum in digital integrated circuits to produce transient currents that can cause changes in logic levels at digital gates due to charging and discharging of gate capacitance, or transistors being turned on or off. If the dose rate is high enough, the product of photocurrent and resistance causes a drop in power supply voltage across the metal resistance and power supply voltage actually present at the memory cell, and an error is introduced in the memory cell. This phenomenon is called rail-span collapse.

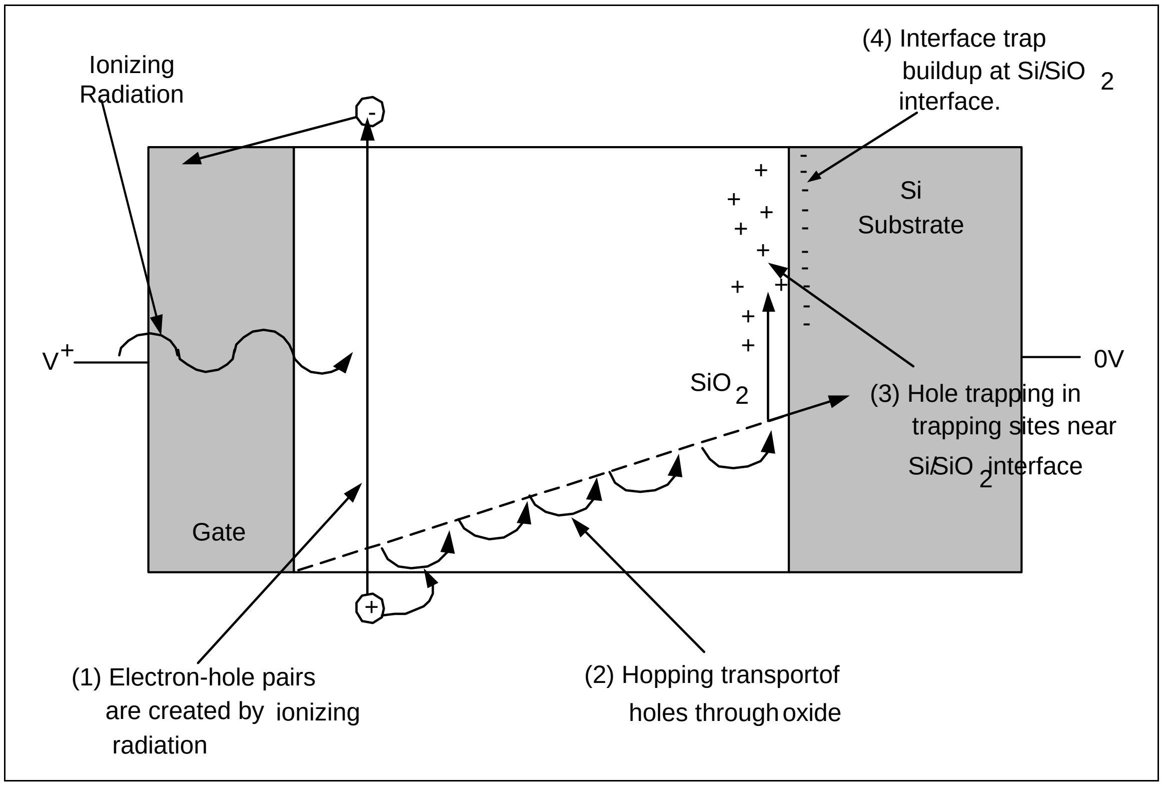
2.3 Ionizing Radiation Effects on MOS devices and IC's

The total-dose ionization problem that occurs in MOS systems is due to the radiation-induced charging, normally positive, of the thin gate-oxide region and isolation oxides, and in the buried oxide for Silicon On Insulator technology. The charge-induced fields result in voltage offsets or shifts in the turn-on voltages of the devices; these offsets or shifts lead to circuit degradation and failure. The incident radiation creates electron-hole (e-h) pairs. The energy required is approximately 17±1 eV [BENE86] to generate an e-h pair in SiO2. This will result in a total number of e-h pairs generated per unit dose in SiO2 of approximately $4 \times 10^{13} cm^{-3} rad^{-1}$(SiO2). The radiation-induced oxide charging problem is complicated by the details of the time dependence of the radiation response of the simple MOS structure shown in Fig 2.6, having to do with a wide variation in the characteristic time scales for the various physical processes involved. The complexity of the time-dependent response has implications in prediction of circuit response.

In this section we try to analyze the problem of radiation interaction with MOS structures with the aim of explaining the more important aspects, which are essential for a physical comprehension of the degradation observed in electronic devices and circuits under radiation, and its understanding is also essential to design effective radiation damage mitigation techniques.

2.3.1 Overview of Ionizing Radiation response of MOS Structures

The MOS radiation response involves several different processes [MCLE99]. Each of these processes depends on time, temperature, **applied field**, process history, etc. as mentioned in the previous section. A basic illustration of the overall radiation response of the MOS transistor is shown in Fig. 2.4.



Ionizing Radiation
(4) Interface trap buildup at Si/SiO2 interface.
Si Substrate
V+
0V
SiO2
(3) Hole trapping in trapping sites near Si/SiO2 interface
Gate
(1) Electron-hole pairs are created by ionizing radiation
(2) Hopping transport of holes through oxide

Fig. 2.4 The basic radiation effect in MOS transistors

The four main processes involved in the radiation response of MOS devices are illustrated in Fig. 2.4, [OLDH99, MCLE99]. First, the ionizing radiation acts with the gate oxide layer to produce electron-hole pairs.

The second process in Fig. 2.4 is the slow transport of holes toward the oxide-silicon interface **due to the presence of the electric field**. It is this transport that explains the short term recovery of MOS devices. When the holes reach the interface, process 3, they become trapped in vacancy sites and this is the main cause of the permanent threshold voltage shift in MOS devices.

The fourth process is the buildup of interface states in the substrate near the interface. A negative space charge region is created near the interface because of the positive charge buildup of process 3. These four processes are discussed in greater detail in the sections that follow.

2.3.1.1 Electron Hole-pair (e-h) creation

As previously mentioned, the part of a MOS transistor that is most sensitive to ionizing radiation is the oxide insulating layers. Consider Fig. 2.5, a positive bias voltage is applied to the gate terminal as in the case of NMOS. When ionizing radiation strikes the gate oxide, electrons are freed from the oxide molecules and are swept by the direction of the electric field towards the gate terminal. The free holes move in the direction of the substrate.

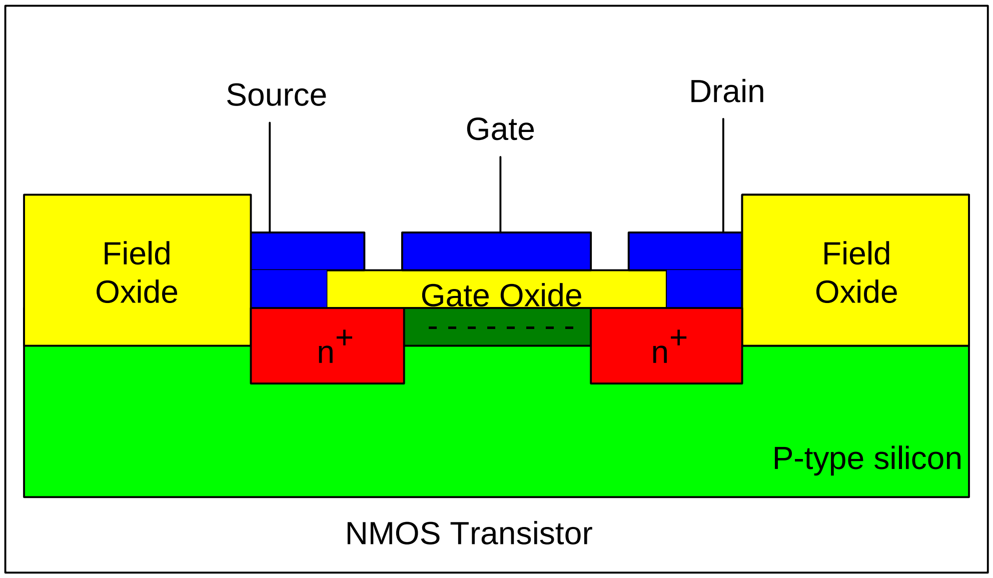


Fig 2.5 NMOS transistor

Some fraction of the electron-hole pairs recombine depending on the type of incident particle and **the applied gate to substrate voltage, i.e. the electric field**. The mobility of the electron is orders of magnitude larger than that of the holes in the gate oxide, and is swept away very quickly in the direction of the gate terminal. The time for the electrons to be swept away is on the order of 1 ps [MCLE99].

Two primary models of recombination have been developed. The columnar model applies when the e-h pairs are close together, and thus a large number recombine. The geminate model applies when the e-h pairs are widely separated, so that a much smaller number of carriers will recombine [MA89]. The surviving holes cause a negative voltage shift in the electrical characteristics of MOS devices. These changes appear in the threshold voltage (VT) or flatband voltage (VFB) for MOS capacitors. These changes can be separated into two components: the voltage shift due to charge in the oxide, ΔVot, and that due to the interface traps, ΔVit. The fractional yield of carriers, those escaping recombination in SiO2, is discussed as a function of the applied field for various sources of radiation in [MA89]. The number of the surviving holes determines the initial response of the device after a short pulse of radiation.

2.3.1.2 Hopping transport of holes

At room temperature and over a period of time, on the order of a few picoseconds to seconds, the holes undergo a hopping transport through the oxide **in response to any electric field present.** They move to the Si substrate for the figure depicted above. This hole transport is the second major effect of the MOS response.

The hole transport process is dispersive in time. Two models have been proposed for this dispersive transport: a) hopping transport where the holes directly tunnel between localized trap sites within the SiO2 band gap, and b) multiple trapping, where the holes are trapped at localized trap sites moving within the oxide due to drift and diffusion between trapping events. This dispersive transport process is sensitive to **applied field**, temperature and oxide thickness. Both of these models can be mathematically described by the continuous-time random walk (CTRW) model [MA89].

2.3.1.3 Deep Hole Trapping

When the holes reach the SiO_2 interface, some are captured in long term trapping sites causing negative voltage shifts. This long-lived, radiation induced voltage shift component is the most commonly observed form of radiation damage in MOS devices. Hole trapping and annealing are sensitive to the processing of the oxide, **applied field**, and temperature [MACL89]. This long-term trapping of holes near the SiO_2 interface is the third major effect of the MOS radiation response.

2.3.1.4 Radiation induce interface traps

The radiation induced interface traps at the SiO_2 interface is the fourth and final component of the MOS radiation response. The interface traps gives rise to a voltage shift component that depends on the silicon surface potential [MCLE99]. Both prompt interface traps, present immediately after a radiation pulse, and a delayed time-dependent buildup of states that can continue for thousands to tens of thousands of seconds at room temperature can be seen. It is also important to note that the radiation induced interface trap generation is strongly dependent on the processing steps of MOS devices, as well as on other variables, such as temperature and **applied field** [MA89, LENA99].

The time-dependent recovery curve in Fig 2.6 shows the radiation-induced shift in threshold voltage as a function of log time. The NMOS device is under positive gate bias at room temperature after exposure to an ionizing radiation pulse of ~1 ms. This figure relates the major features of the response to each of the primary processes discussed above. The initial shift (shown in red) is governed by the electron/hole pair creation in the SiO2 bulk and by the initial recombination processes. The early annealing (shown in blue) is due to the hole transport process. The remaining shift in V_T is due to the deep hole trapping near the SiO2/Si interface. This anneals linearly with log time. The solid curve in Fig 2.6 corresponds to transport, trapping, and annealing of holes alone. In addition to long-term annealing of trapped holes, however, a buildup of radiation induced interface traps may occur, typically in the time regime between ~10^{-2} and 10^{-3} s, which is indicated by the green curve in Fig 2.6. If the interface trap contribution is large, the change in threshold voltage becomes positive, giving rise to what is called super recovery or rebound [MA89].

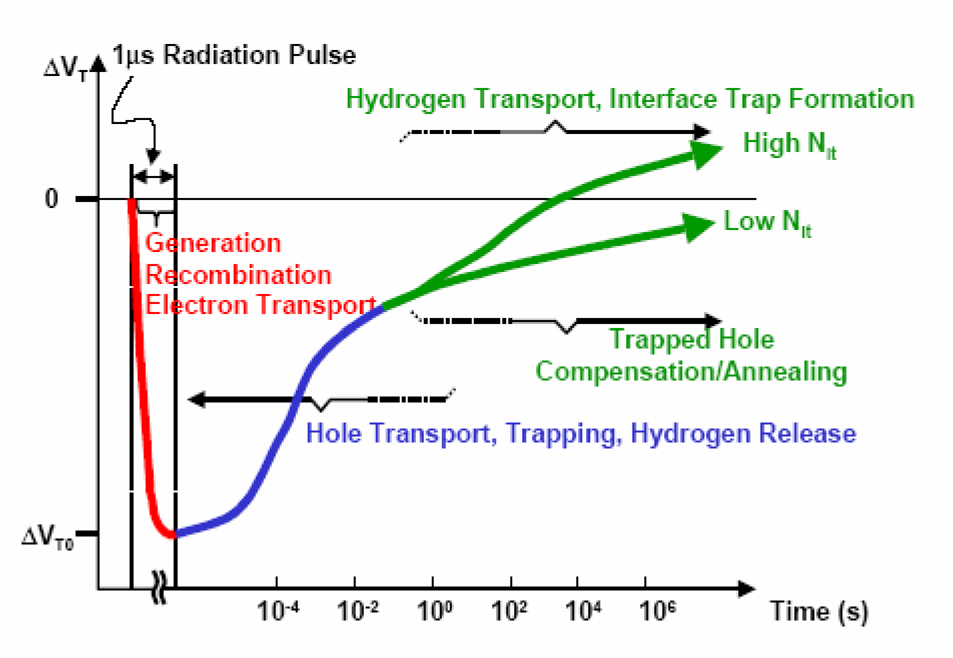


Fig 2.6 Summary of the transient response of an NMOS transistor's threshold voltage to a radiation pulse [OLDH99]

2.4 Consequences of radiation on the electrical parameters of a MOS Transistor

As the dose absorbed by a device increase, more electron-hole pairs are created and becomes trapped in the silicon oxide or interface traps. The number of electron-hole pairs is proportional to the amount of energy absorbed in the device, hence, the total damage is roughly proportional to the dose absorbed by the device [OLDH99, MCLE99].

The number of deep hole traps in the bulk of a thermally grown silicon dioxide layer given today's techniques is usually fairly small. Most of the traps are located near the Si/SiO2 interface, or near the gate electrode/SiO2 interface. The holes generated by ionizing radiation in the bulk of an oxide layer will be swept under a positive gate bias towards the SiO2/Si interface, and some fraction of them will be trapped, depending on the hole trap density and capture cross-section.

One of the major electrical consequences of the radiation induced charging of the silicon oxide is a shift in the voltage operating points for the devices such as the threshold voltage V_T for MOSFETs. The threshold voltage can be written as

$$V_T(t) = V_T^{\ 0} + \Delta V_T(t) \qquad (1)$$

where $V_T^{\ 0}$ is the threshold voltage before irradiation and $\Delta V_T(t)$ is the change in the threshold voltage due to radiation exposure and is time dependant [OLDH99].

Based on the discussion in section 2.3.1, the threshold voltage shift can be broken into three components:

$$\Delta V_T(t) = \Delta V_{ST}(t) + \Delta V_{OT}(t) + \Delta V_{IT}(t) \qquad (2)$$

- $\Delta V_{ST}(t)$ is due to the generated mobile holes transporting in the oxide.
- $\Delta V_{OT}(t)$ is due to the trapped holes near the oxide-silicon interface.
- $\Delta V_{IT}(t)$ is due to the charged interface traps.

These components can be expressed explicitly as [OLDH99]:

$$\Delta V_{ST}(t) = \frac{-q}{C_{ox}} \int_0^{t_{ox}} \frac{\rho(x,t)}{t_{ox}} dx \qquad (3)$$

$$\Delta V_{OT}(t) = \left(\frac{-q}{C_{ox}} \right) \Delta N_{OT}(t) \qquad (4)$$

$$\Delta V_{IT}(t) = \frac{-\Delta Q_{IT}(t)}{C_{ox}} \qquad (5)$$

q is the electronic charge,

t_{ox} is the oxide thickness,

C_{ox} is the oxide capacitance per unit area ($C_{ox} = \frac{\varepsilon_{ox}}{t_{ox}}$ where ε_{ox} is the dielectric constant of the oxide).

$\rho(x,t)$ is the charge distribution in the oxide per unit volume as a function of the distance from the gate oxide interface (x) and time.

The first observation from equation 3 is that the voltage shift due to this contribution is negative when the charge is positive. This can easily be understood by considering for example the p-channel transistor. The positive charge trapped in the oxide repels the holes in the inversion channel. Hence, one needs to apply a more negative voltage to the gate to create the same inversion condition. Another important consideration is that the closer the charge distribution to the silicon-oxide interface, the bigger the threshold voltage shift. For example, if we have a charge distribution close to the gate-oxide interface with a constant density of A, for $x < Z$ and zero for $Z < x < t_{ox}$, i.e.

$$\rho(x,t) = A \quad \text{for} \quad x < Z \quad \text{and} \qquad (6)$$

$$\rho(x,t) = 0 \quad \text{for } Z < x < t_{ox} \qquad (7)$$

Thus, we have,

$$\Delta V_{ST}(t) = \frac{-AZ}{\varepsilon_{ox}} \qquad (8)$$

Suppose now we have a similar distribution in the oxide but this time close to the silicon-oxide interface, i.e.

$$\rho(x,t) = 0 \quad \text{for} \quad 0 < x < t_{ox} - Z \tag{9}$$

$$\rho(x,t) = A \quad \text{for} \quad t_{ox} - Z < x < t_{ox} \tag{10}$$

The voltage shift becomes,

$$\Delta V_{ST}(t) = \frac{-AZ}{\varepsilon_{ox}} (\frac{t_{ox}}{Z} - 1) \tag{11}$$

From equation 11, we have an additional multiplicative factor > 1 (since $t_{ox} > Z$) which is not present in equation 8. Thus the change in threshold voltage shift is indeed larger for the charge distribution closer to the silicon-oxide interface. This conclusion was also mentioned earlier. (The positive charge buildup in PMOS devices is less severe than in NMOS, because the charges get trapped at the gate oxide interface. Thus, the charge buildup in PMOS devices is less effective in shifting the threshold voltage of the device [SROU82]).

Trapped holes can be removed or neutralized by compensating electron trapping, either by thermal annealing, or by tunneling of electrons from the silicon substrate or gate. Complete thermal annealing often requires temperatures above 300°C. The temperature for annealing depends on the distribution of energies inside the SiO2 band gap for the trapped holes, as shallower trap levels emit charge at lower temperature [MA89]. The tunneling annealing process is roughly linear with log(t) dependence, where t is the time after irradiation. Tunneling probability decreases with distance. Because of this decrease in probability due to distance, only defects within 4-5 nm from either interface are neutralized due to tunneling. However, this also means that for very thin oxides (< 10 *nm*), significant neutralization of trapped holes could occur via tunneling in a relatively short time interval.

Field oxides used to isolate MOS devices, inclusive of isolation layers used in SOI technology, all suffer the same type of changes described in the above sections. These changes result in parasitic leakage currents that must be correctly taken into account when a particular device or architecture radiation analysis is performed.

2.4.1 The time-dependant response of MOS structures

In this section we further try to aid in the understanding of the time-dependant response of the MOS structure by means of Fig. 2.7.

Consider Fig 2.7 the pre-irradiation condition is depicted in part (a). The corresponding C-V curve for this condition is depicted in Fig. 2.8 with $t = 0^-$. At $t = 0$ (part (b) of Fig 2.7), the radiation pulse occurs causing electron/hole pairs across the oxide bulk. In a time on the order of picoseconds (part (c) of Fig. 2.7), some of the electron hole pairs recombine, while the highly mobile electrons gets swept toward the gate and collected (part (d) of Fig 2.7), **under the influence of the positive gate voltage applied**. The magnitude of the shift in the flatband voltage ($\Delta V_{fb}\left(0^+\right)$) is maximum at this time. The holes then begin their relatively slow transport toward the silicon-oxide interface, where a fraction of them become trapped in long term trapping sites, while others gets collected by the substrate electrode. This is illustrated in part (e) of Fig. 2.7. Because less positive charge remains in the oxide, the C-V curve has annealed back partially. The final charge configuration (at $t = t_2$) after completion of the hole transport is depicted in part (f) of Fig 2.7.

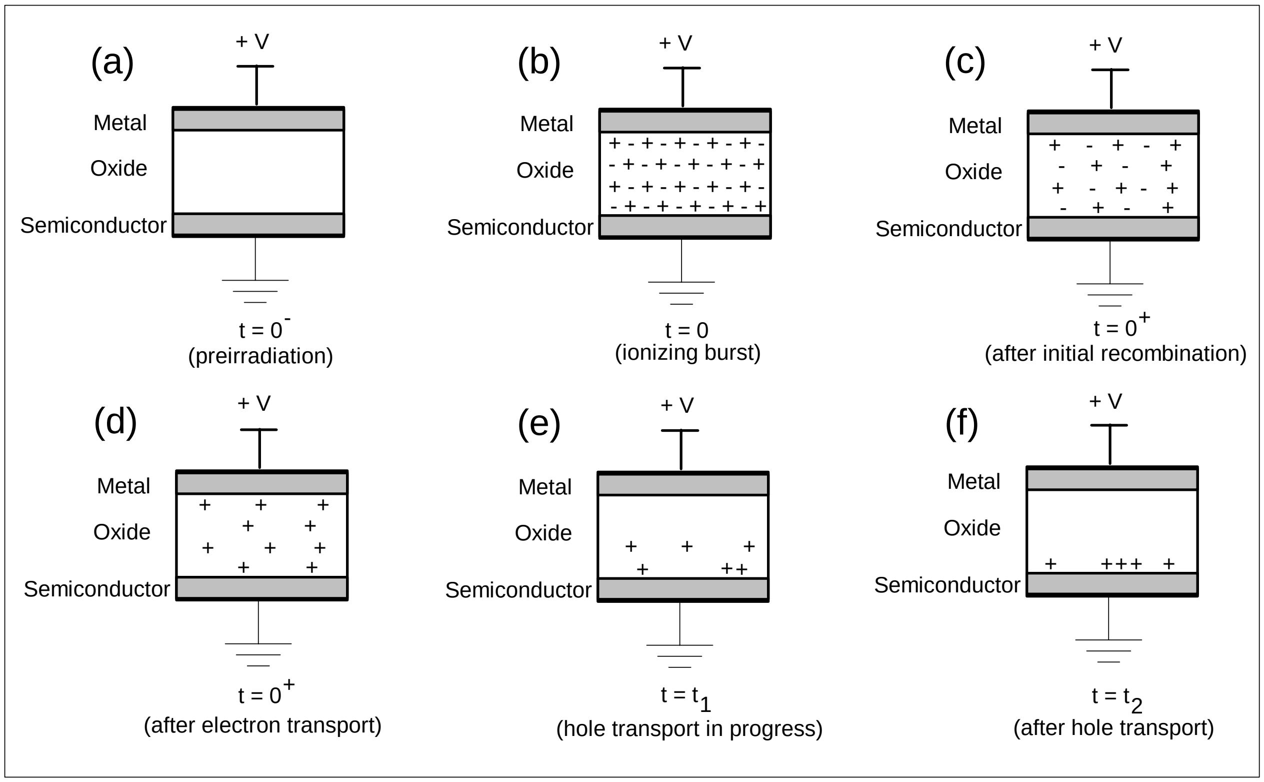


Fig 2.7 Charge generation, recombination, transport and trapping [OLDH99]

Only the long term, trapped holes remain near the silicon-oxide interface, giving rise to a long-term flatband voltage shift in the C-V characteristics.

The actual situation is more complicated, since there can be long term annealing of the trapped holes near the silicon-oxide interface due to either electron tunneling from the Si substrate or thermal injection from the traps [OLDH99].

Because of several processes that are involved in the radiation response of MOS oxides, with each having different characteristic times, the overall time history of the recovery can be very complex [MCLE89, OLDH99]. This has important implications for testing procedures, hardness assurance, and prediction.

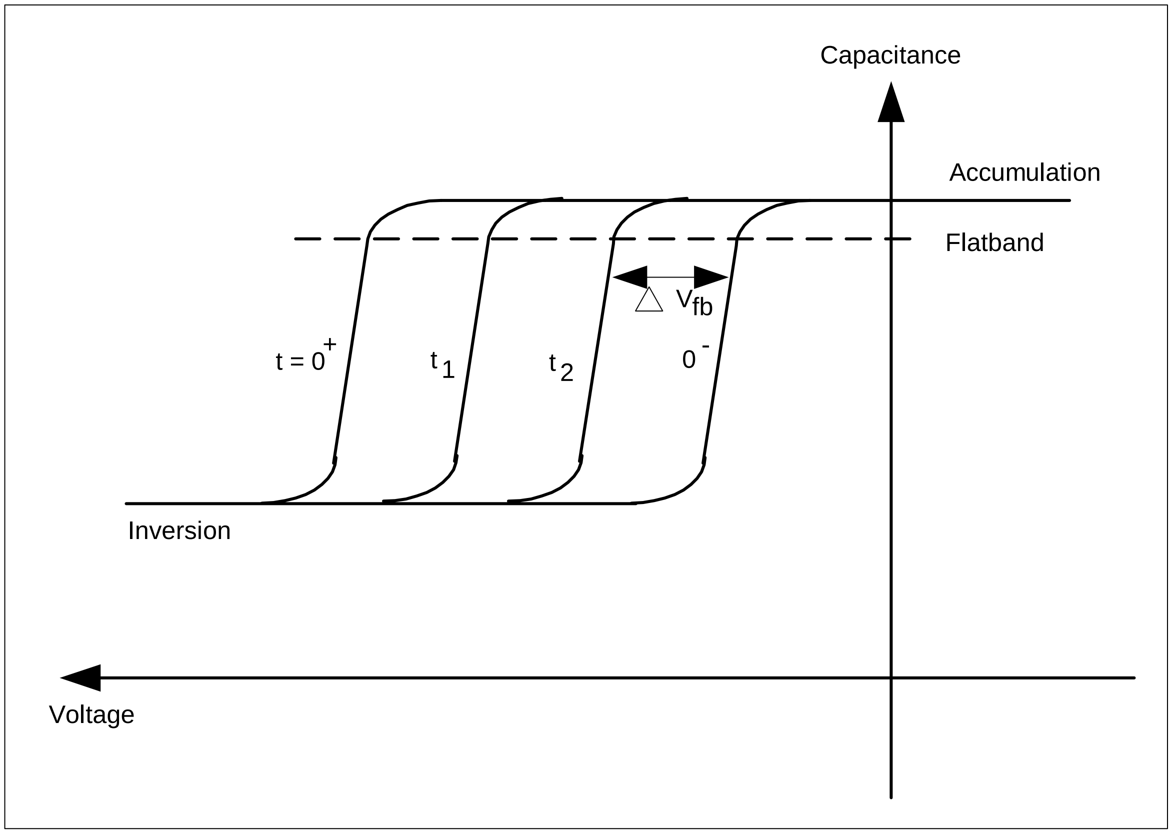


Fig 2.8 Radiation induced shift of the C-V curves for a MOS capacitor [OLDH99]

2.4.2 Increase in Transistor leakage current

Another major problem with CMOS circuits is latchup, where a low resistance path between the power supply and ground is formed. These paths can be formed by parasitic bipolar transistors which are built into the CMOS structure, one pnp and one npn, if the gains of these transistors are large enough that they are driven into saturation.

In integrated circuits using MOS and CMOS technology, radiation damage results in device malfunction. One of the most significant effects of radiation damage in MOS structures is an increase in leakage current [WANG04].

The leakage current is current flowing through a transistor which should be biased off, or current flowing between adjacent transistors. Low resistance paths which allow leakage currents can be formed by the increase in density of interface traps, or by shifts in a transistor's threshold voltage caused by trapped oxide charges.

In linear CMOS devices, leakage currents impair the high input impedance which MOSFETs usually have. In an MOS transistor which is normally biased off, the drain to source leakage currents is less than 1 pA before irradiation, but increase to about 1 nA at 100 krad(Si), and hundreds of nanoamperes after 300 krad(Si) [MA89].

For complex integrated circuits, this can lead to significant increases in the power supply current. **In the studies of total dose effects on FPGAs, the first sign of damage noted was an increase in power supply current due to the onset of leakage currents** [WANG04].

As the individual transistors making up a complicated CMOS integrated circuit such as a microprocessor or FPGA are damaged by ionizing radiation, the characteristics of the overall circuit will change. The damage to individual transistors and CMOS inverters can result in such effects as increased power supply current (due to leakage currents or transistors which should be off turning on), logic failures, latchup effects, or changes in circuit timing. The response to radiation of an integrated circuit made up of thousands of logic gates is, however, difficult to predict [WANG04].

2.5 Single Event Effects

SEE are caused by ionization, as a consequence of the impact of a heavy ion (cosmic ray) or proton. The ionization induces a current pulse in a p-n junction. SEE covers both SEU, or 'soft error', and Single Event Latch-up (SEL).

2.5.1 Single Event Effects classifications

SEE covers [LABE96]:

- Single Event Upset (SEU), or 'soft error'
- Single Event Latch-up (SEL)
- Single Event Burnout (SEB)
- Single Event Gate Rupture (SEGR)
- Single Event Snapback (SES)

2.5.1.1 Single Event Upset

The deposited charge is sufficient to flip the value of a digital signal. Single Event Upsets normally refer to bit flips in memory circuits, but may also in some cases directly affect digital signals in logic circuits. Fig 2.9 illustrates how an energetic particle can produce a spurious electrical signal. The particle produces charges along its path, in the form of electrons and holes. These are collected at the source and drain, and a current pulse appears. This can be large enough to produce an effect like that of a normal signal applied to the transistor.

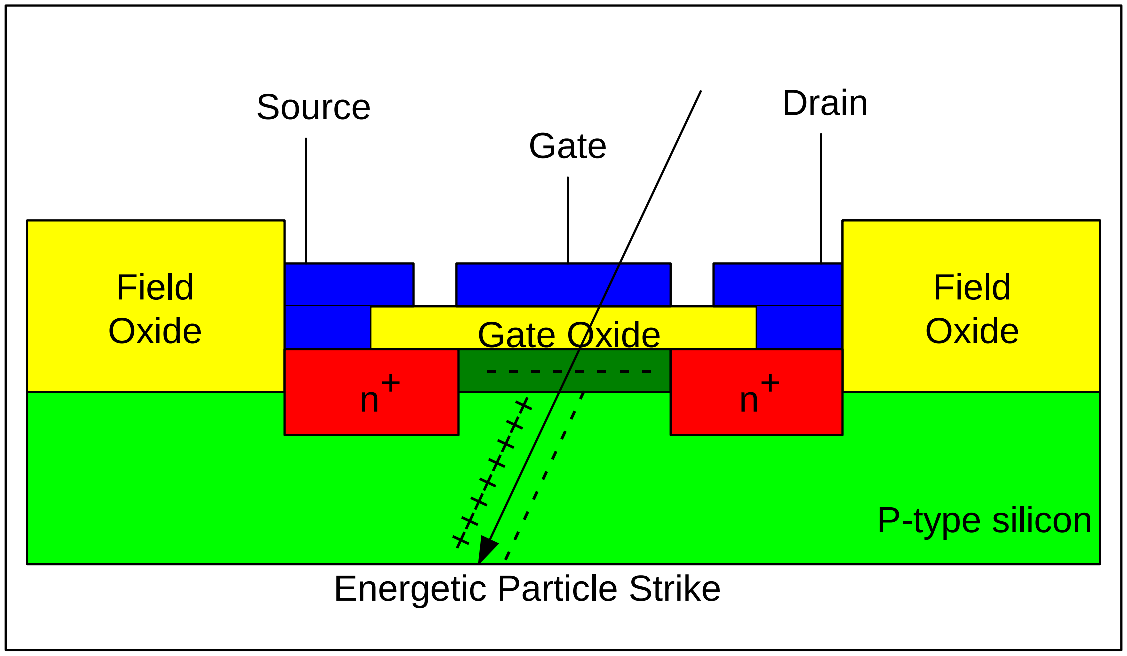


Fig 2.9 Energetic Particle strike in the MOS Transistor

2.5.1.2 Single Event Latch-up

Bulk CMOS technologies (not Silicon On Insulator) have parasitic bipolar transistors that can be triggered by a locally deposited charge to generate a kind of short circuit between the power supply and ground. CMOS processes are made to prevent this from occurring under normal operating conditions, but a local charge deposition from a traversing particle may potentially trigger this effect. Single event latch-up may be limited to a small local region, or may propagate to affect large parts of the chip. The large currents caused by this short circuit effect can permanently damage components if they are not externally protected against the large short circuit current and the related power dissipation.

2.5.1.3 Single Event Burnout

Single event burnout refers to destructive failures of power MOSFET transistors. This destructive failure mechanism is normally associated with failures in the main switching transistors of switching mode power supplies. In SEB an ion that traverses the transistor structure through the source can induce a current flow that turns on the parasitic npn transistor below the source thereby initiating forward biased second breakdown. This leads to device destruction if sufficient short-circuit energy is available.

2.5.1.4 Single Event Gate Rupture

In SEGD an ion that traverses the transistor through the gate, but avoids the p-regions, can generate a plasma filament through the n-epi layer that applies the drain potential to the gate oxide, damaging (increased gate leakage) or rupturing the gate oxide insulation (device destruction).

2.5.1.5 Single Event Snapback

Single event snapback is similar to SEL, but without the PNPN structure. It is induced in N-channel MOS transistors that switch large currents. The effect is that the transistor drain junction is forced open and stays open.

Table 2.2 shows a resume of different Single Event Effects classified by device and by sensitive areas [DENT00].

Table 2.2 SEE categories by device and by sensitive areas

Device Type	Sensitive Area	SEU Types
Memories	Memory cells Control logic	Bit flips Bit flips if sequential, transient if combinational
Combinational Logic	Combinational logic	Transient
Sequential Logic	Sequential Logic	Bit flips
FPGA's	Combinational Logic Sequential Logic	Transient if combinational CLBs, bit flips if CLBs based on LUTs. Bit flips
Microprocessors	Registers, caches, sequential, control logic Combinatorial logic	Bit flips Transients
ADCs, DACs	Analog Portion Digital Portion	Transients Bit flips or transient depending of the design
Linear ICs	Analog area	Transients
Photodiodes	Photodiode	Transients

2.5.2 SEU effects

The most common circuit sensitive to SEU is the memory element. The memory cell is designed so that it has two stable states, one that represents a stored '0' and one that represents a stored '1.' In each state, two transistors are turned on and two are turned off (SEU target drains). A bit-flip in the memory element occurs when an energetic particle causes the state of the transistors in the circuit to reverse. This phenomenon occurs in many microelectronic circuits including memory chips and microprocessors. In a spaceborne computer, for example, a bit-flip could randomly change critical data, randomly change the program, or confuse the processor to the point that it crashes.

The occurrence of the SEU in a CMOS latch or SRAM cell is illustrated in Fig 2.10. When ion strikes at the reverse biased drain junction of the NMOSFET in the "off" state, it causes the node voltage to drop from high to low. This transition propagates along the feedback loop and tries to rewrite the state. In the mean time there is a recovery process that the "on" PMOSFET keeps pulling the struck node back to the original high state. The competition between the feedback process and recover process governs the SEU response [WANG04]. If the feedback process is longer, the node is recovered. If the recover process is longer, the node changes state and an SEU occurred.

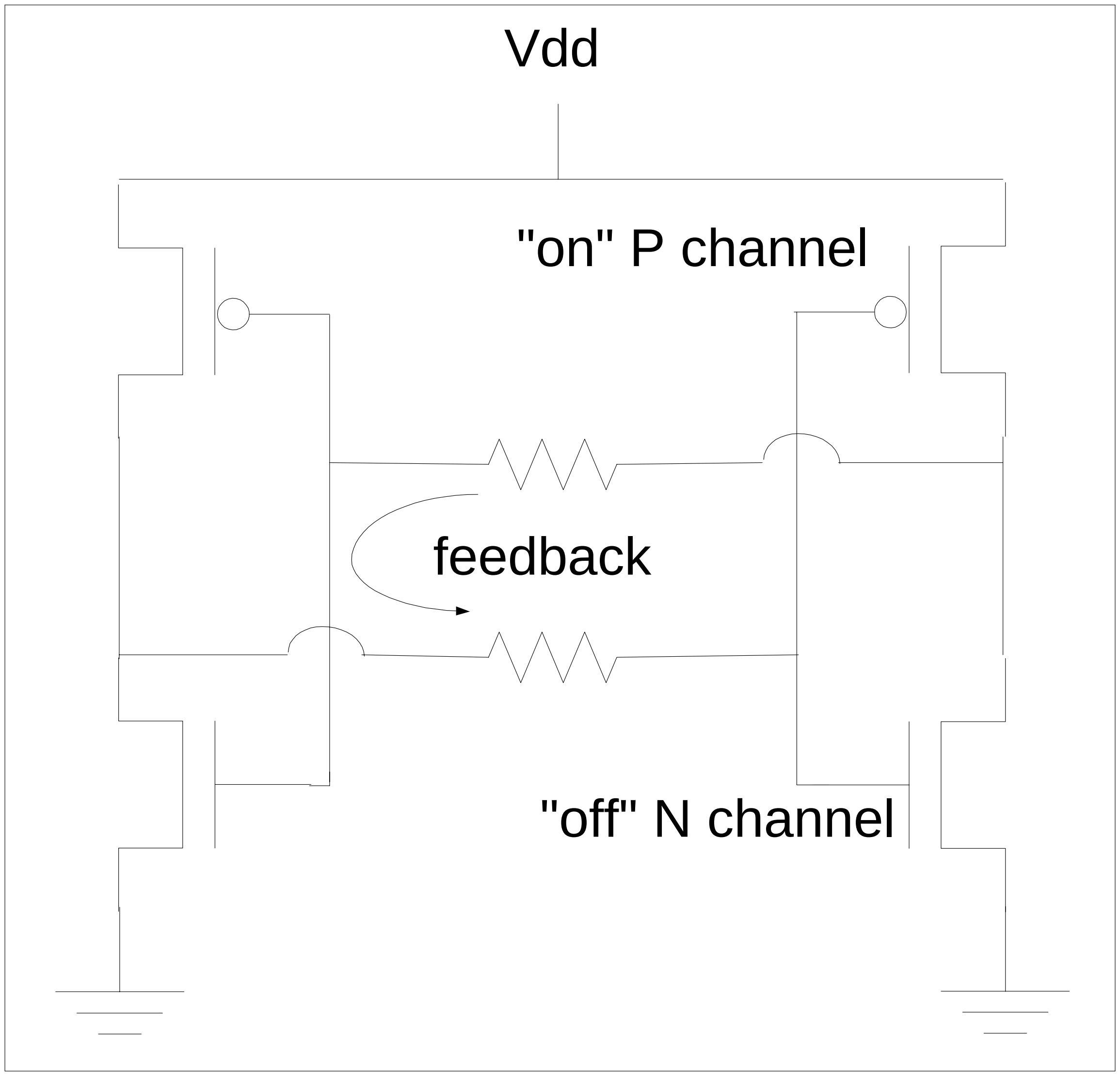


Fig 2.10 Competition between the feedback process and Recover process governs the SEU response of a latch (or SRAM cell)

Charged particles can also induce transient current pulses in combinatorial logic, in global clock lines, and in global control lines. These single event transients (SETs) have only minor effects in present 0.8 to 0.7 micron technologies since the speed of these circuits is insufficient to propagate a 100 to 200 ps SET over any appreciable distance through the circuit. Fig 2.11 shows a typical sequential circuit topology. An upset in the combinational logic can generate an error that is going to be stored in the flip-flop U2 if the speed of the circuit is high enough to propagate the error before the clock change the state of the flip-flop. If the speed is not high enough, the upset in the combinational logic will disappear before the clock change the state of the flip-flop U2, for example.

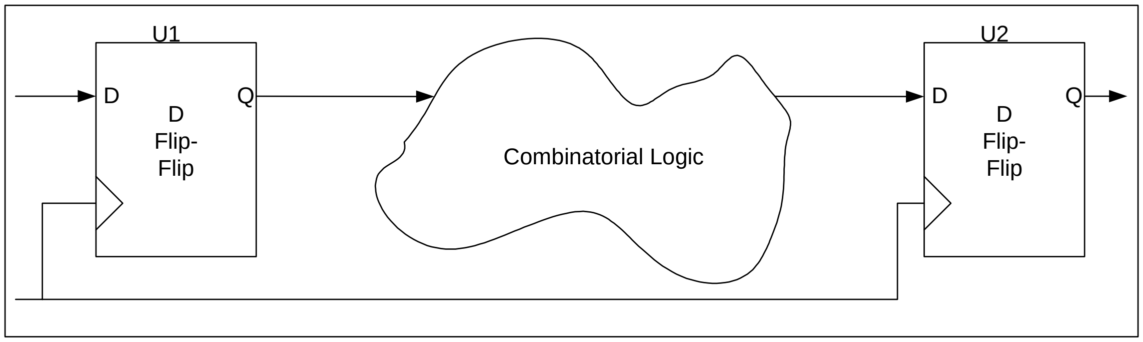


Fig 2.11 Typical sequential circuit topology

However, as smaller feature size and thus faster technologies are becoming strongly used in spacecraft systems where transient pulses generated by charged particle hits can be indistinguishable from normal circuit signals, an upset in the combinational logic can be propagated fast to flip-flops input provoking errors in the circuit.

Consider Fig 2.12, if the ion-strike-induced transient pulse can propagate through the network and result in an error in a storage element, an SEU occurs. This type of SEU is often referred as combinational logic SEU, SET, or SET-induced SEU.

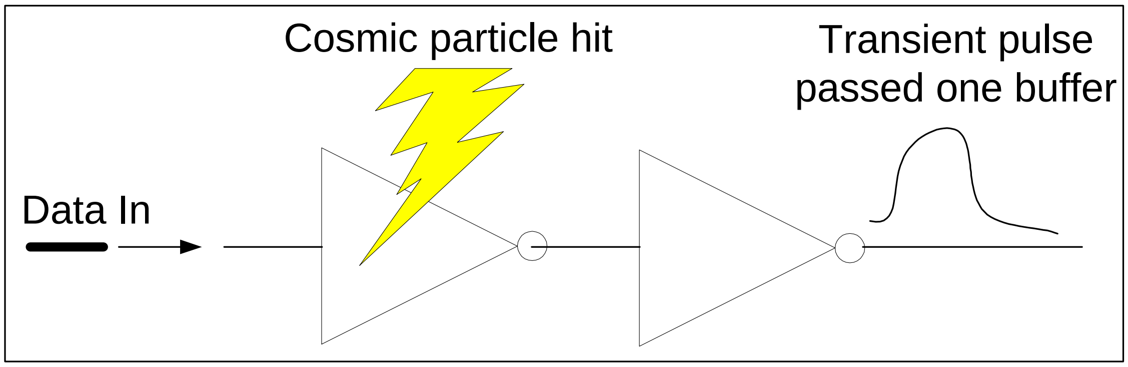


Fig 2.12 SET occurs when the ion induced pulse can propagate through the circuit network [WANG04].

2.6 Displacement Damage

An incident particle or photon capable of imparting energy of about 20 eV to a silicon atom can dislodge it from its lattice site [MA89, MESS92]. Displacement damage creates defect clusters. For example, a 1 MeV neutron transfers about 60 to 70 keV to the Si recoil atom, which in turn can displace roughly 1000 additional atoms in a region of about 0.1 μm radius. Displacement damage manifests itself in two ways; the formation of mid-gap states, and/or a change in doping characteristics. The formation of mid-gap states facilitates the transition of electrons from the valence to the conduction band. In depletion regions, this leads to an increase in the generation current of reverse-biased pn diodes. In forward biased junctions, or non-depleted regions, mid-gap states facilitate charge loss by recombination. States close to the band edges facilitate trapping, where charge is captured and released after a certain time.

2.7 Approaches toward Radiation Hardened Integrated Circuits for TID

Commercial electronics can frequently survive 3-10 krad(Si) of total dose without significant parametric degradation. The failure mechanism is typically field-oxide inversion, resulting in increased leakage current. They can also remain functional (although degraded) from 10-30 krad(Si), but they may suffer a high single-event upset rate or possible latch-up when struck by heavy ions.

There are three possible levels on which the radiation tolerance of a CMOS IC can be improved [KERN88, ALEX96, HUGE03]. The first method would be to modify the manufacturing process of the IC's. For example, by reducing the oxide thickness in a MOS transistor, the device becomes less susceptible to TID degradation, because fewer charges can become trapped in the smaller oxide volume. However, with the shrinking device volume, it becomes more susceptible to SEE, due to a smaller particle energy being necessary to cause an upset.

Some fabrication processes used to harden integrated circuits (ICs) against total ionizing dose are closely guarded secrets, protected either by government classification or company proprietary restrictions. What can be examined, though, are some particulars about the principal process factors affecting the total dose tolerance [NICK03].

