UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

EDUARDO LUIS RHOD

PROPOSAL OF TWO SOLUTIONS TO COPE WITH THE FAULTY BEHAVIOR OF CIRCUITS IN FUTURE TECHNOLOGIES

Porto Alegre
2007

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

EDUARDO LUIS RHOD

PROPOSAL OF TWO SOLUTIONS TO COPE WITH THE FAULTY BEHAVIOR OF CIRCUITS IN FUTURE TECHNOLOGIES

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, da Universidade Federal do Rio Grande do Sul, como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.

Área de concentração: Automação e Instrumentação Eletro-Eletrônica

ORIENTADOR: Luigi Carro

Porto Alegre

2007

EDUARDO LUIS RHOD

PROPOSAL OF TWO SOLUTIONS TO COPE WITH THE FAULTY BEHAVIOR OF CIRCUITS IN FUTURE TECHNOLOGIES

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: ___________________________________

Prof. Dr. Luigi Carro, UFRGS

Doutor pelo Programa de Pós-Graduação em Ciência da Computação, UFRGS, Porto Alegre, Brasil

Banca Examinadora:

Prof. Dr. Marcelo Lubaszewski, UFRGS

Doutor pelo Institut National Polytechnique de Grenoble, Grenoble - França

Prof. Dr. Walter Fetter Lages, UFRGS

Doutor pelo Instituto Tecnológico de Aeronáutica (ITA), Rio de Janeiro - Brasil

Prof. Dra. Fernanda Lima Kastensmidt, UFRGS

Doutora pela Universidade Federal do Rio Grande do Sul, Porto Alegre - Brasil

Coordenador do PPGEE: _______________________________

Prof. Dr. Marcelo Lubaszewski.

Porto Alegre, 25 de abril de 2007.

DEDICATÓRIA

Dedico este trabalho a todos os meus familiares em especial aos meus pais, não só pelo esforço em garantir minha educação, mas também pelo apoio tanto material quanto afetivo. Dedico também a minha namorada e toda a sua família pelo carinho e companheirismo.

AGRADECIMENTOS

Ao Programa de Pós-Graduação em Engenharia Elétrica, PPGEE, e a todos os funcionários da secretaria do curso de Engenharia Elétrica, pelo profissionalismo e seriedade com que exerceram suas atividades. Agradeço em especial a secretária do PPGEE, Miriam Rosek e ao professor Marcelo Lubaszewski, coordenador do PPGEE. Agradeço também a todos os professores e colegas de turma que contribuíram enormemente na minha formação durante o mestrado. Em especial aos professores Flávio Wagner, Erika Cota, Altamiro Susin Agradeço aos colegas de aula Douglas Stein, Márcio Oliveira, Eduardo Brião, Victor Gomes, Fábio Wronski e Elias T. S. Júnior. Aos colegas de laboratório além dos citados acima, Edgar F. Correa, Antonio C. S. B. Filho, Júlio C. B. Mattos, Marco Wehrmeister, Mateus Rutzig, Rodrigo Motta e Dalton Colombo pelas enormes contribuições e ajudas fornecidas durante os trabalhos de pesquisa. Agradeço aos membros do laboratório SiSC da PUCRS por terem me iniciado na pesquisa como bolsista de Iniciação Científica. Ao CNPQ pela provisão da bolsa de mestrado, sem a qual não teria seguido com os estudos.

Agradeço em especial ao meu orientador Luigi Carro, por acreditar no meu trabalho e por conduzir exemplarmente a minha orientação, me conduzindo na busca por um trabalho de qualidade, mas ao mesmo tempo dando espaço para que eu exercesse a minha pesquisa com criatividade e liberdade. Ao amigo e colega de pesquisa Carlos Arthur Lang Lisboa, pelos conhecimentos compartilhados e pela enorme ajuda e contribuição para que os nossos resultados e publicações atingissem o nível que atingiram.

Agradeço a todos os meus familiares e amigos de Lajeado pelo apoio e compreensão da minha ausência nos diversos momentos em que estive dedicado aos meus estudos. Agradeço a minha namorada Alexandra Barcelos e seus familiares pelo companheirismo e

amor. Em especial agradeço aos meus pais Pedro Valentin Rhod e Marta Regina Blasi Rhod pela educação e por todas as oportunidades que me deram sem nunca medir esforços. Agradeço os meus irmãos Guilherme Blasi Rhod e Amanda Rhod por fazerem parte na minha vida.

RESUMO

A diminuição no tamanho dos dispositivos nas tecnologias do futuro traz consigo um grande aumento na taxa de erros dos circuitos, na lógica combinacional e seqüencial. Apesar de algumas potenciais soluções começarem a ser investigadas pela comunidade, a busca por circuitos tolerantes a erros induzidos por radiação, sem penalidades no desempenho, área ou potência, ainda é um assunto de pesquisa em aberto. Este trabalho propõe duas soluções para lidar com este comportamento imprevisível das tecnologias futuras: a primeira solução, chamada MemProc, é uma arquitetura baseada em memória que propõe reduzir a taxa de falhas de aplicações embarcadas micro-controladas. Esta solução baseia-se no uso de memórias magnéticas, que são tolerantes a falhas induzidas por radiação, e área de circuito combinacional reduzida para melhorar a confiabilidade ao processar quaisquer aplicações. A segunda solução proposta aqui é uma implementação de um IP de infra-estrutura para o processador MIPS indicada para sistemas em chip confiáveis, devido a sua adaptação rápida e por permitir diferentes níveis de robustez para a aplicação. A segunda solução é também indicada para sistemas em que nem o hardware nem o software podem ser modificados. Os resultados dos experimentos mostram que ambas as soluções melhoram a confiabilidade do sistema que fazem parte com custos aceitáveis e até, no caso da MemProc, melhora o desempenho da aplicação.

Palavras-chaves: Arquiteturas tolerantes a falhas, arquiteturas baseadas em memória, SoCs confiáveis, técnicas de detecção de erros, taxa de *soft error*.

ABSTRACT

Device scaling in new and future technologies brings along severe increase in the soft error rate of circuits, for combinational and sequential logic. Although potential solutions are being investigated by the community, the search for circuits tolerant to radiation induced errors, without performance, area, or power penalties, is still an open research issue. This work proposes two solutions to cope with this unpredictable behavior of future technologies: the first solution, called MemProc, is a memory based architecture proposed to reduce the fault rate of embedded microcontrolled applications. This solution relies in the use magnetic memories, which are tolerant to radiation induced failures, and reduced combinational circuit area to improve the reliability when processing any application. The second solution proposed here is an infrastructure IP implementation for the MIPS architecture indicated for reliable systems-on-chip due to its fast adaptation and different levels of application hardening that are allowed. The second solution is also indicated for systems where neither the hardware nor the software can be modified. The experimental results show that both solutions improve the reliability of the system they take part with affordable overheads and even, as in the case of the MemProc solution, improving the performance results.

Keywords: Fault tolerant architectures, memory based architectures, reliable SoCs, error detection techniques, soft error rate.

SUMMARY

1. INTRODUCTIONí ... 14
2. CONTEXT OF THE RESEARCHí 18
2.1 RADIATION SOURCES AND THEIR EFFECTSí í í í í í í í í í í í í í í í í 18
2.1.1 Sources of Radiationí .. 18
2.1.1.1 Alpha Particlesí 19
2.1.1.2 High Energy Cosmic Neutronsí í í í í í í í í í í í í í í í í í .. 19
2.1.1.3 Boron Fission Induced by Low Energy Neutronsí í í í í í í í í 19
2.1.2 Effects of SEUs and SETs in Digital Circuitsí í í í í í í í í í í í í . 20
2.2 Metrics to Evaluate the Vulnerability of Circuits to Soft Errorsí í í í í . 21
2.2.1 Failures in Time (FIT)í .. 22
2.2.2 Mean Time to Failure ó MTTFí í í í í í í í í í í í í í í í í í í 22
2.2.3 The Soft Error Rate Estimationí í í í í í í í í í í í í í í í í í í ... 22
2.3 Mitigation Techniques for SEUs and SETsí í í í í í í í í í í í í í 24
2.3.1 Process Modification Related Techniquesí í í í í í í í í í í í í 25
2.3.2 Component Hardening Techniquesí í í í í í í í í í í í í í í í í 26
2.3.3 Circuit Design SEU and SET Hardware Mitigation Techniquesí í í í .. 27
2.3.3.1 Hardware Error Detection Techniquesí í í í í í í í í í í í í 28
2.3.3.2 Hardware Error Detection and Correction Techniquesí í í í í í .. 29
2.3.3.2.1 Triple Modular Redundancy ó TMRí í í í í í í í í í í í í í í 29
2.3.3.2.2 Error Detection and Correction Code ó EDACí í í í í í í í í í 31
2.3.4 SEU and SET Error Mitigation Techniques for Software-Based Systemsí .. 32
2.3.4.1 Software Implemented Hardware Fault Tolerance (SIHFT) techniques... 33
2.3.4.2 Hardware Techniques for Software-Based Systemsí í í í í í í í í . 36
2.3.4.2.1 Dynamic Implementation Verification Architecture ó DIVAí í í í ... 36
2.3.4.2.2 Simultaneous and Redundantly Treaded (SRT) Processorí í í í í í 37
2.3.4.3 Hybrid Techniquesí .. 38
3. USING MEMORY BASED CIRCUITS TO COPE WITH SEUS AND SETS..... 40
3.1 4x4-bit Memory Based Multiplierí 41
3.2 4-tap, 8-bit FIR Filter Memory Based Circuití í í í ..í í í í í í í í í 47
4. MEMPROC: A MEMORY BASED, LOW-SER EFFICIENT CORE PROCESSOR ARCHITECTUREí . 51
4.1 The MemProc Architectureí .. 51
4.1.1 The Macroinstructionsí í í í í í í í í í í í ..í í í í í í í í í í í 52
4.1.2 The Microcodeí 53
4.1.3 The Arithmetic and Logic Unit ó ALUí í í í í í í í í í í í í í í í í í 53
4.2 Design Strategies that Improved Performanceí í í í í í í í í í í í í í ... 56
4.3 Code Generationí 58
5. MEMPROC: EXPERIMENTAL RESULTSí í í í í í í í í í í í í í í í í í 60
5.1 Architectures compared with MemProcí í í í í í í í í í í í í í í í í í .. 60
5.2 Tools Used in the Fault Injection, Performance and Area Evaluationí í í í . 61
5.3 Fault Rate and Area Evaluationí ... 64
5.4 Performance Evaluationí í í í í í í í í í í ..í í í í í í í í í í í í . 68

6. I-IP: A NON-INTRUSIVE ON-LINE ERROR DETECTION TECHNIQUE FOR SOCSí **72**
6.1 The Proposed Approachí **72**
6.1.1 The I-IPí ... **73**
6.1.2 The I-IP Modulesí ... **76**
6.2 Processor and Application Adaptations for MIPSí í í í í í í í í í í í . **79**
7. I-IP EXPERIMENTAL RESULTSí **81**
7.1 Fault Injection Experimentsí . **81**
7.2 Result Analysisí ... **83**
8. CONCLUSIONS AND FUTURE WORKí í í í í í í í í í í í í í í í í .**86**
8.1 Conclusionsí . **86**
8.2 Future Workí ... **87**
REFERENCESí .**89**
APENDIX A: MEMPROC LIST OF INSTRUCTIONSí í í í í í í í í í í í .**94**
APENDIX B: MEMPROC ARCHITECTURE DESCRIBED IN CACO-PS TOOL...96
APENDIX C: MIPS ARCHITECTURE DESCRIBED IN CACO-PS TOOLí í ..**97**

LIST OF FIGURES

Figure 1.1: Evolution of SER: SRAM vs. logicí í í í í í í í í í í í 15
Figure 2.1: Boron fission induced by low energy neutroní í í í í í í í í í í ... 20
Figure 2.2: Sequential circuití ... 20
Figure 2.3: Combinational circuit without radiation (a) and with radiation (b)í í í ... 21
Figure 2.4: SRAM cell hardened by the inclusion of two feedback resistorsí í í í .. 27
Figure 2.5: Detection of an SEU in a memory element (a) and detection of an SET in a combinational circuit (b) by using space (or hardware) redundancy.í í í 28
Figure 2.6: Use of time redundancy to detect an SET in a combinational circuití í í 29
Figure 2.7: Use of space redundancy to detect an SET in a combinational circuití í .. 30
Figure 2.8: TMR with time redundancyí í í í í í í í í í í í í í í í í í ... 31
Figure 3.1: AND truth tableí ..41
Figure 3.2: Fully combinational 4x4-bit multiplierí .í í í í í í í í í í í í í í .42
Figure 3.3: Column multiplier circuití í í í í í í í í í í ..í í í í í í í í í .43
Figure 3.4: Line multiplier circuití 44
Figure 3.5: Combinational circuit for the 8-bit FIR filter with 4 tapsí í í í í í í ..47
Figure 3.6: 8-bit FIR filter with 4 taps using memoryí í í í í í í í í í í í 48
Figure 4.1: MemProc overall architectureí í í í í í í í í í í í í í í í í ... 51
Figure 4.2: Macroinstruction formatí 52
Figure 4.3: Microinstruction formatí .53
Figure 4.4: ALU for one bit operationí ..54
Figure 4.5: Operation masks used during the addition operationí ...í í í í í í í í . 55
Figure 4.6: 2-bit addition using MemProc ALUí í í í í í í í í í í í í í í í í .. 55
Figure 4.7: 8-bit addition paradigmí . 56
Figure 4.8: Code generation process for MemProcí í í í í í í í í í í í í í í . 59
Figure 5.1: FemtoJava pipeline block schemeí í í í í í í í í í í í í í í í í . 60
Figure 5.2: The MIPS pipeline architectureí í í í í í í í í í í í í í í í í í . 61
Figure 5.3: Error detection schemeí .. 64
Figure 5.4: Mean time to execute each type of instruction for all applicationsí í í ... 69
Figure 5.5: The way MemProc does comparisonsí í í í í í í í í í í í í í í ... 70
Figure 6.1: I-IP overall architectureí 74
Figure 6.2: Original instructioní 74
Figure 6.3: Source operands and result fetchingí í í í í í í í í í í í í í í í .. 74
Figure 6.4: Architecture of the I-IPí .. 78
Figure 6.5: Error detection schemeí .. 82

LIST OF TABLES

Table 2.1: Architectural Vulnerability Factor (AVF) estimation approachesí ...í í ..24
Table 3.1: Area for each solution in number of transistorsí í í í í í í í í .í ... 44
Table 3.2: Architectural Vulnerability Factor and timing results for single and double Faultsí ...46
Table 3.3: Area results for the filter implementations, in number of transistors...í í . 48
Table 3.4: AVF results for single faults in FIR filter implementationsí í í ..í í í 49
Table 5.1: Area and time between fault injectionsí í í í í í í í í í í í í í ..65
Table 5.2: Fault rates for all architecturesí í í í í í í í í í í í í í í í ...67
Table 5.3: Performance when executing benchmark applicationsí í í í í í í í ..68
Table 6.1: Runtime frequency of instructionsí í í í í í í í í í í í í í í .83
Table 6.2: Error detection results for the two architecturesí í í í í í í í í í í 83

LIST OF ABREVIATIONS

AVF Architectural Vulnerability Factor

ALU Arithmetic and Logic Unit

BPSG Boron Phospho-Silicate Glass

BCH Bose-Chaudhuri-Hocquenghem

CACO-PS Cycle-Accurate Configurable Power Simulator

CCA Control Flow Checking using Assertions

CFCSS Control Flow Checking by Software Signatures

CMOS Complementary Metal-Oxide-Semiconductor

DIVA Dynamic Implementation Verification Architecture

DWC Duplication with Comparison

DSP Digital Signal Processing

ECCA Enhanced Control Flow Checking using Assertions

EDAC Error Detection and Correction Code

ED^4I Error Detection by Data Diversity and Duplicated Instructions

FIR Finite Impulse Response

FIT Failures in Time

FRAMs Ferroelectric Random Access Memories

GCC GNU Compiler Collection

LET Linear Energy Transfer

MBU Multiple Bit Upset

MIF Memory Initialization File

MRAMs	Magnetic Random Access Memories
MTTF	Mean Time to Failure
N-MR	Modular Redundancy of order N
IMDCT	Inverse Modified Discrete Cosine Transform
IP	Intellectual Propriety
I-IP	Infrastructure-IP
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RS	Reed-Solomon
SDC	Silent Data Corruption
SE	Soft Error
SER	Soft Error Rate
SETs	Single Event Transient
SEUs	Single Event Upsets
SIHFT	Software Implemented Hardware Fault Tolerance
SIMD	Single Instruction Multiple Data Processor
SoC	System-on-a-Chip
SOI	Silicon-on-Insulator
SRAM	Static Random Access Memory
TMR	Triple Modular Redundancy
TVF	Timing Vulnerability Factor
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Large Instruction Word

1 INTRODUCTION

The constant growth of the semiconductor industry in the past years has led to a great improvement in the fabrication of circuits with smaller and faster transistors. This new technology era allows the fabrication of transistors with 100 nm and even smaller dimensions. It allows the integration of billions of transistors in the same chip, giving the designer the possibility to implement more functions in the same device. In this new scenario, designers are developing systems that use more than one processing component in the same chip, with ever growing computation capabilities. These systems are called *system-on-chip* (SoC) and are used in the development of embedded systems such as cell phones, palm tops, GPS systems, etc.

However, the technology improvement is bringing an increased concern regarding the reliability of these new circuits. Although the good advance in terms of performance, these new generations of technologies are more sensible to process variations due to their reduced dimension transistors. Also, high energy particle strikes, such as neutrons from cosmic rays and alpha particles from packaging material, once a concern only for spatial application devices, are now becoming important sources of radiations that are affecting not only memory components but also logic components at low altitude and even at sea level. These strikes can produce or stimulate bit flips, also known as *single event upsets* (SEUs), or generate transient pulses, known as *single event transient* (SETs), which in certain circumstances, can compromise the correct functionality of the circuit, provoking *soft errors* (SE). A soft error is a random error induced by an event that corrupts the data stored in or produced by the device, but does not damage the device itself.

Not only the number of transistors, but also the chip density, in number of transistors per area unit, has been growing exponentially in the past years. This fact has given researchers a new concern related to multiple faults caused by a single particle hit, which is

called *multiple bit upset* (MBU). This phenomenon, in the past present only at memory devices due to its high density, is now affecting the logic part of a circuit.

The reduced size of transistors provided by nanotechnology circuits makes them faster than the ones in the previous technologies, which allows the circuit to run at higher clock frequencies. This improvement in the clock frequency increases the number of operations that can be performed per time unit. On the other hand, with higher frequencies and consequently lower periods, the circuit is more likely to propagate a transient pulse to generate a bit flip or even a multiple bit flip, according to the number of outputs generated by the hit component. As shown in Figure 1.1, from (BAUMANN, 2005), while the *soft error rate* (SER) of SRAM memories remains almost stable with technological scaling, the SER of logic has been always increasing. This new scenario makes architects more concerned with the impacts of soft errors on their designs. In future and even in todayøs circuits, the SER is becoming as important as the performance or power characteristics. In order to survive in this scenario, it is clear that new fault tolerance techniques must be defined, not only for safety critical systems, but to general purpose computing as well.

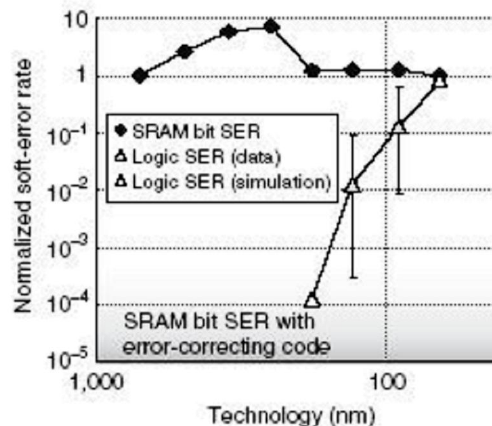


Figure 1.1: Evolution of SER: SRAM vs. Logic, from (BAUMANN, 2005)

Current fault tolerance techniques are effective, with some overhead, for SEUs and SETs. However, they are unlikely to withstand in an efficient way the occurrence of multiple

simultaneous faults that is foreseen with those new technologies (CONSTANTINESCU, 2003; EDENFELD, 2004). To face this challenge, either completely new materials and manufacturing technologies will have to be developed, or fully innovative circuit design approaches must be taken.

Several techniques have been proposed to mitigate SEUs and SETs. There are techniques in all stages of a circuit production, from process modifications to hardware and software design techniques for dedicated systems or general purpose ones. Most of these techniques are able to reduce significantly the number of faults, with some performance and/or area and/or power overheads. Process variation solutions usually are too expensive for low production volumes. Generally, hardware system solutions tend to have a considerable cost in area, while software system solutions somehow affect the resulting performance of the circuit. Thus, the search for reliability in digital systems still lacks efficient solutions, and therefore there is still space for solutions that cope with single and multiple faults without adding undesirable costs to the system development.

Geometric regularity and the extensive use of regular fabrics are being considered as a probable solution to cope with parameter variations and improve the overall yield in manufacturing with future technologies. Regularity brings the reduction of the cost of masks, and also allows the introduction of spare rows and columns that can be activated to replace defective ones in memory circuits (SHERLEKAR, 2004). Together with the proposal of using regular fabrics, the introduction of new memory technologies that can withstand the effects of transient faults, such as ferroelectric and magnetic RAMs (FRAMs and MRAMs, respectively) (ETO, 1998), brings back the concept of using memory to perform computations.

In this work, the use of memory is proposed as a novel mitigation technique for transient faults, by reducing the area of the circuits that can be affected by soft errors. This

way, this work introduces a processor architecture to cope with the SEU/SET problem without imposing any performance overhead, while favoring a regular architecture that can be used to enhance yield in future manufacturing processes. The proposed architecture is a memory-based embedded core processor, named MemProc, designed for use in control domain applications as an embedded microcontroller.

