

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

Ricardo Jardel Nunes da Silveira

FT-JOP: UM PROCESSADOR JAVA PARA APLICAÇÕES
DE TEMPO REAL EM SISTEMAS EMBARCADOS
TOLERANTES A FALHAS

FORTALEZA - CEARÁ
AGOSTO - 2010

Ricardo Jardel Nunes da Silveira

FT-JOP: UM PROCESSADOR JAVA PARA APLICAÇÕES
DE TEMPO REAL EM SISTEMAS EMBARCADOS
TOLERANTES A FALHAS

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. Helano de Sousa Castro
(Orientador)

FORTALEZA - CEARÁ

2010

S591f

Silveira, Ricardo Jardel Nunes da
FT-JOP: um processador Java para aplicações de tempo real em sistemas embarcados tolerantes a falhas / Ricardo Jardel Nunes da Silveira, 2010.
87 f.; il. enc.

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. Circuito integrado 2. Código de hamming 3. Algoritmo CRC
I. Castro, Helano de Sousa (orient.). II. Universidade Federal do Ceará – Programa de Pós-Graduação em Engenharia de Teleinformática. III. Título.

CDD 621.38

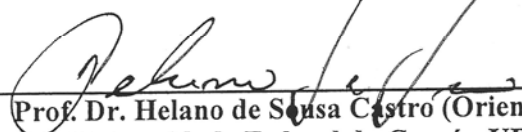
RICARDO JARDEL NUNES DA SILVEIRA

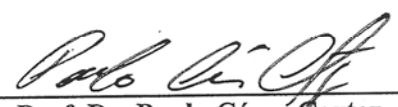
**FT-JOP: UM PROCESSADOR JAVA PARA APLICAÇÕES DE TEMPO REAL EM SISTEMAS
EMBARCADOS TOLERANTES A FALHAS**

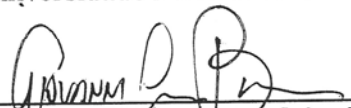
Dissertação submetida à Coordenação do Programa de Pós-Graduação em Engenharia de Teleinformática, da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Engenharia de Teleinformática.
Área de concentração Sinais e Sistemas.


Aprovada em 27/08/2010.

BANCA EXAMINADORA


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


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


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


Prof. Dr. Elmar Uwe Kurt Melcher
Universidade Federal de Campina Grande - UFCG

Ao Super-Ícaro.

Sumário

Lista de Figuras	viii
Lista de Tabelas	ix
Lista de Siglas	x
Resumo	xiii
Abstract	xv
Agradecimentos	xvii
1 Introdução	1
1.1 Motivação	1
1.2 Objetivos	3
1.3 Resumo das Contribuições	3
1.4 Organização	4
2 Sistemas Embarcados de Tempo Real	5
2.1 Introdução	5
2.2 Determinismo e análise temporal	7
2.3 Hardware de tempo real	8
2.4 Escalonamento de processos em sistemas de tempo real	9
2.4.1 Ciclo de execução	9
2.4.2 Prioridade fixa / periódicos	9
2.4.2.1 Problema da inversão de prioridade e solução	10
2.4.3 Algoritmo do <i>deadline</i> mais próximo	11
2.4.4 Escalonamento de tarefas aperiódicas	11
2.5 Gerenciamento de memória	12
2.6 Tolerância a falhas	12
2.6.1 Cálculo de confiabilidade	13
2.6.2 Detecção e correção de erros em uma sequência de dados	14
2.6.2.1 Checagem por redundância cíclica	14
2.6.2.2 Algoritmo de Hamming	14
2.6.2.3 Outros algoritmos de correção de erros	15
2.7 Java para sistemas de tempo real	16
2.7.1 Sistemas atuais	16
2.7.2 Vantagens da linguagem Java	16
2.7.3 Desvantagens da linguagem Java	16

2.7.4	Especificação Java para sistemas de tempo real	17
2.7.5	Especificação Java para sistemas críticos de tempo real	18
2.8	Conclusão	19
3	Projeto de Circuitos Integrados Lógicos Digitais	20
3.1	Introdução	20
3.2	Estado da Arte e Desafios do <i>Deep Sub Micron</i>	22
3.3	Requisitos de Projeto	23
3.4	Projeto e Processo de Fabricação	24
3.5	Fluxos de Projeto	25
3.5.1	<i>Standard Cells</i>	25
3.5.2	<i>Full Custom</i>	28
3.6	Reuso de módulos	29
4	Processador JOP	31
4.1	Implementação da JVM no JOP	31
4.2	Entrada e saída	32
4.3	Interrupções	33
4.4	Requisitos de tempo real	33
4.4.1	Análise de WCET no JOP	33
4.4.2	Garantia de funcionamento	34
4.5	Arquitetura do JOP	34
4.6	Cache de instruções de tempo previsível	34
4.7	Montador de aplicações	35
4.8	Documentação e portabilidade	36
5	JOP Tolerante a Falhas	37
5.1	Ocorrência de evento SEU na memória <i>cache</i> interna ao JOP	38
5.2	Técnica de Proteção de Instruções (TPI)	39
5.2.1	Fluxo dos <i>Bytecodes</i> no JOP x Fluxo dos <i>Bytecodes</i> no FT-JOP	39
5.2.2	Codificador de Hamming	40
5.2.3	Bloco extra de memória	41
5.2.3.1	Decodificador de Hamming	41
5.3	Ocorrência de evento SEU na memória externa ao JOP	41
5.4	Técnica de proteção de métodos	42
5.4.1	Software para adicionar CRC aos métodos	42
5.4.2	Verificador da integridade dos métodos	44
5.4.3	Percepção da falha em nível sistêmico	45
5.5	Aplicabilidade das técnicas de TPI e TPM	45
5.6	Síntese do JOP em Silício	46
5.6.1	Sistema de verificação funcional	46
5.6.2	Substituição das BRAMs	46
5.6.3	Inicialização da <i>Stack</i> RAM	47
5.6.4	Adição de uma JTAG para gravação da Flash e RAM externas	47
5.6.5	Adicionar registradores de configuração dos periféricos	47
5.6.6	Revisão do código RTL	48
5.6.7	Conclusão	48

6	Resultados	49
6.1	Resultados dos Testes de injeção de falhas em simulação	49
6.2	Resultados da síntese física em FPGA	49
6.3	Descrição e Resultados dos Testes de injeção de falhas em FPGA	50
6.4	Discussão dos Resultados	51
6.4.1	Aumento de área de silício do JOP	51
6.4.2	Eficácia dos testes	51
6.4.3	Temporização	51
7	Conclusões e Trabalhos Futuros	52
	Referências Bibliográficas	54
	Anexo 1 - Publicações Relacionadas a Esta Dissertação	60

Lista de Figuras

2.1	implementação em camadas de um sistema de tempo real.	7
2.2	ciclo executivo, adaptado de (TIMESYS, 2002).	9
2.3	inversão de prioridades, adaptado de (TIMESYS, 2002).	11
2.4	sequência de falha, erro e defeito, adaptado de (CASTRO, 1992).	12
2.5	fases de execução de uma aplicação com modelo da especificação SCJ, adaptado de (SCHOEBERL et al., 2007).	19
3.1	consumo x desempenho das tecnologias para implementação de sistemas computacionais.	21
3.2	níveis de projeto de circuitos integrados, adaptado de (RABAEY; POLLICE; WEST, 2003).	22
3.3	fluxo de desenvolvimento de um circuito lógico integrado digital, baseado em <i>standard cells</i>	26
3.4	fluxo de desenvolvimento de um circuito integrado (lógico digital ou analógico) em nível de transistores (<i>full custom</i>).	29
4.1	diagrama de blocos do JOP, adaptado de (SCHOEBERL, 2009).	32
4.2	fluxo de montagem de aplicativo JOP, adaptado de (SCHOEBERL, 2009).	36
5.1	<i>floorplan</i> do JOP original, adaptado de (RAMOS et al., 2010).	38
5.2	Configurações do JOP - (A) JOP original (B) JOP Original e TPI (C) JOP Original, TPI e TPM.	40

Lista de Tabelas

5.1	componentes de memória utilizados.	41
5.2	instruções utilizadas no método <code>main</code> do programa <code>HelloWorld.java</code>	43
6.1	comparação entre o FT-JOP e o JOP original.	50

Lista de Siglas

ABS	Anti-lock Breaking Systems
API	Application Program Interface
ASIC	Application-Specific Integrated Circuit
BCH	Bose-Chaudhuri-Hocquenghem
BICMOS	Bipolar Junction Transistors and CMOS
BVM	Brazil-IP Verification Methodology
CDK	Cadence Design Kit
CI	Circuito Integrado
CISC	Complex Instruction Set Computer
CLDC	Connected Limited Device Configuration
CMOS	Complementary Metal Oxide Semiconductor
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
CRC	Cyclical Redundancy Check
DMA	Direct Memory Access
DRC	Design Rule Check
DVD	Digital Video Disc
ECC	Error-Correcting Code
EDF	Early Deadline First
ESD	Electrostatic Discharge
FEC	Forward Error Correction
FPGA	Field Programable Gate Array
FPS	Fixed Priority Scheduling
FT-JOP	Fault Tolerant Java Optimized Processor

GDSII	Gerber Data Stream Information Interchange II
GPL	Gnu Public License
HDL	Hardware Description Language
IP	Intellectual Property
ITRS	The International Technology Roadmap for Semiconductors
J2ME	Java 2 Micro Edition
JIT	Just In Time
JOP	Java Optimized Processor
JSR	Java Specification Request
JVM	Java Virtual Machine
LESC	Laboratório de Engenharia de Sistemas de Computação
LVS	Layout Versus Schematics
MPW	Multi Project Wafer
NDA	Non-Disclosure Agreement
OVM	Open Verification Methodology
PCELL	Parameterized Cell
RAM	Random Access Memory
RAMB	Random Access Memory Block
RHBD	Radiation Hardening By Design
RISC	Reduced Instruction Set Computer
RMA	Rate Monotonic Analysis
ROM	Read Only Memory
RTAPP	Real-Time Application
RTHW	Real-Time Hardware
RTL	Register Transfer Level
RTOS	Real-Time Operating System
RTS	Real-Time System
RTSJ	Real-Time Specification for Java
SCJ	Safety Critical Java
SDF	Standard Delay Format
SECS	Single Error Correction Single Error Detection

SEU	Single Event Upset
SOI	Silicon on Insulator
SOS	Silicon on Sapphire
SPICE	Simulated Program with Integrated Circuits Emphasis
SRAM	Static Random Access Memory
TCL	Tool Control Language
TMR	Triple Modular Redundancy
TPI	Técnica de Proteção de Instruções
TPM	Técnica de Proteção de Métodos
USART	Universal Synchronous Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit
WCET	Worst Case Execution Time

Resumo

Sistemas embarcados de tempo real têm severas restrições de garantia de funcionamento, tais como: confiabilidade, requisitos temporais, consumo, peso e volume total ocupado. Considerando o projeto de circuitos integrados, para atingir o nível de confiabilidade requerido por esses sistemas, podem-se usar processos de fabricação de circuitos integrados para aplicações específicas, como por exemplo a tecnologia SOI/SOS (*Silicon on Insulator/Silicon on Sapphire*), que é inerentemente resistente a radiação e utilizada principalmente em aplicações militares e aeroespaciais. Outra abordagem consiste na aplicação de técnicas de garantia de funcionamento em nível de projeto do circuito integrado. Além disso, no *software* e no nível sistêmico (ou seja, na integração dos componentes eletrônicos em uma placa de circuito impresso) também é possível aplicar técnicas para aumentar a confiabilidade destes sistemas. Os circuitos integrados manufaturados utilizando processos específicos têm alto custo devido ao pequeno volume de produção destes. Pelo mesmo motivo (por serem específicos), os mesmos estão em geral, três gerações atrasados em relação ao estado da arte. Portanto, é preferível fazer o projeto de um circuito integrado tolerante a falhas e poder manufaturá-lo usando um processo que esteja no estado da arte. Do ponto de vista do desenvolvimento de *software*, os recursos fornecidos pela linguagem de programação utilizada para programar tais sistemas podem ter implicação no nível de confiabilidade dos mesmos. O uso de linguagens com um alto nível de abstração, como Java, pode diminuir a ocorrência de erros de programação, reduzindo assim o número de falhas de projeto, introduzidas durante o desenvolvimento. O *soft ip core* JOP (*Java Optimized Processor*) para FPGAs (*Field Programmable Gate Array*) é uma implementação otimizada da máquina virtual Java, em nível de *hardware*, para aplicações de tempo real. Neste trabalho, são propostas técnicas de *hardware* para o *soft ip core* JOP, que detectam e corrigem erros ocorridos na região de código da memória

SRAM (*Static Random Access Memory*). As técnicas propostas aumentam a confiabilidade do sistema e mantêm as características de tempo real do *soft ip core* JOP. O *soft ip core* JOP modificado foi implementado em uma FPGA Virtex 4 e as características de frequência de operação e número de portas lógicas são comparadas com o *soft ip core* original do JOP.

Palavras-chave: tolerância a falhas, processador Java, sistemas embarcados de tempo real, CMOS, circuito integrado.

Abstract

Real-time embedded systems have severe dependability restrictions such as: time, consumption, weight and total taken volume requirements. Considering the design of integrated circuits (ICs), in order to reach the reliability level required by these systems, one may use specific integrated circuits manufacturing processes, such as SOI/SOS technology (Silicon on Insulator/Silicon on Sapphire) that is inherently resistant to radiation and primarily used in military and aerospace applications. Another approach involves applying dependability techniques in the integrated circuit design phase. Furthermore, at software and system level (i.e. in the integration of electronic components in a printed circuit board) it is also possible to apply techniques to increase the reliability of these systems. The integrated circuits manufactured through specific processes have high cost due to their small production volume. Besides those specific-process ICs are in general three generations late in relation to the state of the art. Therefore, it is preferable to make the design of a fault tolerant integrated circuit and be able to manufacture it by a state of the art process. On the other hand, from the point of view of software development, the resources provided by the programming language used to program such systems may influence the system reliability level. The use of high abstraction level languages like Java, may diminish the occurrence of programming errors and decrease the number of non-predicted faults inserted during the development phase. The JOP (Java Optimized Processor) soft ip core for FPGAs (Field Programmable Gate Array) is an optimized implementation of the Java virtual machine, in the scope of hardware, for real-time applications. In this work we propose hardware design techniques for JOP soft ip core, which detect and correct errors in SRAM (Static Random Access Memory) memory code area. The proposed techniques increase system reliability and leave the features of real-time of JOP soft ip core untouched. The modified soft ip core JOP was implemented on a Virtex 4 FPGA and both the operating frequency and number of logic gates are compared to

the original soft ip core JOP.

Keywords: fault tolerance, Java processor, real-time embedded systems, CMOS, integrated circuit.

Agradecimentos

Dedico meus sinceros agradecimentos para:

- Deus pelo pão nosso de cada dia;
- o meu orientador Helano de Sousa Castro, pela orientação e incentivo;
- aos alunos de graduação David Viana e Pedro Lázaro pelo empenho e dedicação a este trabalho;
- o meu amigo Jilseph pelo empenho e dedicação a este trabalho;
- os meus Pais Diassis e Liduina pelo amor incondicional;
- a minha esposa Valdiana Moraes pelo amor e carinho;
- o meu filho Ícaro por todos os sorrisos que me revigoraram;
- o meu irmão Jarbas Aryel e família (Ismênia) pelo incentivo;
- a minha irmã Gardênia e família (Valmir, Amanda e Rebeca) pelo amor dedicado;
- o meu Amigo e companheiro de mestrado Alexandre Coelho pelas revisões e valiosas sugestões;
- o meu Amigo Vanilson Leite pelas revisões e valiosas sugestões;
- o meu Professor Paulo Cesár Cortez pelo constante incentivo;
- o meu Professor Mário Fiallos, por me iniciar no mundo da pesquisa;
- os companheiros de trabalho do LESC pelas revisões e sugestões;
- o LESC (Laboratório de Engenharia de Sistemas de Computação) por apoiar integralmente a realização das disciplinas;
- Fundação Cearense de Apoio ao Desenvolvimento Científico e tecnológico (FUN-CAP) 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;
- e
- o Programa Brazil IP e seus organizadores pelo apoio ao LESC.

Capítulo 1

Introdução

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 susceptíveis a falhas causadas pela colisão de partículas altamente energizadas com as estruturas nanométricas de silício presentes nos circuitos integrados modernos.

1.1 Motivação

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 a uma inversão permanente de um *bit*. Embora existam processos de fabricação de circuitos integrados especializados em produção de circuitos integrados tolerantes a 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 a radiação em nível de projeto do circuito integrado (ou RHBD - *Radiation Hardening By Design*), independentemente do processo de fabricação, pois, isso reduz os custos do sistema e permite utilizar os mais modernos processos de fabricação existentes.

Devido ao tamanho reduzido e a alta frequência de operação dos circuitos eletrônicos digitais modernos, os mesmos estão cada vez mais susceptíveis a ruído. Por isso, problemas antes somente encontrados em sistemas submetidos a radiações em nível espacial (missões espaciais), hoje são enfrentados em sistemas operando em nível terrestre. Portanto, percebe-se cada vez mais a necessidade de utilização de técnicas de tolerância a

falhas não somente em sistemas embarcados em missões espaciais, mas também nos sistemas terrestres. Dentre alguns exemplos dessas aplicações terrestres podem-se citar a indústria automobilística, bancária, e várias outras aplicações em que os requisitos temporais e de alta disponibilidade são prioritários para o correto funcionamento do sistema.

A linguagem de programação C é atualmente a mais utilizada para desenvolvimento de *software* para sistemas embarcados, tanto para o sistema operacional quanto para a aplicação. Isto pode ser facilmente demonstrado por uma análise dos compiladores comerciais disponíveis para os processadores modernos de sistemas embarcados cujos sistemas operacionais devem suportar aspectos específicos destes sistemas.

Usualmente, o sistema operacional é responsável por funções de suporte a tempo real, gerenciamento de memória e comunicação inter-processos. Tais recursos são disponibilizados para a aplicação por meio de chamadas de sistema (*System Calls*) e de uma API (*Application Program Interface*) (TANENBAUM, 2008).

O uso de uma linguagem de alto nível de abstração traz benefícios, do ponto de vista do desenvolvimento do sistema, tais como diminuir a probabilidade de erros de codificação e a redução do tempo de desenvolvimento de um sistema (BURNS; WELLINGS, 2009). Java é uma linguagem de alto nível atualmente muito utilizada e com extenso suporte para o desenvolvimento de sistemas computacionais. Além do alto nível de abstração, esta linguagem possui em seu núcleo (a Máquina Virtual Java) recursos comumente implementados em nível de sistema operacional, tais como comunicação inter-processo e escalonamento de tarefas.

Em uma implementação tradicional de sistemas embarcados de tempo real, baseada em um processador de uso geral e um sistema operacional de tempo real, essas vantagens (provenientes pelo uso de Java) podem ter um custo elevado em termos de recursos computacionais, que é incompatível com as severas restrições de recursos computacionais em sistemas embarcados. Esta incompatibilidade pode ser resolvida pelo uso de um processador específico para a linguagem Java, como proposto por Schoeberl (SCHOEBERL, 2008) e vários outros (BECK; CAIRO, 2006; HALFHILL, 2000; KREUZINGER et al., 2003; PUFFITSCH; SCHOEBERL, 2007). No entanto, em nenhum destes trabalhos, é discutido a garantia de funcionamento (*dependability*) do processador. Quando comparado com os demais processadores Java, o JOP (*Java Optimized Processor*) se diferencia, por exemplo, em relação a características de tempo real, porém também sem haver preocupação com requisitos de garantia de funcionamento (SCHOEBERL, 2003; SCHOEBERL et al., 2010). Estas são as principais razões para a escolha do processador JOP como base para este trabalho.

Neste trabalho, são propostas técnicas de tolerância a falhas para proteger a memórias SRAM de código interna do JOP contra SEUs. Além disso, propõe-se realizar a compilação do JOP modificado por tais técnicas em uma FPGA.

1.2 Objetivos

O presente trabalho tem por objetivo geral dotar de garantia de funcionamento o processador Java de tempo real JOP e prototipá-lo em FPGA. No decorrer do desenvolvimento deste trabalho, os seguintes objetivos específicos foram perseguidos:

- avaliar as técnicas de tolerância a falhas propostas para melhorar o nível de confiabilidade do JOP;
- validar as técnicas de tolerância a falhas implementadas em nível lógico;
- implementar em FPGA o JOP com as técnicas de tolerância a falhas validadas.

1.3 Resumo das Contribuições

Esta dissertação agrega as seguintes contribuições para o desenvolvimento de processadores Java de tempo real, mais especificamente no quesito tolerância a falhas:

1. concepção, implementação e avaliação de uma **técnica de proteção de instruções** contra SEUs (*Single Event Upsets*);
2. concepção, implementação e avaliação de uma **técnica de proteção de métodos** contra SEUs;
3. uma análise das modificações necessárias para a implementação do processador JOP em tecnologia CMOS (*Complementary Metal Oxide Semiconductor*); e
4. implementação em FPGA do FT-JOP (*Fault Tolerant Java Optimized Processor*).

A primeira contribuição foi publicada em um artigo completo no congresso WSCAD-SSC 2009 (Simpósio em Sistemas Computacionais) enquanto as outras contribuições serão submetidas para um periódico (SILVEIRA et al., 2009).

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 características e requisitos de tempo e de garantia de funcionamento para sistemas de tempo real.

No Capítulo 3 são apresentados um fluxograma, uma metodologia e algumas ferramentas para projeto de circuitos integrados lógicos digitais.

O *soft ip core* de tempo real JOP, sobre o qual foram validadas as técnicas de tolerância a falhas propostas neste trabalho, é abordado no Capítulo 4.

No Capítulo 5 tratam-se as técnicas de tolerância a falhas para a *cache* do JOP e as modificações necessárias de serem feitas no JOP em nível lógico para implementá-lo em silício.

Os resultados das simulações das técnicas de tolerância a falhas para a memória de código, assim como os testes realizados utilizando FPGA são apresentados no Capítulo 6.

As conclusões deste trabalho e os trabalhos futuros são apresentados no Capítulo 7.

Capítulo 2

Sistemas Embarcados de Tempo Real

Um sistema embarcado (*embedded system*) é um sistema computacional de propósito especial projetado para realizar uma ou mais funções dedicadas (GANSSE; BARR, 2003) e, frequentemente, possui restrições de volume, peso, consumo e de execução em tempo real. As restrições de tempo real se dividem em requisitos temporais e de garantia de funcionamento. Essas características são facilmente identificadas nos vários exemplos de sistemas embarcados, tais como telefones celulares, sistemas de freios ABS (*Anti-lock Breaking Systems*), computadores de bordo de aviões, dentre outros.

2.1 Introdução

Com o objetivo de introduzir alguns conceitos básicos de sistemas de tempo real, utilizam-se alguns exemplos do cotidiano. Suponha que você está vendo um filme em seu *laptop* de última geração. Ainda que raramente, é possível perceber pequenas interrupções momentâneas na execução do filme.

Em outro momento, você pode estar vendo um filme em seu aparelho reproduzidor de DVDs (*Digital Video Discs*). Para este simples aparelho, que tem desempenho computacional e custo muito inferior ao seu *laptop*, as mesmas interrupções momentâneas não são percebidas. Isso se deve ao fato de que o seu *laptop* não estar equipado com um sistema operacional de tempo real, tampouco com um *hardware* de tempo real, ou, colocando de outra forma, seu *laptop* não é um sistema de tempo real. Para esta aplicação, do ponto de vista computacional, o desempenho médio do seu *laptop* é muito superior ao do seu aparelho de DVD, mas este último é um sistema de tempo real do tipo *soft*. Além disso, o *laptop* não executa apenas essa tarefa, ao passo que o DVD, por ser um sistema dedicado, o faz. Neste sentido, este sistema atende aos requisitos de executar uma tarefa no tempo

requerido. Isto constitui o que se chama de previsibilidade.

