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

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), São José dos Campos - 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.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

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 today's 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.

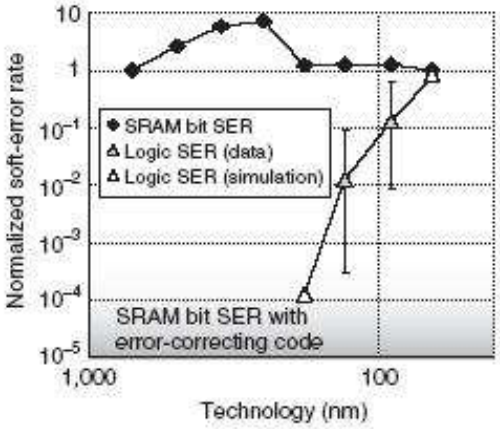


Figure 1.1: Evolution of SER: SRAM vs. Logic, from (BAUMANN, 2005)

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 (PATTERSON, 2002) 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.

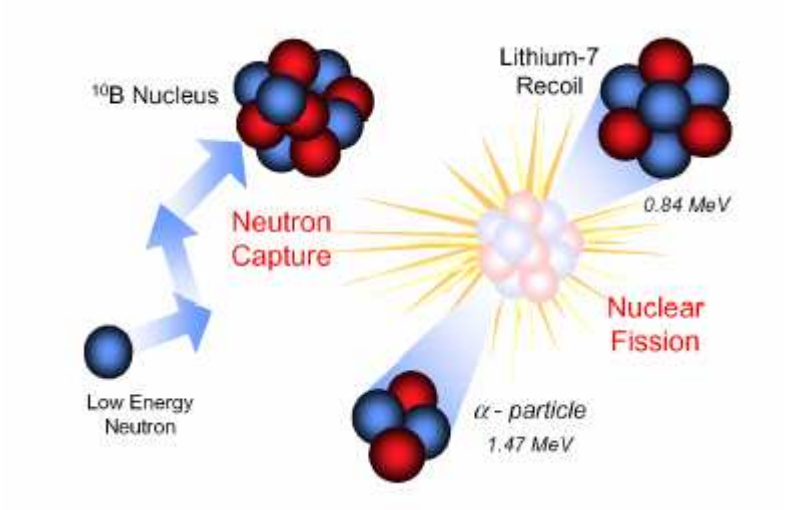


Figure 2.1: Boron fission induced by low energy neutron, from (BAUMANN, 2001).

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.

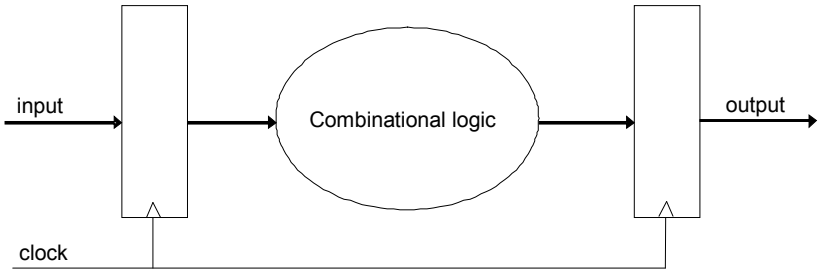


Figure 2.2: Sequential circuit.

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.

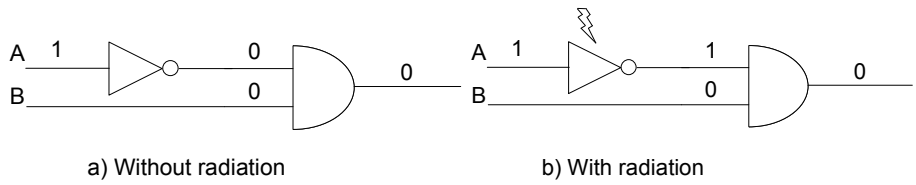


Figure 2.3: Combinational circuit without radiation (a) and with radiation (b).

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 – 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 i^{th} element of the system.

The $SER^{nominal}$ for the i^{th} 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 i^{th} element. The AVF_i, architectural vulnerability factor (also known as logic derating) represents the probability that and error in the i^{th} 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 hit, making this shielding technique obsolete for spatial 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 μm. 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).

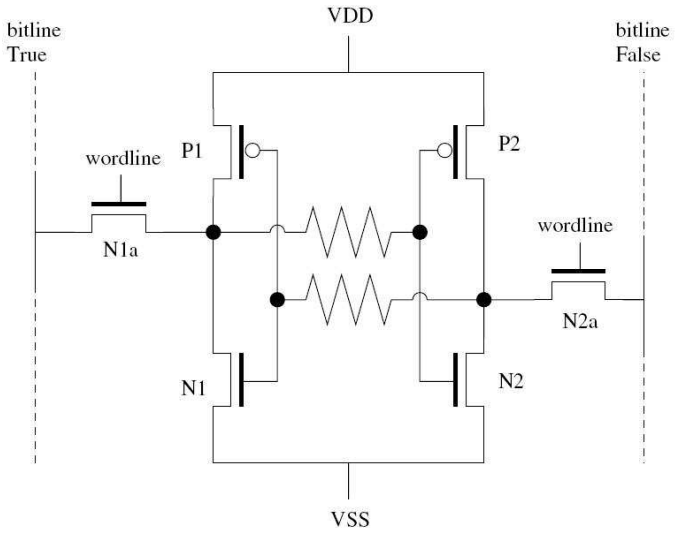


Figure 2.4: SRAM cell hardened by the inclusion of two feedback resistors

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.

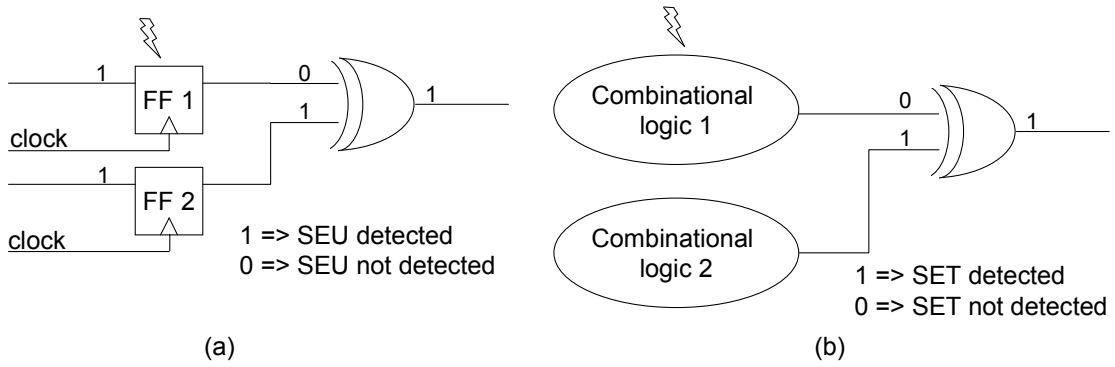


Figure 2.5: Detection of an SEU in a memory element (a) and detection of an SET in a combinational circuit (b) by using space (or hardware) redundancy.

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.

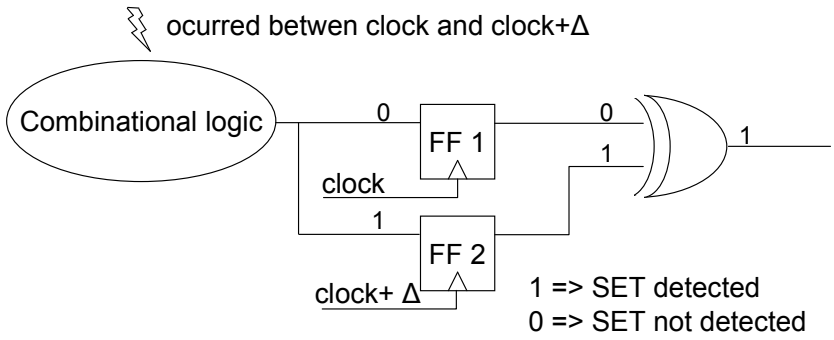


Figure 2.6: Use of time redundancy to detect an SET in a combinational circuit.

The circuit designer must set the "Δ" 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.

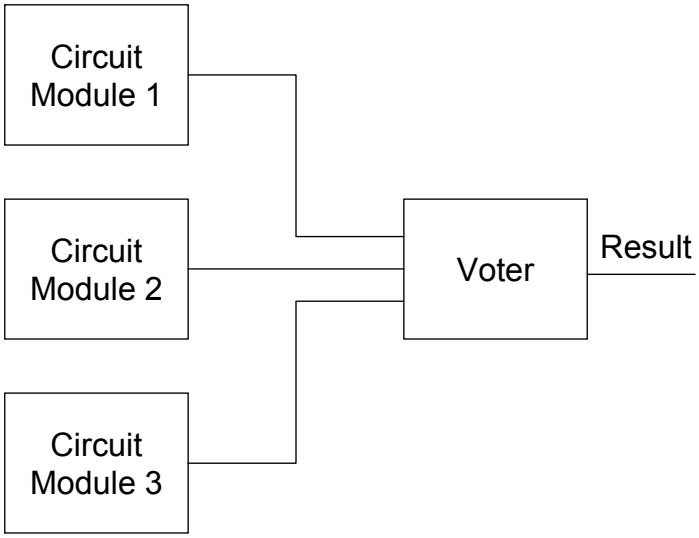


Figure 2.7: Use of space redundancy to detect an SET in a combinational circuit.

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 "Δ" delays adds some extra complexity to the circuit design.

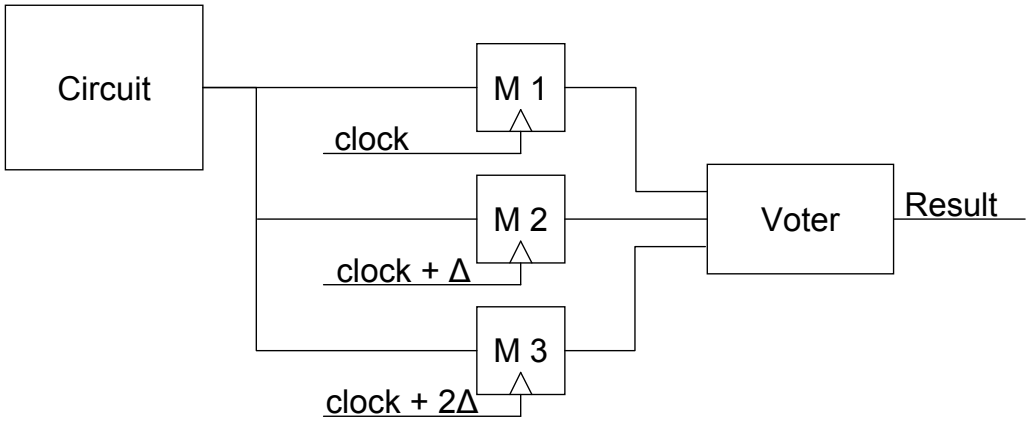


Figure 2.8: TMR with time redundancy.

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 program's 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

"*Dynamic verification*", a hardware-based technique, is detailed in (AUSTIN, 2000) for a pipelined core processor. It uses a "functional checker" 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 processor's 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 processor's 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 processor's 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:

$$\text{Size} = I^2 \times O \tag{3}$$

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.

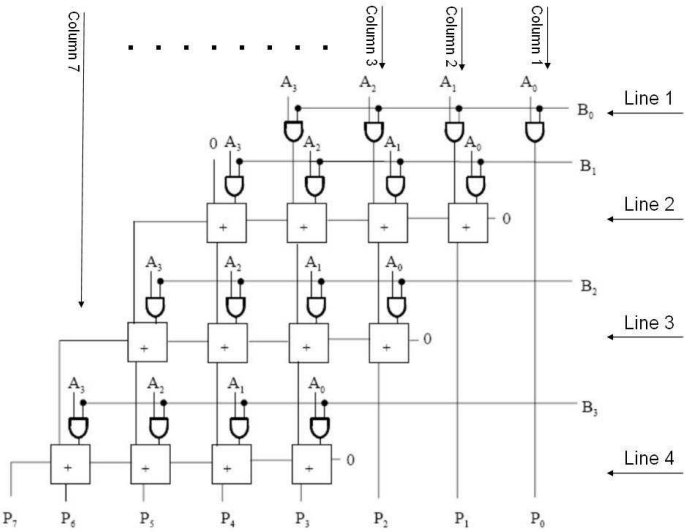


Figure 3.2: Fully combinational 4x4-bit multiplier.

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.

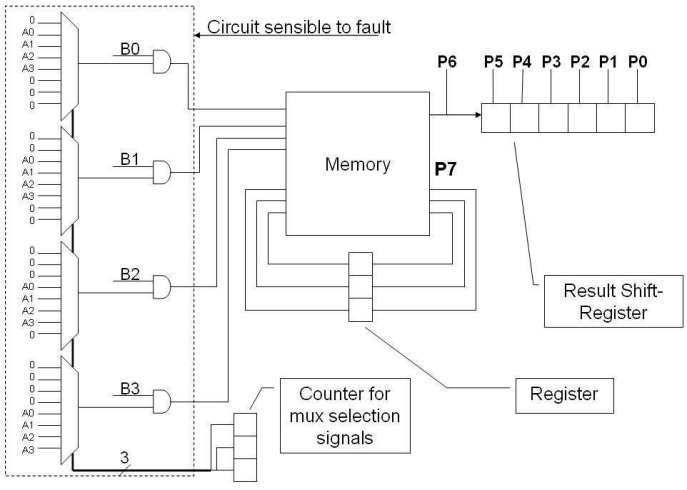


Figure 3.3: Column multiplier circuit.

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.

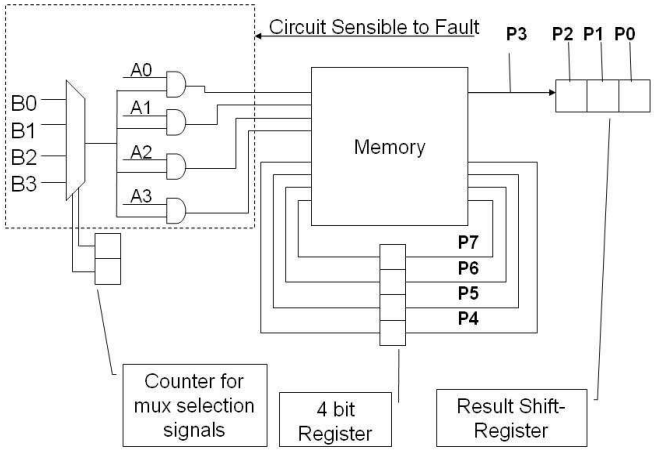


Figure 3.4: Line multiplier circuit.

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 μm 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.

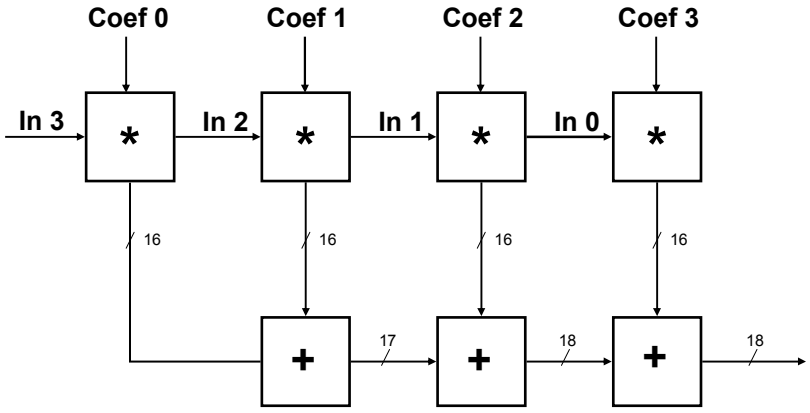


Figure 3.5: Combinational circuit for the 8-bit FIR filter with 4 taps.

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

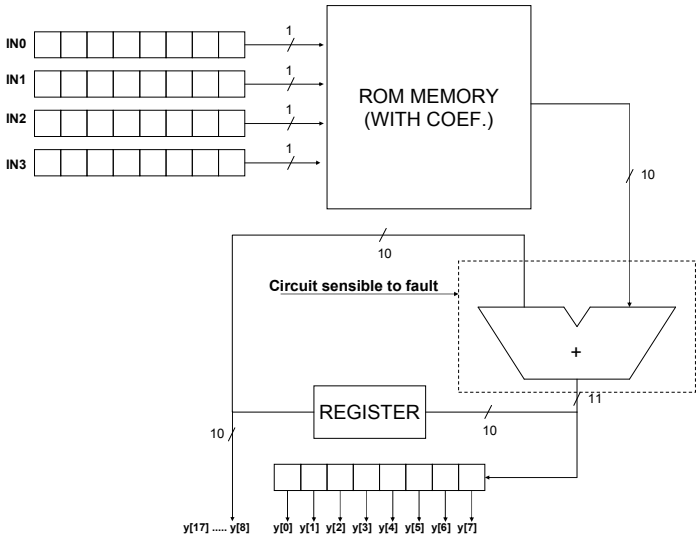


Figure 3.6: 8-bit FIR filter with 4 taps, using memory.

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

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

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.

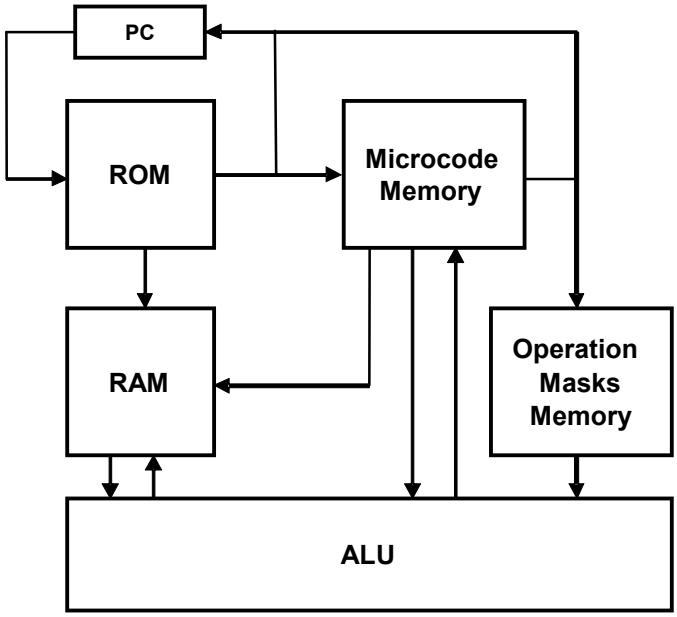


Figure 4.1: MemProc overall architecture.

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.

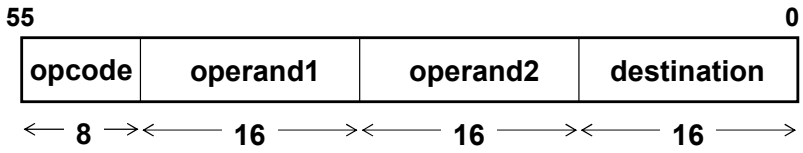


Figure 4.2: Macroinstruction format.

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.

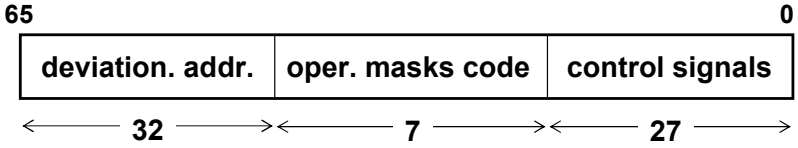


Figure 4.3: Microinstruction format.

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 “operation masks memory” 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.

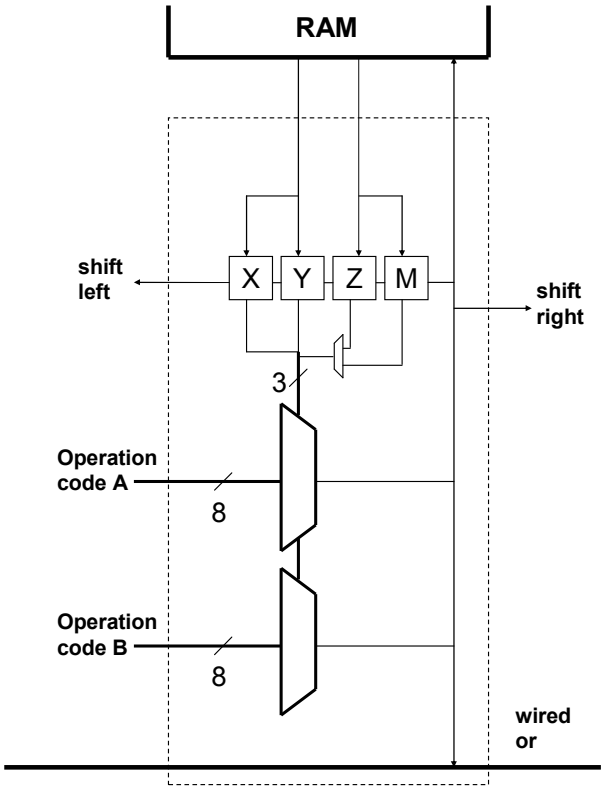


Figure 4.4: ALU for one bit operation.

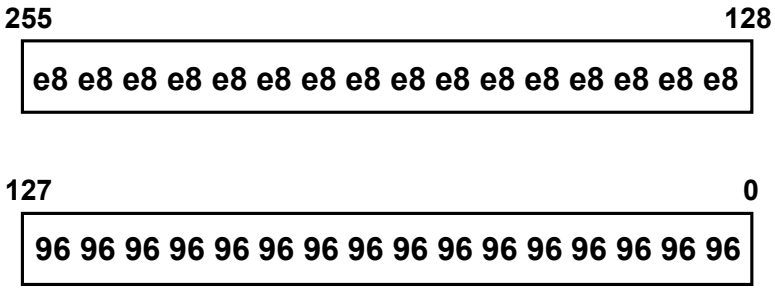


Figure 4.5: Operation masks used during the addition operation.

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 "sum" and the "cout" (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 "or" operation of all the multiplexors' 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.

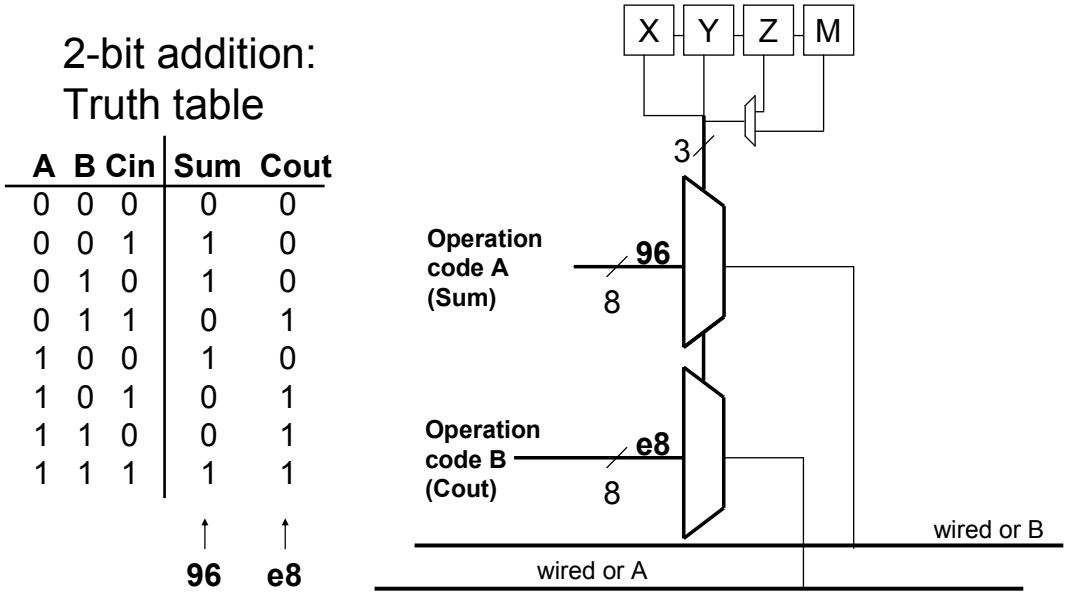


Figure 4.6: 2-bit addition using MemProc ALU.

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 “M” 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.

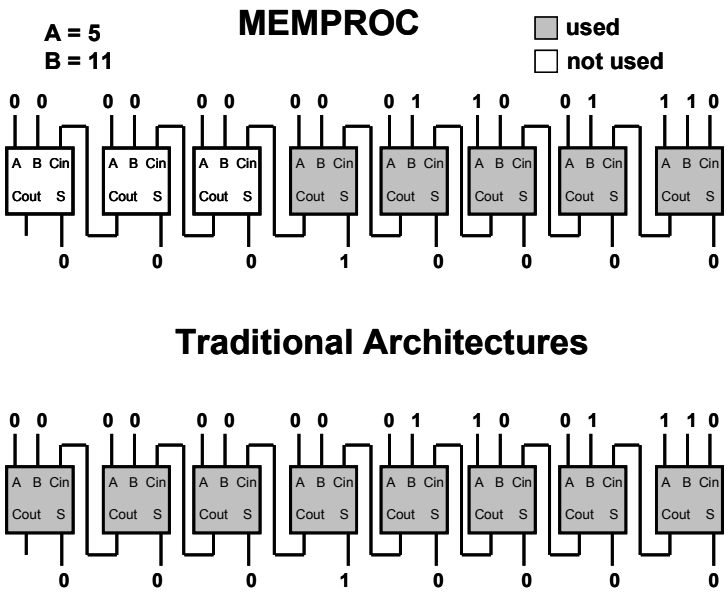


Figure 4.7: 8-bit addition paradigm.

