FACULTY OF
PHYSICS AND ASTRONOMY

HEIDELBERG UNIVERSITY

DIPLOMA THESIS
IN PHYSICS

SUBMITTED BY
HEIKO ENGEL
BORN IN
SINSHEIM, GERMANY

JUNE 2009

Development of a Fault Tolerant Softcore CPU for SRAM based FPGAs

This diploma thesis has been carried out by **Heiko Engel** at the

Kirchhoff Institute for Physics

under the supervision of

Prof. Dr. Udo Kebschull

Entwicklung einer fehlertoleranten Softcore CPU für FPGAs

In Umgebungen mit erhöhter Teilchenstrahlung ist der fehlerfreie Betrieb SRAM-basierter feldprogrammierbarer Hardware nicht mehr garantiert. Radioaktive Strahlung kann sowohl die Konfiguration, als auch den Zustand dieser Geräte und damit ihr Verhalten ändern. Gängige Ansätze benutzen dreifach redundante Logik (TMR) mit Mehrheitsentscheiden um strahlungsbedingtes Fehlverhalten zu kompensieren. Dies bringt jedoch ein erhebliches Maß an zusätzlicher Logik mit sich. Diese Diplomarbeit stellt eine fehlertolerante Softcore-CPU für FPGAs vor, die durch die Kombination von zweifach redundanter Logik und kontinuierlichem Schreiben der FPGA-Konfiguration mit einem geringeren Maß an zusätzlicher Logik auskommt. Die Wirksamkeit der angewandten Methoden konnte sowohl mit Fehlersimulationen, als auch im Teilchenstrahl Experiment nachgewiesen werden.

Development of a Fault Tolerant Softcore CPU for FPGAs

In radiative environments, the accurate operation of SRAM based field programmable hardware cannot be guaranteed. Radiation can alter configuration and state of these devices and thus change their behavior. Common approaches use *triple modular redundancy* (TMR) in combination with majority voters to compensate radiation induced errors. However, this comes with a large area overhead. This thesis proposes a fault tolerant softcore CPU for FPGAs with reduced area overhead by using double modular redundant logic in combination with continuous FPGA configuration writing. The effectiveness of the applied methods could be verified with both error simulation and particle beam experiments.

Contents

List of Figures

List of Tables

1 Introduction and Motivation

With the invention of the transistor in the middle of the 20th century, a new era in electronic devices was started. The previously used tubes could gradually be replaced with these low voltage, low area, highly flexible circuit elements. Research over the years led to continuously decreasing structure sizes allowing the creation of systems with millions of transistors on a single chip. However, creating custom application specific integrated circuits (ASICs) is a time consuming and expensive task. The manufacturing costs can only be mitigated by producing large numbers of chips. An approach to make microelectronic devices available to a broader spectrum of applications was the development of microcontroller systems, starting in the 1970s. Several different modules, all produced in large series and thus cheap, can be combined to form a custom electronic system. But even with increasing functionality, these devices cannot reach the performance of ASICs and their hardware based functionality is fixed once assembled.

This gap has been recognized by Ross Freeman, founder of Xilinx[1], in 1984. He invented the field programmable gate array (FPGA). The patented idea was to supply a single chip with "a plurality of configurable logic elements variably interconnected in response to control signals to perform a selected logic function" [Fre89]. Freeman postulated this device to be affordable for customers due to decreasing costs for transistors over time. Xilinx is now a multi billion dollar enterprise and "the worldwide leader in programmable logic solutions" [Xilun]. The possibility to use the same chip for arbitrary circuits opened a new field of applications. Complex tasks previously implemented as sequential microprocessor instructions can be parallelized in FPGAs. The concrete functionality of an FPGA has only roughly to be specified during the construction of an electronic device and thus significantly increases the time-to-market. The possibility for users to change the configuration of the FPGA offers the potential to supply several different applications with the same device or "upgrade" an existing implementation with bug-fixes or new functionality. FPGAs developed from ASIC prototyping platforms to solutions for digital signal processing, high performance computing, defense, aerospace and high energy physics applications.

Especially in high energy physics experiments, FPGAs have become an important part. Data rates produced by current particle detectors are orders too high to be saved completely. A trigger system is required to select the interesting events to be saved for further

[1] http://www.xilinx.com

analysis. This trigger decision requires a lot of calculations and data management impossible with common sequential processor systems. Fast ASIC solutions are very expensive for small production series and give no flexibility once implemented, so FPGAs are used to parallelize these calculations and evaluate a trigger condition. Another FPGA application is the implementation of embedded systems for controlling tasks. FPGAs are able to run full operating systems. The flexibility on both, the hardware and the software aspects opens a broad field of applications. FPGAs have successfully been integrated in the current experimental buildup at CERN[2] and play a key role in the development of the *Condensed Baryonic Matter*[3] (CBM) experiment at GSI Darmstadt.

However, there are some problems in using commercial off-the-shelf (COTS) FPGAs in particle detectors, avionics or space missions. These FPGAs are susceptible to radiation induced configuration and user logic changes. The FPGA itself is usually not damaged, but its current configuration is altered due to radiation. The correct behavior of the circuit in a radiative environment can therefore not be guaranteed.

This work addresses mitigation strategies for radiation induced effects in SRAM based FPGAs applied on a fault tolerant softcore CPU. Chapter two describes the targeting FPGA architecture and chapter three explains the possible effects of radiation on these FPGAs. Chapter four shows an overview of previously done work to mitigate radiation effects. This work's approach, in contrast to others, and how it was implemented is described in chapter five and six. The effectiveness of the applied methods on the implemented designs have been verified with both error simulation and particle beam test as described in chapters seven and eight. Finally, all relevant aspects of the work will be drawn together in chapter nine.

[2]http://www.cern.ch
[3]http://www.gsi.de/fair/experiments/CBM/

2 FPGA Architecture

A Field Programmable Gate Array (FPGA) is an array of several different logic elements packed into a single semiconductor device. In contrast to common electronic devices, the logic functions of the single internal elements and their connection with each other are volatile and can be programmed by the user at any time. Several input and output pins can easily be used to make connections to almost any other electronic component with the common electrical signalling standards. Only FPGAs from Xilinx[1] have been used in this work, so architectures from other vendors have been neglected in the following descriptions.

The elements forming the array structure in a Xilinx Virtex-4 FPGA are called *Configurable Logic Blocks* (CLB). As the name implies, the behavior of these logic blocks and their connection with each other are configurable by the user. A configurable switch matrix attached to any CLB grants access to the chip wide routing network. The CLBs are arranged periodically in rows and columns and cover the main part of the FPGA. Apart from the CLBs, there are some further configurable columns containing *Block-RAMs* (BRAM) as on-chip storage, *Digital Signal Processors* (DSP) for specific calculations, *Multi Gigabit Tranceivers* (MGT) for fast serial communication, and *Input/Output Buffers* (IOB) as connection into and out of the FPGA. The FPGAs from Xilinx' FX line have also hard wired *IBM PowerPC RISC Processor Cores* [Xil08d]. A sample arrangement is shown in figure 2.1.

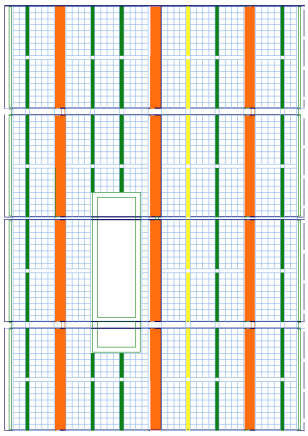
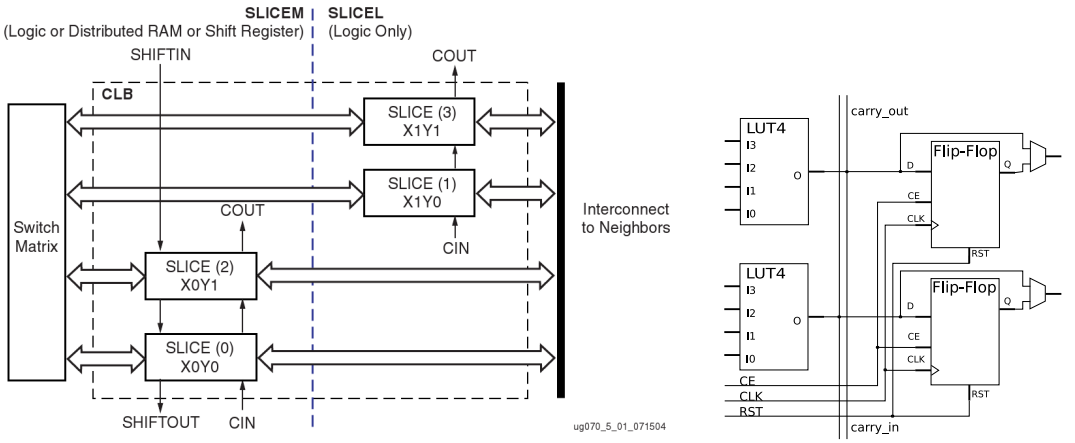


Figure 2.1: Simplified floorplan from Xilinx PlanAhead for a Virtex-4 FX20 FPGA. The blue bordered rectangles are CLBs, BRAMs are colored green, IOBs orange, DSPs yellow. The white rectangle in the center is a PowerPC core.

The CLBs are subdivided into finer grained programmable logic blocks called *slices*, fast connections between them and connections to the chip-wide routing network. There are two types of slices called SLICEM and SLICEL, M standing for *memory* and L for *logic*. The look-up tables in the SLICEM components possess the additional ability that their configuration doesn't need to be completely static during runtime, so they can be used as cheap distributed memories or shift registers. Both types of slices are available in equal numbers. A CLB consists of two columns, each with two slices. The left column contains

[1] http://www.xilinx.com



(a) Contents of a CLB [Xil08e] (b) Contents of a slice

Figure 2.2: Arrangement of slices within a CLB (left) and simplified internals of a slice (right)

SLICEMs, the right SLICELs. Each CLB is connected to its neighbors and a chip-wide configurable routing network. A sketch of a CLB is shown in figure 2.2(a).

The slices mainly consist of look-up tables (LUT), flip-flops (FF), Multiplexers (MUX), and some basic gates. Look-up tables are small programmable memory blocks that produce an output vector to any input vector. The input vector can be understood as an address-line selecting one of the memory values and giving this value to the LUT-output. Current FPGA architectures have look-up tables with four to six inputs and one to two outputs. With the according configuration, any boolean function of the input vectors can be realized. Therefore FPGAs do not need to supply basic gates like AND, OR, NOT, XOR etc. as any combination of those gates can be realized with look-up tables. Unlike the gate implementation, the propagation delay through a LUT is independent of its logic function.Virtex-4 FPGAs have look-up tables with four inputs and one output. The flip-flops can be used as edge-triggered D-type flip-flops or as level sensitive latches. The control signals *reset*, *set* and *clock_enable* can be used as synchronous or asynchronous signals and the initial- and reset-values can be specified via configuration. Two LUTs and two flip-flops are grouped with some basic gates and multiplexers to form a slice. Both flip-flops in a slice use the same *clock*, *reset*, *set* and *clock_enable* signals which cannot come directly from a LUT within the same slice. These flip-flop control signals have the ability to be inverted at the entrance to the slice. The flip-flop's data input line is connected to the preceding LUT's output and the slices' output can come from the FF or directly from the LUT. Furthermore, there are *carry* and *shift* lines connecting proximate slices to allow the implementation of fast adders, comparators or shifters. Figure 2.2(b) illustrates the components within a slice, a more detailed description can be found in the *Virtex-4 User Guide* [Xil08e].

The routing between and within the different elements is done with multiplexers and

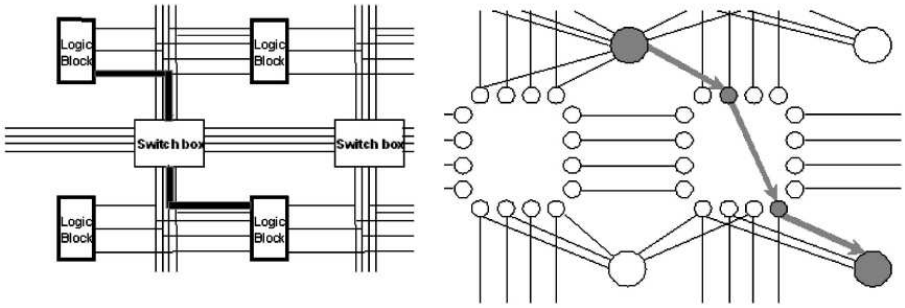


Figure 2.3: Model of the FPGA routing net. The connections between the available wires are configurable with programmable interconnect points (PIP). (Source: [SV06]).

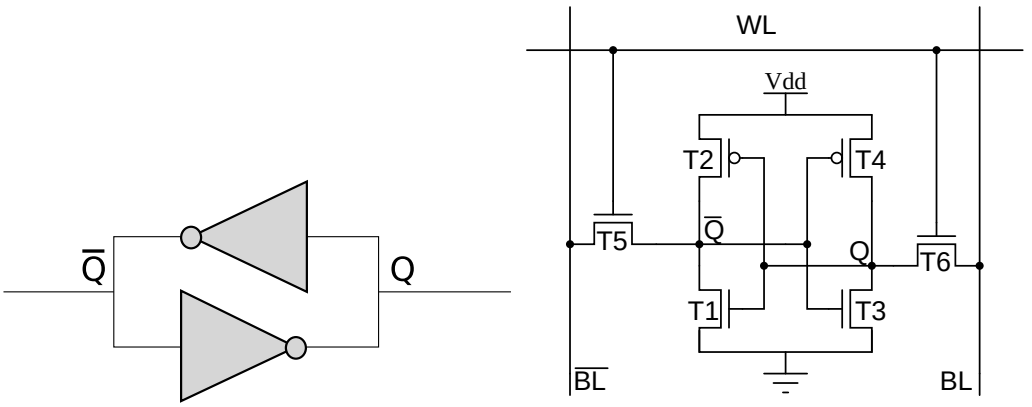
programmable interconnect points (PIP). Multiplexers with statically configured select lines control the signaling within the slices. Outside the slices, the routing is done via static wires. Only the connection between, not the wires themselves are configurable with PIPs. A model of the PIP based routing network is shown in figure 2.3. The PIPs are grouped in switch matrices on CLB level as shown in figure 2.2(a).

The clock signals for synchronous designs have a second, separate routing net spanning throughout the whole device. This global clock net can reach any clock inputs within the FPGA and can be controlled with clock buffers and digital clock managers (DCM). Apart from the global clock net, the FPGA is further divided into several clock regions to allow different local clocks for specific parts of the design.

All of the elements mentioned above have an underlying layer controlling their behavior. Any look-up table to be used must be filled with initial values to act like the desired logic function. Connections between two logic elements must be made by combining according parts of the routing net via switchable interconnect points. Any flip-flop can hold a reset value and can be defined as synchronous or asynchronous. Any IO-Buffer can be used as input or output, with several possible voltage levels just by setting the according configuration bits. This configuration layer is read- and writable for the user and thus makes the FPGA *field programmable*.

The configuration of the FPGA can be done by using one of the vendor supplied configuration interfaces. All of these interfaces are fed with a bitfile containing the values for any configuration bit in the FPGA. Details on the configuration interfaces and the configuration process are further described in chapter 7.

The whole FPGA used in this thesis, including configuration and user logic, is made of SRAM cells. SRAM stands for *Static Random Access Memory*. "Static" in this case means that the memory cell will keep its current value as long as it is powered on. This is in contrast to *dynamic* RAM (DRAM), which stores its values in small capacitors that lose their charge if not refreshed periodically. If the device is powered off, both types



(a) SRAM cell schematic

(b) six transistor CMOS implementation of an SRAM cell

Figure 2.4: SRAM cell schematic as bistable inverters (left) and its six transistor CMOS implementation (right)

of memory lose their contents. *Random Access* means that the memory contents can be read and written at any time. A SRAM cell is a bistable circuit element. It can easily be understood by imagining two oppositely arranged parallel inverters as shown in figure 2.4(a). The cell always holds two values: the value Q and its inversion $\overline{Q}$. As long as the drive strength on Q and $\overline{Q}$ is smaller than the drive strength of the inverters, the current value will be held as long as power is applied to the inverters. If the drive strength on Q and $\overline{Q}$ exceeds this limit, a new value is written into the cell, according to the levels of Q and $\overline{Q}$. The actual CMOS implementation shown in figure 2.4(b) is only little more complicated. On reads, both bit lines BL and $\overline{BL}$ are precharged weakly. By selecting the cell with its word line (WL), the transistors T5 and T6 get conductive and therefore Q and $\overline{Q}$ drive the bit lines to their stored values. On writes, the bit lines are driven stronger than T1-T4 can do. By selecting the cell with its word line the previously stored values get overwritten with the values of BL and $\overline{BL}$. For more information about field effect transistor (FET) properties or circuits see [HH89].

3 Radiation Effects in SRAM-based FPGAs

After a brief theoretical background, this chapter gives an overview on any possible effects of radiation on semiconductor electronics. A focus is set on their impact on SRAM based FPGA applications.

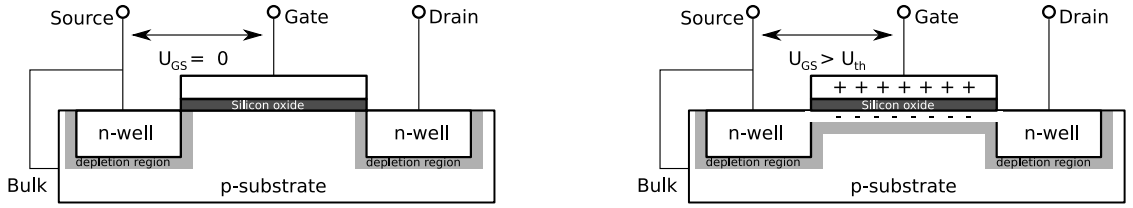
3.1 Theoretical Background

3.1.1 Electric Conductivity in Semiconductors

The correct theoretical explanation of electrical conductivity in crystalline solids is a highly complex problem requiring quantum mechanical calculations of the time dependent Schrödinger equation in external fields. However, it is mostly sufficient to use semi-classical models of electron wave packets with effective mass and speed, derived from their interaction with the crystal lattice in form of electron-phonon-scattering, electron-defect-scattering or electron-electron-scattering. This leads to a theory of electrical conductivity being highly dependent on temperature, lattice structure and its defects like foreign, missing, additional or displaced atoms.

The conductivity can be described with a band structure represented by an orbital-like model of atoms in a lattice. There are two bands: the valance and the conduction band. Charge carriers in the valence band are bound to lattice atoms, whereas the carriers in the conduction band are free to move. By exciting an atom in the lattice, an electron is lifted to the conduction band, leaving a hole in the valence band. With an electric field attached to this solid, the now free electron will move along the field. The left hole is filled with electrons from a neighbor atom by leaving a hole there. Thus, the hole effectively moves to the opposite direction. Both, the electrons and the holes give their share to the conductivity, but one can easily imagine, that the mobility of electrons is higher than the mobility of the holes. More information about models of electric conductivity and band structures in solids can be found in [Kit04].

A solid is an electrical conductor if there are always free electrons in the conduction band, and it is an isolator if there are no free carriers in this band. As thermal energy can excite atoms, this is a temperature dependent issue. Semiconductors are solids with a small band gap of one to four eV and they are isolators for small temperatures. The conductivity increases with temperature as more and more electron hole pairs can be excited with



(a) Nonconducting NMOS transistor

(b) active NMOS transistor

Figure 3.1: Simplified sketch of NMOS functionality

the thermal energy. Conductivity can further be affected by adding foreign atoms into the lattice (doping). Adding atoms with more or less valence electrons than the base material results in a n- or p-doped semiconductor. This creates states for electrons or holes between the former valance and conduction bands and therefore moves the Fermi energy of the solid. Combining p and n-doped silicon gives the well known depletion regions and potential effects exploited by diodes and transistors.

3.1.2 MOSFET Basics

A metal oxide semiconductor field effect transistor (MOSFET) is a voltage controlled current source. It has four connections: drain, source, gate and bulk. The gate is isolated from the other ports with a thin layer of silicon oxide and the current from source to drain is controlled by the voltage applied between gate and source. The bulk connection is used to keep the substrate on a well defined potential and is usually directly connected to the source. The functionality is exemplary shown on a n-type MOSFET (NMOS) in figure 3.1. The p-substrate holds a lot of free holes, but only few free electrons, whereas the n-wells hold more free electrons than holes. Without a positive voltage applied between gate and source, the connection between source and drain acts like a reverse biased diode and only small sub-threshold leakage currents can flow. By applying a positive voltage between gate and source, an electric field between gate and bulk is formed. Exceeding a specific threshold voltage, this field pushes the holes away and pulls some of the free electrons in the p-substrate to the interface between bulk and gate oxide. This forms a n-conducting channel between source and drain. The voltage applied on the gate can therefore control the current between source and drain.

Devices with the opposite doping of NMOS are PMOS transistors. In PMOS transistors the bulk substrate is n-doped and the source and drain wells are p-doped. The functionality is quite similar, but the PMOS is conducting without a gate source voltage applied and is based on the holes as charge carriers. PMOS transistors are therefore slower than their n-doped counterparts. PMOS transistors can be placed on the same substrate as NMOS transistors, if a n-doped region for the whole PMOS is created in the p-substrate or vice versa.

The combination of both, PMOS and NMOS transistors on the same substrate is called *Complementary MOS* (CMOS) technology and is nowadays one of the most frequently used logic family. Any logic gates can be created with CMOS. The SRAM cell in figure 2.4(b) is an example, too.

3.1.3 Passage of Particles through Matter

Almost any kind of radiation and particle can have an effect passing through matter. The effect depends on the incoming particle, its energy and the target material. Concerning radiation effects in semiconductors, the following radiation particles and effects can be distinguished:

- charged leptons like electrons or muons
  → bremsstrahlung and ionization effects
- charged hadrons like protons or α particles up to heavy ions
  → ionization and nuclear effects
- charge less hadrons like neutrons
  → no direct ionization, nuclear effects only
- γ particles
  → ionization, coulomb scattering, pair production

Both charged leptons and charged hadrons can interact with matter by ionizing atoms in the target material. The incoming particle ionizes the target material's atoms by scattering with their electrons. This ionization process creates electron hole pairs along the particle's way through the target material. The electron hole pairs can recombine within a short time as long as they are not separated by external electric fields. The energy required for the creation of a single electron hole pair in silicon semiconductors has been measured to 3.6 eV [Lut07], whereas in silicon oxide 17 eV are required [Sch96]. The energy loss for charged hadrons due to ionization can be calculated with the Bethe-Bloch-formula giving an energy loss per distance:

$$\frac{dE}{dx} = 2\pi N_0 r_e^2 m_e c^2 \rho \frac{Z}{A}\frac{z^2}{\beta^2}[ln(\frac{2m_e\gamma^2 v^2 W_{max}}{I^2} - 2\beta^2 - \delta] \tag{3.1}$$

with N_0 the Avogadro Number, r_e the classical electron radius $\frac{e^2}{4\pi m_e c^2}$, m_e the electron mass, Z and A the atomic number and weight of the medium, z the charge of the incident particle, ρ the density of the medium, I an effective ionization potential, v the velocity of the incident particle, $\beta = \frac{v}{c}$, $\gamma = \frac{1}{\sqrt{1-\beta^2}}$, δ a density correction and W_{max} the maximum energy transfer in a single collision. The calculation of I, δ and W_{max} can be found in [Leo94] chapter 2.2.2.

The resulting unit of the energy loss is $[\frac{MeV}{cm}]$. A typical unit in radiation tests of electronic devices is the Linear Energy Transfer (LET) defined as $[\frac{MeVcm^2}{mg}]$. This can be achieved by

dividing formula 3.1 by the target material's density ρ. The Calculation of the energy loss of a specific particle going through a defined material can easily be done for the entry point to the material, but gives non trivial differential equations, if the particle significantly slows down within the material. There are some simulators for energy loss determinations like SRIM[1], FLUKA[2] or the TVDG LET Calculator[3] to simplify life with these calculations. The ionization losses as calculated above cover the main part of energy transfer for heavy ion and hadron scattering. This model is not sufficient for light charged particles like electrons.

Bremsstrahlung occurs, when light charged particles are influenced by the electric field of the target material's atom nuclei. The nuclei change the trajectory of the incoming particle due to acceleration in the nuclei's electric field. Any acceleration of a charged particle leads to an emission of bremsstrahlung in form of a photon. The energy loss due to Bremsstrahlung is dependent on the incoming particle's energy, the screening of the target material's nucleon charge and therefore the target material itself. Due to the small lepton mass, bremsstrahlung covers the main part of energy loss for charged leptons above a few MeV ([Leo94], p.37). For low energies, ionization is dominating over Bremsstrahlung. The energy loss of charged leptons is therefore a sum of ionization losses and bremsstrahlung losses. The formula for calculating the ionization losses for electrons is quite similar to the Bethe-Bloch formula shown above, but takes into account that electrons are much lighter than hadrons and therefore their trajectory significantly changes on collisions. Furthermore, the scattering is now done between identical, indistinguishable particles. For calculation details and examples, see [Leo94] chapter 2.4.

Nuclear effects occur due to the scattering of incoming particles with the nuclei of the target material. This includes elastic and inelastic scattering resulting in movement, excitation or decay of the target nuclei. Moving atoms in a semiconductor can lead to lattice defects having a direct impact on the semiconductor's band structure. Nuclear scattering is not directly ionizing, but the results from scattering induced nuclear reactions can be. A typical neutron induced nuclear reaction in silicon is the decay of p-doping Boron-10 into Lithium and an ionizing α particle.

γ particles interact with the electrons of the target material. This can result in electrons being excited to higher orbitals or atoms being ionized. For high energies, the γ particles can change to fermion anti-fermion pairs like electron and positron or quark and anti-quark within the target material (pair production). These particles can then interact with the material via ionizing or nuclear effects as described above.

[1] *http://www.srim.org*
[2] *http://www.fluka.org*
[3] *http://tvdg10.phy.bnl.gov/LETCalc.html*

3.2 Radiation Effects in SRAM Cells

The radiation processes affecting the correct behavior of semiconductor circuits in general, and particularly SRAM cells, are mainly ionization and nuclear scattering. An ionizing particle going through a semiconductor leaves a number of electron hole pairs along its way. These electron hole pairs would recombine within short time if they are not separated by electric fields. A running semiconductor device however holds a lot of electric fields because every single transistor relies on them. These fields are quite strong because the distances between two electric poles are in the order of several nanometers down to a few atom layers for the gate oxide. The electron hole pairs created from a charged particle going through a transistor with strong electric fields will not be able to completely recombine before they are separated by the electric field. The consequence are additional free charges in the semiconductor with different mobilities. The mobility of the electrons is higher than the mobility of the holes as described in chapter 3.1.1. Nuclear scattering processes play a role as these interactions can create ionizing particles or move atoms in the semiconductor's lattice structure. Both of these processes lead to two independent effects: cumulative effects and single event effects. Single event effects result, as the name implies, from single radiation particles, whereas the cumulative effects rely on the accumulation of effects from several particles during the whole device's lifetime.

3.2.1 Cumulative Effects

Cumulative effects are gradual effects during the whole lifetime of the radiated device. They rely on the accumulation of radiation effects and lead to a failure when a certain limit is reached. There are two types of cumulative effects: total ionizing dose (TID) and displacement.

TID effects result from charge collection in the transistor's gate oxide, at the silicon-to-oxide interface or in the field oxide between transistors. If electron hole pairs in the oxide do not recombine immediately after the particle strike, they get separated by the electric field. The electrons can leave the oxide quickly due to their higher mobility. The transportation of holes according to the external field is much slower than for electrons as they need to *hop* between localized states in the oxide [Sch96]. During their way to the gate-to-oxide or silicon-to-oxide interface, the holes may get trapped, forming positive oxide charges for both p- and n-channel transistors. These charges screen or increase the electric field from gate to bulk and thus shift the threshold voltage and affect the leakage current. These trapped oxide charges however can be annealed even with room temperature over time.

A further effect of holes moving through the oxide is the release of hydrogen ions out of the oxide structure. These ions can move to the silicon-to-oxide interface where they may

become interface traps [Sch96]. These traps form states in the band gap of the semiconductor exactly at the silicon-to-oxide interface being responsible for the voltage controlled conductivity of the transistor. For p-channel transistors, these traps are predominantly in the lower part of the band gap allowing positive charges. Traps in the upper part of the band gap are mostly formed for n-channel transistors enabling negative charges. In combination with the trapped positive charge in the oxide, these effects may compensate or add up. The interface traps do not anneal with room temperature like the oxide charge buildup and degrade the device with shifted threshold voltage and decreased carrier mobility [Sch96]. This leads to timing errors, increased current or uncontrollable transistors switching.

The TID value used for device characterization is the energy deposited in the material of interest in form of ionization. The unit of the TID is *Gray* or *rad* where $1\ \frac{Joule}{kg} = 1\ Gray = 100\ rad$. The value for the TID rate in a particle beam experiment can be derived from the LET value calculated with the Bethe-Bloch formula 3.1 or one of the simulators:

$$TID\ rate\ [\frac{rad}{s}] = 100 \cdot LET \cdot \phi \cdot e/A \tag{3.2}$$

The factor 100 is for the conversion from *Gray* to *rad*, LET is the calculated energy transfer in $[\frac{eV\ cm^2}{\mathbf{kg}}]$, ϕ the particle flux through the FPGA in $[\frac{1}{s}]$, e the elementary charge $1.602 \cdot 10^{-19}\ Joule$ and A the contributing FPGA area in $[cm^2]$. This calculation is due to its units independent of the thickness of the contributing oxide volume.

The maximum total ionizing dose an FPGA can handle depends on its silicon oxide and gate size and therefore its manufacturing process technology. There is a military standard testing method for TID resistance called *MIL-STD 883 Test Method 1019* [Dep97] to allow comparison between different electronic devices. TID measurements from Xilinx [FDLH08] according to this testing procedure gave a maximum TID resistance of 100 krad for the early Virtex devices with 220 nm technology. Decreasing technology size reduced the volume of TID susceptible silicon oxide and the gate volume and therefore led to an increased TID resistance. The Virtex-II architecture manufactured in 150nm technology already had a tolerance of about 200 krad and the following Virtex-II Pro FPGAs reached 250 krad according to test procedure 1019. The Virtex-4 devices used in this work are manufactured in 90nm technology and have a total ionizing dose resistance of around 300 krad. According to [FDLH08], the modern architectures in 90nm technology can handle doses up to 1 Mrad with "proper design margins". Virtex-5 devices in 65nm technology are supposed to handle even more.

A further cumulative effect is displacement. The recoil from a high energetic radiation particle can move the target particle in the semiconductor's lattice structure. The moved lattice atom generates an interstitial and leaves a vacancy. With enough energy, the hit atom can displace further lattice atoms and generate a defect cascade. These defects generate traps, can influence the carrier mobility or increase the thermal generation of electron hole pairs. According to [Sch96], about 90% of interstitial vacancy pairs recombine

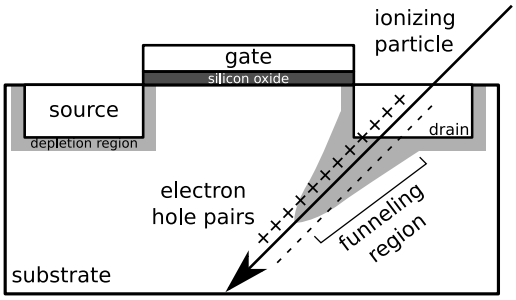


Figure 3.2: Single ionizing particle going through a p-n junction. The particle creates electron hole pairs that may get separated by electric fields before they can recombine. The resulting charge can have several effects. The charge in the funneling region contributes to the charge collection. Charges in deeper substrate regions do not affect the device.

within a minute after irradiation and the displacement effects are relatively unimportant for MOS transistors.