Em outra situação, suponha que se deseja projetar um sistema computacional que irá controlar os *flaps* das asas de um avião durante o pouso. Para testes de laboratório, executa-se o *software* em seu *laptop* para controlar os *flaps* através de um dispositivo com interface USB (*Universal Serial Bus*). Novamente pode-se perceber, assim como na situação anterior, que apesar dos *flaps* operarem bem na maior parte do tempo, em algumas situações, ainda que seu *software* comande os *flaps* corretamente, estes não obedecem os comandos no tempo correto, demonstrando algumas paradas não previsíveis. Durante os testes, pode-se perceber que, algumas vezes, seu *laptop* travou, e que nessa situação, os *flaps* continuaram a se mover continuamente na última velocidade comandada pelo computador, antes do mesmo travar. Neste caso, identifica-se a segunda característica principal de um sistema de tempo real: confiabilidade.

Ainda que as interrupções do filme possam incomodar, certamente continuar-se-à a ver filmes em *laptops*. Por outro lado, mesmo que o sistema computacional embarcado em um avião possua alto custo, em relação ao custo de um *laptop*, certamente este último não será usado para a computação de bordo de um avião. Isso porque o sistema embarcado no avião é um exemplo de sistema de tempo real do tipo *hard*, o qual envolve riscos de vidas humanas. É nesse cenário que surge a necessidade dos sistemas computacionais de tempo real, os quais, além de serem projetados para terem um pequeno tempo médio de execução, estão também preocupados com questões como *deadlines* e confiabilidade.

Para se projetar um sistema de tempo real, todas as partes que o compõem devem ser de tempo real: *hardware* (RTHW - *Real Time Hardware*), sistema operacional (RTOS - *Real Time Operating System*) e aplicação (RTApp - *Real Time Application*). A Figura 2.1 mostra duas implementações bastante comuns de sistemas de tempo real. A primeira é baseada em uma aplicação de tempo real (RTApp), a qual implementa os requisitos funcionais do sistema, um sistema operacional de tempo real (neste exemplo, o RT Linux), o qual faz o gerenciamento dos recursos e o escalonamento de tarefas, e um *hardware* apropriado para tempo real (simbolizado aqui como RTHW). Na segunda implementação, não existe RTOS e a própria aplicação (RTApp - *Real Time Application*) implementa o objetivo propriamente dito do sistema, além de promover o gerenciamento dos recursos. Este último é também conhecido como ciclo executivo, pois, as tarefas são listadas sequencialmente em um laço.

Algumas das técnicas de projeto que os engenheiros usam para atender a estes requisitos de tempo e confiabilidade são redundância, paralelismo real, algoritmos de escalonamento de tempo real e simplificação do sistema (BURNS; WELLINGS, 2009). Obviamente,

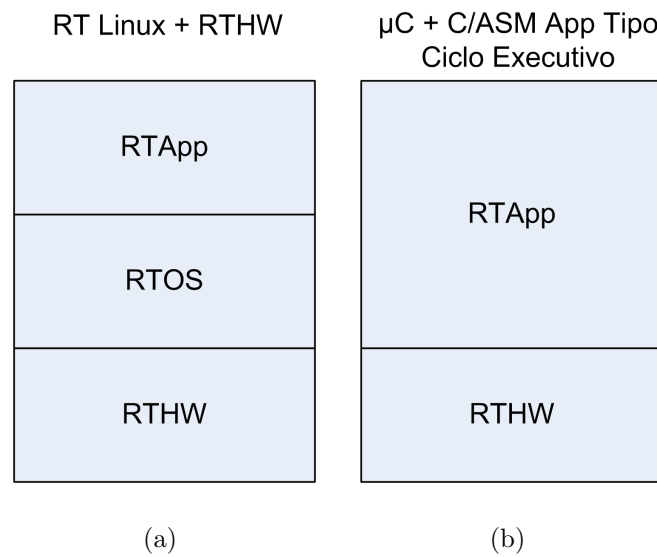


Figura 2.1: implementação em camadas de um sistema de tempo real.

isso pode envolver uma penalização associada ao tempo de desenvolvimento, custo final e desempenho médio do sistema. Portanto, estas técnicas de projeto devem ser aplicadas com parcimônia e especificidade para cada sistema.

2.2 Determinismo e análise temporal

Os sistemas computacionais modernos usualmente têm recursos disponíveis, tais como: Acesso Direto à Memória (DMA), *cache* de memória, *pipeline*, *branch prediction*, gerenciador de memória, escalonador de processos, funções de entrada/saída e comunicação inter-processos. Em um sistema comum (que não é de tempo real), estes recursos dão ao usuário do sistema, ou ao ambiente que o circunda, a impressão de que várias tarefas estão acontecendo em paralelo e prezam por um desempenho médio. Entretanto, um sistema de tempo real precisa restringir o não-determinismo encontrado em sistemas concorrentes comuns. Neste sentido, procura-se simplificar o sistema de forma a facilitar a análise do mesmo. No entanto, é importante entender que algumas penalidades podem ocorrer por simplificar bastante o sistema. Primeiro, a análise pode ser extremamente pessimista e pode não reproduzir a real situação ocorrida na maior parte do tempo. Logo, pode-se gerar um outro problema clássico de sistemas de tempo real, a introdução de *jitter*, que é a diferença entre os tempos de execução no melhor caso e no pior caso. A segunda penalidade, a rigidez imposta ao sistema, significando que uma pequena modificação no sistema como, por exemplo, inserir um novo processo, pode acarretar uma completa re-análise temporal e mesmo em outras modificações no sistema.

Em sistemas de tempo real as tarefas são executadas em tempo determinado, visando a operação normal destes sistemas. Assim, instantaneamente, cada tarefa está associada a um marco no tempo, ou seja, a um *deadline*. A partir deste, se o sistema não responder ao estímulo da entrada, este pode sofrer uma transição para um estado indesejável. Em sistemas de tempo real do tipo *hard*, um *deadline* perdido pode levar a uma falha catastrófica. Já em sistemas de tempo real do tipo *soft*, a perda de um *deadline* pode levar o sistema a funcionar fora de sua especificação. A garantia de que *deadlines* serão cumpridos baseia-se na análise temporal da execução do sistema no pior caso. Nesta análise, calcula-se o tempo exato de execução do pior caso ou WCET (*Worst Case Execution Time*) (SCHOEBERL et al., 2010). Este cálculo é realizado baseado no modelo temporal do *hardware*, no algoritmo de escalonamento do sistema operacional e na aplicação, podendo ser realizado manualmente, mas preferencialmente com o auxílio de uma ferramenta de *software*.

Para se obter o WCET de forma analítica, é necessário se obter o comportamento temporal do processador em questão. No entanto, o modelo temporal detalhado dos processadores modernos é não trivial devido às características, tais como *caches*, *pipelines* e *branch prediction*. Estas características ajudam a reduzir o tempo médio de execução, mas podem ser difíceis de prever seus impactos no WCET. Além disso, muitas informações necessárias podem ser proprietárias e difíceis de serem obtidas, mesmo sobre NDA (*Non-Disclosure Agreement*).

2.3 Hardware de tempo real

O hardware de um processador moderno contém estruturas paralelas que o auxiliam em suas tarefas, tais como DMA (*Direct Memory Access*), *cache* de memória, *pipeline* e *branch prediction*. Estas estruturas introduzem maior complexidade em seu modelo temporal.

A *cache* de memória de um processador é um recurso que aumenta o desempenho médio de um sistema computacional, mas por outro lado, introduz uma grande imprevisibilidade de tempo de execução de uma tarefa. Isto porquê o algoritmo de *cache* pode acertar ou errar quais são as próximas instruções ou dados a serem utilizados. Para solucionar este problema, pode-se desabilitar a *cache* ou fazer uso de um processador mais simples, sem este recurso. Em um primeiro instante, esta abordagem pode parecer como uma fuga ao problema, por falta de competência para se fazer a análise complexa do sistema com o uso de *cache*. No entanto, essa abordagem pode ser justificada pela seguinte máxima na

comunidade de projetistas desses sistemas: “simplicidade contribui para confiabilidade”. Neste caso, ou seja, em sistemas de tempo real do tipo *hard*, deseja-se a solução mais segura, ainda que menos elegante.

Além da complexidade do modelo temporal, de uma maneira geral, esses modelos ou mesmo as informações necessárias para montá-los, não estão facilmente disponíveis, devendo o interessado entrar em acordo legal (NDA - *Non Disclosure Agreement*) com o fornecedor do circuito integrado para tentar obtê-los.

2.4 Escalonamento de processos em sistemas de tempo real

2.4.1 Ciclo de execução

Para o problema de escalonamento de processos, a solução simplificada é o uso de execução cíclica, contínua e repetida de uma sequência de tarefas. Cada tarefa ocupa uma pequena parte do tempo como está mostrado na Figura 2.2, chamada de *minor frame* (ou *frame* secundário) e a soma de todas as tarefas ocupa um tempo conhecido por *major frame* ou *frame* principal (BURNS; WELLINGS, 2009).

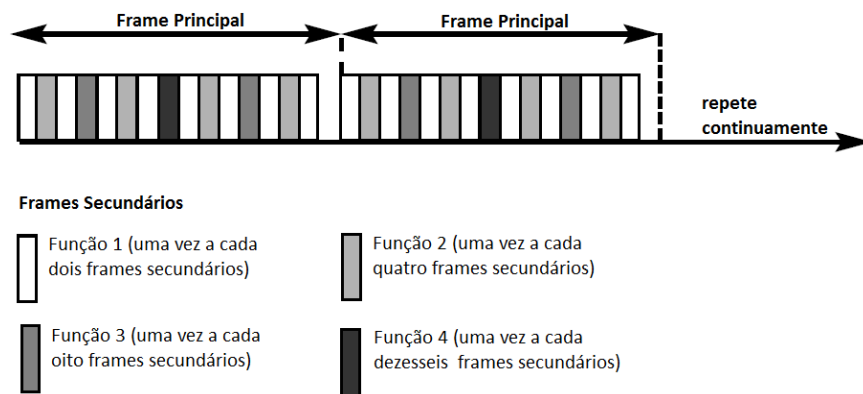


Figura 2.2: ciclo executivo, adaptado de (TIMESYS, 2002).

2.4.2 Prioridade fixa / periódicos

A maioria dos sistemas operacionais de tempo real usa um escalonador de prioridade fixa preemptivo (ou FPS - *Fixed Priority Scheduling*). Nesse caso, o conjunto de processos é fixo, e a cada um destes é atribuída uma prioridade também fixa. Preemptivo significa que, sempre que um processo de maior prioridade está pronto para ser executado, e um

processo de menor prioridade estiver em execução, este último é imediatamente interrompido para que o processo de maior prioridade seja executado. Este esquema é preferido por permitir uma reação mais rápida dos processos de alta prioridade. Esta forma de atribuir prioridades é ótima no sentido de que, se um conjunto de processos P é escalonável, então a atribuição de prioridades monotônicas em função da frequência de execução é uma das soluções para o problema de escalonamento do conjunto de processos P , sem perdas de *deadlines* (SCHOEBERL, 2009; BURNS; WELLINGS, 2009).

Considere um conjunto de N processos P . Sejam T_i e C_i , o período de execução e tempo máximo de execução do processo i , respectivamente. O uso computacional do sistema é calculado da seguinte forma:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \quad (2.1)$$

Demonstra-se que se $U \leq N \cdot (2^{\frac{1}{N}} - 1)$ e se as prioridades dos processos forem atribuídas obedecendo à monotonicidade da frequência de execução ($\frac{1}{T_i}$) e com preempção, então, o conjunto de processos P é escalonável, e portanto todos os *deadlines* são sempre obedecidos. Com o crescimento de N , a expressão converge assintoticamente para 0,63. Esta condição é suficiente, mas não necessária para que o conjunto de processos P seja escalonável. A razão de não ser necessária é que, em muitos casos, é possível ter um uso computacional maior e ainda assim, o sistema ser escalonável (LIU; LAYLAND, 1973).

2.4.2.1 Problema da inversão de prioridade e solução

Usando um algoritmo de prioridade fixa, depara-se com o problema de inversão de prioridade, situação na qual um processo de alta prioridade é bloqueado por um processo de baixa prioridade que detém um recurso compartilhado entre estes processos. A Figura 2.3 exemplifica esta situação. Durante todo o período em que a tarefa de maior prioridade (assinalada com a cor preta) está esperando que o processo de menor prioridade saia da região crítica, o que é razoável. Porém não é aceitável que, nesta condição, o processo de prioridade intermediária interrompa a execução do processo de menor prioridade, pois, por transferência, o processo de maior prioridade está esperando por este (TIMESYS, 2002).

Uma solução para esse problema é o uso de um protocolo de sincronização, como por exemplo o protocolo de herança de prioridades. Usualmente, a implementação deste protocolo está associada às modificações nas funções de acesso à região crítica. A RTSJ (*Real Time Specification for Java*), por exemplo, prevê uma solução baseada em herança de prioridades para resolver o problema de inversão de prioridades.

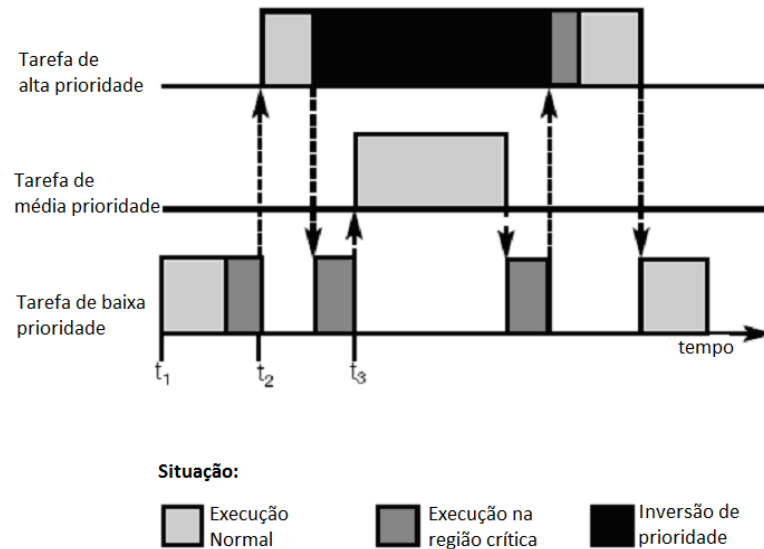


Figura 2.3: inversão de prioridades, adaptado de (TIMESYS, 2002).

2.4.3 Algoritmo do *deadline* mais próximo

Neste algoritmo, também conhecido como EDF (*Early Deadline First*), sempre é escalonada para execução a tarefa que está com seu *deadline* mais próximo. Liu e Layland demonstraram que, se o uso computacional U de um conjunto de tarefas for menor do que 1, então este conjunto é escalonável seguindo este algoritmo e, portanto, os *deadlines* serão sempre cumpridos (LIU; LAYLAND, 1973).

Apesar desse algoritmo permitir um uso computacional efetivo do sistema (100%), as prioridades são calculadas dinamicamente, e isto dificulta a implementação do escalonador. Além disso, em situações não previstas de carga máxima, o determinismo do sistema é menor do que se este fosse baseado em prioridades fixas.

2.4.4 Escalonamento de tarefas aperiódicas

As tarefas aperiódicas são disparadas por eventos tais como requisições do operador, mensagens de emergência, notificação de alcance de limiar, um botão pressionado ou o movimento de um mouse, dentre outros exemplos.

Em particular, uma abordagem para lidar com eventos aperiódicos em sistemas de tempo real, é através do uso de um servidor aperiódico. Este servidor deposita “passes”, os quais são revalidados depois de um certo período após terem sido usados. Quando um evento aperiódico ocorre, este verifica se existem “passes” disponíveis no servidor. Se existirem, o sistema imediatamente processa o evento, e então escalona a criação de outro “passe”, baseado nas políticas de criação de “passes”. Um servidor aperiódico

impõe previsibilidade em tarefas aperiódicas e, portanto, torna-as adequadas para serem escalonadas, utilizando os algoritmos de EDF ou FPS explicados nas seções anteriores.

Após mostrar e discutir nesta seção os principais algoritmos utilizados, para escalonamento de processos em sistemas de tempo real, na próxima seção discorre-se, sucintamente, sobre o gerenciamento de memória em sistemas de tempo real.

2.5 Gerenciamento de memória

Em sistemas de tempo real, uma abordagem bastante adotada para o gerenciamento de memória é criar todos os processos e alocar memória estaticamente, na fase de inicialização do sistema. Portanto, antes da operação propriamente dita do sistema, toda a memória a ser utilizada durante todo o tempo de missão deve estar previamente alocada. A penalização é que o total de memória necessário para o sistema é significativamente maior do que no caso de alocação dinâmica de memória, no qual a alocação é feita em tempo de execução e desalocada sempre que esta região não for mais necessária.

2.6 Tolerância a falhas

Uma falha pode causar um erro e este por sua vez, um defeito conforme mostrado na Figura 2.4. Idealmente, persegue-se detectar, confinar e corrigir a falha antes que esta cause um erro. Por isso, a falha é objeto de estudo primário de sistemas tolerantes a falhas.

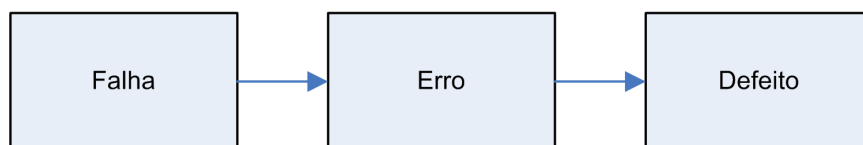


Figura 2.4: sequência de falha, erro e defeito, adaptado de (CASTRO, 1992).

As falhas podem ser classificadas como antecipadas e não antecipadas. Uma falha de projeto, por exemplo, é uma falha não antecipada, pois certamente o projeto não foi feito errado de propósito (ANDERSON; LEE, 1981).

Após detectar uma falha, o sistema deve imediatamente tentar identificar a extensão dos possíveis erros e danos já causados por esta falha. Após isto, medidas devem ser tomadas para que o erro não se propague, ou seja, para que outras partes do sistema não operem baseados em dados errôneos fornecidos pelas partes do sistema que foram afetados

pela falha. Essa fase é conhecida como confinamento da falha.

O processo de correção de falhas envolve, quando possível, recuperar todos os erros causados por esta. Na impossibilidade de recuperar o sistema, este pode tentar operar de forma degradada. Suponha por exemplo, um sistema de telefonia. Uma operação em modo degradado poderia permitir, ou garantir apenas a realização de chamadas de emergência.

Em última hipótese, o sistema deve tentar entrar em modo de falha seguro. No modo de falha seguro, o sistema, ciente de que está defeituoso, não continua a operar, pois, suas execuções podem causar danos maiores ainda. Assim, deve-se prever na arquitetura do sistema, recursos adicionais (de *hardware* e de *software*) que suportem procedimentos para tolerância a falhas. A arquitetura de um sistema tolerante a falhas é projetada de forma a fornecer ao sistema recursos extras, de *hardware* e de *software*, para realizar as ações já descritas de detecção e correção de falhas.

Uma das principais arquiteturas de *hardware* para sistemas tolerantes a falha, é a arquitetura de redundância modular tripla ou TMR (*Triple Modular Redundancy*). Na redundância modular tripla, o sistema computacional é replicado três vezes e um votador é adicionado. Os três sistemas computacionais funcionam em paralelo (*hot stand-by*) e o resultado considerado correto é aquele apontado por no mínimo dois módulos. O votador, que é um ponto de falha comum, por ser um sistema muito menos complexo que uma das réplicas do sistema computacional, usualmente tem uma probabilidade de falha pequena comparada aos módulos computacionais. Ainda assim, é possível replicar também o votador. Note que em um sistema TMR a falha somente é detectada após a mesma ter causado um erro computacional, porém antes que o erro se torne um defeito.

2.6.1 Cálculo de confiabilidade

O cálculo da confiabilidade de um sistema é o ponto de partida para a decisão de se adicionar técnicas de tolerância à falhas. Pela comparação entre a confiabilidade esperada e a confiabilidade atual do sistema, o projetista pode tomar decisões.

Para sistemas simples, sem redundância, estes cálculos são triviais e se baseiam em probabilidade simples. Por exemplo, a confiabilidade de um sistema série é calculada multiplicando-se a confiabilidade de cada um dos componentes do sistema, dada por (CASTRO, 1992)

$$R_s = \prod_{i=1}^n R_i, \quad (2.2)$$

sendo R_s a confiabilidade do sistema série e R_i a confiabilidade do componente i .

Em sistemas com redundância, os módulos replicados podem entrar em funcionamento desde o início da fase de missão (*hot standby*), ou dinamicamente de acordo com as necessidades do sistema (*cold standby*). Baseado em um diagrama de estados dos módulos do sistema, o cálculo da confiabilidade pode ser feito baseado em cadeias de Markov (SHOUMAN, 2002).

2.6.2 Detecção e correção de erros em uma sequência de dados

Sistemas com correção de erro sem reenvio de informações são também conhecidos como FEC (*Forward Error Correction*). Nesse caso, o código de correção de erros ou ECC (*Error Correction Code*), que são dados redundantes, são enviados em um único pacote ou bloco juntamente com os dados originais.

2.6.2.1 Checagem por redundância cíclica

CRC (*Cyclic Redundancy Check*) é um algoritmo utilizado para detecção de erros durante transmissão de dados. Para detectar tais erros, o CRC utiliza-se de um complexo polinômio para gerar um número baseado no dado a ser transmitido. O cálculo do CRC é feito pelo dispositivo que irá enviar esses dados (transmissor) e, após a transmissão, pelo dispositivo que os recebeu (receptor). Caso os dois dispositivos tenham obtido os mesmos valores de CRC, a transmissão ocorreu livre de falhas. Existem várias formas de calcular o CRC, diferem-se pelo polinômio adotado e pela forma de entrada dos dados, paralelamente ou serialmente (SPRACHMANN, 2001; JOSHI; DUBEY; KAPLAN, 2000).

2.6.2.2 Algoritmo de Hamming

O algoritmo de Hamming é capaz de corrigir e detectar erros em uma sequência de *bits* e é relativamente simples e de fácil implementação, tanto em *software*, como em *hardware*. A limitação do algoritmo de Hamming está nas suas habilidades para realizar correções. É possível detectar e corrigir um erro em um único *bit*. O algoritmo de Hamming é usualmente especificado pelo tamanho do bloco em *bits* e o número de *bits* desse bloco que se refere ao dado original. Por exemplo, Hamming (255, 247), refere-se a uma codificação de Hamming com blocos de tamanho 255 *bits*, sendo 8 *bits* redundantes.

De uma maneira geral, definem-se (SHOOMAN, 2002)

$$\begin{cases} n \text{ como o número de } bits \text{ de redundância; } e \\ (2^n - 1) \text{ o tamanho total do bloco, incluindo } n \text{ } bits \text{ de redundância.} \end{cases} \quad (2.3)$$

Assim, tem-se que o número de *bits* de dados em um bloco é $(2^n - 1) - n$.

Os *bits* redundantes são calculados por uma série de operações “ou exclusivo” (**xor**) sobre os *bits* de entrada, e então montados em um bloco contendo os *bits* de entrada e os *bits* de Hamming, com a ordem de sequenciamento desses *bits* ditada pelo algoritmo.

Para verificar se ocorreu um erro, os valores esperados dos *bits* redundantes são calculados (com base nos *bits* de dados recebidos) e comparados com os *bits* de Hamming recebidos. Caso haja qualquer divergência entre *bits* calculados e recebidos, significa que um erro foi detectado.

Para recuperar o erro, o algoritmo, através de operações ou exclusivo descobre qual a posição do bloco contém um bit errado. Com a posição dada, para realizar a correção basta inverter o *bit* na posição apontada pelo algoritmo. É importante ressaltar que é possível detectar e corrigir um erro ocorrido em apenas um bit.