The first step in hardening a CMOS IC against total dose radiation is to minimize voltage shifts, or their impact in the circuit, due to radiation induced charge trapping in the gate and field oxides. Two approaches can be used, either individually or in parallel: reducing the number of holes trapped in dielectric layers, and compensating for the trapped holes with trapped electrons.

The easiest way to minimize the trapped-hole density, as mentioned above, is to thin the oxide [MA89]. A clean gate oxide less than 4-5 nm thick, which is typical of today's commercial integrated circuits, can usually survive up to 100 krad(Si) or more with no process changes. On the other hand, where local oxidation of silicon is used, field oxides must remain thick to meet isolation and planarity requirements.

Minimizing the trapped-hole density in them is much trickier and requires special processing. Adding electron traps to offset hole traps is another method of countering radiation effects on field or isolation oxide structures. The addition of electron traps within the a-SiO2 structure is achieved through implantation or introduction of an element. Hardness levels in excess of 300 krad(Si) can be achieved [MA89]. A specific approach for radiation hardened ICs by manufacturing changes is outlined below.

Use of ion implanted silicon-dioxide films: It is observed that radiation-induced interface-trap buildup is suppressed using ion implanted SiO2. By using a large arsenic ion dose, the interface-state buildup can be suppressed by one order of magnitude. It is found that interface-state buildup depends on the ion dose, the gate bias during irradiation and the annealing atmosphere [MA89].

After applying this technique to a conventional bipolar process, the current gain in lateral pnp transistors degraded by only 10% after 10 Mrad(SiO2) irradiation. Radiation induced trapped positive charges can be compensated by implanting aluminium. Aluminium in SiO2 films acts as an electron trap, compensating for the positive trapped charges when in the right concentrations. Si atoms can also be implanted into silicon dioxide films. The Si changes the SiO2 stoichiometry to an SiOx stoichiometry, providing electron traps. Process conditions must be controlled. Temperatures over 900_C cause Si to diffuse rapidly. The diffusing Si tends to form nano-crystals, reducing its compensating properties [NICK00].

The advancement in modern fabrication technologies and a need for faster and smaller IC's, have led to devices evolving to become immune to TID effects, though not completely. There is not much that the system engineer can do as far as the fabrication method is concerned, and to use radiation hard components would defeat the main objective of keeping the satellite cost as low as possible. Therefore, although important, we will not concentrate on this aspect.

The second method involves the use of special layout techniques, which solved the problem of radiation induced leakage currents and SEL and reduce the vulnerability of error due to SEU. A new radiation tolerant transistor structure can be obtained without any process fabrication modifications [SAMI04]. The NMOS transistor and field leakage normally induced by ionizing radiation can be remedied by acting on the work function of the transistor gate at the transistor edges. The technique also works in a CMOS process where transistor source and drains are silicided. This method decreases device density that can be achieved with the technology and slows down digital circuits due to an increase in node capacitances. It was demonstrated that the functionality of the transistor structure and its radiation tolerance was intact up to 40 Mrad(SiO_2) [SAMI04].

The third method in which to make CMOS IC's radiation tolerant is to use special circuit architectures that are less sensitive than others to the changes in the device characteristics due to radiation. Several methods can be employed to harden a circuit for TID induced effects. Modeling the radiation-induced variation as a function of the total dose of several transistor parameters, such as the trans-conductance and the threshold voltage, allows to foresee the drift of the circuit operating point, and therefore one can design the circuit in order to make it flexible enough to tolerate these drifts.

Using special circuit architectures is also an effective way to obtain SEU hardening (design hardening) [MCCO99]. The basic idea is to provide memory elements with an appropriate feedback devoted to restoring data when corrupted by an ion hit. Radiation Hardened By Design (RHBD) methodology is described in [ALEX96].

Using radiation shields, typically of a tungsten/copper alloy, is another choice. They can either be built into the package structure, or be attached to the top and bottom [MA89]. While they are effective in reducing the electron component of the total dose radiation, they are much less effective in lessening the proton radiation.

Conceptually, the radiation induced oxide charge buildup problem is a simple principle. It is only when one tries to quantify the details of the radiation response that one realizes the complexities involved in the radiation response of the MOS transistor. For example, the radiation response of a MOS transistor has a very complex time-dependant response which is not only important to understand the physics of the response, but also for the practical issues of testing, predicting and hardness assurance [OLD99].

2.8 Radiation Effects in FPGA's

A **field-programmable gate array** is a semiconductor device containing programmable logic components and programmable interconnects. It is the densest and most advanved programmable logic device. The FPGA allows a designer to implement large digital designs with relative ease at any time and location.

There is increasing interest in the use of FPGAs for many space-based computing operations. FPGAs are generally slower than their ASIC counterparts, can't handle as complex a design, and draw more power. However, they have several advantages such as a shorter time to market, ability to re-program in the field to correct errors, and lower engineering costs. Since this is ideal for spacecraft applications, the space community has actively evaluated radiation effects for most new FPGAs being introduced. While FPGAs offer several benefits for space-based electronics, they are sensitive to TID as well as SEE [WHIR03].

2.8.1 FPGA Architectures

FPGA's typically consists of multiple copies of a basic programmable logic element (LE) or logic blocks, Fig. 2.13. The logic element can implement a network of several logic gates that can then feed into 1 or two flip-flops. Logic elements are arranged in a column or matrix on the chip. To perform more complex operations, logic elements can be connected to other logic elements on the chip using a programmable interconnection network (Switching Architecture) [HAMB03].

A key aspect in the design of an FPGA is its switching architecture, which comprises the resources that are used to interconnect the device's logic blocks. There are three different FPGA switch technologies, SRAM, Flash and Antifuse.

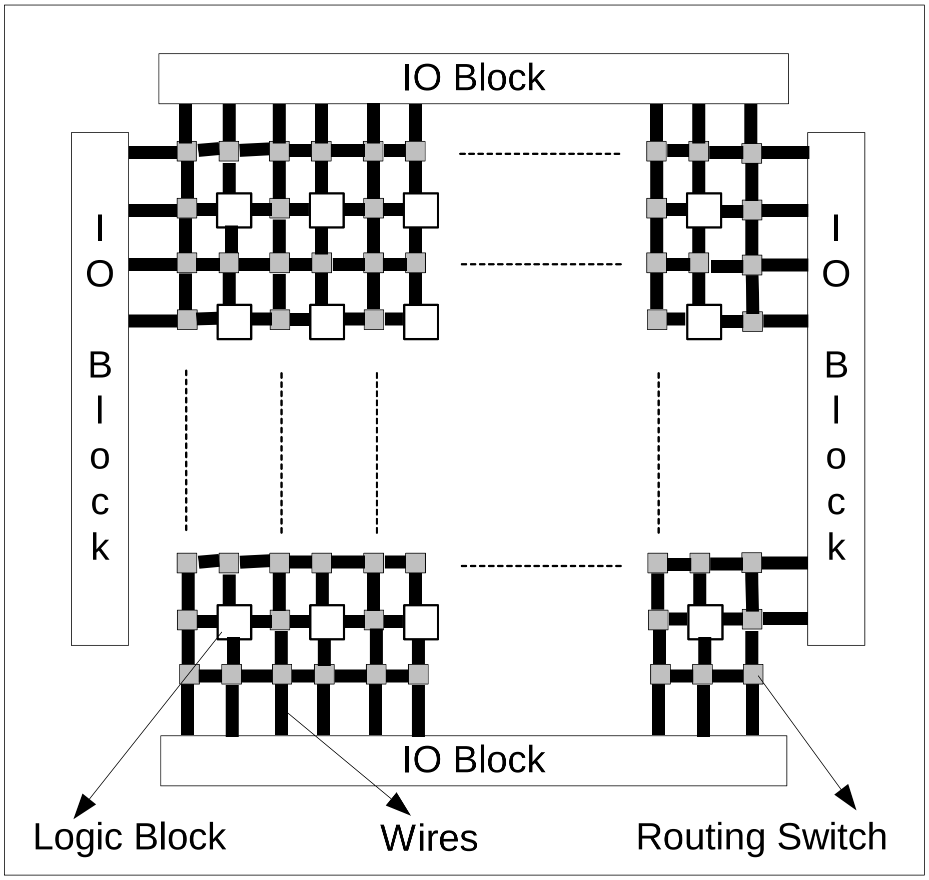


Fig 2.13 General FPGA Architecture

2.8.1.1 SRAM Based FPGA's

SRAM-based FPGA's are increasingly being utilized for satellite and deep space applications [FABU00]. The advantages of these types of devices in these applications are numerous and well known, including the ability to create standard multi-platform application modules, the ability to re-configure the architecture on orbit or in space in response to changing mission requirements, the ability to make last minute design changes and the reduced time from design to flight.

A simplified version of the FPGA Logic Block is illustrated in Fig. 2.14.

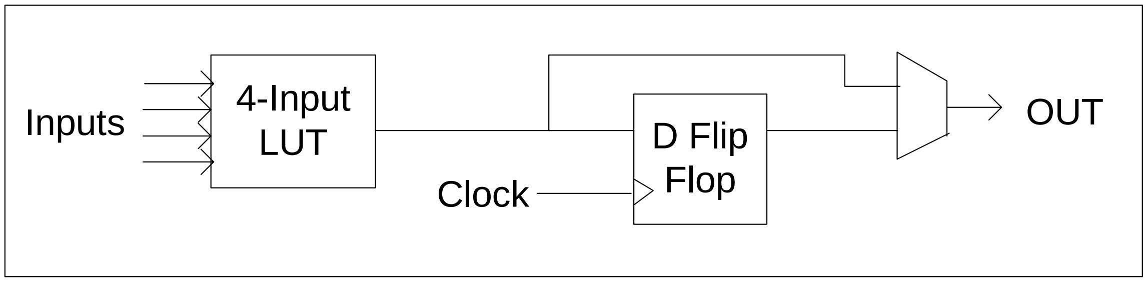


Fig 2.14 FPGA Logic Block

The Logic Block consists of a 4 input look-up table (LUT), a flip flop and a multiplexer. There is one output which can either be the registered or unregistered depending on the multiplexer output [BETZ05].

Each logic block input pin can connect to any one of the wiring segments in the channel adjacent to it. Each logic block output pin can connect to any of the wiring segments in the channels adjacent to it.

Similarly, an I/O block can connect to any one of the wiring segments in the channel adjacent to it. For example, an I/O block at the top of the chip can connect to any of the W wires (where W is the channel width) in the horizontal channel immediately below it.

The FPGA routing can be unsegmented or segmented. In unsegmented routing, each wiring segment spans only one logic block before it terminates in a switch or routing box. By turning on some of the programmable switches within a switch box, longer paths can be constructed. In segmented routing, row and column channels spans the entire device.

Whenever a vertical and a horizontal channel intersect there is a *switch or routing box*. In this architecture, when a wire enters a switch box, there are programmable switches that allow it to connect to other wires in adjacent channel segments. The pattern, or topology, of switches used in this architecture is the planar or domain-based switch box topology. In this switch box topology, a wire in track number one connects only to wires in track number one in adjacent channel segments, wires in track number 2 connect only to other wires in track number 2 and so on. The Fig. 2.15 illustrates the connections in a routing box.

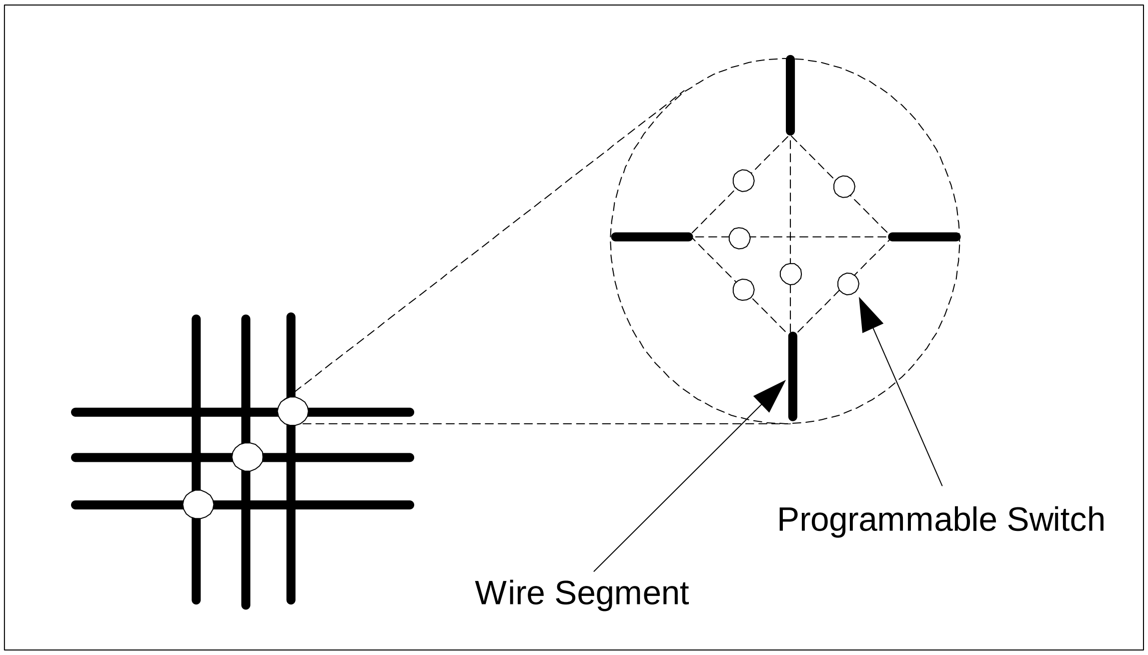


Fig 2.15 Routing Switch Topology

An example of how SRAM cells could be used to implement the routing switch in this type of FPGA is illustrated in Fig. 2.16. The SRAM cell controls a single NMOS pass-transistor.

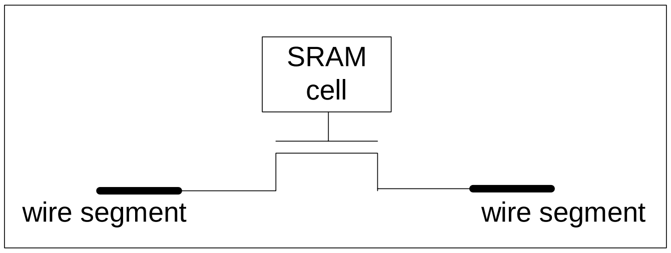


Fig. 2.16 SRAM Based Routing Switches using pass-transistors

The routing architecture above *is not* the routing architecture that is actually implemented by SRAM-based FPGAs. As shown below Fig. 2.17, commercial SRAM-based FPGAs normally place a buffer between routing tracks and the input pins to which they can connect to enhance speed. As well, to save area, the connection from routing wire segment to input pin is made via a multiplexer, not via a set of independent pass transistors. The select lines of the multiplexer are controlled by SRAM cells. It is possible to connect a logic block *output* pin to multiple wire segments in commercial FPGAs.

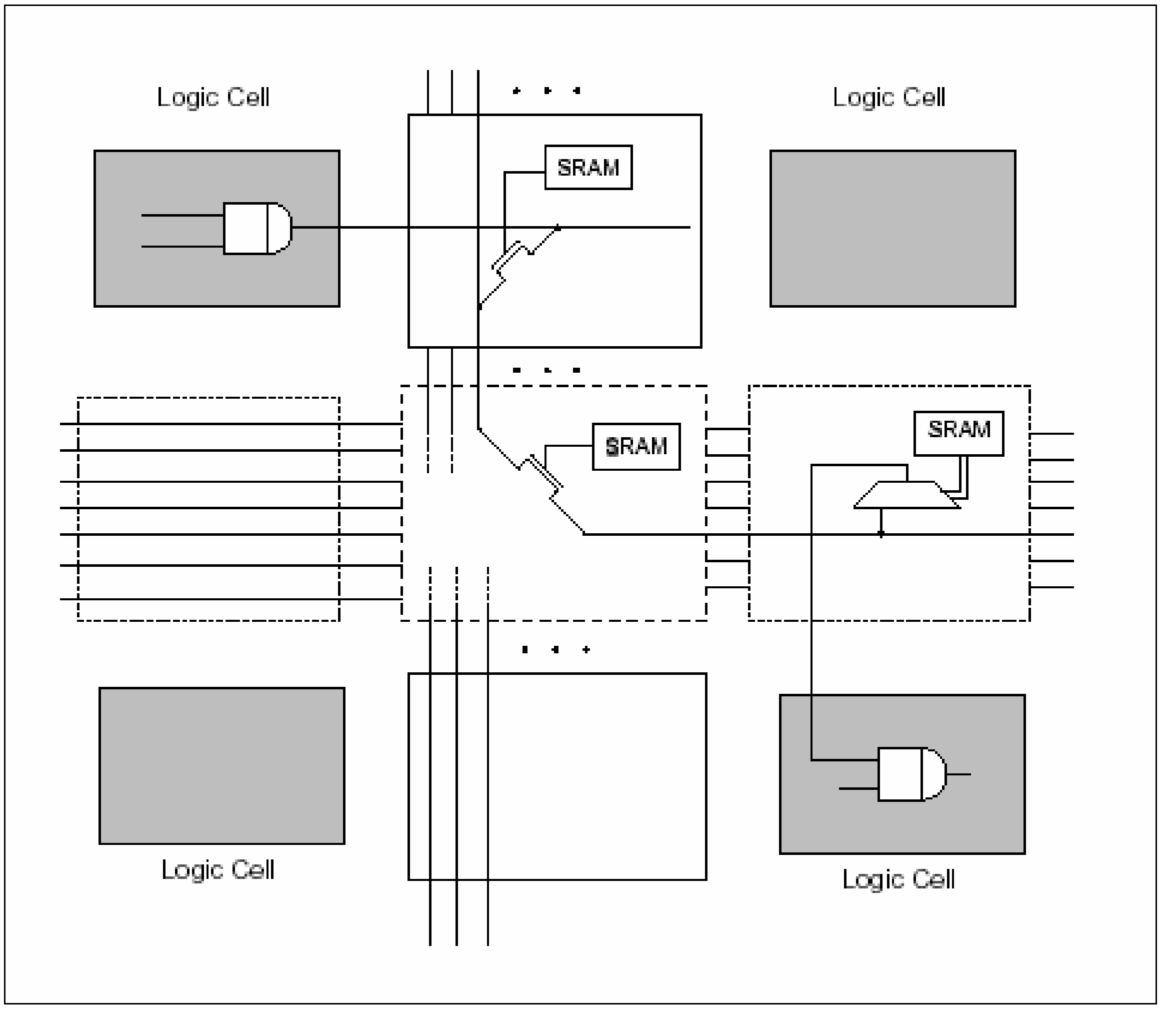


Fig. 2.17 SRAM Based Routing Switches using multiplexers

2.8.1.2 Antifuse Based FPGA's

The other type of programmable switch used in FPGAs is the antifuse. Antifuses are originally open-circuits and take on low resistance only when programmed. Antifuses are suitable for FPGAs because they can be built using modified CMOS technology [BROW97]. As an example, Actel's antifuse structure, known as PLICE, is depicted in Fig. 2.18. The figure shows that an antifuse is positioned between two interconnect wires and physically consists of three sandwiched layers: the top and bottom layers are conductors, and the middle layer is an insulator. When unprogrammed, the insulator isolates the top and bottom layers, but when programmed the insulator changes to become a low-resistance link. PLICE uses Poly-Si and n+ diffusion as conductors and ONO as an insulator, but other antifuses rely on metal for conductors, with amorphous silicon as the middle layer.

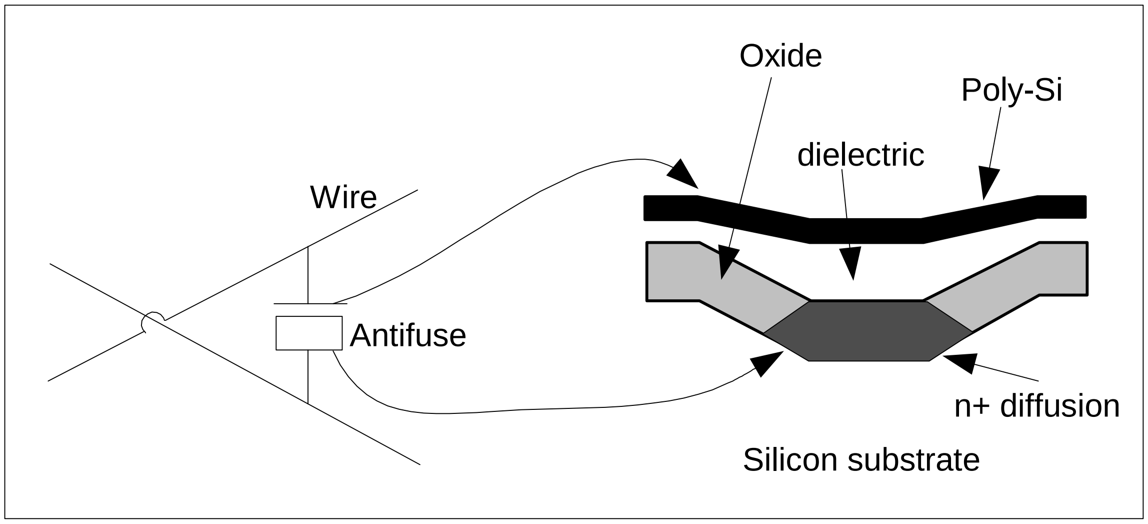


Fig 2.18 The Actel Antifuse Structure

2.8.1.3 Flash Based FPGA's

This section gives a brief description of the structure and operation of the floating gate switch as shown in Fig. 2.19 [WANG04]. The switch element consists of two floating gate NMOS-transistors: A switch transistor turns on-or-off the data path, and a program/sense transistor programs the floating gate voltage and senses the current during threshold voltage measurement. These two transistors share the same control gate and the same floating gate. The modulation of the threshold voltage enables turn-on/off of the switch transistor. The threshold is determined by the charge stored in the floating gate. Fowler-Nordheim tunneling through the thin gate oxide is the mechanism that modulates the stored charge during programming and erasing. The floating gate switch is "programmed" to a low threshold state to turn the switch on, and "erased" to a high threshold state to turn it off. Fig. 2.20 shows the structure of the floating gate transistor. It is an NMOS transistor with a stacked gate. Between the Si substrate and floating gate is the tunnel oxide, and between the floating gate and control gate is the inter-poly oxide-nitrideoxide (ONO) composite dielectric.

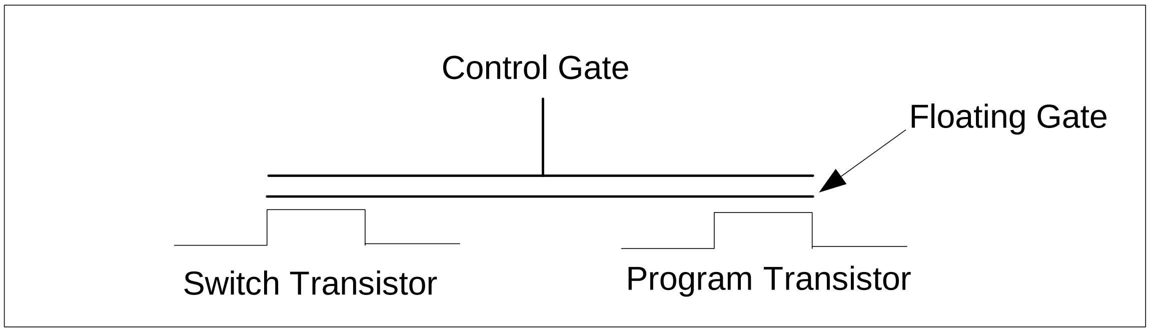


Fig 2.19 Schematic of the physical structure of the floating gate switch [SAMI04]

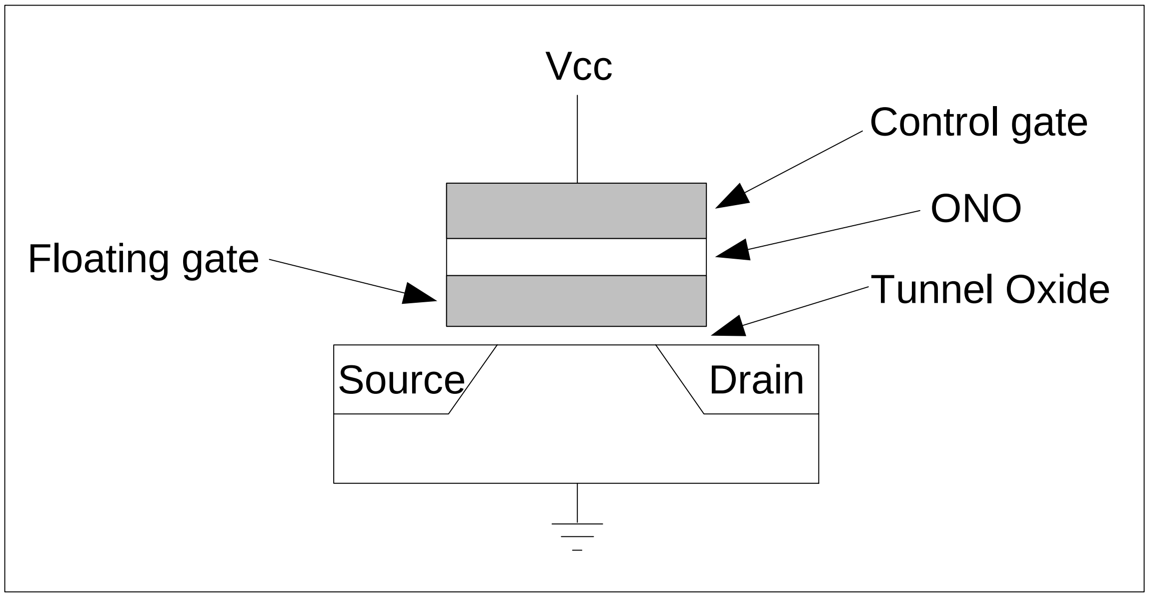


Fig. 2.20 Flash transistor

2.8.2 SEE Effects in FPGA

Single-event upsets in the FPGA affect the user design flip- flops, the FPGA configuration bitstream, and any hidden FPGA registers, latches, or internal state. Configuration bitstream upsets are especially important because such upsets affect both the state and operation of the design. Configuration upsets may perturb the routing resources and logic functions in a way that changes the operation of the circuit. The effects of SEUs in the device configuration memory are not limited to modifications in the memory elements, but they may also produce modifications in the interconnections inside CLBs and among different CLBs, thus giving rise to totally different circuits from those intended [BELL04].

Triple Module Redundancy (TMR) is often exploited for hardening digital logic against SEUs in safety-critical applications. As an instance, TMR is often exploited to design fault-tolerant memory elements to be employed in sequential digital logic. Unfortunately, non radiation-hardened FPGAs present insufficient protection of memory elements in both the mapped circuit, and the configuration memory. As a result, particles hitting the configuration memory can change dramatically the logic functionally of the mapped circuit, as well as the circuit's memory elements. Techniques are therefore required to evaluate the impact of SEUs affecting FPGAs configuration memory, and to avoid undesired changes of the circuit mapped on the FPGA [STER].

2.8.3 Impact of FPGA architecture on radiation response

The differences in the radiation response in different FPGA technologies originate in the switches [WANG04]. The Antifuse FPGA routing switch is completely immune to TID effects, and its sensitivity is only determined by its logic part. This is intuitive correct if one considers that the Antifuse switch structure consists of an insulator isolating the top and bottom layers, but when programmed the insulator changes to become a low-resistance link. Therefore, when programmed the switch basically becomes metal or one routing wire touching another routing wire.

The SRAM FPGA's on the other hand consists of SRAM memory cells comprising the configuration memory of the device. Therefore, compared to an Antifuse FPGA, its sensitivity is increased because of the added effects on the SRAM switches. The TID sensitivity of Flash Based FPGA's will likely be determined by the floating gate switches [WANG04, SNYD89].

The non-volatile antifuse and Flash switches are insensitive to SEE. The logic modules thus determine the sensitivity of the device. SRAM-based FPGA has the biggest disadvantage in that its switch is very sensitive to the SEU. For example, even in real time operation, cosmic-neutron induced soft errors in the SRAM switches can be detected at a typical ground location anywhere.

Hardwired SEU hardenings of non-volatile switch based FPGAs are economically viable because only the logic modules need to be hardened. SRAM-based is difficult to be SEU hardened by hardware solutions. So far, there is no solution without very expensive area penalties. Some software mitigation techniques were proposed and used. However, due to the complexity of the SEU effects on the SRAM-based FPGA, its understanding and subsequent hardening are still open for research at this moment.

Chapter 3
Switched Modular Redundancy

3.1 Introduction

The reader may have noticed in the previous chapter that the author highlighted sentences where the electric field is mentioned. This was done deliberately, because the effect that the gate bias or electric field across the MOS capacitor, has on the radiation response of the MOS oxide, is a very import matter which will be considered more closely in this chapter and forms the basis for the novel Total Ionizing Dose mitigation technique, called Switched Modular Redundancy.

3.2 The Effect of Gate Bias on the MOS Radiation Response

This section forms the theoretical basis for the novel modular redundancy method. It is a well known fact in the physics that a charged particle is accelerated in the presence of an electric field. However, in a solid, electrons will move around randomly in the absence of an applied electric field. Therefore if one averages the movement over time there will be no overall motion of charge carriers in any particular direction. On applying an electric field charge carriers will on average move in a direction aligned with the electric field, with positive charge carriers such as holes moving in the direction of field, and negative charge carriers moving in the opposite direction. In process 1 of Fig 2.4, **if we apply a zero bias to the gate terminal in the presence of ionizing radiation, both the free electrons and holes will remain near their point of origin, and therefore have a greater probability of recombination.**

Further, since no electric field is present, the holes will not be transported toward the silicon-oxide interface for a NMOS transistor and visa versa for the PMOS transistor (process 2 of Fig. 2.4). Hence, the holes will remain near their point of origin. **If the**

radiation field is removed without a gate bias being applied, the effect of the radiation field on the response on the MOS oxide should be minimal.

Several studies have been done that considers the effect of the gate bias on the radiation response of MOS oxides. The amount of threshold shift in MOS devices caused by ionizing radiation is strongly dependant on the bias voltage applied to the gate both during and after radiation. Further, it has been reported that the trapped positive charge near the oxide-silicon interface anneal quickly when irradiated in an unbiased condition [STAN85].

The research by [STAN85] showed that the effect of alternating bias on the radiation response of MOS devices were a **reduced** amount of hole trapping and interface state buildup in N-channel devices. In P-channel devices a reduced amount of hole trapping were also evident.

[OKAB90] studied the effects of high frequency ac bias on the response of MOS devices due to ionizing radiation. Radiation induced interface traps were annealed out **during** irradiation and post-irradiation annealing when an ac bias was applied with a zero offset voltage. Further, the recovery of 40 MHz biased devices agreed with that of 860 MHz biased ones for the same number of alternating cycles of ac gate bias voltage. The authors concluded that the high frequency ac bias was responsible and the total number of cycles may be relevant for the annealing of radiation induced interface traps.

Very large annealing rates have been reported for the total dose damage in a commercial microprocessor by [JOHN83]. The annealing rates had a complex dependence on bias conditions and dose rate. It was reported that these effects can cause large errors in total dose testing procedures.

The effect of bias switching on the growth and annealing of trapped holes and interface states were investigated by [FREI87]. Radiation induced annealing of the trapped charges under zero bias has been observed. The work has confirmed the importance, as reported by [STAN85], of radiation induced annealing.

Since the trapped positive charge near the oxide-silicon interface anneal quickly when irradiated in an unbiased condition, it therefore follows that the threshold voltage shift in MOS devices will be less severe for the gate terminal in an unbiased condition. Thus, for devices which are subject to gate bias cycling, the maximum acceptable dose is higher than if the irradiation bias were applied continuously.

By adding redundancy and applying a resting policy, one can significantly prolong the useful life of MOS components in space. However, a significant buildup of interface states continues during irradiation, even at zero bias [STAN85]. Therefore caution should be applied.

3.3 The Switched Modular Redundancy Method

The fact that the rate of the threshold voltage shift in MOS devices is strongly dependant on the bias voltage applied to the gate terminal is a very important phenomenon that can be exploited, since we have direct control and access to the voltage applied to the gate terminal.

If for example, two identical gates were under the influence of radiation and the gate voltage is alternated between the two, then the two gates should be able to withstand more total dose radiation than using only one gate. This redundancy could be used in a circuit to mitigate for total ionizing dose.

The more a MOS transistor is in use (i.e. switched ON); the more positive charge will be accumulated over time. This implies that gates that are longer in the ON state in a circuit will degrade faster than their idle (OFF) counter parts. Hence “ON” gates will suffer first and cause a circuit malfunction.

Consider Fig. 3.1. When input A is 0, Transistor T1 is in the ON state and T2 in the OFF state. Thus, in this situation, T1 is degrading in the influence of ionizing radiation, and T2 is annealing. This is illustrated in Table 3.1 with an X for degrading and √ for annealing. Therefore, if we have a 50% duty cycle between the two transistors, the circuit should last longer than any other duty cycle. In a real digital circuit, the 50% duty cycle will not be the case. Hence, A might be 0 more often and the result is that T1 will degrade faster. The only requirement for this circuit to malfunction is for one transistor to fail. A solution to this problem would be to add redundancy with an identical module in parallel.

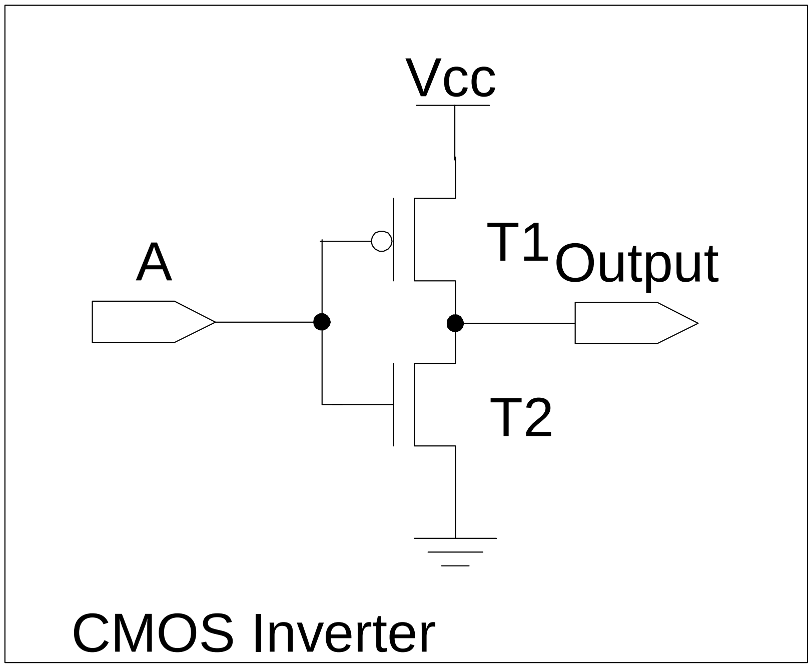


Fig 3.1 CMOS Inverter Circuit

Table 3.1

A	T1	T2
0	X	√
1	√	X

The redundant group consists of two identical components. While the one is in use, the other components inputs are varied between logic 1 and 0. Hence, while the one component is in use, the other component is given a time to anneal.

The same method can be applied similarly to the other fundamental logic gates. This is illustrated in the following diagrams. Functionally, the circuit is still the same as the original circuit, however, it is more radiation tolerant than the original.

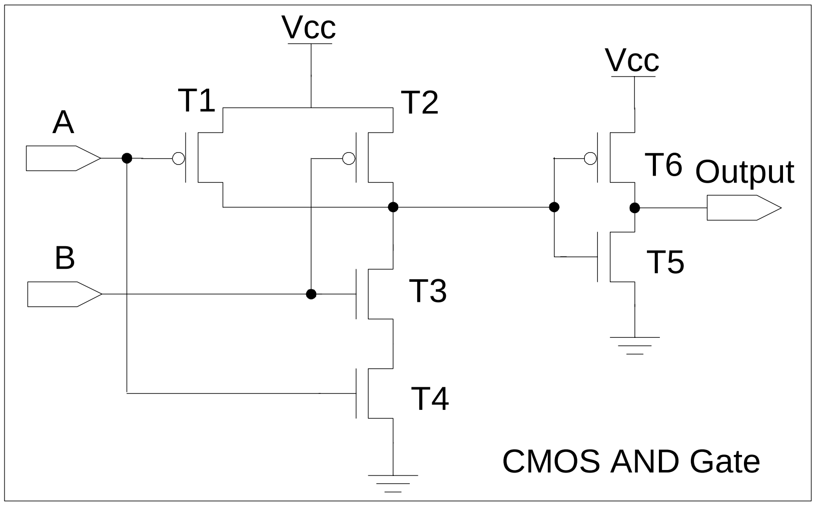


Fig 3.2 CMOS AND Gate Circuit

Table 3.2

A B	T1	T2	T3	T4	T5	T6
0 0	X	X	√	√	X	√
0 1	X	√	X	√	X	√
1 0	√	X	√	X	X	√
1 1	√	√	X	X	√	X

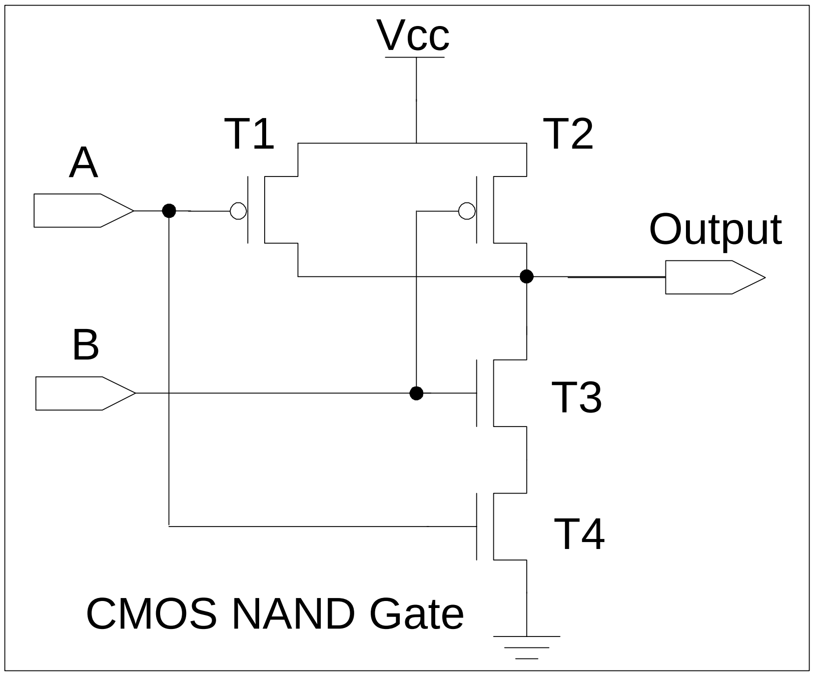


Fig 3.3 The CMOS NAND Gate Circuit

Table 3.3

A B	T1	T2	T3	T4
0 0	X	X	√	√
0 1	X	√	X	√
1 0	√	X	√	X
1 1	√	√	X	X

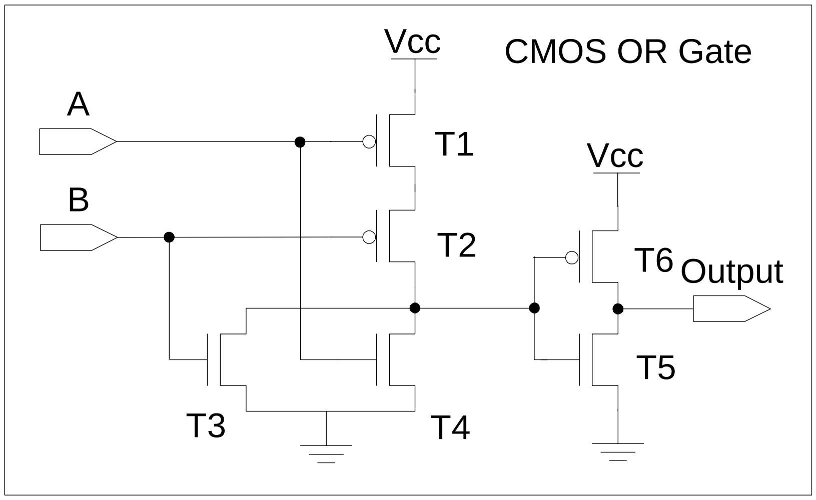


Fig 3.4 The CMOS OR Gate Circuit

Table 3.4

A B	T1	T2	T3	T4	T5	T6
0 0	X	X	√	√	X	√
0 1	X	√	X	√	√	X
1 0	√	X	√	X	√	X
1 1	√	√	X	X	√	X

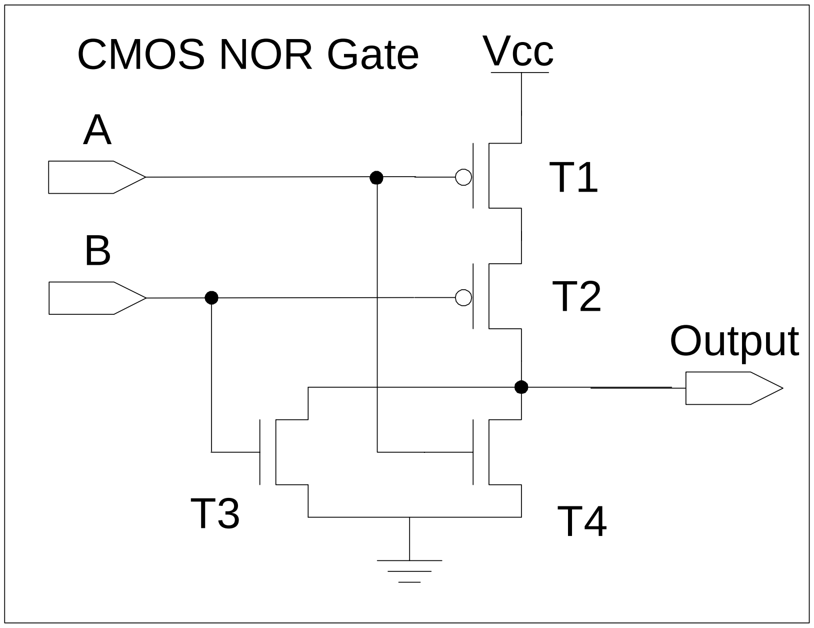


Fig 3.5 The CMOS NOR Gate Circuit

Table 3.5

A B	T1	T2	T3	T4
0 0	X	X	√	√
0 1	X	√	X	√
1 0	√	X	√	X
1 1	√	√	X	X

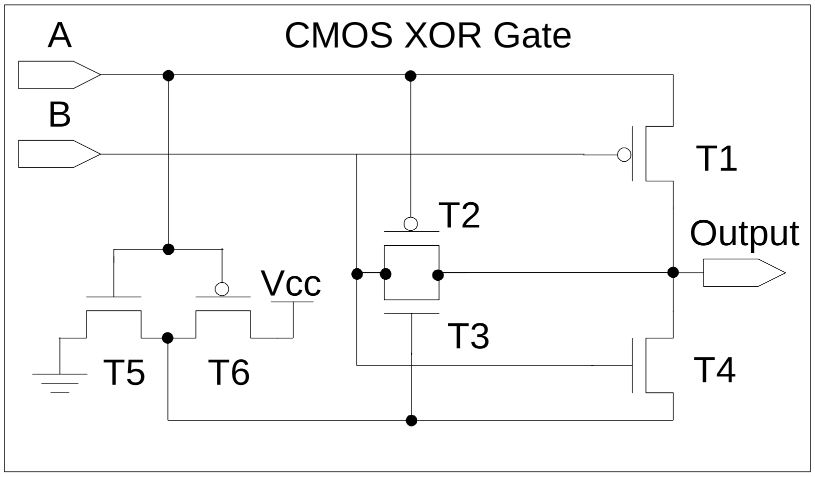


Fig 3.6 The CMOS XOR Gate Circuit

Table 3.6

A B	T1	T2	T3	T4	T5	T6
0 0	X	X	X	√	√	X
0 1	√	X	X	X	√	X
1 0	X	√	√	√	X	√
1 1	√	√	√	X	X	√

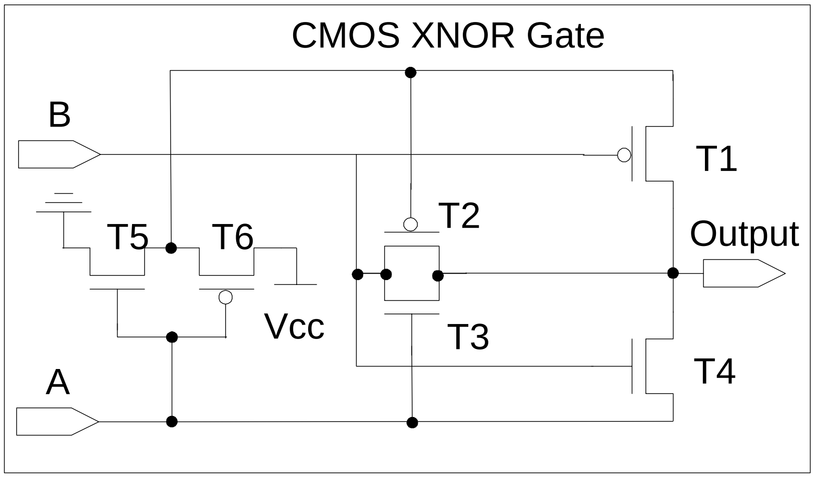


Fig 3.7 The CMOS XNOR Gate Circuit

Table 3.7

A B	T1	T2	T3	T4	T5	T6
0 0	X	√	√	√	√	X
0 1	√	√	√	X	√	X
1 0	X	X	X	√	X	√
1 1	√	X	X	X	X	√

The results from the tables clearly show that in order for the transistors to anneal in each gate, the inputs have to be varied between logic 0 and 1. Hence, both inputs has to be logic 0, thereafter both inputs has to be logic 1.