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 ‘1s’ 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 “for loop” 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 MemProc’s 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.

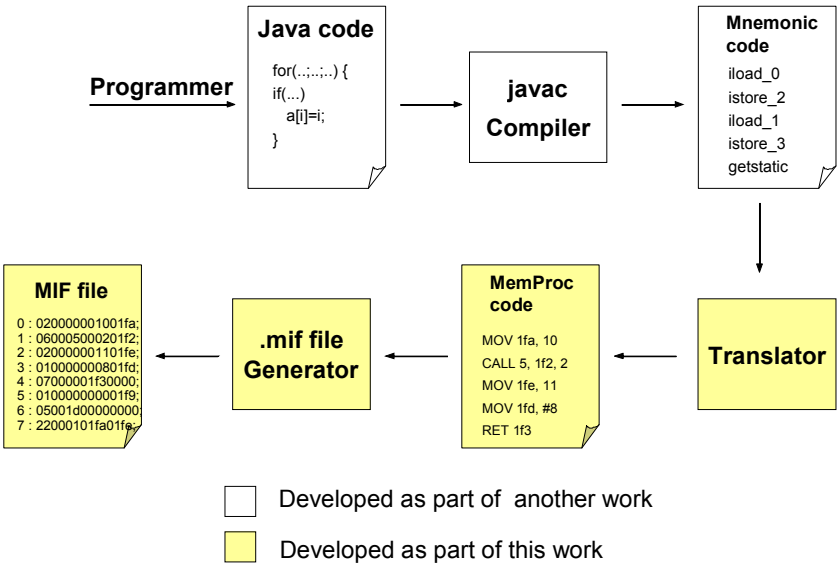


Figure 4.8: Code generation process for MemProc.

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.

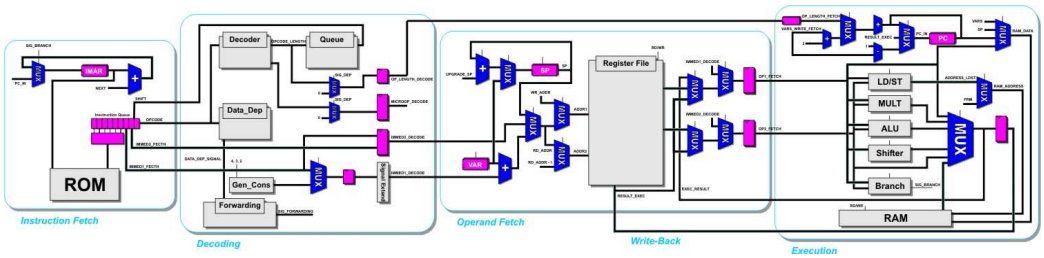


Figure 5.1: FemtoJava pipeline block scheme.

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.

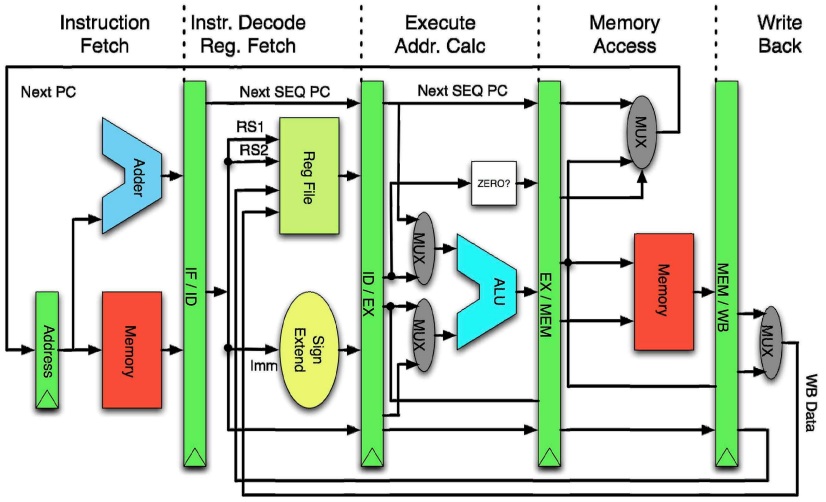


Figure 5.2: The pipelined MIPS architecture.

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 "conversion program" created the faulty architecture correctly, a simple test was done. The "faulty" 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 "conversion program" 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 μm cell library in the Leonardo Spectrum tool (MENTOR, 1981). This tool was also used to evaluate the area consumption in terms of "equivalent gates" 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 "equivalent gate" like the ones from the Leonardo tool. The 2:1 multiplexors were considered to be equal to one "equivalent gate" and the other multiplexors were constructed with 2:1 multiplexors, to be calculated as one "equivalent gate" 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.

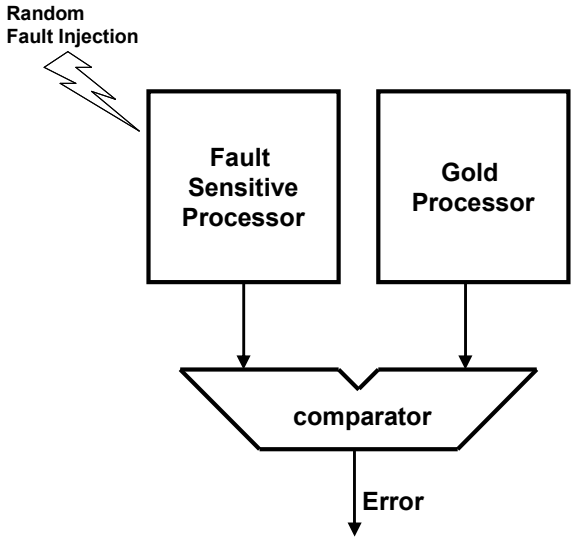


Figure 5.3: Error detection scheme.

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 “bubble sort” 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 FemtoJava's 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, MemProc's 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 MemProc's 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.

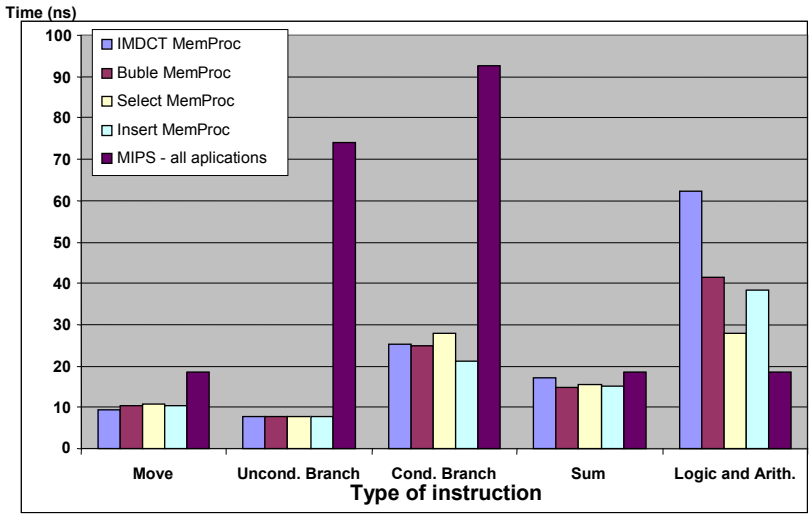


Figure 5.4: Mean time to execute each type of instruction for all applications

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 MemProc's 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 '1' 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.

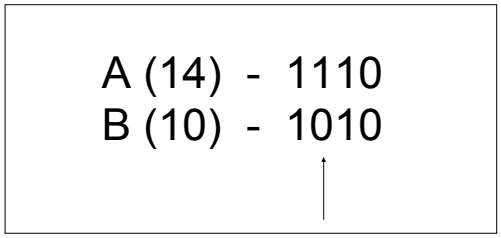


Figure 5.5: The way MemProc does comparisons.

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 ‘1’ bit in a position that the other value does not have is the ‘A’ value, therefore the ‘A’ value is greater than the ‘B’ 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 ‘1’ bit in a position that the other value does not have is the lowest one, due to the 2’s 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 application's 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.

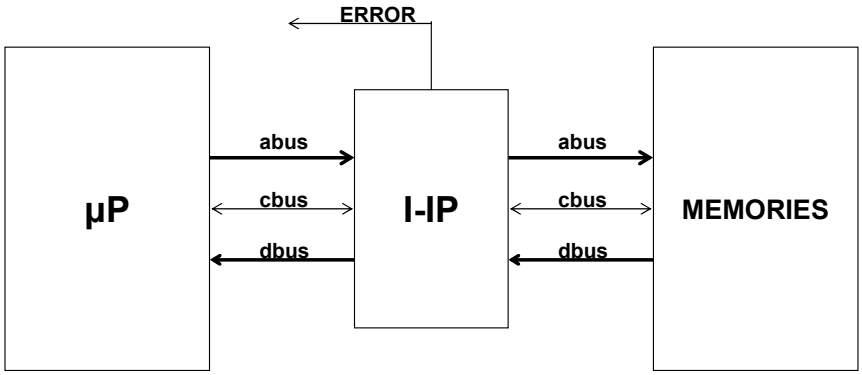


Figure 6.1: I-IP overall architecture.

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 "Fetch logic", and handles the details of the communication with the memory.
3) Fetch logic: issues to the "Memory interface" the commands needed for loading a new instruction in the I-IP and feeding it to the "Decode logic".
4) Decode logic: decodes the fetched instruction, whose address in memory is A, and sends the details about the instruction to the "Control unit". 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 "other", as described in item "c". 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 "CPU interface", it activates the "Fetch logic". Then, depending on the information produced by the "Decode logic", 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 processor's 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.

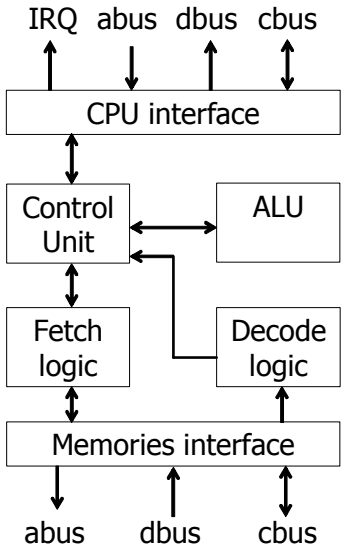


Figure 6.4: Architecture of the I-IP.

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 "CPU interface", the "Memory interface", the "Fetch logic", and the "Control unit" 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 "Decode logic" 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.

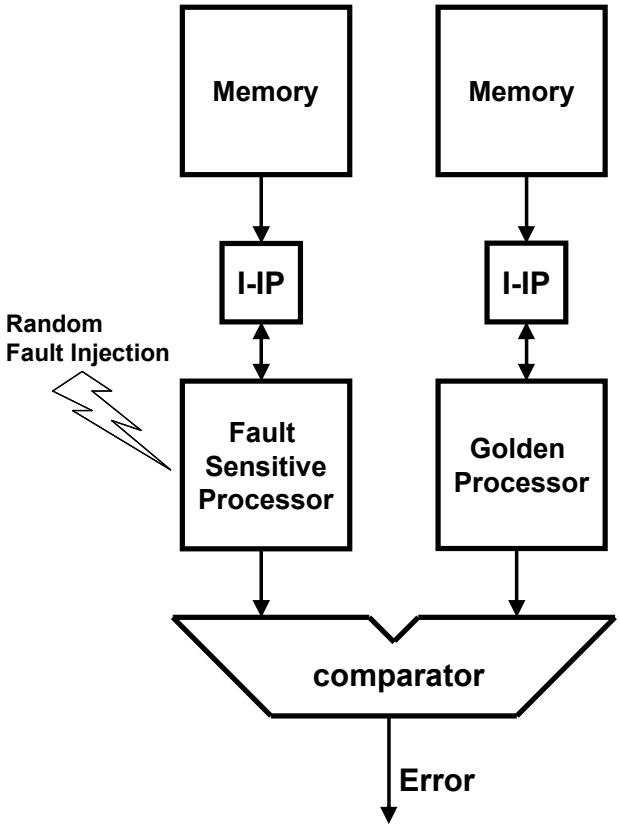


Figure 6.5: Error detection scheme.

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 program's 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. **Proceedings…** 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. **Proceedings…** [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. **Proceedings…** 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. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, p. 297–302, 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. **Proceedings…** Los Alamitos, CA**:** IEEE Computer Society, p. 349, Sept. 2003a.

BECK Fº, 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. **Proceedings…** [S. l.: s. n.], 2003b. p. 239-244.

BECK, Fº. 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, **Proceedings…** 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. **Proceedings…** 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. **Proceedings…** 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. **Proceedings…** [S. l.: s. n.], p. 367–370, 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. **Proceedings…** 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. 73–74.

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 …** 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 0.18µ. **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. **Proceedings…** 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.]. **Proceedings…:** [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. 43–52, 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. **Proceedings…** [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 μm full CMOS SRAM cell using stacked cross couple with enhanced soft error immunity. In: IEEE INTERNATIONAL DEVICES MEETING, IEDM, 1998, San Francisco, USA. **Proceedings…** 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. **Proceedings…** 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. **Proceedings…** [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. **Proceedings…** [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 …** 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. **Proceedings…** 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. **Proceedings…** 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.2229–2233, Dec. 1994.

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

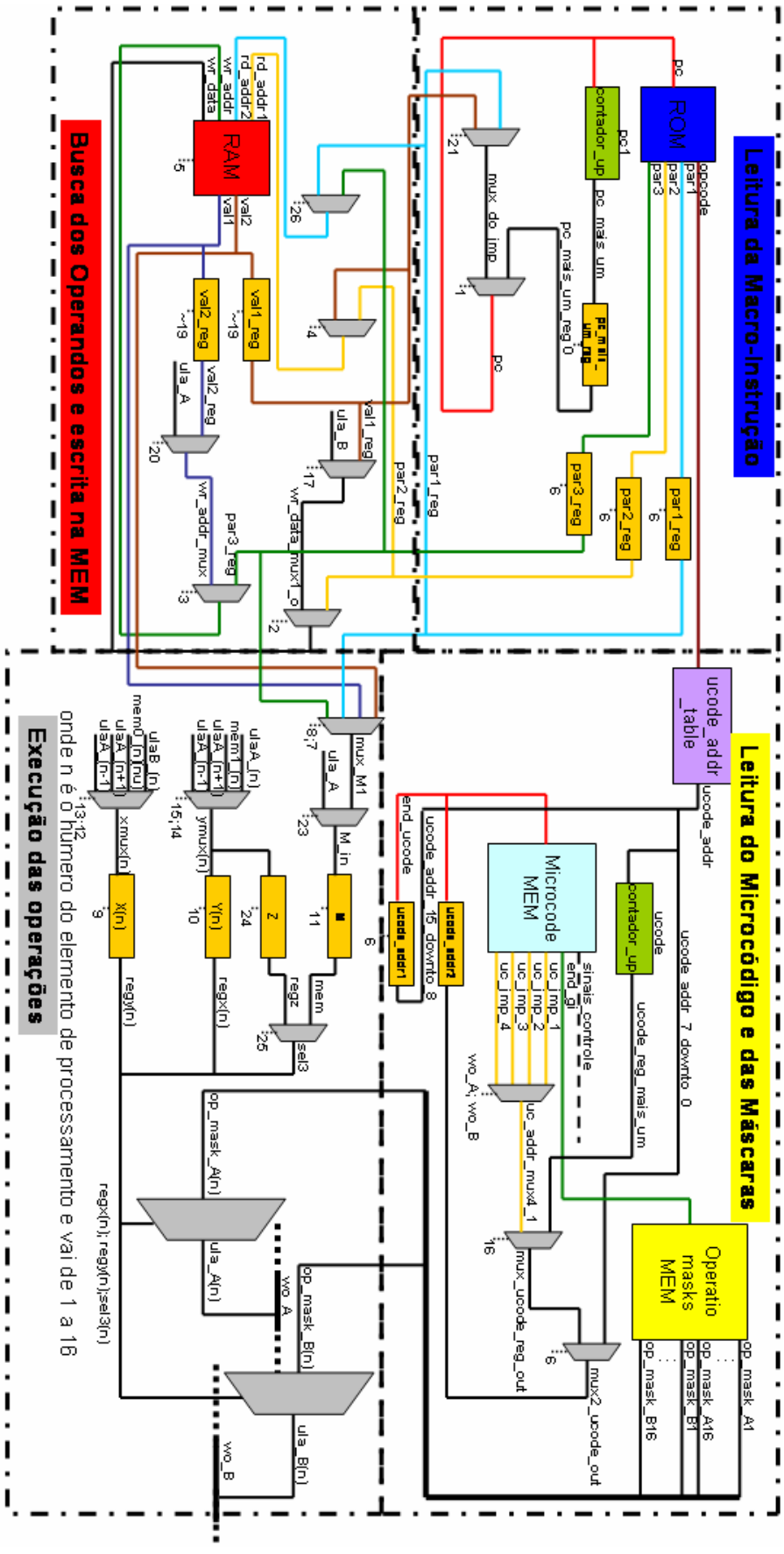
ZIEGLER, J. F.; LANFORD, W. A. The effect of sea level cosmic rays on electronic devices. **Journal of Applied Physics,** [S. 1.], p. 4305-4311, June 1981.

APENDIX A: MEMPROC LIST OF INSTRUCTIONS

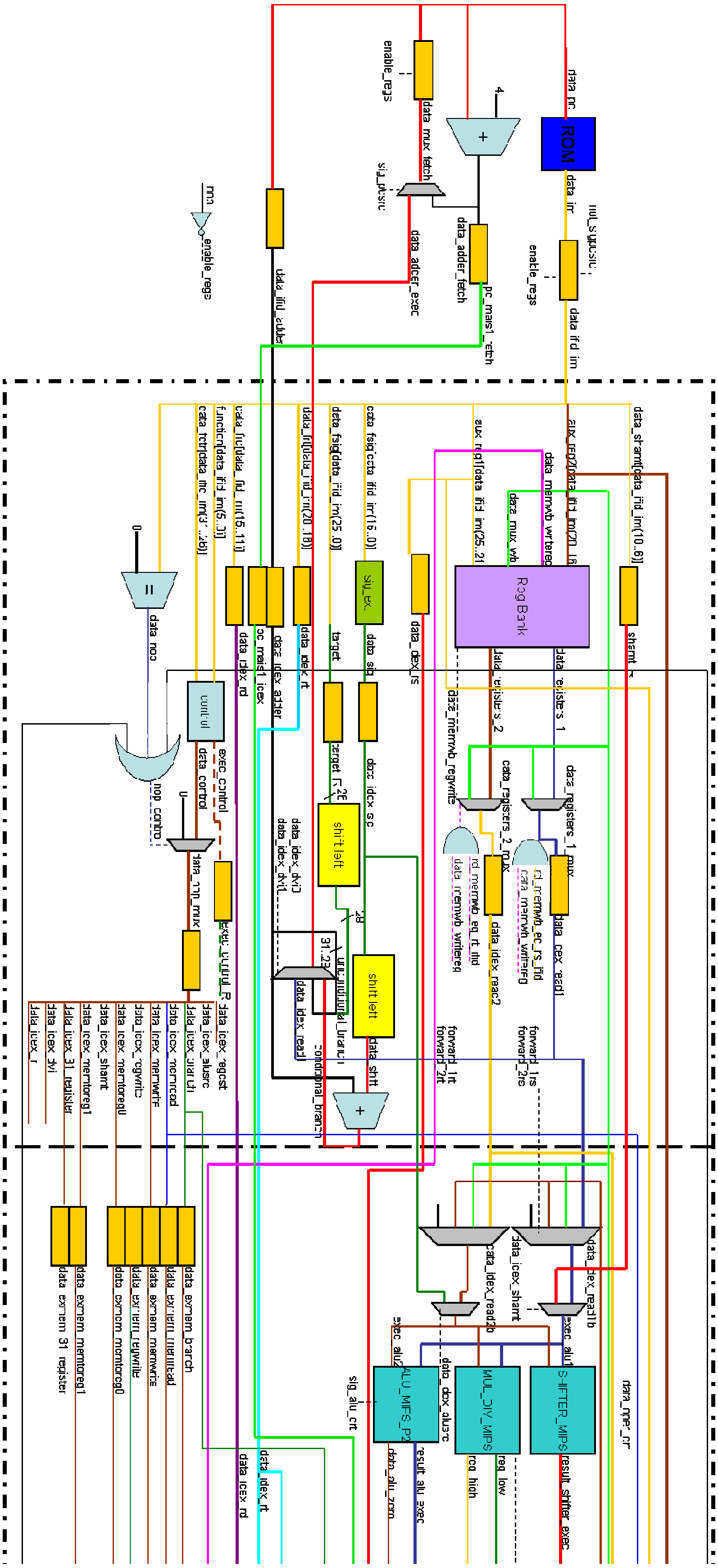
Instruction	Syntax	Description	Number of cycles	
			min	max
NOP	nop	-	1	1
MOV	mov k, m	copy constant k to mem. addr. ‘m’	2	2
	mov m2, m1	copy value in mem. addr. ‘m1’ to mem. addr. ‘m2’	3	3
	mov m2, *m1	copy value indicated by the pointer in ‘m1’ to mem. addr. ‘m2’	3	3
	mov *m2, m1	copy value in mem.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

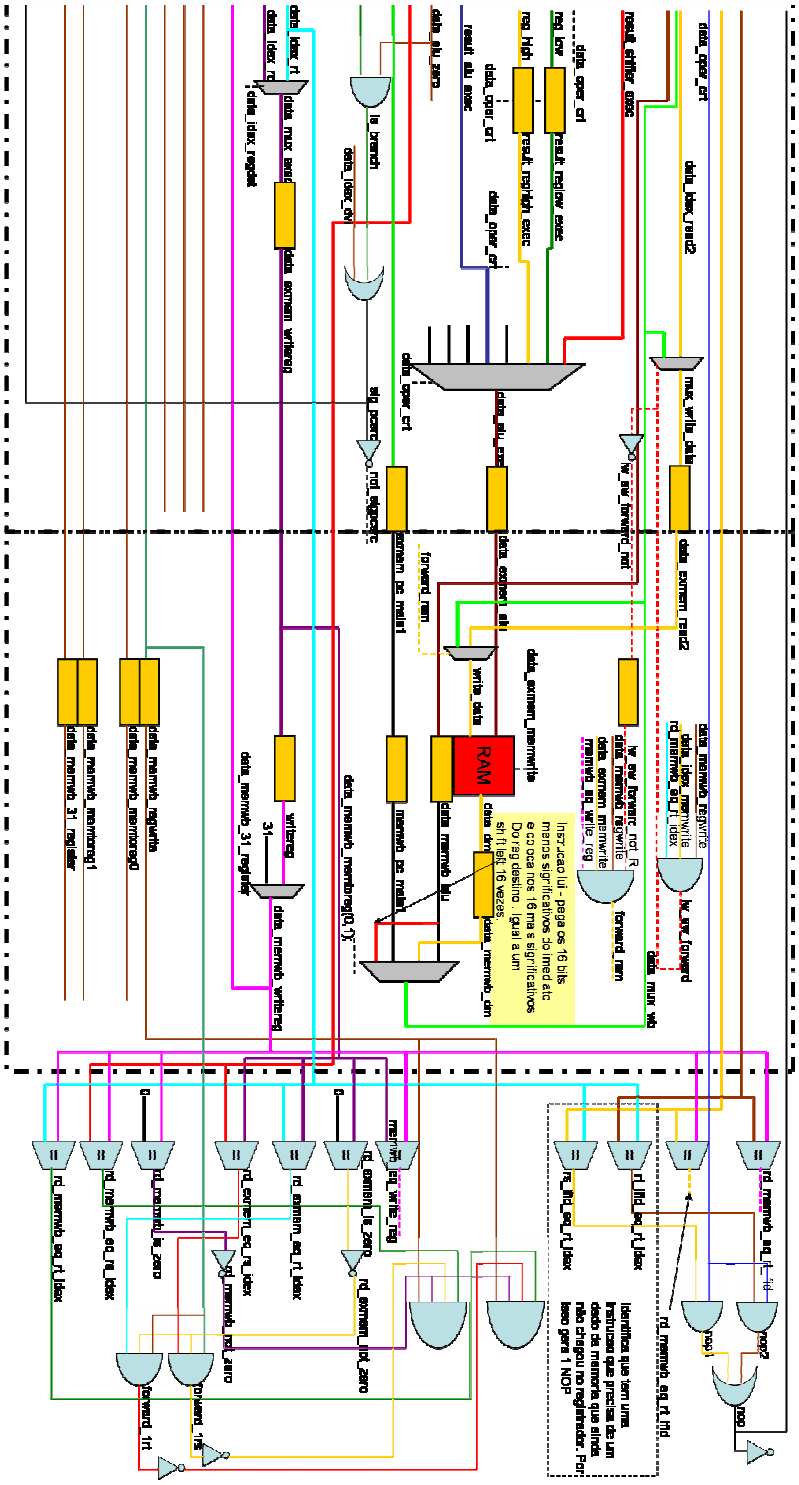
Instruction	Syntax	Description	Number of cycles	
			min	max
ADD	add d, m, k	adds value in mem. addr. ‘m’ to constant ‘k’ and stores in mem. addr. ‘d’	3	18
	add d, m1, m2	adds value in mem. addr. ‘m1’ to value in mem. addr. ‘m2’and stores in mem. addr. ‘d’	3	18
SUB	sub d, m, k	subtracts value in mem.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

APENDIX B: MEMPROC ARCHITECTURE DESCRIBED IN CACO-PS TOOL



APENDIX C: MIPS ARCHITECTURE DESCRIBED IN CACO-PS TOOL




RAM
Instrucao lui - pega os 16 bits
menos significativos do imed etc
e coloca nos 16 mais significativos
Dai 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 reginador. Por
isso gera 1 NOP

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

EDUARDO LUIS RHOD

SÍNTESE

Porto Alegre
2007

1 INTRODUÇÃO

O constante crescimento da indústria de semicondutores nos últimos anos tem contribuído para o desenvolvimento de circuitos com transistores cada vez menores e mais rápidos. Está surgindo uma nova era da tecnologia de circuitos chamada de nanotecnologia, permitindo a integração de bilhões de transistores no mesmo chip. Neste novo cenário, os projetistas de circuitos integrados estão desenvolvendo sistemas com mais de um elemento de processamento, criando o que chamamos de sistemas em chip (system-on-chip (SoC)) encontrados nos sistemas chamados embarcados como por exemplo celulares, sistemas de posicionamento global (GPS), sistemas de frenagem ABS, etc.