Modificando-se o algoritmo original de Hamming pelo acréscimo de mais um *bit* de redundância no tamanho do bloco, é possível detectar até dois *bits* errados. O *bit* adicional é um *bit* de paridade do restante do bloco. O processo de correção é idêntico ao algoritmo original e corrige também apenas um *bit*.

2.6.2.3 Outros algoritmos de correção de erros

Existem algoritmos com uma correção de erros mais robusta, capazes de corrigir vários *bits* errados, como Reed Solomon e BCH (Bose-Chaudhuri-Hocqueenghem). Estes algoritmos são preferíveis para correções de erros em canais ruidosos ou em memórias *flash* do tipo **nand** com células multicamada (*Multi Layer Cell*), em que a probabilidade de erros em vários *bits* é alta. Reed Solomon é um caso particular de BCH e tem menor eficiência. Apesar de serem mais eficientes, esses algoritmos são muito mais complexos do que Hamming e, portanto, consomem muito mais células lógicas ou ciclos de relógio (*clock*).

2.7 Java para sistemas de tempo real

2.7.1 Sistemas atuais

A linguagem de programação C é a principal linguagem utilizada atualmente para desenvolvimento de *software* para sistemas embarcados, tanto para o sistema operacional quanto para a aplicação. Funções de suporte a tempo real, gerenciamento de memória e comunicação inter-processos são usualmente desempenhadas por um sistema operacional de tempo real e são acessíveis pela aplicação através de Chamadas de Sistema (TANENBAUM, 2008) e de uma API (*Application Program Interface*).

2.7.2 Vantagens da linguagem Java

As principais características da linguagem Java são: ser fortemente tipada; ter extensa checagem em tempo de execução e compilação; não ter ponteiros; possuir monitores; ter meios de comunicação inter-processos; ser *multi-thread*; ter checagem e tratamento de exceções. Para sistemas operando sobre uma máquina virtual Java, essas funções são desempenhadas pela máquina virtual e estão disponíveis para serem usadas pela aplicação, através de estruturas próprias da linguagem de programação Java e não através de chamadas de sistema. Essa abordagem também é utilizada por outras linguagens de programação clássicas de sistemas de tempo real, como Occam 2, Modula 2 e Ada (BURNS; WELLINGS, 2009). Além disso, a linguagem Java não trabalha com ponteiros e tem uma extensa checagem de exceções em tempo de compilação e de execução, o que a torna menos propensa a erros do programador em relação à linguagem C. Portanto, a linguagem Java permite que o programador de sistemas embarcados use um nível de abstração maior, dando foco no desenvolvimento da aplicação.

2.7.3 Desvantagens da linguagem Java

Em sistemas embarcados, a Máquina Virtual Java pode ser executada no topo de um sistema operacional Linux, por exemplo. Versões recentes da JVM (*Java Virtual Machine*) utilizam recursos como compilação JIT (*Just in Time*) para otimizar o desempenho dos sistemas. No entanto, essa solução não é interessante para sistemas embarcados, devido à restrição de recursos do mesmo quando comparados a um computador pessoal. Uma outra desvantagem é a introdução de *jitter* no tempo de execução das aplicações, devido ao sistema de *garbage collector* (SCHOEBERL; PUFFITSCH, 2010; SCHOEBERL, 2010).

Schoeberl analisou estas restrições da linguagem Java e fez uma implementação eficiente em hardware de uma máquina virtual Java com aplicação em sistemas embarcados de tempo real (SCHOEBERL, 2009).

2.7.4 Especificação Java para sistemas de tempo real

Existem, particularmente, dois tipos de sistema de tempo real que são bastante desenvolvidos em Java, e estão, há vários anos, aguardando por uma especificação e implementação de tempo real para Java: Sistemas Financeiros e Sistemas Embarcados. Apesar da requisição por uma especificação Java ser bem antiga (data de antes do ano 2000), que aliás é a primeira JSR (*Java Specification Request*), a RTSJ (*Real Time Specification for Java*) ainda é um trabalho em andamento.

Diferentemente de outras linguagens de programação, quando se fala de Java, não é simplesmente da semântica e das funções disponíveis em uma linguagem de programação, mas, além disso, referem-se a um ambiente de execução, a máquina virtual Java. Essa máquina virtual Java pode interfacear diretamente com o *hardware* ou através de um sistema operacional. Em um ambiente de desenvolvimento para tempo real em Java, todos os elementos supracitados devem ser de tempo real, ou seja, a linguagem Java, JVM, RTOS e o *hardware*. Como as características do *hardware* e dos sistemas operacionais de tempo real foram discutidas nas seções anteriores, nós iremos nos deter em JVM e na linguagem de programação Java.

Dois pontos de vista distintos podem ser definidos em relação à especificação RTSJ: o do projetista de máquinas virtuais Java de tempo real e o do programador de aplicações Java de tempo real. Para o primeiro, as características de previsibilidade e confiabilidade precisam ser adicionadas à JVM. Nessa direção, os principais problemas estão relacionados ao gerenciamento de memória (*Garbage Collector*); escalonamento de *threads*; sincronização de *threads*; gerenciamento de tempo com alta resolução; tempo máximo de repostas a interrupções.

De acordo com a RTSJ, o *garbage collector* não deve atuar na área de memória utilizada pelas *threads* de tempo real do tipo *noheap*, pois, a memória alocada para essas *threads* é completamente estática e reside na área denominada *immortal*. Esta é uma região reservada para alocações estáticas, ou seja, na qual o *garbage collector* não atua. Notadamente, a solução adotada para a concorrência por recursos entre *garbage collector* e *noheap threads* foi eliminar a região crítica. Ainda do ponto de vista do projetista de máquinas virtuais, para projetar uma máquina virtual Java compatível com o padrão

RTSJ, o algoritmo de FPS deve ser utilizado para escalonamento de processos, além da adoção de uma solução para o problema de inversão de prioridades.

Do ponto de vista do segundo, o programador de aplicações, este necessita de uma API e semântica adequadas para controlar o comportamento temporal do sistema. Neste sentido, A RTSJ define um conjunto de classes e métodos para facilitar a implementação de sistemas de tempo real. Os primeiros passos são instalar na JVM a extensão RTS e renomear todas as instanciações de `java.lang.Thread` para `javax.realtime.RealtimeThread`. Porém, como não existe uma solução única para programação de sistemas de tempo real, é necessário a cooperação do programador de aplicações para o correto desenvolvimento do sistema.

2.7.5 Especificação Java para sistemas críticos de tempo real

Por consumir muitos recursos do sistema, a RTSJ não é adequada para implementação de uma JVM para uso em sistemas embarcados (SCHOEBERL, 2009). Schoeberl implementou uma máquina virtual Java para sistemas embarcados que atende a um subconjunto da RTSJ.

Não obstante, a JSR de número 302 refere-se à extensão do padrão Java 2 Micro Edition (J2ME) para aplicação em sistemas com nível de confiabilidade crítico (*hard real time*). A especificação baseia-se na RTSJ e contém as características mínimas necessárias para um sistema ser certificado como DO-178B compatível (SCHOEBERL et al., 2007). Esta especificação foi nomeada SCJ (*Safety Critical Java*) e encontra-se em fase de desenvolvimento (JAVA COMMUNITY PROCESS, 2008).

Conforme pode ser visto na Figura 2.5, três fases são claramente distinguíveis no fluxo de execução de um sistema baseado em SCJ: inicialização, missão e recuperação. A fase de inicialização não é de tempo real. Nesta fase, todos os objetos são instanciados e assim permanecem por toda a fase de missão, pois não há *garbage collector* na SCJ. Na fase de missão, que é de tempo real, o sistema deve realizar sua função propriamente dita, e assegurar os requisitos de tempo real da missão. Na ocorrência de uma falha que não possa ser tratada pelos mecanismos de tolerância a falhas, sem comprometer a funcionalidade do sistema, o mesmo é levado à fase de recuperação. Nesta fase, os danos são avaliados, a falha é confinada, e dependendo da gravidade do problema, o sistema pode entrar em modo de falha segura ou recuperar a aplicação, e retornar à fase de inicialização.

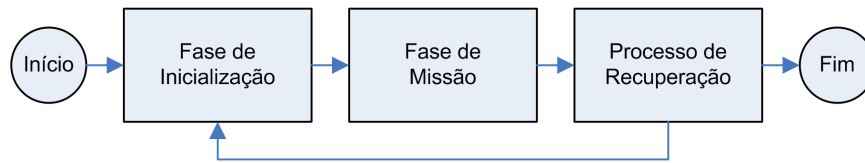


Figura 2.5: fases de execução de uma aplicação com modelo da especificação SCJ, adaptado de (SCHOEBERL et al., 2007).

2.8 Conclusão

Um processador ideal para sistemas de tempo real deve ter um modelo temporal previsível e de fácil análise, ao mesmo tempo, deve dispor dos recursos dos processadores modernos para que tenha um bom desempenho médio.

Deve-se investir para melhorar a confiabilidade de um sistema tanto maior a probabilidade da ocorrência de uma falha e mais custoso seu impacto. Para sistemas de tempo real do tipo *hard*, toda e qualquer característica que possa melhorar a confiabilidade do sistema deve ser considerada, ainda que outras variáveis sejam penalizadas. Os demais sistemas de tempo real (que não envolvem riscos de vidas humanas) são do tipo *soft*, como por exemplo um sistema de recepção de multimídia para entretenimento.

Capítulo 3

Projeto de Circuitos Integrados Lógicos Digitais

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 os *emails*, dentre outros exemplos existentes. Os dispositivos semicondutores, tais como transistores, diodos e circuitos integrados são itens necessários para construção dos equipamentos eletrônicos. Destes itens, os circuitos integrados merecem atenção especial por terem embutido, em um único chip, inúmeras funções e dispositivos eletrônicos.

3.1 Introdução

O projeto de um circuito integrado tem elevado custo de desenvolvimento e de prototipação. A decisão por iniciar o projeto de um novo circuito integrado deve passar por uma análise detalhada de viabilidade técnica e mercadológica. Do ponto de vista técnico, esta decisão está associada aos requisitos da aplicação, tais como desempenho e consumo. Ao se comparar na ordem decrescente de consumo e de aumento de desempenho, conforme mostra a Figura 3.1, tem-se processadores de uso geral, processadores de uso específico, FPGAs (*Field-Programmable Gate Arrays*), CPLDs (*Complex Programmable Logic Device*) e ASICs (*Application-Specific Integrated Circuit*). Os ASICs apresentam o menor consumo e o maior desempenho, justamente por serem específicos para uma determinada aplicação (BURSKY, 2004).

Do ponto de vista mercadológico, as principais variáveis são: o volume de produção esperado, o tempo e o custo necessários para o desenvolvimento. O custo do desenvolvimento deverá ser diluído no volume de produção. Entretanto, se o tempo necessário para

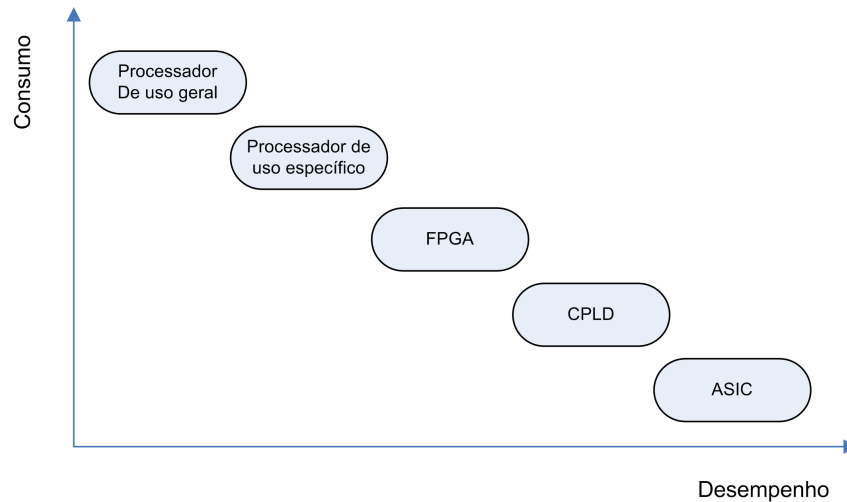


Figura 3.1: consumo x desempenho das tecnologias para implementação de sistemas computacionais.

desenvolver o CI se sobrepuser muito à janela de oportunidade de vendas do produto, então não valeria a pena iniciar seu projeto. Destacam-se dois exemplos que estão nos extremos da relação volume de produção x custo de desenvolvimento. Considere, primeiramente, a decisão que foi tomada para se projetar um CI para aplicação em tocadores portáteis MP3. Do ponto de vista técnico, apenas um ASIC poderia atingir os requisitos de baixo consumo de um dispositivo dedicado a esta aplicação. O volume de produção esperado seria muito grande e um CI para esta aplicação não possui um custo de projeto tão elevado. Logo, a decisão por iniciar este CI é óbvia. Agora, considere um projeto de um equipamento para interfacear com redes óticas e que precisa manipular um volume de dados da ordem de dezenas de Gigahertz. Este equipamento é utilizado por operadoras de telecomunicações e tem uma produção esperada de poucas centenas de unidades e, além disso, tem um custo de produção da ordem de dezenas de milhares de dólares. Neste caso, o projeto de um CI para manipular esse volume de dados é não trivial. Mesmo usando as tecnologias mais modernas de fabricação para viabilizar a operação nesta frequência, o leiaute deverá ser muito otimizado e, portanto completamente manual. Logo, teria o projeto um custo muito elevado e longo tempo de desenvolvimento. Neste sentido, a decisão pelo uso de uma FPGA de alto desempenho e o não início de um novo projeto de CI seria a mais acertada. Circuitos eletrônicos podem ser projetados em vários níveis e em ordem decrescente de complexidade, tem-se: sistema, módulo, porta lógica, circuito e processo (RABAEY; POLLICE; WEST, 2003), conforme mostra a Figura 3.2.

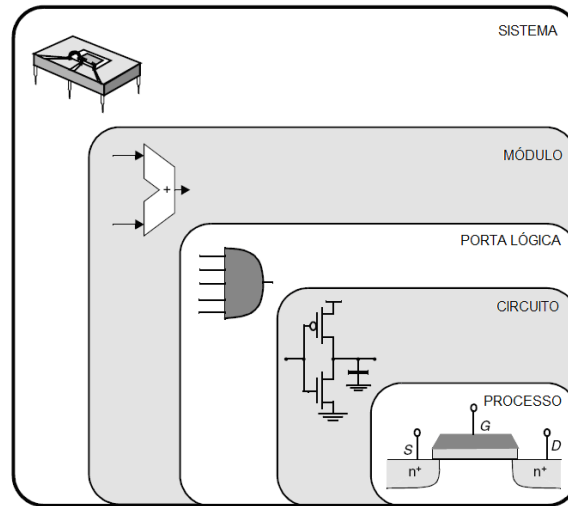


Figura 3.2: níveis de projeto de circuitos integrados, adaptado de (RABAEY; POLLICE; WEST, 2003).

3.2 Estado da Arte e Desafios do *Deep Sub Micron*

Em função do volume de produção, os processos de fabricação para circuitos digitais (ou quase totalmente digitais) baseados em tecnologia CMOS (*Complementary Metal-Oxide-Semiconductor*) dominaram o mercado, sobrepondo-se à tecnologia bipolar. Para circuitos digitais, a vantagem da tecnologia CMOS está no consumo reduzido de energia. Por outro lado, os projetistas de circuitos analógicos, além de trabalharem com a tecnologia CMOS para desenvolverem seus circuitos, que é menos adequada do que a tecnologia bipolar, são obrigados a se adequar a uma redução contínua da tensão de operação (de 12V da tecnologia bipolar para os 3.3V da tecnologia CMOS atual). A faixa de operação pequena de tensão (0-3.3V) exige circuitos analógicos com uma melhor tolerância a ruídos (RAZAVI, 2000).

Nos dias atuais, existem disponíveis processos BICMOS (*Bipolar Junction Transistors and CMOS*), os quais permitem ter no mesmo circuito integrado, circuitos desenvolvidos em tecnologia bipolar e circuitos desenvolvidos em tecnologia CMOS. O processador Pentium Pro, por exemplo, foi desenvolvido usando esta tecnologia. De acordo com o ITRS (*The International Technology Roadmap for Semiconductors*), não existe uma alternativa com condições reais para substituir a tecnologia CMOS, pelo menos para os próximos 15 anos, o que nos permite prever que a dimensão da tecnologia de fabricação dos transistores (comprimento do gate e outros), irá cair de 32nm em 2005 (nó tecnológico de 90nm), para 6nm em 2020 (nó tecnológico de 14nm), permitindo um salto sem precedentes em termos de desempenho e complexidade.

O tamanho da tecnologia de manufatura de circuitos integrados foi reduzido, com

o passar dos anos, seguindo uma progressão geométrica de razão aproximadamente 0,7: $1\mu\text{m}$, $0,7\mu\text{m}$, $0,5\mu\text{m}$, $0,35\mu\text{m}$, $0,25\mu\text{m}$, 180nm , 130nm , 90nm , 65nm , 45nm , 32nm , 22nm (BAKER, 2007). Este quadro sugere que nos próximos anos, a taxa de redução poderá gerar transistores ainda menores (15nm , 10nm , 7nm , $3,5\text{nm}$, $2,5\text{nm}$, $1,8\text{nm}$, $1,3\text{nm}$, $0,9\text{nm}$). Bastante interessante é o fato de que isto é consistente com a lei de Moore (MOORE, 1965). Como resultado da redução contínua das dimensões (inclusive espessura das camadas de silício, dióxido de silício e metal) e da tensão de operação dos dispositivos de silício, o custo da lógica e o consumo de energia diminuem significativamente a cada geração, enquanto a velocidade de operações dos transistores aumenta. Na era sub-micron (tecnologia de manufatura com característica menor do que $1\mu\text{m}$), acreditava-se que o processo de litografia não avançaria além do limite de $0,35\mu\text{m}$ (era do *deep sub micron*), pelo fato de esta dimensão ser próxima do comprimento de onda da luz (HORGAN, 2008). O processo de litografia avançou muito além e hoje, dispositivos comerciais são fabricados na tecnologia de 65nm . Assim, como consequência da redução contínua das dimensões das estruturas dos CIs, problemas relacionados à confiabilidade têm sido cada vez mais críticos. Dois problemas são apontados pelo ITRS como desafios para as próximas gerações de circuitos integrados: projeto para testabilidade e projeto para confiabilidade.

A preocupação em garantir a testabilidade e a confiabilidade dos CIs é parcialmente transferida do nível de processo para o nível de projeto (ATIENZA et al., 2008). Antes, apenas sistemas com altos requisitos de confiabilidade precisavam de técnicas de tolerância a falhas, pois, os dispositivos recebidos pelo integrador eram em sua maioria sem falhas. No novo cenário que se apresenta, devido ao tamanho dos canais de transistores e conexões estarem abaixo do comprimento de onda da luz, o controle das variações de processo é cada vez mais complexo. Por isto, o processo de fabricação necessita de auxílio do projetista do circuito integrado para que, ainda que exista alguma falha no circuito integrado devido ao processo de fabricação, este opere normalmente, graças a um projeto tolerante a falhas. Neste sentido, mesmo para uma aplicação do cotidiano, a falha será um evento considerado normal e o circuito integrado deverá ser capaz de detectar e corrigir a falha (NICOLAIDIS, 2007; ROBERTS; KIM; MUDGE, 2008; ATIENZA et al., 2008). Além disso, deseja-se que o preço pago, em termos de área de silício e de desempenho, seja o mínimo possível.

3.3 Requisitos de Projeto

Os requisitos de projeto de um circuito integrado constituem o ponto de partida para o desenvolvimento do mesmo. É necessário que todos os requisitos sejam especificados antes

da fase de planejamento do projeto. Requisitos específicos da aplicação, tais como: tempo de execução de uma operação; testabilidade; classe de temperatura; consumo; proteção contra ESD (*Electrostatic Discharge*); imunidade a ruído conduzido e irradiado; tolerância a radiação; segurança dos dados; suporte a *debug*, entre outros, deverão ser explicitamente declarados no documento de requisitos (WESTE; HARRIS, 2004). Os requisitos de funcionamento lógico de um circuito, ou seja, a relação entre sinais de entrada e de saída, pode ser feita em forma de um documento de especificação de requisitos ou ainda através da implementação da funcionalidade deste circuito em *software*. Neste caso, um programa executável, e possivelmente seu código fonte, são fornecidos como ponto de entrada para o projetista de circuitos lógicos digitais.

3.4 Projeto e Processo de Fabricação

Baseado nos requisitos, a tecnologia de fabricação deverá ser selecionada. Vários arquivos de tecnologia usualmente fornecidos pelos fabricantes de circuitos integrados são necessários para a continuação do projeto: DK (*Design Kit*), *standard cells*, *PAD Cells* e Gerador de Memórias. Uma Biblioteca de *standard cells* é um conjunto de portas lógicas desenvolvidas e testadas para um determinado processo de fabricação. Para cada porta lógica, têm-se esquemático, leiaute, características temporais e elétricas. O leiaute de cada porta lógica é desenhado dentro de uma área retangular e a altura desse retângulo é a mesma para todas as portas lógicas. Os pontos de conexão para alimentação da porta lógica estão sempre no topo e na base da célula (WESTE; HARRIS, 2004). No caso de uma memória, cada *bit* é formado por 6 (seis) ou mais transistores.

Esses *bits* estão dispostos em forma de uma matriz bidimensional e representam estruturas de silício extremamente regulares, e por isso existem *softwares* para geração automática de módulos de memória. Esses *softwares* são conhecidos como geradores de memória ou ainda compiladores de memória. São específicos para o processo de fabricação e podem ser fornecidos pelo fabricante do circuito integrado. O processo de fabricação determina regras para o desenvolvimento do leiaute, tais como: espaçamento mínimo entre conexões e dimensões mínimas das conexões. Essas regras são passadas ao projetista por escrito e em forma de arquivos de tecnologia. Esses arquivos são específicos para o *software* de leiaute. Para o *software* da Cadence, por exemplo, esses arquivos estão armazenados na forma de um Kit que é conhecido como CDK. A partir das informações disponíveis no CDK, o *software* de leiaute é capaz de auxiliar o engenheiro de leiaute, alertando-o sempre que uma regra de leiaute for violada. *Softwares* automáticos de leiaute

também fazem uso do CDK para gerarem leiautes compatíveis com o processo selecionado (WESTE; HARRIS, 2004).

Para universidades, pequenas empresas e centros de projetos de circuitos integrados, o caminho mais rápido para se ter acesso a esses kits é através de instituições que oferecem serviços de MPW (*Multi Project Wafer*), como por exemplo, o serviço Mosis da Universidade da Carolina do Norte. Para preservar a propriedade intelectual, as células fornecidas pela fábrica não contêm informações a respeito do leiaute interno das células, pois são somente *front-end kits* (USC, 2010). Para alguns processos de manufatura, existem kits e bibliotecas de *standard cells*, desenvolvidas por universidades e que são inteiramente abertas e gratuitas (NCSU, 2010; VTVT, 2010; OSU, 2010). Para fins de aprendizado, esses kits são, sem dúvidas, extremamente úteis. Para fins de desenvolvimento, esses kits não são, possivelmente, tão otimizados como os kits fornecidos pelos fabricantes de CIs.

3.5 Fluxos de Projeto

Dois fluxos principais de projeto se distinguem: *full custom* e *standard cells*. No fluxo de projeto baseado em *standard cells*, um circuito definido utilizando uma HDL (*Hardware Description Language*) é sintetizado em termos de portas lógicas, que são as *standard cells*, enquanto que no fluxo de projeto *full custom*, os circuitos são desenhados em nível de transistores. Em um mesmo projeto, podem-se utilizar ambos os fluxos, porém em partes diferentes do projeto. No projeto de processadores, é comum desenvolver o bloco de controle utilizando *standard cells* e desenvolver a unidade de execução em um fluxo *full custom*, o que permite otimizar em termos de área e de desempenho esta unidade. Nesta seção, descrevem-se ambos os fluxos de projeto. Nessa descrição, serão mencionadas ferramentas como exemplo.