This concept is best explained by means of an example. Consider Fig 3.2 together with Table 3.2. When both inputs to the circuit is logic 0; T1, T2 and T5 are degrading while T3, T4 and T6 are annealing in the presence of ionizing radiation. When both inputs are logic 1; T1, T2 and T5 are degrading, while T3, T4 and T6 are annealing. However, the time for the transistors to anneal is much shorter than the time it takes to degrade [STAN85, DRES86, SCHW83]. The net results being that the transistors would be able to withstand more total dose radiation.

3.4 Applying the SMR Principle in FPGA’s

As mentioned above, the SMR methodology would be to duplicate each gate in a circuit, then selectively only activating one gate at a time allowing the other to anneal during its off cycle. The SMR algorithm is coded in the “C” language and the code is provided in Appendix D as well as on the accompanying CD.

In the proposed design methodology, the design engineer need not be concerned about radiation effects when describing the hardware implementation in a hardware description language. Instead, the design engineer makes use of conventional design techniques.

When the design is complete, it is synthesized to obtain the gate level netlist in edif format. The edif netlist is converted to structural VHDL code during synthesis. The structural VHDL netlist is fed into the SMR “C” algorithm to obtain the identical redundant circuit components. The resultant file is also a structural VHDL netlist.

The generated VHDL netlist or SMR circuit is then mapped to the FPGA. **However, the SMR algorithm as explained will not provide TID tolerance for an FPGA** implementation as the internal structure of the FPGA is much more complicated. For one, the design engineer has no access to individual gates in an FPGA, and in fact, there are no fundamental gates in FPGA’s. The FPGA consists of logic blocks that are configured to implement a function that represents a fundamental gate. Consider Fig 3.8 which is a representation of the internal architecture of the SRAM FPGA. It consists of the routing the routing matrix, LUT, Flip Flops and output buffers as discussed in the previous chapter.

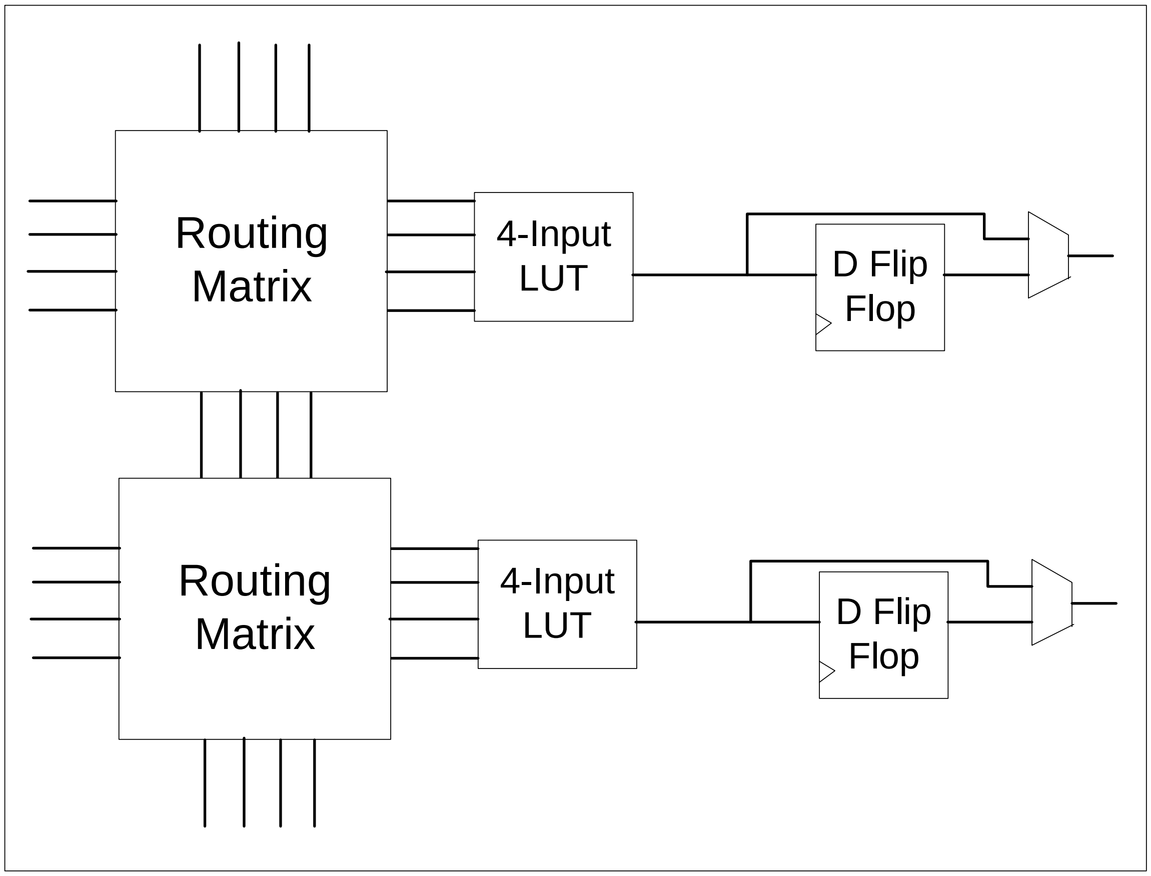


Fig 3.8 Internal architecture of the SRAM FPGA

FPGA devices contain dense arrays of memory cells with a large amount of memory state within a relatively small amount of circuit area. Much like SRAM and DRAM, SRAM-based FPGAs contain large amounts of memory cells within a device and are especially sensitive to radiation. As an example, the Virtex V1000 FPGA contains almost six-million bits of internal state. As suggested in Table 3.8, this relatively large amount of internal state is used for several important purposes.

Table 3.8

Memory Type	No. of Bits	FPGA %
Configuration	5 810 048	97.4 %
Block RAM	131 072	2.2 %
User Flip-Flops	26 112	0.4 %
Total	5 967 232	100 %

User Flip-Flops: An important architectural component of all FPGAs are user programmable flip-flops. User designs exploit these flip-flops to implement common sequential logic circuits such as state machines, counters, and registers. User flip-flops in most digital technologies are susceptible to ionizing radiation because the flip-flops are configured with SRAM bits.

User Memory: Modern FPGAs provide blocks of internal memory larger than the typical look-up table. This block memory is used for traditional random access memory functions such as data storage, buffering, FIFO, etc. For example, the Virtex family includes a set of internal dual-ported BlockRAM memories that each provide 4096-bits of randomly accessible memory. With 32 BlockRAM memories, the Virtex V1000 FPGA offers 131,072 bits of internal memory. Dense static memory such as the BlockRAM is especially susceptible to ionizing radiation.

Configuration Memory: As shown in Table 3.8, 97% of the known memory cells within the Virtex V1000 device are devoted to configuration memory. These memory cells define the operation of the configurable logic blocks, routing resources, input/output blocks, and other programmable FPGA resources. Like other static memory cells, configuration memory is susceptible to ionizing radiation. Errors within the configuration memory are especially troublesome as they may *change* the operation of the circuit. Any spacecraft that utilizes SRAM-based FPGAs must anticipate and mitigate against radiation degradation within the device configuration memory.

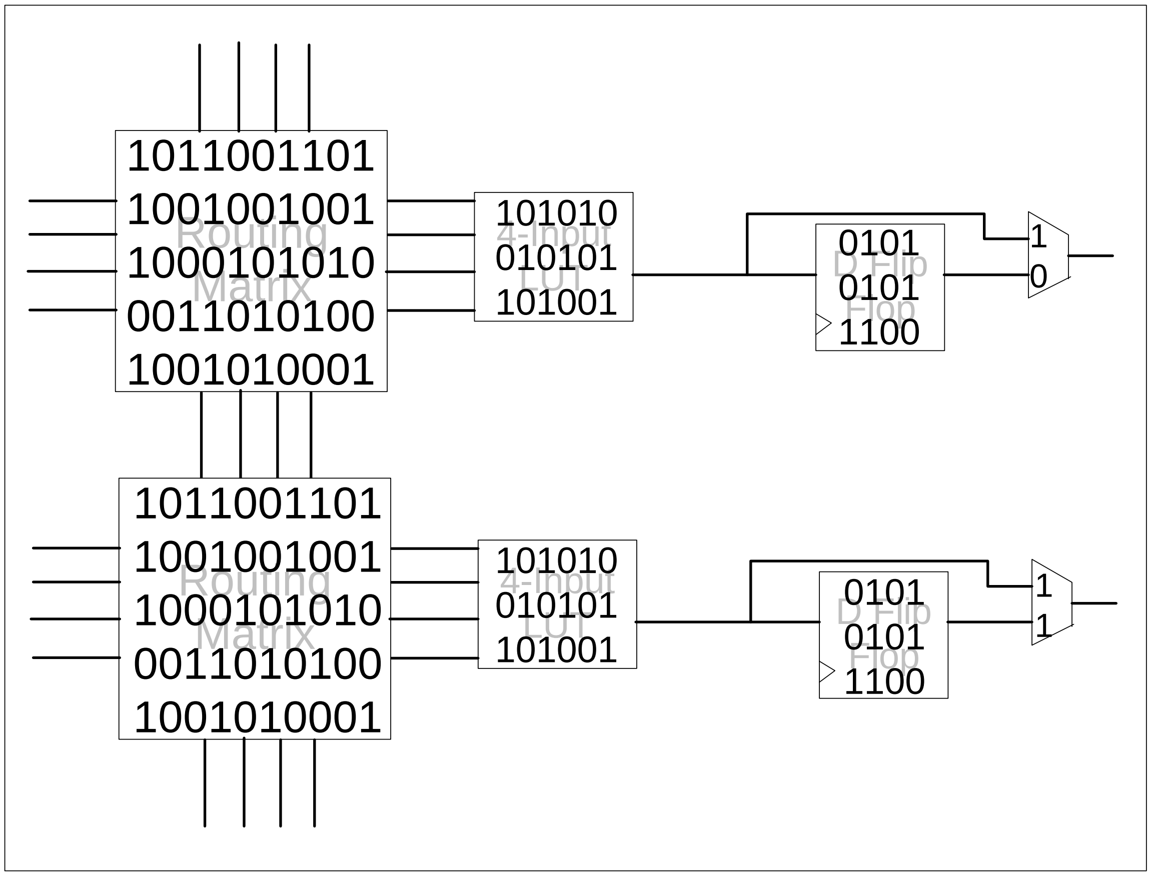


Fig 3.9 Comparative use of SRAM bits in the internal structure of the SRAM FPGA.

Fig 3.9 illustrates the importance and comparative use of SRAM bits in the internal structure of the SRAM FPGA.

As previously mentioned, the SMR algorithm as explained in section 3.3 will not provide TID tolerance for an FPGA. Even if we provided redundancy in the logic part of the FPGA, we still would have a configuration memory that is constant for a particular FPGA implementation. For the SRAM based FPGA as well as the Flash-based FPGA, **switched** redundancy has to be provided in its configuration memory.

When the SRAM and Flash based FPGA is configured, the configuration memory determines the logic functionality of the FPGA. Hence, for a particular SRAM or Flash based FPGA implementation, the configuration memory is constant. Therefore, if we apply the SMR algorithm to an SRAM or Flash based FPGA implementation, **we not only** duplicate the logic part of the FPGA, but as a consequence, we also duplicate the configuration memory.

The configuration memory controls the functionality of the FPGA, and depending on the FPGA architecture, consists of either SRAM or Flash memory cells. However, even if we provided cycled redundancy in the logic part of the SRAM or Flash based FPGA, we still have a constant and static configuration memory that degrades under the influence if ionizing radiation. Thus, for the SRAM or Flash based FPGA, in order to mitigate for TID effects, we also have to provide the cycled redundancy to the configuration memory of the FPGA.

This reduces the problem of TID mitigation for the SRAM and Flash based FPGA to that of reconfigurable computing. This concept can be best explained by means of a diagram. Consider Fig 3.10, which illustrates a simplified depiction of the interconnection between the configuration memory, and the logic part of the FPGA.

If we apply the SMR algorithm to this system, the configuration memory will also be duplicated. Thus, for the SRAM FPGA, we can reset one SRAM cell in the redundant group, while the original circuit is still configured with the other SRAM cell. This basically amounts to in-circuit reconfiguration.

FPGA's can be partially reconfigured to implement Dynamically loadable Hardware Plugin (DHP) modules. A tool called PARBIT has been developed that transforms FPGA configuration bitfiles to enable DHP modules. With this tool it is possible to define a partial reconfigurable area inside the FPGA and download it into a specified region of the FPGA device [EDSO01]. Thus, the above theory can be physically implemented in FPGA's.

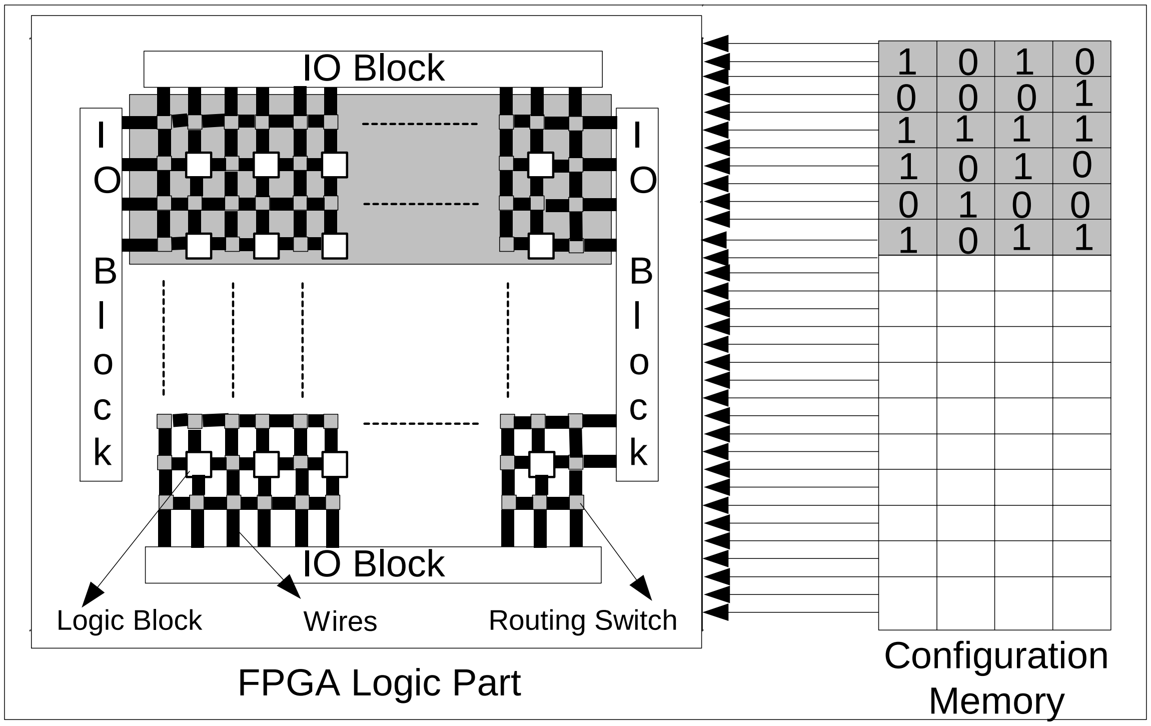


Fig 3.10 Configuration memory and Logic interconnection

Another, and much simpler, way of solving this problem would be to provide no redundancy at all. We can simply load the same circuit into a different part of the configuration memory dynamically, as depicted in Fig 3.11. After some time, the circuit will be reconfigured as in Fig 3.10, and then back to that in Fig 3.11, and so on.

Thus, we define a partial reconfigurable area inside the FPGA and download it into a specified region of the FPGA device. In essence, configuration memory swapping is provided between the duplicated memory cells while the FPGA circuit is in operation.

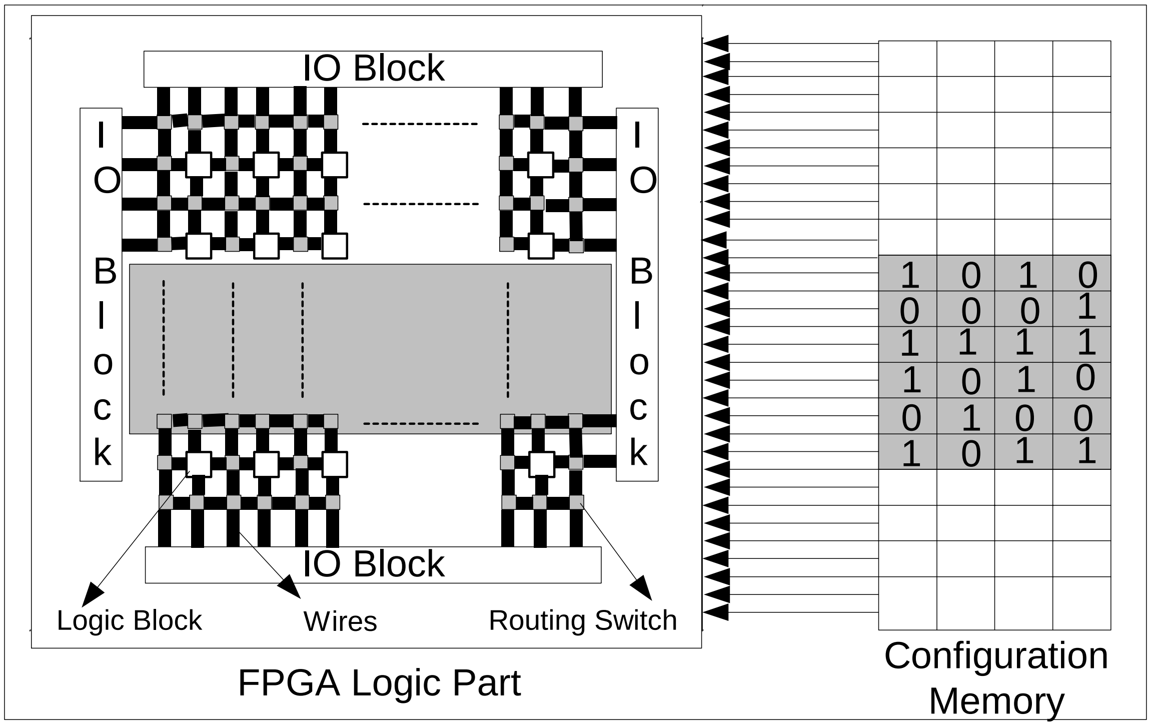


Fig 3.11 Configuration memory swapping

Chapter 4
Experimental Setup and Methodology

4.1 Introduction

The response of MOS devices to radiation is very variable and it is thus impossible at present to use theory alone to predict the device response [HOLM02]. Therefore, testing integrated circuits in a severe radiation environment in advance to their use in operational systems is very important and it will help to reduce the probability of failures in future space applications.

The sensitivity evaluation of a circuit with respect to radiation can be done by:

- the analysis of flight data issued from spacecraft operating in the actual environment, i.e. space projects.
- ground testing,
- fault injection.

Fault injection is normally used to perform SEU testing on electronic circuits, while actual ground tests are performed in order to test the ionizing dose performance of electronic circuits. In this chapter we describe the experimental setup for the ground testing as well as the radiation facility.

4.2 Radiation Source and Facility

The source most often used for ionizing radiation testing and the source used in this study is Co-60, which emits gamma rays (photons) of energy 1.173 and 1.332 MeV. Co-60 is used for industrial irradiations, sterilization, radiotherapy and biological research. The radiation testing for this study was performed utilizing the Co-60 source at the Agriculture Research Council (ARC) in Stellenbosch, Western Cape, South Africa and was readily available.

Ionization due to gamma rays provides useful simulation of the ionization due to the radiation environment of space [HOLM02]. In the radiator facility at the ARC, a cylinder of Co-60 is suspended underground in the radiation room. A diagram of the radiation facility is given in Fig 4.1.

The device under test (DUT) is arranged on a moving tray and is placed near the Co-60 source and their response to the radiation can be monitored continuously by means of the extension wires leading out of the radiation room.

The source at the ARC facility has a radiation dose rate of 2.5 krad/hour (0.7 rad/s) at a distance of 1.3m. Depending on the distance from the source, the radiation dose can be varied, however, testing was performed at a dose rate of 2.5 krad/hour. The dose rate used is considerably larger than that found in space applications, and it can easily be argued that the evolution of the power consumption increase and consequent failure would not at all be the same in a real environment. However, that the radiation response of the CMOS transistor depends on the dose rate is a common misconception. What is true is that the response is time dependent. For example, if one irradiates two samples to 100 krad, one at 10 krad/min for 10 minutes, and the other at 5 krad/min for 20 minutes, then the response at the end of the exposure will probably be different, because damage is annealing during the exposure. If one measures a time dependent quantity at 10 minutes and also at 20 minutes, there is no reason to expect the results to be the same. But for CMOS, if you measure both samples at the same time (20 minutes), the results will be exactly the same, even though the dose rate was different. This was first identified by Derbenwick as an apparent dose rate effect (as opposed to a true dose rate effect) in 1977 [DERB77]. Since then, numerous others have reached the same conclusion. The most elegant experimental demonstration of this idea was by Fleetwood et al. [FLEE88]. They varied exposure rate by 11 orders of magnitude, and annealing time by up to nine orders of magnitude, and found no true dose rate effects. That is, there were no dose rate effects if different annealing times were properly accounted for.

A very recent experimental verification is provided in [SCHW07] where the authors investigated dose rate effects in CMOS ICs at dose rates from 0.2 to 2 x 10^9 rad/s. Radiation-induced degradation of CMOSIIIA 16KSRAMs was dominated by radiation induced charge buildup in gate oxide transistors for dose rates of 100 rad/s and lower. For these devices, laboratory irradiations can be used to estimate device radiation-induced degradation at lower dose rates (e.g., space), because the same mechanism led to radiation-induced degradation.

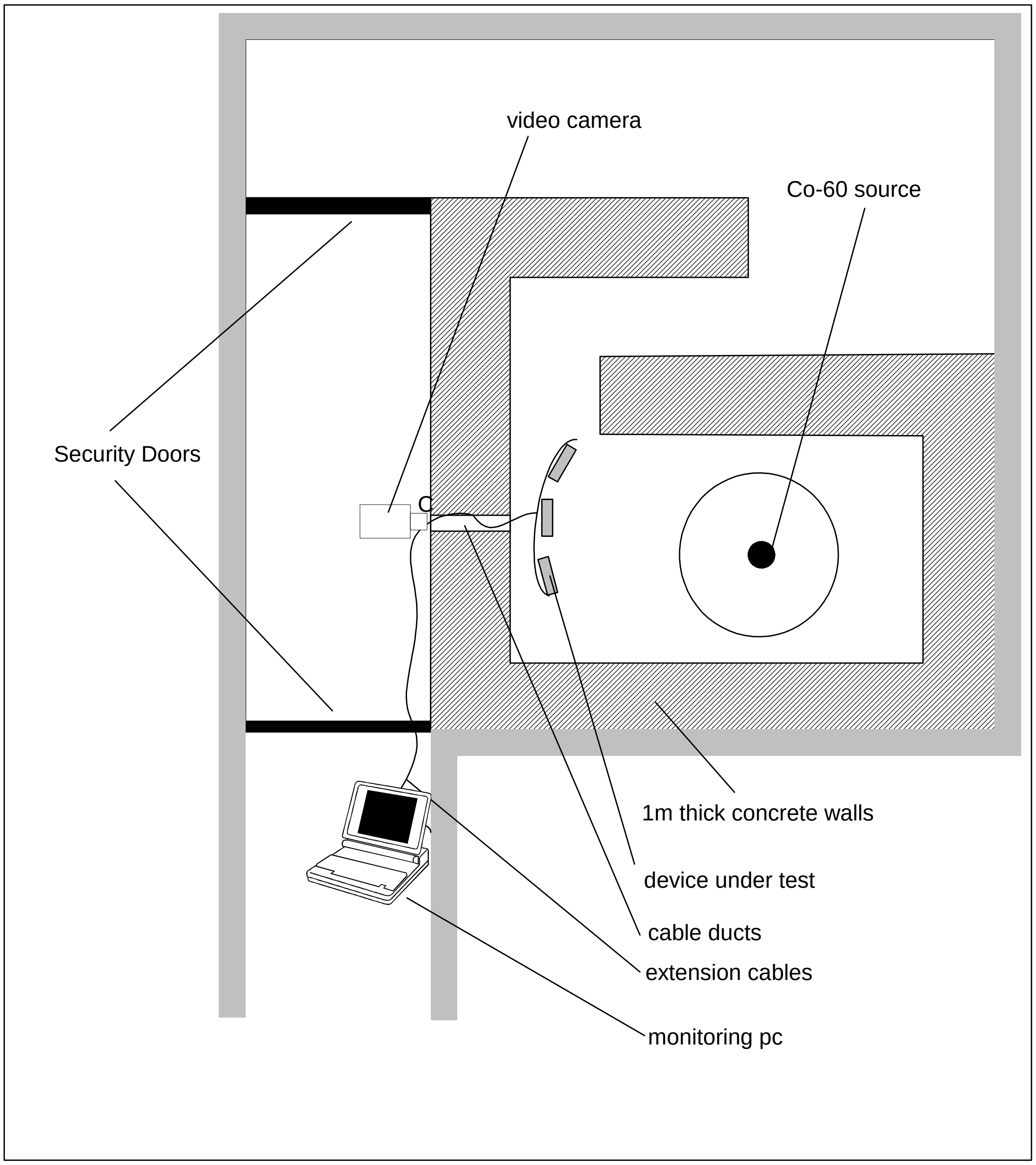


Fig 4.1 Radiation Facility Layout

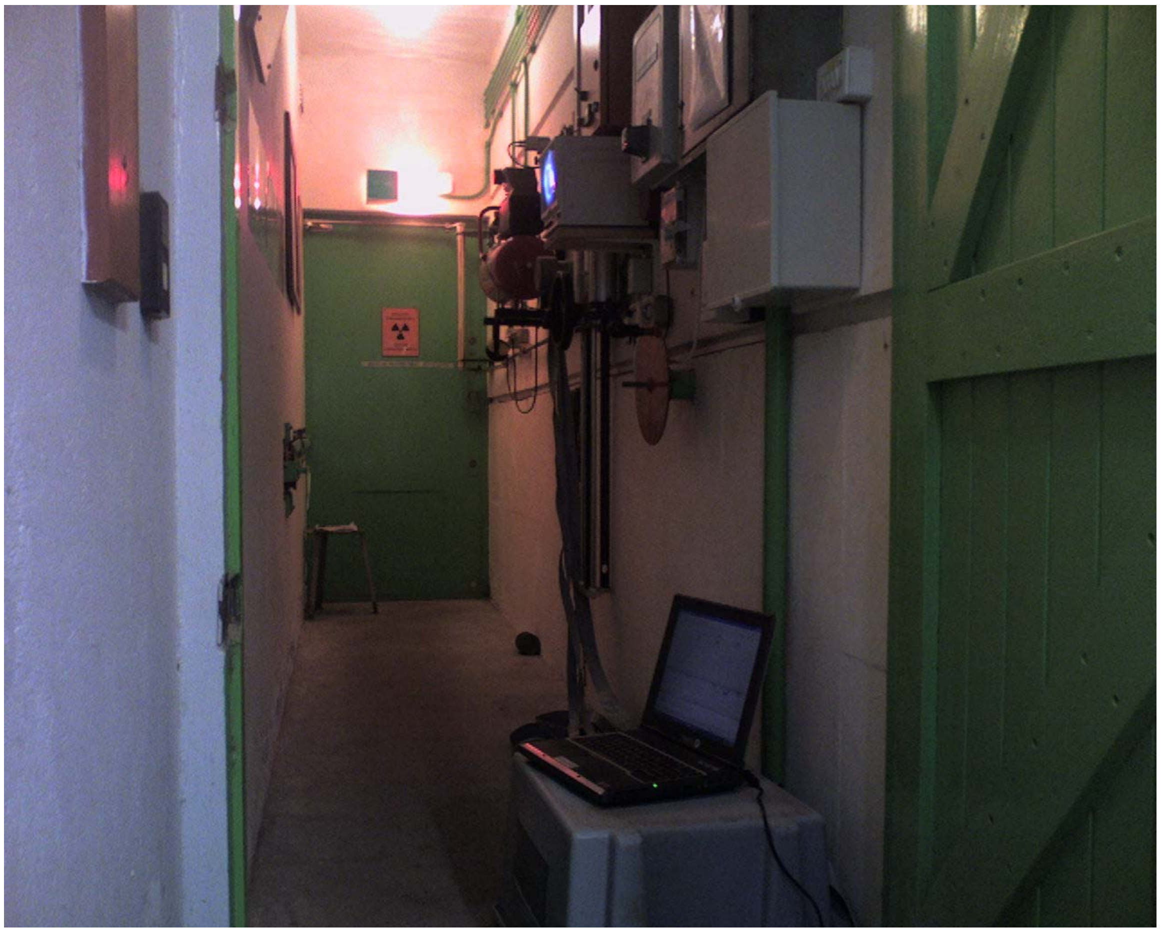


Fig 4.2 Radiation facility entrance

Fig 4.2 shows the entrance to the radiation facility at the ARC. Past the entrance security door in the passage, various monitoring equipment can be seen, as well as the cable ducts. The security camera provides a visual of the radiation room on a monitor. Our monitoring PC connected to the extension cables can also be seen.

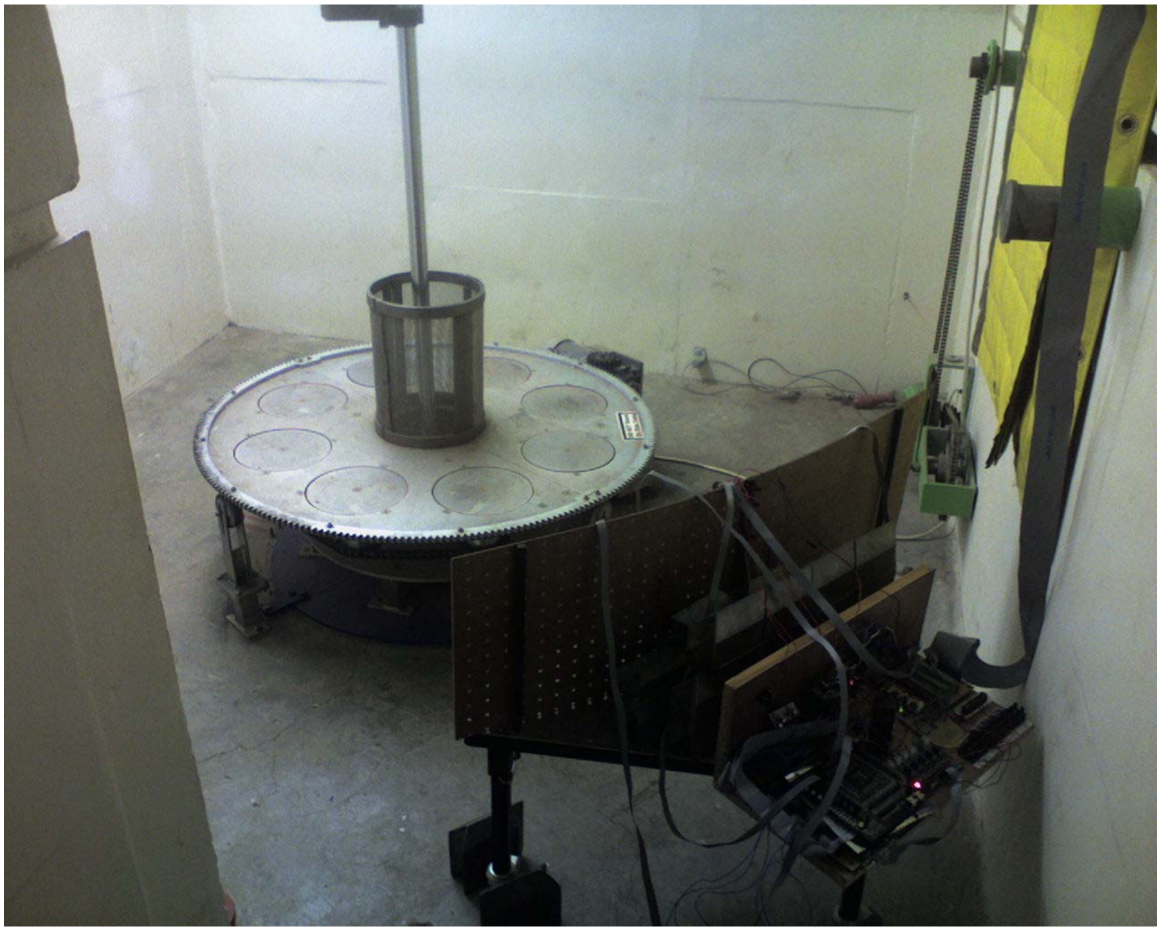


Fig 4.3 Inside the radiation room

Fig 4.3 shows the inside of the radiation room. The Co-60 source is suspended underground via a steel rod in the middle of the room. The DUT is located on a movable trolley in front of the Co-60 source, as indicated in the diagram. The distance from the source determines the radiation dose rate. In our case, the DUT is at a distance of 1.3m which translates into a radiation dose rate of 2.5 krad per hour. Also indicated in Fig 4.3 are the extension cables which link the internal monitoring equipment with the PC outside in the passage via the cable ducts.

Two layers of solid lead bricks of 5cm thickness is located between the back of the trolley and the front plane where the DUT's are suspended. The purpose of the lead bricks is to protect the internal monitoring equipment from radiation damage.

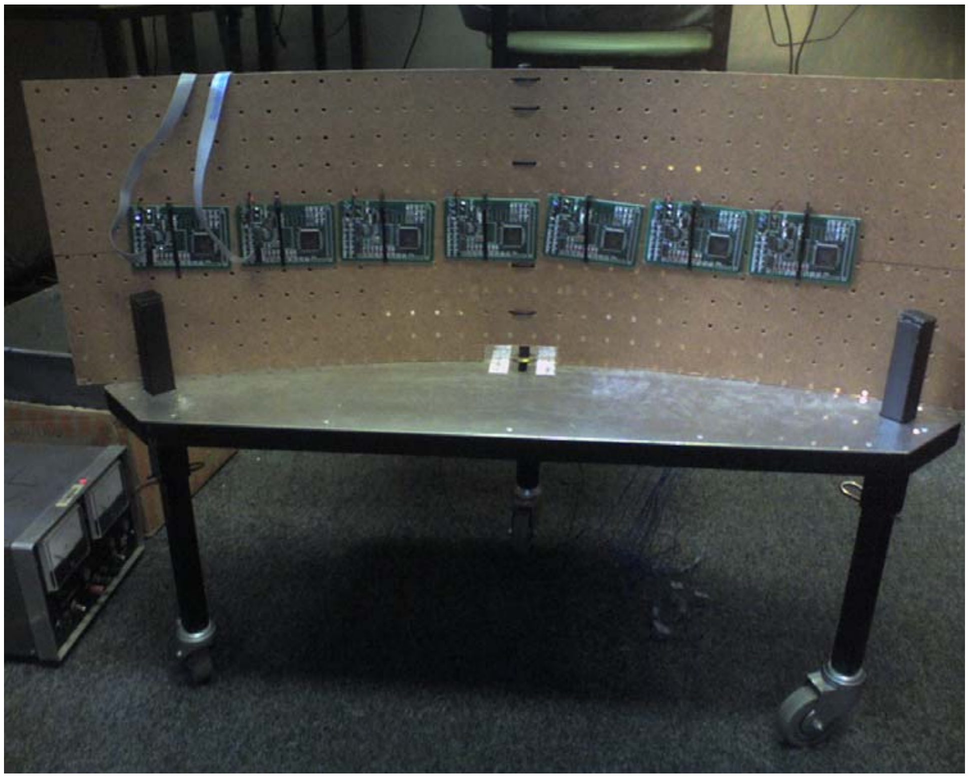


Fig 4.4 Devices under test suspended on the movable trolley

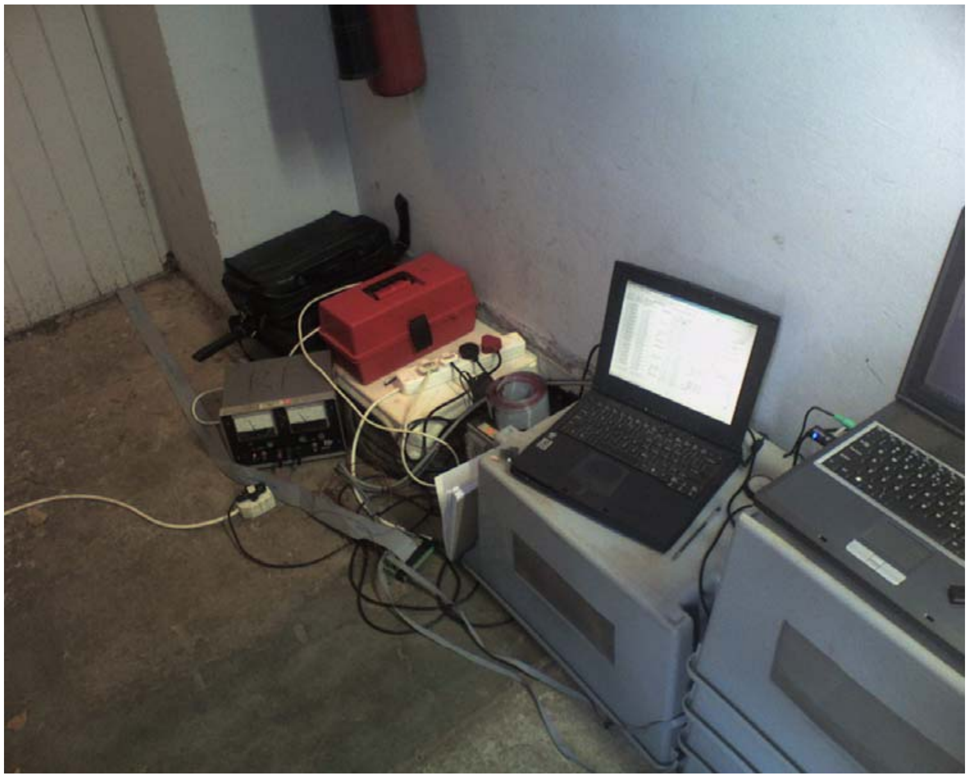


Fig 4.5 Two PC's and power supplies are located outside the radiation facility

4.3 Test Methodology

The test methodology depends on the DUT type. For example, the methodology used for memories consists mainly in to write a data pattern, to wait a loop read out and to compare to expected values. The methodology for processors is more complex. The test can be [BRY98]:

- *Dynamic* - actively exercise a DUT during beam exposure while counting errors, generally by comparing DUT output with a reference device or other expected output. Devices may have several dynamic test modes, such as *Read/Write* or *Program-Only*, depending on their function. Clock speeds may also affect radiation test results.
- *Static* - load device prior to beam irradiation, then retrieve data post-run, counting errors. In this case there is the worst case estimation of the error rate.
- *Biased (SEL only)* - DUT is biased and clocked while I_{CC} (power consumption) is monitored for latch-up or other destructive conditions.

Electronic test equipment for controlling and observing the DUT behavior during its exposition to radiation must be built according to the system and the radiation facility.

Total Ionizing Dose Performance was examined at a dose rate of 0.7 rads (Si)/sec (2.5 kRad/hour). Testing included in-situ monitoring of key parameters such as Icc, as well as full functional test pre- and post-dose. In addition, at various cumulative dose steps, devices were tested for full functional circuit behavior using the specific vendor's comprehensive test programs. Devices were also reconfigured at various dose steps in order to implement the SMR algorithm, as explained in the previous chapter. Variation in the performance of the devices with total absorbed dose is presented in the following chapter, together with the enhanced performance obtained by the SMR method. An illustration of the test setup is shown in Fig 4.6 and Fig 4.7.

The FPGA's are connected to the PC via current sensors, a 12-bit Analogue to Digital

Converter and a microprocessor. A software program runs on the PC that monitors and logs the FPGA power supply currents every 2s. The interface between the FPGA IO blocks and the PC was a FPGA based processor (Actel ProASIC Plus APA075PQ208). The purpose of the microprocessor was to monitor and log the FPGA IO's logic values every 2s. The expected values of the IO ports are stored on the processor and compared to the FPGA IO logic values, if correct a 0 is sent to the PC, if incorrect (i.e. an error), a 1 is sent to the PC.

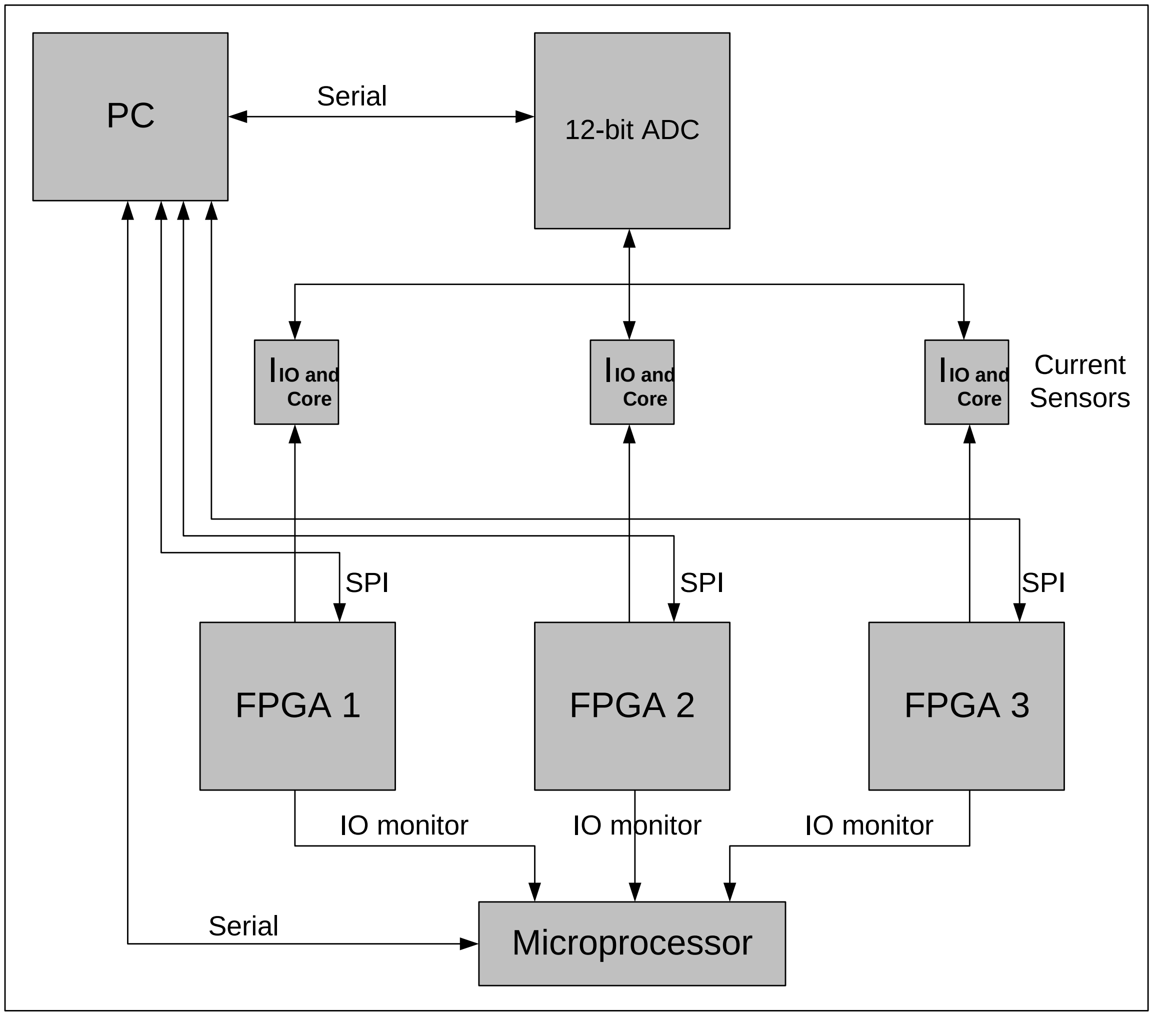


Fig. 4.6 FPGA TID Test setup

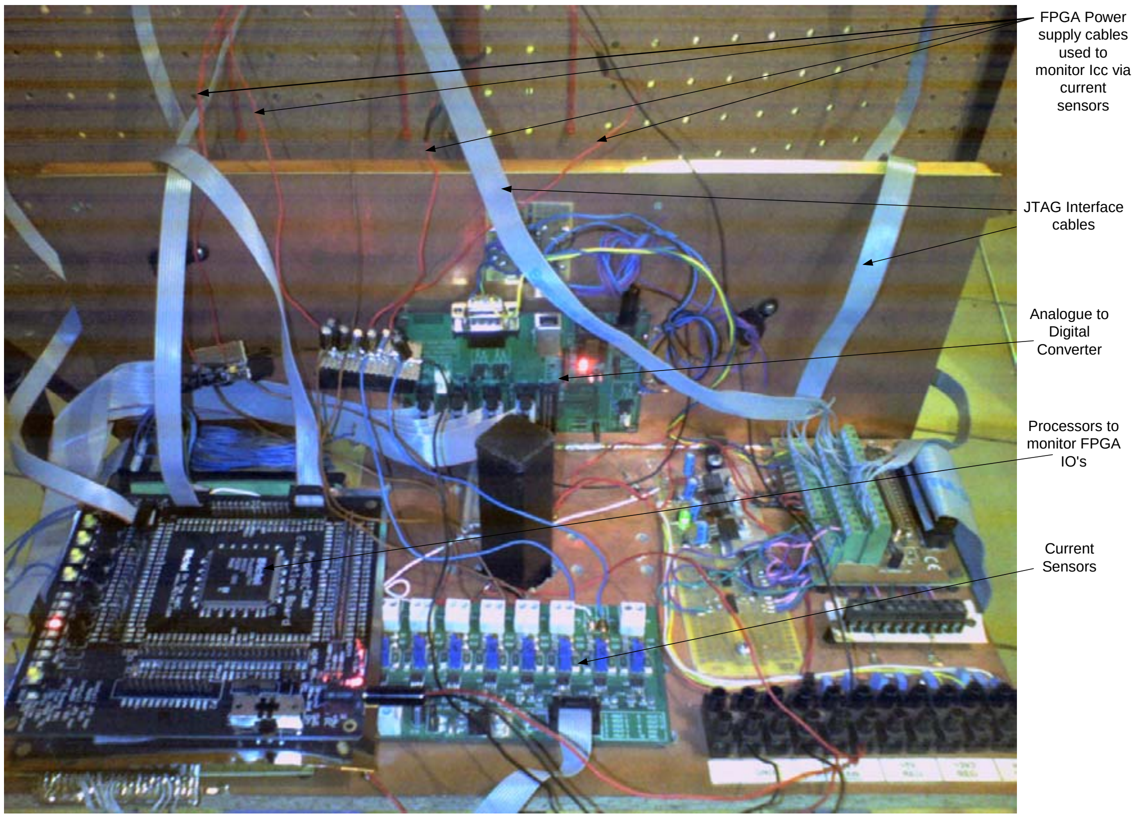

FPGA Power supply cables used to monitor Icc via current sensors
JTAG Interface cables
Analogue to Digital Converter
Processors to monitor FPGA IO's
Current Sensors

Fig 4.7 Monitoring devices at the back of the radiation trolley

In-circuit reconfiguration and JTAG monitoring is provided by means of the Serial Port Interface (SPI). During irradiation testing, the FPGA's were reconfigured; successful reconfiguration is an indication that the configuration memory of the FPGA is still functional. Separate design files were compiled with the Altera Quartus VHDL design software, with each design file representing a different part of the configuration memory.