There are situations in which neither the hardware nor the software can be modified, due to the high costs involved in adding extra hardware or when the source code is not available. In these cases, alternative techniques are needed for providing the system with an adequate level of dependability. To deal with this kind of applications, this work proposes a second alternative to improve the reliability in digital systems, that combines on-line software modifications with a special-purpose hardware module (known as infrastructure IP, or I-IP) which was previously proposed in (BERNARDI, 2006). The development of an I-IP core to improve reliability of the MIPS RISC processor is presented in this work.

This work is divided as follows: in the second chapter the context of this work is reviewed. In the third chapter, the first experiments on using memory based circuits to improve reliability are presented. The fourth chapter describes the developed architecture and its key characteristics that contributed to the good fault tolerance and performance results. The fifth chapter presents the obtained experimental results, in terms of fault coverage, area, and performance. The sixth chapter presents the second solution that was developed to cope with the faulty behavior of future technologies without applying any change to the hardware or the software of the system. The seventh chapter presents the obtained results for the second solution, in terms of fault detection, and its area and performance overhead. In the eighth chapter the conclusions and possible evolution of the work that is presented here are discussed.

2 CONTEXT OF THE RESEARCH

This chapter presents a description of the different types of spatial radiation that can produce or stimulate bit flips in circuits. This chapter is divided in three sections. In the first one, the most commonly found sources of radiation and their effects in digital circuits are discussed. In the second section, the most used metrics that are applied to measure the vulnerability of the circuits are described and, in the third and last section, some of the most used and known techniques applied to detect and mitigate errors in digital circuits are presented and analyzed, together with a discussion of the positive and negative aspects of each technique.

2.1 RADIATION SOURCES AND THEIR EFFECTS

There are different types of space radiation that can cause soft errors. In this section the most known and relevant types of radiation sources that can cause SEUs and SETs are presented, and the effects that SEUs and SETs can cause,. and which are the conditions to an error occur are discussed (HEIJMEN, 2002).

2.1.1 Sources of Radiation

The main sources of radiation catered from space are:

a) alpha particles;

b) high energy cosmic neutrons;

c) boron fission induced by low energy neutrons.

There are other kinds of particles that can cause soft errors, like heavy ions for instance, but they will not be discussed here because they are only relevant for aero-space applications, due to their occurrence only in space or in the highest parts of the earth atmosphere.

2.1.1.1 Alpha Particles

An alpha particle is a doubly ionized helium atom, made of two protons and two neutrons. Alpha particles can be found in circuits packaging materials, solder points of the integrated circuits or in wafers, which are thin slices of semi-conductor material, upon which circuits are constructed. When an alpha particle hits a beta or gamma ray, it looses energy and generates transient current pulses that, depending on their intensity, can cause an SEU (single event upset) which can result in a soft error if it compromises the correct functionality of the circuit.

2.1.1.2 High Energy Cosmic Neutrons

This kind of particle is formed by the collision of galactic particles and solar wind particles with the terrestrial atmosphere. Most of cosmic rays are reflected or captured by the geomagnetic field of the earth, and only 1% of the high energy cosmic neutrons hit the earth surface, generating a flux of 25 neutrons/cm^2.hr (ZIEGLER, 1981) with energy higher than 1 MeV (1 million electron volt) at sea level. Only neutrons with 5 MeV or higher energy are capable of generating soft errors.

2.1.1.3 Boron Fission Induced by Low Energy Neutrons

Another form of radiation can occur when low energy neutrons interact with boron atoms (BAUMANN, 1995). As a result, a lithium core and an alpha particle are generated by fission, as depicted in Figure 2.1. Both particles resulting from this reaction are capable of generating SEUs or SETs that can cause the undesired soft errors.

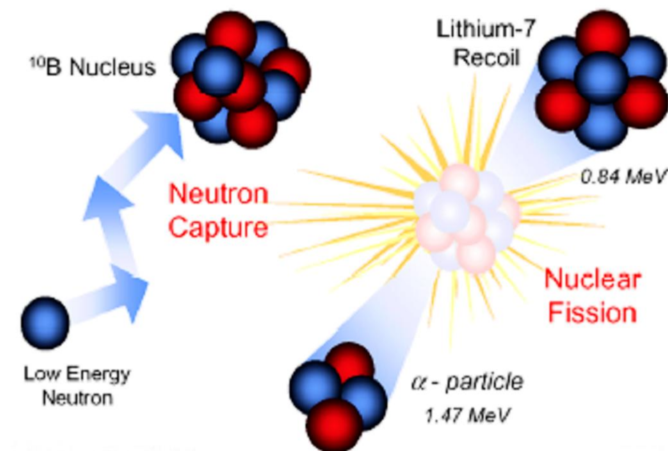


Figure 2.1: Boron fission induced by low energy neutron, from (BAUMANN, 2001).

2.1.2 Effects of SEUs and SETs in Digital Circuits

A particle hit can affect a combinational as much as a sequential part of a circuit (ALEXANDRESCU, 2002). In sequential circuits, like the one shown in Figure 2.2, SEUs can occur only in memory elements (the registers in Figure 2.2). On the other hand, the combinational components can be affected by SETs which, given the right circumstances, can cause an error. The hit of a radiation particle in a memory element does not imply that an SEU will be registered. In order to an SEU occur, it is necessary that this particle has enough charge to create a significant current pulse. In other words, it is necessary that the charge generated by the particle is greater than or equal to the so called critical charge ($Q_{critical}$) of the hit element. The $Q_{critical}$ will be explained with more details in the next section.

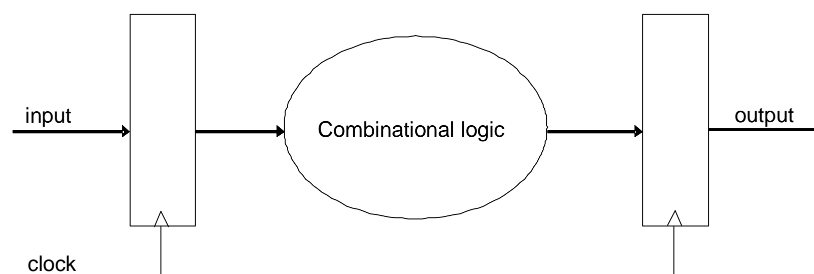


Figure 2.2: Sequential circuit.

The occurrence of an SET in combinational logic, does not mean that an error will result. In order to an error occur, a combination of events must happen, allowing the SET to be captured or generate and erroneous operation. First, it is necessary that the charge generated by the radiation source be equal or higher than the $Q_{critical}$ of the element that was hit. Second, the combinational circuit must be fast enough to propagate the error, and third, the logic of the architecture must allow that the wrong logic value that was generated propagates to some memory element during its latching window or generate an erroneous operation. In Figure 2.3, one can see an example in which the combinational circuit does not allow the SET propagation. The figure shows a little combinational circuit in two different situations. In the first situation (a), the circuit is free of the radiation effects, while in the second (b) the circuit is being affected by a source of radiation. One can see that in booth circuits the result is the same even in the presence of radiation.

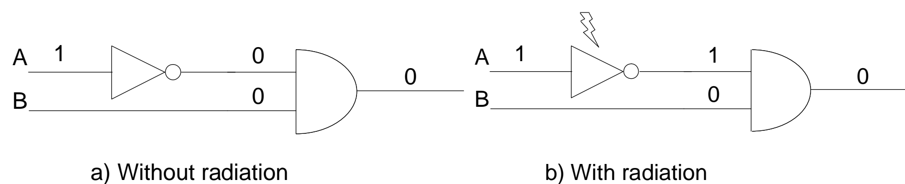


Figure 2.3: Combinational circuit without radiation (a) and with radiation (b).

2.2 METRICS TO EVALUATE THE VULNERABILITY OF CIRCUITS TO SOFT ERRORS

The vulnerability of a circuit to soft errors indicates the probability of the circuit to have an error. This probability indicates to the end user how much he can rely on the correct operation of the circuit. With the growing concern about circuit reliability, companies are using some metrics to evaluate their products. In this section, some of the most used metrics proposed by scientists and designers to evaluate the vulnerability of the circuits, which is known as the soft error rate (SER), are presented.

2.2.1 Failures in Time (FIT)

The fault rate of a circuit can be measured through the number of failures that occur in a certain period of time. This metric is known as *Failures in Time*, or *FIT*. If a circuit has a fault rate of 1 FIT, it means that in a period of 1 billion hours 1 fault will probably occur. Some companies, like IBM, are using this metric as a reference to the design of their products. IBM sets its target for undetected errors caused by SEUs to 114 FIT (BOSSEN, 2002), which means that 1 fault may occur in the time range of about 9 million (8.771.930 to be more precise) hours of device operation. The additive property of FIT makes it convenient for calculation of the fault rate of large systems, because the designer just needs to sum the FIT of all components that are part of the system to have the system FIT.

2.2.2 Mean Time to Failure ó MTTF

Another metric that can be applied to measure the fault rate of a system is the mean time to failure. Differently from the FIT, the MTTF is more intuitive, because it indicates the mean time that will elapse before an error occurs. The MTTF has an inverse relation to the FIT, which is expressed by the following equation :

$$MTTF(hours) = \frac{10^9}{FIT} \quad (1)$$

2.2.3 The Soft Error Rate Estimation

The soft error rate (SER) of a system can also be expressed in terms of the nominal soft error rates of individual elements that are part of the system, such as SRAMs, sequential elements such as flip-flops and latches, combinational logic, and factors that depend on the circuit design and the microarchitecture (NGUYEN, 2003; SEIFERT, 2004), as follows:

$$SER^{design} = \sum_i SER_i^{nominal} \times TVF_i \times AVF_i \quad (2)$$

where i stands for the ith element of the system.

The $SER^{nominal}$ for the ith element is defined as the soft failure rate of a circuit or node under static conditions, assuming that all the inputs and outputs are driven by a constant voltage. The TVF_i, time vulnerability factor (also known as time derating) stands for the fraction of the time that the element is susceptible to SEUs that will cause an error in the ith element. The AVF_i, architectural vulnerability factor (also known as logic derating) represents the probability that and error in the ith element will cause a system-level error.

The $SER^{nominal}$ is defined by the probability of occurrence of an SEU in a specific node of the element. This probability depends on the element type, transistor size, node capacitance and other characteristics of the element. For instance, to estimate the $SER^{nominal}$ for a latch, one must know the $Q_{critical}$, which identifies the minimum charge necessary to cause the element to fail. This can be done by injecting waveforms of alpha an neutron particle hits on all relevant nodes. Then, it is necessary to evaluate the alpha and neutron flux to which the circuit is submitted. More details can be found in (NGUYEN, 2003).

The timing vulnerability factor can be summarized as the fraction of time that the element can fail. For example, the timing vulnerability factor of a latch is equal to the portion of the time that the latch is in its store mode. For combinational logic, the timing vulnerability factor depends on its type, which can be data path or control path. More details on these and other TVF evaluation aspects can be seen in (NGUYEN, 2003; SEIFERT, 2004).

The architectural vulnerability factor of an element can be understood as the probability that a fault in that element causes an error in the system. In Table 2.1 some approaches to estimate the AVF, its major issues, advantages and disadvantages are presented.

Table 2.1: Architectural-vulnerability-factor (AVF) estimation approaches

Approach	Description	Major issues	Advantages	Disadvantages
Fault injection	Inject error(s) and simulate to see if injected error(s) cause(s) system-level error(s) by comparing the system response with simulated fault-free response	* Which inputs to simulate; * How many errors to inject; * Which signals to inject errors in; * Which signals to use for comparison.	* Applicable to any design; * Easy automation.	* Long simulation time (several days or weeks) for statistically significant results; * Dependence on chosen stimuli.
Fault-free simulation	Perform architectural or logic simulation and identify situations that do not contribute to system-level errors, such as unused variables and dead instructions.	* Which inputs to simulate; * How to identify situations that do not contribute to system-level errors.	* Much faster compared to fault injection; * Easy automation.	* Applicable to very specific designs and not general enough; * Dependence on chosen stimuli.

source: (MITRA, 2005)

The AVF value of an element depends on its inputs and also on how important that element is for the circuit considering its functionality. As an example, suppose that the contents of a flip-flop are erroneous. If the flip-flop output drives to an AND gate with another signal whose logic value is 0, the error will have no effect on the output of the AND gate.

2.3 Mitigation Techniques for SEUs and SETs

In the first years of spatial exploration, the reliability of the circuits started to become an important concern for designers. At that time, the major technique used to protect circuits was shielding. This shielding technique worked by reducing the particle flow to smaller levels and consequently, reducing the number of errors caused by particle hit to zero. During many years this technique was widely used in aero-spatial applications and guaranteed the correct

operation of the circuits. However, with the technology evolution up to nanometer scale, circuits became more susceptible to particle hits, making this shielding technique obsolete for special circuits and even for circuits to be used at sea level.

Trying to reach the level of reliability that once belonged to shielding, scientists have proposed several techniques in the past years, each one with its pros and cons, to mitigate SEUs and SETs. In this section, some of these techniques are presented and their costs, in terms of area and processing time overheads, are discussed.

2.3.1 Process Modification related techniques

Several process solutions have been proposed to reduce SER sensitivity of circuits, including the usage of well structures, buried layers, deep trench isolation, and implants at the most sensitive nodes. Also wafer thinning has been proposed as a way to reduce SEU sensitivity (DODD, 2001). It was shown that the overall SEU threshold LET (linear energy transfer) can be significantly increased if the substrate thickness is reduced to 0.5 m. In practice, however, several criteria would have to be met to make the thinning of fully processed wafers possible. Another reduction of the SER can be achieved by reducing to almost zero the contribution of errors caused by the particles resulted by the boron fission reaction. This can be done by eliminating BPSG (boron phosphor-silicate glass) from the process flow. If the use of BPSG is necessary, enriched ^{11}B could be used in the BPSG layers (BAUMANN, 2001). Silicon-on-insulator (SOI) technologies are relatively insensitive to soft errors. Applying SOI technology instead of the corresponding bulk process improves the SER with a factor in the range of 2 to 8 (HARELAND, 2001). However, the cost of materials, especially of the wafers, is higher for SOI. In general, these process modification solutions are expensive and are applied just for a few designs.

2.3.2 Component Hardening Techniques

There are two basic approaches to improve SER sensitivity at the circuit level. On one hand, the components applied in the design can be modified such that they become less susceptible to soft errors. The main goal of this approach, often named design hardening, is to manufacture SER-reliable circuits using standard CMOS processing without additional masks (VELAZCO, 1994). On the other hand, one can accept that soft errors occur at a certain rate and include extra circuitry to detect and correct them. Error detection and correction techniques are discussed in the next subsection.

Solutions to reduce the SER sensitivity of components can be categorized as techniques to increase the capacitance of the storage node, to reduce the charge collection efficiency, or to compensate for charge loss. The applied design style can have an important effect on SER. For instance, in (SEIFERT, 2001) it is demonstrated that level-sensitive latches using transmission gates are more sensitive than edge-triggered static latches, because the former use floating nodes to store information.

Another method to improve SER sensitivity is to enlarge the critical charges by increasing the capacitance of the storage nodes. In fact, if all critical charges are sufficiently large, alpha particles are not able to upset a circuit and neutrons are the only source of soft errors that can affect the circuit. In (KARNIK, 2001), an explicit feedback capacitor is added to the node capacitances. In (OOTSUKA, 1998), a SER-hardened SRAM cell used stacked cross-coupled interconnects to increase the capacitor area. Enlargement of the node capacitances are not only applied in memory design, but were also shown to be an efficient way to improve the SER sensitivity of sequential or domino nodes in high-performance circuits (KARNIK, 2002). The main drawback of increasing the node capacitances is that generally the cell area is increased affecting the memory overall area. The SER sensitivity of SRAM cells and latches can also be improved by adding feedback resistors between the

output of one inverter and the input of the other, as shown in Figure 2.4. This SRAM cell topology was proposed in (SEXTON, 1991). The transient pulse induced by an ionizing particle is filtered by the two resistors, which slow down the circuit such that it does not have sufficient time to flip state. However, the inclusion of feedback resistors in a memory element has the drawback that the write speed is lowered (VELAZCO 1994).

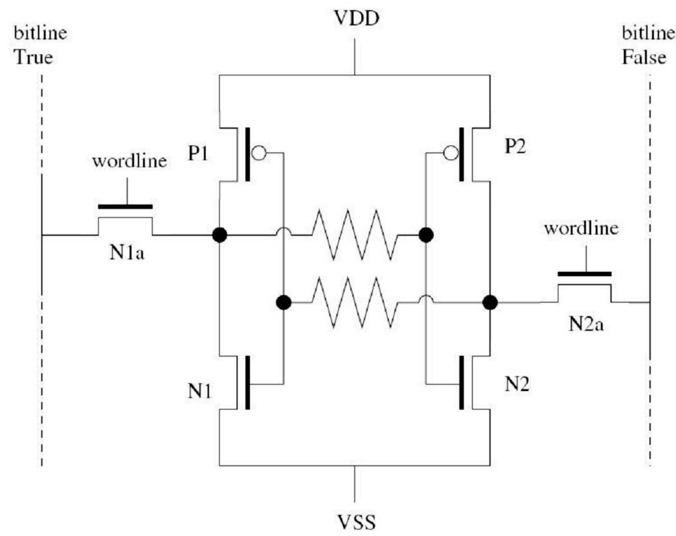


Figure 2.4: SRAM cell hardened by the inclusion of two feedback resistors

2.3.3 Circuit Design SEU and SET Hardware Mitigation Techniques

As stated in a previous subsection, process modification solutions are expensive and are used just in few designs with high volume. Also, component hardening techniques involve costs in energy, area and performance that sometimes may not be reasonable for manufacturers. Therefore, the development of techniques not related to the process variation or component modification has been stimulated during the past years, and some design based mitigation techniques have been proposed for the scientific community. In this section, some of the most know and widely used design techniques that have been proposed by researchers worldwide are presented. These techniques are divided into two main groups: error detection techniques and error detection and correction techniques.

2.3.3.1 Hardware Error Detection Techniques

The error detection techniques are based in redundancy to detect if an error has occurred. This redundancy can be hardware redundancy, also know as space redundancy, or time redundancy. The hardware redundancy approach called duplication with comparison (DWC) is based in the duplication of the module which failing behavior has to be detected, followed by the comparison of the outputs of both modules. If the results do not match, an error signal is activated. This technique can be used to detect either SETs in combinational circuits or SEUs in memory elements. Figure 2.5 illustrates these two situations, time (situation a) and space (situation b) redundancy, to detect SEU and SET, both with one error detected.

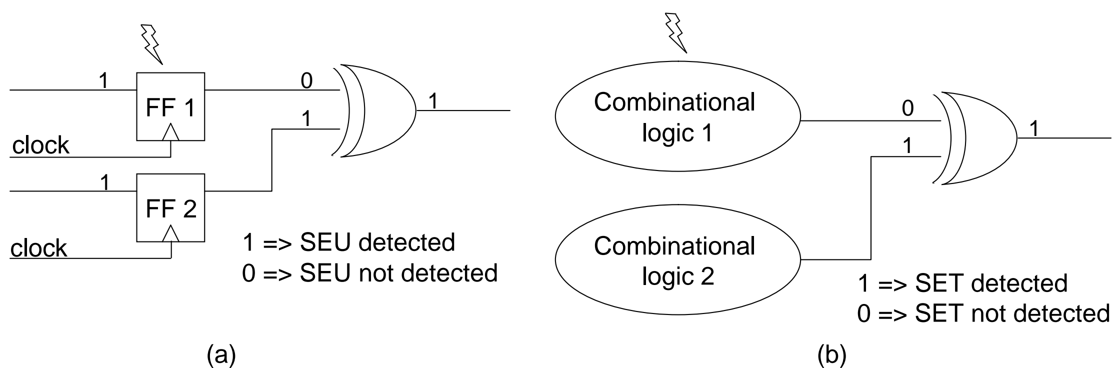


Figure 2.5: Detection of an SEU in a memory element (a) and detection of an SET in a combinational circuit (b) by using space (or hardware) redundancy.

Time redundancy can be used to detect SETs in combinational logic. This technique detects SETs by capturing the output of the combinational circuit in two different moments in time. The two captured values are compared, and in case of different values, an SET detection is indicated. Figure 2.6 illustrates the use of time redundancy to detect an SET in a combinational circuit.

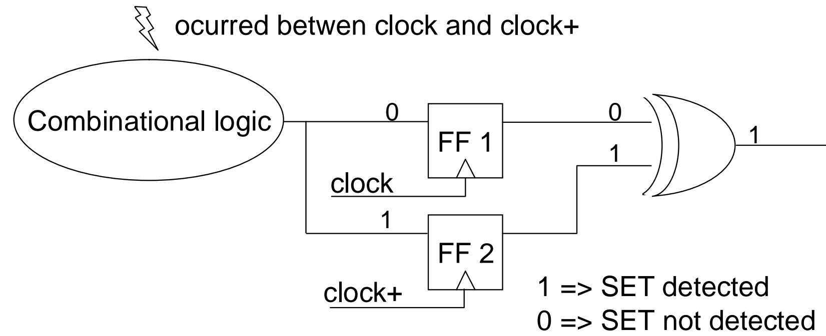


Figure 2.6: Use of time redundancy to detect an SET in a combinational circuit.

The circuit designer must set the õ ö time wide enough to allow the SET propagation, but also short enough not to loose the pulse. If a particle hits one of the memory elements used to capture the values, an SEU will be registered and an SET will be erroneously detected. The main drawbacks of detection techniques based on duplication are: the hardware area is more than doubled, and they are only able to detect the events, and not to avoid the occurrence of an error. This way, if the designer wants the circuit to operate correctly, it is necessary that the event detection flag indicates that the operation needs to be repeated and the wrong value must be discarded.

