UNIVERSIDADE FEDERAL DO CEARÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE TELEINFORMÁTICA

Alexandre Augusto da Penha Coelho

FT-OPENRISC 1200: UM PROCESSADOR DE ARQUITETURA RISC TOLERANTE A FALHAS PARA SISTEMAS EMBARCADOS

FORTALEZA - CEARÁ
OUTUBRO - 2010

Alexandre Augusto da Penha Coelho

FT-OPENRISC 1200: UM PROCESSADOR DE ARQUITETURA RISC TOLERANTE A FALHAS PARA SISTEMAS EMBARCADOS

DISSERTAÇÃO

Dissertação submetida ao corpo docente da Coordenação do Programa de Pós-Graduação em Engenharia de Teleinformática da **Universidade Federal do Ceará** como parte dos requisitos necessários para obtenção do grau de MESTRE EM ENGENHARIA DE TELEINFORMÁTICA.

Área de concentração: Sinais e Sistemas

Prof. Dr. Paulo César Cortez
(Orientador)

Prof. Dr. Helano de Sousa Castro
(Co-orientador)

FORTALEZA - CEARÁ
2010

C614f Coelho, Alexandre Augusto da Penha
FT-OPENRISC 1200: um processador de arquitetura Risc tolerante a falhas para sistemas embarcados / Alexandre Augusto da Penha Coelho. -- Fortaleza, 2010.
129 f. ; il. enc.

Orientador: Prof. Dr. Paulo César Cortez
Co-orientador: Prof. Dr. Helano de Sousa Castro
Área de concentração: Sinais e Sistemas
Dissertação (Mestrado) - Universidade Federal do Ceará, Centro de Tecnologia, Depto. de Engenharia de Teleinformática, Fortaleza, 2010.

1. Microeletrônica. 2. FPGA. 3. EDAC. I. Cortez, Paulo César (Orient.). II. Castro, Helano de Sousa. III. Universidade Federal do Ceará – Programa de Pós-Graduação em Engenharia de Teleinformática. III. Título.

CDD 621.38

ALEXANDRE AUGUSTO DA PENHA COELHO

FT-OPENRISC 1200: UM PROCESSADOR DE ARQUITETURA RISC TOLERANTE A FALHAS PARA SISTEMAS EMBARCADOS

Dissertação submetida à Coordenação do Programa de Pós-Graduação em Engenharia de Teleinformática, da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Engenharia de Teleinformática.
Área de concentração Sinais e Sistemas.

Aprovada em 28/10/2010.

BANCA EXAMINADORA

Prof. Dr. Paulo César Cortez (Orientador)
Universidade Federal do Ceará - UFC

Prof. Dr. Helano de Sousa Castro
Universidade Federal do Ceará - UFC

Prof. Dr. Elias Teodoro da Silva Júnior
Instituto Federal de Educação, Ciência e Tecnologia do Ceará – IFCE

Prof. Dr. Giovanni Cordeiro Barroso
Universidade Federal do Ceará - UFC

Prof. Dr. Edna Natividade da Silva Barros
Universidade Federal de Pernambuco - UFPE

A Deus, aos meus Pais, Augusto e Cristina,
à minha esposa Aline pelo
incentivo e apoio.

Sumário

Lista de Figuras viii

Lista de Tabelas x

Lista de Siglas xii

Resumo xiv

Abstract xv

Agradecimentos xvi

1 Introdução 1
- 1.1 Motivação . . . 1
- 1.2 Objetivos . . . 3
- 1.3 Resumo das Contribuições . . . 3
- 1.4 Organização . . . 4

2 Princípios de Tolerância a Falhas 5
- 2.1 Introdução . . . 5
- 2.2 Falhas, Erros e Defeitos . . . 6
- 2.3 Prevenção de Falhas . . . 7
- 2.4 Estratégias de Tolerância a Falhas . . . 8
  - 2.4.1 Redundância de *hardware* . . . 8
    - 2.4.1.1 Redundância de *hardware estática* . . . 8
    - 2.4.1.2 Redundância de *hardware* dinâmica . . . 9
  - 2.4.2 Redundância de *software* . . . 10
  - 2.4.3 Redundância de tempo . . . 10
  - 2.4.4 Detecção de falhas . . . 11
    - 2.4.4.1 Verificação por replicação . . . 11
    - 2.4.4.2 Verificação por temporização . . . 12
    - 2.4.4.3 Verificação por codificação . . . 12
- 2.5 Avaliação Quantitativa da Confiabilidade . . . 12
  - 2.5.1 Taxa de falha e função de confiabilidade . . . 13
  - 2.5.2 MTTF - Tempo Médio para Falhar (*Mean Time to Failure*) . . . 15
  - 2.5.3 MTTR - Tempo médio para Reparo (*Mean Time to Repair*) . . . 15
  - 2.5.4 MTBF - Tempo Médio entre Falhas (*Mean Time Between Failures*) 15
  - 2.5.5 Cobertura da falha . . . 16
  - 2.5.6 Estimativa da confiabilidade . . . 16

2.5.7 Modelagem combinatorial . . . 17
2.6 Funções de Confiabilidade de Sistemas . . . 19
2.6.1 Configuração TMR . . . 19
2.7 Conclusão . . . 19

3 Efeitos da Radiação Sobre os Semicondutores 21
3.1 Introdução . . . 21
3.2 Os Efeitos da Radiação Sobre os Circuitos Integrados . . . 24
3.3 *Single Event Effects* (SEEs) . . . 25
3.3.1 *SEE* Destrutivo . . . 25
3.3.2 *Hard SEE* . . . 26
3.3.3 *Soft SEE* . . . 26
3.3.3.1 *Single Event-Upset* (SEU) . . . 26
3.3.3.2 *Single Event-Transient* (SET) . . . 29

4 Códigos Corretores de Erros 34
4.1 Introdução . . . 34
4.2 Códigos de *Hamming* . . . 35
4.3 Código de Hamming + Paridade . . . 37
4.4 Códigos de Reed-Solomon . . . 37
4.4.1 Polinômio Primitivo . . . 38
4.4.2 Codificação Reed-Solomon . . . 41
4.4.2.1 Codificação Sistemática com um Registrador de Deslocamento de $(n-k)$ Estágios . . . 43
4.4.3 Decodificação Reed-Solomon . . . 46
4.4.3.1 Cálculo da Síndrome . . . 47
4.4.3.2 Localização dos Erros . . . 48
4.4.3.3 Valores dos Erros . . . 50
4.4.3.4 Correção do Polinômio Recebido . . . 51
4.5 Conclusão . . . 52

5 Processador OpenRISC 1200 53
5.1 Arquitetura do OpenRISC 1200 . . . 53
5.2 A CPU do OpenRISC 1200 . . . 54
5.2.1 Unidade de Instrução (*Instruction Unit*) . . . 55
5.2.2 Registradores de Propósito Geral (*GPRs - General Purpose Registers*) 55
5.2.3 Unidade de Carga/Armazenamento (LSU - *Load/Store Unit*) . . . 56
5.2.4 *Pipeline* de Operações sobre Inteiros (*Integer Execution Pipeline*) . 56
5.2.5 Unidade de DSP *MAC Unit* . . . 57
5.2.6 Unidade de Sistema (*System Unit*) . . . 57
5.2.7 Exceções (*Exceptions*) . . . 57
5.3 Cache de Dados e Instruções (*Data and Instruction Cache*) . . . 58
5.4 Unidade de Gerenciamento de Memória (MMU - *Memory Management Units*) . . . 59
5.5 Gerenciamento de Energia (*Power Management*) . . . 59
5.6 Unidade de Depuração (*DEBUG*) . . . 59
5.7 Temporização Integrada (*Tick Timer*) . . . 60
5.8 Controlador de Interrupções Programável (*Programmable Interrupt Controller*) . . . 60

5.9 Barramento *Wishbone* 61
5.10 *Software* e Ferramentas para o OpenRISC 1200 61
5.11 Conclusão 62

6 FT-OpenRISC 1200: Um Processador Tolerante a Falhas 63
6.1 Pontos de Falhas no OpenRISC 1200 64
6.2 Detecção de Erros e Tolerância a Falhas 65
6.2.1 Registradores tolerantes a SEU e SET 66
6.2.2 *Software* TMRlesc 67
6.2.3 Proteção da Lógica Combinacional 69
6.2.4 Proteção da memória 71
6.2.4.1 Código de Hamming 72
6.2.4.2 Código Reed-Solomon 72
6.2.4.3 Memória *Cache* Tolerante a Falhas 72

7 Resultados e Discussões 76
7.0.5 Resultados da síntese física em FPGA 76
7.0.6 Análise da frequência de operação 77
7.0.7 Descrição e resultados dos testes de injeção de falhas em FPGA . . 77
7.0.7.1 Injeção de falhas em *flip-flops* 78
7.0.7.2 Injeção de falhas na memória *cache* 78
7.0.8 Eficácia dos testes 83

8 Conclusões, Contribuições e Trabalhos Futuros 84

Referências Bibliográficas 87

Apêndices 97

A Códigos de Bloco 97
A.1 Matriz Geradora 98
A.2 Códigos Sistemáticos 100
A.3 Matriz de Verificação de Paridade 102
A.4 Capacidade de Correção de Um Código Linear 102
A.5 Síndrome 104

B HDL do Processador OpenRISC 1200 106
B.1 CPU *Core top-level* 109
B.2 *Top-level* do OpenRISC 1200 111

Lista de Figuras

2.1 sequência de Falha, Erro e Defeito, adaptado de (PRADHAN, 1996) 7
2.2 um típico sistema TMR. 9
2.3 um típico sistema NMR. 9
2.4 um típico sistema *stand-by*. 10
2.5 curva da banheira mostrando a taxa de falhas de um componente em relação ao tempo. 14

3.1 exemplo da curva da seção transversal por LET, adaptado de (SCHWANK et al., 2010). 23
3.2 comportamento da região sensível de um transistor, dada a ação de uma partícula carregada, adaptado de (WESTE; HARRIS, 2010). 24
3.3 forma do pulso de corrente gerado após a colisão de uma partícula carregada com a região sensível do silício, adaptado de (DODD; MASSENGILL, 2003). . 25
3.4 *single event upset* (SEU) em uma célula de memória SRAM, adaptado de (KASTENSMIDT; CARRO; REIS, 2006). 27
3.5 MBU provocado por uma simples partícula, adaptado de (KASTENSMIDT; CARRO; REIS, 2006). 28
3.6 *single-event transient* (SET) e sua possível propagação em um bloco combinacional, adaptado de (NIEUWLAND; JASAREVIC; JERIN, 2006). 30
3.7 mascaramento lógico de uma porta NAND. 31
3.8 processo de atenuação de um pulso de tensão quando este se propaga por portas lógicas inversoras (ALEXANDRES; ANGHEL; NICOLAIDIS, 2002). . . . 31
3.9 mascaramento por latching Window. 32
3.10 modelo de messenger (MESSENGER, 1982) para um pulso de tensão gerado a partir de um SET. 33

4.1 Registro de deslocamento. 44
4.2 Codificador LFSR para RS(7,3). 45

5.1 arquitetura do OpenRISC 1200, adaptado de (OPENCORES, 2001) 54
5.2 arquitetura da CPU do microprocessador openRISC 1200, adaptado de (OPENCORES, 2001) 55
5.3 estrutura da Cache 58

6.1 diagrama de blocos do OpenRISC 1200 original. 64
6.2 diagrama de blocos de execução e acesso a memória no OpenRISC 1200 original. 65
6.3 TMR no registrador. 67
6.4 registradores replicado por TMRlesc: votador Triplicado (a) e votador simples (b) 68
6.5 registradores replicado por TMRlesc. 69
6.6 lógica sequencial e combinacional sem TMR 70
6.7 topologia de circuito digitais combinacional e sequencial 70
6.8 TMR para lógica combinacional e sequencial 71
6.9 diagrama de bloco de uma memória tolerante a falhas. 73
6.10 matriz de memória com os três possíveis tipos de falhas. 73
6.11 esquemático de uma palavra de memória protegida por Reed-Solomon e Hamming. 74
6.12 esquemático de uma memória tolerante a falhas, utilizando Hamming e Reed-Solomon. 75
6.13 codificador utilizando Hamming e Reed-Solomon do FT-OpenRISC 1200. . 75

7.1 memória *cache* do OpenRISC 1200 dividida em linhas e colunas 78
7.2 memória *cache* do OpenRISC 1200 dividida em blocos 79
7.3 falhas simples, duplas e múltiplas em uma memória 80

A.1 estrutura da palavra-código em um código sistemático 100

B.1 periféricos conectados ao OpenRISC 1200. 107

Lista de Tabelas

4.1 exemplo de código de bloco (6,3). 36
4.2 exemplo de código de bloco (6,3). 37
4.3 exemplo de código de bloco (6,3). 38
4.4 mapeamento de elementos no corpo para $GF(2^3) com f(x) = 1 + X + X^3$. 39
4.5 classe de polinômios primitivos 40
4.6 cálculo dos elementos de GF(2^4) 41
4.7 mapeamento de soma de elementos para $GF(2^3)$ 42
4.8 mapeamento de multiplicação de elementos para $GF(2^3)$ 42
4.9 função XOR de 3 entradas 44
4.10 conteúdo dos registradores no processo de codificação **LFSR** 45

7.1 comparação de área ocupada entre o OpenRISC 1200 e o FT-OpenRISC 1200 76
7.2 aumentos percentuais do FT-OpenRISC 1200 em relação ao OpenRISC 1200. 77
7.3 frequência de operação do OpenRISC 1200 e do FT-OpenRISC 1200 77
7.4 detecção e correção de falhas simples no FT-OpenRISC 1200 80
7.5 detecção e correção de falhas duplas no FT-OpenRISC 1200 81
7.6 detecção e correção de múltiplas falhas no FT-OpenRISC 1200 83

A.1 exemplo de código de bloco (6,3). 98

B.1 macros e respectiva interferência no sistema. 108
B.2 macros e respectiva interferência no sistema 109
B.3 módulos presentes no CPU Core, correspondência com arquivo e respectiva funcionalidade. 110
B.4 continuação dos módulos presentes no CPU Core, correspondência com arquivo e respectiva funcionalidade 111
B.5 arquivos e funcionalidade correspondentes para um determinado módulo do OpenRISC 1200. 112

B.6 continuação de arquivos e funcionalidade correspondentes para um determinado módulo do OpenRISC 1200. 113

Lista de Siglas

API	Application Program Interface
ASIC	Application-Specific Integrated Circuit
BCH	Bose-Chaudhuri-Hocquenghem
CI	Circuito Integrado
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CRC	Cyclical Redundancy Check
DMA	Direct Memory Access
DSP	Digital Signal Processing
DVD	Digital Video Disc
ECC	Error-Correcting Code
FEC	Forward error correction
FPGA	Field Programable Gate Array
GCC	GNU C Compiler
GNU	GNU is Not Unix
GPL	General Public License
HDL	Hardware Description Language
MAC	Multiplicator and Accumulator
RAM	Random Access Memory
RAMB	Random Access Memory Block
RHBD	Radiation Hardening by Design
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory

RS	Reed-Solomon
SEU	Single Event Upset
SRAM	Static Random Access Memory
TMR	Triple Modular Redundancy

Resumo

Atualmente, FPGAs (*Field-Programmable Gate Arrays*) têm sido utilizados amplamente em projetos de sistemas eletrônicos digitais. A fim de prover uma quantidade de recursos programáveis que satisfaçam às necessidades dos projetos atuais, os dispositivos FPGAs são fabricados utilizando tecnologias CMOS (*Complementary Metal-Oxide Semiconductor*). Com a contínua redução das dimensões dos transistores fabricados com essas tecnologias, a probabilidade da ocorrência de falhas transientes causadas por radiação vem aumentando consideravelmente. Tais falhas podem ocorrer quando uma partícula carregada, proveniente de uma fonte de radiação ionizante, atinge uma região sensível de um transistor. Para diminuir os efeitos causados por falhas transientes em FPGAs, existem disponíveis no mercado dispositivos FPGAs tolerantes a radiação. Contudo, além de seu custo unitário ser alto, os mesmos em geral, estão três gerações atrasados em relação ao estado da arte, portanto o seu uso fica limitado a poucos casos. Esta realidade motiva o estudo de técnicas economicamentes viáveis para reduzir os efeitos de falhas transientes na lógica. Neste trabalho, propõe-se a utilização e validação de técnicas de tolerância a falhas em um processador OpenRISC 1200 para garantir o seu correto funcionamento sem a necessidade de utilizar FPGAs tolerantes a radiação. É proposto ainda uma arquitetura para proteger a lógica e a memória do processador OpenRISC 1200 contra *Single Event Upset* (SEUs) e *Single Event Transients* (SETs). Toda a arquitetura é descrita em HDL e sintetizada utilizando o *software* ISE da Xilinx para FPGAs da família Virtex-4. A injeção de falhas nos circuitos foi realizada através das JTAGs, a fim de simular falhas neste sistema. O processo de injeção de falhas permite validar a eficiência das técnicas de tolerância a falhas empregadas em nível da lógica sequencial/combinacional do processador OpenRISC 1200, assim como em sua memória de dados. Além desta validação, realizou-se uma comparação, em termos de tempo de execução e área, entre o processador OpenRISC 1200 e o modificado (FT-OpenRISC 1200).

Abstract

Currently , FPGAs field-programmable gate arrays have been widely used in digital electronic systems design. In order to provide an amount of programmable resources that meet the needs of current projects, FPGA devices are manufactured using CMOS Complementary Metal-Oxide-Semiconductor technologies known as submicron or nanometric. With the continued downsizing of transistors manufactured with these technologies, the likelihood of occurrence of transient faults caused by radiation has been considerably increasing. Such failures may occur when a charged particle, originated from a source of ionizing radiation, strikes a sensitive region of a transistor. To reduce the effects caused by transient faults in FPGAs, there are off-the-shelf protected FPGA devices. However, in addition to their unit cost being extremely high, and they in general staying three generations before state of art, their use is limited to few cases. This fact motivates the study of techniques to reduce the transient fault effects in user logic whose use is more economically viable. In this work, we propose the use and validation of fault tolerance techniques in an OpenRISC 1200 processor to ensure its proper operation discarding the need of using protected FPGAs. An architecture is proposed to protect processor logic and memory against Single Event Upset (SEUs) and Single Event Transients (SETs). The whole architecture is described in HDL and synthesized through Xilinx ISE software for Virtex IV family FPGAs. The fault injection in circuits was accomplished through the JTAG interface in order to simulate faults in this system. The process of fault injection allowed the validation of the circuits as transient fault tolerant architectures and of data memory protection techniques. Besides this validation, we carried out a performance evaluation, based on execution time and occupied area, of OpenRISC 1200 processor under test.

Agradecimentos

Dedico meus sinceros agradecimentos para:

– Deus por existirmos;

– os meu orientadores Helano de Sousa Castro e Paulo Cesár Cortez, pelas orientações e incentivos;

– os meus Pais Augusto e Cristina pelo amor incondicional;

– a minha esposa Aline Gomes pelo amor e paciência;

– a minha irmã Karine e família (Carlinhos, Carlos Fernando, Mel e Arthur) pelo amor e incentivo dedicado;

– a minha irmã Isabel pelo incentivo e apoio;

– o meu amigo e companheiro de mestrado Jardel Silveira pelas revisões e valiosas sugestões;

– os companheiros de trabalho do LESC pelas revisões e sugestões;

– o meu Professor Alexandre Moraes, por me iniciar no mundo do *software* livre;

– o LESC (Laboratório de Engenharia de Sistemas de Computação) por me disponibilizar para realização das disciplinas;

– a CAPES pelo suporte financeiro;

– o MCT e Cadence pelo acesso às ferramentas EDA;

– a Xilinx pelos kits de desenvolvimento doados à UFC e licenças de ferramentas EDA.

Capítulo 1

Introdução

Questões de desempenho e custo dominaram os esforços de otimização dos projetos de microprocessadores até os anos 90. Entretanto, atualmente, a confiabilidade vem se tornando o principal interesse em projetos de arquitetura de microprocessadores (SUBRAMANIAN, 2006; BAUMAN, 2002). Esta crescente demanda por projetos que apresentam, em sua arquitetura, características de tolerância a falhas, está relacionada tanto ao fato destes microprocessadores operarem em velocidades cada vez maiores (da ordem de Ghz) quanto às aplicações, que hoje, necessitam de processamento em tempo real.

Por outro lado, o espaço para aplicações de sistemas eletrônicos digitais cresceu muito na última década e, como resultado, a necessidade por características de tolerância a falhas tornou-se fundamental nestes sistemas. Embora os sistemas de missão crítica contenham frequentemente redundância de dispositivos para permitir a operação continuada na presença de falhas operacionais, outros sistemas, com espaço limitado para dispositivos redundantes, requerem a detecção da falha bem como a recuperação da mesma, muitas vezes com o mínimo de redundância dos dispositivos.

Diante deste cenário, percebe-se cada vez mais a necessidade de utilização de técnicas de tolerância a falhas nos sistemas atuais, antes restritas às aplicações espaciais e médicas, e hoje presentes em diversas aplicações tais como: automotivas, internet, fincanceiras e várias outras aplicações nas quais os requisitos temporais e de alta confiabilidade são prioritários para o correto funcionamento de tais sistemas.

1.1 Motivação

Os sistemas eletrônicos de tempo real embarcados em missões espaciais estão sujeitos aos elevados níveis de radiação presentes no espaço. Por isso, tais sistemas estão muito

sujeitos a falhas causadas pela colisão de partículas pesadas, também classificadas como hádrons, com as estruturas nanométricas de silício presentes nos circuitos integrados modernos. Os hádrons são partículas de interação forte que mantêm os núcleos unidos para formar os núcleos atômicos, como por exemplo os prótons, nêutrons e píons.

Particularmente para circuitos integrados contendo blocos de memória SRAM (*Static Random Access Memory*), a principal consequência destas colisões são os SEUs (*Single Event Upsets*), que correspondem à inversão permanente de um *bit*. Embora existam indústrias de circuitos integrados especializadas em produção de circuitos integrados tolerantes à radiação, devido ao pequeno volume de produção desses circuitos, os mesmos têm preços elevados. Além disso, essas fábricas estão, em geral, duas ou três gerações atrasadas em relação à tecnologia de fabricação do estado da arte (FLEETWOOD, 2004). Portanto, é muito importante garantir tolerância à radiação em nível de projeto do circuito integrado, independentemente do processo de fabricação, pois isso reduz os custos do sistema e permite utilizar os mais modernos processos de fabricação existentes.

Uma etapa importante para se projetar um microprocessador tolerante a falhas é estudar os efeitos e a propagação das falhas nesse dispositivo. Os efeitos das falhas nos microprocessadores modernos foram estudados em diversos trabalhos. No trabalho de Wang (WANG et al., 2004), uma análise do efeito de falhas transientes no Alfa 21264 e AMD Athlon foi caracterizada pela simulação, baseada na injeção de falha. Uma análise dos efeitos de erros transientes nas memórias *cache* da arquitetura do processador SPARC V8 por injeção e simulação de falhas, foi apresentado por Rebaudengo (REBAUDENGO; REORDA; VIOLANTE, 2003). Efeitos de SEUs em memórias *cache* dos processadores comerciais foram estudados por Rebaudengo (REBAUDENGO; REORDA; VIOLANTE, 2002). Em (GAISLE, 2002), a injeção da falha utilizando cargas com íons pesados provou a eficiência do microprocessador tolerante a falhas LEON-FT. O projeto e comportamento de um microcontrolador 8051 tolerante a SEU protegido por um único código de correção de erros na presença de múltiplos SEUs foi investigado por Lima (LIMA et al., 2002). Técnicas de tolerância a falhas para proteger a memória *cache* SRAM de código, interna ao processador Java (JOP), contra SEU, foi investigada por Silveira (SILVEIRA et al., 2009).

Neste trabalho é apresentada uma proposta de projeto de um microprocessador, usando técnicas de tolerância a falhas aplicadas em nível de codificação HDL (*Hardware Description Language*) no microprocessador OpenRISC 1200 de 32 *bits* (OPENCORES, 2001). Este trabalho tem como objetivo desenvolver mecanismos de tolerância a falhas para o processador OpenRISC 1200 contra SEU e SET, bem como, através de técnicas combinadas de polinômios de correção de erros EDAC (*Error Detection and Correction*),

proteger a memória do processador. Neste trabalho também foram realizados testes de confiabilidade no OpenRISC 1200 modificado (FT-OpenRISC 1200), através de injeção de falhas pela JTAG, para avaliar o grau de confiabilidade deste.

Neste trabalho, são propostas técnicas de tolerância a falhas para proteger a memória SRAM de código interna do OpenRISC 1200 contra SEUs, bem como a lógica sequencial e combinacional contra SET. Além disso, faz-se a síntese do OpenRISC1200 modificado (FT-OpenRISC 1200) por tais técnicas em uma FPGA para avaliar a sua confiabilidade contra SEUs e SETs.

1.2 Objetivos

O presente trabalho tem por objetivo geral desenvolver um protótipo do processador OpenRISC 1200 em FPGA com técnicas de tolerância a falhas em nível de codificação HDL. Alguns outros objetivos específicos devem ser alcançados:

- avaliar as técnicas de tolerância a falhas propostas para melhorar o nível de confiabilidade do OpenRISC 1200;
- validar as técnicas de tolerância a falhas implementadas em nível lógico.

1.3 Resumo das Contribuições

Esta dissertação agrega as seguintes contribuições para o desenvolvimento de processadores tolerantes a falhas:

1. concepção, implementação e avaliação de uma técnica de proteção do circuito sequencial contra S**EUs (*Single Event Upsets*)**;
2. concepção, implementação e avaliação de uma técnica de proteção do circuito combinacional contra **SET (*Sigle Event Transient*)**;
3. concepção, implementação e avaliação de uma técnica de correção de erros para proteção de memórias contra falhas múltiplas;
4. concepção, implementação e avaliação de um *software* de análise de código fonte Verilog para aplicar técnica de *Triple Modular Redundancy* (TMR) nos flip-flops;

1.4 Organização

Uma vez introduzida a motivação deste trabalho e os objetivos a que se propõe, expõe-se a seguir como o restante está apresentado. No Capítulo 2, são descritas as técnicas de tolerância a falhas. No Capítulo 3 descrevem-se os efeitos de SEU e SET em circuitos digitais. No Capítulo 4 descrevem-se as técnicas utilizadas para proteção de memórias cache. No Capítulo 5 descrevem-se a arquitetura do OpenRISC 1200 original.

O *soft core* FT-OpenRISC 1200, assim como as técnicas de tolerância a falhas, para o circuito combinacional/sequencial, e a técnica de correção de erros, para proteção de memórias contra falhas múltiplas, são descritos no Capítulo 6. As conclusões, contribuições e trabalhos futuros são apresentados no Capítulo 7.

Capítulo 2

Princípios de Tolerância a Falhas

A importância do correto funcionamento dos sistemas computacionais é percebida em diversas áreas de aplicações críticas, tais como: sistemas de defesa, telecomunicações, sistemas bancários, controle de usinas de energia, aplicações espaciais, dentre outras. Muitos desses sistemas precisam manter seu correto funcionamento muitas vezes por um longo período de tempo. Além disso, alguns não podem contar com facilidade de manutenção no local, como alguns sistemas espaciais ou sistemas computacionais instalados em locais de difícil acesso. Nestas situações, os projetistas destes sistemas devem empregar técnicas e metodologias de tolerância a falhas apropriadas, para garantir o correto funcionamento dos sistemas durante toda a missão.

Sistemas tolerantes a falhas são sistemas que possuem alto grau de dependabilidade, a qual indica a qualidade e a confiança depositada no serviço fornecido por um dado sistema, tendo como principais atributos os conceitos de confiabilidade e disponibilidade.

A confiabilidade é a capacidade de um sistema operar corretamente dentro de condições definidas, durante certo período de funcionamento e estar operacional no início desse período, já a disponibilidade é a probabilidade de o sistema estar operacional quando a utilização deste for necessária.

2.1 Introdução

Tolerância a falhas é a habilidade de um circuito e/ou um sistema continuar a execução correta das suas tarefas (sem degradação de desempenho), mesmo diante da ocorrência de falhas em seu *hardware* e/ou *software*, evitando assim prejuízos físicos e materiais (KOREN; KRISHNA, 2007).

A abordagem tradicional para obtenção do requisito confiabilidade em sistemas de computação está fortemente relacionada com a prevenção de falhas. A prevenção de falhas em um sistema é realizada garantindo que todas as possíveis causas de defeitos sejam removidas, antes que o sistema entre em operação. Portanto, técnicas de tolerância a falhas são usadas em sistemas de computação como forma de lidar com condições de erro que podem surgir durante suas vidas operacionais. Estas condições de erro podem resultar tanto de fatores externos quanto internos, sendo o relacionamento entre causa e efeito melhor entendido sob a luz de três conceitos básicos: falhas, erros e defeitos (ANDERSON; LEE, 1981; GAMA, 2008; BOLZANI, 2005; PRADHAN, 1996).

2.2 Falhas, Erros e Defeitos

As falhas podem ocorrer tanto em nível de *hardware* quanto de *software*, sendo estas a causa dos erros. Componentes envelhecidos e interferências externas são exemplos de fatores que podem levar um sistema à ocorrência de falhas. As falhas de *hardware* podem ser classificadas em permanentes, transientes e intermitentes (ANDERSON; LEE, 1981; KOREN; KRISHNA, 2007).

As falhas permanentes ocorrem no meio físico e são provocadas através de falhas no processo de fabricação e/ou pelo envelhecimento dos componentes do sistema. Curtos circuitos, nós abertos e *stuck-at* são exemplos de falhas permanentes. Esta última é um tipo de falha onde um nó do circuito está sempre no mesmo nível lógico seja ele zero (*stuck-at-zero*) ou um (*stuck-at-one*).

Outro tipo de falhas são as transientes, que ocorrem durante a vida útil dos componentes e são provocadas por adversidades e/ou fenômenos ambientais aleatórios onde o sistema está implementado. Variações da tensão de alimentação e interferências eletromagnéticas são exemplos de falhas transientes. Um tipo de falha transiente bastante comum, são as do tipo *bit flip* (inversão de bits), ocasionadas por interferências externas, que resultam em uma mudança temporária no nível lógico em um determinado elemento armazenador de um circuito. Esta mudança pode ocorrer em ambos os sentidos de zero para um ou de um para zero.

Existem ainda as falhas intermitentes que são caracterizadas pela ocorrência temporária e cíclica do erro a partir de variações das condições externas e/ou ambientais do sistema; vibrações e variações da temperatura são exemplos de causas de falhas intermitentes.

Uma falha em um circuito lógico refere-se a um mau funcionamento físico e/ou lógico que causa um evento indesejável. O interesse maior é atender satisfatoriamente as especificações de projeto e as necessidades dos usuários de um sistema. Um defeito é definido como um desvio de especificação. Define-se que um sistema está em estado errôneo, ou em erro, se o processamento posterior, a partir deste estado, pode levá-lo a um defeito. Finalmente, define-se falha ou falta como a causa física do erro (ANDERSON; LEE, 1981; SHOOMAN, 2002).

Com base nas definições de falhas, mau funcionamentos, erros e defeitos, torna-se evidente que são todos eventos indesejáveis em um sistema computacional e, de acordo com o contexto, referem-se a diferentes domínios. Na Figura 2.1 é apresentada um diagrama simplificado, sugerido por (PRADHAN, 1996), para ilustrar os conceitos de falha, erro e defeito. Neste diagrama, as falhas estão associadas ao universo físico, os erros ao universo da informação e os defeitos ao universo do usuário.

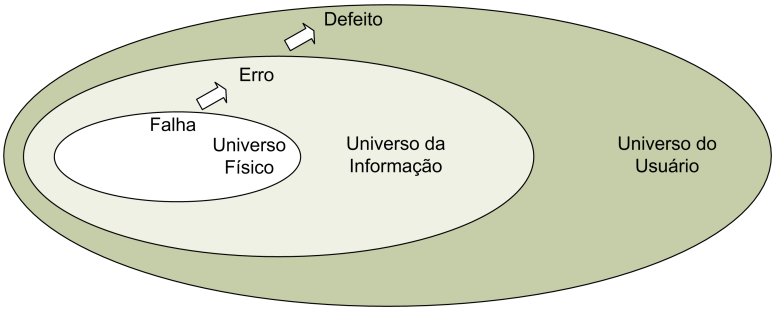

Figura 2.1: sequência de Falha, Erro e Defeito, adaptado de (PRADHAN, 1996)

2.3 Prevenção de Falhas

Tomando-se como base que falhas são inevitáveis, pois componentes físicos envelhecem e/ou sofrem com interferências externas, sejam ambientais ou humanas, surge a necessidade de prevenção de falhas. Neste caso, todos os esforços devem ser concentrados na eliminação das possíveis fontes de falhas, antes que um sistema entre em operação. Isso garante que, para possíveis falhas conhecidas, um sistema já estará apto a tratá-las e, por conseguinte, continuar com seu correto funcionamento. De um modo geral, a prevenção de falhas é utilizada em conjunto com técnicas de tolerância a falhas.

2.4 Estratégias de Tolerância a Falhas

No caso de sistemas que requeiram o emprego de tolerância a falhas, o projetista deve enxergar o sistema de modo mais realístico, ou seja, considerando as falhas como eventos normais que devem ser tratados. Nesse sentido, um sistema tolerante a falhas é aquele que, sem nenhuma intervenção humana, é capaz de lidar com falhas operacionais e de projeto, e de manter o desempenho do sistema dentro de suas especificações (CASTRO, 1992).