3.5.1 *Standard Cells*

A Figura 3.3 mostra o fluxo de desenvolvimento, baseado em *standard cells* de um circuito lógico integrado digital. Conforme mostrado nesta figura, a saída de cada fase, sempre que possível, é confrontada com a especificação original ou com a saída de fase anterior, de forma a tornar o fluxo de desenvolvimento mais robusto.

Após gerar a especificação de requisitos, conforme descrito anteriormente, o projetista lógico descreverá o circuito lógico, para fins de síntese, em nível RTL (*Register Transfer Level*), na linguagem de sua escolha, VHDL, Verilog ou outra.

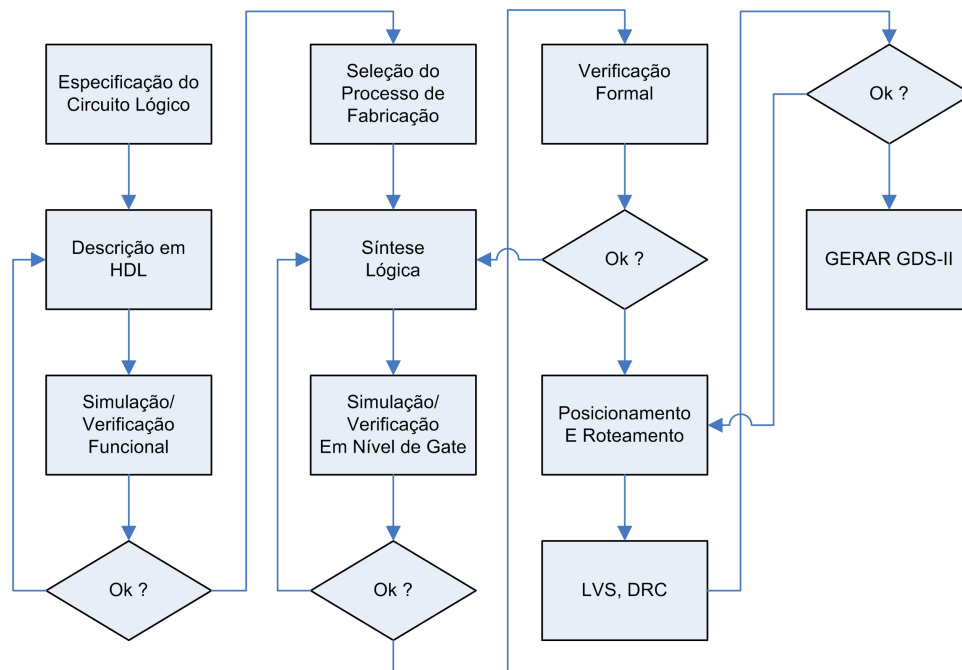


Figura 3.3: fluxo de desenvolvimento de um circuito lógico integrado digital, baseado em *standard cells*.

Utilizando-se uma metodologia de verificação funcional, a implementação desse circuito lógico deve ser verificada contra a especificação original. Uma metodologia de verificação que se provou eficaz é a metodologia VeriSC (MELCHER et al., 2006), tendo esta metodologia evoluído para a metodologia BVM (*Brazil-IP Verification Methodology*), que substituiu com vantagens a metodologia VeriSC. Esta metodologia (BVM) foi implementada na linguagem SystemVerilog, utilizando conceitos e biblioteca de OVM (*Open Verification Methodology*). Esta metodologia tem como objetivo aumentar a produtividade do engenheiro na realização no processo verificação funcional (OLIVEIRA, 2010).

O circuito lógico deve ser testado em FPGA, e todo o fluxo de projetos para FPGA deverá ser seguido antes de continuar a sequência deste fluxo. Na fase de síntese as restrições temporais dos sinais, incluindo frequência de operação do circuito, são definidas. A saída desta fase é um arquivo de descrição de *hardware* estrutural, baseado nos modelos de portas lógicas disponíveis na biblioteca de *standard cells*, que consiste de um *netlist* em formato Verilog. Duas ações podem ser tomadas para garantir que a síntese lógica seja realizada com sucesso: uma simulação em nível de portas lógicas e uma verificação formal.

A verificação formal é um processo rápido e simples do ponto de vista do usuário, realizado, por exemplo, com o auxílio da ferramenta Conformal (DRECHSLER, 2010). A simulação em nível de portas lógicas é lenta, por consumir muitos recursos computacionais, tendo em vista que cada porta lógica (podem ser milhares ou milhões) será simulada

como uma chave, levando em consideração características como atraso que estão disponíveis no arquivo SDF (*Standard Delay Format*), fornecido junto com as *standard cells*. O projetista poderá optar por não realizar esta simulação, desde que a ferramenta de síntese garanta que as restrições temporais dos sinais sejam cumpridas. Para o leiaute do circuito integrado, utiliza-se, por exemplo, a ferramenta Encounter e as seguintes etapas são realizadas:

- primeiramente, a área de silício ocupada é definida;
- os *pads* são posicionados nas bordas da região de silício;
- anéis de alimentação (vcc e gnd) são desenhados com espaçamento, posicionamento e camada de metal definidos pelo usuário;
- as portas lógicas são posicionadas, manual ou automaticamente;
- a árvore de *clock* é sintetizada, automática ou manualmente; e
- finalmente as conexões entre as portas lógicas são realizadas, manual ou automaticamente, utilizando área e camadas de metal indicadas pelo usuário.

A inserção de portas lógicas de reserva (*spare gates*) poderá representar um ganho de tempo e de recursos financeiros, caso algum erro seja detectado durante a fase de testes. Para a manufatura de uma segunda versão, ao invés de se alterar todas as máscaras, é possível que se precise alterar apenas algumas delas, portanto reduzindo o custo da prototipação da segunda versão do silício. É importante lembrar que na etapa de síntese lógica, as características elétricas das conexões são consideradas ideais. Ao fazer o leiaute, estas conexões ideais passam a ser reais. Neste sentido, as características dessas conexões devem ser então consideradas para verificar se as restrições temporais e elétricas estão sendo cumpridas. Isto é feito com o auxílio da ferramenta Celt IC.

Uma vez que as células fornecidas pela fábrica, para universidades e pequenas empresas, não contém informações a respeito do leiaute interno das células (pois são somente *front-end kits*), torna-se mais complicado fazer o processo de verificação final LVS (*Layout Versus Schematics*), que verifica o esquemático gerado pela síntese lógica contra o circuito lógico extraído a partir do leiaute final (usando o *software* Virtuoso). Um outro processo final de checagem, o DRC (*Design Rule Check*), o qual verifica se as restrições de leiaute impostas pelo arquivo de tecnologia estão sendo seguidas, não pode ser realizado integralmente devido à falta do leiaute interno das células. Portanto, o processo como um todo depende das fases anteriores e deve ser correto por construção. Todos os passos

acima podem ser realizados utilizando interfaces gráficas e inserindo os dados de entrada manualmente, ou de forma completamente automatizada através de scripts em TCL (*Tool Control Language*) e Makefile. Este último é particularmente interessante para refazer o projeto inteiro, após uma modificação, com apenas um único comando. Então o arquivo de manufatura no formato GDSII (*Gerber Data Stream Information Interchange II*) é gerado e pode ser enviado por exemplo, ao Mosis, o serviço de manufatura da universidade Carolina do Norte para universidades e pequenas empresas. No Mosis, as células serão instanciadas, ou seja, o leiaute interno das *standard cells* serão combinados com o leiaute do circuito integrado. Finalmente, o arquivo de leiaute é enviado para o fabricante do circuito integrado.

3.5.2 *Full Custom*

No fluxo *full custom*, o circuito lógico e o leiaute são inteiramente desenvolvidos em nível de transistor. Com este fluxo de projeto, é necessária uma equipe de engenharia, pois cada parte do circuito deve ser manualmente projetada. Como resultado, obtém-se um projeto muito otimizado em termos de área e velocidade de execução. A Figura 3.4 mostra o fluxo de desenvolvimento de um módulo ou de um circuito integrado utilizando um fluxo *full custom*.

Após a especificação, que deve ser elaborada em forma de um documento de especificação de requisitos, seleciona-se o processo no qual o módulo ou CI será fabricado. Selecionada a tecnologia, a fábrica que detém a mesma deverá fornecer um kit de desenvolvimento, que contém células básicas, como transistores, resistores, capacitores e *pads*. Essas células são usualmente conhecidas como PCELLs ou células parametrizáveis. São parametrizáveis porque é possível escolher o comprimento e largura de um transistor antes de instanciá-lo em um circuito.

De posse do kit de desenvolvimento, e a partir da especificação, o circuito eletrônico será projetado utilizando as PCELLs disponíveis no kit de desenvolvimento. Para cada PCELL, o projetista do circuito eletrônico deverá informar os parâmetros da referida célula (W, a largura, e L, o comprimento, no caso de transistores). Em função do circuito eletrônico, incluindo-se os parâmetros e modelos das PCELLs, uma simulação do circuito descrito em linguagem SPICE (*Simulated Program with Integrated Circuits Emphasis*) deverá ser realizada para verificar se o projeto está de acordo com as especificações. O passo seguinte é a elaboração do leiaute, que compõe-se do posicionamento das PCELLs e roteamento das conexões entre estas. Após o leiaute, faz-se uma re-simulação do circuito,

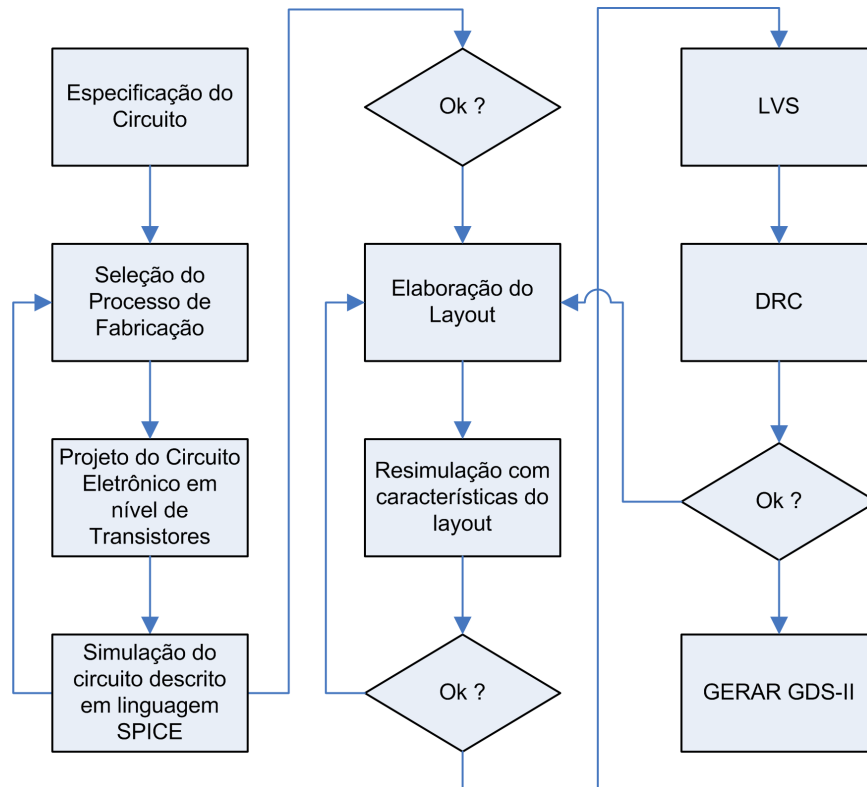


Figura 3.4: fluxo de desenvolvimento de um circuito integrado (lógico digital ou analógico) em nível de transistores (*full custom*).

porém agora incluindo-se as características elétricas dos componentes parasitas ¹ gerados pelas conexões internas do circuito integrado. Na etapa seguinte, verifica-se se o leiaute está de acordo com o esquema elétrico projetado através de um procedimento conhecido como LVS. Finalmente, é verificado se as regras de leiaute impostas pelas restrições da tecnologia em uso estão sendo cumpridas; isto é feito por um procedimento identificado como DRC (*Desing Rule Check*).

3.6 Reuso de módulos

Módulos desenvolvidos utilizando fluxo *full custom* ou *standard cells* em projetos anteriores podem ser reusados em novos projetos, desde que tenham sido desenvolvidos com tal finalidade (SANTOS et al., 2008). Além da redução do tempo de desenvolvimento, por serem módulos testados em projetos anteriores, diminuem o risco do projeto atual. O desenvolvimento para reuso baseia-se no uso de padrões de interconexão entre módulos de propriedade intelectual. Por exemplo, para módulos *soft*, a descrição lógica, em nível

¹Os finos “fios” metálicos usados para as conexões, assim como o acoplamento eletromagnético das camadas que compõem o circuito podem gerar dispositivos indesejáveis, tais como indutores e capacitores, os quais são classificados como componentes parasitas.

comportamental ou estrutural é reusada. Portanto, o leiaute deve ser refeito para o processo de fabricação selecionado. Restrições temporais e elétricas deverão ser garantidas para este novo leiaute do módulo. Módulos do tipo *hard* são específicos para um processo de fabricação e incluem o leiaute do módulo. Desta forma, o reaproveitamento é total e não é necessário refazer e verificar as restrições temporais e elétricas para o leiaute do módulo.

Capítulo 4

Processador JOP

Um processador Java é uma implementação da máquina virtual Java. Essa implementação não é necessariamente completa em *hardware*, pois uma Máquina Virtual Java contém funções complexas como, por exemplo, escalonamento, gerenciamento e intercomunicação de processos. O custo de implementar todos esses recursos em *hardware* pode tornar a implementação não viável. Portanto, o conceito de um processador Java difere de um processador comum, onde apenas elementos de *hardware* estão envolvidos. Dessa forma, um processador Java é uma implementação baseada em *hardware* e, possivelmente, em algum *software*.

Para um processador Java de tempo real, a característica de tempo real do processador deve permear tanto o *software*, como o *hardware* deste processador. O JOP (Java Optimized Processor) (SCHOEBERL, 2008) é uma implementação em *hardware* e *software* de uma máquina virtual Java de tempo real, baseada no perfil J2ME (*Java 2 Micro Edition*) CLDC (*Connected Limited Device Configuration*) e na especificação SCJ (*Safety Critical Java*). Este processador pode ser implementado como *soft ip core* em FPGAs Xilinx ou Altera e, diferentemente da JVM (*Java Virtual Machine*) que é uma máquina CISC (*Complex Instruction Set Computer*), o JOP é, internamente, uma máquina RISC (*Reduced Instruction Set Computer*) (SCHOEBERL et al., 2010; PATTERSON; HENNESSY, 2008), e portanto, contém seu próprio conjunto de instruções.

4.1 Implementação da JVM no JOP

Os *bytecodes* Java são decodificados através de um *pipeline* de instruções equivalentes nativas do JOP. Para alguns *bytecodes* Java, existe uma equivalência biunívoca com instruções nativas do JOP, as quais são executadas em um único ciclo de *clock*. *Bytecodes*

de média complexidade são traduzidos em uma sequência de instruções nativas do JOP, encontradas em uma tabela contida em uma área de memória ROM (*Read Only Memory*), chamada de *JVM microcode* ou Micro-código da Máquina Virtual Java. *Bytecodes* mais complexos, como por exemplo, a instrução **new**, são implementados na própria linguagem Java e, portanto, traduzidos para sequências dos demais *bytecodes*, em tempo de execução. Para otimizar o desempenho de instruções específicas, é possível implementar a mesma em *hardware*. O JOP, assim como a JVM original, é uma “*stack machine*”, ou seja, ao invés de realizar operações sobre um conjunto de registradores, como ocorre em uma arquitetura x86, as operações são realizadas sobre os itens que estão no topo da pilha.

4.2 Entrada e saída

Os dispositivos de E/S do JOP estão mapeados em memória e, do ponto de vista de *software*, os pinos do JOP são acessados pelos métodos `Native.rdMem()` e `Native.wrMem()`. Do ponto de vista do *hardware*, eles são implementados pelo *soft ip core* **scio**, que pode ser visto na Figura 4.1.

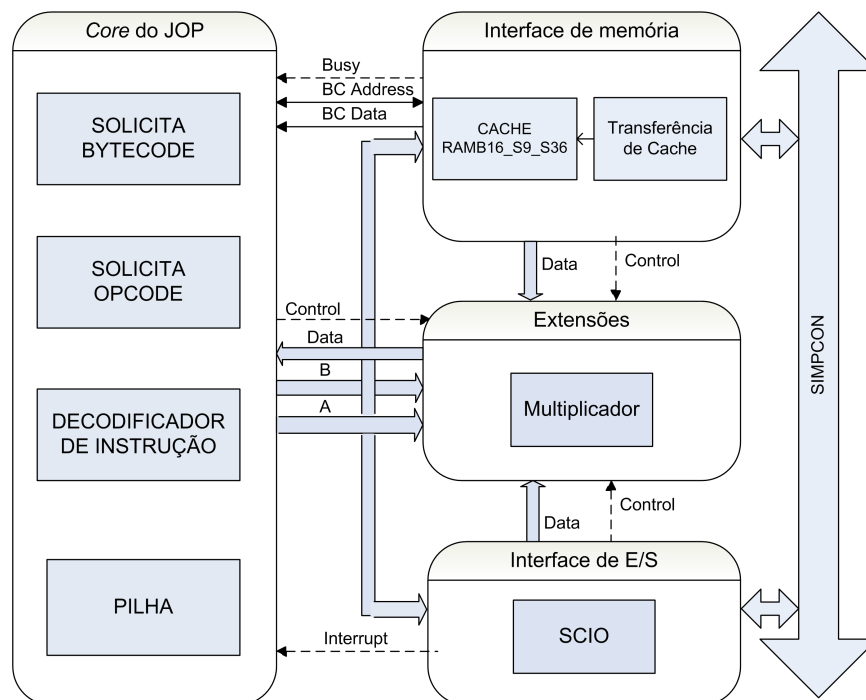


Figura 4.1: diagrama de blocos do JOP, adaptado de (SCHOEBERL, 2009).

4.3 Interrupções

Interrupções são usadas para sinalizar eventos externos, como por exemplo, detectar que um botão foi pressionado. Quando uma interrupção ocorre, o processador simplesmente pára de executar o código atualmente apontado pelo registrador contador de programa, e desvia a execução para uma rotina de interrupção. Além disso, o ambiente do processo interrompido é salvo para que esse possa ser restaurado posteriormente. Isso inclui salvar os registros da CPU (*Central Processing Unit*) e os registradores de *status* do processador. Estas ações tornam possível o retorno da execução do código original quando a rotina de interrupção tiver finalizada.

No JOP, as interrupções geram *bytecodes* especiais (`sys_int`), os quais são inseridos pelo *hardware* de forma transparente, na sequência de *bytecodes* a serem executados. Manipuladores de interrupção podem ser implementados da mesma forma que os *bytecodes*, ou seja, em micro-código ou em Java (KORSOLM; SCHOEBERL; RAVN, 2008).

4.4 Requisitos de tempo real

Aplicações de tempo real para o JOP são explicitamente separadas em duas partes: fase de inicialização e fase de missão. Na fase de inicialização são criados todos os objetos que serão usados durante toda a execução da aplicação e, portanto, áreas de memórias são alocadas e inicializadas. Nesta fase não existe garantia de tempo real. Na fase de missão, as *threads* são executadas concorrentemente de acordo com o algoritmo de escalonamento.

4.4.1 Análise de WCET no JOP

Por ter sido desenvolvido para ser usado em sistemas embarcados com aplicações de tempo real, a arquitetura do processador JOP permite calcular com facilidade o WCET (*Worst Case Execution Time*) de uma tarefa.

A máquina virtual Java do JOP implementa classes que permitem desenvolver aplicações de tempo real. Essas classes não são compatíveis com o padrão RTSJ (*Real Time Specification for Java*) (MIKHALENKO, 2008), pois apenas um subconjunto deste padrão é implementado. Embora as áreas de código e de pilha do JOP utilizem memória de *cache*, o modelamento do comportamento da memória *cache* no JOP é perfeitamente previsível no tempo. Isso se deve ao fato de, diferentemente de outros processadores, não ocorrerem

“*cache misses*” na *cache* do JOP, ou seja, cada instrução solicitada pelo processador à *cache* estará (exceto as instruções `invoke` e `return`) necessariamente previamente carregada na *cache* (SCHOEBERL et al., 2010).

4.4.2 Garantia de funcionamento

O JOP não implementa em sua arquitetura técnicas de garantia de funcionamento. Portanto, para aplicações de tempo real do tipo *hard*, ou seja, que envolvem riscos de vidas humanas, o projetista do sistema deverá assegurar a garantia de funcionamento em nível sistêmico. No JOP, uma falha de *hardware*, como por exemplo, um *opcode* ilegal ou um erro de paridade de memória, levará o sistema a um *shutdown* (SCHOEBERL, 2009).

4.5 Arquitetura do JOP

O JOP é composto de quatro blocos principais (ver Figura 4.1): Interface de memória, *core* do JOP, interface de E/S (**scio**) e “extensões”. O bloco de interface de memória comunica-se com os controladores de memórias externas através do barramento de comunicação **simpcon**. Os controladores de memória, por sua vez, comunicam-se, através dos pinos do processador com as memórias SRAM e *flash*. O bloco “*core* do JOP” é responsável por decodificar e executar os *bytecodes* fornecidos pela interface de memória e comandar as demais partes do processador. O bloco interface de E/S comunica-se com controladores de E/S, tais como porta USB e Serial RS232, através do barramento **simpcon**. Estes controladores de E/S, por sua vez, comunicam-se com os dispositivos do mundo externo através dos pinos do processador. O bloco de “extensões” serve para agregar funções de co-processadores matemáticos sem realizar modificações no núcleo do processador.

4.6 Cache de instruções de tempo previsível

Em Java, ou em qualquer linguagem orientada a objetos, um objeto é composto de dados (variáveis) e das funções que operam sobre seus dados, que são os métodos (MCLAUGHLIN; POLLICE; WEST, 2006).

Soluções convencionais de *cache*, como por exemplo aquelas baseadas em mapeamento direto ou em mapeamento associativo, introduzem uma imprevisibilidade da execução de um algoritmo, o que é incompatível com um sistema de tempo real. A solução de *cache*

de código do JOP, proposta por Schoeberl em (SCHOEBERL, 2004), é completamente diferente das soluções convencionais e consiste em fazer *cache* de métodos inteiros, e não de instruções (*bytecodes*) (SCHOEBERL, 2005).

No JOP, os *bytecodes* `invoke`, `return` e seus derivados são escritos em micro-código (uma sequência de instruções do processador RISC interno ao JOP). Essa sequência de instruções contém uma chamada à instrução `stbcrd` e outra chamada à instrução `ldbcstart`, as quais disparam a atuação da *cache* (cópia da memória RAM externa para memória RAM interna).

A instrução `ldbcstart` insere o endereço do início do método no topo da pilha. Em seguida, a instrução `sbtcrd` é executada e o valor do topo da pilha, o qual neste instante contém o endereço e tamanho de um método, é transferido para o subsistema de memória. Essa operação inicia a transferência da memória principal para a memória *cache*, através de um DMA (*Direct Memory Access*). Nenhum outro acesso à memória externa é permitido durante a leitura dos *bytecodes* de um método.

Com essa abordagem, apenas os *bytecodes* `invoke` e `return` disparam a atuação da *cache*. Portanto, os únicos *bytecodes* que possivelmente terão um tempo de execução variável serão `invoke` e `return`. Essa variação do tempo de execução pode ser calculada em função do tamanho do método e das características das memórias interna e externa. Essa abordagem facilita a análise do WCET das aplicações do JOP.