2.3.3.2 Hardware Error Detection and Correction Techniques

With the necessity of not only detecting but also correcting the soft errors, researches have proposed some detection and correction techniques based on redundancy of modules. In this section some techniques that rely on redundancy to improve systems reliability are presented.

2.3.3.2.1 Triple Modular Redundancy - TMR

The triple modular redundancy (JOHNSON`, 1994) first proposed by Von Neumann in 1956, uses the redundancy of modules to guarantee the correct functionality of the circuits in which it is implemented. This technique is based on the triplication of the protected module in a way that, if any of the three modules fails, the other two will guarantee the correct operation of the system. The redundancy used in this technique can be time redundancy or space redundancy. In Figure 2.7, the use of space redundancy of the component that is being

protected, together with a voter block, is illustrated. The voter is the module that votes, or chooses, for the majority result from the component blocks to be the circuit result.

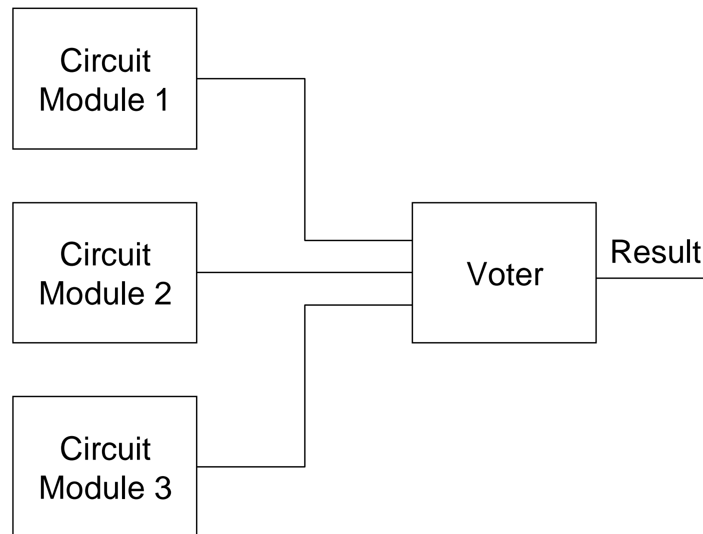


Figure 2.7: Use of space redundancy to detect an SET in a combinational circuit.

Since all the three modules operate in parallel, this technique corrects any failure in one of the three modules with the performance penalty of the voter delay. On the other hand, the area overhead is more 200%, due to the triplication of the protected module and the voter. Depending on the size of the module, this area penalty can be a price that the designer can not afford. In Figure 2.8, the use of TMR with time redundancy to correct a fault in one module is illustrated. The TMR with time redundancy only triplicates the memory elements responsible for capturing the result of the circuit at different moments in time. If we compare the area of both TMRs, the time and the space one, we can say that the time TMR has the lower area overhead if the size of the circuit is smaller than the memory element. On the other hand, the time redundancy TMR will have bigger performance penalty due to the different need to capture the circuit values at three different moments in time. Also, the clock circuit with the two õ ö delays adds some extra complexity to the circuit design.

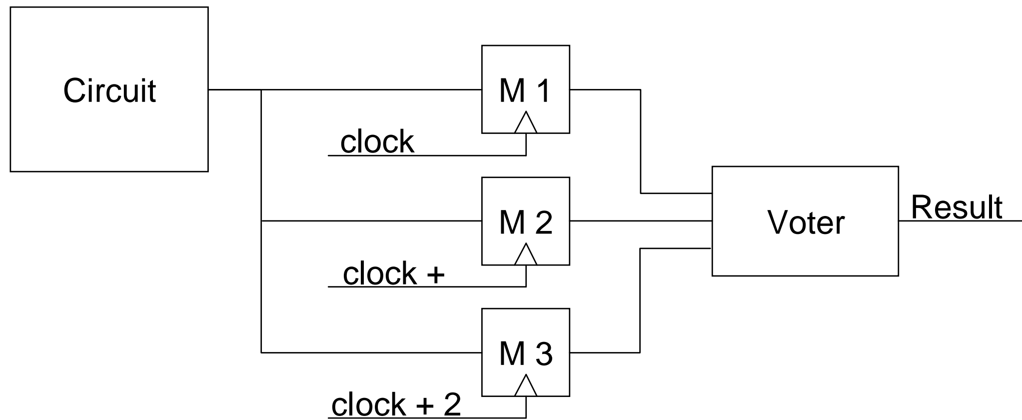


Figure 2.8: TMR with time redundancy.

However, the voter is not free of faults and if a fault hits the voter, the system reliability can be compromised. It is important to mention that the TMR technique is only effective against single faults and in case of a double faults, which means two faults affecting each one a different module, the voter can choose a wrong answer as if it were correct. To guarantee the system reliability against multiple faults, the redundancy has to be increased. This way, N-MR - Modular Redundancy of order N, uses a higher number of modules to guarantee that the majority of the modules operates correctly. In case of double faults, the number of duplicated modules must be five. This way, if two blocks fail, the other three will operate correctly and the voter will be able to choose the right result from the majority. Despite its tolerance to multiple faults, the N-MR has a huge area overhead, which gets to more then 400% for the 5-MR, due to the addition of four copies of the protected module and the voter block. Also, the size of the voter grows geometrically when compared to the TMR version. Since the voter is sensible to faults, the reliability of the system can be compromised if the size of the voter grows too much.

2.3.3.2.2 Error Detection and Correction Codes - EDAC

Error detection and correction codes are commonly used to protect storage devices against single and multiple events. There are examples of software techniques (SHIRVANI, 2000), which will be discussed in the next section, and hardware techniques (REDINBO,

1993) that perform SEU mitigation using EDAC. An example of EDAC is the Hamming code, which is useful to protect memories against SEUs because of its efficient ability to correct single upsets per coded word with reduced area and performance overheads (HENTSCHKE, 2002). However Hamming code is not effective in protecting memories against multiple bit upsets (MBUs). For this kind of event, researchers have proposed Bose-Chaudhuri-Hocquenghem (BCH) and Reed-Solomon (RS) codes, based on finite-field arithmetic (also known as Galois field).

BCH codes can correct a given number of bits at any position of the word, whereas RS codes group the bits in blocks to correct them. The drawback of these two approaches is that they have complex and iterative decoding algorithms, and use tables of constants in the algorithm. However, some studies have shown that the elimination of the table constants can simplify the RS codes (NEUBERGER, 2003). In (NEUBERGER, 2005), the authors propose a technique to improve the RS code through the individual optimization of the multipliers for specific constants. However, the area overhead imposed by the parity bits required for this technique may not be low for devices with small storage capacity. Also, the coder and decoder blocks, necessary to the generation of the parity bits and the correction of faults, are not protected against faults, and its correct functionality is crucial for the reliability of this technique. Therefore, their reliability must be guaranteed by some other protection technique.

2.3.4 SEU and SET Error Mitigation Techniques for Software-Based Systems

In the previous sections some hardware techniques used to mitigate soft errors that add some penalty in area, performance, or both, have been presented. However, when we are dealing with complex architectures made of many different components, such as computer architectures for instance, we can not simply triplicate the whole system like the TMR technique proposes. This way, other solutions with less overhead must be proposed to

guarantee the reliability of these systems. This section presents some solutions for software-based systems, divided into three broad categories: software-implemented techniques, which exploit detection mechanisms implemented purely in software, hardware-based ones, which add extra hardware, and hybrid ones, that combine both software and hardware error detection mechanisms.

2.3.4.1 Software Implemented Hardware Fault Tolerance (SIHFT) techniques

Software based detection and correction techniques are based on modifying the software executed by the processor, introducing some sort of redundancy, so that faults are detected before they become errors. They focus on checking the consistency between the expected and the executed program flow, either by inserting additional code lines or by storing flow information in suitable hardware structures. In the next sections, some of these software based techniques will be discussed with their pros and cons.

Software implemented hardware fault tolerance techniques exploit the concepts of information, operation, and time redundancy to detect the occurrence of errors during program execution. Some of those techniques can be automatically applied to the source code of a program, thus simplifying the task of software developers and reducing development costs significantly.

Techniques aiming at detecting the effects of faults that modify the expected programøs execution flow are known as *control flow checking* techniques. These techniques are based on partitioning the code of the program into basic blocks (AHO, 1986). Among the most important solutions based on the notion of basic blocks proposed in the literature, there are the *Enhanced Control Flow Checking using Assertions* (ECCA) (ALKHALIFA, 1999), the *Control Flow Checking using Assertions* (CCA) (MCFEARING, 1995), and the *Control Flow Checking by Software Signatures* (CFCSS) (OH, 2002b) techniques.

ECCA is able to detect all single inter-block control flow errors, but it is neither able to detect intra-block control flow errors, nor faults that cause an incorrect decision in a

conditional branch. In (ALKHALIFA, 1999), ECCA technique was tested with an SET of benchmark applications, and was able to detect an average of 98% of the control flow errors, with a minimum of 78.5% and a maximum of 100% obtained for one of the benchmarks. Although the authors claim that this technique implies in minimal memory and performance overheads, the exact figures are not presented in the paper. However, the implementation of the technique requires modification of the application software and a non trivial performance/overhead analysis, and for this reason the authors themselves propose the development of a preprocessor for the GCC compiler to insert the assertions in the code blocks to be fortified.

The CFCSS technique works assigning a single and unique signature to each basic block of the program. The runtime signature is held by a global variable and, in the absence of errors, the variable contains the signature associated to the current basic block. At the beginning of the program, the global variable is initialized with the signature of the first block then, at the beginning of each basic block, an additional instruction computes the signature of the destination block from the signature of the source block by computing the XOR function between the signature of the current node and the signature of the destination node. If the control can enter from multiple blocks, an adjusting signature is assigned in each source block and used in the destination block to compute the signature. As a limitation, CFCSS cannot cover control flow errors if multiple nodes share multiple destination nodes. The use of control flow assertions was also proposed in (GOLOUBEVA, 2003) by inserting additional assertions to check the control flow of the program. An SET of 16 benchmarks has been hardened against transient errors using the proposed technique, and tested with SEU fault injection in the bits of the immediate operands of branch instructions. The results have shown that this approach has an improvement over CFCSS (OH, 2002b) and ECCA (ALKHALIFA,

1999), however the technique proved to be very expensive in terms of memory and performance overheads, even though the overheads are application dependent.

CCA, ECCA and CFCSS only detect control flow errors in the program. As far as faults affecting program data are considered, several techniques have been recently proposed that exploit information and operation redundancy (CHEYNET, 2000; OH, 2002a). The most recently introduced approaches modify the source code of the application to be hardened against faults by introducing information redundancy and instruction duplication, and adding consistency checks to the modified code to perform error detection. The approach proposed in (CHEYNET, 2000) exploits several code transformation rules that require duplication of each variable and each operation among variables. The approach proposed in (OH, 2002a), named *Error Detection by Data Diversity and Duplicated Instructions* (ED^4I), consists in developing a modified version of the program, which is executed along with the original one. If results mismatches are found, an error is reported. Both approaches introduce overheads in memory and execution time. The approach proposed in (CHEYNET, 2000) minimizes the latency of faults; however, it is suitable to detect transient faults only. Conversely, the approach proposed in (OH, 2002a) exploits diverse data and duplicated instructions, and thus is suitable for both transient and permanent faults. As a drawback, its fault latency is generally grater than in (CHEYNET, 2000). The ED^4I technique requires a careful analysis of the size of used variables, in order to avoid overflow situations.

Although very effective, SIHFT techniques may introduce time overheads that limit their adoption only to applications in which performance is not a critical issue. Also, in some cases they imply a memory overhead to store duplicated information and additional instructions, what demands an extensive work from the application programmer when the automation is not possible. These approaches also require access to the source code of the

application, precluding the use of commercial off-the-shelf software components from a library.

2.3.4.2 Hardware Techniques for Software-Based Systems

Software based solutions usually impose high cost to the system performance, which for certain types of applications are simply not acceptable. For this kind of systems, hardware techniques are more indicated, as their performance overhead is lower. In this section, some hardware based solutions to cope with SEUs and SETs in software based systems are presented.

2.3.4.2.1 Dynamic Implementation Verification Architecture - DIVA

õ*Dynamic verification*ö, a hardware-based technique, is detailed in (AUSTIN, 2000) for a pipelined core processor. It uses a õfunctional checkerö to verify the correctness of all computations executed by the core processor. The checker only permits correct results to be passed to the commit stage of the processor pipeline. The so-called DIVA architecture relies on a functional checker that is simpler than the core processor, because it receives the instruction to be executed together with the values of the input operands and of the result produced by the core processor. This information is passed to the checker through the re-order buffer (ROB) of the processorøs pipeline, once the execution of an instruction by the core processor is completed. Therefore, the checker does not have to care about address calculations, jump predictions and other complexities that are routinely handled by the core processor. Once the result of the operation is obtained by the checker, it is compared with the result produced by the core processor. If they are equal, the result is forwarded to the commit stage of the processorøs pipeline, to be written to the architected storage. When they differ, the result calculated by the checker is forwarded, assuming that the checker never fails. If a new instruction is not released for the checker after a given time-out period, the pipeline of the core processor is flushed, and the processor is restarted using its own speculation recovery mechanism, executing again the instruction. The DIVA approach cannot be implemented in

SoCs based on FPGAs that have an embedded processor, because the checker is implemented inside the processorøs pipeline. Also, it assumes that the checker never fails, due to the use of oversized transistors in its construction and extensive verification in the design phase.

Originally conceived as an alternative to make a core processor fault-tolerant, this work also evolved to the use of a similar checker to build self-tuning SoCs. To demonstrate the benefits of the proposed solution, the authors implemented the DIVA architecture for the Alpha 21264 and created the so called REMORA (WEAVER, 2001). Results of an architectural simulation of nine SPEC95 benchmarks showed that the performance penalty was less than 1%. The area and power overheads were 6% and 1.5% respectively. Although the good results, the authors do not indicate which fault injection model was used. Also, in case of memory bit flips the technique will not be reliable, because both processors will use corrupted data to perform the operations.

2.3.4.2.2 Simultaneous and Redundantly Treaded (SRT) Processor

In (REINHARDT, 2000) the authors propose the use of a simultaneous and redundantly treaded processor, which is derived from a Simultaneous Multi Threaded (SMT) Processor (DEAN, 1996; DEAN, 1998), to detect faults by running two copies of the same thread at the SRT processor. The authors introduce the concept of the sphere of replication, which indicates what components will have the redundant execution mechanism to detect faults. All activity and states within the sphere are replicated, either in time or in space. Values that cross the boundary of the sphere of replication are the outputs and inputs that require comparison and replication, respectively, and the components that are out of the sphere of replication need other fault detection techniques. The proposed technique brings some challenges that are not present at a lock-stepped, physically-replicated design, like deciding when to compare the outputs and also when and which inputs need to be replicated. To solve these questions, the authors propose the use of some queues and buffers to indicate and store the values that need to be compared and keep the values that need to be replicated.

More details can be found in (REINHARDT, 2002). The proposed technique is only a detection technique and needs a recovery mechanism or some jump trigger to a safe state to guarantee the reliability of the processor. Also, the authors do not provide the overheads in terms of area and performance implied by the proposed approach.

2.3.4.3 Hybrid Techniques

Hybrid techniques such as (BERNARDI, 2006) combine some SIHFT techniques with an infrastructure IP core in the SoC. The software running on the processor core is modified by inserting instruction duplication and information redundancy together with some instructions for communication with the I-IP. The I-IP works concurrently with the main processor, implements consistency checks among duplicated instructions, and verifies whether the correct program execution flow is executed. Such techniques are effective, since they provide a high level of dependability while minimizing the added overhead, both in terms of memory occupation and performance degradation, but they require the availability of the source code of the application.

There are cases in which the software of the application is not available or the costs involved in modifying the application software are too high. To solve this problem, the authors of (LISBOA, 2006) proposed the idea of introducing an I-IP between the 8051 multi-cycle processor and the instructions memory, making the I-IP replace on-the-fly the fetched code by a hardened one.

In this work a hybrid solution, such as the one proposed in (LISBOA, 2006) for the MIPS RISC pipelined architecture, is also presented, and its effectiveness in detecting control flow errors and instruction hardening caused by particle hits in the architecture registers, without adding any memory overhead or architecture modification of the MIPS architecture, is demonstrated.

In this chapter several techniques to improve the fault tolerance in all the stages of a system production circuit design have been presented. As it was previously mentioned,

process modification techniques usually increase the production costs. On the other hand, hardware redundancy techniques imply in high area overhead (greater than 200%), while software redundancy techniques generally adds undesirable performance and memory overheads.

The work presented here proposes two different solutions that provide improvement in the system reliability without the overheads implicit in the existing solutions. The first approach of this work proposes the replacement of the combinational circuit by magnetic memory based circuits, which is intrinsically protected against radiation induced bit flips due to its magnetic way of storing information. As we are just replacing part of the circuit, the area overhead introduced by this technique is potentially low. Also, due to some key architectural control techniques, the performance results showed that the proposed architecture not only has no small overhead but is faster than the equivalent non protected architectures that were compared to this work.

The second solution presented in this work is a hybrid technique that uses an I-IP to improve the system reliability through instruction hardening and the detection of control flow errors. This technique implies in neither memory overheads nor requires any modification of the hardware, like the other software and hybrid techniques do.

3 USING MEMORY BASED CIRCUITS TO COPE WITH SEUS AND SETS

The use of memory not only as a storage device, but also as a computing device, has been a subject of research for some time. In order to explore the large internal memory bandwidth, designers decided to bring some functions executed by the processor into memory, to make effective use of all these available data. In (ELLIOTT, 1999) the Computational-RAM is presented, bringing processor functions into the memory. This technique was originally used as a SIMD Processor (Single Instruction Multiple Data Processor) in some DSP applications. Also, memories come with intrinsic protection against manufacturing defects due to its spare columns and spare rows that can be activated to replace the malfunctioning ones. Also, as it was previously mentioned, they can be protected by Reed-Solomon codes, such as the one proposed in (NEUBERGER, 2005), with relatively low overhead.

The fact that the contents stored in new memory technologies like MRAMs and FRAMs can not be flipped by particle hits, together with the fact that faults affecting logic components are becoming as common or even more than the ones affecting memory elements, makes the use of memory based circuits a good design strategy to implement more robust circuits for future technologies. So, if we reduce the quantity of combinational circuit, by replacing it with memory components, we will reduce the overall architectural vulnerability factor (AVF) and, consequently, the soft error rate.

To test this assumption, two memory based circuit for a 4x4-bit multiplier, and one memory based circuit for a 4-tap 8-bit Finite Impulse Response (FIR) filter were implemented, and compared with their combinational counterparts through single and double simultaneous fault injection campaigns (RHOD, 2006a). All memory elements were protected with the RS code proposed in (NEUBERGER, 2005), to tolerate multiple bit flips.

3.1 4X4-BIT MEMORY BASED MULTIPLIER

In those circuits based on the use of memory, the memory works as a truth table that receives the inputs and returns the outputs according to the implemented function. Since the size of a truth table depends on the width of the input and output, the memory size, in bits, also depends on the input and output widths. This relationship can be described as follows:

$$\text{Size} = I^2 \times O \qquad (3)$$

where I and O are the input and output widths, respectively, both in bits.

For instance, let us consider an AND gate with two inputs A and B. The memory element that would replace this gate would have 2 inputs, representing the A and B values, and one output, to drive the result of the AND operation. The memory size would be equal to 2^2 x 1, which gives us 4 bits. In Figure 3.1 the truth table of the 2 bits AND operation is illustrated.

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Figure 3.1: AND truth table.

In Figure 3.1, one can identify the inputs A and B which in our memory circuit will become our address bits, and the result column indicated by the column C, which will be the 4-bits memory content. The memory content has to be organized according to the truth table, which for this AND example means the positions 0, 1 and 2 have to hold the value ‘0’ and the position 3 holds the value ‘1’.

It is important to mention here some self imposed design restrictions that we had to comply with and that led us to the proposed solutions for the multiplier test case:

Very small memories are not area efficient, because a significant area is needed to implement the decoders and a smaller proportion of area is used for data storage;

The size of the memory used to replace the combinational parts is smaller than the size of the memory needed to implement the whole function, in our case, the 4x4-bit multiplication; otherwise, we would have a fully truth table implementation of the function of the circuit. So, in this case, the memory size must be smaller than 2048 bits;

The size of the combinational circuit must be smaller than the size of the fully combinational circuit of the 4x4 bit multiplier of Figure 3.2, since the goal is to avoid faults in the combinational circuit part.

In order to illustrate the different ways that a memory based circuit can be implemented using memory, two different solutions for the 4x4-bit multiplier, with different amounts of memory and combinational circuits, were proposed. The first one, here called *the column multiplier*, had more combinational circuit and less memory than the second one, here called *the line multiplier*. Using simulated fault injection to calculate the fault propagation rates of these two solutions, we compared the obtained results with the Architectural Vulnerability Factor (AVF) of the 4x4-bit multiplier implemented with the fully combinational circuit shown in Figure 3.2.

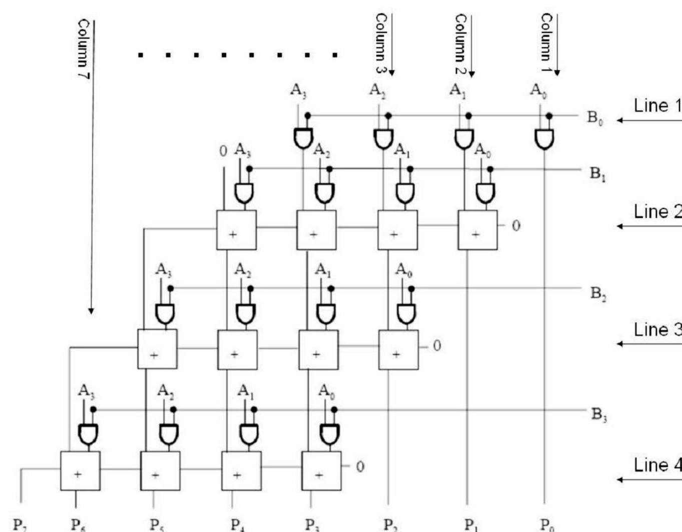


Figure 3.2: Fully combinational 4x4-bit multiplier.