3.2.2 Single Event Effects

Single event effects (SEE) are the effects of single ionizing particles going through semiconductor electronics. The deposited charge can have different effects from temporary or correctable *soft errors* up to permanent and uncorrectable *hard errors*. As described above, an ionizing particle creates a high density electron hole plasma along its way through the semiconductor. The energy deposited within the semiconductor is defined as the linear energy transfer (LET) and can be calculated as shown in chapter 3.1.3. Under normal circumstances, only the charges deposited in the top silicon region should actually affect the circuit. But a charged particle going through a p-n junction extends the depletion region along its path. This effect is called *funneling*. The electron hole pairs in the funneling region therefore contribute to the total accumulated charge. If this total charge is big enough, it can considerably affect the circuit's behavior. But even if funneling significantly extends the volume from which the charge may be accumulated into the substrate, the contributing volume is in the top few micrometers of the substrate.

Hard Errors

Hard errors are errors that result in a physical damage of the radiated device. These errors cannot be recovered. Hard errors can be distinguished between single event burnout (SEBO), single event gate rupture (SEGR) and single event latch-up (SEL). They mostly induced by heavy ion particles because a lot of energy has to be deposited to cause these effects. Single event burnout affects solely high power devices like power MOSFETs, IGBTs and power diodes. Particles going through these multi-layered p-n devices can create a positive feedback of internal parasitic transistors getting conducting until breakdown and destruction of the device. Single event gate rupture is an effect in mainly power MOSFETs, but has also been observed in MOS transistors [Wro87]. Under normal conditions, the electric field between gate and substrate is large, but not large enough to

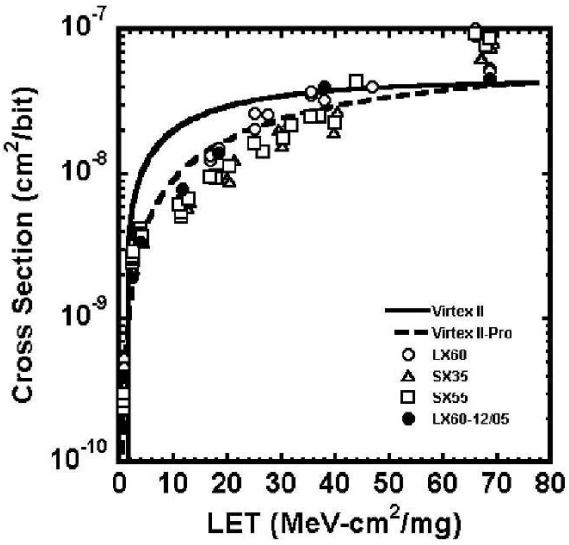


Figure 3.3: Comparison of Virtex-II and Virtex-4 heavy ion SEU cross sections vs. linear energy transfer (LET). The cross section starts increasing from a certain threshold LET and goes into saturation for high values of LET. The measured curves can be fitted using a Weibull fit. (Source: [GKS+06])

rupture the isolating silicon oxide layer. In case of an ion striking through gate oxide and substrate leaving an electron hole plasma, this plasma may get conducting. With a strong electric field, the resulting current could melt the oxide and destroy the device. Single event latch-up is an effect of closely placed p- and n-channel MOSFETs on the same substrate. The combination of differently doped regions automatically leads to parasitic transistors. This is no problem as long as all of these regions are kept to a fixed and well defined potential. But the situation can easily change if an ionizing particle deposits charge in the parasitic devices. The change of state of a single transistor can produce a positive feedback by enabling surrounding parasitic transistors leading to a latch-up. This latch-up can be interrupted by cutting the power supply, otherwise the device will be destroyed.

Hard errors are usually not a concern for CMOS circuits in space or particle detector applications because these errors are either very unlikely or the radiation energy is too low.

Soft Errors

Soft errors are errors that do not cause physical damage to a semiconductor device. These errors are temporary or can be corrected by reconfiguring the device. There are two types of soft errors: single event transients (SET) and single event upsets (SEU).

Single event upsets (SEU) are flipped bits of memory cells. As this work addresses SRAM based FPGAs, SRAM cells are here used for explanation. If an ionizing particle deposits enough charge in an SRAM cell, the content of the memory cell can change. Further, not any point within an SRAM cell will be eligible to change the memory content with any amount of charge. Thus, to produce an SEU a critical amount of charge in a sensitive

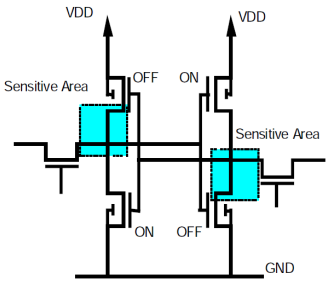


Figure 3.4: SEU sensitive volumes of SRAM cells. Only particle strikes within these colored areas and with enough energy are able to flip the memory cell. (Source: [Xila])

volume must be exceeded. The sensitive volume of an SRAM cell is sketched in figure 3.4. Particles with less energy or outside the sensitive volume will therefore just give a short current pulse without changing the memory cell's content. The dependency of SEU cross section and linear energy transfer (LET) is exemplary shown for a Virtex-4 FPGA and heavy ion radiation in figure 3.3. Below a certain threshold LET, there will not be any SEUs possible, because the deposit charge in the sensitive area will not be sufficient. Exceeding this threshold LET, more and more SEUs can be induced. This cross section seems to saturate for high values of LET and therefore simply depends on the probability of hitting the right parts of the SRAM cell. The shape of this cross section vs. LET behavior can mathematically described with a Weibull fit. A detailed explanation of the shape of this cross section vs. LET curve in combination with the mathematical formulas describing the Weibull model can be found in [Edm96]. In an FPGA single event upsets show up as flipped memory bits, in both, the device configuration and the user logic flip-flops. The possible effects of SEUs in an FPGA are shown in chapter 3.3

The second class of soft errors are single event transients (SET). SETs are based on the same radiation effect as SEUs, but do not require to hit the sensitive area of an SRAM cell. The deposit of charge in the semiconductor leads to short current pulses. The magnitude of these pulses highly depends on the deposited charge and the capacity of the hit line. These pulses are usually in the order of 100 to 200 picoseconds in CMOS circuits and can propagate as glitches in the user logic [ME00]. SETs can have serious consequences if the affected signal is sampled by a system clock as shown in figure 3.5. As long as a SET arrives at a flip-flop's data input in the absence of the sampling clock edge, the SET will not have an effect. This changes, if both, the SET glitch and the sampling clock edge come approximately at the same time. A SET being sampled correctly at a clock edge becomes static and has the same effects as a direct SEU. A SET occurring at a certain moment may violate setup or hold times and therefore produce unpredictable or even metastable flip-flop outputs. Further, the clock or reset lines may be affected by SETs, too. This could lead to desynchronized or spontaneously reseted parts of the FPGA. The rate of SETs becoming glitches is dependent of the flux and energy of the ionizing radiation. However, the probability of a SET being sampled and becoming an SEU is additionally dependent on the clock frequency. The more often a value is sampled, the higher the chance to hit a glitch.

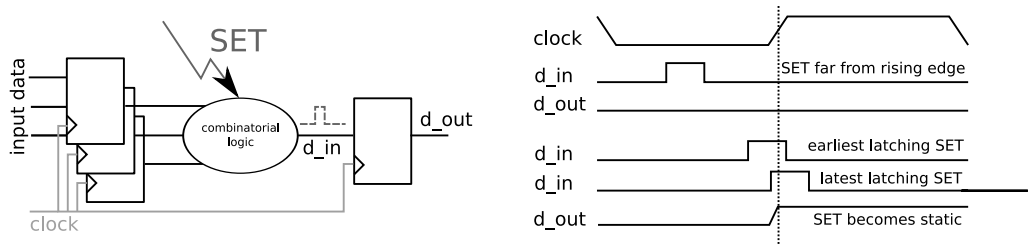


Figure 3.5: Effects of SETs in synchronous designs. The SET induced glitch on *d_in* does not have an effect on *d_out* as long as it occurs apart from a rising clock edge at the flip-flop's input. A SET being sampled at a rising clock edge becomes a static upset and shows the same effects as a direct SEU. If the glitch violates setup or hold times, the flip-flop's output may be unpredictable or even metastable.

Single event upsets (SEU) and single event transients (SET) are the mainly observed effects when using semiconductor electronics in a radiative environment.

3.3 SEU Categories

The Virtex-4 FX20 FPGA used in this work has about 7 million configuration bits and around 17.000 user flip-flops. The probability of changing an user flip-flop is therefore small compared to the probability of changing a configuration bit. A single event upset in the device configuration can have several effects on the running design. A classification of possible SEU effects in a Virtex-I FPGA has been done in [GCZ03]. According to this source, around 80% of all SEUs affect the routing. The majority of remaining SEUs can be identified as look-up table value changes and upsets of the bits controlling miscellaneous functionality of the whole CLB or IOB. An overview of the most common SEU effects is shown in figure 3.6

3.3.1 Routing Effects

The SEU effects on the FPGA routing net have extensively been studied for Virtex devices by Xilinx [GCZ03] and Sterpone & Violante [SV06]. As described in chapter 2, the routing of signals through the FPGA is done with programmable interconnect points (PIP). The behavior of these PIPs is controlled by configuration bits. Long lines may be assisted with switchable buffers. A SEU in a PIP may open or shorten two wires. An open simply disconnects the two lines, so no further signal transmission between them is possible. A PIP short can affect both sides of the PIP as both logic levels now depend on each other. SEUs in the buffer control bits only affect the driven wire and do not have a direct feedback effect. Another routing effect are SEUs in the select line configuration of multiplexers (MUX). If these bits are changed, a different signal will be forwarded through the MUX.

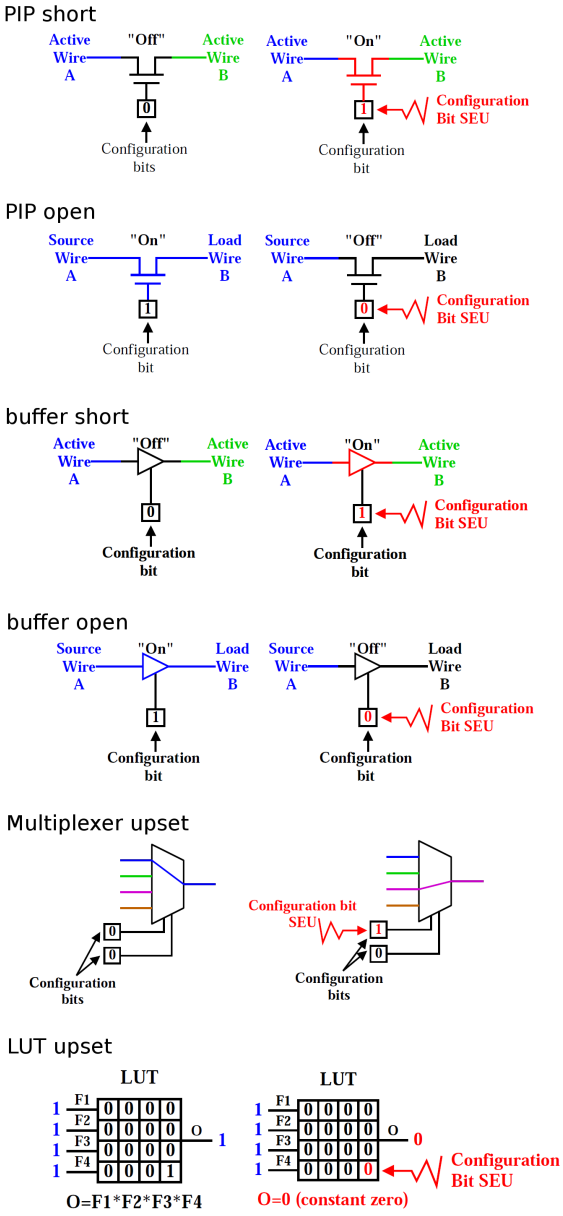


Figure 3.6: Overview of SEU effects in routing and look-up tables. Programmable interconnect points (PIP) in the routing net can be shortened or opened affecting both sides of the PIP. Shorts or opens in buffers have only an effect on the load wire. Multiplexers controlled by configuration bits can select wrong signals on SEUs. A SEU in the LUT values leads to a change of the implemented logic function. SEU effects on flip-flops, CLB control bits or on any other FPGA block beside the CLBs are not shown in this picture. These images are taken from [GCZ03].

Both MUX, buffer and PIP shorts may increase the current consumption as wires driven with V_{dd} could get connected to grounded wires. According to the work of Sterpone & Violante, a single event upset can even have multiple effects on pairs of routing nets. The authors claim, that a single event upset can shorten two nets, delete multiple connections or re-route an existing net by deleting the old and adding a new net due to a decoded PIP configuration in Virtex FPGAs. Unfortunately, no studies on SEU routing effects for Virtex-4 devices could be found. The architecture and the arrangement of blocks in the FPGA changed since Virtex-I, but the principles should have stayed the same.

3.3.2 SEUs in Slices

The main contents of the slices are a look-up table (LUT), a flip-flop (FF), several multiplexers, gates and wires connecting anything with each other as described in chapter 2. A single event upset in a LUT changes one of the memory cells. This implies a change of the currently implemented logic function and therefore can lead to a change of the device's functionality. The look-up tables of SLICEMs have the ability to be used as distributed memory or shift register. The operating mode is defined with configuration bits and can thus be changed by SEUs. Upsets in a flip-flop can directly change its content, but can also change its configuration. The initial- and reset values plus whether it is used as latch or flip-flop is defined with configuration bits. Another set of configuration bits controls whether the *clock*, *clock_enable* or *reset* inputs are inverted. A SEU in these bits is likely to produce timing errors as the flip-flop may get its control signals at the wrong clock edge. The multiplexers and routing lines within the slices can be affected in the same way as described for the inter-slice routing.

3.3.3 SEUs in I/O Buffers

The Virtex input/output buffers (IOBs) deliver a lot of configuration options. This configuration decides whether the buffer is used as input, output or bi-directional tri-state buffer and defines the electrical standard to be used out of 16 possibilities. Furthermore, each IOB contains three flip-flops to enable registering of the values read or to be written. Radiation induced configuration changes in these bits could have a large impact on the whole system of FPGA and its peripherals. The behavior of the Virtex IOBs in the presence of SEUs has been studied by Wirthlin, Rollins, Caffrey and Graham in 2002 [RWCG02]. They recognized that only one single bit out of 324 IOB configuration bits and two two-bit-combinations of flipped configuration bits were able to flip the IOB's output value or to actually change an input pin to an actively driven output pin for Virtex FPGAs. The probability of destabilizing a whole system due to SEUs in the FPGA IOBs is therefore relatively low. As for the routing effects, no explicit Virtex-4 characterization of IOB SEU susceptibility could be found, but it looks like the IOBs did not change a lot.

3.3.4 SEUs in BRAMs

The block RAM (BRAM) memories are made of SRAM storage cells as well, so they are also susceptible to SEUs. According to a Virtex-4 SEU study from George, Koga, Swift, Allen, Carmichael and Tseng in 2006 [GKS+06], the BRAMs have an even higher cross section for SEUs than the CLB configuration bits. The authors assume an explanation for this result in differences in the manufacturing. The CLB SRAM cells have larger channels, a thicker oxide and contain more metal than the BRAM cells. The actual cross sections for BRAMs and CLBs compared to Virtex-II devices in proton and heavy ion beam tests can be found in [GKS+06]. However, the number of BRAM bits on the FPGA is smaller than the number of CLB/IOB configuration bits, so the reduced probability of hitting a BRAM instead of an non-BRAM bit may mitigate this increased BRAM SEU cross section.

3.3.5 SEUs in further Parts of the FPGA

According to [GKS+06], SEUs in Virtex-4 FPGAs can also affect the power-on-reset (POR) circuit initiating a full or partial reset of the FPGA. Furthermore, the configuration ports like JTAG or SelectMAP could be hit, interrupting configuration processes or requiring a power cycle to reconfigure the FPGA. An upset characterization of the PowerPC hard core processor implemented in the Xilinx FX series was done for both Virtex-II and Virtex-4 by Allen, Swift and Miller in 2007 [ASM07]. As the PowerPC core will not be available anymore in the further Virtex-6 / Spartan-6 devices, these characterizations become obsolete. A specific upset study on DSP-blocks, MGT-blocks or clock managers could not be found. Another effect that can be found in literature is the SEU susceptibility of *half latches*. Half latches keep a line on a defined potential and are more efficient than doing the same by using LUTs. These circuit elements showed an SEU susceptibility in the Virtex and Virtex-II FPGAs but are no concern for Virtex-4 FPGAs [ASCT07].

3.4 Multi Bit Upsets

The recent technology changes lead to continuously decreasing device structure sizes, whereas the regions affected by ionizing radiation remain unchanged. The probability of single ionizing particles affecting several transistor structures or several SRAM cells increases with decreasing technology size. The consequences are increasing numbers of multi bit upsets (MBUs), single particle strikes changing several SRAM cells at a time. The cross sections for multi bit upsets in Virtex, Virtex-II, Virtex-II Pro and Virtex-4 FPGAs in proton and heavy ion beams are published in [QGK+05]. According to this source, MBUs have hardly been a problem in the first Virtex FPGAs but are an increasing concern for the newer devices. In proton tests with Virtex-4 devices, around 3% of all upsets can be identified as multi bit upsets. Heavy ion tests with Virtex-II and Virtex-II Pro FPGAs

demonstrated that up to a third of all events at the highest tested LET value were MBUs. The newer Virtex-5 FPGAs tested in [QMG+07] reached even 59% of MBUs in all upsets with heavy ion tests.

4 State of the Art

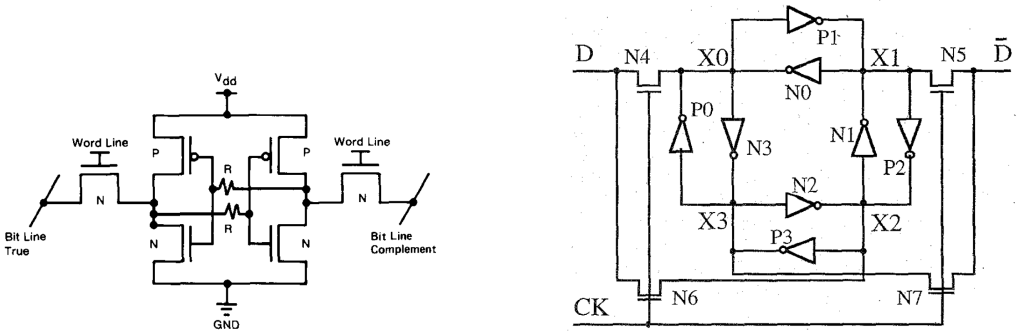
This chapter gives an overview of the commonly known and used SEU and SET mitigation techniques. There are mitigation strategies for all abstraction layers starting from modified CMOS circuits applying at the lowest level up to radiation tolerant FPGA architectures. Redundancy and data encoding can be applied in the user logic to detect or correct upsets. The exploitation of FPGA features and combinations of several methods have shown an increased radiation tolerance. Mitigation aspects for higher abstraction layers are touched in the last part of this chapter.

4.1 Radiation Hardened CMOS Logic

Any of the radiation effects described above rely on the principle of connecting differently doped silicon areas and voltage levels. A first effort would therefore be, to harden these CMOS cells against single event and cumulative effects. On the one hand, the amount of possible charge collection from a single particle can be decreased, one the other hand, the amount of charge required in a sensitive volume to flip a bit can be increased.

An approach to the first method is the use of an insulating layer between the doped areas and the substrate. These Silicon-on-Insulator (SOI) devices significantly reduce the amount of collectible charge as the depletion- and funneling regions are limited by the insulating layer. The complete isolation between n-well and p-well structures gives additionally an increased latch-up resistance, because parasitic pnpn-structures do not exist anymore [Fac99]. Another positive effect of SOI-structures is the reduced capacitance improving power consumption and maximum frequency. This layer is mostly made of silicon oxide or sapphire (Silicon-on-Sapphire, SOS). A similar approach, but with an increased area consumption, is the use of isolating guard rings around the transistors to remove parasitic transistors and to keep the substrate on a fixed potential [Bak07].

In order to increase the critical charge required in the sensitive volume, the capacitance of the nodes can be increased by increasing their size. A higher capacitance leads to a lower voltage swing with the same amount of charge collected. A popular approach is the use of additional resistors within the memory cell. These resistors affect the timing parameters of the cell and can compensate short current pulses. According to [Sch96], these resistors may not even increase the circuit area. A sample implementation is shown in figure 4.1(a).



(a) Resistor hardened SRAM cell

(b) Dual interlocked storage cell

Figure 4.1: The resistor hardened memory cell (left) can compensate short current pulses. The image is taken from [KSR+88]. The principle of a dual interlocked storage cell (DICE) is shown on the right. The image is taken from [CNV96]. Both approaches decrease the SEU susceptibility of SRAM cells.

There are further approaches hardening SRAM cells by using different design techniques. An example is the dual interlocked storage cell (DICE) [CNV96] shown in figure 4.1(b). This cell uses redundancy on CMOS level to store multiple instances of the desired values. Even if one part of the cell is modified by SEUs, the remaining instances restore the correct state of the hit part. The advantage of approaches like this is that they just require a change in the cell design, not in the manufacturing process like the SOI approach. A similar approach was chosen in [BV93] by creating a heavy ion tolerant memory cell (HIT cell).

The main disadvantage of all of these CMOS techniques is the effort to build an FPGA with them. They are a good option for custom ASIC designs but would be too expensive for small series of FPGAs with radiation tolerant CMOS designs.

4.2 Radiation Tolerant FPGA Architectures

The commercial off-the-shelf (COTS) SRAM based FPGAs have shown to be susceptible to both, radiation induced single event effects and cumulative effects. Apart from these devices, there are several architectures and technologies delivering increased radiation tolerance to FPGAs. The main representatives are flash FPGAs, antifuse FPGAs and radiation hardened SRAM FPGAs.

4.2.1 Flash FPGAs

Flash memories are unlike SRAM cells non-volatile and therefore do not lose their stored value when powered off. The principle of flash memories relies on the use of floating gate

(a) Floating gate transistor

(b) non-conducting and fused ONO structure

Figure 4.2: Flash and antifuse principles

transistors. The gates of these transistors are completely isolated from the other parts of the device. Above this floating gate is another control gate. By applying voltages on the control gate, the floating gate can be charged and uncharged by exploiting tunneling effects. Once the charge is deposited on the floating gate, it will remain there, regardless of whether the device is connected to a power supply or not. This non-volatile gate charge leads to a decreased power consumption compared to SRAM based implementations.

The main advantage of flash FPGAs regarding the operation in radiative environments is that the floating gate requires much more charge than a single ionizing particles can deposit or compensate. According to [SWC+99], a heavy ion with a LET of 37 $\frac{MeVcm^2}{mg}$ can only contribute less than 1% of the total charge on a floating gate. FPGAs built with this flash technology are therefore mostly resistant against single event upsets in the configuration memory. The user flip-flops are expected to be as susceptible to SEUs as their SRAM counterparts from the same manufacturing technology. SETs on the routing nets are still a concern. The effects of single event gate rupture (SEGR) are possible during configuration and the flash technology delivers no increased total ionizing dose resistance compared to SRAM based architectures of the same manufacturing process [SWC+99].

The disadvantage of flash based FPGAs is that they require additional manufacturing steps beyond the standard CMOS process and are therefore often some technology generations behind the newest CMOS technologies. Furthermore, the array structure of user configurable logic elements (*tiles*) is much simpler than in the Xilinx Virtex architectures and the supported clock frequencies are lower. The current flash FPGAs offer also on-chip RAM, but are missing things like digital signal processors (DSPs), multi gigabit transceivers(MGTs) or integrated Ethernet PHYs. Actel[1] is one of the most important manufacturers for flash based radiation hardened FPGAs.

4.2.2 Antifuse FPGAs

Another possibility to achieve increased radiation tolerance is the use of antifuse FPGAs. Actually antifuse FPGAs are more similar to ASICs than to SRAM FPGAs, as they are

[1] http://actel.com

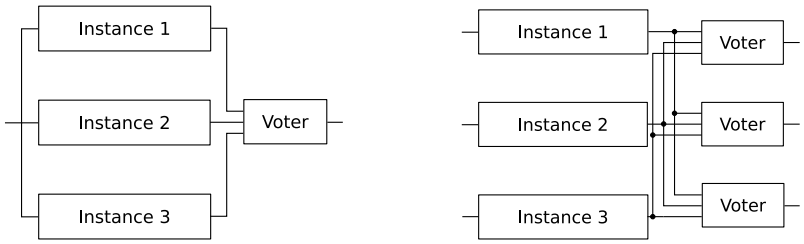
non-volatile and their configuration is writable only once. After that initial write, the configuration is static and cannot be changed anymore. These FPGAs use antifuse switches consisting of an oxide-nitride-oxide (ONO) layer sandwich. The programming can be done by applying high voltage pulses to the desired ONO structures and therefore fusing them. The TID resistance of current antifuse FPGAs is in the order of 300 krad [Act08] and the configuration can, due to its static nature, not be modified by SEUs. There are even antifuse FPGAs with SEU-hardened registers and integrated SRAM scrubbers like the Actel *RTAX-S/SL RadTolerant FPGA* [Act08] available for space applications. Those devices allow an excellent implementation of radiation tolerance, but their static nature denies any application requiring the ability to change the currently implemented design. An antifuse FPGA implementation of the LEON3FT fault tolerant softcore SPARC CPU system for avionic and space applications has been done by Actel in their *RTAX-S/SL* chips. The LEON3FT uses error correcting codes in any memories for up to four errors per 32 bit word or cache tag. A more detailed description of the LEON3FT Actel implementation can be found in [Aer09].

4.2.3 Radiation Tolerant Xilinx FPGAs

Xilinx delivers radiation tolerant versions of their SRAM based devices for space and military applications. These versions are based on the commercial Xilinx FPGAs and are currently available for Virtex-II, Virtex-II Pro, Virtex-4 and Virtex-5 architecture. For Virtex-4 architecture, these FPGAs are called Virtex-4 QPro-V [Xil08b] and are available in the same three platforms as their commercial counterparts: LX-series for high performance logic, SX-series for signal processing and FX-series with a PowerPC core. Xilinx delivers the Virtex-4 QPro-V FPGAs with a guaranteed resistance against total ionizing dose and single event latch-up combined with an SEU characterization. According to [Xil08b], the devices can handle a total ionizing dose of at least $250krad(Si)$ and are immune to single event latch-up up to a heavy ion linear energy transfer (LET) of $100\frac{MeVcm^2}{mg}$. The main difference to the commercial Virtex-4 FPGAs are

- a thin epitaxial layer during the wafer manufacturing process to increase SEL immunity
- a well defined test procedure for each wafer lot regarding the electrical specifications and timing parameters in combination with TID levels
- a full characterization for proton and heavy ion effects
- an increased maximum temperature range

Due to these improvements and characterizations, the Virtex-4 QPro-V FPGAs are much more expensive than their commercial counterparts.



(a) TMR with one voter (b) TMR with three voters

Figure 4.3: Triple Modular Redundancy (TMR) with one or three majority voters. TMR with one voter is resistant to single errors in one of the instances, but the voter itself is unsecured. By securing the voting process with triplication, every part of the surrounding design is needed three times.

4.3 Redundancy

Any of the radiation hardened FPGA architectures mentioned above have still a remaining susceptibility to at least SEUs in the user flip-flops and SETs in the routing net. The standard SRAM FPGAs are further vulnerable to changes in the device configuration as shown in chapter 3.3.

One solution to these problems is redundancy. Redundancy can be applied to almost any grain, from multiple instances of whole macroscopic systems like computers down to the replication of single gates within a device. If the same object exists in multiple instances, the probability for every instance failing decreases with the degree of redundancy. By assuming not more than one upset at a time, the duplication of an instance allows to detect differences between them. By triplicating the instances, at least a majority of correct results can be obtained. Higher orders of redundancy allow a higher reliability or a higher number of possible upsets at a time.

Redundancy can be implemented both temporal and spacial. Temporal redundancy allows to compare results from different times of execution and can be used to mitigate temporary errors. Spacial redundancy is implemented as several parallel instances calculating simultaneously in order to mitigate static errors.

4.3.1 Triple Modular Redundancy

Spacial redundancy is mostly implemented with Triple Modular Redundancy (TMR) and can be applied at any level. Regarding FPGA applications, TMR can imply the use of three independent FPGAs mounted on the same or even on different chips, the use of redundant IP-cores within a single FPGA down to triplication and voting of single look-up tables or flip-flops. Three independent modules do the same operation and their results

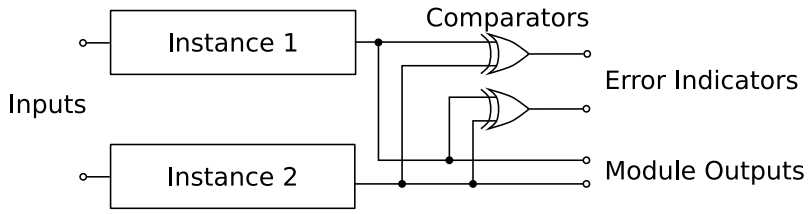
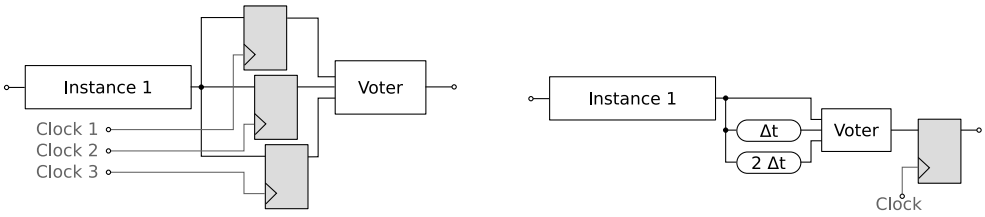


Figure 4.4: Implementation of Double Module Redundancy (DMR) / Duplication With Comparison (DWC). This method can only detect erros and is not able to determine which of the outputs is the faulty one.

are compared. A majority voter can determine the correct result even if one of the modules supplied a wrong value. But this principle does only work as long as the voter and the common input line for all three instances are correct. This cannot be guaranteed if they are implemented with FPGA logic. In order to deal with susceptible inputs and voters, all of the logic has to be triplicated. Three input signals were independently handled by three module instances and their outputs are voted with three majority voters. This version can even handle one upset in an instance and one upset in a voter simultaneously and still returns a majority of correct results. Sample implementations of TMR are shown in figure 4.3(a) and 4.3(b). The penalty of this method is its resource usage. As every part of the design has to be triplicated, the area increase for the triplication is a factor of three. Furthermore, voters have to be added. A voter for three signals can be implemented in one look-up table (LUT3). According to the grain of TMR implementation this can be one further LUT3 for each single flip-flop down to one additional LUT3 for each output of the IP-core or the whole system. The overall resource usage is therefore at least greater than three times the original usage and can even grow to a factor of six [WRCG03]. The power consumption of TMR-hardened designs will increase with the resource usage and the additional logic in each signal path leads inevitably to a decreased timing performance. TMR will also provide an improved tolerance to SETs, because even if one branch of the logic is temporarily affected by a SET, the majority will still hold the correct values.

4.3.2 Double Modular Redundancy

A spacial redundancy implementation with reduced area overhead but also reduced functionality is Double Modular Redundancy (DMR). As for TMR, DMR can be applied at any level, from whole systems down to single logic elements. DMR implies the duplication of logic blocks. The independent evaluation of two identical logic blocks with identical inputs allows to detect errors by comparing their results. If the results differ, one of the instances produced a wrong output. This method is often also named as *duplication with comparison* (DWC) [dLKNH$^+$04]. The limitation of this method is that a system with solely DMR cannot decide which of the signals is the faulty one in case of a difference. Like for TMR, it has to be assured that the input lines and the comparison logic deliver



(a) Sampling with different clocks

(b) Delaying the output signal

Figure 4.5: Examples for temporal sampling to mitigate SET effects. By sampling a signal with three differently shifted clocks (left), at most one flip-flop can get a wrong value due to SETs. A similar approach is to delay the data output in two steps (right).

reliable results, so they may have to be implemented with DMR, too. A sketch of a DMR implementation is shown in figure 4.4. Assuming single configuration upsets again, both comparators will indicate errors if the upset affected one of the instances or any routing net to the comparators. If single errors in the compare logic appear, at least one of the comparators will give a correct result. This, of course, could lead to *false positives*. There is also an increased tolerance to SETs as they could be detected by the comparators, however this tolerance is smaller than with TMR because SETs affecting the outputs may be missed by the comparators due to different signal propagation delays. The increase of area usage is a factor of two for the duplication of any logic plus the logic required for the comparison. The power consumption will increase with resource usage. Both, the area usage and the power consumption are lower than for TMR. This method will also decrease the maximum frequency or increase the latency because additional logic in the data path is required to compare outputs.