During reconfiguration, a different design file was loaded into memory. For example, design file 1 would be loaded, after some accumulated dose design file 2 would be loaded, then design file 1 again and so on. No extra functionality is needed; the new design is simply downloaded to the FPGA by means of the SPI.

The following table gives a summary of the parameters that were tested during the TID tests of the FPGA's. However, the Power supply current (ICC) is the most important parameter for monitoring TID effects [WANG04].

Slight annealing at room temperature was observed over night for each test, as the radiation facility was shut down. These anneals resulted in degraded devices returning to a slightly improved performance.

Table 4.1

	Parameter	Logic Function
1	I_{CC}	FPGA Power Supply
2	Configuration	Configuration Memory
3	IO Ports	FPGA functionality

The devices were tested with a voltage regulator (LM7805) on board and without the regulator. It was thought that the regulator's TID tolerance exceeds that of the FPGA and therefore the FPGA could be tested with the regulator on board. However, no difference in the FPGA radiation response was observed with or without the regulator on the PCB board.

4.4 Devices Tested:

SRAM-based FPGA: 10K10TC144-4 CMOS based from Altera with 10,000 to 250,000 typical gates.
Operating voltage: 5V
Configuration: In-circuit reconfigurability (ICR) via a SPI.
Gate Oxide: SiO_2

The FPGA devices were configured with a ring counter code circuit that will be used to configure the device. The circuit is a synchronous digital logic design clocked at 25 MHz. The circuit occupied 8% of the logic elements in the EPF10K10TC144-4 FPGA. The ring counter code is given in appendix B.

4.4.1 Altera Flex10K Architecture

The flexible logic element matrix (FLEX), is a FPGA device with 10 000 gates. The Flex device is configured by loading internal static random access memory (SRAM), and thus looses its configuration memory whenever power to the device is lost [ALTE03]. Thus is a real system, an external low cost serial programmable read only memory (PROM) is normally used to automatically load programming information when the device powers up.

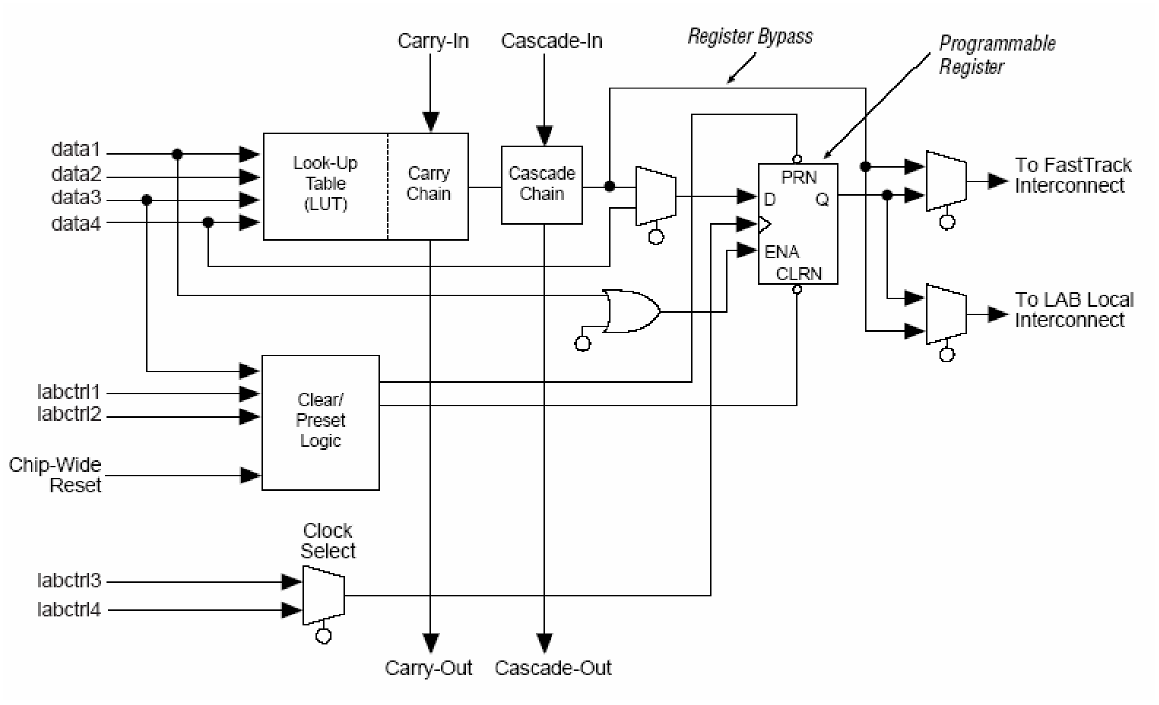


Fig 4.8 Flex 10K Logic Element [ALTE03]

Consider Fig 4.8 which shows the Flex 10K logic element (LE). A logic gate is implemented by making use of LUT's. The LUT is a high speed 16x1 SRAM. Four inputs are used to access the LUT's memory. The truth table for the gate can be loaded in the LUT's SRAM during configuration. A single LUT can thus model any network of gates with 4 inputs and 1 output.

More complex gate networks require interconnections with additional neighboring logic elements. The LUT output is fed into a D flip-flop and then to the interconnection network. The clock, Clear and Preset can be driven by the internal logic or an external Io pin. Carry and Cascade chains connect to all LE's in the same row.

Fig 4.9 shows the logic array block (LAB). A LAB is composed of 8 LE's. Both programmable local LAB and chip-wide row and column interconnects are available. Carry chains are provided to support faster addition operations.

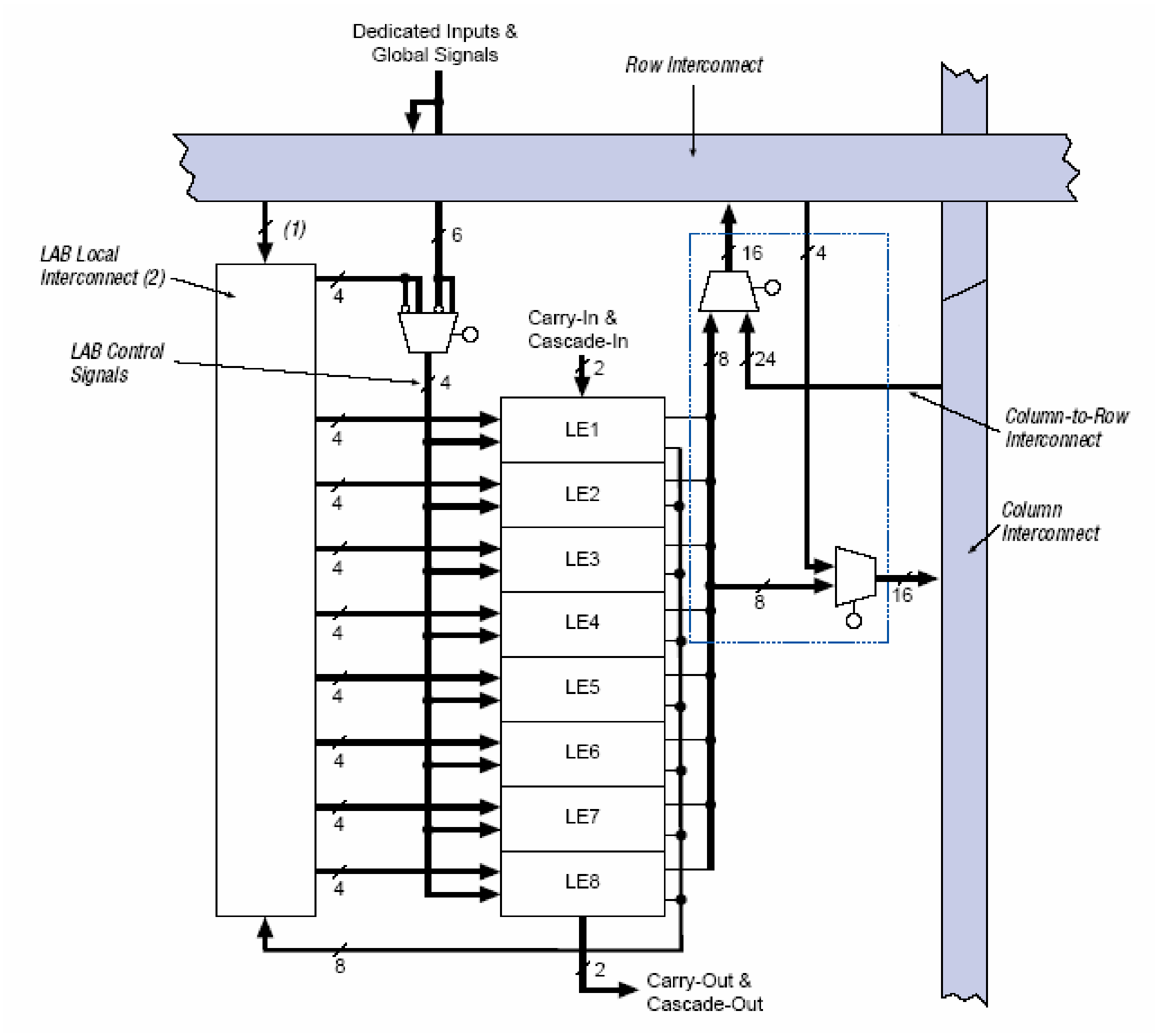


Fig 4.9 Flex 10K logic array block [ALTE03]

Fig 4.10 shows the flex 10K device architecture. A matrix of LAB's and embedded array blocks (EAB) are connected via programmable row and column interconnects. The flex 10K device contains 72 LAB's and 3 EAB's.

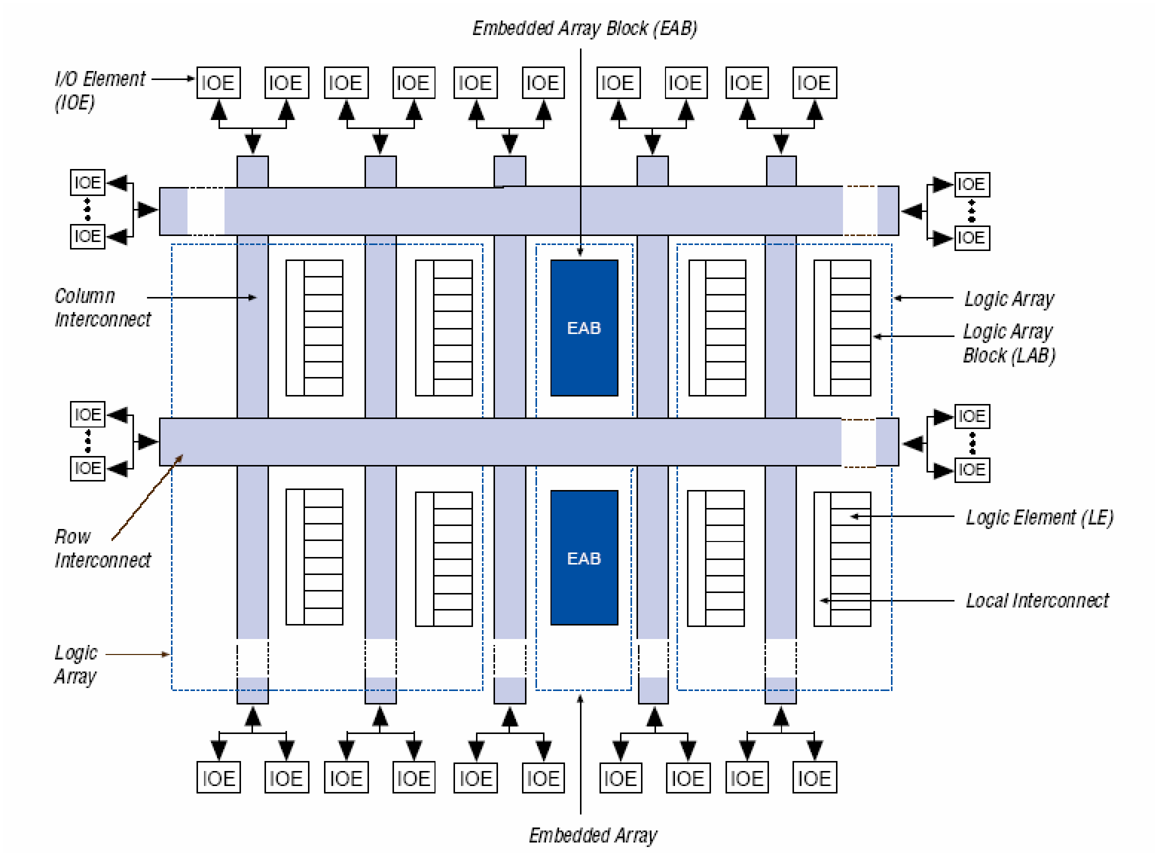


Fig. 4.10 Flex 10K architecture

An EAB contains 2048 bits of memory. Each EAB can be configured as 256x8, 512x4, 1024x2, or 2048x1 SRAM. In some cases, EAB's can be used to implement gate level logic. As an example, a 4x4 multiplier can be implemented by storing the multiply truth table in a single EAB.

Input-output elements (IOE) are located at each of the devices IO pins. IOE's contain a programmable tri-state driver and an optional 1-bit flip-flop register. Each IO pin can be programmed as input, output, output with a tri-state driver, or even tri-state bi-directional with or without a register.

4.4.2 Altera Max Plus II Floorplan Editor

The CAD tool used to program the FPGA was the Altera Max Plus II software with the code written in VHDL as previously stated. A very useful feature of the Max Plus II CAD tool is the Floorplan editor.

The floorplan editor is a visual tool to assist expert users in manually placing and moving portions of logic circuits to different logic cells inside the FPGA. This is normally done in an attempt to achieve faster timing or better utilization of the FPGA. Floorplanning is typically used only on very large designs that contain subsections of hardware with critical high-speed timing [KLUW01].

For non-expert users, use of the Max Plus II compiler's automatic place and route tools is normally used. Automatic place and route is performed during the compile process.

The floorplan editor was used to implement the SMR algorithm as stated in chapter 3. We load the circuit into a different part of the configuration memory dynamically by making use of the floorplan editor.

Fig 4.11 shows a diagram of the floorplan editor. Different versions of the Max Plus II software may place the logic at different locations within the chip. One implementation of the OR-gate design is shown in Fig 4.11.

There is a lot of empty space in the floorplan of Fig 4.11 since the Flex 10K10 can contain up to 10 000 logic gates. Pins and logic cells used in the design are color coded. If you move the logic cell to another location, it will make small changes to the circuit timing because of changes in the interconnect delays inside the FPGA.

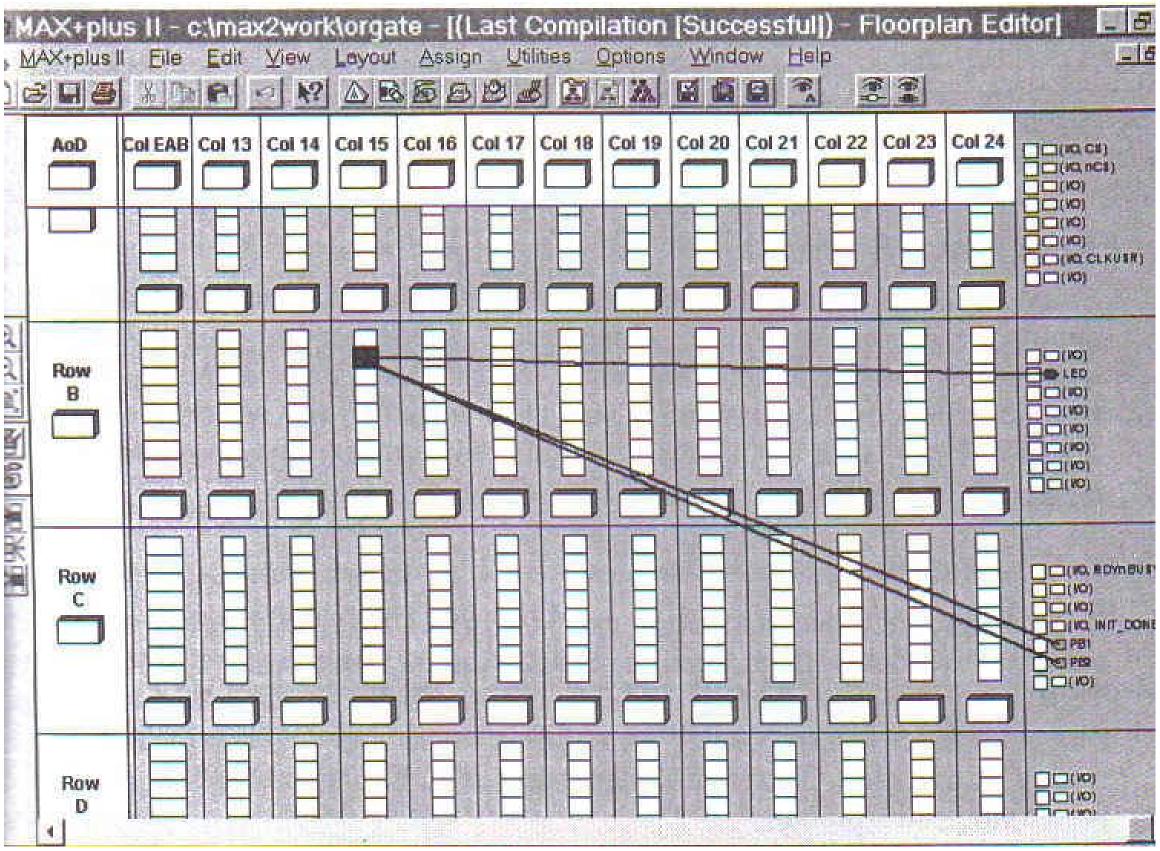


Fig 4.11 Floorplan layout with internal FPGA placement of OR-gate logic cell and IO pins

4.5 Radiation testing PCB boards

The PCB board's that was designed and manufactured for the purpose of radiation testing of the FPGA's is shown in Figures 4.12 and 4.13. These boards were designed for the TID testing of the Altera SRAM FPGA. The schematics can be found in appendix A.

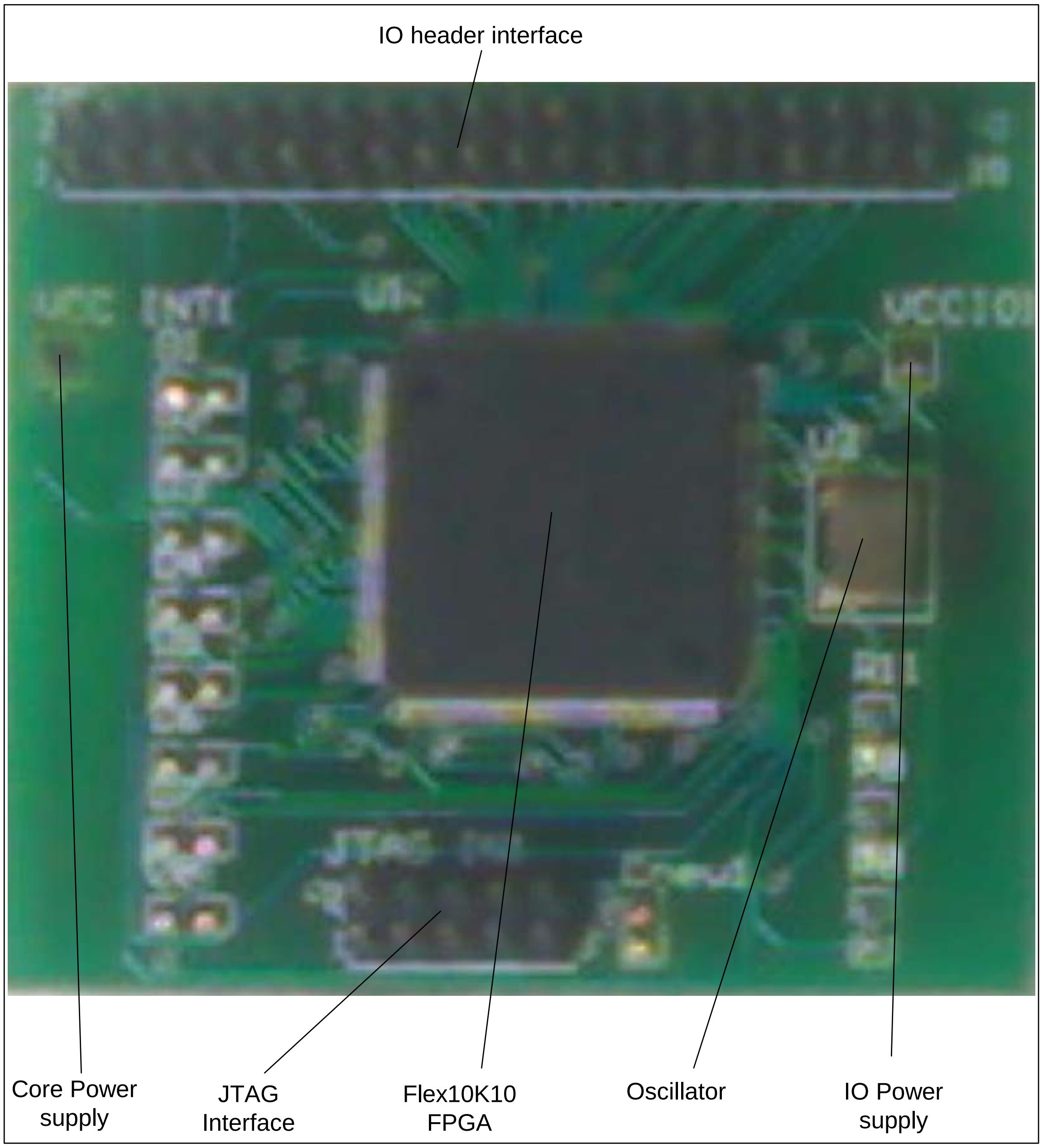


Fig. 4.12 FPGA PCB test board with IO and Core power supply separated

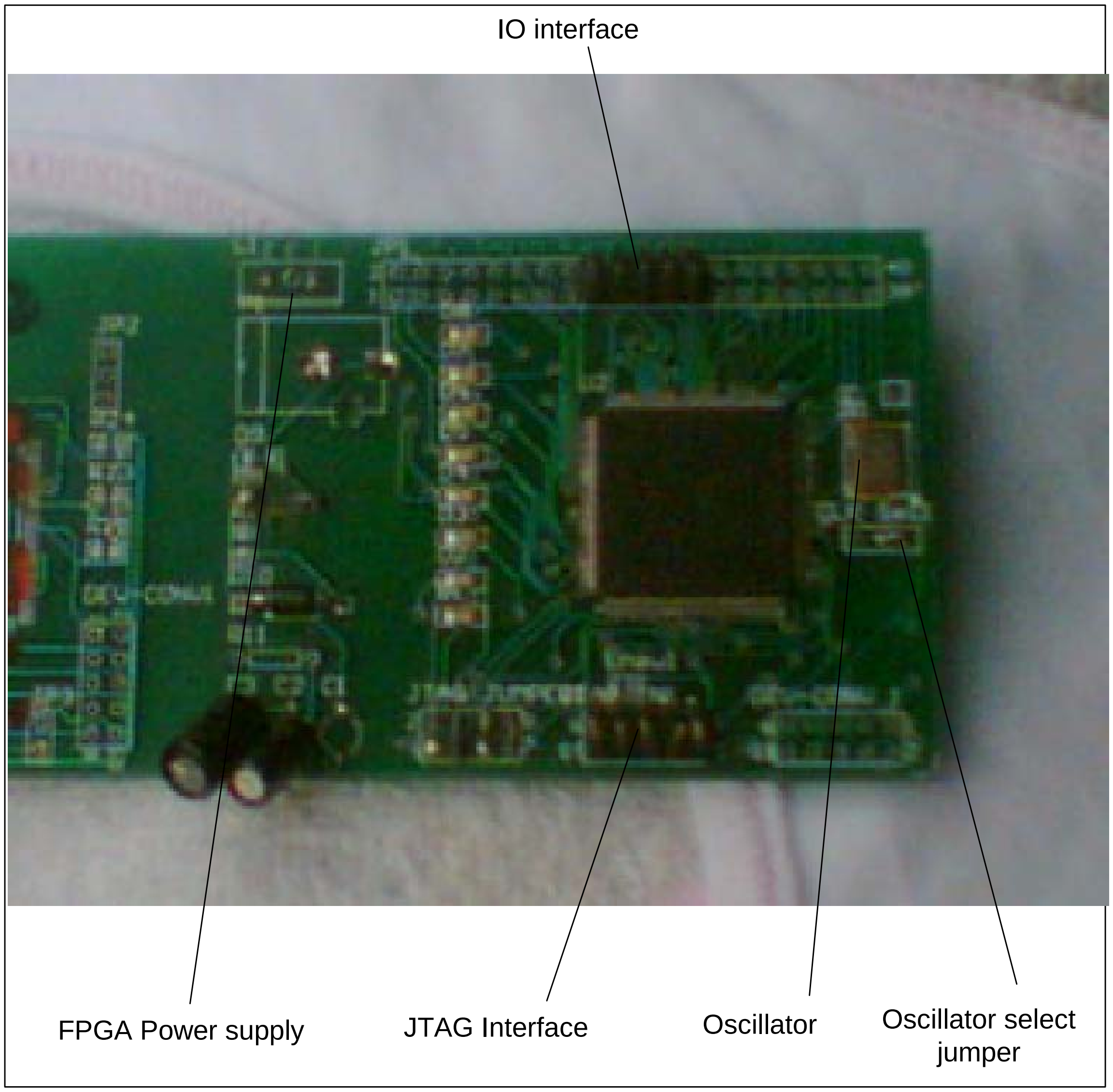


Fig. 4.13 FPGA PCB test board with clock signal select jumper

Chapter 5
Experimental Results

5.1 Testing the Resting Policy on the effects of the FPGA radiation response

In order to test the resting policy on SRAM FPGA's, the following setup was used. In the first instance, the FPGA was tested using normal operation and running a ring counter code (Case 1 in Fig. 5.1), while in the second case the FPGA was tested, also running the same code, however the power to the FPGA's were cycled (Case 2 in Fig. 5.1). The idea, as described in chapter 3, would be that the FPGA components would not degrade under the influence of ionizing radiation during its off cycle and therefore increasing its lifetime. Fig. 5.1 shows a comparison between normal FPGA operation (Case 1) and FPGA power cycling (Case 2). The dose rate as previously stated was 0.7 rads (Si)/sec, or about 2.5 krad/h. The floor plan for the ring counter code as placed in the FPGA is shown in Fig 5.3. Whenever power to the FPGA was restored, the same floorplan was loaded into the FPGA.

The FPGA's in case 1 of Fig. 5.1 started consuming more power supply current at about 15 krad and gradually increased in power supply current. Intermittent IO errors started occurring at 18 kRad (Fig 5.2) until about 22 kRad when functional failure occurred, at which point current monitoring was stopped for case 1 of Fig. 5.1. The FPGA's in case 2 of Fig. 5.1 also started consuming more power supply current, however at a slower rate as case 1. In this case, when the current drops to zero, power to the FPGA is switched off. Spacing of the power cycling dips towards the end of the test was different to the spacing before about 21krad. The reasons for this is that after 21krad, the slope of the curve increases dramatically with increasing absorbed dose, thus, power to the FPGA's has to be cycled faster to prevent functional failure.

For case 2, the FPGA's were still functionally intact after 30 krad when the testing was stopped. This result shows that one can make use of system or device redundancy to increase the lifetime of SRAM FPGA's in space. However, in order to increase the FPGA's lifetime without making use of redundancy at the system level, one would have to provide redundancy internally by means of the SMR algorithm. Further testing was performed with the Altera FPGA, with the conditions as shown in Fig. 5.4.

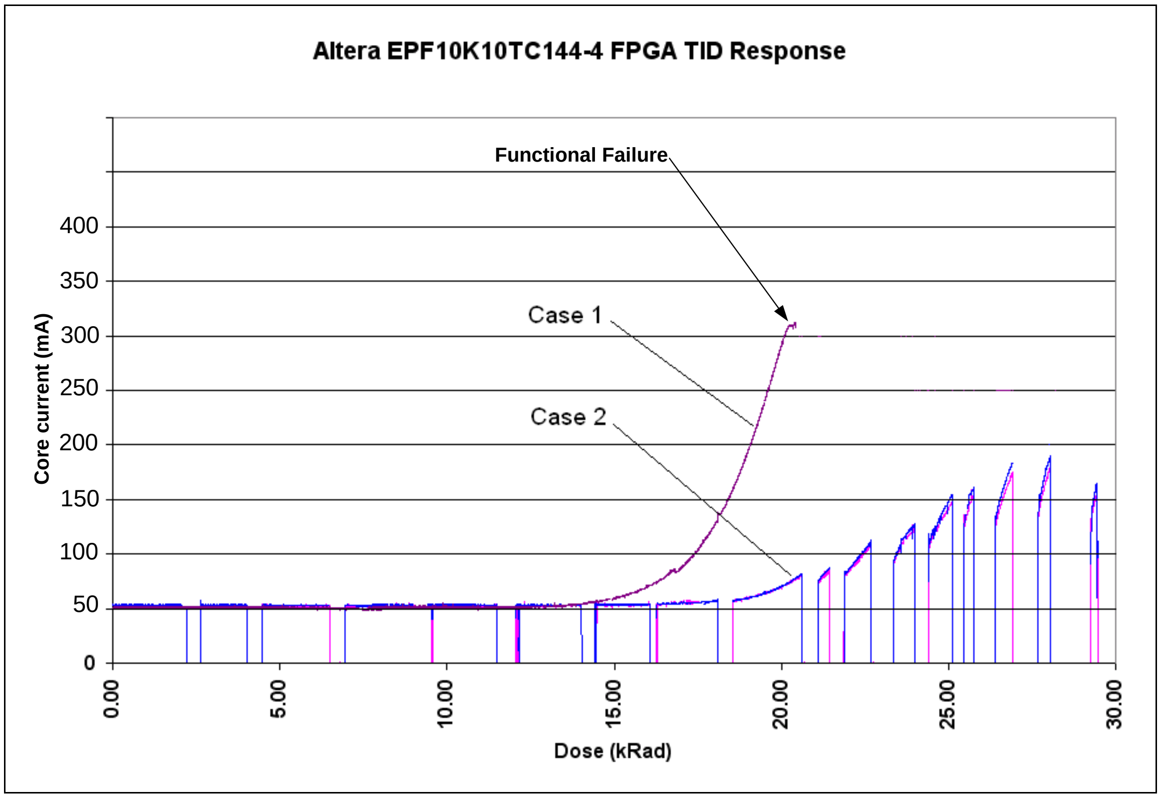


Fig. 5.1 Comparison between normal FPGA operation (Case 1) and FPGA power cycling (Case 2) for the Altera EPF10K10TC144-4 SRAM based FPGA

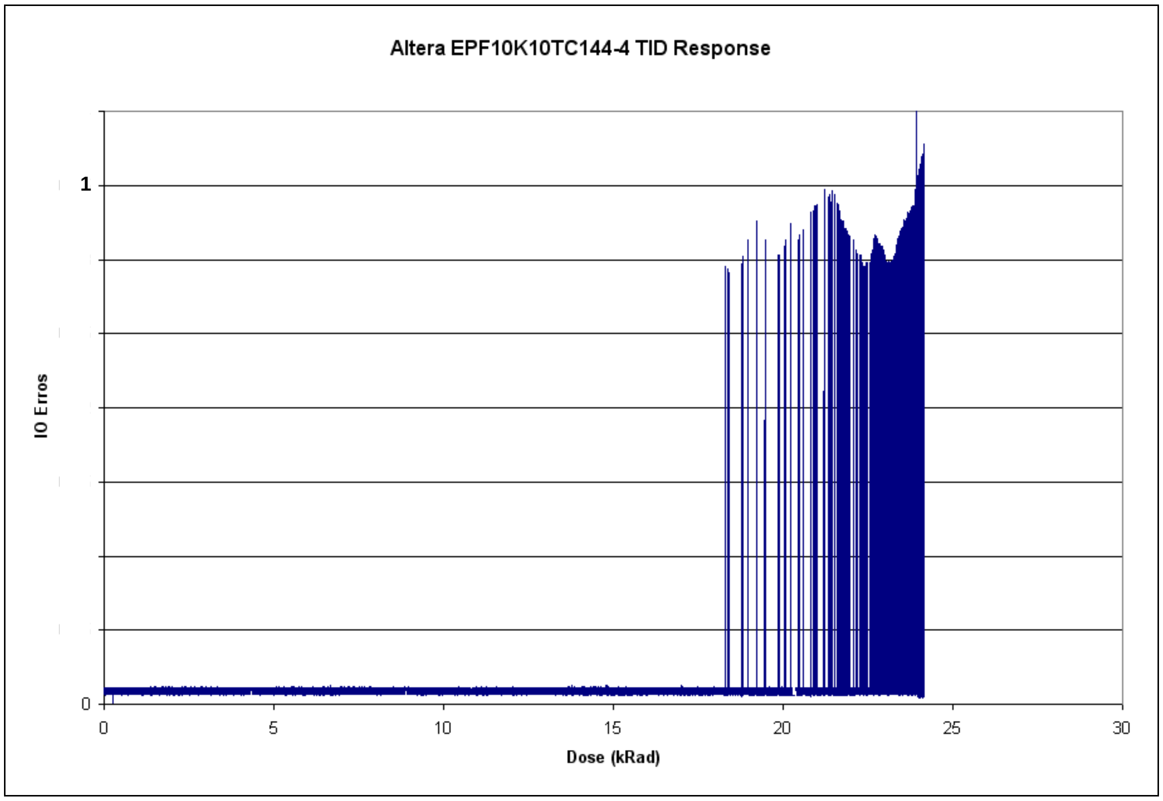


Fig. 5.2 FPGA IO errors for Fig 5.1, case 1.

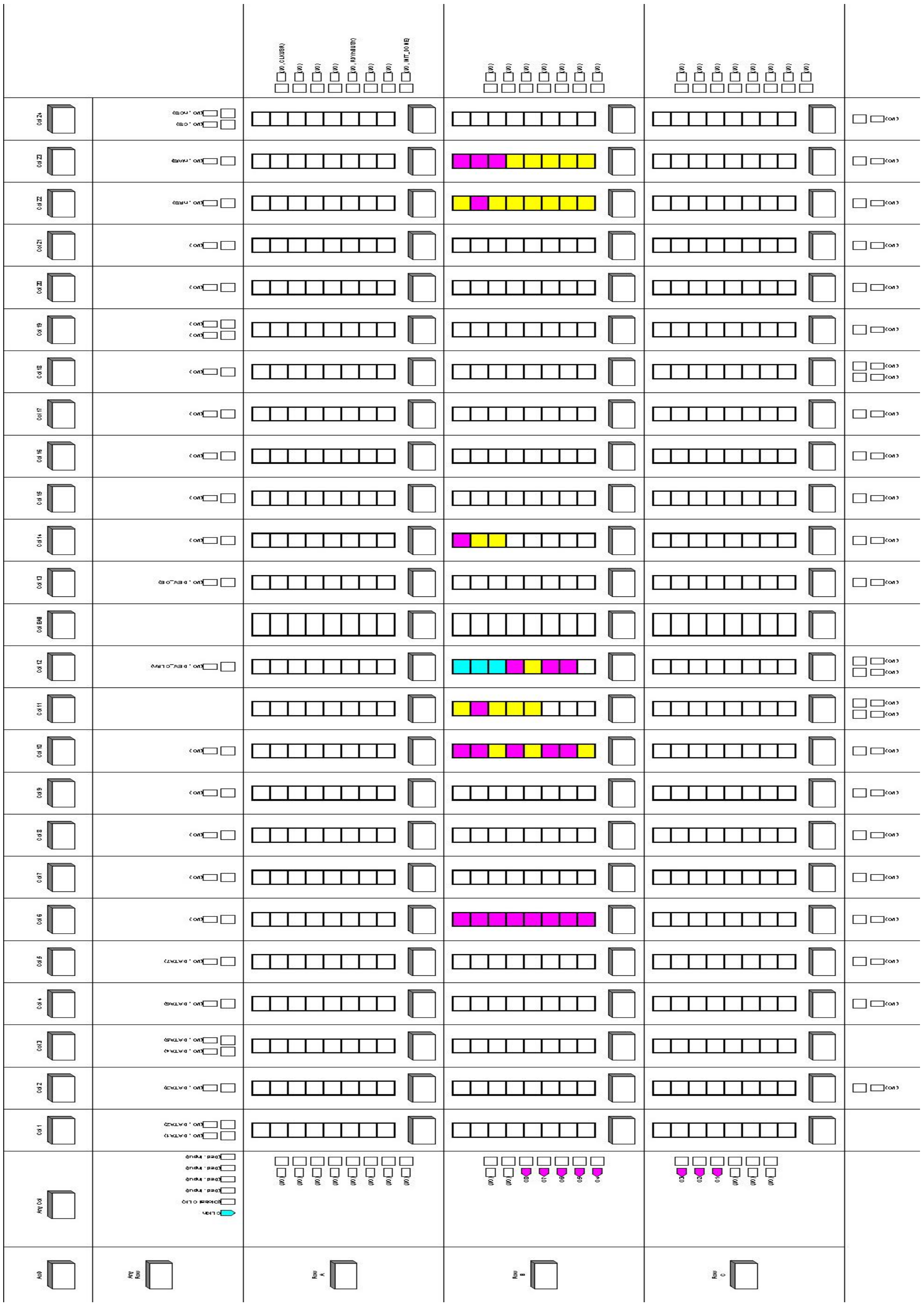


Fig 5.3 Floorplan for Fig 5.1, case 1 and case 2

5.2 Testing the effect of the clock and configuration memory on the FPGA radiation response

For the FPGA configured, but not clocked (case 2, Fig 5.4), the TID response is the same as with the normal operation (Case 1). Thus for case 2 of Fig. 5.4, the configuration memory is constant, but the switching matrix is not clocked. The zero clocking appears to have no effect. Thus, one could infer that the configuration memory plays an important role in the radiation response of the SRAM FPGA. This is because, although the FPGA is not clocked, the configuration memory is still constant, and degrades under the influence of ionizing radiation, as stated in chapter 3. The floorplan for the configured FPGA is the same as in the previous section.

In case 3 of Fig 5.4, the FPGA is configured, and then tested in a radiation field as in case 1. However, after 2.5 krad, the configuration memory is reset, without switching off the FPGA power. For this case, the TID tolerance is much better than normal operation (Case 1). In fact, it is similar to when power was completely reset as in Fig. 5.1.

This result agrees with the theoretical analysis of chapter 3. Since when the FPGA configuration memory is reset, its SRAM cells (or transistors) are reset and thus does not degrade in the presence of ionizing radiation. This is a very favorable result, because by building redundancy into the configuration memory (i.e. internally to the FPGA), one could significantly increase the lifetime of the SRAM FPGA in a radiation environment.

Spacing of the reconfiguration cycling dips towards the end of the test was different to the spacing at the beginning. The reasons for this is that toward the end of the test, as previously observed with the power cycling case where the slope of the curve increases dramatically with increasing absorbed dose, the power to the FPGA's has to be cycled faster to prevent functional failure.

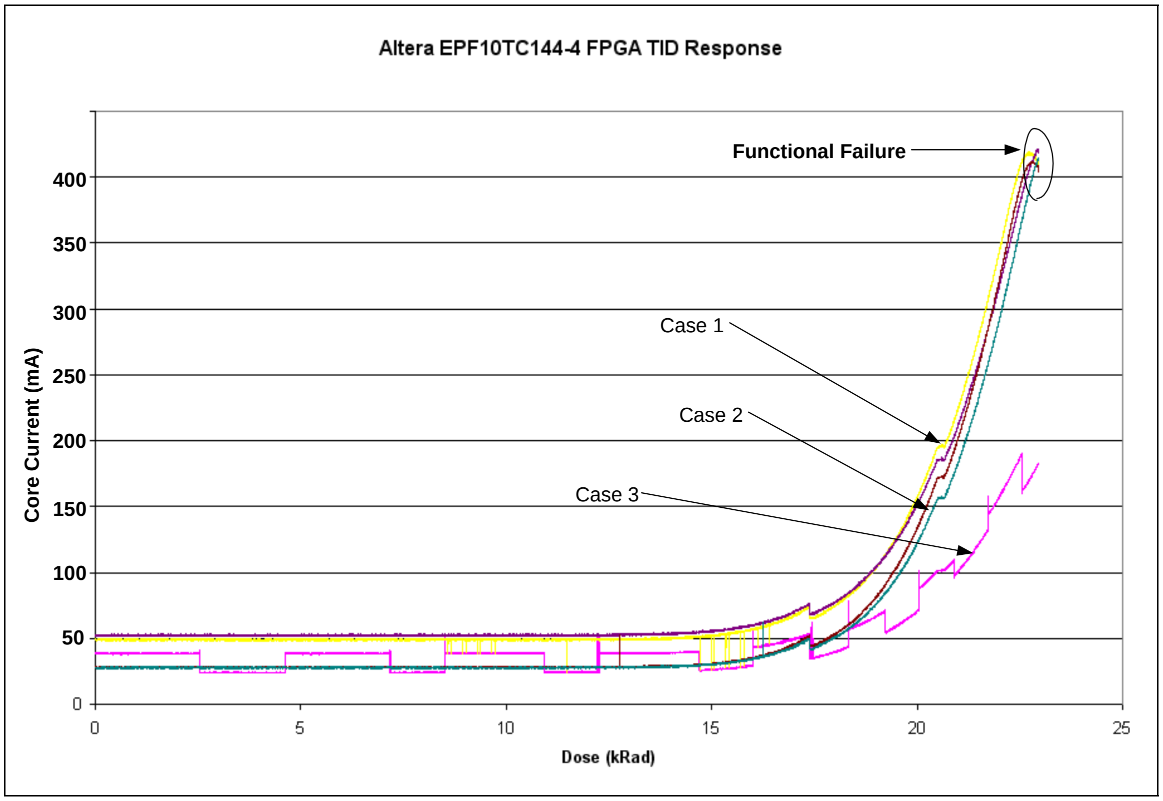


Fig. 5.4 Case 1: Normal operation as in case 1 above. Case 2: The FPGA is configured, but the clock signal is removed. Hence, no switching takes place in the switch matrix. Case 3: Normal operation, however the configuration memory were reset every 2.5 krad, and then reprogrammed again after a further approx 2.5 krad. For the time that the configuration memory is cleared, the power to the FPGA is still on.