The column multiplier, as the name implies, makes the multiplication column by column.

Therefore, to perform a 4x4-bit multiplication, 7 cycles of operation are necessary. During the first cycle, all operations required to generate bit P0 (Figure 3.3) of the product are performed. During the second cycle of operation, bit P1 is generated, and so on, until the last cycle, when bits P6 and P7 are generated. In Figure 3.3 one can see the implemented column multiplier circuit. In this circuit, memory performs the function of one to three full adders of a column, depending on the column that is being calculated. Figure 3.3 also shows that some additional circuitry has been added in order to properly generate control signals. To save the carry-out signals for the next cycle, a 3-bit register is used. A 6-bit shift register was also required to save and shift the product. Another control requirement was a 3-bit counter to generate the selection signals for the multiplexer.

In Figure 3.3, the combinational circuit that is sensible to faults is highlighted with a dashed rectangle.

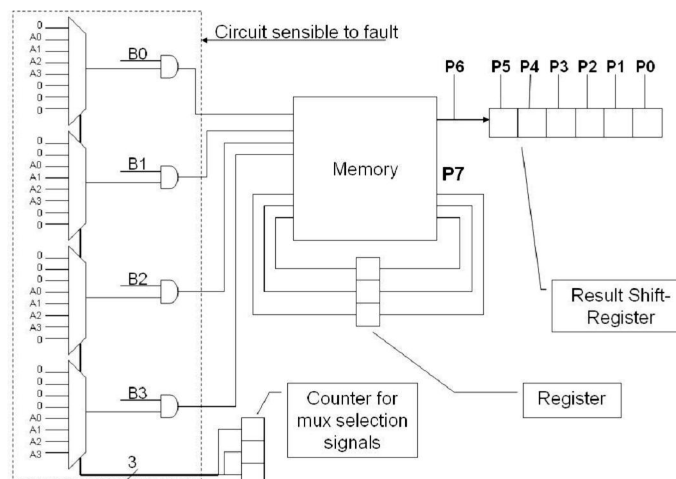


Figure 3.3: Column multiplier circuit.

In the Line multiplier circuit, multiplication is performed line by line. In this case, the number of cycles necessary to make a multiplication is equal to the number of bits of the

inputs, which in our case are four. During the first three cycles, only one result bit per cycle is generated and the four remaining bits are calculated in the last cycle.

In Figure 3.4 we can see the implemented line multiplier circuit. In this circuit, memory performs the function of all 4 full-adders in a line. Like in the previous implementation, it was also necessary to include some additional circuitry for control and to save some values from one cycle to other. But in this circuit only a 3-bit shift-register to store and shift the product was necessary, against the 6-bit register used in the previous solution.

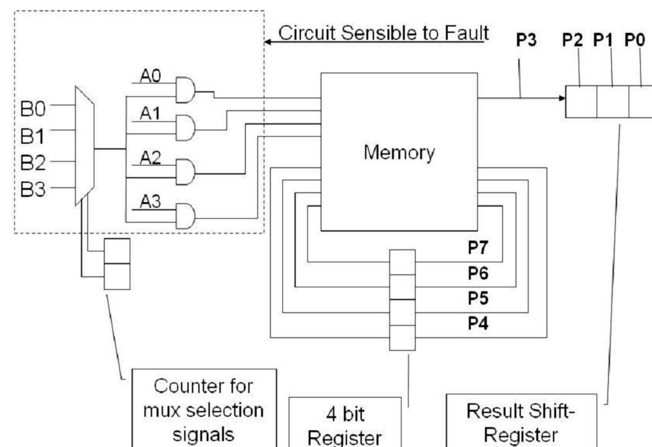


Figure 3.4: Line multiplier circuit.

The area characteristics of the proposed solutions are compared with those of the TMR and 5-MR in Table 3.1. This table also shows the costs of the Reed Solomon protection used for registers and the memory, together with the coder and decoder area costs, which were obtained using the tool proposed in (NEUBERGER, 2005).

Table 3.1: Area for Each Solution in number of transistors.

Circuit	Comb. Circuit	Flip-Flops	Voter	Memory	RS Cod. /Decoder	Total
5-MR	3,270	-	672	-	-	4,392
TMR	2,232	-	240	-	-	2,472
Combin.	744	-	-	-	-	744
Column	200	468	-	3,048	96	3,812
Line	42	346	-	7,650	96	8,134

To evaluate the area, we have considered that each bit of ROM memory demands 4 transistors. For the logic gates we computed the area as follows: 6 transistors for AND, OR and XOR gates, 4 for NAND and NOR gates and 12 for each flip-flop.

One important thing that must be taken into consideration is the additional unprotected area that the voters add to the TMR and 5-MR solutions. In TMR, the voter is almost 15% of the total area, and in 5-MR it is more than 28%. In the memory solutions, the area added for the Reed-Solomon encoder and decoder is less than 4% in the column multiplier solution and less than 2% in the line multiplier solution.

The injection of faults was simulated using CACO-PS (Cycle-Accurate Configurable Power Simulator) (BECK, 2003a), a cycle-accurate, configurable power simulator, which was extended to support single and double simultaneous transient faults injection. The simulator works as follows: first, it simulates the normal operation of the circuit and stores the correct result. After that, for each possible fault combination in the circuit, the simulation is repeated. Then, the output of each simulation is compared to the correct one. If any value differs, it means that the fault was propagated to the output. All the process is repeated again, for each combination of input signals of the circuit. Both implementations of the multiplier using memory were compared with the fully combinational solution and with the classical TMR and 5-MR solutions. The resulting fault propagation rates can be seen in Table 3.2, for single and two simultaneous faults injection. In the same table, one can also find the critical path timing of all solutions. These results were obtained with electrical simulation of the circuits. We used the Smash Simulator for 0.35 m technology.

In Table 3.2, one can see that the architectural vulnerability factor for single faults (3rd column) and for double faults (4th column) was higher in the solutions using memory than in the TMR and 5-MR ones. That happened because we have reduced the area susceptible to faults, and consequently increased the influence of that portion of the circuit in the final

result. But, if we take into account that the circuit with less area has less probability to be affected by a transient fault, and make a proportional AVF evaluation (5th and 6th column), as the percentage of observable faults at the output, one can see the benefits of the proposed solutions.

Table 3.2: Architectural Vulnerability Factor and Timing Results for Single and Double Faults

Circuit	#of gates that fail	AVF % (1 fault)	AVF % (2 faults)	Prop. AVF % (1 fault)	Prop. AVF % (2 faults)	Critical Path Timing (ns)
5-MR	492	8.80	20.50	8.80	20.50	18.5
TMR	268	5.49	16.26	2.99	8.86	18.2
Combin.	76	49.11	63.60	7.59	9.82	17.5
Column	33	15.92	28.05	1.07	1.88	15.0
Line	9	36.22	54.07	0.66	0.99	16.5

When contrasting the results in Tables 3.1 and 3.2, one can notice that the 5-MR solution almost doubles the area required for TMR, and also increases by a factor of 2.5 the percentage of faults that are propagated to the output of the circuit. That happens due to the significant increase in non-protected area introduced by the voter in the 5-MR approach. The conclusion, then, is that future solutions based upon increasing the redundancy in terms of modules will no longer be a good alternative when multiple simultaneous faults become a concern. Another important observation is that, depending on the design alternative, the area versus fault tolerance trade-off may impact quite differently, according to the adopted solution, when contrasted with the TMR approach. For the column multiplier, the area increases 1.5 times, while the fault propagation percentage is reduced 4.7 times. For the line multiplier, however, the area increases by a factor of 3.2, while the AVF decreases by a factor greater than 8.

When one looks at the timing results in Table 3.2, one can notice that the critical path in the memory solutions has decreased. That happened because the proposed memory solutions reduced most of the combinational circuit, and added a memory and flip-flop based

circuit that contributes less to the critical path than the combinational circuit that was replaced. On the other hand, the total computation time has increased by a factor of almost 4 for the line memory and almost 7 for the column memory. That happened because the new memory solutions compute the multiply in 4 and 7 cycles, for the line and column memory solutions, respectively. The final computation time is bigger for the memory based circuits than for the others. It is important to remember that the objective of this work was to show that, when replacing a fully combinational circuit with a protected memory and a smaller combinational circuit, we can have some benefits in terms of reliability, which for this memory circuit was 4.7 for the column solution and more than 8 for the line solution, respectively.

3.2 4-TAP, 8-BIT FIR FILTER MEMORY BASED CIRCUIT

In a second case study we implemented a 4-tap, 8-bit FIR filter. We compared the fully combinational solution (Figure 3.5) with a solution using our approach, with memory replacing part of the combinational circuit.

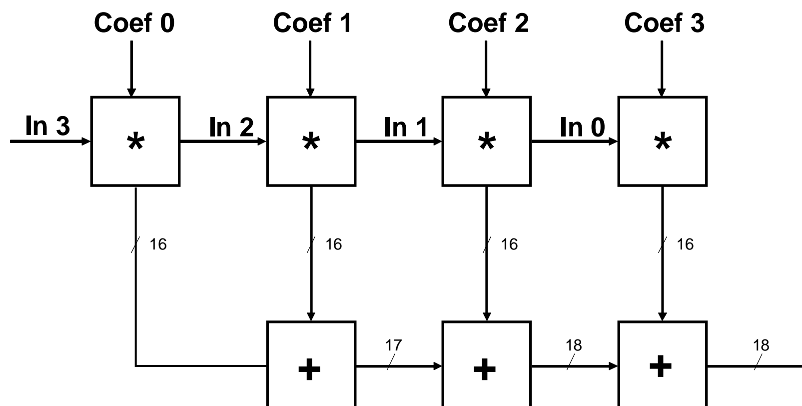


Figure 3.5: Combinational circuit for the 8-bit FIR filter with 4 taps.

The filter implementation using memory to replace part of the combinational logic is illustrated in Figure 3.6.

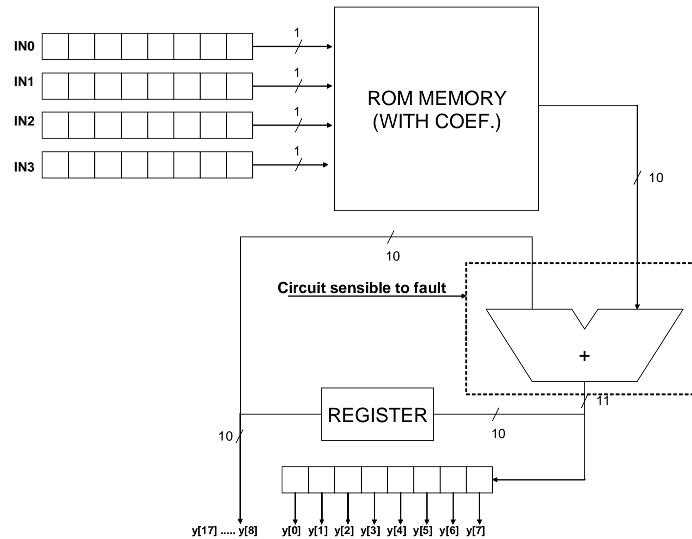


Figure 3.6: 8-bit FIR filter with 4 taps, using memory.

The filtering function is performed in 8 cycles and the memory function can be described by the following equation:

$$y(n) = \sum_{k=0}^{M-1} c_k\, x_{M\text{-}k}(n) \tag{4}$$

where n is the bit position (from 0 to 7), k is the tap number (from 0 to 3) and M is the order of the filter.

In our solution using memory, we pipelined the multiply and add operations, in order to reduce the memory size. The comparison between the area of the combinational filter and the memory one is shown in Table 3.3.

Table 3.3: Area results for the filter implementations in number of transistors.

Filter Circuit	Combin. Circ.	Flip-flops	Memory	RS cod./dec.	Total
Combinational	16,494	-	-	-	16,494
Memory based	540	1,700	900	484	3,624

Since this is a pipelined filter, it was necessary to include a 10-bit adder to add the partial products generated in each cycle, and drive the result to the output. We also included a

register to store the sum from one cycle to the next and an 8-bit shift register to shift and store the 8 least significant bits, which are generated one per cycle.

Differently from the multiplier, it was not possible to simulate the injection of all possible combinations of faults in the filter in an exhaustive way, because it would take too long to get the results.

However, from the experience with a previous case study, where we noticed that only a small number of randomly injected faults (less than one percent of the total number of possible faults) was necessary to reach an approximately stable result, in terms of percentage of faults that propagate to the output, we decided to use a randomly generated set of input combinations and single/double faults injection to evaluate the AVF for the fully combinational solution and for the one using memory.

To implement the fault injection in a faster way, the filter was implemented in VHDL and both filter architectures have been synthesized in an FPGA (Altera EP20K200EFC484-2X). The results are shown in Table 3.4, for single and double faults.

In this case study we can see, from Table 3.4, that the proportional AVF of the memory solution was more than 20 times smaller than that of the combinational solution for single and double simultaneous faults.

Table 3.4: AVF results for single fault in FIR filter implementations.

Circuit	# of gates that fail	Proportional AVF (1 fault)	Proportional AVF (2 faults)
Combinational	1,631	48,21	67.35
Memory	50	1.39	2.11

It is clear that the memory based solution has a greater performance overhead, but as it was stated before, our objective with this work was to show the reduction of the AVF that one can obtain when using a memory based circuit instead of the traditional combinational one, since, as stated in the equation 2 from the previous chapter, the SER of a circuit is

proportional to its AVF. So, if we reduce the circuit AVF it is the same as reducing the circuit soft error rate.

In this chapter, two different applications where traditional combinational circuits were replaced by memory based ones to reduce their AVF were presented. We saw that, depending on the application, and also on the designer strategy, different AVF reductions can be achieved. Despite the good results obtained using this idea, it is not possible to propose a memory based circuit for every combinational circuit that exists nowadays. Also, this idea was proposed to improve only hardware modules, and sometimes it is simply too expensive to convert a software algorithm into a hardware one to improve its reliability. This way, in the next chapter a memory based core processor architecture is presented, as an evolution of the idea proposed in this chapter.

4 MEMPROC: A MEMORY BASED, LOW-SER EFFICIENT CORE PROCESSOR ARCHITECTURE

In the previous chapter a study on how the use of memory based circuits can affect the AVF, and consequently the SER, of a circuit was presented. Despite the good results, the proposed memory solutions imply some performance and area overheads and require a different design for every application. Also, the costs involved to implement the proposed idea for software applications might not be worth.

In this chapter, an innovative general purpose memory-based core processor, designed to be reliable against SETs and SEUs, without adding significant performance or area overhead is presented. At the same time, we favor a regular architecture that can be used to enhance yield in future manufacturing processes. This architecture is called MemProc and was presented in (RHOD, 2006b).

4.1 THE MEMPROC ARCHITECTURE

The processor architecture that is presented here is a multi-cycle 16-bit processor with a Harvard architecture that performs its operations using a microcode memory. Figure 4.1 shows the main functional blocks of the proposed architecture.

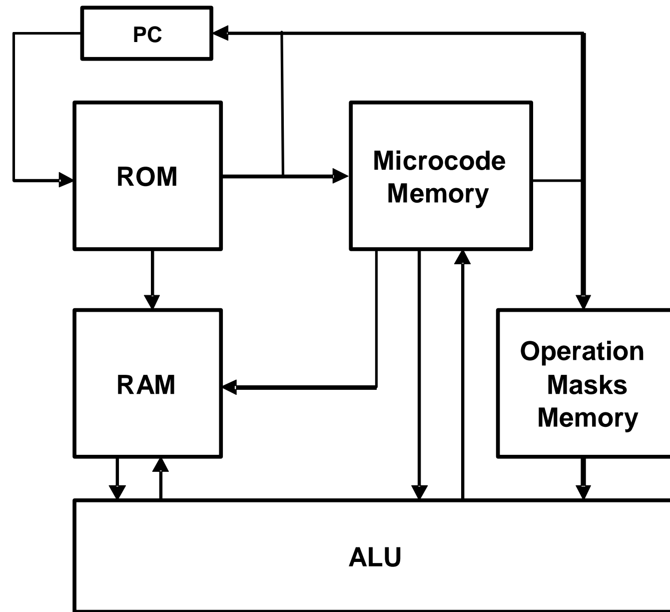


Figure 4.1: MemProc overall architecture.

The microcode memory receives the initial address of the microcode that executes the current operation from the ROM memory and generates the control signals for the data memory, ALU, and operation masks memory. The operation masks memory is responsible for passing the operation masks to the ALU. All arithmetic and logic operations results are stored in the RAM memory, and the register bank is also mapped into this memory. Each logic or arithmetic instruction takes at least 4 cycles to be executed. During the first cycle, the fetch and decoding of the instruction are performed by the microcode memory, and all the operands are fetched from the RAM memory during the second cycle. During the third cycle the operation is executed, and its result is stored in the RAM memory during the fourth cycle.

4.1.1 The Macroinstruction

The macroinstruction of the MemProc architecture is 56 bits wide and is unique for all types of instructions. In Figure 4.2, the macroinstruction format is illustrated, with its different fields and the width of each field. The *opcode* field of the macroinstruction represents the code of the operation that is being executed and is 8 bits wide. The *operand1* and *operand2* fields can indicate the address or the value of the operands used in the

instruction. The *destination* field indicates the destination address of the operation that is being executed. In case of a branch instruction, the *destination* field indicates the address of the branch.

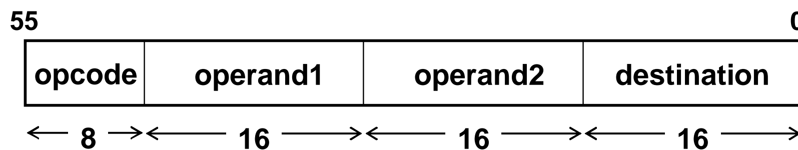


Figure 4.2: Macroinstruction format.

As mentioned before, the MemProc architecture is a multi-cycle machine and, depending on the instruction, the execution of the operation can take different numbers of cycles. For instance, the instruction MOV can take from 2 to 4 cycles to be executed, depending on the types of the operands. A complete list of the 48 instructions that were implemented in MemProc up to now, and the number of cycles that each instruction takes in the execution stage is in the Appendix A.

4.1.2 The Microcode

The microinstructions of the proposed architecture are 66 bits wide and composed by three fields: the *deviation address*, the *operation masks code* and the *control signals*, as it can be seen in Figure 4.3.

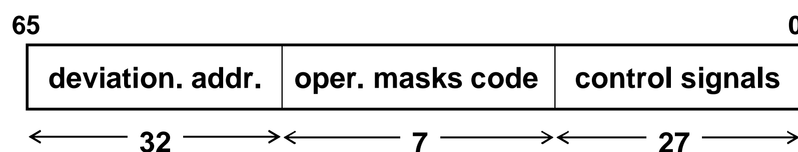


Figure 4.3: Microinstruction format.

The *deviation address* field stores 4 possible deviation addresses in the microcode. This field was introduced to allow deviations in the microcode to accelerate some instructions and also to allow the reuse of the code. The *operation masks code* indicates to the õoperation masks memoryö which are the operation masks that will be used in the next execution cycle.

The operation masks will be explained in more details in the next section. The *control signals* field generates the control signals for all hardware structures of the architecture, such as: memory enable, multiplexors selection and register enable signals.

4.1.3 The Arithmetic and Logic Unit - ALU

The MemProc architecture was designed with the purpose of reducing the area that is more sensitive to SEUs and SETs. As mentioned before, there are several ways to protect memory with low overhead, like EDAC or by using intrinsically protected memories like MRAM, as it is used in this work. However, when it comes to protect the combinational logic, the costs in area are relatively high. This way, in this architecture it is proposed to use simplified combinational logic hardware and more memory elements to improve the architecture reliability in the presence of particle hits. Therefore, due to its simplicity, the ALU of MemProc is based on the Computational RAM approach (ELLIOTT, 1999).

The ALU is composed by 8:1 multiplexors, which are able to generate all the minterms for a given 3-bit boolean function, according to the values of bits X, Y, and Z (or M). Figure 4.4 depicts a MemProc ALU block for processing 1bit of data. The complete MemProc ALU is 16-bit wide and its 16 blocks work in parallel, being able to perform bit serial arithmetic and logic operations. To accelerate addition operations, two 8:1 muxes are used, instead of a single one, as done in the Computational RAM approach; one is responsible to calculate the sum and the other, to calculate the carry out.

The operation masks feed the ALU to calculate all arithmetic, logic and conditional branch operations. Each line of the operation masks memory has 32 masks, with 8-bit width, which gives a total of 256 bits of information. Figure 4.5 highlights a complete line of operation masks used during the addition operation.

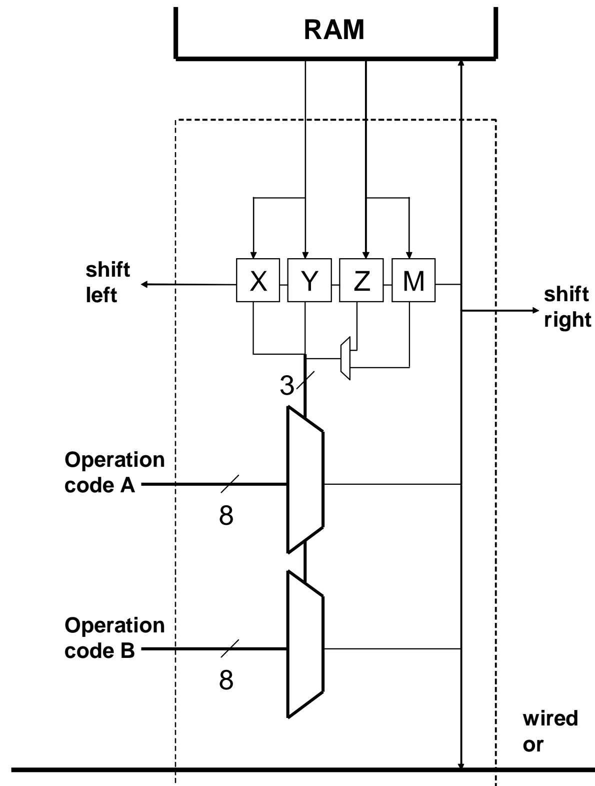


Figure 4.4: ALU for one bit operation.