4.7 Montador de aplicações

O processo de montagem de uma aplicação para o JOP é descrito no ramo direito da Figura 4.2. Conforme pode ser visto no ramo direito desta Figura, após a compilação, usando o compilador `javac`, o aplicativo `jopizer` gera o programa a ser executado pelo JOP. Este programa poderá ser gravado em uma memória *Flash* (aplicação *standalone*) conectada ao JOP ou enviado a este através da porta USB (*Universal Serial Bus*) ou Serial RS232. No ramo esquerdo dessa Figura é apresentado o processo de montagem do hardware do JOP, ou seja, a criação de um arquivo de configuração de uma FPGA (*bitstream*) contendo o *soft ip core* JOP.

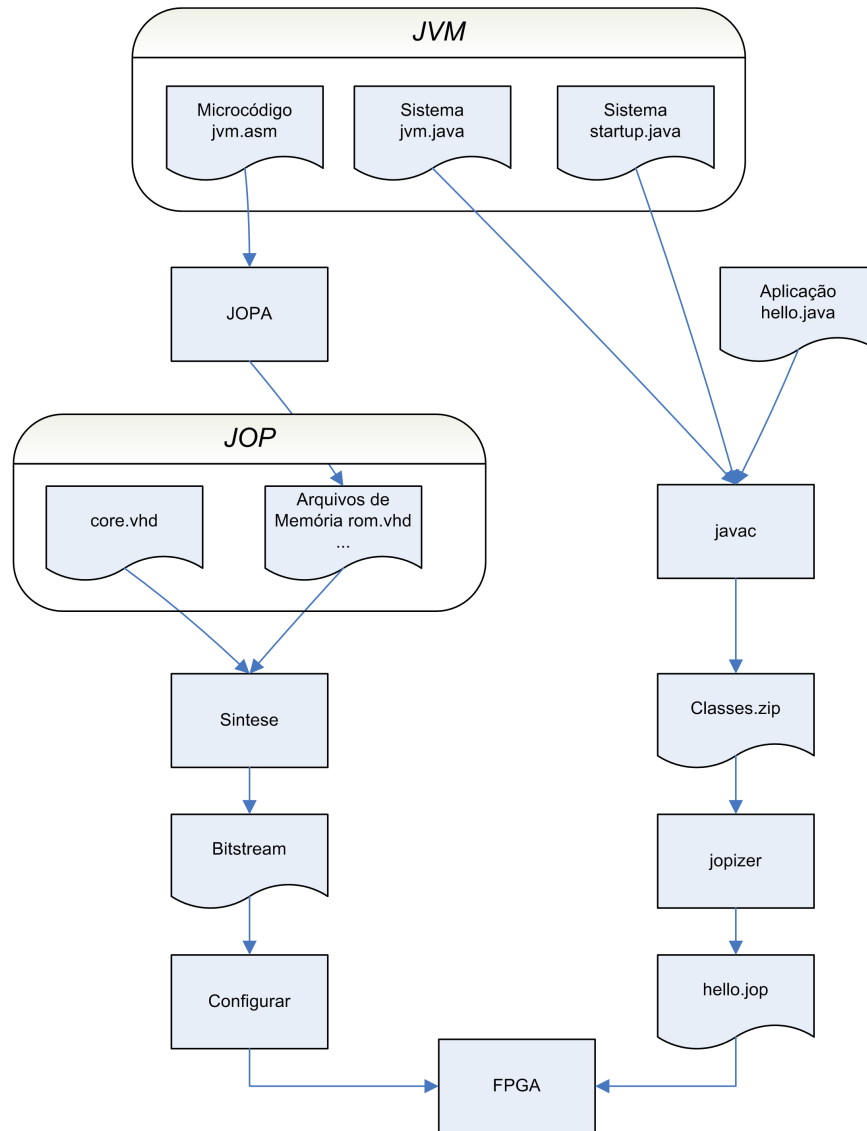


Figura 4.2: fluxo de montagem de aplicativo JOP, adaptado de (SCHOEBERL, 2009).

4.8 Documentação e portabilidade

Além das características técnicas do JOP, descritas anteriormente, podemos destacar também a ampla documentação disponível, a sua portabilidade, pois já foi implementado em diversas placas de desenvolvimento de FPGA (Xilinx e Altera) disponíveis no mercado. Por último, mas não menos importante, a disponibilidade do código fonte do JOP para *download* pelo site <http://www.opencores.org> e licenciado sob a GPL (*Gnu Public License*) versão 3. Atualmente o JOP é utilizado em dois sistemas comerciais, sendo um deles com requisito de tempo real, e em vários sistemas de pesquisa (SCHOEBERL, 2008). Portanto, o JOP se insere como uma excelente alternativa para plataforma base, de pesquisa e desenvolvimento, de novas técnicas de sistemas de tempo real.

Capítulo 5

JOP Tolerante a Falhas

O código de um programa de um sistema embarcado pode ser armazenado de forma permanente em uma memória não volátil do tipo ROM (*Read Only Memory*), que é um tipo de memória tolerante a SEUs. No entanto, não é possível realizar atualizações do programa armazenado nesta memória. Por outro lado, as memórias do tipo *flash* e SRAM (*Static Random Access Memory*), as quais permitem atualização do programa armazenado, são susceptíveis a SEUs (KASTENSMIDT; CARRO; REIS, 2006; HUANG et al., 1998). Caso os erros de memória de programa armazenados nestas memórias não sejam tratados, podem ser disseminados para outras partes do sistema e provocar uma falha catastrófica.

Para detectar e corrigir erros na região de dados de uma memória, existem técnicas bastante efetivas implementadas em *software*. Entretanto, as técnicas aplicadas em nível de *software* para detectar e corrigir erros na região de código de uma memória não são eficazes, pois, o próprio *software* corretor de erros pode estar corrompido (neste também reside a memória de programa). Nesse sentido, neste Capítulo são propostas duas técnicas, para detectar e corrigir erros nas memórias RAM interna (*cache*) e externa do processador JOP, em nível de *hardware* de forma a aumentar a confiabilidade do sistema de *cache* de métodos descritos no Capítulo 4.

5.1 Ocorrência de evento SEU na memória *cache* interna ao JOP

O *floorplan*¹ do JOP original para a tecnologia XFAB XH035 é apresentado na Figura 5.1. Os 4 (quatro) blocos inferiores mostrados nesta Figura, identificados por 1, 2, 3 e 4, representam a região ocupada pela memória *cache* do JOP. Em termos percentuais, a área do bloco de memória *cache* do JOP equivale a 44,46% (RAMOS et al., 2010) da área total do JOP original, excetuando-se a área ocupada pelos *pads* e anéis de alimentação de *vcc* e *gnd*. Quando submetido a radiação, e dado que uma partícula altamente energizada tenha atingido o processador, existe portanto uma alta probabilidade de que a memória *cache* tenha sido atingida, e possivelmente afetada por um SEU. No evento da ocorrência de um SEU na memória interna (*cache*) do JOP, isto provavelmente levaria o sistema baseado neste processador, a uma falha.

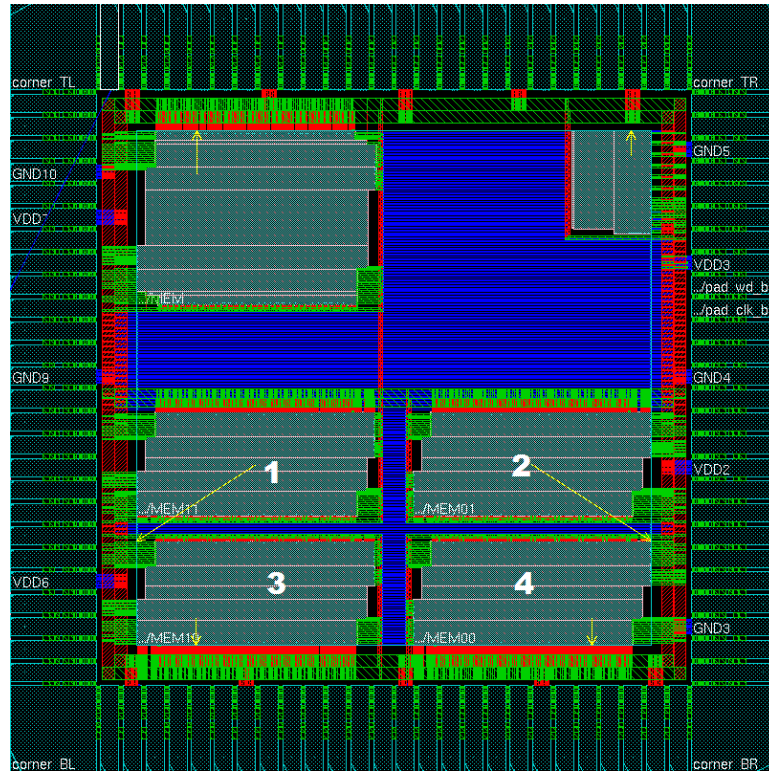


Figura 5.1: *floorplan* do JOP original, adaptado de (RAMOS et al., 2010).

¹Este *floorplan* do JOP original é um resultado preliminar, de um projeto em desenvolvimento no LESC, que faz parte do programa Brazil-IP.

5.2 Técnica de Proteção de Instruções (TPI)

Para lidar com o problema abordado anteriormente, propõe-se o uso de blocos detectores e corretores de erros na memória *cache* do JOP (SILVEIRA et al., 2009).

A técnica de proteção de instruções (ou TPI) detecta e corrige erros ocorridos nos *bytecodes*, desde o armazenamento destes na *cache* até o início de sua execução pelo *core* do JOP. O erro é detectado e corrigido imediatamente antes do *core* iniciar a execução do *bytecode*. Portanto, essa é uma técnica de verificação de último instante (*last minute check*) e que mascara a falha (CASTRO, 1992).

Utiliza-se redundância da *cache* interna para proteger as instruções armazenadas nesta memória. As instruções são armazenadas durante um curto período de tempo na memória *cache*, existindo uma baixa probabilidade de que dois *bits* de uma mesma instrução sejam invertidos antes que outra instrução seja escrita nesta mesma posição de memória. Esta idéia é similar ao princípio no qual se baseia a técnica de *scrubbing* para proteção da memória de configuração de FPGAs, ou seja, antes que ocorram dois SEUs, aquela posição de memória deve ter sido reescrita com o mesmo, ou com outro dado (KASTENS-MIDT; CARRO; REIS, 2006; CASTRO; COELHO; SILVEIRA, 2008). Baseado neste princípio, escolheu-se um codificador de Hamming SECSSED (*Single Error Correction Single Error Detection*) para codificação dos dados da memória *cache*.

5.2.1 Fluxo dos *Bytecodes* no JOP x Fluxo dos *Bytecodes* no FT-JOP

A memória *cache* do processador JOP é uma memória com duas portas para leitura e/ou escrita, identificadas como porta A e porta B. A porta A desta memória permite acessar 2048 endereços, cada um contendo uma palavra de 8 *bits* de dados. A porta B permite acessar 512 endereços, cada um contendo 32 *bits* de dados. No projeto original do JOP, os dados são transferidos da memória externa (SRAM ou *flash*) para a memória *cache* através da porta B e, a partir da porta A, os *bytecodes* são transferidos para o *core* do JOP que irá executar a instrução. Portanto, no JOP original, as instruções seguem o seguinte fluxo: memória externa, *cache* e núcleo do JOP.

O projeto do JOP foi modificado neste trabalho, de modo que simultaneamente à escrita de um *bytecode* (tamanho de 8 *bits*) na memória *cache*, 4 *bits* extras de redundância (*bits* de Hamming) são calculados e armazenados em uma memória *cache*, de redundância, que se posiciona em paralelo com a *cache* original, conforme mostrado na Figura 5.2 -

parte B. Esses *bits* extras são calculados por um *core* escrito em RTL (*Register Transfer Level*), que implementa um codificador de Hamming.

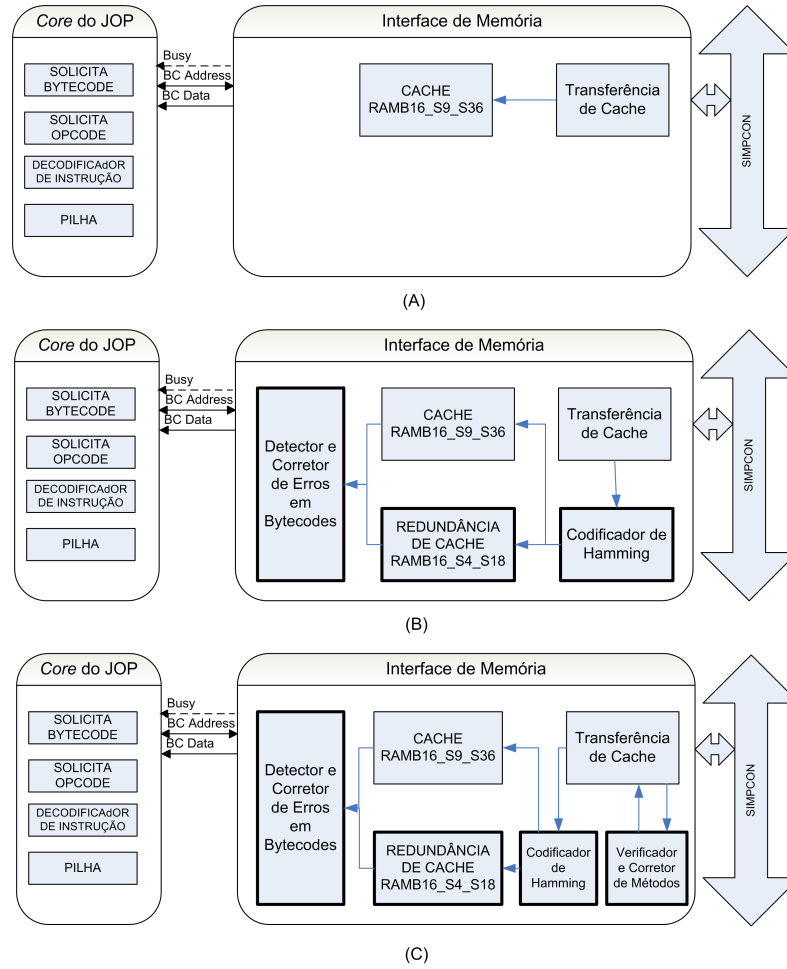


Figura 5.2: Configurações do JOP - (A) JOP original (B) JOP Original e TPI (C) JOP Original, TPI e TPM.

Foram desenvolvidos 2 (dois) *ip cores* e especificada uma memória interna da FPGA para a implementação da Técnica de Proteção de Instruções (TPI), os quais podem ser vistos na Figura 5.2 (parte B). A seguir, listamos e descrevemos cada um destes blocos:

1. codificador de Hamming;
2. detector e corretor de erros em *bytecodes* (Decodificador de Hamming); e
3. redundância de *cache* (RAMB16_S4_S18).

5.2.2 Codificador de Hamming

Foi desenvolvido um bloco codificador de Hamming SECSED, que é inserido entre a memória externa e a memória *cache* do JOP. Este bloco recebe 32 *bits* de dados (ou

quatro instruções, pois cada uma tem 8 *bits*) e gera uma saída de 48 *bits*.

5.2.3 Bloco extra de memória

Além do bloco codificador de Hamming, foi acrescentado ao JOP uma memória, também de porta dupla, para armazenar os *bits* extras gerados pelo codificador de Hamming. Esta é uma memória do tipo RAMB16_S4_S18. De acordo com a Tabela 5.1, a porta A desta memória permite acessar 4096 endereços, cada um contendo uma palavra de 4 *bits* de dados, e a porta B permite acessar 1024 endereços, cada um contendo 16 *bits* de dados.

Tabela 5.1: componentes de memória utilizados.

Componente	Porta A	Porta B
RAMB16_S9_S36	2048 x 8	512 x 32
RAMB16_S4_S18	4096 x 4	1024 x 16

5.2.3.1 Decodificador de Hamming

Imediatamente após a memória *cache* fornecer um *bytecode* para o *core* do JOP, porém antes de ser executado, um *core* decodificador de Hamming lê os 4 *bits* armazenados na *cache* de redundância, conforme mostrado na Figura 5.2 - parte B. Com base nesses 12 (doze) *bits*, oito *bits* do *bytecode* mais quatro *bits* de Hamming, esse *core* verifica se houve alguma inversão de bit em algum dos *bits* do *bytecode*. Em caso afirmativo, isto significa que houve uma falha. Neste caso, o *core* decodificador de Hamming corrige automaticamente o *bytecode*, desde que apenas um bit tenha sido invertido. Finalmente, o *bytecode* correto é entregue para a execução por parte do *core* do JOP.

5.3 Ocorrência de evento SEU na memória externa ao JOP

Durante a inicialização do processador JOP, todo o código é transferido da memória *flash* para a memória RAM. Após o código estar na memória RAM, o sistema de *cache* transfere, sob demanda, métodos inteiros da memória externa para a memória de *cache* interna. Por último, após um método estar completamente carregado na memória interna,

o *core* solicita e executa os *bytecodes*, um a um. No caso de ocorrência de um SEU na memória externa (*Flash* ou RAM) do JOP, isto provavelmente levaria o sistema baseado neste processador, a um defeito.

5.4 Técnica de proteção de métodos

Para proteger o JOP deste tipo de ocorrência, foi desenvolvida neste trabalho a técnica de proteção de métodos. Esta técnica detecta e corrige erros ocorridos desde o processo de transferência do código da memória *flash* até a gravação deste na memória *cache*. O erro é detectado e corrigido antes de iniciar a execução do método. Portanto, essa é uma técnica de checagem antecipada (*early check*).

5.4.1 Software para adicionar CRC aos métodos

Foi desenvolvido um software (de nome **ProtegeMetodo**) para calcular o CRC32 de cada método, e adicionar este dado ao final de cada método, no arquivo a ser gravado na memória de programa do JOP. Portanto, essa técnica requer que o CRC (*Cyclical Redundancy Check*) (SPRACHMANN, 2001) de todos os métodos contidos no programa da aplicação sejam calculados e anexados à memória de código do JOP. Isto é feito pelo aplicativo **ProtegeMetodo** na fase de montagem do programa.

Para ilustrar o funcionamento da técnica de proteção de métodos, considera-se o programa `HelloWorld.java` mostrado abaixo, o qual envia uma mensagem para a porta serial do processador JOP.

```
// Código fonte do HelloWorld
package test;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World from JOP!");
    }
}
```

Como pode ser visto neste código fonte do programa `HelloWorld`, o método `main` invoca o método `println` da classe `System.out`. Apresenta-se a seguir o código fonte de baixo nível do método `main` gerado pelo compilador `javac`.

```
// Bytecode do HelloWorld

test.HelloWorld:main([Ljava/lang/String;)V
Code(max_stack = 2, max_locals = 2, code_length = 9)
0:   getstatic_ref java.lang.System.out Ljava/io/PrintStream; (2)
3:   ldc "Hello World from JOP!" (3)
5:   invokevirtual java.io.PrintStream.println (Ljava/lang/String;)V (4)
8:   return
```

Nota-se que os *bytecodes* `getstatic_ref`, `ldc`, `invokevirtual` e `return` são utilizados para a implementação do método `main`. Os *opcodes* relativos a estas instruções, assim como a quantidade de operandos que cada uma dessas instruções deve receber são mostrados na Tabela 5.2. Mostra-se abaixo o código de máquina gerado pelo compilador, o

Tabela 5.2: instruções utilizadas no método `main` do programa `HelloWorld.java`.

Instrução da JVM	Opcod	Operandos
<code>getstatic_ref</code>	224	2
<code>ldc</code>	18	1
<code>invokevirtual</code>	182	2
<code>return</code>	177	0

qual deve ser gravado na memória externa do processador JOP.

```
// JOPizer do hello world

// 177: main([Ljava/lang/String;)V
-536833006, // 224 0 148 18
45481987, // 2 182 0 3
-1325400064, // 177 0 0 0
```

Neste código de máquina, cada linha contém uma palavra de 32 *bits* e portanto, 4 *bytes*. Cada um destes *bytes* pode representar um *opcode* ou um operando de um *opcode*. Por exemplo, na linha 1, observa-se o opcode 224, que de acordo com a Tabela 5.2, representa a instrução `getstatic_ref`. Ainda na linha 1, encontram-se os *bytes* 0 e 148, que são operandos da instrução anterior (`getstatic_ref`), e o *byte* 18 que representa a instrução `ldc`. Mostra-se a seguir o código de máquina gerado pelo programa `ProtegeMetodo` para este exemplo.

```
// JOPizer do hello world

// 177: main([Ljava/lang/String;)V
-536833006,    // 224    0    148   18
45481987,      // 2      182   0    3
-1325400064,   // 177    0    0    0
1334286165,    // CRC: 0x4F879755
```

Como pode ser visto neste código gerado pelo programa `ProtegeMetodo`, uma palavra de 32 *bits* extra foi adicionada ao código de máquina. Esta palavra extra é o resultado do algoritmo CRC32 aplicado sobre todas as palavras que compõem o método.

5.4.2 Verificador da integridade dos métodos

Sempre que um novo método for carregado na memória *cache* de código, a *cache* de métodos tolerante a falhas e de tempo previsível faz, em paralelo com a carga do método, o cálculo do CRC deste. Para realizar este cálculo foi adicionado ao bloco de interface de memória do JOP, um bloco verificador e corretor de métodos, conforme apresentado na Figura 5.2 - parte C. Ao final da carga do método na memória *cache*, porém antes de executá-lo, o valor calculado do CRC do método é comparado com o valor do CRC do método lido da memória. Nessa situação, caso os valores dos CRCs do método, lido da memória e calculado sejam diferentes, isto significa que ocorreu uma falha no sistema.

Logo após a detecção da falha, uma exceção é gerada e o método em falha é impedido de ser executado, evitando assim que um erro seja gerado por essa falha. É importante salientar que esta técnica detecta uma falha antes que a mesma possa causar um erro e, portanto, não causa um defeito no sistema.

Quando uma falha é detectada, o conteúdo do método do próprio CRC pode estar corrompido, pois, os dois valores (lido e calculado) estão armazenados na mesma memória. No entanto, estatisticamente é muito mais provável que o método esteja corrompido e não o CRC. Uma possível estratégia para aumentar a confiabilidade e detectar qual dos dois está realmente corrompido, seria armazenar o CRC de forma redundante, ou seja, em duas ou mais áreas distintas da memória SRAM.

Com a modificação proposta por esta técnica, o *cache* de métodos do JOP continua a ter previsibilidade de tempo, pois, não houve alterações no núcleo do processador ou em seu *pipeline*, mas somente na interface de memória. Após a detecção do erro, duas abordagens podem ser seguidas pelo projetista do sistema. Na primeira, o projetista

pode optar por não tentar corrigir o método e levar imediatamente o sistema a um modo de falha segura. Numa segunda abordagem, indicada para sistemas de tempo real do tipo *hard*, a análise de WCET deve considerar, além do tempo de cálculo do CRC, o tempo de correção do método, que pode ser feita, por exemplo, a partir de uma memória de programa secundária (externa ao JOP). Na primeira abordagem, o sistema tem um WCET menor que na segunda; em compensação possui uma confiabilidade menor do que o segundo caso, porém ainda maior que a *cache* de métodos original do JOP.

