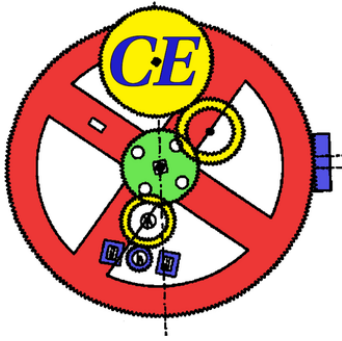




MSc THESIS

A Universal Fault Injector for Structural Simulation Environments

NC Veerman

Abstract



CE-MS-2009-33

Smaller feature size, greater chip density, and minimal power consumption all lead to an increased number of faults in computing systems. Among these faults are soft errors. They are mainly caused by incident radiation. When a particle strikes a semiconductor and deposits enough energy along its path, charge builds up and the transistor temporarily has a different state. If this state is captured by a memory element such as a latch, the system can produce a faulty result. Due to the current technology trends ever decreasing the feature size and minimizing power consumption, reliability is becoming a serious concern in the current and future designs. More and more research is dedicated to fault tolerance of computing systems. One of the problems this research faces is a lack of adequate fault injection tools necessary to perform experimental evaluation of novel fault tolerance techniques. The best experimental setup would consist of a physical hardware placed into an appropriate environment. This is, however, in most cases not feasible due to extremely high cost, and also long time the experiments would require. Moreover, even if the manufacturing of the design is actually planned, the evaluation

should be performed before the manufacturing. Hence, a simulation-based fault injection evaluation is necessary. Nowadays researchers usually implement their own fault injectors on simulation platforms they use for the design evaluation. This sometimes requires very much time and effort. A universal solution is very desirable, which could be easily adapted for different designs. Although it is impossible to create a fault injector compatible with all the existing simulators, there exist structural frameworks such as Unisim [1], that can be used for fast prototyping and are even able to interoperate with already existing simulators. This project focuses on creating a universal fault injector for the Unisim environment, which can be used in Unisim-based designs of different granularity with a minimal effort. It models transient and permanent types of faults, transient meaning single and multiple cycles. It can model single bit and multiple bit faults. It can be used for simulating memory and logic faults. The case studies show how the injector can be used when designing a superscalar processor with fault tolerance features. The fault injector can be implemented in an existing module of a simulator, or a new module can be placed in between existing modules. Different fault tolerance techniques are evaluated and compared using the fault injector.

A Universal Fault Injector for Structural Simulation Environments

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

NC Veerman
born in Amsterdam, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

A Universal Fault Injector for Structural Simulation Environments

by NC Veerman

Abstract

Smaller feature size, greater chip density, and minimal power consumption all lead to an increased number of faults in computing systems. Among these faults are soft errors. They are mainly caused by incident radiation. When a particle strikes a semiconductor and deposits enough energy along its path, charge builds up and the transistor temporarily has a different state. If this state is captured by a memory element such as a latch, the system can produce a faulty result. Due to current technology reliability is becoming a serious concern in current and future designs. More and more research is dedicated to fault tolerance of computing systems. One of the problems this research faces is a lack of adequate fault injection tools necessary to perform experimental evaluation of novel fault tolerance techniques. The best experimental setup would consist of physical hardware placed into an appropriate environment. This is, however, in most cases not feasible due to the extremely high cost, and also the long time the experiments would require. Moreover, even if the manufacturing of the design is actually planned, the evaluation should be performed before the manufacturing. Hence, a simulation-based fault injection evaluation is necessary. Nowadays researchers usually implement their own fault injectors on simulation platforms they use for the design evaluation. This sometimes requires very much time and effort. A universal solution is very desirable, which could be easily adapted to different designs. Although it is impossible to create a fault injector compatible with all the existing simulators, there exist structural frameworks such as Unisim [1], that can be used for fast prototyping and are even able to inter operate with already existing simulators. This project focuses on creating a universal fault injector for the Unisim environment, which can be used in Unisim-based designs of different granularity with a minimal effort. It models transient and permanent types of faults, transient meaning single and multiple cycles. It can model single bit and multiple bit faults. It can be used for simulating memory and logic faults. The case studies show how the injector can be used when designing a superscalar processor with fault tolerance features. The fault injector can be implemented in an existing module of a simulator, or a new module can be placed in between existing modules. Different fault tolerance techniques are evaluated and compared using the fault injector. A novel idea is to use the injector to measure the reduction in soft error rate of a system using CPU speed-up techniques. From simulations it is shown these results can in fact reduce the soft error rate, depending on the size of the memory used for storing the instructions. Furthermore ECC is implemented on a memory and it is shown this technique corrects single bit errors. At a different granularity two whole power PC Processors are used in a duplication scheme. The two processors run the same program, and a detector check the data written to memory. If the data differs a fault is detected.

Laboratory : Computer Engineering
Codenummer : CE-MS-2009-33

Committee Members :

Advisor: Ben Juurlink, CE, TU Delft

Chairperson: Kees Goossens, CE, TU Delft

Member: Demid Borodin, CE, TU Delft

Member: Sorin Cotofana, CE, TU Delft

Member: Nick van der Meijs, CAS, TU Delft

Contents

List of Figures	vii
List of Tables	ix
Acknowledgements	xi
1 Introduction	1
1.1 Problem Description	1
1.2 Work Structure	2
1.3 Thesis Organisation	2
2 Related Work	5
3 Hardware Faults	7
3.1 Transient Faults	7
3.2 Origins of SEU	7
3.3 SEU in Combinational Logic	10
3.4 SEU in SRAM	11
3.5 SEU in DRAM	12
3.6 Multiple Bit Upset	15
3.7 Estimating the SER	15
3.8 Calculating Block SER	16
3.9 Conclusions	18
4 Fault Tolerance Techniques	19
4.1 Circuit-Level Techniques	19
4.2 Hardware Redundancy	20
4.3 Time Redundancy	20
4.4 Conclusions	21
5 Simulation Environment	23
5.1 Unisim Environment	23
5.2 The DLX-Processor	24
5.3 The MIPS-1 ISA Implementation.	27
5.4 The MIPS Superscalar	29
5.5 Conclusions	32
6 Fault Injector	33
6.1 Fault Injector for Combinational Logic	33
6.2 Fault Injector for Memory	35

6.3	Implementation	36
6.4	User Guide	37
6.5	Conclusions	40
7	Case Studies	41
7.1	Benchmarks	41
7.2	Instruction Precomputation	44
7.3	Instruction Memoization	46
7.4	Instruction Duplication with Memoization	49
7.5	Error Correcting Memory	50
7.6	Power PC Duplication	51
7.7	Conclusions	52
8	Conclusions	53
8.1	Summary	53
8.2	Conclusion	54
8.3	Future Work	55
	Bibliography	59

List of Figures

3.1	Particle strike	8
3.2	Linear energy transfer (LET) versus depth curve for 210-MeV chlorine [2] ions in silicon	9
3.3	Charge generation and collection in a reversed biased-junction [3]	10
3.4	Masking of errors[4]	11
3.5	A typical SRAM cell	12
3.6	One transistor DRAM cell[2]	13
3.7	DRAM bit-line strike [5]	14
3.8	Estimated SER of 16×16 multiplier [6]	16
3.9	Combining soft error rates	17
4.1	Comparison of the ion track between NON-SOI(a) and SOI(b)	19
4.2	Concurrent error detection scheme [7]	21
5.1	The DLX pipeline [8]	25
5.2	The DLX pipeline with distributed control [8]	26
5.3	The MIPS simulator [9]	28
5.4	The superscalar MIPS simulator	30
5.5	The instruction fetch module	31
5.6	The instruction decode module	31
5.7	The ALU/memory address calculation module	32
6.1	The superscalar MIPS simulator with fault injectors	34
6.2	The memory fault injector	35
7.1	Instruction precomputation [10]	44
7.2	Superscalar MIPS with precomputation and fault injector	45
7.3	Program flow of the compound ALU	45
7.4	Superscalar MIPS with memoization and fault injector	47
7.5	Memoization	47
7.6	Instruction duplication with precomputation [10]	50
7.7	Duplication of Power PC (PPC)	52

List of Tables

3.1	Assumed technology parameters	16
4.1	Triple redundancy	20
7.1	Number of soft errors using precomputation	46
7.2	Number of soft errors using memoization, with a table of 4 entries	48
7.3	Number of soft errors using memoization, with a table of 8 entries	48
7.4	Number of soft errors using memoization, with a table of 12 entries	48
7.5	Number of soft errors using memoization, with a table of 16 entries	49
7.6	Number of extra clock cycles needed by duplication and memoization . .	49
7.7	Simulation of long lasting soft errors	50
7.8	Number of soft errors using precomputation	51
7.9	PPC duplication	51

Acknowledgements

First I would like to thank Ben Juurlink, the responsible professor at TU Delft. I would like to thank him for his advise and guidance during the project.

I also would like to very much thank Demid Borodin for his advice, and the input and ideas he gave. I enjoyed the discussion we had during the project.

I would like to thank my parents and girlfriend for their support during the project.

NC Veerman
Delft, The Netherlands
september 17, 2009

Introduction

1.1 Problem Description

Smaller feature size, greater chip density, and minimal power consumption all lead to an increased number of faults in computing systems. For designs manufactured at advanced technology's, such as 90 nm, 65 nm and on wards, system level soft errors are much more frequent than in the previous generations. An error is a result of an erroneous hardware block. A soft error is such an error, except the hardware block is temporarily producing an erroneous result.

Soft errors, also called single event upsets, are mostly radiation-induced transient errors. They are caused by particle strikes hitting a sensitive area of a logic element (e.g. a memory bit, the input of an AND port). Crosstalk can also be the cause of a soft error, only the probability is much smaller and there is a wide area of possible fixes. The most common fixes are increased spacing, wire re-ordering, and shielding.

The susceptibility of a device to soft errors caused by radiation can be lowered by radiation hardening. This involves increasing the capacitance of a node so the range of particles with energy high enough to cause an upset is reduced. Designers can also choose to accept soft errors will occur and design systems with appropriate error detection and correction. Nowadays, Error-Correcting Code (ECC) memory is commonly used in server systems[11]. In logic circuits, the concept of triple modular redundancy[12] is well known. This technique uses majority voting to produce correct results.

Smaller feature size leads to an increased number of faults in processors. Since radiation hardening techniques also suffer from this phenomena, fault tolerance through added hardware is becoming more important. To be able to investigate architectural techniques for fault tolerance, a software tool that injects faults into a processor simulator is needed.

The goals of this project are (1) to inject accurately modeled faults in a processor simulator, (2) implement fault tolerance techniques, and (3) investigate the improvement in soft error rate of these techniques using the fault injector.

The fault injector should be universal, it should be able to inject faults into different parts of the processor. Since different fault tolerance techniques target different faults, the fault injector should be able to inject different kind of faults. Most faults are single transient faults, meaning a single bit is affected and the fault has disappeared before the next clock-cycle. However there are cases where the fault lasts multiple cycles. So the injector should have the possibility to specify the length of the fault to be injected. Some faults may affect multiple bits. If, for example, a particle strikes two adjacent paths in a processor multiple bits are affected. The proposed injector can be set to inject a fault in multiple bits.

Faults in memories in contrast to logic faults need to be remembered by the simulator.

If a fault in a memory location occurs, the erroneous value will remain in memory until it is overwritten by a new value. The injector should have the possibility to remember the faults and remove them when the value in the memory address is overwritten.

When researchers investigate new fault tolerance techniques, strange phenomena could happen when injecting a fault. Most simulators cannot repeat the same fault injection sequence. In the proposed fault injector it is possible to repeat a fault injection sequence, to be able to fully investigate the issues concerning the phenomena.

1.2 Work Structure

To achieve the given objectives, the following steps have been taken in succession:

1. Investigate causes and manifestations of soft errors. Soft errors are transient errors. The most common cause is radiation. Radiation causes a particle strike, if this strike is in a sensitive region of a circuit a soft error can occur. Not all particle strikes in sensitive regions are propagated through the circuit. Logical, electrical and temporal masking can occur. Soft errors in logic circuits can be accurately modeled by a transient bit flip, in memory elements the error is preserved until it is overwritten.
2. Inject faults into a processor simulator. The Unisim environment [1] is chosen because it is a structural simulation environment. A simulator in Unisim can be decomposed into several modules. A fault class has been made which can inject faults into a module.
3. A MIPS processor has been implemented in Unisim. This implementation has enough granularity to fully take advantage of the fault injector class. The processors in the library of Unisim do not.
4. Fault tolerance techniques have been investigated. There are schemes using hardware or time redundancy and production technology techniques.
5. Memoization, precomputation and duplication in combination with memoization or precomputation have been implemented. An Error Correcting Code(ECC) for memories is implemented. Duplication of a whole processor from the Unisim library is implemented. Faults are injected using the proposed injector and the improvement in soft error rate of the system is estimated and measured through simulation.

1.3 Thesis Organisation

This report is organised as follows:

- Chapter 2 presents related work.
- Chapter 3 describes the common causes for soft errors.

-
- Chapter 4 contains current techniques for reducing soft error susceptibility.
 - Chapter 5 describes the simulation environment.
 - Chapter 6 the proposed fault injector is introduced.
 - Chapter 7 contains a case study of memoization, precomputation, duplication and fault detecting memory. The improvements of the soft error rates for this techniques are calculated through experiments.
 - Chapter 8 draws conclusions of this work.

Related Work

There has been done much work in the area of fault injection. One approach has been to inject physical faults into the target system hardware. Usually a (proton)source is used to inject charged particles into certain areas of a processor [13] [14]. The benefit of this approach is the fact that the injected particles cause real hardware faults. However it is difficult to control the place where the particle injects faults, what kind of faults and how many faults a particle causes. Furthermore specialised hardware is needed not only to inject the particle, but also to acknowledge if a fault occurred. Usually extra hardware is needed to be able to know the state the processor is in. Extra pins are usually used in this case. Because of the high speed of modern processors, high speed monitoring hardware is also needed.

Software fault injection(SWIFI) is a method which relies on the software to inject faults. Ferrari [15] is a tool that modifies the control structure of a process, that in turn alters the execution state of the target program. Two processes are run concurrently, one being the fault injector and one being the target program. The Ferrari tool is also responsible for collecting the data. Xception [16] uses advanced debugging and performance monitoring features existing in modern processors. By directly programming the debugging hardware inside the target processor, Xception can inject faults with minimum interference with the target application. Doctor [17] is a SWIFI which can inject faults into a system, into the processor, memory and communication. The fault injector manipulates the data on a bus for communication faults. Processor faults are emulated by changing or adding an instruction at the compilation of a program. Memory faults are handled by a layer on top of the memory controller.

Software fault injection has as advantage that it can be used on physical processors. However, the fault models are usually not very realistic. The program has to be altered, so the source code of the programs to be run need to be known in advance. Furthermore it is not portable since the system has to be very well known. Furthermore extra software or hardware is needed to be able to collect the statistics of the fault injected runs. The proposed fault injector has the advantage that it is targeted at structural simulation environment. Because of the modular structure the fault injector can be easily placed in between different modules of the simulator. Furthermore the techniques described above are not really suited for fault tolerance research. To be able to test the fault tolerance first a whole processor needs to be developed. With the proposed injector the fault tolerance technique can be investigated very early in the design stage. Injecting faults by means of adding instructions at compilation time means you have to adapt the compiler or manually add instruction to the assembly code. Either one is a very time consuming job.

Processors simulated or designed using hardware description languages such as VHDL and Verilog can be easily altered to induce faults. Since there are logical values trans-

ferred between components, one can easily alter these values. MEFISTO[18] is a tool which can automatically inject faults into VHDL-models at gate-level, register-level and chip-level. Advantages of this technique are the controllability and observability, and the fact it can be used at different abstraction levels. Furthermore it can be used during the design phase of the processor. A drawback is the simulation speed, which is very low compared to SystemC for example.

SPICE (Simulation Program with Integrated Circuit Emphasis) [19] is a general-purpose analog electronic circuit simulator. In spice models very realistic faults can be modelled. A transient fault can be modelled as a voltage spike. A drawback, however, is that it cannot be used to simulate a whole processors. Implementing a whole processor in spice is a huge amount of work and the simulation time would be infeasible.

Many researchers use processor simulators and adapt these to perform fault injection experiments. However this is a very time consuming work, which often takes more time than implementing the fault tolerance technique. Often the collecting of data is very complicated, due to the interwoven character of processor simulators. For example, in [10] the SimpleScalar [20] simulator is adapted to inject simple faults into the arithmetic logic unit. The proposed injector would not only save programming time, but also could inject more sophisticated faults such as faults with a duration of multiple clock cycles.

Concluding the proposed fault injector clearly has advantage in the portability and reusability. If a processor is implemented in a structural simulation environment with enough granularity, the proposed fault injector could easily be placed in different places of the processor simulator. Another key advantage over the techniques described above is the fact that fault patterns can be replicated. A disadvantage compared to fault injectors targeted at live systems is the simulation speed.

Hardware Faults

Hardware faults that arise during the system operation are best qualified by their duration, there are permanent or long-term faults and transient faults. A permanent fault is caused for example by a short circuit caused by thermal ageing, improper manufacture or misuse. A permanent fault can only be removed by means of replacement. Transient faults are triggered by environmental conditions such as radiation. Transient faults rarely do any long lasting damage to the component, but they can produce an erroneous state for a short period of time. Studies have shown transient faults occur far more often than permanent ones and are much harder to detect.