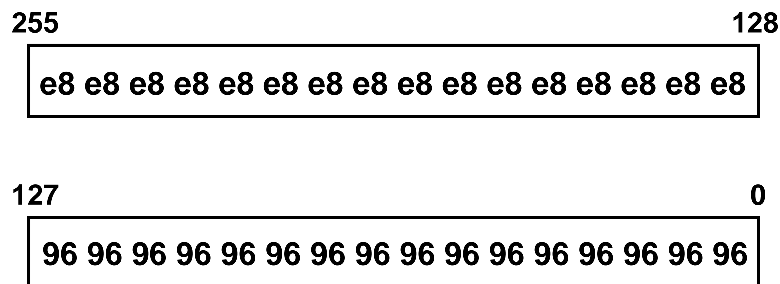


Figure 4.5: Operation masks used during the addition operation.

Figure 4.6 illustrates how the ALU works. In this figure, an addition operation for one bit of the ALU is presented. One can see, from the truth table, that the hexadecimal values of the operation masks for the õsumö and the õcoutö (carry out) outputs of the multiplexors are 96 and e8, respectively. Also in Figure 4.6, one can see the presence of two wired-or buses. These buses implement an õorö operation of all the multiplexorsø outputs. These wired-or busse are extremely important to allow the control of stopping an arithmetic operation as soon as the final result is ready, and also for the improvement in performance when executing

conditional branch instructions. The way these gains are achieved will be explained in more details in the section that follows.

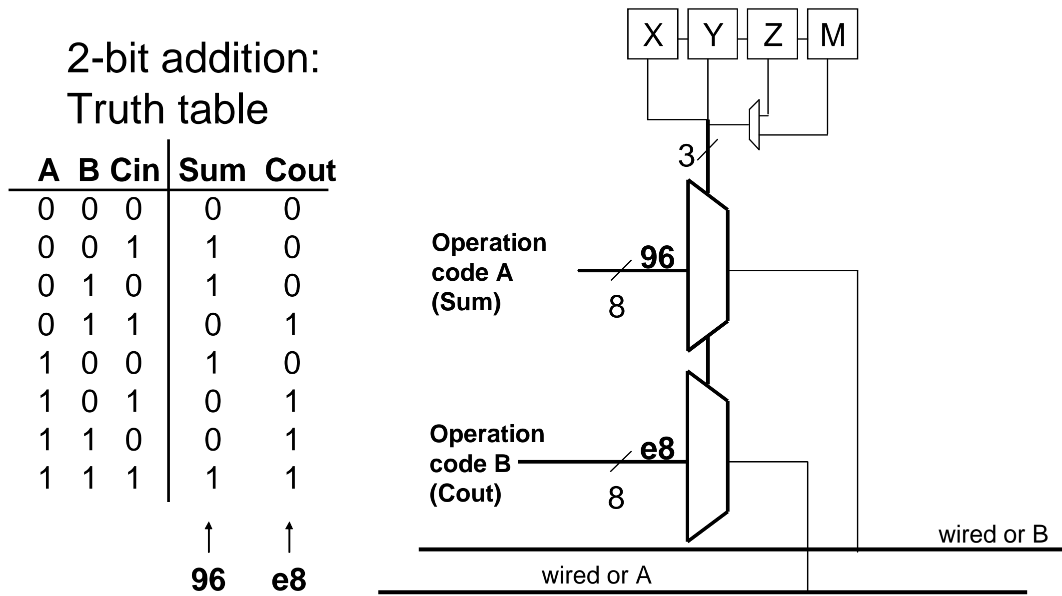


Figure 4.6: 2-bit addition using MemProc ALU.

4.2 DESIGN STRATEGIES THAT IMPROVED PERFORMANCE

The MemProc architecture was designed to have a simplified combinational hardware to reduce the sensible area of the architecture. As it was explained in the previous section, the ALU is capable of 1-bit operations only; therefore, this introduces a performance degradation for operations that need information from a previous cycle to compute the next cycle, like for instance the addition operation, that needs the carry out from one cycle to calculate the sum and the next carry out values. In order to accelerate some operations, we introduced the wired-or buses and also an extra flip-flop called õMö to accelerate multiply operations.

The way MemProc achieves its high performance is based on the fact that it the execution of any operation takes only the exact number of cycles necessary to get the operation result. In traditional computer architectures, the ALU does its arithmetic and logic

operations using combinational hardware which takes always the same time to perform the complete operation, regardless of the value of the operands. In MemProc, the hardware executes only the number of cycles necessary to get the result, according to the carry propagation chain. To explain it clearly, Figure 4.7 illustrates this paradigm with an 8-bit addition operation.

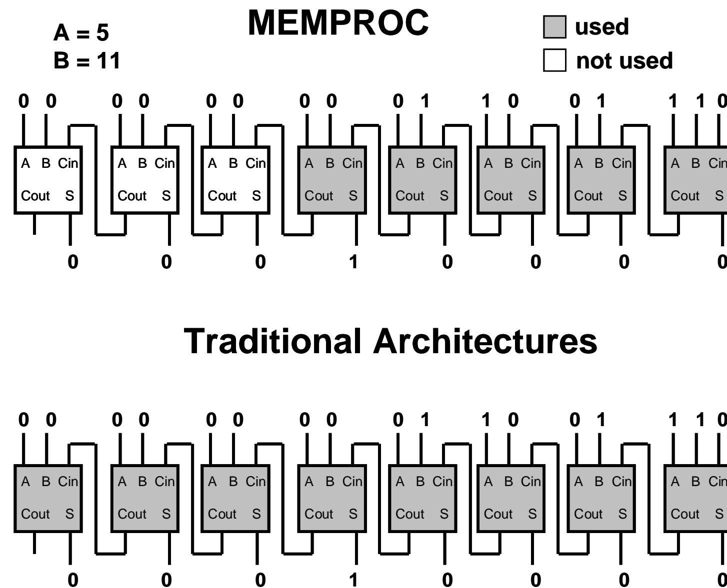


Figure 4.7: 8-bit addition paradigm.

In Figure 4.7 we can see that MemProc needs to wait only for 5 of the 8 operating units to complete their operations in order to get the result, which means that it takes 5/8 of the time that traditional architectures require to perform this addition. To detect when the operation is finished, MemProc uses the wired-or bus to evaluate when there are no more carry outs to propagate, which means that the addition is finished. This way, we can say that the proposed architecture takes advantage on the value of the operands. For instance, one addition can require from 3 to 18 cycles to be performed, depending on the number of carries to be propagated, which depends on the value of the operands.

Multiplications are also performed in order to take advantage of the value of the operands, since the number of cycles depends on the number of bits equal to zero in the operands. The multiplication operation is the same as a sequence of sums and shifts of one

operand, and the number of sums is proportional to the number of ÷1sø that the operands have. Therefore, the number of required cycles decreases as the number of bits equal to zero in the operands increases.

So, in general, the lower the values of the operands the lower is the number of cycles it will take to perform an operation. One could say that if the values of the operands are high the proposed approach would not have any advantage. Nevertheless, in (RAMPRASAD, 1997) results show that the transition activity for some multimedia benchmarks is more intense in the 8 least significant bits. This means that, for this kind of application, most of the data tends to be in the range from 0 to 255, and can be represented in 8 bits, which gives us a low probability of the necessity of more than 8 carry propagations.

Other gains arising from the strategy of computing just the necessary can be achieved when we are dealing with the õfor loopö control structure. Most of the time, this loop structure is used to count up by one,to control the number of repetitions of some block of code. If we analyze just the addition operation present in this loop, we will see that this addition operation produces no carry in 50% of the additions and only one carry in other 25% of the cases. This way, we can assume that for this kind of loop structure, the MemProc architecture will take 3 cycles for 50% of the additions and 4 cycles for other 25%. More results related to MemProc performance gains will be shown in the next chapter.

4.3 CODE GENERATION

In other to accelerate code generation, it was decided to generate the MemProc program code based in another language. Instead of making a compiler or modifying an existing one to, it was created a C program to work as a translator from the Java compiled code to MemProcøs language. In Figure 4.8 the code generation process created for MemProc is illustrated. During the first step, the application programmer writes the Java code of the

application. Since MemProc does not support dynamic space allocation, nor recursive functions, the programmer can not use these programming resources when writing the Java application code. Therefore, all variables and methods must be created statically to have their space reserved. After the code is written, it is compiled and the mnemonic Java code (Java bytecodes) is produced. At the next step, the code translator is applied and the MemProc instruction code is obtained. Then, in the last step, the MIF generation program, also developed as part of this work, is ran to obtain the Memory Initialization File (.mif) of the MemProc program code.

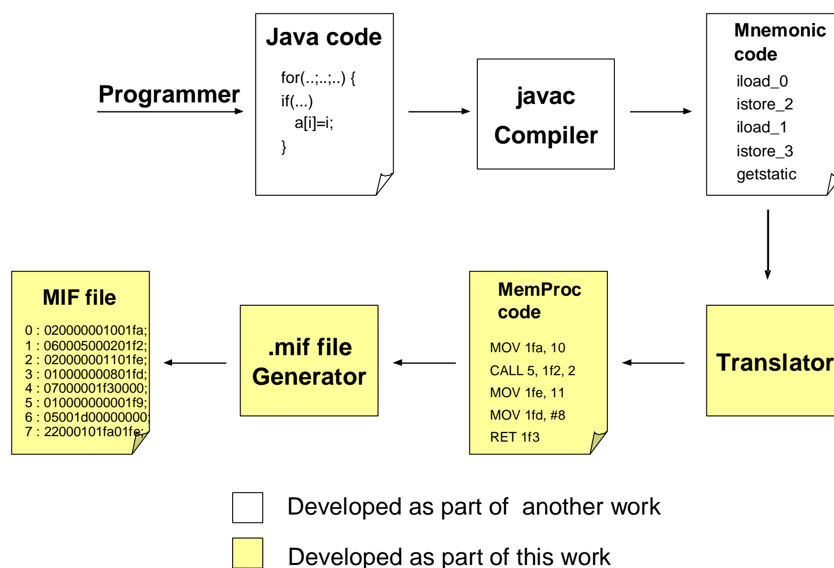


Figure 4.8: Code generation process for MemProc.

From the blocks that compose Figure 4.8, the code translator was the one that demanded more time to be finished. That happened because the source code (the Java one), is a stack based code, which in other words means that all operands need to be stacked before they are used in any operation. On the other hand, the MemProc architecture is more like a RISC one, which needs fewer instructions to perform the same operation. So, it was necessary for the translator to make an intensive code analysis in other to find the correct operands for each operation in the MemProc code.

5 MEMPROC: EXPERIMENTAL RESULTS

In order to evaluate the feasibility of the proposed architecture, both in terms of fault tolerance, area, and performance, extensive simulations have been executed, to compare the MemProc architecture with two different architectures: a 16-bit processor, with a 5-stage pipeline, named FemtoJava (BECK, 2003b) and a well known RISC architecture, the MIPS processor (PATTERSON, 2002). In the first section of this chapter the characteristics of the two architectures that are being compared with MemProc are presented. The second section shows the tools that were used to evaluate the architectures. At the third section the fault rate and area evaluation experimental results are explained. Finally, in the last section, the performance results of the proposed architecture are presented.

5.1 ARCHITECTURES COMPARED WITH MEMPROC

The first architecture that was compared with MemProc is the pipelined version of the FemtoJava processor family. This processor family has a Harvard architecture that executes Java bytecodes based on stack operations. The first version of the FemtoJava processor was the multicycle version proposed by (ITO, 2001). The next version was the 16 and 32 bits, 5-stage, pipelined version (BECK, 2003b), also called FemtoJava Low Power. In other to explore dynamic parallelism and parallelism during compilation time, a superscalar and a VLIW (Very Large Instruction Word) versions (BECK, 2004) have also been proposed. In this work, the MemProc architecture is compared with the 16-bit 5-stage, pipelined version presented in Figure 5.1.

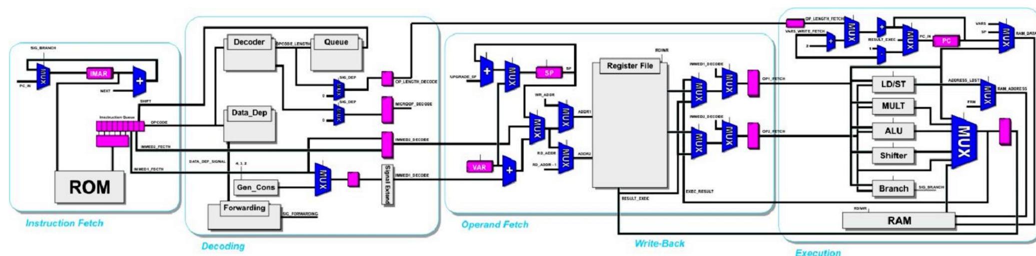


Figure 5.1: FemtoJava pipeline block scheme.

The FemtoJava architecture illustrated in Figure 5.1 is a pipelined architecture with 5 stages: the instruction fetch stage, the decoding stage, the operand fetch stage, the write-back stage and the execution stage. This architecture also counts with the forwarding unit in other to accelerate the delivery of operands to the execution stage.

The other architecture that was used to evaluate the MemProc architecture is the 5-stage pipelined MIPS illustrated in Figure 5.2. It is a Harvard architecture with a reduced instruction set. Differently from FemtoJava, the MIPS architecture has the instruction decode together with the operand fetch. On the other hand, the FemtoJava processor has the data memory access together with the execution stage and in the MIPS processor they are in different stages.

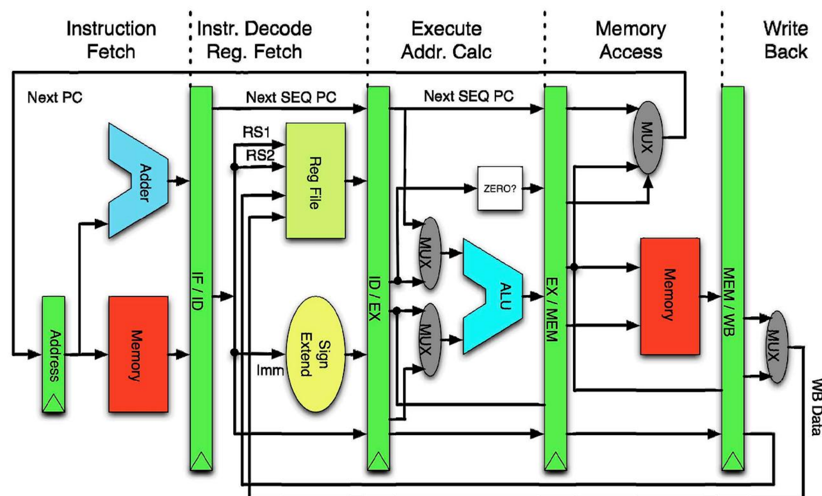


Figure 5.2: The pipelined MIPS architecture.

5.2 Tools Used in the Fault Injection, Performance and Area Evaluation

All architectures were described in a tool named CACO-PS (BECK, 2003a). As the name says, this tool is a cycle-accurate simulator, which performs the architectural behavioral simulation cycle by cycle. The CACO-PS tool uses basically three descriptions files to work.

An architecture description file is used to list the components that take part of the architecture, its inputs, outputs and control signals.

A behavioral description file describes all components that are part of the architecture. In this file, the behavior of each component is described using the C language, and this description allows each component to be instantiated as many times as necessary, and in any architecture where it is required.

The third and last file is the power description file. This file has the description of the function that will be executed to calculate the power consumption of the component, according to its transition activity. In this work the power consumption was not evaluated, therefore this file was not necessary.

The CACO-PS tool has the option to load the program and data codes from a memory initialization file.

The FemtoJava architecture description file was already described by another student, so it was only necessary to describe the MemProc architecture and the MIPS one in order to run the performance evaluation. Both architecture description files can be found in the Appendixes B and C respectively.

For the fault injection procedure it was necessary to add different components to the architectural description file, in order to simulate the faulty behavior of all the three architectures. To simulate the behavior of SETs in the combinational hardware, a component to flip the selected bit of the hit component output just for the duration of one cycle was created. On the other hand, if the component that was hit is a memory element, the kind of event generated is a SEU, whose effects remain active until a new value is written in the memory element. To simulate this faulty behavior, a function already present in the tool was used to write values in memory elements.

To describe each architecture version for fault injection, it was necessary to add one component to inject fault for each component in the architecture, according to the type of event the component receives, SEU or SET. In other to accelerate the insertion of these extra components for all the three architectures, it was created a C program that reads the architecture description file and creates the architecture description file for fault injection automatically, saving time and avoiding human errors in the conversion. To test if the õconversion programö created the faulty architecture correctly, a simple test was done. The õfaultyö architecture ran a selected application without the fault injection and the program result was compared with the normal architecture running the same application. This test was repeated for all available applications and, since the final result was the same for both architectures, it was verified that the õconversion programö did the conversion with no error.

In other to evaluate the maximum frequency each architecture supports, the architectural critical paths of the three architectures were described in VHDL and synthesized for a 0.35 m cell library in the Leonardo Spectrum tool (MENTOR, 1981). This tool was also used to evaluate the area consumption in terms of õequivalent gatesö for the more complex components such as the decoders and queues and registers of the FemtoJava and the MIPS architecture. The other components were evaluated as follows: all AND, XOR, NAND, NOR and NOT gates were considered to have the same area, equivalent to one õequivalent gateö like the ones from the Leonardo tool. The 2:1 multiplexors were considered to be equal to one õequivalent gateö and the other multiplexors were constructed with 2:1 multiplexors, to be calculated as one õequivalent gateö for each 2:1 multiplexor.

5.3 FAULT RATE AND AREA EVALUATION

To evaluate the fault rate of the processors, random faults were injected during their operation. During fault injection, the behavior of each processor was compared to the behavior of its fault free version when executing the same application with the same data.

Since some faults may hit parts of the circuit which are not being used at a specific moment in time, to detect if a fault has been propagated or not it is not necessary to compare the value of all functional units or registers. It is only necessary to compare those components that are vital for the correct operation of the system. For the FemtoJava and the MIPS processors, the units to be checked are the program counter, in order to detect wrong branches, and the RAM data and address registers during write operations, to identify silent data corruption (SDC). In the case of MemProc, besides the program counter, the microcode counter was checked to identify wrong branches and the write address and write data registers contents were checked to identify SDC. Figure 5.3 depicts the fault injection scheme implemented to measure fault rate in both processors. The CACO-PS tool has also been used to implement the fault injection and detection circuits.

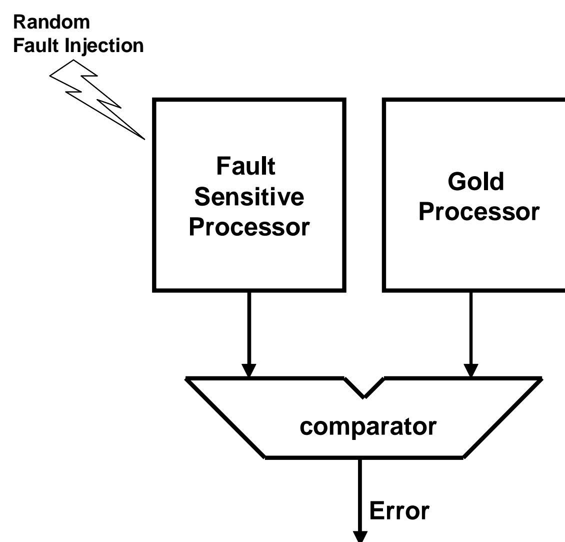


Figure 5.3: Error detection scheme.

It is clear that the probability of a component being hit by a fault increases with the area of the component. So, to be as realistic as possible, the random fault injector was implemented following this probabilistic fault behavior. To do so, it was created a file with all the important information about the components, such as component size in number of gates, the component type (memory or combinational), number of outputs and outputs widths. When the fault injection process starts, this component information file is loaded by the random fault injector and is used to determine which is the component that fails in each fault injection cycle, according to a probability based on its area.

Another important variable in the fault injection process is the amount of faults that are injected and the interval between the fault injections. In this work, we decided to use a so called environmental acceleration (MITRA, 2005), otherwise, we would have to wait for long simulation times in order to get an error. To make calculations easier, we assumed that the particle flow is able to produce 1 SEU or SET per cycle in the FemtoJava processor, which is the one with the biggest sensible area. To calculate the corresponding number of faults per cycle for the MIPS and the MemProc processors according to their sensible area and maximum frequency, it was used the area and frequency information that was obtained with the Leonardo Spectrum tool, as explained in the previous section. Table 5.1 illustrates those results and the corresponding time between fault injections for all processors.

Table 5.1: Area and time between fault injections.

Architecture	ROM (bits)	Op. Masks mem. (bits)	Opcode mem. (bits)	# of sensible gates	Max. freq. (MHz)	# faults per cycle	Time bet. fault inj.
MemProc	1,792	19,712	40,326	1,409	254	1/130	514,3 ns
MIPS	2,488	-	-	9,619	54	1/4	75,3 ns
FemtoJava	600	-	-	23,918	33	1	30,3 ns

The first column of Table 5.1 presents the size of the ROM memory, also know as code memory, for the õbubble sortö application. We can see that the FemtoJava architecture has the lowest memory consumption. That happened because the Java code operates using the

operands that are at the top of the stack and stores the result of the operation at the top of the stack automatically. This strategy saves some space, since the instruction does not need to indicate where the operands are, nor where the result needs to be stored. Also, the MemProc and the MIPS instructions have always the same size, even if the instruction does not use all its width. On the other hand, the Java code has instructions with 1, 2, and 3 bytes of width, and consequently does not waste memory space as the MIPS and the MemProc do. Consequently, the FemtoJava decoder is more complex and demands more area than the MIPS and the MemProc ones.