5.4.3 Percepção da falha em nível sistêmico

Uma falha de *hardware* no JOP original, como por exemplo, um *opcode* ilegal, leva o sistema a um *shutdown* (SCHOEBERL, 2009). Neste trabalho, o processador foi modificado para que, na ocorrência de uma falha de *hardware*, uma exceção seja gerada. Segue abaixo um exemplo de código de como o programador deve fazer o tratamento de uma falha de *hardware*.

```
static int saved_sp;
// Executado em caso de uma exceção
// gerada pelo hardware
static void except() {
    saved_sp = Native.getSP();
    if (Native.rdMem(Const.IO_EXCPT)==Const.EXC_CRCCACHE)
    {
        // Tratamento da exceção
        handleException();
    }
}
```

5.5 Aplicabilidade das técnicas de TPI e TPM

As duas técnicas propostas neste Capítulo (TPI e TPM) podem ser utilizadas para aumentar a confiabilidade do processador JOP, tanto em FPGA, como em tecnologia CMOS. No entanto, para o caso de FPGAs, é necessário aplicar em conjunto com as técnicas de TPI e/ou TPM, uma técnica que proteja a memória de configuração da FPGA contra SEUs, como por exemplo a técnica de *scrubbing* (KASTENSMIDT; CARRO; REIS,

2006), ou ainda a técnica de votação de CRC de frames da FPGA, proposta no trabalho desenvolvido por (CASTRO; COELHO; SILVEIRA, 2008). É importante ressaltar que estas duas últimas técnicas (*scrubbing* e votação de CRCs) não protegem as memórias BRAMs (Bloco de memória RAM) da FPGA utilizadas pelo processador. Portanto, para o caso de FPGAs, as técnicas de proteção da *cache* e de proteção de configuração da FPGA são complementares. Para o caso de tecnologia CMOS, as técnicas de TPI e TPM podem ser aplicadas diretamente.

5.6 Síntese do JOP em Silício

Até a escrita desse trabalho, apenas foi detectada a implementação do JOP para FPGAs. Neste sentido, uma outra contribuição deste trabalho é elencar as modificações necessárias e/ou desejáveis, nos códigos fontes do JOP, para a prototipação do JOP em tecnologia CMOS. Esta contribuição adicional representa um grande passo na evolução desse processador.

5.6.1 Sistema de verificação funcional

Conforme visto no Capítulo 3, uma das etapas importantíssimas do fluxo de projeto de circuitos integrados é a verificação funcional do chip. Os códigos fontes do JOP incluem somente um *testbench* de simulação para execução do programa HelloWorld discutido na Seção 5.4.1. Neste sentido, para a prototipação do JOP em tecnologia CMOS, o primeiro passo é definir e implementar um sistema eficiente de verificação funcional. No Capítulo 3 é feita uma descrição sucinta da metodologia BVM *Brazil-IP Verification Methodology*, a qual poderia ser aplicada para o JOP.

5.6.2 Substituição das BRAMs

De acordo com o que foi visto nos Capítulos 3 e 4, a implementação do JOP em FPGA utiliza dois blocos de memória RAM disponíveis na FPGA, um para implementação do bloco *stack* e outro para implementação da memória *cache*. Além das duas memórias RAM, o JOP necessita de uma memória ROM para armazenamento da implementação e mapeamento de alguns *bytecodes* Java em micro-código. Através do Serviço Mosis (USC, 2010) ou mesmo em contato direto com o fabricante do chip, é possível conseguir os softwares geradores de memória para a tecnologia escolhida. Estes devem ser utilizados para gerar as memórias do JOP.

Os softwares geradores de memória, além de gerarem os blocos de memória, geram um modelo (escrito em uma HDL) do bloco de memória gerado. Estas memórias são, possivelmente, diferentes em termos de lógica de acesso e temporização, daquelas encontradas nas FPGAs. Assim, faz-se necessário modificar o JOP para que este trabalhe corretamente com as memórias da tecnologia na qual será fabricado. Para realização desta tarefa, o modelo da memória gerado pelo *software* será bastante útil.

5.6.3 Inicialização da *Stack* RAM

Para compreender a inicialização da *stack* RAM, deve-se lembrar que para um sistema implementado em FPGAs baseadas em SRAM, é necessário gravar o *bitstream* na FPGA sempre que o sistema inicializa. No caso do JOP, sua implementação se utiliza desta fase (inicialização) para, além de gravar a configuração da FPGA, inicializar a memória *stack* RAM do JOP. Neste sentido, para a implementação do JOP em tecnologia CMOS, faz-se necessário projetar um circuito lógico capaz de inicializar a memória *stack* RAM.

5.6.4 Adição de uma JTAG para gravação da Flash e RAM externas

A implementação do JOP em FPGA utiliza a interface JTAG, disponível na FPGA, para realizar o carregamento do programa, na memória de código do processador. No caso do processador em tecnologia CMOS, é necessário o desenvolvimento da interface JTAG, assim como a funcionalidade de carregamento do programa através desta interface.

5.6.5 Adicionar registradores de configuração dos periféricos

Processadores implementados em silício dispõem usualmente de vários registradores para configuração da funcionalidade de periféricos, tais como porta serial (USART - *Universal Synchronous Asynchronous Receiver Transmitter*) e temporizadores. A configuração dos periféricos do processador JOP em FPGA é realizada de forma fixa (ou *hard coded*) no próprio código VHDL. Isso é viável, pois, sistemas baseados em FPGA podem ser reconfigurados simplesmente através do carregamento de um novo *bitstream*. No entanto, para a prototipação do JOP em silício, recomenda-se a criação de registradores e circuitos que permitam a configuração dos periféricos do JOP por *software*.

5.6.6 Revisão do código RTL

Observa-se no código do JOP original em FPGA alguns erros clássicos de codificação de circuitos lógicos em FPGAs (CHAMBERS, 1997a, 1997b). Para ilustrar esta situação, toma-se como exemplo trechos do código onde o uso da estrutura de programação **case** da linguagem VHDL é utilizada, sem o devido cuidado de satisfazer todas as condições possíveis. Isto é inaceitável mesmo para circuitos implementados em FPGA, os quais podem ser facilmente reconfigurados. Em um circuito integrado, o problema agrava-se, podendo levar à remanufatura completa do circuito integrado. Portanto, recomenda-se uma revisão completa do circuito do JOP em busca de erros clássicos de codificação como o erro descrito acima.

5.6.7 Conclusão

Três configurações do JOP são mostradas na Figura 5.2: JOP Original, JOP Original modificado com TPI e JOP modificado com TPI e TPM. Como pode ser visto nesta Figura, em relação ao JOP original, a técnica de TPI adicionou três módulos de hardware. Em relação ao JOP modificado com a técnica de TPI, a técnica de TPM acrescenta um módulo de hardware, o verificador e corretor de métodos. Todos os módulos de *hardware* (exceto bloco de memória) e software necessários para implementação de ambas as técnicas foram codificados, simulados e testados em FPGA, tanto de forma isolada, como de forma integrada.

Para não modificar o ciclo de execução do JOP, os três *ip cores* (Codificador de Hamming, Decodificador de Hamming e CRC32) desenvolvidos neste trabalho usam apenas lógica combinacional, e portanto são massivamente paralelos.

Capítulo 6

Resultados

Neste Capítulo, são descritos os resultados dos testes de injeção de falhas realizados, tanto em nível de simulação como em FPGAs.

6.1 Resultados dos Testes de injeção de falhas em simulação

O JOP modificado pelo uso da técnicas de TPI e TPM (FT-JOP) foi simulado utilizando a ferramenta NC-VHDL para avaliar se houve degradação de seu funcionamento. Para avaliar a eficácia da técnica foi desenvolvido um módulo injetor de falhas escrito em VHDL, que seleciona aleatoriamente e inverte um *bit* de cada um dos *bytecodes* de um programa em execução pelo JOP. Para fins de avaliação, foi desenvolvido e executado um programa que conta de 1 a 65535, e envia o número atual da contagem pela porta serial do JOP. Então, foram inseridos erros nos *bytecodes* desse programa, e 100% destes foram automaticamente detectados e corrigidos.

6.2 Resultados da síntese física em FPGA

A ferramenta de software ISE da Xilinx foi usada para realizar a síntese de três configurações do processador JOP para a FPGA FX25 da família Virtex 4 da Xilinx. Utilizou-se as configurações *default* desta ferramenta. A Tabela 6.1 mostra uma comparação, em termos de frequência de operação, recursos de elementos lógicos (*slices*) e de memória RAM de 3 combinações do JOP original e das duas técnicas propostas.

Tabela 6.1: comparação entre o FT-JOP e o JOP original.

Configuração do JOP	FPGA Slices	RAM (Kbits)	Freq (MHz)
JOP Original	1762	16	130
JOP Original e TPI	1808	24	130
JOP original, TPI e TPM	1870	24	130

6.3 Descrição e Resultados dos Testes de injeção de falhas em FPGA

As técnicas propostas foram aplicadas ao JOP em FPGA, com o intuito de avaliar os recursos lógicos (portas lógicas) extras necessários, e testar o FT-JOP quando submetido a injeção de falhas. Como o objetivo do teste foi validar as técnicas de proteção de *cache*, foram inseridos erros diretamente na *cache* do processador e não na memória de configuração da FPGA. Portanto, nenhuma técnica de proteção da memória de configuração da FPGA foi embarcada para realização destes testes.

O primeiro ensaio realizado testa a técnica de TPI, aonde além do JOP modificado com a técnica de TPI, foi embarcado na FPGA, um circuito lógico que conecta o barramento de saída da memória *cache* aos pinos externos da FPGA, nos quais estão conectados DIP *Switches*. Desta forma, pode-se facilmente inserir erros permanentes do tipo *stuck-at* (1 ou 0) e erros temporários aleatórios, simplesmente mudando o estado das chaves.

Para todas as falhas inseridas nas quais apenas um *bit* do barramento é invertido, houve detecção e correção automática de 100% destes *bits* pelo uso da técnica de proteção de instruções. No caso de inversão de dois *bits*, não houve detecção ou correção e o sistema parou de responder aos comandos.

O segundo ensaio realizado testa a técnica de TPM. Neste caso, o código a ser executado pelo JOP foi intencionalmente modificado manualmente, utilizando um software editor de arquivos binários. Conforme esperado, o JOP não executou o método corrompido e desviou a execução para uma rotina de tratamento de exceção.

6.4 Discussão dos Resultados

6.4.1 Aumento de área de silício do JOP

Conforme pode ser visto na Tabela 6.1, a implementação da técnica de TPI utilizou 46 *slices* adicionais da FPGA. Isto representa apenas 2,61% do tamanho original do JOP. Conforme demonstrado pela simulação e pelos testes com o sistema embarcado na FPGA, a técnica mostrou-se eficaz, pois corrigiu em tempo real as falhas injetadas.

Para o JOP modificado utilizando ambas as técnicas de TPI e TPM, observa-se um aumento de 108 *slices* da FPGA em relação ao JOP original, ou seja, apenas 6,13%.

6.4.2 Eficácia dos testes

O FT-JOP, além de detectar e corrigir 100% dos erros de inversão de apenas um *bit* gerados por *single SEUs* na *cache*, é capaz de detectar que o código foi corrompido ainda quando estava na memória externa ao processador. O método corrompido pode ser recuperado a partir de uma outra memória, caso exista, mas é importante ressaltar que isto levaria o sistema a operar em um modo degradado, pois as condições de tempo real não poderiam ser mais garantidas.

Portanto, a injeção de falhas através de simulação e testes demonstraram a eficácia da proposta. No entanto, os testes realizados não simulam com perfeição um ambiente submetido a elevados níveis de radiação. Neste sentido, considera-se necessário avaliar o funcionamento do FT-JOP quando submetido ao bombardeamento de partículas altamente energizadas. Estes testes devem permitir avaliar a confiabilidade do processador e podem ser realizados, por exemplo, no LIN - Laboratório de Instrumentação Nuclear da UFRJ, assim que o FT-JOP for prototipado em silício.

6.4.3 Temporização

Importante ainda notar que, de acordo com a Tabela 6.1 a frequência atingida pela ferramenta de síntese foi a mesma. Além disso, não foram realizadas alterações no núcleo do JOP, e portanto, o ciclo de execução deste se mantém, bem como suas características de tempo real. Por fim, os blocos codificador de Hamming, decodificador de Hamming e CRC32 são completamente combinacionais, e portanto não incluem ciclos de máquina adicionais na execução de *bytecodes* pelo FT-JOP.

Capítulo 7

Conclusões e Trabalhos Futuros

Neste trabalho foram propostas uma técnica de proteção de instruções (TPI) e uma técnica de proteção de métodos (TPM) para aplicação no processador de tempo real JOP (*Java Optimized Processor*). Estas técnicas utilizam algoritmo de Hamming e CRC, respectivamente. Ambos os algoritmos foram implementados em hardware e de forma massivamente paralela para aumentar a confiabilidade do processador Java, enquanto mantém seu desempenho e sua característica de tempo real. A TPI adiciona quatro *bits* de Hamming, que são armazenados em conjunto com cada *bytecode* Java (palavra de 8 *bits*) do processador JOP. Estes quatro *bits* são usados para detectar e corrigir erros na memória de código. A TPM detecta a integridade de um método Java através do uso do *core* CRC32, especialmente desenvolvido para este propósito.

O *Soft IP Core* FTP-JOP foi desenvolvido a partir da aplicação das técnicas de TPI e TPM no JOP original. Para fins de verificação da eficácia das técnicas propostas e análise de recursos extras necessários, O FT-JOP foi implementado em uma FPGA Virtex 4 FX25. Foram realizados testes em FPGA e de simulação, usando um módulo de injeção de falhas desenvolvido neste trabalho, que demonstraram que a técnica de TPI corrige 100% dos erros simples (apenas um *bit* invertido) ocorridos nos *bytecodes* Java. Além disso, os testes também mostraram que a TPM é capaz de detectar se um método Java estar corrompido. Portanto, o FT-JOP é capaz de detectar e corrigir erros na memória de código. Isto permite manufaturar o FT-JOP (*Fault Tolerant Java Optimized Processor*) utilizando-se os mesmos processos de fabricação de silício que são utilizados para fabricar chips comerciais, ao invés de usar processos de fabricação específicos para chips tolerantes a radiação. Logo, reduz-se consideravelmente o custo por área de silício.

De acordo com a pesquisa bibliográfica realizada, não foi encontrado nenhum processador Java que tenha simultaneamente garantia de tempo real e de funcionamento.

Portanto, este trabalho é original no sentido de conceber, a partir do JOP, um processador Java de tempo real tolerante a falhas. Estas características são importantíssimas para sistemas embarcados de tempo real para aplicações que envolvem risco de vidas humanas.

Conforme argumenta-se no início do Capítulo 5, o bloco de memória *cache* ocupa uma área de 44,46% do processador JOP original, sendo esta área altamente suscetível a SEUs, daí a importância de se proteger a *cache* do JOP. No entanto, quando se compara o FT-JOP a outros processadores tolerantes a falhas (GAISLER, 2002, 1994; QUACH, 2000; CHECK; SLEGEL, 1999; COTA et al., 2001; MEINHARDT et al., 2009; S.; SUBRAMANIAN; SOMANI, 2006), fica claro que, em trabalhos futuros, algumas melhorias ainda podem ser feitas no JOP, no sentido de aumentar sua confiabilidade, tais como:

- proteger os registradores do processador contra SEU, como proposto em (GAISLER, 2002);
- proteger os *flip flops* do processador;
- proteger da unidade de execução contra SEU.

Referências Bibliográficas

ANDERSON, T.; LEE, P. A. *Fault Tolerance: Principles and Practice*. Upper Saddle River, NJ, USA: Prentice-Hall, 1981.

ATIENZA, D. et al. Reliability-aware design for nanometer-scale devices. In: *ASP-DAC '08: Proceedings of the 2008 Asia and South Pacific Design Automation Conference*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2008. p. 549–554. ISBN 978-1-4244-1922-7.

BAKER, J. *CMOS Circuit Design, Layout, and Simulation*. 2nd. ed. Somerset, NJ, USA: Wiley-IEEE Press, 2007.

BECK, A.; CAIRO, L. Low power Java processor for embedded applications. In: GLESNER, M. et al. (Ed.). *VLSI-SOC: From Systems to Chips*. Boston, MA, USA: Springer Boston, 2006, (IFIP International Federation for Information Processing, v. 200). p. 213–228.

BURNS, A.; WELLINGS, A. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Upper Saddle River, NJ, USA: Addison-Wesley, 2009.

BURSKY, D. Understanding the options lets you optimize system performance. *Electronic Design*, may 2004.

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.

CHAMBERS, P. *The Ten Commandments of Excellent Design*. USA, 1997.

_____. *The Ten Commandments of Excellent Design VHDL Code Examples*. USA, 1997.

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.

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.

DRECHSLER, R. *Advanced Formal Verification*. New York, NY, USA: Springer US, 2010. ISBN 1441954201.

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

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*. Austin, TX: IEEE Press, 1994. p. 128 –130.

_____. 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.

GANSSE, J.; BARR, M. *Embedded Systems Dictionary*. 1. ed. Gilroy, CA, USA: CMP Books, 2003. ISBN 978-1578201204.

HALFHILL, T. R. Imsys hedges bets on Java. *Microprocessor Report*, p. 1–4, August 2000.

HORGAN, J. *Computational Lithography*. Novembro 2008.

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.

JAVA COMMUNITY PROCESS. *Java Specification Request 302*. Setembro 2008. Disponível em: <<http://jcp.org/en/jsr/detail?id=302>>.

JOSHI, S. M.; DUBEY, P. K.; KAPLAN, M. A. *A new parallel algorithm for CRC generation*. June 2000. 1764–1768 p.

KASTENSMIDT, F. L.; CARRO, L.; REIS, R. *Fault-Tolerance Techniques for SRAM-based FPGAs*. 1. ed. New York, NY, USA: Springer, 2006.

KORSHOLM, S.; SCHOEBERL, M.; RAVN, A. P. Java interrupt handling. In: *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*. Orlando, Florida, USA: IEEE Computer Society, 2008.

KREUZINGER, J. et al. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, v. 27, n. 1, p. 19–31, 2003.

LIU, C.; LAYLAND, J. Scheduling algorithms for multiprogramming in hard real-time environment. *JACM*, p. 46–61, 1973.

MCLAUGHLIN, B.; POLLICE, G.; WEST, D. *Head First Object-Oriented Analysis and Design*. 1. ed. Sebastopol, CA, USA: O'Reilly, 2006.

MEINHARDT, C. et al. Recovery scheme for hardening system on programmable chips. In: *Test Workshop, 2009. LATW '09. 10th Latin American*. Buzios, RJ, Brasil: IEEE Press, 2009. p. 1–6.

MELCHER, E. U. K. et al. Silicon validated ip cores designed by the brazil ip network. In: *IP-SOC 2006 conference proceedings*. Grenoble, France: Design and Reuse, 2006. p. 437–441.

MIKHALENKO, P. *Real-Time Java: An Introduction*. September 2008. Disponível em: <<http://www.onjava.com/pub/a/onjava/2006/05/10/real-time-java-introduction.html?page=1>>.

MOORE, G. E. Cramming more components onto integrated circuits. *Electronics*, v. 38, n. 8, p. 114–117, April 1965.

NCSU. *North Carolina State University (NCSU) Cadence Design Kit*. 2010. Disponível em: <http://www.eda.ncsu.edu/wiki/NCSU_CDK>.

NICOLAIDIS, M. Graal: A new fault tolerant design paradigm for mitigating the flaws of deep nanometric technologies. In: *Proceedings of IEEE INTERNATIONAL TEST CONFERENCE*. Santa Clara, CA, USA: IEEE Press, 2007.

OLIVEIRA, H. F. de A. Dissertação de mestrado, *BVM: Reformulação da metodologia de verificação funcional VeriSC*. Campina Grande, PB, Brasil: UFCG, Junho 2010.

OSU. *Oklahoma State University Standard Cells*. 2010. Disponível em:

<<http://vcag.ecen.okstate.edu/projects/scells/>>.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. 4. ed. California, USA: Morgan Kaufmann, 2008.

PUFFITSCH, W.; SCHOEBERL, M. picoJava-II in an FPGA. In: *Proceedings of the 5th 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*. Vienna, Austria: ACM Press, 2007. p. 213–221.

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.

RABAEY, J.; POLLICE, G.; WEST, D. *Digital Integrated Circuits*. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

RAMOS, C. C. L. et al. *Leiaute do JOP em tecnologia CMOS - Projeto LESC/Brazil IP*. Fortaleza, CE, Brasil, Junho 2010.

RAZAVI, B. *Design of Analog CMOS Integrated Circuits*. 1. ed. Burr Ridge, IL, USA: McGraw-Hill, 2000.

ROBERTS, D.; KIM, N. S.; MUDGE, T. On-chip cache device limits and effective fault repair techniques in future nanoscale technology. *Microprocessors and Microsystems Journal*, p. 244–253, 2008.

S., G. T.; SUBRAMANIAN, V.; SOMANI, A. Seu mitigation techniques for microprocessor control logic. In: *EDCC '06: Proceedings of the Sixth European Dependable Computing Conference*. Washington, DC, USA: IEEE Computer Society, 2006. p. 77–86. ISBN 0-7695-2648-9.

SANTOS, F. et al. ipprocess: A usage of an ip-core development process to achieve time-to-market and quality assurance in a multi project environment. In: *Proceedings of IP 2008 - IP based System Design Conference*. Grenoble, France: Design and Reuse, 2008.

SCHOEBERL, M. JOP: A Java optimized processor. In: *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*. Catania, Italy: Springer, 2003. (LNCS, v. 2889), p. 346–359. ISBN 3-540-20494-6. ISSN 0302-9743.

_____. A time predictable instruction cache for a Java processor. In: *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*. Agia Napa, Cyprus: Springer, 2004. (LNCS, v. 3292), p. 371–382. ISSN 0302-9743.

SCHOEBERL, M. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. Tese (Phd Thesis) — Vienna University of Technology, 2005.

_____. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 2008.

_____. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. Scotts Valley, CA, USA: CreateSpace, 2009. ISBN 1438239696.

_____. Scheduling of hard real-time garbage collection. *Real-Time Systems*, accepted, 2010. Disponível em: <<http://www.jopdesign.com/doc/hrtsgc.pdf>>.

SCHOEBERL, M. et al. A hardware abstraction layer in java. *Trans. on Embedded Computing Sys.*, ACM, 2010.

SCHOEBERL, M.; PUFFITSCH, W. Non-blocking real-time garbage collection. *Trans. on Embedded Computing Sys.*, ACM, 2010.

SCHOEBERL, M. et al. Worst-case execution time analysis for a java processor. *Software: Practice and Experience*, v. 40/6, p. 507–542, 2010.

_____. A profile for safety critical Java. In: *10th IEEE International Symposium on Object And Component-Oriented Real-Time Distributed Computing (ISORC'07)*. Santorini Island, Greece: IEEE Computer Society, 2007. p. 94–101. ISBN 0-7695-2765-5.

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.

SPRACHMANN, M. Automatic generation of parallel crc circuits. *IEEE Design and Test of Computers*, v. 18, p. 108–114, May-June 2001.

TANENBAUM, A. S. *Modern Operating Systems*. 3. ed. Upper Saddle River, NJ, USA: Pearson Education, 2008.

TIMESYS. *The Concise Handbook Of Real-Time Systems*. 1.3. ed. Pittsburgh, PA, USA, 2002.

USC. *The Mosis Service*. 2010. Disponível em: <<http://www.mosis.com>>.

VTVT. *Virgina Technology VLSI for Telecommunications Standard Cells*. 2010. Disponível em: <<http://www.vtvt.ece.vt.edu/vlsidesign/cell.php>>.

WESTE, N.; HARRIS, D. M. *CMOS VLSI Design: A Circuits and Systems Perspective*. 3. ed. Upper Saddle River, NJ, USA: Addison Wesley, 2004.