3.1 Transient Faults

Transient fault have many names. In literature they are also known as single event effects or soft errors, they are caused by single event upsets(SEU). As defined by the JEDEC standard JESD89A single event effects (SEE) indicate any measurable or observable change in state or performance of a microelectronic device, component, subsystem or system resulting from a single energetic particle strike. A soft error is defined as an erroneous output signal from a latch or memory cell that can be corrected by performing one or more normal functions of the device containing the latch or memory cell. High speed circuits are becoming more vulnerable to single-event effects, including multiple-bit upsets. These are the result of scaling and thus higher clock frequencies and lower operating voltages. In the next section these complications are explored further.

3.2 Origins of SEU

There are two primary methods by which ionising radiation releases charge in a semiconductor device: direct ionisation by the incident particle itself and ionisation by secondary particles created by nuclear reactions between the incident particle and the struck device. Both these mechanisms can lead to semiconductor malfunction.

When an energetic charged particle passes through a semiconductor material it frees electron-hole pairs along its path as it loses energy. When a particle has lost all it's energy it comes to rest in the semiconductor. The total path length is called range. The LINEAR Energy Transfer(LET) is a measure of the energy transferred while the particle travels through. LET is defined by

$$L_{\Delta} = \frac{dE_{\Delta}}{dx} \quad (3.1)$$

This typically is expressed in units of $MeV \cdot cm^2/mg$. this is a combination of the energy lost by the particle per unit path length MeV/cm divided by the density of the

material mg/cm^3 .

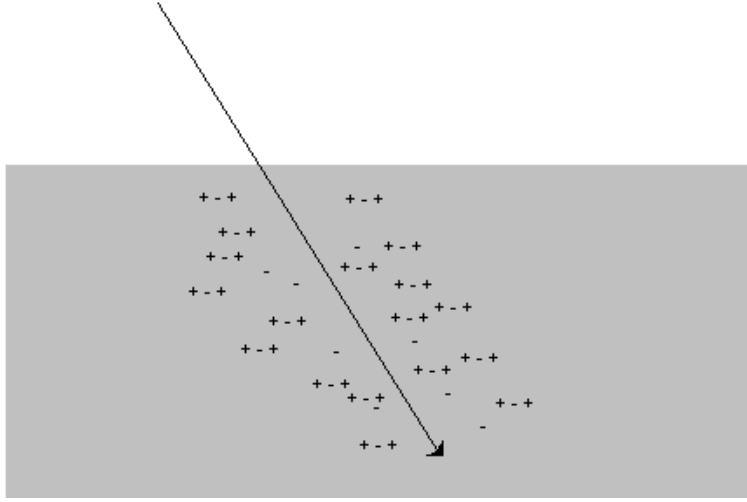


Figure 3.1: Particle strike

From the LET the charge deposition can be easily calculated. In Silicon a LET of $97 mg/cm^3$ corresponds to a charge deposition of $1 pC/\mu m$. In figure 3.2 the relation between the energy transfer and the depth the particle has travelled is depicted. When a single particle strikes a semiconductor and doesn't collide with other particles, it is called direct ionisation. Direct ionisation is the primary charge deposition mechanism for upsets caused by heavy particles. However in [21] the direct ionisation of protons, a light particle, is investigated and reported.

Direct ionisation by light particles usually do not deposit enough energy to build up enough charge to cause single event upsets. However this doesn't mean these particles can be neglected. The lighter particles, protons and neutrons, can produce upsets due to indirect ionisation. As a high energy light particle travels into the semiconductor, it may have an inelastic collision with a nucleus. Inelastic meaning there is loss of kinetic energy. Possible reactions from this collision are other collisions that produce Silicon recoils, the recoil of a daughter nucleus causing the emission of alpha or gamma particles, and spallation reactions. Spallation is the process in which a heavy nucleus emits a large number of nucleons as a result of being hit by a high-energy particle.

Any of these reactions can now deposit energy along their paths by direct ionisation. These particles are much heavier than the incident particle, proton or neutron, and they deposit higher charge densities as they travel. This means they may be cause an upset along their path.

Inelastic collision products typically have fairly low energies and do not travel far from the collision site. Reaction products tend to be forward-scattered in the direction of the original particle. This means the SEU sensitivity is also a function of the angle of incidence. If a nuclear reaction due to the collision has occurred, the charge deposition by the product particles is the same as for a directly ionising strike by a heavy ion.

The most sensitive regions of a microelectronic device are usually reverse-biased p/n

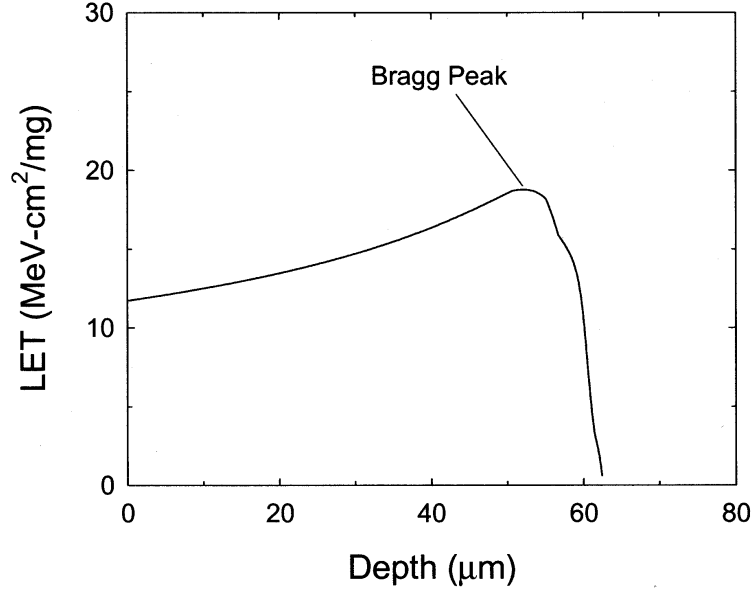


Figure 3.2: Linear energy transfer (LET) versus depth curve for 210-MeV chlorine [2] ions in silicon

junctions. In a reverse-biased junction depletion region the high field strength causes drift processes, which collect the charge a travelling particle deposits. This in turn leads to a transient current at the junction contact. Strikes near a depletion region can also result in significant transient currents as carriers diffuse into the vicinity of the depletion region and thus can also be efficiently collected.

Shortly after the discovery of SEU in the late 1970's, researchers at IBM used numerical device simulators to compute the response of reverse-biased pn junctions to alpha-particle strikes [22] [23]. These simulations showed the existence of a transient disturbance in the junction electrostatic potential, this was named the field funnel. Charge generated along the particle track can locally collapse the junction electric field. The funnelling effect can increase charge collection at the struck node by extending the junction electric field away from the junction and deep into the substrate. Because of this effect the charge deposited some distance from the junction can be collected through the drift process. As can be seen in figure 3.3.

Newer studies have shown that another charge-collection mechanism may exist in case of sub-micron MOS transistors. These mechanisms require considering the whole transistor [24] reference. This mechanism was revealed by 3-D alpha particle simulation and has been verified through experiments. The effect is called alpha-particle source-drain penetration effect (ALPEN). This effect comes into play when a particle passes through both the source and the drain. Immediately following the impact, the electrostatic potential in the channel region is disturbed. This leads to a significant but short source-drain conduction current that mimics the "on" state of the transistor.

Another mechanism exists when electrons or holes released by a particle strike are confined to a well or body region. For example, an n-channel p-well MOS transistor,

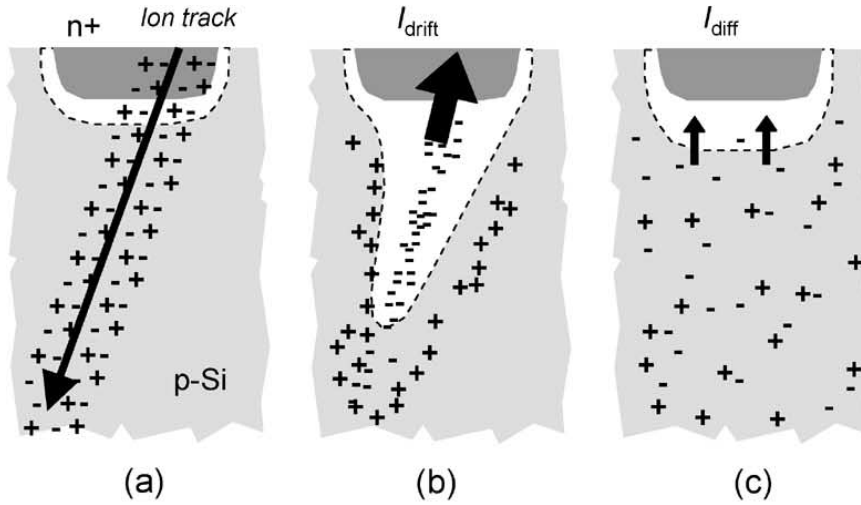


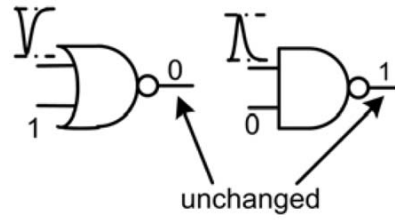
Figure 3.3: Charge generation and collection in a reversed biased-junction [3]

electrons induced by a particle strike can be collected at the drain/well junction or the well/substrate junction. Holes left in the well however increase the well potential, thus lowering the source/well potential barrier, and the source injects electrons into the channel. These electrons can be collected by the drain and can add to the particle-strike induced current, thus increasing the probability of an upset. This is known as the bipolar transistor effect, since the electrons are injected over the sourcewell barrier and the source acts as the emitter, the channel as the base region and the drain as the collector. By reduction of the channel length, the "base" width is decreased and the bipolar effect becomes more pronounced.

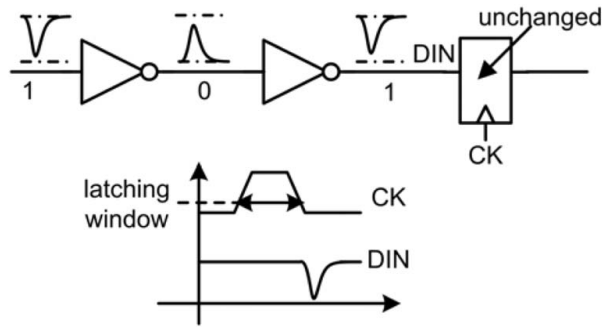
3.3 SEU in Combinational Logic

Any node in a combinational circuit can be struck by a particle and generate a voltage transient which can propagate to the combinational logic and cause an error if latched by a sequential element such as a memory-cell. If a particle strike generates enough charge for an upset in a transistor, it might not be received as an error for the software operation. It could be masked by one of the following phenomena:

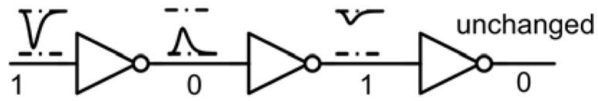
1. *logical masking* occurs when a particle strikes a portion of the combinational logic whose result can not affect the output due to subsequent gate whose result is completely determined by it's other input values. This phenomena is illustrated in figure 3.4(a). The upset will not be latched and there will be no soft error.
2. *Temporal masking* occurs when the transient pulse reaches a latch but not at the time the latch captures it's input. This can be seen in figure 3.4(b).
3. *Electrical masking* occurs for transients with bandwidths higher than the cutoff frequency of the CMOS circuit. The pulse amplitude may reduce, the rise and fall



(a) Logical masking



(b) Temporal masking



(c) Electrical masking

Figure 3.4: Masking of errors[4]

times increase and the transient will disappear. This can be seen in figure 3.4(c). Low frequency transients with sufficient initial amplitude will be amplified, since most logic gates are nonlinear and have substantial voltage gains.

3.4 SEU in SRAM

A typical SRAM cell is shown in figure 3.5. When an ion strikes one of the sensitive regions of the cell, charge collected by the junction results in a transient current in the transistor. As current flows through the restoring transistor sources current in an

attempt to balance the particle-induced current. This transistor however has a finite amount of current drive and channel conductance, meaning the current flow caused by a particle strike induces a voltage drop at its drain. This voltage drop is the main reason for upsets in SRAM cells.

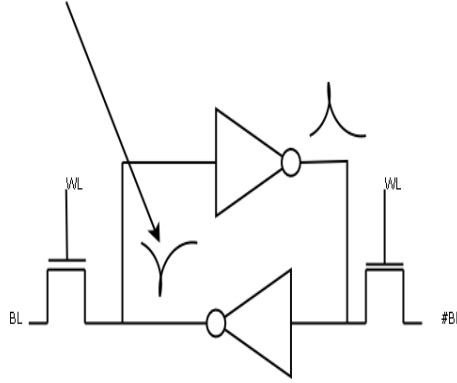


Figure 3.5: A typical SRAM cell

If the particle strike causes a transient in one of the nodes, the disturbance can propagate through to the second inverter and induce a transient in this inverter. The second inverter, gives the first inverter a wrong value and consequently the two nodes will flip and the memory-cell will store a false value. The cell will only hold a right value once it is rewritten or in the highly unlikely case a second particle will strike a sensitive area of the cell again.

Upsets can also occur when a particle strikes the bit-line. During a read operation a bit-line is discharged by a small current from a memory cell. The bit is read as true or false based on the voltage differential developed on the bit-lines during the access period of the memory cell. If a particle strikes close to the junction of an access-transistor of any cell attached to the bit-line, the voltage differential is disturbed. The smallest charge that needs to be deposited in order to cause a bit upset is called the critical charge (Q_{crit}).

3.5 SEU in DRAM

Dynamic random access memory (DRAM) is a type of random access memory that stores each bit of data in a separate capacitor within an integrated circuit. Since real capacitors leak charge, the information eventually fades unless the capacitor charge is refreshed periodically. Because of this refresh requirement, it is called a dynamic memory as opposed to SRAM and other static memory. If a particle strikes in a DRAM-cell, the charge is altered. If the charge was above and is now below a critical level, the refresh logic of the DRAM now sets the cell to a new state. The same holds if a cell started as below the critical level. The refresh-rate has an influence on the SEU vulnerability. Just before a refresh the charge of a set capacitor/bit is at its lowest. Now only little has to be done to remove some more charge so the charge is below the critical charge.

The biggest source of soft errors in Dram is caused by single event charge collection within each binary cell. These are caused by particle strikes in or near the capacitor or the source of the access transistor. This can be seen in figure 3.6. An upset in a cell is usually seen as a transition from state $1 \rightarrow 0$. In the late 1980s it was discovered that the ALPEN effect can occur in DRAM [24]. The ALPEN effect causes the capacitor to build up charge and thus has a transition from $0 \rightarrow 1$.

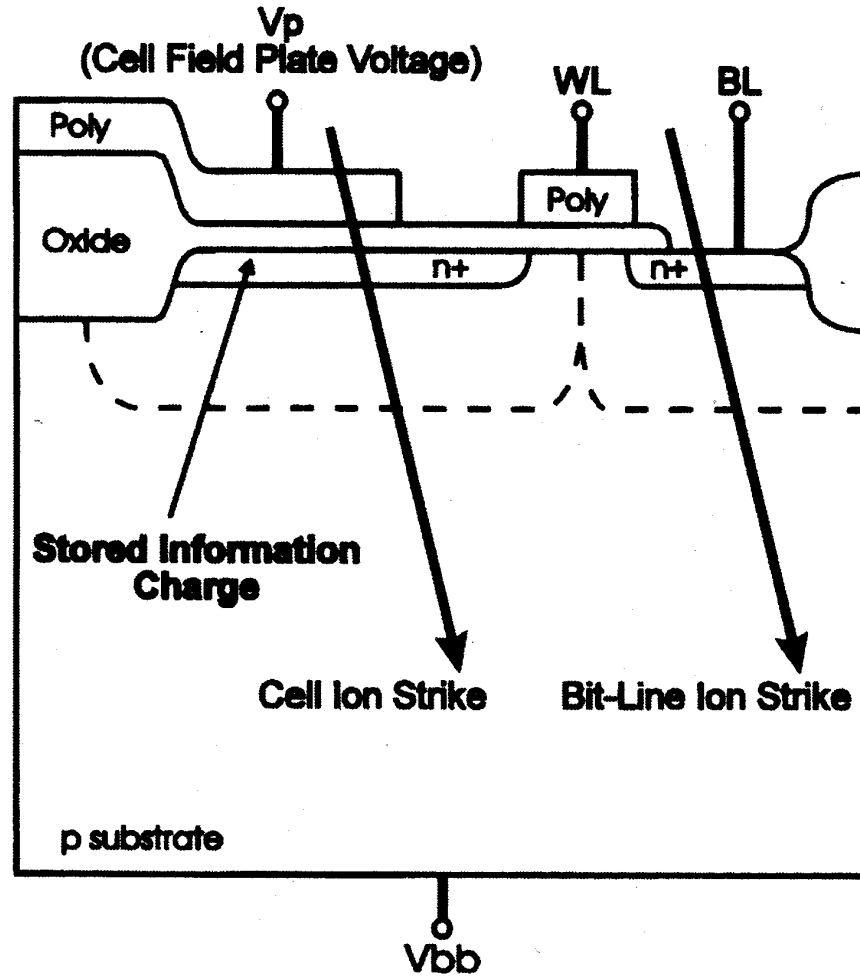


Figure 3.6: One transistor DRAM cell[2]

Particle strikes on bit(access)lines can also induce a soft error. During a read cycle the bit lines are in a floating voltage state. During this time DRAM cells are sensitive to the collection of charge into diffusion regions which are electrically connected to the bit access lines. This collection could arise from a particle hitting an access-transistor drain or the differential sense amplifier [25].

Bit line strikes are only possible during the floating precharge and sensing stages, and therefore are very dependent of the duty cycle. The bit line soft error rate is inversely proportional to DRAM cycle time. Cell upsets are independent of the cycle time. So

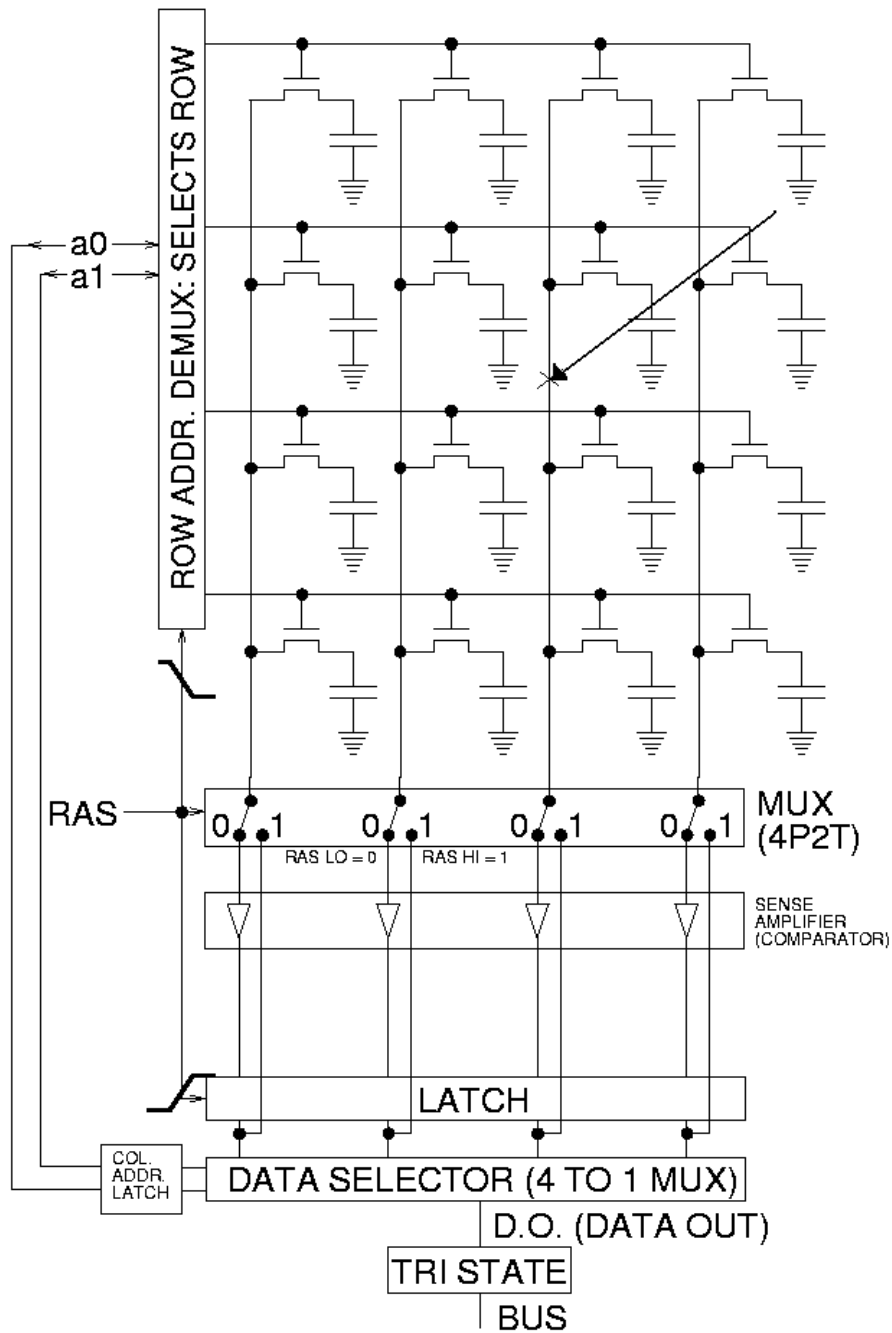


Figure 3.7: DRAM bit-line strike [5]

as the cells are becoming smaller and the clock speeds increases, bit line errors become increasingly important.

in 1988 Rajeevakumar et al. discovered a new upset mechanism in DRAM[26]. The combined effect of bit line and storage cell charge collection during a read operation

causes an error. Individually these processes usually don't collect enough charge for an error. This effect is called the combined cell-bit line Failure mode and for low clock frequencies it dominates both cell and bit line error rates.

In determining the SEU vulnerability the cell technology is a very important factor. There are a number of different storage structures and they have a big influence on the SEU susceptibility.

3.6 Multiple Bit Upset

Single-event multiple-bit upsets (MBUs) occur when a single particle strike causes more than one error in an IC. The greater the angle of incidence the greater the probability the charge track may intercept several sensitive regions and cause multiple upsets. This probability however is very small. It is reported in [27] that only 2% of all SEU in a 90 nm cache SRAM result in an MBU.

3.7 Estimating the SER

Many experimental studies have shown that the probability of a particle strike causing a soft error depends exponentially on the critical charge. This is also known as the *cross section* of the node as it depends linearly on the sum of the sensitive diffusion areas. This relationship has been confirmed using a semi-empirical model in [28] and is commonly used to model soft error rates [29]. The equation for all circuits is:

$$UR = F * K * A * e^{-\frac{Q_{crit}}{Q_s}} \quad (3.2)$$

Where UR is the upset rate, F is the neutron flux in particles per second, A is the area of the circuit susceptible to neutron strikes, K is a constant for a particular technology, Q_s is the charge collection efficiency. The collection efficiency Q_s and the constant K depend on technology and supply voltage.

In [29] the authors observe that since $Q_s(\text{NMOS}) > Q_s(\text{PMOS})$, the soft error rate due to strikes on NMOS-drains are about two orders of size greater than that of strikes on PMOS-drains. Furthermore, when a particle strikes an nMOS-source/drain junction it can only discharge a node only logic '1' s may be upset. Hits in an pMOS-source/drain junction may only charge the node, thus only logic '0' s can be upset. Hence we can use (3.2) to calculate the soft error rate of logic circuits more explicit:

$$SER = \sum_{i=1}^n \frac{t_{f,i}}{T} P_{obs,i} (P_{H,i} UR_{n,i} + P_{L,i} UR_{p,i}) \quad (3.3)$$

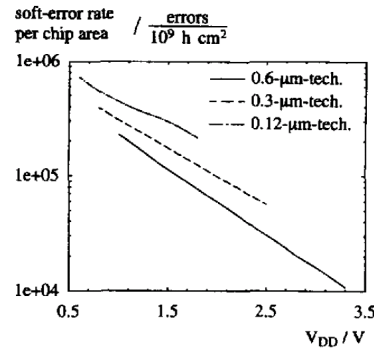
where n denotes the number of nodes in the circuit, $t_{f,i}$ is the time a dynamic register is floating. This accounts for the *temporal masking*. This should be smaller than clock period T . $P_{obs,i}$ is the probability for an upset to cause an observable upset at the output. $P_{H,i}$ and $P_{L,i}$ are the probability of the connected node holding a logical '1' or logical '0' respectively. $UR_{n,i}$ and $UR_{p,i}$ denote the upset rate for hits in an nMOS-source/drain junction and a pMOS-source/drain junction respectively.

Techonology	0.6 μm	0.3 μm	0.12 μm
normal supply voltage V_{DD}	3.3	2.5	1.5
channel length	0.5	0.25	0.1
channel width	1.9	0.95	0.4
Layout aea of 16×16 multiplier	0.59	0.14	0.024
Source-drain area	2.0×1.9	1.0×0.95	0.42×0.4

Table 3.1: Assumed technology parameters

An example is the SER of a pipelined 16×16 multiplier. The multiplier uses a carry/save array existing of full adder cells. This cicruit has approximatly 12800 transistors. $P_{obs,i}$ is very close to one. If an upset in one of the node occurs, the partial product of the cycle is erroneous. Since the multiplier continous with this erroneous results, the final outcome is erroneous as well. The probability a node has a logic value of '1' is the same as having logic value '0', so $P_{L,i} = P_{H,i} = 0.5$. The time a capturing register is floating, $t_{f,i}$, is assumed to be half the clock cycle.

By using the data listed in Table 3.1 and assuming the chip package as main particle source, the figure of Figure 3.8 can be estimated. It should be noted the SER greatly increases by lowering the supply voltage.

Figure 3.8: Estimated SER of 16×16 multiplier [6]

3.8 Calculating Block SER

Untill now only *raw* soft errors were discussed. Not all raw soft errors cause a program to fail. For example, a soft error in a functional unit that is not currently processing an instruction or an SRAM cell that is not storing usefull data will not harm execution. Such an error is also masked. A method to take this masking into account is using the Architectural Vulnerability Factor (AVF). A hardware structure's AVF is the probability that a fault in that particular structure will result in an user-visible error [30]. The AVF for a component can be calculated as the percentage of time the component contains Architecturally Correct Execution bits(i.e., the bits that affect the final program output).

To calculate the SER rate of a component we can use:

$$SER = AVF * SER_{raw} \quad (3.4)$$

The AVF is dependent on multiple factors and is application dependable. For certain instructions errors are masked. For example, if the instruction is the *set less than* instruction (SLT), if the first source register is smaller than the second, the destination is set to one. If the first source register has value 12 and the second 100 the outcome is one. If this outcome is used in a *branch if equal to zero* instruction, the branch will not be taken. If a fault was injected into the SLT instruction and the outcome became 4, the branch will also not be taken. This phenomena can be named application masking.

The system soft error rate can be calculated by: Combining all the SER of a system of k components to calculate the system Soft Error Rate gives:

$$SER_{sys} = \sum_{i=1}^k AVF_i * SER_{raw_i} \quad (3.5)$$

A diagram of this proces is shown in figure 3.9.

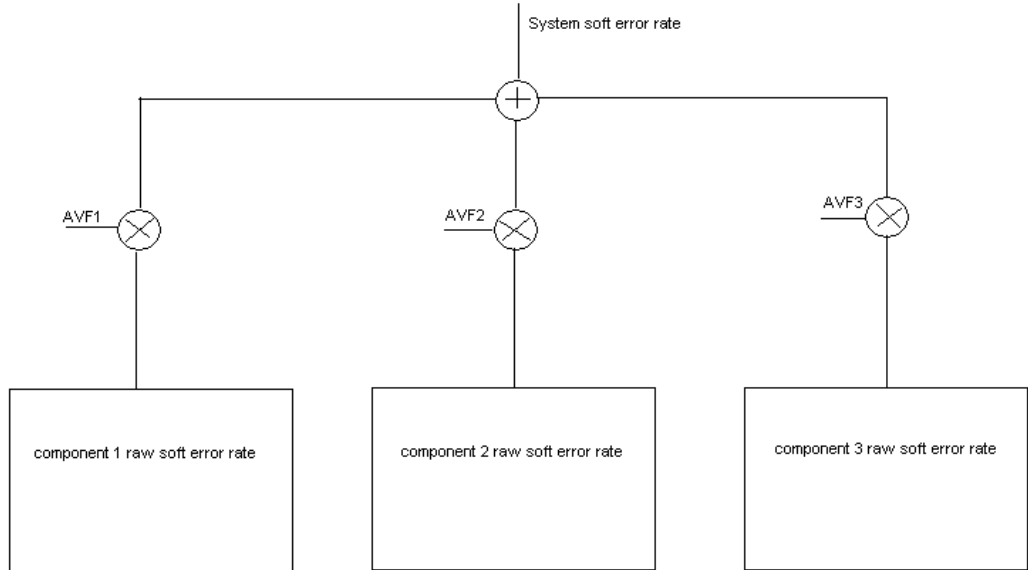


Figure 3.9: Combining soft error rates

3.9 Conclusions

In this chapter the main causes for soft errors are described. Soft errors are mainly caused by particle strikes. In this chapter single event upsets caused by particle strikes are described for combinational logic, SRAM and DRAM. Furthermore, multiple bit upsets are described and a method to estimate soft error rate (SER) of a circuit is shown. Finally a method to calculate the SER of a combination of circuits is shown.

Fault Tolerance Techniques

There are several strategies to provide soft-error protection. They can be divided into three methods, hardware redundancy, time redundancy and circuit-level techniques. In this chapter these three fault tolerance techniques are described.

4.1 Circuit-Level Techniques

The main circuit-level technique used for soft error mitigation is radiation hardening. Radiation hardened are often manufactured on insulating substrates instead of the usual semiconductors wafers. This is known as Silicon On Insulator(SOI). In normal technologies, charge liberated by a particle strike can be collected by the source/drain junction from several microns deep. In SOI technologies, the buried oxide in theory limits charge collection to the charge liberated in the top silicon layer underneath the gate. Because this layer is normally very thin compared to the active thickness of NON-SOI technology, the sensitive area of SOI is considerably less. This can make SOI IC's considerable less susceptible to producing soft errors from particle strikes. However particle strikes can trigger a bipolar mechanism that limits the SEU hardness of SOI circuits.

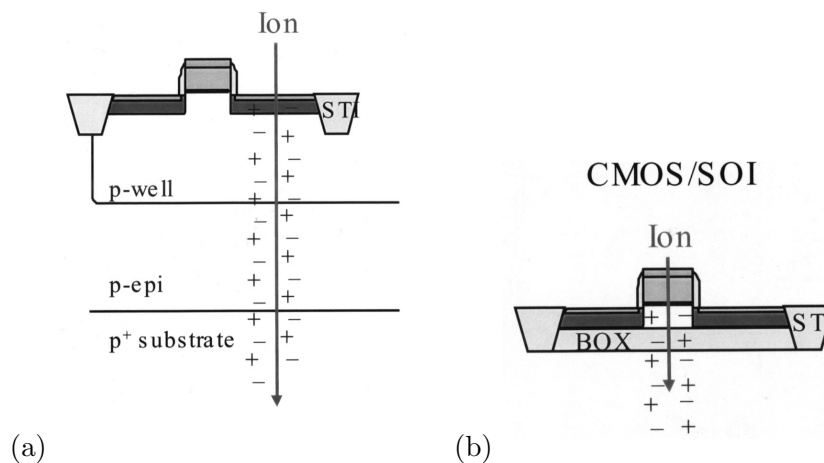


Figure 4.1: Comparison of the ion track between NON-SOI(a) and SOI(b)

In circuit level hardened circuits although there are less soft errors, every soft error goes undetected. Furthermore it gives area overhead and power overhead compared to a non-hardened circuit. It has been used for many years in military and space applications.

triplet received	interpreted as
000	0
001	0
010	0
100	0
111	1
110	1
101	1
011	1

Table 4.1: Triple redundancy

4.2 Hardware Redundancy

There are several soft error mitigation techniques using hardware redundancy. The oldest one is triple modular redundancy. Here three (sub)systems perform the same process, and at the end the results are compared. Majority voting is used to submit the real outcome. The upside of this technique is that it will mostly detect and correct all errors, except when two systems are producing an erroneous result. Then the result cannot be corrected. Furthermore it doesn't mean a reduction in speed. A downside is the fact it uses three times the area and power than a single system. This can be applied to combinational logic, for example three ALU's, and also for sequential logic. An example is given in Table 4.1. Each bit in the memory is represented by three bits.

The general architecture of a concurrent error detection scheme is shown in figure 4.2. There are many techniques that use this scheme. Here are the most commonly used:

- Duplication, each process is performed twice. If the outcomes agree the output is submitted, else the process is performed twice again.
- Parity prediction, the predicted parity is compared to the outcome parity. This cost less hardware than duplication does.
- Unidirectional error detecting codes assume that all errors are unidirectional; i.e., they change logic '0's to logic '1's and vice versa but never both at the same time.

An advantage of CED over Triple modular redundancy is the fact that CED uses less hardware. A disadvantage is the fact they can only detect faults and recovery measures are necessary.

4.3 Time Redundancy

Software implemented hardware fault tolerance is a time redundant soft error tolerance scheme. Program instructions are executed twice and results are compared. If they differ a soft error is detected. Advantage is there is no need for extra hardware, disadvantage is the time overhead. A program using software fault tolerance can use 40 to 200 percent more time than the original program.