Entretanto, esta evolução na tecnologia está aumentando a preocupação dos projetistas com a confiabilidade desses novos circuitos. Apesar de apresentarem melhor desempenho, estas tecnologias de última geração são mais sensíveis à variação dos processos de fabricação por possuírem dimensões muito reduzidas. Além disso, o choque de partículas de alta energia, como nêutrons provenientes do espaço, até certo tempo consideramos preocupantes apenas para aplicações espaciais, estão agora se tornando fontes de radiações significativas e estão afetando não só componentes de memórias mas também componentes de lógica combinacional instalados ou funcionando no nível do mar. O choque destas partículas pode produzir ou estimular a mudança no valor lógico dos circuitos de momória, também conhecida como SEUs ou *single event upsets*, ou gerar pulsos transientes conhecidos como SETs ou *single envent transients,* os quais em certas circunstâncias podem ocasionar erros no funcionamento do circuito, ocasionando os assim chamados SE ou *soft errors*.

Outro motivo de preocupação dos pesquisadores reside no fato de que não só o número de transistores, mas também a densidade dos transistores no chip têm crescido

exponencialmente nos últimos anos. Estes crescimentos favorecem a ocorrência de múltiplas falhas decorridas de apenas uma partícula se chocar com o circuito. A este fenômeno damos o nome de MBUs ou *multiple bit upsets,* presente até pouco tempo apenas em circuitos de memória, porém agora pode se manifestar em circuitos de lógica combinacional.

Outro problema decorre do fato que com transistores mais rápidos, teremos períodos de relógio mais curtos, e conseqüentemente, o tempo de duração das falhas poderá durar mais de um ciclo de relógio, além de facilitar a propagação de pulsos transientes para serem capturados por elementos de memória gerando os chamados "bit-flip" ou "multiple-bit flips".

Para sobrevivermos neste novo cenário, está claro que novas técnicas de tolerância a falhas precisam ser definidas, não apenas para a segurança de sistemas críticos, mas também para qualquer sistema computacional.

As técnicas de tolerância de tolerância a falhas que conhecemos são eficientes, com um certo custo adicional, para mitigar SEUs e SETs porém, não são capazes de suportar múltiplas falhas que serão previstas nas tecnologias futuras (CONSTANTINESCU, 2003; EDENFELD, 2004). Para enfrentar este desafio, novos materiais e tecnologias de produção precisam ser desenvolvidas ou também novas técnicas de projeto de circuitos que sejam tolerantes a falhas precisam ser propostas.

Diversas técnicas visando mitigar os SEUs e SETs foram propostas nos últimos anos. Existem técnicas que atuam em todos os estágios de fabricação de um circuito, deste as etapas relacionadas ao processo de fabricação até técnicas de projeto que modificam o software e/ou o hardware para sistemas dedicados ou sistemas de propósito geral. A maioria destas técnicas são capazes de reduzir significamente o número de falhas, incluído para isso algum custo adicional em desempenho e/ou área e/ou potência. As soluções relacionadas ao processo de fabricação são geralmente muito caras para pequenos volumes de produção. Geralmente, técnicas de hardware tendem a acrescentar custo considerável em área de circuito, enquanto

que técnicas de software afetam de alguma maneira o desempenho final do circuito. Portanto a procura por uma solução que forneça confiabilidade aos circuitos na presença de falhas simples e/ou múltiplas continua sendo um tópico de muito interesse que ainda requer soluções eficientes.

A regularidade geométrica nos circuitos e o uso extensivo de fabricação de circuitos regulares estão sendo considerados como uma possível solução para lidar com as variações tecnológicas e aumentar a taxa de produtividade na fabricação dos circuitos nas tecnologias futuras. A regularidade traz a redução no custo das máscaras e permite também a introdução de linha e colunas sobressalentes que podem ser ativados para substituir linhas e colunas de memória defeituosas (SHERLEKAR, 2004). Juntamente com a proposta de usar a fabricação regular, a utilização de novas tecnologias de memória que podem suportar os efeitos de falhas transientes, tais como as memórias RAM ferroelétricas e magnéticas (ETO, 1998), trás de volta o conceito de computar utilizando memória.

Neste trabalho, o uso de memória é proposto como uma nova técnica de mitigação para falhas transientes, através da redução da área do circuito que pode ser afetada por *soft errors*. Desta forma, este trabalho introduz uma arquitetura de processadorque pode ser usada para aumentar a taxa de produtividade nos processos de manufatura do futuro. A arquitetura proposta é um núcleo de processador embarcado baseado em memória, aqui chamado de MemProc, projetado para o uso em aplicações de controle como um microcontrolador embarcado.

Existem casos em que nem o hardware nem o software podem ser modificados, devido aos altos custos envolvidos em adicionar hardware extra ou até mesmo quando não possuímos o código para modificá-lo. Nestes casos, técnicas alternativas são necessárias para prover ao sistema um nível adequado de confiança. Para lidar com este tipo de aplicações, este trabalho propõe uma segunda alternativa para aumentar a confiabilidade nos sistemas digitais, que

combina modificação do software a ser executado em tempo de execução com um módulo de hardware de propósito específico conhecido como *infrastructure IP* ou I-IP, proposto previamente em (BERNARDI, 2006). O desenvolvimento de um núcleo de I-IP para melhorar a confiabilidade do processador de arquitetura RISC conhecido como MIPS (PATTERSON, 2002) é apresentado neste trabalho.

2 REVISÃO BIBLIOGRÁFICA

Nos primeiros anos de exploração espacial, a confiabilidade dos circuitos era garantida através de técnicas de blindagem. Esta técnica funcionava através da redução do fluxo das partículas que atingiam o circuito e conseqüentemente, reduzindo o numero de erros causados pelo choque de partículas a zero. Entretanto, com a evolução tecnológica os circuitos se tornaram mais sensíveis ao choque das partículas, o que tornou a técnica de blindagem obsoleta em se tratando de proteger circuitos de tecnologia submicrométrica.

Buscando atingir o mesmo nível de proteção garantido pela blindagem, os cientistas vêm propondo diversas técnicas para mitigar os SEUs e os SETs. Nesta seção são apresentadas algumas destas técnicas e seus respectivos custos em termos de área e tempo de processamento.

Diversas técnicas relacionadas aos processos de fabricação foram desenvolvidas dentre as quais podemos destacar o *wafer thining* (DODD, 2001), a eliminação do BPSG (*boron phosphor-silicate glass*) do processo de fabricação (BAUMANN, 2001) e o uso de SOI ao invés do processo de poço tradicional (HARELAND, 2001). Entretanto, o custo dos materiais envolvidos nestas técnicas é caro e se aplicam apenas a alguns projetos em especial. Existem também técnicas chamadas de técnicas de fortificação de células. Estas técnicas podem ser categorizadas em: técnicas para aumentar a capacitância dos nós críticos (KARNIK, 2001; OOTSUKA, 1998; KARNIK 2002); técnicas para reduzir eficiência da propagação das cargas que afetam o circuito ou técnicas para compensar a perda de cargas (SEXTON 1991). Estas técnicas geralmente comprometem ou o consumo de energia do

circuito, ou o tempo de propagação do circuito ou a área do circuito e as vezes mais de um desses fatores.

Devido aos altos custos das técnicas de proteção aplicadas na fase de processo, os cientistas buscaram técnicas que afetassem os outros níveis presentes na fabricação de um CI. Técnicas relacionadas ao projeto de circuitos foram apresentadas pela comunidade científica e basicamente se dividem em técnicas de detecção de erros e técnicas de detecção e correção de erros. Dentre as técnicas de detecção propostas, podemos citar como uma das mais conhecidas a técnica de duplicação com comparação ou *duplication with comparrison* (DWC). Essa técnica se baseia em duplicar o módulo de HW a ser protegido e comparar a resposta dos dois módulos. Caso as respostas dos dois módulos forem diferentes, um erro foi detectado. Esta técnica geralmente acarreta em um considerável custo de área de circuito já que ela exige que o modulo a ser protegido seja duplicado, além de exigir nova computação para garantir a resposta correta. Para solucionar o problema da re-computação os cientista propuseram a técnica assim chamada de ou *triple modular redundancy* ou TMR (JOHNSON, 1994). Esta técnica se baseia em triplicar o modulo a ser protegido e adicionar um votador para escolher a resposta correta com base nas três respostas dos módulos. A resposta que mais for apresentada pela maioria dos módulos (pelo menos 2 módulos) é a resposta certa. Esta técnica além de acrescentar um grande custo de área de circuito, não garante o correto funcionamento no caso de uma falha atingir o votador nem protege o circuito contra falhas múltiplas. Além das técnicas descritas acima, os cientistas propuseram técnicas baseadas em códigos de detecção e correção de erros. Um exemplo clássico destas técnicas que detectam e corrigem erros através de códigos é o código de Hamming. Este código é bastante utilizado para proteger memórias contra falhas simples em palavras de memória (HENTSCHKE, 2002). Entretanto, o código de Hamming não protege memórias contra falhas múltiplas. Neste caso técnicas como os códigos de Bose-Chaudhuri-Hocquenghen (BCH) e Reed-Solomon

(RS) (NEUBERGER, 2005) baseadas em aritmética de campos finitos, (também conhecidos como Campos de Galois). Os códigos de BCH podem corrigir um certo número de bits em qualquer posição da palavra de memória enquanto que o código de RS agrupa os bits em blocos para corrigi-los.

Além das técnicas chamadas de técnicas de hardware acima apresentadas, existem técnicas chamadas de técnicas de software. Estas técnicas são utilizadas quando ou não queremos ou não podemos modificar o hardware do sistema. Podemos dividir as diversas técnicas de proteção de software em três grandes grupos: técnicas implementadas em software, técnicas que adicionam algum hardware e técnicas híbridas, ou seja, que modificam o software e adicionam algum hardware. Dentre as técnicas implementadas em software podemos destacar: a técnica *Enhanced Control Flow Checking using Assertions* (ECCA) proposta por (ALKHALIFA, 1999), *Control Flow Checking using Assertions* (CCA) proposta por (MCFEARING, 1995), *Control Flow Checking by Software Signatures* (CFCSS) proposta por (OH, 2002b), e a técnica *Error Detection by Data Diversity and Duplicated Instructions* (ED^4I) proposta por (OH, 2002a). Apesar de muito efetivas, as técnicas implementadas em software introduzem custos adicionais no tempo de processamento que limitem sua utilização apenas para aplicações em que o tempo de processamento não é uma variável crítica. Além disso, essas soluções implicam em custo adicional de memória para guardar o código extra que deve ser adicionado pelo projetista. Estas soluções também necessitam de acesso ao código fonte da aplicação, o que impossibilita o uso de componentes de software recém lançados que possuem seu código fonte fechado.

Visto que os custos de se adotar uma solução de software para aumentar a tolerância a falhas podem ser muito altos e muitas vezes impagáveis, os cientistas propuseram outras soluções que se caracterizam por adicionar elementos de hardware em um sistema de processamento de software. Dentre as técnicas que se encontram neste domínio de atuação,

podemos citar uma técnica chamada de DIVA, do inglês *Dinamic Verification,* proposta por (AUSTIN, 2000). É uma técnica aplicada a processadores com pipeline que se utiliza de um verificador funcional para verificar se as operações que estão sendo executadas pelo processador principal estão corretas. Por ser mais simples do que o processador principal, o hardware verificador possui um hardware extremamente simples, pois é formado apenas pelo estágio de execução do processador principal, já que ele apenas recebe a indicação da operação a ser verificada e os operandos a serem utilizados. Apesar de possuir um custo adicional de hardware bem reduzido, esta solução não pode ser aplicada em sistemas em chip baseados em FPGAs que possuem um processador embarcado, pois o hardware verificador é implementado dentro do pipeline do processador principal. Além disso, está técnica assume que o verificador nunca falha já que o mesmo é construído com transistores maiores e conseqüentemente mais tolerantes a falhas provocadas pelo choque de partículas.

Buscando a combinação dos benefícios das técnicas de software com as técnicas de hardware, os cientistas propuseram técnicas chamadas de técnicas híbridas. Em (BERNARDI, 2006), os autores propõem o uso de re-execução das operações aritméticas e lógicas, através da duplicação de instruções e acrescentando instruções de comunicação com um hardware externo. Esta re-execução é comandada por um bloco de hardware chamado de I-IP do inglês *Infrastructure IP* que trabalha concorrentemente com o processador principal, executando as mesmas operações que o processador principal e comparando os resultados com os obtidos pelo processador principal. Caso os resultados das operações forem diferentes, o processador principal recebe a ordem de executar novamente a operação para obter um resultado correto. Além disso, esta técnica permite verificar se o processador principal está executando corretamente as instruções de desvio. Isso ocorre através da monitoração do contador de programa ou *program counter PC* pelo I-IP. Está técnica possui o inconveniente de exigir que o projetista modifique o código fonte. Existem situações em que esta exigência não pode ser

atendida ou exige muito trabalho do projetista. Pensando em solucionar este problema, em (LISBOA, 2006), é proposto um I-IP que se localiza entre o processador principal e a memória. Este I-IP identifica quais as instruções que estão sendo requisitadas pelo processador principal e envia uma nova seqüência de instruções para que o processador execute duplamente cada instrução para fazer a verificação se a operação foi realizada corretamente. Desta forma esta técnica não necessita que o código da aplicação seja modificado. Neste trabalho, uma solução híbrida tal como a solução proposta em (LISBOA, 2006) é apresentada para o processador MIPS de arquitetura RISC, sem exigir qualquer modificação no código da aplicação ou na arquitetura do processador MIPS.

Neste capítulo foram apresentadas diversas técnicas de tolerância a falhas que atuam em todos os estágios de produção de circuitos e sistemas baseados em processadores que executam software. Foi visto que as técnicas relacionadas a produção dos circuitos geralmente acarretam em custos de produção muito altos. Por outro lado, as técnicas de redundância de hardware implicam em um custo de área que pode ultrapassar 200% enquanto que as técnicas de redundância de software geralmente prejudicam o desempenho ou aumentam o tamanho da memória do sistema.

O trabalho aqui apresentado propõe duas soluções diferentes que melhoram a confiabilidade do sistema sem acrescentar os custos que são adicionados pelas soluções previamente apresentadas. A primeira solução apresenta a substituição de circuito combinacional por um circuito construído quase que na sua totalidade com memórias magnéticas. Estas memórias são intrinsecamente protegidas contra os erros causados pelo choque de partículas altamente carregadas. Foi visto que com algumas técnicas arquiteturais de controle os resultados em termos de tempo de processamento desta nova arquitetura quando comparados com a tradicional arquitetura do processador MIPS.

A segunda solução aqui apresentada foi uma técnica híbrida chamada de I-IP aplicada ao processador MIPS para tornar suas operações mais robustas sem necessitar alteração no código das aplicações ou na arquitetura do processador.

3 SÍNTESE DOS RESULTADOS

3.1 PRIMEIRA TÉCNICA: MEMPROC – PROCESSADOR BASEADO EM MEMÓRIA.

Para avaliar os pontos fortes e fracos da técnica proposta, foram realizadas simulações para comparar a arquitetura proposta com outras duas arquiteturas, a primeira, um processador chamado de FemtoJava (BECK, 2003b), com arquitetura de pilha e pipeline de 5 estágio. A segunda, o conhecido processador MIPS (PATTERSON, 2002). Foi utilizada a ferramenta CACO-PS para simular o comportamento das arquiteturas sob a presença de falhas. Além disso, foi utilizada a ferramenta Leonardo Spectrum para avaliar os custos de área de circuito necessários para implementar as arquiteturas.

Os resultados mostraram que a arquitetura proposta apresenta maior área em relação às duas arquiteturas comparadas, porém com melhor desempenho e mais robustez ao choque de partículas altamente carregadas. Isto se deve ao fato de que a arquitetura proposta se baseia em substituir o uso de circuito operacional por memória. Esta substituição acarretou em um acréscimo considerável em área da arquitetura proposta por outro lado, melhorou a tolerância a falhas, pois a memória acrescentada é imune à falhas causadas pelo choque de partículas.

3.2 Segunda Técnica: I-IP – Uma Técnica Não-Intrusiva para Detecção de Erros para Sistemas em Chip.

Esta técnica apresenta um IP (núcleo de propriedade intelectual) que pode ser inserido em um sistema em chip sem nenhuma mudança na arquitetura do núcleo do processador. Este IP é capaz de monitorar a execução da aplicação e detectar erros ocasionados por falhas transientes em instruções lógicas e aritméticas além de instruções de deslocamento do fluxo de execução do processador. Esta solução não exige nenhuma mudança no código da aplicação, o que reduz em muito o tempo do projetista para introduzir o I-IP para monitorar qualquer arquitetura desejada.

A arquitetura do I-IP foi descrita no simulador CACO-PS, e foi simulada junto com a arquitetura do processador MIPS previamente descrita para simulação da primeira solução aqui apresentada. A arquitetura do processador MIPS junto com I-IP foram simuladas em ambiente com injeção de falhas. Foi observado que nem todas as falhas que causaram erros foram detectadas. As falhas que atingiram os bits de seleção do banco de registrador não puderam ser detectadas pois este tipo de erro troca o valor do operando a ser utilizado tanto pelo I-IP quanto pelo processador principal. Desta forma, as duas vezes em que a operação é executada (pelo I-IP e pelo processador principal) a resposta obtida foi igualmente errada e, sendo igual, o erro não é detectado. Já os custos adicionados relacionados ao desempenho da aplicação se mostraram pequenos, já que o I-IP executa as suas tarefas em paralelo a execução do processador principal.

4 CONSIDERAÇÕES FINAIS

Este trabalho apresentou duas soluções candidatas a lidar com o problema ocasionado pela presença de SEUs e SETs. Estes eventos preocupam os projetistas dos sistemas que serão utilizados no futuro e até mesmo dos que estão sendo fabricados atualmente. A primeira solução apresentou a arquitetura MemProc, construída com memórias magnéticas ao invés circuitos combinacionais. A segunda solução, o núcleo I-IP para o processador MIPS, é indicado para os casos em que nem o hardware do processador e nem o software da aplicação podem ser modificados.

Ambas a soluções apresentadas possuem seus prós e contras. No caso da primeira solução, a área final do MemProc aumentou principalmente pelo acréscimo das duas memórias (a memória de microcódigo e a memória de operação de máscaras). A penalidade de área foi mais de 2 vezes maior do que a área do MIPS e 1,3 vezes maior do que a área do FemtoJava. Por outro lado, o tempo médio entre falhas foi 17 vezes melhor, ou seja, menor que o tempo apresentado pelo MIPS e mais de 49 vezes do que o tempo apresentado pelo FentoJava. Em se tratando do desempenho a arquitetura proposta também apresentou ganhos, 1,2 vezes mais rápida do que o MIPS e 2,2 vezes mais rápida que o FentoJava. A solução proposta apesar de não se tratar de uma solução definitiva, aponta para um caminho em que as arquiteturas de computadores terão que se preocupar mais com a questão da confiabilidade e a tolerância à falhas.

No caso da segunda solução apresentada neste trabalho, os resultados mostram que a idéia pode ser implementada em qualquer tipo de arquitetura RISC ou SISC. Apesar dos custos em desempenho serem consideravelmente altos, os custos em termos de área adicional

de circuito são apenas cerca de 15% com uma taxa de erros detectados de 74,5%. Porém a grande vantagem desta abordagem é que ela não é nem intrusiva no software e nem no hardware, o que a torna uma solução atrativa para um certo nicho de sistemas em que as outras soluções não podem ser implementadas.

ANEXOS – PUBLICAÇÕES

A seguir são apresentados os artigos publicados referentes aos dois assuntos desenvolvidos no decorrer desse trabalho. São 5 publicações no total, sendo 3 referentes a primeira solução proposta e 2 sobre a segunda solução aqui apresentada. Dentre estas 3 pubicações, pode-se destacar o artigo de título: "Using Memory to Cope with Simultaneous Transient Faults" apresentado no 7º Latin-American Test Workshop por ter recebido o prêmio de Best Paper Award. Com relação a segunda idéia apresentada neste trabalho, foram publicados 2 artigos sendo um destes um artigo publicado em revista IEEE. Os artigos seguem apresentados no formato da publicação e em ordem cronológica referente a data da publicação.

Title: Using Memory to Cope with Simultaneous Transient Faults

Authors:

Rhod, Eduardo
Depto. de Engenharia Elétrica
Av. Osvaldo Aranha, 103/206-B
90035-190 - Porto Alegre, RS, Brasil
Phone: +55 51 3316 3516
elrhod@eletro.ufrgs.br

Lisbôa, C. A. L.
Instituto de Informática
Av. Bento Gonçalves, 9500, Bloco IV
91501-970 - Porto Alegre, RS, Brasil
Phone: +55 51 3316 7748
calisboa@inf.ufrgs.br

Carro, Luigi
Av. Osvaldo Aranha, 103/206-B
90035-190 - Porto Alegre, RS, Brasil
Phone: +55 51 3316 3516
Depto. de Engenharia Elétrica
carro@eletro.ufrgs.br

Contact Person and Presenter:
Rhod, Eduardo
Depto. de Engenharia Elétrica
Av. Osvaldo Aranha, 103/206-B
90035-190 - Porto Alegre, RS, Brasil
Phone: +55 51 3316 3516
elrhod@eletro.ufrgs.br

Using Memory to Cope with Simultaneous Transient Faults

Rhod, Eduardo
Depto. de Engenharia Elétrica[1]
elrhod@eletro.ufrgs.br

Lisbôa, C. A. L.
Instituto de Informática[2]
calisboa@inf.ufrgs.br

Carro, Luigi
Depto. de Engenharia Elétrica[1]
carro@eletro.ufrgs.br

Abstract

Transistors in future technologies will be so small that they will be heavily influenced by electromagnetic noise and SEU induced errors. As a consequence, simple gates will behave as expected only a fraction of the time, and the probability of double soft errors occurring at the same time will increase. In order to face this challenge, new design approaches must be taken. In the memory design arena, several techniques have already been proposed in order to protect data. As an alternative to fully combinational circuits and in order to reduce the area susceptible to faults, the use of a companion memory that allows circuits to withstand up to two simultaneous faults is proposed here. The proposed solution has been analyzed in two test cases, through the simulation of fault injection. Experimental results show a significant reduction in the sensible area of the circuit, and in the probability of double simultaneous faults being propagated to the output of the circuit.

1. Introduction

As the microelectronics industry moves towards nanotechnologies, systems designers become increasingly concerned about the reliability of future devices, which will have propagation delays shorter than the duration of transient pulses induced by radiation attack, as well as smaller transistors, which will be more sensitive to the effects of electromagnetic noise, neutron and alpha particles that may cause transient faults, even in fully tested and approved circuits.

For this less reliable technology, it is likely that common gates, such as a simple NAND, will behave as expected only a fraction of the total time.

In order to survive in this new 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. Current fault tolerance techniques are effective for single event upsets (SEUs) and single event transients (SETs). However, they are unlikely to withstand the occurrence of multiple simultaneous faults that is foreseen with those new technologies [1,2].

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.

In this paper we propose a new approach to cope with this faulty behavior of gates, through the use of memory as an alternative to combinational circuits, in order to provide tolerance to multiple soft faults in digital circuits. The reason to extend the use of memory in current circuits is that several techniques have already been proposed in order to protect the data stored in memory from the effects of SEUs, even in the presence of two simultaneous upsets.

By combining memory circuits with combinational circuits one can obtain a hardened version of the original combinational circuit, at the expense of extra area. Experimental results show that, for two simultaneous faults, the proposed approach is more robust than TMR or N-MR.

This paper is organized as follows: section 2 describes related work, while in section 3 we describe the memory protection scheme used in this work and the test cases used in the evaluation of the proposed solution. Section 4 reports the results obtained in the implementation of the test case circuits. In section 5 we discuss the results obtained in the experiments and our plan for future work on this project.

2. Related Work

The possibility of increased incidence of soft errors due to noise or high-energy particles in the next technology generation is already a topic of concern [1-3]. These soft errors are not caused by poor design techniques or process defects, but rather they derive from the incidence of external radiation and/or electromagnetic noise, which become stronger as technology features shrinks, and there are fewer electrons to form the transistor channel.

What turns a soft error in combinational circuits into a major concern nowadays is that the higher frequencies to be reached by future circuits will lead to cycle times shorter than the duration of transient pulses caused by radiation and/or electromagnetic noise. Therefore, those pulses will have a higher probability of affecting the output of combinational circuits, long enough to be captured and stored as

[1] Av. Osvaldo Aranha, 103/206-B - 90035-190 - Porto Alegre, RS, Brasil - Phone: +55 51 3316 3516
[2] Av. Bento Gonçalves, 9500, Bloco IV - 91501-970 - Porto Alegre, RS, Brasil – Phone: +55 51 3316 7748

incorrect values in memory elements. Besides that, shrinking transistor dimensions and lower operating voltages will make circuits more sensible to neutron and alpha particles, which also induce transient pulses.

Considering current technology trends, it is clear that multiple upsets or electromagnetic noise will impact circuit behavior beyond the single upset hypothesis. This way, a design paradigm able to withstand multiple simultaneous upsets must be devised.

Several techniques to maintain circuit reliability even under those critical conditions have been proposed, including hardware implemented parity code and source level code modification [4], time/space redundancy [5-6], triple modular redundancy (TMR) and double modular redundancy with comparison (DWC) with concurrent error detection (CED) [7]. However, all these techniques are targeted to the occurrence of a single upset in a given time interval.

In parallel, research aiming the protection of memories against faults has also evolved. The first approaches, using the Hamming code, allowed to protect the data against single bit flips in a data word, allowing the correction of the wrong bit, as well as detection of double bit flips (without correction). The Reed-Solomon approach [8] is a solution that allows the detection and recovery of faulty bits in a single symbol, but is still susceptible to errors when bit flips occur in two separate symbols. The size of the symbols can be adjusted according to the application and the method allows the correction of as many faults (in a single symbol) as the number of bits in the symbol.

Recently, another work [9] proposed the combination of the Hamming and Reed-Solomon protection schemes in order to cope with simultaneous bit flips in two contiguous symbols. With this technique, one can build memories that tolerate two simultaneous faults occurring in the same symbol or in two symbols in different words in a memory array. This solution also allows the detection and correction of double bit flips in adjacent symbols of the same word, once a special placement of the RS coded words and does not impose an excessive penalty in area nor in delay.

Our work proposes the use of a combination of a combinational circuit and a memory as a replacement to the original combinational circuit, thereby reducing the area susceptible to double simultaneous faults and, in consequence, the probability of transient faults affecting the output of the circuit. This technique does not completely eliminate the occurrence of errors, but reduces the probability of those to a level that is acceptable for many practical applications such as multimedia and graphics processing, as we show in a filter test case.

Another strong argument to the use of memory is its intrinsic fault protection that comes with its spare columns and spare rows, like it is done today in DRAMs.

3. Using Protected Memory to Reduce Combinational Logic

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 (which can get up to 2.9 Tbytes/s), designers decided to bring some functions executed by the processor into memory, to make effective use of all these available data. The so-called Computational-RAM, presented in [10], brings processor functions into the memory. This technique was originally used as a SIMD (Single Instruction Multiple Data) Processor in some DSP applications.

3.1 Area Analysis

In the proposed approach, the protected 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:

$$\text{Size} = 2^{I} \times O, \quad (1)$$

where I is the input width and O is the output width, both in bits.

Since the memory size grows exponentially with the width of the inputs, it is unthinkable to use it as the sole implementation means to develop a circuit able to withstand two simultaneous faults. Our proposal is to replace part of the combinational circuit by a memory that withstands two simultaneous faults, without adding a meaningful amount of unprotected hardware, like TMR and N-MR approaches do by adding a voter.

At first sight, it seems that the amount of area that one must add to the circuit will make the proposed solution unfeasible, from the area cost standpoint. However, as will be shown in the test cases discussed in section 4, this drawback can be significantly reduced with adequate design approaches, as shown in next subsection.

3.2 Timing Analysis

When replacing combinational logic with memory, a delay is added to the computing time of the function, due to the memory access time, the set and propagation times of flip-flops and the number of cycles required to compute the new function. However, at the same time, the critical path of the combinational logic is significantly reduced, and this must be taken into account when comparing the performance of the proposed solution with that of the fully combinational circuit.

Calling T_{comb} the total computing time of the original combinational function, T_{prop} the computing time of the proposed solution, and N the number of cycles required to perform the equivalent combinational function, one has:

$$T_{prop} = N * (T_{mem} + T_{newcomb} + T_{setFFs})$$

and the goal of the designer will be to minimize the T_{prop} / T_{comb} ratio.

Even if the total delay time of the proposed solution is larger than the one of the combinational approach, by using the memory-based technique one is trading performance for reliability.

4. Test Cases

In order to explain the proposed technique, two test cases have been simulated: one 4x4-bit multiplier is implemented using two different approaches to replace combinational elements with memory, and a 4-tap FIR filter with 8-bit inputs and coefficients is implemented to illustrate the results of our proposal in a practical application circuit.

4.1 The 4x4-bit Multiplier

To show the reduction of the fault propagation rate obtained using our proposed approach, we chose one 4x4-bit multiplier as a case study. To illustrate this, we implemented two solutions with different amounts of memory and combinational circuit. The first one, here called *the column multiplier*, has 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 fault rate of the 4x4-bit multiplier implemented with the fully combinational circuit shown in Figure 1.

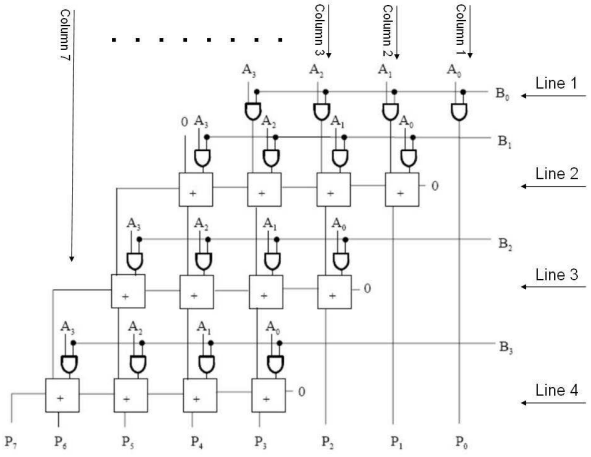


Figure 1. Fully combinational 4x4 bit multiplier

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 shown in Figure 1, since the goal is to avoid faults in the combinational circuit part.

4.1.2 The Column 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 1) 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 2 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 2 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. All the registers required in this additional circuitry, as well as the memory, are protected using the Reed Solomon method [9].

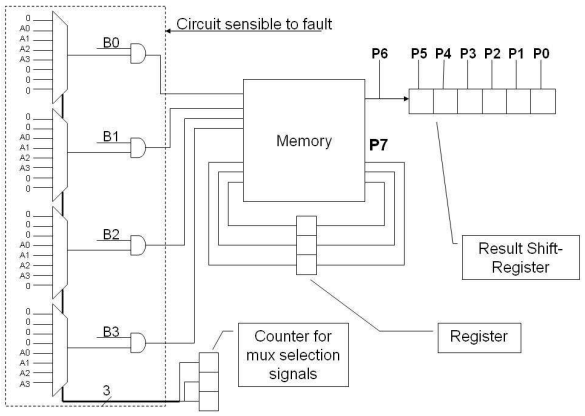


Figure 2. Column multiplier circuit.

The combinational circuit that is sensible to faults is highlighted in Figure 2 with a dashed rectangle.

4.1.1 The Line Multiplier

In this circuit the 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, that in our case is four. During the first three cycles, only one result bit per cycle is generated. The four remaining bits are calculated in the last cycle. In Figure 3 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.

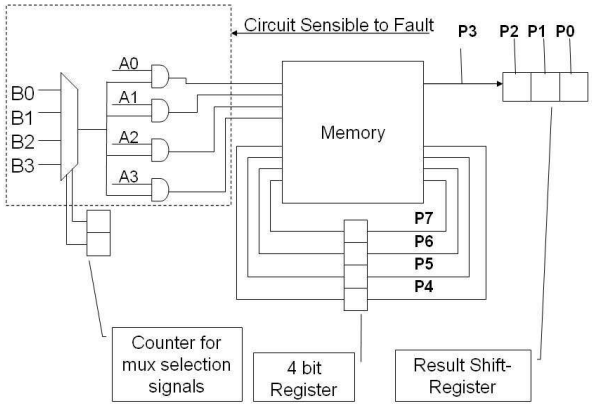


Figure 3. Line multiplier circuit.

The size of the counter that generates the selection signals for the multiplexer was also smaller (2 instead of 3). On the other hand, the shift register used to store signals from one cycle to the other was larger. This and other area characteristics from all the solutions are compared in Table 1. This table also shows the costs of the Reed Solomon protection, both for memory and registers.

Table 1. Area for each solution (# of transistors).

Solution	Combin. Circuit	Flip-flops	Voter	Memory	RS cod/dec	Total
5-MR	1520		608	-	-	2128
TMR	912		160	-	-	1072
Comb	304			-	-	304
Column	100	288		1536	80	2004
Line	36	296		3840	80	4252

The combinational circuit for the line multiplier was smaller than the one in the column multiplier. That happened because in the line multiplier only one bit of input B is necessary for the AND operations, while in the column multiplier each bit of input B is necessary for the AND operation with one bit of input A. On the other hand, the total area of the column multiplier was smaller than the one of the line multiplier. That happened because in the line multiplier more output signals from a cycle become input signals to the next cycle and this makes memory size grow. 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.

4.1.3 Fault Injection Simulations and Results

The fault injection was simulated using CACO-PS [11], a cycle-accurate, configurable power simulator, which was extended to support single and double simultaneous transient fault 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, the fault was propagated to the output. All the process is repeated again, for each combination of signals of the circuit.

Both implementations of the multiplier using memory were compared with the fully combinational solution of Figure 1 and with the classical TMR solution [12]. The resulting fault propagation rates can be seen in Table 2, for single fault injection, and in Table 3 for two simultaneous faults injection.

Table 2. Fault rate for single faults.

Circuit	# of gates that fail	Fault rate (%)	Proportional fault rate (%)
5-MR	532	8.80	8.80
TMR	268	5.49	2.77
Combinational	76	49.02	7.00
Column	33	46.82	2.90
Line	9	70.23	1.19

Table 3. Fault rates for two simultaneous faults.

Circuit	# of gates that fail	Proportional fault rate (%)
5-MR	532	20.50
TMR	268	8.19
Combinational	76	8.95
Column	33	4.19
Line	9	1.53

In Table 2, one can see that the fault rate (3rd column) has increased in the solutions using memory. 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 fault rate evaluation (4th column), as the percentage of observable faults at the output, one can see the benefits of the proposed solutions. Therefore, in tables 3, 5, and 6, only the proportional fault rate is shown.

4.1.4 Area versus Fault Tolerance Trade-off

When contrasting the results in tables 1 and 3, 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 will be a concern.

Another important observation is that, depending on the design alternative, the area × fault tolerance trade-off may impact quite differently the adopted solution, when contrasted with the TMR approach. For the column multiplier, the area increases almost twice, while the fault propagation percentage is reduced by the same ratio. For the line multiplier, however, the area increases by a factor of 4, while the fault rate decreases by a factor greater than 5.

4.2 The 4-tap FIR Filter

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

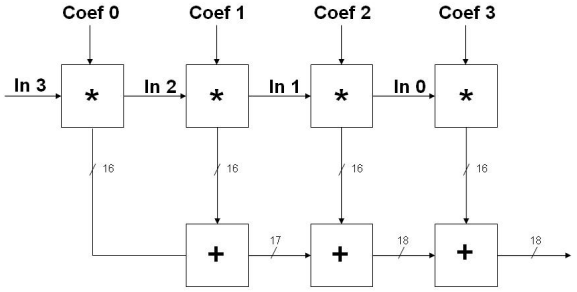


Figure 4. Combinational circuit for the 8-bit FIR filter with 4 taps.

The filter implementation using memory to replace part of the combinational logic is illustrated in Figure 5.

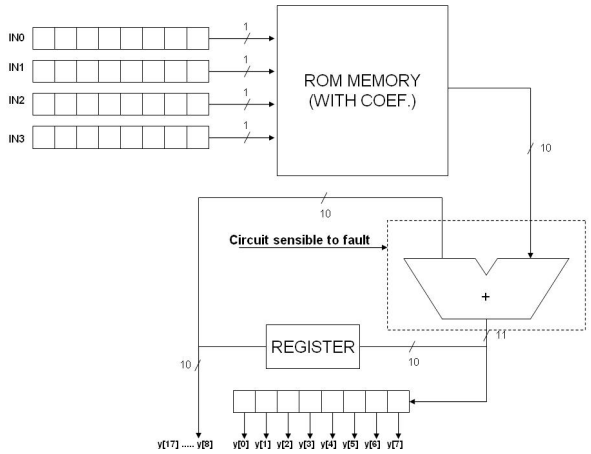


Figure 5. Filter implementation using memory.

4.2.1 The proposed approach: using memory

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

Erro!

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

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 area data is shown in Table 4. Since this is a pipeline filter, it was necessary to add 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 generated in each cycle.

Table 4 – Areas of the filter implementations (# of transistors)

Solution	Combin. circuit	Flip-flops	Memory	RS enc/dec	Total
Comb.	6524		-	-	6524
Mem.	200	760	320	552	1832

4.2.2 Fault Injection Simulations and Results

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 fault injection to evaluate the fault rate for the fully combinational solution and for the one using memory.

To implement the fault injection in a faster way, we implemented the filter in VHDL and synthesized both filter architectures, using a FPGA (Altera EP20K200EFC484-2X). The results are shown in Table 5, for single faults, and in Table 6, for double faults.

Table 5. Fault rate results for single faults in FIR filter implementations.

Solution	# of gates that fail	Proportional fault rate (%)
Comb.	1631	48.21
Memory	50	2.58

Table 6. Fault rate results for two simultaneous faults in FIR filter implementations.

Solution	# of gates that fail	Proportional fault rate (%)
Comb.	1631	67.35
Memory	50	2.96

In this case study we can see that the proportional fault rate of the memory solution is more than 20 times smaller than the combinational solution for single and double simultaneous faults. The area figures in table 4 show that the memory

solution has less than 3 times the area of the combinational solution.

5. Conclusions and Future Work

This work proposed the use of a memory, protected against double simultaneous transient faults using a Reed-Solomon scheme, in order to reduce the area of combinational logic susceptible to faults.

Two different circuits have been implemented in order to analyze the feasibility of the proposed solution in terms of tolerance to double simultaneous faults, and through fault injection simulations the reduction of the fault propagation rates has been demonstrated.

5.1 Discussion

The experiments conducted with the injection of faults in the example circuits have shown that the solution indeed reduces the probability of double simultaneous faults affecting the output of the circuit, since the area subject to faults (remaining combinational logic) is much smaller than that in the fully combinational implementations. Also, the hypothesis that increased redundancy in the logic (n-MR) provides higher reliability was not confirmed, since when two simultaneous faults occur, strategies like the one proposed here provide better results than n-MR, as shown by the obtained results.

However, since we are replacing combinational logic with memory, there is a penalty in performance, which has yet to be improved.

In any case, when comparing performance, the cost of failure must also be taken into consideration, since there are many applications in which, if the circuit fails, the cost will be unbearable. For those applications, fault tolerance is the major concern and, therefore, the proposed solution will be the most indicated, trading performance for reliability.

5.2 Future Work

The next step in the exploration of the use of memory to replace combinational logic will be the evaluation of the impact of this technique in the computing time of digital circuits for different purposes. From the area figures shown in Tables 1 and 4, it can be seen that there is no direct relation between the area of the fully combinational circuit and that of the solutions using memory. Concerning this issue, there is a large design space to be explored, with different area reduction/increase ratios, according to the expertise of each designer, as can be noticed comparing the areas obtained for the different multiplier implementations (Table 1) and in the filter implementation (Table 4). The definition of design techniques that will lead to the optimal results, in terms of area vs. fault tolerance, is another topic to be addressed in future works, aiming the automation of this design step.

This work is part of a larger research project, aiming the development of new design techniques tailored to future technologies. Those techniques will be applied in the construction of a whole processor that will be tolerant to multiple simultaneous transient upsets.

6. References

[1] Constantinescu, C., "Trends and Challenges in VLSI Circuit Reliability", *IEEE Micro*, vol. 23, no. 4, pp. 14-19, IEEE Computer Society, New York-London, July/August 2003.

[2] Edenfeld, D.; Kahng, A.B.; Rodgers, M.; Zorian, Y., "2003 Technology Roadmap for Semiconductors", *IEEE Computer*, vol. 37, pp. 47-56, IEEE Computer Society, New York-London, January 2004.

[3] Semiconductor Industry Association. International Technology Roadmap for Semiconductors - ITRS 2003,http://public.itrs.net/Files/2003ITRS/Home2003.htm. Accessed in 17/11/2005.

[4] Pflanz, M. and Vierhaus, H. T., "Online Check and Recovery Techniques for Dependable Embedded Processors", *IEEE Micro*, vol. 21, number 5, pp. 24-40, IEEE Computer Society, New York-London, September-October 2001.

[5] Anghel, L., Alexandrescu, D. and Nicolaidis, M., "Evaluation of soft error tolerance technique based on time and/or space redundancy", in *Proceedings of 13th Symposium on Integrated Circuits and Systems Design (ICSD 2000)*, pp. 237-242, IEEE, Manaus, Brazil, September 2000.

[6] Anghel, L. and Nicolaidis, M., "Cost Reduction and Evaluation of a Temporary Faults Detection Technique", in *Proceedings of Design, Automation and Test in Europe Conference (DATE 2000)*, pp. 591-598, ACM, Paris, France, March 2000.

[7] Lima, F., Carro, L. and Reis, R., "Techniques for Reconfigurable Logic Applications: Designing Fault Tolerant Systems into SRAM-based FPGAs", in Proceedings of the International Design Automation Conference, DAC 2003, pp. 650-655, ACM, New York, 2003.

[8] Houghton, A. D., The Engineer's Error Coding Handbook. London, UK: Chapman & Hall, 1997.

[9] Neuberger, G., Kastensmidt, F., Reis, R. **"**An Automatic Technique for Optimizing Reed-Solomon Codes to Improve Fault Tolerance in Memories**"**, in *IEEE Design and Test of Computers*, Volume 22, Issue 1, pp. 50-58. IEEE Computer Society, New York, 2005.

[10] Elliott, D.G., Stumm, M., Snelgrove, W.M., Cojocaru, C., Mckenzie, R., "Computational RAM: implementing processors in memory**",** *Design & Test of Computers, IEEE***,** vol. 16, no. 1, pp. 32-41, IEEE Computer Society, New York-London, Jan/Mar 1999.

[11] Beck F^{o}, A. C. S., Mattos, J. C. B., Wagner, F. R. and Carro, L., "CACO-PS: A General Purpose Cycle-Accurate Configurable Power-Simulator", in Proceedings of the 16th Brazilian Symposium on Integrated Circuits and Systems Design (SBCCI 2003), Sep. 2003.

[12] Johnson, B. W., Design and Analysis of Fault Tolerant Digitals Systems: Solutions Manual. Reading, MA: Addison – Wesley publishing Company, October 1994.

Fault Tolerance Against Multiple SEUs using Memory-Based Circuits to Improve the Architectural Vulnerability Factor

Rhod, Eduardo[1] elrhod@ece.ufrgs.br
Lisbôa, C. A. L.[2] calisboa@inf.ufrgs.br
Álisson Michels[1] alisson.michels@ufrgs.br
Carro, Luigi[1,2] carro@ece.ufrgs.br

Abstract

Technology trends for semiconductors forecast a higher incidence of soft errors caused by radiation on digital circuits implemented using sub 65nm technologies. As a consequence, the single-fault paradigm, which has been the model for circuit designers for many years, will no longer hold, and new design approaches are necessary to generate circuits that are able to withstand multiple simultaneous upsets. Newer memory technologies, like magnetic RAM, ferroelectric RAM and flash, are not affected by high energy particle strikes. Therefore, using small memories as building blocks, this work proposes to replace parts of combinational circuits with those intrinsically protected memories, thus reducing the overall architectural vulnerability factor (AVF), and, consequently, the soft error rate. The proposed approach has been applied to the implementation of multipliers and a FIR filter and the simulated injection of double transient faults has confirmed the initial assumptions, with a 8 to 30 times reduction in AVF.

1. Introduction

The constant growth of the semiconductor industry has led to great improvements in the performance of electronic devices. However, this performance gain brought increased concern regarding the reliability of these advanced circuits. As nanotechnologies arrive, circuits are becoming more susceptible to high energy particle strikes such as neutrons from cosmic rays and alpha particles from packaging material. These strikes can produce or stimulate bit flips, also known as single event upsets (SEUs), which can compromise the correct functionality of the circuit, provoking soft errors (SEs). Soft error rates (SER) for logic circuits used to be negligible compared with the failure rate of memory devices. However, for 100 nm technologies and beyond, logic SER must to be taken into account. For a comprehensive introduction to the subject of soft errors, the reader is referred to [1].

Current fault tolerance techniques are effective for SEUs and single event transients (SETs). However, they are unlikely to withstand the occurrence of multiple simultaneous faults that is foreseen with those new technologies [2, 4]. The probability of multiple-bit upsets increases relatively fast with technology scaling, compared with the probability of an upset in a single bit. In the near future SER will become as important as the performance or power characteristics of electronic circuits.

In this paper, we propose an alternative method to reduce the soft error rate of combinational circuits, by reducing its architectural vulnerability factor (AVF), through the use of a companion memory. Our technique aims to reduce the area susceptible to single and double bit flips, using a memory based circuit in the place of the traditional combinational one.

Soft errors occur in SRAM and DRAM devices, but not in ferroelectric RAMs (FRAMs), magnetic RAMs (MRAMs), or flash memories [5]. Our proposal is to reduce SER by using a magnetic memory, and some additional combinational circuit, to replace the fully combinational one.

In order to explain the proposed technique, two different versions of a 4x4-bit multiplier using memory to replace logic have been implemented. As a practical example of the application of our technique, the implementation of a FIR filter, is also presented in Section 5. Our results show that, with some penalty in area and computational time, we can improve the AVF by a factor of 8 against the TMR solution. With no protection technique, our solution decreased the AVF by a factor of 30. By reducing the overall AVF one consequently reduces the SER.

This paper is organized as follows: section 2 describes related work, while section 3 details the SER estimation approach used in this paper. Section 4 explains the memory characteristics and its influence in area and performance results. Section 5 reports the test cases and its results in terms of area, performance and AVF. In section 6 we discuss the results obtained in the experiments and our plans for future work on this project.

[1] Departamento de Engenharia Elétrica. Av. Osvaldo Aranha, 103/206-B - 90035-190 - Porto Alegre, RS, Brasil - Phone: +55 51 3316 3516
[2] Instituto de Informática. Av. Bento Gonçalves, 9500, Bloco IV - 91501-970 - Porto Alegre, RS, Brasil - Phone: +55 51 3316 7748

2. Related Work

Soft errors derive from the incidence of external radiation, whose effects become stronger as technology features shrink, and there are fewer electrons to form the transistor channel. Shrinking transistor dimensions and lower operating voltages will make circuits more sensible to neutron and alpha particles, which induce transient pulses.

Several techniques to maintain circuit reliability have been proposed, including hardware implemented parity code and source level code modification [6], time/space redundancy [7, 8], triple modular redundancy (TMR) and double modular redundancy with comparison (DWC) with concurrent error detection (CED) [9]. However, all these techniques are targeted to the occurrence of a single upset in a given time interval.

In parallel, research aiming the protection of memories against faults has also evolved. The first approaches, using Hamming code [10], allowed one to protect the data against single bit flips in a data word, allowing the correction of the wrong bit, as well as detection of double bit flips (without correction). The Reed-Solomon approach [10] is a solution that allows the detection and recovery of faulty bits in a single symbol, but is still susceptible to errors when bit flips occur in two separate symbols. The size of the symbols can be adjusted according to the application and the method allows the correction of as many faults (in a single symbol) as the number of bits in the symbol.

Recently, another work [11] proposed the combination of the Hamming and Reed-Solomon protection schemes in order to cope with simultaneous bit flips in two contiguous symbols. With this technique, one can build memories that tolerate two simultaneous faults occurring in the same symbol or in two symbols in different words in a memory array. This solution also allows the detection and correction of double bit flips in adjacent symbols of the same word, once a special placement of the RS coded words and does not impose an excessive penalty in area nor in delay.

3. Soft Error Rate

The soft-error-rate (SER) of a design can be expressed by the amount of errors manifested in a given period of time. One of the most known ways to measure the SER is evaluating the Failure in Time (FIT) of the design [12]. For example, a soft error rate of 10 FIT means that the circuit will have 10 errors in 1 million years. Another very commonly used metric to express SER is the Mean Time to Failure (MTTF). FIT and the MTTF are inversely related.

The soft-error rate of a design can also be expressed [13, 14] as follows:

$$SER^{design} = \sum_{i} SER_i^{\text{nominal}} \times TVF_i \times AVF_i \quad (1)$$

Where i represents the i^{th} element of the design.

The $SER^{nominal}$ is defined by the probability of a SEU occurring on a specific node of the element. This probability depends on the element type, transistor size, node capacitance, and other static characteristics of the element. For instance, to estimate the $SER^{nominal}$ for a latch, one must know Q_{critic}, which identifies the minimum charge necessary to cause the element to failure. More details can be found in [13].

The timing vulnerability factor (TVF) can be summarized as the fraction of time that the element can fail. For combinational logic, the timing vulnerability factor depends on the type of logic, which can be data path or control path. More details on these and other TVF evaluation can be seen in [13,14]. In this paper we propose an alternative to reduce the soft error rate by reducing the overall architecture vulnerability factor (AVF) of the system.

The architecture vulnerability factor of an element can be understood as the probability that a fail in that element cause a fail in the circuit. The AVF value of an element depends on its inputs and also on the importance of that element for the circuit. In this paper we estimate the AVF by injecting faults and detecting if those faults have caused system errors.

4. Using Magnetic Memory to Reduce Combinational Logic

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 our approach, 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:

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

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

Since the memory size grows exponentially with the width of the inputs, it is unthinkable to use it as the sole implementation means to develop a circuit able to withstand one or more simultaneous faults. Our

proposal is to replace part of the combinational circuit by a magnetic memory, without adding a meaningful amount of unprotected hardware, like TMR and N-MR approaches do by adding a voter.

At first sight, it seems that the amount of area that one must add to the circuit will make the proposed solution unfeasible, from the area cost standpoint. However, as will be shown in the test cases discussed in section 5, this drawback can be significantly reduced with adequate design approaches.

When replacing combinational circuits by memories, a delay is added to the computing time of the function, due to the memory access time, the flip-flops set and propagation times and the number of cycles required to compute the new function. However, at the same time, the critical path of the new combinational logic is significantly reduced, and this must be taken into account when comparing the performance of the proposed solution with that of the fully combinational circuit. These aspects are further discussed in section 5.

5. Test Cases

In order to explain the proposed technique, two test cases have been simulated: one 4x4-bit multiplier is implemented using two different approaches to replace combinational elements with memory, and a 4-tap FIR filter with 8-bit inputs and coefficients is implemented to illustrate the results of our proposal in a practical application circuit.

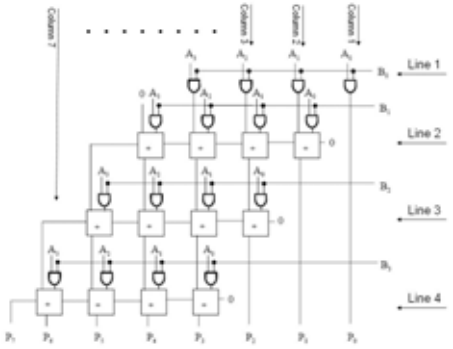


Figure 1. Fully Combinational 4x4-bit Multiplier

We implemented two architectures for the 4x4-bit multiplier, with different amounts of memory and combinational circuits. The first one, here called *the column multiplier*, has 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 AVF of the 4x4-bit multiplier implemented with the fully combinational circuit shown in Figure 1.

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.

In Figure 2 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 2 also shows that some additional circuitry has been added in order to properly generate control signals. The combinational circuit that is sensible to faults is highlighted in Figure 2 with a dashed rectangle.

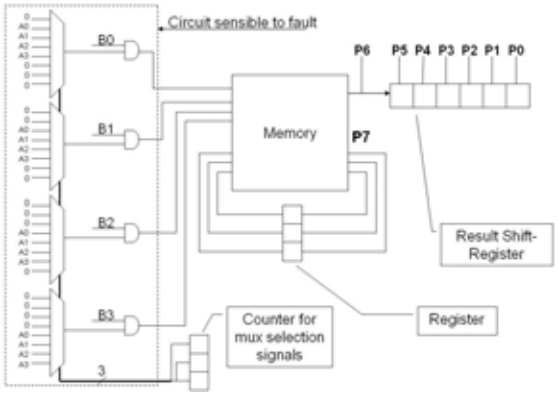


Figure 2. Column Multiplier Circuit.

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

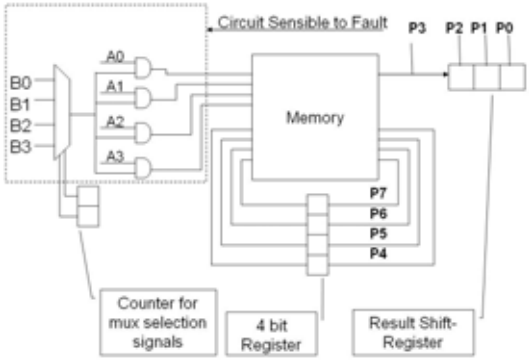


Figure 3. Line multiplier circuit.

In Figure 3 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.

Area characteristics from all the solutions are compared in Table 1. This table also shows the costs of the Reed Solomon protection used for registers.

Table 1. Area for Each Solution (# 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
Comb.	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 1 transistor. 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 [15], a cycle-accurate, configurable power simulator, which was extended to support single and double simultaneous transient fault 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, the fault was propagated to the output. All the process is repeated again, for each combination of signals of the circuit. Both implementations of the multiplier using memory were compared with the fully combinational solution and with the classical TMR solution. The resulting fault propagation rates can be seen in Table 2, for single and two simultaneous fault 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 μm

In Table 2, one can see that the architectural vulnerability factor (3rd 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. Therefore, in table 4, only the proportional AVF is shown.

Table 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
Comb.	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 1 and 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 will be a concern. Another important observation is that, depending on the design alternative, the area × fault tolerance trade-off may impact quite differently 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 2, one can notice that the critical path in the memories 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 substituted. On the other hand, the total computational 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.

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

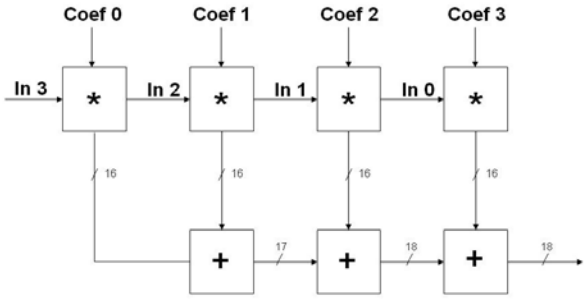


Figure 4. Combinational Circuit for the 8-bit FIR Filter with 4 Taps.

The filter implementation using memory to replace part of the combinational logic is illustrated in Figure 5.

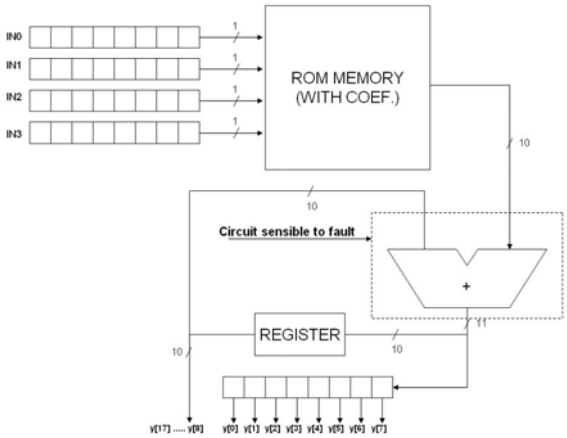


Figure 5. The Proposed Approach: Using Memory.

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

$$y(n) = \sum_{k=0}^{M-1} c_k \, x_{M-k}(n) \qquad (3)$$

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 area data is shown in Table 3.

Table 3. Areas of the Filter Implementations (# of Transistors)

	Comb.	Flip-Flops	Memory	RS Cod./ Decoder	Total
Combin.	16,494	-	-	-	16,494
Memory	540	1,700	900	484	3,624

Since this is a pipeline filter, it was necessary to add 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 generated in each 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 fault injection to evaluate the AVF for the fully combinational solution and for the one using memory.

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

Table 4. AVF Results for Single Faults 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

6. Conclusions

This work proposed the use of a memory, in order to reduce the area of combinational logic susceptible to faults.

Two different circuits have been implemented in order to analyze the feasibility of the proposed solution in terms of tolerance to double simultaneous faults, and through fault injection simulations the reduction of the AVF has been demonstrated.

The experiments conducted with the injection of faults in the example circuits have shown that the solution indeed reduces the probability of double simultaneous faults affecting the output of the circuit, since the area subject to faults (remaining combinational logic) is much smaller than that in the fully combinational implementations. Also, the hypothesis that increased redundancy in the logic (n-MR) provides higher reliability was not confirmed, since when two simultaneous faults occur, strategies like the one proposed here provide better results than n-MR, as shown by the obtained results.

However, since we are replacing combinational logic with memory, there is a penalty in performance, which has yet to be improved. In any case, when comparing performance, the cost of failure must also be taken into consideration, since there are many applications in which, if the circuit fails, the cost will be unbearable. For those applications, fault tolerance is the major concern and, therefore, the proposed solution will be the most indicated, trading performance for reliability.

Even if the total delay time of the proposed solution is larger than the one of the combinational approach, by using the memory-based technique one is trading performance for reliability.

This technique does not completely eliminate the occurrence of errors, but reduces the probability of those to a level that is acceptable for many practical applications such as multimedia and graphics processing [16].

The next step in the exploration of the use of memory to replace combinational logic will be the evaluation of the impact of this technique in the computing time of digital circuits for other purposes.

7. References

[1] Ziegler J. F. *et al.*, "IBM experiments in soft fails in computer electronics (1978–1994)," *IBM J. Res. Devel.*, vol. 40, no. 1, pp. 3–18, Jan. 1996.

[2] Constantinescu, C., "Trends and Challenges in VLSI Circuit Reliability", *IEEE Micro*, vol. 23, no. 4, pp. 14-19, IEEE Computer Society, New York-London, July/August 2003.

[3] Edenfeld, D.; Kahng, A. B.; Rodgers, M.; Zorian, Y., "2003 Technology Roadmap for Semiconductors", *IEEE Computer*, vol. 37, pp. 47-56, IEEE Computer Society, New York-London, January 2004.

[4] Semiconductor Industry Association. International Technology Roadmap for Semiconductors - ITRS 2003,http://public.itrs.net/Files/2003ITRS/Home2003.htm. Accessed in 17/11/2005.

[5] Eto, A.; Hidaka, M.; Okuyama, Y.; Kimura, K.; Hosono. M., "Impact of neutron flux on soft errors in MOS memories.", In *Proc. IEEE Int. Dev. Meet. (IEDM)*, pages 367–370, 1998.

[6] Pflanz, M.; Vierhaus, H. T., "Online Check and Recovery Techniques for Dependable Embedded Processors", *IEEE Micro*, vol. 21, number 5, pp. 24-40, IEEE Computer Society, New York-London, September-October 2001.

[7] Anghel, L.; Alexandrescu, D.; Nicolaidis, M., "Evaluation of soft error tolerance technique based on time and/or space redundancy", in *Proceedings of 13th Symposium on Integrated Circuits and Systems Design (ICSD 2000)*, pp. 237-242, IEEE, Manaus, Brazil, September 2000.

[8] Anghel, L.; Nicolaidis, M., "Cost Reduction and Evaluation of a Temporary Faults Detection Technique", in *Proceedings of Design, Automation and Test in Europe Conference (DATE 2000)*, pp. 591-598, ACM, Paris, France, March 2000.

[9] Lima, F.; Carro, L.; Reis, R., "Techniques for Reconfigurable Logic Applications: Designing Fault Tolerant Systems into SRAM-based FPGAs", in Proceedings of the International Design Automation Conference, DAC 2003, pp. 650-655, ACM, New York, 2003.

[10] Houghton, A. D., "The Engineer's Error Coding Handbook.", London, UK: Chapman & Hall, 1997.

[11] Neuberger, G.; Kastensmidt, F.; Reis, R., **"**An Automatic Technique for Optimizing Reed-Solomon Codes to Improve Fault Tolerance in Memories**"**, in *IEEE Design and Test of Computers*, Volume 22, Issue 1, pp. 50-58. IEEE Computer Society, New York, 2005.

[12] Mukherjee, S. S.; Emer, J.; Reinhardt, S. K., "The Soft Error Problem: An Architectural Perspective", in Proc. 11th International Symposium on High-Performance Computer Architecture (HPCA), Feb. 2005, San Francisco, pp. 243-247.

[13] Nguyen, H.T.; Yagil, Y., "A Systematic Approach to SER Estimation and Solutions", Proc. IEEE Int'l Reliability Physics Symp., IEEE Press, 2003, pp. 60-70.

[14] Seifert, N.; Tam, N., "Timing Vulnerability Factors of Sequentials", IEEE Trans. Device and Materials Reliability, Sept. 2004, pp. 516-522.

[15] Beck Fº, A. C. S.; Mattos, J. C. B.; Wagner, F. R.; Carro, L., "CACO-PS: A General Purpose Cycle-Accurate Configurable Power-Simulator", in Proceedings of the 16th Brazilian Symposium on Integrated Circuits and Systems Design (SBCCI 2003), Sep. 2003.

[16] Johnson, B. W., "Design and Analysis of Fault Tolerant Digitals Systems: Solutions Manual. Reading", MA: Addison – Wesley publishing Company, October 1994.

A non-intrusive on-line control flow error detection technique for SoCs

E. L. Rhod[1], C. A. Lisbôa[2], L. Carro[2], M. Violante[3], M. Sonza Reorda[3]

[1]*Universidade Federal do Rio Grande do Sul, Escola de Engenharia, Dep. Engenharia Elétrica*
[2]*Universidade Federal do Rio Grande do Sul, Instituto de Informática, Dep. Informática Aplicada*
[3]*Politecnico di Torino, Dipartimento di Automatica e Informatica*
eduardo.rhod@ufrgs.br,{calisboa, carro}@inf.ufrgs.br,{massimo.violante,matteo.sonzareorda}@polito.it

Abstract

The time to market demands of embedded systems make the reuse of software and hardware components a mandatory design approach, while the growing sensitivity of hardware to soft errors requires effective error detection techniques to be used even in general purpose systems. Control flow error detection techniques are usually either hardware or software intrusive, requiring modification of the processor architecture or changes in the application software. This paper proposes a non-intrusive and low cost technique to be used in SoC designs, that is able to detect errors affecting the program counter with very small area and performance overheads, without the need of any changes in the core processor hardware nor in the application software.

1. Introduction

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.

At the same time, the technology evolution towards nanoscale brings along higher sensitivity of the hardware to radiation induced soft errors, caused by collisions of particles with the silicon. These events generate transient pulses that may change the logic output value of the affected gate and the wrong value may be propagated through an open logic path in the circuit, and eventually be stored in a memory element. Formerly a concern only for mission critical or space applications, the increase of the soft error rate of circuits manufactured with new technologies turned this topic into a challenge for system designers.

Many different techniques for soft errors mitigation have already been proposed, but most of them require modification of the hardware or software of the SoC's core processor, or even both. Besides that, these techniques frequently imply heavy area and/or performance penalties, that may not be bearable for a given application.

This paper proposes the use of an infrastructure IP (I-IP) that can be inserted in the SoC without any change in the core processor architecture, and is able to monitor the execution of the application and detect control flow errors generated by transient faults affecting the program counter of the core processor. The proposed technique does not require any modification of the application software, being non-intrusive from both hardware and software standpoints.

In fault injection tests of the proposed solution, performed for an implementation of the I-IP for a SoC based on a pipelined RISC core processor, running a benchmark application, all errors affecting the program counter have been detected.

This paper is organized as follows. Section 2 presents an overview of generic and control flow error detection techniques, section 3 describes the proposed approach, and section 4 presents the results obtained with one implementation of the I-IP for a MIPS based SoC. Section 5 summarizes the conclusions and points to future works.

2. Transient errors detection techniques

This section focuses on techniques to detect transient errors, i.e., those resulting from faults that may affect the behavior of a system in a temporary fashion, thereby allowing the system to eventually recover from the error and return to a consistent state. Recovery techniques are beyond the scope of this paper.

2.1. Generic online error detection techniques

Many on-line, or concurrent, error detection techniques have been proposed so far. Those techniques aim to detect errors that occur during the normal operation of systems, and therefore cannot be detected by test procedures in the manufacturing process.

According to the architectural level at which the technique is applied, they can be divided into *circuit-level* and *system-level* techniques. Self-checking circuits, error detecting codes, and parity schemes are examples of circuit-level techniques, while replication and watchdog processors are system-level approaches [1].

Another usual classification of those techniques divides them into *hardware based*, *software based*, and *hybrid techniques* [2].

Hardware based solutions generally imply the use of redundancy or reconfiguration [3]. Examples are the triple modular redundancy approach (TMR) [4], the use of checker circuits that run in parallel with the main processor to verify its operations [5], and the use of spare units that can replace a faulty one when an error is detected [6]. The main drawback of those approaches is the need to modify the hardware of the system to be hardened, which precludes their use to harden tested IP core processors or commercial off-the-shelf processors embedded in FPGA chips, which are frequently used in SoCs designs as a means to shorten the time to market of products.

In [2], a hardware based technique that uses an external infrastructure IP that monitors the core processor buses to harden applications, and therefore does not require modification of the core processor architecture nor of the application software, was proposed. Using consistency check for data processing instructions and also checking the core processor control flow, the implementation of the proposed solution to harden one application running on an 8051 microcontroller was proven to detect more than 81% of errors when only one instruction was hardened by the I-IP, and more than 87% with two hardened instructions.

Software based techniques are also system-level. In general, they imply modification of the application to be hardened, and this requires changes and/or additions to the source code of the application, which is not always available. Besides that, solutions in this class require additional memory to store the hardened application, and also have significant performance penalties [7, 8, 9].

Hybrid techniques, such as [10, 14], try to leverage on the strengths of hardware and software based ones, in order to optimize the area and performance overheads according to the target market and application.

2.2. Specific techniques for control flow error detection

This work focuses specifically on those techniques aiming to protect the system against transient faults that cause SEUs in the program counter of a processor during its operation, thereby causing control flow errors, which are a subset of the types of errors that may occur during the normal operation of a system. Control flow error detection techniques are generally based in the use of assertions or signature analysis [3], and some of them are commented in the following paragraphs.

The use of watchdog processors is sometimes suggested also for control flow check. In [11], an active watchdog processor executes the program concurrently with the core processor, and checks if its program flow proceeds as that executed by the main processor. This approach, however, has heavy penalties both in terms of application performance and the additional area required for the watchdog processor. Two alternative approaches, using a passive watchdog processor that computes a signature while observing the main processor's bus and performs consistency checks whenever the program enter or leaves a basic block within the program graph, are proposed in [12, 13]. While their area overhead is much smaller than that of active watchdog processors, there is a performance overhead introduced by instructions needed to communicate with the watchdog.

In [14], two software based and one hardware based mechanisms have been combined, in order to provide increased tolerance to transient faults, and this arrangement allowed the detection of 93% of the control flow errors. This work presented improvements obtained with the combination of the three techniques over the use of them separately, but no overhead analysis was provided.

In [15], a technique called Enhanced Control-Flow Checking Using Assertions (ECCA), which combines the use of application and system level mechanisms to provide on-line detection of control flow errors, was proposed. Tested with a set of benchmark applications, the ECCA technique 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.

Control Flow Checking by Software Signatures (CFCSS) is another important software based technique proposed in [8], which provided a dramatic reduction in the number of undetected control flow errors when tested with a set of benchmark applications. This technique also requires modification of the application code, and does not detect all control flow errors, due to some limitations in the detection of certain types of errors [9].

The use of control flow assertions was also proposed in [9]. This software based approach requires the introduction of additional executable assertions to check the control flow of the program. A 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 [8] and ECCA [15], however the technique proved to be very expensive in terms of memory and performance overhead, even though the overheads are application dependent.

The hardware based technique proposed in [2] aims to protect a core processor against transient faults, using the SEU in the memory elements of the processor as the fault model. As mentioned above, this technique did not focus only in the control error detection, but also in consistency check for the execution of data processing instructions. Being a non-intrusive technique that does not require modifications either in the application software nor in the hardware of the core processor,

there is no need to access the source code of the application. However, the technique did not detect all control errors.

Another software based fault tolerant approach, using time redundancy, proposed in [17], is evaluated in [3], using a Toshiba TX 49 commercial processor, a 64-bit radiation hardened processor, designed for space applications, and injecting faults only in the program counter. The experiments resulted in the detection of more than 96% of the injected faults.

The technique proposed in this paper is a system level, hardware based, non-intrusive one. It does not require any modification of the application software to be run in the SoC, allowing the reuse of applications supplied only in object code format. Moreover, the I-IP is connected externally to the buses of the core processor, allowing its implementation without any change in the core internal architecture.

3. Proposed Technique

The proposed technique is based on the insertion of an I-IP between the processor core of the SoC and the memory storing the instructions the processor core executes. This way, the I-IP is able to intercept the fetch addresses sent by the core processor and check if the correct control flow is being followed. This overall architecture is illustrated in Figure 1.

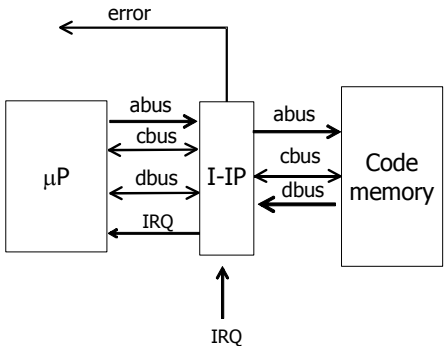


Figure 1. Overall architecture

By checking the fetch addresses, the I-IP monitors the value of the program counter to identify if its contents are correct or not, according to the control flow of the program being executed. When the I-IP finds an invalid address, it raises a flag indicating that an error was found. It is important to mention here that the I-IP does not correct the error, it only indicates that a fetch from a wrong address was attempted.

In this work, as done in [5], we assume that the I-IP is hardened by design and test, being tolerant to any kind of faults that can lead to any type of failure compromising its correct operation. We also assume that the code memory is protected through some EDAC (Error Detection and Correction) scheme, so that values stored in memory cannot be corrupted. Finally, we assume that the bus connecting the instruction cache to the processor is not accessible, as it often happens for processor cores, and therefore we assume that the instruction cache either does not exist, or is disabled.

To detect when the program counter has a wrong value, the I-IP identifies all fetched instructions by decoding and classifying them into two main groups:

Branch instructions: are all the instructions that can cause a deviation in the program control flow, such as unconditional jump or conditional branch instructions;

Other instructions: all other instructions that do not have the capacity to change the value of the program counter, such as logic and arithmetic data processing instructions, are classified in this group.

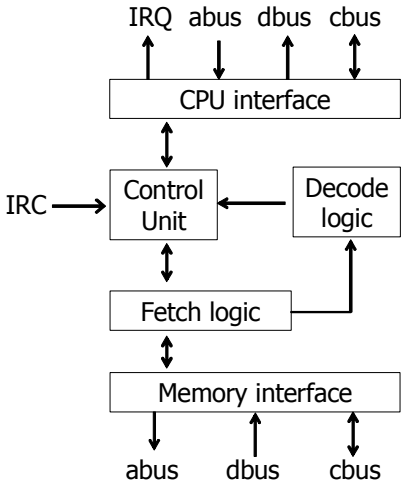


Figure 2. Architecture of the I-IP

Figure 2 shows the different hardware blocks of the I-IP, which will be described in the following paragraphs.

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 rest of the I-IP.

2) *Memory interface*: connects the I-IP with the memory storing the application the processor executes. This module executes commands coming from the *Fetch logic* and handles the details of the communication with the memory.

3) *Fetch logic*: issues to the *Memory interface* the commands needed for loading a new instruction into the I-IP and feeding it to the *Decode logic*.

4) *Decode logic*: decodes the fetched instruction, whose address in memory is *A*, and sends the details about the instruction to the *Control unit.* This module classifies instructions according to the previously described two categories: *branch instructions* and *other instructions*. The *branch instructions* are also sub-classified as *conditional branches,* such as the MIPS instruction *beq* (*branch if equal or zero*) or *unconditional branches*, such as the *j* instruction (*jump to address*).

5) *Control unit*: supervises the operation of the I-IP. Upon receiving a request for an instruction fetch from the *CPU interface*, it activates the *Fetch logic*. Then, depending on the information produced by the *Decode logic*, it either issues to the main processor a different sequence of instructions, as explained in the next paragraph, or sends to the processor the original instruction. Moreover, it implements the operations

needed for control-flow check, and, in case of error detection, the error flag is activated. Finally, it receives interrupt requests and forwards them to the processor core at the correct time. A special care is taken with the interrupt request handling by the I-IP, when a sequence of instructions is sent to the core processor instead of the original one. In order to allow a proper return from interrupt service subroutines to the point in the application program where the next instruction is located, in those cases the I-IP only forwards the interrupt request to the core processor after all instructions of the substitute sequence have been sent. In other words, the substitute sequence operation is treated as an atomic operation, which cannot be interrupted.

In order to correctly decode the instructions, when a new core processor is adopted for the first time, the I-IP must be tailored to the instruction set architecture of the target processor. The designer needs to program the decoding rules in the *Decode logic* block to indicate to the *Control unit* block if the current instruction is a branch instruction or not, and, depending on the branch instruction, if it is necessary to issue to the processor a different sequence of instructions. Such situation happens when the branch destination address is not included in the instruction word, together with the instruction *opcode*, as in the *jr* (*jump through register*) MIPS instruction. In this case, the I-IP needs to get the address value stored in the register, to check if the branch was taken correctly. Therefore, the designer has to provide the sequence of instructions that the I-IP will send to the processor in order to receive the branch address and return to the normal program flow, to continue execution.

Auxiliary tools to help the designer in the process of adapting the I-IP to a given target core processor, following the same model and design flow proposed in [2], will be developed as part of future work.

The topology of the proposed approach allows the I-IP to be used in any kind of SoC implementation in which the code memory is external to the processor core.

The use of this approach brings two advantages when compared to the alternatives commented in Section 2. First, from the software standpoint, it is non-intrusive, since the designer does not need to know the application code nor to modify this code. Second, since the I-IP is inserted outside the core processor and connects to it through to already available buses, no changes to the core processor architecture are necessary, which also characterizes this approach as non-intrusive in terms of hardware.

4. Case study: hardening a MIPS core

In this section we will discuss the experimental results obtained with an implementation of the proposed I-IP to harden the control flow of a widely used RISC core processor: a pipelined MIPS.

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:

a) 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;
b) 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* instruction, instead of the *jal* instruction. The first instruction is used to save the correct address in the register used to store the return address, and the *j* instruction performs the jump to the subroutine entry point.

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 stored address with the one in the program counter. Therefore, the I-IP has an internal circular register file, used to store up to four addresses that will be compared to the program counter a few cycles later.

4.1. Fault injection experiments

To evaluate the performance of the I-IP in control flow error detection, an in house tool named CACO-PS (Cycle Accurate Configurable Power Simulation) [16] was used to simulate the architecture of the SoC and check the results of fault injection.

The I-IP and the MIPS architecture were described in the language used by CACO-PS, which is System C like. During the fault injection procedure, 2,000 faults were injected randomly in time, causing SEUs in randomly chosen bits of the MIPS program counter register, while executing a software implementation of the Viterbi algorithm for encoding a stream of data.

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 an error occurred. 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. Figure 3 illustrates the error detection scheme described here.

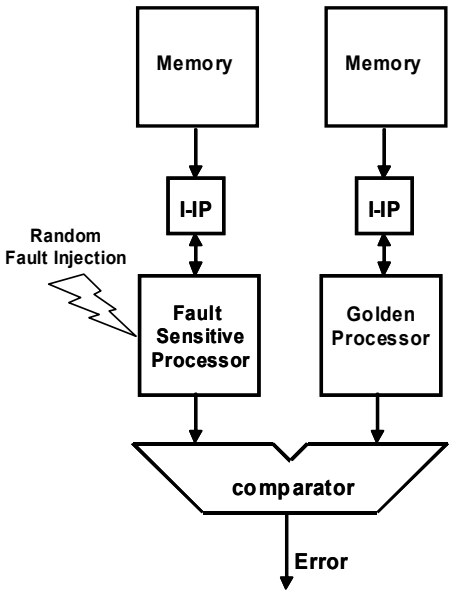


Figure 3. Error detection scheme

4.2. Fault coverage analysis

The proposed I-IP detected all SEU errors affecting the program counter bits during the fault injection experiments using the Viterbi application. Some special detection cases deserve a closer analysis, and are described in the following paragraphs.

First, one must recall that the MIPS core processor uses byte addressing to read from the code memory. Therefore, all instruction addresses must be a multiple of 4 and, in some MIPS implementations, a value different from zero in the two low order bits of the PC generates a processor exception. In the implementation used in our experiments, this was not true, and the I-IP itself was responsible to detect those errors by checking if the two bits are equal to zero for every fetch.

Second, approximately 3% of the detected errors were caused by faults that hit the 3rd least significant bit of the program counter. Due to the instruction addressing scheme described in the previous paragraph, when a bit flip from '1' to '0' of the 3rd bit occurs, it is equivalent to keep in the program counter the same value that was used in the previous cycle, what would make the processor execute the same instruction again, sometimes affecting the results produced by the program execution. In order to cope with these cases, the I-IP checks if two consecutive fetch operations ask for the same memory address and, if so, sends back a *nop* instruction to the core processor. On the other hand, when a SEU in the 3rd bit causes a bit flip from '0' to '1', one instruction is skipped and not executed; in this case, the I-IP detects the error because the fetch address is not the expected one (address *A* + size of the instruction).

Finally, although in this experiment all errors have been detected, it is important to highlight that there is one case in which the proposed I-IP fails in detecting an incorrect branch. This very unlikely situation can happen only with conditional branch instructions, for which there are two alternative valid destination addresses, one to be used when the branch is taken (A_{taken}), and the other when the program has to proceed with the instruction immediately following the branch (A_{next}). It happens when a fault occurs exactly during the execution cycle of a conditional branch instruction, and that fault changes the value of the program counter in such a way that the corrupted value is, by extreme coincidence, equal to the other valid value for that branch instruction, i.e., A_{taken} is transformed in A_{next}, or vice versa.

In order to detect such situation, it would be necessary to add new capabilities to the I-IP, allowing it to duplicate all instructions that affect the condition flags of the core processor and keep, inside the I-IP, replicas of the condition codes, in order to use their values to check the evaluation of conditions during the execution of conditional branch instructions. The additional area that would be required to implement these features, as well as the performance and area penalties that would be incurred, make this alternative not feasible.

4.3. Performance and area analysis

As commented in Section 2, most of the control flow error detection techniques proposed so far imply heavy performance and/or area overheads.

The technique proposed in this paper, besides its good error detection capability, requires very small performance and area overheads, as shown in Table 1.

Table 1. Performance and area overheads

	Without I-IP	With I-IP	Overhead
Performance (# of cycles)	8,779	9,972	13.59%
Area (# of gates)	38,340	41,982	9.5%

The performance information in Table 1 relates only to the execution of the specific Viterbi algorithm used in the experiments, which is normally executed by the core processor in 8,779 cycles. As described in Section 4, when the I-IP is present in the SoC, it replaces branches and *jump and link* instructions by sequences of instructions, thereby introducing some overhead. For the Viterbi algorithm, 1,193 additional cycles were necessary, a 13.59% increase in the computation time. However, depending on the use of those specific instructions, other applications may behave differently.

Concerning the area overhead, the circular register file used by the I-IP to keep the addresses to be later compared with the PC contents is responsible for most of it. Because our technique has been designed to be non-intrusive, the I-IP must store locally all information needed to check the control flow from outside the target processor.

As opposed to the performance overhead, the area overhead is application independent, which means that the area overhead figures in Table 1 will remain the same for any other application.

5. Conclusions and future work

This paper proposes the use of an infrastructure IP as a means to detect control flow errors caused by transient faults affecting the program counter of the core processor in a SoC.

The technique is non-intrusive, both from the core processor hardware and application software standpoints, and in our experiments has been able to detect all errors caused by SEUs affecting the program counter bits of a pipelined MIPS processor executing a benchmark application, with small performance and area overheads, when compared with most of the alternative techniques discussed in Section 2.

As to the fault detection capability, the proposed technique has shown to perform better than related ones, with an error detection coverage of 100% for the benchmark application used in the tests, and with only one non detectable situation, which has an almost negligible probability of occurrence.

The next steps in this project 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.

6. References

[1] Mahmood, A., and McCluskey, E. J., "Concurrent Error Detection Using Watchdog Processors – a Survey", IEEE Transactions on Computers, vol. 37, no. 2, IEEE Computer Society, New-York-London, February 1988, pp 160-174.

[2] Lisbôa, C. A. L., Carro, L., Sonza Reorda, M., and Violante, M. "Online Hardening of Programs against SEUs and SETs", in *Proceedings of the 21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems - DFT 2006*, IEEE Computer Society, Los Alamitos, CA, October 2006, pp. 280-288.

[3] Torellas, S., Nicolescu, B., Velazco, R., Valderas, M. G., and Savaria, Y., "Validation by fault injection of a Software Error Detection Technique dealing with critical Single Event Upsets", in Proceedings of the 7th IEEE Latin-American Test Workshop (LATW 2006), Evangraph, Porto Alegre, RS, Brasil, March 2006, pp. 111-116.

[4] B. W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems: Solutions Manual*, Addison-Wesley Publishing Company, Reading, MA, October 1994.

[5] Austin, T., "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design". In *MICRO32 - Proceedings of the 32nd ACM/IEEE International Symposium on Microarchitecture*, pages 196-207, Los Alamitos, CA, November, 1999.

[6] Breveglieri, L, Koren, I, and Maistri, P., "Incorporating Error Detection and Online Reconfiguration into a Regular Architecture for the Advanced Encryption Standard". In *Proceedings of the 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems - DFT 2005*, IEEE Computer Society, Los Alamitos, CA, October 2005, pp. 72-80.

[7] K. H. Huang, J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations", *IEEE Transactions on Computers*, vol. 33, December 1984, pp. 518-528.

[8] N. Oh, P.P. Shirvani, and E.J. McCluskey. "Control flow Checking by Software Signatures", *IEEE Transactions on Reliability*, Vol. 51, No. 2, March 2002, pp. 111-112.

[9] Goloubeva, O., Rebaundengo, M., Sonza Reorda, M., and Violante, M., "Soft Error Detection Using Control Flow Assertions", in Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance (DFT 2003), IEEE Computer Society, Los Alamitos, CA, November 2003, pp. 581-588.

[10] P. Bernardi, L. M. V. Bolzani, M. Rebaudengo, M. Sonza Reorda, F. L. Vargas, M. Violante. "A New Hybrid Fault Detection Technique for Systems-on-a-Chip", *IEEE Transactions on Computers*, Vol. 55, No. 2, February 2006, pp. 185-198.

[11] M. Namjoo, "CERBERUS-16: An architecture for a general purpose watchdog processor", in *Proceedings of the 13th International Symposium on Fault-Tolerant Computing (FTCS-13)*, 1983, pp. 216-219.

[12] K. Wilken, J.P. Shen, "Continuous signature monitoring: low-cost concurrent detection of processor control errors", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 9, No. 6, June 1990, pp. 629-641.

[13] J. Ohlsson, M. Rimen, "Implicit signature checking", in *Digest of Papers of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, 1995, pp. 218-227.

[14] Miremadi, G., and Torin, J., "Evaluating Processor Behavior and Three Error-Detection Mechanisms Using Physical Fault-Injection", IEEE Transactions on Reliability, vol. 44, no. 3, IEEE Computer Society, New-York-London, September 1995, pp 441-454.

[15] Alkhalifa, Z., Nair, V. S. S., Krishnamurthy, N., and Abraham, J. A., "Design and Evaluation of System-Level Checks for On-line Control Flow Error Detection", IEEE Transactions on Parallel and Distributed Systems, vol. 10, no. 6, IEEE Computer Society, New-York-London, May-June 1999, pp 627-641.

[16] A. C. S. Beck Fº, J. C. B. Mattos, F. R. Wagner, and L. Carro, "CACO-PS: A General Purpose Cycle-Accurate Configurable Power-Simulator", in Proceedings of the 16th Brazilian Symposium on Integrated Circuits and Systems Design (SBCCI 2003), Sep. 2003.

[17] Nicolescu, B., Savaria, B., and Velazco, R., "Software detection mechanisms providing full coverage against single bit-flip faults". IEEE Transactions on Nuclear Science, vol. 51, no. 6, part 2, IEEE Computer Society, New-York-London, May-December 2004, pp 3510-3518.

A Low-SER Efficient Core Processor Architecture for Future Technologies

E. L. Rhod, C. A. Lisbôa, L. Carro

Universidade Federal do Rio Grande do Sul
Escola de Engenharia and Instituto de Informática
Porto Alegre, RS, Brazil
eduardo.rhod@ufrgs.br, calisboa@inf.ufrgs.br, carro@inf.ufrgs.br

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 have started to be investigated by the community, the full use of future resources in circuits tolerant to SETs, without performance, area or power penalties, is still an open research issue. This paper introduces MemProc, an embedded core processor with extra low SER sensitivity, and with no performance or area penalty when compared to its RISC counterpart. Central to the SER reduction are the use of new magnetic memories (MRAM and FRAM) and the minimization of the combinational logic area in the core. This paper shows the results of fault injection in the MemProc core processor and in a RISC machine, and compares performance and area of both approaches. Experimental results show a 29 times increase in fault tolerance, with up to 3.75 times in performance gains and 14 times less sensible area.

1. Introduction

Previously a concern only for mission critical applications, errors due to the effects of transient pulses produced by radiation and other interferences, called *soft errors*, are now being generally considered by the design community, since these errors are very likely to occur in future technologies. While successful mitigation techniques, and new memory technologies such as MRAM and FRAM, have already been devised to protect memories against soft errors, the protection of combinational logic, mainly against multiple simultaneous upsets, is a relatively recent concern and still lacks efficient solutions [1].

Due to the variability of their vulnerability periods, the SER of combinational logic is harder to quantify, and so far the mitigation of soft errors in those circuits has been dealt with through redundancy and larger transistor architectures, with obvious costs in area, power and even performance. The technology evolution towards nanoscale leads to the possibility of manufacturing chips with up to 10^{12} devices. Not only the number of transistors, but also the speed of the circuits has increased with the advent of deep sub-micron technology. All together, the result is a higher sensitivity of combinational logic to soft errors. As shown in Figure 1, from [2], while the SER of SRAM memories remains almost stable with technological scaling, the SER of logic has been always increasing.

For future technologies, solutions that impose redundancy or larger areas impair the ability to explore the advantages of the technology evolution. Therefore, new paradigms must be adopted in the design of combinational circuits to be manufactured using those technologies.

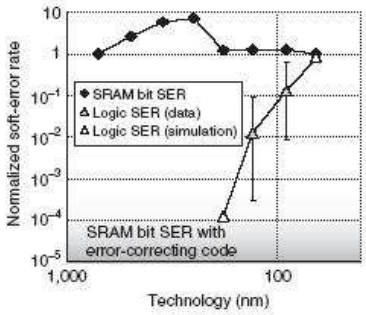


Figure 1. Evolution of SER: SRAM vs. logic [2]

Geometric regularity and the extensive use of regular fabrics is being considered as a probable solution to cope with parameter variations and improve the overall yield in manufacturing with future technologies. Together with the reduction of the cost of masks, regularity allows the introduction of spare rows and columns that can be activated to replace defective devices [3].

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 [2], brings back the concept of using memory to perform computations. Already proposed in the past [4], but precluded as a general purpose solution due to poor performance and high cost, the use of memory now is proposed here as a novel mitigation technique for transient faults, by reducing the area of the circuits that can be affected by soft errors.

In this paper, we try to cope with the SEU/SET problem without imposing area or performance overhead, at the same time that we favor a regular architecture that can be used to enhance yield in future manufacturing processes. We introduce a memory-based embedded core processor architecture, named MemProc, designed for use in control domain applications as an embedded microcontroller. It is a microcoded multicycle core processor that uses a reduced combinational logic and some extra memory to reduce the incidence of soft errors. Our technique reduces the area of sequential logic, which is sensible to faults, by using intrinsically protected memories.

The performance was evaluated by running in MemProc different applications selected from the targeted domain and comparing the results with those obtained using a pipelined RISC architecture.

This paper is organized as follows: section 2 discusses related work and highlights the differences between the proposed architecture and other alternatives. Section 3 describes the MemProc architecture, explaining how its simplified ALU works and which are the main reasons for the good performance results. Section 4 describes the fault injection process, and presents simulation data that confirms the superiority of MemProc concerning fault rates, as well as the metrics of MemProc in terms of circuit area and performance, in comparison to the pipelined RISC core processor. In section 5 we comment the achieved results and also future work.

2. Related Work

The reliability of circuits manufactured in future technologies became a major topic of discussion and research in recent years [5, 6], imposing tolerance to transient faults as a mandatory design concern.

Among different approaches to cope with soft errors found in the literature, the use of spatial or time redundancy dominates as the major technique.

The use of time redundancy to avoid undesirable errors, exploiting microarchitectural techniques that are already incorporated in the processor due to performance reasons, has been proposed in [7], and a penalty of up to 30% in performance is incurred. The use of simultaneous multithreading to detect transient faults is also proposed in [8]. The area cost of such duplication techniques is obviously high.

In [9], a self-repairing unit for microprogrammed processors is proposed. In that work, the authors used a dedicated built-in self-test (BIST) architecture to provide an online status – either good or faulty – for each block in the execution unit. For each processor microinstruction, they defined a sequence of microinstructions that can execute the same operation using only fault-free units. This approach has a significant area and performance overhead due to the BIST and fault-free units added to the circuit.

In [10], the authors propose the use of a self-stabilizing microprocessor to cope with any combination of soft errors. The paper presents only the initial studies of the behavior of the self-stabilizing processor in the presence of soft errors. Whenever affected by a transient fault, the processor is able to converge to a safe state, from which the normal fetch-decode-execute sequence can be resumed during fault-free periods. Besides presenting the design scheme for the processor, a new technique for the analysis of the effects of soft errors is introduced, which instead of using simulation is based in an upper bound algorithm that does not take into account the fault masking effects of the circuit.

The use of memory as a computing device, has been subject of research in the past. In order to explore the large internal memory bandwidth, designers proposed to bring some functions executed by the processor into memory [4]. This technique apparently has been discarded due to its limited field of application.

Back to the fault tolerance arena, another strong argument to the use of memory to perform computation functions is its intrinsic protection against defects, due to the use of spare columns and spare rows, such as in DRAMs. More recently, the fact that new memory technologies, such as ferroelectric RAMs (FRAMs), magnetic RAMs (MRAMs), and flash memories, are virtually immune to soft errors, due to their physical characteristics [2], makes those types of memories an important additional resource for the implementation of fault tolerant systems. MRAMs are also more energy efficient than other non-volatile memory technologies, since they consume less power during read and write operations [11].

Since memories are regular structures by nature, memory systems will also benefit from the foreseen advantages that regular fabrics will provide for future technology.

The fact that the proposed MemProc processor relies heavily in the use of memories adds the benefits arising from regularity and immunity against soft errors to the solution proposed in this paper. In order to highlight the fault tolerance of the design, we injected faults and compared the results with those

obtained for another core processor (MIPS), using the same simulation tool. In this process, two implementations of each architecture running in parallel have been simulated and faults have been injected in one of them, comparing the produced results for each possible single event transient occurrence. Therefore, all the possible fault incidence cases have been considered, even those in which the faults are masked by the architecture and do not generate errors.

3. The Architecture of MemProc

The architecture proposed in this paper is a microcoded multicycle 16-bit core processor with Harvard architecture, in which part of the datapath has been replaced with memory, thereby reducing the amount of combinational logic. In Figure 2(a) the main functional blocks of the proposed architecture are shown.

The application code, which is also called the *macroinstrucion code*, is stored in the ROM memory. The instructions in this code, as usual, indicate the operations to be performed and their operands. The *microcode memory* receives the initial microcode address of 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.

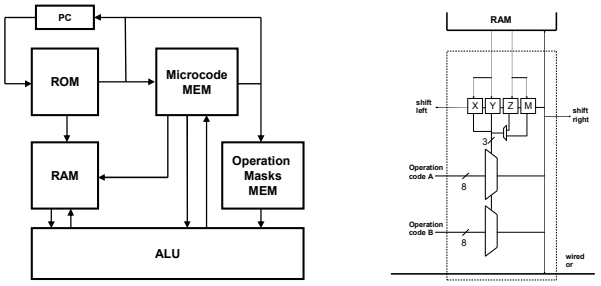


Figure 2. (a) MemProc architecture (b) ALU for one bit operation (c) 2-bit addition using MemProc ALU

In the MemProc ALU, operations are performed 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 2(b) depicts a MemProc ALU block for processing 1-bit operands.

The complete MemProc ALU is 16-bit wide and their 16 blocks work in parallel, being able to perform bit serial arithmetic and logic operations. All operation mask values are independent from each other, so each processing element of the ALU can perform a different Boolean function. To accelerate addition operations, we use two 8:1 multiplexors instead of a single one; one multiplexor is used to calculate the sum and the other to calculate the carry out. An extra flip-flop, called "M", was also added, to accelerate multiplications.

In figure 2(c) the addition of two 1-bit operands is used to illustrate how the ALU works. We can see from the truth table the operation masks for the "sum" and the "cout" (carry out) outputs of the multiplexors. Also in figure 2(c), we can se the presence of a wired-or bus. This bus implements an "or" operation of all the multiplexors' outputs. This wired-or bus is an extremely important element in what we call "compute only the necessary to get the result", which will be discussed in the following paragraph.

The way MemProc achieves its high performance is based on the fact that it computes just the necessary cycles to get the operation result. In traditional computer architectures, the ALU does its arithmetic and logic operations using combinational hardware that always takes the same time to compute the operation, regardless of the value of the operands. MemProc executes only the number of cycles required to get the result, depending on the carry propagation chain.

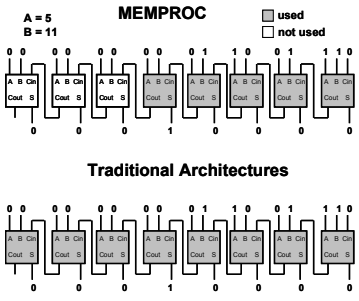


Figure 3. 8-bit Addition paradigm.

In Figure 3 we can see that MemProc requires only 5 of the 8 operation units to perform the addition of two 8-bit operands, which means that it takes 5/8 of the time required by traditional architectures to perform this operation. 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 has 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. On the other hand, store operations require only 2 cycles.

In multiplications, the number of cycles depends on the number of bits equal to zero in the operands. The number of required cycles decreases as the number of bits equal to zero in the operands increases. One could say that if the values of the operands are high the proposed approach would not have any advantage. However, as shown in [12], the transition activity for some multimedia benchmarks is more intense in the 8 least significant bits.

4. Experimental Results: fault tolerance, area and performance metrics

The fault rate of a circuit, also known as soft error rate, can be expressed by the amount of errors that affect the circuit in a certain period of time.

The soft-error rate of a design can also be expressed by the nominal soft-error rate of the individual circuit elements that compose the design, like memory structures such as SRAMs, sequential elements such as flip-flops and latches, combinational logic and its architectural and timing vulnerability characteristics [13, 14] as follows:

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

where i represents the i^{th} element of the design.

The $SER^{nominal}$ for the i^{th} 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, which will cause an error in the i^{th} element. The AVF_i, architectural vulnerability factor (also known as logic derating) represents the probability that and error in the i^{th} element will cause a system-level error. In this study the time vulnerability factor was not taken into account [13, 14].

One of the most usual ways to measure the SER of a circuit is evaluating the number of Failures in Time (FIT), which means one error every 10^9 hours [13, 14]. A soft error rate of 10 FIT means that the device will generate 10 errors in 1 million years. Another commonly used metric to express SER is the Mean Time to Failure (MTTF). As an example, a MTTF of 1000 hours means that, in average, one error occurs after 1000 hours of device operation. FIT and MTTF are inversely related, i.e., less FIT means better SER, while higher MTTF means better SER [15]. In this paper we use the MTTF metric to measure the fault tolerance of the proposed architecture and MIPS.

In order to evaluate the feasibility of the architecture proposed in this paper, both in terms of fault tolerance, area, and performance, extensive simulations have been executed, using an in-house developed simulation tool named CACO-PS (a System C-like simulator) [16]. The comparisons have been made against the well-known MIPS 16-bit RISC architecture, with a 5-stage pipeline and forwarding unit [17], widely used in real-world embedded processors.

4.1 Fault Rate Evaluation

To evaluate the fault rate of the processors, random faults were injected in both MIPS and MemProc 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 MIPS processor, the units to be checked are the program counter, in order to detect wrong branches, and wrong data or address values 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 contents were checked to identify SDC.

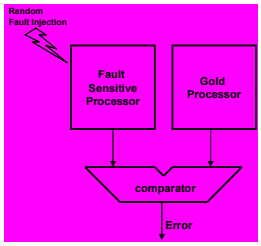


Figure 4. Error Detection Scheme

Figure 4 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.

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, we have implemented the random fault injector following this probabilistic fault behavior. To do so, we have created a file with all the important information about the components, such as component size, number of outputs and outputs widths. Then, 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 in every cycle. In this work, we decided to use a technique called environmental acceleration [18], 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 every 2 cycles in the MIPS processor, which is indeed a high rate assumption. To calculate the corresponding number of faults per cycle for the MemProc processor, we have used the Leonardo Spectrum tool [19] to generate area and timing information from the VHDL descriptions of both processors. Table 1 shows those results and the corresponding time gap between faults for both processors. As one can see, this time gap is inversely proportional to the number of SET sensible gates of the processors, showing that the lower the sensible area of a circuit, the lower is the probability of a particle hit affecting that circuit.

As shown in Table 1, the combinational circuit (# of sensible gates) in the MemProc architecture is very small when compared to the size of its memory elements. In our approach, the memory elements are considered to be immune to soft errors, since we are supposing the use of new memory technologies, such as MRAM, FRAM, and flash memories, as mentioned before. In order to allow a fair comparison, during the fault injection process all memory elements of both architectures have been considered immune to soft errors.

Table 1. Area and number of faults per cycle.

Architecture	MemProc	MIPS
ROM (bits)	1,792	2,720
RAM (bits)	512	512
Op. Masks Mem. (bits)	128 x 256	-×-
Microcode Mem. (bits)	1024 x 68	-×-
# of sensible gates	679	9,619
Frequency (MHz)	254	54
time between faults (ns)	523.62	37.04

The fault injection process injected random faults according to the probability of the component being hit and also the calculated number of faults per cycle. In this process, faults were injected until one error or a silent data corruption (SDC) was detected, in order to determine the time to failure. This process was repeated 100 times, and the mean time to failure calculated as the average time to failure in the 100 experiments. The results are shown in Table 2, which lists the fault injection results for the MemProc and MIPS processors.

Table 2. Fault rates for both architectures.

Architecture	MemProc	MIPS
# of cycles	585,945	4,320
# of injected faults	4,404	2,160
# of errors + SDCs	100	100
MTTF (μs)	23.068	0.798

The first line of Table 2 shows the number of cycles each processor had to execute until 100 errors or SDCs were detected. The second line presents the number of faults injected during the process. The third line shows the total number of errors and SDCs that occurred during this process. The fourth line shows the corresponding Mean Time To Failure value, showing that the MTTF of the MemProc architecture is almost 29 times bigger than the MIPS's one. These results show the significant reduction in the MTTF that can be obtained by using the proposed architecture.

4.2 Performance Evaluation

The evaluation of the MemProc performance was made using a cycle accurate simulation tool (CACO-PS [16]) to measure the number of cycles taken by the proposed architecture while executing a set of benchmarks.

Using the description language of CACO-PS, which is similar to System C, both MemProc and MIPS architectures were described and simulated. The performance evaluation was done using four different application programs, with different processing characteristics: three sort algorithms and the IMDCT (Inverse Modified Discrete Cosine Transform, part of the MP3 coding/decoding algorithm) function, which were executed both in MemProc and MIPS. Those applications have been selected because are widely used in the target domain (control applications), and also because they use most of the operations implemented in the MemProc instruction set.

The maximum frequency of operation for both architectures was evaluated using VHDL descriptions, and the Leonardo Spectrum tool.

The obtained results are shown it Table 3, in which we can see that MemProc executes the bubble sort algorithm in approximately 4.7 thousand cycles, while MIPS takes half this number of cycles to perform the same. 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 that of MIPS. On the other hand, the critical path of MemProc is determined by the access time of the microcode memory, while in MIPS the critical path is determined by the multiplier delay. So, the maximum frequency of MemProc is more than 4 times higher than that of MIPS, and, as consequence, the MemProc is almost 3 times faster than MIPS running the sort algorithms.

Table 3. Performance when executing benchmark applications

Application	MIPS (54 MHz)	
	# of Cycles	Computation Time (μs)
Bubble Sort	2,280	42.2
Insert Sort	1,905	35.3
Select Sort	1,968	36.4
IMDCT	38,786	718.3

Application	MemProc (254 MHz)		Perform. Ratio
	# of Cycles	Computation Time (μs)	
Bubble Sort	4,720	18.4	2.29
Insert Sort	2,508	9.8	3.60
Select Sort	2,501	9.7	3.75
IMDCT	142,961	562.8	1.28

The analysis of the results when executing IMDCT shows that MemProc was only 1.28 times faster. That happens because this algorithm uses 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 multicycle machine, while MIPS is a pipelined one, which is expected to be faster than its multicycle version. So, we can conclude that if we were comparing MemProc with MIPS multicycle version, performance results would be even better. Also, the performance gains of MemProc comes from the fact that the number of cycles it takes to perform an operation depends both on the operation and on the values of the operands. For instance, let us consider that MIPS needs 1 cycle to perform an add operation. Since the frequency of MemProc is almost 5 times higher, if the operands are such that the number of carry cycles are less than 5, MemProc will finish the addition operation earlier than MIPS. Also, store operations take only 2 cycles in MemProc, which is more than 2 times faster than in MIPS.

In order to stress that the primary goal of the proposed technique is fault tolerance, and not performance, the execution of the IMDCT application has been executed once again, this time with MemProc running at 198.44 MHz, which gives the same computation time for both MIPS and MemProc running that application. The fault injection process was then repeated for MemProc running at that frequency and the MTTF has been recomputed. The resulting MTTF was 18.7 μs, which is still more than 23 times longer than that of MIPS. This confirms our claim, that the approach proposed in this paper is well suited to a higher reliability embedded processor.

5. Conclusions and Future Work

This work proposes a novel fault tolerant architecture for embedded core processors for use in control applications, which uses microcoded memory to execute macroinstructions, and uses as ALU sixteen 8:1 multiplexors to perform all logic and arithmetic operations.

Simulation results have shown that the Mean Time to Failure of the proposed architecture is more than 29 times longer than the MIPS one, due to the reduction of the area sensitive to faults, without having any performance degradation, on the contrary, with improved performance. Also, results showed that, despite requiring several cycles to execute its bit serial operations, MemProc was 1.28 times faster than MIPS. While the main goal of this work was to propose a new fault tolerant architecture, the performance gains come from the fact that MemProc exploits the benefits of using bit serial operations, and differently from MIPS, it can require less cycles to make the same operation, depending on the value of the operands.

The proposed architecture, while not being a final solution, reflects our 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, while providing much more reliability against transient faults. In order to stress that fault tolerance is the major goal of this work, a lower frequency version of MemProc, which delivers exactly the same performance of the alternative architecture for the sample application, has been used in one experiment.

6. References

[1] Rossi, D., Omaña, M., Toma, F. and Metra, C., "Multiple Transient Faults in Logic: An Issue for Next Generation ICs ?", in Proceedings of th 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2005), pp. 352-360, IEEE Computer Society, Los Alamitos, CA, Oct '05.

[2] Baumann, R., "Soft Errors in Advanced Computer Systems", IEEE Design and Test of Computers, vol. 22, no. 3, pp 258-266, IEEE Computer Society, May-June 2005.

[3] Sherlekar, D., "Design Considerations for Regular Fabrics", in Proceedings of the 2004 International Symposium on Physical Design (ISPD 2004), pp. 97-102.

[4] Elliott, D.G., Stumm, M., Snelgrove, W.M., Cojocaru, C., Mckenzie, R., "Computational RAM: implementing processors in memory", Design & Test of Computers, IEEE, vol. 16, no. 1, pp. 32-41, IEEE Computer Society, Jan/Mar 1999.

[5] Constantinescu, C., "Trends and Challenges in VLSI Circuit Reliability", IEEE Micro, vol. 23, no. 4, pp. 14-19, IEEE Computer Society, New York-London, July-August 2003.

[6] Semiconductor Industry Association. International Technology Roadmap for Semiconductors – ITRS 2005, last access July, 2006. http://www.itrs.net/Common/2005ITRS/Home2005.htm.

[7] Rotenberg, E., "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," in Digest of Papers of the 29th Annual International Symposium on Fault-Tolerant Computing, pp. 84-91, IEEE Computer Society, New York-London, 1999, ISBN: 0-7695-0213-X.

[8] Reinhardt, S. K., and Mukherjee, S. S., "Transient Fault Detection via Simultaneous Multithreading," in Proceedings of

the 27th Annual International Symposium on Computer Architecture (ISCA 2000), pp. 25-36, ACM Press, May 2000.
[9] Benso, A.; Chiusano, S.; Prinetto, P.,"A self-repairing execution unit for microprogrammed processors", in IEEE Micro, vol. 21, issue 5, pp. 16-22, IEEE Computer Society, New York-London, sept-oct 2001.
[10] Dolev, S.; Haviv, Y.A., "Self-Stabilizing Microprocessor: Analyzing and Overcoming Soft Errors" in IEEE Transactions on Computers, vol. 55, no. 4, pp. 385-399, IEEE Computer Society, New York-London, April 2006, ISSN: 0018-9340.
[11] Tehrani, S. et al., "Magnetoresistive Random Access Memory using Magnetic Tunnel Junctions", in Proceedings of the IEEE, vol. 91, no. 5, pp 703-714, IEEE Computer Society, London-New York, May 2003. ISSN: 0018-9219.
[12] Ramprasad, S., Shanbhag, N. R., Hajj, I. N., "Analytical Estimation of Transition Activity from Word-level Signal Statistics", in Proc. of the 34th Design Automation Conference (DAC'97), pp. 582-587, IEEE Comp. Soc., June 1997.
[13] H.T. Nguyen and Y. Yagil, "A Systematic Approach to SER Estimation and Solutions," Proc. IEEE Int'l Reliability Physics Symp., IEEE Press, 2003, pp. 60-70.
[14] N. Seifert and N. Tam, "Timing Vulnerability Factors of Sequentials," IEEE Trans. Device and Materials Reliability, V4, N3, Sept. 2004, pp. 516-522.
[15] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The Soft Error Problem: An Architectural Perspective," in Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA), pp. 243-247, IEEE Computer Society, Los Alamitos, CA, Feb. 2005, San Francisco.
[16] A. C. S. Beck Fº, J. C. B. Mattos, F. R. Wagner, and L. Carro, "CACO-PS: A General Purpose Cycle-Accurate Configurable Power-Simulator", in Proceedings of the 16th Brazilian Symposium on Integrated Circuits and Systems Design (SBCCI 2003), Sep. 2003.
[17] Patterson, D.A., and Hennessy, J. L.. Computer Architecture: a Quantitative Approach, 3rd Edition, Elsevier Science & Technology Books, June 2002. ISBN: 1558605967.
[18] Mitra, S., Seifert, N., Zhang, M., Shi, Q., Kim, K.S., "Robust system design with built-in soft-error resilience", in Computer, vol. 38, issue 2 pp. 43-52, feb 2005.
[19] Last access: July, 2006. http://www.mentor.com/products/ fpga_pld/synthesis/leonardo_spectrum/.

J Electron Test
DOI 10.1007/s10836-007-5018-2

Hardware and Software Transparency in the Protection of Programs Against SEUs and SETs

Eduardo Luis Rhod · Carlos Arthur Lang Lisbôa · Luigi Carro · Matteo Sonza Reorda · Massimo Violante

Received: 27 November 2006 / Accepted: 18 June 2007

Abstract Processor cores embedded in systems-on-a-chip (SoCs) are often deployed in critical computations, and when affected by faults they may produce dramatic effects. When hardware hardening is not cost-effective, software implemented hardware fault tolerance (SIHFT) can be a solution to increase SoCs' dependability, but it increases the time for running the hardened application, as well as the memory occupation. In this paper we propose a method that eliminates the memory overhead, by exploiting a new approach to instruction hardening and control flow checking. The proposed method hardens an application online during its execution, without the need for introducing any change in its source code, and is non-intrusive, since it does not require any modification in the main processor's architecture. The method has been tested with two widely used architectures: a microcontroller and a RISC processor, and proven to be suitable for hardening SoCs against transient faults and also for detecting permanent faults.

Keywords SEU · SET · Infrastructure IP · Instruction hardening

Responsible Editor: N. A. Touba

E. L. Rhod (✉)
Escola de Engenharia,
Universidade Federal do Rio Grande do Sul,
Porto Alegre, Brazil
e-mail: eduardo.rhod@ufrgs.br

C. A. L. Lisbôa · L. Carro
Instituto de Informática,
Universidade Federal do Rio Grande do Sul,
Porto Alegre, Brazil

C. A. L. Lisbôa
e-mail: calisboa@inf.ufrgs.br

L. Carro
e-mail: carro@inf.ufrgs.br

M. Sonza Reorda · M. Violante
Dipartimento di Automatica e Informatica, Politecnico di Torino,
Torino, Italy

M. Sonza Reorda
e-mail: matteo.sonzareorda@polito.it

M. Violante
e-mail: massimo.violante@polito.it

1 Introduction

The introduction of new semiconductor technologies is making possible the implementation of very powerful computer systems: constantly shrinking feature sizes allow the integration of more functions in a single chip, while a higher clock frequency significantly increases the number of operations that can be performed per time unit. Although designers can exploit all these benefits for devising highly efficient systems, known as systems-on-a-chip (or SoCs), those new technologies are very sensitive to soft errors, induced either by particles strikes, or by other types of interferences. Therefore, when a SoC is intended for safety or mission-critical applications, designers must guarantee that soft errors have negligible impact on its behavior.

SoCs are often designed resorting to intellectual property (IP) cores. IP cores are usually guaranteed only to function correctly (since debugged and validated by IP vendors), while their correct behavior in presence of soft errors is normally not guaranteed. Therefore, it is up to the designers of safety or mission-critical SoCs to guarantee that their systems are hardened against soft errors.

As far as processor cores are concerned, a possible approach to guarantee dependability lies in the adoption of

the so-called Software Implemented Hardware Fault Tolerance (SIHFT) techniques [9, 10, 17]. They are based on modifying the software executed by the processor (or controller), introducing some sort of redundancy, so that errors are detected before they become failures. SIHFT techniques are characterized by their ease of use, however, their adoption is often limited by the high overhead they introduce, both in terms of memory, and of performance.

Memory occupation increases due to the additional information the software has to deal with, and the extra instructions to process them. Performance degradation arises from the execution of redundant instructions inserted in the software [9, 10, 17].

A hybrid approach, that minimizes the introduced overhead by combining software modifications with a special-purpose hardware module (known as infrastructure IP core, or I-IP core) is presented in [5].

SIHFT, and even its more optimized hybrid implementation, may increase memory occupation by a factor ranging from 2 up to 7. As to redundant instructions, they are intrinsic to the concept of SIHFT and cannot be avoided; they can only be minimized, as shown in [5], but at a cost of additional silicon area occupation.

In some types of applications, neither SIHFT nor its hybrid version are applicable, because of stringent costs or power budget. Moreover, when the SoC runs commercial off-the-shelf software components, SIHFT is simply not applicable, since the source code is not available. In this case, alternative techniques are needed for providing the system with an adequate level of dependability.

In this paper we propose a novel technique to cope with errors that may affect a processor core, which combines ideas from SIHFT and hybrid techniques, and introduces the novelty of *hardware and software transparency*. The technique exploits instruction hardening and consistency check by performing twice the computation of selected instructions, and controlling the consistency of the attained results. The main novelties of this work are that the designer is freed from the burden of modifying the source code of the application running on the main processor core, and that the implementation of the I-IP is non-intrusive as far the main processor architecture is concerned. Indeed, an I-IP located close to the main processor takes care of on-the-fly instruction hardening, consistency checks, and control flow checking, by transparently modifying the sequence of instructions fetched by the processor. The I-IP constantly monitors the instructions fetched by the main processor, and each time a data processing instruction is recognized, it generates a sequence of instructions that are passed to the main processor, executes itself the data processing instruction and compares the results with what is computed by the main processor. The I-IP also recognizes the branch instructions the main processor fetches, and checks the correctness of the control flow execution.

Several advantages stem from this approach. Since the source code of the application running on the processor core is no longer needed, commercial off-the-shelf software components can be hardened, as well. Since the application running on the main processor is hardened on-the-fly, no additional instructions need to be stored and information redundancy is not necessary. Designers only have to guarantee the consistency of the information the SoC stores in memory, for example by resorting to cost-effective codes like parity.

The approach is scalable, allowing the designers to trade the area overhead and the performance degradation the I-IP introduces for the attained dependability level. Designers decide which of the instructions the processor executes should be replicated on-the-fly by the I-IP, thus minimizing the amount of hardware resources needed to implement the I-IP. Finally, this approach can be used to detect the occurrence of transient faults (SEUs affecting the processor's memory elements, and SETs affecting the processor's combinational logic during normal operation of the SoC), and also permanent faults (during manufacturing final tests).

The paper is organized as follows. Section 2 presents an overview of the already available approaches to harden a processor-based SoC. Section 3 details the approach presented in this paper, while Section 4 presents an experimental analysis of the approach using two of the most popular architectures for SoC core processors: one microcontroller (the Intel 8051 microcontroller, which has been subject of our previous work [11]), and one RISC processor (a 16-bit pipelined MIPS processor). Finally, Section 5 draws some conclusions.

2 Previous Work

Error detection techniques for software-based systems can be organized in three broad categories: software-implemented techniques, which exploit purely software detection mechanisms, hardware-based ones, which exploit additional hardware, and hybrid ones, that combine both software and hardware error detection mechanisms. Such techniques focus on checking the consistency between the expected and the executed program flow, recurring to the insertion of additional code lines or by storing flow information in suitable hardware structures, respectively.

SIHFT techniques exploit the concepts of information, operation, and time redundancy to detect the occurrence of errors during program execution. In the past five years some techniques have been developed that can be automatically applied to the source code of a program, thus

simplifying the task for software developers: the software is indeed hardened by construction, and the development costs can be reduced significantly. Moreover, the most recently proposed techniques are general, and thus they can be applied to a wide range of applications.

Techniques aiming at detecting the effects of faults that modify the expected program's execution flow are known as *control flow checking* techniques. These techniques are based on partitioning the program's code into basic blocks (sequences of consecutive instructions in which, in the absence of faults, the control flow always enters at the beginning and leaves at the end) [1].

Among the most important solutions based on the notion of basic blocks proposed in the literature, there are the techniques called *Enhanced Control Flow Checking using Assertions* (ECCA) [2] and *Control Flow Checking by Software Signatures* (CFCSS) [17].

ECCA is able to detect all the 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 on a conditional branch. CFCSS cannot cover control flow errors if multiple nodes share multiple nodes as their destination nodes. As far as faults affecting program data are considered, several techniques have been recently proposed that exploit information and operation redundancies [6, 16]. The most recently introduced approaches modify the source code of the application to be hardened against faults by introducing information redundancy and instruction duplication. Moreover, consistency checks are added to the modified code to perform error detection. The approach proposed in [6] exploits several code transformation rules that mandate for duplicating each variable and each operation among variables. Moreover, each time a variable is read, a consistency check between the variable and its replica should be performed.

Conversely, the approach proposed in [16], 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 unmodified program. After executing both the original and the modified versions, their results are compared: an error is detected if any mismatch is found.

Both approaches introduce overheads in memory and execution time. By introducing consistency checks that are performed each time a variable is read, the approach proposed in [6] minimizes the latency of faults; however, it is suitable for detecting transient faults only, since the same operation is repeated twice. Conversely, the approach proposed in [16] exploits diverse data and duplicated instructions, and thus it is suitable for both transient and permanent faults. As a drawback, its fault latency is generally greater than in [6]. The ED^4I technique requires a careful analysis of the size of used variables, in order to avoid overflow situations.

SIHFT techniques are appealing, since they do not require modification of the hardware running the hardened application, and thus in some cases they can be implemented with low costs. However, although very effective in detecting faults affecting both program execution flow and program data, the software-implemented approaches may introduce significant time overheads that limit their adoption only to those applications where performance is not a critical issue. Also, in some cases they imply a non-negligible increase in the amount of memory needed for storing the duplicated information and the additional instructions. Finally, these approaches can be exploited only when the source code of the application is available, precluding its application when commercial off-the-shelf software components are used.

Hardware-based techniques exploit special purpose hardware modules, called *watchdog processors* [12], to monitor the control flow of programs, as well as memory accesses. The behavior of the main processor running the application code is monitored using three types of operations.

Memory access checks consist in monitoring for unexpected memory accesses executed by the main processor, such as in the approach proposed in [15], where the watchdog processor knows at each time during program execution which portion of the program's data and code can be accessed, and activates an error signal whenever the main processor executes an unexpected access.

Consistency checks of variables contents consists in controlling if the value a variable holds is plausible. By exploiting the knowledge about the task performed by the hardened program, watchdog processors can validate each value the main processor writes or reads through range checks, or by exploiting known relationships among variables [13].

Control flow checks consist in controlling whether all the taken branches are consistent with the Program Graph of the software running on the main processor [14, 18, 21, 25]. As far as the control flow check is considered, two types of watchdog processors may be envisioned. An *active watchdog processor* executes a program concurrently with the main processor. The Program Graph of the watchdog's program is homomorphic to the main processor's one. During program execution, the watchdog continuously checks whether its program evolves as that executed by the main processor or not [14]. This solution introduces minimal overhead in the program executed by the main processor; however, the area overhead needed for implementing the watchdog processor can be non-negligible. A *passive watchdog processor* does not execute any program; conversely, it computes a signature by observing the main processor' bus. Moreover, it performs consistency

checks each time the main program enters/leaves a basic block within the Program Graph. A cost-effective implementation is described in [25], where a watchdog processor observes the instructions the main processor executes, and computes a runtime signature. Moreover, the code running on the main processor is modified in such a way that, when entering a basic block, an instruction is issued to the watchdog processor with a pre-calculated signature, while the main processor executes a NOP instruction. The watchdog processor compares the received pre-computed signature with that computed at runtime, and it issues an error signal in case of mismatch. An alternative approach is proposed in [18], where the watchdog processor computes a runtime signature on the basis of the addresses of the instructions the main processor fetches. Passive watchdog processors are potentially simpler than active ones, since they do not need to embed the Program Graph, and since they perform simpler operations: signature computation can be demanded to LFSRs, and consistency checks to comparators. However, an overhead is introduced in the monitored program: instructions are indeed needed for communicating with the watchdog.

Dynamic verification, another hardware-based technique, is detailed in [3] for a pipelined core processor. It uses a "functional checker" to verify the correctness of all computation 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 processor's 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 processor's 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 core processor's pipeline is flushed, and the processor is restarted using its own speculation recovery mechanism, executing again the instruction.

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 [24].

While being a well balanced solution, in terms of area and performance impacts, the DIVA approach has two main drawbacks. First, since the checker is implemented inside the processor's pipeline, it cannot be implemented in SoCs based on COTS processors or FPGAs that have an embedded off-the-shelf processor, such as an ARM or Power PC core. Second, the fundamental assumption behind the proposed solution is that the checker never fails, due to the use of oversized transistors in its construction and also to extensive verification in the design phase. In case this is not feasible, the authors suggest the use of conventional alternatives, such as TMR and concurrent execution with comparison, which have been already studied in several other works.

Hybrid techniques (for example [5]) combine the adoption of some SIHFT techniques in a minimal version (thus reducing their implementation cost) with the introduction of an I-IP into the SoC. The software running on the processor core is modified so that it implements instruction duplication and information redundancy; moreover, instructions are added to communicate to the I-IP the information about basic block execution. The I-IP works concurrently with the main processor, it implements consistency checks among duplicated instructions, and it verifies whether the correct program's execution flow is executed by monitoring the basic block execution.

Hybrid techniques are effective, since they provide a high level of dependability while minimizing the introduced overhead, both in terms of memory occupation and performance degradation. However, in order to be adopted they mandate the availability of the source code of the application the processor core should run, and this requirement cannot be always fulfilled.

The idea of introducing an I-IP between the processor and the instructions memory, and of charging the I-IP of substituting on-the-fly the fetched code with hardened one, was preliminarily introduced in [20]. However, the I-IP proposed in [20] was much simpler (it does not include either an ALU or a control unit), and is not supported by a suitable design flow environment, as proposed here. Moreover, the performance overhead of the method in [20] was significant, and the method cannot cover permanent faults.

3 The Proposed Approach

The approach we developed 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 application's source code is not available, by exploiting the following concepts:

- *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 memory address is compared with the expected one, and an error is notified if a mismatch is detected.

In the following we describe how these concepts are implemented, firstly by stating the assumptions upon which it is based on, then by describing the overall architecture of the SoC adopting the proposed technique, and finally by detailing the I-IP architecture and the design flow to support its deployment in a SoC.

3.1 Assumptions

The system we intend to protect 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 two different well known and widely used architectures: the Intel 8051 microcontroller and a MIPS RISC processor.

We assume that a suitable I-IP core can be inserted in the SoC, able to implement instruction hardening and control flow checking. The SoC can be either implemented through an ASIC, or an FPGA embedding a processor core (such as the Xilinx's Virtex II Pro, which embeds a PowerPC core, or Actel's M7 proASIC3/E, which embeds an ARM core).

The I-IP we propose is inserted between the memory storing the code and the main processor core, and monitors each instruction fetch operation. In this work we assume that the bus connecting the instruction cache to the processor is not accessible, as it often happens for processor cores, and therefore we assume that the instruction cache either does not exist, or is disabled. While, at first sight, this assumption might be considered a weakness of the proposed approach, because cache memories play a significant role in performance improvement, it is important to highlight that the original goal of this work was to work with COTS components, in which one seldom has access to the cache-processor connection. Given that condition, one must remeber that in several applications (e.g., in the automotive market) low cost microcontrollers, which often do not have caches, are used, and our method would be perfectly tailored to this kind of cores. Moreover, there are several versions of ARM, MIPS or Texas processors in which one can actually disconnect the cache when working in a COTS project, and hence the techniques here proposed could also be used in those cases.

Moreover, we assume that the instruction memory and the data memory located outside the processor are either hardened with suitable error detection/correction codes or implemented using technologies that are intrinsically immune to soft errors. In either case, it must be noted that, in a SoC design, it is often possible to intervene on the memory global structure, by for example including extra circuitry covering error detection and correction, while it is much less common to be able to modify the processor core, which is often bought from third parties, or is reused from previous projects.

The use of error detection and correction codes is a relatively low cost technique, concerning area overhead, because the extra area required for coding grows logarithmically with the size of the memory to be protected, making this technique a natural one to be adopted for high reliability systems. A second alternative to achieve protection of the memory is the use of new memory technologies, such as magnetic RAMs (MRAMs) or flash memories, which are not sensitive to radiation effects, and therefore are intrinsically immune to soft errors [8].

In developing our technique, we adopt the SEU in the processor's memory elements as fault model. Possible fault locations are therefore the register file, and the registers not accessible through the instruction set (for example, the pipeline's boundary registers). However, the approach is general, and it can be exploited to cope with other fault models, like the single stuck-at one, or the single event transient one.

3.2 Overall Architecture

The technique we developed is based on inserting an I-IP between the processor core and the memory storing the instructions the processor core executes, as illustrated in Fig. 1. While the I-IP must be tailored to the specific core processor in a given SoC, according to the design flow described in Section 3.4, the architecture and the technique described here are generic, and can be implemented in any SoC in which additional modules can be inserted.

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

Instruction hardening and *consistency check*: the I-IP decodes the instructions the processor fetches. Each time a data processing instruction is fetched, like that in Fig. 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 Fig. 3, which is sent to the processor.

Therefore, from the point of view of the processor, the fetched instructions are no more those contained in the code memory, but those issued by the I-IP. The sequence of instructions that substitutes each data processing one includes two instructions whose purpose is to send to the I-IP the value of the source operands the instruction elaborates.

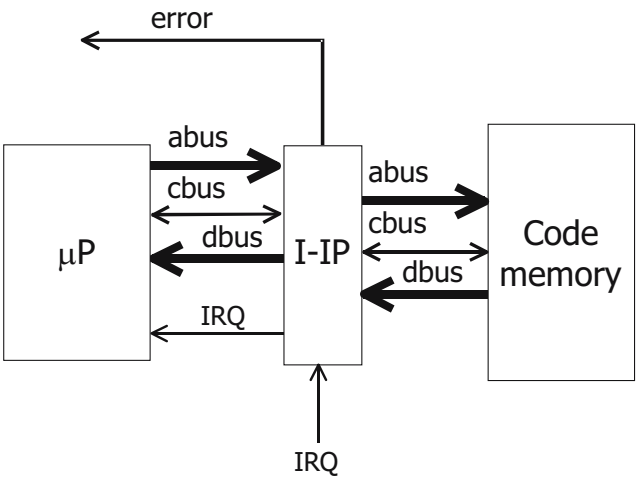


Fig. 1 Overall architecture

The third instruction (in boldface) is the original instruction coming from the program, while the fourth one is used to send the I-IP the computed result. Finally, the last instruction is used to resume the original program execution, starting from the instruction following the replicated one, which is located at address FETCH_ADX +OFFSET, being OFFSET the size of the replicated instruction.

Concurrently to the main processor, the I-IP executes the fetched data processing instructions by exploiting its own arithmetic and logic unit, and it 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.

Some additional comments, specific for the implementation of this technique with pipelined core processors, are presented in Section 4.2.

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 A_{next} in the program 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}).

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

Fig. 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
```

Fig. 3 Source operands and result fetching

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. At the fetch of the next instruction at address A', the I-IP controls if the program is proceeding along the expected control flow. If A' differs from both A_{next} and A_{taken}, the error signal is raised to indicate that an instruction located in an unexpected address has been fetched.

Control flow check is disabled when the processor fetches a return instruction that transfers the execution flow from a subroutine to its caller, or that ends an interrupt service routine. In both these cases a stack is required to implement control flow check correctly. We opted for not implementing this stack since it would increase significantly the I-IP's area occupation.

Interrupt requests are not allowed to interfere with the execution of the sequence of instructions in Fig. 3. For this reason the I-IP receives interrupt requests through the IRQ signal, and it forwards them to the processor core only after the sequence of instructions in Fig. 3 has been completely executed. This approach preserves the correct operation of the I-IP, which is able to harden interrupt service routines, too, at a cost of slightly increased interrupt latency.

3.3 The I-IP

The I-IP we developed is organized as Fig. 4 shows, and it is composed of the following modules:

1 *CPU interface*: it 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 rest of the I-IP.

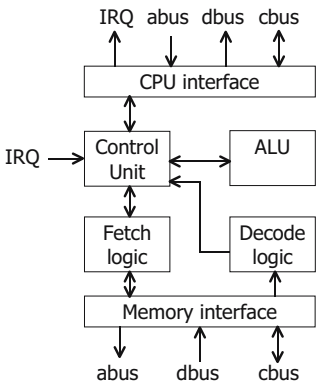


Fig. 4 Architecture of the I-IP

2 *Memory interface*: it connects the I-IP with the memory storing the application the processor executes. This module executes commands coming from the *Fetch logic* and handles the details of the communication with the memory.
3 *Fetch logic*: it issues to the *Memory interface* the commands needed for loading a new instruction in the I-IP and feeding it to the *Decode logic*.
4 *Decode logic*: it decodes the fetched instruction, whose address in memory is A, and sends the details about the instruction to the *Control unit.* This module classifies instructions according to three categories:
   4.1 *Data processing*: if the instruction belongs to the set of instructions the I-IP is able to process, which is defined at design time, the I-IP performs instruction hardening and consistency check. Otherwise, the instruction is treated as "other", as described in item "*c*". Moreover, for the purpose of the control flow check, the address A_{next} of the next instruction in the program is computed, as described in Section 3.2.
   4.2 *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 in Section 3.2.
   4.3 *Other*: the instruction does not belong to the previous categories. The I-IP forwards it to the main processor and it computes the address A_{next} of the next instruction in the program, only, as described in Section 3.2.
5 *Control unit*: it supervises the operation of the I-IP. Upon receiving a request for an instruction fetch from the *CPU interface* it activates the *Fetch logic*. Then, depending on the information produced by the *Decode logic*, it either issues to the main processor the sequence of instructions summarized in Fig. 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 and forwards them to the processor core at the correct time.
6 *ALU*: it implements a subset of the main processor's instruction set. This module contains all the functional modules (adder, multiplier, etc.) needed to execute the data processing instructions the I-IP manages.

Two customization phases are needed for deploying successfully the I-IP in a SoC:

Processor adaptation: the I-IP has to be adapted to the main processor the SoC employs. This customization impacts on the *CPU interface*, the *Memory interface*, the *Fetch logic*, and the *Control unit*, 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 the main processor is executing (mainly affecting the set of data processing instructions to be hardened by the I-IP). This operation impacts the *Decode logic*, and the I-IP's *ALU*, as it defines which instructions the I-IP replicates. In this phase, designers must decide which of the instructions of the program to be executed by the main processor have to be hardened. Moreover, in this phase designers decide how often instructions have to be hardened. This 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. For this reason, we developed an automatic design flow, described in Section 3.4, that supports this phase.

3.4 The Design Flow

We developed the prototype of a tool that supports the automatic design flow depicted in Fig. 5 to implement the application adaptation phase. A *disassembler* tool reads the binary code of the application that should run on the SoC's main processor, and it generates a report describing the application's instruction mix, where the instructions composing the binary code are listed, along with their frequency in the code (i.e., how often an instruction appears in the code). A second tool, the *I-IP generator*, reads the instruction mix, and a set of constraints provided by the designer, and generates the VHDL model for the I-IP, whose *Decode* and *ALU* modules are adapted to the given application. Specific versions of those tools have been developed for each processor core used in the experiments described in Section 4.

The designer's constraints specify which instructions in the instruction mix have to be hardened by the I-IP and which should be ignored (i.e., when fetched, the I-IP forwards them directly to the main processor, without performing any other operation, except control flow check).

In the current implementation of the design flow the designers select the instructions to be hardened, according

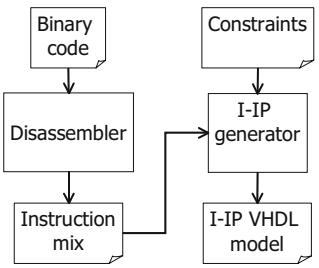


Fig. 5 The design flow to support the application adaptation phase

to their frequency in the code. We are now working to enrich the information we provide to designers with profiling information coming from the analysis of the frequency of execution of each instruction during the evaluation of a representative workload.

4 Experimental Results

To assess the effectiveness of the proposed technique, and its applicability to SoCs based on different core processors, we developed two prototypical implementations of SoCs, one using the Intel 8051 microcontroller as the core processor and the other using a five-stage pipelined MIPS processor. The selection of those two processors was due to their widespread use in the implementation of SoCs by the industry.

4.1 Using a CISC Microcontroller

This experiment implements the Viterbi encoding of a stream of data. The SoC is composed of a processor core implementing the Intel 8051 instruction set, which runs a software implementation of the Viterbi algorithm. We exploited Synopsys' Design Compiler to obtain a gate-level model of the SoC (exploiting an in-house developed technology library). The SoC area occupation is 52,373 μm^2, including the processor core and its instruction/data memory.

In order to evaluate the impact of the adoption of the I-IP on the SoC design, we developed an Intel 8051 compatible version of the I-IP that implements the control flow check approach, as well as instruction hardening and consistency check. Moreover, in order to evaluate the benefits stemming from the adoption of our technique, we evaluated its capability of detecting errors that may affect the SoC.

In particular, we focused on SEUs affecting the processor core's internal memory elements. For this purpose, we performed several fault injection campaigns by exploiting the fault injection environment presented in [7], during which 10,000 randomly selected faults were injected in the processor's memory elements (the register file, the control registers, and the hidden registers the Intel 8051's control unit embeds).

During the experiments, we considered two different implementations of the I-IP's ALU, where different sets of instructions are replicated. The first implementation replicates only the increment (INC) instruction, which is the most used instruction in the program, while the second implementation replicates both the increment and the add (ADD) instructions. The selection of the instructions to be hardened, in these experiments, was based on the static count of instructions in the code to be executed by the core processor. Table 1 lists the results we attained, in terms of percent reduction of failures (i.e., faults for which the outputs of the faulty SoC differ from the expected correct ones) that we observed with respect to the unhardened SoC, area overhead the I-IP introduces, and performance overhead with respect to the unhardened SoC.

As the reader can observe, the approach increases significantly the dependability of the system. In case the most frequently used instruction (INC) is hardened, the number of observed failures is reduced by 81%. When the ADD instruction is also hardened, the reduction of observed failures is about 88%.

Moreover, the figures concerning the area overhead and the performance degradation show that, by selecting which instructions to harden, the designers can trade dependability for area increase and speed reduction.

Please note that the introduced area overhead is far below the amount of silicon area needed to duplicate the whole system, as well as lower than that of the alternative approaches we adopted to harden the same system. When the SIHFT approach described in [6] is exploited, its area overhead (due to the increased memory requirement) is significantly higher than that of the approach presented here, while the performance degradation and the fault detection capabilities of the two methods are comparable. When the hybrid approach [5] is considered, its area overhead is still higher than the approach presented in this paper. However, the hybrid approach presented in [5] shows lower performance degradation and slightly higher fault detection capability. The method presented in [24] can not be easily compared, since it was implemented for a different processor. However, that method is clearly characterized by a lower area overhead, a higher performance overhead, and a lower fault coverage.

The experiments also show that not all the faults can be detected; 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. For this type of

Table 1 Intel 8051 results

Method	Proposed here		Proposed in [6]	Proposed in [5]
Hardened Instructions	INC	INC ADD	n.a.	n.a.
Reduction of failures (%)	81.3	87.5	81.8	92.5
Area overhead due to I-IP (%)	13.1	15.7	76.2	51.8
Performance overhead (%)	292.0	314.0	388.3	108.9

faults the I-IP fetches the operand for the hardened instruction from a wrong source, as the operation executed by the main processor does. As a result, this type of fault escapes the error detection mechanisms provided by the I-IP. Some of the escaped faults 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 at the wrong time. A typical example of this type of fault is a 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 program's control flow, and therefore it escapes the control flow check that the I-IP employs. Finally, some of the escaped faults affected unhardened instructions (mainly those concerning the return from a subroutine).

As a final remark, it is worth noting that the method is able to detect all the permanent faults affecting the processor's *ALU*, since data manipulation instructions are executed in parallel by the I-IP.

4.2 Using a RISC Microprocessor

In order to check the adequacy of the proposed technique for a different architecture, another implementation of the I-IP, using a SoC with a 5-stage pipelined MIPS processor has also been simulated.

In this experiment, the same fault model (SEUs affecting internal memory elements of the processor) and methodology of the experiment with the 8051 core have been used, but a cycle-accurate simulator [4] has been used to check the fault tolerance of the system.

The simulated SoC now is composed of a processor core that implements the MIPS instruction set, and runs the same Viterbi encoding algorithm used in the experiments described in Section 4.1.

Two different implementations of the I-IP, hardening one or three data processing instructions, have been tested, both providing instruction hardening, consistency check, and control flow check. This time, the choice of instructions to be hardened in the experiment was based on runtime statistics, shown in Table 2, and not on static analysis of the code, as in the 8051 experiment.

First, only the most used data processing instruction (ADDU) has been hardened, and the fault injection process executed. In a second step, a different version of the I-IP has been implemented, hardening also the second and third most executed data processing instructions in the Viterbi application (ANDI and SRA), and the fault injection experiments repeated.

Because the experiments with the MIPS core have been done using a cycle-accurate simulator, which takes a long time to run the application once, only 1,000 fault injection campaigns have been executed with each of the above mentioned implementations of the I-IP using the MIPS core, and the obtained results are shown in Table 3.

Table 2 Runtime frequency of instructions

Viterbi execution (7,182 instructions)		
Instruction	Frequency	Percent
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

The fault injection process consisted in running the Viterbi application 1,000 times, and for each execution one fault has been injected. The cycle in which the fault was injected and the specific bit of the internal memory element that the fault affected were choosen randomly in each execution.

In order to be able not only to detect errors using the I-IP, but also to check if every error was detected or not, the fault injection experiments were executed using two copies of the system running in the simulator, and injecting faults in only one of them, using the results of the second copy (golden processor) as a reference to check the results of the computation.

At the end of every execution cycle, the contents of the program counter, the memory address register, and the memory data register of the faulty system were compared with those of the golden one, in order to check if an error that was not detected by I-IP occurred. If so, information concerning this error was logged.

Table 3 MIPS results

Hardened Instructions	Control flow + instructions		Control flow only
	ADDU	ADDU ANDI SRA	
Reduction of failures (%)	74.5	79.2	64.6
Area overhead due to I-IP (%)	12.7	12.9	13.6
Performance overhead (%)	99.0	196.8	9.5

Moreover, when an injected fault caused an error that was detected by the I-IP, the execution was interrupted and all the pertinent information about the fault was logged.

Since the works described in [5] and [6] did not use the same core processor, the comparison with the results obtained in the current experiments is not possible.

Contrasting the figures for reduction of failures in Table 3 with those obtained for the 8051 microcontroller, shown in Table 1, one can see that smaller percentages of faults have been detected by the I-IP for the MIPS architecture while running the Viterbi application. Also, in this work we have logged separately not only the errors in the execution of the hardened instructions in different experiments, but also the control flow errors detected by the proposed mechanism. The results shown in Table 3 are further analyzed in Section 4.3.

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 that significant, because the I-IP implemented to work 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.

Nevertheless, it is important to highlight that the area overhead, for both processors, is very small, when compared to other approaches.

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 shown in Fig. 3, each time an instruction that must be hardened is fetched from memory by the core processor.

4.3 Fault Coverage Analysis

As shown in the previous sections, the proposed approach is adequate for use with SoCs based on CISC microcontrollers or RISC processors, but the ability to detect faults in the MIPS implementation is 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. In the specific case of the MIPS version used in the simulations, the register file alone represents 56% of the total memory elements inside the processor, the remaining 44% corresponding to the pipeline registers, special ALU registers (such as the high-order and low-order registers used only in multiplications) and other memory elements used by the control logic. This was confirmed by the analysis of the log reports of the fault injection experiments.

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, in Section 4.1, 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.

While the requirement for register file protection may seem unfeasible, the analysis of recent industry trends shows that it may become a standard design practice in the near future. High-end processors available in the market already provide EDAC protection for internal memory elements [19]. Also, during a panel discussion at the International Test Conference 2005 [22, 23], in which the adequate level of concern for soft errors was discussed, audience members from the industry reported that the protection of internal registers against soft errors is already a practice being adopted also for processors targeted at the desktop and high-performance embedded systems markets.

This overall tendency to protect internal register banks goes in the same direction that favors the I-IP technique proposed here.

5 Conclusion

In this paper, a new approach able to harden SoCs against transient errors in the processor core they embed has been presented. The method is based on introducing in the SoC a further module (I-IP), whose architecture is general, that needs to be customized to the adopted processor core.

The I-IP monitors the processor buses and performs two main functions: when the processor fetches a data processing instruction belonging to a design time selected set, it acts on the bus and lets the processor fetch a sequence of instructions generated on-the-fly, instead of the original one. The sequence of instructions allows the I-IP to get the operands of the original data processing instructions, which

is then executed both by the processor and by the I-IP; the results obtained by the processor and the I-IP are then compared for correctness. The I-IP also checks the correctness of the address used by the processor to fetch each new instruction, thus detecting a number of control flow errors.

The method is inspired in SIHFT and hybrid techniques, but it does not introduce any memory overhead in the hardened system (code redundancy is introduced on-the-fly). Moreover, no change is required on the application code, whose source version is not required to be available. Finally, the method allows designers to trade-off costs and reliability, mainly by suitably selecting the subset of data-manipulation instructions to be hardened.

In order to validate the proposed approach, two implementations of the I-IP with different core processor architectures have been simulated and tested against fault injection, using SEUs in the internal memory elements as the fault model. The experimental results show that the approach is able to increase the dependability of a SoC based on a processor core, while giving to designers the possibility of trading off dependability with area overhead and performance degradation.

The experiments with MIPS also have shown that most of the non detected errors are due to SEUs affecting the register file of the processor before the operands are read and their values forwarded to the I-IP. In order to avoid these errors, the register file should be protected using EDAC techniques, what would imply in changes in the internal architecture of the processor.

Since one of the main goals of the technique proposed here is to be non-intrusive, i.e., not to modify the architecture of the core processor, an important outcome of this study is that the use of EDAC techniques to protect the contents of the internal registers of the core processor should be a recommended design rule to be adopted by the industry for SoCs using future technologies.

In a future extension of this work, assuming that in future processors all internal memory elements will be protected against SEUs, the effectiveness of the I-IP to protect the SoC against other types of transient faults affecting the combinational logic of the processor will be assessed.

References

1. Aho A, Sethi R, Ullman J (1986) Compilers: principles, techniques and tools. Addison-Wesley, Reading, MA
2. Alkhalifa Z, Nair VSS, Krishnamurthy N, Abraham JA (1999) Design and evaluation of system-level checks for on-line control flow error detection. IEEE Trans Parallel Distrib Syst 10(6):627–641 (Jun)
3. Austin TM (2000) DIVA: a dynamic approach to microprocessor verification. Journal of Instruction Level Parallelism 2(May)1–6 http://www.jilp.org/vol2
4. Beck F, Mattos JCB, Wagner FR, Carro L (2003) CACO-PS: a general purpose cycle-accurate configurable power-simulator. In: Proceedings of the 16th Brazilian symposium on integrated circuits and systems design (SBCCI 2003), September 2003
5. Bernardi P, Bolzani LMV, Rebaudengo M, Sonza Reorda M, Vargas FL, Violante M (2006) A new hybrid fault detection technique for Systems-on-a-Chip. IEEE Trans Comput 55(2):185–198 (Feb)
6. Cheynet P, Nicolescu B, Velazco R, Rebaudengo M, Sonza Reorda M, Violante M (2000) Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. IEEE Trans Nucl Sci 47(6 part 3): 2231–2236 (Dec)
7. Civera P, Macchiarulo L, Rebaudengo M, Sonza Reorda M, Violante M (2001) Exploiting circuit emulation for fast hardness evaluation. IEEE Trans Nucl Sci 48(6):2210–2216 (Dec)
8. Eto A, Hidaka M, Okuyama Y, Kimura K, Hosono M (1998) Impact of neutron flux on soft errors in MOS memories. In: Proceedings of the IEEE international electronic devices meeting (IEDM), IEEE Computer Society, Los Alamitos, CA, pp 367–380
9. Goloubeva O, Rebaudengo M, Sonza Reorda M, Violante M (2003) Soft-error detection using control flow assertions. In: Proceedings of the 18th IEEE international symposium on defect and fault tolerance in VLSI systems—DFT 2003, November 2003, pp 581–588
10. Huang KH, Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. IEEE Trans Comput 33:518–528 (Dec)
11. Lisbôa CAL, Carro L, Sonza Reorda M, Violante M (2006) Online hardening of programs against SEUs and SETs. In: Proceedings of the 21st IEEE international symposium on defect and fault tolerance in VLSI systems—DFT 2006, IEEE Computer Society, Los Alamitos, CA, October 2006, pp 280–288
12. Mahmood A, McCluskey EJ (1988) Concurrent error detection using watchdog processors—a survey. IEEE Trans Comput 37 (2):160–174 (Feb)
13. Mahmood A, Lu DJ, McCluskey EJ (1983) Concurrent fault detection using a watchdog processor and assertions. In Proceedings of the IEEE international test conference 1983 (ITC '83), pp. 622–628
14. Namjoo M (1983) CERBERUS-16: an architecture for a general purpose watchdog processor. In: Proceedings of the 13th international symposium on fault-tolerant computing (FTCS-13), pp 216–219
15. Namjoo M, McCluskey EJ (1982) Watchdog processors and capability checking. In: Proceedings of the 12th international symposium on fault-tolerant computing (FTCS-12), pp 245–248
16. Oh N, Mitra S, McCluskey EJ (2002) ED^{4}I: error detection by diverse data and duplicated instructions. IEEE Trans Comput 51 (2):180–199 (Feb)
17. Oh N, Shirvani PP, McCluskey EJ (2002) Control flow Checking by Software Signatures. IEEE Trans Reliab 51(2):111–112 (Mar)
18. Ohlsson J, Rimen M (1995) Implicit signature checking. In: Digest of papers of the 25th international symposium on fault-tolerant computing (FTCS-25), pp 218–227
19. Quach N (2000) High availability and reliability in the Itanium processor. IEEE MICRO 20(5):61–69 (Sep–Oct)
20. Schillaci M, Sonza Reorda M, Violante M (2006) A new approach to cope with single event upsets in processor-based systems. In: Proceedings of the 7th IEEE Latin–American test workshop—LATW 2006, March 2006, pp 145–150
21. Schuette MA, Shen JP (1987) Processor control flow monitoring using signatured instruction streams. IEEE Trans Comput 36 (3):264–276 (Mar)
22. Stolicny C (2006) ITC 2005 panels. IEEE Des Test Comput 20 (5):164–166 (Mar–Apr)
23. Vijaykrishnan N (2005) Soft-errors: is the concern for soft errors overblown? In: Proceedings of the IEEE international test conference 2005 (ITC 2005), November 2005 (2 pages)

24. Weaver C, Gebara FF, Austin T, Brown R (2002) Remora: a dynamic self-tuning processor. University of Michigan CSE Technical Report CSE-TR-460-02, July 2002. University of Michigan, MI, USA
25. Wilken K, Shen JP (1990) Continuous signature monitoring: low-cost concurrent detection of processor control errors. IEEE Trans Comput-Aided Des Integr Circuits Syst 9(6):629–641 (Jun)

Eduardo Luis Rhod was born in Lajeado, Brazil, in 1981. He received his degree in Automation and Control Engineering from Pontificia Universidade Católica do Rio Grande do Sul in 2004. He is finishing his master degree in Electrical Engineering at Universidade Federal do Rio Grande do Sul and his research interests are computer architecture and fault tolerant systems.

Carlos Arthur Lang Lisbôa was born in Porto Alegre, Brazil, in 1950. He received the Civil Engineering and MSc degrees from Universidade Federal do Rio Grande do Sul (UFRGS), Brazil, in 1971 and 1976, respectively. He is a faculty member of the Applied Informatics Department at the Informatics Institute of UFRGS since 1972. He is currently enrolled in the Ph.D. program of the Graduation Program in Computer Science at UFRGS. His research interests are computer architecture and fault tolerant systems.

Luigi Carro was born in Porto Alegre, Brazil, in 1962. He received the Electrical Engineering and the MSc degrees from Universidade Federal do Rio Grande do Sul (UFRGS), Brazil, in 1985 and 1989, respectively. From 1989 to 1991 he worked at ST-Microelectronics, Agrate, Italy, in the R&D group. In 1996 he received the Ph.D. degree in the area of Computer Science from Universidade Federal do Rio Grande do Sul (UFRGS), Brazil. He is presently a professor at the Applied Informatics Department at the Informatics Institute of UFRGS, in charge of computer architecture disciplines at the undergraduate level. He is also a member of the Graduation Program in Computer Science at UFRGS, where he is responsible for courses on embedded systems, digital signal processing, and VLSI design. His primary research interests include mixed-signal design, digital signal processing, mixed-signal and analog testing, and rapid system prototyping. He has published more than 120 technical papers on those topics and is the author of the book Digital systems Design and Prototyping (in Portuguese) and co-author of the book Fault-Tolerance Techniques for SRAM-based FPGAs, 1. ed., Dordrecht: Springer, 2006.

Matteo Sonza Reorda took his M.S. degree in Electronics (1986) and Ph.D. degree in Computer Engineering (1990) from Politecnico di Torino, Italy. Currently, he is a Full Professor at the Department of Computer Engineering of the same institution. His main research interests include testing and fault tolerant design of electronic systems. He published more than 200 papers on these topics. He has been the General (1998) and Program Co-chair (2002, 2003) of the IEEE International Online Testing Symposium. Currently, he is the chair of the committee on Test Generation, Simulation and Diagnosis of the DATE 2007 conference.

Massimo Violante received the M.S. degree in Computer Engineering (1996) and the Ph.D. degree in Computer Engineering (2001) from Politecnico di Torino, Italy. Currently, he is an Assistant Professor with the Department of Computer Engineering of the same institution. His main research interests include design, validation, and test of fault-tolerant electronic systems.