Anexo 1 - Publicações Relacionadas a Esta Dissertação

Técnica de Proteção de Bytecodes para Processador Java em Tecnologia CMOS

Jardel Silveira (jardel@lesc.ufc.br), David Viana (david@lesc.ufc.br)
Helano Castro (helano@lesc.ufc.br), Alexandre Coelho (alexandre@lesc.ufc.br)
Jarbas Silveira (jarbas@lesc.ufc.br)
Universidade Federal do Ceará
LESC (<http://www.lesc.ufc.br>)
Fortaleza, CE, Brasil

Resumo

O soft core JOP (Java Optimized Processor) para FPGAs (Field Programmable Gate Array) é uma implementação otimizada da máquina virtual Java em hardware, para aplicações de tempo real. No entanto, este processador não contempla em sua arquitetura técnicas de tolerância a falhas. O trabalho descrito neste artigo é parte de um esforço maior para tornar o processador JOP um processador tolerante a falhas. Neste artigo, apresentamos os resultados da aplicação de uma técnica de tolerância a falhas, proteção de memória através de ECC (Error Correction Code), no soft core JOP, que detecta e corrige erros na área destinada ao código da memória SRAM (Static Random Access Memory). A ocorrência da falha é percebida no nível sistêmico através de uma exceção, característica esta disponível na linguagem Java. Este artigo apresenta resultados inovadores na medida em que não existem registrados na literatura outro processador Java de tempo real e tolerante a falhas.

1 Introdução

Os sistemas eletrônicos de tempo real embarcados em missões espaciais estão sujeitos aos elevados níveis de radiação existentes no espaço. Por isso, tais sistemas estão sujeitos a falhas causadas pelas colisões de partículas altamente energizadas contra estruturas nanométricas de silício presentes nos circuitos integrados modernos.

Particularmente para circuitos integrados contendo blocos de memória SRAM, a principal consequência destas colisões é a ocorrência de SEUs (*single event upsets*), que são a inversão permanente do valor de um bit. Existem fábricas especializadas na produção de circuitos integrados tolerantes à radiação. No entanto, devido ao pequeno volume de produção desses circuitos integrados, os mesmos têm preços elevados. Portanto, é muito importante garantir

tolerância à radiação no âmbito do projeto do circuito integrado, independentemente do processo de fabricação, pois isto reduz os custos do sistema e permite utilizar os mais modernos processos de fabricação existentes.

Devido ao tamanho reduzido e a alta frequência de operação dos circuitos eletrônicos digitais modernos, os mesmos estão cada vez mais suscetíveis a ruído. Por isso, problemas antes somente encontrados em sistemas submetidos a radiações com intensidade similar à espacial, hoje são enfrentados em sistemas operando em ambiente terrestre. Aplicações seguras constituem outro exemplo em ambiente terrestre que requerem técnicas de tolerância a falhas, pois falhas em sistemas seguros podem ser exploradas por *crackers* para descobrir chaves secretas armazenadas na memória interna de um circuito integrado [13].

Diante desse cenário, percebe-se cada vez mais a necessidade da utilização de técnicas de tolerância a falhas não somente em sistemas embarcados em missões espaciais, mas também nos sistemas terrestres. Dentre alguns exemplos dessas aplicações terrestres podemos citar a indústria automobilística, bancos e outras aplicações cujos requisitos temporais e de alta disponibilidade são prioritários para o correto funcionamento do sistema.

Notavelmente, a linguagem de programação C é atualmente a mais utilizada para desenvolvimento de *software* para sistemas embarcados, tanto para o sistema operacional quanto para a aplicação. Isto pode ser facilmente demonstrado por uma análise dos compiladores comerciais disponíveis para os processadores modernos de sistemas embarcados.

Usualmente, o sistema operacional é responsável por funções de suporte a tempo real, gerenciamento de memória e comunicação inter-processos. Tais recursos são disponibilizados para a aplicação por meio de chamadas de sistema (*system calls*) [21] e de uma API (*Application Program Interface*).

O uso de uma linguagem de alto nível de abstração traz benefícios do ponto de vista do desenvolvimento do

sistema, tais como diminuir a probabilidade de erros de codificação e a redução do tempo de desenvolvimento de um sistema [2]. Java é uma linguagem de alto nível muito utilizada e com grande suporte para o desenvolvimento de sistemas *standalone*. Além do alto nível de abstração, a linguagem Java traz no seu núcleo (Máquina Virtual Java) recursos comumente implementados em nível de sistema operacional, tais como comunicação inter-processo e escalonamento de tarefas.

Em uma implementação tradicional de sistemas embarcados de tempo real, baseada em um processador de uso geral e um sistema operacional de tempo real, essas vantagens têm um custo elevado em termos de recursos computacionais, o que é incompatível com as severas restrições de recursos computacionais em sistemas embarcados. Esta incompatibilidade pode ser resolvida pelo uso de um processador específico para linguagem Java, como proposto por Schoeberl em [19] e vários outros[1][9][10][15]. No entanto, para nenhum destes processadores, esses autores discutem a garantia de funcionamento (*dependability*) do processador. Quando comparado com os demais processadores Java, o JOP [19] se destaca, por exemplo, em relação a sua característica de tempo real, porém também desconsiderando requisitos de garantia de funcionamento. Por isso, escolhemos o processador JOP como plataforma base para este trabalho.. Note que existem outros processadores de uso geral tolerantes a falhas [8][7][16][4][6][11], mas nenhum deles é um processador Java tolerante a falhas e de tempo real.

Neste trabalho, propomos uma técnica de tolerância a falhas para proteger a memória cache SRAM de código interna ao JOP contra SEUs. Não se pretende com a aplicação desta técnica esgotar a discussão sobre tolerância a falhas no JOP, no entanto, por proteger uma região extremamente crítica, no caso a memória de código, e que não é viável de ser protegida via técnicas de software, consideramos a mesma de extrema importância. Informações sobre o JOP necessárias para a compreensão do problema, bem como os detalhes desta e os resultados obtidos são apresentados nas seções que se seguem.

2 Processador JOP

Um processador Java é uma implementação da máquina virtual Java. Essa implementação não é necessariamente completa em *hardware*, pois uma Máquina Virtual Java contém funções complexas como, por exemplo, escalonamento, gerenciamento e intercomunicação de processos. O custo de implementar todos esses recursos em *hardware* pode tornar a implementação não viável. Portanto, o conceito de um processador Java difere de um processador comum, no qual apenas elementos de *hardware* estão envolvidos. Dessa forma, um processador Java é uma

implementação baseada em *hardware* e, possivelmente, em algum *software*.

Para um processador Java de tempo real, sua característica de tempo real deve permear tanto o seu *software*, como o seu *hardware*.

O JOP (Java Optimized Processor) é uma implementação em *hardware* e *software* de uma máquina virtual Java de tempo real, baseada no perfil J2ME (*Java 2 Micro Edition*), CLDC (*Connected Limited Device Configuration*) e na especificação SCJ (*Safety Critical Java*). Este processador é implementado como *soft core* em FPGAs Xilinx ou Altera e, diferentemente da JVM (*Java Virtual Machine*), que é uma máquina CISC (*Complex Instruction Set Computer*) [14], o JOP é, internamente, uma máquina RISC (*Reduced Instruction Set Computer*) [14] e contém seu próprio conjunto de instruções.

2.1 Requisitos de tempo real

Aplicações de tempo real para o JOP são explicitamente separadas em duas partes: Fase de Inicialização e Fase de Missão. Na fase de inicialização são criados todos os objetos que serão usados durante toda a execução da aplicação e, portanto, áreas de memórias são alocadas e inicializadas. Nesta fase não existe garantia de tempo real. Na fase de missão, as *threads* são executadas concorrentemente de acordo com o algoritmo de escalonamento.

2.1.1 Análise de WCET no JOP

Por ter sido desenvolvida para ser usada em sistemas embarcados com aplicações de tempo real, a arquitetura do processador JOP permite calcular com facilidade o WCET (*Worst Case Execution Time*) de uma tarefa.

A máquina virtual Java do JOP implementa classes que permitem desenvolver aplicações de tempo real. Essas classes não são compatíveis com o padrão RTSJ (*Real Time Specification for Java*) [12], pois apenas um subconjunto deste padrão é implementado. Embora as áreas de código e de pilha do JOP utilizem memória de *cache*, o modelamento do comportamento desta no JOP é perfeitamente previsível no tempo. Pois, diferentemente do que acontece em outros processadores, no JOP não ocorrem “*cache misses*”, ou seja, cada instrução solicitada pelo processador à *cache* estará sempre previamente carregada na *cache* [17].

2.1.2 Garantia de funcionamento

O JOP não implementa em sua arquitetura técnicas de garantia de funcionamento. Portanto, para aplicações de tempo real do tipo *hard*, ou seja, que envolvem risco para vidas humanas, o projetista do sistema deverá assegurar a garantia de funcionamento em nível sistêmico. No JOP, uma falha de *hardware*, por exemplo um *opcode* ilegal ou

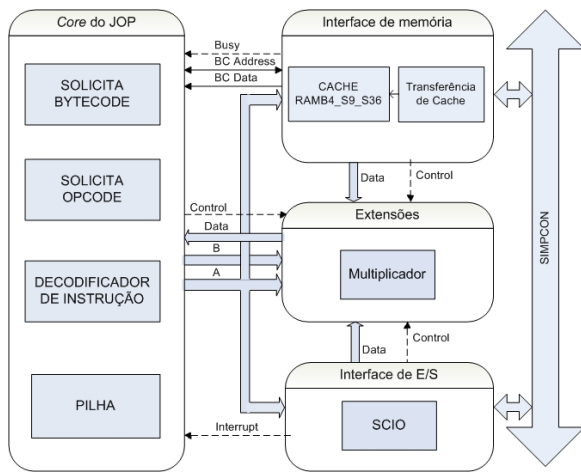


Figura 1. Diagrama de Blocos do JOP

um erro de paridade de memória, levará o sistema a um shutdown [18].

2.2 Arquitetura do JOP

O JOP é composto de quatro blocos principais (ver Figura 1): Interface de memória, *core do JOP*, interface de E/S (*scio*) e “extensões”. O bloco de interface de memória comunica-se com os controladores de memórias externas através do barramento de comunicação **simpcon**. Os controladores de memória, por sua vez, comunicam-se, através dos pinos do processador com as memórias SRAM e *flash*. O bloco “*core do JOP*” é responsável por decodificar e executar os *bytecodes* fornecidos pela interface de memória e comandar as demais partes do processador. O bloco interface de E/S comunica-se com controladores de E/S, tais como porta USB e Serial RS232, através do barramento **simpcon**. Estes controladores de E/S, por sua vez, comunicam-se com os dispositivos do ambiente externo através dos pinos do processador. O bloco de “extensões” serve para agregar funções de co-processadores matemáticos sem realizar modificações no núcleo do processador.

2.3 Documentação e portabilidade

Além das características técnicas do JOP, descritas anteriormente, podemos destacar também a ampla documentação disponível, a sua portabilidade, pois já foi implementado em diversas placas de desenvolvimento de FPGA (Xilinx e Altera) disponíveis no mercado, e por último, mas não menos importante, a disponibilidade do código fonte do JOP para *download* pelo site <http://www.opencores.org> e licenciado sob a GPL (Gnu Public License) versão 3. Atualmente o JOP

é utilizado em dois sistemas comerciais, tendo um deles a característica de tempo real, e em vários sistemas de pesquisa [19]. Portanto, o JOP se posiciona como uma excelente alternativa para plataforma base, de pesquisa e desenvolvimento, de novas técnicas de sistemas de tempo real.

3 JOP Tolerante a Falhas

Erros na memória de código de um sistema computacional são críticos por serem armazenados permanentemente e podem, portanto, causar sucessivos erros no processo de computação que fizer uso dos dados errôneos. Para detectar e corrigir erros na memória de dados, existem técnicas bastante efetivas implementadas em *software*. No entanto, as técnicas aplicadas no âmbito de *software* para detectar e corrigir erros na memória de código não são eficazes, pois o próprio *software* pode estar corrompido. Nesse sentido, propomos o uso de uma técnica no âmbito de *hardware* para detectar e corrigir erros nas memórias RAM interna (*cache*) ao processador JOP, de forma a aumentar a confiabilidade do sistema de *cache* de métodos descritos na seção anterior.

Durante a inicialização do processador JOP, todo o código é transferido da memória *flash* para a memória RAM. Ao fim da transferência de código para a RAM, o sistema de *cache* transferirá, sob demanda, métodos inteiros da memória externa para a memória de *cache* interna. Por último, após o método estar completamente carregado na memória interna, o *core* irá solicitar e executar os *bytecodes*, um a um.

A Figura 2 mostra o diagrama de blocos do JOP modificado. Quando comparado ao diagrama de blocos do JOP original (ver Figura 1), nota-se que três novos blocos foram adicionados ao bloco de interface de memória da arquitetura original do JOP. Estes blocos se referem à implementação da técnica de proteção de instruções:

1. Codificador de Hamming;
2. Detector e corretor de erros em *bytecodes* (Decodificador de Hamming);
3. Redundância de cache (RAMB4_S4_S18);

3.1 Técnica de proteção de instruções

Esta técnica detecta e corrige erros ocorridos nos *bytecodes*, desde o armazenamento destes na *cache* até o início da execução destes pelo *core* do JOP. O erro é detectado e corrigido imediatamente antes de o *core* iniciar a execução do *bytecode*. Portanto, essa é uma técnica de verificação de último instante (*last minute check*)[5].

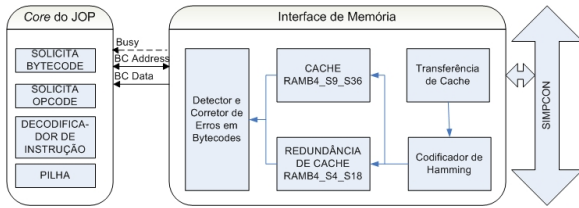


Figura 2. Diagrama de Blocos do JOP modificado

Simultaneamente à escrita de um *bytecode* (tamanho de 8 bits) na memória *cache*, 4 bits extras de redundância (bits de Hamming) são calculados e armazenados em uma memória *cache* de redundância (ver Figura 2), que se posiciona em paralelo com a *cache* original. Esses bits extras são calculados por um *core* escrito em RTL (*Register Transfer Level*), que implementa um codificador de Hamming.

Imediatamente após a memória *cache* fornecer um *bytecode* para o *core* do JOP, porém antes de ser executado, um *core* decodificador de Hamming lê os 4 bits armazenados na *cache* de redundância (Ver Figura 2). Com base nesses 12 (doze) bits (oito bits do *bytecode* mais quatro bits de Hamming), esse *core* irá verificar se houve alguma inversão de bit em algum dos bits do *bytecode*. Em caso afirmativo, isto significa que houve uma falha. Neste caso, o *core* decodificador de Hamming irá corrigir automaticamente o *bytecode*, desde que apenas um bit tenha sido invertido. Finalmente, o *bytecode* correto será entregue para a execução por parte do *core* do JOP.

3.2 Percepção da falha no âmbito sistêmico

Uma falha de *hardware* no JOP original, como por exemplo, um *opcode* ilegal, leva o sistema a um *shutdown* [18]. Neste trabalho, o processador foi modificado para que, na ocorrência de uma falha de *hardware*, uma exceção seja gerada.

4 Resultados

O JOP modificado pelo uso da técnica de proteção de instruções foi simulado utilizando a ferramenta NC-VHDL para avaliar se houve regressão de seu funcionamento. Para avaliar a eficácia da técnica foi desenvolvido um módulo injetor de falhas escrito em VHDL. Esse módulo gera aleatoriamente eventos do tipo SEU nos BYTECODES de um programa em execução pelo JOP, sendo que não deve existir mais que um dos bits invertidos em um mesmo BYTECODE. A técnica mostrou-se eficaz, tendo em vista que todos os bytecodes corrompidos com somente um bit inver-

Tabela 1. Tabela Comparativa entre o JOP Modificado e o JOP Original

	FPGA Slices	RAM (Kbits)	Freq (MHz)
JOP	3150	16	100
JOPFT	3201	24	100

tido, foram automaticamente corrigidos. Para aqueles com dois bits invertidos, uma exceção foi gerada para ser tratada por rotina específica para este fim. Além da simulação, o JOP modificado foi embarcado na placa Spartan-3 starter kit board [22] da Digilent. Essa placa contém uma FPGA Spartan 2 XC3S200 e 1MB de memória SRAM. Neste caso, a injeção de falhas foi realizada através de mudanças do nível lógico dos pinos da FPGA. Dois tipos de falhas foram injetadas: stuck-at (1 ou 0) e inversão de bits aleatórios. Para todas as falhas inseridas, de um tipo ou de outro, houve detecção e correção automática pelo uso da técnica de proteção de instruções. Em termos numéricos, a possibilidade de detecção de falhas pode ser facilmente calculada com base no algoritmo de hamming aplicado a 8 bits de dados úteis e 4 bits de redundância [20].

A Tabela 1 compara a área de FPGA, em termos de *slices* e de memória RAM para o JOP original e o JOP modificado (JOPFT) pela técnica proposta.

5 Conclusão

De acordo com a pesquisa bibliográfica realizada, não foi encontrado nenhum processador Java que tenha simultaneamente garantia de tempo real e de funcionamento. Portanto, este trabalho é único no sentido de conceber, a partir do JOP, um processador Java de tempo real tolerante a falhas. Estas características são importantíssimas para sistemas embarcados de tempo real para aplicações que envolvem risco de vidas humanas.

A técnica proposta utiliza algoritmo de hamming implementado em hardware para aumentar a confiabilidade do processador Java. 4 bits de Hamming são armazenados em conjunto com cada *bytecode* Java (palavra de 8 bits) do processador JOP. Estes 4 bits são usados para detectar e corrigir erros na memória de código. Desta forma, agregamos ao processador JOP a capacidade de detectar e corrigir erros na memória de código. Isto permite manufaturar o JOPFT utilizando-se os mesmos processos de fabricação de silício, que são utilizados para fabricar chips comerciais, ao invés de usar processos de fabricação específicos para chips tolerantes a radiação. Logo, reduz-se drasticamente o custo por área de silício.

Finalmente é importante destacar que proteger a

memória de código do JOP é um passo importante para aumentar sua confiabilidade. Outras técnicas, como a técnica de votação de CRC de frames, anteriormente proposta pelos autores em [3] e também as técnicas propostas em [8] estão sendo aplicadas ao JOP e avaliadas em termos de confiabilidade, custo adicional em termos de área de silício, desempenho e implicações na características de tempo real do JOP, assim como foi feito para a técnica descrita neste trabalho. Portanto, como trabalho futuro, JOP modificado será testado com esta e outras técnicas de tolerância a falhas. Destacamos ainda que esse trabalho foi aprovado para ser manufaturado na tecnologia CMOS IBM 130 nm gratuitamente no service Mosis. Portanto, o JOP modificado será prototipado no processo IBM 130 nm, para então ser submetido a testes de funcionamento quando sob bombardeamento de partículas altamente energizadas. Estes testes permitirão avaliar a confiabilidade do processador e poderão ser realizados, por exemplo, no LIN - Laboratório de Instrumentação Nuclear da UFRJ.

Agradecimentos

Os autores gostariam de agradecer a Funcap pelo suporte financeiro através do seu programa de bolsas de mestrado (Processo BMD0008-00052.01.05). Agradecemos também o Ministério da Ciência e Tecnologia (MCT) do Brasil, a Sociedade Brasileira de Microeletrônica (SBMICRO), a Cadence e a Anacom que viabilizaram o programa universitário da Cadence no Brasil. Agradecer ainda a Xilinx pelo apoio fornecido através de seu programa universitário. Agradecer também o Mosis, IBM e ARM que viabilizarão a manufatura do circuito integrado fruto deste trabalho. E ainda o DETI, FCPC, LESC, Flextronics Institute of Technology (FIT) e Flextronics por apoiarem este trabalho. E por último, mas não menos importante, a Martin Schoeberl por distribuir livremente o código fonte do JOP sob a licença GPL.

Referências

- [1] A. C. Beck and L. Carro. Low power java processor for embedded applications. In *Proceedings of the 12th IFIP International Conference on Very Large Scale Integration*, pages 213–228, December 2003.
- [2] A. Burns and A. Wellings. *Real-Time Systems and Their Programming Languages*. Addison-Wesley, 1991.
- [3] H. Castro, A. A. Coelho, and R. J. Silveira. Fault-tolerance in fpga's through crc voting. In *SBCCI '08: Proceedings of the 21st Annual Symposium on Integrated Circuits And System Design*, pages 188–192, New York, NY, USA, 2008. ACM.
- [4] M. A. Check and T. J. Slegel. Custom s/390 g5 and g6 microprocessors. *IBM J. RES. DEVELOP*, 43(5/6):671–680, SEPTEMBER/NOVEMBER 1999.
- [5] H. de Sousa Castro. *Fault Tolerance Through Reconfigurability: Applications In Space Instrumentation*. Phd thesis, The University of Sussex, June 1992.
- [6] Érika Cota, F. Lima, S. Rezgui, L. Carro, R. Velazco, M. Lubaszewski, and R. Reis. Synthesis of an 8051-like micro-controller tolerant to transient faults. *J. Electron. Test.*, 17(2):149–161, 2001.
- [7] J. Gaisler. 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*, pages 128–130, June 1994.
- [8] J. Gaisler. A portable and fault-tolerant microprocessor based on the sparv8 architecture. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 409–415, Washington, DC, USA, 2002. IEEE Computer Society.
- [9] T. R. Halfhill. Imsys hedges bets on Java. *Microprocessor Report*, pages 1–4, August 2000.
- [10] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.
- [11] C. Meinhardt, R. Reis, M. Violante, and M. Reorda. Recovery scheme for hardening system on programmable chips. In *Test Workshop, 2009. LATW '09. 10th Latin American*, pages 1–6, March 2009.
- [12] P. Mikhaleenko. Real-time java: An introduction, September 2008.
- [13] C. R. Moratelli, E. Cota, and M. S. Lubaszewski. A cryptography core tolerant to dfa fault attacks. *Journal Integrated Circuits and Systems*, 2(1):14–21, 2007.
- [14] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 3 edition, 2007.
- [15] W. Puffitsch and M. Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 213–221, Vienna, Austria, September 2007. ACM Press.
- [16] N. Quach. High availability and reliability in the itanium processor. *IEEE Micro*, 20(5):61–69, 2000.
- [17] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. Phd thesis, Vienna University of Technology, 2005.
- [18] M. Schoeberl. *Jop Reference Handbook*. 1 edition, 2007.
- [19] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, doi:10.1016/j.sysarc.2007.06.001, 2008.
- [20] M. L. Shooman. *Reliability of Computer Systems and Networks*. Wiley, 1 edition, 2002.
- [21] A. S. Tanenbaum. *Modern Operating Systems*. Pearson Education, 3 edition, 2008.
- [22] Xilinx, Inc. *Spartan-3 Starter Kit Board User Guide*, May 2005. v1.2.

Fault-Tolerance in FPGA's through CRC Voting

Helano Castro
LESC - UFC
Campus do Pici, SN B 723
Fortaleza CE Brazil
(+55) 85 3366 9608
helano@lesc.ufc.br

Alexandre Coelho
LESC - UFC
Campus do Pici, SN B 723
Fortaleza CE Brazil
(+55) 85 3366 9608
alexandre@lesc.ufc.br

Jardel Silveira
LESC - UFC
Campus do Pici, SN B 723
Fortaleza CE Brazil
(+55) 85 3366 9608
jardel@lesc.ufc.br

ABSTRACT

The use of FPGA's for implementing fault-tolerant systems (FTS) has been widely discussed. Many FTS's have been proposed in this context and TMR is by far the most used architecture. However, those implementations have to count on the memory configuration's integrity of those FPGA's, since all the TMR's circuitry is stored into it. In fact, radiation or even electromagnetic noise can disturb the content of the configuration memory, with disastrous results for the system. In this paper we propose a way of dealing with this problem by using the FPGA's CRC as its signature. In case of an error derived from a kind of fault mentioned above, that signature will change. By voting those signatures in TRM architectures we can not only detect the faults, but we can recover from them by copying the memory configuration of a faultless FPGA into a faulty one. We discuss the difficulties of implementing this technique and the workarounds used to get over those difficulties. Finally we implement an experiment to validate the idea.

Categories and Subject Descriptors

B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance.

General Terms

Measurement, Reliability, Experimentation, Theory, Design, Verification.

Keywords

Cyclic redundancy check, fault tolerance, FPGA, partial reconfiguration.