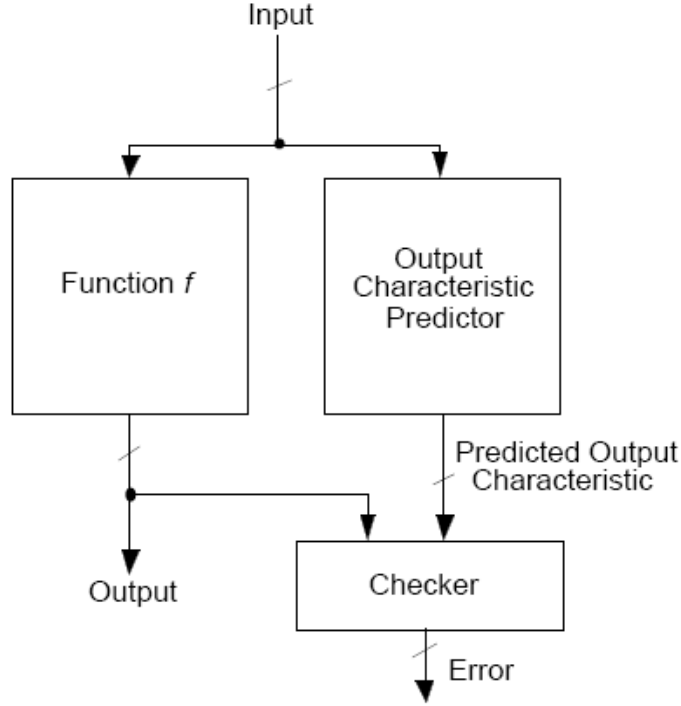


Figure 4.2: Concurrent error detection scheme [7]

Another time redundant mechanism is multi-threading. The same instruction sequence is executed using two threads and the results are compared. This technique can detect almost all soft errors. Some extra area is needed for saving the result of the thread. This technique takes about 20 to 40 percent extra time.

Multistrobe detects and corrects error by strobing outputs of the same logic block multiple times by delayed clocks. If the soft error remains during the strobing time this fault tolerance technique can not detect the error. There is a small performance overhead when detecting an error, and a large overhead when correcting a soft error.

4.4 Conclusions

In this chapter we have briefly described the three main strategies for fault tolerance. Each of these techniques will introduce overhead. In case of circuit level techniques power overhead is introduced as well as cost overhead. Hardware redundancy introduces area overhead and consequently cost overhead. Time redundancy mainly introduces performance overhead.

Simulation Environment

To achieve universality of our fault injector, we target structural simulation environments. In this work we use Unisim described in Section 5.1. We implement a DLX simulator (Section 5.2), a MIPS simulator (Section 5.3) and finally a superscalar MIPS architecture (Section 5.4).

These implementations were necessary because the implementations in the Unisim library are of too low granularity. In the Unisim library there are an implementation of a Power PC core and an ARM-core. The cores are implemented as one big module. The higher the granularity the better the place a fault is injected is controlled.

5.1 Unisim Environment

The experiments in this work are performed using the Unisim environment [1]. Unisim is chosen because it is a structural simulation environment. Unisim is implemented as a layer on top of SystemC. A structural simulator can be formed by several modules, each corresponding to a hardware block. Structural simulators are opposed to monolithic simulators. Monolithic simulators, such as SimpleScalar[20], are simulators built as a single piece of software describing the whole processor.

Structural simulators have several benefits over monolithic simulators:

- Reuse: because simulators are decomposed into modules, it is much easier to extract and reuse parts of a simulator. For instance different cache architectures can be easily tried out, provided they have the same processor-cache and cache-memory interfaces. Because monolithic simulators do not reflect the modularity of processor architectures, it can be exceedingly difficult to update one or a few architecture components.
- Hardware to simulator mapping: structural simulators allow the intuitive mapping of the target architecture blocks into simulator modules.
- Easy modification of simulators: in a monolithic simulator, the different simulator parts are usually so much interwoven, that modifying a part can have unexpected effects on other parts. For instance, using a variable containing the outcome of a branch at a wrong place in the code could cause the branch result signal too propagate too early in the architecture pipeline.

SystemC is already a structural simulation environment, however it has several downsides. The communication protocol between modules is not normalised. SystemC provides a syntax for writing modules interfaces, but they do not specify how the modules should exchange data. As a result, two separately developed modules usually cannot

communicate. Furthermore there is no attempt at reusing control. Modules describe the hardware block functionality, but most of the simulator code corresponds to hardware control. The control code within a module describes the interaction with other modules. As a result, if a module is altered, the control code must be rewritten, almost removing the reuse capability of a structural simulator.

Unisim provides four key features over SystemC:

1. Normalised communications and reusing control. Unisim defines a communication protocol which characterises interactions between modules and embeds it all in the modules communication interfaces. This is called control abstraction.
2. Interoperability. Unisim provides the ability to inter operate with existing simulators. This is possible via wrapper modules, which can connect to existing simulators.
3. Library. Unisim comes with a library of simulator components and full models.
4. Capabilities. Unisim allows capabilities to connect to modules to perform functions, such as power modelling, sampling, statistics, etc.

The communication/control mechanisms in Unisim are based upon SystemC ports. A three-way handshake is used to communicate between modules. First the information which module A sends to module B is inserted in the data part of the signal. If module is ready to accept the incoming data, the accept part of the signal is set to true. The third part of the handshake is the enable part of the signal. Module A can set the enable signal to true to let module B know it can start using the data. The third part is essentially the decentralised control.

5.2 The DLX-Processor

The DLX is a RISC processor architecture design by John L. Hennessy and David A. Patterson, the principal designers of the MIPS and the Berkeley RISC designs respectively [9]. The DLX is essentially a cleaned up and simplified version of the MIPS-1 processor, with a simple 32-bit load/store architecture. Intended primarily for teaching purposes, the DLX design is widely used in university-level computer architecture courses. Figure 5.1 depicts the DLX block diagram.

This figure shows a 5-stage pipelined DLX-processor. The same ALU is used for computing memory addresses, branch addresses and arithmetic/logic operations. The main hardware components are: the Instruction Memory, the Register File, the ALU and the Data Memory. In a structural simulator the code remains very close to the block diagram. The connections between the modules closely correspond to the datapath links. In the block diagram of Figure 5.1 there is a large block called control. As mentioned before a benefit of Unisim is the decentralisation of control, so hardware blocks can easily be replaced by other implementations. If control is implemented as a single, separate module, inserting or removing a module in the simulator may have a significant impact on the behaviour of the simulator. To counteract the change in behaviour a complete rewrite of the control module could be necessary. A simple solution to this problem is

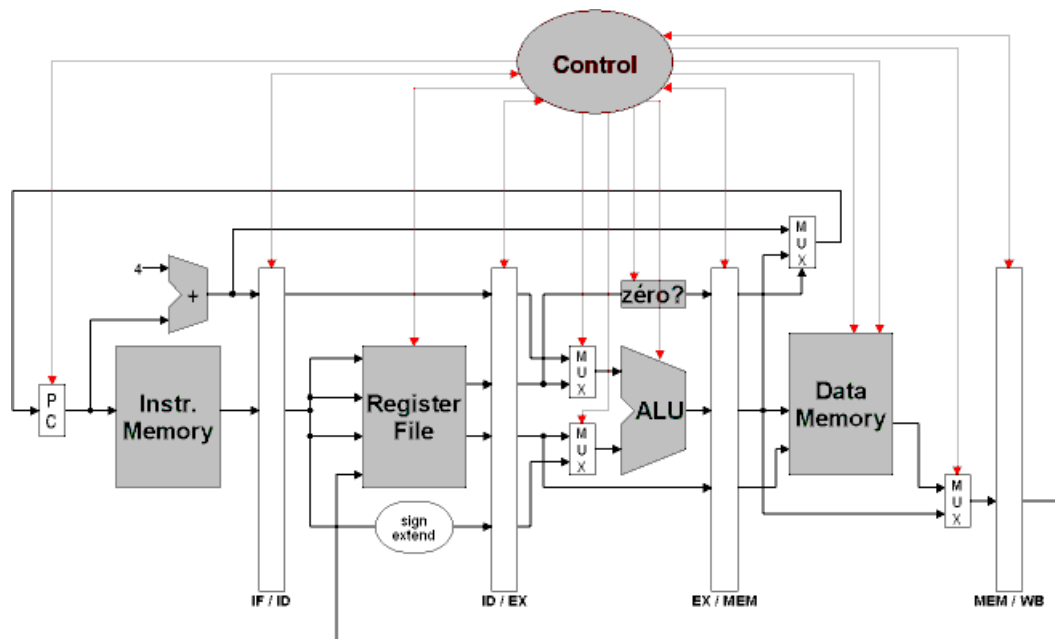


Figure 5.1: The DLX pipeline [8]

to distribute control over each hardware block. Now each block is in charge of how it interacts with only the modules it is connected to. If a module is modified, removed or added, only the distributed control part of the modules it is interacting with are affected. One of the key features of structural simulation is the similarity between the hardware block diagram and the corresponding simulator code. In several architectures control is more centralised than distributed. In Unisim this paradigm is broken to improve code re-usability.

The block diagram with distributed control is depicted in Figure 5.2. The grey boxes correspond to the modules of the DLX-simulator. To enhance re-usability of modules across simulators, the centralised control is distributed to the different modules composing the simulator. Control is both part of the module behaviour implemented through processes, and part of the communication scheme. Besides being connected to other modules, a module can also be connected to a clock signal. The process within a module can be made sensitive to the raising edge or the falling edge of the clock.

Every clock cycle begins with the Instruction Memory module reading the opcode pointed to by the Program Counter (PC) from memory. The opcode is then presented to the Register File module. At the end of a clock cycle the PC is updated if the value is accepted by the Register File module or if a branch has to be taken. At the start of a clock cycle, if there is an instruction in its input latch, the Register File module fetches the values from the registers and presents the instruction with values at the output of the module. If the values aren't ready or there is no instruction in the latch, the output remains empty. At the end of a clock cycle the module empties its latch if the output is accepted and sets a flag signalling the destination register of the instruction in the output latch is not ready. Also it performs the writeback of previous instructions. If

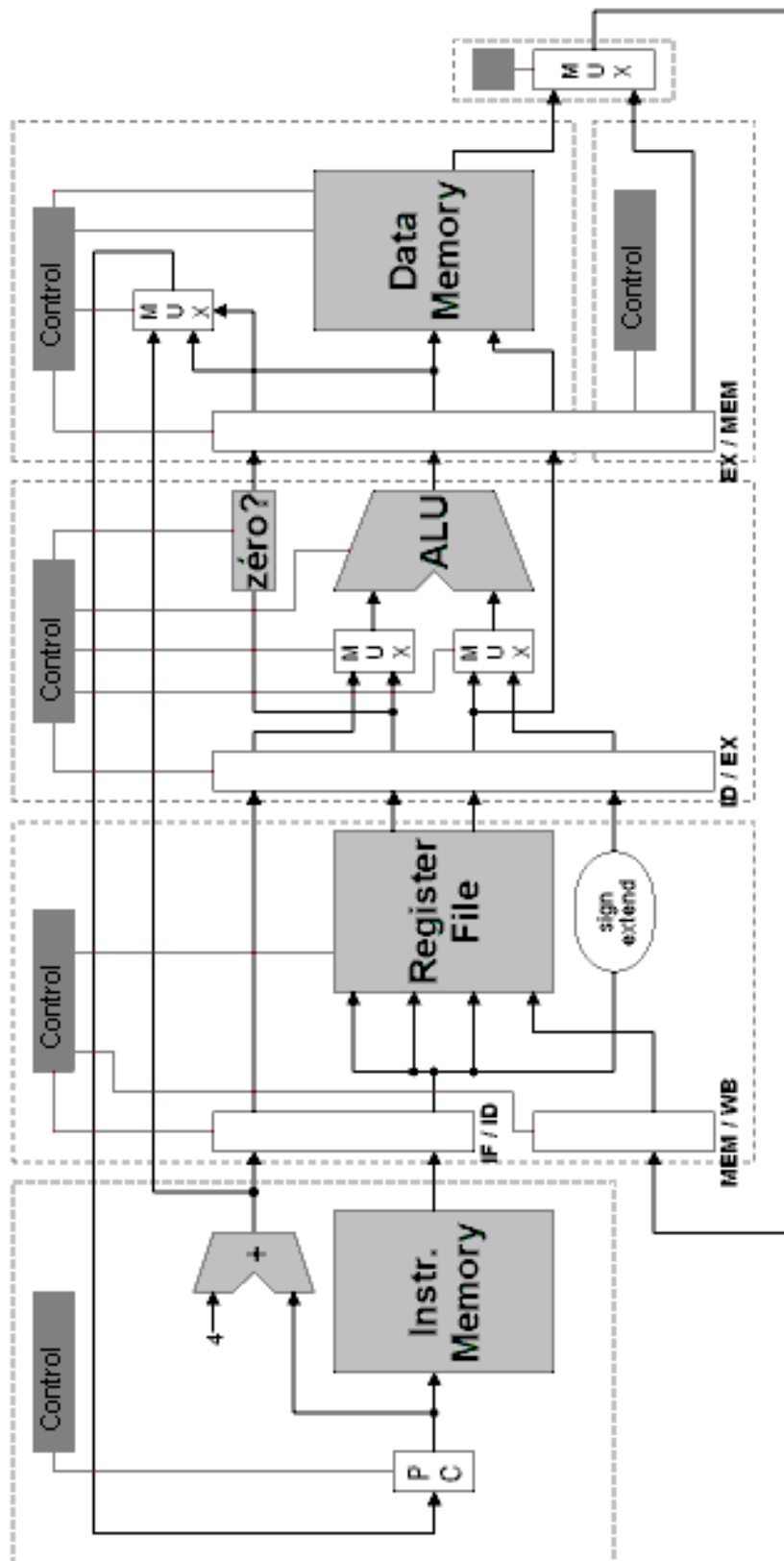


Figure 5.2: The DLX pipeline with distributed control [8]

the ALU module input latch is empty at the start of a clock cycle, the instruction is read from the Register module. If it is a branch instruction, the calculated address is presented on the wires between the ALU and the Instruction Memory module. The output to the multiplexer remains empty. If the instruction is a memory load or store, the calculated address is presented to the Data Memory module. If it is an other instruction the computation result is presented to the multiplexer and the wire to the Instruction Memory module remains empty. At the end of a clock cycle, the latch is emptied if the result is excepted by one of the connected modules. The Data Memory module loads the value from the address at the start of a clock cycle, if there is an instruction in it's latch and the instruction is a load instruction. At the end of a clock cycle store instruction are handled and if it's output is accepted by the module responsible for writeback the latch is emptied. Finally if the latch is empty and there is an instruction presented to the input, the instruction is stored in it's latch. The Bus Arbiter module, the rightmost block in 5.2, simply connects the ALU module result and Data Memory module output to the write-port of the Register File.

5.3 The MIPS-1 ISA Implementation.

DLX programs can only use 15 instructions, this is not very realistic for modern processors. To be able to simulate more realistic programs the DLX simulator is adapted to support the MIPS-1 instruction set. For this instruction set a cross compiler version of gcc is available. This means c-programs can be compiled for the MIPS simulator.

To allow the DLX-processor to support the MIPS-1 instruction set, first the data send between to modules needs to be augmented. Part of the data send between modules is the instruction. The instruction set now supports 60 instead of 15 instructions. The Instruction Decode module also needs an addition to be able to recognise the new instructions. Also the ALU needs to be extended to execute the new instructions.

A hazard is a situation that prevents the next instruction to be executed. If a pipelined processor issues a branch instruction, most commonly a branch predictor is used to predict the next instructions to execute. If the prediction is wrong, the instruction already issued are flushed. The DLX simulator uses software introduced no-operation instructions(nops), to ensure correct execution of the program. The three instructions after a branch are nops, so the processor effectively stalls till the outcome of the branch is known.

The MIPS R3000 processor is the base for the Unisim MIPS simulator. In the MIPS simulator the branch condition and branch address calculation are moved into the ID-module. This removes the need for nops, because the new address is in affect after the branch delay slot. This movement only partly resembles the real world. In the MIPS R3000 processor the branch condition is only calculated in the ID module for conditional branches using a zero test. The MIPS simulator doesn't support floating point operations in accordance to the R3000 which needs a co-processor for handling floating point instructions.

Now the simulator can execute gcc compiled programs. However, since gcc compiled programs follow the Executable and Linking Format (ELF) standard, an ELF-loader needed to be implemented.