A estratégia mais comum em sistemas tolerantes a falhas é o uso de redundância em diferentes níveis, *software* e/ou *hardware*, e/ou temporal. O uso de recursos redundantes empregados nesses sistemas geralmente não tem como objetivo uma melhoria de desempenho e sim ajudar um sistema a contornar eventuais falhas, mascarando ou contornando os erros. A redundância em sistemas tolerantes a falhas pode ser classificada em: *hardware*, *software*, e temporal.

2.4.1 Redundância de *hardware*

A redundância de hardware consiste do uso de circuitos adicionais para detectar e corrigir erros e pode ser dividida em dois tipos: estática e dinâmica.

2.4.1.1 Redundância de *hardware estática*

Na redundância estática ou mascaramento, componentes adicionais são utilizados para mascarar os efeitos de uma falha, normalmente utilizando-se componentes em paralelo. Uma vez que os efeitos das falhas são mascarados, a detecção da falha pode ser difícil quando esta técnica é aplicada em nível de componentes (CASTRO, 1992). Nesse sentido, seu uso em sistemas reparáveis pode vir a ser indesejável, uma vez que o processo de isolamento do componente falho deve ter que contar com outros mecanismos para localizá-lo. Entretanto, como a falha não é sentida pelo sistema, esta técnica pode ser muito atrativa em aplicações de tempo real, uma vez que não é necessário tempo adicional para detectar e isolar a falha.

Um exemplo típico do uso desse tipo de redundância é a Redundância Modular Tripla (TMR - *Triple Modular Redundancy*). Na Figura 2.2 é mostrado um sistema TMR onde três cópias de um módulo são usadas em lugar de um único módulo. A saída de cada módulo passa por um circuito votador que deixa fluir somente a versão correta da informação, isto é, aquela fornecida por pelos menos dois dos módulos. Os resultados de

cada operação são executados em paralelo pelos módulos e são verificados pelo votador, na base de "a maioria vence".

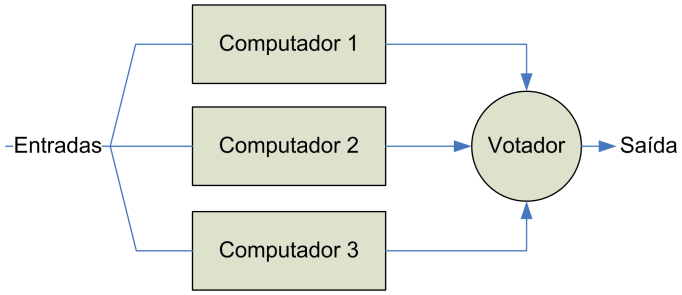

Figura 2.2: um típico sistema TMR.

Uma extensão de sistemas TMR é a redundância modular NMR (*N-uple Modular Redundancy*), mostrada na Figura 2.3. Nesse caso, existem (2n+1) módulos redundantes e, pelo menos (n+1) módulos são requeridos para concordar em um resultado; assim, esta estrutura é capaz de tolerar falhas em n módulos simultaneamente (ANDERSON; LEE, 1981; KOREN; KRISHNA, 2007).

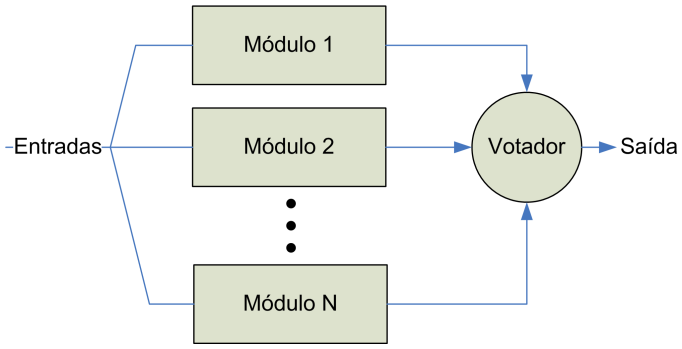

Figura 2.3: um típico sistema NMR.

Um outro exemplo de redundância estática é o uso de códigos de correção de erros em unidades de memória, na qual é necessária a utilização adicional de hardware e informações, como por exemplo, códigos de correção de *Hamming* (HAMMING, 1950), código *Reed-Solomon* (REED; SOLOMON, 1978) bem como o código *Matrix* (ARGYRIDES; ZARANDI; PRADHAN, 2007).

2.4.1.2 Redundância de *hardware* dinâmica

Em contraste com a redundância de hardware estática, na redundância de *hardware* dinâmica, os módulos redundantes existem como unidades de reserva, conforme mostrado na Figura 2.4, e podem substituir um módulo falho, automaticamente ou manualmente. Um módulo reserva pode ainda estar nos seguintes estados: ativo (ligado) ou passivo

(desligado). No caso de redundância dinâmica, a falha tem que ser detectada primeiro, uma vez que esta não é mascarada.

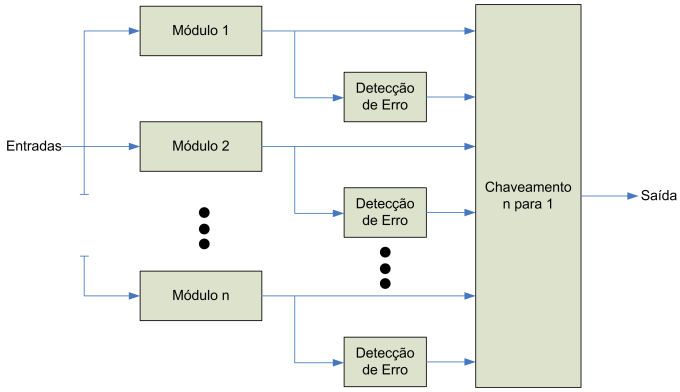

Figura 2.4: um típico sistema *stand-by*.

2.4.2 Redundância de *software*

A Redundância de *software* é usada para ajudar um sistema a superar tanto falhas de *hardware* como de *software*, consistindo de macro e micro programas e instruções extras classificadas em dois tipos de redundância de *software*: estática e dinâmica.

Na redundância de *software* estática, programas replicados são executados concorrentemente em máquinas separadas e seus resultados são comparados por votação (uma versão de *software* de um NMR). Um exemplo de tal redundância é a programação com N-versões (HANMER, 2007; FUHRMAN; CHUTANI; NUSSBAUMER, 1995). A técnica de programação N-versões, também chamada de programação diversitária, foi sugerida por Elmendorf (ELMENDORF, 1972) e desenvolvida por Avizienis e Chen (CHEN; AVIZIENIS, 1978).

Na redundância de *software* dinâmica, uma cópia do estado de um módulo é registrado (salvo) periodicamente em alguns pontos do programa, conhecidos como pontos de verificação (*checkpoints*), durante a operação do sistema. A recuperação de um erro é alcançada retrocedendo a execução do programa (*rolling back*) para o último ponto de verificação, onde o status do programa e toda a informação necessária para uma recuperação bem sucedida está armazenada (HANMER, 2007).

2.4.3 Redundância de tempo

Neste caso a redundância é implementada através da repetição de operações e é normalmente usada para detectar falhas transientes (LALA, 1985; KWAI; PARHAMI, 1996).

Exemplos dessa técnica são: re-execução (*rollback*) de instruções, segmentos de programas ou programas inteiros e re-envio de mensagens em um sistema distribuído. Em operações *rollback*, a re-execução de programas começa de pontos pré-definidos em um programa (*checkpoints*), conforme explicado na seção anterior. O microprocessador 8051 tolerante a falhas proposto por Lima (KASTENSMIDT; CARRO; REIS, 2006) e o processador LEON3-FT proposto por Gaisler (GAISLE, 2002), são exemplos de sistemas, implementados em FPGAs, que fazem extensivo uso desta técnica.

2.4.4 Detecção de falhas

O processo de detecção de falhas é o ponto de partida para qualquer técnica de tolerância a falhas. De um modo geral, quanto mais detecção de falhas em um sistema, mais confiável ele será, na medida em que essas falhas possam ser recuperáveis ou mascaráveis. Dependendo da aplicação, não é factível equipar o sistema com um grande número de facilidades para detecção de falhas, sobretudo devido ao custo e *overheads* impostos por extensivas verificações.

Enquanto as técnicas de detecção de falhas podem variar de sistema para sistema, a maioria delas pode ser situada em um dos seguintes casos apresentado por (KOREN; KRISHNA, 2007): verificação por replicação, temporização e codificação.

2.4.4.1 Verificação por replicação

Verificação por replicação consiste da replicação da atividade do sistema associada a algum método de comparação de suas saídas. Esta é uma das medidas mais completas para se detectar falhas em um sistema de um computador (KOREN; KRISHNA, 2007). É normalmente usada com o objetivo de detectar falhas no *hardware*, e pode ser implementada tanto em nível de circuito como em nível de subsistema. Devido a grande importância da unidade que compara as saídas dos módulos replicados e por sua função ser muito crítica, algumas vezes esta também é replicada. Um exemplo é o votador em alguns sistemas TMR.

Um outro exemplo de um sistema empregando tal técnica é o microprocessador 8051 tolerante a falhas desenvolvido por Lima (KASTENSMIDT; CARRO; REIS, 2006), no qual algumas partes do microprocessador são triplicadas, e suas saídas são checadas por votadores antes que sejam transmitidas. Outro sistema que emprega verificação por replicação é o processador LEON3-FT proposto por Gaisler (GAISLE, 2002).

2.4.4.2 Verificação por temporização

Verificação por temporização é particulamente eficaz na detecção de erros no *software* em programas replicados (HANMER, 2007). Nessa técnica, a execução de uma operação não deve exceder um tempo máximo pré-determinado, caso contrário uma exceção de *timeout* (tempo ultrapassado) é gerada para indicar que um defeito ocorreu.

Esta técnica faz uso de um contador que pode ser implementado tanto por *hardware* como por *software*. Quando um temporizador de *hardware* é utilizado, este é chamado de temporizador cão de guarda (*watchdog timer*) e tem que ser periodicamente zerado pelo programa em execução no processador. Caso isso não ocorra, devido a um desvio indesejável da execução do programa, como por exemplo, o temporizador sinaliza a ocorrência do erro e reinicializa o sistema.

2.4.4.3 Verificação por codificação

A técnica de verificação por codificação usa uma representação redundante de objetos como forma de detectar erros. Esta representação mantém um relacionamento fixo com a representação original (não redundante) do objeto, e uma exceção deve ser levantada se esse relacionamento for corrompido, como resultado de um erro (ANDERSON; LEE, 1981; KOREN; KRISHNA, 2007).

O exemplo mais comum de verificação por codificação é o uso de checadores de paridade, em que um *bit* de paridade está associado com um número de *bits*, normalmente uma palavra, e seu relacionamento é tal que a soma módulo 2 de todos os bits é 0 (zero), paridade par, ou 1 (um), paridade ímpar. A principal desvantagem de checadores de paridade está no fato que estes não podem detectar erros resultantes de falhas que invertam um número par de *bits*. Para superar esse tipo de falha, outros esquemas de codificação são adotados, por exemplo: código de *Hamming*, checagem por redundância cíclica-CRC (CASTRO; COELHO; SILVEIRA, 2008), códigos aritméticos (*checksums*) (FUJIWARA E.T.; PRADHAN, 1990), *Reed-Solomon* (REED; SOLOMON, 1978) e *Matrix* (ARGYRIDES; ZARANDI; PRADHAN, 2007).

2.5 Avaliação Quantitativa da Confiabilidade

Os requisitos de confiabilidade para sistemas de computadores têm se tornado mais rígidos à medida que as aplicações assumem características de sistemas críticos. O estabelecimento do programa espacial desempenhou um papel importante nos avanços da

pesquisa no campo de confiabilidade em sistemas de computadores (CASTRO, 1992). Uma vez que missões espaciais, usualmente, contam enormemente com o funcionamento adequado de seus computadores embarcados, esses computadores são projetados tomando-se os aspectos de confiabilidade em alta consideração.

A confiança colocada nos sistemas de computadores pode ser expressa como a probabilidade requerida para que o sistema funcione continuamente, ao passo que a confiabilidade $R(t)$ é a probabilidade condicional de que um sistema funcione corretamente por um período de tempo de 0 a t, dado que este estava operacional no tempo $t = 0$.

Essa confiança é normalmente expressa como a probabilidade mínima necessária para que um sistema funcione continuadamente, fornecendo o serviço especificado, durante um intervalo de tempo fixo. Um dado sistema crítico é conveniente para determinada aplicação se sua confiabilidade inerente é maior do que a confiabilidade requerida. Às vezes, entretanto, sua confiabilidade é menor do que a requerida, então, surge a necessidade de provê-lo com alguns mecanismos para aumentar sua confiabilidade. Neste sentido, faz-se necessário o uso de métodos de avaliação de confiabilidade para ajudar o projetista a decidir onde e como empregar técnicas de tolerância a falhas.

Existem várias formas de avaliação quantitativa relacionada com a confiabilidade de sistemas, incluindo: taxa de falhas, Tempo Médio para Falhar (MTTF), Tempo Médio para Reparo (MTTR), Tempo Médio entre Falhas (MTBF), cobertura de falhas e análise de confiabilidade.

2.5.1 Taxa de falha e função de confiabilidade

A taxa de falhas de um componente é o número esperado de falhas de um tipo de dispositivo em um período de tempo e é usualmente denotada como λ. Se, por exemplo, um componente falha, em média, uma vez a cada 1000 horas, este componente possui uma taxa de falhas média de uma falha por 1000 horas, ou 1/1000 falhas/hora (CASTRO, 1992).

A taxa de falhas de um componente e/ou dispositivo é mensurada em defeitos por unidade de tempo e é diretamente proporcional ao tempo de vida do componente e/ou dispositivo. Uma representação usual para a taxa de falhas de componentes de *hardware* é dada pela curva da banheira, mostrada na Figura 2.5. Nesta curva podem-se distinguir três fases, a primeira fase é a da mortalidade infantil, na qual componentes fracos e mal fabricados falham no início da operação. A segunda fase, denomina-se vida útil, na qual as taxas de defeitos são constantes, ou seja, as falhas são de natureza aleatória e pouco se

pode fazer para evitar. Por último tem-se a fase de envelhecimento, neste caso inicia-se o término da vida útil do componente e a taxa de falhas cresce continuamente, causadas principalmente pelo desgaste e envelhecimento dos componentes.

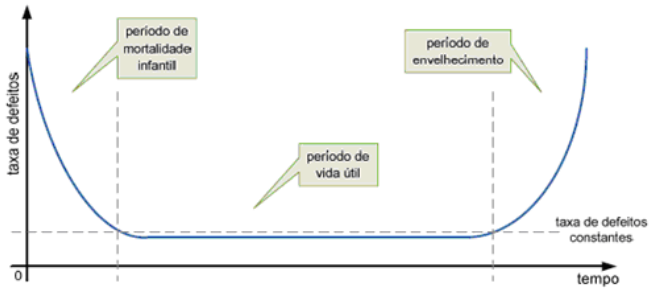

Figura 2.5: curva da banheira mostrando a taxa de falhas de um componente em relação ao tempo.

Os componentes de *hardware* somente apresentam taxa de falhas constante durante um período de tempo chamado de vida útil, que segue após uma fase com taxa de defeitos decrescente chamada de mortalidade infantil. Para acelerar a fase de mortalidade infantil, os fabricantes recorrem à técnica de *burn-in*, na qual é efetuada a remoção de componentes defeituosos sendo estes substituídos por componentes que já sobreviveram à fase de mortalidade infantil através do processo de aceleração de operação (NG; LOW; DEMIDENKO, 2008).

Quando a taxa de falhas é constante, uma distribuição exponencial negativa é utilizada para representar a função de confiabilidade, e tem a a forma (CASTRO, 1992):

$$R(t) = e^{-\lambda t}, \tag{2.1}$$

em que λ é a taxa de falha e t é o tempo. A relação entre a confiabilidade e o tempo, conforme a equação 2.1, é conhecida como *lei exponencial de falha*, e é a relação mais usada para modelar a função de confiabilidade de sistemas.

Quando a análise de confiabilidade é feita, usualmente assume-se que em $t = 0$ todos os componentes estão operacionais, e que em $t = \infty$ todos os componentes devem falhar, logo: $R(0) = 1$ e $R(\infty) = 0$.

O tempo inicial $t = 0$ pode ser qualquer tempo durante o intervalo no qual a taxa de falha do sistema é constante. Isso é uma consequência da suposição de que a taxa de falhas de cada componente do sistema é constante nesse período. A estimativa da taxa de falhas de um componente é um ponto crucial quando se avalia a confiabilidade de um sistema.

2.5.2 MTTF - Tempo Médio para Falhar (*Mean Time to Failure*)

O tempo médio para falhar, comumente conhecido como MTTF (*Mean Time to Failure*), é o tempo esperado que um sistema deve operar antes da ocorrência da primeira falha e sua expressão em função da confiabilidade $R(t)$ é dada por (SHOOMAN, 2002):

$$MTTF = \int_0^\infty R(t)dt, \quad para \quad R(\infty) = 0. \tag{2.2}$$

Para uma função de distribuição exponencial, substituindo-se a equação 2.1 em 2.2, a expressão se resume a:

$$MTTF = \int_0^\infty e^{-\lambda t}dt = \frac{1}{\lambda}. \tag{2.3}$$

2.5.3 MTTR - Tempo médio para Reparo (*Mean Time to Repair*)

O tempo médio para reparo MTTR (*Mean Time to Repair*) é o tempo médio necessário para reparar um sistema. Pela definição, o MTTR depende da habilidade do pessoal de manutenção para reparar o sistema em falha. O MTTR é usualmente expresso em termos da taxa de reparo μ dada por (SHOOMAN, 2002)

$$MTTR = \frac{1}{\mu} \tag{2.4}$$

em que μ é o número médio de reparos por período de tempo.

2.5.4 MTBF - Tempo Médio entre Falhas (*Mean Time Between Failures*)

O tempo médio entre falhas MTBF (*Mean Time Between Failures*) é o tempo associado com a próxima falha, supondo que uma já ocorreu. Após a ocorrência da falha, o sistema deve experimentar um período de tempo sob reparo que depende de seu MTTR. Uma vez que o sistema está novamente operacional, a próxima falha deve ocorrer de acordo com o seu MTTF. Assim, o MTBF pode ser calculado desta foma:

$$MTBF = MTTF + MTTR \tag{2.5}$$

2.5.5 Cobertura da falha

A cobertura da falha é um parâmetro associado com algumas das fases relacionadas com as estratégias de tolerância a falhas: detecção, localização, confinamento e recuperação. Nesse sentido, quatro tipos de coberturas podem ser identificadas: cobertura da detecção, cobertura da localização, cobertura do confinamento e cobertura da recuperação. A cobertura associada com cada fase é a medida do sucesso no desempenho das operações relacionadas com cada uma. Assim, por exemplo, a cobertura de detecção é a medida da habilidade de um sistema detectar falhas. Uma vez que o sucesso de um sistema se recuperar de uma dada falha está associado com seu sucesso na detecção, localização e confinamento da falha. A cobertura de recuperação é usualmente utilizada para expressar a cobertura de uma falha (SHOOMAN, 2002). A expressão matemática para a cobertura de falha de um sistema é dada por:

$$C = P(\text{recuperação da falha / ocorrência da falha}), \tag{2.6}$$

e é lido como a probabilidade de se recuperar de uma falha, dado que esta tenha ocorrido.

2.5.6 Estimativa da confiabilidade

A estimativa da confiabilidade de um sistema pode ser obtida usando basicamente dois métodos: experimental e analítico. No método experimental um conjunto de N objetos, cuja confiabilidade deve ser estimada, é colocado em operação por um período de tempo t. O número de objetos que falham N_f durante esse período é então observado. O número de sobreviventes é $N - N_f$ e a confiabilidade dos objetos pode ser estimada como:

$$R(t) = \frac{N - N_f}{N}. \tag{2.7}$$

Duas dificuldades se apresentam com esse método. A primeira é o fato de que o número de objetos requeridos para executar um experimento deve ser consideralvelmente grande. A segunda dificuldade está associada com o período de tempo do experimento. O método analítico, por outro lado, usa modelos matemáticos a fim de obter a confiabilidade do sistema. Uma técnica analítica bastante utilizada é a modelagem combinatorial (SHOOMAN, 2002).

2.5.7 Modelagem combinatorial

Na modelagem combinatorial, um sistema é dividido em módulos independentes. Cada módulo é assumido possuir uma probabilidade de estar operacional P_i, ou uma confiabilidade em função do tempo $R_i(t)$. O objetivo é determinar a probabilidade P_{sys}, ou a função $R_{sys}(t)$ para o sistema operar corretamente. Nesta modelagem, algumas propriedades devem ser assumidas:

1. as falhas nos módulos são independentes;
2. uma vez que um módulo falhou, é assumido que este sempre gere resultados incorretos;
3. o sistema é considerado falho se não satisfizer os requisitos mínimos de funcionalidade;
4. uma vez que um sistema entrou em estado falho, falhas subsequentes não podem retorná-lo ao seu estado funcional.

É comum, quando usando essa técnica, descrever um sistema como uma coleção de componentes que são conectados em série, paralelo, malha, ou uma combinação desses. O modo como os componentes são conectados no modelo está relacionado com suas contribuições para mantê-lo operacional. Entretanto, esse relacionamento pode diferir substancialmente da topologia real do sistema em análise, e é somente usado para deduzir os eventos cujas ocorrências o deixam operacional.

Dois ou mais componentes estão em série, no contexto de análise de confiabilidade, quando todos devem funcionar corretamente a fim de que um dado sistema se matenha operacional. Assim, se um componente falhar, o sistema deve falhar. A confiabilidade de um sistema série, contendo n componentes é dada por:

$$R_s = R_1 * R_2 * R_3 * ..... * R_n, \tag{2.8}$$

em que R_s é a confiabilidade do sistema e R_n é a confiabilidade do componente n.

Um conjunto de componentes, de um sistema, é dito estar em paralelo se um deles é necessário funcionar corretamente para que este sistema se mantenha operacional. Assim, todos os componentes têm que falhar para que o sistema falhe. A confiabilidade de um sistema paralelo, contendo n componentes é dada:

$$R_p = 1 - \prod_{i=1}^{n}(1 - R_i(t)), \tag{2.9}$$

em que R_p é a confiabilidade do sistema e R_i é a confiabilidade do componente i.

O conceito de redundância, visto neste Capítulo, está diretamente associado com o uso de componentes e/ou subsistemas em paralelo, como forma de mascarar as falhas em um sistema. Entretanto, o uso de redundância nem sempre aumenta a confiabilidade de um sistema. O aumento da confiabilidade introduzida pelo uso de redundância depende da quantidade de elementos redundantes empregados e da confiabilidade das partes redundantes utilizadas no sistema (SHOOMAN, 2002).

De um modo geral, sistemas usados na prática são tipicamente uma combinação de subsistemas série e paralelo (SHOOMAN, 2002). Para um sistema obedecendo a lei exponencial de falhas, a confiabilidade de um componente n é:

$$R_n = e^{-\lambda_n t}, \tag{2.10}$$

em que λ_n é a taxa de falhas do componente n. As equações (2.8) e (2.9) podem ser re-escritas respectivamente como

$$R_s = e^{-\lambda_1 t} * e^{-\lambda_2 t} * e^{-\lambda_3 t} * ..... * e^{-\lambda_n t}, \tag{2.11}$$

$$R_p = 1 - \prod_{i=1}^{n}(1 - e^{-\lambda_i t}). \tag{2.12}$$

Em particular, se os sistemas acima fossem constituídos de n subsistemas idênticos, suas funções de confiabilidade são, respectivamente

$$R_s = e^{-n\lambda_n t}, \tag{2.13}$$

$$R_p = 1 - (1 - e^{-\lambda_n t})^n \tag{2.14}$$

e seus respectivos MTTF dados por

$$MTTF_s = \frac{1}{(\lambda_1 + \lambda_2 + \lambda_3 + ... + \lambda_n)}, \tag{2.15}$$

$$MTTF_p = \frac{ln(n)}{\lambda}. \tag{2.16}$$

2.6 Funções de Confiabilidade de Sistemas

Nessa seção é apresentada uma avaliação de confiabilidade relacionada com os sistemas TMR, além de algumas de suas variações mais comuns. A análise das configurações mostradas a seguir supõe que cada módulo individualmente obedece à lei exponencial de falha, ou seja, possui uma taxa de falhas constante λ. Além disso, supõe-se que todos os módulos estão inicialmente no estado operacional.

2.6.1 Configuração TMR

Um sistema TMR opera corretamente quando pelo menos dois dos três módulos estão operacionais. Considerando inicialmente que nenhum reparo é previsto, pode-se enumerar todos os estados do sistema conforme mostrado na equação 2.17 a seguir

$$(R+F)^3 = R^3 + 3R^2F + 3RF^2 + F^3, \tag{2.17}$$

em que R e F são respectivamente, a confiabilidade e a não-confiabilidade de um único módulo. Escolhendo somente aqueles estados que se enquadram na definição de um sistema TMR, alguns termos são eliminados e a equação 2.17 pode ser reescrita como:

$$R_{tmr} = R^3 + 3R^2F = R^3 + 3R^2(1-R) = 3R^2 - 2R^3, \tag{2.18}$$

em que R_{tmr} é a confiabilidade de um sistema TMR.

Considerando que cada módulo obedece á lei exponencial de falhas, a confiabilidade do sistema TMR em função do tempo R_{tmr} e o seu $MTTF_{tmr}$ são dados respectivamente por:

$$R_{tmr}(t) = 3e^{-2\lambda t} - 2e^{-3\lambda t}, \tag{2.19}$$

$$MTTF_{tmr} = \int_0^\infty (3e^{-2\lambda t} - 2e^{-3\lambda t})dt = \frac{5}{6\lambda}. \tag{2.20}$$

2.7 Conclusão

Como pode-se observar, sistemas que fazem uso das técnicas de tolerância a falhas por replicação, tendem a ser mais confiáveis que sistemas que não fazem o uso desta. Neste

sentido, muitos sistemas computacionais tolerantes a falhas utilizam em partes de sua estrutura, replicação por TMR. É o que acontece, por exemplo, em sistemas computacionais de missão espacial, pois nestes os altos índices de radiação podem ocasionar falhas nos circuitos integrados que compõem estes sistemas. No capítulo seguinte é apresentado um estudo dos efeitos da radiação sobre os circuitos integrados digitais.

Capítulo 3

Efeitos da Radiação Sobre os Semicondutores

Os equipamentos de tecnologia da informação e comunicação estão presentes no cotidiano do homem, seja quando utiliza um celular para se comunicar ou um *laptop* para ler mensagens, dentre outros exemplos existentes. Os dispositivos semicondutores, tais como transistores, diodos e circuitos integrados são necessários para a construção de equipamentos eletrônicos. Destes dispositivos, os circuitos integrados merecem atenção especial por terem embutido, em um único chip, vários circuitos eletrônicos.

3.1 Introdução

O constante avanço da tecnologia *CMOS - Complementary Metal Oxide Semiconductor*, tem contribuído para se aumentar a densidade de integração e a velocidade de operação dos circuitos digitais. Isto tem causado uma diminuição drástica das dimensões dos transistores. Diante disso, alguns ajustes no processo de fabricação *CMOS* têm sido necessários, dentre estes a redução da tensão de limiar (*threshold*) e da tensão de alimentação. Tais ajustes, aliados aos efeitos causados pela redução das dimensões dos transistores, como a diminuição das capacitâncias dos nós internos, têm tornado os circuitos cada vez mais suscetíveis a falhas transientes, também referenciadas na literatura como *soft errors* (KEANE et al., 2010; IROM, 2002; MAVIS; EATON, 2002).

Atualmente, a principal fonte de falhas transientes nos circuitos integrados é a radiação (BAUMANN, 2001; BAUMAN, 2005). No espaço, a atividade solar é a principal responsável pela radiação, a qual é constituída por partículas carregadas (ou energéticas), tais como elétrons, prótons ou íons pesados, ou de radiação eletromagnética, a qual pode

ser constituída por raios X, raios gama ou luz ultravioleta (BARTH, 1997; BAUMANN, 2001).

A influência da radiação nos materiais é medida através da energia e do fluxo de partículas. Este fluxo corresponde ao número de partículas por segundo que atravessam um material em uma área de $1cm^2$ $[1/s.cm^2]$. Quanto mais próximo da superfície da terra, menor é o fluxo e a energia das partículas provenientes da radiação espacial (NORMAND, 1996).

Aparentemente, a preocupação com o funcionamento de circuitos integrados operando na superfície da terra parece um tanto distante. Porém, o constante avanço da tecnologia CMOS tem tornado o tamanho dos transistores cada vez menor. Dessa forma, as capacitâncias internas a estes também têm reduzido seus valores. O resultado deste processo é uma maior vulnerabilidade dos circuitos integrados a falhas provocadas por partículas de menor intensidade energética, justamente as mais frequentes na superfície da terra (NICOLAIDIS, 1999; JOHNSTON, 2000).

Na superfície da terra, três tipos de mecanismos são responsáveis pela geração de partículas energéticas, entre estas partículas podem-se citar as partículas alfa, os raios cósmicos de alta energia e os raios cósmicos de baixa energia (BAUMAN, 2005; BORKAR, 2005).

As partículas alfa, tratando-se de circuitos digitais encapsulados, são emitidas por impurezas tais como Urânio e Tório presentes no próprio encapsulamento do circuito. Já a interação entre raios cósmicos de alta energia e átomos na atmosfera, é responsável pela geração de nêutrons de alta e de baixa energia. Os nêutrons de alta energia produzem partículas carregadas nas colisões com o material atingido. Por outro lado os nêutrons de baixa energia, geram partículas energéticas quando interagem com o Boro, elemento muito utilizado na dopagem de material tipo **P** no silício.

Vários fatores devem ser considerados para que a ação de uma partícula venha a se tornar efetivamente uma falha. Um destes fatores é a coleta de carga (*Charge collection*) ou Q_{col}, que se refere à quantidade de carga depositada no silício quando da ação de uma partícula. Outro importante fator é chamado de carga crítica (*Critical Charge*) ou Q_{crit}, que corresponde à sensibilidade do silício quanto ao excesso de cargas depositadas. Estas cargas dependem também de vários outros fatores. Para que um *soft-error* realmente se caracterize como uma falha transiente, Q_{coll} deve ser maior que Q_{crit}. Para se ter uma idéia, a colisão de um íon com o substrato de silício produz um par elétron lacuna para cada $3,6eV$ (eletron Volt) de energia perdida pelo íon (BAUMAN, 2005).

Normalmente, utiliza-se o termo *LET - Linear Energy Transfer* para medir a quantidade de energia perdida pela partícula por unidade de comprimento, quando esta atravessa um material (BAUMAN, 2005; DODD; MASSENGILL, 2003), cuja unidade de medida é $MeV/(mg/cm^2)$. O *LET* mínimo que pode causar um *bit-flip*, ou seja um *upset*, é chamado de *LET threshold* (LET_{th}) (MCNULTY et al., 1999; SCHWANK et al., 2010). Há muitos níveis de robustez, de acordo com a quantidade de fluxo de energia transferida para o silício que pode manter o circuito funcionando corretamente. Em média, as aplicações espaciais, por exemplo que operam em órbita baixa, e aplicações militares devem ser robustas para LETs superior a $40MeV/(mg/cm^2)$ (KASTENSMIDT; CARRO; REIS, 2006).

Ao contar o número de *upsets* e conhecendo quantas partículas atravessam um dado circuito, pode-se calcular a probabilidade de uma particular partícula provocar uma virada de *bit* (*upset*). Esse número resultante, que é o número de *upsets* dividido pelo número de partículas por cm^2 que causam *upsets*, é chamado de seção transversal, conhecido também como seção de choque e é mensurado em unidades de cm^2/componente. Por conseguinte, a sensibilidade de um dispositivo a um *upset* é medida por uma função da seção transversal σ em termos da LET. Na Figura 3.1 é apresentado um exemplo da curva da seção transversal σ por LET.

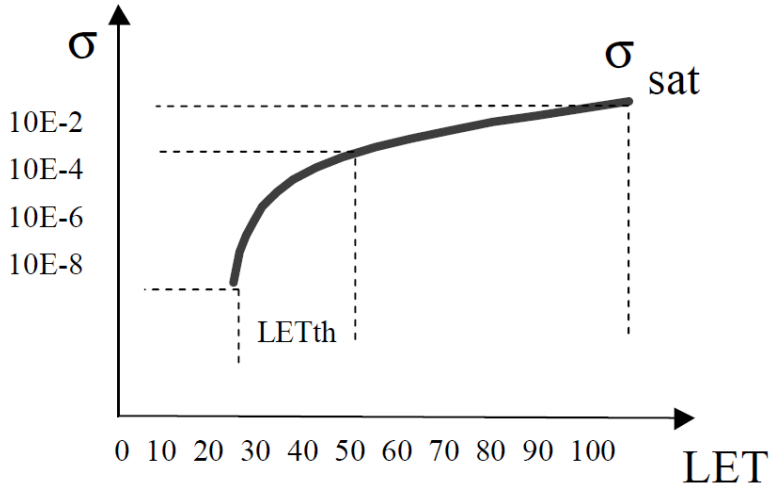

Figura 3.1: exemplo da curva da seção transversal por LET, adaptado de (SCHWANK et al., 2010).

Analisando a curva da Figura 3.1, pode-se afirmar que nenhum erro ocorre na presença de partículas com LET menor do que 25 MeV. Para partículas com 25 MeV, mais de 100.000.000 partículas devem percorrer a área sensível do circuito para disparar um *upset*. Para partículas com 50 MeV, um fluxo de 10.000 partículas por segundo é necessário para provocar um *upset*, assim como, um fluxo de 100 partículas por segundo com um LET de 100 MeV é necessário para desencadear um *upset*.

3.2 Os Efeitos da Radiação Sobre os Circuitos Integrados

Diversos efeitos podem ser observados quando ocorre a colisão de partículas com materiais semicondutores dos circuitos integrados. Os efeitos variam desde uma simples falha no funcionamento dos circuitos até mesmo um dano permanente nestes.

A junção de polarização reversa do dreno do transistor que se encontra desligado (estado *off*) é considerada a região mais sensível do transistor, devido às suas características internas de polarização (MESSENGER, 1982; BAUMAN, 2005; GADLAGE, 2004). Na Figura 3.2 é ilustrado o comportamento do silício quando da presença de uma partícula carregada.

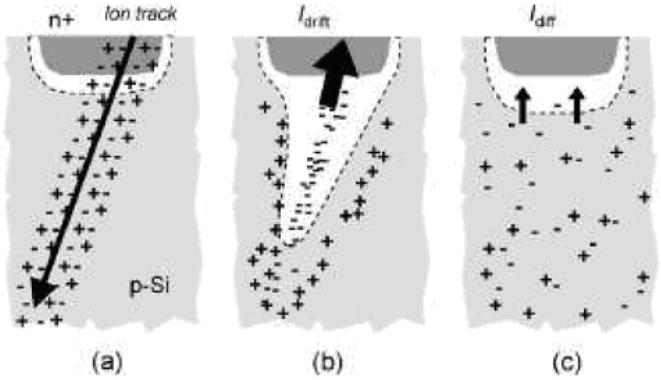

Figura 3.2: comportamento da região sensível de um transistor, dada a ação de uma partícula carregada, adaptado de (WESTE; HARRIS, 2010).

Quando a partícula choca-se com a região sensível do transistor, uma trilha de ionização, constituída por pares elétron-lacuna e uma alta concentração de portadores, é formada por onde a partícula passa, conforme ilustrado na Figura 3.2 (a). Devido à proximidade da trilha com a região de depleção da junção, é gerado então um campo magnético. Este campo por sua vez, faz com que a região de depleção tome a forma de um funil (WESTE; HARRIS, 2010; BAUMAN, 2005; MESSENGER, 1982) e as cargas negativas sejam atraídas rapidamente para a região positiva do silício (dreno), conforme ilustrado na Figura 3.2 (b). Após a região de depleção retomar ao seu formato natural por difusão, as cargas negativas continuam a se deslocar para a região positiva, mas de maneira mais lenta, conforme ilustrado na Figura 3.2 (c). Como resultado deste fenômeno, é gerado um pulso de corrente entre o dreno do transistor e o substrato ou poço, conforme o tipo de transistor (O'BRYAN, 2002). Caso essa corrente possua amplitude e duração suficientes para carregar ou descarregar um nodo de saída, um pulso de tensão transiente é gerado. Tal pulso de tensão pode ser interpretado como uma mudança de nível lógico, caso este

tenha uma amplitude maior do que a margem de ruído da porta subsequente (DODD; MASSENGILL, 2003). A quantidade de ionização e a intensidade da corrente gerada pela colisão de partículas carregadas com o silício são diretamente proporcionais à quantidade de carga liberada pela partícula (LIMA et al., 2002; KARNIK T.; HAZUCHA, 2004).

A forma do pulso de corrente gerado pela colisão de uma partícula pesada no silício é ilustrada na Figura 3.3. É possível verificar de maneira transparente as três fases da ação da partícula no silício.

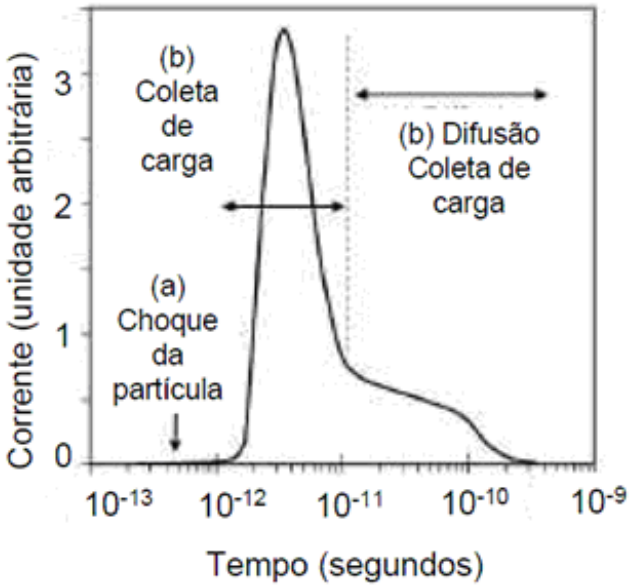

Figura 3.3: forma do pulso de corrente gerado após a colisão de uma partícula carregada com a região sensível do silício, adaptado de (DODD; MASSENGILL, 2003).

3.3 *Single Event Effects* (SEEs)

Fenômenos causados pela ação da colisão de partículas em circuitos integrados são também referenciados na literatura como SEEs - *Single Event Effects*. Dependendo das consequências causadas pela ação de uma partícula, estes podem ser divididos em três categorias (O'BRYAN, 2002; LIMA et al., 2002): *SEE* destrutivo, *Hard SEE* e *Soft SEE*.

3.3.1 *SEE* Destrutivo

O *SEE* destrutivo é caracterizado por um dano físico permanente no circuito causado pela ação de uma partícula. O SEE mais comun é chamado *Single Event Latchup* - SEL. Dada a ação de uma partícula, se a quantidade de energia desprendida da partícula

ultrapassar os limites de corrente suportados pelo transistor, uma falha permanente pode se deflagrar no circuito tornando-o defeituoso (WESTE; HARRIS, 2010; MCDONALD et al., 2000).

3.3.2 *Hard SEE*

Um desvio de comportamento permanente em uma das funcionalidades do circuito caracteriza um *hard error*. O fenômeno, neste caso, caracteriza-se como *SHE - Single Hard Error*. Um bit de memória com um valor fixo em "1"ou "0"é um exemplo deste tipo de caso (MCDONALD et al., 2000).

3.3.3 *Soft SEE*

As falhas provocadas pela colisão de partículas energizadas com as regiões sensíveis dos circuitos que não o danificam são chamadas de *Soft SEE*. A ação da partícula apenas causa um mau funcionamento momentâneo do circuito, de forma que, instantes depois, o circuito continua a funcionar normalmente. Esta característica de não permanência de falhas é a responsável por caracterizar os *SEEs* como *soft errors* (KARNIK T.; HAZUCHA, 2004).

3.3.3.1 *Single Event-Upset* (SEU)

Uma partícula carregada pode colidir diretamente com uma região sensível de um elemento de memória. Caso isto venha a acontecer, conforme a quantidade de energia liberada pela partícula, o valor armazenado na célula de memória pode vir a ser invertido, efeito este também chamado de *bit-flip*. A este fenômeno é atribuído o nome de *Single Event-Upset* ou *SEU* (WESTE; HARRIS, 2010; BAUMANN, 2001; DODD; MASSENGILL, 2003; NICOLAIDIS, 1999).

Considerando as tecnologias submicrônicas atuais, a probabilidade de ocorrência de *SEUs* é bem maior do que a probabilidade de ocorrência de *SETs*. Isto se deve principalmente ao fato das células de memória serem projetadas com transistores de dimensões mínimas ou próximas das mínimas (BAZE M.; BUCHNER, 1997). Assim, mecanismos de geração de *SEUs* e também técnicas para proteção contra *SEUs* têm sido muito investigadas nos últimos anos. Entretanto, conforme já explicitado anteriormente, com a evolução da tecnologia CMOS, os circuitos têm se tornado cada vez mais vulneráveis aos efeitos de colisões de partículas carregadas, de modo que a ocorrência de SETs também

tem aumentado significativamente nos últimos anos.

O comportamento de uma célula de memória *SRAM* (*Static Random Access Memory*) quando da colisão de uma partícula carregada é mostrado na Figura 3.4. Note que inicialmente o circuito está estável, conforme está mostrado na Figura 3.4 (a). Após uma partícula carregada colidir com o dreno do transistor PMOS (*P-channel Metal-Oxide-Semiconductor*) ($MP0$), um pulso de tensão é gerado na saída do transistor, conforme a Figura 3.4 (b). Isto faz com que seja carregada, mesmo que momentaneamente, a capacitância associada a este dreno, ligando assim o transistor que o controla. Levando-se em conta as características de projeto, típicas de qualquer tipo de elemento de memória, que utilizam dimensionamentos diferentes dos transistores e laços de realimentação, o resultado da carga do capacitor pode ser interpretado erroneamente como uma mudança de nível lógico. Dessa forma, acontece indesejavelmente uma troca do valor armazenado na célula de memória caracterizando um SEU, da forma como apresentado na Figura 3.4 (c).

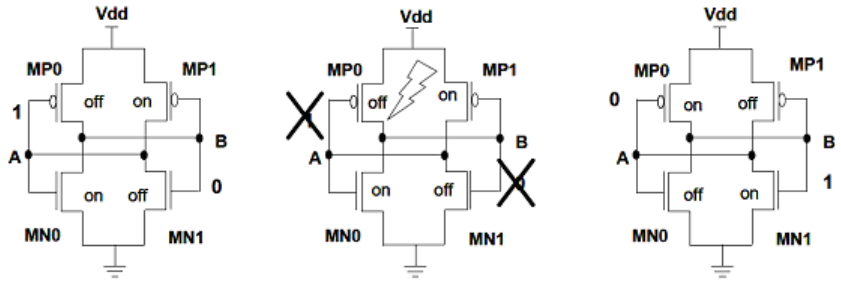

Figura 3.4: *single event upset* (SEU) em uma célula de memória SRAM, adaptado de (KASTENSMIDT; CARRO; REIS, 2006).

Embora SEU seja de maior interesse em aplicações espaciais, falhas transientes múltiplas também chamadas de *Multiple Bits Upsets* (MBUs), começam a ser importantes por causa das dimensões nanométricas dos transistores das tecnologias atuais (DODD; MASSENGILL, 2003; LIMA et al., 2002). Quando um único íon de alta energia passa pelo silício, pode energizar duas ou mais células de memória adjacentes (Reed, 1997). As MBUs podem ser induzidas por ionização direta ou por influência nuclear de outros átomos. É mais provável que a alta energia da partícula provoque falhas duplas, enquanto falhas múltiplas são causadas por um aumento no ângulo de incidência da partícula. Experimentos em memórias sob altos fluxos de prótons e íons mostraram que a probabilidade de falhas múltiplas provocadas por um único íon vem aumentando ao longo dos anos (Wrobel, 2001; Johansson, 1999; Buchner, 2000). Assim como SEU, a ionização direta ou recuo nuclear podem induzir MBUs, como é apresentado na Figura 3.5 (KASTENSMIDT;

CARRO; REIS, 2006).

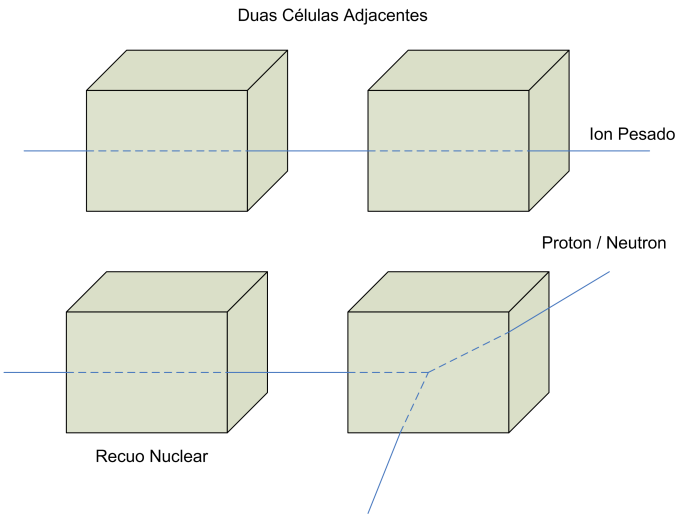

Figura 3.5: MBU provocado por uma simples partícula, adaptado de (KASTENSMIDT; CARRO; REIS, 2006).

Em Reed (REED, 1997), experiências em memórias atravessadas por fluxos de prótons e íons pesados, mostram múltiplos *upsets* provocados por um único íon. Durante estas experiências, foi observado MBUs em vários ângulos de incidência para LET superior a 25 $MeV/(mg/cm^2)$.

O resultado destas experiências é a classificação das MBUs em três tipos. O primeiro tipo de MBU ocorre quando uma única partícula atinge dois nós adjacentes, localizados em duas células de memória distintas. Este evento é classificado como um efeito de segunda ordem. Este tipo de MBU pode ser evitado em nível de posicionamentos específicos. O segundo tipo de MBU ocorre quando uma única partícula atinge dois nós adjacentes, localizados na mesma célula de memória. Este evento é classificado como um efeito de terceira ordem e pode ser evitado em nível de restrições no leiaute. Neste caso, duas ou mais partículas carregadas são necessárias para gerar múltiplos *upsets* sendo que a probabilidade desta ocorrência está diretamente relacionada com o posicionamento das células de memória. O terceiro tipo de MBU ocorre quando múltiplas partículas se chocam provocando múltiplos *upsets* em vários nós do silício. Este evento pode ser analisado como grupos de SEUs representados como se fossem um único evento.

Com base no trabalho desenvolvido por Reed (REED, 1997), a maioria dos múltiplos *upsets* localizados em células adjacentes são provocados por uma única partícula. Existe uma probabilidade muito baixa de que mais de uma partícula carregada, interagindo com células adjacentes, provoque distúrbios em intervalos menores do que $1s$. Em outro trabalho (VELAZCO; REZGUI; ECOFFET, 2000), são mostrados alguns resultados de SEU,

em memórias SRAM (Hitachi e MHS) utilizadas durante vôos espaciais. Um total de 691 *upsets* foram detectados para o período de tempo analisado, 333 deles provenientes da SRAM Hitachi e 358 ocorrem na memória SRAM MHS. A distribuição de *bit flips* na palavra de *bits* da memória foi uniforme. Alguns duplos *upsets* de bits também foram detectados, 8 duplos *upsets* na Hitachi e 3 na memória da MHS. Foi observado também que transições $1-0$ ocorrem com mais frequência do que $0-1$ nas duas memórias testadas.

3.3.3.2 *Single Event-Transient* (SET)

Historicamente, a preocupação com falhas em circuitos integrados provocadas por partículas energizadas, estava relacionada apenas com SEU, ou seja, partículas que se chocam diretamente com elementos de memória. Isto se deve principalmente à densidade espacial dos elementos de memória e à quantidade de informação que estes podem armazenar. Porém, o constante avanço da tecnologia CMOS tem tornado substancial a preocupação com os efeitos causados por partículas que se chocam com a parte combinacional de circuitos integrados. Como resultado da redução das dimensões dos transistores e consequente diminuição das capacitâncias internas, a probabilidade de que um pulso consiga se propagar através de um circuito tem aumentado (MAHESHWARI; KOREN; BURLESON, 2003).

Se uma partícula carregada colide com uma região sensível de um circuito combinacional, ocasionando a geração de um pulso transiente de tensão, o fenômeno associado recebe o nome de *Sigle-Event Transient* ou SET (ALEXANDRES; ANGHEL; NICOLAIDIS, 2002; KASTENSMIDT; CARRO; REIS, 2006). Se o pulso de corrente gerado possuir amplitude e duração suficientes para carregar ou descarregar uma capacitância de saída, gerando assim um pulso de tensão transiente, tal pulso de tensão pode vir a ser interpretado como uma mudança de nível lógico. Em média, a duração deste tipo de pulso de tensão provocado por um SET varia entre 1 ps e 100 ps (ANGHEL; NICOLAIDIS, 2000; HEIJMEN; NIEUWLAND, 2006).

Embora a geração de um pulso de tensão indesejável no circuito não se caracterize como algo previsto, este não necessariamente deve provocar uma falha transiente no circuito. Esta falha somente se concretiza, caso este pulso venha a se propagar pelo circuito e por sua vez, venha a ser capturado (erroneamente) por um elemento de memória, caracterizando um erro. Na Figura 3.6 é apresentado o comportamento de um circuito, dada a colisão de uma partícula energizada com uma região sensível de um transistor pertencente ao bloco combinacional. Deve-se salientar que tal fato se caracteriza como uma falha transiente no circuito, haja vista que na próxima borda de relógio, o valor errôneo

armazenado pelo registrador será sobrescrito, fazendo com que o circuito continue a operar normalmente.

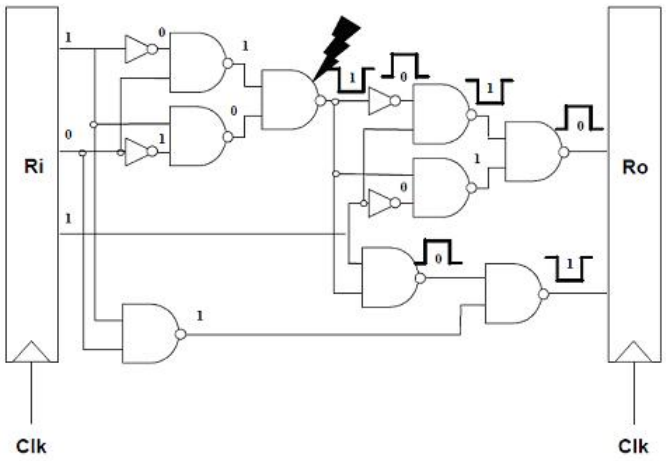

Figura 3.6: *single-event transient* (SET) e sua possível propagação em um bloco combinacional, adaptado de (NIEUWLAND; JASAREVIC; JERIN, 2006).

Conforme mostrado na Figura 3.6, um efeito causado por um SET somente se caracterizará como uma falha no funcionamento do circuito caso o pulso de tensão se propague pelo circuito e posteriormente, seja armazenado por um elemento de memória. Neste ponto, é possível perceber que a probabilidade de um pulso se propagar até um elemento de memória, ou até mesmo a uma saída primária de um circuito, tende a ser inversamente proporcional ao número médio de níveis lógicos entre o ponto em que um pulso é gerado e as saídas primárias, ou registradores propriamente ditos. Assim, quanto maior o número de portas lógicas que um pulso transiente precisa atravessar, maior a probabilidade deste ser mascarado antes de ser armazenado por um elemento de memória ou ser enviado para uma saída primária (NIEUWLAND; JASAREVIC; JERIN, 2006).

Dentre os mecanismos que contribuem para que haja uma diminuição da probabilidade de que um pulso venha a se propagar por um circuito pode-se citar os mascaramentos lógico, elétrico e por *latching window*.

O mascaramento lógico está associado à controlabilidade das portas lógicas. Se um pulso, ao se propagar por um circuito, atingir uma das entradas de uma porta lógica, este pode vir a ser mascarado logicamente caso esta porta possua o valor controlador em pelo menos uma de suas outras entradas. Na Figura 3.7 é mostrado um exemplo desta situação. nnote que, independente do valor que esteja presente na entrada B, o valor da saída da porta NAND permanece imutável.

O mascaramento elétrico se dá devido às propriedade elétricas das portas lógicas no caminho do pulso. Quanto maior a duração de um pulso causado por um SET maior é a

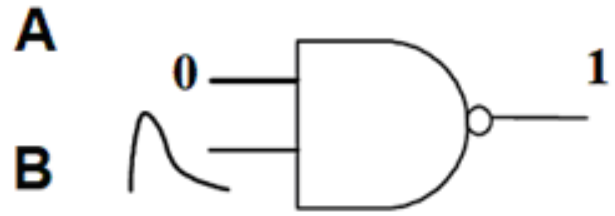

Figura 3.7: mascaramento lógico de uma porta NAND.

probabilidade deste conseguir se propagar por um circuito e eventualmente, ser capturado por um elemento de memória.

Se a duração de um pulso, provocado por um SET, possuir uma largura maior que o tempo de propagação de uma porta, este por sua vez, não é atenuado. Porém, se a largura do pulso possuir um valor menor do que o tempo de propagação da porta, este pulso tenderá a ser atenuado. Caso o pulso venha a ter uma largura menor do que metade do tempo de propagação de uma porta, então o pulso é completamente atenuado caracterizando-se um mascaramento elétrico (NICOLAIDIS, 1999; ALEXANDRES; ANGHEL; NICOLAIDIS, 2002). Na Figura 3.8 é apresentado o processo de atenuação de um pulso do tipo SET quando da passagem por uma sequência de inversores.

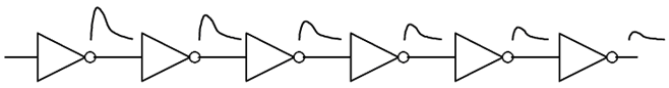

Figura 3.8: processo de atenuação de um pulso de tensão quando este se propaga por portas lógicas inversoras (ALEXANDRES; ANGHEL; NICOLAIDIS, 2002).

Por fim, além dos tipos de mascaramentos citados acima, existe também o mascaramento por *latching window* que ocorre quando um pulso atinge um elemento de memória fora da janela de amostragem. Se t_{su} e t_h são os tempos de preparação (*setup*) e manutenção (*hold*) de um registrador de saída e t é o instante em que ocorre a borda ativa do relógio, então o pulso é necessariamente capturado, caso tenha duração mínima entre $(t - t_{su})$ e $(t + t_h)$. Caso o pulso termine antes de $(t - t_{su})$ ou inicie depois de $(t + t_h)$, diz-se que ocorre um mascaramento por *latching window* (SHIVAKUMAR et al., 2002; ANGHEL; LEVEUGLE; VANHAUWAERT, 2005; WIRTH et al., 2008). Caso a duração de um pulso venha a possuir uma largura maior do que o período de relógio do circuito, este é necessariamente capturado por um elemento de memória (GADLAGE et al., 2005).

O exato momento em que um pulso é capturado por um elemento de memória, ou seja,

quando este alcança um registrador dentro de sua janela de amostragem está ilustrado na Figura 3.9. Esta Figura ilustra ainda alguns exemplos de pulsos mascarados.

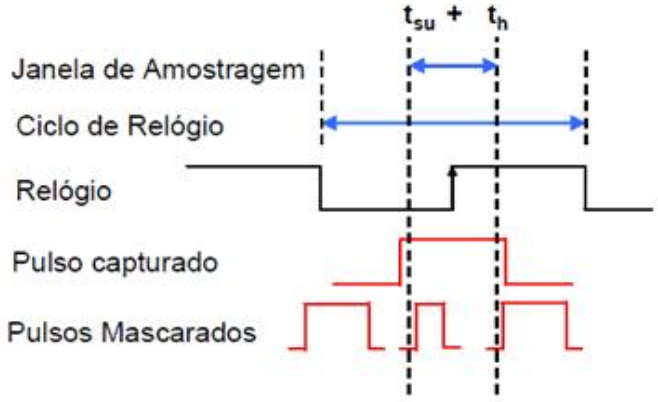

Figura 3.9: mascaramento por latching Window.

Um outro fator importante com relação à probabilidade de um *SET* tornar-se uma falha transiente é a frequência de operação dos circuitos digitais. À medida que esta se torna maior, maiores são as chances de uma falha ser capturada por um elemento de memória, haja vista o aumento de janelas de amostragem proporcionadas por frequência de operação maiores (GADLAGE et al., 2005).

Com o intuito de analisar computacionalmente os efeitos causados pela ação de partículas no silício, particularmente os efeitos causados pelo pulso gerado por um *SET*, alguns autores propõem alternativas de modelagem de um pulso de tensão. Na Figura 3.10 é ilustrado um modelo proposto por Messenger (MESSENGER, 1982). O modelo caracteriza o fenômeno de deposição de carga na saída de uma porta lógica como uma fonte de corrente, $I(t)$, cujo comportamento obedece a dupla exponencial

$$I(t) = I_0(e^{(-t/t_\alpha)} - e^{(-t/t_\beta)}), \tag{3.1}$$

em que I_0 é a corrente máxima, t_α é a constante de tempo da junção e t_β é a constante de tempo para estabelecer o impacto inicial da partícula. A Figura 3.10 mostra a modelagem proposta por Messenger (MESSENGER, 1982) e as formas de onda típicas para uma tecnologia de $100nm$, considerando diversos valores de I_0 e t_β.

A modelagem de um pulso de tensão transiente é de suma importância quando se deseja realizar a análise de sua propagação em um circuito eletrônico. Embora a equação de Messenger, Equação 3.1, modele os efeitos causados pela ação de partículas nos circuitos, sua utilização para estes fins é bastante difícil, dada a complexidade para modelagem de cada porta. Assim, vários outros modelos têm sido propostos e utilizados com este

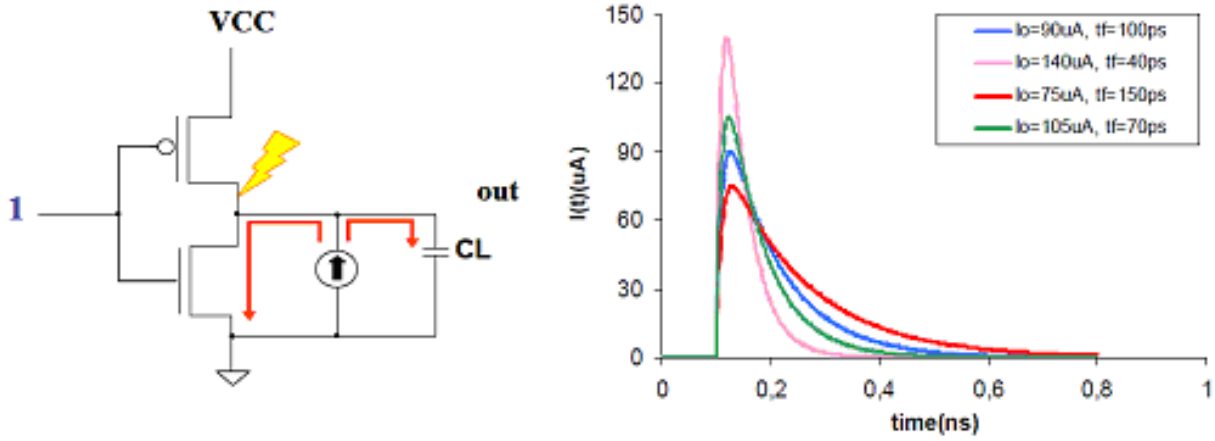

Figura 3.10: modelo de messenger (MESSENGER, 1982) para um pulso de tensão gerado a partir de um SET.

mesmo fim. Para uma discussão mais detalhada consultar as referências (ALEXANDRESCU et al., 2004; OMANA et al., 2003; ANGHEL; ALEXANDRESCU; NICOLAIDIS, 2000; DAHLGREN; LIDEN, 1995).

No próximo capítulo é discutido como os códigos corretores de erros podem ser interligados para lidar com alguns tipos de falhas discutidos neste capítulo.

Capítulo 4

Códigos Corretores de Erros

Os códigos corretores de erros podem ser encontrados na transmissão de informações em diversos dispositivos digitais utilizados em nosso dia a dia, tais como computadores, televisão digital, telefone celular, leitor de CD (*Compact Disc*) e DVD (*Digital Versatile Disc*) dentre outros. Um código corretor de erros é, essencialmente, uma maneira organizada de acrescentar algum dado adicional a cada informação para ser transmitida ou armazenada, de modo que permita, ao recuperar a informação, detectar e corrigir os erros cometidos no processo de armazenamento ou transmissão da informação (HEFEZ ABRAMO. E VILLELA, 2002).

4.1 Introdução

Os códigos corretores de erro podem ser classificados como: códigos de bloco e códigos de árvore. Um código de bloco é o mapeamento de k símbolos de entrada em n símbolos de saída. O número n pode ser maior ou igual a k, se for maior, os símbolos adicionais são redundantes e usados com a finalidade de prover deteção e/ou correção de erros. Este código não possui memória, pois a cada grupo de k símbolos é gerado um grupo de n símbolos definido.

Códigos de árvore são caracterizados por serem gerados num codificador que possua memória. Os códigos convolucionais são um subconjunto dos códigos de árvore. Um codificador convolucional aceita k símbolos na entrada e produz n símbolos na saída que são gerados em função de $v + k$ símbolos de entrada, assim, se v maior que zero, então o sistema possui memória. Com isso, temos que a razão entre os símbolos de entrada, no caso representados por k, e os símbolos de saída é $r = k/n$, sendo definido como a taxa de código.

4.2 Códigos de *Hamming*

Códigos de Hamming são códigos binários utilizados para detecção e correção de erros e constituem uma família de códigos de bloco lineares com $n = 2^c - 1, k = 2^c - c - 1, c = n - k$ e $d_{min} = 3$, para qualquer inteiro $c \geq 2$, em que c é o número de bits de paridade, n é o número total de bits da palavra código resultante da codificação (*Palavra-código*), k é o número de *bits* da informação a ser codificada e d_{min} é a distância mínima de *Hamming* (PROAKIS, 2002).

Estes códigos foram inventados por *Richard Wesley Hamming* (1915-1998) que publicou no *The Bell System Technical Journal* em abril de 1950 o artigo *Error Detecting and Error Correcting Codes* descrevendo o funcionamento e características destes códigos (HAMMING, 1950). De Hamming e Shooman (HAMMING, 1950; SHOOMAN, 2002) são extraídos algumas características:

- Para determinar o número de *bits* de dados (ou *bits* de informação) , k, e o número de *bits* de paridade c, utiliza-se a seguinte relação:

$$2^c \geq k + c + 1 \tag{4.1}$$

- A taxa de código ($r = k/n$) é a relação entre a quantidade de bits de informação (k) a ser codificada e a palavra código resultante da codificação (n). É uma relação adimensional e representa a eficiência do código.

- A relação entre a quantidade de *bits* de paridade (c) e a quantidade de *bits* da mensagem (k) resulta no *overhead* do código.

- Cada código de *Hamming* é representado na forma (n, k). Ex.: *Hamming (7,4)*

NaTabela 4.1 é mostrada a relação entre os *bits* de dados (k), os *bits* de paridade (c), o tamanho da palavra código (n), a taxa de código (r) e o *overhead* de alguns códigos de *Hamming*.

Hamming introduziu o uso de vários *bits* de paridade gerados a partir da informação a ser codificada, para detectar e corrigir erro num único *bit*. Um *bit* de paridade diz se o número de 1s de um grupo de *bits* é par ou impar. Ele é acrescentado para que o número de 1s, em um grupo de *bits*, seja par ou ímpar (FLOYD, 2007).

Para explicar o processo de codificação será utilizada uma informação com o tamanho de 4 *bits* ($k = 4$). Da Tabela 4.1, com $k = 4$, tem-se $n = 7$ e $c = 3$. Desta forma, tem-se

Tabela 4.1: exemplo de código de bloco (6,3).

Hamming (n,k)	k	c	n	r	*Overhead (%)*
(7,4)	4	3	7	0,571	75
(12,8)	8	4	12	0,667	50
(21,16)	16	5	21	0,762	31,25
(38,32)	32	6	38	0,842	18,75

3 *bits* de paridade (P_1, P_2 *e* P_3) e 4 bits de dados (D_1, D_2, D_3 *e* D_4), totalizando 7 *bits* de *palavra-código* e cada *bit* ocupa uma posição numerada de 1 até 7. Os *bits* de paridade estão localizados nas posições cujos números correspondem as potências de 2 $(1, 2, 4)$ e cada *bit* de paridade provê uma verificação em outros determinados *bits* no código total (FLOYD, 2007).

Cada posição da palavra-código possui uma representação binária (posição $1 = 001_2$). O número da posição em binário do *bit* de paridade P_1 possui valor 1 no dígito mais a direita ($1 = 001_2$). Esse *bit* de paridade verifica todas as posições, incluindo ele mesmo, que têm 1s no *bit* mais a direita. Portanto, o *bit* de paridade P_1 verifica as posições dos *bit* $1, 3, 5$ e 7. O número da posição em binário do *bit* de paridade P_2 possui valor 1 no digito do meio ($2 = 010_2$) e verifica as posições $2, 3, 6$ e 7. O *bit* de paridade P_3 ocupa a posição 4 e em binário possui valor 1 no digito da esquerda ($4 = 100_2$) e verifica as posições $4, 5, 6$ e 7.

A detecção e correção são proporcionadas para todos os *bits* da palavra código, inclusive os *bits* de paridade. Na decodificação é realizada a verificação dos *bits* de paridade da palavra código recebida. Para $k = 4$, há 3 bits de paridade (P_1, P_2 e P_3) que formam uma palavra binária $P_3P_2P_1$, em que P_1 é o *bit* menos significativo. Essa palavra é chamada de síndrome. Se não ocorrer nenhum erro, a síndrome é zero. Se ocorrer um simples erro, a síndrome é diferente de zero e o valor indica a posição do *bit* errado. Para corrigí-lo, basta inverter o seu valor (01 ou 10). Se ocorrerem dois erros, a síndrome é diferente de zero, entretanto, a posição indicada do *bit* errado é incorreta. Os códigos de *hamming* com $d_{min} = 3$ detectam e corrigem um erro (conhecido como SECSED - *Single Error Correction and Single Error Detection*) ou podem ser usados para detectar erros duplos (DED - *Double Error Detection*) (SHOOMAN, 2002).

4.3 Código de Hamming + Paridade

Adicionando um *bit* extra de paridade no código de *Hamming*, é possível aumentar a distância mínima para 4 ($d_{min} = 4$). Este código de *Hamming* com um *bit* de paridade extra é chamado de código de *Hamming + Paridade* ou código de *Hamming Estendido*. Isto dá ao código a capacidade de detectar e corrigir um erro e, ao mesmo tempo, detectar um duplo erro (conhecido como SECDED - *Single Error Correction and Double Error Detection*). Ou podem ser usados para detectar três erros (*TED - Triple Error Detection*).

A diferença no processo de codificação entre o *Hamming convencional* e o *Hamming + Paridade* (*Hamming Estendido*) é a adição de um *bit* de paridade no final do processo de codificação.

No processo de decodificação, a diferença entre os códigos é a verificação da paridade P. No código de *Hamming + paridade* podem haver quatro situações que são interpretadas da seguinte maneira (Tabela 4.2):

Tabela 4.2: exemplo de código de bloco (6,3).

Síndrome	*Paridade*	*Interpretação*
Zero	P = 0	Não ocorreu nenhum erro
Diferente de Zero	P = 1	Ocorreu um erro que pode ser corrigido
Diferente de Zero	P = 0	Ocorreu erro duplo mas não pode ser corrigido
Zero	P = 1	Um erro ocorreu no bit de paridade P

Na Tabela 4.3 é mostrada a relação entre os bits de dados (k), o tamanho da Palavra-Código (n) e o overhead de alguns códigos de Hamming + Paridade.

4.4 Códigos de Reed-Solomon

Em 1960, Irving Reed e Gus Solomon publicaram um artigo no *Journal of the Society for Industrial and Applied Mathematics* (REED; SOLOMON, 1978), descrevendo uma nova classe de códigos corretores de erros chamados Códigos *Reed-Solomon*. Esses se mostraram bastante eficientes sendo atualmente utilizados em diversas aplicações, entre estas as da área espacial. Nessa seção são apresentados os algoritmos de codificação e decodificação dessa classe de códigos corretores de erros.

Tabela 4.3: exemplo de código de bloco (6,3).

Hamming + Paridade (n,k)	*k*	*n*	*Overhead (%)*
(8,4)	4	8	100
(13,8)	8	13	62,50
(22,16)	16	22	37,50
(39,32)	32	39	21,87

Códigos *Reed-Solomon* (RS) são códigos cíclicos não-binários constituídos de sequências de *mbits*, onde m é qualquer inteiro positivo maior que 2 (BOLZANI, 2005). Um código cíclico é aquele que, a partir de um vetor U pertencente ao subespaço vetorial do código cíclico, é possível gerar todos os demais códigos através do deslocamento sucessivo de U. O vetor U é descrito como: $U = (\mu_0\, \mu_1\, \mu_2\, \ldots\, \mu_{n-1})$.

Sequências RS(*Reed-Solomon*) são representadas na forma $RS(n, k)$, em que n representa o número total de símbolos, e k o número total de símbolos a serem codificados.

$$(n, k) = (2^m - 1, 2^m - 1 - 2t) \tag{4.2}$$

Na Equação 4.2, t representa a capacidade de correção de símbolos com erros do código, e $r = 2t$ o número de símbolos de paridade anexados à mensagem a ser transmitida.

Códigos *RS* são particularmente úteis para correção de erros em rajada. Também podem ser usados eficientemente em canais onde o conjunto de símbolos de entrada é consideravelmente grande.

4.4.1 Polinômio Primitivo

A definição de uma classe de polinômios, chamada polinômios primitivos, se faz necessária para definir os códigos Reed-Solomon. A seguinte condição é necessária e suficiente para garantir que um polinômio seja primitivo. Um polinômio irredutível, $f(X)$ de grau m é dito como primitivo se o menor inteiro positivo n para o qual $f(X)$ divide $X^n + 1$ é $n = 2^m - 1$. Um polinômio irredutível não pode ser fatorado em polinômios de ordem menor. Por exemplo, o polinômio $x^2 + x + 1$ é irredutível, mas $x^2 + 1$ não, porque $(x + 1)(x + 1) = x^2 + 2x + 1 = x^2 + 1$ (módulo 2). Entretanto, nem todos os polinômios

irredutíveis são primitivos. Na Tabela 4.4 é apresentado o mapeamento dos 8 elementos pertencentes ao corpo de *Galois* representado por $GF(2^3)$ com $f(x) = x^3 + x + 1$. Há diferentes maneiras para representar um polinômio, entre elas a notação binária e a notação decimal. As notações na sequência apresentada na Equação 4.3 representam o mesmo polinômio.

$$x^5 + x^2 + 1 = 100101_2 = 37_{10} \tag{4.3}$$

Tabela 4.4: mapeamento de elementos no corpo para $GF(2^3) com f(x) = 1 + X + X^3$

	X^0	X^1	X^2
0	0	0	0
α^0	1	0	0
α^1	0	1	0
α^2	0	0	1
α^3	1	1	0
α^4	0	1	1
α^5	1	1	1
α^6	1	0	1
α^7	1	0	0

Na Tabela 4.5 é apresentada uma lista de polinômios primitivos, com m variando de 3 a 24. No caso de aplicações espaciais no padrão CCSDS (*Consultative Committee for Space Data Systems*), cada pacote é dividido em blocos de 256 *bytes*, sendo necessários 256 elementos que compõem o Corpo de *Galois* para representar cada um dos *bytes*. O polinômio que representa esses elementos é definido como sendo de grau 8, ou seja, $m = 8$.

O seguinte exemplo ilustra o algoritmo RS com símbolos de 4 bits. O gerador polinomial define um corpo finito sobre todas as operações que são calculadas da seguinte maneira: O termo $GF(2^4)$ significa que o corpo finito possui 16 elementos (HAMMING, 1950). O cálculo dos corpos começa com um elemento primitivo α, que nesse caso é 2 ou 0010. Um incremento na potência de α representa cada membro sucessivo pertencente ao corpo; então, α é chamada de raíz primitiva porque sua potência representa todos os membros pertencentes ao corpo diferentes de zero. Na Tabela 4.6 são mostrados os cálculos dos elementos pertencentes ao $GF(2^4)$.

Após definidos todos os elementos pertencentes a $GF(2^4)$, operações de soma, subtração, multiplicação e divisão são executadas. As operações de soma e subtração são as mesmas e trabalham com portas XOR de elementos representados por valores numéricos (KASTENSMIDT; CARRO; REIS, 2006). Por exemplo, $\alpha^5 + \alpha^6 = 0110\ XOR\ 1100 = 1010 =$

Tabela 4.5: classe de polinômios primitivos

m			m	
3	$1+X+X^3$		14	$1+X+X^6+X^{10}+X^{14}$
4	$1+X+X^4$		15	$1+X+X^{15}$
5	$1+X^2+X^5$		16	$1+X+X^3+X^{12}+X^{16}$
6	$1+X+X^6$		17	$1+X+X^3+X^{17}$
7	$1+X^3+X^7$		18	$1+X^6+X^{18}$
8	$1+X^2+X^3+X^4+X^8$		19	$1+X+X^2+X^5+X^{19}$
9	$1+X^4+X^9$		20	$1+X^3+X^{20}$
10	$1+X^3+X^{10}$		21	$1+X^2+X^{21}$
11	$1+X^2+X^{11}$		22	$1+X+X^{22}$
12	$1+X+X^4+X^5+X^{12}$		23	$1+X^5+X^{23}$
13	$1+X++X^3+X^4+X^{13}$		24	$1+X+X^2+X^7+X^{24}$

α^9. Operações de multiplicação e divisão são executadas através de soma e subtração de potência de elementos, lembrando que $\alpha^{15}=1$. Por exemplo, α^2 x $\alpha^4=\alpha^6$; $\alpha^4/\alpha^2=\alpha^2$. O próximo passo é montar a relação de cada elemento do corpo do polinômio. Para simplificar a demonstração, o polinômio utilizado é de grau 3. A Equação 4.4 apresenta α^3 como sendo a soma de α-termos de menor ordem.

$$\alpha^3 = 1+\alpha \tag{4.4}$$

De fato, todas as potências de α podem ser então representadas. Por exemplo, considere a Equação 4.5.

$$\alpha^4 = \alpha.\alpha^3 = \alpha(1+\alpha) = \alpha+\alpha^2 \tag{4.5}$$

Sabendo que $\alpha^4=\alpha+\alpha^2$, considere a Equação 4.6.

$$\alpha^5 = \alpha.\alpha^4 = \alpha(\alpha+\alpha^2) = \alpha^2+\alpha^3 \tag{4.6}$$

Das Equações 4.4 e 4.6 obtém-se

$$\alpha^5 = 1+\alpha+\alpha^2 \tag{4.7}$$

Agora, usando a Equação 4.7, tem-se

$$\alpha^6 = \alpha.\alpha^5 = \alpha(1+\alpha+\alpha^2) = \alpha+\alpha^2+\alpha^3 = 1+\alpha^2 \tag{4.8}$$

Tabela 4.6: cálculo dos elementos de GF(2^4)

Potência	**Cálculo**	**Valor Numérico**
$\alpha = x$	x	$0010_2 = 2_{10}$
$\alpha^2 = x$ x x	x^2	$0100_2 = 4_{10}$
$\alpha^3 = x$ x x x x	x^3	$1000_2 = 8_{10}$
$\alpha^4 = \alpha$ x α^3	$x^4 = x + 1$	$0011_2 = 3_{10}$
$\alpha^5 = \alpha$ x α^4	$x^5 = x^2 + x$	$0110_2 = 6_{10}$
$\alpha^6 = \alpha$ x α^5	$x^6 = x^3 + x^2$	$1100_2 = 12_{10}$
$\alpha^7 = \alpha$ x α^6	$x^7 = x^4 + x^3 = x^3 + x + 1$	$1011_2 = 11_{10}$
$\alpha^8 = \alpha$ x α^7	$x^8 = x^4 + x^2 + x = x^2 + 1$	$0101_2 = 5_{10}$
$\alpha^9 = \alpha$ x α^8	$x^9 = x^3 + 1$	$1010_2 = 10_{10}$
$\alpha^10 = \alpha$ x α^9	$x^{10} = x^4 + x^2 = x^2 + x + 1$	$0111_2 = 7_{10}$
$\alpha^11 = \alpha$ x α^{10}	$x^{11} = x^3 + x^2 + x$	$1110_2 = 14_{10}$
$\alpha^12 = \alpha$ x α^{11}	$x^{12} = x^4 + x^3 + x^2 = x^3 + x^2 + 1$	$1111_2 = 15_{10}$
$\alpha^13 = \alpha$ x α^{12}	$x^{13} = x^4 + x^3 + x^2 + x = x^3 + x^2 + 1$	$1101_2 = 13_{10}$
$\alpha^14 = \alpha$ x α^{13}	$x^{14} = x^4 + x^3 + x = x^3 + 1$	$1001_2 = 9_{10}$
$\alpha^15 = \alpha$ x α^{14}	$x^{15} = x^4 + x = 1$	$0001_2 = 1_{10}$

Usando a Equação 4.8, obtém-se

$$\alpha^7 = \alpha.\alpha^6 = \alpha(1 + \alpha^2) = \alpha + \alpha^3 = 1 = \alpha^0 \tag{4.9}$$

Note que $\alpha^7 = \alpha^0$, e portanto, os oito elementos finitos do corpo de $GF(2^3)$ são

$$0, \alpha^0, \alpha^1, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6 \tag{4.10}$$

Com o desenvolvimento dessas funções é possível obter a relação de todos os elementos pertencentes a $GF(2^3)$, pois cada elemento é representado pela soma de termos de menor ordem. Na Tabela 4.7, são apresentadas as operações de adição, e na Tabela 4.8 é apresentada a multiplicação de elementos em um corpo finito de grau 3 na representação de matrizes.

As tabelas de adição e multiplicação representam uma matriz de dimensão $(2^m - 1)$ x $(2^m - 1)$ e contêm todas as associações aditivas e multiplicativas dos elementos pertencentes a $GF(2^3)$.

4.4.2 Codificação Reed-Solomon

A Equação 4.2 apresenta a forma mais convencional de códigos RS em termos dos parâmetros n, k, t e qualquer inteiro positivo $m > 2$.

Tabela 4.7: mapeamento de soma de elementos para $GF(2^3)$

	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^0	0	α^3	α^6	α^1	α^5	α^4	α^2
α^1	α^3	0	α^4	α^0	α^2	α^6	α^5
α^2	α^6	α^4	0	α^5	α^1	α^3	α^0
α^3	α^1	α^0	α^5	0	α^6	α^2	α^4
α^4	α^5	α^1	α^1	α^6	0	α^0	α^3
α^5	α^4	α^3	α^3	α^2	α^0	0	α^1
α^6	α^2	α^0	α^0	α^4	α^3	α^1	0

Tabela 4.8: mapeamento de multiplicação de elementos para $GF(2^3)$

	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^0	α^0	α^1	α^2	α^3	α^4	α^5	α^6
α^1	α^1	α^2	α^3	α^4	α^5	α^6	α^0
α^2	α^2	α^3	α^4	α^5	α^6	α^0	α^1
α^3	α^3	α^4	α^5	α^6	α^0	α^1	α^2
α^4	α^4	α^5	α^6	α^0	α^1	α^2	α^3
α^5	α^5	α^6	α^0	α^1	α^2	α^3	α^4
α^6	α^6	α^0	α^1	α^2	α^3	α^4	α^5

Um gerador polinomial, representado por $g(x)$, representa todos os polinômios válidos pertencentes ao Corpo de *Galois*. Sendo o gerador polinomial de grau $2t$, este deve ser precisamente $2t$ potências sucessivas de α que são raízes do polinômio gerador (LAPRIE, 1985) (JOHNSTON, 2000). A Equação 4.11 descreve o gerador polinomial com $2t = n-k =$ 4 raízes.

$$g(X) = X^4 - \alpha^3 X^3 + \alpha^0 X^2 - \alpha^1 X + \alpha^3 \tag{4.11}$$

Seguindo o formato de baixa ordem para alta ordem, e trocando os sinais negativos para positivo na Equação 4.11, sendo que em corpos binários +1 = -1, o gerador $g(X)$ pode ser representado na forma

$$g(X) = \alpha^3 + \alpha^1 X + \alpha^0 X^2 + \alpha^3 X^3 + X^4 \tag{4.12}$$

A seguir é demonstrado o passo de codificação para a mensagem apresentada na Equação 4.13.

$$010\ 110\ 111,\ \textit{para}\ \alpha^1,\ \alpha^3\ \textit{e}\ \alpha^5\ \textit{respectivamente}. \tag{4.13}$$

Primeiramente multiplica-se o polinômio da mensagem $\alpha^1 + \alpha^3 X + \alpha^5 X^2$ por $X^{n-k} = X^4$, resultando em $\alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6$.

O próximo passo é dividir o polinômio da mensagem pelo gerador polinomial da Equação 4.12, $\alpha^3 + \alpha^1 X + \alpha^0 X^2 + \alpha^3 X^3 + X^4$. É possível verificar que a divisão de polinômios resulta no seguinte polinômio de paridade:

$$p(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 \tag{4.14}$$

Então, o polinômio do bloco de códigos a ser transmitido pode ser escrito como

$$U(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6 \tag{4.15}$$

4.4.2.1 Codificação Sistemática com um Registrador de Deslocamento de $(n-k)$ Estágios

A utilização de circuitos eletrônicos para realizar a codificação *Reed-Solomon*, levando em conta o polinômio gerador apresentado na Equação 4.12, necessita a implementação de um registrador de deslocamento com realimentação (LFSR - *Linear Freedback Shift Register*).

Um LFSR é composto de um registrador de deslocamento e uma função de retorno. Um registrador de deslocamento é um dispositivo cuja função é deslocar seu conteúdo em posições adjacentes ao registrador ou deslocar o conteúdo para a saída. O conteúdo de um registrador de deslocamento é geralmente composto de '1's e '0's. Se um registrador de deslocamento contém o seguinte padrão de bit "1101", um deslocamento, para a direita nesse caso, resultaria em "0110", com mais um deslocamento para direita obteria-se "0011".

Dois usos para registradores de deslocamento são:

1. conversão entre dados paralelos e serial
2. atraso em uma *stream* de *bit* serial

A função de conversão, entre paralelo e serial, pode ser feita dos seguintes modos: preencher todas as posições do registrador de deslocamento de uma única vez (paralelo) e então deslocar os bits para a saída (serial). A função de atraso simplesmente desloca os bits do final de um registrador de deslocamento para outro, fornecendo um atraso igual

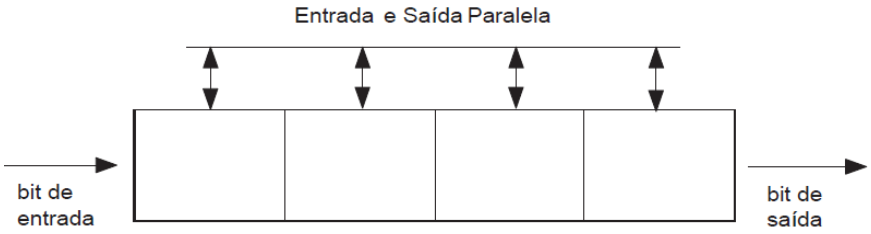

Figura 4.1: Registro de deslocamento.

ao comprimento do registrador de deslocamento. Na Figura 4.1 é apresentado o fluxo do processo de conversão de dados paralelos em serial e vice-versa.

Em um LFSR, os bits contidos nas posições selecionadas no registrador de deslocamento são combinados no mesmo tipo de função e o resultado é colocado de volta nos registradores de entrada. A função de retorno em um LFSR possui alguns nomes: *XOR* (ou exclusivo), paridade ímpar ou soma módulo 2. Qualquer que seja o nome, a função é simples: 1) Somar os valores de *bits* selecionados, 2) se a soma for ímpar, a saída da função é 1; senão a saída é 0. Na Tabela 4.9 é mostrado a saída para uma função XOR de 3 entradas:

Tabela 4.9: função XOR de 3 entradas

Entrada A	Entrada B	Entrada C	Saída
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

O uso de um circuito para codificar uma sequência de 3 símbolos na forma sistemática com o Código $RS(7,3)$ requer a implementação de uma LFSR, como mostrado na Figura 4.2. Pode-se verificar que os fatores de multiplicação da esquerda para a direita correspondem aos coeficientes do polinômio na Equação 4.12 (baixa ordem para alta ordem). O código $RS(7,3)$ é constituído de $2^m - 1 = 7$ símbolos, e cada símbolo possui $m = 3$ *bits*.

A operação implementada pelo codificador da Figura 4.2 resulta em palavras-código na forma não sistemática, procedendo do mesmo modo como feito em códigos binários. Os passos podem ser descritos como segue:

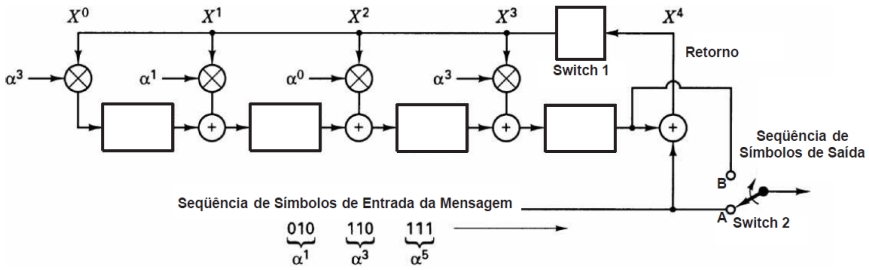

Figura 4.2: Codificador LFSR para RS(7,3).

Tabela 4.10: conteúdo dos registradores no processo de codificação **LFSR**

Fila de Entrada			**Ciclo de Clock**	**Registradores**				**Retorno**
α^1	α^3	α^5	0	0	0	0	0	α^5
	α^1	α^3	1	α^1	α^6	α^5	α^1	α^0
		α^1	2	α^3	0	α^2	α^2	α^4
		-	3	α^0	α^2	α^4	α^6	-

- A chave 1 é fechado durante os primeiros k ciclos de *clock* para permitir o deslocamento dos símbolos da mensagem em $(n-k)$ estágios no registrador de deslocamento.
- A chave 2 está na posição A durante os primeiros k ciclos de *clock* para permitir transferência de símbolos da mensagem diretamente para um registrador de saída.
- Depois de transferir os k_s símbolos da mensagem para o registrador de saída, a chave 1 é aberta e a chave 2 é movida para a posição B.
- Os $(n-k)$ ciclos de *clock* restantes apagam os símbolos de paridade contidos no registrador de saída.
- O número total de ciclos de *clock* é igual a n, e o conteúdo do registrador de saída é o polinômio $p(X) + X^{n-k}m(X)$, em que $p(X)$ representa os símbolos de paridade e $m(X)$ os símbolos da mensagem em forma de polinômio.

Os passos operacionais durante os primeiros $k = 3$ deslocamentos do circuito codificador da Tabela 4.10 são os seguintes:

Depois do terceiro ciclo de clock, o registrador contém os 4 símbolos de paridade, representados por α^0, α^2, α^4 e α^6, como mostrado na Tabela 4.10. Então, a chave 1 do circuito é aberta, a chave 2 é levada para a posição B e os símbolos contidos no registrador são deslocados para a saída. Consequentemente, o *palavra-código* de saída, escrito na forma polinomial, pode ser representado como apresentado na Equação 4.16.

$$U(X) = \sum_{n=o}^{6} u_n X^n$$

$$U(X) = \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6 \quad (4.16)$$
$$= (100) + (001)X + (011)X^2 + (101)X^3 + (010)X^4 + (110)X^5 + (111)X^6$$

4.4.3 Decodificação Reed-Solomon

De forma a exemplificar o algoritmo de decodificação, uma mensagem de teste codificada na forma sistemática usando um código $RS(7, 3)$ resultou em uma *palavra-código* polinomial descrita pela Equação 4.16. Agora, assume-se que durante a transmissão essa *palavra-código* foi corrompida resultando no recebimento de 2 símbolos com erro. O número de erros corresponde à capacidade máxima de correção do código. Para a *palavra-código* de 7 símbolos desse exemplo, o erro padrão pode ser escrito na forma polinomial apresentada na Equação 4.17.

$$U(X) = \sum_{n=0}^{6} e_n X^n \quad (4.17)$$

Para esse exemplo, dois símbolos de erros são representados como α^2 e α^5 conforme a Equação 4.18.

$$e(X) = 0 + 0X + 0X^2 + \alpha^2 X^3 + \alpha^5 X^4 + 0X^5 + 0X^6$$
$$= (000) + (000)X + (000)X^2 + (001)X^3 + (111)X^4 + (000)X^5 + (000)X^6 \quad (4.18)$$

Como visto anteriormente, os quatro primeiros elementos do polinômio representam a paridade e os outros três elementos representam a mensagem a ser transmitida. Um símbolo de paridade foi corrompido com 1-*bit* de erro (α^2), e um símbolo de dados corrompido com 3 bits de erro (α^5). O polinômio da *palavra-código* recebida com erro $r(X)$ é então representada pela soma da *palavra-código* transmitida com o polinômio de erro padrão como segue:

$$r(X) = U(X) + e(X) \quad (4.19)$$

Seguindo a Equação 4.19, soma-se $U(X)$ da Equação 4.16 com e(X) da Equação 4.18 para produzir:

$$r(X) = (100) + (001)X + (011)X^2 + (100)X^3 + (101)X^4 + (110)X^5 + (111)X^6$$
$$= \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^0 X^3 + \alpha^6 X^4 + \alpha^3 X^5 + \alpha^5 X^6 \quad (4.20)$$

Nesse exemplo de correção de erros de 2 símbolos, existem quatro elementos desconhecidos, duas localizações de erros e dois valores de erros. É importante notar a diferença entre a decodificação binária e a decodificação não binária. Na primeira o decodificador somente precisa encontrar a localização do erro. Sabendo que existe um erro em uma localização particular, basta trocar o *bit* de 1 para 0 ou vice-versa. Símbolos não binários necessitam não somente saber a localização do erro, mas também os valores dos símbolos corretos nas suas localizações (REED, 1997). Uma vez que existem quatro elementos desconhecidos nesse exemplo, quatro equações são necessárias para sua solução.

4.4.3.1 Cálculo da Síndrome

A síndrome é o resultado da verificação de paridade executada em r para determinar se r é um membro válido do conjunto de *palavra-código*. Se de fato r é um membro, então a síndrome S tem o valor 0. Qualquer valor de S diferente de zero indica a presença de erros. Similarmente ao caso binário, a síndrome S é composta de $n-k$ símbolos, $S_i(i = 1, \ldots, n-k)$. Então, para o código $RS(7,3)$ existem quatro símbolos que compõem o vetor de síndromes. Esses valores podem ser calculados a partir do polinômio recebido $r(X)$. Para esse exemplo, os quatro símbolos de síndrome são encontrados como segue:

$$S_1 = r(\alpha) = \alpha^0 + \alpha^3 + \alpha^6 + \alpha^3 + \alpha^1 0 + \alpha^8 + \alpha^1 1$$
$$= \alpha^0 + \alpha^3 + \alpha^6 + \alpha^3 + \alpha^2 + \alpha^1 + \alpha^4 \quad (4.21)$$
$$= \alpha^3$$

$$S_2 = r(\alpha^2) = \alpha^0 + \alpha^4 + \alpha^8 + \alpha^6 + \alpha^1 4 + \alpha^1 3 + \alpha^1 7$$
$$= \alpha^0 + \alpha^4 + \alpha^1 + \alpha^6 + \alpha^0 + \alpha^6 + \alpha^3 \quad (4.22)$$
$$= \alpha^5$$

$$\begin{aligned} S_3 = r(\alpha^3) &= \alpha^0 + \alpha^5 + \alpha^1 0 + \alpha^9 + \alpha^1 8 + \alpha^1 8 + \alpha^2 3 \\ &= \alpha^0 + \alpha^5 + \alpha^3 + \alpha^2 + \alpha^4 + \alpha^4 + \alpha^2 \\ &= \alpha^6 \end{aligned} \tag{4.23}$$

$$\begin{aligned} S_4 = r(\alpha^4) &= \alpha^0 + \alpha^6 + \alpha^1 2 + \alpha^1 2 + \alpha^2 2 + \alpha^2 3 + \alpha^2 9 \\ &= \alpha^0 + \alpha^6 + \alpha^5 + \alpha^5 + \alpha^1 + \alpha^2 + \alpha^1 \\ &= 0 \end{aligned} \tag{4.24}$$

O resultado confirma que a *palavra-código* recebida contém um erro (aqui inseridos) desde que $S \neq 0$.

4.4.3.2 Localização dos Erros

Supondo que existam v erros na *palavra-código* na localização $X_1^j; X_2^j; \ldots X_v^j$. Então o polinômio de erro mostrado nas Equações 4.17 e 4.18 pode ser escrito como

$$e(X) = e_{j_1} X^{j_1} + e_{j_2} X^{j_2} + \ldots + e_{j_v} X^{j_v} \tag{4.25}$$

Os índices $1, 2, \ldots, v$ indicam o $1^0, 2^0, \ldots, v_{esimo}$ erro, e o índice j indica a localização do erro. Para corrigir a *palavra-código* corrompida, cada valor de erro e_{jl} e sua localização X^{jl} , em que $l = 1, 2, \ldots, v$ devem ser determinadas. Define-se um número localizador de erros como $\beta_l = \alpha^{j_l}$. No próximo passo, são obtidos os $n - k = 2t$ símbolos de síndrome substituindo α^i no polinômio recebido para $i = 1, 2, \ldots, 2t$:

$$\begin{aligned} S_1 = r(\alpha) &= e_{j_1}\beta_1 + e_{j_2}\beta_2 + \ldots + e_{j_v}\beta_v \\ S_2 = r(\alpha^2) &= e_{j_1}{\beta_1}^2 + e_{j_2}{\beta_2}^2 + \ldots + e_{j_v}{\beta_v}^2 \\ &\ldots \\ S_{2t} = r(\alpha^{2t}) &= e_{j_1}{\beta_1}^{2t} + e_{j_2}{\beta_2}^{2t} + \ldots + e_{j_v}{\beta_v}^{2t} \end{aligned} \tag{4.26}$$

Existem $2t$ elementos desconhecidos (t valores de erros e t localizações), e $2t$ equações simultâneas. No entanto, essas $2t$ equações não podem ser resolvidas de modo usual por serem não-lineares. Uma técnica que resolva esse sistema de equações é conhecida como algoritmo de decodificação Reed-Solomon.

Quando um vetor de síndromes diferente de zero (um ou mais dos seus símbolos são diferentes de zero) é corrompido (IYER R. K. E KALBARCZYK, 2010), significa que um erro foi recebido. Para isso, é necessário descobrir a localização do erro ou erros. Um polinômio localizador de erro pode ser definido como

$$\begin{aligned}\sigma(X) &= (1+\beta_1 X)(1+\beta_2 X)\dots(1+\beta_v X) \\ &= 1+\sigma_1 X+\sigma_2 X^2+\dots+\sigma_v X^v \end{aligned} \tag{4.27}$$

As raízes de $\sigma(X)$ são $1/\beta_1, 1/\beta_2, \dots, 1/\beta_v$. A recíproca das raízes de $\sigma(X)$ são os números de localização de erros padrão $e(X)$. Usando a técnica de modelagem auto-regressiva é formada uma matriz de síndromes, onde as primeiras t síndromes são usadas para prever a próxima síndrome, que é:

$$\begin{bmatrix} S_1 & S_2 & S_3 & \dots & S_{t-1} & S_t \\ S_2 & S_3 & S_4 & \dots & S_t & S_{t+1} \\ & & & \dots & & \\ S_{t-1} & S_t & S_{t+1} & \dots & S_{2t-3} & S_{2t-2} \\ S_t & S_{t+1} & S_{t+2} & \dots & S_{2t-2} & S_{2t-1} \end{bmatrix} \begin{bmatrix} \sigma_t \\ \sigma_{t-1} \\ \dots \\ \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} -S_{t+1} \\ -S_{t+2} \\ \dots \\ -S_{2t-1} \\ -S_{2t} \end{bmatrix} \tag{4.28}$$

Para o código $RS(7,3)$ corretor de erros de duplos símbolos, o tamanho da matriz é 2 x 2 e o modelo é escrito como

$$\begin{bmatrix} S_1 & S_2 \\ S_2 & S_3 \end{bmatrix} \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} S_3 \\ S_4 \end{bmatrix} \tag{4.29}$$

$$\begin{bmatrix} \alpha^3 & \alpha^5 \\ \alpha^5 & \alpha^6 \end{bmatrix} \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} \alpha^6 \\ \alpha^0 \end{bmatrix} \tag{4.30}$$

Para resolver os coeficientes σ_1 e σ_2 do polinômio localizador de erros $\sigma(X)$, primeiramente calcula-se a inversa da matriz da Equação 4.30, que é:

$$\begin{bmatrix} \alpha^1 & \alpha^0 \\ \alpha^0 & \alpha^5 \end{bmatrix} \tag{4.31}$$

Continuando a Equação 4.30, inicia-se a pesquisa para localizar os erros pela resolução dos coeficientes do polinômio localizador de erro $\sigma(X)$ como segue:

$$\begin{bmatrix} \sigma^2 \\ \sigma^1 \end{bmatrix} = \begin{bmatrix} \alpha^1 & \alpha^0 \\ \alpha^0 & \alpha^5 \end{bmatrix} \begin{bmatrix} \alpha^6 \\ 0 \end{bmatrix} = \begin{bmatrix} \alpha^7 \\ \alpha^6 \end{bmatrix} = \begin{bmatrix} \alpha^0 \\ \alpha^0 \end{bmatrix} \tag{4.32}$$

Das Equações 4.27 e 4.32 tem-se

$$\begin{aligned}\sigma(X) &= \alpha^0 + \sigma_1 X \sigma_2 X^2 \\ &= \alpha^0 + \alpha^6 X + \alpha^0 X^2\end{aligned} \tag{4.33}$$

As raízes de $\sigma(X)$ são recíprocas às localizações dos erros. Uma vez encontradas as raízes, a localização do erro passa a ser conhecida. Em geral, as raízes de $\sigma(X)$ podem ser um ou mais elementos do corpo. Nós determinamos essas raízes por teste exaustivo do polinômio $\sigma(X)$ com cada um de seus elementos, como mostrado a seguir. Qualquer elemento X que resultar em $\sigma(X) = 0$ é uma raiz no qual permite a localização de um erro:

$$\begin{aligned}\sigma(\alpha^0) &= \alpha^0 + \alpha^6 + \alpha^0 = \alpha^6 \neq 0 \\ \sigma(\alpha^1) &= \alpha^2 + \alpha^7 + \alpha^0 = \alpha^2 \neq 0 \\ \sigma(\alpha^2) &= \alpha^4 + \alpha^8 + \alpha^0 = \alpha^6 \neq 0 \\ \sigma(\alpha^3) &= \alpha^6 + \alpha^9 + \alpha^0 = \mathbf{0} \Rightarrow \mathbf{ERRO} \\ \sigma(\alpha^4) &= \alpha^8 + \alpha^10 + \alpha^0 = \mathbf{0} \Rightarrow \mathbf{ERRO} \\ \sigma(\alpha^5) &= \alpha^10 + \alpha^11 + \alpha^0 = \alpha^2 \neq 0 \\ \sigma(\alpha^6) &= \alpha^12 + \alpha^12 + \alpha^0 = \alpha^0 \neq 0\end{aligned}$$

Como visto na Equação 4.27, as localizações dos erros são a inversa das raízes do polinômio. Então, $\sigma(\alpha^3) = 0$ indica que uma raíz existe em $1/\beta_l = \alpha^3$. Logo, $\beta_l = 1/\alpha^3 = \alpha^4$. Similarmente, $\sigma(\alpha 4) = 0$ indica que outra raíz existe em $1\beta_{l'} = 1/\alpha^4 = \alpha^3$, onde (para esse exemplo) l e l' referem-se ao 1^0 e 2^0 erro. Como existem 2 símbolos de erros aqui, o polinômio de erro é configurado da seguinte forma:

$$e(X) = e_{j_1} X^{j_1} + e j_2 X^{j_2} \tag{4.34}$$

Os dois erros foram encontrados nas localizações α^3 e α^4. Note que o índice do número de localização do erro é completamente arbitrário. Então, para esse exemplo, podem ser escolhidos os valores $\beta_l = \alpha^{j_l}$ como $\beta_1 = \alpha^{j_1} = \alpha^3$ e $\beta_2 = \alpha^{j_2} = \alpha^4$.

4.4.3.3 Valores dos Erros

Um erro é denotado por e_{jl} , onde o índice j refere-se à localização do erro e o índice l identifica o l_{esimo} erro. Uma vez que cada valor de erro está associado a uma localização

particular, a notação e_{jl} pode ser simplificada por e_l. Agora, preparado para determinar os valores de erro e_1 e e_2, associados com as localizações $\beta_1 = \alpha^3$ e $\beta_2 = \alpha^4$, qualquer uma das quatro equações de síndrome pode ser usada. Da Equação 4.26, são usados S_1 e S_2:

$$\begin{aligned} S_1 = r(\alpha) = e_1\beta_1) + e_2\beta_2 \\ S_2 = r(\alpha^2) = e_1{\beta_1}^2 + e_2{\beta_2}^2 \end{aligned} \tag{4.35}$$

Nós podemos escrever essas equações em forma de matriz como segue:

$$\begin{bmatrix} \beta_1 & \beta_2 \\ {\beta_1}^2 & {\beta_2}^2 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \end{bmatrix} \tag{4.36}$$

$$\begin{bmatrix} \alpha^3 & \alpha^4 \\ \alpha^6 & \alpha^8 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \end{bmatrix} = \begin{bmatrix} \alpha^3 \\ \alpha^5 \end{bmatrix} \tag{4.37}$$

Para encontrar os valores de erro e_1 e e_2, a matriz na Equação 4.37 é invertida resultando em:

$$\begin{bmatrix} \alpha^2 & \alpha^5 \\ \alpha^0 & \alpha^4 \end{bmatrix} \tag{4.38}$$

Agora, resolvendo a Equação 4.37 para os valores de erros tem-se:

$$\begin{bmatrix} e_1 \\ e_2 \end{bmatrix} = \begin{bmatrix} \alpha^2 & \alpha^5 \\ \alpha^0 & \alpha^4 \end{bmatrix} \begin{bmatrix} \alpha^3 \\ \alpha^5 \end{bmatrix} = \begin{bmatrix} \alpha^5 & \alpha^10 \\ \alpha^3 & \alpha^9 \end{bmatrix} = \begin{bmatrix} \alpha^5 & \alpha^3 \\ \alpha^3 & \alpha^2 \end{bmatrix} = \begin{bmatrix} \alpha^2 \\ \alpha^5 \end{bmatrix} \tag{4.39}$$

4.4.3.4 Correção do Polinômio Recebido

O polinômio de erros é formado a partir das Equações 4.34 e 4.39, resultando no polinômio apresentado na Equação 4.40.

$$\begin{aligned} \overline{e}(X) &= e_1X^{j_1} + e_2X^{j_2} \\ &= \alpha^2X^3 + \alpha^5X^4 \end{aligned} \tag{4.40}$$

O algoritmo demonstrado repara o polinômio recebido, resultando em uma estimativa da palavra-código transmitida, recuperando a mensagem original armazenada na memória. A Equação 4.41 apresenta o processo de restauração da palavra-código armazenada.

$$
\begin{aligned}
\overline{U}(X) &= r(X) + \overline{e}(X) = U(X) + e(X) + \overline{e}(X) \\
r(X) &= (100) + (001)X + (011)X^2 + (100)X^3 + (101)X^4 + (110)X^5 + (111)X^6 \\
\overline{e}(X) &= (000) + (000)X + (000)X^2 + (001)X^3 + (111)X^4 + (000)X^5 + (000)X^6 \\
\overline{U}(X) &= (100) + (001)X + (011)X^2 + (101)X^3 + (010)X^4 + (110)X^5 + (111)X^6 \\
&= \alpha^0 + \alpha^2 X + \alpha^4 X^2 + \alpha^6 X^3 + \alpha^1 X^4 + \alpha^3 X^5 + \alpha^5 X^6
\end{aligned}
\tag{4.41}
$$

4.5 Conclusão

Códigos corretores de erros têm sido muito utilizados em sistemas de telecomunicações para transmissão de mensagens. Contudo, estudos demonstram a eficiência destes códigos para correção de *bits* em memórias (NEUBERGER et al., 2003). Nos próximos capítulos é apresentada uma arquitetura de um microprocessador de propósito geral e seus pontos vulneráveis à radiação. Além disso, é feito uma análise de como estes pontos podem ser trabalhados com técnicas de tolerância a falhas e em alguns pontos com o uso de códigos corretores de erros apresentados neste Capítulo.

Capítulo 5

Processador OpenRISC 1200

O microprocessador OpenRISC 1200 (OR1K, 2009) é um *softcore processor* de 32 bits mantido pelo grupo OpenCores (CORES, 2010), sendo uma iniciativa da comunidade de *hardware* para o desenvolvimento de *soft-cores* de domínio público. O objetivo da comunidade OpenCores é obter uma plataforma de microprocessador que seja facilmente estendida com novos componentes de *hardware* e assim versátil o suficiente para suportar diferentes domínios de aplicações.

O OpenRISC 1200 tem o seu *core* disponível sob a licensa GNU GPL (*General Public License*), ou seja, possui o mesmo direito de uso do sistema operacional Linux. Neste caso, pode-se obter o código fonte, gratuitamente, do microprocessador e alterá-lo, de forma a customizá-lo para a aplicação na qual se deseja utilizar.

Além de ser um microprocessador de 32 *bits* e de código fonte gratuito, possui também, em seu núcleo, um conjunto de instruções de DSP (*Digital Signal Processor*) o que permite a utilização deste processador em aplicações que envolvam processamento digital de sinais. A vantagem de se utilizar este microprocessador, para estas aplicações, é o do processamento DSP ser executado em *hardware* e não por *software*, aumentando assim o desempenho do sistema.

5.1 Arquitetura do OpenRISC 1200

O microprocessador OpenRISC 1200 é baseado na arquitetura MIPS (*Microprocessor without Interlocked Pipeline Stages*), que por sua vez é uma arquitetura de processador do tipo RISC (*Reduced Instruction Set Computing*). Este microprocessador de 32 *bits* é projetado com ênfase em desempenho, simplicidade, baixo consumo de energia, escalabilidade e versatilidade, tendo o seu núcleo codificado na linguagem de descrição de *Hardware*

Verilog.

A arquitetura do microprocessador OpenRISC 1200 é apresentada no diagrama de componentes da Figura 5.1, o qual é composto por um microprocessador (CPU), uma unidade de gerenciamento de memória (IMMU / DMMU), uma unidade de *cache* (ICache / DCache), um controlador de interrupções programável (PIC), uma unidade de depuração (DEBUG), um temporizador (*TICK TIMER*), uma unidade avançada de gerenciamento de energia (POWERM) e um barramento para conexão de componentes externos (WBI / WBD). Nas próximas seções é feita uma descrição detalhada de cada um dos blocos de componentes presentes na Figura 5.1.

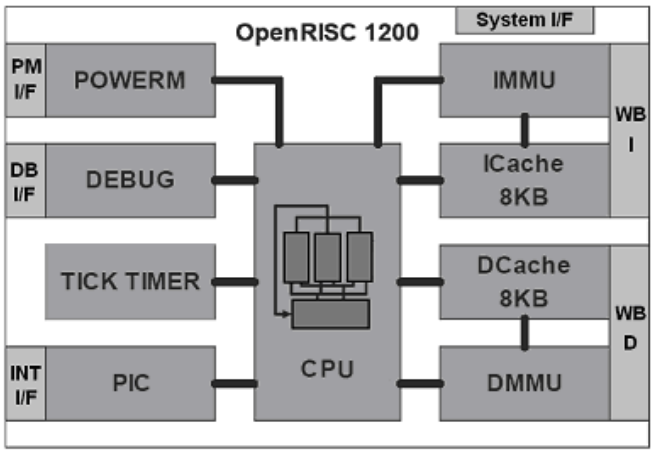

Figura 5.1: arquitetura do OpenRISC 1200, adaptado de (OPENCORES, 2001)

5.2 A CPU do OpenRISC 1200

A Unidade Central de Processamento (*Central Processing Unity - CPU*) tem como função principal o processamento e execução dos programas armazenados na memória principal, mais especificamente, a CPU é responsável pela entrada de dados, saída de resultados, cálculos, comparações, tomada de decisões, emissão de sinais para controlar o processamento e comunicação entre esta e seus dispositivos de entrada e saída.

No microprocessador OpenRISC 1200 a arquitetura da CPU é composta por uma unidade de instrução (*Instruction Unit*), uma unidade de exceção (*Exception*), uma unidade de Sistema (*System*), uma unidade MAC (*MAC Unit*), uma unidade de carga / armazenamento (*Load / Store Unit*) e uma unidade de *pipeline* de operações sobre inteiros (*Integer EX Pipeline*).

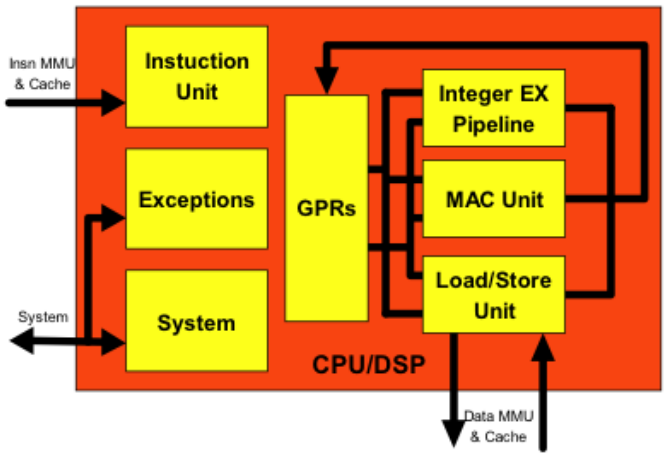

Figura 5.2: arquitetura da CPU do microprocessador openRISC 1200, adaptado de (OPENCORES, 2001)

5.2.1 Unidade de Instrução (*Instruction Unit*)

A unidade de instruções do OpenRISC 1200 implementa um *pipeline* básico de instruções, ou seja, busca instruções do subsistema de memória, despacha-as para a unidade de execução disponível, mantendo um histórico do estado, de modo a poder assegurar um modelo preciso de exceções e que as operações sejam terminadas na ordem correta. A unidade de execução também executa intruções *branch* condicional e *jump* incondicional, além de ser capaz de detetar quando os dados estão disponíveis e certificar-se que não existe outra instrução que irá utilizar o mesmo registro de destino.

A unidade de instruções do OpenRISC 1200 inclui instruções para a multiplicação e a divisão implementadas em *hardware*, sendo que na multiplicação a latência é de 3 ciclos de *clock*, e na divisão a latência é estimada em 64 ciclos de *clock* (ORSRC, 2009).

5.2.2 Registradores de Propósito Geral (*GPRs - General Purpose Registers*)

Os registradores são unidades de armazenamento baseadas em *flip-flops*, utilizados para a manipulação, movimentação e transferência de dados. Podem ser tanto registradores específicos como registradores de endereço, ou ainda registradores de propósito geral, utilizados para qualquer tipo de dado.

O microprocessador OpenRISC 1200 implementa 32 registradores de propósito geral de 32 *bits*. Sua arquitetura suporta o compartilhamento de cópias de registradores de arquivos, o que possibilita trocas rápidas de chaveamento de contexto. A forma como o OpenRISC 1200 implementa os registradores de arquivos de propósito geral, é através de

duas memórias síncronas *dual-port*, com capacidade de 32 palavras de 32 *bits* cada.

5.2.3 Unidade de Carga/Armazenamento (LSU - *Load/Store Unit*)

A Unidade de carga/armazenamento (LSU) tem a função de processar os pedidos das instruções, para as operações de leitura e escrita na memória. Quando a instrução solicita os dados armazeados na memória, tem-se uma operação de *load*, contudo quando a instrução solicita que os dados sejam gravados na memória, tem-se uma operação de *store*.

No microprocessador OpenRISC 1200 a unidade LSU transfere todos os dados entre os GPRs e o barramento interno da CPU. A LSU é implementada como uma unidade de execução independente, de modo que paradas que ocorram no subsistema de memória afetem o *pipeline* principal apenas se existir dependência entre os dados manipulados. As principais características da LSU do OpenRISC 1200 incluem a implementação exclusiva de todas as unidades de *load* e *store* em *hardware*, *buffer* para os endereços de entrada, operação em *pipeline* e alinhamento de endereços para acessos rápidos à memória.

Para a realização de uma instrução de *load* ou de *store*, a LSU do OpenRISC 1200 verifica se todos os operandos estão disponíveis. A sequência executada pela LSU consiste em verificar o operando com o registro do endereço, no qual se deseja carregar ou armazenar os dados, em seguida, verificar o operando com o registro dos dados, para o caso de instruções de *store*, ou o operando com os registros de dados do destino, para o caso de instruções de *load*.

5.2.4 *Pipeline* de Operações sobre Inteiros (*Integer Execution Pipeline*)

A unidade de execução é o componente básico de qualquer microprocessador, pois é nesta unidade que são processadas as operações envolvendo números inteiros. O OpenRISC 1200 implementa os tipos de instruções de inteiros de 32 *bits*, apresentadas na lista abaixo:

- instruções aritméticas;
- instruções de comparação;
- instruções lógicas;

- instruções *shift* e *rotate*.

No OpenRISC 1200 a maioria das instruções de inteiros podem ser executadas em um único ciclo.

5.2.5 Unidade de DSP *MAC Unit*

A unidade de processamento digital de sinais (DSP) do OpenRISC 1200, possui um conjunto de instruções capaz de realizar as operações mais habituais em processamento digital de sinais, que são as de adição, multiplicação e transferência de memória. Este conjunto de operações tem o nome de MAC (multiplicador e acumulador).

No OpenRISC 1200 as instruções MAC são implementadas em *hardware* com operações de 32 bits. A unidade MAC tem duas entradas de 32 bits, e gera como resultado uma saída de 64 bits levando 3 ciclos por operação. Contudo, como as unidades MAC executam em paralelo com o restante do processador, estas podem aceitar uma nova operação a cado novo ciclo de *clock*. A unidade MAC representa aproximandamente 22% da área total do OpenRISC 1200 e é responsável por cerca de 11% do consumo de energia deste processador.

5.2.6 Unidade de Sistema (*System Unit*)

No OpenRISC 1200 a unidade de sistema tem a função de implementar os registradores de propósitos especiais (SPRs) da CPU. Esta unidade conecta todos os sinais da CPU/DSP que não são conectados através da interface de instruções e dados. Além disso, por estar conectada diretamente à unidade de exceções pode, através de seus registradores, lançar uma exceção.

5.2.7 Exceções (*Exceptions*)

As exceções na CPU do OpenRISC 1200 são geradas quando uma fonte indica que ocorreu uma situação especial como, por exemplo, quando uma interrupção externa é acionada, ou quando acontece um erro ao acessar uma região de memória. Além disso, outros casos podem gerar uma exceção no microprocessador, em particular no caso de execução de um *opcode* de instrução que não está presente no OpenRISC 1200, ou por chamadas de sistemas e também por *breakpoints*.

Quando uma determinada exceção ocorre no OpenRISC 1200, o controle é transferido

para uma rotina responsável por atender tal exceção e, assim, carregar o contador de programa *Program Counter* com o endereço do tratador apropriado.

5.3 Cache de Dados e Instruções (*Data and Instruction Cache*)

A memória *cache* é uma memória embutida no processador que serve para armazernar os dados usados com mais frequências. Ela evita, na maioria das vezes, que seja necessário recorrer à memória secundária (RAM), muito mais lenta que a *cache*. A idéia por trás da *cache* é muito simples: colocar na memória *cache* os dados e instruções que são mais comumente utilizados pelo processador.

No OpenRISC 1200 as *caches* de dados e de instruções são configuráveis em vários tamanhos (1KB, 2KB, 4KB e 8KB). As *caches* usadas neste microprocessador seguem o modelo *Harvard*, ou seja, são separadas em *cache* de dados e de instruções. Além disso, as *caches*, quer de dados ou de instruções, são de uma única entrada e mapeadas diretamente no microprocessador.

A *cache* do OpenRISC 1200 usando a configuração padrão está organizada em 512 linhas, sendo que cada linha consiste de 16 bytes de dados, alguns *bits* de estado e endereço das *tag*, conforme apresentado na Figura 5.3. A cache é preenchida através de sequências de 16 *bytes* sempre que a tag e os *bits* de estado correspondentes no endereço são diferentes. Neste preenchimento a palavra é escrita simultaneamente na *cache* e direcionada para a unidade que realizou o pedido, o que minimiza as paradas devido à latência associada ao preenchimento da *cache*.

A *cache* de dados permite que as *tags* sejam armazenadas e executadas para substituição das linhas de memória. No caso da *cache* de instruções, esta se comunica com a unidade de busca (*fetch*), de modo que esta última permite o cálculo do endereço efetivo.

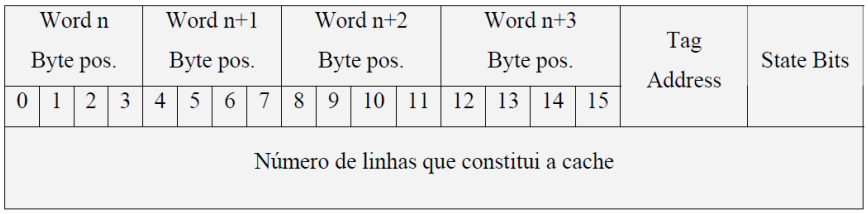

Figura 5.3: estrutura da Cache

A configuração padrão das *caches* é de 8KB, sendo possível ter valores para dimensões de 1KB até 8KByte, o que indica, admitindo *bursts* de 16 *bytes*, que tem-se de 64 até 512

linhas nesta memória.

5.4 Unidade de Gerenciamento de Memória (MMU - *Memory Management Units*)

O OpenRISC 1200 está dotado de memória virtual, realizando a separação entre dados e instruções no tratamento da memória virtual, adotando-se uma arquitetura do tipo Harvard para a mesma.

O valor para o tamanho do TLB (*Translation lookaside buffer*) é variável entre 16 e 256 entradas, quer para o caso de dados, quer para o caso de instruções, sendo este por padrão de uma única entrada.

O espaço de endereçamento é linear sendo que os endereços virtuais são de 32 bits e os endereços físicos de 24 a 32-bits. Existe ainda um esquema que trata da protecção das páginas. O tamanho das páginas é de 8KB estando inerente a cada uma delas os respectivos atributos.

5.5 Gerenciamento de Energia (*Power Management*)

A unidade de gerenciamento de energia, integrada ao OpenRISC 1200, tem como função principal permitir a redução do consumo de energia deste microprocessador, através do controle da CPU e de seus periféricos. Esta unidade é capaz de reduzir o consumo de energia de 2 a 100 vezes e ainda alterar a frequência de operação do OpenRISC 1200, configurando a CPU para o modo de funcionamento de baixa frequência ou modo *idle*. Com esta unidade é possível ativar dispositivos que estavam adormecidos, ou desativar dispositivos que não estão sendo utilizados, através de interrupções. Além disso, tem incorporado um gatilho de relógio (*clock gating*) de modo que o sinal de *clock* somente seja recebido por um determinado dispositivo quando o sinal de controle, daquele dispositivo, estiver ativo, potencializando assim a eficiência da utilização da energia.

5.6 Unidade de Depuração (*DEBUG*)

A unidade de depuração (*debug*) é um mecanismo incorporado a um microprocessador que permite a sua execução controlada, habilitando-o a iniciar, parar e continuar gradualmente o processamento através de um dispositivo/*software* externo conectado à sua

unidade de depuração.

O OpenRISC 1200 tem incorporado uma unidade que permite a realização de depuração. O modo de *debug* não interfere com o funcionamento do OpenRISC 1200 e do sistema. A unidade de *debug* permite que se faça o supervisionamento do OpenRISC 1200 e do sistema em tempo de execução, sendo possível ter acesso ao controle da unidade para realizar o depuramento deste diretamente através da JTAG (Joint Test Action Group) por meio da interface de depuração de *software* GNU GDB.

5.7 Temporização Integrada (*Tick Timer*)

Um temporizador é um dispositivo capaz de medir o tempo, sendo um tipo de relógio especializado. Este pode ser utilizado para controlar a sequência de um evento ou processo. Temporizadores podem ser mecânicos, eletromecânicos, digitais, ou mesmo programas de computador.

O OpenRISC 1200 tem implementado internamente um temporizador *tick timer* ligado diretamente ao seu *clock*. Este temporizador pode ser utilizado pelo sistema operacional para realizar medições de tempo e agendar tarefas do sistema. O temporizador permite que se faça a contagem de 2^{32} ciclos de *clock* e é limitado ao período máximo, entre interrupções, de 2^{28} ciclos. Além disso, o *tick timer* possui ainda outros modos de operação como o *single run*, gerando uma única interrupção, e o *continues timer*, no qual são geradas temporizações bem determinadas.

5.8 Controlador de Interrupções Programável (*Programmable Interrupt Controller*)

O controlador de interrupções recebe interrupções externas direcionando-as para a CPU, atribuindo-lhes níveis de prioridade. No OpenRISC 1200 existem duas interrupções que não podem ser desativadas, respectivamente a 0 e a 1, de um total de 32 interrupções. Desta forma, tem-se disponíveis apenas 30 interrupções mascaráveis. Estas 30 interrupções podem ser mascaradas de forma a servirem aos objetivos específicos de uma determinada aplicação.

5.9 Barramento *Wishbone*

Na arquitetura de um computador, um barramento é um conjunto de linhas de comunicação que permitem a interligação entre dispositivos, como a CPU, a memória e outros periféricos. No caso do OpenRISC 1200 o padrão de barramento adotado para a interconexão de seus periféricos é o do barramento *Wishbone*.

Os desenvolvedores do barramento de comunicação *Wishbone* foram influenciados por três fatores principais. Primeiro, havia a necessidade de uma solução boa e confiável para a integração de núcleos de *hardware* em SOCs (*System-on-Chip*). Segundo, havia a necessidade de uma especificação de uma *interface* comum para facilitar as metodologias de projeto estruturadas para grandes equipes de projeto. Terceiro, eles foram influenciados pelas soluções de integração de sistemas tradicionais, fornecidos por barramentos de microcomputador como o PCI, DMA, dentre outros.

De fato, a arquitetura do *Wishbone* é análoga a um barramento de microcomputador, sendo que: oferece uma solução flexível para integração que pode ser facilmente adaptada a uma aplicação específica; oferece uma variedade de ciclos de acesso ao barramento e de larguras de caminhos de dados para atender a diferentes sistemas; e permite que os núcleos de *hardware* sejam projetados por vários fornecedores.

Em janeiro de 2001, a organização OpenCores adotou o *Wishbone* como um padrão de conectividade entre seus núcleos de *hardware*. Este barramento foi escolhido por ser o único a atender aos requisitos adotados por esta instituição que incluem o uso de um barramento flexível e simples de utilizar, além de ser atualmente completamente aberto, pois sua criadora, a empresa *Silicore Corporation*, tornou-o de domínio público.

5.10 *Software* e Ferramentas para o OpenRISC 1200

O OpenRISC 1200 é um microprocessador de propósito geral que pode ser programado diretamente através de codificação e compilação de um *firmware*, utilizando as ferramentas de compilação e depuração baseadas no GNU *toolchain*. O GNU toolchain é um termo que agrupa uma série de projetos que contêm as ferramentas de programação desenvolvidas pelos integrantes do projeto GNU. Estes projetos formam um sistema integrado que é usado para programar tanto aplicações como sistemas operacionais. Neste sentido pode-se utilizar as ferramentas GCC, para compilação, e GDB, para depuração, das aplicações desenvolvidas para o OpenRISC 1200.

Além do microprocessador OpenRISC 1200 suportar a programação direta, ou seja,

através de *firmware*, este também suporta executar alguns sistemas operacionas como o Linux, o uClinux, OAR RTEMS *real-time* OS.

5.11 Conclusão

Os sistemas embarcados frequentemente têm requisitos de restrição de espaço, peso, consumo de energia e desempenho computacional. O microprocessador OpenRISC 1200 é uma excelente opção para os sistemas embarcados comuns, pois como vimos neste capítulo, possui alto nível de integração e suporta vários sistemas operacionais modernos, alguns destes inclusive são sistemas operacionais de tempo real. Apesar de ser um microprocessador muito versátil, o OpenRISC 1200 não possui requisitos temporais e de tolerância a falhas que torne possível seu uso em aplicações de tempo real.

Algumas alternativas podem ser consideradas para permitir a aplicação do openRISC 1200 em sistemas críticos, que são sistemas computacionais, para os quais uma falha pode ocasionar danos catastróficos e mesmo perda de vidas humanas. A primeira delas é implementar técnicas de tolerância a falhas em nível sistêmico, como por exemplo inserir processadores OpenRISC 1200 redundantes na placa principal do sistema implementando assim um sistema TMR. Esta abordagem é bastante utilizada até o presente momento. No entanto, o avanço contínuo das restrições acima citadas (espaço, peso e consumo) vão de encontro a esta alternativa.

Uma outra opção é implementar o processador openRISC 1200 em FPGAs tolerantes a radiação. Estas FPGAs estão em geral algumas gerações atrasadas em relação aos processos de manufatura do estado da arte, e devido ao seu baixo volume de produção, têm custo elevado.

Uma outra alternativa para permitir a utilização do OpenRISC 1200 em sistemas críticos, á a de re-projetar o OpenRISC 1200 aplicando ténicas de tolerância a radiação em nível de projeto (RHBD - Radiation Hardening By Design), ou seja, projetar um microprocessador OpenRISC 1200 tolerante a falhas em nível de projeto, o qual denomina-se neste trabalho de FT-openRISC 1200 (Fault Tolerante OpenRISC 1200). As características do FT-openRISC1200 vão ao encontro das restrições de sistemas embarcados e permite que o mesmo seja manufaturado utilizando FPGAs, produzidas com os mais recentes processos de fabricação de circuitos integrados, conforme veremos no Capítulo seguinte.

Capítulo 6

FT-OpenRISC 1200: Um Processador Tolerante a Falhas

As memórias de dados utilizadas por um miprocessador normalmente são do tipo SRAM (Static Random Access Memory), sendo estas susceptíveis a SEUs (KASTENSMIDT; CARRO; REIS, 2006; HUANG et al., 1998). Caso os erros de memória de dados armazenados nestas memórias não sejam tratados, estes podem ser disseminados para outras partes do sistema e provocar uma falha catastrófica. Para detectar e corrigir erros na memória de dados existem técnicas bastante efetivas implementadas em *software*. No entanto, as técnicas aplicadas em nível de *software* para detectar e corrigir erros na memória de dados consomem muito processamento por parte da CPU do microprocessador, sendo normalmente executadas sequencialmente, ou seja, a CPU deve parar a execução do que estiver fazendo para então processar os algorítmos de detecção e correção de erros em memória. Nesse sentido, o uso de técnicas em nível de *hardware* para detectar e corrigir erros na memória aumenta o desempenho do sistema, pois estas técnias podem ser executar em paralelo com a CPU de um sistema microprocessado sem impactar o seu desempenho. Além disso, o mascaramento de falhas em nível de *hardware* são importantes para aplicações espaciais devido ao fato do sistema não precisar de tempo adicional para detectar o erro, corrigí-lo e, se for o caso, recuperar-se evitando assim a falha.

Falhas podem ocorrer em qualquer um dos módulos do OpenRISC 1200 descrito no capítulo anterior quando expostos à radiação. Neste sentido, alterações em nível de arquitetura devem ser feitas, para prover, a este processador, tolerância aos efeitos da radiação em seu circuito digital. O restante deste capítulo trata das técnicas de tolerância a falhas implementadas neste processador.

6.1 Pontos de Falhas no OpenRISC 1200

Nos capítulos anteriores foram destacados os principais pontos de falhas, provenientes da radiação, em um semicondutor. No caso de microprocessadores estes pontos se resumem aos registradores (flip-flops e latchs), a lógica combinacional e aos elementos de memória, como por exemplo a *cache*.

No OpenRISC 1200 as unidade dos registradores e da memória *cache* não possuem proteção contra SEU e SET, ou seja, caso ocorra uma inversão de *bits*, em uma destas unidades, o processador poderá gerar um erro e este levar o sistema a falhar. Um exemplo típico é o de um cálculo matemático no qual a operação a ser realizada é de uma soma entre dois números, mas uma inversão de *bits* na ULA (Unidade Lógica Aritmética), poderá alterar esta operação para uma de subtração, afetando todo o restante das outras operações que dependem deste resultado.

Para garantir o correto funcionamento do OpenRISC 1200, mesmo quando da incidência de radiação no seu circuito digital, o primeiro ponto que deve-se atacar é a proteção dos registradores e da lógica combinacional da sua unidade central, ou seja, a CPU. Na Figura 6.1 pode-se observar que a CPU é a unidade central que controla todas as operações e comunicações entres os diversos componentes do OpenRISC 1200. Devido a isso, esta é considerada a unidade que primeiro deve ser protegida, caso contrário, o resultado do processamento poderá estar errôneo.

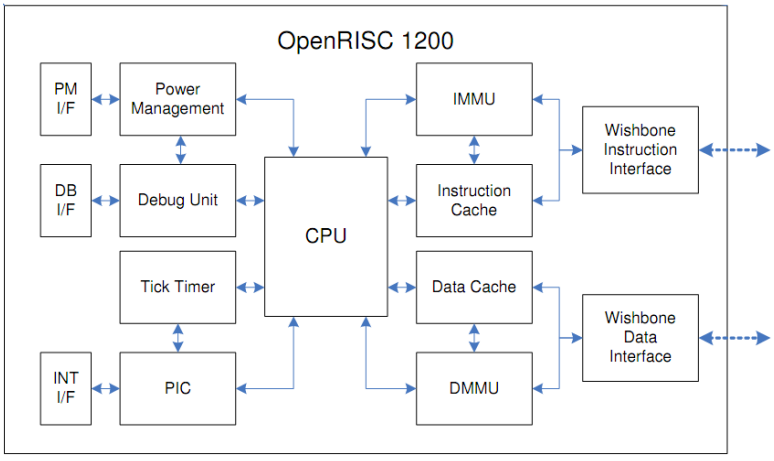

Figura 6.1: diagrama de blocos do OpenRISC 1200 original.

Outro ponto que deve ser observado está na proteção da memória *cache* do OpenRISC 1200. Na Figura 6.2 é apresentado o fluxo pelo qual os dados passam até chegar ao processador. Pode-se notar que o acesso à memória (*Memory Access*) é um ponto crítico, pois nesta região podem ocorrer inversões de *bits*. Neste sentido, o acesso à memória

cache (*Memory Access Stage*) do OpenRISC 1200 deve ser protegida, pois da forma como é implementada, se ocorrer um SEU em sua região, os valores armazenados terão seu valores alterados. Para solucionar esta vulnerabilidade no OpenRISC 1200, propõe-se a utilização de técnicas de correção de erros nesta memória a fim de proteger os dados por esta armazenados.

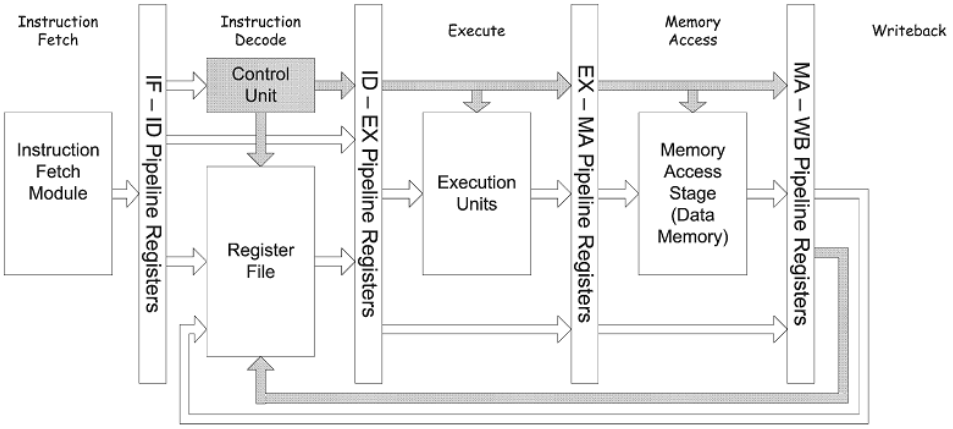

Figura 6.2: diagrama de blocos de execução e acesso a memória no OpenRISC 1200 original.

Nas seções seguintes são apresentadas as técnicas de tolerância a falhas aplicadas neste processador para corrigir suas vulnerabilidades, tornando-o apto a ser utilizado, por exemplo, em aplicações espaciais. Foram realizados testes de injeção de falhas no microprocessador FT-OpenRISC 1200 para garantir a eficácia das técnicas de mascaramento, bem como a eficácia dos códigos corretores de erros aplicadas em nível de projeto do FT-openRISC 1200. Estes testes também serão descritos em seções posteriores.

6.2 Detecção de Erros e Tolerância a Falhas

Atualmente, a principal fonte de falhas transientes nos circuitos integrados é a radiação (BAUMA, 2005; BAUMANN, 2001). Partículas ionizantes, advindas principalmente da atividade solar, ao colidirem com uma região sensível de um circuito, como por exemplo o dreno de um transistor que se encontra desligado (O'BRYAN et al., 2002; MESSENGER, 1982), podem ocasionar a geração de uma trilha de ionização entre o dreno e o substrato do transistor, permitindo o estabelecimento de um pulso de corrente (O'BRYAN et al., 2002). Se essa corrente possuir amplitude e duração suficientes para carregar, ou descarregar um nó de um circuito, gerando assim um pulso de tensão transiente, este pode vir a ser interpretado como uma mudança de nível lógico. Caso o mesmo não venha a

ser mascarado logicamente, ou eletricamente, ou atinja o registrador de saída dentro da janela de amostragem (*latching window*), uma falha não-permanente ocorre no circuito.

Por outro lado, se uma partícula energizada colidir com uma região sensível de um circuito combinacional, ocasionando a geração de um pulso transiente de tensão, tem-se o fenômeno chamado de SET (*Sigle-Event Transient*) (ALEXANDRES; ANGHEL; NICOLAIDIS, 2002; ALEXANDRESCU; ANGHEL; NICOLAIDIS, 2004; VIOLANTE, 2003; ANGHEL; NICOLAIDIS, 2000; ANGHEL; LEVEUGLE; VANHAUWAERT, 2005). Além disso, se uma partícula colide com uma região sensível de um elemento de memória, isso provoca uma mudança do valor lógico armazenado, também chamado de *bit-flip*, que é um exemplo de *Single-Event Upset* ou simplesmente SEU (BAUMA, 2005; NICOLAIDI, 2005; DODD; MASSENGILL, 2003).

A maioria das técnicas de alto nível utilizadas atualmente, tanto para proteção contra SEUs quanto para proteção contra SETs, baseia-se em redundância de *hardware* (CARMICHAEL, 2006), redundância temporal (NICOLAIDIS, 1999) ou em uma combinação de ambas (KASTENSMIDT; CARRO; REIS, 2006). Neste trabalho, propõe-se uma arquitetura baseada em redundância de hardware, que utiliza redundância modular tripla e uma combinação de dois códigos corretores de erros, Hamming e Reed-Solomon.

6.2.1 Registradores tolerantes a SEU e SET

A proteção dos *flip-flops* é fundamental para a tolerância a falhas em um processador, pois os registradores e *latches* podem sofrer inversão de *bit*, causando erro no evento de uma leitura, que pode se propagar, alterando assim o resultado de um dado processamento. Para resolver este tipo de problema a técnica de TMR é normalmente a mais usada para detectar e corrigir os *bits* com poucos prejuízos de desempenho, pois os dados são processados em paralelo e enviados ao votador, que se encarrega de fazer a comparação e enviar o resultado desta votação. Neste contexto, o FT-OpenRISC 1200 usa um projeto tolerante a SEU e SET, baseado no uso de TMR, no intuito de manter o correto funcionamento quando da presença destes tipos de eventos. Assim, a meta para o FT-OpenRISC 1200 é a de tolerar uma falha em qualquer registro, sem a intervenção de *software*, suprimindo assim os efeitos do SEU e SET na lógica combinacional e sequencial.

No projeto utiliza-se redundância modular tripla para proteger os *flip-flops* do OpenRISC 1200, os quais são usados principalmente para armazenamento temporário e máquinas de estado. Este tipo de redundância consiste do uso de três registros em paralelo e um votador que seleciona o resultado da maioria, como mostrado na Figura 6.3.

Os *flip-flops* são continuamente alterados a cada *clock* e, como resultado, qualquer erro de SEU é automaticamente removido dentro de um ciclo de *clock* pois o votador evita que esse erro seja propagado. O benefício deste esquema é o mascaramento e a remoção do erro sem a necessidade de um processamento extra para análise da falha (recuperação do erro).

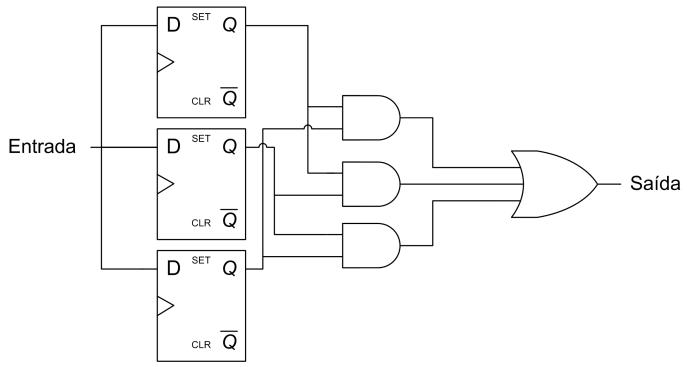

Figura 6.3: TMR no registrador.

Devido à grande quantidade de *flip-flops* presentes no processador openRISC 1200, é desenvolvido um *software*, denominado de TMRlesc, que analisa o código fonte do openRISC 1200 e, através de uma análise, converte automaticamente o código original do processador em um código com registradores tolerante a SEUs. Com este *software* pode-se programar o OpenRISC 1200 com vários tipos de projetos de circuitos digitais a fim de identificar qual a melhor relação tolerância a falhas versus área ocupada na FPGA. Através de parâmetros no *software*, pode-se alterar o tipo de TMR a ser utilizado, inclusive, se for o caso, triplicar o votador ou até mesmo utilizar outro tipo de configuração NMR. O TMRlesc é baseado na idéia proposta na ferramenta da Xilinx TMRtool (XILINX, 2004), assim como nas técnicas de projetos propostas por Carmichael (CARMICHAEL, 2006), por Gaisler Research (HABINC, 2002) e por Martin L. Shooman (SHOOMAN, 2002). Uma análise comparativa entre o funcionametno das ferramentas TMRTool e TMRlesc é bastante interessante, contudo isto não pôde ser realizado pois a Xilinx não disponibiliza a ferramenta Xilinx TMRtool para o Brasil.

6.2.2 *Software* TMRlesc

o *software* TMRlesc é um aplicativo, desenvolvido na linguagem PERL para o sistema operacional Linux, que faz uma analisa do código fonte do microprocessador OpenRISC 1200 e, através desta análise, converte automaticamente o código fonte original do processador em um código com registradores tolerante a SEUs.

Neste trabalho a concepção e o desenvolvido do *software* TMRlesc surgiu da necessidade de utilizar uma ferramenta para automatizar o processo de detecção dos registradores, descritos em linguagem Verilog, replicação destes registradores assim como a inserção de votadores. Na Figura 6.4 é apresentado uma parte do sistema replicado através do *software* TMRlesc.

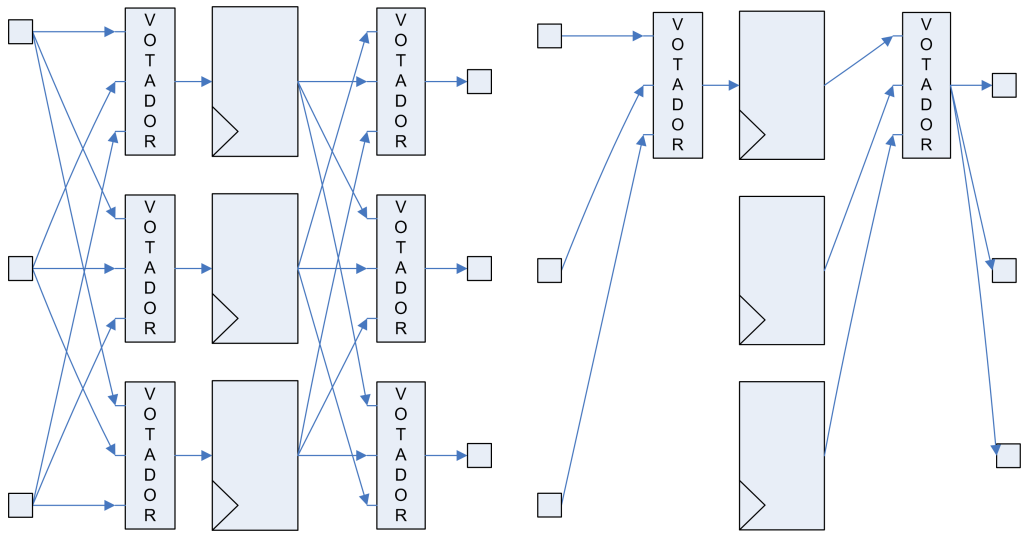

Figura 6.4: registradores replicado por TMRlesc: votador Triplicado (a) e votador simples (b)

O Fluxograma de execução do TMRlesc é apresentado na Figura 6.5. Neste processo, o *software* TMRlesc é executado no diretório raiz do projeto do OpenRISC 1200, ou seja, no diretório em que estão contidos os códigos fontes originais deste processador. Ao ser executado o TMRlesc altera as partes deste código fonte que possuam registradores, isto é possível pois este *software* possui um mecanismo inteligente de busca por registradores utilizando *parsers* e *regex* desenvolvidos especialmente para este fim. Ao encontrar estas estruturas de código de registradores, o TMRlesc altera este código fonte triplicando estas estruturas e inserindo um sistema de votação (votador), interligando assim estas alterações ao restante do sistema do OpenRISC 1200.

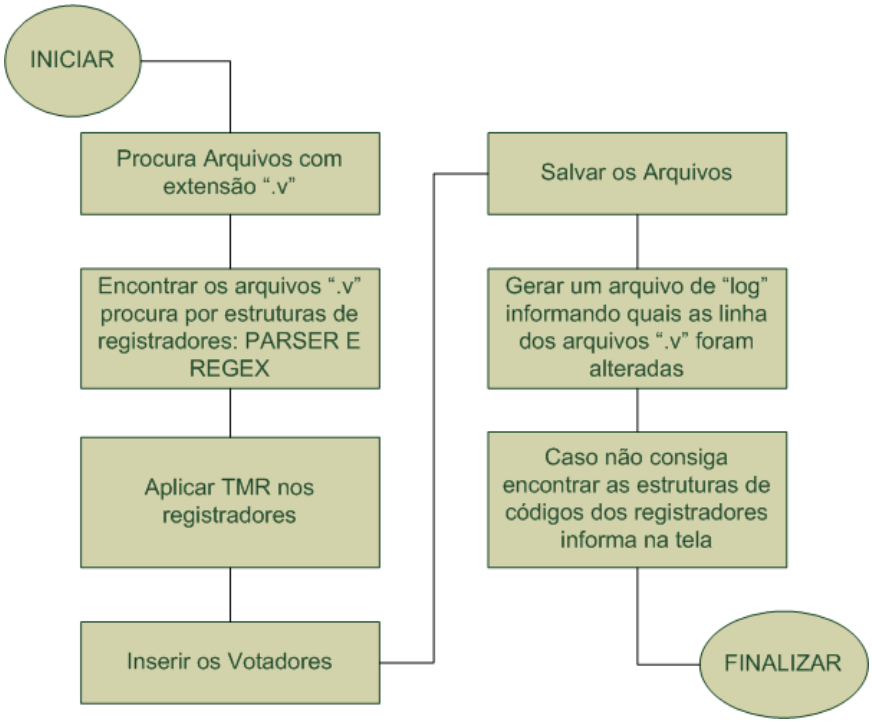

Figura 6.5: registradores replicado por TMRlesc.

Quando a execução do *software* TMRlesc é finalizada, tem-se um novo código Verilog do *core* do processador OpenRISC 1200 já com os registradores triplicados e seus votadores. Este novo código, por ter seu projeto modificado com técnicas de tolerância a falhas, utilizando adição de *hardware* (TMR), quando sintetizado gera um processador OpenRISC 1200 tolerante a SEU em nível de registrador.

6.2.3 Proteção da Lógica Combinacional

Uma única partícula ionizada pode se chocar com os circuitos combinacionais ou com regiões do processador onde se encontram implementadas as lógicas combinacionais e sequenciais, apresentando um efeito de SEU no circuito. Na Figura 6.6 é ilustrada uma topologia típica de circuito encontrado em quase todos os circuitos digitais, que mesclam a lógica combinacional com a sequencial.

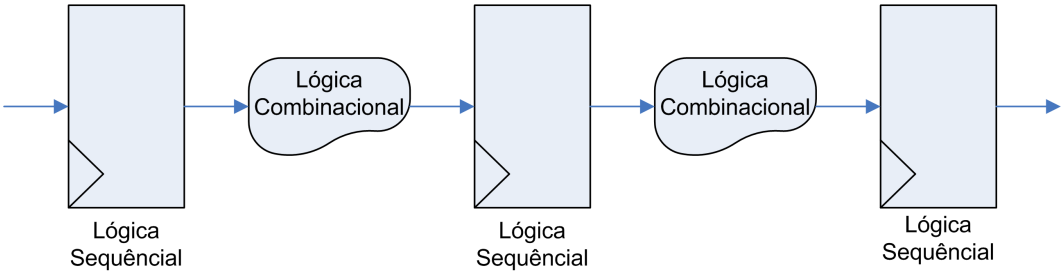

Figura 6.6: lógica sequencial e combinacional sem TMR

Seguindo uma topologia de operação de um circuito digital, como por exemplo o apresentado na Figura 6.7, tem-se que os dados do primeiro *latch* são transferidos para a lógica combinacional em um pulso de *clock*, e assim as operações de lógica são executadas, com a informação recebida pelo primeiro *latch*. A saída da lógica combinacional deve alcançar o segundo *lacth* antes do próximo *clock*. Ainda neste caso, pode-se observar que a informação que é transmitida para o segundo *latch* sofre interferência de dois *upsets*, um na lógica sequencial e outro na lógica combinacional. Isto justifica a necessidade de proteger a lógica sequencial e combinacional para evitar a propagação de erros provenientes deste tipo de falha, no processador.

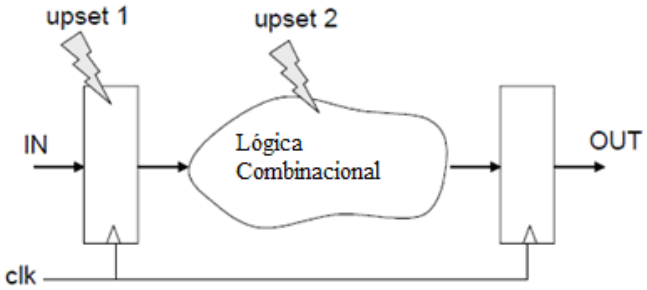

Figura 6.7: topologia de circuito digitais combinacional e sequencial

A lógica combinacional é definida como todo o circuito no qual todos os trajetos conectam diretamente as entradas com as saídas, sem formar lógica de laços. Em outras palavras, os estados da lógica dentro de um bloco de lógica combinacional não são dependentes de estados anteriores. Isto também inclui os circuitos nos quais os resultados precisam de mais de um ciclo de *clock* para ser computado.

A proteção de um bloco da lógica puramente combinacional através de TMR é realizada de forma simples e direta, sendo suficiente criar três cópias independentes do circuito original, conforme ilustrado na Figura 6.8. Assim, na ocorrência de uma falha em alguns dos módulos redundantes, os valores corretos da saída serão mascarados no circuito de votação seguinte, como mostrado na Figura 6.8.

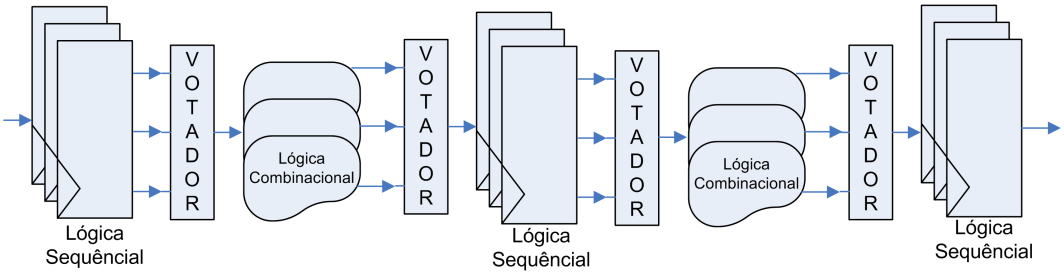

Figura 6.8: TMR para lógica combinacional e sequencial

Deve-se notar que o circuito votador não necessariamente precisa ser introduzido após os módulos de saída, pois a votação também pode ser realizada no bloco sequencial seguinte à lógica. A razão para isto é que, como os estados das saídas são sempre uma função das entradas atuais, ou seja, não é possível um erro "ser armazenado"em uma estrutura combinacional, no próximo *clock*, quando as entradas mudarem o erro já não se encontra mais na lógica. Sendo assim, quando a causa do erro estiver corrigida (por exemplo, uma entrada errônea retornar ao seu valor correto), o módulo volta a produzir outra vez as saídas corretas.

6.2.4 Proteção da memória

Experimentos em memórias sob altos fluxos de prótons e íons mostram que a probabilidade de falhas múltiplas provocadas por um único íon vem aumentando ao longo dos anos (WROBEL et al., 2001; JOHANSSON et al., 1999; BUCHNER et al., 2000). Desta forma, a proteção da memória no FT-OpenRISC 1200 é realizada através do uso de uma técnica baseada em códigos corretores de erros para proteção de memórias contra falhas múltiplas. Esta técnica empregada combina os códigos *Reed-Solomon* e *Hamming* para garantir confiabilidade em presença de falhas múltiplas com reduzidas penalidades em termos de recursos lógicos e de desempenho (NEUBERGER et al., 2003).

O código de *Hamming* (HOUGHTON, 1996) é largamente utilizado para proteger memórias contra SEU por causa de sua capacidade de corrigir falhas simples com reduzidas penalidades de área e desempenho (HENTSCHKE et al., 2002). Por outro lado, *Reed-Solomon* (HOUGHTON, 1996) é um código de correção de erros baseado em blocos, capaz de tratar múltiplos erros. Sendo assim, a combinação destas duas técnicas provê ao FT-OpenRISC 1200, a capacidade de corrigir falhas simples e múltiplas em nível de memória *cache*.

6.2.4.1 Código de Hamming

A implementação do código de Hamming é feita por um bloco combinacional responsável por codificar os dados, bloco codificador, inclusão de *bits* extras na palavra para a paridade (*latches* ou *flip-flops* extras) e outro bloco combinacional responsável por decodificar os dados, denominado bloco decodificador. O bloco codificador calcula os *bits* de paridade e pode ser implementado por um conjunto de portas lógicas *XOR* de duas entradas. O bloco decodificador é mais complexo do que o bloco codificador porque não necessita só detectar a falha, mas também deve corrigí-la. Este bloco é composto basicamente pela mesma lógica que calcula a paridade além de um decodificador que indica o endereço do *bit* que contém o erro. Já o bloco decodificador, além de possuir um conjunto de portas *XOR* de duas entradas. também possui algumas portas *AND* e inversores.

6.2.4.2 Código Reed-Solomon

O processo de codificação usando código Reed-Solomon ocorre através da divisão dos *bits* de dados em símbolos de s *bits*, seguido de uma multiplicação de cada símbolo por constantes apropriadas e, finalmente, operações booleana *XOR* entre os resultados das multiplicações para encontrar os símbolos de paridade R e S.

O algoritmo Reed-Solomon necessita de um grande número de tabelas para as multiplicações. Entretanto para implementação em *hardware*, tabelas têm um custo de operação muito alto. Neste caso, para executar a operação em um só ciclo, são necessárias tabelas para cada multiplicação, aumentando com isso a área ocupada. Na implementação utilizada, essas tabelas foram substituídas por um multiplicador otimizado que utiliza portas XOR e AND (REED; SOLOMON, 1978; NEUBERGER et al., 2003).

O processo de decodificação consiste da divisão dos *bits* recebidos em R, S e símbolos de dados, seguido da multiplicação desses símbolos por constantes, e finaliza com operações de *XOR* dos últimos resultados para encontrar $S0$ e $S1$. Se $S0$ e $S1$ são iguais a zero, nenhum erro ocorreu, caso contrário, $S1/S0$ é a localização do erro e $S0$ é o padrão do erro. Para um decodificador que só detecta se há erros, sem nenhuma correção, os passos são basicamente os mesmos do codificador.

6.2.4.3 Memória *Cache* Tolerante a Falhas

Uma memória de dados de palavra de n bits pode ser protegida contra falhas usando técnicas baseadas em códigos corretores de erros, adicionando-se blocos codificador e decodificador, além de bits extras para armazenar as paridades, conforme apresentado na

Figura 6.9. O codificador e o decodificador podem usar qualquer código de correção de erros. Uma limitação desta abordagem é que os dados são codificados nas operações de escrita e decodificados nas operações de leitura. Então, é possível que ocorra acumulação de falhas e depende da frequência das operações de leitura e escrita da aplicação. No caso de se usar código de *Hamming*, podem-se corrigir falhas em um único *bit* de memória.

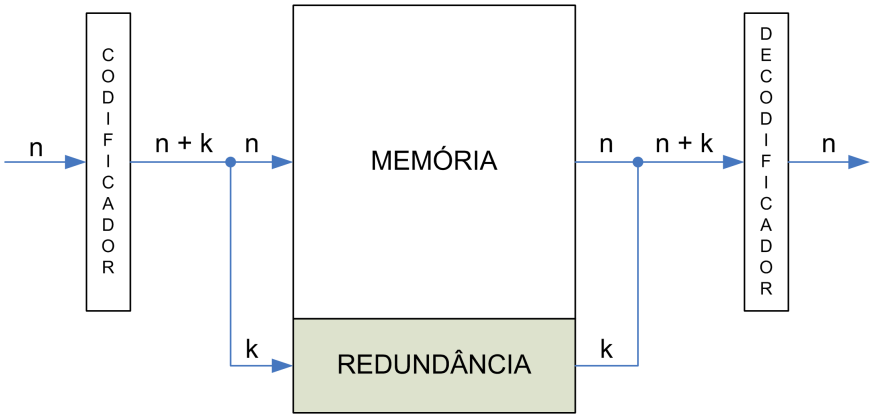

Figura 6.9: diagrama de bloco de uma memória tolerante a falhas.

Para ser capaz de corrigir falhas em múltiplos *bits* da memória, o código Reed-Solomon deve ser utilizado. A palavra de dados é dividida em símbolos, sendo que cada palavra de dados é um código Reed-Solomon diferente. Por exemplo, numa memória de 256 linhas, a palavra de dados usa a linha inteira, e assim cada palavra de dados é dividida em m símbolos de acordo com o tamanho do símbolo e do tamanho dos dados da memória. Múltiplas falhas podem ocorrer em qualquer posição da matriz de memória, mas é mais provável que ocorram em dois *bits* adjacentes que estão no mesmo símbolo (falha tipo **A**), em símbolos adjacentes verticalmente (falha tipo **B**), ou em símbolos adjacentes horizontalmente (falha tipo **C**) conforme ilustrado na Figura 6.10.

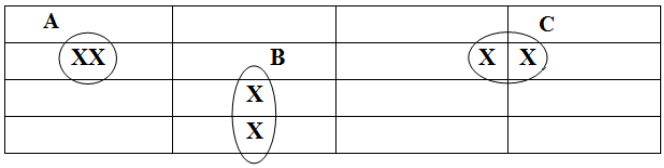

Figura 6.10: matriz de memória com os três possíveis tipos de falhas.

O código Reed-Solomon pode facilmente corrigir falhas do tipo ***A***, pois, é a propriedade essencial deste código, ou seja, múltipla correção de erro em um mesmo símbolo. Falhas do tipo ***B*** podem ocorrer e também são corrigidas pois cada linha é um código Reed-Solomon diferente, então isto é equivalente a dois erros simples em linhas distintas. Entretanto

falhas do tipo ***C*** não são corrigidas, pois é equivalente a erros em dois símbolos diferentes da mesma palavra codificada e o Reed-Solomon implementado não é capaz de corrigir este tipo de erro. Neste caso, para corrigir todos os possíveis erros duplos é necessário a utilização de um código diferente de Hamming ou de Reed-Solomon. A primeira opção é o uso de um código Reed-Solomon com capacidade de corrigir dois símbolos diferentes por palavra codificada. Porêm, este código Reed-Solomon ocupa mais que duas vezes a área e atraso do que o Reed-Solomon de correção de símbolo simples, tornando esta solução inadequada para arquiteturas de memória (Houghton, 1997). A segunda alternativa é o uso de um *bit* protegido por código de Hamming entre os símbolos Reed-Solomon. Como o número de *bits* protegido por Hamming é igual ao número de símbolos protegido por Reed-Solomon, então esta opção não aumenta significativamente o custo em área, tendo em vista a simplificação da implementação de Hamming em *hardware*. Na Figura 6.11 é apresentada a inserção do código de Hamming em uma linha já codificada por código Reed-Solomon (NEUBERGER et al., 2003).

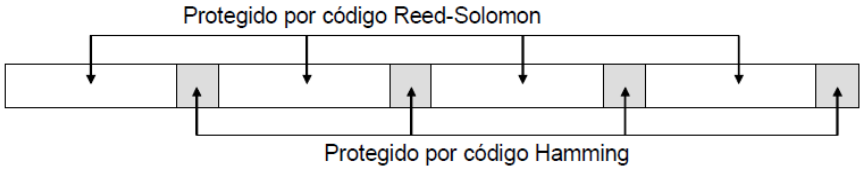

Figura 6.11: esquemático de uma palavra de memória protegida por Reed-Solomon e Hamming.

Como mostrado na Figura 6.12, o codificador recebe os dados a serem gravados na memória e os divide entre dados protegidos pelo código de Hamming, e dados protegidos pelo código de Reed-Solomon para então gravá-los na memória. Desta forma, quando o FT-OpenRISC 1200 precisar destes dados, o decodificador irá pegar todos os dados da memória fazer a checagem por Hamming e Reed-Solomon, retirar os dados de redundância e enviar os dados para que sejam processados pelo FT-OpenRISC 1200.

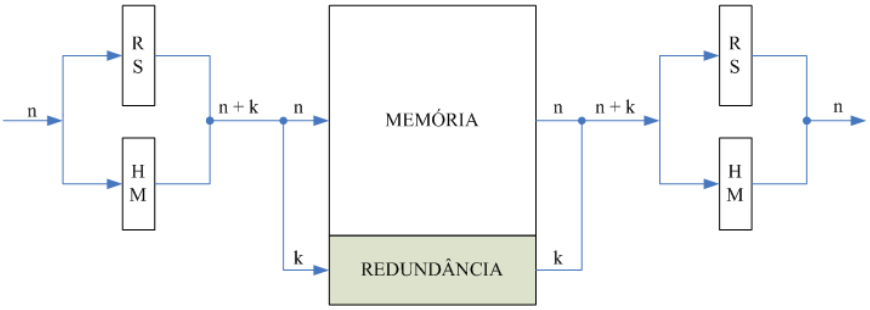

Figura 6.12: esquemático de uma memória tolerante a falhas, utilizando Hamming e Reed-Solomon.

Seguindo o pipeline de operação do OpenRISC 1200, conforme pode ser observado na Figura 6.13, é inserido o codificador no processo de gravação e leitura dos dados da memória *cache*. As operações para se ler e gravar as informações na *cache* de dados deste processador passam pelo processo de codificação de Hamming e Reed-Solomon do FT-OpenRISC 1200.

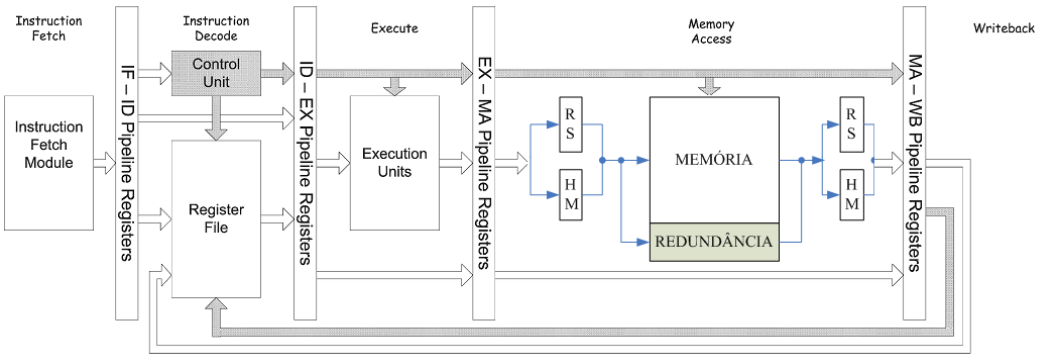

Figura 6.13: codificador utilizando Hamming e Reed-Solomon do FT-OpenRISC 1200.

No próximo Capítulo é analisado e discutido a eficiência das técnicas de tolerância a falhas aplicadas a este processador. Além disso, é apresentado os resultados da síntese deste processador em FPGA, com resultados de área ocupadas da frequência de operação em FPGA da Xilinx.

Capítulo 7

Resultados e Discussões

O objetivo neste Capítulo é apresentar os resultados da síntese do microprocessador FT-OpenRISC 1200, levando-se em conta os custos de área ocupada e de sua frequência de operação em FPGA, além de validar, por experimentos de injeção de falhas, as técnicas de tolerância a falhas aplicadas, em nível de projeto, no FT-OpenRISC 1200.

7.0.5 Resultados da síntese física em FPGA

Para realizar a síntese dos microprocessadores openRISC 1200 e FT-openRISC 1200 para a FPGA 4vfx12ff668-10 da família Virtex 4 da Xilinx, a ferramenta de *software* ISE da Xilinx é utilizada. Uma comparação, em termos dos recursos de elementos lógicos, dos microprocessadores OpenRISC 1200 e FT-OpenRISC 1200 para FPGA é apresentada na Tabela 7.1.

Tabela 7.1: comparação de área ocupada entre o OpenRISC 1200 e o FT-OpenRISC 1200

Configuração da FPGA	OpenRISC 1200	FT-OpenRISC 1200
Number of Slices	1342	2764
Number of Slice Flip Flops	683	1722
Number of 4 input LUTs	2419	5203

Após aplicar as técnicas de tolerância a falhas neste processador, de um modo geral, tem-se um aumento da ordem de $2,5$ vezes em relação ao tamanho original do OpenRISC 1200, como pode ser observado na Tabela 7.2.

Tabela 7.2: aumentos percentuais do FT-OpenRISC 1200 em relação ao OpenRISC 1200.

Configuração da FPGA	FT-OpenRISC 1200
Number of Slices	105,96%
Number of Slice Flip Flops	152,123%
Number of 4 input LUTs	115,089%

7.0.6 Análise da frequência de operação

Importante ainda notar que, de acordo com a Tabela 7.3 a frequência atingida pela ferramenta de síntese foi diferente, como já era de se esperar, pois foram realizadas alterações no *core* do openRISC 1200.

Tabela 7.3: frequência de operação do OpenRISC 1200 e do FT-OpenRISC 1200

Configuração do OpenRISC 1200	Freq de operação (MHz)
OpenRISC 1200(Original)	66.007
FT-OpenRISC 1200(Tolerante a Falhas)	50.378

Pela Tabela 7.3 pode-se notar que a diminuição na frequência de operação do OpenRISC 1200 para o FT-OpenRISC 1200 é de apenas 25%. Sabe-se que ao aplicar técnicas de tolerância a falhas em um sistema, existe um custo associado e, no caso deste microprocessador FT-OpenRISC 1200, o custo é computado de duas formas: a primeira com o aumento da área de utilização em FPGA e a segunda com a diminuição da frequência de operação deste microprocessador. Em contra partida tem-se um microprocessador que pode ser utilizado para aplicações espaciais.

7.0.7 Descrição e resultados dos testes de injeção de falhas em FPGA

Injeção de falhas é uma técnica atraente para avaliação de projetos devido a sua alta flexibilidade em termos de informação temporal e espacial. O processo envolve a

inserção de falhas em determinados alvos de um sistema em um tempo determinado no processo, e monitorar os resultados para definir seu comportamento em resposta a uma falha. Adicionalmente, a metodologia de testes por injeção de falhas tem tempo e custo de testes reduzidos comparado com testes sob radiação tradicionais, feitos em laboratórios de instrumentação nuclear. Além do mais, já foi mostrado que injetando falhas em uma plataforma programável, depois de sintetizada em uma FPGA, pode-se aumentar a velocidade do processo de testes em muitas ordens de magnitude.

Nos experimentos realizados, dois programas são utilizados, a multiplicação e a ordenação por bolhas (bubble sort), os quais são codificados em linguagem C e compilados com a ferramenta GNU GCC, ferramenta esta capaz de gerar executáveis para a arquitetura do microprocessador FT-openRISC 1200.

7.0.7.1 Injeção de falhas em *flip-flops*

A ferramenta de injeção de falhas insere diferentes modelos de falhas em pontos específicos do módulo central do microprocessador FT-OpenRISC 1200. O processo de inserção de falhas é realizado para ambos os programas em execução, separadamente no FT-OpenRISC 1200, e o mesmo se mostrou operante para todas as falhas inseridas.

7.0.7.2 Injeção de falhas na memória *cache*

No OpenRISC 1200 a memória *cache* é dividida em linhas, e cada linha tem capacidade de armazenar 128 *bits*, conforme é discutido nas seções anteriores. Para exemplificar esta divisão das linhas na *cache*, toma-se como base uma memória com capacidade para armazenar 512 *bits*, esta memória é então dividida em quatro linhas, sendo que cada uma destas são de 128 *bits* conforme apresentado na Figura 7.1.

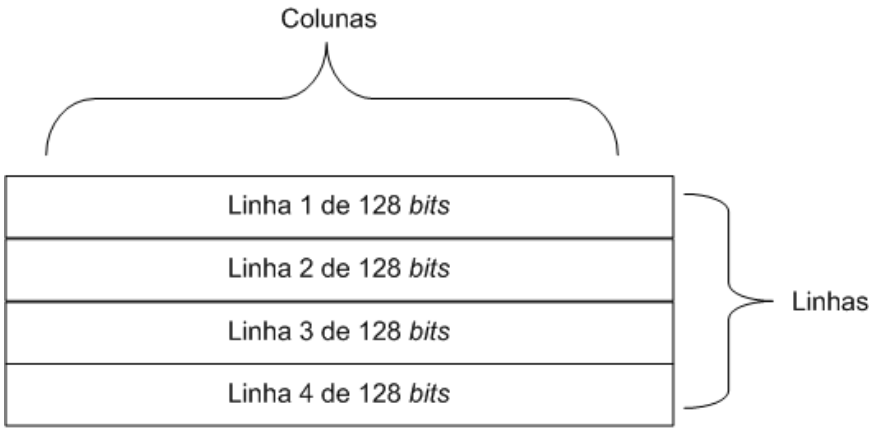

Figura 7.1: memória *cache* do OpenRISC 1200 dividida em linhas e colunas

Na proteção desta memória é utilizado o esquema apresentado na Figura 7.2, na qual pode-se constatar que a memória *cache* é dividida em quatro blocos, sendo que cada bloco tem capacidade para armazenar 32 *bits*, de um total de 128 *bits* por linha.

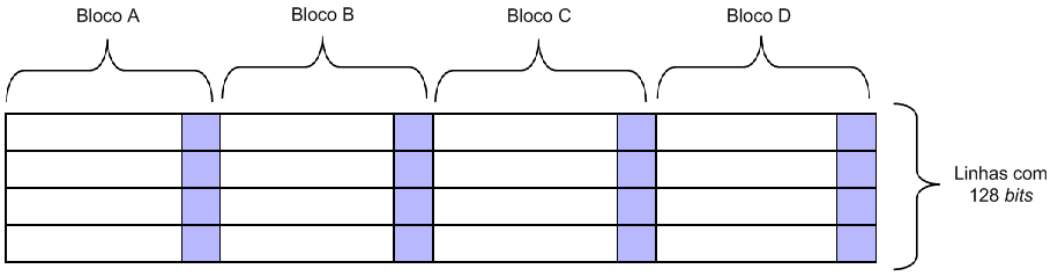

Figura 7.2: memória *cache* do OpenRISC 1200 dividida em blocos

Cada bloco da Figura 7.2 representa dados protegidos pela técnica de proteção de memória adotada neste trabalho. Neste caso, cada bloco é dividido em dois sub-blocos menores, um de 24 *bits* representado pela parte branca e, o outro de 8 *bits*, representado pela parte cinza deste bloco. A parte branca de cada bloco (24 *bits*) é protegida por um símbolo de Reed-Solomon e a parte cinza (8 *bits*), de cada bloco, é protegida por Hamming.

Na Figura 7.3 são apresentados 10 tipos diferentes de falhas que podem ocorrer em células de memórias. Estas falhas ocorrem quando uma partícula ionizada se choca com um transistor da memória ocasionando uma virada de *bit* (SEU), e são classificadas da seguinte forma:

1. falhas do tipo 1 e do tipo 2 ocorrem quando uma partícula ionizada se choca com uma célula de memória, ocasionando a virada de um *bit* (SEU);

2. falhas dos tipo 3, 4, 5 e 6 ocorrem quando uma partícula ionizada se choca com uma célula de memória, se divide devido ao choque e ocasiona duas viradas de *bits* em células vizinhas. Este tipo de falha, discutida no Capítulo 3, é conhecida como MBUs;

3. falhas dos tipo 7, 8, 9 e 10 ocorrem quando uma ou mais partículas ionizadas se chocam com células de memória ocasionando múltiplas viradas de *bits* em células vizinhas. Este tipo de falha, discutido no Capítulo 3, também é conhecida como MBUs;

Para a realização do experimento de injeção e falhas na memória *cache* do FT-OpenRISC 1200, foram injetados, nesta memória, os 10 tipos diferentes de falhas, conforme Figura 7.3. Para cada tipo de falha o código corretor de erros implementado apresenta uma sequência de correção diferente, confome pode ser verificado nas Tabelas 7.4, 7.5 e 7.6.

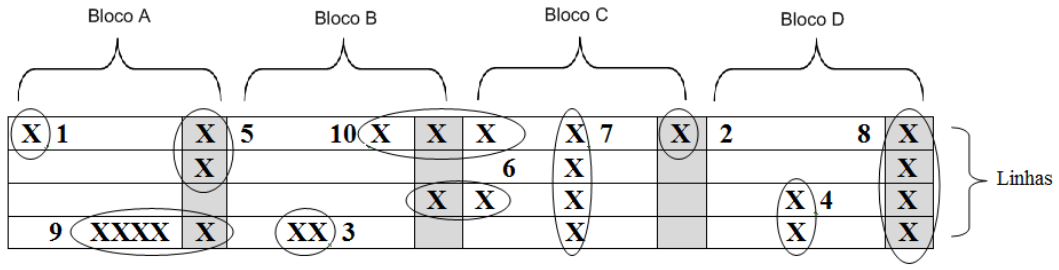

Figura 7.3: falhas simples, duplas e múltiplas em uma memória

Na Tabela 7.4 são apresentados dois tipos de erros simples que podem ocorrer na memória. Estes dois erros classificados como do tipo 1 e do tipo 2, podem ocorrer, em momentos distintos, em duas regiões da memória, conforme Figura 7.3. As falhas do tipo 1 ocorrem numa linha do bloco **A**, ou seja, uma virada de *bit* nos dados protegidos pelo símbolo de Reed-Solomon, já as Falhas do tipo 2 ocorrem numa linha do bloco **C**, ou seja, uma virada de *bit* nos dados protegidos por Hamming. Estes tipos de falhas são corrigidos internamente pelo FT-OpenRISC 1200 pois ambos os códigos (Reed-Solomon e Hamming) são capazes de corrigir erros simples.

Tabela 7.4: detecção e correção de falhas simples no FT-OpenRISC 1200

Tipo de falha	***Localização***	***Efeito do erro no FT-OpenRISC***
1	Símbolo Reed-Solomon	Corrigido erro simples em apenas um símbolo Reed-Solomon
2	*Bit* Hamming	Corrigido, erro simples em apenas um *bit* Hamming

Na Tabela 7.5 são apresentados quatro tipos de erros duplos que podem ocorrer na memória, e que são classificados como dos tipos 3, 4, 5 e 6, podendo ocorrer, em momentos distintos, em quatro regiões da memória, conforme Figura 7.3.

As falhas do tipo 3 ocorrem numa linha do bloco **B**, ou seja, duas viradas de *bit* nos dados protegidos pelo símbolo de Reed-Solomon. Este tipo de erro é corrigido pois o código Reed-Solomon empregado é capaz de corrigir erros duplos no mesmo símbolo da mesma linha.

As falhas do tipo 4 ocorrem em duas linhas diferentes do bloco **D**, ou seja, duas viradas de *bit* nos dados protegidos por símbolos diferentes de Reed-Solomon em linhas diferentes, este tipo de erro é o mesmo que o do tipo 1 ocorrendo duas vezes simultaneamente. Como se trata de falhas ocorridas em linhas diferentes de Reed-Solomon, a técnica utilizada corrige facilmente este tipo de erro.

As falhas do tipo 5 ocorrem em duas linhas diferentes do bloco **A**, ou seja, duas viradas de *bit* nos dados protegidos por Hamming em linhas diferentes, sendo este tipo de erro o mesmo que o do tipo 2 ocorrendo duas vezes simultaneamente. Como se trata de falhas ocorridas em linhas diferentes de Hamming a técnica utilizada corrige facilmente este tipo de erro.

As falhas do tipo 6 ocorrem na mesma linha, contudo em diferentes bloco **B** e **C**, ou seja, uma virada de *bit* nos dados protegidos por Hamming do bloco **B** e uma outra virada de *bit* nos dados protegidos por Reed-Solomon do bloco **C**. Este tipo de falha é uma combinação das falhas do tipo 1 com as do tipo 2. Neste caso, o FT-OpenRISC 1200 é capaz de corrigí-las.

Tabela 7.5: detecção e correção de falhas duplas no FT-OpenRISC 1200

Tipo de falha	***Localização***	***Efeito do erro no FT-OpenRISC***
3	Símbolo Reed-Solomon	Corrigido erro múltiplo em apenas um símbolo Reed-Solomon.
4	Símbolo Reed-Solomon adjacentes verticalmente	Corrigido erros simples em diferentes códigos Reed-Solomon.
5	*Bits* Hamming adjacentes verticalmente	Corrigido erros simples em diferentes códigos de Hamming.
6	*Bit* Hamming e símbolo Reed-Solomon adjacentes horizontalmente	Corrigido erro simples em símbolo Reed-Solomon e erro simples em *bit* Hamming.

Na Tabela 7.6 são apresentados quatro tipos de erros multiplos que podem ocorrer na memória. Estes quatros erros classificados como dos tipos 7, 8, 9 e 10, podem ocorrer, em momentos distintos, em quatro regiões da memória, conforme Figura 7.3.

As falhas do tipo 7 ocorrem em quatro linhas diferentes do bloco **C**, ou seja, quatro viradas de *bit* nos dados protegidos por símbolos diferentes de Reed-Solomon e em linhas diferentes. Este tipo de erro é o mesmo que o do tipo 1 ocorrendo quatro vezes simultaneamente. Como se trata de falhas ocorridas em linhas diferentes de Reed-Solomon, a técnica utilizada corrige facilmente este tipo de erro.

As falhas do tipo 8 ocorrem em quatro linhas diferentes do Bloco **D**, ou seja, quatro viradas de *bit* nos dados protegidos por Hamming em linhas diferentes, sendo este tipo de erro o mesmo que o do Tipo 2 ocorrendo quatro vezes simultaneamente. Como se trata de falhas ocorridas em linhas diferentes de Hamming a técnica utilizada corrige facilmente este tipo de erro.

As falhas do tipo 9 ocorrem na mesma linha e no mesmo bloco **A**, ou seja, quatro viradas de *bit* nos dados protegidos pelo mesmo símbolo de Reed-Solomon e uma virada de *bit* nos dados protegidos por Hamming. Como se trata de múltiplos erros nos dados protegidos pelo mesmo símbolo de Reed-Solomon numa mesma linha e ainda um único erro nos dados protegidos por Hamming, a técnica utilizada é capaz de corrigir todos estes 5 erros.

As falhas do tipo 10 ocorrem na mesma linha, contudo em diferentes blocos **B** e **C**, ou seja, uma virada de *bit* nos dados protegidos pelo símbolo de Reed-Solomon do bloco **B**, uma virada de *bit* nos dados protegidos por Hamming do bloco **B** e uma outra virada de *bit* nos dados protegidos por outro símbolo de Reed-Solomon do Bloco **C**. Este tipo de falha não pode ser corrigido pelo FT-OpenRISC 1200, pois o algoritmo de Reed-Solomon utilizado não suporta correção de erros em símbolos diferentes numa mesma linha, contudo, este tipo de falha é detectado e sinalizado pelo FT-OpenRISC 1200, através de um pino de controle/interrupção.

Quando o FT-OpenRISC 1200 detecta falhas do tipo 10, o mesmo entra em modo de falha seguro, ou seja, sinaliza através de um pino de controle o seu estado de falha. Neste caso, fica a cargo do projetista do sistema operacional / *firmware*, em execução no FT-OpenRISC 1200, implementar uma rotina de tratamento para essa interrupção, ou seja, ele deve definir o que deve ser feito caso essa falha venha a ocorrer.

Tabela 7.6: detecção e correção de múltiplas falhas no FT-OpenRISC 1200

Tipos de falhas	***Localização***	***Efeito do erro no FT-OpenRISC***
7	Símbolos Reed-Solomon verticais	Corrigidos erros simples em diferentes códigos Reed-Solomon.
8	*Bits* Hamming verticais	Corrigido erros simples em diferentes códigos de Hamming.
9	Símbolo Reed-Solomon e *bit* Hamming horizontal	Corrigido erro simples em símbolo Reed-Solomon e erro simples em *bit* Hamming.
10	Símbolo Reed-Solomon, *bit* Hamming e símbolo Reed-Solomon horizontal	Detectado mas não corrigido erro em dois diferentes símbolos Reed-Solomon na mesma linha.

7.0.8 Eficácia dos testes

Durante a fase de injeção de falhas, foram injetadas 2314 falhas na lógica sequencial e combinacional e 1000 falhas na mémoria *cache* de dados. Os resultados dos testes de injeção de falhas pôde constatar a eficiência da proposta. Os testes realizados demostraram que o FT-OpenRISC 1200 projetado utilizando técnicas de TMR e de códigos corretores de erros, além de mascarar as falhas na lógica sequencial e combinacional, provenientes de SEUs e SETs, também pode detectar e corrigir erros na memória *cache*.

Embora a injeção de falhas através de simulação e testes demonstrem a eficácia da proposta, os testes realizados não simulam com perfeição um ambiente submetido a elevados níveis de radiação. Neste caso, considera-se necessário avaliar o funcionamento do FT-OpenRISC 1200 quando este for submetido a um bombardeamento real de partículas. Estes testes permitirão avaliar o grau de tolerância a falhas por radiação do microprocessador, em um ambiente real.

Capítulo 8

Conclusões, Contribuições e Trabalhos Futuros

A probabilidade da ocorrência de falhas transientes em circuitos digitais, causadas por radiação vem aumentando consideravelmente. Neste sentido, este trabalho propõe a utilização e validação de técnicas de tolerância a falhas, em nível de projeto do microprocessador OpenRISC 1200. Estas técnicas garantem o correto funcionamento do microprocessador, mesmo na presença de radiação, para garantir o correto funcionamento sem a necessidade de utilizar FPGAs protegidas. Desta forma, é proposta uma arquitetura para proteger a lógica e a memória SRAM do OpenRISC 1200 contra Single Event Upset (SEU) e Single Event Trasient (SET).

Toda a arquitetura é descrita em HDL e prototipada em FPGAs da Xilinx. A injeção de falhas é realizada através da JTAGs na FPGA, permitindo assim a validação das técnicas de tolerância a falhas transientes e de proteção da memória de dados utilizadas no OpenRISC 1200. Além da validação das técnicas, realizou-se uma avaliação no desempenho do processador modificado em termos de tempo de execução e área ocupada.

Desta forma, falhas simples nos registros foram adicionadas aleatoriamente, e constatadas ineficazes para causar um erro no FT-openRISC 1200. Ou seja, em 100% das vezes que uma falha ocorre nos registros, esta falha é mascarada e o processamento computado corretamente. Contudo quando utilizamos o sistema de votador simples, ou seja, sem replicar, inserimos no sistema um ponto de falha único, e nos casos em que o SEU ocorre no votador o resultado é de 100% de falhas. Desta forma, para as partes críticas do processador faz-se necessário o uso de replicação do votador, tornando assim o sistema mais confiável.

Múltiplas falhas foram aleatoriamente injetadas em todas as células de memória para

avaliar a robustez do método. A experiência foi realizada numa plataforma de prototipação de circuitos programáveis (FPGA FX25 da família Virtex 4 da Xilinx). Os resultados mostram a eficiência do método proposto para todas as falhas simples e duplas e uma grande parte das falhas múltiplas. Todas as falhas duplas e uma grande combinação de falhas múltiplas foram corrigidas pelo método. O único tipo de falha múltipla que só foi detectada, mas não foi corrigida é quando a falha múltipla (três ou mais bits) afeta dois símbolos diferentes do código RS.

Para a técnicas de proteção de memória, é proposto a utilização de algoritmo de Hamming e Reed Solomon, ambos implementados em *hardware* (HDL) para aumentar a confiabilidade do processador OpenRISC 1200. A disposição destes códigos nas linhas de memória, ou seja Hamming nas interfaces de símbolos de Reed-Solomon são usados para potencializar a detecção e correção de erros. Desta forma, agregamos ao processador OpenRISC 1200 a capacidade de detectar e corrigir múltiplos erros na memória.

As principais contribuições, discutidas neste trabalho, incluem:

- uso de triplicação dos registradores;
- proteção da memória *cache* utilizando combinações de códigos de Hamming e Reed-Solomon;
- porte do OpenRISC 1200 para ser executado na FPGA Virtex 4 da Xilinx presente na placa ML403;
- mapeamento das partes principais do código fonte do OpenRISC 1200;
- software de TMR para o OpenRISC 1200.

No entanto, quando compara-se o FT-OpenRISC 1200 com outros processadores tolerantes a falhas (GAISLE, 2002; GAISLER, 1994; QUACH, 2000; CHECK; SLEGEL, 1999; COTA et al., 2001; MEINHARDT et al., 2009), evidencia-se a necessidade de incorporar outras técnicas de tolerância a falhas para torná-lo mais robusto. Neste sentido, podem ser desenvolvidos os seguintes trabalhos:

- a triplicação da árvore de *clocks*
- a proteção da memória externa utilizando combinações de códigos corretores de erros
- a proteção do controlador de memória externa com técnicas de EDAC

- submeter o FT-OpenRISC 1200 a testes de radiação em laboratórios de radiação (LIN - Laboratório de Instrumentação Nuclear)
- tripicar toda a lógica combinacional do FT-OpenRISC 1200

Além disso, em aplicações de tempo real, os processos devem ser atendidos dentro de um intervalo de tempo limite (deadline) e seus resultados devem ser livres de erros. Neste trabalho tratou-se dos problemas de como executar as operações sem falhas, não sendo analisadas as características temporais. Neste sentido, uma análise para aplicações de tempo real deve ser feita com o intuito de verificar o comportamento do FT-OpenRISC 1200 para estes sistemas. Desta forma os seguintes trabalhos podem ser feitos:

- portar o RT-Linux, sistema operacional de tempo real, para este processador e verificar as características de *deadline* de operação;
- testar o escalonador de tempo real, e mensurar se os tempos de execução estão sendo atendidos corretamentes.

Sugere-se ainda aprimorar o *software* de TMR tornando-o flexível, para que além de interpretar Verilog HDL, também possa inferir sobre códigos fontes codificados em outras linguagens de descrição de *hardware*, como por exemplo VHDL. Além disso, testes em outras plataformas de microprocessadores devem ser efetuados, no sentido de aprimorar o algoritmo do *software* de TMR, adotado neste trabalho no FT-OpenRISC 1200, para que seja possível sua utilização em outras plataformas.

Referências Bibliográficas

ALEXANDRES, D.; ANGHEL, L.; NICOLAIDIS, M. New methods for evaluating the impact of single event transients in vdsm ics. In: . Washington, DC, USA: IEEE Computer Society, 2002. p. 99 – 107. ISSN 1063-6722.

ALEXANDRESCU, D. et al. Simulating single event transients in vdsm ics for ground level radiation. *Journal of Electronic Testing*, Springer Netherlands, Norwell, MA, USA, v. 20, p. 413–421, 2004. ISSN 0923-8174. 10.1023/B:JETT.0000039608.48856.33. Disponível em: <http://dx.doi.org/10.1023/B:JETT.0000039608.48856.33>.

ALEXANDRESCU, D.; ANGHEL, L.; NICOLAIDIS, M. Simulating single event transients in vdsm ics for ground level radiation. *Journal of Electronic Testing*, Springer Netherlands, v. 20, p. 413–421, 2004. ISSN 0923-8174. 10.1023/B:JETT.0000039608.48856.33. Disponível em: <http://dx.doi.org/10.1023/B:JETT.0000039608.48856.33>.

ANDERSON, T.; LEE, P. A. *Fault Tolerance: Principles and Pratctice*. [S.l.]: Prentice-Hall, 1981.

ANGHEL, L.; ALEXANDRESCU, D.; NICOLAIDIS, M. Evaluation of a soft error tolerance technique based on time and/or space redundancy. In: *SBCCI '00: Proceedings of the 13th symposium on Integrated circuits and systems design*. Washington, DC, USA: IEEE Computer Society, 2000. p. 237. ISBN 0-7695-0843-X.

ANGHEL, L.; LEVEUGLE, R.; VANHAUWAERT, P. Evaluation of set and seu effects at multiple abstraction levels. In: . Washington, DC, USA: IEEE Computer Society, 2005. p. 309 – 312. ISSN 1530-1591.

ANGHEL, L.; NICOLAIDIS, M. Cost reduction and evaluation of a temporary faults detecting technique. In: . New York, NY, USA: ACM, 2000. p. 591 –598.

ARGYRIDES, C.; ZARANDI, H. R.; PRADHAN, D. K. Matrix codes: Multiple bit upsets tolerant method for sram memories. In: *DFT '07: Proceedings of the 22nd IEEE*

International Symposium on Defect and Fault-Tolerance in VLSI Systems. Washington, DC, USA: IEEE Computer Society, 2007. p. 340–348. ISBN 0-7695-2885-6.

BARTH, J. Applying computer simulation tools to radiation effects problems. In: *IEEE NUCLEAR SPACE RADIATION EFFECTS CONFERENCE, NSREC*. [S.l.]: IEEE Computer Society, 1997. p. 1–83.

BAUMA, R. C. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, v. 5, n. 3, p. 305 – 316, sep. 2005. ISSN 1530-4388.

BAUMAN, R. Soft errors in commercial semiconductor technology: Overview and scaling trends. In: *Proc. IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*. Washington, DC, USA: IEEE Computer Society, 2002. p. 121.01.1–121.01.14.

BAUMAN, R. C. Radiation-induced soft errors in advanced semiconductor technologies. In: *IEEE Transactions on Devices and Materials Reliability*. New York, USA: v.5, n.3,, 2005. p. 305–316.

BAUMANN, R. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *Device and Materials Reliability, IEEE Transactions on*, v. 1, n. 1, p. 17 –22, mar. 2001. ISSN 1530-4388.

BAUMANN, R. C. Soft errors in advanced semiconductor devices - part i: The three radiation sources. In: *IEEE Transactions on Device and Materials Reliability, v.1, n. 1*. [S.l.: s.n.], 2001.

BAZE M.; BUCHNER, S. Attenuation of single event induced pulses in cmos combinational logic. In: *IEEE Transactions on Nuclear Science*. [S.l.: s.n.], 1997.

BOLZANI, L. M. V. *Explorando uma Solução Híbrida: Hardware + Software para a Detecção de Falhas em Systems-on-Chip (SoCs)*. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio Grande do Sul - PUC, 2005.

BORKAR, S. esigning reliable system from unreliable components: The challenges of transistor variability and degradation. In: *IEEE MICRO, v.45, n.7*. [S.l.: s.n.], 2005.

BUCHNER, S. et al. Investigation of single-ion multiple-bit upsets in memories on board a space experiment. *Nuclear Science, IEEE Transactions on*, v. 47, n. 3, p. 705 –711, jun. 2000. ISSN 0018-9499.

CARMICHAEL, C. *Triple Module Redundancy Design Techniques for Virtex FPGAs*. Longmont, CO, USA, July 2006. Application note, v1.0.1.

CASTRO, H.; COELHO, A. A.; SILVEIRA, R. J. Fault-tolerance in fpga's through crc voting. In: *SBCCI '08: Proceedings of the 21st Annual Symposium on Integrated Circuits And System Design*. New York, NY, USA: ACM, 2008. p. 188–192. ISBN 978-1-60558-231-3.

CASTRO, H. de S. *Fault Tolerance Through Reconfigurability: Applications In Space Instrumentation*. Tese (Phd Thesis) — The University of Sussex, June 1992.

CHECK, M. A.; SLEGEL, T. J. Custom s/390 g5 and g6 microprocessors. *IBM J. RES. DEVELOP*, v. 43, n. 5/6, p. 671–680, SEPTEMBER/NOVEMBER 1999. Disponível em: <http://www.research.ibm.com/journal/rd/435/check.pdf>.

CHEN, L.; AVIZIENIS, A. N-version programming: A fault tolerance approach to reliability of software operation. In: *Proceedings of the 8th International Symposium on Fault-Tolerant Computing Systems (FTCS-8)*. Toulouse, France: Fault-Tolerant Computing Systems (FTCS-8), 1978. p. 3–9.

CORES, O. *Open Cores*. 2010. Disponível em: <http://www.opencores.org>.

COTA Érika et al. Synthesis of an 8051-like micro-controller tolerant to transient faults. *J. Electron. Test.*, Kluwer Academic Publishers, Norwell, MA, USA, v. 17, n. 2, p. 149–161, 2001. ISSN 0923-8174.

DAHLGREN, P.; LIDEN, P. A switch-level algorithm for simulation of transients in combinational logic. *Fault-Tolerant Computing, International Symposium on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 0207, 1995.

DODD, P.; MASSENGILL, L. Basic mechanisms and modeling of single-event upset in digital microelectronics. *Nuclear Science, IEEE Transactions on*, v. 50, n. 3, p. 583 – 602, jun. 2003. ISSN 0018-9499.

ELMENDORF, W. Fault tolerant programming. In: *Proceedings of the 2nd International Symposium on Fault-Tolerant Computing Systems (FTCS-2)*. Newton, MA, USA: Fault-Tolerant Computing Systems (FTCS-2), 1972. p. 79–83.

FLEETWOOD, D. M. *Radiation Effects And Soft Errors In Integrated Circuits And Electronic Devices (Selected Topics in Electronics and Systems)*. 1. ed. [S.l.]: World Scientific Publishing Company, 2004. ISBN 9812389407.

FLOYD, T. L. *Sistemas Digitais - Fundamentos e Aplicações*. Porto Alegre, BR: Bookman, 2007.

FUHRMAN, C.; CHUTANI, S.; NUSSBAUMER, H. Hardware/software fault tolerance with multiple task modular redundancy. In: . [S.l.: s.n.], 1995. p. 171 –177.

FUJIWARA E.T.; PRADHAN, D. K. Error control coding in computer. In: *IEEE Computer*. [S.l.]: IEEE, 1990. p. 63–72.

GADLAGE, M. et al. Comparison of heavy ion and proton induced combinatorial and sequential logic error rates in a deep submicron process. *Nuclear Science, IEEE Transactions on*, v. 52, n. 6, p. 2120 – 2124, dec. 2005. ISSN 0018-9499.

GADLAGE, M. e. a. Single event transient pulsewidths in digital microcircuits. In: *IEEE Transactions on Nuclear Science*. [S.l.: s.n.], 2004.

GAISLE, J. A portable and fault-tolerant microprocessor based on the sparc v8 architecture. In: *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2002. p. 409–415. ISBN 0-7695-1597-5.

GAISLER, J. Concurrent error-detection and modular fault-tolerance in a 32-bit processing core for embedded space flight applications. In: *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*. [S.l.: s.n.], 1994. p. 128–130. ISBN 0-8186-5520-8.

GAMA, M. A. *NÚCLEOS IP CORRETORES DE ERROS PARA PROTEÇÃO DE MEMÓRIA EM SoC*. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio Grande do Sul - PUC, 2008.

HABINC, S. *Functional Triple Modular Redundancy (FTMR)*. Göteborg, Sweden, December 2002.

HAMMING, R. W. Error detecting and error correcting codes. In: *The Bell System Technical Journal*. [S.l.]: The Bell System Technical Journal, 1950. p. 147–160.

HANMER, R. *Patterns for Fault Tolerant Software (Wiley Software Patterns Series)*. 1. ed. [S.l.]: Wiley, 2007. ISBN 0470319798.

HEFEZ ABRAMO. E VILLELA, M. L. *Códigos corretores de erros*. Rio de Janeiro, BR: IMPA, 2002. ISBN 85-244-0169-9.

HEIJMEN, T.; NIEUWLAND, A. Soft-error rate testing of deep-submicron integrated circuits. In: *ETS '06: Proceedings of the Eleventh IEEE European Test Symposium*. Washington, DC, USA: IEEE Computer Society, 2006. p. 247–252. ISBN 0-7695-2566-0.

HENTSCHKE, R. et al. Analyzing area and performance penalty of protecting different digital modules with hamming code and triple modular redundancy. In: *SBCCI '02: Proceedings of the 15th symposium on Integrated circuits and systems design*. Washington, DC, USA: IEEE Computer Society, 2002. p. 95. ISBN 0-7695-1807-9.

HOUGHTON, A. *Engineer's Error Coding Handbook*. 1st. ed. New York, NY, USA: Springer, 1996. ISBN 041279070X.

HUANG, T. yuan et al. Improving radiation hardness of eeprom/flash cell by n o annealing. *IEEE ELECTRON DEVICE LETTERS*, v. 19, n. 7, p. 256–258, July 1998.

IROM, F. e. a. Single-event upset in commercial silicon-on-insulator powerpc microprocessors. In: *IEEE INTERNATIONAL SILICON-ON-INSULATOR CONFERENCE*. [S.l.: s.n.], 2002. p. 203–204.

IYER R. K. E KALBARCZYK, Z. *Hardware and Software Error Detection. [Online]*. 2010. Disponível em: <http://www.crhc.uiuc.edu/ kalbar/MotorolaCourse/HW&SW_ErrorDetection.pdf.>.

JOHANSSON, K. et al. Neutron induced single-word multiple-bit upset in sram. *Nuclear Science, IEEE Transactions on*, v. 46, n. 6, p. 1427 –1433, dec. 1999. ISSN 0018-9499.

JOHNSTON, A. Scaling and technology issues for soft error rates. In: *roceedings of 4th Annual Research Conference on Reliability*. [S.l.: s.n.], 2000.

KARNIK T.; HAZUCHA, P. P. J. Characterization of soft errors caused by single event upsets in cmos processes. In: *IEEE Transaction on Dependable and Secure Computing*. [S.l.]: v.1, n.2, 2004. p. 128–143.

KASTENSMIDT, F. L.; CARRO, L.; REIS, R. *Fault-Tolerance Techniques for SRAM-based FPGAs*. 1. ed. www.springer.com: Springer, 2006.

KEANE, J. et al. On-chip reliability monitors for measuring circuit degradation. *Microelectronics Reliability*, n. 50, p. 1039–1053, September 2010.

KOREN, I.; KRISHNA, M. *Fault-Tolerant Systems*. 1. ed. [S.l.]: Morgan Kaufmann, 2007. ISBN 0120885255.

KWAI, D.-M.; PARHAMI, B. Fault-tolerant processor arrays using space and time redundancy. In: . [S.l.: s.n.], 1996. p. 303 –310.

LALA, P. K. (Ed.). *Fault Tolerance and Fault Testable Hardware Design*. Upper Saddle River, NJ, USA: Prentice-Hall, 1985. ISBN 0-13-057887-8.

LAPRIE, J. claude. Dependable computing and fault-tolerance: Concepts and terminology. In: *Proceedings of the IEEE*. [S.l.: s.n.], 1985. p. 2–11.

LIMA, F. et al. Injecting multiple upsets in a seu tolerant 8051 micro-controller. In: . Washington, DC, USA: IEEE Computer Society, 2002. p. 194.

LIN, S. *An Introduction to Error-Correcting Codes*. USA: Prentice-Hall, 1970.

MAHESHWARI, A.; KOREN, I.; BURLESON, N. Techniques for transient fault sensitivity analysis and reduction in vlsi circuits. In: . [S.l.: s.n.], 2003. p. 597 – 604. ISSN 1063-6722.

MAVIS, D.; EATON, P. Soft error rate mitigation techniques for modern microcircuits. In: *INTERNATIONAL RELIABILITY PHYSICS SYMPOSIUM*. [S.l.: s.n.], 2002. p. 216–225.

MCDONALD, P. et al. Destructive heavy ion see investigation of 3 igbt devices. In: . [S.l.: s.n.], 2000. p. 11 –15.

MCNULTY, P. et al. Threshold let for seu induced by low energy ions [in cmos memories]. *Nuclear Science, IEEE Transactions on*, v. 46, n. 6, p. 1370 –1377, dec. 1999. ISSN 0018-9499.

MEINHARDT, C. et al. Recovery scheme for hardening system on programmable chips. In: *Test Workshop, 2009. LATW '09. 10th Latin American*. [S.l.: s.n.], 2009. p. 1–6.

MESSENGER, G. C. Collection of charge on junction nodes from ion tracks. *Nuclear Science, IEEE Transactions on*, v. 29, n. 6, p. 2024 –2031, dec. 1982.

NEUBERGER, G. et al. A multiple bit upset tolerant sram memory. *ACM Trans. Des. Autom. Electron. Syst.*, ACM, New York, NY, USA, v. 8, n. 4, p. 577–590, 2003. ISSN 1084-4309.

NG, Y. H.; LOW, Y. H.; DEMIDENKO, S. Improving efficiency of ic burn-in testing. In: . [S.l.: s.n.], 2008. p. 1685 –1689. ISSN 1091-5281.

NICOLAIDI, M. Design for soft error mitigation. *Device and Materials Reliability, IEEE Transactions on*, v. 5, n. 3, p. 405 – 418, sep. 2005. ISSN 1530-4388.

NICOLAIDIS, M. Time redundancy based soft-error tolerance to rescue nanometer technologies. In: . Washington, DC, USA: IEEE Computer Society, 1999. p. 86 –94.

NIEUWLAND, A. K.; JASAREVIC, S.; JERIN, G. Combinational logic soft error analysis and protection. In: *IOLTS '06: Proceedings of the 12th IEEE International Symposium on On-Line Testing*. Washington, DC, USA: IEEE Computer Society, 2006. p. 99–104. ISBN 0-7695-2620-9.

NORMAND, E. Single-event effects in avionics. *Nuclear Science, IEEE Transactions on*, v. 43, n. 2, p. 461 –474, apr. 1996. ISSN 0018-9499.

O'BRYAN, M. et al. Current single event effects and radiation damage results for candidate spacecraft electronics. In: . Washington, DC, USA: IEEE Computer Society, 2002. p. 82 – 105.

O'BRYAN, M. e. a. Current single event effects and radiation damage results for candidate spacecraft electronics. In: *IEEE RADIATION EFFECTS DATA WORKSHOP*. [S.l.: s.n.], 2002. p. 82–105.

OMANA, M. et al. A model for transient fault propagation in combinatorial logic. In: . [S.l.: s.n.], 2003. p. 111 – 115.

OPENCORES. *OpenRISC 1200 IP Core Specification, Preliminary draft, Rev.0.7*. USA, 2001.

OR1K, O. *or1k*. 2009. Disponível em: <http://opencores.org/project,or1k>.

ORSRC, O. *The OpenRISC 1200 Source Code*. 2009. Disponível em: <http://opencores.org>.

PRADHAN, D. K. (Ed.). *Fault-tolerant computer system design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996. ISBN 0-13-057887-8.

PROAKIS, J. G. *Communications Systems Engineering, 2 Ed*. New Jersey, USA: Prentice Hall, 2002.

QUACH, N. High availability and reliability in the itanium processor. *IEEE Micro*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 20, n. 5, p. 61–69, 2000. ISSN 0272-1732.

REBAUDENGO, M.; REORDA, M.; VIOLANTE, M. An accurate analysis of the effects of soft errors in the instruction and data caches of a pipelined microprocessor. In: . Washington, DC, USA: IEEE Computer Society, 2003. p. 602 – 607. ISSN 1530-1591.

REBAUDENGO, M.; REORDA, M. S.; VIOLANTE, M. Analysis of seu effects in a pipelined processor. In: . Washington, DC, USA: IEEE Computer Society, 2002. p. 112 – 116.

REED, I. S.; SOLOMON, G. Polynomial codes over certain finite fieldserror detecting and error correcting codes. In: *SIAM Journal of Applied Mathematics*. Philadelphia, PA, USA: The Bell System Technical Journal, 1978. p. 3–9.

REED, R. Heavy ion and proton induced single event multiple upsets. In: *IEEE Nuclear and Space Radiation Effects Conference (NSREC)*. [S.l.: s.n.], 1997.

SCHWANK, J. et al. Estimation of heavy-ion let thresholds in advanced soi ic technologies from two-photon absorption laser measurements. *Nuclear Science, IEEE Transactions on*, v. 57, n. 4, p. 1827 –1834, aug. 2010. ISSN 0018-9499.

SHIVAKUMAR, P. et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In: . [S.l.: s.n.], 2002. p. 389–398.

SHOOMAN, M. L. *Reliability of Computer Systems and Networks*. 1. ed. Somerset, NJ, USA: Wiley, 2002.

SILVEIRA, R. J. N. da et al. Técnica de proteção de bytecodes para processador java em tecnologia cmos. In: *Anais do WSCAD SSC 2009: X Simpósio em Sistemas Computacionais*. São Paulo, SP, Brasil: SBC (Sociedade Brasileira de Computação), 2009. ISSN 21762074.

SUBRAMANIAN, G. T. *Control Caching: A fault-tolerant architecture for SEU mitigation in microprocessor control logic*. 2–3 p. Dissertação (Mestrado) — Iowa State University, Ames, Iowa, 2006.

VELAZCO, R.; REZGUI, S.; ECOFFET, R. Predicting error rate for microprocessor-based digital architectures through c.e.u. (code emulating upsets) injection. *Nuclear Science, IEEE Transactions on*, v. 47, n. 6, p. 2405 –2411, dec. 2000. ISSN 0018-9499.

VIOLANTE, M. Accurate single-event-transient analysis via zero-delay logic simulation. *Nuclear Science, IEEE Transactions on*, v. 50, n. 6, p. 2113 – 2118, dec. 2003. ISSN 0018-9499.

WANG, N. et al. Characterizing the effects of transient faults on a high-performance processor pipeline. In: . New York, NY, USA: ACM, 2004. p. 61 – 70.

WESTE, N.; HARRIS, D. M. *CMOS VLSI Design: A Circuits and Systems Perspective*. 4. ed. Upper Saddle River, NJ, USA: Addison Wesley, 2010.

WIRTH, G. I. et al. Modeling the sensitivity of cmos circuits to radiation induced single event transients. *Microelectronics Reliability*, v. 48, n. 1, p. 29 – 36, 2008. ISSN 0026-2714. Disponível em: <http://www.sciencedirect.com/science/article/B6V47-4N6FP41-4/2/a6fee70fa98d954504e6011d3b8e664c>.

WROBEL, F. et al. Simulation of nucleon-induced nuclear reactions in a simplified sram structure: scaling effects on seu and mbu cross sections. *Nuclear Science, IEEE Transactions on*, v. 48, n. 6, p. 1946 –1952, dec. 2001. ISSN 0018-9499.

XILINX. *TMRTool user guide*. 6.2.03. ed. Longmont, CO, USA, September 2004. TMRTool Software V6.2.03i.

Apêndices

Apêndice A

Códigos de Bloco

Nesta seção, são apresentados os conceitos básicos dos códigos de Bloco e em especial de uma subclasse - os códigos de bloco lineares (LIN, 1970). Os códigos de Bloco basicamente, processam a informação bloco por bloco, tratando cada bloco de *bits* de informação de forma independente em relação a outro bloco. Diz-se que um código é linear se duas palavras-código quaisquer do código puderem ser somadas em aritmética módulo 2 para produzir uma terceira palavra-código. A codificação da informação é realizada em duas etapas:

1. A sequência de informação é dividida em blocos de informação, com k dígitos sucessivos de informação;

2. O codificador transforma, então, cada bloco de informação em um bloco maior com n ($n > k$) dígitos binários (palavra-código), de acordo com uma determinada regra de codificação. Este novo bloco é denominado Palavra-Código.

Cada bloco de informação contém k dígitos binários. Portanto, existem 2^k blocos de informação distintos. Consequentemente, existem 2^k palavras-código correspondentes. Este conjunto de 2^k palavras-código é denominado Código de Bloco.

Um Código de Bloco Linear é formado pelo conjunto das 2^k palavras-código que constituem um subespaço do espaço vetorial V_n de todas as palavras-código.

Seja C um codificador que divide a sequência de informação em blocos de 3 dígitos e transforma cada mensagem em uma palavra-código de 6 dígitos, conforme demonstrado na Tabela A.1. Com $k = 3$, existem $2^3 = 8$ informações distintas possíveis. Para cada bloco de informação o codificador gerou uma palavra-código distinta, com 6 dígitos. Assim, a partir de cada uma das palavras-código geradas é possível obter-se a informação original.

Tabela A.1: exemplo de código de bloco (6,3).

Informação	Palavra-Código
000	000 000
001	001 101
010	010 011
011	011 110
100	100 110
101	101 011
110	110 101
111	111 000

A.1 Matriz Geradora

Todas as palavra-código (vetores) de um subespaço vetorial S de V_n podem ser obtidas através de uma combinação linear de um conjunto de palavra-código linearmente independentes. Assim, um código de bloco linear com 2^k vetores código (palavras-código) pode ser representado por um conjunto de k vetores código linearmente independentes. Estes vetores linearmente independentes podem ser organizados como linhas em uma matriz k x n:

$$G = \begin{bmatrix} V_1 \\ V_2 \\ . \\ . \\ V_k \end{bmatrix} = \begin{bmatrix} V_{11} & V_{12} & V_{13} & . & . & V_{1n} \\ V_{21} & V_{22} & V_{23} & . & . & V_{2n} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ V_{k1} & V_{k2} & V_{k3} & . & . & V_{kn} \end{bmatrix} \tag{A.1}$$

Esta matriz é denominada Matriz Geradora do código, pois combinações lineares entre suas linhas geram um código de bloco linear. A palavra-código correspondente a um bloco de informação $m = (m_1, m_2, ..., m_k)$ pode ser obtida por:

$$u = mG \tag{A.2}$$

$$u = (m_1, m_2, ..., m_k) \begin{bmatrix} V_1 \\ V_2 \\ . \\ . \\ V_k \end{bmatrix} \tag{A.3}$$

$$u = m_1 V_1 + m_2 V_2 + ... + m_k V_k \tag{A.4}$$

Conforme já observado, a palavra-código correspondente ao bloco de informação $(m_1, m_2, ..., m_k)$ é uma combinação linear das linhas da matriz geradora G. Este código linear é conhecido como um Código de Bloco (n, k), no qual um bloco de informação com k dígitos é codificado em uma palavra-código com n dígitos para ser armazenado numa memória. A razão $R = k/n$ é denominada Taxa do Código e representa a quantidade de redundância contida na palavra-código.

O código apresentado na Tabela (A.1) , no início desta seção, é um código (6,3) e sua matriz geradora está representada em (A.5).

$$G = \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{A.5}$$

A palavra-código correspondente a informação $m = 101$ é dada por (A.6) e (A.7):

$$u = (1, 0, 1) \begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} \tag{A.6}$$

$$u = 1(100110) + 0(010011) + 1(001101) = 101011 \tag{A.7}$$

Deve-se notar que, devido ao fato de o código de bloco linear ser completamente especificado por sua matriz geradora, o codificador não precisa armazenar todas as palavras código em sua memória. Assim, para o conjunto de 2^k palavras-código apenas k linhas da matriz geradora são armazenadas. Outro ponto importante é que, como a palavra-código atual depende apenas do bloco de informação atual, não há memória envolvida no processo de codificação. Portanto, o codificador pode ser implementado através de uma lógica combinacional.

A.2 Códigos Sistemáticos

É possível definir um processo de codificação em que os k dígitos iniciais de cada palavra-código sejam exatamente os mesmos k dígitos de cada bloco de informação, enquanto os $n-k$ dígitos restantes serão os dígitos de redundância. Um código deste tipo é chamado de Código Sistemático e sua estrutura está representada na Figura A.1.

Palavra-Código	
Bloco de informação	Dígitos de Redundância
k	*n-k*

Figura A.1: estrutura da palavra-código em um código sistemático

Um código linear sistemático (n, k) pode ser descrito por uma matriz geradora, com a seguinte estrutura:

$$G = \begin{bmatrix} 1000\ldots0 & p_{11} & p_{12} & \ldots & p_{1,n-k} \\ 0100\ldots0 & p_{21} & p_{22} & \ldots & p_{2,n-k} \\ 0010\ldots0 & p_{31} & p_{32} & \ldots & p_{3,n-k} \\ & & & \ldots & \\ 0000\ldots1 & p_{k1} & p_{k2} & \ldots & p_{k,n-k} \end{bmatrix} \tag{A.8}$$

onde $p_{ij} = 0$ ou 1.

A matriz identidade é uma matriz quadrada na qual todos os elementos da diagonal principal (elementos de índice a_{ii}, ou seja, a_{11}, a_{22}, etc.) são iguais a 1. Considerando-se que I_k seja a matriz identidade de ordem k e que P seja a matriz $kx(n-k)$ de p_ij, a matriz geradora de um código sistemático pode ser representada por:

$$G = [I_k P] \tag{A.9}$$

A partir das Equações A.2, A.3, A.4 e A.9, a palavra-código correspondente a um bloco de informação $m = (m_1, m_2, ..., m_k)$ é dada por:

$$
\begin{aligned}
u &= (u_1, u_2, u_3, \ldots, u_n) \\
u &= (m_1, m_2, m_3, \ldots, m_k)G \\
u = (m_1, m_2, m_3, \ldots, m_k) &\begin{bmatrix}
1000\ldots0 & p_{11} & p_{12} & \ldots & p_{1,n-k} \\
0100\ldots0 & p_{21} & p_{22} & \ldots & p_{2,n-k} \\
0010\ldots0 & p_{31} & p_{32} & \ldots & p_{3,n-k} \\
 & & & \ldots & \\
0000\ldots1 & p_{k1} & p_{k2} & \ldots & p_{k,n-k}
\end{bmatrix}
\end{aligned}
\tag{A.10}
$$

Pode-se observar, pela multiplicação das matrizes, que:

$$u_i = m_i \quad para \quad i = 1, 2, ..., k \tag{A.11}$$

$$u_{k+j} = p_{1j}m_1 + p_{2j}m_2 + ... + p_{kj}m_k \quad para \quad j = 1, 2, ..., n-k \tag{A.12}$$

Pelas equações A.11 e A.12 verifica-se facilmente que os primeiros k dígitos da palavra-código são exatamente os dígitos do bloco de informação, enquanto os últimos $n-k$ dígitos são funções lineares dos dígitos de informação. Estes $n-k$ últimos dígitos da palavra código são conhecidos como Dígitos de Verificação de Paridade do código.

Considerando-se mais uma vez o código apresentado na Tabela A.1, juntamente com sua matriz geradora, a palavra-código gerada para o bloco de informação (m_1, m_2, m_3) será:

$$
\begin{aligned}
u &= (u_1, u_2, u_3, \ldots, u_6) \\
u = (m_1, m_2, m_3) &\begin{bmatrix}
1 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 1
\end{bmatrix} \\
u &= (m_1, m_2, m_3, m_1 + m_3, m_1 + m_2, m_2 + m_3)
\end{aligned}
\tag{A.13}
$$

Então, $u_1 = m_1, u_2 = m_2, u_3 = m_3$. Os $n-k$ dígitos restantes (u_4, u_5, u_6) são funções lineares dos k dígitos de informação. Assim, para um código de bloco linear na forma sistemática, a complexidade de codificação fica ainda mais reduzida, pois o codificador tem que armazenar apenas a matriz $P_{k(n-k)}$ de p_{ij}, ao invés de armazenar todas as linhas da matriz geradora G.

A.3 Matriz de Verificação de Paridade

Para uma dada matriz $G_{k.n}$ existe uma matriz correspondente $H_{(n-k).n}$, tal que o espaço linha da matriz G é ortogonal a H, isto é, o produto interno de um vetor do espaço linha de G e uma linha de H é zero (PROAKIS, 2002). Então, um código de bloco linear gerado pela matriz G pode ser descrito de forma alternativa: u é uma palavra-código gerada por G se e somente se $uH^T = 0$. Isto significa que os vetores que pertencem ao espaço linha de G também pertencem ao espaço nulo de H. A matriz H é denominada Matriz de Verificação de Paridade. Se a matriz geradora de um código binário sistemático estiver na forma apresentada na equação A.14, sua matriz de verificação de paridade será:

$$H = \begin{bmatrix} p_{11} & p_{21} & \dots & p_{k1} & 1000\dots0 \\ p_{12} & p_{22} & \dots & p_{k2} & 0100\dots0 \\ & & \dots & & \\ p_{1,n-k} & p_{2,n-k} & \dots & p_{k,n-k} & 0000\dots1 \end{bmatrix} = [P^T I_{n-k}] \tag{A.14}$$

onde P^T é a matriz P transposta.

Conforme descrito anteriormente, o espaço linha da matriz G é o espaço nulo de H, e vice-versa. Então, um código de bloco linear pode ser unicamente especificado tanto por sua matriz G, quanto pela H.

A.4 Capacidade de Correção de Um Código Linear

Para as consideraçõess referentes à capacidade de correção de um codigo linear são necessárias algumas definições, apresentadas a seguir:

- O Peso de Hamming de uma palavra-código $v, \omega(v)$, é definido como o numero de elementos de v diferentes de zero. Se $v = (1001011001)$, por exemplo, $\omega(v) = 5$;
- A Distância de Hamming entre dois vetores u e v, $d(u, v)$, é definida como o número de bits nos quais eles diferem. Se $u = (1001011001)$ e $v = (1\mathbf{1}00\mathbf{0}010\mathbf{1}\mathbf{0})$, por exemplo, $d(u, v) = 5$;
- A Distância minima de um código de bloco linear, d_{min}, é definida como a menor distância de *Hamming* entre todos os possíveis pares de palavras-código.

Pela regra da adição módulo-2, verifica-se que:

$$d(u, v) = \omega(u + v) \tag{A.15}$$

Ou seja, a distância entre dois vetores u e v é igual ao peso de sua soma vetorial, $u+v$.

Seja $v = (v_1, v_2, ..., v_n)$ o vetor código (palavra-código) armazenado numa memória, que utiliza um código corretor de erros aleatórios. Devido à influência de interferências presentes no ambiente, o vetor $r = (r_1, r_2, ..., r_n)$ pode ser qualquer uma das 2^n combinações. A diferença entre r e v é chamada Padrão de Erros ou Vetor de Erros causados pelos distúrbios que afetam os dados armazenados na memória e é dado por:

$$\begin{aligned} e &= (e_1, e_2, \ldots, u_n) \\ e &= r + v \\ e &= (r_1, r_2, \ldots, r_n) + (v_1, v_2, \ldots, v_n) \\ e &= (r_1 + v_1, r_2 + v_2, \ldots, r_n + v_n) \end{aligned} \tag{A.16}$$

Quando $e_i = r_i + v_i = 1$, há um erro na posição i do vetor lido r.

A função do decodificador é a de identificar o vetor código v a partir do vetor recebido r. A decodificação de máxima verossimilhança (também denominada decodificação de máxima probabilidade, ou *maximum-likelihood decoding*) irá identificar v como sendo o vetor código mais próximo do vetor recebido r com relação à distância de Hamming, isto é, aquele para o qual $d(v, r)$ é mínimo.

Pode-se demonstrar que um código com distância mínima d_{min} pode corrigir $\left[\frac{d_{min}-1}{2}\right]$ erros (PROAKIS, 2002). Assim, a capacidade de correção deste código fica definida por:

$$t = \left[\frac{d_{min} - 1}{2}\right] \tag{A.17}$$

Retomando-se o código exemplo da Tabela 3.1, verifica-se que sua distância mínima é 3 e, portanto, sua capacidade de correção é de 1 erro. Para um vetor armazenado igual ao padrão (000000) pode-se observar um caso de decodificação incorreta considerando-se a ocorrência de dois erros, um na 3ª posição e outro na 6ª posição. O vetor lido será, então, igual ao padrão (001001). A distância de *Hamming* entre o vetor lido e o vetor código (001101), por exemplo, é igual a 1, enquanto que a distância entre o vetor recebido e o vetor originalmente armazenado é 2. Assim, o decodificador identifica incorretamente o vetor (001101) como sendo a palavra-código armazenada.

A.5 Síndrome

Seja u um vetor código armazenado numa memória. O decodificador obtém um vetor corrompido r, que é uma soma vetorial do vetor originalmente armazenado u com o vetor de erros e. O objetivo do decodificador é recuperar o vetor u a partir de r.

$$r = u + e \tag{A.18}$$

A Síndrome do vetor decodificado r é um vetor com $(n - k)$ componentes, utilizada no processo de detecção e correção de erros e definida pela seguinte equação:

$$s = rH^T \tag{A.19}$$

Pode-se observar que se o vetor decodificado for uma palavra-código ou, em outras palavras, se ele pertencer ao espaço nulo de H, a síndrome será igual a zero. Portanto, se o decodificador calcular a síndrome e o resultado for diferente de zero ele já terá detectado a presença de erros no vetor decodificado, conforme demonstrado no exemplo a seguir.

A matriz geradora do código apresentado na Tabela (A.1) é:

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{A.20}$$

De acordo com a equação A.14, a correspondente matriz de verificação de paridade é:

$$H = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{A.21}$$

A informação (111000) é uma palavra-código, conforme observado na Tabela (A.1). Então, sua síndrome será nula:

$$S = (111000) \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = (000) \tag{A.22}$$

A informação (111001), no entanto, não é uma palavra-código (não pertence ao espaço linha de G ou ao espaço nulo de H). A síndrome deste vetor não será nula:

$$S = (111001) \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = (001) \tag{A.23}$$

No caso de códigos sistemáticos, a síndrome equivale à soma módulo 2 dos dígitos de paridade lidos com os dígitos de paridade recalculados a partir dos dígitos de informação lidos. Isto pode ser facilmente observado, notando-se que a matriz é formada pela matriz P (parte da matriz geradora) e pela matriz identidade I_{n-k}.

Apêndice B

HDL do Processador OpenRISC 1200

A primeira análise que deve ser feita, quando se utiliza *softcore processor*, é a da divisão do código fonte HDL. Neste caso, deve-se observar, no projeto, todos os relacionamentos entre os componetes descrito em HDL, a fim de mapeá-los, facilitando assim possíveis alterações no código fonte.

No caso do OpenRISC 1200 este mapeamento é feito de forma manual, pois, as referências e documentações sobre este processador não estão de forma condensada. Neste sentido, essa análise se deu através do *forum* da comunidade *opencores* e do manual de sua arquitetura. Infelizmente este manual de arquitetura não segue o desenvolvimento corrente do processador, ou seja, a documentação disponível data de 2004 e o código fonte do openRISC 1200 é atualizado quase que diariamente.

Além disso, o OpenRISC 1200 não possui uma codificação comentada, ou seja, os arquivos de códigos fonte praticamente só possuem a codificação da linguagem Verilog e nenhum comentário dos blocos de código.

Outro aspecto que dificulta o uso deste processador, está relacionado ao fato deste ser utilizado em ASIC, ou seja, o HDL deste processador é voltado para ser prototipado em silício. Para a prototipação em FPGA, ajustes no código fonte devem ser feitos visando a FPGA específica que deseja-se utilizar.

Neste trabalho, o primeiro passo para usar o processador OpenRISC 1200 é o de ajustar o HDL para que seja sintetizável na FPAG da XILINX. O outro passo é o de fazer uma descrição em alto nível do código fonte do processador OpenRISC 1200 para consultas, ou seja, mapear as partes principais do código fonte do OpenRISC 1200 para facilitar e agilizar o processo de modificação deste processador, tendo em vista a escassez

de informações técnicas sobre este.

Para simular e posteriormente realizar a síntese física em FPGA do OpenRISC 1200 é necessário ter acesso ao código fonte descrito em Verilog. O código fonte deste microprocessador está disponível para *download* em (OR1K, 2009).

Dos arquivos que existem disponíveis há alguns que merecem especial atenção, como é o caso do *top-level* do CPU/DSP *core*, "or1200_cpu.v", o *top-level* do sistema que inclui o módulo anterior, "or1200_top.v"e os restantes dos periféricos apresentados na Figura B.1. Nesta lista de arquivos, há ainda o arquivo "or1200_defines.v"que contém as macros que controlam o comportamento do sistema e as características que estão implementadas.

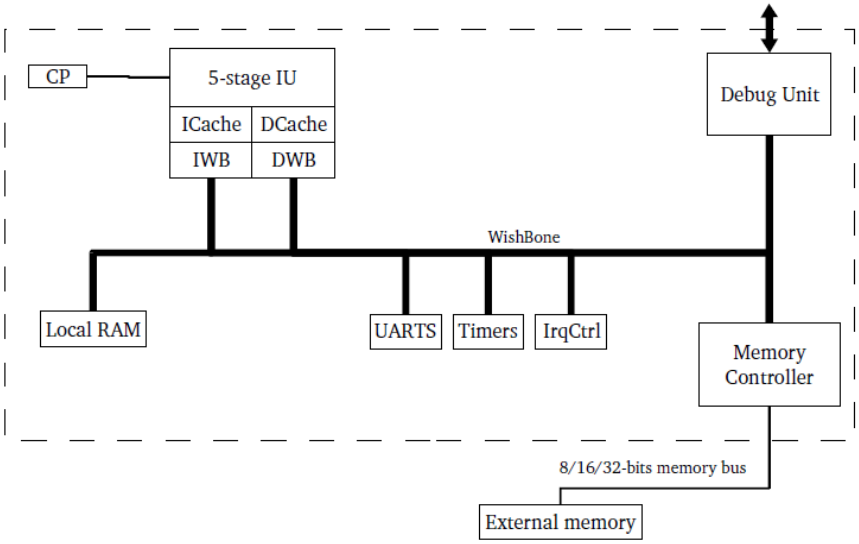

Figura B.1: periféricos conectados ao OpenRISC 1200.

O arquivo "OR1200_defines.v"contém todas as definições que devem ser configuradas para que o sistemas esteja de acordo com as funcionalidades pretendidas. Neste arquivo, de definição "or1200_defines.v", é possível escolher, por exemplo, se a memória *cache* de dados está ou não implementada e, caso esteja, qual o seu tamanho e como realizar a configuração de registros que são responsáveis pelo comportamento desta. Isto é feito através das macros que estão ativadas ou desativadas.

As macros de maior importância, para este trabalho, estão mapeadas na Tabela B.1 e B.2. Nestas tabelas é apresento qual o procedimento indicado, para ter um sistema que responda às necessidades das aplicações a qual se pretende desenvolver. Estas macros são de fato aquelas que permitem a presença dos blocos que constituem o *top-level* do OpenRISC 1200.

Tabela B.1: macros e respectiva interferência no sistema.

Macro	***Configuração do OpenRISC***
OR1200_NO_DC	Se estiver definida faz com que a *cache* de dados não seja implementada.
OR1200_NO_IC	Se estiver definida faz com que a *cache* de instruções não seja implementada.
OR1200_NO_DMMU	Se estiver definida, a MMU para dados não está presente no sistema.
OR1200_NO_IMMU	Se estiver definida, a MMU para instruções não está presente no sistema.
OR1200_IC_1W_4KB	Define o tamanho da *cache* de instruções para 4KB.
OR1200_DC_1W_4KB	Define o tamanho da *cache* de dados para 4KB.
OR1200_IC_1W_8KB	Define o tamanho da *cache* de instruções para 8KB.
OR1200_DC_1W_8KB	Define o tamanho da *cache* de dados para 8KB.
OR1200_SB_IMPLEMENTED	Define se o *Store buffer* está ou não presente no sistema.
OR1200_QMEM_IMPLEMENTED	Define se existe ou não memória interna no OpenRISC 1200

A configuração das *caches* é feita também recorrendo a estas macros que tratam das *tags* sempre presentes nestas unidades. O tratamento da memória virtual, quer para dados quer para instruções, também tem a sua configuração presente neste arquivo. Para as restantes unidades presentes no OpenRISC 1200, existem outros registros associados que devem ser configurados e mapeados no código fonte deste, contudo, as definições relativas aos registros especiais e para os *bits* que os compõem, estão presentes neste arquivo.

Tabela B.2: macros e respectiva interferência no sistema

Macro	***Configuração do OpenRISC***
OR1200_ASIC	Distingue se o suporte de desenvolvimento é um ASIC ou FPGA.
OR1200_BIST	Define se a memória incorporada tem autoteste incorporado
OR1200_PM_IMPLEMENTED	Define se a unidade de tratamento de energia está presente no sistema.
OR1200_DU_IMPLEMENTED	Define se a unidade de *debug* está implementada no OpenRISC 1200.
OR1200_PIC_IMPLEMENTED	Permite implementar ou não a unidade que controla as interrupções.
OR1200_PIC_INTS	Define qual o número de interrupções pode ser suportado no sistema
OR1200_TT_IMPLEMENTED	Define se o *tick timer* está ou não presente no sistema
OR1200_MULT_IMPLEMENTED	Define se a multiplicação está presente no OpenRISC 1200
OR1200_MAC_IMPLEMENTED	Define se existe a unidade MAC no Sistema

B.1 CPU *Core top-level*

Nesta seção é feita uma caracterização para o RTL (*Register Transfer Level*) do processador, através dos módulos que o constituem, bem como pela funcionalidade apresentada por cada um destes.

Na Tabela B.3 B.4 estão os módulos presentes na estrutura do *core* do OpenRISC 1200, fazendo-se a correspondência ao arquivo onde estes se encontram definidos e a funcionalidade desempenhada no *core*.

Desta forma consegue-se então caracterizar o *core* do OpenRISC 1200 de acordo com as associações das macros e módulos, o que facilita a compreenção das várias unidades que estão presentes, assim como a estrutura de hierarquia (*top-down*) deste processador.

Tabela B.3: módulos presentes no CPU Core, correspondência com arquivo e respectiva funcionalidade.

Macro	***Configuração do OpenRISC***	***Função Implementada na CPU***
OR1200_ALU	or1200_alu.v	Unidade Aritmética e lógica presente no CPU.
OR1200_GENPC	or1200_genpc.v	Bloco associado ao *program counter*.
OR1200_IF	or1200_if.v	Bloco associado ao *fetch* das instruções. Program counter e interface com a *cache* de instruções.
OR1200_CTRL	or1200_ctrl.v	Usado para efectuar a decodificação das instruções e possui ainda lógica de controle.
OR1200_WBMUX	or1200_wbmux.v	CPU com pipeline de estágio de Escrita de Retorno (*write-back stage*).
OR1200_RF	or1200_rf.v	Bloco que instancia as memórias usados para a transferência de dados entre registros e CPU.

Tabela B.4: continuação dos módulos presentes no CPU Core, correspondência com arquivo e respectiva funcionalidade

Macro	***Configuração do OpenRISC***	***Função Implementada na CPU***
OR1200_MULT_MAC	or1200_mult_mac.v	Unidade que está associada à ALU, implementa a multiplicação e a operação de MAC.
OR1200_LSU	or1200_lsu.v	Interface entre o CPU e a *cache* de dados.
OR1200_FREEZE	or1200_freeze.v	Trata de todas as paragens e freezes que ocorrem na CPU.
OR1200_EXCEPT	or1200_except.v	Módulo que trata de todo as excepções que possam ocorrer no interior da CPU.
OR1200_OPERANDMUXES	or1200_operandmuxes.v	Multiplexador para dois operandos de leitura do arquivo de registro.
OR1200_CFGR	or1200_cfgr.v	Modulo que realiza a configuração dos diversos registros (VR, UPR).
OR1200_RFRAM_GENERIC	Or1200_rfram_generic.v	Memória genérica usada no registrador de arquivos para guardar os GPRS.

B.2 *Top-level* do OpenRISC 1200

Apartir da caracterização do *core* da CPU, é essencial construir um arquivo *top-level* que monte a CPU com as restantes unidades que permitem fazer um interface com um sistema externo.

Como foi dito, na seção introdutória, o *top-level* do OpenRISC 1200 pode ser configurado mediante as necessidades de um determinado sistema. No entanto, deve-se ter acesso

aos outros arquivos da estrutura do sistema com as características de maior relevância. A forma mais simples de representar essa informação é apresentar a hierarquia e suas respectivas informações sobre os módulos, qual o arquivo correspondente e qual a função desempenhada no sistema. Na Tabela B.5 e B.6 este mapeamento é apresentado.

Tabela B.5: arquivos e funcionalidade correspondentes para um determinado módulo do OpenRISC 1200.

Módulo	***Arquivo***	***Função***
OR1200_iwb_biu	OR1200_iwb_biu.v	Interface *wishbone* para as instruções no *core*.
OR1200_wb_biu	OR1200_wb_biu.v	Interface *wishbone* para dados.
OR1200_immu_top	OR1200_immu_top.v	Unidade de controle de memória virtual para instruções.
OR1200_dmmu_top	or1200_ctrl.v	Unidade de controle de memória virtual para dados.
OR1200_IC_TOP	OR1200_IC_TOP.v	*Cache* do *core* para instruções.
OR1200_DC_TOP	OR1200_DC_TOP.v	*Cache* do *core* para dados.
OR1200_RF	or1200_rf.v	Bloco que instancia as memórias usados para a transferência de dados entre registros e CPU.

Tabela B.6: continuação de arquivos e funcionalidade correspondentes para um determinado módulo do OpenRISC 1200.

Módulo	***Arquivo***	***Função***
OR1200_CPU	or1200_mult_mac.v	CPU/MAC que existe no OpenRISC1200.
OR1200_SB	or1200_sb.v	Módulo para *buffer* de armazenamento (*Store buffer*).
OR1200_DU	OR1200_DU.v	Unidade que permite o *debug* dos programas em tempo real.
OR1200_PIC	OR1200_PIC.v	Módulo que atende as interrupções que venham a ocorrer no interior do CPU ou proveniente do exterior.
OR1200_TT	OR1200_TT.v	*Tick-timer* do sistema, é usado para gerar interrupções periódicas.
OR1200_PM	OR1200_PM.v	Módulo responsável pelo controle dinâmico de potência.
OR1200_QMEM_TOP	OR1200_QMEM_TOP.v	Memória interna que pode ser embarcada no *core* e que tem a vantagem de ter acessos mais rápidos.

Livros Grátis

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

Milhares de Livros para Download:

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

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