1. INTRODUCTION

Critical applications need very reliable computer systems (CS) in order to perform the specified functions should a fault occurs. This problem is somewhat more important because many of those applications have very strict timing requirements and because of that they are known as Real-Time Applications. In order to meet

these requirements, the CS's must have reliability's levels very high. One way of enhancing the CS's reliability is to provide fault-tolerance, which can be achieved by hardware and software replication (temporal replication is also used).

Hardware replication has become very attractive because its cost has decreased dramatically over the last decades, especially when compared to software's cost. However, this approach has its drawbacks. Nowadays, many applications are performed in a special class of systems known as Embedded Systems (ES). ES's have an embedded computer system that is used to help to carry out the actions needed by the application. As ES are getting smaller and smaller, the restrictions concerned with size, weight and power are getting very strict. As a result, redundancy has to be thought very carefully so that those restrictions are not violated.

On the other hand, the use of FPGA's in fault tolerant systems has been much appealing, particularly when use of reconfigurable FPGA's[1][2] is made. In fact, once is possible to replicate processing units inside a single FPGA, the restrictions aforementioned could have more easily met. However, the use of FPGA's brings another challenger to the designer. SRAM-based FPGA's[3] are very susceptible to radiation and electromagnetic noise, what can cause *single-event upsets* (SEU) in the *configuration RAM*[4]. The effects of such hazard on FPGS's are different from those occurring in RAM found in usual computer systems. On FPGA-based designs, such a fault can provoke a bit inversion on its configuration memory, with reflection on the circuit design implemented in it. Worse still, many modern FPGA's do not support random bitstreams, therefore a bit inversion could cause an internal contention, connecting directly different logic's levels (0 and 1). This problem has to be managed when implementing fault-tolerant architectures in a FPGA.

A much well-known fault-tolerant architecture is a TMR (Triple Modular Redundancy) setup. Takahara et al [5] have proposed a TRM that uses instances of Xilinx's *soft core* 32 bits RISC Microblaze, with the voter implemented in SDRAM. Carmichael et al [6][7] describes a technique that can be used to correctly implement a TMR in a single FPGA. Useful as those systems can be, the problem with faults aforementioned has still to be dealt. In this paper we propose a way of dealing with this problem by voting the CRC's of the redundant modules in the FPGA. We also describe the difficulties of implementing the idea on some FPGA's due to unavailable inside information from the manufacturers. Besides we propose and describe the implementation of a setup meant to replicate the problem and we validate the proposal through an experiment.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SBCCI'08, September 1-4, 2008, Gramado, Brazil.

Copyright 2008 ACM 978-1-60558-231-3/08/09...\$5.00.

2. USING CRC FOR FAULT DETECTION IN FPGA

Figure 1 represents a 10-blocks (frames) FPGA where each block can be reprogrammed independently. Each block B_n corresponds to a FPGA's programming unit. The block named *content refresh* rewrites periodically the content of each column B_n , as well as it computes periodically its CRC (Cyclic Redundancy Code). The CRC value of block B_n is stored into a register, named CRC_B_n (where $n = 0 \dots 9$), associated with that block, as is shown in figure 1. Thus, each CRC_B_n contains the CRC of block B_n .

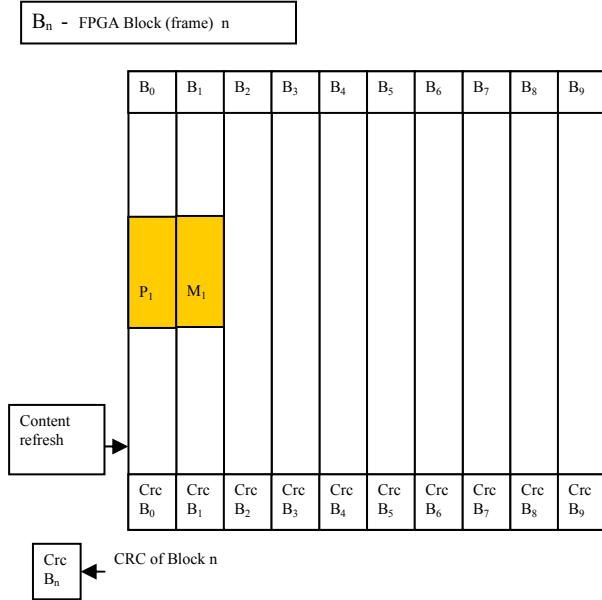


Figure 1: A Simplex system implemented in a FPGA.

Now suppose we wish to implement a computer, hereafter named C_1 , on that FPGA, and that two blocks are used for that purpose: P_1 is used to implement the processor, and M_1 is used to implement the memory. As it can be seen in figure 1, B_0 contains P_1 , and B_1 contains M_1 . Correspondently, CRC_B_0 contains P_1 's CRC and CRC_B_1 contains M_1 's CRC. Therefore CRC_0 represents the CRC of the bits that make up P_1 , while CRC_1 represents the CRC of bits that make up M_1 . As C_1 has only one P_1 and one M_1 , we call it a Simplex System.

Suppose now that one decides (e.g. in order to increase the fault tolerance of the simplex system C_1) to implement a TRM (Triple Redundancy Module) by replicating C_1 in the same FPGA shown in figure 1. Figure 2 shows that scenario, where C_2 (composed by P_2 and M_2) and C_3 (composed by P_3 and M_3) were added to the original project. Note that all units (P 's and M 's) were implemented in disjoint blocks (that restriction will be raised later). A Voter V , used to vote the results produced by the redundant modules, can also be seen in this figure. As the CRC of one P_n processor can change while the CRC of another one is being read, the voter should stop the clock of all processors P_n , calculate the CRC of each P_n , and carry out the voting process. In this paper, we propose to use a Xilinx Virtex II Pro FPGA in

order to use, as a voter, the softcore MicroBlaze Processor. As explained further on, that would make it easier to implement our proposal.

The bits used to implement each C_n block in the TRM system, before it is placed into use, are called the *hardware context*. Usually, the hardware context should not change, unless a fault should occur. In this case, a change in its pattern related to that context would occur. Since this pattern is static, we call it hardware context. When the FPGA is powered and the computers C 's start executing, the computational state of each C changes as a result of its register's values being modified. This change of state resulting of the computation along the time, we call it *software context*, since the pattern of the bits that represents this context is dynamic.

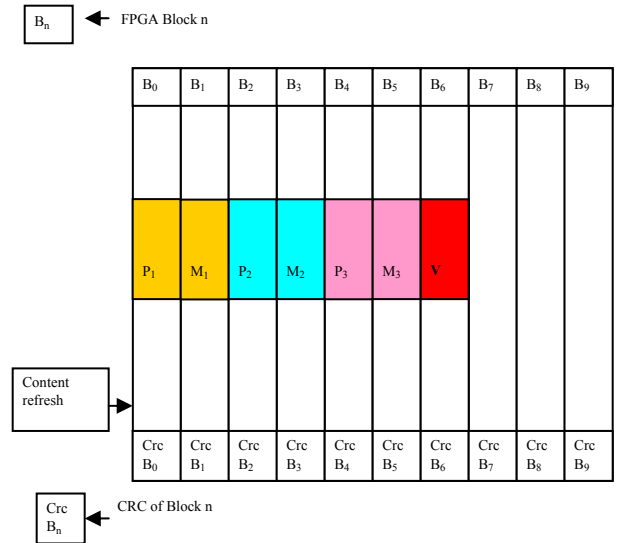


Figure 2: A TMR system implemented in a FPGA.

The CRC of each block B_n would change as a result of both changes in the hardware and software contexts. In another words, CRC changes would result either due to faults or due the evolution of the computation in each C . In a faultless scenario, the CRC of each computer C_n (which is a combination of the CRC of P_n and M_n) should be the same in each C_n that make up the TMR system. Should any C module present a different CRC than the other two modules, this could indicate the presence of a fault in that C module. In our proposal, each module should send the result of its computation to the voter (in our case, the MicroBlaze). The Microblaze[8][9][10] stops the processors' clock and calculates the CRC of each C module. The reason for stopping the clock is to make sure that the vote acquires coherent CRCs, in relation to each C module. After that, the Microblaze carries out the voting process. However, the voting is taken place not on the results furnished by the computers C 's, but on the CRC's of those computers. As explained before, those CRC's are acquired by the Microblaze from the blocks' CRC_B_n 's. If those values happen to be the same, it means that not only no fault occurred with any C hardware (because the hardware context did not change), but the results of the computation for all three

modules are the same as well (software status did not change). In that case, the voter should use as the result of the TMR computation any module's result. Should one C module produce a different result, the majority vote would be the value used by the Voter as the TMR's result of the voting process, and the result of the agreeing modules (with the same CRC's) should be used as the result of the computation. As the FPGA has two MicroBlaze processors, redundancy at the Voter's level would be possible. Figure 3 shows the voting process.

The CRC Fault Detection approach is able to detect any kind of fault, both those occurring in the computer's core and those affecting its computation. In addition, this approach makes it possible to tell if the problem occurred either in the block M_n or block P_n of the processor C_n , providing that these two units are implement in disjoint blocks B_n 's. In case this is not necessary, we can alleviate the initial restriction for these units to be implemented in disjoint blocks.

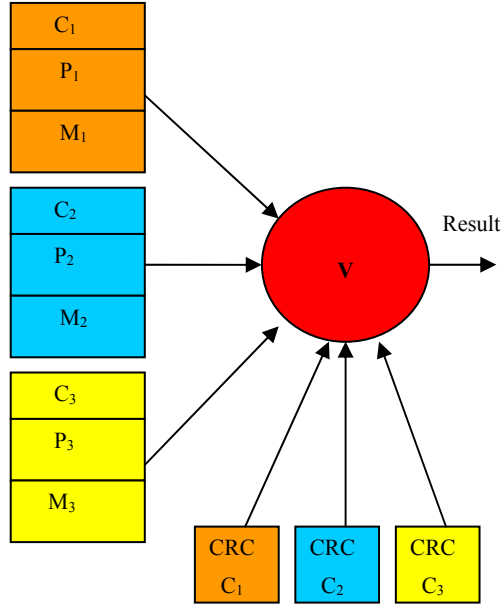


Figure 3: CRC's Voting Approach.

In order to implement our proposal, some obstacles have to be removed. In this session we will explain each one of them as well as the way to work around the problems. As we are working with Xilinx Virtex II Pro[11] FPGAs, we will describe the difficulties of implementing our proposal on them. In the first place, Xilinx does not recommend read-back operations on memory implementations in its FPGA's. As a result, that approach should be used with reservation on modules M 's of figure 2. As a work-around on this issue, a CRC calculation on the contents of those blocks could be made, as usually is done with memory contents. The second obstacle is that the designer has to make sure that each module fits in its block, with no external connections been routed through it. Xilinx routing tool provided to developers does not obey restrictions related to areas where the signals are routed. SEDCOLE et al [12][13] made use of a routing tool that does that job, but that tool was designed to be used only by Xilinx

designers or people authorized by them. Thus, although it may take some effort it is possible to guarantee that restriction, especially if one can use the tool used by SEDCOLE.

As it can be seen in figure 2, the CRC needed to be generated is that associated with modules P_n and M_n (for each computer C_n). However, those modules may not use the whole block (column) size where it is located (these block are referred usually as *frames*). Thus, in order to calculate the CRC of each module we need to use only the bits in the frame that represent its implementation. There lies the third problem with implementing this approach on one-only FPGA. Xilinx does not provide information on the meaning of each bit in the configuration frame for Virtex-II/Virtex-II PRO family. Upegui in (UPEGUI; SANCHEZ, 2005)[14] gives some details on the organization of these bits within a frame, allowing to identify the bits which describe the equation implemented by a LUT. However, more details concerning the rest of this bit stream (routing and other FPGA configuration's resources) is not disclosed by Xilinx.

For all these reasons we were not able to test the idea on a setup containing one-only FPGA, but it should be clear that, providing we have the pieces of information and tools above mentioned, we should be able to do so. As a result we decided to test the idea on a similar configuration that strives to work around the problems aforementioned. This setup is described in the next session.

3. AN EXPERIMENT TO VALIDATE THE PROPOSAL

In order to reproduce the environment needed to test our proposal, we used on our experiment three XUP-V2Pro boards [15][16], each one corresponding to a computer C in the TMR architecture, as shown in figure 4. A personal computer (PC) is connected to each board through RS232 serial ports, whereby the CRC of each computer in the TMR architecture is collected by an application being executed in the PC, which acts as the Voter. The CRC associated with the bit stream of each computer in the FPGA is read via a *readback* operation[17].

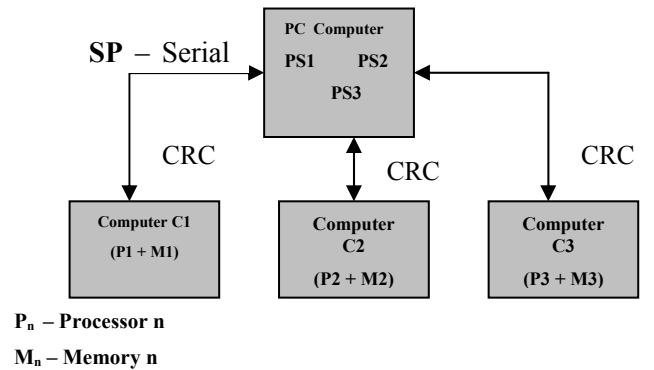


Figure 4: Setup of our experiment. A TMR System with PC Computer acting as Voter.

In each FPGA we embedded a MicroBlaze™ soft processor, developed by Xilinx and made it available in the Software Xilinx

Embedded Development Kit (EDK)[18]. The MicroBlaze has the architecture shown in figure 5.

When the CRCs of all FPGAs are received by the voter application, it votes these results. Should one CRC be in discordance with the others, it is understood as the computer corresponding to that FPGA being faulty. When a fault on a computer is detected via CRC comparison, the voter application automatically discards its computation. In addition to isolating the faulty module, as part of the fault-tolerance recovery action, it could copy both hardware and software bit stream of any faultless FPGA into the faulty one. After that reconfiguration process the Voter could try to use the recovered module in the next cycle. Figure 6 shows that process.

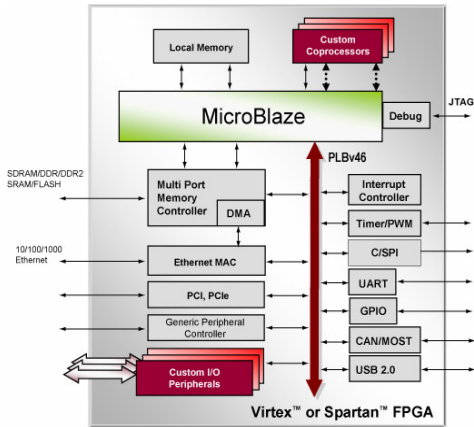


Figure 5: Soft Core MicroBlazer Processor.

Event though the fault occurring in the FPGA does not turn into an error in the computation (i.e. the faulty area was not accessed yet), the fault is detected, since the change in the bits that implement the faulty logic circuit will be reflected on the FPGA's CRC read by a readback operation.

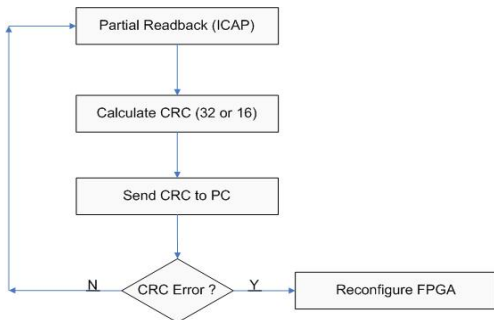


Figure 6: Voter compare the bit stream's CRC of the modules.

The partial FPGA reconfiguration prior to Virtex II/Virtex-IIPro families was realized via the SelectMAP[19] interface or via the JTAG. However, that requires an external controller to send the reconfiguration data. Virtex II/Virtex-IIPro families have an internal interface known as ICAP. ICAP is a reduced set of

SelectMAP, but with the advantage of having distinct input and output ports for data, whereas SelectMAP has one bidirectional port.

The Xilinx's EDK (*Embedded Development Kit*) made it easier to use ICAP by integrating a device named HWICAP, which allows the operation of ICAP via a MicroBlaze software processor. HWICAP is connected to the OPB bus (a proprietary Xilinx bus) [21] and has a control logic and a small cache memory for configuration implemented in a BRAM. The EDK makes available an API [21][22] that can be used to program these devices and that define methods for transferring data between the configuration cache memory and the configuration memory. That facility allows one to perform readback operations, frame by frame (block by block), as well as partial FPGA's reconfigurations. The process running in the PC/Voter was developed by using the ICAP Reconfiguration's API.

Figure 7 shows a block diagram representing each implementation of a computer module in our TMR architecture. Note that we can choose to perform the voting process into the internal MicroBlaze instead of using an external computer (in fact, as a way of increasing the reliability, we could even replicate the MicroBlaze into the FPGA). However, in order to keep firmer control of the experiment, we decide to use the arrangement shown in figure 4.

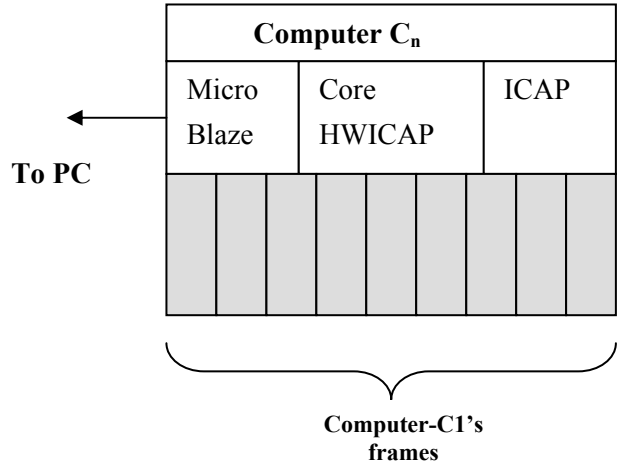


Figure 7: Each FPGA sends its bit streams processed by ICAP via its internal software Microblaze.

4. CONCLUSION AND FUTURE WORK

The use of CRC as a signature of FPGA's healthiness is a very interesting approach. In fact, when this technique is used on a TMR, a change in any FPGA's CRC will indicate both a fault occurring in the memory configuration that implements the system's circuitry and a fault in the computation being carry out by it. As a result, we can use this approach to monitor the status of each module in the TMR. Besides, as the FPGA's configuration memory contains the present status of not only the circuitry but also the status of the computation, a recovery action can be performed by copying the configuration memory contend of a faultless module into a faulty one. Thus this technique can be used with the system on the fly. While some techniques count on the replication of each part of a module and on voting the results of those parts, as part of the

computation, our approach makes the whole fault detection mechanism transparent to the application being performed. This contributes to separate what is application to what is fault tolerance in the system.

As Xilinx does not openly provide information about each bit in each frame configuration of the Virtex-II/Virtex-II PRO FPGA's family, we had to test the proposal by implementing the TMR in different FPGA's. However, the results of our experiment prove that, providing we have those pieces of information we can use it on a one-only implementation. In addition to that, as the routing tools do not obey user's restrictions concerning the way the signals must be routed (which is important to guarantee that the components of each redundant computer is in different frames), we had also to work around that limitation. SEDCOLE [13] used successfully a tool that meets those restrictions. However, this tool is for Xilinx's internal use only and it is not available for outside users. However, this problem can also be worked around providing we have that tool.

The problems mentioned above, of course, do not prevent one from implementing the technique in a setup consisting of only one FPGA.. Our work proved that the technique works and, should we obtain the information above, we could easily use the technique in a one-only FPGA. As a next step on our research we will strive to implement this technique in such a configuration. We also intend to perform the CRC calculation into a specific hardware (VHDL) that will read the CRC via ICAP and will perform the calculation. We believe that CRC calculation will be performed faster, as well as we think that less space will be needed in the FPGA, since the calculation will be performed by the softcore microblaze microprocessor.

It should be said that the approach described in this paper is only a piece of a larger research project which intends to deal with hardware's resources management in the context of fault tolerant reconfigurable systems (mainly by using FPGA's). The way the Voter is handled is an issue that belongs to that scope and it will be discussed in another paper to be published soon. We are also conceiving several set-ups that will measure the latency in the voting process. That, of course, will depend on the way the hardware management deals with the detection and recovery steps of the fault tolerance process. Finally, although we had not provided a formal comparison of our approach with the ones used by other authors, we can say that our approach is superior to the ones quoted in this paper, in the sense that it can be used as a general approach, since it count only on the CRC of each module (regardless if it is a processor or other digital circuit). That will be discussed more extensively in another paper.

5. REFERENCES

- [1] Javier Castilho, Ivan González, Pablo Huerta, J.I. Martinez, "Auto-Reconfiguración sobre FPGAS".
- [2] Blodget B., James-Roxby P., Keller E., McMillan S., Sundararajan P. A., "Selfreconfiguring Platform", FPL'03, pp. 565- 574, Sept 2003.
- [3] M. G. Gericota, G. Alves, M. L. Silva, J. M. Ferreira, "Active Replication: Towards a Truly SRAM-based FPGA On-Line Concurrent Testing", *Proc. IOLTW*, pp. 165-169, 2002.
- [4] KASTENSMIDT, F. L. ; REIS, Ricardo . Designing Single Event Upset Mitigation Techniques for Large SRAM-based FPGAs. Porto Algre: PPGC, 2003.
- [5] TAKAHARA, T. et al. Embedded computer system with soft core cpu for space application. The Military and Aerospace Programmable Logic Device (MAPLD) International Conference, n. P70, p. 1 {6, September 2003}.
- [6] CARMICHAEL, C. Triple Module Redundancy Design Techniques for Virtex FPGAs. [S.l.], July 2006. (xapp197). Application note, v1.0.1.
- [7] CARMICHAEL, C.; CAFFREY, M.; SALAZAR, A. Correcting Single-Event Upset Through Virtex Partial Reconfiguration. [S.l.], June 2000. (xapp216). Application note, v1.0.
- [8] XILINX, INC., MicroBlaze Processor Reference Guide EDK, 2005. DOI= http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf
- [9] XILINX, INC., MicroBlaze Product Brief, 2005. DOI= http://www.xilinx.com/bvdocs/ipcenter/data_sheet/MB_sell_sheet.pdf
- [10] XILINX, INC., http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm
- [11] XILINX, INC., "Virtex-II Platform FPGA User Guide", June 2003.
- [12] SEDCOLE, N. P. Reconfigurable Platform-Based Design in FPGAs for Video Image Processing. Tese (Phd Thesis) | Imperial College of Science, Technology and Medicine, January 2006.
- [13] SEDCOLE, P. et al. Modular dynamic reconfiguration in virtex fpgas. IEE Proceedings Computers and Digital Techniques, v. 153, n. 3, p. 157{164, May 2006.
- [14] UPEGUI, A.; SANCHEZ, E. Evolving hardware by dynamically reconfiguring Xilinx FPGAs. In: MORENO, J. (Ed.). *Evolvable Systems: From Biology to Hardware*. Berlin Heidelberg: Springer-Verlag, 2005. (LNCS, v. 3637), p. 56-65.
- [15] DIGILENT, INC., <http://www.digilentinc.com/Products/Detail.cfm?Nav1=Products&Nav2=Programmable&Prod=XUPV2P>.
- [16] XILINX, INC., <http://www.xilinx.com/univ/xupv2p.html>
- [17] XILINX, INC., "Virtex FPGA Series Configuration and Readback", Application note XAPP502. November 2001.
- [18] XILINX, INC., "Getting Started with the Embedded Development Kit (EDK)"
- [19] XILINX, INC., "Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or SelectMAP Mode", Application note XAPP502. December 2007.
- [20] XILINX, INC., "Designing Custom OPB Slave Peripherals for MicroBlaze", February 2002. Tutorial Report.
- [21] XILINX, INC., "Xilinx 8.2 Libraries Guide".
- [22] XILINX, INC., "Development System Reference Guide 8.2i"