In Table 5.1 we can see why MemProc is called a memory-based processor. In the MemProc architecture, the combinational circuit is very small when compared to the size of its memory elements. In our approach, all memory contents are not sensible to faults, since we are simulating the use of fault tolerant memory technologies, such as MRAM, FRAM, and flash memories, already referred to. Even for the MRAM and FRAM technologies, the decoding circuit is not tolerant to faults. So, to be as realistic as possible, the area corresponding to these circuits was also counted together with the sensible gates of MemProc, and the decoding circuit was constructed as a separated component at the architecture description, to have its behavior simulated during the fault injection. The two MemProc memories have more than 60,000 bits together. If we consider that each 2 bits of memory have the same area of one equivalent gate, then the total area introduced by the memory components is equal to more than 30,000 gates, which makes MemProc have the largest area among the three processors. However, the area corresponding to the memory elements is not sensible to faults.

The fault injection process injected random faults according to the probability of the component being hit, together with the calculated time between fault injections, which is in the 8th column of Table 5.1. In this process, faults were injected until one error or a silent data

corruption (SDC) was detected. This process was repeated 100 times and the mean time to failure for these 100 errors for all the three architectures was calculated and is presented at Table 5.2.

Table 5.2: Fault rates for all architectures.

Architecture	# of injected faults	# of errors	# of SDC	# of cycles	MTTF (μs)
MemProc	4,943	98	2	865,412	31.83
MIPS	2,160	90	10	4,320	1.83
FemtoJava	2,127	84	16	2,127	0.64

Table 5.2 lists the fault injection results for the MemProc, MIPS and Femtojava processors. In the second column, the number of simulated injected faults in the entire process until the detection of 100 errors or SDCs is shown. The third and fourth columns present the number of errors and SDCs that occurred during this process, respectively. The fifth column shows the total number of cycles that were necessary to detect all 100 errors and SDCs. In the last column, one can see the corresponding Mean Time to Failure value. as one can see, the MTTF of the MemProc architecture is more than 49 times bigger than the FemtoJavaøs one.

When comparing the Mean Time to Failure of MemProc and the MIPS architecture, one can see that the MTTF of the proposed architecture is more than 17 times bigger than the MIPS one.

These results show the significant reduction in the MTTF that can be obtained by using the proposed architecture. In the next section, performance results are presented, and show that, despite the fault tolerance improvement introduced by the MemProc architecture, no performance degradation is observed at the proposed architecture when compared to the FemtoJava and the well known MIPS architectures.

5.4 PERFORMANCE EVALUATION

The performance evaluation was done using four different application programs, with different processing characteristics: three sort algorithms (the bubble, insert and select sort algorithms), one DSP algorithm, and the IMDCT (Inverse Modified Discrete Cosine Transform) algorithm, part of the MP3 coding/decoding algorithm, were executed in MemProc, MIPS and FemtoJava architectures. The obtained results are shown in Table 5.3.

Table 5.3: Performance when executing benchmark applications.

Application	MIPS (54 MHz)		FemtoJava (33 MHz)		MemProc (254 MHz)		Performance ratio compared to:	
	# of cycles	Comp. time (μs)	# of cycles	Comp. time (μs)	# of cycles	Comp. time (μs)	FJ	MIPS
Bubble Sort	2,280	42.2	2,468	74.8	4,720	18.4	4.06	2.29
Insert Sort	1,905	35.3	1,571	47.6	2,508	9.8	4.86	3.60
Select Sort	1,968	36.4	1,928	58.4	2,501	9.7	6.02	3.75
IMDCT	38,786	718.3	41,061	1,244.2	142,951	562.8	2.23	1.28

From Table 5.3 we can see that MemProc executes the bubble sort algorithm in approximately 4.7 thousand cycles, while FemtoJava and MIPS take the half of the number of cycles. As stated before, MemProc requires several cycles to perform arithmetic (bit serial) operations, and the number of cycles also depends on the value of the operands. That is the reason why the number of cycles spent by MemProc is higher than the other architectures. On the other hand, MemProcøs critical path is determined by the access time of the microcode memory and the operational masks memory, while in FemtoJava and MIPS the critical path is determined by the multiplier delay. So, the maximum frequency of MemProc is more than 7 times higher than that of FemtoJava and almost 5 times higher than that of MIPS, and, as consequence, the MemProc is more that 4 times faster than FemtoJava and more than 2 times faster than MIPS when running the sort algorithms.

If we look at the results when executing IMDCT we can see that MemProc was only 2.23 times faster than FemtoJava and 1.28 times faster that MIPS. That happened because this algorithm makes intensive use of the multiply instruction, which can take up to 48 cycles to be executed in MemProc. It is important to mention here that MemProc is a multi-cycle machine, while FemtoJava and MIPS are pipelined ones, which are expected to be faster than their multi-cycle versions. So, we can conclude that if we were comparing MemProc with the multicycle versions of FemtoJava and MIPS, performance results would be even better. Also, the performance gains of MemProc come from the fact that the number of cycles it takes to perform an operation depends on the operation and on the operands value. For instance, let us consider that FemtoJava needs 1 cycle to perform one add operation. Since MemProcøs frequency is more than 7 times higher, if the operands are such that the number of carry cycles is less than 7, MemProc will finish the addition operation earlier than FemtoJava.

To evaluate how the value of the operands contributes to the MemProc performance gains, the mean time to execute each type of instruction in the MemProc and the MIPS architectures were simulated for the sorts and the IMDCT algorithms. The results are presented in Figure 5.4. The MIPS architecture always takes the same time to execute each type of instruction, so its results are independent to the application.

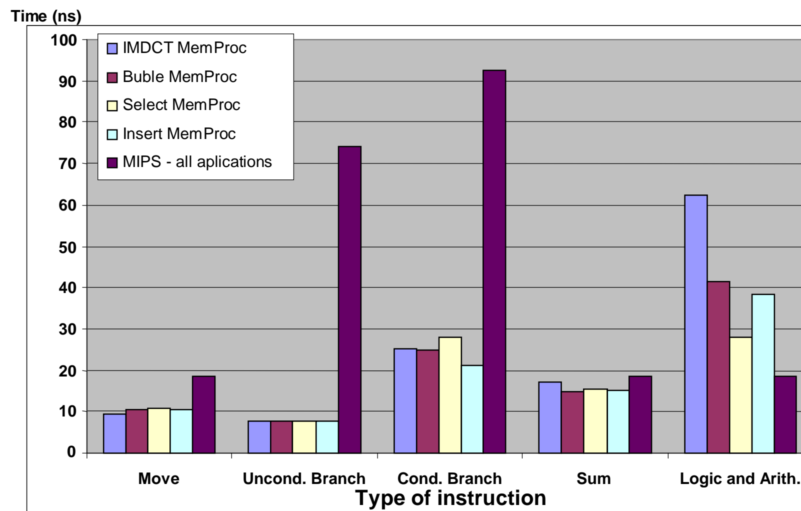


Figure 5.4: Mean time to execute each type of instruction for all applications

In Figure 5.4 one can see that MemProc executes move instructions, conditional and unconditional branches faster than MIPS. On the other hand, it is slower than MIPS to execute logic and arithmetic operations, except for add instructions. That happened because the proposed architecture takes larger number of cycles to implement the multiply and the subtraction instructions. In the IMDCT application, the percentage of arithmetic and logic instructions is 63%, which explains why MemProcøs performance for this application was slower than for the sort applications, in which the percentage of logic and arithmetic instructions was 51%, 36%, and 49% for bubble, select, and insert, respectively. One can conclude that the lower the percentage of arithmetic instructions, such as multiplications and subtractions, the higher is the performance of MemProc in comparison to MIPS. This results show that MemProc has greater performance when executing control flow than data flow applications. The reason for the good performance when executing conditional branches is the unique way MemProc executes the comparison. In traditional architectures, such as MIPS and FemtoJava, the comparison in the conditional branch is done by the subtraction operation. If MemProc would do the comparison using subtraction it would take 18 cycles to get the result of the comparison due to the time it takes to get the last carry propagation, which indicates the signal of the subtraction. The way MemProc does the comparison of two values is by identifying which is the value that has the most significant level -1ø bit in a position that the other value does not have, by using binary search. For instance, let us consider the example in Figure 5.5.

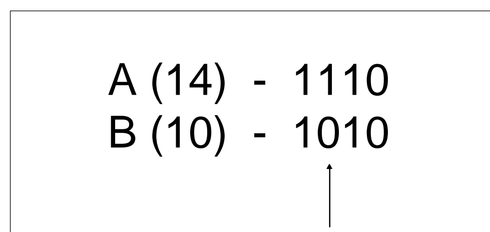


Figure 5.5: The way MemProc does comparisons.

In Figure 5.5, two values in decimal, 14 and 10, and their binary representations are shown. From this figure, one can see that the value that has the most significant level ÷1ø bit in a position that the other value does not have is the ÷Aø value, therefore the ÷Aø value is greater than the ÷Bø value. MemProc performs binary search to find which of the two values is the biggest and, for a value of 16 bits, MemProc takes at most 4 cycles to identify the biggest one. In case of two negative values, the value that has the most significant level ÷1ø bit in a position that the other value does not have is the lowest one, due to the 2øs complement representation for negative values. Consequently, MemProc takes 1 cycle to identify if any of the two numbers is negative, 1 more to see if they are equal and 4 more to identify which is the biggest one, which gives us a total of only 6 cycles at most, to perform any comparison operation. As it was said before, if MemProc would do comparison trough subtraction, it would take 18 cycles, which is 2 times more cycles than MemProc actually takes.

6 I-IP: A NON-INTRUSIVE ON-LINE ERROR DETECTION TECHNIQUE FOR SOCS

The growing demands and competitive needs of the embedded systems market, with ever shrinking time to market requirements, has made the use of SoCs incorporating previously tested IPs, or the use of FPGAs with built-in factory supplied processors, preferred alternatives to provide fast deployment of new products. As to the software of SoCs, the use of standard library applications, for which the source code is not always available, provides another path to fast product development. Even for these systems, the technology evolution towards nanoscale brings along higher sensitivity of the hardware to radiation induced soft errors. For this kind of SoCs, neither the hardware nor the software can be modified, either because of the high costs involved in adding extra hardware, or simply because the hardware is not accessible or the source code is not provided.

In this chapter we describean infrastructure IP (I-IP) that can be inserted in the SoC without any change in the core processor architecture, able to monitor the execution of the application and detect control flow and instruction execution errors generated by transient faults. In the first section the I-IP approach proposed in (LISBÔA, 2006) is presented, together with a description of its internal blocks. The next section describes the adaptations that were implemented in the I-IP for the MIPS processor case study.

6.1 The Proposed Approach

The system to be protected is a SoC where a processor core is used to run a software application, and the proposed approach can be used to harden applications executed by any processor core, independent of its internal architecture. In order to confirm this assumption, we have conducted experiments aiming the implementation of the I-IP in the well known and widely used MIPS RISC processor. The proposed I-IP is inserted between the memory storing

the code and the main processor core, and monitors each instruction fetch operation. In this work it is assumed that the bus connecting the instruction cache to the processor is not accessible from outside the core, as it often happens for processor cores, and therefore it is assumed that the instruction cache either does not exist, or is disabled. Moreover, it is considered that the instruction memory and the data memory located outside the processor are hardened with suitable error detection/correction codes or somehow protected, and so the data read from memory can be considered reliable.

6.1.1 The I-IP

The I-IP aims at minimizing the overhead needed to harden a processor core, with particular emphasis in minimizing the amount of memory used by the hardened application, and in being applicable even when the applicationøs source code is not available, by exploiting the concepts described in the following paragraphs.

Instruction hardening and consistency check: data processing instructions are executed twice, producing two results that are checked for consistency; and an error is notified whenever a mismatch occurs.

Control flow check: each time the processor fetches a new instruction, the fetch memory address is compared with the expected one, and an error is notified if a mismatch is detected.

As stated before, the I-IP is inserted between the processor core and the code and data memories, as illustrated in Figure 6.1, with the indication of the address bus, control bus and data bus. While the I-IP must be tailored to the specific core processor in a given SoC, the architecture and the technique described here are generic, and can be implemented in any SoC in which additional modules can be inserted.

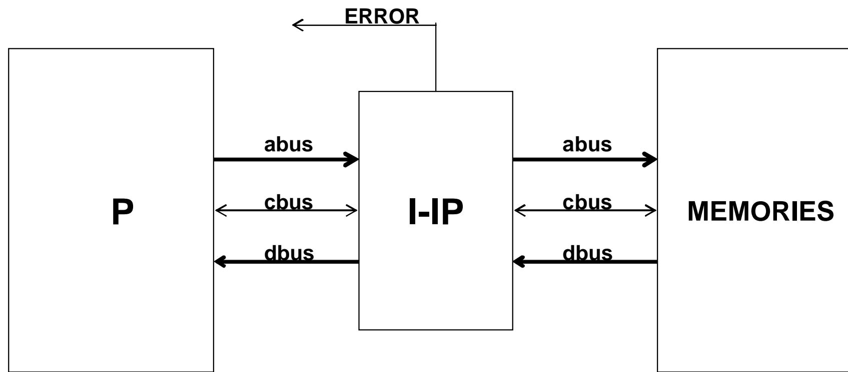


Figure 6.1: I-IP overall architecture.

Whichever the core processor existing in the SoC, the I-IP implementing the concepts of the proposed technique works as follows.

Instruction hardening and consistency check: the I-IP decodes the instructions fetched by the processor. Each time a data processing instruction is fetched, like that shown in Figure 6.2, whose format is `opcode dst,src1,src2`, and which is stored in memory at address `FETCH_ADX`, the I-IP replaces it with the sequence of instructions in Figure 6.3, which is sent to the processor.

```
FETCH_ADX: opcode dst,src1,src2
```

Figure 6.2: Original instruction.

```
store  I-IP-adx,src1
store  I-IP-adx,src2
opcode dst,src1,src2
store  I-IP-adx,dst
branch FETCH_ADX+OFFSET
```

Figure 6.3: Source operands and result fetching.

Therefore, from the point of view of the processor, in this case the fetched instructions are no more those contained in the code memory, but those issued by the I-IP. The sequence of instructions that replaces each data processing one includes two instructions whose purpose

is to send to the I-IP the value of the source operands of the instruction. The third instruction (in boldface) is the original instruction coming from the program, while the fourth one is used to send to the I-IP the computed result. Finally, the last instruction is used to resume the original program execution, starting from the instruction following the original one, which is located as address `FETCH_ADX+OFFSET`, being `OFFSET` the size of the original instruction. Concurrently to the main processor, the I-IP executes the fetched data processing instructions by exploiting its own arithmetic and logic unit, and compares the obtained results with that coming from the processor. In case a mismatch is found, it activates an error signal, otherwise the branch instruction is sent to the core processor, in order to resume its normal program flow.

Control flow check: concurrently with instruction hardening and consistency check, the I-IP also implements a simple mechanism to check if the instructions are executed according to the expected flow. Each time the I-IP recognizes the fetch of a memory transfer, a data processing, or an I/O instruction stored at address A, it computes the address of the next instruction in the program (A_{next}) as $A+offset$, where *offset* is the size of the fetched instruction. Conversely, each time the I-IP recognizes the fetch of a branch instruction, it computes the address of the next instruction in the two cases corresponding to the branch taken situation (A_{taken}) and to the branch not taken one (A_{next}). The former is computed taking into account the branch type, while the latter is computed as $A+offset$, where *offset* is the size of the branch instruction. When the next instruction is fetched from address ‘D’, the I-IP checks if the program is proceeding along the expected control flow by comparing the value of D with the destination address calculated as described here. If D differs from both A_{next} and A_{taken}, the error signal is raised to indicate that a fetch from an unexpected address has been attempted.

6.1.2 The I-IP Modules

The I-IP that was developed is organized as shown in Figure 6.4, and it is composed of the following modules:

1) CPU interface: connects the I-IP with the processor core. It decodes the bus cycles the processor core executes, and in case of fetch cycles it activates the other modules of the I-IP.
2) Memory interface: connects the I-IP with the code and data memories, to allow access to the program instructions and to the data sent by the processor. This module executes commands coming from the õFetch logicö, and handles the details of the communication with the memory.
3) Fetch logic: issues to the õMemory interfaceö the commands needed for loading a new instruction in the I-IP and feeding it to the õDecode logicö.
4) Decode logic: decodes the fetched instruction, whose address in memory is A, and sends the details about the instruction to the õControl unitö. This module classifies instructions according to three categories:
    i. Data processing: if the instruction belongs to the set of instructions that the I-IP is able to harden, which is defined at design time, the I-IP performs instruction hardening and consistency check. Otherwise, the instruction is treated as õotherö, as described in item õcö. Moreover, for the purpose of the control-flow check, the address A_{next} of the next instruction in the program is computed, as described previously.
    ii. Branch: the instruction may change the execution flow. The I-IP forwards it to the main processor and it computes the two possible

addresses for the next instruction, A_{next} and A_{taken}, as described previously.

iii. Other: the instruction does not belong to the previous categories. The I-IP forwards it to the main processor and only computes the address of the next instruction in the program (A_{next}), , as described previously.

5) Control unit: supervises the operation of the I-IP. Upon receiving a request for an instruction fetch from the õCPU interfaceö, it activates the õFetch logicö. Then, depending on the information produced by the õDecode logicö, it either issues to the main processor the sequence of instructions summarized in Figure 6.3, to implement instruction hardening and consistency check, or it sends to the processor the original instruction. Moreover, it implements the operations needed for control-flow check. Finally, it receives interrupt requests (IRQs) and forwards them to the processor core at the correct time. This means that, in case an IRQ is received by the I-IP during the execution of a substitute sequence of instructions sent by the I-IP to the core processor, this IRQ will be forwarded to the core processor only after all the hardening instructions have been fully executed.

6) ALU: it implements a subset of the main processorøs instruction set. This module contains all the functional modules (adder, multiplier, etc.) needed to execute the data processing instructions the I-IP manages. Its complexity varies according to the set of instructions to be hardened, which is chosen at design time.

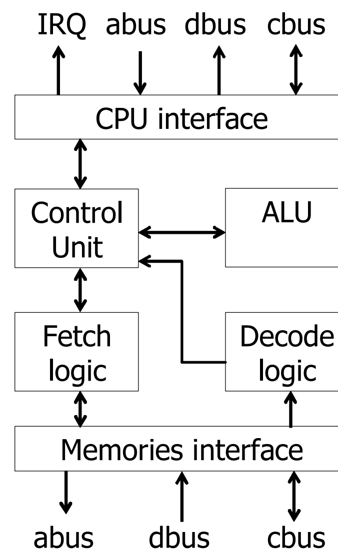


Figure 6.4: Architecture of the I-IP.

Two customization phases are needed to successfully deploy the I-IP in a SoC:

Processor adaptation: the I-IP has to be adapted to the main processor used in the SoC. This customization impacts the õCPU interfaceö, the õMemory interfaceö, the õFetch logicö, and the õControl unitö only. This phase has to be performed only once, each time a new processor is adopted. Then, the obtained I-IP can be reused each time the same processor is employed in a new SoC.

Application adaptation: the I-IP has to be adapted to the application that will be executed by the main processor (mainly affecting the set of data processing instructions to be hardened by the I-IP). This operation impacts the õDecode logicö and the ALU of the I-IP, as it defines which instructions the I-IP will execute and check. In this phase, designers must decide which of the instructions of the program to be executed by the main processor have to be hardened. The application adaptation phase may be performed several times during the development of a SoC, for example when new functionalities are added to the program running on the main processor, or when the designers tune the SoC area/performance/dependability trade-off.

6.2 Processor and Application Adaptations for MIPS

In this section we will present the processor and application adaptations that were implemented in the proposed I-IP to harden the instruction execution and the control flow of the widely used RISC MIPS processor. The MIPS used in our experiments has a 16-bit RISC architecture, with a 5-stage pipeline, and no branch prediction. The selection of this architecture was due to its widespread use in the implementation of SoCs by the industry.

Because the MIPS architecture has a 5-stage pipeline, with fetch, decode, execution, memory write and write back stages, the I-IP works (only from the logical standpoint) as being an additional stage, between the fetch and the decode stages. That happens because the I-IP requires one cycle to decode the fetched instruction and decide which instruction(s) to send to the processor, and that makes the processor receive the fetched instruction one cycle later.

Due to this virtual extension of the number of pipeline stages, the I-IP needs to send a different sequence of instructions, depending on the fetched one, to prevent erroneous situations:

1) In the case of an unconditional branch, the number of instructions that need to be flushed from the pipeline is increased by one, because, as explained before, the I-IP works as an extra pipeline stage. To correct this situation, the I-IP sends to the core processor an extra `nop` (*no operation*) instruction, each time an unconditional branch is fetched.
2) When a `jal` (*jump and link*) - a subroutine call instruction - is executed, the MIPS processor saves the subroutine return address in a register. Since the I-IP causes a delay of one cycle in the execution of instructions, the saved address is also one cycle ahead the correct one. To solve this problem, when fetching a `jal` instruction the I-IP sends to the core processor one instruction that

restores the PC value to the correct one, followed by a `j` (*jump*) instruction, instead of only sending the `jal` one. The first instruction is used to save the correct address in the register that is used to store the return address, and the `j` instruction performs the jump to the subroutine entry point;

3) In case of a `jr` (*jump through register*) instruction, the I-IP needs to get the address value stored in the register that indicates the address, to check if the branch was taken correctly. Therefore, the I-IP has to provide a `sw` (*store word*) instruction to receive the target address of the branch before the original `jr` instruction is executed.