5.3 Testing the SMR Algorithm by means of reconfigurable FPGA computing

Consider Fig. 5.5, after the FLEX 10K device has been configured, it was reconfigured in-circuit by loading to a different part of the configuration memory as described in chapter 4. However, the same IO ports were used during each reconfiguration. Hence, only the internal FPGA core is different for each reconfiguration. This is represented by case 2 and 3. The floorplan for case 1 of Fig 5.5 is the same as indicated in Fig 5.3. However, the floorplan for case 2 and 3 is given in Fig 5.6. As can be seen, the internal core is different to that in Fig 5.3, however the same IO ports are used.

Reconfiguration requires less than 320 ms during system operation. The FPGA with normal operation (case 1) failed functionally at about 20 krad, at which point current monitoring was stopped, whereas the reconfigured FPGA's failed functionally at about 33 krad. With the FPGA reconfiguration a 65% increase in functional lifetime is observed.

In case 4 of Fig 5.5, the FPGA was reconfigured with both a different part of the configuration memory as well as different IO ports. Hence, a different internal core was used during each reconfiguration, as well as different IO ports. The floorplan for this scenario is shown in Fig 5.7.

There seems to be an improved performance in case 4 compared to cases 2 and 3. However, with increasing absorbed dose, case 4's current soon increases and the FPGA failed functionally at 34 krad. The above results indicate that FPGA internal core redundancy provided TID mitigation, however, by provided IO port redundancy does not further add to the functional lifetime of the FPGA.

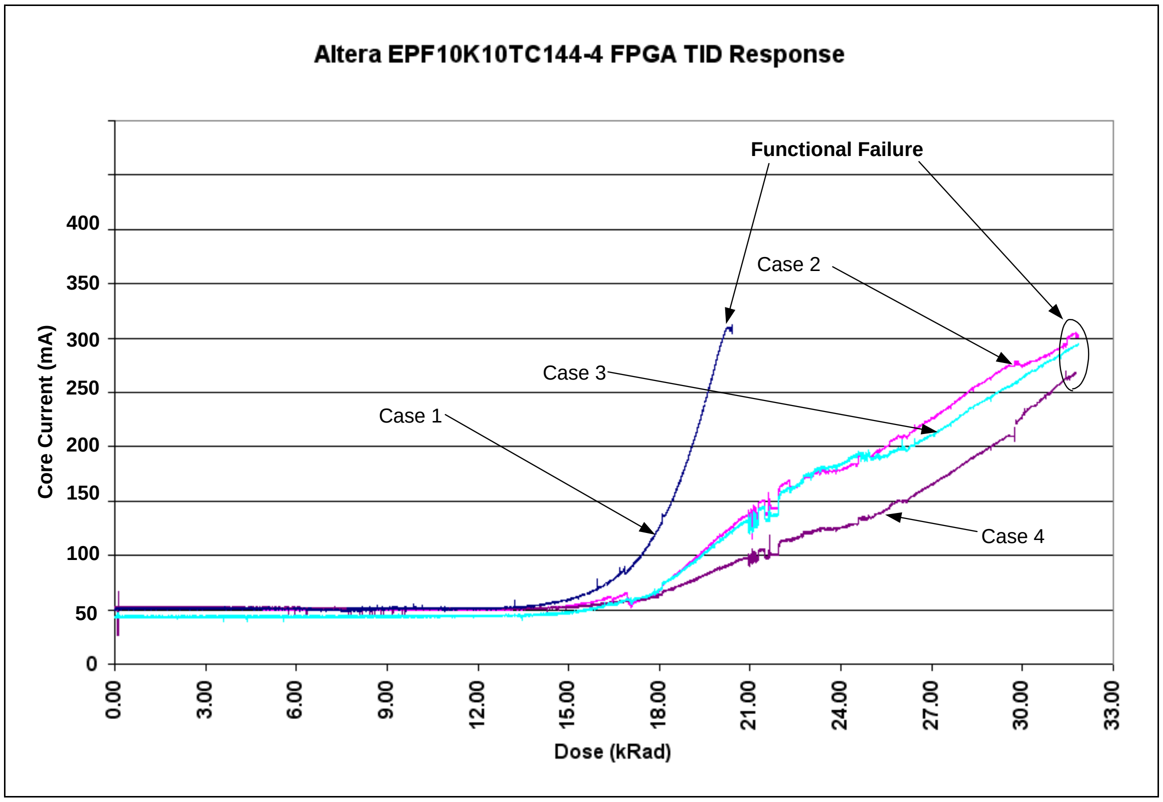


Fig. 5.5 Case 1: Normal Operation. Case 2, 3: Configuration memory is reset every half hour to a different part of the configuration memory. Case 4: Configuration memory is set to a different internal core as well as different IO ports.

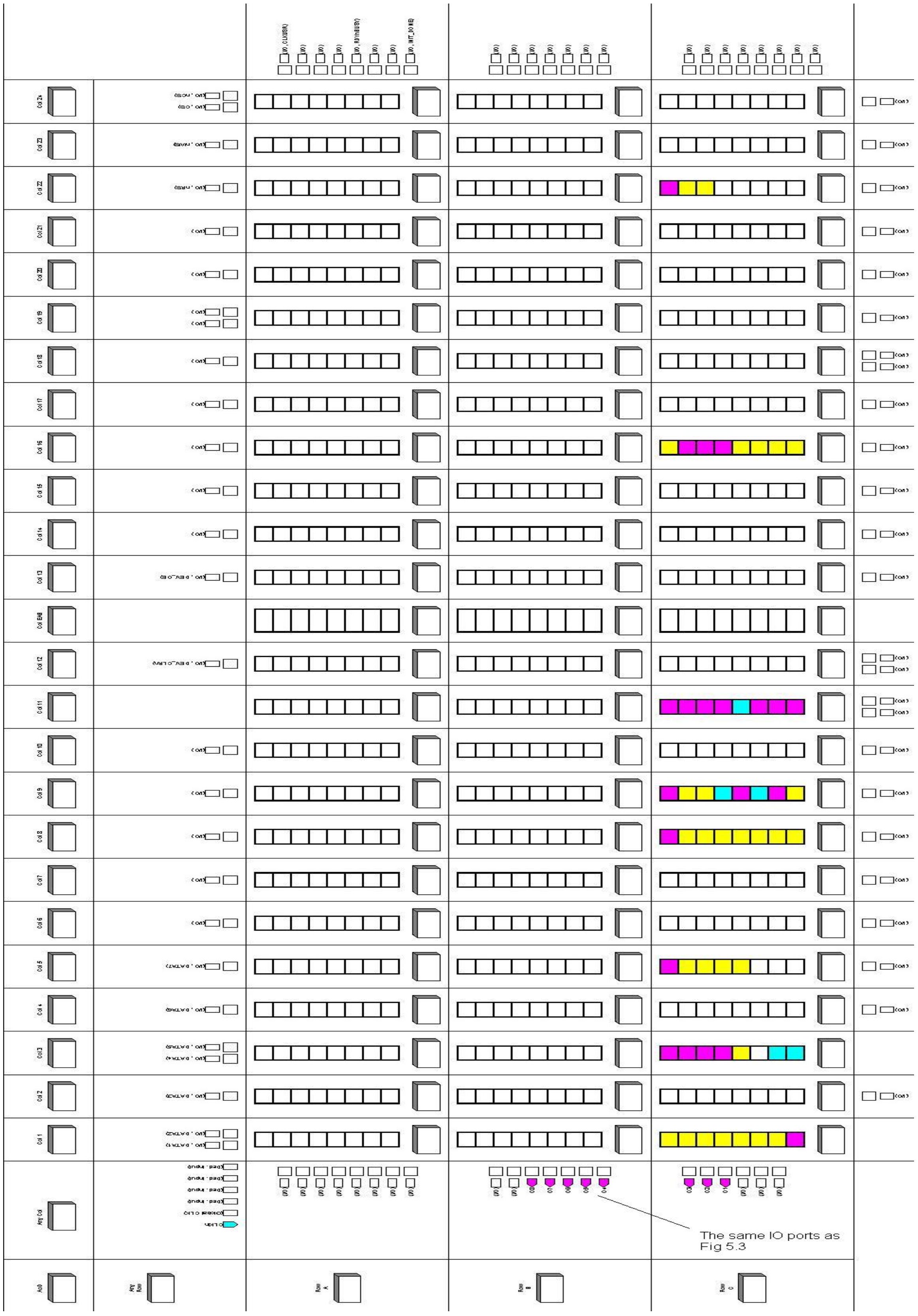


Fig 5.6 Floorplan for Fig 5.5, case 2 and 3

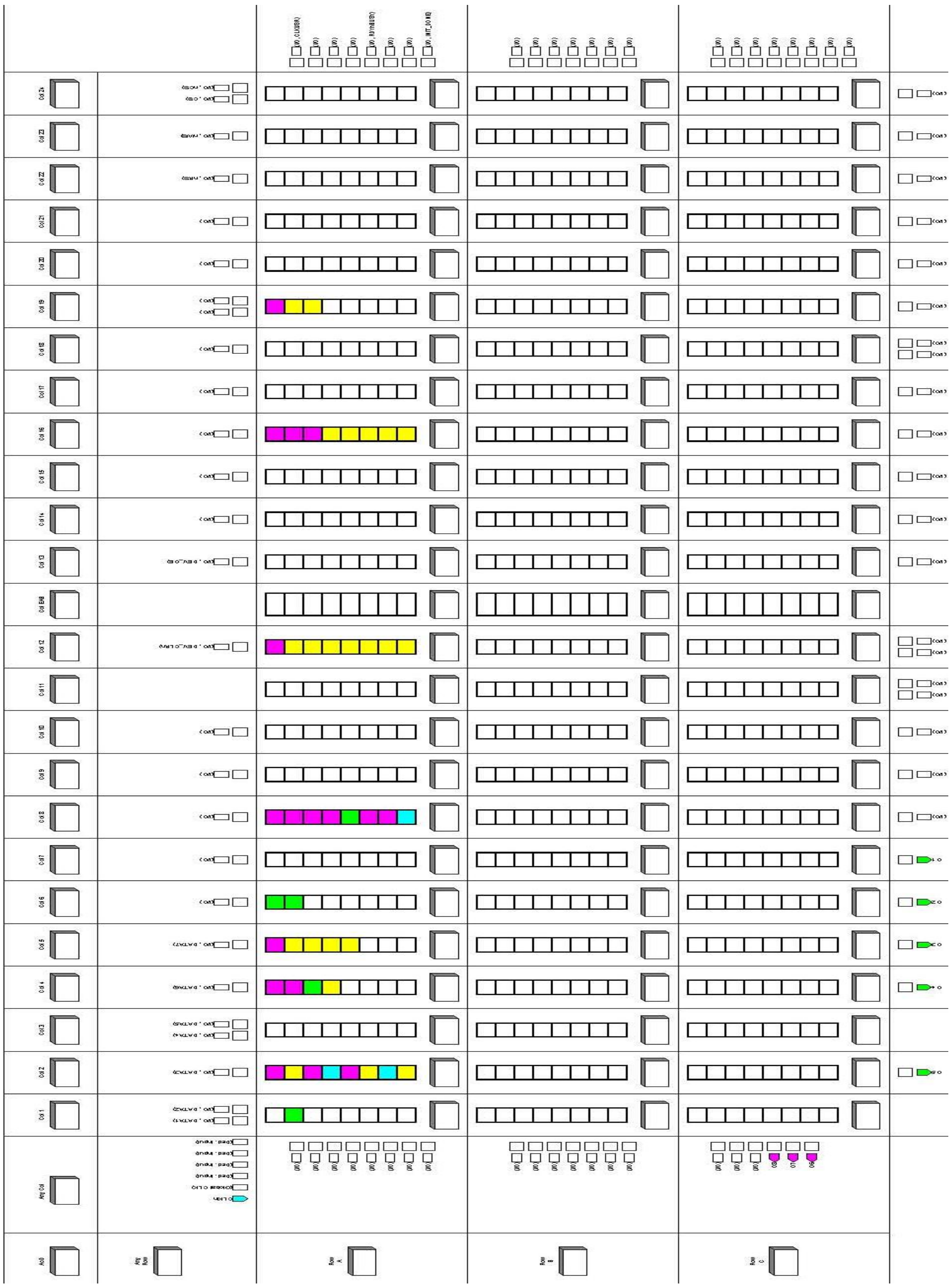


Fig 5.7 Floorplan for Fig 5.5, case 4.

Further, if one looks at Fig 5.8, which shows the IO current of the FPGA versus the absorbed dose, the IO current does not increase for an increase in absorbed radiation dose for the entire measurement period.

This does not mean that the IO ports are immune against TID, however, the results does suggest that the IO ports have a higher tolerance to the radiation than the FPGA core for the measured absorbed dose. One can thus safely say that the FPGA core is the first and main source of the FPGA power supply current increase.

Thus one can make use of the same IO ports for each reconfiguration and thus do not have to make changes to the PCB board.

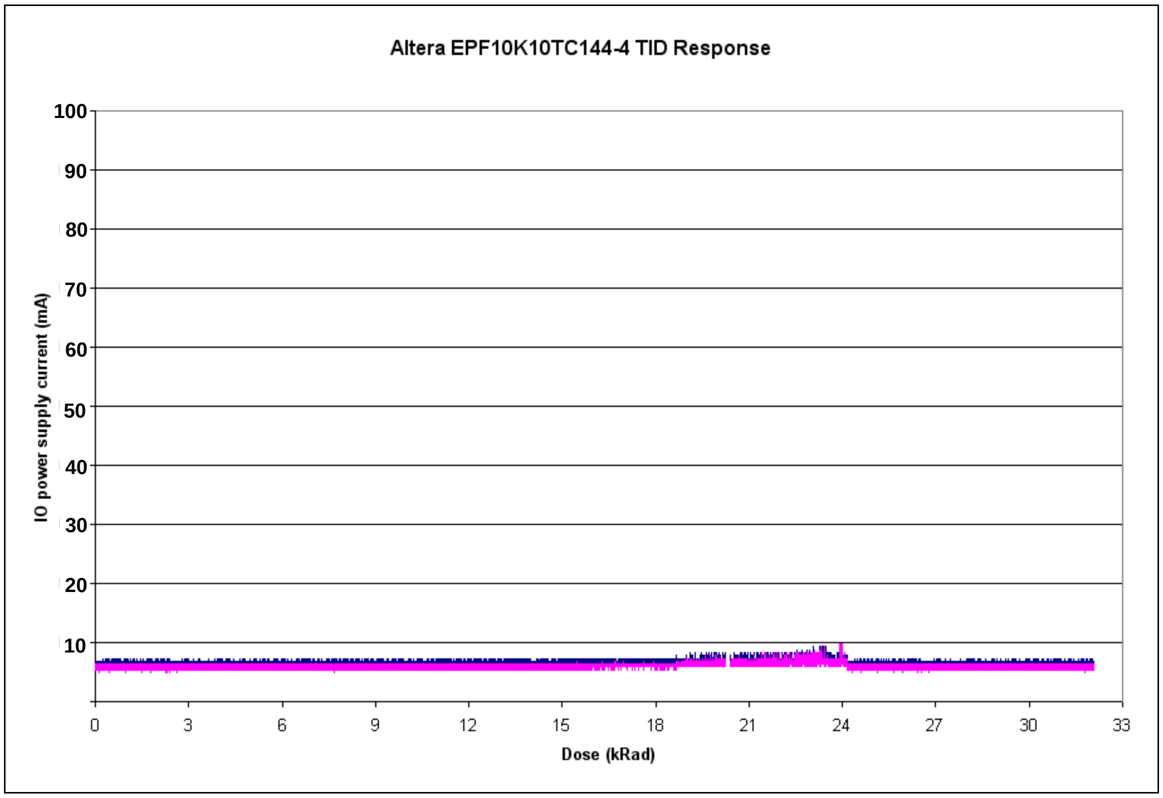


Fig 5.8 IO power supply TID Response

Chapter 6
Conclusions and Recommendations

6.1 Conclusions

For many years, the space radiation community has studied and evaluated radiation hard technologies suitable for space applications. During that time, vendors of radiation hardened technologies have faced a considerable reduction, and the space community has focused its interest more on the use of Commercial-Off-The-Shelf (COTS) components rather than the highly expensive and less advanced Radiation hard components that they traditionally used in the past.

As a consequence of this evolution several semiconductor companies have abandoned the radiation hard electronics market, and now only a very few companies in the world offer radiation-hard technologies. In view of this market trend, and as an answer to the requirement of space applications that requires high performance devices with low power, low cost, high flexibility and time to market as well as the radiation tolerance, we have investigated an alternative approach based on radiation tolerant design techniques in CMOS FPGA COTS technology.

The main contribution of this dissertation was the development of the novel Switched Modular Redundancy (SMR) method for mitigating the effects of space radiation on satellite electronics. It was proposed in this dissertation that if we apply a zero bias to the gate terminal of a MOS transistor in the presence of ionizing radiation, i.e. no electric field across the gate oxide, both the free electrons and holes will on average remain near their point of origin, and therefore have a greater probability of recombination. Thus, the threshold voltage shift in MOS devices will be less severe for the gate terminal in an unbiased condition.

It was further proposed that by adding redundancy and applying a resting policy, one can significantly prolong the useful life of MOS components in space. This redundancy could be used in a circuit to mitigate for total ionizing dose.

We applied the principles of reconfigurable computing to implement the Switched Modular Redundancy Algorithm in order to mitigate for Total Ionizing Dose (TID) effects in FPGA's. It was shown by means of experimentation that this new design technique provides greatly improved TID tolerance for FPGAs.

The method consists of applying Switched Modular Redundancy to the configuration memory in the FPGA. For devices which are subject to gate bias cycling, the maximum acceptable dose is higher than if the irradiation bias were applied continuously. By adding redundancy and applying a resting policy, one can significantly prolong the useful life of MOS components in space. It was shown experimentally that by applying FPGA system redundancy on a power cycling basis, the system lifetime is increased significantly. By resetting the configuration memory, the functional lifetime of the FPGA resembles that of power cycling. By applying redundancy in the configuration memory, the lifetime of the SRAM FPGA was increased in the presence of ionizing radiation. It was also shown through the current consumption of the IO ports, that the IO ports are not as susceptible to radiation as the FPGA core, which is the main cause of the increase in power supply current in the presence of ionizing radiation.

In the experiments presented in this dissertation, two times redundancy was provided. However, in a space application, depending on the size of the circuit compared to the capacity of the FPGA, more than two times redundancy can be used to prolong the lifetime of the device. For example, in order to use the above SMR methodology, the circuit should be at most 50% of the device capacity in order to duplicate the circuit internally. Depending on the satellite orbit, and hence the required total absorbed dose of the mission, the amount of redundancy can be increased to meet the mission requirements.

In our experiments with the Altera EPF10K10TC144-4 SRAM FPGA, we obtained a 65% increase in functional lifetime with two times redundancy, i.e. instead of failing at 20krad, the FPGA failed functionally at 33 krad. In order to further increase the FPGA functional lifetime, one would have to add additional redundancy. However, what complicates the matter is that it does not appear that the total absorbed dose at which functional failure occurs is linear with the number of redundancies. For example, one would have expected two times redundancy to correspond to a functional failure dose of 40 krad, if functional failure with no redundancy occurs at 20 krad. Thus, in order to use this methodology, one would have to test the percentage increase in functional lifetime for each amount of circuit duplication in order to assess the required amount of redundancy. Also, because each FPGA technology is unique in its architecture, and each has its unique response to radiation, every new FPGA has to be tested to determine the percentage increase in functional lifetime. Although the principle of removing the bias on the gate of the MOS transistor should lead to a linear improvement in prolonging the MOS lifetime in the presence of ionizing radiation, the complexity of the circuit in an FPGA and lack of transistor gate bias control, leads to a lower lifetime value than one would expect.

It was shown experimentally that by applying FPGA system or device redundancy on a power cycling basis, the system lifetime is increased significantly. Device redundancy mitigation is an alternative method to internal FPGA mitigation. In terms of prolonged functional lifetime, no distinction can be made between device redundancy and internal FPGA redundancy. However, device redundancy is the most costly solution in terms of both PCB board space and device costs.

It is noted that the scheme does not make provision for Single Event Effects mitigation, however, whenever a reconfiguration cycle is started, the configuration memory is reset and hence any configuration errors due to SEE will be corrected. However, it is not sufficient to update the configuration SRAM memory continuously to remove any bit errors induced by SEEs, since the effect of the configuration change will change the logic which in turn will potentially lead to the change of the internal state of the design, i.e. the state of the various registers and flip-flops [HABI02]. By correcting the configuration SRAM memory, one can repair the logic, but not re-establish the state of the design. Thus, to properly provide for any SEE effects, one would have to apply Triple Modular Redundancy in addition to the SMR methodology.

6.2 Recommendations

1. The proposed technique to enhance TID tolerance of SRAM-based FPGA is based on the availability of additional free resources in the device. Since hardening FPGA to SEEs, which is also mandatory for Space applications, also typically require additional resources and actions involving configuration memory, further research should address how this approach can combine with SEE hardening strategies.

   The author is currently working on a proposal to use the SMR algorithm in order to mitigate for SEU effects in FPGAs. We propose a new design technique for SEU mitigation in the Field Programmable Gate Array configuration memory. The method consists of applying Switched Modular Redundancy to the configuration memory in the FPGA. Used in conjunction with the TID mitigation features as indicated in this thesis, this dual mitigation feature would make it the only known method to cater for both SEU and TID simultaneously. However, further research has to be done in order to confirm the proposed SEU mitigation method experimentally.

   Two papers were written on the new SEU mitigation methods and remain unpublished. The papers can be found on the accompanying CD under the folder "SMR SEU Mitigation Papers" and the simulation code in appendix C. However, it must be stressed that further research needs to be done in order to consider it complete.

2. SRAM and FLASH based FPGA devices are very different, and radiation effects are as a consequence very different. In particular, the configuration memory is fundamentally different. Further testing needs to be done on Flash based FPGA's in order to demonstrate that our approach is also applicable to them.

3. The SMR methodology, as described in chapter 3, would be to duplicate each gate in a circuit, then selectively only activating one gate at a time allowing the other to anneal during its off cycle. The SMR algorithm is coded in the "C" language. In the proposed design methodology, the design engineer need not be concerned about radiation effects when describing the hardware implementation in a hardware description language. Instead, the design engineer makes use of conventional design techniques. When the design is complete, it is synthesized to obtain the gate level netlist in edif format.

   The edif netlist is converted to structural VHDL code during synthesis. The structural VHDL netlist is fed into the SMR "C" algorithm to obtain the identical redundant circuit components. The resultant file is also a structural VHDL netlist. The generated VHDL netlist or SMR circuit is then mapped to the FPGA. However, we never made use of the above method in FPGAs for the reasons given in chapter 3, since there are no fundamental gates in FPGA's and we were more interested in the FPGA configuration memory.

   The SMR code that was developed is capable of the functions mentioned above and further research need to be done that use the SMR code to the gate level in digital circuits.

4. We have used the redundancy method in order to provided TID mitigation to FPGAs. However, there is no reason that the theory of chapter 3 cannot be extended to include other digital electronic circuits as well as analogue electronics. This needs to be further investigated.

References

1. **[ADAM02]** L. Adams, "Guidelines for the use of Electronic Components in the Space Radiation Environment", Technical Report, Issue 2, 18th March 2002.
2. **[ALEX96]** D. R. Alexander, "Design issues for radiation tolerant microcircuits for space", Notes of the Short Course of the IEEE Nuclear and Space Radiation Effects Conference, Indian Wells (California), July 1996, Section V.
3. **[ALEX96]** D. R. Alexander, Design Issues for Radiation Tolerant Microciruits for Space, IEEE Nuclear Science and Radiation Effects Conference Short Course, 1996.
4. **[ALTE03]** "Flex 10K Embedded Programmable Logic Device Family" Data Sheet, January 2003, ver. 4.2.
5. **[ANEL00]** GM Anelli, "CONCEPTION ET CARACTERISATION DE CIRCUITS INTEGRES RESISTANTS AUX RADIATIONS POUR LES DETECTEURS DE PARTICULES DU LHC EN TECHNOLOGIES CMOS SUBMICRONIQUES PROFONDES" PhD thesis, INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, page 29, 2000.
6. **[ANTA00]** www.antarctica.ac.uk / News and Information / Press Releases / 2000 . Insurance industry funds new research into satellite failures. http://www.antarctica.ac.uk/News_and_Information/Press_Releases/2000/20000712-1.html.
7. **[BARR]** M.J. Barry, "Radiation resistant SRAM memory cell", US Patent No. 5,157,625.
8. **[BART97]** BARTH, Janet. Radiation Environment. In: IEEE NSREC Short Course, July 21, 1997. http://flick.gsfc.nasa.gov/radhome/ RPO_slides.htm.
9. **[BELL04]** M. Bellato P. Bernardi1, D. Bortolato, A. Candelori, M. Ceschia, A. Paccagnella, M. Rebaudengo1, M. Sonza Reorda1, M. Violante1 and P. Zambolin, "Evaluating the effects of SEUs affecting the configuration memory of an SRAM based FPGAs", Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2004.

10. **[BENE86]** J. M. Benedetto and H. E. Boesch, “The Relationship Between Co60 and 10 keV X-Ray- Damage in MOS Devices”, IEEE Transactions on Nuclear Science, vol. 33, no. 6, pp. 1318–1323, Dec. 1986.
11. **[BETZ05]** V. BETZ, “FPGA Place-and-Route Challenge”, 2005.
12. **[BROW97]** S. Brown, J. Rose, “Architecture of FPGAs and CPLDs: A Tutorial”, Department of Electrical and Computer Engineering, University of Toronto, 1997.
13. **[BUNS00]** P. E. Bunson, M. Di Ventra, S. T. Pantelides, D. M. Fleetwood, and R. D. Schrimpf, “Hydrogen-Related Defects in Irradiated SiO2”, IEEE Transactions on Nuclear Science, vol. 47, no. 6, pp. 2289–2296, Dec. 2000.
14. **[BUNS99]** P. E. Bunson, M. Di Ventra, S. T. Pantelides, R. D. Schrimpf, and K. F. Galloway, “Ab initio calculations of H+ energetics in SiO2: Implications for transport”, IEEE Transactions on Nuclear Science, vol. 46, no. 6, pp. 1568–1573, Dec. 1999.
15. **[CANA91]** J. Canaris, S. Whitaker, and M.N. Lui, “SEU hardened memory cells for a CCSDS Reed Solomon encoder”, IEEE Trans. on Nuc. Sci, vol 38(6), pages 1471 – 1477, December 1991.
16. **[CANA95]** J. Canaris, S. Whitaker, “Circuit techniques for the radiation environment of space”, IEEE 1995 Custom Integrated Circuits Conference, pages 5.4.1 – 5.4.4, 1995.
17. **[CONL92]** J. F. Conley, Jr. and P. M. Lenahan, “Room-Temperature Reactions Involving Silicon Dangling Bond Centers and Molecular-Hydrogen in Amorphous SiO2 Thin-Films on Silicon”, IEEE Transactions on Nuclear Science, vol. 39, no. 6, pp. 2186–2191, Dec. 1992.
18. **[CONL93]** J. F. Conley, Jr. and P. M. Lenahan, “Room-Temperature Reactions Involving Silicon Dangling Bond Centers and Molecular-Hydrogen in Amorphous SiO2 Thin-Films on Silicon”, Applied Physics Letters, vol. 62, no. 1, pp. 40–42, 1993.
19. **[CONY93]** J. F. Conley, Jr. and P. M. Lenahan, “Molecular Hydrogen, E0 Center Hole Traps, and Radiation-Induced Interface Traps in MOS Devices”, IEEE Transactions on Nuclear Science, vol. 40, no. 6, pp. 1335–1340, 1993.

20. **[DENT00]** Martin Dentan. RADIATION EFFECTS ON ELECTRONIC COMPONENTS AND CIRCUITS. In: CERN Training Course. April 11, 2000. (http://atlas.web.cern.ch/Atlas/GROUPS/FRONTEND /radhard.htm).
21. **[DERB77]** G. F. Derbenwick and H. H. Sander, "CMOS Hardness Prediction for Low-Dose-Rate Environments," IEEE Trans. Nucl. Sci., NS-24, 2244-2247 (1977).
22. **[DODD03]** P. Dodd, L.W. Massengill, "Basic Mechanisms and Modeling of Single-Event Upset in Digital Microelectronics", IEEE Transactions on Nuclear Science,, VOL. 50, NO. 3, June 2003.
23. **[DOOL]** J.G. Dooley, "SEU-immune for gate array, standard cell, and other ASIC applications", US Patent No. 5,311,070.
24. **[DRES86]** P. Dressendorfer, J. Soden, J. Harrington, T. Nordstrom, "The effects of test conditions on MOS radiation-hardness results", IEEE Trans. Nucl. Sci. Vol. NS-28, No. 6, Dec 1986.
25. **[DYER01]** C. Dyer, "Radiation Effects on Spacecraft & Aircraft", *Space Department, QinetiQ, Cody Technology Park, Farnborough, Hampshire, 2001.*
26. **[EDSO01]** Edson L. Horta, John W. Lockwood, "PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGA's)", Washington University, July 06, 2001.
27. **[FLEE03]** D.M. Fleetwood, "Total-Dose Radiation Hardness Assurance", IEEE Trans. Nuclear Science, Vol. 50, No. 3, June 2003.
28. **[FLEE88]** Fleetwood, D.M.; Winokur, P.S.; Schwank, J.R, "Using laboratory X-ray and cobalt-60 irradiations to predict CMOS device response in strategic and space environments" IEEE Trans. on Nucl. Sci, Vol. 35, No. 6, December 1988.
29. **[FLEE95]** D. M. Fleetwood, W. L. Warren, J. R. Schwank, P. S. Winokur, M. R. Shaneyfelt, and L. C. Riewe, "Effects of Interface Traps and Border Traps on MOS Postirradiation Annealing Response", IEEE Transactions on Nuclear Science, vol. 42, pp. 1698–1707, 1995.
30. **[FREI87]** R.K. Freitag, C.M. Dozier, D.B. Brown, " Growth and annealing of trapped holes and interface states using time dependent biases", IEEE Trans on Nucl Sci, Vol. NS-34, No. 6, December 1987.

31. **[FREI93]** R. K. Freitag, D. B. Brown, and C. M. Dozier, “Experimental Evidence of Two Species of Radiation-Induced Trapped Positive Charge”, IEEE Transactions on Nuclear Science, vol. 40, no. 6, pp. 1316–1322, 1993.
32. **[GALL84]** K. F. Galloway, M. Gaitan, and T. J. Russell, “A Simple Model for Separating Interface and Oxide Charge Effects in MOS Device Characteristics”, IEEE Transactions on Nuclear Science, vol. NS-31, no. 6, pp. 1497–1501, 1984.
33. **[HABI02]** S. Habinc, “functional triple modular redundancy,” Gaisler Research, Dec. 2002, Design and Assessment Rep.
34. **[HAMB03]** J.O. Hamblen, M.D. Furman, “Rapid prototyping of digital systems”, Kluwer Academic Publishers, 2003.
35. **[HOLM02]** A. Holm-Siedle, L. Adams, “Handbook of Radiation Effects: Second Edition”, Oxford University Press, 2002.
36. **[HUGE03]** H.L. Huges and J.M. Benedetto, “Radiation Effects and Hardening of MOS Technology: Devices and Circuits”, IEEE Transactions on Nuclear Science, Vol.50, No. 3, June 2003.
37. **[JOHN83]** A. H. Johnston, “ Annealing of total dose damage in the Z80 Microprocessor”, IEEE Trans on Nucl Sci, Vol NS-30, No. 6, December 1983.
38. **[KATZ97]** R. Katz’, K. LaBel’, J.J. Wang, B. Cronquist2, R. Koga’, S. Penzin’, and G. Swift, “Radiation Effects on Current Field Programmable Technologies” IEEE Trans. on Nucl. Sci., Vol. 44, NO. 6, December 1997.
39. **[KERN88]** S. E. Kerns, B. D. Shafer, L. R. Rockett et al., “The Design of Radiation-Hardened ICs for Space: A Compendium of Approaches”, *Proceedings of the IEEE*, vol. 76, no. 11, November 1988, pp. 1470-1509.
40. **[KLUW01]** J.O. Hamblen, M.D. Furman, “Rapid Prototyping of Digital Systems, Second Edition”, Kluwer Academic Publishers, 2003.
41. **[LABE96]** K.A. LaBel and M.M. Gates, “Single Event Effect Mitigation from the System Perspective”, *IEEE Trans. on Nuclear Science*, vol 43, no. 2, pp. 654-660, Apr. 1996.
42. **[LABE98]** K. LaBel, A.H. Johnson, J.L. Barth, R.A. Reed, C.E. Barnes,”Emerging Radiation Hardness Assurance issues: A NASA approach for space flight programs”, NASA Jet Propulsion Laboratory, 1998.

43. **[LABE99]** LABEL, K. et al. Commercial Microelectronics Technologies for Applications in the Satellite Radiation Environment. In: http://flick.gsfc.nasa.gov/ radhome.htm (Nov. 1999).

44. **[LENA99]** P. M. Lenahan, J. J. Mele, J. F. Conley, Jr., R. K. Lowry, and D. Woodbury, "Predicting Radiation Response From Process Parameters: Verification Of A Physically Based Predictive Model", IEEE Transactions on Nuclear Science, vol. 46, no. 6, pp. 1534–1543, Dec. 1999.

45. **[LIMA02]** F.G. Lima, "DESIGNING SINGLE EVENT UPSET MITIGATION TECHNIQUES FOR LARGE SRAM-based FPGA DEVICES" Porto Alegre, February 11th, 2002

46. **[LIU92]** M.N. Lui and S. Whitaker, "Low power SEU immune CMOS memory circuits", IEEE Trans. on Nuc. Sci, vol. 39(6), pages 1679 – 1684, December 1992.

47. **[MA89]** T.P. Ma, P.V. Dressendorfer, "Ionizing radiation effects in MOS Devices and Circuits", John Willey & Sons, 1989.

48. **[MACL89]** F. B. McLean and H. E. Boesch, Jr., "Time-Dependent Degradation of MOSFET Channel Mobility Following Pulsed Irradiation", IEEE Transactions on Nuclear Science, vol. 36, no. 6, pp. 1772–1783, Dec. 1989.

49. **[MCC099]** J. McCollum, "Programmable Elements and Their Impact on FPGA Architecture, Performance, and Radiation Hardness," MAPLD 1999 Proceedings, Laurel, Maryland, Sept. 1999.

50. **[MCLE99]** T.R. Oldham, F.B. Mclean, H.E. Boesch, J.M. McGarrity, "An overview of radiation-induced interface traps in MOS structures", Semicond. Sci. Technol. 4 (1999) 986 – 999.

51. **[MESS92]** George C. Messenger and Milton S. Ash, The Effects of Radiation on Electronic Systems, Van Nostrand Reinhold, July 1992.

52. **[MONR03]** D.Monroe,http://www.louisville.edu/~djmonr01/vanallen.html , April 2003.

53. **[NICK00]** C. J. Nicklaw, M. P. Pagey, S. T. Pantelides, D. M. Fleetwood, R. D. Schrimpf, K. F. Galloway, J. E. Wittig, B. M. Howard, E. Taw, W. H. McNeil, and J. F. Conley, "Defects and Nanocrystals Generated by Si Implantation into a-

SiO2", IEEE Transactions on Nuclear Science, vol. 47, no. 6, pp. 2269–2275, Dec. 2000.

54. **[NICK03]** C.J. Nicklaw, "MULTI-LEVEL MODELING OF TOTAL IONIZING DOSE IN a-SiO2: FIRST PRINCIPLES TO CIRCUITS", PhD Thesis, Graduate School of Vanderbilt University, 2003.

55. **[NICO95]** M. Nicolaidis, T. Calin, F. Vargas, R. Velazco, "A low cost, highly reliable SEU-tolerant SRAM: Prototype and test results", IEEE Trans. on Nuc. Sci, vol. 42(6), pages 1592 – 1598, 1995.

56. **[NICO96]** M. Nicolaidis, T. Calin, and R. Velazco, "Upset hardened memory design for submicron CMOS technology", IEEE Trans. on Nuc. Sci, vol. 43, pages 2874 – 2878, December 1996.

57. **[OKAB90]** T. Okabe, M. Kato, M. Katsueda, H. Kamimura, I. Takei, " High Frequency annealing effects on Ionizing Radiation response of MOSFET", IEEE Trans. On Nucl. Sci, Vol 37, No6, December 1990.

58. **[OLDH99]** T.R. Oldham, "Ionizing Radiation Effects in MOS Oxides", World Scientific Publishing, 1999.

59. **[POIN84]** E. H. Poindexter, G. J. Gerardi, M.-E. Rueckel, P. J. Caplan, N. M. Johnson, and D. K. Biegelsen, "Electronic Traps and Pb Centers at the Si/SiO2 Interface: Band-Gap Energy Distribution", Journal of Applied Physics, vol. 56, no. 10, pp. 2844–2849, 15 Nov. 1984.

60. **[RASH01]** S. N. Rashkeev, D. M. Fleetwood, R. D. Schrimpf, and S. T. Pantelides, "Defect Generation by Hydrogen at Si-SiO2 Interface", Physical Review Letters, vol. 87, no. 16, pp. 165506– 1–165506–4, 15 Oct. 2001.

61. **[RUTE01]** R. Rutenbar et. al. " (When) Will FPGA's kill ASIC's", Proceedings of the 38th conference on Design automation, Las Vegas, Nevada, United States, pg 221 -222, 2001.

62. **[SAKS93]** N. S. Saks, R. B. Klein, R. E. Stahlbush, B. J. Mrstik, and R. W. Rendell, "Effects of Post- Stress Hydrogen Annealing on MOS Oxides After 60Co Irradiation or Fowler-Nordheim Injection", IEEE Transactions on Nuclear Science, vol. 40, no. 6, pp. 1341–1349, 1993.

63. **[SAMI04]** JJ. Wang, et. el. “Total Ionizing Dose Effects on Flash-based Field Programmable Gate Array”, IEEE Trans. on Nucl. Sci., Vol. 51, No. 6, December 2004.

64. **[SAND02]** Sandi Habinc, “Functional Triple Modular Redundancy”, Design and assessment report, Gaisler Research, December 2002.

65. **[SANT03]** A.C.F. Santos, A Frohlich, “Collision Cross Section and the size of a coin”, 2003 Phys. Educ. 38 336-339.

66. **[SCHW07]** J. R. Schwank, F. W. Sexton, M. R. Shaneyfelt, D. M. Fleetwood,” Total Ionizing Dose Hardness Assurance Issues for High Dose Rate Environments”, IEEE Trans. on Nucl. Sci, Vol. 54, No. 4, August 2007.

67. **[SCHW83]** J. Schwank, W. Dawes, “Irradiated silicon gate MOS device bias annealing”, IEEE Trans. Nucl. Sci. Vol. NS-30, No. 6, Dec 1983

68. **[SCHW92]** J. R. Schwank, D. M. Fleetwood, M. R. Shaneyfelt, P. S. Winokur, C. L. Axness, and L. C. Riewe, “Latent Interface-Trap Buildup and Its Implications for Hardness Assurance”, IEEE Transactions on Nuclear Science, vol. 39, no. 6, pp. 1953–1963, Dec. 1992.

69. **[SEXT85]** F. W. Sexton and J. R. Schwank, “Correlation of Radiation Effects in Transistors and Integrated-Circuits”, IEEE Transactions on Nuclear Science, vol. 32, no. 6, pp. 3975–3981, Dec. 1985.

70. **[SHAN90]** M.R. Shaneyfelt, J.R. Schwank, D.M. Fleetwood, P.S. Winokur, K.L. Hughes, F.W. Sexton, “ Field dependence of Interface-trap buildup in Polysilicon and Metal Gate MOS devices” IEEE Trans on Nucl Sci, Vol 37, No. 6, December 1990.

71. **[SHAN91]** M.R. Shaneyfelt, J.R. Schwank, D.M. Fleetwood, K.L. Hughes, “ Charge yield for Cobalt-60 and 10-keV X-ray irradiations of MOS Devices”, IEEE Trans on Nucl Sci, Vol. 38, No. 6, December 1991.

72. **[SMIT94]** Ed Smith “Effects of Realistic Satellite Shielding on SEE Rates”, *IEEE Trans. on Nuclear Science*, vol 41, no. 6, pp. 0018-9499, Dec. 1994.

73. **[SNOE99]** W.J. Snoeys, T.A.P. Gutierrez and G. Anelli,” A New NMOS Layout Structure for Radiation Tolarance”, IEEE Trans. Nucl. Sci, Vol. 49, No. 4, August 1999.

74. **[SNYD89]** E.S. Snyder, P.J. McWhorter, T.A. Dellin, J.D. Sweetman, "Radiation response of Floating Gate EEPROM Memory Cells", IEEE Trans. On Nucl. Sci., Vol. 36, No. 6, December 1989

75. **[SPAC96]** "SPACE RADIATION EFFECTS ON ELECTRONIC COMPONENTS IN LOW-EARTH ORBIT" NASA, PRACTICE NO. PD-ED-1258, April 1996.

76. **[SROU03]** J.R. Srour, C.J. Marshall, P.W Marshall, "Review of Displacement Damage Effects in Silicon Devices", IEEE Trans. Nucl. Sci., Vol. 50, No.3, June 2003.

77. **[SROU82]** J.R. Srour, "Basic Mechanism of Radiation Effects on electronic Materials, Devices, and Integrated Circuits", Technical Report, Defense Nuclear Agency, Washington, 1982

78. **[STAN85]** T. Stanley, D. Neaman, P. Dressendorfer, J. Schwank, P. Winokur, M. Ackermann, K.Jungling, C. Hawkins, W. Grannemann, " The effect of operating frequency in the radiation induced buildup of trapped holes and interface states in MOS devices", IEEE Trans. Nucl. Sci. Vol NS-32, No. 6, Dec 1985.

79. **[STAS88]** E.G. Stassinopoulos, J.P. Raymond, "The Space Radiation Environment for Electronics", Proceedings of the IEEE, Vol. 76, No. 11, November 1988.

80. **[STER]** L. Sterpone, M. Violante, "Analysis of the robustness of the TMR architecture in SRAM-based FPGAs", Politecnico di Torino, Dip. Automatica e Informatica, C.so Duca degli Abruzzi 24, 10129 Torino, Italy

81. **[STES96]** A. Stesmans and V. V. Afanasev, "Thermally Induced Interface Degradation in (111) Si/SiO2 Traced by Electron Spin Resonance", Physical Review B, vol. 54, no. 16, pp. R11 129–132, 15 Oct. 1996.

82. **[STET04]** M. Stettler, M. Caffrey, P. Graham, J. Krone, " Radiation effects and mitigation strategies for modern FPGAs", 10th annual workshop for the LHC and future experiments, Los Almos National Laboratory, USA, 2004.

83. **[TPMA89]** T.P. Ma, "Interface trap transformation in radiation or hot-electron damaged MOS structures", Semicond. Sci. Technol. 4 (1989) 1061 -1079.

84. **[USRA]** U.S. Centennial of Flight Commission, http://www.centennialofflight.gov/essay/Dictionary/RADIATION_BELTS/DI160.htm

85. **[VARG94]** F. Vargas and M. Nicolaidis, "SEU-tolerant SRAM design based on current monitoring", 24th International Symposium on Fault Tolerant Computing, pages 106 – 115, June 1994.

86. **[VELA94]** R. Velazco, D. Bessot, "Two CMOS memory cells suitable for the design of SEU-tolerant VLSI circuits", IEEE Trans. on Nuc. Sci, vol. 41(6), pages 2229 – 2234, 1994.

87. **[VENB93]** J. Venbrux et. el., "Designing and testing of SEU/SEL immune memory and logic circuits in a commercial CMOS process", Record of the 1993 Radiation Effects Data workshop, pages 51 – 55, July 1993.

88. **[WANG04]** J.J. Wang, "Radiation effects in FPGA's", Actel Corporation, 11 May 2004.

89. **[WANG99]** J.J. Wang, R.B. Katz, J.S. Sun, B.E. Cronquist, J.L. McCollum, "SRAM Based Re-programmable FPGA for Space Applications", IEEE Trans. on Nucl. Sci., Vol. 46, NO. 6, December 1999.

90. **[WHIR03]** M. Wirthlin, N. Rollins, M. Caffrey, and P. Graham, "Hardness By Design Techniques for Field Programmable Gate Arrays", Technical Report, Department of Electrical and Computer Engineering, Brigham Young University, Provo, UT, 2003.

91. **[WHIT]** S. Whitaker, "Single event upset hardening CMOS memory circuit", US Patent No 5,111,429.

92. **[ZUKA02]** h.dr. V.Gavryushin, h.dr. A.Žukauskas, "Functional combinations in solidstates**",**http://www.mtmi.vu.lt/pfk/funkc_dariniai/transistor/mos_capacitors.htm

Appendix A

Schematics for Fig 4.12 and 4.13

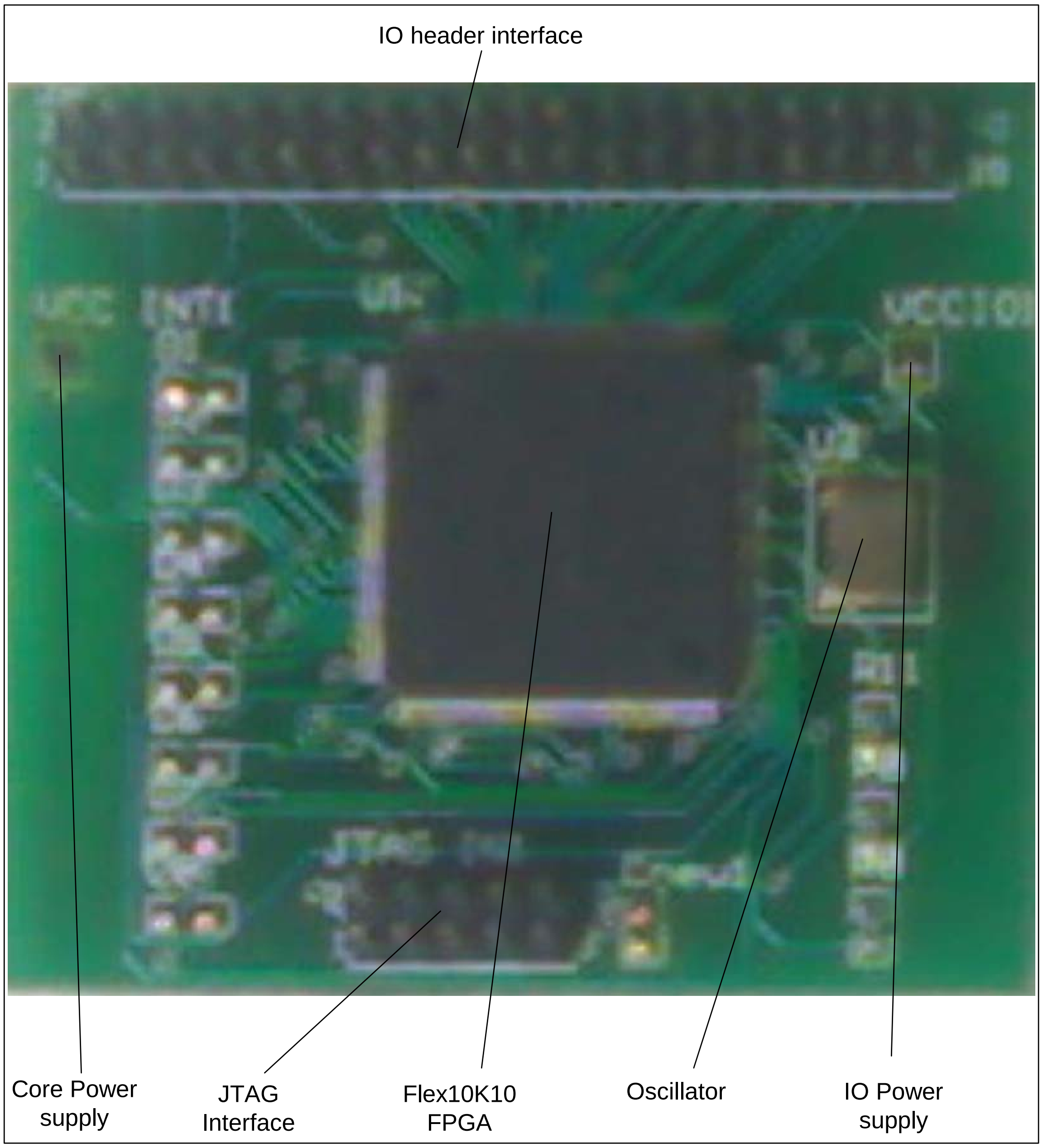
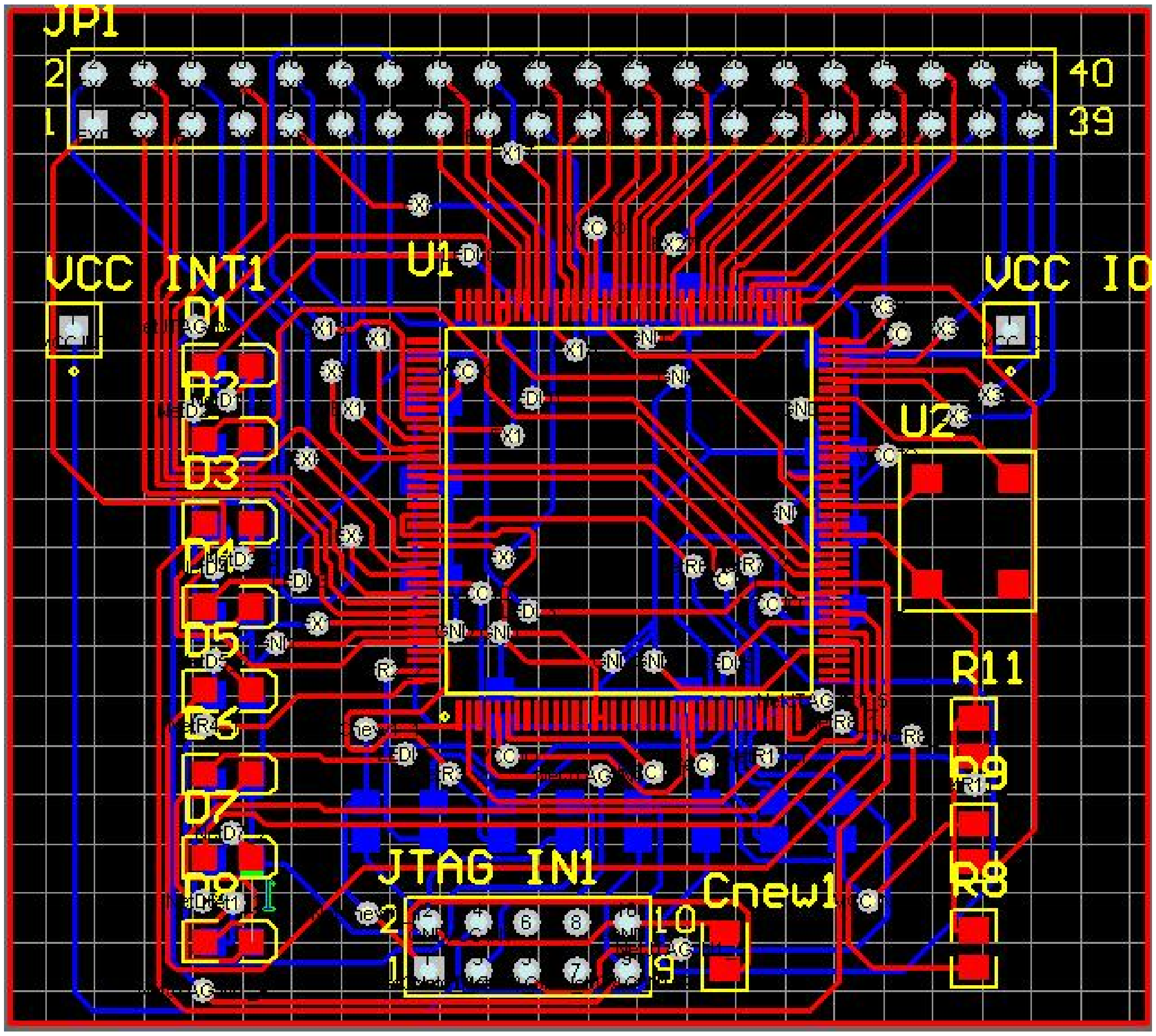
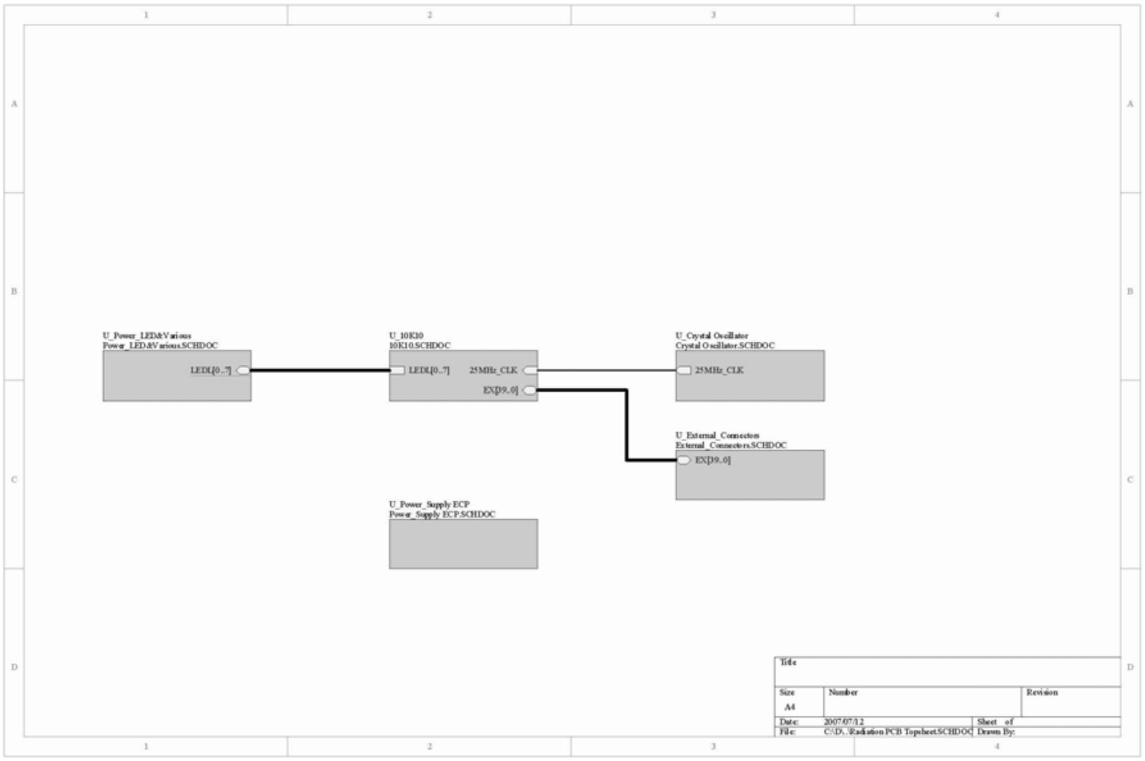
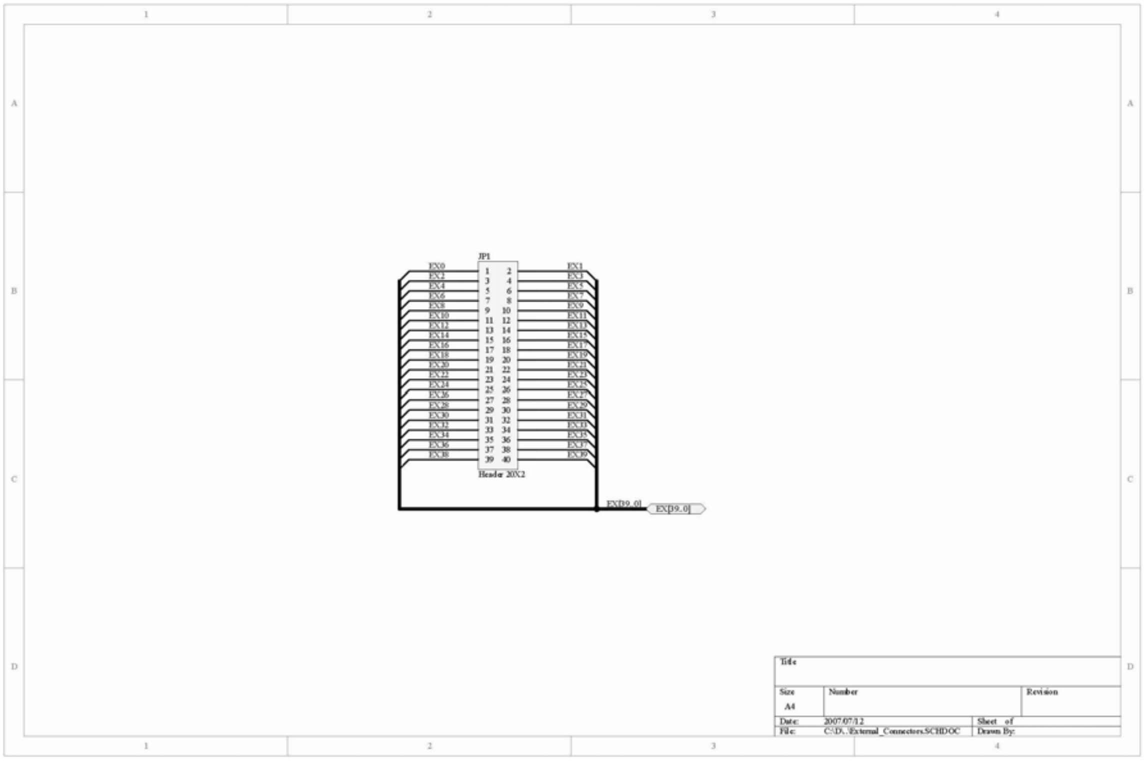
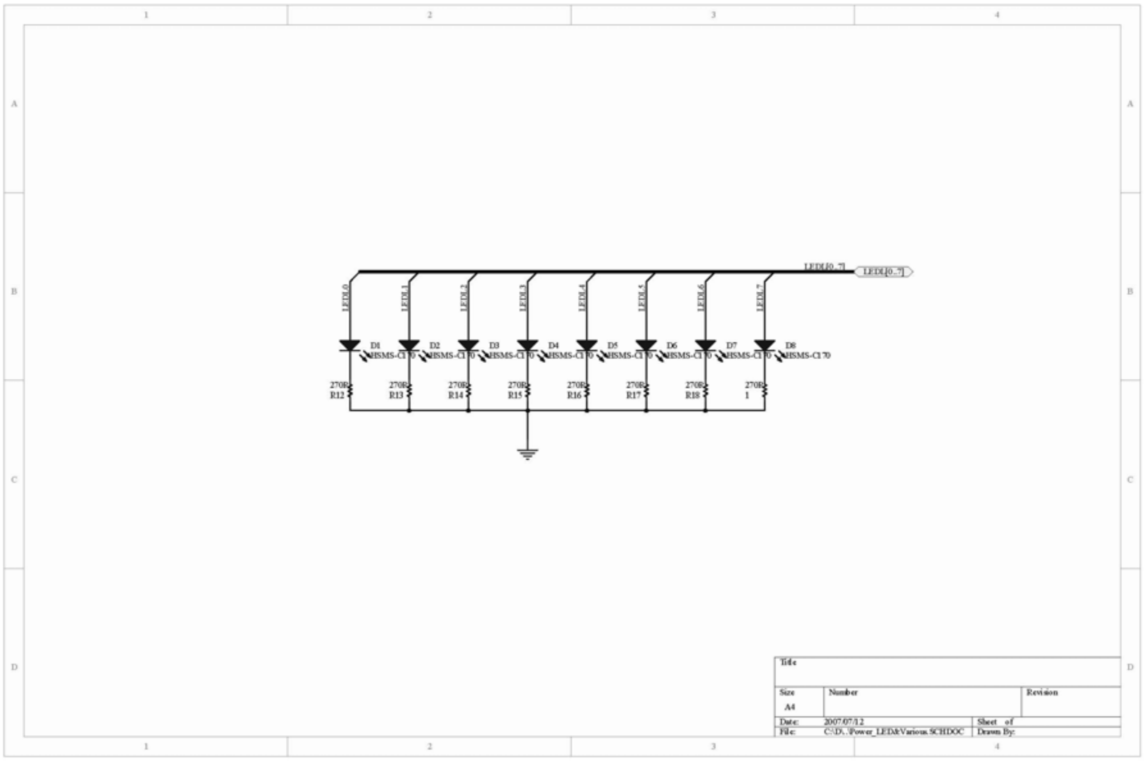


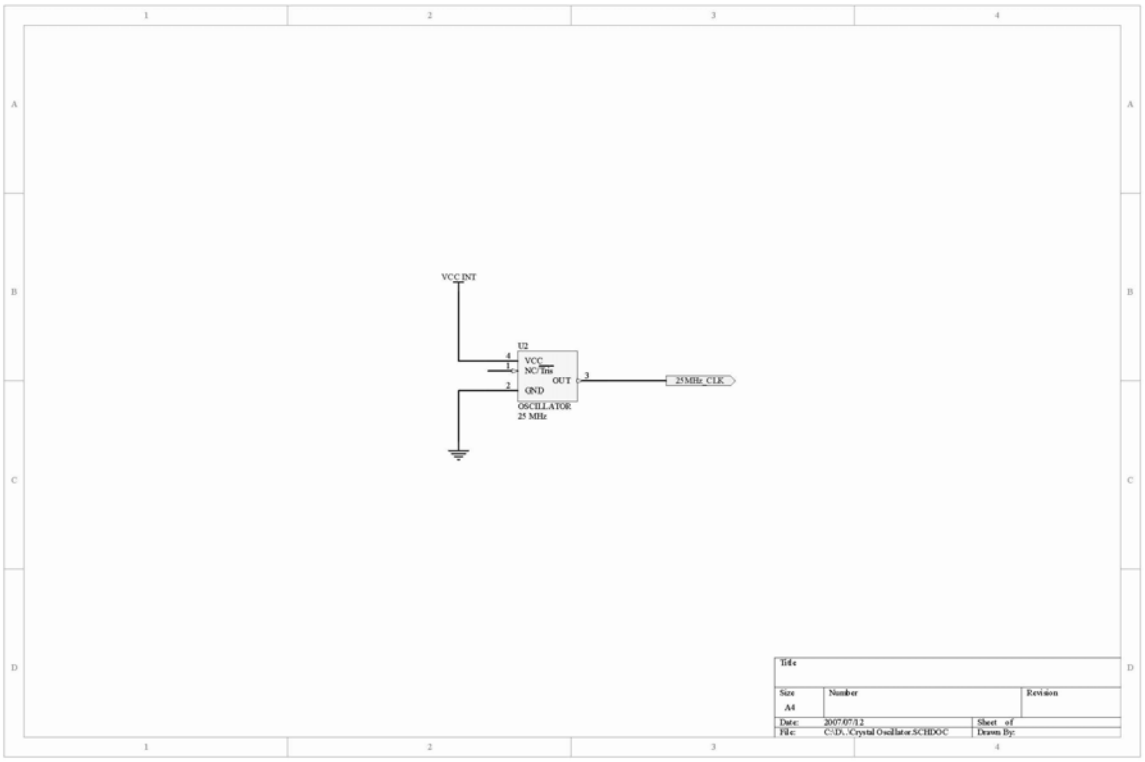
Fig. 4.12 FPGA PCB test board with IO and Core power supply separated

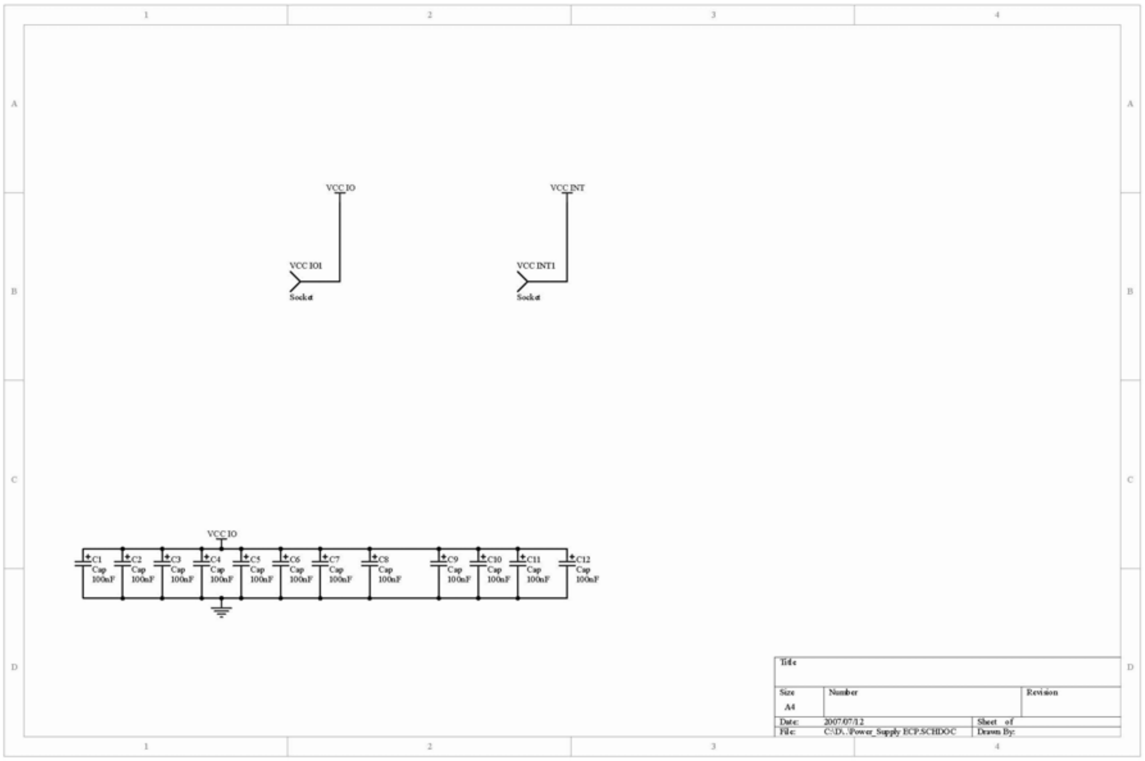

JP1
2
1
40
39
VCC INT1
VCC IO
U1
U2
D1
D2
D3
D4
D5
D6
D7
D8
D9
R11
R8
JTAG IN1
Cnew1
10
9

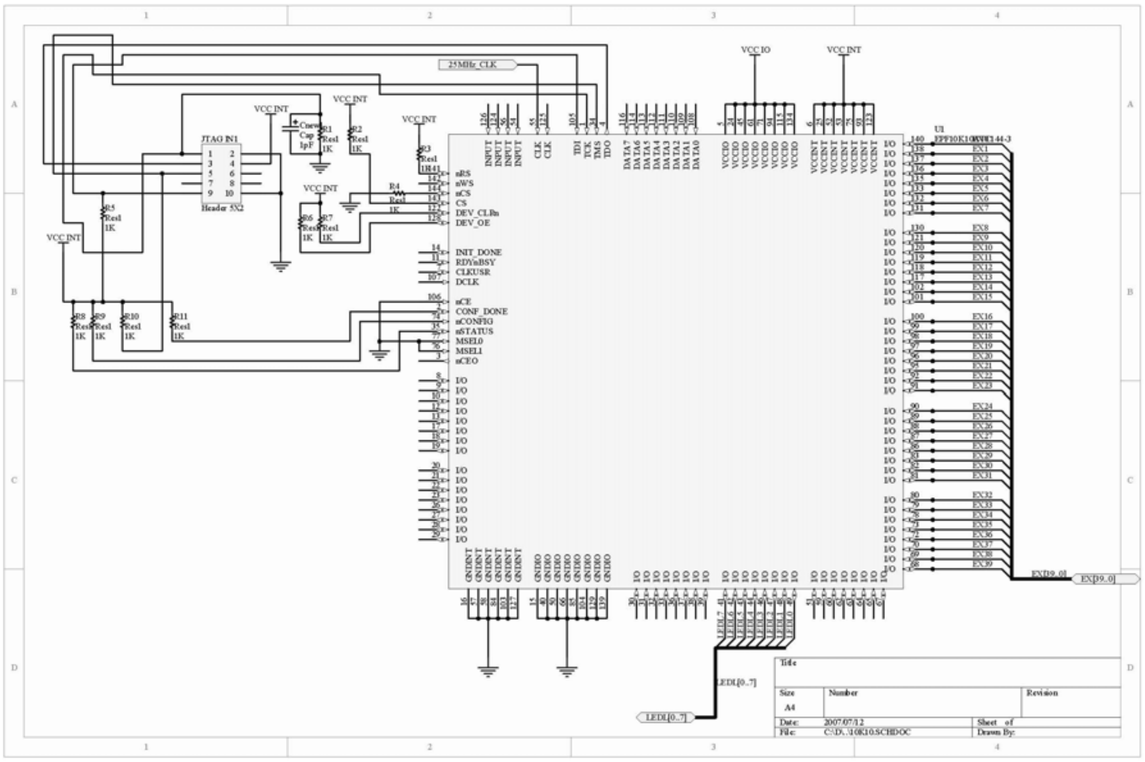

U_Power_LED&Various
Power_LED&Various.SCHDOC
LEDL[0..7]
U_10K10
10K10.SCHDOC
LEDL[0..7]
25MHz_CLK
EX[39..0]
U_Crystal Oscillator
Crystal Oscillator.SCHDOC
25MHz_CLK
U_External_Connectors
External_Connectors.SCHDOC
EX[39..0]
U_Power_Supply ECP
Power_Supply ECP.SCHDOC
Title
Size
A4
Number
Revision
Date: 2007/07/12
Sheet of
File: C:\D..\Radiation PCB Topsheet.SCHDOC
Drawn By:


JP1
Header 20X2
EX0
EX1
EX39
EX[39..0]
EX[39..0]
Title
Size
A4
Number
Revision
Date: 2007/07/12
Sheet of
File: C:\D..\External_Connectors.SCHDOC
Drawn By:


LEDL[0..7]
LEDL0
LEDL1
LEDL2
LEDL3
LEDL4
LEDL5
LEDL6
LEDL7
D1 HSMS-C170
D2 HSMS-C170
D3 HSMS-C170
D4 HSMS-C170
D5 HSMS-C170
D6 HSMS-C170
D7 HSMS-C170
D8 HSMS-C170
270R R12
270R R13
270R R14
270R R15
270R R16
270R R17
270R R18
270R 1
Title
Size A4
Number
Revision
Date: 2007/07/12
Sheet of
File: C:\D..\Power_LED&Various.SCHDOC
Drawn By:


VCC INT
U2
VCC
NC/Tris
OUT
GND
OSCILLATOR
25 MHz
25MHz_CLK
Title
Size A4
Number
Revision
Date: 2007/07/12
Sheet of
File: C:\D..\Crystal Oscillator.SCHDOC
Drawn By:


VCC IO
VCC INT
VCC IO1
Socket
VCC INT1
Socket
C1 Cap 100nF
C12 Cap 100nF
Title
Size A4
Number
Revision
Date: 2007/07/12
File: C:\D..\Power_Supply ECP.SCHDOC
Sheet of
Drawn By:


25MHz_CLK
JTAG IN1
Header 5X2
VCC INT
VCC IO
U1
EPF10K10TC144-3
nRS
nWS
nCS
CS
DEV_CLRn
DEV_OE
INIT_DONE
RDYnBSY
CLKUSR
DCLK
nCE
CONF_DONE
nCONFIG
nSTATUS
MSEL0
MSEL1
nCEO
EX[39..0]
LEDI[0..7]
Title
Size A4
Number
Revision
Date: 2007/07/12
File: C:\D..\10K10.SCHDOC
Sheet of
Drawn By:

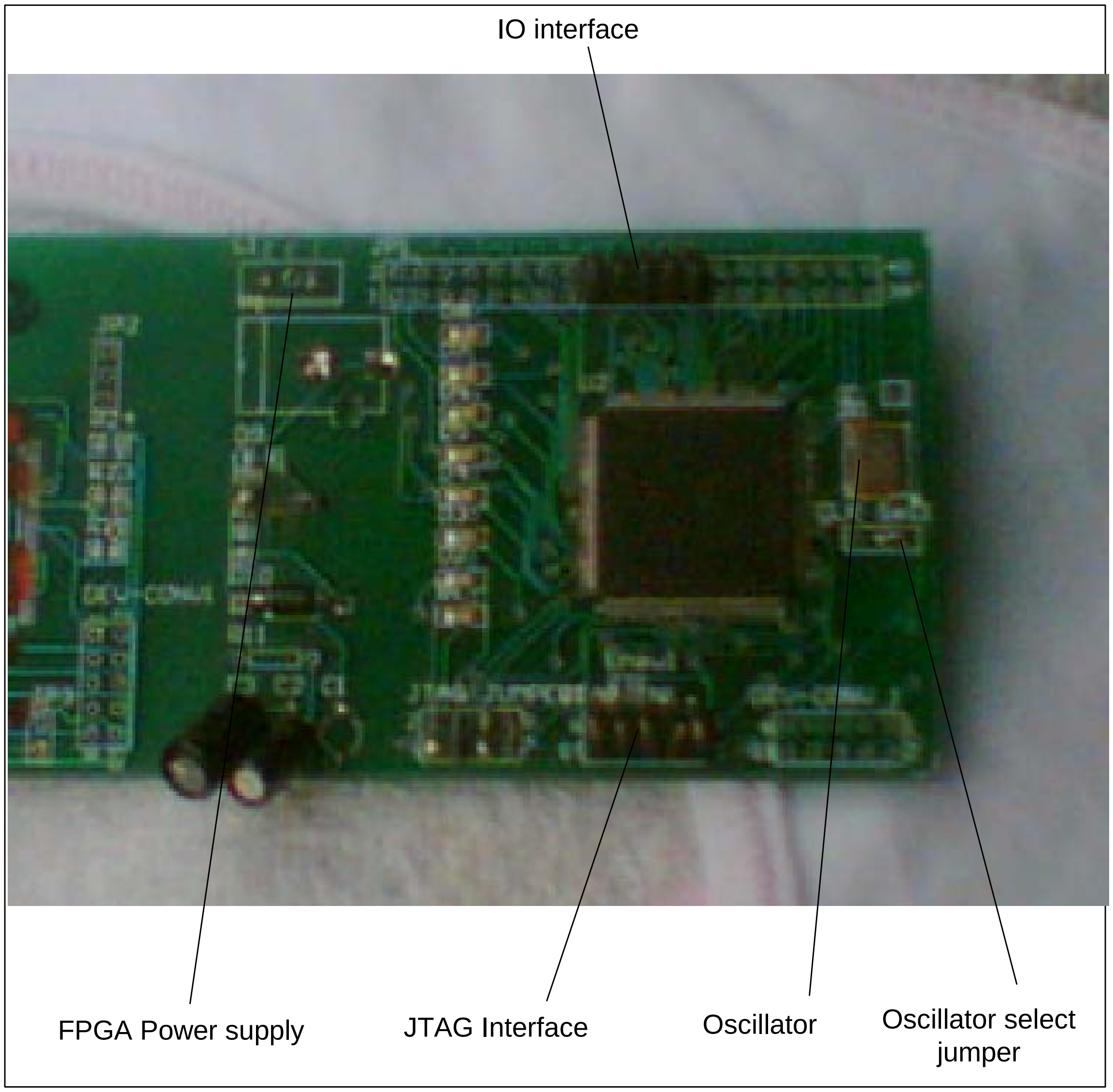
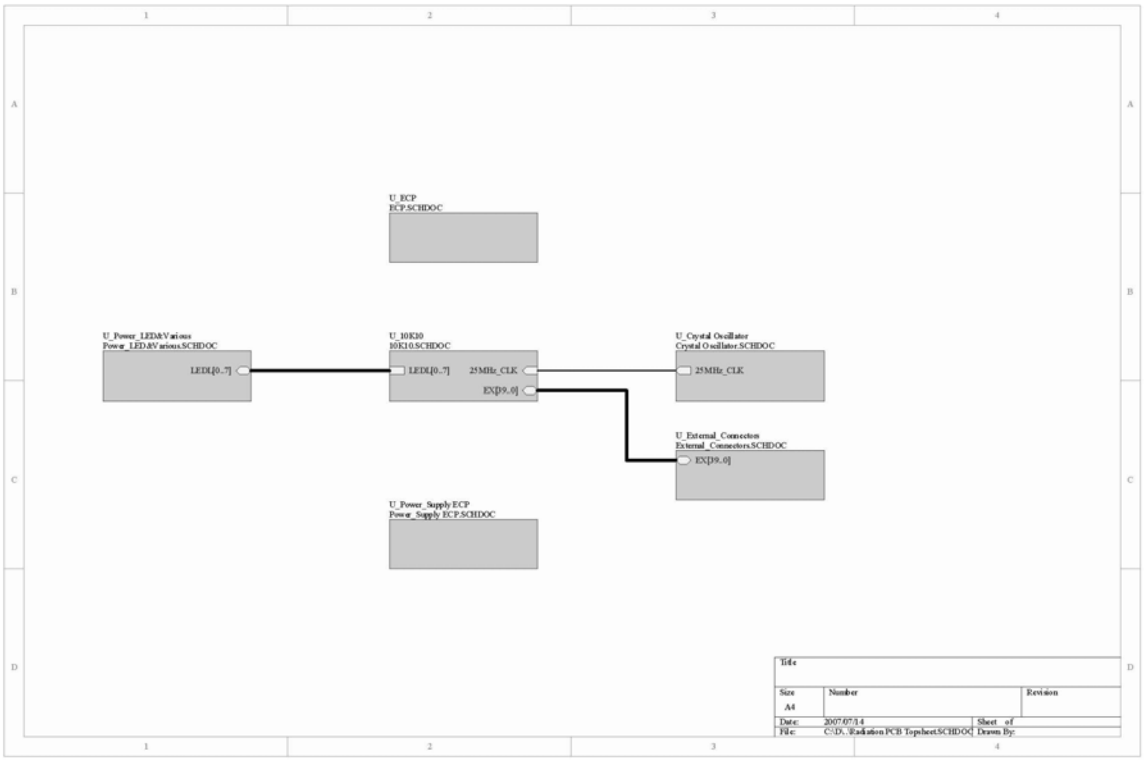
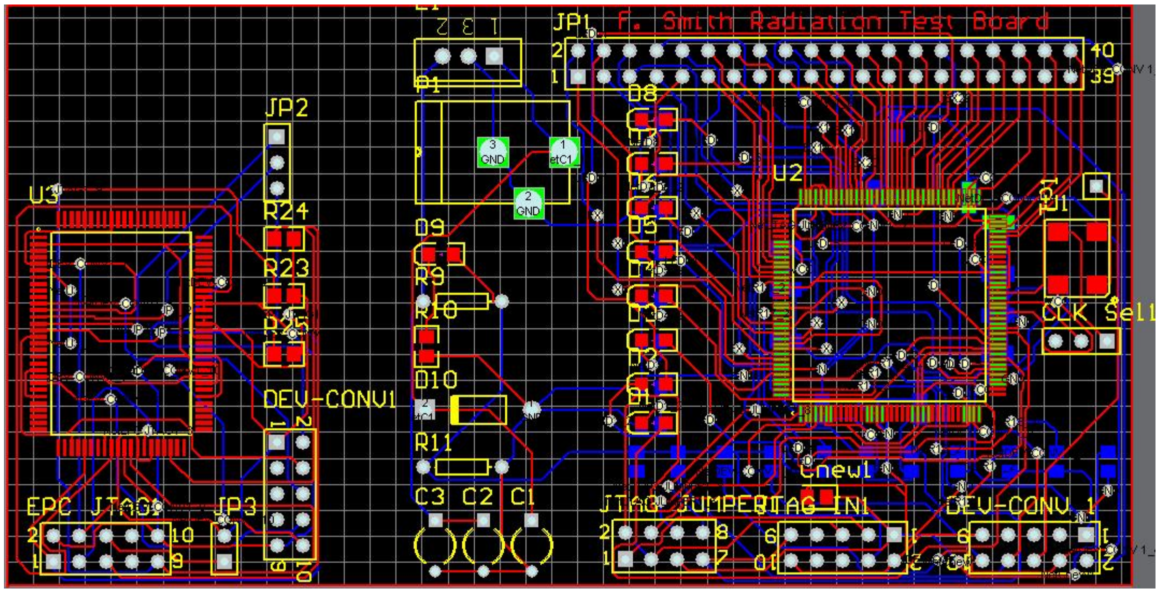
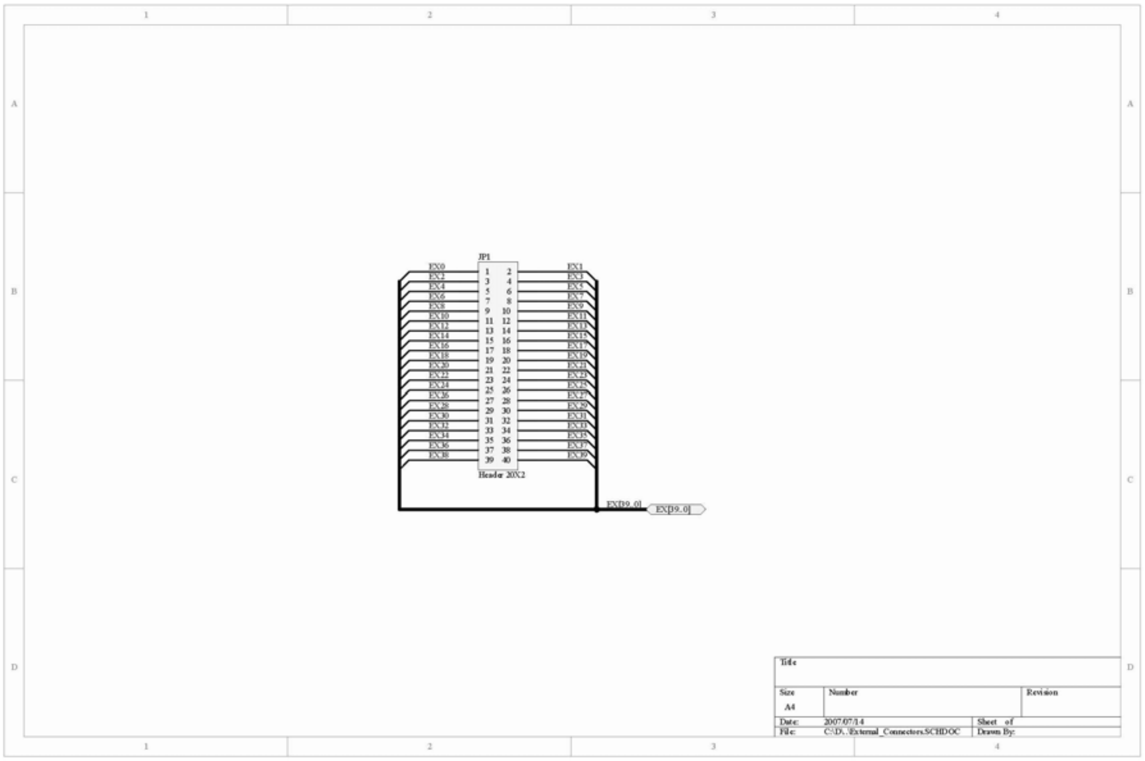


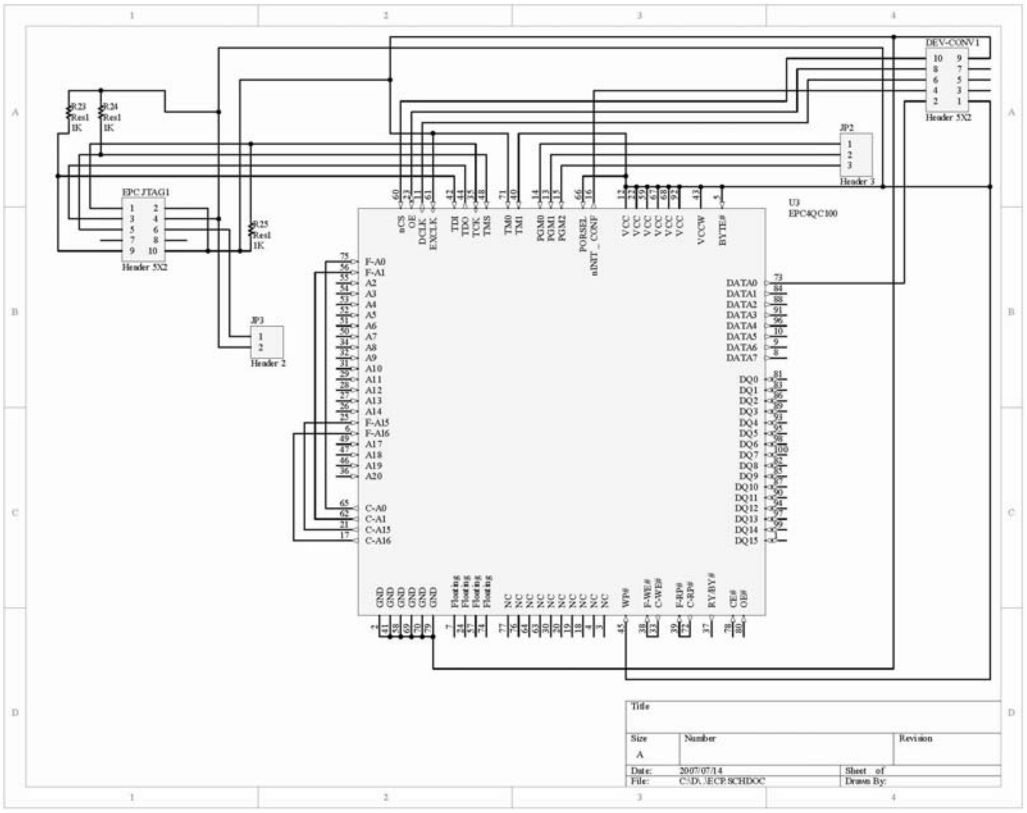
Fig. 4.13 FPGA PCB test board with clock signal select jumper


U_ECP
ECP.SCHDOC
U_Power_LED&Various
Power_LED&Various.SCHDOC
LED[0..7]
U_10K10
10K10.SCHDOC
25MHz_CLK
EX[9..0]
U_Crystal Oscillator
Crystal Oscillator.SCHDOC
U_External_Connectors
External_Connectors.SCHDOC
U_Power_Supply ECP
Power_Supply ECP.SCHDOC
Title
Size A4
Number
Revision
Date: 2007/07/14
File: C:\D..\Radiation PCB Topsheet.SCHDOC
Sheet of
Drawn By:

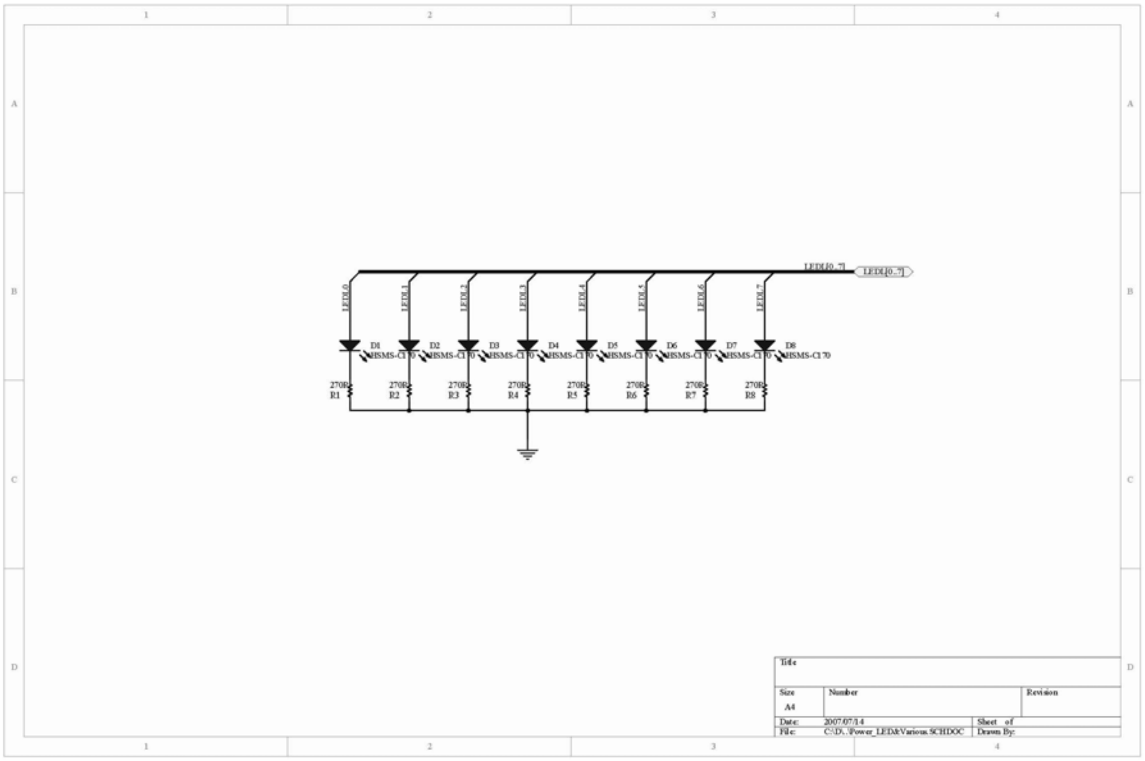

F. Smith Radiation Test Board
JP1
JP2
JP3
U2
U3
R24
R23
D9
D10
R9
R10
R11
D8
DEV-CONV1
DEV-CONV_1
EPC JTAG
JTAG JUMPER
JTAG IN1
CLK Sel1
Cnew1
C1
C2
C3
P1
GND

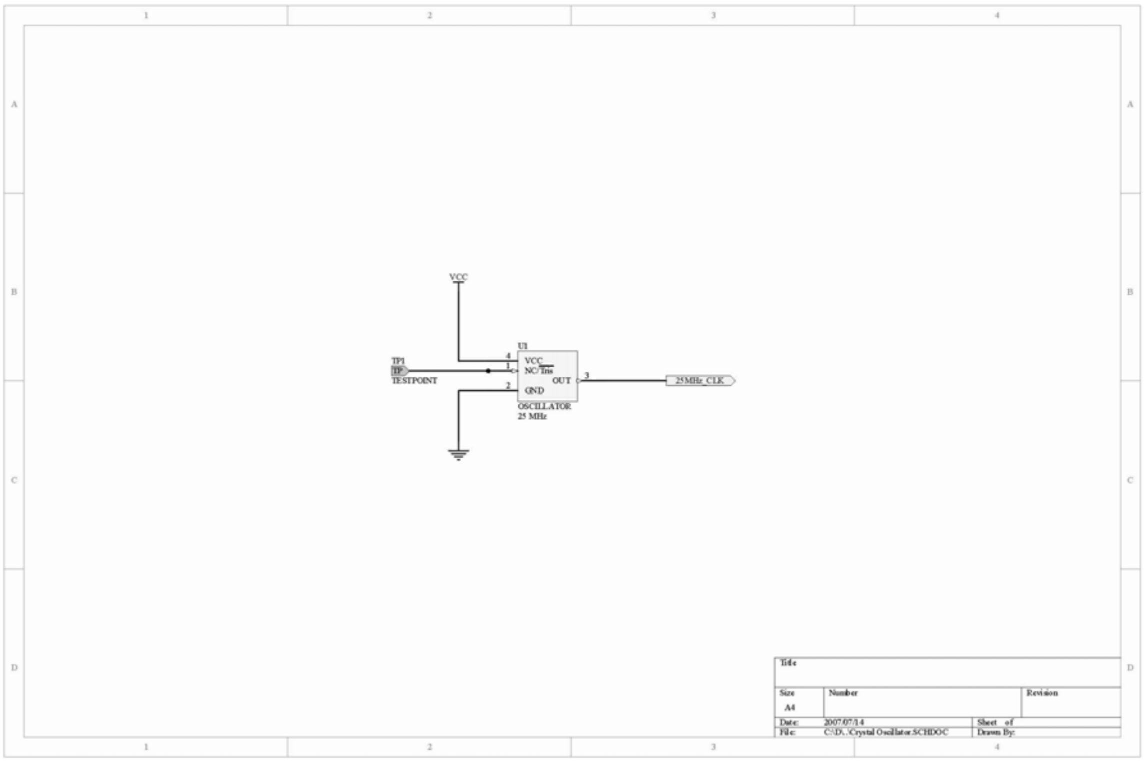


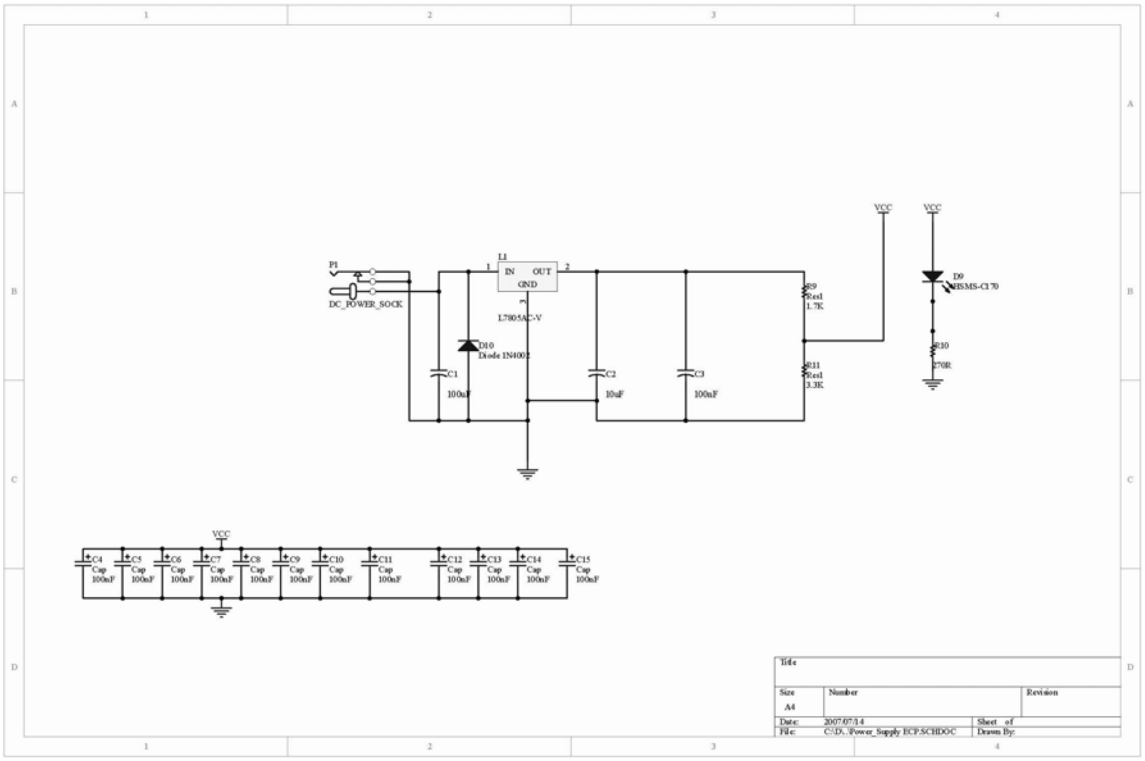
JP1
Header 20X2
EX[39..0]
Title
Size
A4
Number
Revision
Date: 2007/07/14
Sheet of
File: C:\D..\External_Connectors.SCHDOC
Drawn By:



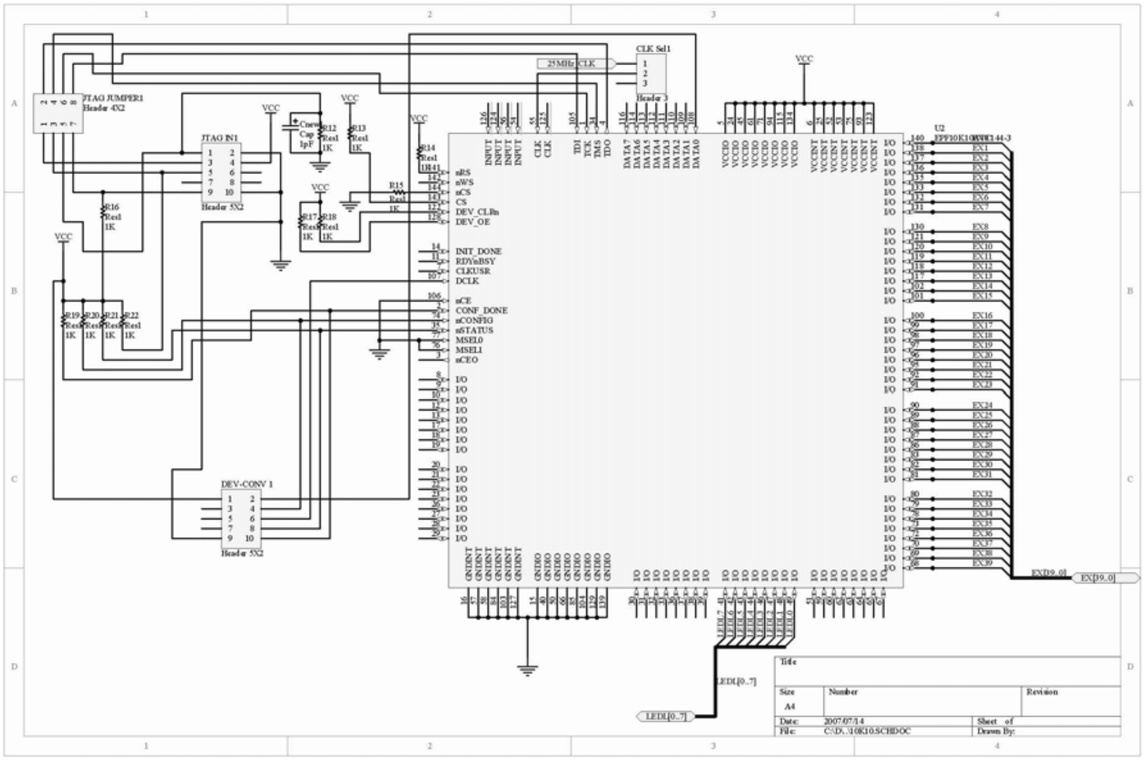
DEV-CONV1
Header 5X2
R23 Res1 1K
R24 Res1 1K
R25 Res1 1K
EPC JTAG1
Header 5X2
JP2
Header 3
JP3
Header 2
U3
EPC4QC100
nCS
OE
DCLK
EXCLK
TDI
TDO
TCK
TMS
TM0
TM1
PGM0
PGM1
PGM2
PORSEL
nINIT_CONF
VCC
VCCW
BYTE#
DATA0
DATA1
DATA2
DATA3
DATA4
DATA5
DATA6
DATA7
DQ0
DQ15
F-A0
F-A1
A2
A20
C-A0
C-A1
C-A15
C-A16
GND
Floating
NC
WP#
F-WE#
C-WE#
F-RP#
C-RP#
RY/BY#
CE#
OE#
Title
Size
A
Number
Revision
Date: 2007/07/14
Sheet of
File: C:\D..\ECP.SCHDOC
Drawn By:


LEDL[0..7]
LEDL0
LEDL1
LEDL2
LEDL3
LEDL4
LEDL5
LEDL6
LEDL7
D1
D2
D3
D4
D5
D6
D7
D8
HSMS-C170
270R
R1
R2
R3
R4
R5
R6
R7
R8
Title
Size
A4
Number
Revision
Date: 2007/07/14
Sheet of
File: C:\D..\Power_LED&Various.SCHDOC
Drawn By:


VCC
U1
TP1
TP
TESTPOINT
VCC
NC/Tris
OUT
GND
OSCILLATOR
25 MHz
25MHz_CLK
Title
Size
A4
Number
Revision
Date: 2007/07/14
Sheet of
File: C:\D..\Crystal Oscillator.SCHDOC
Drawn By:



P1
DC_POWER_SOCK
L1
IN
OUT
GND
L7805AC-V
D10
Diode 1N4002
C1
100nF
C2
10uF
C3
100nF
R9
Res1
1.7K
R11
Res1
3.3K
VCC
D9
HSMS-C170
R10
270R
C4 Cap 100nF
C5 Cap 100nF
C15 Cap 100nF
Title
Size A4
Number
Revision
Date: 2007/07/14
File: C:\D..\Power_Supply ECP.SCHDOC
Sheet of
Drawn By:



JTAG JUMPER1
Header 4X2
JTAG IN1
Header 5X2
CLK Sel1
Header 3
25MHz_CLK
DEV-CONV 1
Header 5X2
U2
EPF10K10TC144-3
nRS
nWS
nCS
CS
DEV_CLRn
DEV_OE
INIT_DONE
RDYnBSY
CLKUSR
DCLK
nCE
CONF_DONE
nCONFIG
nSTATUS
MSEL0
MSEL1
nCEO
EX[39..0]
LED[0..7]
Title
Size A4
Number
Revision
Date: 2007/07/14
File: C:\D..\10K10.SCHDOC
Sheet of
Drawn By:

Appendix B

Ring Counter VHDL Code

```
-- Summary: This is the Selective Modular Redundancy (SMR)
-- algorithm test code written as a part of my PhD thesis.
--
-- Purpose: The program implements the SMR Algorithm on the VHDL code.
--
-- Author              Farouk Smith
--                     Electronic Systems Laboratory
--                     Department of Electronics Engineering
--                     University of Stellenbosch
--                     South Africa
-- email               fsmith@sun.ac.za
-- Time Stamp          October 2005(Cape Town)
-- Current Version     1.3
-- (C) 2005 Farouk Smith
-- Ring Counter

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY ring IS
        PORT (clock : in STD_LOGIC;
                      reset : in std_logic;
                      C0, C1, C2 : out std_logic);
END ENTITY ring;

ARCHITECTURE Behavior of ring IS
        TYPE state_type IS (s0, s1, s2, s3, s4, s5, s6, s7, s8);
        SIGNAL state: state_type;
BEGIN
        next_state_logic: process (clock, reset)
        BEGIN
                IF( reset = '0') THEN
                                state <= s8;
                ELSIF (clock'EVENT AND clock = '1') THEN
                CASE state IS
                        WHEN s0 =>
                        IF( reset = '1') THEN
                                state <= s1;
                        ELSE
                                state <= s8;
```

```
END IF;
WHEN s1 =>
IF( reset = '1') THEN
          state <= s2;
ELSE
          state <= s8;
END IF;
WHEN s2 =>
IF( reset = '1') THEN
          state <= s3;
ELSE
          state <= s8;
END IF;
WHEN s3 =>
IF( reset = '1') THEN
          state <= s4;
ELSE
          state <= s8;
END IF;
WHEN s4 =>
IF( reset = '1') THEN
          state <= s5;
ELSE
          state <= s8;
END IF;
WHEN s5 =>
IF( reset = '1') THEN
          state <= s6;
ELSE
          state <= s8;
END IF;
WHEN s6 =>
IF( reset = '1') THEN
          state <= s7;
ELSE
          state <= s8;
END IF;
WHEN s7 =>
IF( reset = '1') THEN
          state <= s8;
ELSE
          state <= s0;
END IF;
WHEN s8 =>
IF( reset = '1') THEN
          state <= s0;
```

```
                        ELSE
                                state <= s8;
                        END IF;
                        END CASE;
                END IF;
END process;

output_logic: process (state)
BEGIN
CASE state IS
        WHEN s0 =>
                C0 <= '0';
                C1 <= '0';
                C2 <= '0';
        WHEN s1 =>
                C0 <= '1';
                C1 <= '0';
                C2 <= '0';
        WHEN s2 =>
                C0 <= '0';
                C1 <= '1';
                C2 <= '0';
        WHEN s3 =>
                C0 <= '1';
                C1 <= '1';
                C2 <= '0';
        WHEN s4 =>
                C0 <= '0';
                C1 <= '0';
                C2 <= '1';
        WHEN s5 =>
                C0 <= '1';
                C1 <= '0';
                C2 <= '1';
        WHEN s6 =>
                C0 <= '0';
                C1 <= '1';
                C2 <= '1';
        WHEN s7 =>
                C0 <= '1';
                C1 <= '1';
                C2 <= '1';
        WHEN s8 =>
                C0 <= '0';
                C1 <= '0';
                C2 <= '0';
```

```
END CASE;
END process;
END Behavior;
```

Appendix C

```
/*
 * Summary: This is the Selective Modular Redundancy (SMR)
 * algorithm SEU simulation written in my PhD thesis.
 *
 * Purpose: The algorithm is a simulation model to study the number of SEU's
 * that occur as we change the target density, reaction distance, no of SRAM
 * cells in the matrix and the incoming cosmic ray particle flux.
 *
 * Constraints:
 *
 * Inputs: no_of_srams , reaction_distance,density, Io.
 * Outputs: No of SEU's, cross section.
 *
 * Author           Farouk Smith
 *                      Electronic Systems Laboratory
 *                      Department of Electronics Engineering
 *                      University of Stellenbosch
 *                      South Africa
 * email                fsmith@sun.ac.za
 * Time Stamp         April 2007(Cape Town)
 * Version 1.1
 * (C) 2005 Farouk Smith
 */

#include <stdlib.h>
#include <string.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <math.h>

void Usage(char *programName)
{
        fprintf(stderr,"%s usage:\n",programName);
        /* Modify here to add your usage message when the program is
         * called without arguments */
}

/* returns the index of the first argument that is not an option; i.e.
  does not start with a dash or a slash
*/
int HandleOptions(int argc,char *argv[])
{
	int i,firstnonoption=0;

	for (i=1; i< argc;i++) {
		if (argv[i][0] == '/' || argv[i][0] == '-') {
			switch (argv[i][1]) {
				/* An argument -? means help is requested */
				case '?':
					Usage(argv[0]);
					break;
				case 'h':
				case 'H':
					if (!stricmp(argv[i]+1,"help")) {
						Usage(argv[0]);
						break;
					}
					/* If the option -h means anything else
					 * in your application add code here
					 * Note: this falls through to the default
					 * to print an "unknow option" message
					*/
				/* add your option switches here */
				default:
					fprintf(stderr,"unknown option %s\n",argv[i]);
					break;
			}
		}
		else {
			firstnonoption = i;
			break;
		}
	}
	return firstnonoption;
}

int main(int argc,char *argv[])
{
	/*
	if (argc == 1) {
		// If no arguments we call the Usage routine and exit
		Usage(argv[0]);
		return 1;
	}
	*/
	/* handle the program options */
	HandleOptions(argc,argv);
```

```
        /* The code of the application follows */

        int val;
        int i,j,k,scanned,l,m,h;
        int temp[10000];
        double maxrand;        // The maximum number that the rand() funtion can output.
        int seu;
        int noseu;
        float cross;
        float unit_length;     // unit lenght of the matrix, used to determine the density
                               //and Io flux.
        float Io_flux;                  // This is the input flux of particles
        float Io;                       //(i.e. number of incident particles).
        float reaction_distance;        // Reaction distance (i.e. the max distance a particle
                                        //should be from a SRAM cell to cause a SEU).
        int no_of_srams;                // Represents the number of SRAM cells in the
                                        //memory matrix.
        float density;                  // The number of SRAM cells per unit area. The
                                        //higher this number, the lower the density.
//      float matrix_size;              // The size of the matrix with respect to the total
                                        //size of the SRAM cells. For example the density
                                        //can stay the same, but the matrix size can increase.
                                        //Thus the SRAM cells will occupy a small
                                        //congested part of the space.
        struct {
                        char name;
                        double x,y,r;
                        double x_position, y_position; // Represents the position of the
                                                        //SRAM cells in the memory matrix.
                        } sram[20000];

        FILE *fp;
   fp = fopen("temp.txt", "w+r");       // Create and open a temporary file to write the
                                        //output.
        srand( (unsigned int)time( NULL ) );
        maxrand = 32767;

/*********************************************************************/
//These are the Parameters that can be changed.
//      Io = 5000;
        unit_length = 1;
        Io_flux = 20000;
        no_of_srams = 500;
        reaction_distance = 0.01;
        density = 1;
```

```
/*************************************************************************/
        cross = 0;
//      float averageseu = 0;
//for(h=0;h<100;h++)
//      {
//      density = density - 0.01;
//      int totalseu = 0;
        for(m=0;m <300;m++)
                {

                        no_of_srams = no_of_srams + 100;
//                      printf("matrix size : %3.6f\n", matrix_size);
//                      unit_length = unit_length + 1;
//                      density = density - 0.1;
//                      unit_length = no_of_srams/density;
                        Io = Io_flux*unit_length;
//                      no_of_srams = density*unit_length;
//                      density = no_of_srams/unit_length;
//                      reaction_distance = reaction_distance + 0.01;

        for(k=0; k< no_of_srams; k++)           // Create sram cells that equal no_of_srams
                                                //spaced equally.
                {
                        sram[k].x_position = density*k;
                        sram[k].y_position = density*k;
//                      fprintf(fp, "%3.10f,%3.10f\n",sram[k].x_position,
sram[k].y_position);
                }
        Io = density*Io;
        for(k=0; k<Io; k++)
                {
                                sram[k].x = density*no_of_srams*rand()/maxrand;
                // Create a floating random number between 0 and no_of_srams.
                                sram[k].y = density*no_of_srams*rand()/maxrand;
//                              sram[k].r = sqrt(sram[k].x*sram[k].x +
sram[k].y*sram[k].y);
//                      fprintf(fp,"%3.10f\n", sram[k].x);      // send the random number to
                                                                // a file.
                }

        seu = 0;
        noseu = 0;
        for (k=0; k<no_of_srams; k++)           // For each SRAM position, check if a
                                                //incident cosmic particle struct the SRAM.
                {
        // If the cosmic particle is within a distance of reaction_distance of the SRAM
        //then a SEU occurs. The radius of interaction is given by reaction_distance.
                for (j= 0; j<Io; j++)
        {
                                if((sram[j].x < (sram[k].x_position +
reaction_distance))&&(sram[j].x > (sram[k].x_position - reaction_distance)))
                                {
                                        for(l=0; l<Io; l++)
                                        {
                                        if((sram[l].y < (sram[k].x_position +
reaction_distance))&&(sram[l].y > (sram[k].x_position - reaction_distance)))
                                                {
                                                //      scanned = 1;
                                                seu++;
                                                }
                                        }
                                /*
                                        if(scanned == 1)
                                                {
                                                        break;
                                                }
                                */
                                }
                                else
                                {
                                //      scanned = 0;
                                noseu++;
                                }

                        }
/*
                        if(scanned == 1)
                                {
                                                seu++;
                                }
                        else
                                {
                                                noseu++;
                                }
*/
                }

        cross = seu/(density*Io);
        fprintf(fp, "%d,%d,%3.10f\n",no_of_srams, seu, cross);         // send the random
                                                                        // number to a file.
//      totalseu = totalseu + seu;
  }
```

```
//	averageseu = totalseu/100;
//	fprintf(fp,"%2.2f,%2.2f\n",density, averageseu);
//	printf("%d,%2.2f\n",h,averageseu);

//}
	return 0;
}
```

Appendix D

```
/*
 * Summary: This is the Selective Modular Redundancy (SMR)
 * algorithm written as a part of my PhD thesis. This is a gate level
 * version of the algorithm. The original code was written by Praveen Samudrala as part
 * of his MSc thesis on Selective Triple Modular Redundancy and modified for the SMR.
 * Purpose: The program implements the SMR Algorithm on the VHDL code.
 *
 * Constraints: The input VHDL file has to be in a definite order for the
 * algorithm to scan. The VHDL file must be in the structural format as
 * obtained from the Synopsys FPGA Compiler II software.
 *
 * Inputs: Structural VHDL file name.
 * Outputs: SMR file.
 *
 * Author             Farouk Smith
 *                         Electronic Systems Laboratory
 *                         Department of Electronics Engineering
 *                         University of Stellenbosch
 *                         South Africa
 * email                   fsmith@sun.ac.za
 * Time Stamp         October 2005(Cape Town)
 * Modified Ver 1.2     November 2005 for Synopsys FPGA Compiler II
 *                         Structural VHDL Input.
 * Modified Ver 1.2.1 December 2005 to include the SMR Algorithm.
 *
 * Version      1.3     Modified to delete the technology dependant components
 *                         from the MAXPLUS EDIF file, February 2006.
 * Version       1.4       Modified to include the user defined libraries created
 *                         for the fundamental primitive gates. March 2006.
 * (C) 2005 Farouk Smith
 *
 *
 * NOTE: For more information on the SMR method read
 *
 *
 * STDMR and SMR has patent pending as of August 2005
 */

#include "stdmr.h"

namespace Compo
{
```

```
/*
 * This is a Function that counts the number of components from the VHDL netlist
 * The this is done to obtain the structure of the components, and the number of
 * inputs and outputs. The result is stored in gates.num_inputs.
 */

void num_compo (char* cktname)
{
 char start[MAXLENGTH] ;
 char cnamefield[MAXLENGTH] ;
 int file_start, file_count;
 char buf[MAXLENGTH];
 char buf_temp[MAXLENGTH];

 /*
  * Open the input vhdl file for reading
  */

 FILE *fp;
 fp = fopen(cktname,"r");

 if(NULL == fp)
   {
       printf("Error Reading input file!\n");
       exit(1) ;
   }

 /*
  * Scan the VHDL input file; start storing the values after
  * the last "entity" statement. This is done by first counting
  * the number of "entity" statements in the file and storing
  * the value in entity_count. The file is then closed and opened
  * again so that we know at which position the last "entity"
  * statement is, and can now start counting the number of components
  * in the main VHDL entity.
  */

 file.num_entity = 0;
 file.end_entity = 0;
 int help = 0;
 int entity_comp;
 int entity_count = 0;
 do{
       entity_comp = fscanf(fp,"%s",&start);
       if(!strcmp(start,"entity"))
         {
          file.num_entity++;
         }
       else if(!strcmp(start, "end"))
         {
          file.end_entity++;
         }
       else
       {
       }
  }while( entity_comp != EOF);

 fp = fopen(cktname,"r");

 while(1)
  {
       fscanf(fp,"%s",&start);
       if(!strcmp(start,"entity"))
         {
          entity_count++;
         }
       if(entity_count == file.num_entity)
       {
          break;
       }
  }

  while(1)
       {
               fscanf(fp, "%s", &buf_temp);
               if(!strcmp(buf_temp,"is"))
               {
                       break;
               }
       }

       /*
        * After the "is" statement, start reading the first component.
     */

    do{
       entity_comp = fscanf(fp,"%s",&start);
       if(!strcmp(start,"component"))
         {
          file.num_component++;
         }
       else if(!strcmp(start, "end"))
```

```
              {
                file.end_component++;
              }
            else
            {
            }
        }while( entity_comp != EOF);

        printf("Number of entities : %d\n", file.num_entity);
        printf("Number of components : %d\n", file.num_component);

         fp = fopen(cktname,"r");

         int entity_FPGA = 0;
         do{
            entity_FPGA = fscanf(fp,"%s",&start);
            if(!strcmp(start,"FPGA_Compiler_II;"))
              {
                file.num_FPGA++;
              }
        }while( entity_FPGA != EOF);

        fclose(fp);


 }
}

namespace Component_signals
{

  /*
   * This is a Function that counts the number of inputs and outputs for each component
   * in the main VHDL entity.
   * The this is done to obtain the structure of the components, and the number of
   * inputs and outputs. The result is stored in gates.num_inputs.
   */

  void in_out (char* cktname)
  {
    bool start_scan = false ;
    char start[MAXLENGTH] ;
    char    cnamefield[MAXLENGTH],    st[MAXLENGTH],    s1[MAXLENGTH],    s2,
s3[MAXLENGTH], s4[MAXLENGTH], s5[MAXLENGTH];
    int file_start, file_count, i, b;
    char buf[MAXLENGTH], buf1[MAXLENGTH], buf_temp[MAXLENGTH];
```

```
/*
 * Open the input vhdl file for reading
 */

FILE *fp;
fp = fopen(cktname,"r");

if(NULL == fp)
  {
       printf("Error Reading input file!\n");
       exit(1) ;
  }

/*
 * Scan the VHDL input file; start storing the values after
 * the last "entity" statement. This is done by first counting
 * the number of "entity" statements in the file and storing
 * the value in entity_count. The file is then closed and opened
 * again so that we know at which position the last "entity"
 * statement is, and can start storing the VHDL file from there.
 */

int entity_comp;
int entity_count = 0;

 while(1)
  {
       fscanf(fp,"%s",&start);
       if(!strcmp(start,"entity"))
         {
          entity_count++;
         }
       if(entity_count == file.num_entity)
       {
          break;
       }
  }

  int c_is = 0;
  int in = 0;
  int out = 0;
  while(1)
       {
              fscanf(fp, "%s", &buf_temp);
              if(!strcmp(buf_temp,"is"))
```

```
        {
                c_is++;
                break;
        }
}

// The following assigns the primary inputs and primary outputs of the entity.

        if(c_is == 1)
        {
                fscanf(fp, "%s %s", &buf_temp, &buf);
                while(1)
                {
                        fscanf(fp,"%s %c %s %s %s", &s1, &s2, &s3, &s4, &s5);

                        if(!strcmp(s3, "in"))
                        {
                                strcpy(pri_in[in].name, s1);
                                in++;
                        }
                        else if(!strcmp(s3, "out"))
                        {
                                strcpy(pri_out[out].name, s1);
                                out++;
                        }
                        else if(!strcmp(s5, ");"))
                        {
                                break;
                        }
                        else
                        {
                                break;
                        }
                }
        }

while(1)
{
        fscanf(fp, "%s", &buf_temp);
        if(!strcmp(buf_temp,"is"))
        {
                c_is++;
                break;
        }
}
```

```
        pri_inputs = in;
        pri_outputs = out;

        /*
         * After the second "is" statement, start reading the first component.
       */

    for(i = 0; i < file.num_component; i++)
    {

        /*
         * Start scanning in line at a time.
         */

        fscanf( fp, "%s %s", &cnamefield, &st);
        strcpy(components[i].kind, st);
        fscanf( fp, "%s %s", &cnamefield, &st);
        int c_num_inputs = 0;
        int c_num_outputs = 0;
        printf("\n");
        b = 0;
        while(1)
        {

                fscanf(fp,"%s %c %s %s %s", &s1, &s2, &s3, &s4, &s5);

                strcpy(components[i].tri_elim, s1);
                b++;
                if(!strcmp(s3, "in"))
                {
                        c_num_inputs++;
                        strcpy(components[i].identity, s3);
                }
                if(!strcmp(s3, "out"))
                {
                        c_num_outputs++;
                        strcpy(components[i].identity, s3);
                }
                if(!strcmp(s5, ");"))
                {
                   break;
                }
//              printf("Signal identity: %s\n",components[i].identity);
                printf("Components_ tri_elim : %s\n", components[i].tri_elim);
```

```
          }

          fscanf(fp,"%s %s", &s1, &s3);


          components[i].num_inputs = c_num_inputs;
          components[i].num_outputs = c_num_outputs;

          printf("Component        %s        inputs        :        %d\n",components[i].kind,
components[i].num_inputs);
          printf("Component        %s        outputs        :        %d\n",components[i].kind,
components[i].num_outputs);
          printf("Signal identity: %s\n",components[i].identity);
   }
   fclose(fp);
 }
}

namespace Circuit
{

  /*
   * This is a Function that "builds the circuit" from the VHDL netlist
   * The gates and netlists are stored as arrays of the base Gate and
   * Net classes they are scanned.
   */

  void BuildCkt (char* cktname)
  {
   bool start_scan = false ;
   bool scanned    = false ;
   char start[MAXLENGTH] ;

   char gnamefield[MAXLENGTH] ;
   int scannet = 0 ;
   int scangate = 0;
   int i, j, b, k, l, file_start, file_count;
   char buf[MAXLENGTH];
   char buf_temp[MAXLENGTH];
   char    temp1[MAXLENGTH],    temp2[MAXLENGTH],    temp3[MAXLENGTH],
temp4[MAXLENGTH], temp5[MAXLENGTH];
   char elim[MAXLENGTH], selim[MAXLENGTH], elim2[MAXLENGTH];
   char temp_in[MAXLENGTH] ;
   char temp_out[MAXLENGTH] ;
   char temp_w[MAXLENGTH], temp_x[MAXLENGTH], temp_y[MAXLENGTH];
  char s2[MAXLENGTH], s3, s4, s5[MAXLENGTH], s6, s7[MAXLENGTH],
s8[MAXLENGTH], s9[MAXLENGTH], st[MAXLENGTH] ;
  char s10, s11, s12, s14, s15, s16, s17, s18, s19,
s22[MAXLENGTH],s24,s25,s26,s27,s28,
s29[MAXLENGTH],s30,s31[MAXLENGTH],s32,s33,s34[MAXLENGTH],s35,s36,s37[
MAXLENGTH];

  /*
   * Open the input vhdl file for reading
   */

  FILE *fp;
  fp = fopen(cktname,"r");

  if(NULL == fp)
    {
        printf("Error Reading input file!\n");
        exit(1) ;
    }

  /*
   * Scan the VHDL input file; start storing the values after
   * the last "begin" statement. This is done by first counting
   * the number of "begin" statements in the file and storing
   * the value in file_start. The file is then closed and opened
   * again so that we know at which position the last "begin"
   * statement is, and can start storing the VHDL file from there.
   */

  file.start = 0;
  file.end = 0;
  file.sig = 0;
  int help = 0;
  int file_comp;
  do{
        file_comp = fscanf(fp,"%s",&start);
        if(!strcmp(start,"begin"))
          {
           file.start++;
          }
        else if(!strcmp(start, "end"))
          {
           file.end++;
          }
        else
        {
        }
     }while( file_comp != EOF);

     fp = fopen(cktname,"r");
     file_comp = 0;

/* Count the number of signal statements in the VHDL file */

     do{
        file_comp = fscanf(fp,"%s",&start);
        if(!strcmp(start,"signal"))
          {
           file.sig++;
          }
     }while( file_comp != EOF);

    fp = fopen(cktname,"r");
    file_count = 0;

    while(1)
     {
        fscanf(fp,"%s",&start);
        if(!strcmp(start,"begin"))
          {
           file_count++;
          }
        if(file_count == file.start)
        {
           start_scan = true;
           break;
        }
     }


   int gatenum = ZERO;
   int netnum  = ZERO;
   int file_done;
   int outnum = ZERO;
   int in_num = ZERO;

   while(start_scan)
     {
        printf("\n");

        /* Start scanning in one word at a time, after the last "begin" statement.
         * Only scan until a ");" or ";" is found. This represents one line statement
```

```
	 * in VHDL.
	 */

	*buf = NULL;
	do{
		file_done = fscanf(fp, "%s", &buf_temp);
		strcat(buf, buf_temp);
		strcat(buf," ");
		if(!strcmp(buf_temp,");") || !strcmp(buf_temp,";"))
		{
			break;
		}
	}while(file_done != EOF);

	printf("\n%s \n ", buf);

	sscanf( buf, "%s %s", &gnamefield, &st);
//	printf("\n%s %s\n ", gnamefield, st);

//	strcpy(gates[gatenum].contents, strchr(buf, ':'));
//	printf("%s\n", gates[gatenum].contents);

	/*
	 * If the first field is not "end" then store it as the
	 * name of the gate
	 */

	if(!strcmp(gnamefield, "end"))
	  {
	  break;
	  }
	else
	  {
	  strcpy (gates[gatenum].name, gnamefield) ;
	  }

   if(strcmp(st, "<="))
   {
	sscanf( buf, "%s %s %s %s %s %s", &gnamefield, &s2, &s5, &s7, &s9, &s8);

	printf("\nGates name %s\n", gates[gatenum].name);

	strcpy(gates[gatenum].redun, s2);
//	printf("%s \n ", gates[gatenum].redun);

	/*
```

```
         * String s5 hold the gate kind field
         */

        strncpy(gates[gatenum].kind, s5, (strlen(s5)-1));
        strcpy(gates[gatenum].kindname, s5);

        /*
         * We now determine the number of inputs and outputs for this gate by
         * looking at what type of gate it is (from s5), then comparing it to
         * what we got in the first funtion above. Also if input or output is
         * listed first (gates[gatenum].sig_order).
         */

          strcpy(elim, strchr(buf, '('));
          sscanf( elim, "%s %s", &selim, &elim2);
          strcpy(gates[gatenum].tri_elim, elim2); // This is to see if IN1 is listed first.
Used to determine the Sig_order for the TRIBUF.

        for(i = 0; i < file.num_component; i++)
        {
                if(!strcmp(s5, components[i].kind))
                {
                        gates[gatenum].num_inputs = components[i].num_inputs;
                        gates[gatenum].num_outputs = components[i].num_outputs;
                        if(!strcmp(s5, "TRIBUF"))
                        {
                          if(!strncmp(gates[gatenum].tri_elim, "IN1", 3))
                          {
                                strcpy(gates[gatenum].sig_order, "out");
                          }
                          else
                          {
                                strcpy(gates[gatenum].sig_order, components[i].identity );
                          }
                        }
                        else
                        {
                                strcpy(gates[gatenum].sig_order, components[i].identity );
                        }
                }
        }

        printf("The number of gate inputs are : %d\n",gates[gatenum].num_inputs);
        printf("The number of gate outputs are : %d\n",gates[gatenum].num_outputs);
        printf("SI order : %s\n", gates[gatenum].sig_order);
        /*
```

```
 * Store the kind and type of gate.
 */

if(!strncmp(gates[gatenum].kind, "AND", 3))
  {
    gates[gatenum].type = 100;
    if(gates[gatenum].num_inputs == 1)
    {
          strcpy(gates[gatenum].ttlname, "LCELL");
    }
    else if(gates[gatenum].num_inputs == 2)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_and2");
    }
    else if(gates[gatenum].num_inputs == 3)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_and3");
    }
    else if(gates[gatenum].num_inputs == 4)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_and4");
    }
    else if(gates[gatenum].num_inputs == 5)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_and5");
    }
    else if(gates[gatenum].num_inputs == 6)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_and6");
    }
    else if(gates[gatenum].num_inputs == 7)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_and7");
    }
    else if(gates[gatenum].num_inputs == 8)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_and8");
    }
    else if(gates[gatenum].num_inputs == 9)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_and9");
    }
    else if(gates[gatenum].num_inputs == 10)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_and10");
    }
    else if(gates[gatenum].num_inputs == 11)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_and11");
    }
    else if(gates[gatenum].num_inputs == 12)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_and12");
    }
    else
    {}
  }
else if(!strncmp(gates[gatenum].kind, "NAND",4))
  {
    gates[gatenum].type = 101;
    if(gates[gatenum].num_inputs == 1)
    {
          strcpy(gates[gatenum].ttlname, "LCELL");
    }
    else if(gates[gatenum].num_inputs == 2)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nand2");
    }
    else if(gates[gatenum].num_inputs == 3)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nand3");
    }
    else if(gates[gatenum].num_inputs == 4)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nand4");
    }
    else if(gates[gatenum].num_inputs == 5)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nand5");
    }
    else if(gates[gatenum].num_inputs == 6)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nand6");
    }
    else if(gates[gatenum].num_inputs == 7)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nand7");
    }
    else if(gates[gatenum].num_inputs == 8)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nand8");
    }
    else if(gates[gatenum].num_inputs == 9)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nand9");
    }
    else if(gates[gatenum].num_inputs == 10)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nand10");
    }
    else if(gates[gatenum].num_inputs == 11)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nand11");
    }
    else if(gates[gatenum].num_inputs == 12)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nand12");
    }
    else
    {}
  }
else if(!strncmp(gates[gatenum].kind,"OR", 2))
  {
    gates[gatenum].type = 102;
    if(gates[gatenum].num_inputs == 1)
    {
          strcpy(gates[gatenum].ttlname, "LCELL");
    }
    else if(gates[gatenum].num_inputs == 2)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_or2");
    }
    else if(gates[gatenum].num_inputs == 3)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_or3");
    }
    else if(gates[gatenum].num_inputs == 4)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_or4");
    }
    else if(gates[gatenum].num_inputs == 5)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_or5");
    }
    else if(gates[gatenum].num_inputs == 6)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_or6");
    }
    else if(gates[gatenum].num_inputs == 7)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_or7");
    }
    else if(gates[gatenum].num_inputs == 8)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_or8");
    }
    else if(gates[gatenum].num_inputs == 9)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_or9");
    }
    else if(gates[gatenum].num_inputs == 10)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_or10");
    }
    else if(gates[gatenum].num_inputs == 11)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_or11");
    }
    else if(gates[gatenum].num_inputs == 12)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_or12");
    }
    else
    {}
  }
else if(!strncmp(gates[gatenum].kind, "NOR", 3))
  {
    gates[gatenum].type = 103;
    if(gates[gatenum].num_inputs == 1)
    {
          strcpy(gates[gatenum].ttlname, "LCELL");
    }
    else if(gates[gatenum].num_inputs == 2)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nor2");
    }
    else if(gates[gatenum].num_inputs == 3)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nor3");
    }
    else if(gates[gatenum].num_inputs == 4)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nor4");
    }
    else if(gates[gatenum].num_inputs == 5)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nor5");
    }
    else if(gates[gatenum].num_inputs == 6)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nor6");
    }
    else if(gates[gatenum].num_inputs == 7)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nor7");
    }
    else if(gates[gatenum].num_inputs == 8)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nor8");
    }
    else if(gates[gatenum].num_inputs == 9)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nor9");
    }
    else if(gates[gatenum].num_inputs == 10)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nor10");
    }
    else if(gates[gatenum].num_inputs == 11)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nor11");
    }
    else if(gates[gatenum].num_inputs == 12)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_nor12");
    }
    else
    {}
  }
else if(!strncmp(gates[gatenum].kind,"XOR",3))
  {
    gates[gatenum].type = 104;
    if(gates[gatenum].num_inputs == 1)
    {
          strcpy(gates[gatenum].ttlname, "LCELL");
    }
    else if(gates[gatenum].num_inputs == 2)
    {
          strcpy(gates[gatenum].ttlname, "stdmr_xor2");
    }
    else if(gates[gatenum].num_inputs == 3)
    {
        strcpy(gates[gatenum].ttlname, "stdmr_xor3");
    }
    else if(gates[gatenum].num_inputs == 4)
    {
        strcpy(gates[gatenum].ttlname, "stdmr_xor4");
    }
    else if(gates[gatenum].num_inputs == 5)
    {
        strcpy(gates[gatenum].ttlname, "stdmr_xor5");
    }
    else if(gates[gatenum].num_inputs == 6)
    {
        strcpy(gates[gatenum].ttlname, "stdmr_xor6");
    }
    else if(gates[gatenum].num_inputs == 7)
    {
        strcpy(gates[gatenum].ttlname, "stdmr_xor7");
    }
    else if(gates[gatenum].num_inputs == 8)
    {
        strcpy(gates[gatenum].ttlname, "stdmr_xor8");
    }
    else if(gates[gatenum].num_inputs == 9)
    {
        strcpy(gates[gatenum].ttlname, "stdmr_xor9");
    }
    else if(gates[gatenum].num_inputs == 10)
    {
        strcpy(gates[gatenum].ttlname, "stdmr_xor10");
    }
    else if(gates[gatenum].num_inputs == 11)
    {
        strcpy(gates[gatenum].ttlname, "stdmr_xor11");
    }
    else if(gates[gatenum].num_inputs == 12)
    {
        strcpy(gates[gatenum].ttlname, "stdmr_xor12");
    }
    else
    {}
  }
else if(!strncmp(gates[gatenum].kind,"XNOR",4))
  {
    gates[gatenum].type = 105;
```

```
if(gates[gatenum].num_inputs == 1)
{
	strcpy(gates[gatenum].ttlname, "LCELL");
}
else if(gates[gatenum].num_inputs == 2)
{
	strcpy(gates[gatenum].ttlname, "stdmr_xnor2");
}
else if(gates[gatenum].num_inputs == 3)
{
	strcpy(gates[gatenum].ttlname, "stdmr_xnor3");
}
else if(gates[gatenum].num_inputs == 4)
{
	strcpy(gates[gatenum].ttlname, "stdmr_xnor4");
}
else if(gates[gatenum].num_inputs == 5)
{
	strcpy(gates[gatenum].ttlname, "stdmr_xnor5");
}
else if(gates[gatenum].num_inputs == 6)
{
	strcpy(gates[gatenum].ttlname, "stdmr_xnor6");
}
else if(gates[gatenum].num_inputs == 7)
{
	strcpy(gates[gatenum].ttlname, "stdmr_xnor7");
}
else if(gates[gatenum].num_inputs == 8)
{
	strcpy(gates[gatenum].ttlname, "stdmr_xnor8");
}
else if(gates[gatenum].num_inputs == 9)
{
	strcpy(gates[gatenum].ttlname, "stdmr_xnor9");
}
else if(gates[gatenum].num_inputs == 10)
{
	strcpy(gates[gatenum].ttlname, "stdmr_xnor10");
}
else if(gates[gatenum].num_inputs == 11)
{
	strcpy(gates[gatenum].ttlname, "stdmr_xnor11");
}
else if(gates[gatenum].num_inputs == 12)
{
```

```
            strcpy(gates[gatenum].ttlname, "stdmr_xnor12");
          }
          else
          {}
        }
      else if(!strncmp(gates[gatenum].kind, "INV", 3))
        {
          gates[gatenum].type = 106;
          strcpy(gates[gatenum].ttlname, "stdmr_inv");
        }
      else if(!strncmp(gates[gatenum].kind, "FLIP", 4))
        {
          gates[gatenum].type = 107;
          strcpy(gates[gatenum].ttlname, "stdmr_flip_flop");
        }
      else if(!strncmp(gates[gatenum].kind, "TRIBU", 5))
        {
          gates[gatenum].type = 108;
          strcpy(gates[gatenum].ttlname, "TRI");
        }
      else if(!strncmp(gates[gatenum].kind, "FILTE", 5))
        {
          strcpy(gates[gatenum].ttlname, "LCELL");
        }
      else if(!strncmp(gates[gatenum].kind, "DELA", 4))
        {
          strcpy(gates[gatenum].ttlname, "LCELL");
        }
      else
        {
      //   printf("Error in input file at %s\n", gates[gatenum].name);
      //   exit(1);
        }

          printf("Gate kind : %s\n", gates[gatenum].kind);


//    printf("Gate output is : %s\n", &temp_out);

      int inctr = 0 ;

      /*
       * Now that we have got the number of inputs and outputs for this
       * gate, the next input number of fields(...ofcourse not
       * counting the "(" and other fields) will be net names. But first
       * we need to take care of the order. i.e. input or output first.
```

```
 */

 /* sscanf does not work like fscanf. It reads the buffer from the start again.
  * Thus have to get rid of the stuff before the "(", i.e. after the port map.
  */

strcpy(temp4, strchr(buf, '('));

if((!strcmp(gates[gatenum].sig_order, "in")))
{

 /*
  * Now perform the output net operations.
  */

  sscanf( temp4, "%s %s %s %s", &s31, &temp2, &temp3, &temp_out);
  strcpy(temp4, strstr(temp4, temp_out));

        /*
        * Check if the output net has been scanned before. If scanned
         * get the net number and store it in the variable
         * "scannet"
        */

        for (i = 0; i < MAXNETS ; i++)
        {
                if (! strcmp( nets[i].name, temp_out))
                {
                        scanned = true ;
                        scannet = i ;
                        break ;
                }
                else
                {
                        scanned = false ;
                }
        }

        /*
        * same as what we had done for inputs
        */

        if(!scanned)
        {
                /*
                * Store the name of the net scanned
                */
                strcpy ( nets[netnum].name, temp_out ) ;
                /*
                 * Store only the outputs of all gates
                 */
                strcpy( nets[outnum].outname, temp_out);

                if(!strcmp(gates[i].kind, "IN")&&(gates[i].num_inputs!=1))
                {
                nets[netnum].int_output = true;
                }

                /*
                * Add the net number to the output net of gate
                */
                gates[gatenum].out_netnum = netnum;

                /*
                * Store the gate number as the gate whose output is this
                * net
                */
                nets[netnum].fan_out_gateid = gatenum;
                /*
                * Increae the number of nets scanned
                */
                netnum++;
                outnum++;

        }

        /*
        * Net has been scanned before
        */
        else
        {
                /*
                * scan the net number as the number of its output net
                */
                gates[gatenum].out_netnum = scannet;
                /*
                * Store the gate number as the gate whose output is this
                * net
                */
                nets[scannet].fan_out_gateid = gatenum;
        }
```

```
        for( j = 1; j < gates[gatenum].num_outputs; j++)
        {
         sscanf( temp4, "%s %s %s %s %s", &s31,&temp1, &temp2, &temp3,
&temp_out);
         strcpy(temp4, strstr(temp4, temp_out));
                /*
                * Check if the output net has been scanned before. If scanned
                 * get the net number and store it in the variable
                 * "scannet"
                */

                for (i = 0; i < MAXNETS ; i++)
                {
                        if (! strcmp( nets[i].name, temp_out))
                        {
                                scanned = true ;
                                scannet = i ;
                                break ;
                        }
                        else
                        {
                                scanned = false ;
                        }
                }

                /*
                * same as what we had done for inputs
                */
                if(!scanned)
                {
                        /*
                        * Store the name of the net scanned
                        */
                        strcpy ( nets[netnum].name, temp_out ) ;

                        /*
                        * Add the net number to the output net of gate
                        */
                        gates[gatenum].out_netnum = netnum;

                        if(!strcmp(gates[i].kind, "IN")&&(gates[i].num_inputs!=1))
                        {
                        nets[netnum].int_output = true;
                        }
                        /*
                        * Store the gate number as the gate whose output is this
                        * net
                        */
                        nets[netnum].fan_out_gateid = gatenum;
                        /*
                        * Increae the number of nets scanned
                        */
                        netnum++;

                }

                /*
                * Net has been scanned before
                */
                else
                {
                        /*
                        * scan the net number as the number of its output net
                        */
                        gates[gatenum].out_netnum = scannet;
                        /*
                        * Store the gate number as the gate whose output is this
                        * net
                        */
                        nets[scannet].fan_out_gateid = gatenum;
                }
         }

         strcpy(temp4, strstr(temp4, temp_out));

         for( j = 0; j < gates[gatenum].num_inputs; j++)
         {

                /* sscanf does not work like fscanf. It reads the buffer from the start again.
                * Thus have to get rid of the stuff before the inputs.
                */

                sscanf( temp4, "%s %s %s %s %s", &s31,&s37, &temp2, &temp3,
&temp_in);
                printf("Input Net name : %s and its number %d\n",&temp_in,j);
                strcpy(temp4, strstr(temp4, temp_in));
                //  printf("%s\n", &temp4);

                /*
                * Check if the net has been scanned before. If scanned
                * get the net number and store it in the variable
                * "scannet"
                */
                for (i = 0; i < MAXNETS ; i++)
                {
                        if (!strcmp ( nets[i].name, temp_in))
                        {
                                scanned = true ;
                                scannet = i ;
                                break ;
                        }
                        else
                        {
                                scanned = false ;
                        }
                }


                /*
                * Net not scanned before
                */
                if(!scanned)
                {
                        /*
                        * Store the name of the net scanned
                        */
                        strcpy(nets[netnum].name, temp_in) ;

                        strcpy(nets[in_num].inname, temp_in) ;

                        /*
                        * Add the net number to the input list of the gate
                        */
                        gates[gatenum].input_list[inctr] = netnum ;
        //              printf("Net number %d\n", netnum);
                        /*
                        * Add the gate number to the input list of the net
                        */
                        nets[netnum].fan_in_gateid[nets[netnum].fan_out] = gatenum ;
                        /*
                        * Increase the count of nets scanned
                        */
                        netnum++ ;
                        in_num++;
                        inctr++ ;
                }
```

```
            /*
            * Net has been scanned before
            */
            else
            {
                /*
                    * Add the net number to the input list of the gate
                    */
                    gates[gatenum].input_list[inctr] = scannet ;
                    /*
                    * Since it has been scanned before increase the
                    * fanout of the net
                    */
                    nets[scannet].fan_out++ ;
                    /*
                    * Add the gate number to the input list of the net
                    */
                    nets[scannet].fan_in_gateid[nets[scannet].fan_out] = gatenum;
                    inctr++;
            }
        }
      }

/*
    * If the "in" signal is listed first, then do this.
    */

    else
    {

            /* sscanf does not work like fscanf. It reads the buffer from the start again.
            * Thus have to get rid of the stuff before the inputs.
            */

            sscanf( temp4, "%s %s %s %s", &s31, &temp2, &temp3, &temp_in);
            printf("Input Net name : %s and its number %d\n",&temp_in,0);
            strcpy(temp4, strstr(temp4, temp_in));
            //  printf("%s\n", &temp4);

            /*
            * Check if the net has been scanned before. If scanned
            * get the net number and store it in the variable
            * "scannet"
            */
            for (i = 0; i < MAXNETS ; i++)
```

```
	{
		if (!strcmp ( nets[i].name, temp_in))
		{
			scanned = true ;
			scannet = i ;
			break ;
		}
		else
		{
			scanned = false ;
		}
	}


	/*
	* Net not scanned before
	*/
	if(!scanned)
	{
		/*
		* Store the name of the net scanned
		*/
		strcpy(nets[netnum].name, temp_in) ;

		strcpy(nets[in_num].inname, temp_in) ;
		/*
		* Add the net number to the input list of the gate
		*/
		gates[gatenum].input_list[inctr] = netnum ;
//		printf("Net number %d\n", netnum);
		/*
		* Add the gate number to the input list of the net
		*/
		nets[netnum].fan_in_gateid[nets[netnum].fan_out] = gatenum ;
		/*
		* Increase the count of nets scanned
		*/
		netnum++ ;
		inctr++ ;
		in_num++;
	}

	/*
	* Net has been scanned before
	*/
	else
```

```
                {
                        /*
                        * Add the net number to the input list of the gate
                        */
                        gates[gatenum].input_list[inctr] = scannet ;
                        /*
                        * Since it has been scanned before increase the
                        * fanout of the net
                        */
                        nets[scannet].fan_out++ ;
                        /*
                        * Add the gate number to the input list of the net
                        */
                        nets[scannet].fan_in_gateid[nets[scannet].fan_out] = gatenum;
                        inctr++;
                }

        for( j = 1; j < gates[gatenum].num_inputs; j++)
         {

                /* sscanf does not work like fscanf. It reads the buffer from the start again.
                * Thus have to get rid of the stuff before the inputs.
                */

                sscanf( temp4, "%s %s %s %s %s", &s31, &temp1, &temp2, &temp3,
&temp_in);
                printf("Input Net name : %s and its number %d\n",&temp_in,j);
                strcpy(temp4, strstr(temp4, temp_in));
                //  printf("%s\n", &temp4);

                /*
                * Check if the net has been scanned before. If scanned
                * get the net number and store it in the variable
                * "scannet"
                */
                for (i = 0; i < MAXNETS ; i++)
                {
                        if (!strcmp ( nets[i].name, temp_in))
                        {
                                scanned = true ;
                                scannet = i ;
                                break ;
                        }
                        else
                        {
                                scanned = false ;
```

```
				}
			}


			/*
			* Net not scanned before
			*/
			if(!scanned)
			{
				/*
				* Store the name of the net scanned
				*/
				strcpy(nets[netnum].name, temp_in) ;

				strcpy(nets[in_num].inname, temp_in) ;
				/*
				* Add the net number to the input list of the gate
				*/
				gates[gatenum].input_list[inctr] = netnum ;
//				printf("Net number %d\n", netnum);
				/*
				* Add the gate number to the input list of the net
				*/
				nets[netnum].fan_in_gateid[nets[netnum].fan_out] = gatenum ;
				/*
				* Increase the count of nets scanned
				*/
				netnum++ ;
				inctr++ ;
				in_num++;
			}

			/*
			* Net has been scanned before
			*/
			else
			{
				/*
				* Add the net number to the input list of the gate
				*/
				gates[gatenum].input_list[inctr] = scannet ;
				/*
				* Since it has been scanned before increase the
				* fanout of the net
				*/
				nets[scannet].fan_out++ ;
```

```
//                              nets[scannet].gate_in =
                                /*
                                * Add the gate number to the input list of the net
                                */
                                nets[scannet].fan_in_gateid[nets[scannet].fan_out] = gatenum;
                                inctr++;
                        }
                 }

               /*
                * Now perform the output net operations.
                */

//              strcpy(temp4, strstr(temp4, temp_in));

                for( j = 0; j < gates[gatenum].num_outputs; j++)
                {
                 sscanf( temp4, "%s %c %s %s %s", &s31,&s37, &temp2, &temp3,
&temp_out);
                 strcpy(temp4, strstr(temp4, temp_out));
                        /*
                        * Check if the output net has been scanned before. If scanned
                         * get the net number and store it in the variable
                         * "scannet"
                        */

                        for (i = 0; i < MAXNETS ; i++)
                        {
                                if (! strcmp( nets[i].name, temp_out))
                                {
                                        scanned = true ;
                                        scannet = i ;
                                        break ;
                                }
                                else
                                {
                                        scanned = false ;
                                }
                        }

                        /*
                        * same as what we had done for inputs
                        */
                        if(!scanned)
                        {
```

```
            /*
            * Store the name of the net scanned
            */
            strcpy ( nets[netnum].name, temp_out ) ;
            /*
             * Store only the outputs of all gates
             */
            strcpy( nets[outnum].outname, temp_out);

            if(!strcmp(gates[i].kind, "IN")&&(gates[i].num_inputs!=1))
            {
            nets[netnum].int_output = true;
            }
            /*
            * Add the net number to the output net of gate
            */
            gates[gatenum].out_netnum = netnum;

            /*
            * Store the gate number as the gate whose output is this
            * net
            */
            nets[netnum].fan_out_gateid = gatenum;
            /*
            * Increae the number of nets scanned
            */
            netnum++;
            outnum++;

    }

    /*
    * Net has been scanned before
    */
    else
    {
            /*
            * scan the net number as the number of its output net
            */
            gates[gatenum].out_netnum = scannet;
            /*
            * Store the gate number as the gate whose output is this
            * net
            */
            nets[scannet].fan_out_gateid = gatenum;
    }
```

```
            }

           }

        gatenum++;
        printf("gate no %d No of nets = %d\n",gatenum, netnum);

    } /*Closing the if(strcmp(st, "<=")) */

    else
    {
        sscanf( buf, "%s %s %s", &gnamefield, &s2, &temp_out);
        printf("\nGates name %s\n", gates[gatenum].name);
        strcpy(gates[gatenum].redun, s2);
        strcpy(gates[gatenum].outname, temp_out);
        printf("\n%s \n ", gates[gatenum].redun);
        strcpy(gates[gatenum].buf, buf);
        gatenum++;

    //  printf("%s\n", gates[gatenum].buf);
    }

   }     /*Closing the while(start_scan) */

   fclose(fp);
   totalgates = gatenum;
   totalnets = netnum;
   totaloutputs = outnum;
   totalinputs = in_num;
   printf("Total no of gates is %d Total no of nets is %d\n\n",totalgates, totalnets);

  }
}

/*
 * This routine checks and sets int isinput, isoutput flag of that
 * particular net. This is done to find out the primary inputs and
 * primary outputs of the circuit
 */

void SetNetInOut(Gate* gates, Net* nets)
{
  int gatenum, netnum, outnum, in_num ;
  int status, i ;
/*
  for (outnum = 0; outnum < totaloutputs; outnum++)
  {
          printf("Output : %s and its number : %d\n", nets[outnum].outname, outnum);
  }
  for (in_num = 0; in_num < totalinputs; in_num++)
  {
          printf("Input : %s and its number : %d\n", nets[in_num].inname, in_num);
  }
*/

  for(netnum = 0 ; netnum < totalnets ; netnum++)
    {
      /*
       * The net is a primary input net only if this net was not
       * scanned as a output of any gate..or its fan_out_gateid still
       * holds the default value which is "-1" (for some reason i
       * chose -1 ;) )
       */
//       printf("Netnum %d => fanout : %d\n",netnum, nets[netnum].fan_out);
/*
           if( -1 != nets[netnum].fan_out_gateid )
          {
            printf("Net %s is an internal signal\n", nets[netnum].name);
          }

      if( -1 == nets[netnum].fan_out_gateid )
          {
            nets[netnum].isinput = true ;
            printf("Net %s is a primary input\n", nets[netnum].name);
            strcpy(primary[pri_in].name, nets[netnum].name);
            pri_in++;
          }
      else
          {

             // A net is a primary output only if is not an input to any
             // gates.

             // NOTE: There is a flaw here, in-outs are left out this
             // way. I am trying to fix this.
             //

            for(gatenum = 0; gatenum <= nets[netnum].fan_out; gatenum++)
              {
               if( -1 == nets[netnum].fan_in_gateid[gatenum] )
                    {
                     status = 2 ;
```

```
                }
            else
                {
                  status = 0;
                  break ;
                }
          }
        if(status == 2)
          {
            nets[netnum].isoutput = true;
            printf("Net %s is a primary output\n", nets[netnum].name);
          }
       }
       */

       for (i = 0; i < pri_inputs; i++)
       {
               if(!strcmp(nets[netnum].name, pri_in[i].name))
               {
                       nets[netnum].isinput = true;
               }
       }

       for (i = 0; i < pri_outputs; i++)
       {
               if(!strcmp(nets[netnum].name, pri_out[i].name))
               {
                       nets[netnum].isoutput = true;
               }
       }


       for (i =0; i < totalgates; i++)
       {
               if(!strcmp(nets[netnum].name, gates[i].outname))
               {
                       nets[netnum].isoutput = true;
                       printf("TRUE\n");
               }
       }

  }
 return ;
}

 /*
```

```
   * This is a routine inserts the signals clock, reset and VDD as inputs to the
   * DFF.
   */

namespace DFlip_Flop
{

  void dffsignal(Gate *gates, Net *nets)
  {
    int i, j;
    bool scanned = false;
    int scangate = 0;
    printf("Total gates : %d\n", totalgates);
    printf("Total nets : %d\n", totalnets);
    /*
    for(i = 0; i < totalgates ; i++)
      {
       if(!strncmp(gates[i].kind, "DF", 2))
            {
              strcpy(nets[gates[i].input_list[1]].name, "clock");
              strcpy(nets[gates[i].input_list[2]].name, "VCC");
              strcpy(nets[gates[i].input_list[3]].name, "VCC");
            }
       if(!strncmp(gates[i].kind,"XOR",3))
       {
              strcpy(nets[gates[i].input_list[1]].name, "GND");
            }
       }
       */
   }
}

  /*
   * This is a routine that removes (not really) the technology dependant delay
components from the VHDL netlist, as
   * obtained from MAXPLUS II or QUARTUS. Actually it reassigns the signals.
   */
/*
namespace Remove_Delay
{

  void remove(Gate *gates, Net *nets)
  {
    int i, j, k, l;

     for(i = 0; i < totalgates ; i++)
    {
        l = 0;
        if(strcmp(gates[i].kind, "IN")&&(gates[i].num_inputs==1))
        {
                for(j = 0; j < totalgates ; j++)
                {
                        for(k = 0; k < gates[j].num_inputs; k++)
                        {
                                if(!strcmp(nets[gates[i].out_netnum].name,
nets[gates[j].input_list[k]].name))
                                {
                                //      strcpy(            nets[gates[j].input_list[k]].name,
nets[gates[i].input_list[0]].name );
                        //      printf("Gate delay num
                                l++;

                                }
                        }

                }
     }

    }


    //Get rid of the TRIBUF at the output.

    for(i = 0; i < totalgates ; i++)
    {
     if(!strcmp(gates[i].ttlname, "TRI"))
        {
                for(j = 0; j < totalgates ; j++)
                {
                   if((gates[j].num_inputs!=1)||!strcmp(gates[j].name, "IN")) // To get rid
of the delay elements.
                   {
                        if(!strcmp(nets[gates[i].input_list[0]].name,
nets[gates[j].out_netnum].name))
                        {
                                strcpy(nets[gates[j].out_netnum].name,
nets[gates[i].out_netnum].name);
                                nets[gates[j].out_netnum].isoutput = true;

                        //      printf("%s       input       :       %s\n",gates[i].name,
nets[gates[i].input_list[0]].name);
```

```
                        //        printf("%s          output          :          %s\n",gates[i].name,
nets[gates[i].out_netnum].name);
                        //        printf("%s          output          :          %s\n",gates[j].name,
nets[gates[j].out_netnum].name);
                        //        printf("i = %d, j = %d\n\n", i, j);

                        }
                    }
                }

            }
        }

  }
}
*/

/*
 The following routine determines the no of modules required per redundant group
according
 to the STDMR algorith.
*/

namespace Redundant_Modules
{
        void R_modules ( Gate *gates)
        {
          int gatenum;

          /* Now we determine the no of redundant modules required for each gate
            based on the activity of the most idle path according to STDMR.
          */

          for (gatenum = 0; gatenum < totalgates; gatenum++)
          {
                if(!strcmp(gates[gatenum].redun, ":"))
                {
                gates[gatenum].add_redundancy = 1;
                gates[gatenum].modules = 2;
                }
                else
                {
                        gates[gatenum].add_redundancy = 0;
                }
          }
        }
```

```
}

/*
 *  This routine inserts the redundant modules in the VHDL gatelevel netlist file.
 *  The control circuitry is in accordingly.
 */

namespace Insert_Redundancy
{
  void InsertStdmr(Gate *gates, Net *nets, Multi *tempnets, char* cktname, char* outfile)
  {
    long int i, j, q, r, k = 0, count = 0, countn = 0, index, c = 0, b, d, e, f, g;
    int a;
    char temp[200];
    char *separate = " ";
    char gateinstance[10][instance];
    char muxinstance[10][instance];
    char muxinput0[10][instance];
    char muxinput1[10][instance];
    char muxtemp[10][instance];
    char signals[1000][1000];
    bool linescan = false;
    bool muxscan0 = false;
    bool muxscan1 = false;
    char buf_temp[MAXLENGTH];
    char buf_new[50];
    char line_temp[50];
    char buf[MAXLENGTH];
    char                    buf_smr1[MAXLENGTH],                    buf_smr2[MAXLENGTH],
buf_smr3[MAXLENGTH] ;
    char ent[10];
    int intro, aa;
    bool sigscan = false;

    FILE *fp,*fp1,*fp2,*fp3,*fp4;
    fp = fopen("temp.out", "w+r");  /* Create and open a temporary file to write the output
*/
    fp1 = fopen(cktname,"r");  /* Open the original file for reading */
    fp2 = fopen(outfile,"w+a");            /* open the file in write mode..to append it */
    fp3 = fopen("intro.txt","r"); /* Open the intro file */
    fp4 = fopen("components.txt","r"); /* Open the file with the component declarations */

    if(fp == NULL)
      {
        printf("Unable to Create File temp.out");
        exit(0);
```

```
     }
   if(fp1 == NULL)
     {
      printf("Unable to Open the Original File..Exiting!!");
      exit(0);
     }
   if(fp2 == NULL)
     {
      printf("Unable to create the _smr File");
      exit(0);
     }

   /* At this stage the intro is added to the start of the VHDL file */

         for(i=0;i<34;i++)
         {
         fgets(buf,200,fp3);
         fputs(buf, fp2);
         }
//printf("TEST 0\n");
   /*
    * Get rid of all the entities before the last (MAIN) entity.
    * These are basically the entities that describes the components
    * within the MAIN entity, but is not needed since we use the
    * predefined components in the STDMR Library.
    */
/*
    int FPGA_count  = 0;
      while(1)
         {
                  fgets(buf,200,fp1);
                  sscanf( buf, "%s %s",&buf_new, &line_temp);
                  if(!strcmp(line_temp,"FPGA_Compiler_II;"))
                  {
                           FPGA_count++;
                  }
                  if(FPGA_count == (file.num_FPGA - 1))
                  {
                           break;
                  }
         }
*/

//printf("TEST 1\n");
/* Now Start scanning the VHDL file until the first component of the last entity is found
*/
```

```
   r = 0;
   int s = 0;
   while(1)
     {
          fgets(buf,200,fp1);
          sscanf( buf, "%s %s %s %s %s", &line_temp, &buf_temp, &buf_new, &temp,
&ent);
//        printf("TEST 2\n");
          if(!strcmp(line_temp, "use")&&(s == 0))
          {
                  fprintf(fp2, "use IEEE.std_logic_1164.all;\n");
                  fprintf(fp2, "library work;\n");
                  fprintf(fp2, "use work.stdmr_package.ALL;\n\n");
                  s++;
          }
          else if(!strcmp(line_temp, "entity"))
          {
                  sprintf( buf_smr1, "%s %s_smr %s\n", &line_temp, &buf_temp,
&buf_new);
                  fputs(buf_smr1, fp2);
          }
          else if(!strcmp(line_temp, "port"))
          {
                  fprintf( fp2, " port (\n");
                  fprintf( fp2, "   s0 : in std_logic ;\n");
                  fprintf( fp2, "   s1 : in std_logic ;\n");
                  fprintf( fp2, "   clk_out : in std_logic ;\n");
          }
          else if(!strcmp(line_temp, "end")&&(r == 0))
          {
                  fprintf( fp2, "%s %s_smr %s\n", &line_temp, &buf_temp, &buf_new);
                  r++;
          }
          else if(!strcmp(line_temp, "architecture"))
          {
                  sprintf( buf_smr3, "%s %s %s %s_smr %s\n", &line_temp, &buf_temp,
&buf_new, &temp, &ent);
                  fputs(buf_smr3, fp2);
          }
          else
          {
                  fputs(buf, fp2);
          }

          if(!strcmp(ent,"is"))
            {
```

```
            break;
          }
     }
     fputs("\n", fp2);

 /* Add the TTL components to the VHDL file */
   /*
     for(i=0;i<142;i++)
          {
          fgets(buf,200,fp4);
          fputs(buf, fp2);
          }
     fputs("\n", fp2);
     fputs("\n", fp2);
     */
/* Add the signals of the original VHDL file to the smr file at this stage */
//printf("TEST 2\n");
     while(1)
     {
          fgets(buf,200,fp1);
          sscanf( buf, "%s", &line_temp);
//        printf("%s", buf);
          if(!strcmp(line_temp,"signal"))
            {
                  break;
            }
     }
          fputs(buf, fp2);

     while(1)
     {
          fgets(buf,200,fp1);
          sscanf( buf, "%s", &line_temp);
          if(!strcmp(line_temp,"begin"))
            {
                  break;
            }
          fputs(buf, fp2);
     }

     fclose(fp1);

     /*
      * The following section adds the input mux's to the primary inputs.
      */
```

```
    for ( a = 0; a < pri_inputs; a++)
        {
          printf("Net no %d and name = %s\n", a, pri_in[a].name);

                sprintf( muxinput0[a], "inmux_0_%d : stdmr_mux port map( s0, clk_out,
%s, %s_0);\n", a, pri_in[a].name, pri_in[a].name);
          muxinput0[a][strlen(muxinput0[a])] = '\0';

                sprintf( muxinput1[a], "inmux_1_%d : stdmr_mux port map( s1, clk_out,
%s, %s_1);\n", a, pri_in[a].name, pri_in[a].name);
          muxinput1[a][strlen(muxinput1[a])] = '\0';

          fputs(muxinput0[a], fp);
                fputs(muxinput1[a], fp);
        }

   for(i = 0; i < totalgates ; i++)
   {
    /*
     * Delete all the delay components, i.e. all components with a single input
     * except the inverter (INV).
     */
     /*
    if(!(strcmp(gates[i].kind, "IN")&&(gates[i].num_inputs==1)))
    {
    // This is to get rid of the TRI buffer (at the output).

     if(strcmp(gates[i].kind, "TRIBU"))
     {
        */

      for( b = 0; b < gates[i].modules; b++)
        {
      gateinstance[b][0] = '\0';          /*clear the gates string array*/
        }
        for( b = 0; b < gates[i].modules; b++)
        {
      muxinstance[b][0] = '\0';           /*clear the mux string array*/
        }

        for( b = 0; b < gates[i].modules; b++)
        {
      muxtemp[b][0] = '\0';               /*clear the mux string array*/
        }
        //printf("TEST %d\n", i);
        if( gates[i].add_redundancy == 0 )
```

```
            {
                fputs(gates[i].buf, fp);
                fputs("\n", fp);
            }
          else
            {
          /* Replicate the gate gates[i].modules times */

          for ( a = 0; a < gates[i].modules; a++)
           {
                    sprintf( gateinstance[a], "%s_%d : %s port map( ",gates[i].name, a,
gates[i].ttlname);
               gateinstance[a][strlen(gateinstance[a])] = '\0';

           }

           for ( a = 0; a < 2*gates[i].num_inputs; a++)
           {
                    sprintf( muxinstance[a], "mux2_%s_%d : stdmr_mux port map(
",gates[i].name, a);
               muxinstance[a][strlen(muxinstance[a])] = '\0';
           }
       /*
        * Create 2 Input Mux's to each input of the gate.
        */

               for ( a = 0; a < gates[i].num_inputs; a++)
           {
                   sprintf( muxinstance[a] + strlen(muxinstance[a]), "s, clk_out, %s, %s_0",
nets[gates[i].input_list[a]].name, nets[gates[i].input_list[a]].name);

               }

               for ( a = gates[i].num_inputs; a < 2*gates[i].num_inputs ; a++)
           {
                   sprintf( muxinstance[a] + strlen(muxinstance[a]), "s, clk_out, %s, %s_1",
nets[gates[i].input_list[a - gates[i].num_inputs]].name, nets[gates[i].input_list[a -
gates[i].num_inputs]].name);

               }


           for(j=0; j < gates[i].num_inputs ; j++)
               {
                   /*
```

```
                 * Do if the gate is replicated or its inputs signals are internal. In the stdmr
case
                 * the gate is always replicated.
                 */


if((gates[nets[gates[i].input_list[j]].fan_out_gateid].add_redundancy)||(nets[gates[i].input
_list[j]].fan_out_gateid == -1))
                  {
                   for (b = 0; b < gates[i].modules; b++)
                   {
                 sprintf(   gateinstance[b]   +   strlen(gateinstance[b]),   "%s_%d,   ",
nets[gates[i].input_list[j]].name, b);
             }

             for (b = 0; b < gates[i].modules; b++)
             {
                strcpy( tempnets[b].name, nets[gates[i].input_list[j]].name);
             }

             char tempc[MAXLENGTH];
             for (b = 0; b < gates[i].modules; b++)
             {
                 sprintf( tempc, "_%d", b);
                 strcat(tempnets[b].name, tempc);
             }
                         /*

                 for (b = 0; b < gates[i].modules; b++)
                 {
                         strcpy(signals[k],tempnets[b].name);
                         k++;
                 }
                 */
                  }
            else
                  {
                   for (b = 0; b < gates[i].modules; b++)
                   {
              sprintf(    gateinstance[b]    +    strlen(gateinstance[b]),    "%s,    ",
nets[gates[i].input_list[j]].name);
              }
                  }
                 }
```

```
        /*************************************************************
**********************/

            for(j=0; j < gates[i].num_inputs; j++)
            {

if((gates[nets[gates[i].input_list[j]].fan_out_gateid].add_redundancy)||(nets[gates[i].input
_list[j]].fan_out_gateid == -1))
                  {

             for (b = 0; b < gates[i].modules; b++)
             {
                strcpy( tempnets[b].name, nets[gates[i].input_list[j]].name);
             }

             char tempc[MAXLENGTH];
             for (b = 0; b < gates[i].modules; b++)
             {
                 sprintf( tempc, "_%d", b);
                 strcat(tempnets[b].name, tempc);
             }


                 for (b = 0; b < gates[i].modules; b++)
                 {
                         strcpy(signals[k],tempnets[b].name);
                         k++;
                 }


                  }

            else
                  {
              for (b = 0; b < gates[i].modules; b++)
                    {
                         sprintf(   muxinstance[b]   +   strlen(muxinstance[b]),   ",   %s",
nets[gates[i].input_list[j]].name);

              }
                  }
            }



        for(j = 0; j < gates[i].num_inputs; j++)
```

```
            {
          for(index=0; index <= nets[gates[i].input_list[j]].fan_out;index++)
                 {
             if(!gates[nets[gates[i].input_list[j]].fan_in_gateid[index]].add_redundancy )
                     {
                c = 0;
                     }
             else
                     {
                c = 2;
                break;
                     }
                  }
            }

       if(c != 0)
            {
                for (b = 0; b < gates[i].modules; b++)
                {
            sprintf(       gateinstance[b]        +        strlen(gateinstance[b]),        "%s_%d
);\n",nets[gates[i].out_netnum].name,b);
                }

                for (b = 0; b < 2*gates[i].num_inputs; b++)
                {

                sprintf( muxinstance[b] + strlen(muxinstance[b]), " );\n");

                }


                for (b = 0; b < gates[i].modules; b++)
          {
             strcpy( tempnets[b].outname,nets[gates[i].out_netnum].name);
          }


                char tempc[MAXLENGTH];
          for (b = 0; b < gates[i].modules; b++)
          {
            sprintf( tempc, "_%d", b);
            strcat(tempnets[b].outname, tempc);
          }

          for(q = 0; q <= k; q++)
                 {
```

```
            if(strcmp(tempnets[0].outname,signals[q]) == 0)
                   {
               r=2;
               break;
                   }
            else
                   r=0;
                }

        if(r==0)
                {
                        for (b = 0; b < gates[i].modules; b++)
               {
                        strcpy(signals[k],tempnets[b].outname);
                        k++;
               }
                }



               /*
                * The following prints out the multiplexers thats is added to each input
gate.
                */

          /*
            for (b = 0; b < gates[i].num_inputs; b++)
                {
                 if( nets[gates[i].input_list[b]].isinput == true) // Check if the net is a
primary input, if yes, then print as is without the _0 or _1.
              {
                        printf("%s \n", muxinstance[b]);
               fputs(muxinstance[b], fp);
          }
          else
          {}
        }

        for (b = gates[i].num_inputs; b < 2*gates[i].num_inputs; b++)
                {
                 if( nets[gates[i].input_list[b - gates[i].num_inputs]].isinput == true) //
Check if the net is a primary input, if yes, then print as is without the _0 or _1.
              {
                        printf("%s \n", muxinstance[b]);
               fputs(muxinstance[b], fp);
          }
```

```
            else
            {}
           }
           */


                for (b = 0; b < gates[i].modules; b++)
                {
                fputs(gateinstance[b], fp);
           }



        // The following prints out the multiplexers thats added to the output gate.
        // Only if it contains a primary output net.

            if(nets[gates[i].out_netnum].isoutput == true)
            {
           char tempcc2[MAXLENGTH] = "\0";
           char tempcc3[MAXLENGTH] = "\0";
           char tempcc4[MAXLENGTH] = "\0";

           sprintf(gateinstance[0], "outmux_%d : stdmr_mux port map ( s1, ", count);
                count++;

             for (b = 0; b < gates[i].modules; b++)
             {
                sprintf( tempcc2, "%s_%d ,",nets[gates[i].out_netnum].name, b);
                strcat(tempcc3, tempcc2);
             }

           sprintf( tempcc4, "%s );\n",nets[gates[i].out_netnum].name);
           strcat(tempcc3, tempcc4);
           strcat(gateinstance[0], tempcc3);

           fputs(gateinstance[0], fp);
          }
            }

        else
            {
                for ( b = 0; b < gates[i].modules; b++)
                {
                sprintf(   gateinstance[b]   +   strlen(gateinstance[b]),   "%s_%d   );\n",
nets[gates[i].out_netnum].name,b);
           }
```

```
            fputs(gateinstance[0], fp);
            for (b = 0; b < gates[i].modules; b++)
            {
                fputs(gateinstance[b], fp);
            }
             }
          }
//     }
//      }
//      else
//      {}
      }

   /* If there is only one instance of this gate then only its first input is
      listed in the signals part of the VHDL file. However, its fan out could
      be more than one of it is the input of a gate that is replicated. Thus,
      have to list its inputs as well. This problem is now solved. See
      explanation above.
   */

   fputs("\nend FPGA_Compiler_II;", fp);
   fclose(fp);

   fp = fopen("temp.out","r");
   int lineno = 0 ;
   float newline = 0;
   q = 0;
   printf("Value of K = %d\n", k);
   while(1)
     {
//      printf("TEST IT 1??? %d\n", q);
      fprintf(fp2,"signal           %s", signals[0]);
      strcpy(lines[0].name, signals[0]);
      q++;
//    printf("TEST IT 2 %d\n", q);

            /*
             * Check if the signal has been added to the file before.
             */

      for(i=1; i < k; i++)
           {
         for (j = 0; j < k ; j++)
              {
                 if (!strcmp (lines[j].name, signals[i]))
                   {
                        linescan = true ;
                        lineno = i ;
                        break ;
                      }
                    else
                      {
                        linescan = false ;
                      }
                 }
           //   printf("TEST IT 3 %d\n", i);

                 // String contents not scanned before

                if(!linescan)
                  {

                      // Store the name of the signal scanned

                        strcpy(lines[i].name, signals[i]);
                        fprintf(fp2," ,%s", lines[i].name);
                        lineno++;
                        newline++;


                        // The following comparison is needed so that we only have 7 signals
                        // per line.


                        if(roundf(newline/7) == newline/7)
                        {
                                fprintf(fp2, "\n");
                                fprintf(fp2, "       ");
                        }

                   }


                // String contents has been scanned before, then do not add it to the file, i.e.
                // do nothing.

                else
                  {

                  }
               }
//            printf("TEST %d\n", -1);
//       fprintf(fp2," %s : std_logic;\n\n", signals[k-1]);
        fprintf(fp2," : std_logic;\n\n");
        fprintf(fp2,"%s\n\n", "begin");

            while(1)
              {
            fgets(temp,200,fp);
            if(strncmp(temp,"end FPGA_Compiler_II;",21) == 0)
                  {
                fputs(temp,fp2);
                break;
                  }
            fputs(temp,fp2);
              }
            fclose(fp2);
            fclose(fp);
            break;
      }
  }
}


int main()
{
  char cktname[MAXLENGTH] ;
  char timing[MAXLENGTH];
  char basename[MAXLENGTH] ;
  char outfile[MAXLENGTH] ;
  char probfile[MAXLENGTH] ;

  printf("Enter the circuit name without extension: \n") ;
  scanf("%s", cktname) ;

  strcpy(outfile, cktname) ;
  strcpy(basename, cktname) ;
  strcat(outfile,"_smr.vhd") ;
  strcat(cktname, ".vhd") ;

  Compo::num_compo (cktname) ;
  Component_signals::in_out (cktname) ;
  Circuit::BuildCkt (cktname) ;
  SetNetInOut(gates, nets);
//  Remove_Delay::remove(gates,nets);
  DFlip_Flop::dffsignal(gates,nets);
  Redundant_Modules::R_modules (gates);
  Insert_Redundancy::InsertStdmr(gates, nets, tempnets, cktname, outfile);
  return 0 ;
}
```

```
//This is the stdmr.h library file for the main Code.

#include "iostream"
#include "string"
#include "cmath"
#include "fstream"
#include "stdio.h"
#include "iomanip"
#include "time.h"
#include "strings.h"

#define ZERO 0
#define JUNK 2.0
#define MAXINPUTS 100
#define MAXGATES 5000
#define MAXNETS 5000
#define MAXLENGTH 1000
#define MAXFANOUTS 1000
#define SHORTLENGTH 1000
#define instance 100000

const int AND   = 100 ;
const int NAND  = 101 ;
const int OR    = 102 ;
const int NOR   = 103 ;
const int XOR   = 104 ;
const int XNOR  = 105 ;
const int INV   = 106 ;
const int DFF  = 107 ;
const int TRIBU = 108;


class Gate
{
public:
  /*
   * Gate name
   */
    char name[MAXLENGTH] ;

    char outname[MAXLENGTH];

   /*
    * Used to eliminate the TRIBUF
    */
    char tri_elim[MAXLENGTH] ;
```

```
/*
 * Individual Gate line contents
 */
 char contents[MAXLENGTH];

/*
 * Gates TTL part name
 */
 char ttlname[MAXLENGTH];

/*
 * single assignment names
 */
  char buf[MAXLENGTH] ;

  char redun[10];
  /*
   * Gate kind (AND, NAND etc...)
   */
  char kind[SHORTLENGTH] ;


  char kindname[SHORTLENGTH] ;

  /*
   * Gate signal order, i.e. is the input or output listed first.
   */
  char sig_order[SHORTLENGTH];

  /*
   *Gate time information
   */
   char time[MAXLENGTH] ;
   /*
   * Gate type 100 for and and so on
   */
  int type ;
  /*
   * number of inputs
   */
  int num_inputs ;
  /*
   * number of outputs
   */
  int num_outputs ;
```

```
    /*
     * List of inputs
     */
    int input_list[SHORTLENGTH] ;
    /*
     * output net number
     */
    int out_netnum ;
    /*
     * gate activity
    */
    float activity;
    /*
     * No of redundant modules required.
     */
    float modules;
    /*
     * flag to indicate whether this gate has to get redundancy.
     */
     int add_redundancy;

     bool sensitivity ;
    /*
     * flag to indicate whether this gate has to be triplicated ot no
     */
    bool triplicate ;
    /*
     * Default constructor
    /*
     * Default constructor
     */
    Gate () ;
} ;

Gate::Gate()
{
    /*
     * Assign default values
     */
    add_redundancy = false;
}


class Net
{
public:
```

```
/*
 * Net name
 */
char name[MAXLENGTH] ;

char outnew[MAXLENGTH];
 /*
Net timing information
*/
 char timename[MAXLENGTH] ;
 /*Net logic state per clock cycle
 */
 char logic[MAXLENGTH];

 char output[MAXLENGTH];
/*
 * fan out the net
 */
int fan_out ;
/*
 * Used to store the delay nets for comparison.
 */
char delay_name[MAXLENGTH];
char outname[MAXLENGTH];
char inname[MAXLENGTH];
/*
 * number of gates whose input is this net. Infact this array
 * store the gate numbers
 */
int fan_in_gateid[MAXFANOUTS] ;
/*
 * Gate number whose output is this net
 */
int fan_out_gateid ;
/*
 * Signal probability value of the net
 */
float prob ;
/*
 * flag to indicate if this net is a primary input
 */
bool isinput ;
/*
 * flag to indicate if this net is a primary output
 */
bool isoutput ;
```

```
    bool int_output;
    /*
     * Net activity
     */
    int activity;
    /*
     * Default constructor
     */

    Net() ;
};

Net::Net()
{
  /*
   * Assign default values
   */

  int j ;
  fan_out = 0 ;
  fan_out_gateid = -1 ;
  isinput = false ;
  isoutput = false ;
  int_output = false ;
  prob = JUNK ;
  for(j = 0 ; j <MAXFANOUTS ; j++)
    {
      fan_in_gateid[j] = -1 ;
    }

}

class Multi
{
public:
  /*
   * name
   */
    char name[MAXLENGTH] ;

    char outname[MAXLENGTH];

    /*
     * Default constructor
     */
    Multi () ;
} ;

Multi::Multi()
{
    /*
     * Assign default values
     */

}

class Sig
{
public:
  /*
   * name
   */
    char name[MAXLENGTH] ;

    char outname[MAXLENGTH];

    /*
     * Default constructor
     */
    Sig () ;
} ;

Sig::Sig()
{
    /*
     * Assign default values
     */

}

class Line
{
public:
  /*
   * name
   */
    char name[MAXLENGTH] ;

    /*
     * Default constructor
     */
    Line () ;
} ;

Line::Line()
{
    /*
     * Assign default values
     */

}

class Files
{
public:
  /*
   * name
   */
    int start ;
    int end ;
    int sig;
    int num_entity ;
    int end_entity ;
    int num_FPGA ;
    int num_component ;
    int end_component ;

    /*
     * Default constructor
     */
    Files () ;
} ;

Files::Files()
{
    /*
     * Assign default values
     */

}

class Component
{
public:
    /*
     * component kind (AND, NAND etc...)
     */
    char kind[SHORTLENGTH] ;
    /*
     * component signal identity (i.e. input or output)
     */
    char identity[SHORTLENGTH] ;
    /*
     * number of inputs
     */
    int num_inputs ;
    /*
     * number of outputs
     */
    int num_outputs ;
    /*
     * Check if input or output is listed first
     */
    bool sig_order ;

   /*
    * Used to eliminate the TRIBUF
    */
    char tri_elim[MAXLENGTH] ;
    /*
     * Default constructor
     */
    Component () ;
} ;

Component::Component()
{
    /*
     * Assign default values
     */
     sig_order = false ;
}

class primary_in
{
public:
    /*
     * Insert only the primary inputs in the class.
     */
    char name[MAXLENGTH] ;

    primary_in () ;
} ;
```

```
primary_in::primary_in()
{
   /*
    * Assign default values
    */
}

class primary_out
{
public:
   /*
    * Insert only the primary inputs in the class.
    */
   char name[MAXLENGTH] ;

   primary_out () ;
} ;

primary_out::primary_out()
{
   /*
    * Assign default values
    */
}


class gate_in
{
public:
   /*
    * Insert only the primary inputs in the class.
    */
   char name[MAXLENGTH] ;

   gate_in () ;
} ;

gate_in::gate_in()
{
   /*
    * Assign default values
    */
}
/*
 * Declare an array of classes
 */
Gate gates[MAXGATES];
Net nets[MAXNETS];
Multi tempnets[MAXNETS];
Sig   temp[MAXNETS];
Line lines[MAXLENGTH];
Files file;
Component components[MAXLENGTH];
primary_in pri_in[MAXNETS];
primary_out pri_out[MAXNETS];

/*
 * Global variables
 */
int totalgates = ZERO;
int totalnets = ZERO;
int totalouts = ZERO;
int pri_inputs = ZERO;
int pri_outputs = ZERO;
int totaloutputs;
int totalinputs;
int clock_cycle;
int outgates[2000];
```

Appendix E

Flatband diagram [ZUKA02]

The flatband diagram is by far the easiest energy band diagram of the MOS transistor. The term flatband refers to fact that the energy band diagram of the semiconductor is flat, which implies that no charge exists in the semiconductor. The flatband diagram of an aluminum-silicon dioxide-silicon MOS structure is shown in Figure E1. Note that a voltage, V_{FB}, must be applied to obtain this flat band diagram. Indicated on the figure is also the work function of the aluminum gate, Φ_M, the electron affinity of the oxide, χ_{oxide}, and that of silicon, χ, as well as the bandgap energy of silicon, E_g. The bandgap energy of the oxide is quoted in the literature to be between 8 and 9 electron volt. The reader should also realize that the oxide is an amorphous material and the use of semiconductor parameters for such material can justifiably be questioned.

The flat band voltage is obtained when the applied gate voltage equals the workfunction difference between the gate metal and the semiconductor. If there is also a fixed charge in the oxide and/or at the oxide-silicon interface, the expression for the flatband voltage must be modified accordingly. This is necessary to offset any existing electric fields due to the presence of a fixed charge.

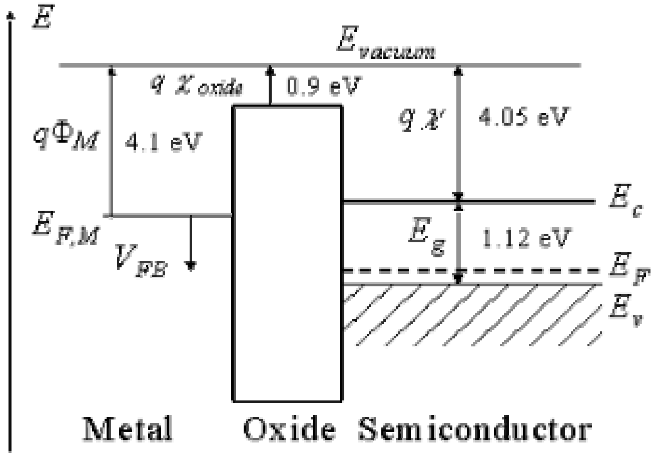


Fig E1 Flatband energy diagram of a metal-oxide-semiconductor (MOS) structure consisting of an aluminum metal, silicon dioxide and silicon.

Appendix F
The Accompanying CD-ROM

A CD-ROM accompanies this thesis. This CD-ROM contains the following information:

- This thesis document in ".doc" as well as ".pdf" format can be found in the "Thesis Document" directory.
- Many of the documents referenced in this thesis can be found in the "Referenced papers and other useful documents" directory.
- Source code for the SMR algorithm can be found in the "code" directory.
- The images used in this thesis have been included in various formats in the "Thesis Diagrams" directory.
- The papers published as part of this dissertation can be found in the directory "Published Work"
- The PCB design files of the radiation test boards can be found in the directory "PCB Files"
- The unpublished SMR papers can be found in the directory "SMR SEU mitigation papers"
- Additional FPGA information can be found in the directory "FPGA info"