Double Module Redundancy is often combined with concurrent error detection (CED) schemes. Regarding a module F with input i and output F(i), a CED checker module predicts specific output characteristics of the concerned module F without the need of doing the identical operation. These predicted characteristics can be used to detect errors and can extend a DMR system with CEDs in each module. This extension makes it possible for the system to decide, which of both modules is the faulty one. As only characteristics are used for prediction, the number of detectable errors depends on the module and the used characteristics. Several concurrent error detection schemes have been evaluated and compared in [MM00].

4.3.3 Temporal Redundancy

Temporal redundancy evaluates the result of a module at different points of time. This form of redundancy addresses transient faults only, because static faults would give the same wrong result all the time. One possible implementation of temporal hardware redundancy is to sample the output of a module with different relatively shifted clocks as

shown in figure 4.5(a). As long as the transient width of the SET is smaller than the phase shift of the clocks, the SET will have no effect on the subsequent logic. If a transient occurs in the considered module, it will not arrive at the flip-flops data inputs in the same time with more than one of the three sampling clock edges. Thus, the worst case would be that one of the flip-flops actually samples the transient fault. Cases like this are then eased by the subsequent voter. Approaches to build temporal sampling latches relying on this principle are presented in [ME00].

Another approach to temporal redundancy is to sample differently delayed outputs of the same module with one clock as shown in figure 4.5(b). As long as the transient width is smaller than the signal delay, not more than one of the lines will be upset at a time. The voter will therefore keep the output stable. This approach is not well fitting for FPGA applications because fixed signal delays are not easy to implement.

The temporal redundancy approach with several sampling clocks can be combined with triple modular redundancy to achieve SEU and SET tolerance [RWS+07].

4.4 Error Detection and Correction

Another aspect of fault tolerance is the implementation of error detecting or even error correcting codes. The simplest example for error detection is the use of a parity bit. A parity bit is an additional bit added to a word of bits determining whether there is an odd or even number of logic ones in the considered word. The combination of word and parity bit makes it possible to detect single bit flips in the word or the parity bit. A parity bit can easily be created in hardware as it is only the XOR-operation of any bits in the word. The single parity bit algorithm is not capable of detecting more than one upset, because any even number of flipped bits will result in the same parity bit. However,there are several algorithms providing multiple error detection.

The Hamming error correcting code (ECC) is an example for an algorithm able to correct up to one and detect up to two errors. This code uses seven additional bits on a 32 bit word and eight bits on 64 bit words. The algorithm for calculating Hamming bits uses only XOR operations of bits in the word, but due to cleverly chosen combinations of these operations, single errors in both the word and the Hamming bits can be corrected with the according correction logic. This algorithm and sample HDL implementations of Hamming encoder and decoder are described in [Tam06].

Both parity and Hamming codes are common techniques when targeting applications using lossy mediums but reliable decoding/encoding hardware. More complex codes like *Reed-Solomon-Codes* are used for CompactDics, mobile phones or digital video broadcasting (DVB).

The theory of encoding combinations of bits can further be applied to FPGAs using state machines. State machines control the sequential behavior of hardware modules. The current state is hold in registers and the next state is derived from the current state and/or further inputs. Each state of a state machine is encoded as a unique sequence of bits. The most common encodings are *Gray*, using the binary representation of an incrementing number, or *One Hot*, using one bit per possible state. Single event upsets in these state bits can alter the current state to another or invalid state and thus induce an unexpected behavior of the whole system. By encoding these states with a hamming distance, invalid states due to SEUs can be detected and corrected. Several state machine applications and encodings have been compared with a focus on SEU resistance in [BT04].

4.5 Scrubbing

Previous investigations have shown that radiation can alter the state of the FPGA configuration bits and therefore change the behavior of the system. A logic consequence to achieve increased fault tolerance is, to try to keep this configuration mostly static. The Xilinx Virtex-4 FPGAs offer several interfaces for initial FPGA configuration and for reconfiguration during runtime. These interfaces allow any part of the FPGA configuration to be read back or written while the device is operating. The technique of writing only parts of the configuration is called dynamic partial reconfiguration (DPR) and is also a well known instrument for dynamic circuits and object orientated system approaches [AGM+08]. The advantages of reconfiguration during runtime in the presence of single event upsets is that the FPGA configuration can be read back, compared to a correct bitstream and rewritten into the FPGA correctly. This allows to identify, correct and avoid the accumulation of SEUs in the FPGA configuration memory.

The idea of correcting upsets with reconfiguration is called *scrubbing* and has already been published for the first Virtex FPGAs [AP98]. The implementation on the current Virtex-4 FPGAs is presented in a Xilinx application note [CT08]. There are basically two approaches to exploit this reconfiguration feature for SEU mitigation. The first approach is to continuously write the correct bitstream into the FPGA, regardless of whether there are upsets or not. This technique is easy to implement with a state machine reading the configuration from a memory and writing it into the FPGA. The second approach is to read back the configuration first, check if there are configuration errors by using checksums or by comparing it to a correct bitstream and finally correcting the configuration only if errors were found. Xilinx even recommends to read back a corrected configuration again after the write, to ensure that the error was actually corrected.

Reconfiguration can be done with either an FPGA internal configuration port accessible from the user logic called ICAP or via one of the external configuration ports like JTAG or SelectMAP. The configuration process is further described in chapter 7. The disadvantage of the internal reconfiguration port is that it is implemented with common FPGA logic

and therefore susceptible to SEUs itself. A comparison of the reliability of both, the internal and the external configuration ports has been published in [BPP+08]. Performing the reconfiguration via SelectMAP gives the opportunity to monitor the configuration control signals in order to detect errors in the configuration interface.

4.6 FastBoot

The idea of FastBoot is to take a snap shot of the complete FPGA system during runtime including all flip-flop states. By using the captured flip-flop states as the flip-flop reset states in a new bitfile, the whole system could be restored to this captured state by uploading the newly created bitfile. This technique can be used to mitigate the effects of uncorrectable errors. If snap shots are taken regularly, the latest snap shot can be used as reprogramming bitfile in case of uncorrectable errors. The system would restart at the latest captured point and does not need to restart "from zero". The current states of the flip-flops are accessible via the Xilinx FPGA configuration interfaces. There is still some work to do when using FastBoot in combination with peripherals requiring a power up sequence, but the reboot time of a system in case of uncorrectable errors could be reduced. More information about FastBoot can be found in [MK09].

4.7 Shielding

A quite simple SEE mitigation scheme is the shielding of the electronic device. Particles that do not reach the FPGA cannot induce radiation effects. Shielding is highly dependent on the radiation and can have negative effects, if the radiation is not completely stopped but only slowed down. The energy deposition of radiation in matter is a function of its energy and can increase with decreasing energy. High energetic particles can thus have a lower LET resulting in a lower SEU probability than the same particles with a lower energy. This effect is known as *Bragg peak* and can be derived from the Bethe-Bloch formula presented in chapter 3.1.3.

4.8 Automated and Combined SEE Mitigation Implementations

All of the methods presented above give an increased tolerance to single event effects. The TMR implementation is handled as the best "general purpose" solution for radiation tolerant FPGA designs and was compared to temporal redundancy, quadded logic and state machine encoding in [MMPW07]. There are several papers about how TMR can be implemented efficiently like [WRCG03], [Car06] or [KSCR05], however, they require a lot

of work to harden existing designs. Tool flows that automatically apply TMR techniques already exist.

Xilinx TMR-Tool is one possibility to automate the hardening of an existing FPGA design with TMR techniques. The tool promises complete SEU and SET immunity [Xilb] by applying triple modular redundancy in the whole design. It can be integrated into the common implementation tool flow and is advised to be used along with scrubbing.

Wirthlin et al. are developing a Java API for analyzing, creating and modifying EDIF netlists for FPGA desings. This approach includes tools to automate the implementation of both TMR and DWC. This software is open source and released under GNU GPL. A complete description with documentation and download links can be found on the project's homepage [Bri].

A further approach to improve the SEU tolerance of TMR designs is presented in [SV06]. The authors describe a custom *reliability orientated place and route algorithm*. This algorithm performs reliability orientated mapping and routing between logic functions in a way, that SEUs affecting two different connections are not possible anymore.

A common FPGA design uses a lot of configuration bits but not all of these bits actually affect the running design on SEUs. There are several approaches to indicate, which configuration bits in a design are actually *sensitive* to SEUs. In order to reduce the costs of full TMR regarding area and power consumption, there are approaches to apply TMR only to subsets of the original FPGA design [PCG+06] [SRK04]. One of these approaches is called "Partial TMR" and classifies all configuration bits according to their effects on the design. Only the SEU critical parts of the design are triplicated with TMR and therefore allow a trade-off of high SEU tolerance and acceptable area costs.

An approach applying triple modular redundancy with HDL functions is presented in [Hab02]. Radiation tests comparing the Xilinx TMR-Tool approach with this functional TMR approach have been accomplished with Virtex-II FPGAs by Saab Ericsson Space AB [SM04]. Among others, these tests have been done using a combination of scrubbing and redundant logic to avoid the accumulation of upsets.

There are also works addressing a significantly lower area overhead compared to the TMR techniques. The double modular redundancy approach has proved to be able to detect errors while scrubbing is able to correct them. The combination of both scrubbing and DMR/DWC enables the creation of systems with increased SEE tolerance and area costs significantly lower than TMR. One implementation is published in [BQS07]. The authors implemented several modules in DMR connected to a scrubbing mechanism. The detection of an error induces a partial reconfiguration of the module reporting the error. Another approach exploiting the combination of DMR and scrubbing is presented in [RVMR09]. An embedded softcore processor has been implemented twice as a master-checker-system. A checker logic implemented in TMR is used to detect differences between master and checker, to initiate the creation of checkpoints and to force a rollback

of previously stored checkpoints in case of errors. The presence of a memory scrubber is only assumed at this point.

The idea of implementing an additional *Watchdog Processor* was already published in 1988 [MM88]. A small and simple co-processor monitors the behavior of the main processor. Detected errors can be signaled to the checked processor or an external error handling unit.

A SEU and SET mitigation technique combining TMR and DMR with temporal redundancy and scrubbing is described in [LCR03]. The authors duplicate all logic and detect errors by comparison. The outputs of each of their duplicated modules were sampled with two clocks to mitigate SET effects. Any flip-flop is built with TMR to correct user logic upsets and scrubbing is used to correct the FPGA configuration.

4.9 Fault Tolerance in Higher Abstraction Layers

All hardware fault tolerance techniques proposed above rely on redundant or additional hardware. There are further approaches regarding fault tolerance on software level without the need of further hardware, but with increased runtime.

One aspect of providing software based fault tolerance to an embedded system could be to regularly perform a context save of all registers, the memory and the program counter. The latest snap shot can be restored, if an uncorrectable error occurs [RVMR09]. Another approach is redundant execution of software. Any command could be executed identically twice or with shifted operands to detect malfunctions in the hardware. One approach in this area using redundant execution in combination with validation instructions is published as *software implemented fault tolerance* (SWIFT) [RCV+05].

There are probably plenty of other methods providing fault tolerance on higher abstraction layers, yet, as this field is not relevant for the purpose of this work, only a short insight was intended.

5 Approach

The previous chapters have shown that SRAM based FPGA systems are susceptible to radiation induced memory changes and therefore cannot guarantee correct behavior of every single part of the FPGA in radiative environments. However, exploiting SEU mitigation techniques as described above can lead to a significantly increased SEU tolerance. The following chapter gives an overview of SEU mitigation techniques applied on this work.

5.1 Radiation Tolerance in a Multilayer System

The aim of this work is to apply fault tolerance techniques to SRAM based reconfigurable FPGAs not by blindly triplicating all logic, but by exploiting mitigation techniques on different layers to keep the area overhead on a moderate level. The commercial off-the-shelf (COTS) SRAM architecture is chosen not because of any improved radiation tolerance features compared to the competing architectures, but due to its flexibility and price. Indeed, according to the requested application, a radiation hardened SRAM FPGA, a flash based FPGA or even an antifuse solution may give a significantly better radiation tolerance. The aim is rather to find out what is possible with these cheap and well spread SRAM based FPGAs. The SRAM FPGA market is continuously growing, the flexibility and the possibilities for applications have increased with every Virtex generation, so why not use them in a radiative environment, if it is possible with a fair effort?

Different SEE mitigation techniques are used in this work to develop a radiation tolerant softcore CPU for SRAM based FPGAs. This CPU is designed to be part of a radiation tolerant multilayer system spanning all layers of modern FPGA based embedded systems. This includes the FPGA configuration layer and the actual FPGA system architecture with CPU and peripherals. An operating system contributes an abstraction layer for the hardware and allows applications to use it. A sketch of the layer structure is shown in figure 5.1. All these layers will give their share to an overall radiation hardened system by applying different error and radiation effect mitigation techniques on each level. As shown above, the FPGA configuration layer allows to be read and written in order to correct upsets or to apply FastBoot techniques. The implemented logic can be extended with redundancy to detect or even correct single event effects. An operating system could implement redundant execution of codes or context saves and software could be hardened with appropriate coding techniques or error detection schemes.

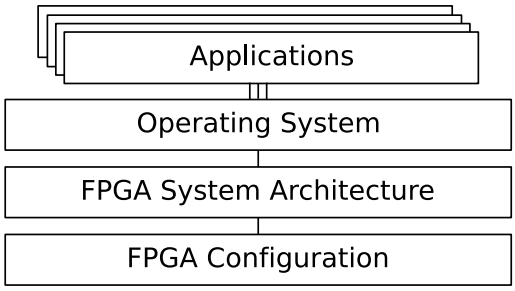


Figure 5.1: Layer structure of modern FPGA based embedded systems. In order to provide radiation tolerance in such a multilayer system, any layer can give a share by applying layer specific mitigation methods.

5.2 FPGA SEU Mitigation Techniques for the lowest Layers

A radiation tolerant softcore CPU for SRAM based FPGAs as part of a multilayer system covers only the lowest abstraction layers. Operating system or software aspects have no effect as long as a base for executing single commands has not yet been created. The mitigation techniques used in this work therefore refer to the lowest two layers: the FPGA configuration and the user logic implementation.

The most important issue when dealing with radiation effects in SRAM based FPGAs is to minimize the number of configuration upsets at a time. A system accumulating configuration upsets over time will almost in any case have a limit of errors it can deal with. As shown in the previous chapter, Xilinx Virtex-4 FPGAs offer the possibility to read or write their configuration during runtime. This feature can be used to correct configuration upsets while the system is running. Thus, the scrubbing technique as described above is one of the key features of this approach. The effectiveness of scrubbing is dependent on how fast the configuration can be written. The higher the scrubbing frequency for a given upset rate, the lower the statistical number of configuration upsets at the same time. This recommends the use of a fast configuration interface. Configuration errors due to single event upsets cannot be avoided, but they can be corrected with scrubbing within a short period of time. The application of FastBoot techniques is thinkable, but has not been examined in this work.

The second part of SEU mitigation is hardening the user logic implementation. The use of cheap commercial off-the-shelf (COTS) FPGAs as base architecture requires the use of redundancy techniques with an area overhead. As shown above, double modular redundancy (DMR) is just able to detect errors, but not to chose the correct result . Therefore, a general reaction on a detected error cannot be formulated. Many projects advice the use of triple modular redundancy (TMR) instead of DMR techniques to ensure that at least the majority of three signals is correct. However, the disadvantage of TMR is an increased area requirement of more than three times the original design.

The main aspect of this work is the exploitation of the combination of both, scrubbing and redundant logic. Scrubbing guarantees that within a short period of time a configuration error will be corrected. With this knowledge, it is sufficient to use double modular

redundancy to detect errors and wait for them to be corrected by the scrubbing mechanism. This of course demands that a software application tolerates a CPU waiting for an error to be corrected by scrubbing. Consequently, a system required to handle realtime conditions will hardly be possible with this method and time dependent software has to be considered carefully. If these limitations do not exclude the desired application, this method gives a remarkably increased radiation tolerance with a significantly smaller area overhead than TMR.

5.3 Physical System Layout

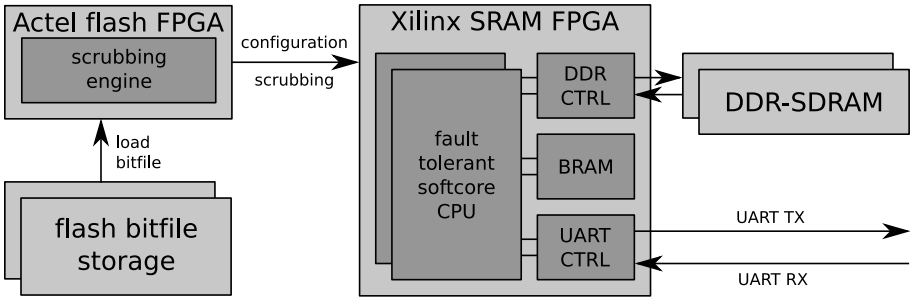


Figure 5.2: System layout. The light gray boxes are physical elements whereas the dark gray boxes represent the according FPGA configuration. A small flash FPGA is used to load bitfiles from a flash memory and perform scrubbing on the SRAM FPGA. The SRAM FPGA is the main part of the system holding the softcore CPU and its peripheral controllers.

The physical layout of the fault tolerant system is shown in figure 5.2. The heart of the whole system is an SRAM based Xilinx Virtex-4 FPGA. This FPGA will hold the softcore System-on-Chip (SoC) containing the fault tolerant CPU, a system bus, internal peripherals and controllers for external peripherals. These controllers can access the pins to the physical peripherals. Exemplary shown are DDR-SDRAM chips and a Universal Asynchronous Receiver Transmitter (UART) connection. The BRAMs are pure FPGA internal memory blocks. Possible further peripherals like Ethernet, SD-Card, MGTs etc. are not shown here. The initial FPGA configuration and the scrubbing process is performed by a small Actel flash FPGA connected to the Virtex' configuration interface. This flash FPGA continuously runs a scrubbing engine and uses dedicated flash memories to store the configuration bitfiles. The scrubbing engine already existed from a previous work [Roh08] and has hardly been touched in this work. The only change was the addition of a signal indicating the completion of a scrubbing cycle by sending a short pulse to the SRAM FPGA.

5.4 Choosing a suitable Softcore CPU

The base of a multilayer system is a CPU with a well defined instruction set. This instruction set allows commands to be sequentially executed and therefore software to be developed. With an appropriate system of CPU and peripherals, even an operating system can be used. As the wheel does not need to be reinvented, it should be considered to implement a CPU with an existing instruction set or better an existing processor architecture definition. This allows existing compilers and software to be used and simplifies the implementation of software and operating systems. A freely available radiation tolerant implementation of a CPU could not be found, so a standard softcore CPU has to be modified. As already mentioned, some requirements on the architecture simplify the future extension to a radiation tolerant multilayer system. The main demands on the softcore CPU with these purposes are:

- simple 32 bit architecture, not stack based
- HDL source code available to adapt fault tolerance
- HDL code actually synthesizable, not only a simulation model
- software compiler / GCC available to allow software development
- easy attachment of further peripherals
- optional: existing linux port

There are several different softcore CPUs for FPGAs available. The most famous softcore CPUs are Xilinx' Microblaze, Altera's[1] Nios and Gaisler's[2] LEON series. Unfortunately, none of them is really applicable for the requested needs. More promising are free custom implementation like the cores published on *opencores.org*.

Microblaze

Xilinx' Microblaze is a 32 bit RISC processor highly optimized for Xilinx FPGAs. It has a huge amount of configuration options including Memory Management Unit (MMU), cache, floating point unit, interrupt- and exception handling up to connections with several peripherals over different bus protocols. All these options are set with a graphical user interface (GUI) and do not need to touch any line of HDL code. In a standard Xilinx subscription, Microblaze comes as pre-synthesized netlists, so the actual HDL code can neither be seen nor changed. The full details about Microblaze can be found in [Xil08a]. The high optimization, complexity and the not freely available VHDL source code gave the criterion for exclusion.

[1] http://www.altera.com
[2] http://www.gaisler.com

LEON

Quite similar arguments fit to the LEON-processors of Gaisler . There are currently two active versions of LEON: LEON2 and its successor LEON3. LEON is a SPARC[3] CPU with a lot of configuration options and peripherals. Like Microblaze, there are MMU, cache, interrupt handling, bus connection and debug support. There is also a fault tolerant version of LEON3 available as described in chapter 4.2.2, but this VHDL design is meant as ASIC/antifuse FPGA prototype for avionic and space applications but not for SRAM FPGAs. This means, this design handles only bit upsets in system memory, caches and registers. It does not take any changes in routing or logic behavior into account as this cannot happen in an antifuse FPGA or ASIC implementation. In contrast to the standard LEON3, the fault tolerant version's VHDL code is not available under GNU GPL but must be purchased. Compared to Microblaze, the SPARC instruction set is more complex and things like rotating register windows make life really hard when trying to modify the code. More details about SPARC and its instruction set can be found in the *SPARC V8 Architecture Manual* [SPA].

Nios

Nios is Altera's counterpart to Xilinx' Microblaze and is therefore optimized for Altera FPGAs. The current version is NIOS II. It is also a 32 bit RISC architecture with customizable cache, MMU, debug modules and a lot of peripherals all attachable with a graphical user interface. More information about Nios can be found in [Alt09]. In contrast to Microblaze, Nios gives the possibility to add user defined instruction. Licensing is done through *Synopsys Designware*[4] as third party IP provider. The complexity and the need to port the Altera design to a Xilinx FPGA made a decision against Nios.

Custom Implementations

There are a lot of free implementations of several CPU architectures available on hosts like *opencores.org* or private pages. Unfortunately, a lot of them are rather eight or 16 bit architectures, are incomplete or hardly documented. The DLX architecture presented in the early versions of *Computer Architecture: A Quantitative Approach* [HP96] would have been a good candidate, because any part of the architecture is described in detail in this book. A lot of implementation of the DLX architecture combined with compilers, simulators and sample software could be found. However, none of them was actually synthezisable, but rather a simulation model only.

[3] http://www.sparc.org
[4] http://www.synopsys.com

A promising implementation meeting the most important requirements was found in Steve Rhoads' *Plasma - most MIPS I(TM) opcodes* project [Rho09] hosted on *opencores.org*. It has been chosen, because it is a quite simple 32 bit RISC microprocessor system with the most important peripherals already attached and ready to run. It is written in VHDL and can be synthesized with either a two or three stage pipeline. The CPU is implemented with the MIPS I instruction set able to run code produced from the standard GCC MIPS cross compiler. This CPU does not have a cache or a MMU, thus keeping the whole system simple. There is a lot of sample software from simple test programs up to a custom built small operating system with TCP/IP stack and running a webserver. A software environment for both developing and simulating own applications is also supplied. Even beginners are able to create own application running on the FPGA within some hours. Unfortunately, this design is not very flexible. Any peripheral is "hard wired" to the CPU without using a defined bus protocol. Thus, users need to have a deep knowledge of whats happening inside the CPU to be able to add custom peripherals. Furthermore, the interrupt/exception handling and the co-processor registers are not implemented as the MIPS specification suggests or are partially not existent. The VHDL code is written in a quite unusual way and the synthesized design is comparably slow.

These facts started a deep modification and finally lead to a complete rewrite reusing only minor parts of the original code.

5.5 The base CPU

The final version of the CPU is now quite similar to the MIPS R2000/R3000 system and is supplied with

- 32 bit address and data width, byte addressed, big or little endian
- a five stage pipeline
- a hardware multiplication and division unit
- interrupt and exception handling as suggested by the MIPS specification [KH91]
- a Wishbone bus interface as specified in [Her02]

The implemented instruction set can be found in appendix A. A graphical representation of the CPU system is shown in figure 5.3. The CPU and any peripherals can be controlled using the standard GCC MIPS cross compiler. The implemented MIPS CPU is a three operand architecture with 32 general purpose 32 bit registers and several co-processor registers used for interrupt and exception handling. As its predecessor, it neither has a cache nor a MMU at this time, whereas a cache for this architecture is currently in development. The lack of separate instruction and data caches require that both, instruction fetch and memory access share the same bus. The pipelined execution of commands is separated into *instruction fetch* (IF), *instruction decode* (DE), *register access* (RA), *execute* (EX)

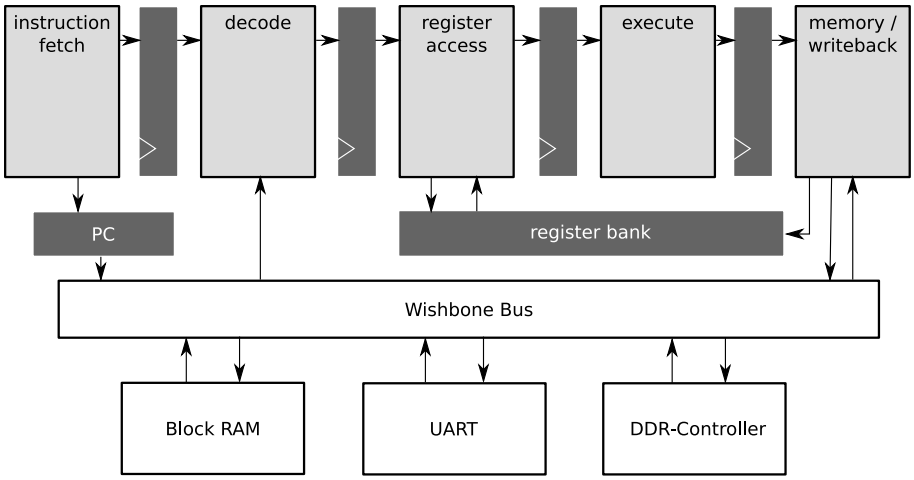


Figure 5.3: Pipeline stages and bus concept of the implemented MIPS CPU. The CPU is divided into five pipeline stages and can be attached to almost any peripherals using a Wishbone bus. Bus access is done using either the program counter (PC) or the address supplied from memory/writeback stage. The currently connected peripherals are an internal BRAM block, a UART interface and a DDR-SDRAM controller.

and *memory / writeback* (MW). IF-stage fetches the next instruction from the bus and DE-stage decodes it and creates the select signals for the multiplexers in the later stages. RA-stage loads the contents of the registers according to these signals. EX-stage performs the requested logic combination of the previously loaded register contents. MW-stage redirects the result back into one of the registers or performs a bus access. The Wishbone bus currently grants access to an internal BlockRAM, a UART controller and a DDR-SDRAM controller.

5.5.1 Interrupt and Exception Handling

Exceptions are a superset of all spontaneous events in the CPU. The currently implemented exceptions are interrupts, integer overflow and SYSCALL/BREAK instructions. All exceptions induce a jump to a predefined address, the *exception handler*. An integer overflow exception occurs, if signed additions or subtractions exceed limit the of the 32 bit number range and therefore produce a wrong result. SYSCALL and BREAK are special instructions directly inducing a jump to the exception handler by software.

An interrupt is an external signal raising an exception and thus forcing the CPU to jump to the exception handler. This exception handler disables further interrupts, saves the interrupted program counter address and all important register contents to memory, determines the source of the interrupt and handles it. After the interrupt is cleared, the saved register contents are restored, interrupts are re-enabled and the CPU returns to the previously interrupted program counter address. If the CPU is the only device being

able to initiate bus transfers, interrupts are the only possibility for peripherals to indicate events. An example for an interrupting peripheral is a UART interface. The CPU can write characters via the UART at any time, but cannot know when the UART will receive a character. One possibility would be, to continuously access the UART interface to check whether a character was received or not (active polling). This is not very efficient. The other possibility is, to make the UART raise an interrupt every time a character is received. The CPU gets interrupted, checks where the interrupt came from and can read the received character from the UART interface for further operations. The MIPS architecture delivers eight different interrupts. Six of them are external signals that can be connected to peripherals. The remaining two interrupts are software interrupts being raised by writing into a specific register by software. All of these interrupts can be enabled or disabled separately and have all the same priority.

5.5.2 The Wishbone Bus and its Peripherals

The Wishbone bus is a simple handshake protocol for communication between different interfaces. The specification can be found in [Her02]. The bus system is divided into master and slave devices, where only the masters can initiate transfers. The implemented CPU acts as a single bus master with its peripherals as bus slaves. The CPU can only initiate single transfers instead of bursts at this time and has a data width of 32 bit. The bus is addressing 32 bit words with a 30 bit address line and refers to single bytes within these words via four bit select lines. Further peripherals can easily be added into the address mapping.

The peripherals currently connected are an internal block memory (BRAM), a UART controller and a DDR-SDRAM controller.

5.6 Applying Fault Tolerance

In order to apply fault tolerance to a softcore CPU on FPGA logic level, some kind of redundancy has to be used. As described above, this approach tries to keep area overhead low by avoiding triple modular redundancy (TMR) in support of combining double modular redundancy (DMR) with scrubbing.

The CPU is fed with a sequence of instructions strongly related on each other. A sample sequence is shown below.

```
ADD R3, R1, R2     # R3 = R1 + R2
SW R3, R5          # write content of R3 to the address stored in R5
JR R3              # Jump to address stored in R3
```

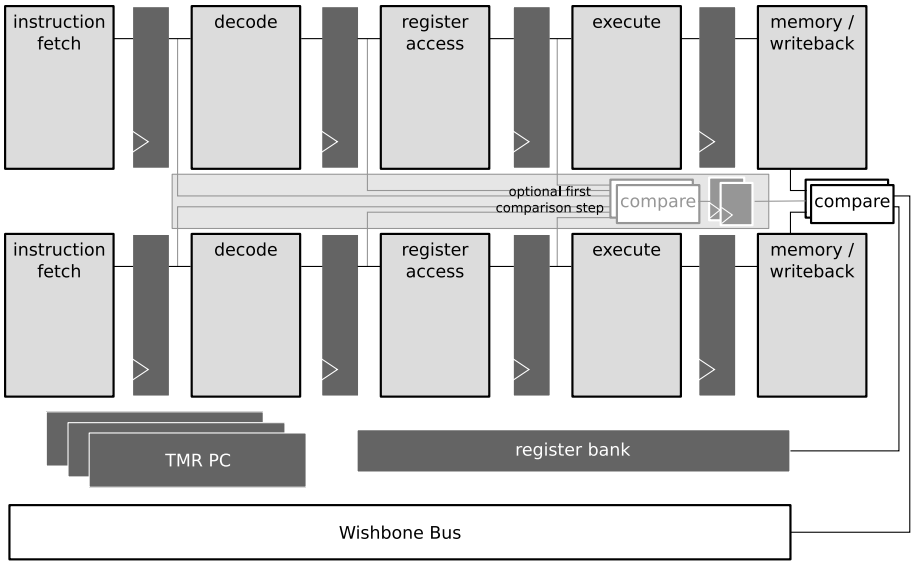


Figure 5.4: Sketch of the fault tolerant softcore CPU. The whole pipeline has been duplicated to detect SEEs and shares a common register bank. Comparison between the pipeline stages is done in two steps. The program counter is the only part of the CPU implemented with TMR.

If one of these instructions fails and returns a wrong result or a wrong destination address, any successive instructions relying on this result will not have any chance to complete correctly. If the addition in the example above fails and R3 gets a wrong value, this value is written to memory with the following instruction. By even jumping to the wrongly calculated address R3 with the third instruction shown above, the behavior of the system gets unpredictable. The CPU will produce wrong output or get stuck in parts of the code it shouldn't be. The key fact is: even if the functional behavior of the system is corrected, wrong register or memory values will affect the system sooner or later. The most critical part in hardening the CPU is therefore to avoid wrong data being written back into a register or in any other part of the memory.

5.6.1 Duplicating the Pipeline

DMR on the softcore CPU has been implemented by duplicating the whole pipeline. Any of the pipeline stages IF, DE, RA, EX and MW have been instantiated twice forming a completely identical second pipeline. Both pipelines operate independently and have their own pipeline control signals, but are fed with identical inputs. In order to detect SEEs, both pipelines are compared with each other. A sketch of this double pipeline CPU is shown in figure 5.4.

There are two possibilities to detect errors in the pipelines. One approach is to monitor a lot of equivalent signals from all pipeline stages to detect an error as early as possible.

The second approach is to reduce the comparison to only the outputs of the MW-stage in order to detect errors just before they would be written back. Both approaches have been implemented during this work.

The comparison between the pipeline stages is designed to be done in two steps. A first step compares the inputs of the stages DE, RA and EX, a second step uses the first step results and the outputs of the MW-stage. Both comparison approaches can be realized by either including the first step's comparison result to the second step or not.

The second comparison step has to be done combinatorially because its result is required in the same pipeline step to detect differences between both pipelines and prevent the writeback of faulty data. The first comparison step is not that time critical, because the instructions in these stages cannot perform any writing to registers or memory. The result of this comparison is registered and used as input to the second comparison step. Even if an error was detected in the first step, but forwarded to the second comparison step one cycle later, the pipeline shifted its instructions at maximum one step. The worst case would be that the instruction raising the error condition arrives at MW together with its error signal. The error is still handled before any wrong pipeline data can be written back.

The same argument allows the comparison of just the input signals to the pipeline stages instead of their results in the first comparison step. An error in the outputs of a pipeline stage may be undetected in one of the early pipeline stages unless the pipeline is shifted one step. The error will then result in a difference of the input signals to the next pipeline stage and gets detected. Errors like these are detected with a delay, but the detection will be in any case before the according instruction completes the MW-stage.

The advantage of splitting the comparison process into two steps is mainly due to performance aspects. In the first pipeline stages, a lot of signals are compared to detect a broad range of possible errors. If all of these signals would be compared in the MW-stage, a lot of comparisons had to be in the data path deciding whether a result can be written back or not. This would significantly decrease the maximum possible clock frequency and therefore the throughput. With this two step approach, only the result of the first comparison has to be included to the MW-stage output comparison.

For both comparison steps, any differences between the two instances of each pipeline stage are detected as errors, if at least one of them is marked as valid. Instructions marked invalid in both instances are not taken into account. The pipeline control signals from both pipelines are compared, too, and are evaluated in the first comparison step.

5.6.2 Duplicating the Compare Logic

As shown in the chapters above, any reduction from redundancy to a single signal is a SEE critical point. The final result of a comparison is typically one bit: '0' if the compared

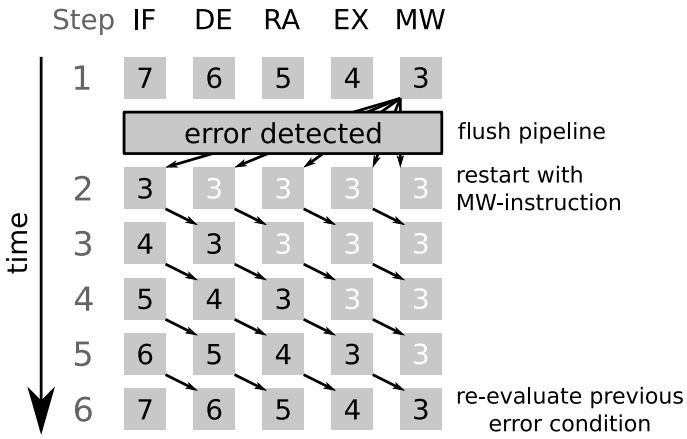


Figure 5.5: CPU behavior on error detection. If an error is detected, the pipeline is flushed by marking all instructions as invalid (white numbering) and restarts with the instruction in MW-stage. By copying the PC of the MW-stage in all pipeline stages, the CPU restarts with this instruction.

signals are equal, '1' if they differ. If this comparison logic gets struck, a reliable error detection is not possible anymore. In order to avoid this single point of failure as far as it is possible, the comparison logic is implemented twice, too. The optional first step and the required second comparison step are implemented with DMR, too. Both second step comparisons use the outputs of each first step comparison as inputs for their own results.

5.6.3 Reacting on Errors

If an error was detected with the compare logic described above, the CPU has to react on it. Error handling is only done according to the output of the second comparison step. This automatically includes errors detected in the first step delayed by one clock cycle. The most important fact is that in any case an error is detected, the CPU will not try to write to any memory. The pipeline is flushed by marking all instructions as invalid and copying the program counter of the MW-stage to any pipeline stages. This induces, that the address of the instruction in MW-stage at the time the error occurred is used as new address for instruction fetch. An example is shown in figure 5.5: While instruction 3 is being prepared for writeback, an error is detected and the result from 3 is not written back. The pipeline is marked as invalid (white numbering) and the program counter from the MW-stage is copied to any stage. The program counter of all states now points to address 3. As this includes the PC in IF-stage, the pipeline continues with fetching instruction 3. Four pipeline steps later, the pipeline's state is the same that resulted in an error before. If the error was a transient, the CPU will continue normally at that point. If the error is a static upset, the error condition will be raised again. This results in a loop until the error gets corrected by scrubbing. The error detected in this example does not necessarily come from instruction 3 but can also be an error raised by any of the earlier pipeline stages.

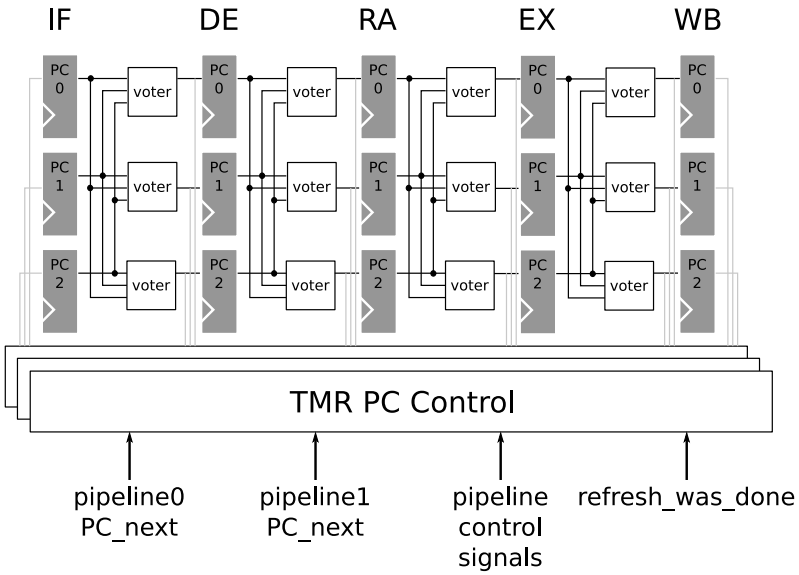


Figure 5.6: Triplicated program counter. The PC address from any pipeline stage is stored triple and voted to inputs of the next stage. The PC control unit handling the signals from both pipelines is implemented with TMR, too. *refresh_was_done* is an external signal from the flash FPGA indicating when a scrubbing cycle has completed.

5.6.4 Triplicating the Program Counter

The error handling techniques above have shown that the program counter is a very important part when recalculating failed instructions. If the program counter is altered or cannot set the according addresses reliably, all of the methods described above will fail. The program counter is the most sensitive part of the CPU and has therefore been implemented with TMR. This is the only part of the CPU that uses TMR. The program counter from any pipeline stage is stored triple and is voted to the inputs of the next pipeline stage. Only if both pipelines give identical control and data signals, the PC array is shifted to the next stages.

The experiment showed that the complexity of the system still holds a probability that raised error conditions may not be cleared with scrubbing. This results in an uncorrectable error and requires a reset of the CPU. In order to detect whether or not the CPU is still working, the program counter in the memory stage is monitored over time. A signal from the flash FPGA indicates when a scrubbing cycle has completed. The combination of both signals allows to detect, if the CPU's MW-stage PC address did not change for several scrubbing cycles and can initiate a reset of the CPU. By resetting the CPU, it restarts from PC address zero.

5.6.5 Register Bank

There are two choices when duplicating the pipeline: Any pipeline gets its own register bank or both pipelines share the same. Both versions have been implemented and tested during this work. The problem for both versions is, that the redundancy cannot be extended to the lowest levels. Any flip-flop has a single data input and a single clock

enable. Any redundant signal has to be reduced to one signal if it is used as data input or write enable. By ensuring, that both instances of data input and both instances of write enable are equal and correct, the result will be a single clock enable signal. The correctness of this single signal cannot be assured without just shifting the problem. This last part from comparison to the flip-flop is still vulnerable to SEEs and though wrong bits being written cannot be completely avoided. The probability of undetected errors in this part of the design can be reduced with the length of the single signals, thus by placing the comparison of the redundant signals as close as possible to the targeting flip-flop.

By giving an own register bank to each pipeline, another form of redundancy is implemented. Assuming undetected single errors during a write to the register bank, at least one instance will hold the correct value. Any further access to the faulty register resulted in two differing values in the pipelines raising an error condition, inducing a pipeline flush and re-executing the instruction. However, this endless loop cannot be recovered with scrubbing, because the error is not based on a configuration upset. Detecting differences between both instances is not a problem at all, but how to react on differences? The first guess would be to re-execute the instruction if the writeback resulted in errors. Now imagine a command like this:

```
ADD R3, R3, R3      # R3 = R3 + R3
```

In order to re-execute this command, the previous value of R3 had to be saved. Extending this to spontaneous errors in registers that are not accessed in the current instruction, a complete second instance of the register bank with a lot of overhead controlling the snapshot/restore procedure would be required. A single general purpose registers bank consists of 31 registers with 32 bit each, thus just below 1000 registers and there are also some co-processor registers. The whole fault tolerant CPU without register bank uses around 2.500 flip-flops. Three or even four instances of a register bank do not get along with a low area approach. Further, if differences between the register banks occurred anyway, the CPU had to be reseted.

The second approach uses one shared register bank for both pipelines. As described above, the data and control signals from both pipelines are used to create a single data and clock enable as close as possible to the targeting register. There is another comparison logic before each flip-flop comparing only the single bits from both pipelines for this flip-flop. If these signals differ and the difference was not yet detected by the two step pipeline comparison, the bit is not written. One of the two data bits to each flip-flop has to be chosen as data input. The inputs from the first pipeline are used here. Reading is done independently for both pipelines. This implementation tolerates, that single bits within a word may not be written even if no error was detected by comparing the pipeline stages. This is a lack of hardware based fault tolerance but gives a great chance to error mitigation techniques on software level.

The demands for a low area implementation would prefer one shared register bank for both pipelines, if the radiation tolerance does not suffer from that. Both versions have been implemented and tested on their SEU susceptibility during this work.

5.6.6 Securing the Wishbone Bus with Hamming Codes

The Wishbone bus is a common source for both pipelines. If wrong instructions were fetched via this bus, both pipelines would be fed with the same faulty data and the error was not detectable by comparison. In order to avoid this situation, the Wishbone bus has to be secured. The Wishbone specification [Her02] allows the use of *tag-fields* for both the address and the data lines. Their contents can be defined by the user. Chapter 4.4 showed, that Hamming codes offer the possibility to detect several errors and therefore give a good candidate to be used as *tags*.

An own implementation of both Hamming encoder and decoder has been done during this work. The Hamming encoder to generate the check bits for a 32 bit input word could be implemented using 44 look-up tables and the decoder to correct single errors in a 32 bit word took 104 LUTs. By recalling, that storing these 32+7 bits in flip-flops needs 20 slices, whereas the decoder needs over 50 slices and by remembering, that the functionality of these slices cannot be guaranteed in a radiative environment, the advantage of error correction gets questionable. As both encoder and decoder are not radiation hardened in the described implementation, their usage in internal logic will make the design more susceptible to SEEs as it would be without these methods.

Due to these facts, only error detection mechanisms and no error correction logic have been used in this work. Both, the data lines and the address lines are secured with seven Hamming bits each. These bits are mapped into the tag fields and are able to detect up to two errors in a 32 bit word. The error detection of data read from the bus is done for each pipeline separately. If the combination of data and tag is faulty, both pipelines will detect an error. If one of the Hamming encoders fails, the result will be detected as difference between equivalent pipeline stages. Feeding both pipelines from the same bus is therefore no longer a single point of failure. Writing on the bus is only done, if both pipelines give identical signals. Otherwise, an error condition is raised. Again, the values from the first pipeline are actually used as output.

If the addressed peripherals receive faulty data or address values, they may respond to the CPU with an *ERR* signal instead of acknowledging the transfer. Receiving an *ERR* has the same effect in the CPU as detecting a difference between pipeline stages: the pipeline is flushed and restarted with the instruction in MW-stage. This functionality is the only fault tolerance technique applied to the peripherals. Further error mitigation strategies for the peripherals have not been evaluated in this work.

6 Implementation of the Fault Tolerant MIPS CPU

This chapter gives a detailed overview of the fault tolerant softcore CPU including the concrete behavior and its reactions on detected errors. Both, the functionality provided by the MIPS architecture and the modifications to increase fault tolerance are described here. Apart from the description of fault tolerance techniques for each module, this chapter gives assistance to add new peripherals to the system or write custom software. A non-hardened version of the CPU is presented to create a reference for the fault tolerant CPU regarding SEU susceptibility, resource usage and current consumption.

6.1 Target Devices and Tool Flow

The target device for the whole fault tolerant system is a *Syscore 1* board developed at Kirchhoff Institute for Physics in Heidelberg. The heart of this board is a Xilinx Virtex-4 FX20 FPGA (XC4VFX20-11FF672). The Virtex-4 FPGAs are manufactured in ten layer triple-oxide 90nm CMOS technology and are assembled in flip-chip geometry. The FX20 version of this device generation offers

- around 17.000 four-input LUTs and flip-flops
- 68 BRAM blocks with 18 kbit each, though around 1.200 kbit internal memory
- four digital clock managers (DCM)
- 32 digital signal processor (DSP) blocks
- one PowerPC 405 Hardcore CPU
- two Ethernet MACs
- eight multi gigabit tranceiver (MGT) blocks

The Xilinx FPGA is connected to DDR-SDRAM, Ethernet PHY, UART, USB controller, SD-Card slot and several other general purpose input and output ports as shown in figure 6.1. The DDR-SDRAM chips assembled on the Syscore 1 board are Micron MT46V64M16 -6TA chips. Two of these chips are arranged in parallel to get the double data width of 64 bit and an overall memory of 256 MByte. Beside the Virtex-4 FPGA, there is a small Actel ProASIC3 A3P125 flash FPGA. The Actel FPGA is manufactured in seven layer 130nm

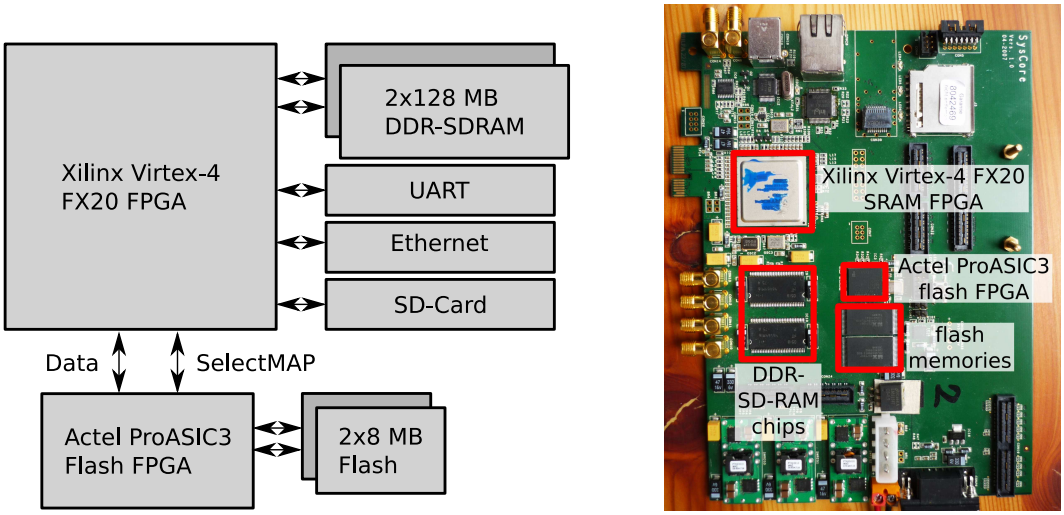


Figure 6.1: Sketch and photo of the Syscore 1 board

CMOS technology and offers around 3.000 tiles. This FPGA has access to two 8 MByte flash memory chips and is amongst others connected to the Virtex' SelectMAP interface. It is though able to configure and reconfigure the Xilinx FPGA.

The user logic on both FPGAs is described with the hardware description languages VHDL or Verilog. The Xilinx FPGA logic is synthesized with the Xilinx *XST* compiler. Implementation is done using Xilinx *Place and Route* (PAR) tools. Xilinx *bitgen* is used to create the FPGA configuration bitfile. All of these tools can be controlled and configured using Xilinx *ISE* as graphical user interface (GUI). Xilinx *PlanAhead* allows graphical floor-planning and *FPGA-Editor* shows the actual implementation of a placed and routed design. The configuration bitfiles are uploaded into the FPGA using Xilinx *Impact* with either a graphical interface or as command line tool. All of these tools are used as they come in *Xilinx ISE Design Suite* 9.1 and 10.1.

The Actel ProASIC3 flash FPGA comes with an own design flow. The HDL code is written in Verilog and can be synthesized with Synplify. *Place and Route* and the creation of the programming file is done with *Actel Designer Software*. The Actel Libero GUI gives a graphical front end to these tools. Uploading the programming file into the flash FPGA is done with an Actel *FlashPro3* programming cable and software.

6.2 The Actel Flash FPGA

The main purpose of the Actel flash FPGA is to perform the continuous configuration writing (scrubbing) on the Xilinx FPGA. As this FPGA is based on a flash architecture, it is mostly immune against radiation induced configuration upsets and only needs to

address SETs and user logic SEUs. The Xilinx bitfiles to be written are stored in the attached flash memories. The bitfiles for the Actel FPGA have hardly been touched for this thesis and are taken from a previous work described in [Roh08]. This work delivers a tool chain for writing bitfiles into the flash memories and includes a design for the Actel FPGA to perform the scrubbing operation on the Xilinx FPGA. The Actel FPGA design includes a flash controller for reading and writing the flash memories and a state machine that writes the Xilinx bitfiles from the flash memory to the Xilinx SelectMAP interface. The current operation of the Actel FPGA is determined with two jumpers.

Unfortunately, the Actel FPGA does not have a ready-to-use UART, Ethernet or USB connection in order to write Xilinx bitfiles into the flash memories. On the other hand, the only possibility to write the flash memories is to use the Actel FPGA. A sketch of this architecture is shown in figure 6.1. The tool chain for writing the bitfiles into the flash memories therefore requires both FPGAs. A UART connection with a PC is established with the Xilinx FPGA, running a PowerPC system with the standard Xilinx IP-Cores and a custom peripheral for driving the data lines between the Xilinx and the Actel FPGA. The data to be written is forwarded to the Actel FPGA and finally written into the flash memories by the flash controller running on the Actel FPGA. This implementation in both FPGAs does not use any fault tolerance techniques, yet. Changing the contents of the flash memories in a radiative environment is not reliably possible at this time.

There are some minor modifications done on the Actel design and the writing tool chain during this work. A signal has been added in the Actel design, that sends a single pulse from the internal 40MHz clock to the Xilinx FPGA via one of the data lines. This pulse is sent each time the scrubbing state machine in the Actel FPGA has completed one scrubbing cycle. The addition of this signal allows the CPU on the Xilinx FPGA to check whether a configuration refresh could correct a single event upset or not. The modifications in the writing tool chain consist of porting of the PC UART interface software from Windows to Linux. Both, the new Actel bitfile and the Linux UART software interface can be found on the CD attached to this thesis.

The scrubbing state machine currently implements only "blind" scrubbing. The partial bitfile is written continuously into the Xilinx SRAM FPGA, but the current configuration is neither read back, nor is the configuration interface checked for SEU-induced malfunctions. Unfortunately, the scrubbing is performed relatively slow at this time. The state machine requires more than a second to perform a single write cycle. Theoretically, writing the Xilinx bitstream via a 33 MHz eight bit bus should be possible in less than 100 ms. The limiting factors in the current design have not been investigated yet. The Actel design currently does not contain any fault tolerance techniques. Due to the flash architecture, the probability for SEUs in the configuration memory is very low, but SEUs and SETs in the user logic can still occur like in any SRAM device. Future versions of the Actel design should consider SET mitigation techniques and redundant modules. A new implementation of the whole Actel design "from scratch" is currently in development.

6.3 Hierarchy of the SRAM FPGA Design

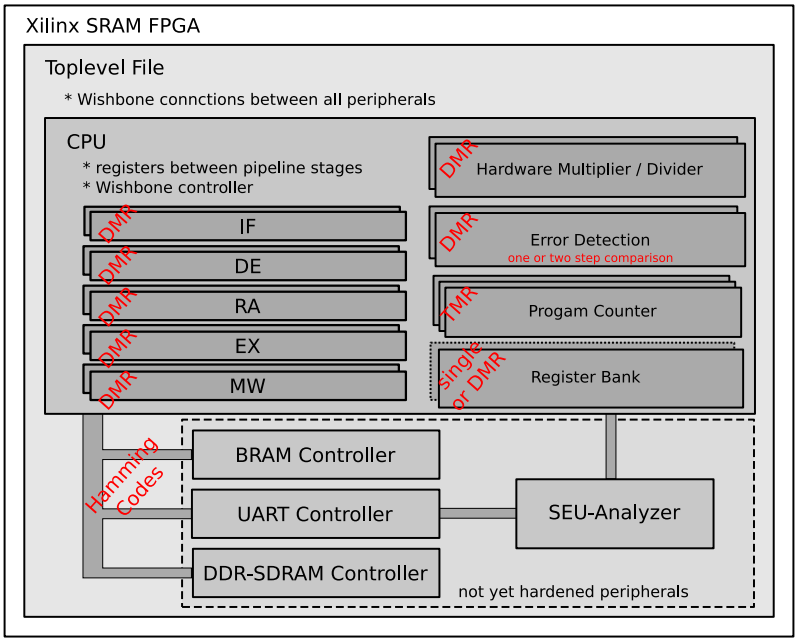


Figure 6.2: Hierarchy of the SRAM FPGA design. The applied error mitigation techniques are attached in red.

The complete hierarchy of the implemented softcore CPU system is shown in figure 6.2. All elements are plotted according to their hierarchy in the HDL implementation and are further described in this chapter. The toplevel file instantiates the CPU, a BRAM controller, a UART controller and a DDR-SDRAM controller, all connected with a Wishbone bus. A DCM as clock generator delivers the clocks required for the bus, the CPU and any peripherals. SEU-analyzer is a module built to report detected errors and the current status of the CPU via UART. Any pipeline stage within the CPU, the hardware multiplier/divider modules and the error detection mechanisms are implemented with DMR. Error detection is implemented in both versions, with one or two step comparison. Both instances of the pipelines use either one common register bank or are supplied with one register bank for each instance. Both pipelines are fed from a triplicated program counter. The Wishbone bus uses Hamming codes to detect bus errors. This work's SEE mitigation techniques are addressed to the softcore CPU only, so there have no mitigation techniques been applied on the peripherals during this work.

The whole SRAM FPGA design is written with VHDL using a modular approach. In order to keep the code clear, the component declarations, the type definitions and some basic system settings like endianess, interrupt handler address, processor ID or UART baud rate are implemented using VHDL packages.

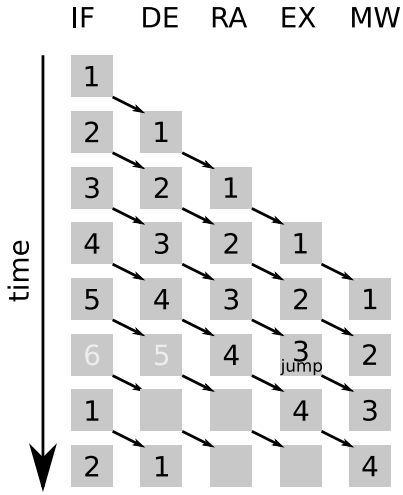


Figure 6.3: The pipelining concept. Any pipeline stage handles one instruction. The results from every pipeline stage is synchronously shifted to the next stage. Instruction 3 exemplary fulfills a jump condition in EX, so the instructions in IF and DE are invalid and needlessly fetched. Instruction 4 is executed as branch delay slot. The pipeline continues with the new address (here: 1).

6.4 Implementation of the Fault Tolerant CPU

The following sections will describe both the functionality of the fault tolerant CPU and how this is implemented in this work. The basic functionality of a CPU in general will only roughly be described here, so the reader may be redirected to [HP96]. The original plasma project implements a large subset of the MIPS I instruction set. This has been kept for this work and extended with the "return from exception" (RFE) instruction. The complete implemented instruction set is shown in appendix A. An architecture implementing this MIPS I instruction set could be found in the MIPS R2000/R3000. The fault tolerant CPU is leaned on this architecture regarding pipeline organization and coprocessor registers, but does not implement caches, memory management unit (MMU) or virtual addressing / translation lookaside buffer (TLB). A detailed description of the MIPS R2000/R3000 architecture can be found in [KH91].

6.4.1 The Pipelining Concept

The whole CPU is implemented using a five stage pipeline with *instruction fetch* (IF) *instruction decode* (DE), *register access* (RA), *execute* (EX) and *memory/writeback* (MW). Dealing with a single instruction from IF to MW includes a lot of logical steps. These steps can be handled in a single clock cycle using a slow clock or in several cycles using a fast clock. The idea of pipelining is to parallelize the handling of sequential instructions using a fast clock. This concept is shown in figure 6.3. In the first step, instruction 1 is fetched. While instruction 1 is decoded, the next instruction (2) can already be fetched. If both of these completed their stage, the whole pipeline content is shifted to its next stage by fetching instruction 3, decoding instruction 2 and performing register access for instruction 1. Assuming a single instruction needs n clock cycles for each stage, it takes $5n$ clock cycles for

the completion of a single instruction. As the pipeline handles five instructions in parallel, the CPU completes one instruction every n cycles in the optimal case. The benefit of pipelining is obvious: parallelization increases the throughput without significantly changing the latency.

But there are problems with pipelining, too. Imagine instruction 1 writes its result to a register and instruction 2 needs this value to calculate its own result: the writing of instruction 1 is performed in MW, whereas instruction 2 wants to read this value in RA. While instruction 2 is in RA, instruction 1 has not yet reached MW but is still calculating the result. There are two solutions to this problem: Instruction 2 has to wait for instruction 1 completing MW by negating the advantages of pipelining or instruction 1 makes its result available to instruction 2 before it is written back in MW. The second solution is applied here: By using bypass lines, the result of one instruction can be forwarded to one of its followers, so the pipeline does not have to be stalled until a single instruction has completed. This bypassing is done automatically if needed.

A program counter (PC) stores the addresses of the currently executing instructions. This program counter is stored for each instruction in each pipeline stage. The PC in the IF-stage is used as address to fetch the next instruction and is incremented automatically, unless there is no jump to a specific PC-address.

The worst case for pipelining is a code with a lot of jumps. In most cases, the decision whether a jump has to be done or not is not available until EX-stage. By reaching EX, the following two instructions are already in the pipeline and a third is currently fetched. If the jump has to be done, at least two of these following instructions have been fetched needlessly and have to be marked as invalid in the pipeline. The pipeline now continues with the calculated jump address. The more jumps in the software, the less effective is the pipelining concept. The MIPS architecture offers the benefit that for the common conditional and unconditional jump commands, the instruction following the jump is executed, too. This is called *branch delay slot* and lowers the cost of jumps in a pipelined architecture a little bit. The branch delay slot is not executed on exceptions.

6.4.2 Description of the Pipeline Stages

Instruction Fetch Stage (IF)

The IF-stage is the only stage with no own entity definition. Most of the functionality of the IF-stage is placed within the EX-stage, because the program counter for the next instruction to be fetched relies on whether a jump is done or not. Without a jump, the program counter for the IF-stage is incremented by one for each instruction. If a jump condition is fulfilled, the new program counter is calculated within the EX-stage module.

The correct program counter is used as Wishbone bus address to fetch the next instruction. The fetched instruction is stored in the *opcode0* and *opcode1* registers for each pipeline respectively.

Decode Stage (DE)

The DE-stage decodes the instruction currently stored in the *opcode0/1* registers. This entity is mostly taken from the original Plasma project [Rho09]. According to the bits in the opcode, this entity generates the select signals for the registers to be loaded in RA-stage, the select signals for the logic combination of them in EX-stage and the select signals for the result to be written back in MW-stage. Parts of the select signals are implemented using VHDL type definitions to keep the code readable. Further, there is a first detection, whether bypass lines are required to load the needed values from currently executing instructions.

Register Access Stage (RA)

The RA-stage fetches the requested register values by applying the according registers addresses at the register bank. The read values are redirected to the EX-stage. Again, the signals of the preceding instructions are checked, to see if the read values are out of date and bypassing values is required. The select signals concerning EX-stage or MW-stage are just forwarded.

Execute Stage (EX)

The EX-stage actually performs the requested logic operation. Two operands are selected according to the supplied select signals out of the register values fetched in RA, the current PC address or the signed or unsigned lower part of the instruction word. These operands are combined in the requested logical operation. In case of signed addition or subtraction, it is checked whether the result exceeded the 32 bit number space (overflow exception). A further multiplexer selects the correct output for the following pipeline stage. The register values read in RA-stage are compared to determine possible branch conditions. This pipeline stage also holds the calculation for the IF PC address, because its determination is strongly related to what's happening in EX-stage. In cases of jumps/branches or exceptions, the PC calculated from the EX-stage values is used, in any other cases, the current PC is incremented by one. The EX-stage also redirects the input values for multiplication or division to the according modules.

Memory/Writeback Stage (MW)

The last pipeline stage prepares the calculated values to be written into the requested destination register or to any peripherals via the Wishbone bus. There are several possibilities how data can be written to peripherals. The instruction set supports the write of both halfwords (16 bit) and bytes (8 bit) additionally to the option of writing full 32 bit words. If halfwords or bytes are written, the data has to be aligned within the 32 bit Wishbone bus data according to the configured *endianess* (big or little endian). The write enable and select signal for the Wishbone bus are created here, too. In case of halfword or byte bus reads, the incoming data can be interpreted as signed or unsigned requiring the data to be sign extended before it is written to a 32 bit register. If data has to be written to one of the registers in the register bank, MW-stage creates the write_enable signals for the according registers.

6.4.3 Hardware Multiplier and Divider

Both multiplication and division is implemented in hardware for signed and unsigned 32 bit words. Multiplication is realized with DSP-Blocks, the multiplication module is generated with Xilinx Core Generator. In order to realize both signed and unsigned multiplication, the module has been created as 33x33 bit signed multiplier and the highest bits are set according to the requested operation and the sign of the input vectors. Multiplication is currently done in six clock cycles. Division is done using the same algorithm as provided with the plasma project:

```
long upper=a, lower=0;
a = b << 31;
for(i = 0; i < 32; ++i)
{
 lower = lower << 1;
 if(upper >= a && a && b < 2)
 {
   upper = upper - a;
   lower |= 1;
  }
 a = ((b&2) << 30) | (a >> 1);
 b = b >> 1;
}
```

a and *b* are dividend and divisor, the values of *upper* and *lower* are the requested quotient and remainder. The implementation of this algorithm has been rewritten and partially

```
type t_de_out_ra_in is record
  valid : std_logic;
  rfe : std_logic;
  opcode : std_logic_vector(25 downto 0);
  rs_index : std_logic_vector(5 downto 0);
  rt_index : std_logic_vector(4 downto 0);
  rd_index : std_logic_vector(5 downto 0);
  alu_func : t_alu_func;
  shift_func : t_shift_func;
  sign : t_sign;
  mult_func : t_mult_func;
  branch_func : t_branch_func;
  bypass : t_de_bypass;
  a_source : t_a_source;
  b_source : t_b_source;
  c_source : t_c_source;
  mem_source : t_mem_source;
  mem_mode : t_mem_mode;
  pc_source : t_pc_source;
end record;
```

```
signal pl0_de_out: t_de_out_ra_in;
signal pl0_ra_in : t_de_out_ra_in;
[...]
pl0_seq : process(clk)
begin
  if rising_edge(clk) then
    if rst='1' then
      [...]
      pl0_ra_in <= reset_ra_in;
    else
      [...]
      -- register to RA-stage
      pl0_ra_in <= pl0_de_out;
      [...]
    end if;
  end if;
end process;
```

Figure 6.4: Implementation of pipeline data signals with records. The left column shown the record definition. Registers for all elements within the record are inferred in the right column with one line of VHDL code.

pipelined for the use with clock frequencies of around 100 MHz. The division is done in 33 clock cycles.

Both multiplier and divider have been implemented twice, so every pipeline uses its own instances.

6.4.4 HDL Dual Pipeline Implementation

The pipeline has been implemented using a modular approach. The logic of each pipeline stage is implemented as a combinatorial process placed in an own entity. There are registers between two subsequent combinatorial pipeline stages. The inputs, outputs and the data transfer between two successive pipeline stages are mainly realized with records. The records allow to easily put registers between all outputs of one entity and the inputs to the next entity. This is independent of the number of signals and keeps the code clear. The example in figure 6.4 shows the record and the register generation between DE-stage and RA-stage. All outputs of DE-stage are inputs for RA-stage. By grouping them with a record (left), the flip-flops for any signals within this record can be generated with one line of HDL code (right).

Any pipeline stage is instantiated twice forming an identical second pipeline instance. The data lines between adjacent pipeline stages in one pipeline do not have an effect on the second pipeline and both instances have their own set of pipeline control signals.

However, the registers between adjacent stages share a common clock enable for both pipelines. All of these pipeline registers are implemented with one VHDL process and their common clock enable is derived from both pipeline's control signals.

6.4.5 Register Bank Organization

The register bank consists of 32 general purpose registers (GPRs) with 32 bit each. The first register, R0, is read-only and always contains zero. The other registers are read and writable and are used for different purposes from the compiler. An overview of the GPRs is shown in table 6.1. A typical instruction loads the values from two of these registers, combines them with a logical operation and writes the result back to one of these registers. An example is:

```
ADD R3, R1, R2      # R3 = R1 + R2
```

Register	Name	Function
R0	zero	read-only, always contains zero
R1	at	Assembler temporary
R2-R3	v0-v1	Function return values
R4-R7	a0-a3	Function parameters
R8-R15	t0-t7	Function temporary values
R16-R23	s0-s7	Saved registers across function calls
R24-R25	t8-t9	Function temporary values
R26-R27	k0-k1	Reserved for exception handler
R28	gp	Global pointer
R29	sp	Stack pointer
R30	s8	Saved register across function calls
R31	ra	Return address from function call

Table 6.1: General purpose registers. This information is taken from [Rho09].

Additionally to the general purpose registers, there are some special registers accessible. The registers HI and LO are used to store the results from multiplication and division. These registers can be accessed via the MFHI, MFLO, MTHI and MTLO instructions (see appendix A). In case of multiplication, the result of multiplying two 32 bit values may be up to 64 bits wide. The HI register stores the upper 32 bit of this result and the LO register stores the lower 32 bit. In case of division, both the quotient and the remainder are calculated. The quotient is written to LO and the remainder to HI.

Another set of special registers are the co-processor 0 (CP0) registers. They are accessible via the MTC0 and MFC0 instructions (see appendix A) and are used for interrupt and exception handling and control. Table 6.2 shows the implemented registers.

Register	Name
CP0-R12	Status register
CP0-R13	Cause register
CP0-R14	Exception Program Counter (EPC)
CP0-R15	Processor Revision Identifier (PRId)

Table 6.2: Implemented co-processor 0 registers

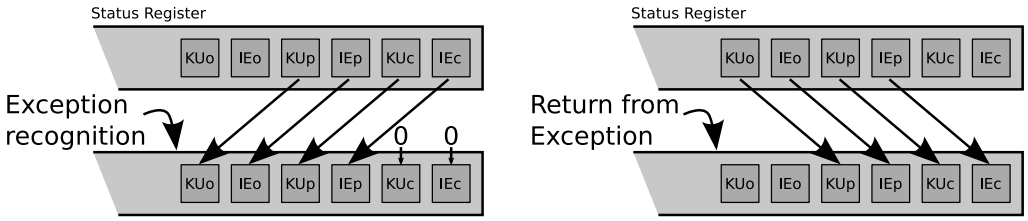


Figure 6.5: Status register on exceptions. All flags are shifted left on exceptions while disabling interrupts and kernel mode. The RFE instructions restores the previous state

Status Register

The status register regulates if interrupts are allowed, which interrupts are allowed and whether the CPU is currently in kernel or user mode. The status register format is shown in table 6.3. The Co-processor Usable (CU) flag controls the usability of the four possi-

Bits	31:28	27:16	15:8	7:6	5	4	3	2	1	0
Content	CU	0	IM	0	KUo	IEo	KUp	IEp	KUc	IEc

Table 6.3: Status register format

ble co-processor units and is statically implemented to "0001". The Interrupt Mask (IM) controls, which interrupts are enabled. There are eight interrupt lines, six external lines and two software interrupts. The interrupt mask allows to enable only a subset of all interrupts. Bits 15:10 are mapped to the external interrupts and bits 9:8 to the software interrupts. The remaining bits are Interrupt Enable (IE) and Kernel/User mode (KU) bits in three instances: old (IEo, KUo), previous (IEp, KUp) and current (IEc and KUc). Only the *current* values are writable. The current Interrupt Enable (IEc) dominates all settings in the interrupt mask. If IEc is zero, interrupts are completely deactivated regardless of the settings in IM. If IEc is one, the IM settings apply. In case of an exception, the *old* values are overwritten with the *previous* values, the *current* values are written into the *previous* fields and the new *current* values are set to zero. This deactivates all interrupts and allows exception handling. After this is done, the *Return From Exception* (RFE) instruction reverses the shift: the *previous* values are copied to the *current*, and the *old* values are copied to the *previous* fields. A graphical representation of these shifts is shown in figure 6.5.

Cause Register

The cause register describes the cause of the last exception and is organized as shown in table 6.4. The Branch Delay (BD) flag indicates, whether the last exception occurred in a

Bits	31	30	29:28	27:16	15:8	7	6:2	1:0
Content	BD	0	CE	0	IP	0	ExcCode	0

Table 6.4: Cause register format

branch delay slot. The return address had to be adjusted in case of a BD-slot. The Co-processor Enable (CE) flag could indicate the number of the co-processor, if a *co-processor unusable* exception was raised. This functionality is not implemented, so these fields are statically zero. The Interrupt Pending (IP) field indicates, which of the eight possible lines raised an interrupt in case of an interrupt exception. IP(7:2) map to the external interrupts and IP(1:0) map to the software interrupts. These IP(1:0) are writable for the users. Writing a '1' in there raises an interrupt condition, if the according IM and IEc are set in the status register. The Exception Code (ExcCode) field defines the type of exception. The implemented exceptions are shown in table 6.5. Both, the BD-flag and the

ExcCode	Description
0	Interrupt
8	SYSCALL exception
9	Breakpoint exception
12	Arithmetic overflow exception

Table 6.5: Implemented exceptions and their exception codes

ExcCode are written automatically each time an exception occurs. By reading this cause register during the exception handling, the CPU can determine the cause of the exception and can react on it accordingly.

Exception Program Counter (EPC)

The EPC is a 32 bit read-only register containing the address of the interrupted instruction due to any exception. The value of this register is used to resume the program after an exception has been handled. This register value is automatically written each time an exception occurs.

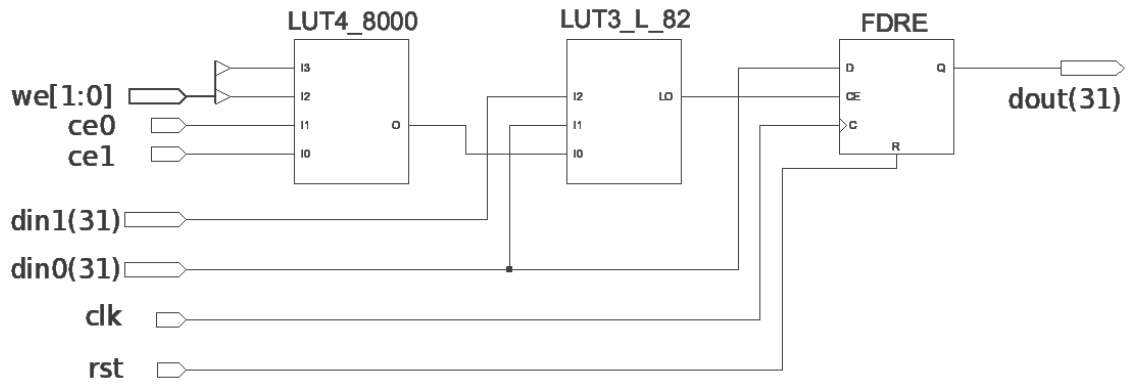


Figure 6.6: Technology schematic of a single register in the common register bank for both pipelines. Screenshot from Xilinx Technology Viewer, modified for readability.

Processor Revision Identifier (PRId)

The PRId is a 32 bit read-only register. It contains information about the implementation and revision level of the CPU. As the implemented architecture is similar to the MIPS R3000 CPU, the according value 0x00000200 has been implemented. This value is taken from [KH91] and can be changed in the system configuration package.

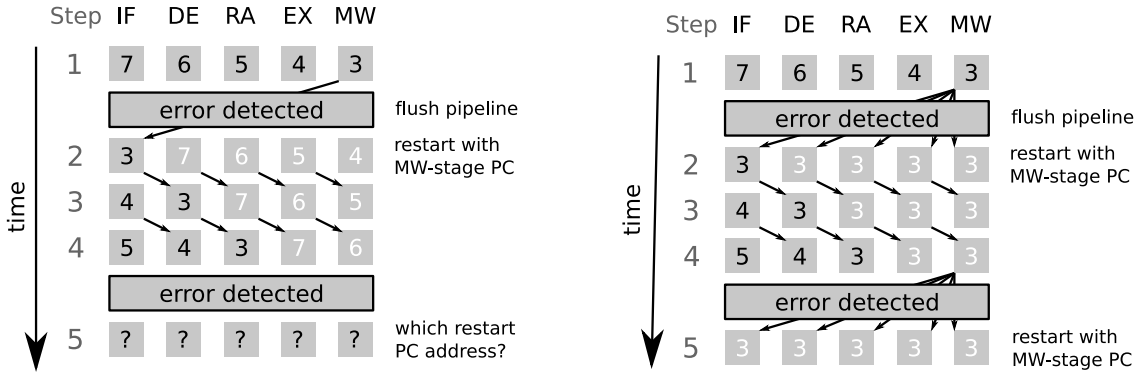
6.4.6 Register Bank Implementation

As described in the chapters above, the register bank is implemented in two versions: one common register bank for both pipelines or one register bank for each pipeline. The dual register bank implementation does not include specific error detection mechanisms. Each register bank "blindly" writes what it gets from its pipeline, so no special implementation is chosen in this case.

If both pipelines share a common register bank, the signals from both pipelines have to be evaluated to set correct values. Each MW-stage returns a select signal, defining which register shall be written. A entity called *get_write_enable* returns a write_enable signal according to the pipeline control signals, the identity of both MW-stages and the result from the first pipeline comparison step. This entity is implemented twice, too. Any register in the common register bank is therefore fed with:

- the data signals from both pipelines: din0, din1
- the write_enable signal from both *get_write_enable* entities: we[1:0]
- the select signal for each register from both pipelines: ce0, ce1

As described in chapter 5.6.5, the double redundancy is kept as long as possible, in order to rather do not perform the write than to write a wrong value. The implementation



(a) Without copying the PC addresses

(b) With copying the PC addresses

Figure 6.7: PC behavior on errors. Without copying the PC addresses to any pipeline stages on errors, a reliable restart address could not be determined (left). The right picture shows the implemented behavior

is done with the instantiation of single look-up tables and flip-flops as shown for one flip-flop in figure 6.6. The write_enable vector and the two select lines are checked to be all equal to '1' in the first look-up table. The second LUT uses this signal as input and includes both data inputs. Only if both data lines are equal and the result from the first LUT is '1', a clock_enable signal is sent to the flip-flop, sampling din0 at the next rising clock edge. Both LUTs and the flip-flop have been instantiated with their primitives "by hand" and are replicated 32 times with a for-loop. This entity is used for any register in the register bank except for the cause and status registers.

6.4.7 Program Counter Implementation

As described in the chapters above, the program counter is the only part of the CPU that is implemented with triple modular redundancy. Three instances of the stored program counter addresses for the instructions in all pipeline stages are working in conjunction. All three instances use the signals from both pipelines and voted values of their own outputs as inputs. The majority voting of signals, vectors and records is implemented with VHDL functions. The output of current PC addresses to any of the pipeline stages is done from all three instances. Any pipeline stage uses its own voters to reduce the triple signals to singles.

In case of an error detected by the CPU, the program counter address of the instruction currently in MW-stage is copied to any pipeline stages. In case of a regular jump, the address of the jump instruction is used for all stages from DE to MW. This feature allows to use the MW-stage PC address as restart address in any cases of an error, without having to consider, which of the instructions currently in the pipeline are valid or have not yet been executed.

An example showing the purpose of this method is shown in figure 6.7. One of the stages suddenly gives a wrong result due to an SEU and the error gets detected. If the PC addresses remained unchanged, the restart point could not be determined reliably. The first detected error can be handled well by restarting with the MW-instruction. The problem points out, if a second error is detected during the pipeline refill. If the MW-instruction (instruction 6) would be chosen in this second case, the CPU would omit the instructions 3-5. The two step comparison technique does not allow to detect, which of the early pipeline stages failed and therefore can not determine the correct restart point if the MW-PC address is not copied to any stage on errors. Determining the correct restart address without copying the MW-stage PC would require additional logic. It had to be recorded, which instructions have already been executed and which instructions are not valid or may no be executed at all due to regular jumps.

In every instance of the program counter, there is a process checking whether the PC address in MW-stage changed during the last clock cycles. In combination with the signal from the Actel FPGA, indicating when a scrubbing cycle has completed, this can be used to detect if scrubbing could correct an error. If the PC address in MW-stage is stuck for several scrubbing cycles, the CPU state is uncorrectable with scrubbing and needs to be reseted. The TMR-voted signal of this condition is used to reset the CPU automatically. This reset currently affects only the CPU and has no impact on the other peripherals.

6.4.8 Error Detection and Error Handling

As described above, error detection is done in two stages. A first step compares the inputs to the pipeline stages DE, RA and EX with their equivalents from the second pipeline. A further input to the first comparison step are the registered values of the pipeline control signals from both pipelines. An error condition is raised, if equivalent pipeline stages differ or if equivalent pipeline control signals differ. Differences in the pipeline stages do only contribute, if at least one of both instances is marked as valid. Pipeline data marked as invalid in both instances cannot raise an error condition. The result of this comparison is registered. This first comparison stage is implemented twice, too, so there are two instances of the comparison result. The result of this comparison is not time critical, because none of the early pipeline stages can write to memory. It has just to be ensured, that the instruction raising the error condition does not complete MW-stage before the error is handled. This will never be the case with just one register step. Even if an error was raised from EX-stage and this instruction is shifted to MW-stage in the following clock cycle, it will not have completed MW-stage before the error is detected.

The second comparison step is done asynchronously, its result is required in the same clock cycle to prevent faulty data being written. This second step compares all outputs of the MW-stages, all data being prepared to be put on bus and the results for the next program counter address. Furthermore, it continuously checks for Hamming code errors

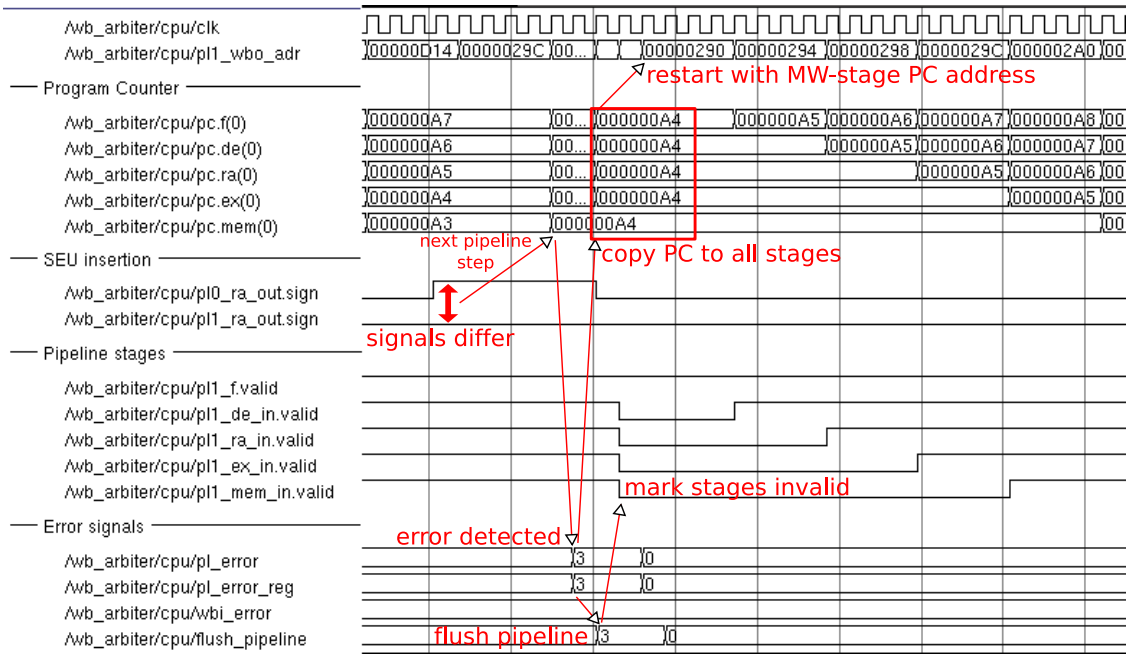
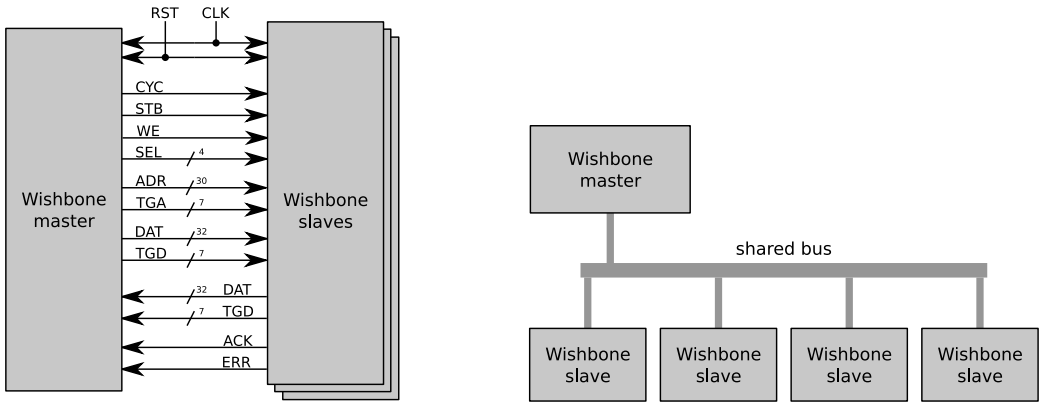


Figure 6.8: Behavior on errors shown in a Modelsim simulation.

in the data received from the bus and takes the result from the first comparison step into account. As for the first step, both, the comparison and the Hamming code checker are implemented twice and both results from the first steps are fed into each of this second step comparators. So, this also results in two independent error signals. An error is detected, if one of those two signals differs from zero.

A second implementation has been done by solely using the second comparison step, but including the pipeline control signals. The inputs to the early pipeline stages are not compared in this approach. This should save a lot of combinatorial logic, but will not detect errors in the early pipeline stages. The effects on the overall radiation tolerance of this approach are studied in chapter 8.

If an error is detected in the second comparison step, the instruction in MW-stage will not yet be completed. This error condition makes the program counter to copy the MW-PC address to all stages and induces the *flush_pipeline* signal. This signal marks both instances of DE, RA, EX and MW as invalid and the pipeline restarts with the instruction previously in MW-stage. In case of a transient fault, a second attempt to process the failed instruction will succeed and the CPU can continue its operation. In case of a static upset, a re-execution of the failed instruction will not correct the error. An error like this will be detected every time after the pipeline restarted and will flush it again. The CPU will stay in this loop until the error gets corrected by scrubbing. If the error is corrected, the pipeline will get correct values and this error condition is not raised again.



(a) Implemented Wishbone signals

(b) shared bus principle

Figure 6.9: Wishbone bus signals and shared bus principle

The detection of an error is exemplary shown in figure 6.8. The *sign* output of the RA-stage is used as SEE affected signal. The output from the second pipeline is correctly zero, whereas the output of the first pipeline is forced to a logic one for a short period of time. As *sign* is an output of one of the early pipeline stages, the difference is not detected immediately. After the whole pipeline content gets shifted one stage, the previous RA-stage output becomes an input to EX-stage. The inputs from this stage contribute to the first step comparison and the error gets detected one clock cycle later. The second stage handles this result combinatorially. The output from the first step comparison is *pl_error_reg* and the output from the second comparison step is *pl_error*. This second comparison step result induces that the PC address from MW-stage is copied to any stages and that the *flush_pipeline* is raised. Flushing the pipeline is performed by marking the DE, RA, EX and MW instructions as invalid. The PC from MW-stage is then used to fetch the failed instruction again. For clarity reasons, only one instance of the TMR program counter, only the valid-signals from one pipeline and only the differing pipeline signals are shown in the described figure.

6.5 Wishbone Bus

The Wishbone bus is meant as a public domain common interface between different IP-core components, its exact specification can be found in [Her02]. Wishbone defines a standard data exchange protocol between modules without regulating their IP-core functionality. This bus protocol is recommended by *opencores.org* to assist the connectivity between different IP-cores. The Wishbone bus is a master/slave architecture, where only masters can initiate a bus transaction and the slaves are only allowed to respond. The architecture is not limited to a single bus master, however bus arbitration has to be done if there are multiple masters.

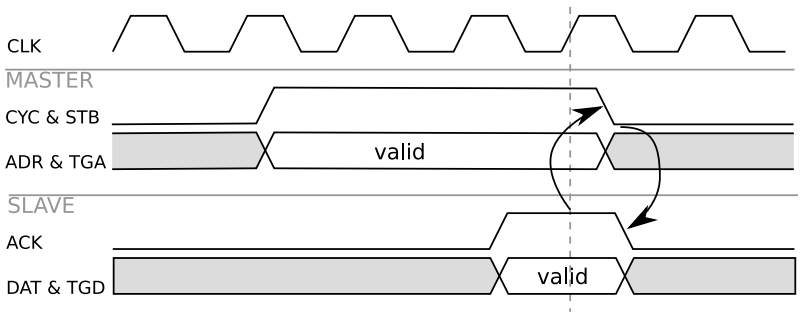


Figure 6.10: Wishbone read cycle. The master asserts CYC & STB in combination with valid ADR and TGA. WE and SEL are assumed to be zero. The slave responds with ACK and valid DAT & TGD. After the ACK was sampled with the following rising edge, the master de-asserts CYC & STB whereupon the slave releases its ACK signal.

In this implementation, the CPU is the only bus master and its peripherals are connected as bus slaves. The bus is used as single shared bus for all peripherals. The Wishbone signals and their widths as they are implemented in this work are shown in figure 6.9(a). The clock and reset lines are fed from a configurable digital clock manager (DCM) and are used for both, the bus, the CPU and the peripherals. A Wishbone transfer is initiated by the master setting CYC and STB to '1' in combination with a valid address (ADR+TGA) and a valid write enable (WE). In case of a write access, data (DAT+TGD) and select (SEL) have to be valid, too. The master waits for the slave to reply with any of the transfer terminating signals ACK or ERR. On read cycles, the slave's data output (DAT+TGD) has to be valid while ACK is '1'. The data is sampled at the first rising edge while ACK or ERR are '1' and the master releases CYC & STB. The slave can then pull the termination signal back to '0'. A sample read cycle is shown in figure 6.10.

CYC is meant to make connection to a slave and STB is meant to actually initiate a transfer. The current implementation of the CPU uses both signals synchronously, they have always the same value. There was no need to hold a connection to a slave without transferring data, so these signals could be used to implement redundancy in the bus. ADR is the address field and TGA is its tag field holding the Hamming bits for the current address. Accordingly, TGD is the tag field for the data lines (DAT) holding their Hamming bits. The select (SEL) line is used to address single bytes or halfwords within the data words for write cycles. It consists of four bit for a 32 bit data width. In combination with WE='1', valid values for SEL are "1111" to write the full word, ("0011", "1100") to write the lower or upper halfword or ("1000", "0100", "0010", "0001") to write one of the bytes within the word.

The CPU's output signals WE, SEL, DAT, TGD, ADR and TGA are statically connected to all peripherals. Only the control signals CYC and STB are multiplexed to only one slave at a time according to the current bus address. The slaves' output signals DAT, TGD, ACK and ERR are all multiplexed to the CPU's inputs according to the address, too.

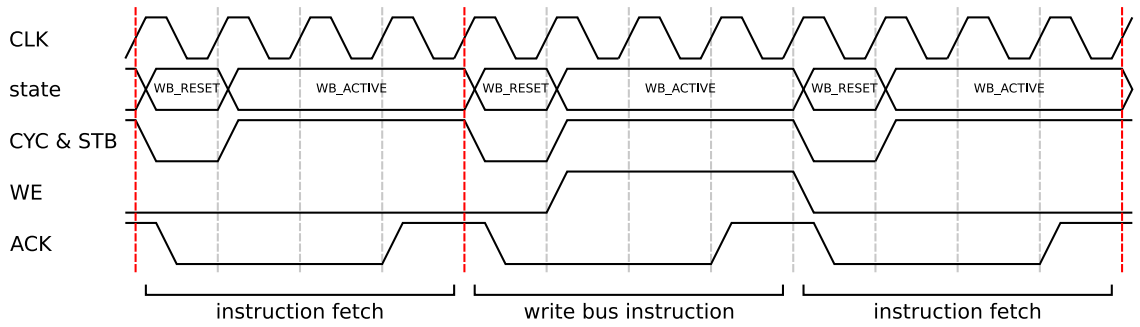


Figure 6.11: Behavior of the Wishbone state machine. The *WB_RESET* state keeps CYC & STB low for one cycle. The *WB_ACTIVE* state waits for the slave responding with a cycle termination signal. The pipeline is shifted one step at the red dotted lines. The second bus cycle shows the situation when MW and IF have to share the same bus.

6.5.1 The Wishbone State Machine

The Wishbone master within the CPU is implemented as a small state machine with states *WB_RESET* and *WB_ACTIVE*. *WB_RESET* forces both CYC and STB to zero, whereas they are high in *WB_ACTIVE* and the CPU waits for one of the cycle termination signals ACK or ERR. This behavior is shown in figure 6.11. The data and address lines are not shown in this figure, but are assumed to be valid. The red lines indicate the points where each instruction is shifted to the next pipeline stage. Exemplary shown is a regular register accessing instruction in the first bus cycle. IF fetches the next instruction, while MW only uses the internal register bank and does not need to access the bus. The second bus cycle shows the situation when both IF and MW have to access the bus. IF has to fetch the next instruction, whereas MW in this example wants to write a value to a peripheral via bus. As both stages have to share the same bus, the request has to be handled sequentially. In the first phase of the cycle, MW-stage performs its writing to the bus. In the second phase, IF fetches the next instruction. The sequential handling of both stages requesting bus access is controlled by the signals *pause_pipeline* and *pause_pipeline_reg*. This situation can be accelerated by using dedicated instruction and data caches.

Bus Access in Combination with Multi-Cycle Instructions

With most of the instructions, the bus access will be the speed limiting part. The majority of the instructions will complete their current pipeline stage within one clock cycle. A bus access currently takes at least four clock cycles. However, there are some multi-cycle instructions like MULT, MULTU, DIV and DIVU. They may require more clock cycles than a bus access, so the CPU waits for them to complete before it proceeds to the next instruction and therefore the next bus access. While the CPU is waiting for multi-cycle instructions to complete, the Wishbone state machine will remain in *WB_ACTIVE*, so

CYC and STB will stay high regardless of an ACK or ERR signal. The peripherals have to be able to keep their cycle termination signals high as long as CYC & STB stay active.

Skipping needless Instruction Fetches on Jumps

In case of branch/jump instructions, the jump condition is evaluated in EX-stage. The following two instruction are already handled by DE and RA and the third following instruction is currently fetched (IF). In case of a regular jump, the instruction in RA is executed as branch delay slot, but the instructions in DE and IF have to marked as invalid. The jump condition is available within the first clock cycle of the according instruction in EX-stage. The Wishbone bus is meanwhile in *WB_RESET*, so this bus cycle can still be canceled to mitigate the costs of the jump in this pipelined architecture. The number of needlessly fetched instructions on jump conditions can therefore be reduced to the one instruction in DE-stage. The implementation of this bus cycle canceling is done with the *wb_skip* signal.

6.5.2 Fault Tolerance Aspects of the Wishbone Bus

As mentioned above, the Wishbone bus uses seven bit Hamming codes to protect both, the address and the data fields. Both pipelines calculate their values for DAT, TGD, ADR and TGA independently. These values are compared and checked for identity before the values from the first pipeline are put on bus. New values are only put on bus, if the Wishbone state machine is in *WB_RESET*, so these values should not change during a bus access. The Hamming encoding for a 32 bit word is done with a custom built VHDL function called *hamming_encode(std_logic_vector(31 downto 0))*. This function returns the Hamming bits as a *std_logic_vector(6 downto 0)*. The algorithm to calculate the Hamming bits is taken from Xilinx Application Note 988 [CT08] and is synthesized using around 44 LUTs for a 32 bit word.

6.5.3 Adding new Peripherals

New Wishbone peripherals can be added in the design's toplevel file *wb_arbiter.vhd* and do not need to touch any signals within the CPU. The currently connected peripherals are listed in a VHDL type definition and use records for the incoming and outgoing signals:

```
signal from_bram, from_ddr, from_uart : wb_slave_out_master_in;
signal to_bram, to_ddr, to_uart : wb_slave_in_master_out;
type PERIPHERALS is (NONE, BRAM_CTRL, DDR_CTRL, UART_CTRL);
```

The following steps have to be done to add a new peripheral:

- add the peripheral to the signals and types described above
- add the new peripheral type to the *data_signals* process
- connect the broadcast signals to the new peripheral
- add the new peripheral in the *address_mapper* process with the desired address range
- connect the according Wishbone control signals in the *control_signals* process

6.5.4 The current Address Mapping

Every peripheral connected to the Wishbone bus is listening to a defined range of addresses. The current address mapping is shown in table 6.6. This mapping is chosen arbitrary and can easily be changed in the *address_mapper* process in the toplevel file. Accessing an unmapped address results in a bus cycle termination via ERR. The UART controller currently occupies an address range of 128 MByte, however using only two registers internally.

Start	End	Size	Peripheral
0x00000000	0x00001FFF	8 kByte	BlockRAM
0x10000000	0x17FFFFFF	128 MByte	DDR-SDRAM
0x20000000	0x27FFFFFF	-	UART

Table 6.6: Implemented Wishbone address mapping

6.6 Peripherals

6.6.1 UART

The UART controller is mostly taken from the original Plasma project and extended with a Wishbone interface. It is implemented to handle one start bit, eight data bits and one stop bit. The baud rate can be set in the configuration package via *COUNT_VALUE* by setting the number of system clock cycles required for one UART bit. This value is currently set to 115200 Hz for a clock frequency of 71 MHz. The UART controller distinguishes two addresses: one common address to fetch the read data or to write data via UART, and a second address to access the read-only UART status register. The status register indicates, whether the UART is currently busy writing or if new data has been received. There is no write FIFO implemented in the UART. If the CPU wrote data to the UART while it is busy, the UART would make the CPU to wait for the transfer to finish before answering with ACK. By checking the status register before writing, the CPU can determine, whether the UART is busy and can continue with other calculations in the meantime. The current

address mappings for both addresses are shown in table 6.7 and the bit explanation of the status register are described in table 6.8.

Address	
0x20000000	UART_READ and UART_WRITE
0x20000020	UART_STATUS

Table 6.7: Address mapping for the UART and status bit explanation

UART_STATUS bits	
0	'1': new data available
1	'1': UART is busy writing
2-31	unused

Table 6.8: UART status bit explanation

The status bit zero indicating new data is also used as output to the CPU and is connected to one of the CPU's interrupt lines. Any time a character is received, this bit goes high and raises an interrupt in the CPU. This bit stays high as long as the received word is not read from the CPU.

There are two versions of the UART included, one version with conventional UART behavior (*uart_wb.vhd*) and one version to be used in conjunction with SEU simulation or beam tests (*uart_wb_seusim.vhd*). This second version continuously transmits a diagnosis vector to report the current state of the CPU. The lower four bit are used for this diagnosis vector and the upper four bit remain writable via Wishbone bus.

6.6.2 Block-RAM

The BRAM instance is created via *Xilinx Core Generator* and currently uses four RAMB16 blocks with 16 kbit each. The current total block memory is therefore 8 kByte. This amount is chosen arbitrary and was enough for small test software. There are a lot of unused BRAMs available, so this may be extended if needed. The locations of these current four RAMB16 blocks are set in the constraint file. The contents of the BRAMs can be changed in the final bitfile using the Xilinx *data2mem* software. This allows the CPU software to be changed in the final bitfile. The work flow to change the software is further shown in chapter 6.9. The BRAM instance has been extended with a small state machine to enable access via Wishbone. A BlockRAM instance using the Xilinx ECC-BlockRAMs with redundancy is currently in development.

6.6.3 DDR-SDRAM Controller

The DDR-SDRAM controller used in this work is taken from [Pai07] and has been ported to the Syscore 1 board. Most of the required changes could be done in the controller's configuration package by setting the appropriate address widths, the refresh frequency and the IDELAY value. The change of the address widths required a minor change in the address path to retain the refresh-controlling bits at the correct position within the address line. A further change in the controller was required, because it used differential output buffers at a site where the Virtex-4 FPGA did not support them. They had to be replace by two regular output buffers with opposite clocks. The ported version for the Syscore 1 board is attached to this work.

6.6.4 SEU-Analyzer

The SEU-analyzer is a module to record and report the errors detected by the CPU. This module is not required for the system to run, but is used to analyze the behavior of the CPU during SEU simulation and beam tests. It is not connected to the Wishbone bus, but accesses several CPU signals directly. The inputs and outputs to this module are:

```
sim_pl_error : in std_logic_vector(1 downto 0);
sim_wbi_error : in std_logic_vector(1 downto 0);
sim_mem_pc0 : in std_logic_vector(31 downto 2);
sim_mem_pc1 : in std_logic_vector(31 downto 2);
sim_mem_pc2 : in std_logic_vector(31 downto 2);
sim_jmp_error : in std_logic_vector(2 downto 0);
uart_ack : in std_logic;
diag_data : out std_logic_vector(3 downto 0)
```

sim_pl_error is the two bit error signal resulting from the second step comparison within the CPU. If this value is differing from "00" an error in any part of the CPU was detected. *sim_wbi_error* is a sub-element from *sim_pl_error* indicating only errors detected via Hamming encoding of the Wishbone bus signals. The signals *sim_mem_pc0/1/2* are the current values of the PC addresses in MW-stage from all three instances of the program counter. The voted majority of these signals is again monitored for changes. *sim_jmp_error* indicates when the CPU resets itself due to a stuck MW-stage PC for more than three scrubbing cycles. These four error conditions are mapped to the four bit diagnosis vector *diag_data* shown in table 6.9. The UART is configured to continuously transmit the content of its internal data register. The upper four bit of this eight bit register remain writable via Wishbone bus, whereas the lower four bit are fed with the diagnosis output of the SEU analyzer. The diag_data register is reset each time after its contents have been accepted to be written by *uart_ack* and accumulates all events occurring during the write.

diag_data bit	value	meaning	value	meaning
0	'1'	general error detected	'0'	no error detected
1	'1'	Wishbone error detected	'0'	no Wishbone error
2	'1'	CPU auto-reset done	'0'	no reset
3	'1'	PC in MW-stage still changing	'0'	PC stuck

Table 6.9: Encoding of the SEU-analyzer diagnosis output

Any event driving one of the diag_data lines for longer than one clock cycle will be captured and transmitted via UART in the next write cycle. If no error occurs, the UART will continuously transmit $0x8_{hex}$ in the lower four bits: $0x8_{hex} = "1000"_2 \rightarrow$ no general error, no Wishbone error, no reset, and the PC is still changing.

6.7 Coding Techniques for Redundant Logic

The modern HDL compilers do a lot of effort to optimize the logic describes with an hardware description language (HDL). They try to simplify the user's input, map it to the target technology and detect identical or unused signals in order to use the required resources optimally. Logic may also be duplicated for timing issues. This is a great benefit when writing high level HDL code, but can make a lot of trouble when trying to infer redundancy intentionally. With the standard compiler settings, any detected redundancy will be removed. This may automatically remove any SEE mitigation logic and thus, will not result in an enhanced fault tolerance even if the redundancy is described with the HDL.

One starting point is to deny the removal of equivalent registers. The option to disable equivalent register removal in XST for the whole project can be found in systhesis options: *Synthesis Options* → *Xilinx Specific Options* → *Equivalent Register Removal*. This may not be the desired case, if several modules are implemented redundantly whereas each single module is designated to be optimized. The solution to this is to use VHDL attributes. Attributes can be applied to wide range of objects, among others to signals, types, functions, labels, entities etc. One of the XST supported attributes is *equivalent_register_removal*:

```
architecture Behavioral of cpu_wb is
[...]
attribute equivalent_register_removal : string;
signal pl_error_reg : std_logic_vector(1 downto 0);
attribute equivalent_register_removal of pl_error_reg: signal is "no";
[...]
```

The equivalent attribute for the Synplicity compiler is *syn_preserve*:

```
library synplify;
use synplify.attributes.all;
[...]
architecture Behavioral of cpu_wb is
[...]
attribute syn_preserve : boolean;
signal pl_error_reg : std_logic_vector(1 downto 0);
attribute syn_preserve of pl_error_reg: signal is true;
[...]
```

This piece of code will make *pl_error_reg* to be implemented with two registers even if both of its values are detected to be identical. This has been used on most of the redundant signals. The XST attribute *keep* may be useful, too, as it denies nets to be absorbed into logic blocks.

Simply denying the removal of equivalent registers may not be sufficient to implement redundancy. Even if the registers are preserved, the logic creating the data input for the registers may be collapsed into one instance feeding both registers. This can be avoided by implementing the module as an own entity and instantiating it twice. The contents of different instantiations are usually not merged.

A possibility to replicate single look-up tables is to instantiate them with switched inputs and therefore different configurations. XST does not detect their identity. This has been used in the TMR program counter for an enable signal.

6.8 The non-hardened CPU

In order to allow an efficiency estimation of the implemented SEE mitigation methods, the area overhead of redundancy and power consumption, the same system of CPU and peripherals has also been built without any radiation tolerance techniques applied. This non-hardened version of the CPU uses the same pipeline modules as the fault tolerant implementation, but is lacking any redundancy or error detection mechanisms. The pipeline has been implemented in one single instance, using one register bank. The results are written back to one of the registers or to the bus as they drop out of MW-stage. There are no tag-fields carrying Hamming codes in the Wishbone bus and the ERR transfer termination signal is not supported. The program counter is implemented as a single instance and does not monitor, if the MW-stage PC is still changing. The addition of a module recording and reporting the current state of the CPU like the SEU-analyzer described above makes also no sense in this implementation. This version of the CPU allows

the same software to be used as the previously described fault tolerant implementation and behaves in the same way on interrupts or exceptions.

6.9 Running Software on the CPU

The software for both versions of the CPU can be developed using a standard MIPS cross compiler environment. The plasma project offers a Windows version of the MIPS GCC ELF compiler for download from its project page [Rho09]. A linux version of the *GNU toolchain for MIPS Processors* can be downloaded from CodeSourcery[1]. The light edition of this package is free and includes any tools required to compile own software. In order to create software with the MIPS I instruction set, the according *architecture* flag has to be set when running GCC. The listing below shows a section from the makefile, mostly taken from the plasma project:

```
GCC_MIPS  = $(BIN_MIPS)/mips-linux-gnu-gcc $(CFLAGS)
CFLAGS = -Wall -c -s -march=mips1
```

The CPU starts executing the code from its internal address 0x00000000. This is the lowest BRAM address. Code to be executed has therefore to be put into the BRAMs. The plasma project includes some software tools to automatically create a VHDL file containing the BRAMs with appropriate content. This is no flexible solution, because the whole system had to be re-synthesized to change the software. The later part of this section will show, how the BRAM content can be changed in the final bitfile without re-synthesizing the whole design. However, this tool flow can be used and modified to get the compiled software as a list of 32 bit instructions. The *convert* tool coming with the plasma project can be used to convert the compiled MIPS binary into an ASCII list of 32 bit hex instructions. This format is, apart from a small header, mostly similar to the coe-file format used for BRAM initialization files and allows the change of the BRAM content for both synthesis and simulation. The coe file can be created with a small script (*make_coe.pl*).

6.9.1 Creating Own Applications

The applications coming with the plasma project mainly consist of three parts. An Assembler file is used as initialization file. Pointers are set, memory areas are cleared and the jump to the C-code entry point is included. Furthermore, there is some basic interrupt / exception handling and some special functions are written in Assembler, too. The second part is a C-file containing basic function definitions and the high level interrupt and exception handling. The third part is the actual user application. This part can be written in C, too, and can use all functions defined in one of the prior parts.

[1] http://www.codesourcery.com/sgpp/lite/mips

Part 1: The Assembler File

This file is the entry point to the software world and is executed from its beginning. This part is mostly taken from the plasma project. A sample entry point file is shown below:

```
   .comm InitStack, 512       #Reserve 512 bytes for stack
   .text
   .align 2
   .global entry
   .ent entry
entry:
   .set noreorder
   la    $gp, _gp             #initialize global pointer
   la    $5, __bss_start      #$5 = .sbss_start
   la    $4, _end             #$4 = .bss_end
   la    $sp, InitStack+488   #initialize stack pointer
$BSS_CLEAR:
   sw    $0, 0($5)            #write zero to address in $5
   slt   $3, $5, $4
   bnez  $3, $BSS_CLEAR       #restart with BSS_CLEAR if bss_end is not reached
   addiu $5, $5, 4            #branch delay slot: increment $5 by 4
   jal   main                 #jump to main() (see part 3)
   nop                        #nop in branch delay slot
```

These lines of code initialize the global and stack pointers according to the information from the compiler. The *$BSS_CLEAR* routine overwrites the whole memory range for .bss with zero. This is a nice example for the efficient usage of the branch delay slot. As long as the address loop is running, the *BNEZ* instruction gives the jump condition to restart, whereas the incrementation of the address pointer is done in the branch delay slot. If the loop completed, the program will jump to the *main()* function defined in the C-file of part three.

A second example for code in the Assembler file is the basic interrupt / exception handling. If an exception occurs, the CPU jumps to the exception handler address. This is currently set to 0x0000003C, so a valid piece of code has to be at this address. In this case, this is reached by "counting" the instructions in the Assembler file and placing the exception handler code at the right place. Parts of the exception handler are shown below:

```
#Save all temporary registers
  sw    $1,  16($29)    #at
  sw    $2,  20($29)    #v0
  sw    $3,  24($29)    #v1
  sw    $4,  28($29)    #a0
  sw    $5,  32($29)    #a1
  sw    $6,  36($29)    #a2
  sw    $7,  40($29)    #a3
  sw    $8,  44($29)    #t0
  sw    $9,  48($29)    #t1
  sw    $10, 52($29)    #t2
  sw    $11, 56($29)    #t3
  sw    $12, 60($29)    #t4
  sw    $13, 64($29)    #t5
  sw    $14, 68($29)    #t6
  sw    $15, 72($29)    #t7
  sw    $24, 76($29)    #t8
  sw    $25, 80($29)    #t9
  sw    $31, 84($29)    #lr
#load EPC
  mfc0  $26, $14
#adjust return address according
#to cause and BD-flag
[...]
#save return address
  sw    $26, 88($29)
  mfhi  $27
  sw    $27, 92($29)    #hi
  mflo  $27
  sw    $27, 96($29)    #lo
#use cause as function argument
  mfc0  $4,  $13
  jal   OS_InterruptServiceRoutine
[...]
```

```
[...]
#Restore all temporary registers
  lw    $1,  16($29)    #at
  lw    $2,  20($29)    #v0
  lw    $3,  24($29)    #v1
  lw    $4,  28($29)    #a0
  lw    $5,  32($29)    #a1
  lw    $6,  36($29)    #a2
  lw    $7,  40($29)    #a3
  lw    $8,  44($29)    #t0
  lw    $9,  48($29)    #t1
  lw    $10, 52($29)    #t2
  lw    $11, 56($29)    #t3
  lw    $12, 60($29)    #t4
  lw    $13, 64($29)    #t5
  lw    $14, 68($29)    #t6
  lw    $15, 72($29)    #t7
  lw    $24, 76($29)    #t8
  lw    $25, 80($29)    #t9
  lw    $31, 84($29)    #lr
  lw    $26, 88($29)    #EPC
  lw    $27, 92($29)    #hi
  mthi  $27
  lw    $27, 96($29)    #lo
  mtlo  $27
  addi  $29, $29, 104   #adjust sp
#jump to return address
  jr    $26
#BD-slot: return from exception
  rfe
[...]
```

The left column shows how the temporary register values are stored to memory. The address of the interrupted instruction is automatically stored in the EPC co-processor register. This value is loaded and modified to be used as return address. In case of an interrupt, the interrupted instruction must be used as return address. In case of a SYSCALL or BREAK instruction, using the same value would result in an endless loop. Another aspect is the branch delay flag in the cause register. If this value is set, the EPC value has at least to be decremented by four to point to the preceding jump instruction as return address. The values from the HI and LO registers are saved, too. After the context save, the program jumps to the *OS_InterruptServiceRoutine*. This function is part of the function definition file described in part two. After returning from this function, the previous context is restored (right column) and the CPU jumps to the according return address. The *RFE* instruction in the branch delay slot restores the previous values of the

interrupt enable (IEp) and kernel/user mode (KUp) flags as described in chapter 6.4.5.

A third example is the *irq_enable()* function. Functions like this are easier in Assembler than in C and the definition in the Assembler file makes them accessible to any of the C files.

```
.global irq_enable
.ent irq_enable
irq_enable:
   .set noreorder
#set IRQ-mask to 0xFF and IEc to 1
   ori   $24, $0, 0xff01
   jr    $31
   mtc0  $24, $12       #enable interrupts
   .set reorder
.end irq_enable
```

Part 2: The Function Definitions

This file is used for function definitions in C. Sample functions are the high level exception handler *void OS_InterruptServiceRoutine(int cause)* with its cause parameter described above or simple functions to read or write memory.

```
#define MemoryRead(A) (*(volatile unsigned int*)(A))
#define MemoryWrite(A,V) *(volatile unsigned int*)(A)=(V)

void putchar(int value)
{
  while(!(MemoryRead(IRQ_STATUS) & IRQ_UART_WRITE_AVAILABLE));
  MemoryWrite(UART_WRITE, value);
}
```

This example shows a very simple UART driver. The status register of the UART is read to check whether the UART is busy or not and waits for it to be ready. After that, the requested value is written. This, for example, may be extended to deliver functions for writing strings or numbers. The function definition file may be used for any user defined functions.

Part 3: The User Application

The third part is the actual user application. This application can access any functions described in the previous parts. The Assembler file in the first part initiates a jump to the user application's *main()* function.

```
extern void irq_enable();
extern int puts(const char *string);

int main()
{
  irq_enable();
  puts("Hello World\n");
  while(1);
  return 0;
}
```

Putting it all together

All three parts have to be combined to a single application. They are compiled for the MIPS I instruction set with a MIPS GCC and put together by the linker. An example makefile is shown below:

```
myapp:
  $(AS_MIPS) -o boot.o boot.asm
  $(GCC_MIPS) functions.c
  $(GCC_MIPS) myapp.c
  $(LD_MIPS) -Ttext 0 -eentry -Map test.map -s -N -o test.axf \
    boot.o functions.o myapp.o
  @$(DUMP_MIPS) --disassemble test.axf > test.lst
  convert_bin.exe
```

boot.asm is the Assembler file from part one, *functions.c* is the function definition file from part two and *myapp.c* is the user application from part three. The linker combines all three compiled binaries to one program. The disassembly is optional, but can help to debug applications. The resulting *test.axf* has still a file header. This header is removed by the *convert* tool supplied with the plasma project, so the binary can directly be executed from the CPU.

There are a lot of free MIPS simulators to test the software on a common PC, the plasma project even supplies its own, but none of them have been used or tested in this work.

6.9.2 Changing BRAM Contents

In order to run software on the CPU, the instructions have to be placed into the BRAMs. This can be done before sythesis as described above, or it can be done in the final bitfile. Changing the bitfile saves a lot of time, because the design does not have to be synthesized again. Xilinx delivers a tool called *Data2Mem* to change the BRAM contents of a

bitfile. This tool is capable of changing the content of any BRAM block within the FPGA, so it has to be known where the "used" BRAMs are placed on the chip. It is advised to fix the locations of the used BRAMs with location contraints in the user constraint file (UCF) during implementation. A sample placement contraint is shown below:

```
INST "BRAM/blockram/BU2/U0/blk_mem_generator/valid.cstr/ramloop[3].ram.r/
  v4_init.ram/SP.SINGLE_PRIM.SP" LOC = "RAMB16_X3Y1";
INST "BRAM/blockram/BU2/U0/blk_mem_generator/valid.cstr/ramloop[2].ram.r/
  v4_init.ram/SP.SINGLE_PRIM.SP" LOC = "RAMB16_X3Y0";
INST "BRAM/blockram/BU2/U0/blk_mem_generator/valid.cstr/ramloop[1].ram.r/
  v4_init.ram/SP.SINGLE_PRIM.SP" LOC = "RAMB16_X4Y1";
INST "BRAM/blockram/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/
  v4_init.ram/SP.SINGLE_PRIM.SP" LOC = "RAMB16_X4Y0";
```

This information can be used to tell *Data2Mem* which BRAMs are used and how they are arranged. This has to be done via a *BMM* file as shown below:

```
ADDRESS_MAP mymap PPC405 0
ADDRESS_SPACE BRAM COMBINED INDEX_ADDRESSING[0x00000000:0x00001FFF]
ADDRESS_RANGE RAMB16
BUS_BLOCK
  wb_arbiter/BRAM/BU2/U0/blk_mem_generator/valid.cstr/ramloop[3].ram.r/
    v4_init.ram/SP.SINGLE_PRIM.SP [31:24] PLACED=X3Y1;
  wb_arbiter/BRAM/BU2/U0/blk_mem_generator/valid.cstr/ramloop[2].ram.r/
    v4_init.ram/SP.SINGLE_PRIM.SP [23:16] PLACED=X3Y0;
  wb_arbiter/BRAM/BU2/U0/blk_mem_generator/valid.cstr/ramloop[1].ram.r/
    v4_init.ram/SP.SINGLE_PRIM.SP [15:8] PLACED=X4Y1;
  wb_arbiter/BRAM/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/
    v4_init.ram/SP.SINGLE_PRIM.SP [7:0] PLACED=X4Y0;
END_BUS_BLOCK;
END_ADDRESS_RANGE;
END_ADDRESS_SPACE;
END_ADDRESS_MAP;
```

The locations of the requested BRAMs are specified and an address space for the whole BRAM content is created. This address space is only to define the relative organization of the used BRAMs and has nothing to do with the CPU's address space. The current configuration shows four BRAM blocks arranged in parallel, where each block contributes eight bit to the resulting 32 bit word stored in all four blocks.

In combination with this *BMM* file, *Data2Mem* can be fed with data. The compiled binary file without header dropping out of the compilation process sketched above has to be converted into a format *Data2Mem* understands. This format is chosen to be *MEM*, an ASCII

format showing the hexadecimal representation of the instructions. The SRecord package[2] supplies a tool called *srec_cat* to convert the binary file into this *MEM* format. The code snippet below shows the part of the make flow creating the *MEM* file and changing the BRAM content of a bitfile.

```
srec_cat $(SW)/test.bin -binary -o $(PROG).mem -vmem 8
data2mem -bm $(BMM_file) -bt $(BIT_file) -bd $(PROG).mem -o b $(PROG).bit
```

test.bin is the software binary without header. This file is converted to *MEM*. The *BMM* file describes the organization and arrangment of the targeting BRAMs, *BIT_file* is the bitfile to be altered and the *MEM* file contains the new BRAM content. According to this information, *Data2Mem* creates a new bitfile with the requested content in the appropriate BRAMs. This file can directly be uploaded into the FPGA.

[2]http://srecord.sourceforge.net/

7 Partial Bitfiles and SEU Simulation

This chapter describes how partial bitfiles can be created and how they can be used for both scrubbing and SEU simulation. Simulating SEUs within the real hardware without the need of a radiative environment allows a reliable estimation of the SEU tolerance of a specific design implementation.

7.1 Xilinx Configuration Protocol

At the end of each synthesis tool flow is a bitfile containing the final value for each configuration bit in the targeted FPGA. This bitfile can be uploaded into Xilinx FPGAs using one of the available configuration interfaces: Serial, SelectMAP or JTAG. Serial mode configures, as the name implies, the FPGA with one bit per configuration clock cycle. SelectMAP provides an eight bit bi-directional data interface to the FPGA, which can be used for both reading and writing the configuration. The JTAG interface gives an access port to the configuration using the *IEEE 1149.1 Test Access Port and Boundary Scan Architecture*. This standard is a widely used test and debugging possibility. More Information about JTAG can be found in [MK09]. The actual configuration process is controlled by several configuration options. These options control the write process and must be set before any FPGA configuration data is written. Therefore, the bitfile to be uploaded consists of the actual FPGA configuration data trailed by commands to write the options registers and initiate the actual configuration writing. A complete overview of all configuration registers and the process of configuration is described in the *Virtex-4 Configuration User Guide* [Xil08e], chapter 7: *Configuration Details*.

7.2 Creating Partial Bitfiles

The bitfiles created by the Xilinx tools are per default full bitfiles. Full bitfile means, a complete set of configuration commands is used. This includes commands for resetting all flip-flops and input/output registers to their initialization values and pulling all interconnects to high impedance during the whole configuration process. A sample configuration command sequence is shown in table 7.1. Furthermore, every full bitfile contains the initial configuration of any BRAM block on the chip and overwrites any previous content. These commands are not wanted for dynamic reconfiguration, as their use would

make the whole design restart from its reset state. The purpose of scrubbing is, to continuously write the same partial bitfile during runtime without resetting the device after each write and without overwriting dynamic values. To enable partial reconfiguration, several requirements have to be fulfilled:

- Allow SelectMAP pins to persist: In a standard FPGA design, the pins used for SelectMAP do not need to remain reserved for programming purposes only and are per default freed for any user designs. If reconfiguration using direct SelectMAP access is needed, these pins must not be freed after the initial programming and must be persisted as SelectMAP pins. This can be reached by enabling *Create ReadBack Data Files* and *Allow SelectMAP Pins to Persist* via properties of *Generate Programming File → Readback Options* or by setting the *-g Persist* option in *bitgen*. This option is important if configuration is done from the Actel FPGA, but does not affect JTAG configuration.
- Drive DONE pin high: The DONE pin is an FPGA output pin indicating, if the current programming cycle has finished. This option can be set in properties of *Generate Programming File → Startup Options*. This is also required for SelectMAP programming.
- Do not use LUTs as distributed memory or shift registers: Distributed memories or shift registers are implemented in look-up tables of SLICEMs as described in section 2 and their dynamic values are therefore overwritten on reconfiguration. There seem to be possibilities on Virtex-4 to use these anyway by setting appropriate values in the configuration options, but this has not been tried in this work.

A partial bitfile can be generated by taking out any of the reset or high impedance commands. In order to additionally preserve the dynamic BRAM content, this partial bitfile can be reduced to a version without BRAM contents.

7.2.1 Full bitfile format

The default full bitfile format is shown in table 7.1 and consists of the following parts:

- Bitfile header: contains mainly the filename of the bitfile and the date. This header has no fixed length as it depends on the filename. The bitfile header is not transmitted to the FPGA and is used by the Xilinx tools only.
- Fixed synchronization word: 0xAA995566. This word signals the beginning of the actual data. Any word from here is a configuration command or FPGA configuration data and is transfered to the FPGA.
- Commands to write device ID, CRC checksums and configuration options registers. The device ID in the bitfile must match the device ID stored on the FPGA. If they differ, the configuration will not be written. Checking the device ID can be disabled

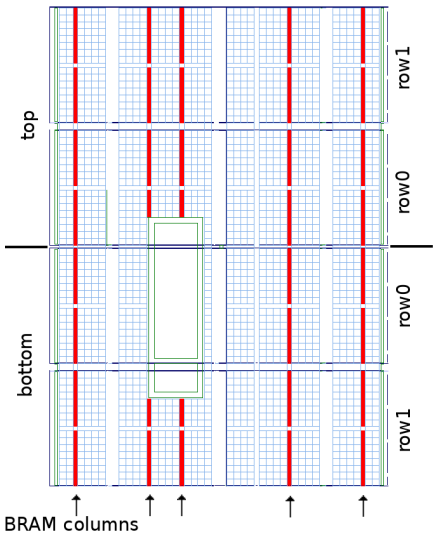


Figure 7.1: Modified PlanAhead screenshot showing the BRAM organization within the Xilinx Virtex-4 FX20 FPGA. The BRAM columns are colored red. The white rectangle in the center is the PowerPC core.

by setting the appropriate options. A description of all configuration options can be found in the *Virtex-4 Configuration User Guide* [Xil08c].

- FPGA configuration data to set any configuration bit within the whole FPGA. This is initiated by a Type-2 command [Xil08c] specifying the number of configuration words to be written.
- Commands to reset the device and set all interconnects to high impedance.
- Commands to release the reset condition and start the device operation.

7.2.2 Removing BRAM Contents

The internal organization of the FPGA and the number of configuration bits required for specific resources like CLBs, IOBs, BRAMs, DSPs and so on is roughly described in *Xilinx Application Note 988* [CT08]. In combination with FPGA-Editor or PlanAhead, the organization for the specific device can be deduced. The BRAM contents are the last data written into the FPGA during the configuration and can therefore be removed by some simple calculations. As shown in figure 7.1, the Virtex-4 FX20 is arranged in two halves (top / bottom half) and two rows on each half. The rows are counted beginning from the middle, so the device in top-down-view consists of: top row1, top row0, bottom row0, bottom row1. Each row has five columns of BRAM blocks. The "missing" BRAMs due to the PowerPC did not show any effect and the experiment proved that they can be counted in this calculation as if they were there.

The smallest writable unit is a *frame*. Each frame consists of 41 words with 32 bits per word. A BRAM content column requires 64 frames for configuration. With five columns

in two rows and two halves this results in:

$$N_{BRAMwords} = 41 * 64 * 5 * 2 * 2 = 52480 \ \ words = 0xCD00_{hex} \ \ words$$

This means, the complete BRAM contents for all Block RAMs on the FPGA are stored in the last 0xCD00 words of the FPGA configuration data. Looking into a full bitfile at the Type-2 command shows, that a full bitfile uses 0x36EF0 words for a complete configuration. By changing this number to 0x36EF0 - 0xCD00 = 0x2A1F0 and cutting out the last 0xCD00 words of configuration data one can create a bitfile without any BRAM contents.

7.2.3 Getting Rid of the Reset Commands

The bitfile is a binary file, but any hex code used is documented in the *Virtex-4 Configuration User Guide* [Xil08c]. The commands responsible for resetting and high impedance operations are *START*, *DGHIGH / LFRM* and *GRESTORE*. These commands ensure in a full bitfile that the device starts from a well defined state. In order to use scrubbing, a bitfile without these commands but with the correct FPGA configuration data is required. Scrubbing has no effect on the running design, if there are no errors to be corrected. If errors are corrected with scrubbing or if the partial bitfile is used to inject errors into the FPGA configuration, these changes will come asynchronously to any device clock, they do not need to comply with the internal setup- or hold times.

As partial bitfiles are only used after an initial full configuration, some of the configuration options do not need to be set again in the partial bitfile. A good starting point for creating partial bitfiles is to let Xilinx' *bitgen* create a partial bitfile using the *-r* option and looking into that. A comparison between the original full bitfile and the partial bitfile is shown in table 7.1. The main differences are:

- Mask and control registers do not need to be written again
- Amount of data to be written is reduced by the BRAM contents
- GRESTORE, DGHIGH/LFRM and START are taken out
- CRC is not used
- number of NOPs and order of the commands slightly differ

7.2.4 Automating the Creation of Partial Bitfiles

A little C program called *parbitgen* has been written during this work and is attached to this thesis. The program automatically creates partial bitfiles out of full bitfiles by removing both, reset commands and BRAM contents. It is currently configured for Virtex-4 FX20 FPGAs, but it should be no problem to extend this to other devices.

Full bitfile configuration sequence:	Partial bitfile configuration sequence:
SYNC WORD (aa995566)	SYNC WORD (aa995566)
NOP	NOP
WRITE REG CMD(4): RCRC(7)	WRITE REG CMD(4): RCRC(7)
2x NOP	2x NOP
WRITE REG COR(9): 11043fe5	WRITE REG DEVICE_ID(12): 01e64093
WRITE REG DEVICE_ID(12): 01e64093	WRITE REG COR(9): 11043fe5
WRITE REG CMD(4): SWITCH(9)	
NOP	
WRITE REG MASK(6): 00000600	
WRITE REG CTL(5): 00000600	
1149xNOP	
WRITE REG MASK(6): 00000600	
WRITE REG CTL(5): 00000000	
WRITE REG CMD(4): NULL(0)	
NOP	5x NOP
WRITE REG FAR(1):00000000	WRITE REG CMD(4): WCFG(1)
	NOP
WRITE REG CMD(4): WCFG(1)	WRITE REG FAR(1): 00000000
NOP	NOP
WRITE REG FDRI(2) with 00036ef0 words:	WRITE REG FDRI(2) with 0002a1f0 words:
36EF0 words of FPGA configuration... .	2A1F0 words of FPGA configuration...
WRITE REG CRC(0): 0000defc	
WRITE REG CMD(4): GRESTORE(10)	
NOP	
WRITE REG CMD(4): DGHIGH/LFRM(3)	
99x NOP	101x NOP
WRITE REG CMD(4): GRESTORE(10)	
NOP	
WRITE REG CMD(4): NULL(0)	
NOP	
WRITE REG FAR(1): 00008ac0	
WRITE REG CMD(4): START(5)	
NOP	
WRITE REG MASK(6): 00000008	
WRITE REG CTL(5): 00000008	
WRITE REG CRC(0): 0000defc	
WRITE REG CMD(4): DESYNCH(13)	WRITE REG CMD(4): DESYNCH(13)
16x NOP	4x NOP

Table 7.1: Comparison between full and partial bitfile configuration command sequences. The main differences in the configuration commands are colored red whereas the actual configuration data, which covers the main part of the bitfile, is not shown here

7.2.5 Partial Bitfiles Used for Scrubbing

The resulting bitfile is used by the peripheral Actel FPGA running the scrubbing mechanism. Due to the fact, that BRAM contents are left out of this file, it is significantly smaller in size. The full bitfile had 0x36ef0 configuration words, the partial bitfile has only 0x2a1f0 words. As the scrubbing cycle time is dependent of the words to be written, writing a partial bitfile without BRAM contents gives a speedup of factor 1.3 on a Virtex-4 FX20. If BRAMs are not used at all, one could consider calculating out the BRAM interconnect frames as well.

7.3 Error Injection through Partial Reconfiguration

Working with these partial bitfiles mentioned above allows them not only to be used for scrubbing. Changing one single bit in the FPGA configuration part of the partial bitfile should have the same effect as a radiation particle flipping a single SRAM cell. Writing a correct partial bitfile has the same effect as doing scrubbing. In order to write defective bitfiles with a single configuration bit flipped, the CRC mechanisms of the configuration protocol have to be disabled. Without disabling these, a CRC error would be detected and the bitfile would not be written. CRC can be disabled in ISE by unchecking *Enable Cyclic Redundancy Checking (CRC)* in properties of *Generate Programming File → General Options*.

The partial bitfiles can therefore be exploited to simulate SEUs within the real hardware implementation. The flipped bit in the bitfile will change a bit in the FPGA configuration. This bit will flip during the write of the configuration and asynchronous to any system clock. This means, the flipping bit may violate setup- and hold times. These are the same effects that can be seen on radiation induced direct SEUs. SETs cannot be simulated in this way because partial reconfiguration is not capable of giving short pulses on the FPGA routing nets or configuration bits. Unfortunately, the exact meaning of the currently flipped bit in the design cannot be reconstructed as the correlation between bitfile bits and actual FPGA configuration bits is not officially published at this time. However, there are approaches to decode bitfiles like shown in [NR08]. Nevertheless, a rough estimation in which part of the design the flipped bit will affect the logic, can be done by counting the frames.

Device	CLB cols	IO cols	DSP cols	CLK cols	MGT cols	PAD cols
Virtex-4 FX20	36	3	1	4	2	1

Table 7.2: Row contents of a Virtex-4 FX20 configuration row. The number of rows for CLBs, IOs, DSPs, CLKs and MGTs can be found in PlanAhead and FPGA-Editor. The PAD-frame is written at the end of each row and is roughly documented in [Xil08c].

	CLB	IO	DSP	CLK	MGT	BRAM interconnect	BRAM content	PAD frame	
configuration frames per element	22	30	21	2	20	20	64	1	

Table 7.3: Number of configuration frames per element in a Virtex-4 FPGA. This information is taken from [CT08] and [Xil08c].

The FPGA is configured beginning at the middle writing top row0 → top row 1 → bottom row0 → bottom row1. According to PlanAhead and FPGA-Editor, one row in the Virtex-4 FX20 has the columns shown in table 7.2. The ordering of these columns can be seen in PlanAhead and FPGA-Editor, the number of frames required for each component can be found in table 7.3 and [CT08]. By reconstructing the frame number of the flipped bit within the bitstream, one can decide in which half (top/bottom), row (0/1) and frame position (CLB, IO, CLK, DSP, MGT) the flipped bit will affect the targeted design. The logic affiliated with this frame can be seen in FPGA-Editor. Unfortunately, this is the finest possible grain with this method. The exact part of the logic affected with a certain configuration bit cannot be reconstructed.

7.3.1 Floor-Planning

As shown above, SEU simulation can easily be done by modifying the bitfiles. Configuration errors can be injected into any part of the FPGA. This is a great benefit, but also a problem in this case. This thesis covers a radiation hardened CPU only without using radiation hardened peripherals. Injecting errors in these peripherals would give wrong statistics about the radiation tolerance of the CPU. The solution to this is to use floor-planning: The CPU is placed in a well defined area of the FPGA, whereas the peripherals are forced into another area. Injecting errors into the CPU's area only gives a reliable statistic of the CPU's radiation tolerance.

There are several solutions for reconfiguring specific parts of the FPGA only. Xilinx delivers some methods granting reliable signal transfer between static and dynamic FPGA areas via bus macros and allows writing of rectangular areas within the FPGA by "jumping" to different frame addresses during the write. All these methods deliver more functionality and complexity as needed in this case. As already mentioned above, the FPGA configuration rows are being written in the order top row0 → top row1 → bottom row0 → bottom row1. By cutting the configuration after bottom row0, only 3/4 of the FPGA would be written and/or changed by flipped bits. This means 75% of the chip area could be used for the CPU and 25% for all the peripherals. Unfortunately, 3/4 of the chip was not sufficient for the CPU with increasing fault tolerance, so parts of bottom row0 had

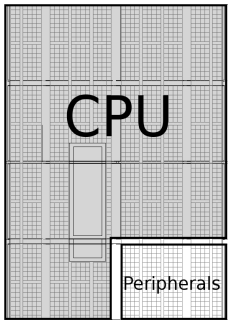


Figure 7.2: Sketch of the Xilinx FX20 floor-plan for SEU simulation. The CPU is placed in the top three rows plus the half of the forth row. The whole CPU area is chosen in this way, because the lowest row is written last and any row is written from left to right. Placing the CPU according to these facts allows "partial reconfiguration" by simply interrupting the write at a certain point.

to be used for the CPU, too. Stopping the configuration within a row requires the calculations sketched above. The rows are written from left to right. Therefore, the left half of the row can additionally be used for the CPU. This includes 1 MGT column, 2 CLK columns, 1 IO column and 18 CLB columns. These numbers are extracted from PlanAhead's device overview for a FX20. Combining this with tables 7.2 and 7.3 gives a total number of 3*952 frames for the three rows plus 432 frames for the half of the forth row to be written on error injection. A sketch of the floor-plan is shown in figure 7.2.

7.4 CPU Testing

Both, the fault tolerant implementations and the non-hardened version of the CPU have been tested for SEU tolerance using single error injection as mentioned above. The fastest way of testing many different flipped bits in respect of their effect on a running design is to use SelectMAP on chip level with a peripheral controller directly connected to the FPGA. In case of the Syscore board, this could be done using the Actel FPGA which is responsible for scrubbing. An appropriate Actel design for injecting random errors into a bitfile did not exist at this time and there are only few connections that could be used as status and debug output. Due to this fact, the SEU simulation had to be done using JTAG. A great advantage of this method is, that it can easily be seen in the uploading software whether the programming has succeeded or not. The main disadvantage is, that programming over JTAG is much slower than using SelectMAP. It takes two to three seconds for each bitfile to be loaded into the FPGA. In order to simulate SEUs via JTAG for an approach that relies on scrubbing, the scrubbing cycles have to be performed via JTAG as well. The simulation of a single bit flip therefore requires the write of an erroneous bitfile plus at least one write of the correct partial bitfile to simulate the scrubbing. The required time for the simulation of one bit flip is the sum of these write times and is in the order of five seconds per flipped bit. The FPGA area used for the CPU design includes 3288 frames = 3288 * 41 words = 3288 * 41 * 32 bit $\approx$ 4 million bit. Even without taking into account, that some flipped bits may need more than one configuration refresh for the CPU to continue its operation, the required time for simulating any flipped bit in this

area would take more than 200 days. A full testing of all bits in the design is thus not possible and the results obtained from these tests are statistical. The bits to be flipped are chosen randomly using the common C functions.

The fault tolerant versions of the CPU require more resources than the plain version. A bit flip in an unused area of the chip will not have an effect on the running design. The costs of additional chip area for the SEU mitigation methods producing further SEU sensible bits have to be taken into account. In order to get comparable results, all versions of the CPU are placed in the same FPGA area shown in figure 7.2, one CPU at a time. Error injection is done on the whole available area for the CPUs. This compensates the effects of differing resource utilization and makes all tested CPU versions directly comparable.

Test programs running on the CPUs

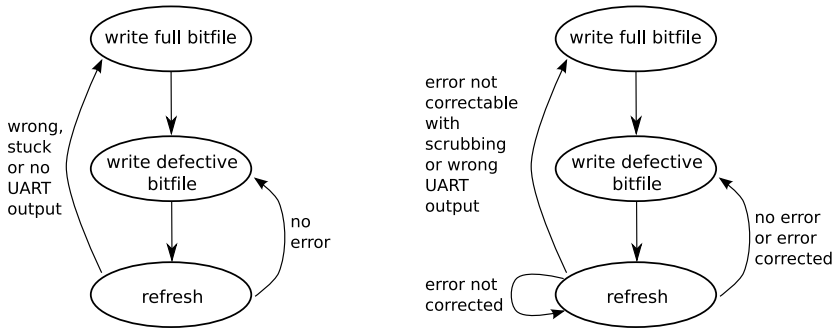
As described in chapter 6.6.4, the lower 4 bit of the UART output are used for CPU debugging in the fault tolerant versions. These bits show, whether the CPU is still running or why it has stopped. The upper 4 bit of the UART output can be used by both versions for the software running on the CPU. These bits indicate the progress of the program. The test program used covers the main aspects of the CPU:

- Calculate Fibonacci numbers: Lots of register transfers and additions
- Shift left/right: The shift amounts are calculated during runtime
- Multiply and divide

Specific bits of each calculation step's result are used as UART output. If no error occurs, the Fibonacci numbers return 0x10, shifting returns 0x20 and multiply / divide returns 0x30. This program runs in an endless loop and takes a few milliseconds for a single run through all three stages of the test program. The source code of the test program can be found on the CD attached to this work. The UART output of the FPGA system is monitored by a PC, parsing progress and debug bits. According to this data, FPGA programming, error-injection or refresh is initiated using the Xilinx tools.

SEU Simulation in the Non-Hardened CPU

The CPU is placed in the area shown in figure 7.2, the SEU simulation procedure is shown in figure 7.3(a). At the beginning, a full bitfile is uploaded. This design is reconfigured with a defective partial bitfile containing a single faulty bit. Independently of the CPUs behavior, this configuration error is corrected with a partial bitfile containing the valid design (refresh). The refresh is done to prevent that multiple bits can be flipped at the same time. If this bit flip did not produce any effect on the CPU, the next random bit is chosen to be flipped for the next iteration of the testing procedure. Wrong, stuck or no UART outputs show the results of wrong internal computations, dead lock situations or



(a) Testing of the non-hardened version

(b) Testing of the fault tolerant version

Figure 7.3: Test procedures for both versions of the CPU.

accesses to non-existing addresses. Each of these effects clearly indicate a malfunction of the CPU and force a complete reprogramming of the FPGA with a full bitfile. The number of flipped bits and the number of detected errors are recorded in combination with a detailed log of the UART output.

SEU Simulation in the Fault Tolerant Implementations

Things get little more complicated when the fault tolerant CPU design is tested. A stuck UART output for example does not have to be a malfunction, as the CPU may stop on errors. Only if the scrubbing cannot repair this, it may be counted as error. A sketch of the SEU simulation procedure is shown in figure 7.3(b). At the beginning, a full bitfile is loaded. After that, a defective bitfile with one randomly flipped bit is uploaded and the system's output is monitored to detect errors and malfunctions. A configuration refresh by uploading a clean partial bitfile to correct the previously flipped bit is done after each error injection. If no functional error was detected or the current error could be repaired with the configuration refresh, the simulation process continues with flipping the next random configuration bit. If the CPU cannot continue its operation after a refresh, the refresh is repeated several times without injecting new flipped bits until the error disappears or it is marked as uncorrectable. An error is defined as uncorrectable, if the CPU is stuck for more than six refresh cycles. This limit is chosen, because the CPU tries to reset itself after being stuck for three refresh cycles. If resetting itself fails twice, the system can be seen as irrecoverably stuck. Another reason for an irrecoverable error is a wrong UART output. In both cases, the SEU simulation procedure restarts by writing the full bitfile again including a full FPGA reset. As above, the number of flipped bits as well as the numbers of corrected and uncorrectable errors are recorded in combination with a detailed log file.

8 Results

In this work, several versions of the CPU have been implemented: a non-hardened version and three different fault tolerant versions. All versions are based on the same implementation and have the same functionality regarding the supported instruction set and their behavior towards commands or exceptions. The SEE mitigation techniques applied to the fault tolerant versions are

- continuous configuration scrubbing
- double modular redundancy applied on the pipeline
- two step or single step comparison for error detection
- Hamming codes and error signals in the Wishbone bus
- DMR register bank or single register bank with error detection
- triple modular redundancy program counter
- uncorrectable error detection and automatic CPU reset

The non-hardened version does not apply any of these fault tolerance techniques listed above. This version can be used as reference for the fault tolerant implementations to evaluate the effectiveness of the applied methods.

All implemented versions of the CPU have been compared with each other regarding SEU susceptibility, resource usage and power consumption. The SEU tests have been conducted with both, SEU simulation as described in chapter 7 and real particle beam experiments.

8.1 Resource Usage and Power Consumption

As described in the previous chapter, one non-hardened CPU and several fault tolerant CPU implementations have been done in this work. The synthesis results from all versions tested on SEU susceptibility are shown in table 8.1. This table shows the synthesis result from synthesizing solely the CPU, peripherals are not included. Both, the absolute values of the occupied resources and the relative values compared to the non-hardened implementation are shown.

Elements absolute values	non-hardened version	FT version DMR reg.bank 2 step comp.	FT version single reg.bank 2 step comp.	FT version single reg.bank 1 step comp.
slices	2416	5732	6503	5532
flip-flops	2254	4698	3605	3605
look-up tables	3502	10415	11687	10303
DSPs	4	8	8	8
relative values				
slices	100%	237%	269%	229%
flip-flops	100%	208%	160%	160%
look-up tables	100%	297%	334%	294%
DSPs	100%	200%	200%	200%

Table 8.1: Comparison of synthesis results for all tested versions of the CPU

The fault tolerant implementation with two instances of the register bank uses a little more than twice the registers of the non-hardened version. This is because any logic is duplicated and there are some further registers for Hamming bits and fault detection mechanisms. The number of look-up tables has grown to approximately the triple of the non-hardened version. This is due to the comparison logic and the Hamming encoder as both require exclusively LUTs. The fault tolerant version with one single register bank saves more than 1000 registers by omitting a second register bank. However, the number of LUTs increases by approximately the same value because any register requires an additional LUT to generate the clock_enable according to the data input signals, as shown in chapter 6.4.6. The smallest fault tolerant implementation was be done by reducing the comparison to less signals and only one step. This saves a lot of LUTs. The DSPs for all fault tolerant versions have doubled compared to the non-hardened version because any pipeline uses its own multiplication unit.

The power consumption measurements have only been conducted roughly by monitoring the current consumption of the whole Syscore board with the power supply display. This includes a complete system with CPU, BRAMs and UART controller. The values for all implemented versions plus the current consumption of the board without any design loaded have been recorded and are summarized in table 8.2. The current consumption increases with increasing resource usage.

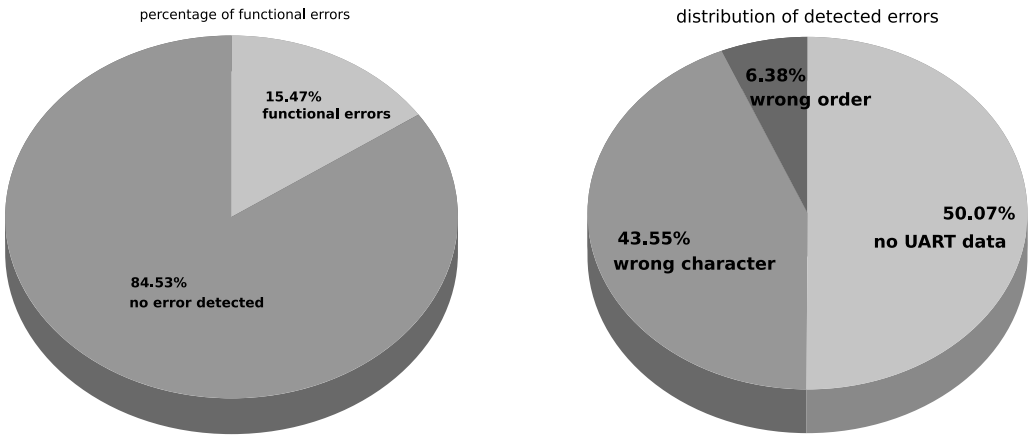
Design	none	non-hardened	DMR reg.bank 2 step comp.	single reg.bank 2 step comp.	single reg.bank 1 step comp.
Current (mA)	280±10	390±10	450±10	440±10	430±10

Table 8.2: Current consumption

8.2 SEU Simulation Results

All implemented CPU versions have been tested for SEU susceptibility with SEU simulation as described in chapter 7. Single errors have been injected into the bitstream to flip single configuration bits within the FPGA. The reaction of the CPU on this configuration change has been monitored via the CPU's UART output. As described above, the number of configuration bits is too large to test a design against configuration upsets in every bit within a reasonable period of time. Thus, statistics of the SEU susceptibility have been created with random error injection. Any tested version of the CPU has been placed in the same area on the chip to take their different resource usage into account.

8.2.1 The Non-Hardened CPU



(a) Percentage of functional errors

(b) Distribution of functional errors

Figure 8.1: Number of functional errors and their distribution for the non-hardened CPU.

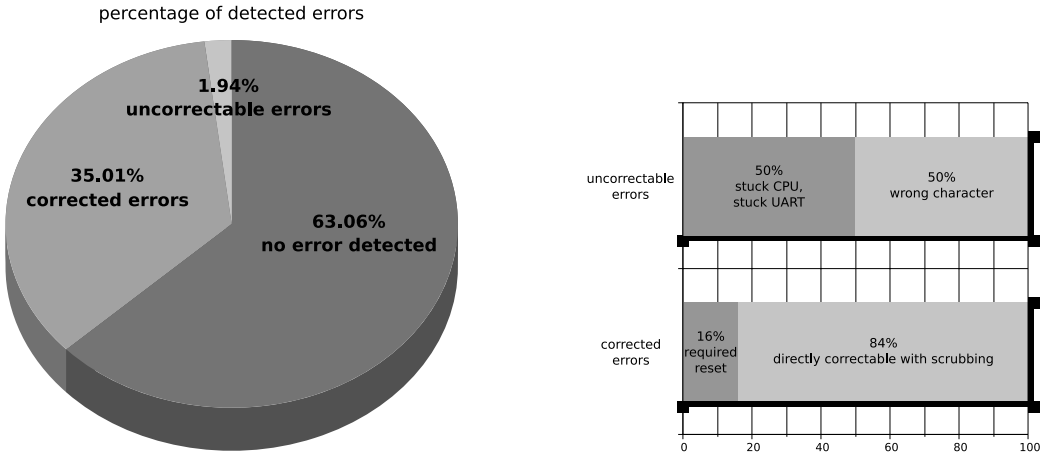
The non-hardened version of the CPU was tested with 129.784 flipped configuration bits, one flipped bit a time. 20,079 of the flipped bits lead to a functional error of the CPU. This equates 15.47% of the total number of flipped bits. The functional errors have been distinguished into *no UART output*, *wrong character* and *correct character, but in a wrong order*. 50% of the functional errors have shown up with a stopped UART output. This is most likely a jump to a wrong piece of code or a completely stuck CPU. 43.55% of the functional errors have been recorded to send a wrong character. The remaining 6.38% of functional errors could be identified as suitable characters, but in a wrong order compared to the previous characters. This may be a faulty conditional branch, for example. A graphical representation of the total number of functional errors and their distribution is shown in figure 8.1.

8.2.2 The Fault Tolerant Implementations

The testing scheme for the fault tolerant implementations was the same as for the non-hardened CPU, however, with changed error conditions. A stuck CPU is not an error anymore, if it can be recovered with scrubbing. No differentiation between wrong character and wrong character order has been made, both cases have been covered with the wrong character error. Three different implementations of the fault tolerant CPU have been tested with SEU simulation:

- DMR register bank, two step comparison
- single register bank, two step comparison
- single register bank, single step comparison

DMR Register Bank, Two Step Comparison



(a) percentage of detected errors

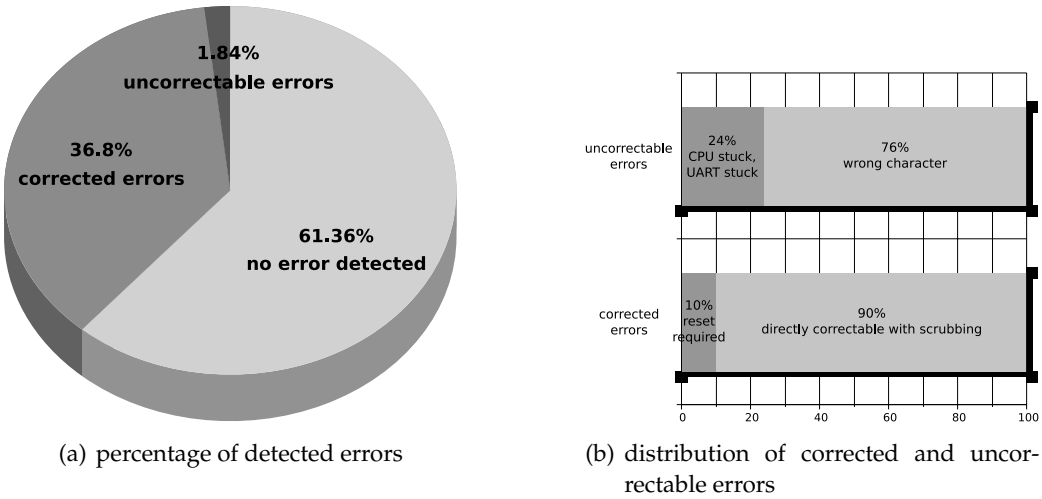
(b) distribution of the corrected errors

Figure 8.2: SEU simulation result for the DMR register bank implementation in the fault tolerant CPU

The DMR register bank version was the first implemented version of the fault tolerant CPU. It has been tested with 48,903 single configuration bit flips. A total of 36.94% of all flipped bits have had an impact on the running CPU. 35.01% of all flipped bits have led to detected errors that could be recovered with scrubbing. 1.94% of all flipped bits have led to uncorrectable errors, either with wrong output, no output or with unrecoverable error conditions. These numbers are shown in figure 8.2. The number of corrected errors can further be distinguished into the number of upsets directly correctable with scrubbing and the number of errors successfully triggering an internal CPU reset. 16% of all correctable errors had to be corrected with an automated reset, the remaining 84% have

directly been correctable with scrubbing. The uncorrectable errors have consisted of 50% stuck CPU or UART and 50% wrong UART output.

Single Register Bank, Two Step Comparison



(a) percentage of detected errors

(b) distribution of corrected and uncorrectable errors

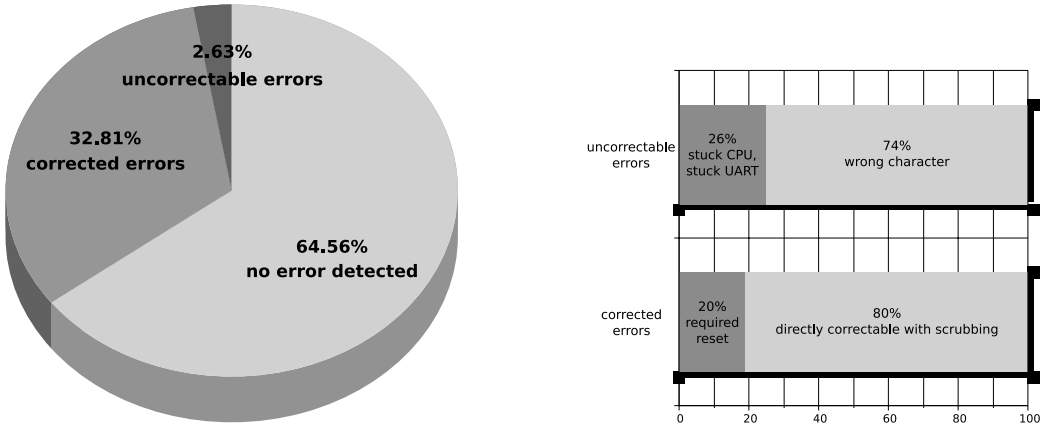
Figure 8.3: SEU simulation results for the single register bank implementation

A second implementation with one common register bank for both pipelines has been tested in the same way. 86,780 configuration bits have been flipped during the SEU simulation of this implementation. The results are shown in figure 8.3. In this version, only 38.64% of all flipped bits had an impact on the running CPU. 36.80% led to errors correctable with scrubbing and 1.84% could have been identified as uncorrectable errors. The correctable errors have consisted of 90% errors directly correctable with scrubbing and 10% errors requiring an automated CPU reset. The uncorrectable errors have compromised 76% wrong characters and only 24% of stuck CPU or UART.

Single Register Bank, Single Step Comparison

In this version only the results from MW-stage have been used to detect errors between both pipeline instances. Signals from the early pipeline stages have not been compared here, however, comparison of the Wishbone Hamming bits and the pipeline control signals have remained unchanged. 53,062 flipped configuration bits have been injected into this implementation. The overall susceptibility to configuration changes has been measured to 35.44%, where 32.81% of the upsets could have been corrected with scrubbing and 2.63% have led to uncorrectable errors. 20.0% of the correctable errors have required

a CPU reset, thus 80.0% could have directly been corrected with scrubbing. The distribution of uncorrectable errors has been similar to the other single register bank implementation. These results are also shown in figure 8.4.



(a) percentage of detected errors

(b) distribution of corrected and uncorrectable errors

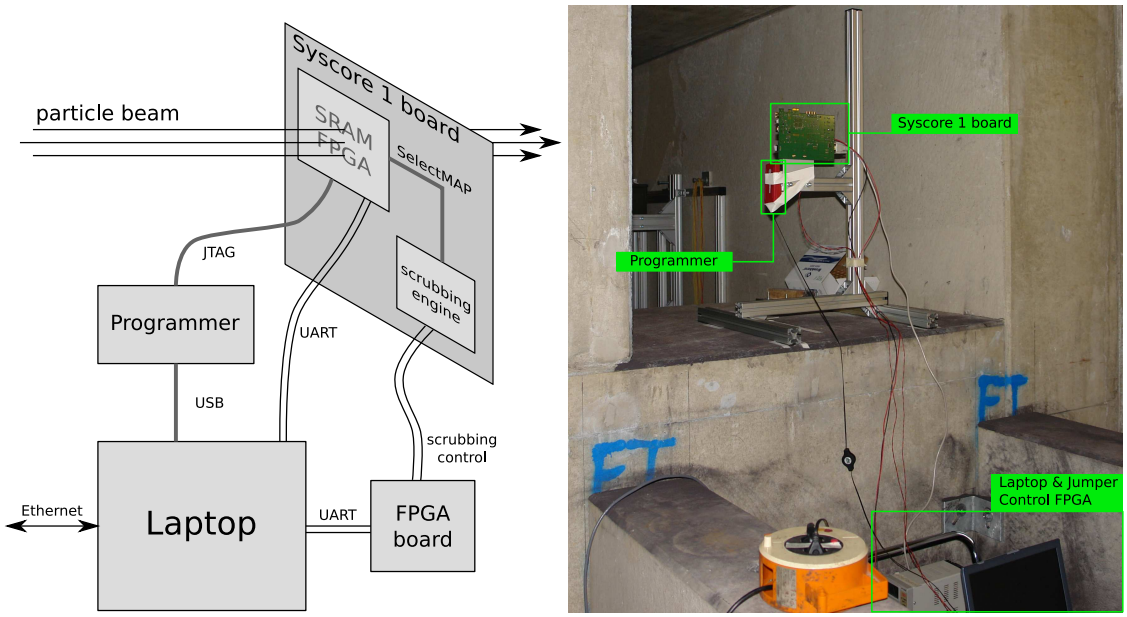
Figure 8.4: SEU simulation results for the single step comparison implementation

8.2.3 Summary

All tested fault tolerant versions of the CPU have shown an significantly increased resistance against spontaneous configuration changes compared to the non-hardened implementation, however their overall percentage of SEU susceptible bits increased with their resource usage.

The most promising implementation has been found in the fault tolerant version with one common register bank for both pipelines and the two step comparison technique. The probability for uncorrectable errors could have been reduced from 15.5% for the non-hardened CPU down to 1.84% for this fault tolerant implementation. This means an increase of factor 8.4. Furthermore, the uncorrectable errors in this implementation have mainly consisted of wrong output characters. These errors are most likely based on undetected register bank errors and are good candidates to be mitigated with fault tolerant techniques on software level.

The implementation with two register banks has given comparable error rates, but does not fit well the needs for a future fault tolerant multilayer system. Correcting differences between both register banks reliably is an extensive task in hardware and impossible for software. The single step comparison implementation has given the best result regarding resource usage, but has not delivered as good error rates in SEU simulation as the other two versions have done.



(a) Sketch (b) Photo (Source: J. Gebelein)

Figure 8.5: Sketch and photo of the beam test arrangement.

8.3 Beamtime Results

A non-hardened version and a fault tolerant implementation of the CPU have been tested in a particle beam experiment at GSI Darmstadt. A Syscore 1 board has been placed directly into the beam, around 15 meters behind the last accelerator magnet. A programming cable was connected to the Xilinx SRAM FPGA and to a laptop. The jumpers controlling the scrubbing process have been managed by using a second FPGA, also connected to the laptop. A sketch of the assembly is shown in figure 8.5. The laptop and the FPGA controlling the jumpers have been placed out of the beam. The laptop has been accessible via Ethernet and has thus been able control both, the SRAM FPGA configuration and the scrubbing process. As the Syscore board has been placed fix in this position, the contents of the flash bitfile memories could have only been changed during beam breaks.

The beam particles have consisted of Ruthenium-96 ions with an energy of 1.69 GeV/u. These particles have almost completely been ionized, 42 of the 44 electrons have been stripped off. According to the calculations described in chapter 3.1.3, this complies with a linear energy transfer of approximately $3.3 \frac{MeV\ cm^2}{mg}$. In order to derive an SEU cross section, the value can be compared with the previous measurements of cross section versus LET. The Weibull fit for Virtex-4 FPGAs is shown in figure 8.6. A LET of $3.3 \frac{MeV\ cm^2}{mg}$ corresponds to an SEU cross section of approximately $(2 \ldots 4) \cdot 10^{-9} \frac{cm^2}{bit}$. The Virtex-4

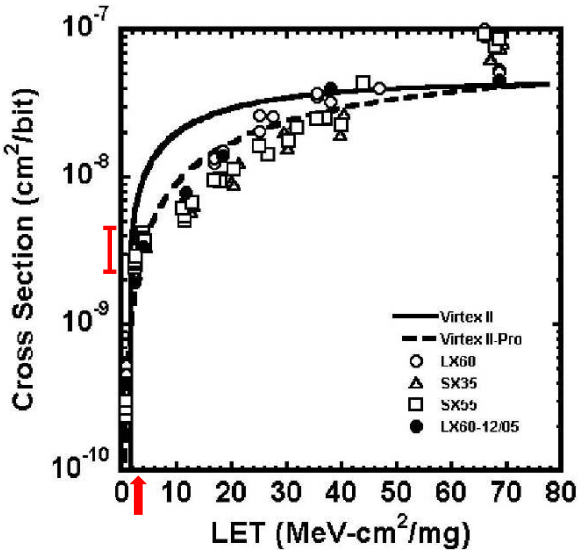


Figure 8.6: SEU cross section vs. LET for Virtex-4 FPGAs taken from [GKS+06]. The LET value for the concerning beam test is highlighted.

FX20 FPGA has around 7,200,000 configuration bits and covers an area of approximately one square centimeter. By multiplying the cross section by the number of configuration bits and dividing by the FPGA area, the number of SEUs per incident particle results in $0.014 \ldots 0.029$. Therefore the probability for a single particle of this beam to produce an SEU in the FPGA is around one to three percent. The beam particles have arrived in *spills*, one spill has taken around 15 seconds and has included $5 \cdot 10^6$ ions. At the location of the FPGA, this beam has been expanded to an area of 10 to 30 cm^2. The number of ions going through the FPGA has therefore been in the order of 10^5 per spill. In combination with the SEU cross section, this results in an expected SEU ratio of approximately $2,400 \ldots 14,500$ upsets per spill. Unfortunately, the incident particle distribution in time during a spill has not been known.

The time of a single scrubbing cycle has been in the order of one second and the expected upset rate of the particle beam has been in the order of $10^3 \cdots 10^4$ per 15 seconds. By recalling that the double modular redundancy implementation is constructed to detect and mitigate single errors, this beam test cannot give results directly comparable to the SEU simulation above. This upset rate has been way to high to be eased by the implemented fault tolerance techniques, because a significant probability has been given to alter both instances of a double modular redundancy system. A reliable error detection has thus not been possible anymore.

Both, the fault tolerant CPU in combination with scrubbing and the non-hardened version without scrubbing have been tested in the beam. Regrettably, only the fault tolerant version with two instances of the register bank has been ready to be tested at this time, so no comparison between different fault tolerant implementations has been achieved. The same test program as for the SEU simulation has been used in the CPU: calculation of Fibonacci series, shift operations and multiply / divide operations. The UART output

has been monitored to record the current status and to detect errors.

The non-hardened version has been uploaded into the FPGA using the programming cable and time has been measured until the CPU has produced wrong results or got stuck. A collection of data by programming the FPGA 2894 times has given an average runtime of 1.2 seconds. However, there have been a lot of cases where the CPU did not send a single character before the whole system got corrupted by SEUs.

The fault tolerant CPU has been run in combination with scrubbing, so the Actel flash FPGA has been responsible for initial configuration and continuous reconfiguration via SelectMAP. The operation mode of the Actel FPGA has been controlled via the jumper lines connected to the second FPGA and could have been set from the laptop via a second UART connection. By sending the according signals to this jumper control FPGA, the initial configuration and the scrubbing could have been triggered. A measurement has been started by initiating a full configuration followed by scrubbing. The CPU's output has been monitored to get the current software state and CPU status. In this case, runtime has been defined as the time, the CPU "does not do anything wrong". This tolerates a stuck CPU waiting for reconfiguration as well as the automated resets, but does not allow a stuck or wrong UART output. The measurement has also been canceled, if the UART output did not change anymore. These measurement conditions has given an average runtime of 15.7 seconds over a sample size of 4930 measurements. This time indicates how long the CPU has not done anything wrong and has included a lot of automated resets. It does not necessarily indicate the progress of the running software.

The absolute SEU rate is not the limiting factor regarding the fault tolerance of the radiated FPGA design. More important is the interaction of scrubbing frequency and SEU frequency. As long as only few configuration bits are upset during the same scrubbing cycle, the DMR approach will be able to handle them. With a scrubbing frequency in the order of one second, this has not been possible in this beam experiment. There have been several measurements, where the CPU did not output a single character or got stuck from the first second. However, there have been some examples where scrubbing could directly correct detected errors or the automated reset could recover a stuck CPU.

Some positive examples of error mitigation techniques showing their effectiveness during the beam test are listed in figure 8.7. The example in the left column shows the application of the automated internal CPU reset. The CPU detects an error during the first second of its runtime and waits for this error to be corrected. The error is uncorrectable with scrubbing, but after three scrubbing cycles, the CPU resets itself. After the reset, the CPU was able to continue its work for further two seconds until the next error was detected. The example in the right column shows the co-operation of scrubbing and error detection. After the initial programming, the CPU was able to run four seconds before an error was detected. The following scrubbing cycle could correct this error within the same second, so the CPU could continue. One second later, the CPU got stuck again due

```
--initial programming
03-18 14:30:51  everything fine
03-18 14:30:51  pipeline error
03-18 14:30:51  pc_mem does not change
--refresh
--refresh
--refresh
03-18 14:30:57  pipeline error
03-18 14:30:57  CPU reset(29)
03-18 14:30:57  everything fine
--refresh
03-18 14:30:59  pc_mem does not change
03-18 14:31:01  pipeline error
...
```

```
--initial programming
03-18 14:31:47  everything fine
03-18 14:31:51  pipeline error
03-18 14:31:51  pc_mem does not change
03-18 14:31:51  pipeline error
--refresh
03-18 14:31:51  everything fine
03-18 14:31:52  pipeline error
03-18 14:31:52  pc_mem does not change
03-18 14:31:53  pipeline error
--refresh
03-18 14:31:53  everything fine
03-18 14:31:58  pipeline error
03-18 14:31:58  pc_mem does not change
03-18 14:31:59  pipeline error
--refresh
03-18 14:31:59  everything fine
03-18 14:32:05  pipeline error
...
```

Figure 8.7: Two examples of log files showing the effectiveness of the applied SEE mitigation techniques during the beam test. The left column shows how the fault tolerant CPU resets itself on uncorrectable errors. The right column shows the successful co-operation of scrubbing and error detection.

to detected errors. Repairing this error with scrubbing again made the CPU to run another five seconds before the next error was detected. A third error could be corrected few seconds later.

In order two compare the results of the non-hardened and the fault tolerant CPU, it has to be noted that the measurement on the non-hardened version has returned the time the CPU operated correctly, whereas the measurement on the fault tolerant version has returned the time the CPU did not do anything wrong. This is a difference in this implementation.

During the whole beam time experiment, a total number of $4.21 \cdot 10^{11}$ Ru-96 Ions per cm^2 have passed through the experiment. The expansion of the beam profile at the FPGA board has been estimated to be 10 to 30 cm^2. This results in approximately $(1.4 \ldots 4.2) \cdot 10^{10}$ ions through the FPGA. The total ionizing dose deposited within the FPGA during the whole beam time has been calculated as described in chapter 3.2.1 to be between 740 krad and 2.2 Mrad. The imprecisely flux numbers due to the estimated beam profile area do not allow a more accurate specification. Unfortunately, the current consumption of the FPGA has not been monitored during the beam test, so a comparison to the TID measurement from Xilinx can hardly be conducted. The only information on TID resistance in this case is that the FPGA has not shown any malfunctions after the beam test and is still completely usable.

The measurements have also shown that the configuration interfaces are susceptible to

radiation induced malfunctions. Several cases could have been noticed, where the Actel FPGA has not been able to program the Xilinx FPGA. This could mostly been recovered by uploading an arbitrary bitfile from the laptop via JTAG. The JTAG configuration interface has shown some errors, too. The device ID of the FPGA could temporary not be read correctly, but could be corrected with several attempts.

A lot of problems have occurred by trying to change the content of the flash bitfile memories during a beam break. The transfer from the UART over both FPGAs into the memories has been extremely slow, but has seemed to work in principle. The reason for this has not been determined yet, because the whole Actel design is currently being rewritten.

In further particle beam tests, the monitoring of FPGA current consumption should be considered in order to see the effects of TID and prevent a possible damage of the device. Furthermore, a more accurate measurement of particle flux and beam profile would help to determine the expected error rates and doses more reliably. With a fixed particle flux per area, the upset rate within the FPGA could be reduced by angling the device in the beam. This would result in a reduced effective FPGA area and thus in a reduced number of upsets. However, too flat angles can increase the probability of multi bit upsets. In case of few beam breaks, a mechanism to move the FPGA out of the beam could help to change the flash memories or to prevent damage due to high TID values.

8.4 Outlook

The first steps towards a radiation tolerant FPGA system have been done and gave promising results, however there is still some work to do. The Wishbone state machine is currently not yet hardened. An error here may cut the CPU from its peripherals. Furthermore, there is also the possibility to add a tag to the Wishbone control signals, so the reliability of these signals could be enhanced. This has not been implemented yet. There is currently no floor-planning done to spatially separate redundant modules. This may help to mitigate upsets affecting several parts of the logic.

There is currently no fault tolerance implemented in the peripherals. This has necessarily to be done in order to achieve a fault tolerant system with the existing fault tolerant CPU. An approach using ECC protected block RAMs is currently in development. This ECC approach may be extensible to the DDR-SDRAM. Regarding the development of software or even an operating system for this CPU, a hardware timer will most likely be required. How a hardware timer and thus time dependent software can be implemented into a CPU that is allowed to be stuck for some time has to be considered carefully. A fault tolerant cache improving the performance of accessing the DDR-SDRAM memory is currently in development. A memory management unit may be negligible for simple operating systems.

The tool flow to run custom software on the CPU currently consists of parts of the plasma project and own snippets. The converting between several different file types is quite cumbersome. This may not be sufficient for future applications and should be possible to be realized more efficiently.

Regarding SEU simulation, the SEU susceptibility measurements could be extended to several configuration errors at the same time. However, SEU simulation as it was conducted is very time consuming. In this work, only a random subset of all possible configuration bits could be tested and with multiple upsets, the number of possible upset combinations would explode. A reduced time consumption for simulating both, single and multiple upsets could be reached by using the Actel flash FPGA to directly inject errors via SelectMAP. Programming via JTAG as shown in chapter 7 takes up to three seconds, programming via SelectMAP should be possible within several milliseconds.

9 Conclusion

Previous research has shown that SRAM based electronics are susceptible to radiation induced memory changes and current pulses. In a field programmable gate array, these effects lead to temporary or static modifications of the device configuration or affect directly the implemented user logic. However, there are several techniques to mitigate the effects of radiation in SRAM based FPGA designs.

In this work, a fault tolerant softcore CPU for SRAM based FPGAs has been developed. Apart from the common approach to implement the FPGA logic with triple modular redundancy coming with a large area overhead, this work targets a fault tolerant implementation with lower resource usage. The combination of double modular redundancy to detect errors and continuous FPGA configuration scrubbing to correct upsets is used as base technique to built a fault tolerant system.

This work's CPU is equipped with a five stage pipeline, hardware multiplier and divider, interrupt and exception handling and a Wishbone bus connection. This CPU has been extended with several error mitigation techniques. The whole pipeline has been implemented twice to detect errors via differences between both instances. In order to detect differences reliably, the error detection logic has also been duplicated and has been implemented and tested in different versions. A focus has been set on the denial of faulty data being written back to registers or any part of the memory. In case of detected errors, the CPU flushes the pipeline and starts to re-execute the failed instruction until the error is corrected with scrubbing. A promising implementation of the internal register bank has been found by providing one common register bank with error detection mechanisms for both pipelines. The program counter has proven to be a very sensible part of the CPU, so a triple implementation has been necessary. A Wishbone bus has been used to connect arbitrary peripherals and it has been secured with error signals and error detecting codes on both, address and data lines.

Several fault tolerant implementations and a non-hardened reference implementation with the same functionality have been evaluated regarding single event upset susceptibility, resource usage and power consumption. The resource usage has grown due to the implemented redundancies, but has still been significantly below the area overhead required for triple modular redundancy. The SEU tests have been conducted with both, SEU simulation via error injection and real particle beam experiments. The results from SEU simulation have shown an increased resistance against single configuration upsets

with a factor 8.4 compared to the non-hardened version. The effectiveness of the applied error mitigation methods has also been verified with a heavy ion particle beam experiment. A direct comparison between simulation and beam test results has not been possible because SEU simulation has addressed to single upsets during one scrubbing cycle and the beam experiment has produced hundreds of upsets in the same period.

As a result it can be stated that the applied radiation tolerance techniques could remarkably mitigate radiation induced single event effects. A significantly increased SEU resistance has been measured without the need of implementing any part with triple modular redundancy.

Appendix A

Implemented Instruction Set

The implemented instruction set is a full MIPS I instruction set except unaligned load and store operations (LWL, LWR, SWL, SWR). The MIPS GCC does usually not use these instructions, so this should not be a limitation. As there is no co-processor and no cache, all co-processor instructions except MFC0 and MTC0 and all cache control instructions have been omitted. The implemented instruction set is shown in the table below and is taken from [Rho09] and [KH91].

Opcode	Name	Action
ADDI rt, rs, imm	Add Immediate	rt=rs+imm
ADDIU rt, rs, imm	Add Immediate Unsigned	rt=rs+imm
ADD rd, rs, rt	Add	rd=rs+rt
ADDU rd, rs, rt	Add Unsigned	rd=rs+rt
ANDI rt, rs, imm	And Immediate	rt=rs&imm
AND rd, rs, rt	And	rd=rs&rt
BEQ rs,rt,offset	Branch On Equal	if(rs==rt) pc+=offset*4
BGEZAL rs,offset	Branch On >= 0 And Link	r31=pc; if(rs>=0) pc+=offset*4
BGEZ rs,offset	Branch On >= 0	if(rs>=0) pc+=offset*4
BGTZ rs,offset	Branch On > 0	if(rs>0) pc+=offset*4
BLEZ rs,offset	Branch On <= 0	if(rs<=0) pc+=offset*4
BLTZAL rs,offset	Branch On < 0 And Link	r31=pc; if(rs<0) pc+=offset*4
BLTZ rs,offset	Branch On < 0	if(rs<0) pc+=offset*4
BNE rs,rt,offset	Branch On Not Equal	if(rs!=rt) pc+=offset*4
BREAK	Breakpoint	epc=pc; pc=Exception Handler
DIV rs,rt	Divide	HI=rs%rt; LO=rs/rt
DIVU rs,rt	Divide Unsigned	HI=rs%rt; LO=rs/rt
JALR rs	Jump And Link Register	rd=pc; pc=rs
JAL target	Jump And Link	r31=pc; pc=target<<2
JR rs	Jump Register	pc=rs
J target	Jump	pc=pc_upper | (target<<2)
LB rt,offset(rs)	Load Byte	rt=*(char*)(offset+rs)
LBU rt,offset(rs)	Load Byte Unsigned	rt=*(Uchar*)(offset+rs)
LBU rt,offset(rs)	Load Halfword Unsigned	rt=*(Ushort*)(offset+rs)

Opcode	Name	Action
LH rt,offset(rs)	Load Halfword	rt=*(short*)(offset+rs)
LUI rt, imm	Load Upper Immediate	rt=imm<<16
LW rt,offset(rs)	Load Word	rt=*(int*)(offset+rs)
MFC0 rt,rd	Move From Coprocessor	rt=CPR[0,rd]
MFHI rd	Move From HI	rd=HI
MFLO rd	Move From LO	rd=LO
MTC0 rt,rd	Move To Coprocessor	CPR[0,rd]=rt
MTHI rs	Move To HI	HI=rs
MTLO rs	Move To LO	LO=rs
MULT rs,rt	Multiply	HI,LO=rs*rt
MULTU rs,rt	Multiply Unsigned	HI,LO=rs*rt
NOR rd, rs, rt	Nor	rd=!(rs | rt)
ORI rt, rs, imm	Or Immediate	rt=(rs | imm)
OR rd, rs, rt	Or	rd=(rs | rt)
RFE	Return From Exception	
SB rt,offset(rs)	Store Byte	*(char*)(offset+rs)=rt
SH rt,offset(rs)	Store Halfword	*(short*)(offset+rs)=rt
SLL rd,rt,sa	Shift Left Logical	rd=rt<<sa
SLLV rd,rt,rs	Shift Left Logical Variable	rd=rt<<rs
SLTI rt, rs, imm	Set On Less Than Immediate	rt=rs<imm
SLTIU rt, rs, imm	Set On Less Than Immediate Unsigned	rt=rs<imm
SLT rd, rs, rt	Set On Less Than	rd=rs<rt
SLTU rd,rs,rt	Set On Less Than Unsigned	rd=rs<rt
SRA rd,rt,sa	Shift Right Arithmetic	rd=rt>>sa
SRAV rd,rt,rs	Shift Right Arithmetic Variable	rd=rt>>rs
SRL rd,rt,sa	Shift Right Logical	rd=rt>>sa
SRLV rd,rt,rs	Shift Right Logical Variable	rd=rt>>rs
SUB rd,rs,rt	Subtract	rd=rs-rt
SUBU rd,rs,rt	Subtract Unsigned	rd=rs-rt
SW rt,offset(rs)	Store Word	*(int*)(offset+rs)=rt
SYSCALL	System Call	epc=pc; pc=Exception Handler
XORI rt,rs,imm	Exclusive Or Immediate	rt=rs^imm
XOR rd,rs,rt	Exclusive Or	rd=rs^rt

Bibliography

[Act08] ACTEL CORPORATION, **Oct. 2008**. *RTAX-S/SL RadTolerant FPGAs*. URL `http://www.actel.com/documents/RTAXS%5fDS.pdf`

[Aer09] AEROFLEX GAISLER AB, **Feb. 2009**. *LEON3-FT SPARC V8 Processor Data Sheet and User's Manual*, 1.1.0.7 ed.

[AGM+08] ABEL, N., GRULL, F., MEIER, N., BEYER, A. and KEBSCHULL, U., **Sep. 2008**. *Parallel hardware objects for dynamically partial reconfiguration*. In FPL, pp. 563–566. doi:10.1109/FPL.2008.4630009.

[Alt09] ALTERA CORPORATION, **Mar. 2009**. *Nios II Processor Reference Handbook*.

[AP98] ALFKE, P. and PADOVANI, R., **Sep. 1998**. *Radiation Tolerance of High-Density FPGAs*. MAPLD.

[ASCT07] ALLEN, G., SWIFT, G., CARMICHAEL, C. and TSNG, C., **Apr. 2007**. *Initial Single Event Effects Testing of the Xilinx Virtex-4 Field Programmable Gate Array*. Single Event Effect Symposium, Long Beach, California. Presentation.

[ASM07] ALLEN, G. R., SWIFT, G. M. and MILLER, G., **Jul. 2007**. *Upset Characterization and Test Methodology of the PowerPC405 Hard-Core Processor Embedded in Xilinx Field Programmable Gate Arrays*. In Proc. IEEE Radiation Effects Data Workshop, vol. 0, pp. 167–171. doi:10.1109/REDW.2007.4342559.

[Bak07] BAKER, R. J., **Nov. 2007**. *CMOS Circuit Design, Layout, and Simulation, Revised Second Edition*. Wiley-IEEE Press, 2 ed. ISBN 9780470229415.

[BPP+08] BERG, M., POIVEY, C., PETRICK, D., ESPINOSA, D., LESEA, A., LABEL, K. A., FRIENDLICH, M., KIM, H. and PHAN, A., **Aug. 2008**. *Effectiveness of Internal Versus External SEU Scrubbing MitigationStrategies in a Xilinx FPGA: Design, Test, and Analysis*. IEEE Transactions on Nuclear Science, 55:2259–2266. doi:10.1109/TNS.2008.2001422.

[BQS07] BOLCHINI, C., QUARTA, D. and SANTAMBROGIO, M. D., **Mar. 2007**. *SEU Mitigation for SRAM-Based FPGAs through Dynamic Partial Reconfiguration*. GLSVLSI.

[Bri] BRIGHAM YOUNG UNIVERSITY. *BYU EDIF Tools Home Page*. URL `http://reliability.ee.byu.edu/edif/`

[BT04] BURKE, G. and TAFT, S., **Sep. 2004**. *Fault Tolerant State Machines*. MAPLD.

[BV93] BESSOT, D. and VELAZCO, R., **Sep. 1993**. *Design of SEU-hardened CMOS memory cells: the HIT cell*. In RADECS, pp. 563–570. doi:10.1109/RADECS.1993.316519.

[Car06] CARMICHAEL, C., **Jul. 2006**. *Triple Module Redundancy Design Techniques for Virtex FPGAs*. Xilinx, v1.0.1 ed.

[CNV96] CALIN, T., NICOLAIDIS, M. and VELAZCO, R., **Dec. 1996**. *Upset hardened memory design for submicron CMOS technology*. IEEE Transactions on Nuclear Science, 43(6):2874–2878. doi:10.1109/23.556880.

[CT08] CARMICHAEL, C. and TSENG, C. W., **Mar. 2008**. *Xilinx Application Note 988: Correcting Single Event Upsets in Virtex-4 Platform FPGA Configuration Memory*. Xilinx.

[Dep97] DEPARTMENT OF DEFENCE, USA, **Dec. 1997**. *MIL-STD 883, Method 1019.5: Ionization Radiation (Total Dose) TestProcedure*. Department of Defence, USA.
URL `http://www.medivactech.com/std883inc%5B1%5D.pdf`

[dLKNH+04] DE LIMA KASTENSMIDT, F. G., NEUBERGER, G., HENTSCHKE, R., CARRO, L. and R. REIS, R., **Dec 2004**. *Designing fault-tolerant techniques for SRAM-based FPGAs*. IEEE Design & Test of Computers, 21(6):552–562. doi:10.1109/MDT.2004.85.

[Edm96] EDMONDS, L. D., **Dec. 1996**. *SEU cross sections derived from a diffusion analysis*. IEEE Transactions on Nuclear Science, 43(6):3207–3217. doi:10.1109/23.552719.

[Fac99] FACCIO, F., **Dec. 1999**. *Radiation Effects in the Electronics for CMS*.
URL `http://web.mst.edu/~umrr/cf116.pdf`

[FDLH08] FABULA, J. J., DELONG, J. L., LESEA, A. and HSIEH, W.-L., **2008**. *The Total Ionizing Dose Performance of Deep Submicron CMOS Processes*. In MAPLD. Presentation.

[Fre89] FREEMAN, R., **Sep. 1989**. *Configurable electrical circuit having configurable logic elements and configurable interconnects*. US Patent 4870302.

[GCZ03] GRAHAM, P., CAFFREY, M. and ZIMMERMANN, J., **2003**. *Consequences and Categories of SRAM FPGA Configuration SEUs*. Tech. rep., XILINX Research Labs.

[GKS+06] GEORGE, J., KOGA, R., SWIFT, G., ALLEN, G., CARMICHAEL, C. and TSENG, C. W., **Jul. 2006**. *Single Event Upsets in Xilinx Virtex-4 FPGA Devices*. In Proc. IEEE Radiation Effects Data Workshop, pp. 109–114. doi:10.1109/REDW.2006.295477.

[Hab02] HABINC, S., **Dec. 2002**. *Functional Triple Modular Redundancy (FTMR)*. Gaisler Research, 0.2 ed.

[Her02] HERVEILLE, R., **Sep. 2002**. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. OpenCores Organization, revision b.3 ed.

[HH89] HOROWITZ, P. and HILL, W., **Jul. 1989**. *The Art of Electronics*. Cambridge University Press, 2nd ed. ISBN 0521370957.

[HP96] HENNESSY, J. and PATTERSON, D., **1996**. *Computer Architecture: a Quantitative Approach, 2nd Edition*. Morgan Kaufman. ISBN B001D8LCJG. URL `http://amazon.com/o/ASIN/B001D8LCJG/`

[KH91] KANE, G. and HEINRICH, J., **1991**. *MIPS RISC Architecture (2nd Edition)*. Prentice Hall PTR. ISBN 0135904722.

[Kit04] KITTEL, C., **Nov. 2004**. *Introduction to Solid State Physics*. Wiley, 8 ed. ISBN 9780471415268.

[KSCR05] KASTENSMIDT, F. L., STERPONE, L., CARRO, L. and REORDA, M., **Mar. 2005**. *On the optimal design of triple modular redundancy logic for SRAM-based FPGAs*. In Proc. Design, Automation and Test in Europe, pp. 1290–1295. doi:10.1109/DATE.2005.229.

[KSR+88] KERNS, S. E., SHAFER, B. D., ROCKETT, J., L. R., PRIDMORE, J. S., BERNDT, D. F., VAN VONNO, N. and BARBER, F. E., **Nov. 1988**. *The Design of Radiation-Hardened ICs for Space: a Compendium of Approaches*. IEEE Proceedings, 76(11):1470–1509. doi:10.1109/5.90115.

[LCR03] LIMA, F., CARRO, L. and REIS, R., **2003**. *Reducing pin and area overhead in fault-tolerant FPGA-based designs*. In FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays, pp. 108–117. ACM, New York, NY, USA. ISBN 1-58113-651-X. doi:http://doi.acm.org/10.1145/611817.611834.

[Leo94] LEO, W. R., **Feb. 1994**. *Techniques for Nuclear and Particle Physics Experiments: A How-to Approach*. Springer, 2nd ed. ISBN 9783540572800.

[Lut07] LUTZ, G., **Jul. 2007**. *Semiconductor Radiation Detectors: Device Physics*. Springer. ISBN 9783540716785.

[ME00] MAVIS, D. G. and EATON, P. H., **Dec. 2000**. *SEU and SET Mitigation Techniques for FPGA Circuit and Configuration Bit Storage Design*. MAPLD.

[MK09] MÜLLER-KLIESER, S., **Mar. 2009**. *Entwurf und Implementierung eines adaptiven, strahlentoleranten eingebetteten Systems am Beispiel eines Read-Out-Controllers*. Diploma thesis, Kirchhoff Institute for Physics, University of Heidelberg.

[MM88] MAHMOOD, A. and MCCLUSKEY, E., **1988**. *Concurrent Error Detection Using Watchdog Processors-A Survey*. IEEE Transactions on Computers, 37(2):160–174. ISSN 0018-9340. doi:http://doi.ieeecomputersociety.org/10.1109/12.2145.

[MM00] MITRA, S. and MCCLUSKEY, E. J., **Oct. 2000**. *Which concurrent error detection scheme to choose ?* In Proc. International Test Conference, pp. 985–994. doi:10.1109/TEST.2000.894311.

[MMPW07] MORGAN, K. S., MCMURTREY, D. L., PRATT, B. H. and WIRTHLIN, M. J., **Dec. 2007**. *A Comparison of TMR With Alternative Fault-Tolerant Design Techniques for FPGAs*. IEEE Transactions on Nuclear Science, 54(6):2065–2072. doi:10.1109/TNS.2007.910871.

[NR08] NOTE, J.-B. and RANNAUD, E., **2008**. *From the bitstream to the netlist*. In FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays, pp. 264–264. ACM, New York, NY, USA. ISBN 978-1-59593-934-0. doi:http://doi.acm.org/10.1145/1344671.1344729.

[Pai07] PAINKE, F., **Feb. 2007**. *Hardware-Software Interface and Data Flow of the ALICE HLT*. Diploma thesis, Kirchhoff Institute for Physics, University of Heidelberg.

[PCG+06] PRATT, B., CAFFREY, M., GRAHAM, P., MORGAN, K. and WIRTHLIN, M., **Mar. 2006**. *Improving FPGA Design Robustness with Partial TMR*. In Proc. 44th Annual. IEEE International Reliability Physics Symposium, pp. 226–232. doi:10.1109/RELPHY.2006.251221.

[QGK+05] QUINN, H., GRAHAM, P., KRONE, J., CAFFREY, M., REZGUI, S. and CARMICHAEL, C., **Dec. 2005**. *Radiation-Induced Multi-Bit Upsets in Xilinx SRAM-Based FPGAs*. IEEE Transactions on Nuclear Science, 52:2455–2461.

[QMG+07] QUINN, H., MORGAN, K., GRAHAM, P., KRONE, J., CAFFREY, M. and LUNDGREEN, K., **Dec. 2007**. *Domain Crossing Errors: Limittions on Single Device Triple-Modular Redundancy Circuits in Xilinx FPGAs*. IEEE Transactions on Nuclear Science, 54(6):2037–2043.

[RCV+05] REIS, G. A., CHANG, J., VACHHARAJANI, N., RANGAN, R., AUGUST, D. I. and MUKHERJEE, S. S., **2005**. *Software-controlled fault tolerance*. ACM Trans. Archit. Code Optim., 2(4):366–396. ISSN 1544-3566. doi:http://doi.acm.org/10.1145/1113841.1113843.

[Rho09] RHOADS, S., **2009**. *Plasma - most MIPS I(TM) opcodes*.
URL `http://opencores.org/?do=project&who=mips`

[Roh08] ROHR, D., **Mar. 2008**. *Syscore Board v1.0 Flash Interface*. Kirchhoff Institute for Physics, University of Heidelberg, Internal Labreport.

[RVMR09] REORDA, M. S., VIOLANTE, M., MEINHARDT, C. and REIS, R., **Apr. 2009**. *A Low-Cost SEE Mitigation Solution for Soft-Processors Embedded in Systems On Programmable Chips*. DATE.

[RWCG02] ROLLINS, N., WIRTHLIN, M. J., CAFFREY, M. and GRAHAM, P., **Sep. 2002**.

Reliability of Programmable Input/Output Pins in the Presence of Configuration Upsets. MAPLD.

[RWS+07] REZGUI, S., WANG, J. J., SUN, Y., CRONQUIST, B. and MCCOLLUM, J., **Jun. 2007**. *SET Characterization and Mitigation of ACTEL Flash-Based FPGAs in Heavy Ions and Protons Beams*. Microelectronics Reliability & Qualification Workshop. Presentation.

[Sch96] SCHWANK, J. R., **Jul 1996**. *Space and Military Radiation Effects in Silicon-on-Insulator Devices*. Tech. rep., Sandia National Laboratories, Albuquerque, New Mexico.

[SM04] STURESSON, F. and MATTSSON, S., **Nov. 2004**. *Application-like Radiation Test of XTMR and FTMR Mitigation Techniquesfor Xilinx Virtex-II FPGA*. Tech. rep., Saab Ericsson Space AB.

[SPA] SPARC INTERNATIONAL INC. *The SPARC Architecture Manual Version 8*.

[SRK04] SAMUDRALA, P. K., RAMOS, J. and KATKOORI, S., **Oct. 2004**. *Selective triple Modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs*. IEEE Transactions on Nuclear Science, 51(5):2957–2969. doi: 10.1109/TNS.2004.834955.

[SV06] STERPONE, L. and VIOLANTE, M., **Jun. 2006**. *A new reliability-oriented place and route algorithm for SRAM-based FPGAs*. IEEE Transactions on Computers, 55(6):732–744. doi:10.1109/TC.2006.82.

[SWC+99] SPEERS, T., WANG, J. J., CRONQUIST, B., MCCOLLUM, J., TSENG, H., KATZ, R. and KLEYNER, I., **1999**. *0.25um FLASH Memory Based FPGA for Space Applications*. MAPLD.

[Tam06] TAM, S., **Aug. 2006**. *XAPP645: Single Error Correction and Double Error Detection*. Tech. rep., Xilinx.

[WRCG03] WIRTHLIN, M., ROLLINS, N., CAFFREY, M. and GRAHAM, P., **Sep. 2003**. *Hardness by Design Techniques for Field Programmable Gate Arrays*. MAPLD.

[Wro87] WROBEL, T. F., **Dec. 1987**. *On Heavy Ion Induced Hard-Errors in Dielectric Structures*. IEEE Transactions on Nuclear Science, 34:1262–1268. ISSN 0018-9499. doi:10.1109/TNS.1987.4337463.

[Xila] XILINX. *Radiation Effects & Mitigation Overview*. Presentation.
URL `http://www.xilinx.com/esp/mil%5faero/collateral/presentations/radiation%5feffects.pdf`

[Xilb] XILINX. *Xilinx TMRTool*. Sell Sheet.
URL `http://www.xilinx.com/esp/mil%5faero/collateral/tmrtool%5fsellsheet%5fwr.pdf`

[Xil08a] XILINX, **Aug. 2008**. *MicroBlaze Processor Reference Guide*.

[Xil08b] XILINX, **Dec. 2008**. *Radiation-Tolerant Virtex-4 QPro-V Family Overview v1.2*.

[Xil08c] XILINX, **Apr. 2008**. *Virtex-4 Configuration User Guide v1.10*.

[Xil08d] XILINX, **Sep. 2008**. *Virtex-4 Family Overview v3.0*.

[Xil08e] XILINX, **Jun. 2008**. *Virtex-4 FPGA User Guide v2.5*.

[Xilun] XILINX, **2009 Jun.** *About Xilinx*.
URL `http://www.xilinx.com/company/about.htm`

Erklärung zur selbständigen Verfassung

Ich versichere, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, im Juni 2009

Heiko Engel