Due to the pipelined architecture of MIPS, the I-IP must wait a few cycles until a branch is executed and only then compare the calculated destination address with the one in the program counter. Therefore, the I-IP has an internal circular register file, used to store up to four destination addresses, that will be compared to the program counter a few cycles later. In the next chapter the experimental results of the I-IP alternative for the MIPS RISC processor are presented and compared with the results presented in (LISBOA, 2006).

7 I-IP EXPERIMENTAL RESULTS

This chapter presents the reduction of failures that can be obtained by applying the I-IP technique to the MIPS architecture and compares the achieved results with those of the implementation of the I-IP in the 8051 processor obtained in (LISBOA, 2006). In the first section, the fault injection procedure that was implemented in order to test the proposed IP is described. The second section presents the fault detection results obtained for the two architectures, the 8051 and the MIPS, together with the area and performance overhead discussions.

7.1 FAULT INJECTION EXPERIMENTS

To evaluate the performance of the I-IP in instruction hardening and control flow error detection, the tool named CACO-PS (Cycle-Accurate Configurable Power Simulator), described in a previous chapter, was used to simulate the architecture of the SoC and check the results of fault injection.

The I-IP and the MIPS architectures were described in the language used by CACO-PS. The fault model used in all experiments is the SEU in internal memory elements of the core processor. During the fault injection procedure, 2,000 faults were injected randomly in time and space, causing SEUs in randomly chosen bits of the MIPS architecture registers, while executing a software implementation of the Viterbi algorithm for encoding a stream of data, like it was done in (LISBOA, 2006) for the 8051 processor.

To detect if a fault caused an error, two copies of the SoC (including the MIPS core processor, the I-IP and independent code memories), both running the same application, have been used. Faults have been injected in one of the two architectures, while the other remained free of faults. Then, at every core processor cycle, the simulation tool compared the value of the program counters from both copies, to check if a control flow error occurred. In order to

check if an instruction execution error occurred, the RAM memory content was also monitored, by comparing the address and the data of the memory write operations.

At the same time, all errors detected by the I-IP were recorded in a log file, indicating the type of error that was detected and other information used in the analysis of the simulation results, which will be discussed in the next section. Figure 6.5 illustrates the error detection scheme described here.

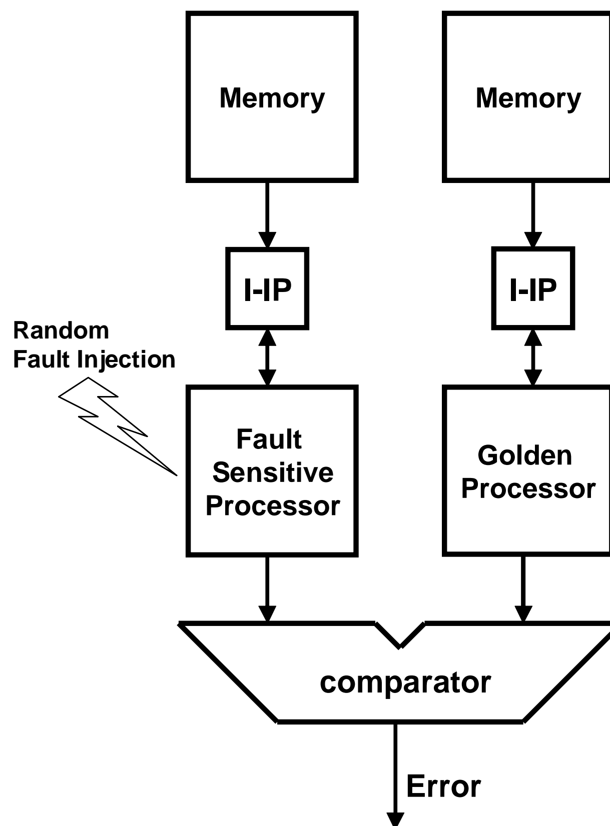


Figure 6.5: Error detection scheme.

To evaluate how the quantity of hardened instructions impacts the area and performance overheads, two experiments were implemented, one hardening only the ADDU instruction and the other hardening the ADDU, ANDI and SRA instructions. In the MIPS experiment, the choice of instructions to be hardened in the was based on runtime statistics, shown in Table 6.1, and not on static analysis of the code, as in the 8051 experiment.

Table 6.1: Runtime frequency of instructions.

Viterbi execution (7,182 instructions)		
Instruction	Frequency	%
LW	2,105	29.3
SW	1,349	18.8
ADDU	1,072	14.9
ANDI	716	10.0
SRA	716	10.0
ADDIU	429	6.0
SLL	271	3.8
SUBU	152	2.1
JALL	77	1.1
SRL	76	1.0
JR	76	1.0
Others	143	2.0

Because the experiments with the MIPS core have been done using a cycle-accurate simulator, only 1,000 faults have been injected in each of the implementations of the I-IP with the MIPS core, and the obtained results are shown in Table 6.2.

Table 6.2: Error detection results for the two architectures.

Application	8051		MIPS	
Hardened Instructions	INC	INC ADD	ADDU	ADDU ANDI SRA
Reduction of failures (%)	81.3	87.5	74.5	79,2
Area overhead due to I-IP (%)	13.1	15.7	12.7	12.9
Performance overhead (%)	292.0	314.0	99.0	196.8

7.2 RESULT ANALYSIS

The experiments results described in (LISBOA, 2006) have shown that not all the faults can be detected by the I-IP in the 8051 processor. Indeed, some failures have been observed for the hardened SoC. Some of the escaped faults affected memory elements that change the configuration of the processor core. For example, they change the register bank select bit, switching from the used register bank to the unused one. This kind of fault makes

both the I-IP and the main processor fetch the operand from a wrong source, which makes them produce the same wrong operation result. Since the I-IP detects faults by testing if the two results are different, these faults escape from the error detection mechanisms provided by the I-IP. The other type of faults that escaped affect the execution of branch instructions in such a way that the taken branch is consistent with the program control flow, but it is taken to the wrong destination. A typical example of this type of fault is an SEU affecting the carry bit of the processor status word that hits the SoC before a conditional branch is executed. In this case, the wrong execution path is taken, based on a wrong value of the carry flag. However, the control flow is transferred to a legal basic block, which is consistent with the programøs control flow, and therefore it escapes the control flow check that the I-IP employs. Finally, some of the escaped faults affected un-hardened instructions, mainly the LW (load word) and the SW (store word) instructions, due to its high occurrence in the program.

When it comes to the area overhead analysis, one can see from Table 6.2 that the I-IP introduces a slightly smaller area overhead in the MIPS based SoC, due to the fact that the MIPS core processor is much more complex, and therefore larger, than the 8051 microcontroller. However, the reduction was not very significant, because the I-IP implemented with the MIPS core must keep track of the evolution of the instructions inside the pipeline, which also requires a more complex hardware than that of the I-IP for the 8051. Concerning performance, the implementation for MIPS has provided a significantly smaller overhead. At this point, it is worth to recall that the performance overhead is mainly due to the execution of additional instructions sent by the I-IP to the core processor, as it was presented in the previous chapter, each time an instruction that must be hardened is fetched from memory by the core processor.

When analyzing the percentage of reduction of failures, one can see that the ability to detect faults in the MIPS implementation was smaller than that in the 8051 implementation.

The fault model used in all experiments is the SEU in internal memory elements of the core processor. Therefore, since the MIPS processor is pipelined, there is a larger amount of memory elements subject to SEUs in its architecture than in the 8051 microcontroller, where most of the memory elements are registers used for data or address storage, not for control.

The use of a cycle-accurate simulator in the experiments with the MIPS processor, however, provided more information about the cases in which faults are not detected by the I-IP, thereby allowing a more detailed analysis of the problem. So, besides those cases already mentioned for the 8051 microcontroller, our analysis has shown that, among the undetected faults, a large number was due to SEUs affecting the register file of the MIPS processor before the operands are read and their values forwarded to the I-IP. In those cases, the same corrupted data values are used by the core processor and by the I-IP during the parallel execution of the data processing instruction, and therefore the results are the same and no error is flagged. These findings point out that the protection of some internal memory elements of the core processor, such as the register file, would be an improvement factor for the fault coverage, when the approach proposed here is applied.

8 CONCLUSIONS AND FUTURE WORK

8.1 CONCLUSIONS

In this work, two candidate solutions to cope with the SEU and SET problem that is concerning designers of digital systems for future and even current technologies were presented. The first solution presented here was the MemProc processor core architecture, based on the use of memory technologies not sensible to SEU and reduced combinational circuits. The second solution was the I-IP core for the MIPS processor, which is proposed for cases where neither the hardware nor the software of the system can be modified.

Both solutions have their pros and cons. As an example, in the MemProc case the final area of the solution, increased mainly due to the two memories (the microcode and the operation masks memories), was more than 2 times larger than the one in MIPS and 1.3 times than that of FemtoJava, but on the other hand, the fault tolerance and performance results have shown a 17 times bigger mean time to failure, and more than 1.2 performance gain when compared to MIPS and more than 49 times bigger MTTF and 2.2 performance gain when compared to FemtoJava. The proposed architecture, while not being a final solution, reflects the focus in the search for new processor design alternatives that might be used in the future, when current ones will start to fail due to the weaknesses of new technologies. It innovates in several design features, even providing better performance when compared to a well known architecture for embedded applications (the MIPS processor), while providing much more reliability against transient faults.

In the case of the I-IP core, results have shown that this approach can be implemented for any kind of architecture, either RISC (like the MIPS case study presented here) or CISC (like the 8051 presented in (LISBOA, 2006)). Although the performance overheads are considerably high, due to the number of instructions hardened, the area overhead is below

15.7%, with more than 74.5% of the errors detected. The great advantage of this approach is that it is neither hardware nor software intrusive, which makes it easily adaptable for any kind of processor core, with the possibility of different configurations in the number of hardened instructions and control flow error detection.

In this work, two different solutions were presented to cope with particle hit induced events that is foreseen in the new technologies. Although the presented solutions do not eliminate the possibility of a soft error occurrence, a significant reduction in the soft error rate and error detection percentage, with considerably low overheads, were achieved with the solutions here presented, which represent an important step towards a complete and feasible solution for reliable systems in future technologies.

8.2 FUTURE WORK

The MemProc processor architecture presented here has shown great performance results due to the architectural innovations that accelerated addition and comparison operations, as it was described in a previous chapter. Although, some operations, such as subtraction and multiplication, still need to be improved in other to reduce the number of cycles they take to be executed, which are now 18 for the subtractions, and from 35 to 49 for the multiplications. Another point that can be improved in the MemProc architecture is the reduction of the size of the microcode and the operation masks memories, which will positively impact the area overhead introduced by these components.

In the case of the I-IP, the next steps will be the repetition of the experiments with a broader set of benchmark applications, and the development of tools to automate the generation of new I-IP versions for other core processors, according to the set of instructions that need to be hardened, and also the type of control flow instructions to be monitored.

Since this work proposes two widely different solutions, an innovative and interesting evolution of this work is the integration of both solutions in an unique architecture designed for fault tolerant applications. This way, the I-IP core will have to be modified to harden the MemProc instruction set and monitor the two main sources of control deviation, which are the microcode and the ROM memories. Since the MemProc architecture is vulnerable at the instruction execution sector, the I-IP would complement the good results of MemProc by hardening the instructions that are being executed. On the other hand, the I-IP vulnerability stands at the memory elements, which in the MemProc case are hardened by the MRAM technology. This way, the integration of these two solutions promises to provide good results in terms improving the fault tolerance during the execution of critical applications.

REFERÊNCIAS

AHO, A.; SETHI, R.;ULLMAN, J. **Compilers:** principles, techniques and tools. Boston: Addison-Wesley Longman Publishing Co., Inc, 1986. 796 p. ISBN:0-201-10088-6.

ALEXANDRESCU, D.; ANGHEL, L.; NICOLAIDIS, M. New methods for evaluating the impact of single event transients in VDSM ICs. In: IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS WORKSHOP, DFT, 17., Vancouver, Canada, November 2002. **Proceedingsí** Los Alamitos, CA: IEEE Computer Society, p. 99-107, 2002. ISBN: 0-7695-1831-1.

ALKHALIFA, Z. et al. Design and evaluation of system-level checks for on-line control flow error detection. **IEEE Transaction on Parallel and Distributed Systems.** [S. l.] v. 10, n. 6, p. 627-641, June 1999.

ANGHEL, L.; NICOLAIDIS, M. Cost reduction and evaluation of a temporary faults detection technique. In: DESIGN, AUTOMATION, AND TEST IN EUROPE CONFERENCE, DATE, Paris, France, 2000a. **Proceedingsí** [S. l.]: ACM, Mar. 2000, p. 591-598.

ANGHEL, L.; ALEXANDRESCU, D.; NICOLAIDIS, M. Evaluation of soft error tolerance technique based on time and/or space redundancy. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 13., Manaus, Brazil 2000. **Proceedingsí** Los Alamitos, CA: IEEE Computer Society, Sept. 2000b, p. 237-242.

AUSTIN, T. M. DIVA: A dynamic approach to microprocessor verification. **The Journal of Instruction-Level Parallelism.** [S. l.], v. 2, May. 2000. Disponível em: <http://www.jilp.org/vol2>. Acesso em: November 2006.

BAUMANN, R. C. et al. H. Boron compounds as a dominant source of alpha particles in semiconductor devices. In: IEEE INTERNATIONAL RELIABILITY PHISICS SYMPOSIUM, 1995, Las Vegas, USA. **Proceedingsí** Los Alamitos, CA: IEEE Computer Society, p. 297ó302, 1995.

BAUMANN, R. C. Silicon amnesia: a tutorial on radiation induced soft errors. In: IEEE INTERNATIONAL RELIABILITY PHYSICS SYMPOSIUM, 2001. [S. l.] **Technical Note.** O arquivo pdf contendo os slides pode está disponível mediante requisição ao autor.

BAUMANN, R. C. Soft errors in advanced computer systems. **IEEE Design and Test of Computers**, v. 22, n. 3, p. 258-266, May/June. 2005.

BECK Fº, A. C. S. et al. CACO-PS: a general purpose cycle-accurate configurable power-simulator. In: BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGNS, SBCCI, 16., São Paulo, Brazil, 2003. **Proceedingsí** Los Alamitos, CA**:** IEEE Computer Society, p. 349, Sept. 2003a.

BECK F^{o}, A. C. S.; CARRO, L. Low power java processor for embedded applications. In: INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI-SOC, 2003, 12., Darmstadt, Germany. **Proceedingsí** [S. l.: s. n.], 2003b. p. 239-244.

BECK, F^{o}. A. C. S. et al. A VLIW low power java processor for embedded applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGNS, SBCCI, 17., Pernambuco, Brazil, Sept. 2004, **Proceedingsí** New York, NY: ACM Press, Sept. 2004, p 157-162.

BERNARDI, P. et al. A new hybrid fault detection technique for systems-on-a-chip, **IEEE Transactions on Computers**, [S. l.], v. 55, n. 2, p. 185-198, Feb. 2006.

BOSSEN, D.C. CMOS soft errors and server design. In: IEEE INTERNATIONAL RELIABILITY PHYSICS SYMPOSIUM, IRPS, Dallas, USA, April 2002. **Reliability Physics Tutorial Notes:** [S. l], IEEE Press, April 2002.

CONSTANTINESCU, C. Trends and challenges in VLSI circuit reliability. **IEEE Micro**, v. 23, n. 4, p. 14-19, New York-London: IEEE Computer Society, Jul.Aug. 2003.

CHEYNET, P. et al. Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. **IEEE Transactions on Nuclear Science.** New York, v. 47, n. 6, p. 2231-2236, Dec. 2000.

DEAN, M. et al. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 23., Philadelphia, USA, May 1996. **Proceedingsí** New York, NY: ACM Press, p. 191-202, May 1996.

DEAN, M. et al. Simultaneous multithreading: maximizing on-chip parallelism. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 25., Barcelona, Spain, June 1998. **Proceedingsí** New York, NY: ACM Press, p. 533-544, June 1998.

DODD, P.E. et al. Impact of substrate thickness on single-event effects in integrated circuits. **IEEE Transaction on Nuclear Science.** New York, USA, v. 48, n. 6, p. 1865-1871, Dec. 2001.

EDENFELD, D. et al. Technology Roadmap for Semiconductors. **IEEE Computer**, New York-London, v. 37, p. 47-56, Jan. 2004.

ELLIOTT, D.G. et al. Computational RAM: implementing processors in memory. **IEEE Design & Test of Computers,** New York, USA, v. 16, n. 1, p. 32-41, Jan/Mar. 1999.

ETO, A. et al. Impact of neutron flux on soft errors in MOS memories. In: IEEE INTERNATIONAL ELECTRON DEVICES MEETING, IEDM, San Francisco, USA, Dec. 1998. **Proceedingsí** [S. l.: s. n.], p. 367ó370, 1998.

GOLOUBEVA, O. et al. Soft error detection using control flow assertions. In: IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE, DFT, 18., Boston, USA, 2003. **Proceedingsí** Los Alamitos, CA**:** Computer Society, Nov. 2003, p. 581-588.

HARELAND, S. et al. Impact of CMOS process scaling and SOI on the soft error rates of logic processes. In: SYMPOSIUM ON VLSI TECHNOLOGY, Kyoto, Japan, 2001. **Digest of Technical Papers**. [S. l.: s. n.], June 2001, p. 73ó74.

HEIJMEN, T. Radiation-induced soft errors in digital circuits: a literature survey. **Philips Electronics Nederland BV 2002.** [S. l.: s. n.], p. 7-20, 2002.

HENTSCHKE et al. Analyzing area and performance penalty of protecting different digital modules with hamming code and triple modular redundancy. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGNS, 15., Porto Alegre, Brazil, 2002. **Proceedings í** Los Alamitos, CA: IEEE Computer Society, 2002. p. 95-100.

ITO, S.; CARRO, L.; JACOBI, R. Making java work for microcontroller applications. **IEEE Design & Test**, New York, v. 18, n. 5, p.100-110, Sept.Oct. 2001.

JOHNSON, B. W. **Design and Analysis of Fault Tolerant Digital Systems:** solutions manual. Reading, MA: Addison-Wesley Publishing Company, Oct. 1994.

KARNIK, T. et al. Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18 . **Digest of Technical Papers**. VLSI Circuits 2001. [S. l.], 2001, p. 61-62.

KARNIK, T. et al. Selective node engineering for chip-level soft error rate improvement. **Digest of Technical Papers**. VLSI Circuits. [S. l.], 2002, p. 204-205.

LISBOA, C. A. L. et al. Online hardening of programs against SEUs and SETs. In: IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS, 21, 2006, Washington DC, USA. **Proceedingsí** Los Alamitos, CA**:** IEEE Computer Society, 2006.

MENTOR, G. **Leonardo Express**: version 2.11.15.0. Mentor Graphics Inc. 1981. Disponível em: <http://www.mentor.com>. Acesso em: Nov. 2006.

MCFEARING, L.; NAIR, V.S.S. Control-Flow Checking Using Assertions. In: INTERNATIONAL WORKING CONFERENCE ON DEPENDABLE COMPUTING FOR CRITICAL APPLICATIONS, 5., 1995, [S. l.]. **Proceedingsí :** [S. l.: s. n.], Sept. 1995.

MITRA, S. et al. Robust system design with built-In soft-error resilience. **Computer Society**. [S. l.]: v. 38, i. 2, p. 43ó52, Feb. 2005.

NEUBERGER et al. Multiple bit upset tolerant SRAM memory. **ACM Transactions on Desing Automation Electronic Systems.** [S. l.]: v. 8, n. 4 p. 577-590, Oct. 2003.

NEUBERGER, G.; KASTENSMIDT, F. G. L.; REIS, R. An Automatic Technique for Optimizing Reed-Solomon Codes to Improve Fault Tolerance in Memories. **IEEE Desing & Test for Computers:** design for yield and reliability [S. l.: s. n.], p. 50-58, Jan/Feb. 2005.

NGUYEN, H. T.; YAGIL, Y. A systematic approach to SER estimation and solutions. In: IEEE INTERNATIONAL RELIABILITY PHISICS SYMPOSIUM, 41., 2003, Dallas, USA. **Proceedingsí** [S. l.]: IEEE Press, 2003. p. 60-70.

OH, N.; MITRA, S.; MACCLUSKEY, E.J. ED4I: error detection by diverse data and duplicated instructions. **IEEE Transactions on Computers.** [S. l.] v. 51, n. 2, p. 180-199, Feb. 2002a.

OH, N.; SHIRVANI, P.P.;MCCLUSKEY, E.J. Control flow checking by software signatures. **IEEE Transactions on Reliability**. [S. l.] v. 51, n. 2, p. 111-112, March 2002b.

OOTSUKA, F. et al. A novel 0.20   m full CMOS SRAM cell using stacked cross couple with enhanced soft error immunity. In: IEEE INTERNATIONAL DEVICES MEETING, IEDM, 1998, San Francisco, USA. **Proceedingsí** New York**:** IEEE, 1998, p. 205-208.

PATTERSON, D.A.; HENNESSY, J. L. **Computer Architecture:** a quantitative approach**.** 3 ed., Amsterdam: Elsevier Science & Technology Books, June 2002. ISBN: 1558605967.

RAMPRASAD, S.; SHANBHAG, N. R.; HAJJ, I. N. Analytical estimation of transition activity from word-level signal statistics. In: DESIGN AUTOMATION CONFERENCE, 34., 1997, Anaheim, USA. **Proceedingsí** New York: IEEE Computer Society, June 1997, p. 582-587.

REDINBO, G.; NAPOLITANO, L.; ANDALEON, D. Multibit correction data interface for fault-tolerant systems. **IEEE Transactions on Computers.** [S. l.]: v. 42, n. 4. p. 433-446, Apr. 1993.