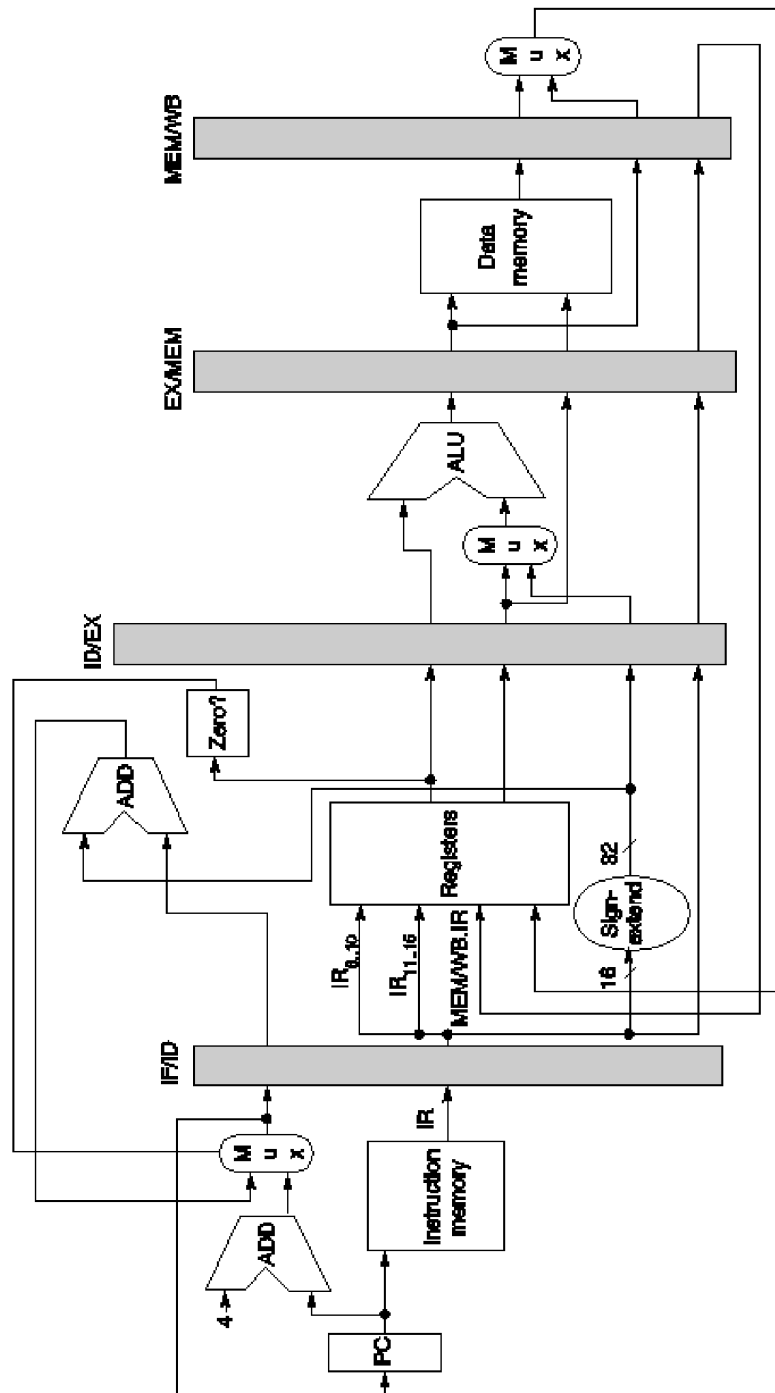


Figure 5.3: The MIPS simulator [9]

5.4 The MIPS Superscalar

Nowadays, many modern processor designs use superscalar processing. To be able to measure the relation between the performance overhead of a fault tolerance technique and the number of functional units, the MIPS-1 simulator is extended to become a superscalar architecture, shown in Figure 5.4. The first module changed is the Instruction Fetch module. The connection between the Instruction Fetch and Decode module is widened to four instructions, starting with the instruction pointed to by the program counter (PC). At the start of a clock cycle the instructions are updated. If a branch address is submitted to the module, the module checks at the end of the cycle if the condition is true, and if so the PC is updated to the branch address. The module can be seen in the figure below.

The Instruction Decode module must now be able to accept the incoming instructions. The module has a four instructions wide queue. This queue stores the instructions which are offered on the ports coming from the Instruction Fetch module. At the end of a clock cycle, the instructions are fed into the free positions of the instruction queue. At the start of a clock cycle, every entry of the instruction queue is checked whether the operands are ready and there is no Write After Read issue.

This issue is also known as a WAR hazard. It occurs when an instruction tries to write to a register when an earlier fetched instruction still needs to read it. It can be solved by storing the register values along with the instruction as soon as the register is ready. If the register is stored with the instruction, then the later fetched instruction can be issued.

Write After Write (WAW) hazards occur when two instructions try to write to the same register. They need to be executed in order, to make sure the right value is in the register in the future. This is done by keeping track of the destination register and stalling the issue of a later instruction with the same destination.

Read After Write (RAW) hazards are stalls in the pipeline caused by trying to read a register which is the destination of an unfinished earlier instruction. This can be avoided by keeping track of the readiness of each register.

The techniques explained above are known as scoreboarding. If all operands are ready and there is no chance of a conflict, the instruction is placed into a buffer of a functional unit or a memory address calculation unit and the place in the instruction queue is freed. At the end of the cycle, the instruction queue is reordered so it is actually more of a First In First Out (FIFO) buffer. If an instruction is accepted by a functional unit or a memory address calculation unit, the buffer is emptied. If there is data on the write-back lines of the module, they are now written into the registers, setting the status of the register to ready. If at the start of the cycle a branch is detected in the instruction queue, the instructions after the branch delay slot are unable to be executed. If the branch is taken, the instructions after the delay slot are flushed out of the instruction queue. If the branch is not taken every instruction in the instruction queue can be issued again. Every integer instruction has a 1-clock cycle delay, except the integer multiply and integer divide instructions. These have a delay of 6 and 32 cycles respectively. These can only be executed by one of the two ALUs. In the next section it will become clear why. The ID-module is shown in figure 5.6.

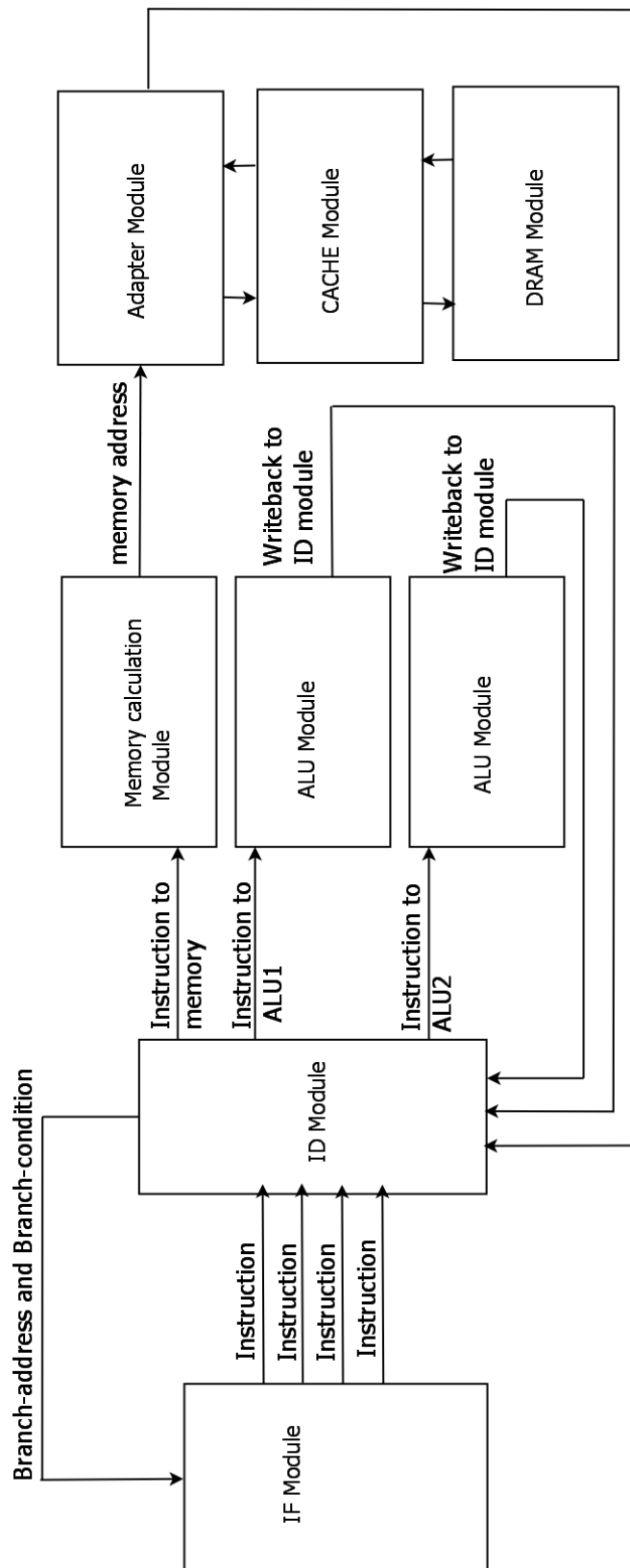


Figure 5.4: The superscalar MIPS simulator

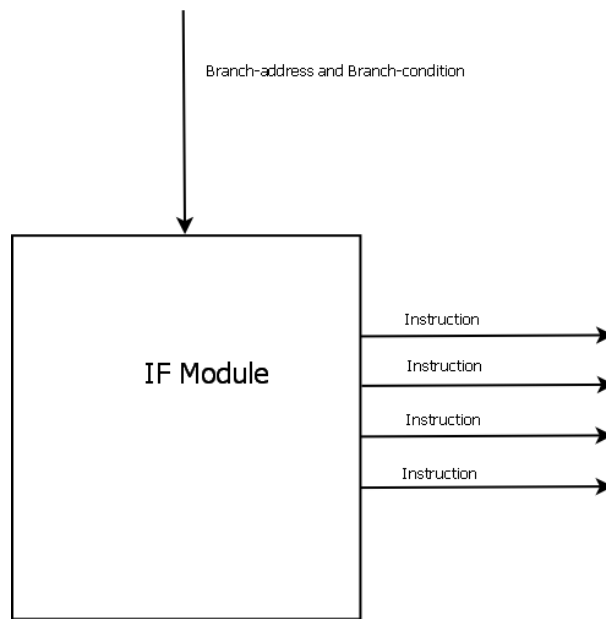


Figure 5.5: The instruction fetch module

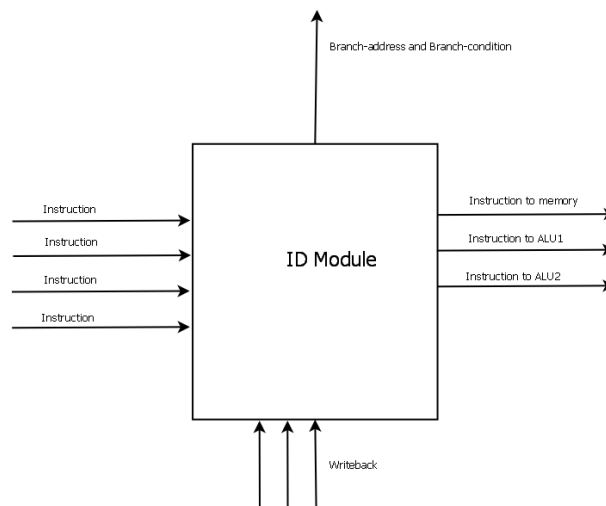


Figure 5.6: The instruction decode module

The ALU module fills its buffer at the end of a clock cycle. At the start of the next cycle, the value is computed and is copied to the out-port. An exception are the divide and multiply operations. These instructions store their result into two 32 bit registers called HI and LOW, corresponding to the most and least significant 32 bits respectively. These values can be copied into the general purpose registers by using the MFLO and MFHI, move from hi/low, instructions. This is the reason divide, multiply, MFHI and MFLO can only be executed by one of the two ALU's. The out-port of the ALU'S are directly connected to the write-back-sections of the Instruction Decode

Module. If the simulated processor has a bus, a bus module can be placed in between. The memory address calculation module calculates the memory address and adds it to the (memory)instruction. The instruction is then passed to the memory module.

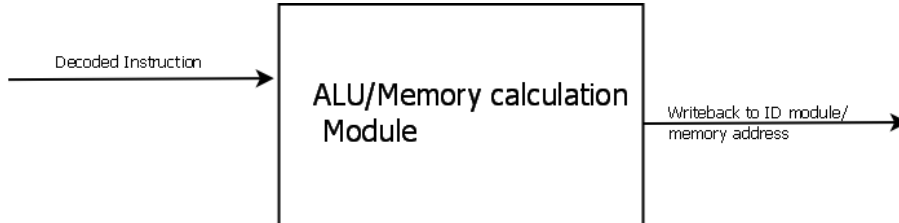


Figure 5.7: The ALU/memory address calculation module

The Memory module can be simulated to be a very simple data memory which can have a steady delay or a more complicated cache-dram module. In the Unisim library there is Write-back blocking cache and a Dram memory provided. These two modules communicate through different data than the MIPS processor simulator does, so a translation is needed. A memory adaptor has been made which translates the MIPS-memory requests to standard memory requests. The complete simulator can be seen in Figure 5.4.

5.5 Conclusions

In this chapter the simulation environment has been described. The Unisim environment is chosen because it is a structural simulation environment. It is based upon SystemC, which is a popular environment used for design and simulation of embedded solutions. Since SystemC is used for architectural exploration it is an excellent environment for researching fault tolerance techniques.

Unisim provides four key features over SystemC:

1. Normalised communications and reusing control. Unisim defines a communication protocol which characterises interactions between modules and embeds it all in the modules communication interfaces. This is called control abstraction.
2. Interoperability. Unisim provides the ability to inter operate with existing simulators. This is possible via wrapper modules, which can connect to existing simulators.
3. Library. Unisim comes with a library of simulator components and full models.
4. Capabilities. Unisim allows capabilities to connect to modules to perform functions, such as power modelling, sampling, statistics, etc.

The processor cores present in the Unisim library do not have enough granularity to inject faults into. A MIPS processor simulator has been developed and is further extended to become a superscalar MIPS processor simulator. These two processor simulators have enough granularity to be able to inject faults and investigate fault tolerance techniques.

Fault Injector

In chapter 3 it is shown faults can be modelled as bit flips. In this chapter we propose a class which can be coupled to a module. This class provides two possibilities, the class can be integrated into an existing module, or a new module can be made which can then be placed between two modules. The new module, models faults in the sender of the signal. The fault injector class can inject different types of faults:

- Random: a fault is injected with a probability of 1 divided by a user specified fault frequency.
- Periodic: a fault is injected periodically at a user specified frequency.
- Repeatable: a known fault injection pattern is followed.
- Permanent: when a fault is injected it stays in effect till the end of the simulation.

In case of a *Random* type of fault, a trace file can be made. This file contains the clock cycle when a fault was injected and which bit was affected. This can be used to repeat the fault pattern. In case of permanent fault, it is possible to specify if the time a fault is injected is given randomly or by a specific clock cycle.

The fault class can also be used to inject multi-bit faults. Since the module using an instance of the class is responsible for corrupting the data, it can apply the fault to different signals within the module. Furthermore, the minimal and maximal number of clock cycles a fault is applied can be specified. One could also apply multiple instances of the class in sequence. This counteracts the fact that the class can only inject one fault at a time.

6.1 Fault Injector for Combinational Logic

A diagram of the MIPS superscalar simulator with fault injectors for the ALUs can be seen in Figure Figure 6.1. In this case, faults in the ALUs can be modelled by a new adaptor module which intervenes on the write-back signal. The adaptor modules are in bold.

For combinational logic it is however more realistic to integrate the class in existing modules. For example, in case of the ALU module of Figure 5.4, this modules handles single cycle and multi cycle operations. If a multi cycle operation is performed, the module with integrated class can simulate particle strikes in anyone of the cycles. During a multi cycle operation, the ALU doesn't produce output until the operation has finished. If the separate fault module injects a fault during one of these cycles it is not captured by the next module, since there are no control signals send. Since an integrated fault injector knows the ALU is executing it can wait for the result and then inject the fault.

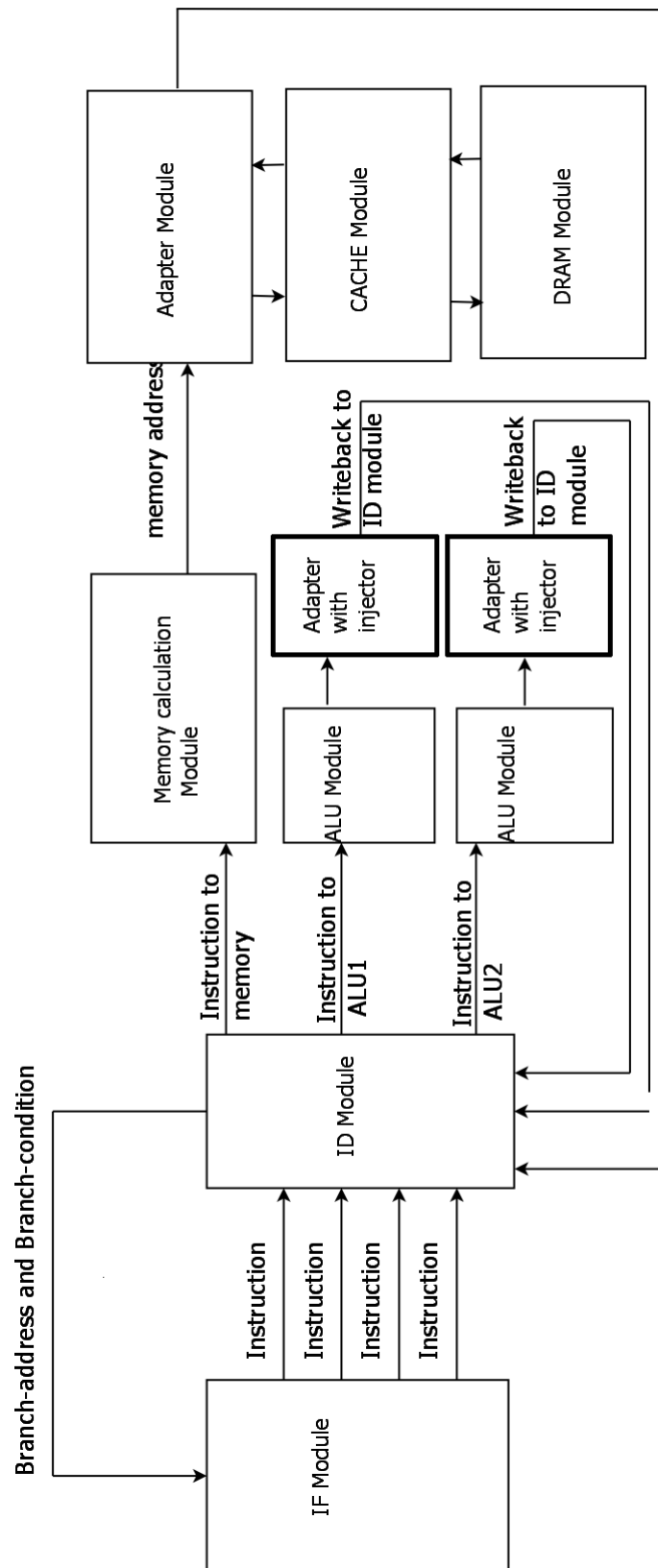


Figure 6.1: The superscalar MIPS simulator with fault injectors

Furthermore, in case of the MIPS simulator using an adaptor injector, faults injected when multiplying or dividing are not captured. The MIPS architecture stores the results of integer multiply and integer divide instructions in two specialised registers, HI and LO. The multiply and divide instructions have an execution time of 6 and 32 cycles respectively. When executing these instruction an injected fault of the separate module doesn't affect the result of the operations. When implemented in the module itself, the faults are captured by corrupting the values stored in the HI/LO registers.

Note that in order to use the class with an external module, you have to know very little about the internals of the module in which you inject fault. While in the more realistic case of integrating the class you need to understand most of the internals of the module.

6.2 Fault Injector for Memory

Up to now we discussed faults injected into combinational modules. A fault in a memory cell is modelled by a permanent bit flip. This means every instruction loading from the memory address reads the corrupted value. Unlike combinational faults, it is only removed if the value is overwritten by a store instruction or in the highly unlikely case another fault occurs in the same bit. An adaptor module mimicking this behaviour is made. If a fault is injected, the memory adaptor stores the corrupted value in a table, along with the address. Now if a load request is issued to the memory, first it is checked whether the address is in the table. If it is found, the answer of the memory to the request is replaced with this value. If a load is issued to the memory the adaptor checks whether the address is in the table. If so the entry is removed from the table. Figure 6.2 shows this solution.

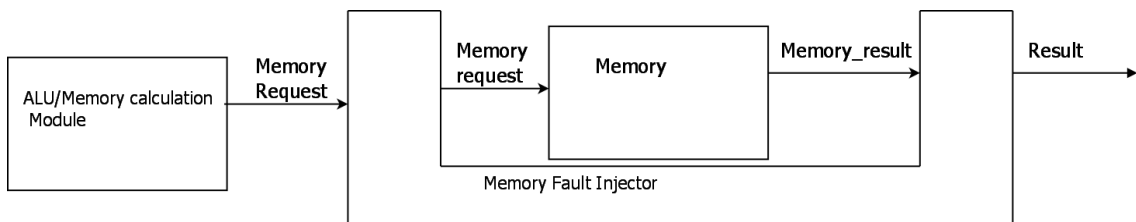


Figure 6.2: The memory fault injector

If a load is issued, the module can inject a fault in the result of this read action. If this is not the case, you can have many simulations which don't read any faulty memory value.

If a load is issued the memory adaptor calls the function `find_in_table`.

```

find_in_table(address)
{ search the table for address
  if found, return the offset
  else return -1
}

```

Now the adaptor knows if an address is present in the table or not. If it is, the table entry is placed in the output latch of the adaptor. If it is not present, the request is passed to the memory module and a fault could be injected in the result of the load request. If a fault is injected the function `put_in_table` is called. A description can be found below.

```
put_in_table(address, data)
{ Extend the table to make room for a new entry
  Put the address and the erroneous data in the table
}
```

In case of a write request, the function `remove_from_table` is called and the request is copied to the memory module. A functional description is shown below.

```
remove_from_table(address)
{ Remove the entry with the address from the table
}
```

In case an erroneous memory address is overwritten, the entry is removed from the table. If the address is requested by a load instruction the address is not found in the table and the (correct) value is read from the memory module.

6.3 Implementation

The fault injector class has several properties important for the type of fault to inject:

- Type: specify if it is a random, periodic, sequential or permanent type of fault.
- Fault_Freq: the fault frequency.
- Fault_Place: in which bit the fault needs to be injected.
- MAX_Fault_length: the maximum length a transient fault can occur.
- MIN_Fault_length: the minimum length a transient fault can occur.
- Fault_Length: the current number of cycles a fault should be injected.
- Fault: boolean value read by module indicating whether to inject a fault.

An object of the fault injector class is made in every module selected to inject faults. At the start of every clock cycle the function `inject()` is called. `Inject()` sets the boolean `Fault` if it is time to inject a fault. `Fault_Place` and `Fault_Length` are also set by this function. Since the fault injector class has no clock signal, the adaptor is responsible for timing. If `Inject()` is called and there is a fault in progress, the function decreases the time the fault is still in session. If the time is zero the fault is no longer applied. In case of a permanent fault, the time is not decreased.

In case a pattern needs to be made, the c++ directive `MAKE_TRACE` needs to be defined. This causes the function *Inject()* to write a tracepoint to a file. A tracepoint is defined as follows:

```
struct tracepoint{
unsigned long cycle; //The clock cycle when to inject
int Fault\_Length;//The length of the fault
int Fault\_Place;//The bit position to inject into
};
```

If a pattern needs to be followed, first the entire pattern is read into memory. Then the following two variables are set:

- `Time_In_Trace`: which tracepoint to follow
- `Max_Time_In_Trace`: which is the last tracepoint

In this case *Inject()* check whether the cycle specified in the current tracepoint is equal to the current clock cycle, and if so inject the fault given in the tracepoint.

6.4 User Guide

The usage of the fault injector in an Unisim simulator is best shown by means of an example. An example of a separate fault injector module for combinational logic is given below.

```
#include fault_injector.h
template
< int fault_frequency, int max_fault_length, int min_fault_length
>
class Adaptor: public module
{public:
    /* Ports */
    inport < instruction > in;
    outport < instruction > out;
    inclock clock;

    fault_injector *nfi;
    uint32_t mask;
    /* Constructor */
    Adaptor(const char *name):module(name)
    { sensitive_method(on_accept) << out.accept;
      sensitive_method(on_data) << in.data;
      sensitive_method(on_enable) << in.enable;
      sensitive_pos_method(start_of_cycle) << clock;
      //a new fault injector of RANDOM type is instantiated
```

```

//the data size is 4 bytes
    nfi = new fault_injector(fault_frequency, max_fault_lenght,
        min_fault_lenght, 4, fault_injector::TYPE_RANDOM, this->name());
}

void start_of_cycle()
{ nfi->inject(); //decide wether it is time to inject a fault
}

void on_data()
{ if(inAda.data.something())
    { instruction inst = inAda.data;
      if(nfi->fault) //is it time to inject a fault
      { mask = 0x01 << nfi->fault_place;
        inst.rd_value = inst.rd_value ^ mask; //flip the erroneous bit
          //rd_value is the part of the signal to corrupt
      }
    }
    out.data = inst; //no fault is injected, pass the data through
}
else
{ outAda.data.nothing();
}

void on_accept()
{
    in.accept = out.accept; //pass the accept signal through
}

void on_enable()
{
    out.enable = in.enable; //pass the enable signal through
}
};

```

It should be noted this adaptor class only has a clock signal for the timing of the fault injector. Only data signals are corrupted, the rest of the signals is simply passed through to the next module, so no control code is needed. Also it is not necessary to know how control is implemented in the simulator.

The superscalar MIPS with fault injectors from Figure 5.4 has been implemented in Unisim. the following code segments list shows an example of a superscalar MIPS processor simulator.

```

#define MAKE_TRACE 1 //make a trace file
#define FAULT_INJECTION_FREQUENCY 40000
//a fault is injected approximatlyonce every 40000 clock cycles
#define MAX_FAULT_LENGTH 1 //The fault has a maximum length of 1 clock cycle

```

```

#define MIN_FAULT_LENGTH 1 //The fault has a minimum length of 1 clock cycle
class GeneratedSimulator : public Simulator, public MyDefs
{public:
typedef Adapter<FAULT_INJECTION_FREQUENCY, MAX_FAULT_LENGTH, MIN_FAULT_LENGTH> MyAda;

IF_mod *if_mod;
ID_mod *id_mod;
EX_mod *ex_mod;
ALU_mod *alu1_mod;
ALU_mod *alu2_mod;
DataMemory *dmem;
MyAda *adaptor1;
MyAda *adaptor2;

    GeneratedSimulator()
    { char buf[255];

        // Module instantiation //
        if_mod = new IF_mod("if_mod");
            .
            .
        adaptor1 = new MyAda("adaptor1");
        adaptor2 = new MyAda("adaptor2");

        // clock connections //
        if_mod->clock(global_clock);
            .
            .
        adaptor1->clock(global_clock);
        adaptor2->clock(global_clock);

        // Module connections //
        if_mod->instr1 >> id_mod->read1;
        if_mod->instr2 >> id_mod->read2;
        if_mod->instr3 >> id_mod->read3;
        if_mod->instr4 >> id_mod->read4;

        id_mod->mem_out >> ex_mod->in;
        id_mod->pc_out >> if_mod->branch;
        id_mod->alu1 >> alu1_mod->in;
        id_mod->alu2 >> alu2_mod->in;

        dmem->out >> id_mod->write1;
        alu1_mod->out_result >> adaptor1->in;
        adaptor1->out >> id_mod->write2;
        alu2_mod->out_result >> adaptor2->in2;
        adaptor3->out2 >> id_mod->write3;
        ex_mod->out_mem_address >> dmem->request;
    } //GeneratedSimulator()
}; //class GeneratedSimulator

```

It can be seen from this example it is very easy to implement a module with fault injector into an existing simulator. The only thing one should know about the simulator is which part of the communication signal to corrupt.

6.5 Conclusions

In this chapter we have proposed a fault injector class. The fault injector can inject single cycle faults, multi-cycle faults, permanent faults and make and follow a pattern. The module that uses the class is responsible for injecting the fault proposed by the injector class in the signal.

The fault injector can be integrated into an existing module or a new separate module can be implemented. A separate module can inject faults into data send by the previous module. Thus it can simulate errors in the previous module and the bus. A memory adaptor has been developed. It wraps around a memory module. It keeps track of the address where a fault is injected. If this address is accessed before the erroneous value is overwritten the memory adaptor sends the erroneous value.

An example of a superscalar MIPS with fault injectors has been given.

Since combinational logic is becoming more susceptible to soft errors than memory, fault tolerance techniques targeted at combinatorial logic are becoming increasingly important. In this chapter some fault tolerance schemes for ALUs are investigated. Benchmarks are simulated and faults are injected using the fault injector proposed in 6. The number of faults and produced soft errors are collected and the reduction in soft error rate of the affected functional unit is calculated.

We propose to reduce the soft error rate of a functional unit by having the most frequently used instructions be dealt with by a functional unit with a lower soft error rate. The soft error rate of these two functional units combined is bound to be lower than that of the original unit. This will be tested using precomputation [31] in Section 7.2 and memoization [32] in Section 7.3. Precomputation is a technique used for improving a processors performance. It reuses data profiled off-line instead of computing the instructions. Memoization stores the result of an earlier instruction and reuses the result if the same operation and operands need to be computed. We propose that using precomputation and memoization the soft error rate of an ALU can be reduced.

In Section 7.4 a processor using instruction duplication and Memoization will be tested. When using instruction duplication every instruction is executed twice, the results of these executions are compared and if they are equal the result is written back.

The ECC will be tested in Section 7.5. Finally in Section 7.6 the fault injector is tested at a different granularity. Two Power PC processors from the Unisim library will run the same program in parallel.

7.1 Benchmarks

Because the MIPS and superscalar MIPS simulators are lacking an operating system, there is no file system and no memory management system. This means only small kernels can be run. The following kernels have been chosen, Matrix Multiplication, Image Addition, Fibonacci Sequence and Sum of Absolute Differences.

Matrix Multiplication is used in many multimedia applications. The listing is shown below.

```
void mult_matrix(void)
{
    int i, k, j, res;

    for( i=0; i<M; i++ )
    {
        for( k=0; k<P; k++ )
```

```

    {
        res = 0;
        for( j=0; j<N; j++ )
            res += MatrixX[i][j] * MatrixY[j][k];

        if( res > 255 ) MatrixR[i][k] = 255;
        else MatrixR[i][k] = res;
    }
}

```

This function has as input MatrixX of size $M \times N$ and MatrixY of size $N \times P$ the result is MatrixR of size $M \times P$. At the start of the simulation, MatrixX and MatrixY are loaded into the data-memory module of the simulator. At the end of the simulation the result, MatrixR, is read from the data-memory module and stored. The result can then be compared to the correct output, for example the result from a fault free simulation. If they differ one can assume a soft error occurred.

Image addition is a much used image operation. The main code can be seen below.

```

void add_images(void)
{
    int i, j, sum;

    for(i=0; i<N; i++)
        for (j=0; j<M; j++ )
        {
            sum = ImageX[i][j] + ImageY[i][j];
            if (sum>255) ImageX[i][j] = 255;
            else ImageX[i][j] = sum;
        }
}

```

This kernel takes two bitmap images of size $N \times M$. The header of the images is simply copied to the output, since these should be the same for both images. At the start of the simulation the bitmap data of ImageX and ImageY are copied into the data-memory module. At the end of the simulation the result is extracted from the data-memory module and stored in a result bitmap. Now the result can be compared to the correct result bitmap. And again if they differ, a soft error occurred.

Fibonacci numbers are used in a variety of applications. For example they are used in audio compression algorithms and in many models of biological settings.

```

void fibonacci(void)
{
    int i;
    int last = 1, last_last = 0;

    numbers[0] = 0;
}

```

```

    if( NUMBER == 0 ) return;

    numbers[1] = 1;

    if( NUMBER == 1 ) return;

    for( i = 2; i <= NUMBER; i++ )
        numbers[i] = numbers[i-1] + numbers[i-2];
}

```

NUMBER is defined as the last Fibonacci number. The result is the array *numbers*. This kernel is interesting for fault injection, if for example the first number is erroneous due to a soft error, every other number will also be erroneous.

The Sum of absolute differences is also a kernel which is widely used in multimedia kernels. For example, it is used for motion estimation for video encryption. The source code is shown below.

```

int sad()
{
    int i, j, tmp, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
        {
            tmp = block1[i][j] - block2[i][j];
            sum += ABS(tmp);
        }

    return sum;
}

```

The input of this function are *block1* and *block2*, both of size $N \times M$. Usually in video encoding the blocks are 16×16 . So this is a very small kernel.

The Power PC core found in the Unisim library has OS support, and thus can run bigger benchmarks. The following benchmarks have been chosen:

- Cjpeg encodes a file in other graphic file format than jpeg to a jpeg file. the jpeg file format is a lossy image compression method.
- ADPCM stands for Adaptive Differential Pulse Code Modulation. It is family of speech compression and decompression algorithms.
- Mesa is a 3d rendering program.
- Pgp is a computer program that provides cryptographic privacy and authentication.

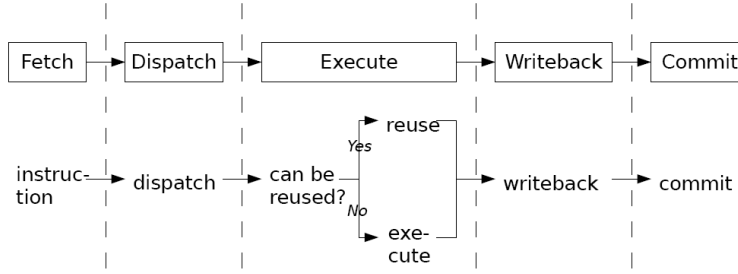


Figure 7.1: Instruction precomputation [10]

7.2 Instruction Precomputation

With precomputation an application is profiled off-line. Profiling means storing the most used operations, inputs and outputs an ALU needs to execute. The most frequent used operation-operand pairs are stored in a table (P-table), along with their results. Figure 7.1 illustrates instruction execution with precomputation. An instruction proceeds regularly until the execution stage. In this stage first a P-table lookup is performed. If the necessary result is found, it will become the output of the execution stage, otherwise the instruction is issued to the ALU. Since populating the P-table is done off-line, the profiling data may be assumed faults free if the profiling system is reliable. If the P-table is implemented using for example ECC, the soft error rate will be much lower than that of an unprotected ALU.

The notion compound ALU is introduced, which is the combination of the precomputation table and logic and a normal ALU. We propose a novel methodology to quantify the soft error rate of the compound ALU:

$$SER_{new} = (1 - hit_rate) \times SER_{ALU} + hit_rate \times SER_{P-table} \quad (7.1)$$

SER_{new} is the soft error rate of the compound ALU. SER_{ALU} is the soft error rate of the ALU. The SER of the control logic is assumed to be very small and is neglected. When using sophisticated ECC the SER of the P-table will be very close to zero. So the Equation 7.1 becomes:

$$SER_{new} = (1 - hit_rate) \times SER_{ALU} \quad (7.2)$$

From [10] it follows that the average hit-rate is about 15 percent. By applying Equation 7.2 the SER of the new unit is estimated to be 0.85 of the original SER. Furthermore, a hit in the P-table costs 1 cycle, even for multi-cycle operations such as multiplications and divisions. This means the execution time of multi cycle operations will decrease.

The implementation in Unisim is based on the superscalar MIPS Section 5.4. Since Unisim is a structural environment, the only module needing changes is the ALU module. The P-table is part of the module, and is simulated by a structure existing of the operands and the opcode. When an instruction is send to the ALU it checks whether it is listed in the P-table and thus precomputed. If it is, a boolean value is set to true. At the start

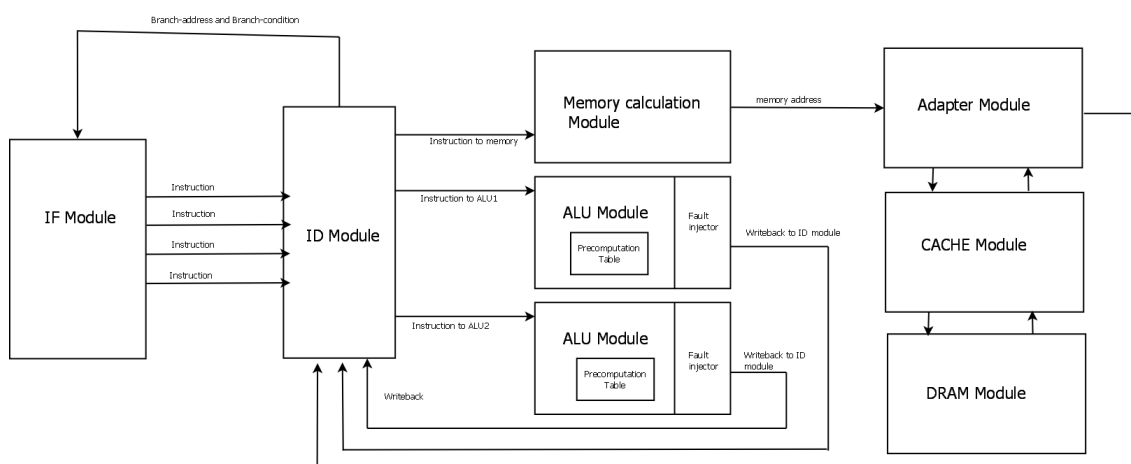


Figure 7.2: Superscalar MIPS with precomputation and fault injector

of the next cycle, if the boolean value is set no fault can be injected and the result of the compound ALU will be non-erroneous. The program flow is shown in Figure 7.3.

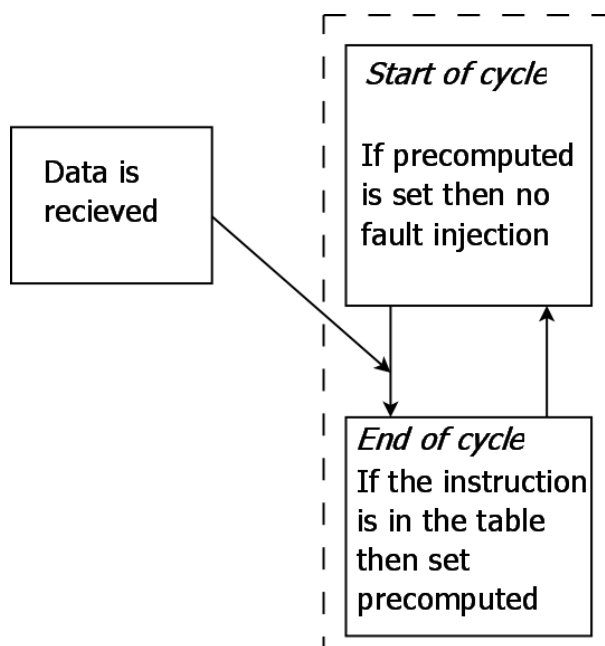


Figure 7.3: Program flow of the compound ALU

Faults are simulated by the injector by flipping a bit at the output of the execution module at a random time. However the ALUs are not busy all the time. If a fault is injected in the time an ALU is idle this fault will not be seen by another module, and will not become a soft error. To inject the same number of faults in the simulation with and without precomputation the trace functionality of the injector is used.

The P-table profiling data is based on the most used instructions. The simulation

Kernel	$softerrors_{old}$	$softerrors_{new}$	SER_{new}/SER_{old}	hit rate
Matrix Multiplication	1822	1622	0.917	0.088
Image addition	1611	1413	0.877	0.123
Fibonacci	2035	1700	0.835	0.174
Sum of absolute differences	11809	10606	0.898	0.123

Table 7.1: Number of soft errors using precomputation

results are shown in Table 7.1.

The hit rate is calculated by collecting the number of table accesses and the number of actual hits. $Softerrors_{old}$ is the number of soft errors in the situation no precomputation is applied. $Softerrors_{new}$ is the number of soft errors when using precomputation. The result of the experiments are as predicted by equation 7.1. The influence of the improvement in SER of the ALUs doesn't mean the improvement in block soft error rate is the same. This is dependent on the fraction of time the ALU is executing and the executing instruction.

We propose that by carefully choosing the instruction which are precomputed, the reduction in block SER can be further increased. Block SER means the SER of all the ALUs. If there are many divide instruction, which take 35 cycles in the MIPS processor, one could reduce this to one cycle by precomputation. This decreases the AVF of the block and thus decreases the system SER. However since the idle time of the other components is decreased the AVF of these components increases, thus increasing the system SER.

For example take the following program:

```
int a = 10, b = 5, c;
c = a/b;
```

This takes 21 cycles if the division is precomputed and 33 cycles if the division is not precomputed. Thus in the case of no precomputation the probability a fault is injected during the execution of the program is 1.5 times higher than if the division is precomputed.

7.3 Instruction Memoization

Memoization uses the same instruction reuse principle as precomputation. The only difference is that the entries of the table are produced on-line instead of off-line. At the writeback stage the instruction is also sent to the memoization module which updates the table. This technique is developed as a technique to improve processor speed, if the table however is protected against soft errors it can be used for SER reduction as well. Multicycle instructions transform to single cycle instruction when the instruction is present in the table and reduces FUs pressure. A disadvantage of this scheme is that it introduces power overhead as well as area overhead in regards to precomputation. This is caused by the fact if an operation is not present in the table, the table needs

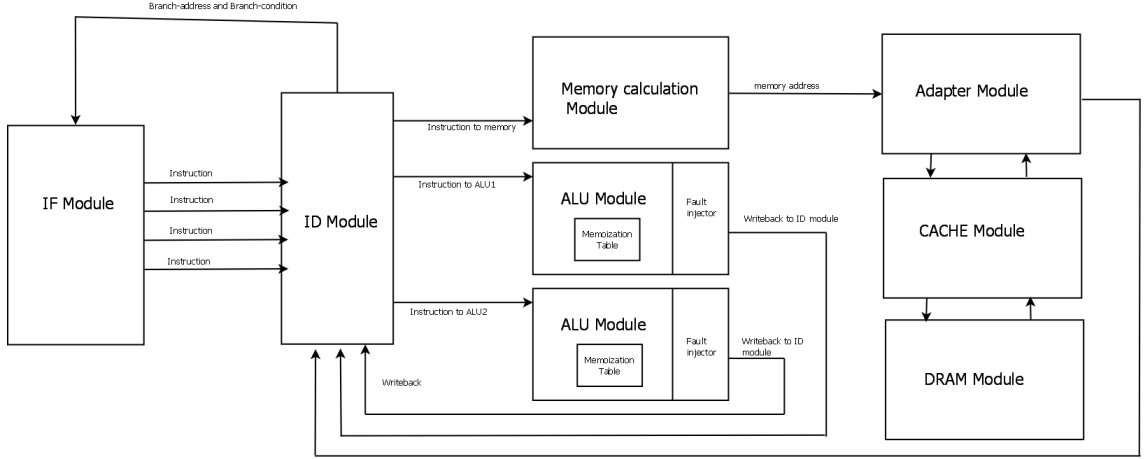


Figure 7.4: Superscalar MIPS with memoization and fault injector

to be updated after the execution of the instruction. Furthermore if a faulty result is written to the memoization table, the fault is repeated for every hit for that specific instruction-operands combination.

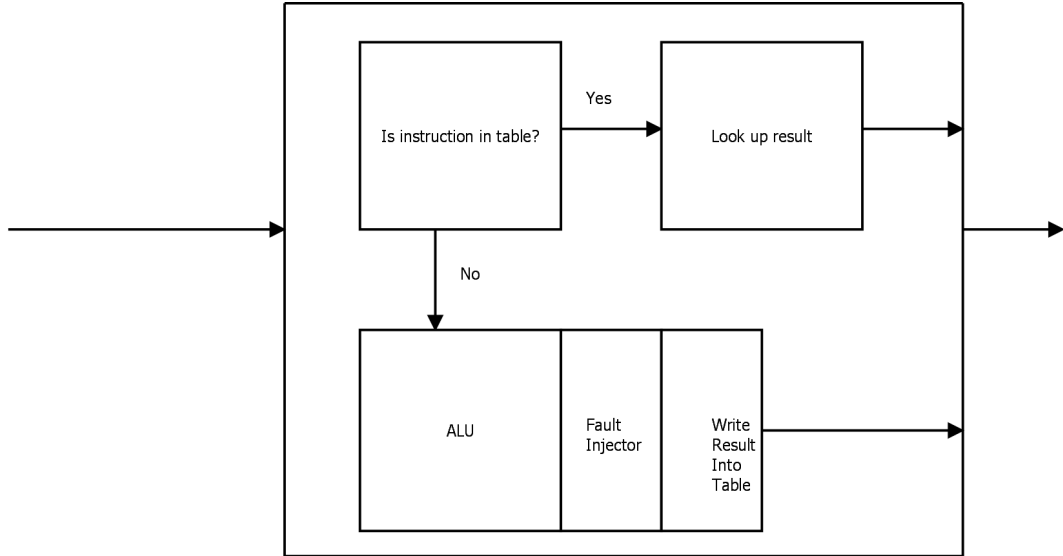


Figure 7.5: Memoization

The soft error rate can now be calculated as follows:

$$SER_{new} = (1 - HIT_RATE) \times SER_{ALU} + HIT_RATE \times SER_{memtable} \quad (7.3)$$

The probability of an erroneous result produced by the memoization table depends on the probability that the result affected by a soft error is written back to the table and the probability this result is reused. The probability a faulty instruction is written

Kernel	$softerrors_{old}$	$softerrors_{new}$	SER_{new}/SER_{old}	hit rate
Matrix Multiplication	616	1116	1.82	0.33
Image addition	479	501	1.05	0.23
Fibonacci	572	454	0.79	0.33
Sum of absolute differences	1232	777	0.63	0.48

Table 7.2: Number of soft errors using memoization, with a table of 4 entries

Kernel	$softerrors_{old}$	$softerrors_{new}$	SER_{new}/SER_{old}	hit rate
Matrix Multiplication	552	498	0.90	0.33
Image addition	512	343	0.67	0.36
Fibonacci	579	373	0.64	0.47
Sum of absolute differences	1232	737	0.60	0.48

Table 7.3: Number of soft errors using memoization, with a table of 8 entries

to the table is $1 - HIT_RATE \times SER_{ALU}$. The probability an instruction is reused is the average number of look-ups (LU).

$$SER_{memtable} = LU \times (1 - HIT_RATE) \times SER_{ALU} \quad (7.4)$$

Combining Equation 7.3 and Equation 7.4 gives the following equation:

$$SER_{new} = (1 - HIT_RATE) \times (SER_{ALU} + HIT_RATE \times LU \times SER_{ALU}) \quad (7.5)$$

In table 7.2 the improvement of SER is calculated from the simulated results. The (re)usage of a wrong result is also counted as a soft error, since the ALU produces an erroneous result.

The high hit rate is explained by the relative small sizes of the kernels. The kernels all have a small number of instruction in the loop, this means a high hit rate is easily achieved. If memoization is used, an optimisation would be to split the program into small parts, each run consecutively. Another notable phenomena is the sometimes very high number of soft errors when using memoization. This is caused by the relative long lifetime of some instructions. If for instance an instruction early in the program suffers

Kernel	$softerrors_{old}$	$softerrors_{new}$	SER_{new}/SER_{old}	hit rate
Matrix Multiplication	592	2270	3.83	0.33
Image addition	594	694	1.17	0.37
Fibonacci	567	296	0.52	0.54
Sum of absolute differences	1251	487	0.39	0.68

Table 7.4: Number of soft errors using memoization, with a table of 12 entries

Kernel	$softerrors_{old}$	$softerrors_{new}$	SE_{new}/SE_{old}	hit rate
Matrix Multiplication	572	223	0.38	0.65
Image addition	596	501	0.84	0.55
Fibonacci	576	223	0.39	0.69
Sum of absolute differences	1181	468	0.40	0.69

Table 7.5: Number of soft errors using memoization, with a table of 16 entries

from a soft error, is then stored in the memoization table and used 500 times after that, 500 times a wrong result is used. If more simulation results are obtained, this effect will surely be less obvious to see. A counteract measure for this effect is to limit the lifetime of instruction in the memoization table.

7.4 Instruction Duplication with Memoization

Instruction duplication is fault tolerance techniques that issues every instruction twice. A fetched instruction proceeds normally until the execution stage. There it is issued to the corresponding functional unit (FU) twice. The results are compared at the writeback stage and an error is signalled on a mismatch. This scheme uses time redundancy for fault detection. Permanent and long lasting faults will not be detected, because the same (wrong) result is produced twice in this case and then compared. Single cycle errors, such as produced by particle strikes, will almost always be detected. The probability a particle hits the same bit twice is at least the square of the probability of a strike divided by the number of bits the signal is made off.

Parashar et al. [33] proposed using memoization to counteract the time redundancy when using duplication. This scheme is depicted in figure 7.6.

This scheme detects every single error at the cost of extra clock cycles. It is interesting to observe the relation between the amount of soft errors and the number of extra clock cycles needed. This is shown in Table 7.6.

Kernel	faults injected	$softerrors_{ALU}$	extra cycles	uncorrected
Matrix Multiplication	1472	377	40.0%	0
Image addition	1080	363	27.8%	0
Fibonacci	2966	819	33.1%	0
Sum of absolute differences	13320	4098	27.8%	0

Table 7.6: Number of extra clock cycles needed by duplication and memoization

This fault tolerance technique corrects all single cycle soft errors at the expense of 32% extra clock cycles. For longer lasting faults the improvement in soft error rate can be calculated. This is show in Table 7.7. If a long lasting soft error disappears before the instruction is re-executed the soft error is detected and can be recovered by executing it ones more. However if the error is still in effect during re-execution, the erroneous

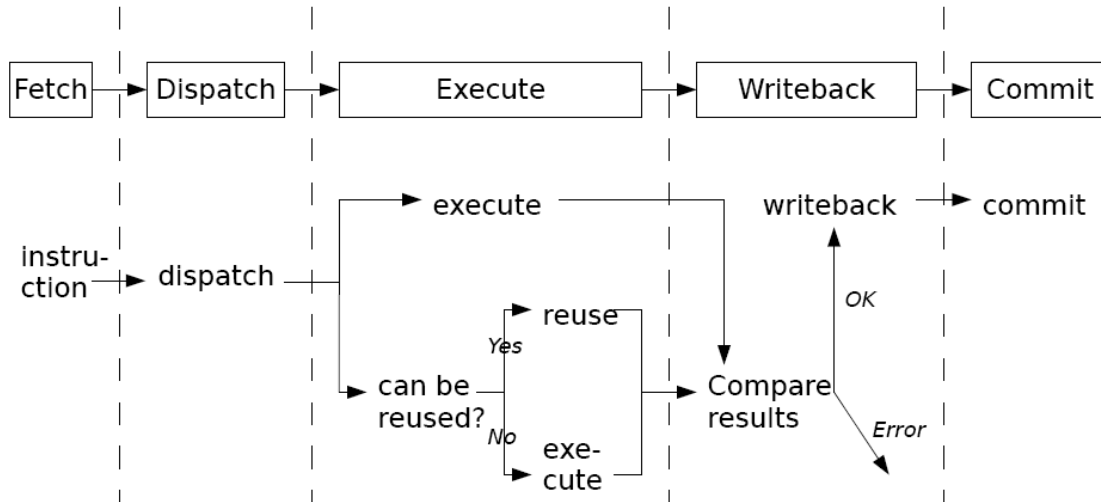


Figure 7.6: Instruction duplication with precomputation [10]

Fault length	faults injected	$softerrors_{ALU}$	wrong results
1	10100	2668	0
2	10587	7188	3234
3	11153	11066	4291
4	11567	13025	5842

Table 7.7: Simulation of long lasting soft errors

result is submitted or re-executed again, until there is no difference. This depends on the recovery mechanism.

From Table 7.7 it can be seen that the percentage of soft errors that occur in the ALU are shown at the output roughly stays the same at 50%. It is expected the percentage will rapidly increase with still longer faults. In the case of a fault length of 4 it can be seen the number of soft errors in the ALU is higher as the number of injected faults. That is caused by the fact the fault stays in affect for 4 cycles, thus you could actually say $4 \times$ the number of faults are injected.

7.5 Error Correcting Memory

To demonstrate the power of error correcting memory, the memory of the superscalar MIPS simulator is extended to use Hamming encoding [34]. Each memory location stores a word, 32 bits, and parity bits (6 bits). The following algorithm is applied:

1. Mark all bit positions that are powers of two as parity bits. (positions 1, 2, 4, 8, 16, 32)

Kernel	$softerrors_{noECC}$	$softerrors_{ECC}$	percentage corrected
Matrix Multiplication	945	0	100%
Image addition	722	0	100%
Fibonacci	650	0	100%
Sum of absolute differences	5409	0	100%

Table 7.8: Number of soft errors using precomputation

2. All other bit positions are for the data to be encoded. (positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, etc.)

For example 1000 1110 1101 0011 1111 0000 1010 0101 becomes XX1X000 X1110 110X1 0011 1111 0000 1X010 0101. Where X is a parity bit. When decoding, if a bit is flipped in the memory the parity of at least one of the extra positions is wrong. If for example bit position number nine of the original value is flipped, the parity of position number one and eight will be wrong. The position of the wrong bit in the memory value is now calculated by adding the positions of the incorrect parity bits.

To calculate the improvement of SER of the memory, the simulator without error correcting code (ECC) is run until a soft error occurs. The injected faults pattern is stored and rerun in a simulator with ECC. ECC is very powerful and will correct every single bit transient error. The simulation results are shown in Table 7.8.

ECC corrects all single bit transient errors at the expense of 18.75% area overhead.

7.6 Power PC Duplication

Duplication of two processors is a hardware redundant fault tolerance technique. The diagram showing duplicated Power PC(PPC) cores is depicted in Figure 7.7.

In normal case the detector passes the messages between the PPC cores and the cache as if it was a single core. However, if the data from the PPC's to the cache are different, a soft error occurred and the simulation is stopped. The probability a soft error is introduced into both cores at the same time is negligible small. In Table 7.9 the simulation results are shown.

Program	faults injected	soft errors	detected	missed
cjpeg	45	42	42	0
adpcm	45	41	41	0
mesa	45	44	44	0
pgp	45	45	45	0
Total	180	172	172	0

Table 7.9: PPC duplication

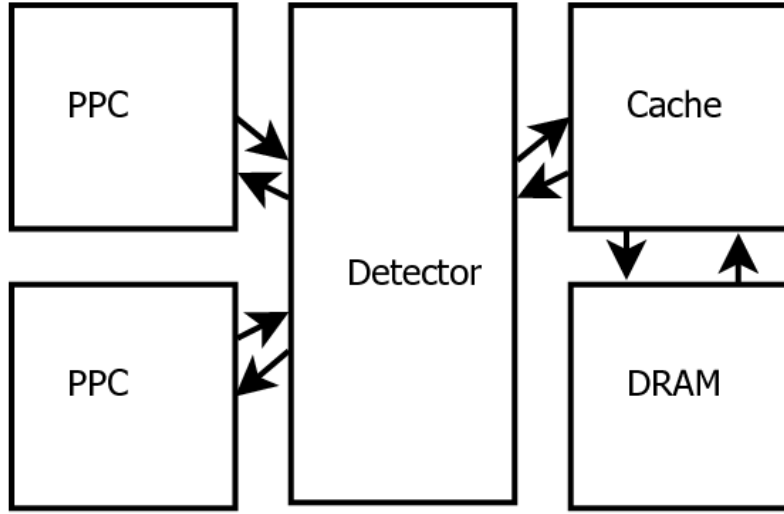


Figure 7.7: Duplication of Power PC (PPC)

7.7 Conclusions

In this chapter several fault techniques have been implemented and tested using the proposed fault injector. It is proposed that instruction precomputation can be used as a fault tolerance technique. A novel method of estimating the reduction in soft error rate is proposed. The SER can be estimated using

$$SER_{new} = (1 - hit_rate) \times SER_{ALU}$$

. Memoization is used as fault tolerance techniques, in this case a disadvantage is the fact erroneous results can be reused. Thus not lowering the soft error rate. A countermeasure could be to limit the lifetime of instruction in the memoization table.

Duplication in conjuncting with memoization has been implemented. All single cycle soft errors are corrected at the expense of 32% extra clock cycles. For faults lengths longer than two cycles, the fault tolerance of the scheme rapidly decreases. A memory protected by ECC has been implemented. ECC corrects all single bit transient errors at the expense of 18.75% area overhead. Finally duplication of two whole PPC cores at a much lower granularity has been implemented. All soft errors are detected, but correction techniques should be implemented to become a fault tolerant system.

Conclusions

8.1 Summary

Hardware faults are best qualified by their duration, there are permanent or longterm faults and transient faults. A permanent fault can only be removed by means of replacement. Transient faults are triggered by environmental conditions and rarely do any long lasting damage. They can however produce an erroneous state for a short period of time.

Transient faults have many names, in literature they are mostly known as soft errors. They are caused by single event effects. SEE are defined as any measurable or observable change in state or performance of a microelectronic device, component, subsystem or system resulting from a single energetic particle strike. A soft error is defined as an erroneous output signal from a latch or memory cell that can be corrected by performing one or more normal functions of the device containing the latch or memory cell.

When an energetic particle strikes a semiconductor it deposits energy along the path it travels inside the semiconductor. The amount of energy it deposits is described by the Linear Energy Transfer(LET).

If a particle strikes a DRAM-cell the charge is altered. If the charge is altered from below to above a threshold voltage and vice-versa, causing a state transition from $0 \rightarrow 1$ and $1 \rightarrow 0$ respectively.

When a particle strikes a sensitive region in an SRAM-cell, charge collected by the junction results in current flowing through the restoring transistor. Since this transistor has a limited amount of current drive and channel conductance, a voltage drop at its drain occurs. This voltage drop is the main source for upsets in SRAM cells.

A strike in combinational logic can generate a transient fault and if captured by a sequential element can cause an error. It might however not be received as an error by the software by one of the following reasons:

1. Logical masking, the output of the gate is determined by the other inputs.
2. Temporal masking, the erroneous output of a gate is not captured by a latch.
3. Electrical masking, the bandwidth of the transient is higher than the cutoff frequency of the CMOS circuit and the transient dies out.

Because of decreasing feature size, the operating voltage and the critical charge of transistors are lowered. These facts make modern semiconductors more susceptible to soft errors. Fault tolerance techniques are much needed.

Nowadays researchers usually implement their own fault injectors on simulation platforms they use for the design evaluation. This sometimes requires very much time and

effort. A universal solution is very desirable, which could be easily adapted for different designs. Although it is impossible to create a fault injector compatible with all the existing simulators, there exist structural frameworks such as Unisim [1], that can be used for fast prototyping and are even able to inter operate with already existing simulators. A universal fault injector for the Unisim environment is proposed, which can be used in Unisim-based designs of different granularity with a minimal effort.

The fault injector can simulate transient faults of different lengths, multi-bit or single bit. These faults can be simulated into memory elements and combinatorial logic. In this project a simulator of a MIPS processor has been developed. The MIPS processor simulator is extended to a superscalar architecture. The simulator is used in conjunction with the fault injector, to test several fault tolerance techniques.

Instruction precomputation is a technique used for improving a processor's performance [31]. It avoids the re-execution of the most frequent used instructions by using profiling information collected off-line. However, we propose instruction precomputation can also be used for fault tolerance. The soft error rate of the combined ALU can be estimated by:

$$SER = (1 - hit_rate) * SER_{ALU} + hit_rate * SER_{P-table}$$

Instruction memoization is another technique used for improving a processor's performance. Memoization avoids redundant computations by the results of previous executions. A drawback is the fact a faulty result can also be stored and then reused. This means memoization itself won't improve the soft error rate.

Instruction duplication is fault tolerance technique, it uses time redundancy to detect and correct errors. However, instruction duplication seriously degrades performance [10]. A counter measure may be the use of memoization. Instruction duplication with memoization will detect all single cycle errors. However not all multi-cycle errors will be detected.

Error correcting memory can be used for storing the information for memoization or precomputation. This will lower the probability a soft error occurs in the memory element. Error corrected memory using Hamming encoding has been tested and in simulations of single bit upsets it corrects all soft errors.

Finally duplication of Power PCs is tested. Two whole Power PC processors, running the same program, are connected to a detector. The detector and the two processors effectively act as one processor, until the two processors produce different results. The detector detects it and counter measures can be taken to correct the result. Since the probability the two processors produce the same erroneous result caused by an injected fault is negligibly small, the detector will detect all faults.

8.2 Conclusion

Not every fault becomes an error. In combinatorial circuits there is logical, temporal and electrical masking. Furthermore in both combinatorial and sequential circuits there is application masking. If an erroneous value is never used, the program runs without problems.

A software based simulation is a good start for investigating a fault tolerance technique. However one can not make realistic estimates about the soft error rate of a

semiconductor device. This is caused by the fact one has to know everything about the processor. One has to know the size and depth of every path in the processor. This would give so much data a simulation can never be done in reasonable time. With the proposed injector it is however possible to make a good estimate of the reduction in soft error rate.

The fault injector can be used to inject faults into busses, logic and memories. It can be placed into a separate module of the simulator or be integrated into existing modules.

If the fault injector is used as a separate module, placed in between two modules, a drawback is the fact that it doesn't know what the module to inject faults into is doing. For example, a multiplication in an ALU is computed by adding partial products in multiple cycles. If a fault occurs in one of the cycles, the outcome will be erroneous. If the fault injector injects in one of these cycles, the outcome will still be correct. A solution to this is integrating the fault injector into the ALU module in this case.

An advantage of the fault injector in the Unisim environment is the fact you can test the fault tolerance at a very early stage. No manufacturing is necessary so the costs are very low. The simulation time for the fault injector in the Unisim environment is however very large. A hardware simulation would be much faster. With the proposed fault injector it is possible to inject faults in precisely the desired location and at the desired time. In hardware simulation this is not possible. Furthermore a fault can be precisely replicated by the injector. Thus meaning it is an excellent tool for fault tolerance researchers to simulate their design and calculate the improvement in SER.

It is proposed processor speed up techniques such as precomputation can be used also for fault tolerance. Using the superscalar MIPS simulator and the fault injector the fault tolerance properties of precomputation have been confirmed. The relationship between the hit rate and the SER of a precomputation table can be estimated and is said to be

$$SER = (1 - hit_rate) * SER_{ALU} + hit_rate * SER_{P-table}$$

Through simulations this relationship is confirmed.

8.3 Future Work

A drawback for the fault injector is that if it is used as a separate module it doesn't have information about the module it is injecting faults into. This means you cannot inject faults at a different rate for different instructions. If an ALU is responsible for shift instructions and integer arithmetic, it is likely to have different soft error rates for either one. It can be possible by for example breaking up the ALU into different parts, such as a multiplier an adder and so on. But this would slow down simulation even more. A solution to this problem might be to inject faults at the rate of the longest path, thus the highest fault frequency. Read the instruction executed in the fault injector, and only produce a fraction of the errors. For example, inject faults at a frequency of once every 10000 cycles for an adder and assume the SER of the shift logic is halve the adder, only allow half of the faults to inject if a shift instruction is executed.

Furthermore if an environment has further divided the clock signal, thus not only calculating on the start and end of a clock cycle, the faults could also be injected more

sophisticated. This is however defined by the simulation environment. The injector could however easily be ported to another structural simulation environment.

The scalar and superscalar MIPS processor simulator could be extended to support system calls, so a file system and an memory management system could be implemented. This would mean larger programs could be run and furthermore also faults can be injected into these stages and thus the reduction in SER of a complete system could be estimated.

Bibliography

- [1] August. D, Chang.J., Girbal. S., Gracia-Perez.D., Mouchard. G., Penry. D. Temam. O., and Vachharajani. N., “ UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development”, *Computer Architecture Letters*, vol. 6, pp. 45–48, 2007.
- [2] L.W. Dodd, P.E.; Massengill, “Basic mechanisms and modeling of single-event upset in digital microelectronics”, pp. 583–602, 2003.
- [3] R.C. Baumann, “Radiation-Induced Soft Errors in Advanced Semiconductor Technologies”, *IEEE Transactions on Device and Materials Reliability*, vol. 5, pp. 305–316, 2005.
- [4] P.E. Dodd and L.W. Massengill, “Modeling and Simulation of Single Event Effects in Digital Devices and ICs”, vol. 4, pp. 1854–1878, 2008.
- [5] “Wikimedia”, <http://www.wikimedia.com/>.
- [6] T. Juhnke and H. Klar, “Calculation of the Soft Error Rate of Submicron CMOS Logic Circuits”, vol. 30, pp. 830–834, 1995.
- [7] Subhasish Mitra and Edward J. McCluskey, “WHICH CONCURRENT ERROR DETECTION SCHEME TO CHOOSE”, pp. 985–994, 2000.
- [8] “UNISIM: UNIted SIMulation environment webpage”, <http://unisim.org/>.
- [9] J.L. Hennessy and D.A.Patterson, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann, 3 edition, 2003.
- [10] D. Borodin, B.H.H. Juurlink, and S. Kaxiras, “Instruction Precomputation for Fault Detection”, in *DSD-2009: 12th Euromicro Conf. on Digital System Design*, Aug 2009, pp. 91–99.
- [11] Feng Qin, Shan Lu, and Yuanyuan Zhou, “SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs”, *IEEE Electron*, vol. 2, 2005.
- [12] RE Lyons and W Vanderkulk, “The use of triple-modular redundancy to improve computer reliability”, *IBM*, 1962.
- [13] J.L. Kaschmitter, D.L. Shaeffer, N.J. Colella, C.L. McKnett, and P.G. Coakley, “Operation of commercial R3000 processors in the Low Earth Orbit(LEO) space environment”, vol. 38, 1991.
- [14] D.L. Shaeffer, J.R. Kimbrough, S.M. Denton, J.L. Kaschmitter, J.W. Wilburn, R.W. Davis, N.J. Colella, and D.B. Holtkamp, “High energy proton SEU test results for the commercially available MIPS R3000 microprocessor and R3010 floating point unit”, vol. 38, 1991.

- [15] G. Kanawati and J. Abraham, "FERRARI: A tool for the validation of system dependability properties", *Int. Symp. Fault Tolerant Comput. Boston*, p. 336345, 1992.
- [16] Joo Carreira, Henrique Madeira, and Joo Gabriel Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers", 1998.
- [17] S. Han, H.A. Rosenberg, and K.G. Shin, "DOCTOR: An Integrated Software Fault InjeCTiOn EnviRonment", 1993.
- [18] E. Jenn, J. Arlat, m. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: the MEFISTO tool", 1994.
- [19] L.W. Nagel and D.O. Pederson, "SPICE (Simulation Program with Integrated Circuit Emphasis)", pp. 985–994, apr 1973.
- [20] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling", *IEEE*, vol. 2, pp. 59–67, 2002.
- [21] J. Bar, J. Levinson, M. Victoria, and W. Hajdas, "Direct Processes in the Energy Deposition of Protons in Silicon", *IEEE Nuclear science*, vol. 43, pp. 2820–2826, 1996.
- [22] C. M. Hsieh, P. C. Murley, and R. R. OBrien, "A field-funneling effect on the collection of alpha-particle-generated carriers in silicon devices", *IEEE High-Performance Computer Architecture*, vol. 11, pp. 103–105, 1981.
- [23] C. M. Hsieh, P. C. Murley, and R. R. OBrien, "Collection of charge from alpha-particle tracks in silicon devices", *IEEE Electron*, vol. 30, pp. 668–693, 1983.
- [24] E. Takeda, K. Takeuchi, D. Hisamoto, T. Toyabe, K. Ohshima, , and K. Itoh, "A cross section of α -particle-induced soft-error phenomena in VLSIs", *IEEE Trans. Electron. Devices*, vol. 36, pp. 2567–2575, 1989.
- [25] E. Takeda, K. Takeuchi, D. Hisamoto, T. Toyabe, K. Ohshima, and K. Itoh, "A soft error rate model for MOS dynamic RAMs", *IEEE J. Solid-State Circuits*, vol. 17, pp. 362–367, 1982.
- [26] T. V. Rajeevakumar, W. Henkels N. Lu, W. Hwang, and R. Franch, "A new failure mode of radiation-induced soft errors in dynamic memories", *IEEE Electron. Device Lett.*, vol. 9, pp. 644–646, 1988.
- [27] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong, "Characterization of Multibit Soft Error Events in Advanced SRAMs", *Proc. IEEE Intl Electronic Device Meeting*, pp. 519–522, 2003.
- [28] T. Heijmen, "Analytical semi-empirical model for SER sensitivity estimation of deep-submicron CMOS circuits", *Proceedings of the 15th IEEE International On-line Testing Symposium*, 2006.

- [29] P. Hazucha and C. Svensson, “Impact of CMOS technology scaling on the atmospheric neutron soft error rate”, *IEEE Transactions on Nuclear Science*, vol. 47, pp. 2586–2594, 2000.
- [30] Arijit Biswas, Paul Racunas, Razvan Cheveresan, Joel Emer, Shubhendu S. Mukherjee, and Ram Rangan, “Computing Architectural Vulnerability Factors for Address-Based Structures”, 2005.
- [31] J. Yi Joshua, Resit Sendag, and David J. Lilja, “Increasing Instruction-Level Parallelism with Instruction Precomputation”, in *Euro-Par 2002 Parallel Processing*, Jan 2002, pp. 27–58.
- [32] D. Michie, “Memo Functions and Machine Learning”, vol. 218, pp. 19–22, 1968.
- [33] A. Parashar, S. Gurumurthi, , and A. Sivasubramaniam, “A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy”, in *ISCA-04: Proc. of the 31st Annual Int. Symp. on Computer Architecture*, 2004, pp. 376–386.
- [34] R.W. Hamming, “Error Detecting and Error Correcting Codes”, *The Bell System Technical Journal*, vol. 29, 1950.

Curriculum Vitae



NC Veerman was born in Amsterdam, on may 10th 1981. He obtained his VWO exam in 1999 at the Hervormd Lyceum West in Amsterdam. He did his bachelor in electrical engineering at the Faculty of Electrical Engineering, Mathematics and Computer Science of the Delft University of Technology.