REINHARDT, S. K.; MUKHERJEE, S. S. Transient fault detection via simultaneous multithreading. In: INTERNATIONAL COMPUTER ARCHITECTURE, ISCA, 27., 2000, Vancouver, Canada. **Proceedingsí** [S. l.: s. n.], June 2000. p. 25-36.

SEIFERT, N. et al. Historical trend in alpha-particle induced soft error rates of the Alpha microprocessor. In: IEEE INTERNATIONAL RELIABILITY PHYSICS SYMPOSIUM, IRPS, 2001, Orlando, USA. **Proceedingsí** [S. l.: s. n.]: Apr.May 2001, p. 259-265.

SEIFERT, N.; TAM, N. Timing vulnerability factors of sequentials. **IEEE Transactions on Device and Materials Reliability.** [S. 1.], v. 4, n. 3, p. 516-522, Sept. 2004.

SEXTON, F.W. et al. SEU simulation and testing of resistor-hardened D-latches in the SA3300 microprocessor. **IEEE Teansaction on Nuclear Science.** [S. l.], v. 38, n. 6, p. 1521-1528, Dec. 1991.

SHERLEKAR, D. Design considerations for regular fabrics. In: INTERNATIONAL SYMPOSIUM ON PHYSICS DESIGN, ISPD, 2004, Phoenix, USA. **Proceedings í** New York: ACM Press, Jan. 2004, p. 97-102.

SHIRVANI, P.; SAXENA, N.; MCCLUSKEY, E. Software implemented EDAC protection against SEUs. **IEEE Transactions on Reliability.** [S. l.], v. 49, n. 3, p. 273-284. Sept. 2000.

RHOD, E. L.; LISBOA, C. A. L. ; CARRO, L. . Using memory to cope with simultaneous transient faults. In: IEEE LATIN-AMERICAN TEST WORKSHOP, LATW, 7., 2006, Buenos Aires, Argentina. **Proceedingsí** Porto Alegre: Evangraf, 2006, v. 1, p. 151-156.

RHOD, E. L. ; LISBOA, C. A. L. ; CARRO, L. . A low-SER efficient processor architecture for future technologies. In: DESIGN, AUTOMATION AND TEST IN EUROPE

CONFERENCE, DATE, 2007, Nice, France. **Proceedingsí** Los Alamitos, CA: IEEE Computer Society, 2007, v. 1, p. 1448-1453.

VELAZCO, R. et al. Two CMOS memory cells suitable for the design of SEU-tolerant VLSI circuits. **IEEE Transaction on Nuclear Science**. [S. l.], v. 41, n. 6, p.2229ó2233, Dec. 1994.

WEAVER, C.; AUSTIN, T. A fault tolerant approach to microprocessor design. In: THE INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 2001. **Proceedingsí** [S. l.], Jul. 2001. p. 411-420.

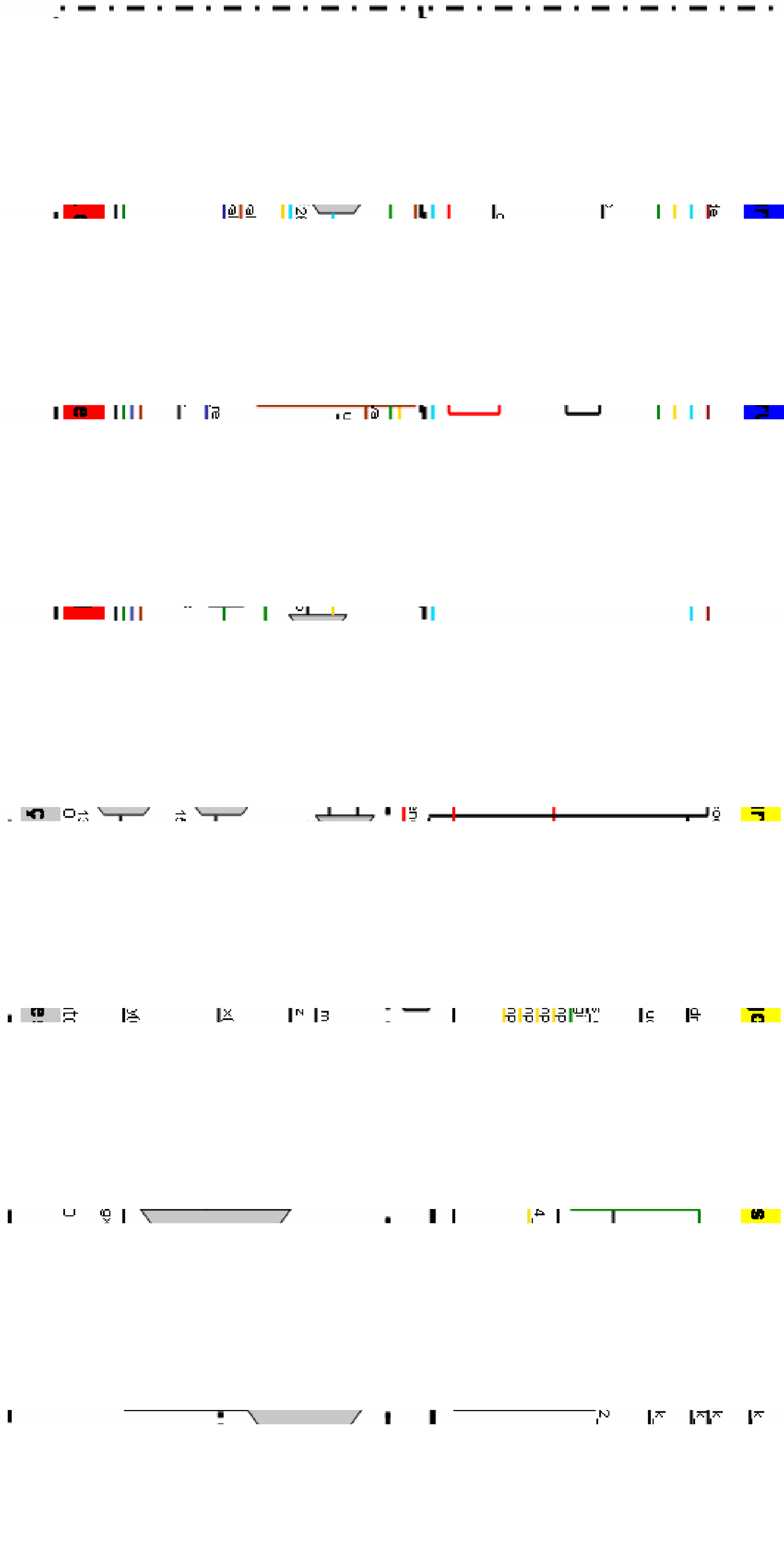
ZIEGLER, J. F.; LANFORD, W. A. The effect of sea level cosmic rays on electronic devices. **Journal of Applied Physics,** [S. 1.], p. 4305-4311, June 1981.

APENDIX A: MEMPROC LIST OF INSTRUCTIONS

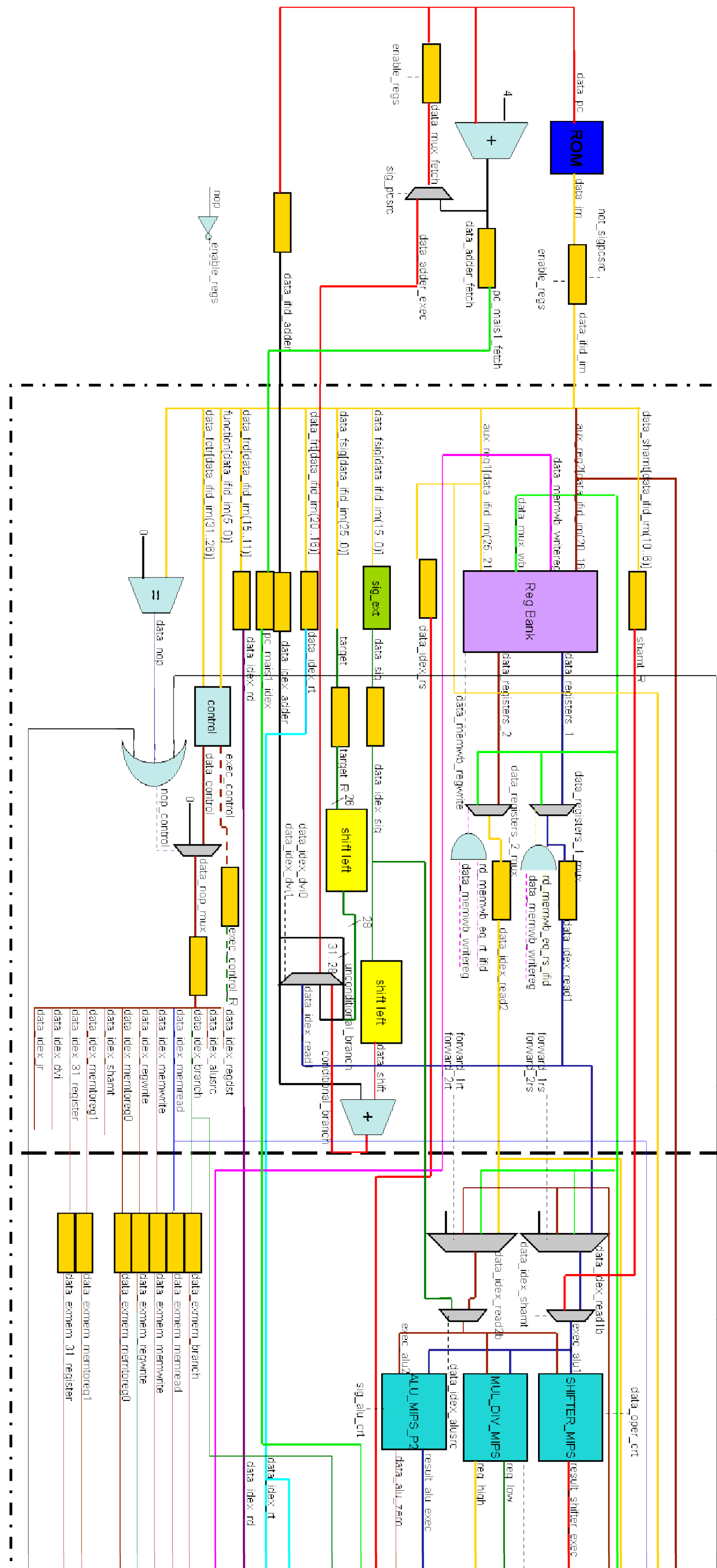
Instruction	Syntax	Description	Number of cycles	
			min	max
NOP	nop	-	1	1
MOV	mov k, m	copy constant k to mem. addr. ÷mø	2	2
	mov m2, m1	copy value in mem. addr. ÷m1øto mem. addr. ÷m2ø	3	3
	mov m2, *m1	copy value indicated by the pointer in ÷m1ø to mem. addr. ÷m2ø	3	3
	mov *m2, m1	copy value in mem. addr. ÷m1øto mem. addr. indicated by the pointer ÷m2ø	4	4
IF_ICMPEQ	if_icmpeq k, m, j	if constant ÷kø equal to value in mem. addr. ÷mø then jump to addr. ÷jø	4	4
	if_ icmpeq m1, m2, j	if value in mem. addr. ÷m1ø equal to value in mem. addr. ÷m2øthen jump to addr. ÷jø	4	4
IF_ICMPNE	if_icmpne k, m, j	if constant ÷kø not equal to value in mem. addr. ÷mø then jump to addr. ÷jø	4	4
	if_ icmpne m1, m2, j	if value in mem. addr. ÷m1ønot equal to value in mem. addr. ÷m2øthen jump to addr. ÷jø	4	4
IF_ICMPLT	if_icmplt k, m, j	if constant ÷kø less then value in mem. addr. ÷mø then jump to addr. ÷jø	4	9
	if_icmplt m, k, j	if value in mem. addr. ÷m1øless then constant ÷køthen jump to addr. ÷jø	4	9
	if_icmplt m1, m2, j	if value in mem. addr. ÷m1ø less then value in mem. addr. ÷m2øthen jump to addr. ÷jø	4	9
IF_ICMPLE	if_icmple k, m, j	if constant ÷kø less or equal then value in mem. addr. ÷møthen jump to addr. ÷jø	4	9
	if_icmple m, k, j	if value in mem. addr. ÷m1øless or equal then constant ÷køthen jump to addr. ÷jø	4	9
	if_icmple m1, m2, j	if value in mem. addr. ÷m1øless or equal then value in mem. addr. ÷m2øthen jump to addr. ÷jø	4	9
IF_ICMPGT	if_icmpgt k, m, j	if constant ÷kø greater then value in mem. addr. ÷mø then jump to addr. ÷jø	4	9
	if_icmpgt m, k, j	if value in mem. addr. ÷m1ø greater then constant ÷kø then jump to addr. ÷jø	4	9
	if_icmpgt m1, m2, j	if value in mem. addr. ÷m1øgreater then value in mem. addr. ÷m2øthen jump to addr. ÷jø	4	9
IF_ICMPGE	if_icmpge k, m, j	if constant ÷kø greater or equal then value in mem. addr. ÷møthen jump to addr. ÷jø	4	9
	if_icmpge m, k, j	if value in mem. addr. ÷m1ø greater or equal then constant ÷køthen jump to addr. ÷jø	4	9
	if_icmpge m1, m2, j	if value in mem. addr. ÷m1øgreater or equal then value in mem. addr. ÷m2øthen jump to addr. ÷jø	4	9
IFEQ	ifeq m, j	if value in mem. addr. ÷møis equal to zero then jump to addr. ÷jø	4	4
IFNE	ifne m, j	if value in mem. addr. ÷mø is not equal to zero then jump to addr. ÷jø	4	4
IFLT	iflt m, j	if value in mem. addr. ÷møis less then zero then jump to addr. ÷jø	4	4
IFLE	ifge m, j	if value in mem. addr. ÷møis less or equal to zero then jump to addr. ÷jø	4	4
IFGT	ifgt m, j	if value in mem. addr. ÷mø is greater then zero then jump to addr. ÷jø	4	4
IFGT	ifge m, j	if value in mem. addr. ÷møis greater or equal then zero then jump to addr. ÷jø	4	4

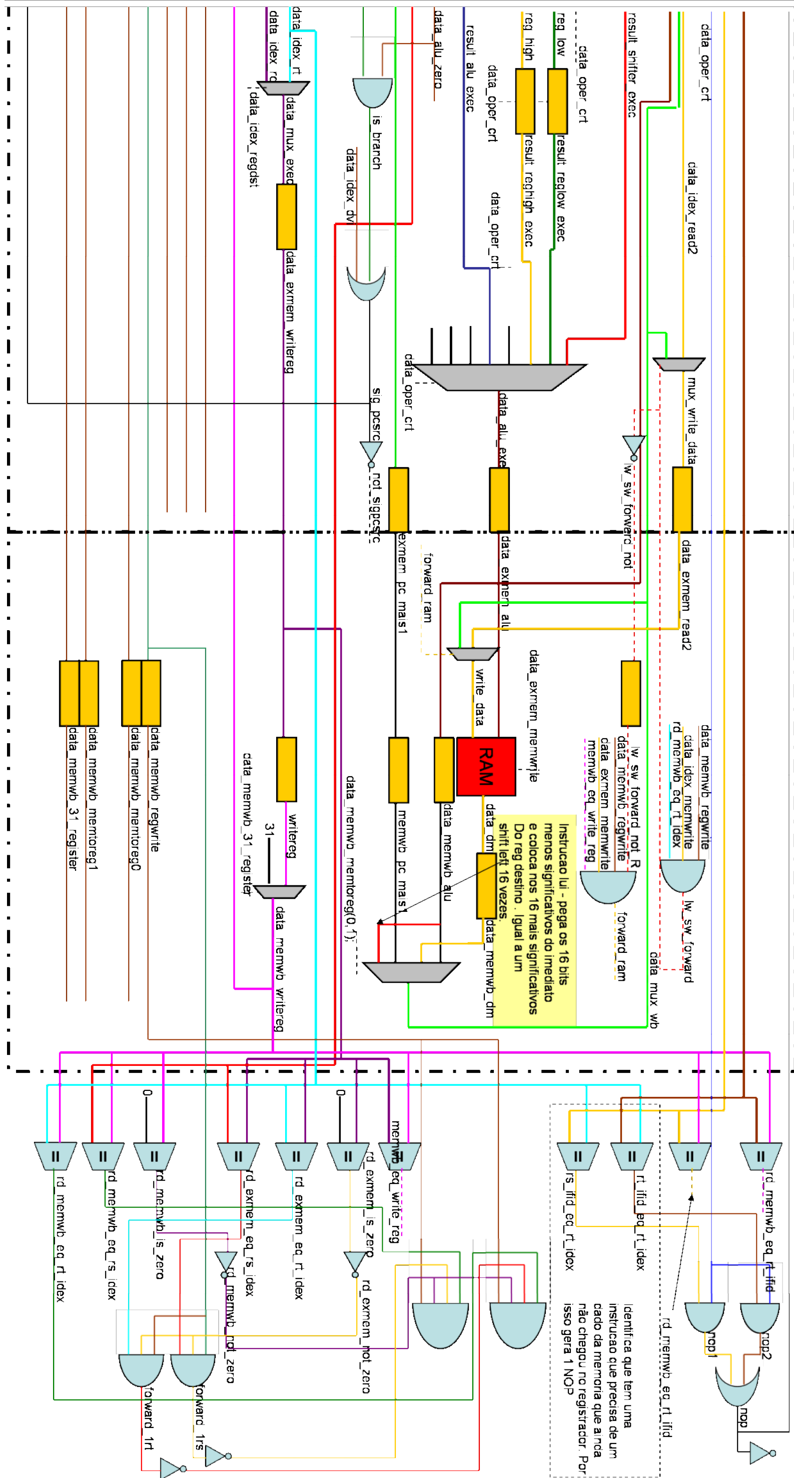
Instruction	Syntax	Description	Number of cycles	
			min	max
ADD	add d, m, k	adds value in mem. addr. ÷mø to constant ÷kø and stores in mem. addr. ÷dø	3	18
	add d, m1, m2	adds value in mem. addr. ÷m1ø to value in mem. addr. ÷m2ø and stores in mem. addr. ÷dø	3	18
SUB	sub d, m, k	subtracts value in mem. addr. ÷mø from constant ÷kø and stores in mem. addr. ÷dø	18	18
	sub d, k, m	subtracts constant ÷kø from value in mem. addr. ÷mø and stores in mem. addr. ÷dø	18	18
	sub d, m1, m2	subtracts value in mem. addr. ÷m1ø from value in mem. addr. ÷m2ø and stores in mem. addr. ÷dø	18	18
ADDC	addc d, m, k	adds with carry value in mem. addr. ÷mø to constant ÷kø and stores in mem. addr. ÷dø	3	18
	addc d, m1, m2	adds with carry value in mem. addr. ÷m1ø to value in mem. addr. ÷m2ø and stores in mem. addr. ÷dø	3	18
MUL	mul d, m, k	multiply value in mem. addr. ÷mø by constant ÷kø and stores in mem. addr. ÷dø	35	49
	mul d, m1, m2	multiply value in mem. addr. ÷m1ø by value in mem. addr. ÷m2ø and stores in mem. addr. ÷dø	35	49
IUSHR	iushr d, m, k	unsigned shift right the value in mem. addr. ÷mø ÷kø times	3	18
	iushr d, m1, m2	unsigned shifts right the value in mem. addr. ÷mø the value in mem. addr. ÷m2ø times	3	18
ISHL	ishl d, m, k	shifts left the value in mem. addr. ÷mø ÷kø times	3	18
	ishl d, m1, m2	shifts left the value in mem. addr. ÷m1ø the value in mem. addr. ÷m2ø times	3	18
NEG	neg d, m	negates value in mem. addr. ÷mø and stores in mem. addr. ÷dø	3	18
AND	and d, m, k	make õlogic andö with the value in mem. addr. ÷mø with the constant ÷kø and stores in mem. addr. ÷dø	3	3
	and d, m1, m2	make õlogic andö with the value in mem. addr. ÷m1ø and ÷m2ø and stores in mem. addr. ÷dø	3	3
OR	or d, m, k	make õlogic orö with the value in mem. addr. ÷mø with the constant ÷kø and stores in mem. addr. ÷dø	3	3
	or d, m1, m2	make õlogic orö with the value in mem. addr. ÷m1ø and ÷m2ø and stores in mem. addr. ÷dø	3	3
JMP	jmp d	jumps to the destination in mem. addr. ÷mø	3	3
CALL	call f, r	jumps to subroutine in the address ÷fø and stores the return addres in mem. addr. ÷rø	2	2
RET	ret r	returns to the address in mem. addr. ÷rø	3	3

APENDIX B: MEMPROC ARCHITECTURE DESCRIBED IN CACO-PS TOOL



APENDIX C: MIPS ARCHITECTURE DESCRIBED IN CACO-PS TOOL




Instrucao lui - pega os 16 bits menos significativos do imediato e coloca nos 16 mais significativos Do reg destino. Igual a um shift left 16 vezes.
Identifica que tem uma instrucao que precisa de um dado da memoria que ainda não chegou no registrador. Por isso gera 1 NOP
RAM
data_oper_crt
data_idex_read2
mux_write_data
data_exmem_read2
result_shifter_exec
reg_low
reg_high
result_reglow_exec
result_reghigh_exec
result_alu_exec
data_alu_zero
data_alu_exec
data_exmem_alu
data_exmem_memwrite
write_data
forward_ram
is_branch
sig_pcsrc
data_mux_exec
data_exmem_writereg
data_idex_regdst
data_memwb_regwrite
data_memwb_memtoreg1
data_memwb_memtoreg0
data_memwb_31_register
data_memwb_writereg
data_memwb_alu
data_memwb_dm
data_mux_wb
memwb_pc_mas1
exmem_pc_mas1
lw_sw_forward
lw_sw_forward_not
nop
nop1
nop2
forward_1rt
forward_1rs

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo