PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO DE ENGENHARIA ELÉTRICA

SOLUÇÕES HÍBRIDAS DE HARDWARE/SOFTWARE PARA A DETECÇÃO DE ERROS EM SYSTEMS-ON-CHIP (SoC) DE TEMPO REAL

LEONARDO BISCH PICCOLI

PORTO ALEGRE
Agosto, 2006

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO DE ENGENHARIA ELÉTRICA

SOLUÇÕES HÍBRIDAS DE HARDWARE/SOFTWARE PARA A DETECÇÃO DE ERROS EM SYSTEMS-ON-CHIP (SoC) DE TEMPO REAL

LEONARDO BISCH PICCOLI

Orientador: Prof. Dr. Fabian Luis Vargas

Dissertação apresentada ao Programa de Mestrado em Engenharia Elétrica, da Faculdade de Engenharia da Pontifícia Universidade Católica do Rio Grande do Sul, como requisito parcial à obtenção do título de Mestre em Engenharia Elétrica.

PORTO ALEGRE
Agosto, 2006

SOLUÇÕES HÍBRIDAS DE HARDWARE/SOFTWARE PARA A DETECÇÃO DE ERROS EM SYSTEMS-ON-CHIP (SoC) DE TEMPO REAL

CANDIDATO: LEONARDO BISCH PICCOLI

Esta dissertação foi julgada para a obtenção do título de MESTRE EM ENGENHARIA ELÉTRICA e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade Católica do Rio Grande do Sul.

_____

Prof. Dr. Flávio A. Becon Lemos

Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

BANCA EXAMINADORA

_____

Prof. Dr. Fabian Luis Vargas - Presidente

_____

Prof. Dr. Eduardo Augusto Bezerra - PUCRS

_____

Prof. Dr. Rubem Dutra Ribeiro Fagundes - PUCRS

AGRADECIMENTOS

Agradeço à minha noiva, por sempre ter me apoiado, incentivado para seguir em frente.
Agradeço aos meus pais, irmã e avó, que nunca me deixaram desistir, sempre dando-me apoio em todas as horas e conselhos sempre úteis para a minha vida.
Ao meu orientador, pelas grandes oportunidades de trabalho e aprendizado.
Aos meus amigos e colegas, por me apoiarem e ajudarem no meu crescimento pessoal e profissional, sempre descontraindo nos momentos de maior estresse.
Sem essas pessoas maravilhosas nada disso teria acontecido.
Obrigado por acreditarem em mim. Estimo muito vocês!

RESUMO

Nos últimos anos, o crescente aumento do número de aplicações críticas envolvendo sistemas de tempo real aliado ao aumento da densidade dos circuitos integrados e a redução progressiva da tensão de alimentação, tornou os sistemas embarcados cada vez mais susceptíveis à ocorrência de falhas transientes.

Técnicas que exploram o aumento da robustez de sistemas em componentes integrados (SoC) através do aumento do ciclo de trabalho do sinal de relógio gerado por um bloco PLL para acomodar eventuais atrasos indesejados da lógica [1] são possíveis soluções para aumentar a confiabilidade de sistemas eletrônicos. Diz-se que estes sistemas utilizam técnicas de "*error avoidance*". Outras técnicas cujo objetivo não é o de evitar falhas, mas sim o de detectá-las, são ditas técnicas de "*error detection*". Este trabalho aborda esse segundo tipo de técnica para aumentar a confiabilidade de sistemas eletrônicos; ou seja, aborda o desenvolvimento de técnicas que realizam a detecção de erros em tempo de execução do sistema.

Sistemas de tempo real não dependem somente do resultado lógico de computação, mas também no tempo em que os resultados são produzidos. Neste cenário, diversas tarefas são executadas e o escalonamento destas em função de restrições temporais é um tema de grande importância. Durante o funcionamento destes sistemas em ambientes expostos à interferência eletromagnética (EMI), existe a enorme probabilidade de ocorrerem falhas transientes. Assim, a utilização de técnicas capazes de detectar erros evita que dados errôneos se propaguem pelo sistema até atingir as saídas e portanto, produzindo um defeito e/ou comprometendo a característica temporal do sistema. Basicamente, as técnicas de detecção são classificadas em duas categorias: soluções baseadas em *software* e soluções baseadas em *hardware*.

Neste contexto, o objetivo principal deste trabalho é especificar e implementar uma solução baseada em *software* (descrito em linguagem C e inserida no núcleo do Sistema Operacional de Tempo Real - RTOS) ou baseada em *hardware* (descrito em linguagem VHDL e conectada no barramento do processador) capaz de detectar em tempo de execução eventuais erros devido a falhas ocorridas no sistema. As falhas consideradas neste trabalho são aquelas que afetam a execução correta do fluxo de controle do programa. A solução proposta é inovadora no sentido de se ter como alvo sistemas SoC com RTOS multitarefa em ambiente preemptivo. A solução proposta associa a estes sistemas, técnicas híbridas de

detecção de erros: baseadas em *software* (YACCA [2,3]) e em *hardware* (WDT [4,5], OSLC [6,7] e SEIS [8,9,10]).

Diferentes versões do sistema proposto foram implementadas. Em seguida, foram validadas em um ambiente de interferência eletromagnética (EMI) segundo a norma IEC 62132-2 [11] que define regras para os testes de circuitos integrados expostos à EMI irradiada. A análise dos resultados obtidos demonstra que a metodologia proposta é bastante eficiente, pois apresenta uma alta cobertura de falhas e supera os principais problemas presentes nas soluções propostas na literatura. Ou seja, associa uma menor degradação de desempenho com um menor consumo de memória e uma maior cobertura de falhas.

Palavras chaves: Sistemas Embarcados de Tempo Real; Aplicações Críticas; Falhas de Fluxo de Controle; Soluções Baseadas em Hardware/Software; Interferência Eletromagnética (EMI).

ABSTRACT

The always increasing number of critical applications requiring real time systems associated with integrated circuits, high density and the progressive system power supply reduction, has made embedded systems more sensitive to the occurrence of transient faults.

Techniques that explore the robustness increase in integrated circuits (SoC) by means of increasing the clock duty-cycle generated by the PLL block, in order to accommodate eventual undesired delays through the logic [1] are possible solutions to increase electronic systems reliability. It is said that such systems use “error avoidance” techniques. Other techniques whose goal is not to avoid fault occurrence, but instead, to detect them, are said “error detection” techniques. This work is focused on the second type of techniques in order to increase electronic systems reliability. In other words, this work proposes the development new techniques to perform fault detection at system runtime.

Real-time systems depend not only on the logical computation result, but also on the time at which these results are produced. In this scenario, many tasks are executed and the efficient time scheduling is a great concern. During system execution in electromagnetic interference (EMI) exposed environments, there is the large probability of transient faults occurrence. Thus, the use of fault detection techniques prevents faults from propagating through the system till primary outputs and them producing systems defect (and/or compromising the time characteristic of the system). Basically, these detection techniques are classified in two main categories: solutions based on software and solutions based on hardware.

In this context, the goal of this work is to specify and to implement a solution based on software techniques (described in C language and inserted in the RTOS kernel) and/or hardware (described in VHDL language and connected on the processor bus) that is capable of performing real time detection of eventual errors in Systems-on-Chips. The faults considered in this work are these that affect the correct processor control flow. The proposed solution is innovative int the sense of having as target systems, those operating is a preemptive multitasking RTOS environment. Therefore, the proposed techniques perform fault detection based on a hybrid solution that combines software (YACCA [2,3]) with hardware (WDT [4,5], OSLC [6,7] and SEIS [8,9,10]).

Several system versions have been proposed and implemented. Then, they were validated in on electromagnetic environment according to the standard IEC 62132-2 [11], witch defines rules for testing integrated circuits under radiated EMI. The obtained results demonstrate that the proposed methodology is very efficient, since it yields a high fault detection coverage higher than those proposed by other methodology on the literature. In other works, the proposed work associates the smallest system performance degradation with the smallest memory overhead and the highest fault detection coverage.

Key-words: Real-Time Embedded Systems; Critical Applications; Control Flow Faults; Hardware/Software Based Approaches; Electromagnetic Interference (EMI).

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

LISTA DE ABREVIATURAS

AC - Alternating Current

ANSI - American National Standards Institute

AP - Application Processor

App - Application

ASIC - Application Specific Integrated Circuit

ASIS - Asynchronous Signatured Instruction Streams

BFI - Branch Free Intervals

BID - Branch-Free Interval Identifier

BRAM - Block RAM

BSSC - Block Signature Self-Checking

CCA - Control Flow Checking by Assertions

CFCSS - Control Flow Checking Signature by Software

CFG - Control Flow Graph

CFID - Control Flow IDentifier

CI - Circuit Integrated

CLB - Configurable Logic Blocks

COTS - Commercial-of-the-shelf

CPU - Central Processing Unit

CRC - Cyclic Redundancy Check

CSM - Continuous Signature Monitoring

DC - Direct Current

DIR - Direction

DUT - Device Under Test

E/S - Entrada/Saída

ECCA - Enhanced Control Flow using Assertions

ECI - Error Capturing Instructions

EMC - Electromagnetic Compatibility

EOP - End of Procedure

FFT - Fast Fourier Transform

FPGA - Field Programmable Gate Array

GDB - GNU Debugger

GSR - Global Shift Register

GTEM - Gigahertz Transverse Electromagnetic

HDL - Hardware Description Language

HSG - Hardware Signature Generator

HW - HardWare

HW/SW – HardWare / SoftWare

IEC - International Electrotechnical Commission

I-IP - Infrastructure Intellectual Property

INTI - Instituto Nacional de Tecnología Industrial (Argentina)

ISR - Interrupt Service Routines

JTAG - Joint Test Action Group

LED - Light Emitting Diode

LFSR - Linear Feedback Shift Register

LSB - Last Significant Byte

LUT - Look-Up Tables

LZW - Lempel-Ziv-Welch

MISRA - Motor Industry Software Reliability Association

MMU - Memory Management Unit

MSB - Most Significant Byte

NFETS - Negative Field Effect Transistor

NMOS - Negative Metal Oxide Silicon

OPB - On-chip Peripheral Bus

ORA - Output Response Analyzer

OSLC - On-line Signature Learning and Checking

PIP - Programmable Interconnection Points

PC - Personal Computer

PCB - Printer Circuit Board

PFETS - Positive Field Effect Transistor

RAM - Randon Access Memory

RE - Radiated Energy

RI - Radiated Immunity

RM - Rate Monotonic
ROM - Read Only Memory
RTOS - Real-Time Operating Systems
SAR - Signature Analysis Register
SEIS - Signature Encoded Instruction Stream
SEU - Single-Event Upsets
SG - Signature Generator
SIHFT - Software Implemented Hardware Fault Tolerance
SoC - System-on-Chip
SOP - Start of Procedure
SRAM - Static Random Access Memory
SRL - Shift-Register Latch
SW – SoftWare
TAP - Port Access Port
TF – Tolerância a Falhas
VLSI - Very Large Scale Integration
VSWR - Voltage Standing Wave Radio
WCET - Worst Case Execution Time
WD - Watchdog
WDDP - Watchdog Direct Processing
WDP - Watchdog Processor
WDP-IP - Watchdog Processor - Infrastructure Intellectual Property
WDP-IP/HW – Versão do WDP-IP implementado em *hardware*
WDP-IP/SW - Versão do WDP-IP implementado em *software*
WDT - Watchdog Timer
XMD - Xilinx Microprocessor Debugger
YACCA - Yet Another Control-Flown Checking using Assertions

Observação: Diversas abreviaturas serão mantidas em inglês, pois para um leitor habituado com os temas abordados se torna muito mais agradável a leitura do texto. Quanto à terminologia utilizada na área de confiabilidade será mantido o proposto por (VERÍSSIMO, 1989) em [12].

1 INTRODUÇÃO

1.1 Motivação

Os sistemas embarcados de tempo real são empregados em praticamente todas as atividades da nossa sociedade, inclusive as mais essenciais. Telefonia e finanças são exemplos de atividades essenciais fortemente dependentes de sistemas de *software*. Sistemas como esses, em que a sua interrupção ou mal funcionamento representam riscos para vidas humanas, bens patrimoniais ou ao meio-ambiente, são denominados sistemas de missão crítica ou, simplesmente sistemas críticos. Nesse contexto um dos principais requisitos é a confiabilidade. Um sistema confiável é aquele para o qual podemos, justificadamente, confiar nos serviços que ele oferece [13].

Sistemas embarcados de tempo real já são largamente disseminados na atualidade, num cenário que projeta expansões ainda maiores para os próximos anos na medida em que os equipamentos eletrônicos portáteis se tornem mais baratos e expandam suas capacidades de recursos computacionais. Este crescimento, aliado ao número de aplicações críticas baseadas em sistemas eletrônicos, intensificou a pesquisa relacionada às técnicas de detecção de erros, ou seja, técnicas capazes de agregarem confiabilidade e robustez aos sistemas. Estas técnicas podem ser classificadas a partir de diferentes conceitos. Entretanto, para os modelos de falhas assumidos neste trabalho, as técnicas de detecção de erros podem ser divididas sinteticamente em soluções baseadas em *software* e soluções baseadas em *hardware*.

Neste contexto, visando manter as vantagens oferecidas por cada uma das duas soluções e minimizar as penalidades decorrentes de suas utilizações, este trabalho propõe soluções implementadas em *software* e *hardware*, traçando um comparativo entre as técnicas através de injeções de falhas tanto em *software* quanto em *hardware* para prover a detecção de erros em falhas transientes.

Portanto, os testes para a validação da técnica desenvolvida nesta dissertação utiliza métodos de injeção de falhas em *software* (alteração nos dados armazenados na memória do processador) e em *hardware* (em ambiente de interferência eletromagnética – EMI segundo a norma IEC 62132-2 [11]) propostos na literatura para simular falhas que podem ocorrer no mundo real.

1.2 Visão Geral dos Objetivos

O principal objetivo deste trabalho é especificar, implementar e avaliar soluções em *software* e em *hardware* capazes de detectar erros no tempo de execução de tarefas e erros de fluxo de controle em aplicações críticas de tempo real durante seu funcionamento que podem operar em ambiente multitarefa e preemptivo. Tem também como objetivo a implementação de técnicas altamente flexíveis que não dependem de uma arquitetura de processador específica ou na utilização de uma determinada aplicação ou sistema operacional.

A metodologia proposta nesta dissertação de mestrado é inédita no sentido de utilizar um sistema operacional de tempo real (*Real-Time Operating Systems* - RTOS) multitarefa e preemptivo na detecção de erros que causam a alteração no tempo de execução da aplicação ou que alterem o controle de fluxo do programa. Baseia-se em alguns conceitos para detecção de erros nas técnicas em *software* (YACCA [2,3]) e em *hardware* (WDT [4,5], OSLC [6,7] e SEIS [8,9,10]).

Em uma primeira versão denominada de WDP-IP (*Watchdog Processor - Infrastructure Intellectual Property*), o WDP-IP opera em simbiose com o sistema operacional de tempo real (RTOS) uC/OS-II [14] e monitora o comportamento do tempo estimado de unidade central de processamento (CPU) para a execução de uma ou mais tarefas de uma aplicação em ambiente multitarefa e preemptivo visando a detecção de erros no tempo de execução.

Neste cenário proposto, em uma versão melhorada do WDP-IP [15,16,17] denominada WDP-IP+, implementa também a técnica YACCA propostas por (GOLOUBEVA, 2003) em [2]. Assim, o WDP-IP+ monitora o fluxo de controle para a execução de cada tarefa e o tempo associado para tal execução visando a detecção de erros.

As soluções propostas neste trabalho propõe que as técnicas sejam implementadas parte em *software* e parte em *hardware*. Neste sentido, a inserção de funções (versão do WDP-IP em *software*) ou instruções (versão do WDP-IP em *hardware*) no núcleo do RTOS para o controle do tempo de cada tarefa e a inserção de funções (versão do WDP-IP+ em *software*) ou instruções (versão do WDP-IP+ em *hardware*) redundantes no código do usuário necessárias para o controle do fluxo do programa são implementadas em *software* (ver Tabela 1.2.1). O monitoramento da consistência do sistema pelo WDP-IP é implementado em *hardware* (denominado WDP-IP em hardware) ou em *software* (denominado WDP-IP em software).

Técnica	Característica e Resumo
WDP-IP em *hardware*	Controla o tempo de execução das tarefas. É descrito em VHDL e esta conectado no barramento do processador. A comunicação é através de instruções de leitura ou escrita do processador.
WDP-IP em *software*	Controla o tempo de execução das tarefas. É descrito em C ANSI e esta embarcado no núcleo do RTOS uC/OS-II [14]. A comunicação é através de funções de chamada no RTOS.
WDP-IP+ em *hardware*	Controla o tempo de execução e o controle de fluxo de cada tarefa através do algoritmo YACCA [2,3]. É descrito em VHDL e esta conectado no barramento do processador. A comunicação é através de instruções de leitura ou escrita do processador.
WDP-IP+ em *software*	Controla o tempo de execução e o controle de fluxo de cada tarefa através do algoritmo YACCA [2,3]. É descrito em C ANSI e esta embarcado no núcleo do RTOS uC/OS-II [14]. A comunicação é através de funções de chamada no RTOS.

Tabela 1.2.1: Características das técnicas propostas [15,16,17].

As técnica propostas, a saber, o WDP-IP e o WDP-IP+, foram testadas e validadas em ambiente de interferência eletromagnética (EMI) tendo como referência a norma IEC 62132-2 [11] para testes de circuitos integrados. Em seguida, também foram utilizadas técnicas de injeção de falhas por *software* durante o funcionamento do sistema em uma placa de desenvolvimento comercial da Xilinx [18].

1.3 Apresentação dos Capítulos

Este trabalho foi divido em três partes, conforme descrito abaixo:

Parte I – Fundamentos na página 25 (Capítulos 2 ao 6):

- Capítulo 2 (TOLERÂNCIA A FALHAS): Apresenta uma breve introdução relacionada aos principais conceitos que envolvem a teoria de teste;
- Capítulo 3 (INTRODUÇÃO AOS SISTEMAS EMBARCADOS): Apresenta os principais conceitos dos sistemas embarcados de tempo real e técnicas de análise das estruturas e robustez das aplicações;
- Capítulo 4 (INJEÇÃO DE FALHAS): Explora as técnicas de injeção de falhas, apresentando suas características;
- Capítulo 5 (TÉCNICAS DE DETECÇÃO DE ERROS VIA SOFTWARE): Apresenta técnicas para a detecção de erros em fluxo de controle de sistemas

baseados em SoCs implementadas em *software*. Salienta-se que estas técnicas servem de referência para este trabalho, mais especificamente a técnica YACCA [2,3].

- Capítulo 6 (TÉCNICAS DE DETECÇÃO DE ERROS VIA HARDWARE): Apresenta técnicas para a detecção de erros em fluxo de controle de sistemas baseados em SoCs implementadas em *hardware*. Salienta-se que estas técnicas servem de referência para este trabalho, mais especificamente aquelas apresentadas em WDT [4,5], OSLC [6,7] e SEIS [8,9,10].

Parte II – Metodologia na página 99 (Capítulo 7):

- Capítulo 7 (IMPLEMENTAÇÕES DO WDP-IP): Apresenta a técnica híbrida WDP-IP [15,16,17] proposta neste trabalho para a detecção de erros de tempo de execução e fluxo de controle em ambiente multitarefa e preemptivo em sistemas baseados em SoCs. Salienta-se que a técnica proposta baseia-se nas soluções: YACCA [2,3] (para a detecção de erro no fluxo de controle das tarefas), WDT [4,5] (para o controle do tempo de cada tarefa), OSLC [6,7] (para o auto-aprendizado da assinatura de cada bloco básico, no caso o bloco é uma tarefa e a assinatura é o tempo de execução da tarefa) e SEIS [8,9,10] (para a comunicação entre o processador e o WDP-IP implementado em *hardware*).

Parte III - Resultados e Conclusões na página 133 (Capítulos 8 e 9):

- Capítulo 8 (RESULTADOS): Apresenta os resultados obtidos a partir de experimentos de injeção de falhas reais através de EMI radiada e injeção de falhas via s*oftware* com a inserção de instruções de desvio (*jump*) e *bit-flip* em tempo de execução na área de memória física da placa de desenvolvimento, onde é armazenado o código da aplicação;
- Capítulo 9 (CONCLUSÃO): Apresenta as conclusões relacionadas às técnicas submetidas aos testes no capítulo anterior, as conclusões relacionadas ao desenvolvimento deste trabalho de dissertação e finalmente sugere alguns trabalhos para serem desenvolvidos no futuro.

PARTE I - FUNDAMENTOS

2 TOLERÂNCIA A FALHAS

2.1 Conceitos de Tolerância a Falhas

Tolerância a falhas é a habilidade de um sistema continuar a execução correta (sem degradação) das tarefas mesmo na ocorrência de falhas em *hardware* ou em *software* [19].

Com a inserção dos computadores nos ramos da sociedade moderna, os sistemas computacionais tornaram-se uma ferramenta indispensável, visto os inúmeros benefícios alcançados com seu uso [20]. Tais benefícios tornaram-se tão imprescindíveis que os prejuízos proporcionados quando os sistemas computacionais deixam de funcionar ou funcionam incorretamente, podem ter consequências trágicas, a exemplo de: aeronaves, ferrovias, controle de tráfego de automóveis; sistemas de emergência; controles de vôo de linhas aéreas; sistemas de segurança de usinas nucleares; equipamentos médico-hospitalares [21].

Tolerância a falhas é a melhor garantia de confiança para os usuários sujeitos a erros físicos, de projeto, de interação humana, que vírus ou atos maliciosos possam interromper serviços essenciais.

Sistemas tolerantes a falhas exigem mecanismos para detecção de erros. A concepção dos mecanismos de detecção de erro exige conhecimento das classes de erro e seus efeitos sobre o comportamento do sistema. Além disso, um estudo de efeitos do erro ajuda a caracterizar a susceptibilidade do sistema para falhas. Uma abordagem efetiva para alcançar estas metas são os procedimentos de injeção de falhas. Porém, para obter resultados consistentes, a injeção de falhas deve prover também um modelo de falha que represente igualmente falhas reais encontradas em operações reais [22].

2.2 Objetivos da Tolerância a Falhas

Tolerância a falhas é um atributo que é projetado para um sistema alcançar metas de projeto [23]. Da mesma maneira que um projeto deve apresentar muitas funcionalidades e metas de performance, deve adicionalmente satisfazer numerosos outros requisitos tais como: confiança no funcionamento, confiabilidade, disponibilidade, segurança, desempenhabilidade,

mantenabilidade e testabilidade.

Tolerância a falhas é um atributo de sistema capaz de cumprir os requisitos [24]:

- Dependabilidade (*dependability*): é a qualidade de serviço provido por um sistema particular. Confiabilidade, disponibilidade, segurança, desempenhabilidade, mantenabilidade e testabilidade são exemplos de medidas usadas para quantificar a confiança de um sistema e são medidas quantitativas correspondentes a distintas percepções do mesmo atributo de um sistema [13];
- Confiabilidade (*reliability*): é uma função de tempo definida como a probabilidade condicional que o sistema desempenha corretamente ao longo de um intervalo de tempo;
- Disponibilidade (*availability*): é uma função de tempo definida como a probabilidade que tem um sistema de operar corretamente, e estar disponível para desempenhar suas funções em um determinado intervalo de tempo;
- Segurança (*safety*): é a probabilidade de um sistema apresentar suas funções corretamente ou descontinuar suas funções, de maneira que não corrompa as operações de outros sistemas ou comprometa a segurança das pessoas associadas ao sistema;
- Desempenhabilidade (*performability*): Em muitos casos, é possível projetar sistemas que possam continuar executando corretamente após a ocorrência de falhas de *hardware* ou *software*, mas o nível de desempenho de alguma maneira diminui;
- Mantenabilidade (*maintainability*): é uma medida da facilidade com que um sistema pode ser consertado uma vez que apresentou defeito;
- Testabilidade (*testability*): é a habilidade de testar certos atributos de um sistema. Certos testes podem ser automatizados sob condição de integrar como parte do sistema e melhorar a testabilidade. A testabilidade está claramente relacionada a mantenabilidade pela importância de minimizar o tempo exigido de identificação e localização de problemas específicos.

A questão de tolerância a falhas em computação de tempo real tem ganho atenção dos pesquisadores. Os computadores têm ganho maior espaço em aplicações críticas, sendo que uma maior confiança tem sido um requisito importante para estes sistemas, pois enquanto

aumenta sua autonomia, a intervenção humana têm se reduzido. Situações rígidas onde a demanda por processamento de tempo real, têm conseqüências catastróficas na presença de falhas [25].

2.3 Definições: Falha, Erro e Defeito

As falhas são causadas por fenômenos naturais de origem interna ou externa e ações humanas acidentais ou intencionais [13]. Quando ocorrem falhas no sistema, sua manifestação dá origem aos erros, que subseqüentemente resultam na origem de defeitos que são um desvio do comportamento especificado para o sistema, como mostra a Ilustração 2.3.1.

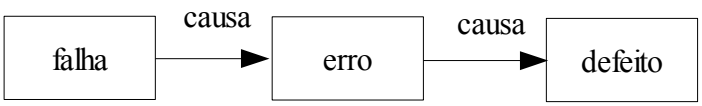


Ilustração 2.3.1: Falha, erro e defeito [13].

Definição [13]:

- A causa de um erro é uma falha;
- Um erro é a manifestação no sistema de uma falha, mas não necessariamente leva a um defeito.
- Um defeito é a manifestação de um erro no serviço entregue;
- O serviço entregue por um sistema é o comportamento conforme é percebido por outro sistema, interagindo com o sistema considerado: seu usuário(s).

A Ilustração 2.3.2 mostra alguns conceitos relacionados à terminologia de tolerância a falhas. Quando uma falha causa uma alteração incorreta no estado da máquina, um erro ocorre. O tempo entre a ocorrência da falha e a primeira percepção de um erro é chamado latência da falha. Embora uma falha permaneça localizada no código ou circuito afetado, múltiplos erros podem se originar da mesma e se propagar através do sistema. Se os mecanismos necessários estão presentes, a propagação do erro será detectada após um período de tempo denominado latência do erro. Quando os mecanismos de tolerância a falhas detectam um erro, diversas ações podem ser iniciadas para a manipulação da mesma, visando conter os erros. A recuperação ocorre se essas ações têm sucesso, caso contrário, o sistema encontra-se com defeito [21].

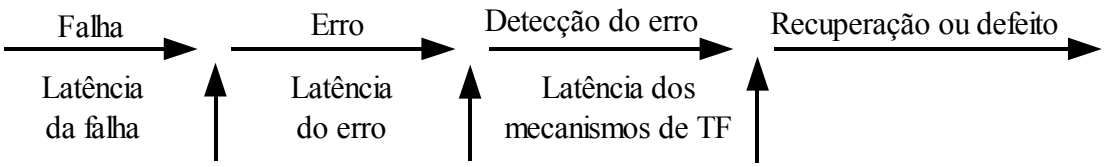


Ilustração 2.3.2: Falha e conseqüências [21].

2.4 Tipos de Falhas

Deve-se entender o comportamento da falha no circuito, a fim de descobrir um modelo de falha. O circuito defeituoso deve produzir um erro em uma de suas saídas para que a falha seja detectada.

Os desafios que devem ser superados para construir um projeto de microprocessador confiável são grandes. Existem muitas origens de falhas, cada uma exigindo atenção especial durante o projeto, verificação, e fabricação. As falhas que podem reduzir a confiabilidade podem ser divididas em três categorias [26,27]:

- Falhas de projeto: resultam de erro humano, ou no projeto ou na especificação de um componente do sistema, resultando em parte na incapacidade de responder corretamente para certas entradas do sistema. A abordagem típica utilizada para detectar estes erros são baseados na verificação por simulação.
- Falhas de fabricação: resultam em uma gama de problemas no processamento, que se manifestam durante a fabricação. Por exemplo: problemas nas etapas durante o processo de metalização podem causar circuitos abertos, ou dopagem imprópria no canal de transistores CMOS podendo causar mudança de limiar de voltagem e retardos em um dispositivo. Os testes neste sistema são realizados adicionando *hardware* específico para teste.
- Falhas operacionais: são caracterizadas pela sensibilidade do componente em condições ambientais. Estes tipos de erros baseados na sua frequência de ocorrência e são divididos em sub-categorias:
    1. Falhas permanentes: são causadas por defeitos de dispositivos irreversíveis em um componente devido a dano, fadiga ou manufatura imprópria. Uma vez ocorrida a falha permanente, o componente defeituoso pode ser restaurado somente pela reposição ou, se possível, pelo reparo [21,28].

2. Falhas transientes: são ocasionadas por distúrbios de ambiente tais como flutuações de voltagem, interferência eletromagnética ou radiação. Esses eventos tipicamente têm uma duração curta, sem causar danos ao sistema (embora o estado do sistema possa continuar errôneo). Falhas transientes podem ser até 100 vezes mais freqüentes que falhas permanentes, dependendo do ambiente de operação do sistema [21]. Devido a falhas transientes afetarem o sistema aleatoriamente e não terem uma característica periódica, são difíceis de detectar e isolar, conseqüentemente se tornam uma grande preocupação, especialmente em aplicações críticas de tempo real. As principais fontes de faltas transientes são radiação e interferência eletromagnética que geram principalmente evento único de pertubação (*Single-Event Upset* ou SEU), voltagem ou pulsos aleatórios de corrente [29]. O crescimento exponencial no número de transistores em circuitos integrados, juntamente com reduções de níveis de voltagem, faz com que a cada geração de microprocessadores haja um aumento na vulnerabilidade de sistemas eletrônicos para falhas transientes [30]. Falhas transientes causadas por radiação HIR (*heavy-ion radiation*) são encontradas no espectro de raio cósmico caraterizando um grande problema para equipamentos eletrônicos utilizados em aplicações espaciais [22].
3. Falhas intermitentes: tendem a oscilar entre períodos de atividade errônea e dormência, podem permanecer durante a operação do sistema. Elas são geralmente atribuídas a erros de projeto que resultam em *hardware* instável [21].
4. Falhas de software: são causadas por especificação, projeto ou codificação incorreta de um programa. Embora aparentemente o *software* não falhe fisicamente após ser instalado em um computador ou envelhece fisicamente da mesma forma que um *hardware*, falhas latentes ou erros no código podem permanecer durante a operação. Isto pode ocorrer sob altas cargas de trabalho ou cargas não usuais para determinadas condições. Tais situações eventualmente conduzem o sistema a defeitos. Sendo assim, injeção de falhas por *software* são usadas principalmente para testes de programas ou mecanismos de tolerância a falhas implementados por *software* [21].

Dentre as falhas citadas, falhas transientes e intermitentes são as maiores causadoras de defeito em computadores. Estima-se em que em bons ambientes, mais de 90% dos defeitos em computadores são causados por falhas transientes [7].

2.5 Efeito das Falhas

Efeitos das falhas podem ser classificados em duas categorias [31]:

- Erros de dados: surgem quando o conteúdo de (ao menos) uma variável armazenada na memória ou em (ao menos) um registrador do microprocessador é alterado;
- Erros de fluxo de controle: surgem quando o conteúdo de uma célula de memória ou registradores do microprocessador que armazenam instruções são alterados, causando a execução incorreta da seqüência de instruções.

Os erros de dados e fluxo de controle podem ser divididos em alterações de dois tipos:

- Tipo S1: Somente alterações que afetam os dados (por exemplo: atribuições, cálculo de expressões aritméticas, etc.);
- Tipo S2: Alterações que afetam o fluxo de controle (por exemplo: testes, laços, procedimento de chamadas e retorno, etc.).

Por outro lado, erros que afetam o código podem ser divididos em dois tipos, dependendo do caminho das alterações, cujo código é modificado:

- Tipo E1: os erros mudam a instrução a ser apresentada pela declaração, sem alterar o fluxo de execução do código (por exemplo: mudando uma instrução de adição por uma de subtração);
- Tipo E2: os erros mudam o fluxo de execução (por exemplo: mudando uma instrução de soma para uma instrução de desvio ou vice-versa).

Uma representação das possíveis transformações causadas por erros é mostrada na Ilustração 2.5.1.

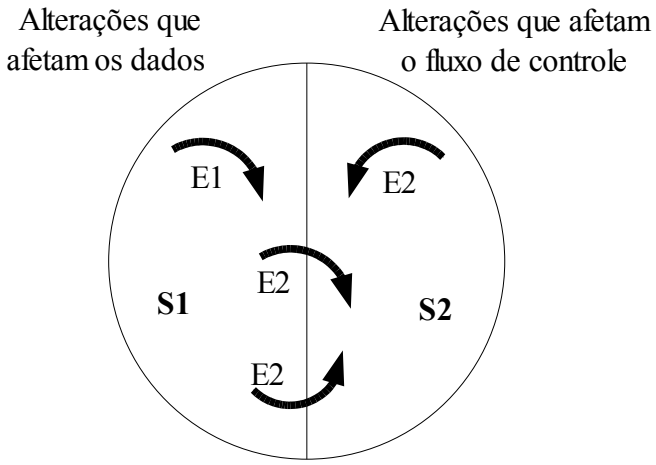


Ilustração 2.5.1: Classificação dos efeitos dos erros [31].

Pulsos de tensão ou de corrente aleatórias são perturbações induzidas nas interconexões do circuito podendo levar a uma interpretação errônea de um nível de lógica. Em sistemas de microprocessadores estas falhas podem ser localizadas em três principais áreas [29]:

- Falha na Memória: é um dos efeitos mais comuns por pertubação em sistemas de microprocessadores. Normalmente, consiste no surgimento de SEU numa posição de memória. Programa e dados armazenados em memória podem ser afetados por estas falhas.
- Falha no Processador: é semelhante a falha de memória. O estado interno do processador é alterado, causando uma execução errônea do programa, efeitos secundários como corrupção dos dados de memória e operações incorretas de periféricos. É causado por SEU num registrador interno do processador.
- Falha no Barramento: estas falhas podem ser causadas por pulsos de tensão ou de corrente aleatórias nos pinos de E/S do microprocessador ou no barramento do sistema. Causam a transferência incorreta de dados e corrupção dos dados durante o processamento (veja Ilustração 2.5.2).

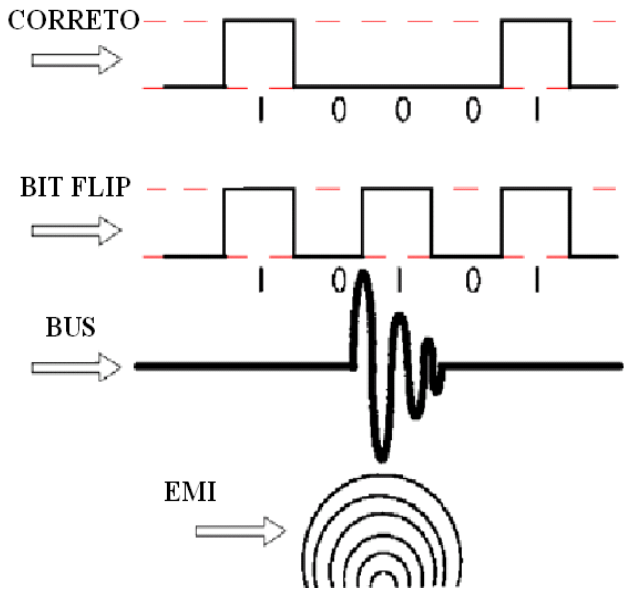


Ilustração 2.5.2: Transferência de dados incorretos [29].

2.6 Defeitos e Modelos de Falhas

Modelos de falhas são utilizados para emulação de falhas ou defeitos em um ambiente de simulação durante a etapa do projeto e devem atender aos seguintes requisitos [32]:

- Refletir com precisão o comportamento dos defeitos reais que podem acontecer durante a fabricação e no processo de manufatura, como também o comportamento de falhas que podem acontecer durante a operação do sistema.
- Deve ter eficiência computacional com respeito ao ambiente de simulação de falha

Serão descritas nas seções seguintes vários modelos de falhas, que nos últimos anos surgiram baseados nos principais defeitos físicos encontrados nos circuitos.

2.6.1 Modelo de Falha *Gate-Level Stuck-at*

Uma simples falha ***stuck-at*** causada em um só nó de um circuito digital, um determinado valor lógico [33]. A falha *stuck-at-1* (sa1) inibe o nó de comutar para o nível lógico “0”, enquanto a falha *stuck-at-0* (sa0) inibe a comutação para o nível lógico “1”. Em lógica de três estados uma falha *stuck-at*-off (saZ) pode ser definida como sendo a proibição do sinal lógico de ser transmitido sobre o barramento.

A Ilustração 2.6.1.1 apresenta a notação utilizada para representar falhas *stuck-at* e a Ilustração 2.6.1.2 o comportamento das mesmas.

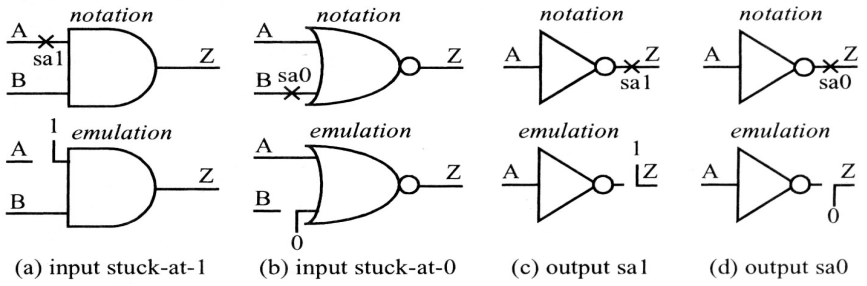
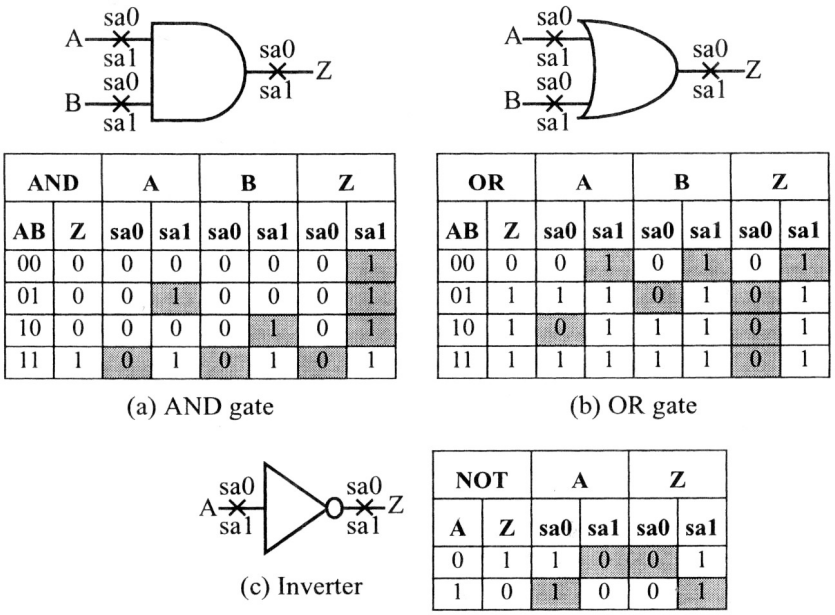


Ilustração 2.6.1.1: Notação do modelo de falha *Stuck-At* [32].



AND		A		B		Z	
AB	Z	sa0	sa1	sa0	sa1	sa0	sa1
00	0	0	0	0	0	0	1
01	0	0	1	0	0	0	1
10	0	0	0	0	1	0	1
11	1	0	1	0	1	0	1

OR		A		B		Z	
AB	Z	sa0	sa1	sa0	sa1	sa0	sa1
00	0	0	1	0	1	0	1
01	1	1	1	0	1	0	1
10	1	0	1	1	1	0	1
11	1	1	1	1	1	0	1

NOT		A		Z	
A	Z	sa0	sa1	sa0	sa1
0	1	1	0	0	1
1	0	1	0	0	1

Ilustração 2.6.1.2: Comportamento do modelo de falha *Stuck-At* [32].

Salienta-se que as falhas *stuck-at* são emuladas como se as portas de entradas e saídas estivessem desconectadas e assim amarradas ao valor lógico "0" (*stuck-at-0*) ou ao valor lógico "1" (*stuck-at-1*).

2.6.2 Modelo de Falha *Transistor-Level Stuck-at*

Este modelo reflete o comportamento exato das falhas de transistores em circuitos CMOS e define que qualquer transistor pode estar *stuck-on* (também denominado *stuck-short*)

ou *stuck-off* (também denominado *stuck-open*).

A Ilustração 2.6.2.1 mostra a emulação do modelo de falha *Transistor-Level Stuck-at* no transistor de entradas CMOS-NOR e a Ilustração 2.6.2.2 o seu comportamento.

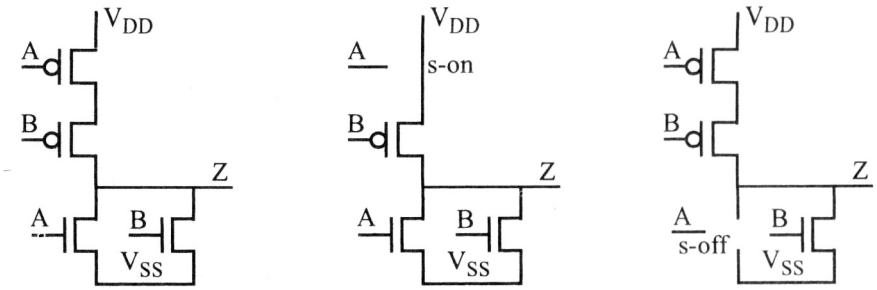


Ilustração 2.6.2.1: Modelo de falha *Transistor-Level Stuck-at* [32].

Fault-Free NOR		A PFET		B PFET		A NFET		B NFET	
AB	Z	s-on	s-off	s-on	s-off	s-on	s-off	s-on	s-off
00	1	1	Z⁻	1	Z⁻	V_Z/I_{DDQ}	1	V_Z/I_{DDQ}	1
01	0	0	0	V_Z/I_{DDQ}	0	0	0	0	Z⁻
10	0	V_Z/I_{DDQ}	0	0	0	0	Z⁻	0	0
11	0	0	0	0	0	0	0	0	0

Ilustração 2.6.2.2: Comportamento do modelo de falha *Transistor-Level Stuck-at* [32].

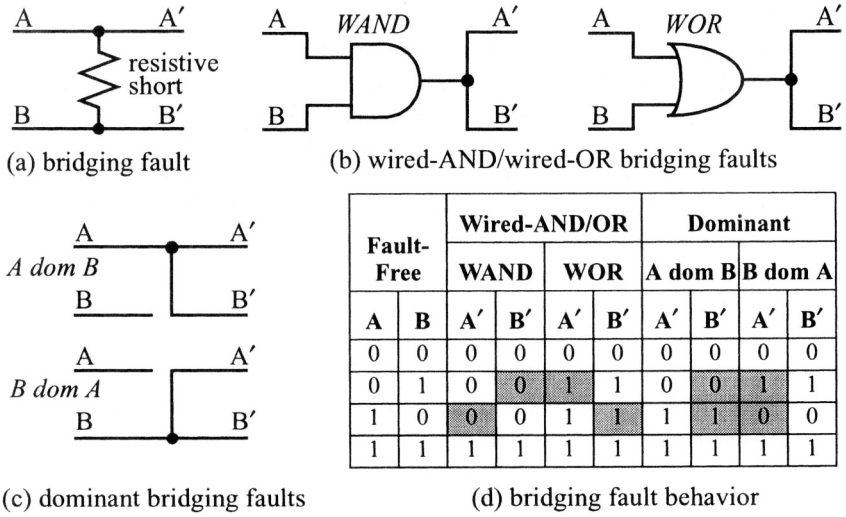
Como podemos observar na Ilustração 2.6.2.1, as falhas *stuck-on* (*s-on*) podem ser emuladas através de um curto circuito entre a fonte e o dreno do transistor, e as falhas *stuck-off* (*s-off*) desconectando-se o transistor do circuito. Alternativamente, falhas *stuck-on* podem ser emuladas desconectando a porta MOSFET do sinal e conectando-a a lógica "1" para transistores NFETS ou a lógica "0" em transistores PFETS. Este procedimento fará com que o transistor esteja sempre conduzindo. Já no que diz respeito a falhas *stuck-off*, elas podem ser emuladas conectando a porta (*Metal Oxide Silicon Field Effect Transistor* ou MOSFET) a lógica "0" para transistores NFET e a lógica "1" para transistores PFET, assim o transistor nunca conduzirá.

A partir da análise das Ilustrações 2.6.1.1 até a 2.6.2.2, observa-se que um mesmo conjunto mínimo de vetores de teste é capaz de detectar simultaneamente falhas *stuck-on*, *stuck-off* e *stuck-at*.

2.6.3 Modelo de Falha *Bridging*

O modelo de falha *bridging* pode ser causado por um componente (exemplo: um transistor) avariado, que toca em outra trilha do circuito ou o cobre em excesso que não foi removido durante a fase de corrosão e curto-circuito em duas ou mais trilhas da placa de circuito impresso [33].

A Ilustração 2.6.3.1 mostra o modelo de falhas *bridging* para dois casos específicos, denominados *wired-AND/wired_OR bridging* e *dominant bridging*.

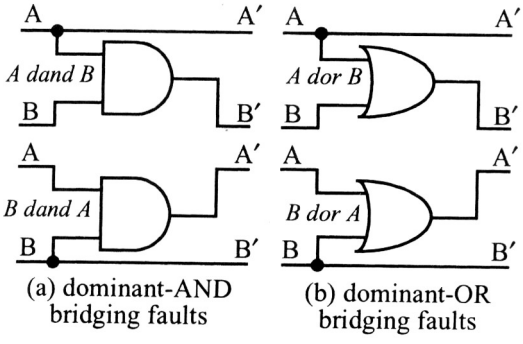


Fault-Free		Wired-AND/OR				Dominant			
		WAND		WOR		A dom B		B dom A	
A	B	A′	B′	A′	B′	A′	B′	A′	B′
0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	0	1	1
1	0	0	0	1	1	1	1	0	0
1	1	1	1	1	1	1	1	1	1

(d) bridging fault behavior

Ilustração 2.6.3.1: Modelos de falhas wired-AND/wired-OR bridging e dominant bridging [32].

Outro modelo de falha *bridging* definido a partir do comportamento observado em curtos que ocorrem em *Application Specific Integrated Circuits* (ASICs) e *Field Programmable Gate Arrays* (FPGAs) é definido como *dominant-AND/OR bridging*. A Ilustração 2.6.3.2 mostra este modelo.



(a) dominant-AND bridging faults

(b) dominant-OR bridging faults

Fault-Free		A dand B		B dand A		A dor B		B dor A	
A	B	A′	B′	A′	B′	A′	B′	A′	B′
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	1	1	1
1	0	1	0	0	0	1	1	1	0
1	1	1	1	1	1	1	1	1	1

(c) dominant-AND/OR bridging fault behavior

Ilustração 2.6.3.2: Modelo de Falha *dominant-AND/OR bridging* [32].

Quando existirem muitas portas lógicas no circuito combinacional, o modelo de falha *bridging* pode resultar em oscilações devido à realimentação em laço no circuito.

Enquanto o modelo de falhas *transistor-level* e *bridging* refletem mais fielmente o comportamento dos defeitos presentes em circuitos, sua emulação e avaliação em simuladores é computacionalmente mais complexa em relação às tradicionais falhas *stuck-at*.

2.6.4 Modelo de Falha *Delay*

Este modelo representa outra importante classe de falhas que ocorrem quando a operação executada pelo circuito é logicamente correta, mas não é executada em tempo hábil [33].

Falhas de *Delay* são muito mais difíceis de emular comparadas aos modelos discutidos anteriormente. O objetivo básico deste tipo de teste é verificar os caminhos entre *flip-flops*, entre entradas primárias e *flip-flops*, e finalmente entre *flip-flops* e saídas primárias, ou seja, verificar através da lógica combinacional se durante a operação na velocidade requerida, algum caminho do sistema apresentou um atraso de propagação do sinal além do permitido.

Tipicamente, o teste de *Delay* consiste na aplicação seqüencial de dois vetores, uma vez que o caminho através da lógica combinacional é carregado com o primeiro vetor enquanto o segundo vetor gera a transição através dos caminhos para a detecção da falha.

2.6.5 Modelo de Falha Múltiplo *Stuck-at*

Durante o processo de fabricação de um determinado dispositivo VLSI ou PCB múltiplos defeitos podem ser inseridos. Para ilustrar com mais clareza, as diferenças em termos de tempo de simulação dos modelos simples versus múltiplos, observe os exemplos, a seguir descritos. Em um circuito com N portas de entradas e saídas, diante do modelo múltiplo de falha *stuck-at* deve-se emular 3^N-1 e do modelo de falha simples apenas 2^N diferentes falhas. A mesma análise pode ser feita diante dos modelos *transistor-level stuck* e para o modelo *wired-AND/OR* ou *dominant bridging*. Já para o modelo *dominant-AND/OR bridging* múltiplo é necessário simular 5^N-1 falhas e no simples 4N falhas.

O modelo de falha simples *stuck-at* é admitidamente uma simplificação da realidade, embora seja uma simplificação comumente feita. Em circuitos reais, particularmente aqueles recentemente fabricados, falhas múltiplas são freqüentes [33].

Entretanto, a alta cobertura de falhas obtidas a partir de modelos de falhas simples garante sua ampla utilização no desenvolvimento e avaliação de testes.

2.6.6 Modelos de Falhas em FPGAs

Dois aspectos importantes devem ser considerados na tecnologia de FPGAs baseada em SRAM [34]:

- SEUs podem alterar os elementos de memória utilizados por aplicações executadas em projetos embarcados;
- SEUs podem alterar o conteúdo da memória do dispositivo de armazenamento das informações de configuração. Por exemplo, SEUs podem alterar o conteúdo de *Look-Up Tables* (LUTs) dentro dos blocos lógicos de configuração (*Configurable Logic Blocks* - CLBs), ou o roteamento dos sinais dentro do CLBs ou entre os CLBs.

Os efeitos dos SEUs na memória de configuração do dispositivo não estão limitados apenas na modificação dos elementos de memória, mas também podem produzir modificações de interconexões dentro dos CLBs, e entre diferentes CLBs, ocasionando circuitos totalmente diferentes daqueles esperados.

Nos dispositivos baseados em SRAM da Xilinx [18], o roteamento do sinal acontece por interconexão de matrizes denominado pontos programáveis de interconexão (*Programmable Interconnection Points* ou PIPs). A ferramenta de posicionamento e roteamento (*place-and-route tool*) pode implementar qualquer rede de conexão de dois módulos do circuito unindo vários PIPs, cada um pertencendo a uma interconexão em ponte. Como resultado, SEUs podem modificar os bits de configuração das interconexões em ponte de um PIP e interromper a propagação do sinal entre CLBs e em larga escala, módulos do circuito.

A Ilustração 2.6.6.1 mostra uma situação livre de falhas. O PIP pode ser utilizado para conectar sinais de entrada (IN_0 a IN_11) originários de recursos do FPGA para sinais de saída (OUT0 a OUT7) e implementar o roteamento de rede Net_1 e Net_2 definidas pela ferramenta de posicionamento e roteamento.

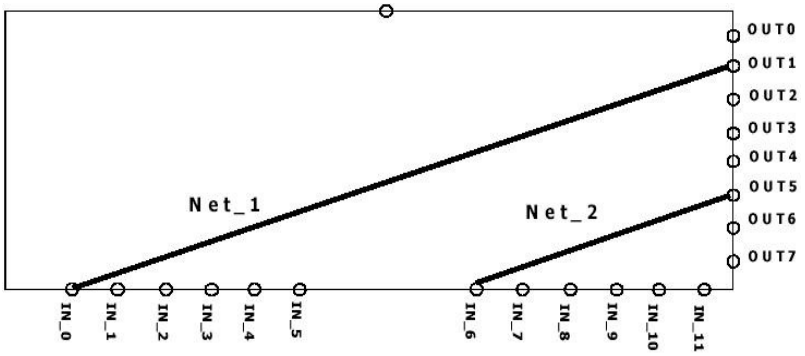


Ilustração 2.6.6.1: Representação esquemática da matriz de interconexão implementada por um PIP [35].

Um novo método para avaliar os efeitos de SEUs nos mecanismos de configuração de memória foi proposto em [35,36]. A partir da configuração livre de falhas apresentada na Ilustração 2.6.1.1, identificou-se os seguintes modelos de falhas:

- ***Aberto:***

A configuração do PIP correspondente à Net_1 é configurado em estado aberto, de modo que IN_0 e OUT1 estejam desconectados. O efeito do SEU pode ser classificado em dois casos distintos para o estado **Aberto** como demonstra a Ilustração 2.6.6.2.

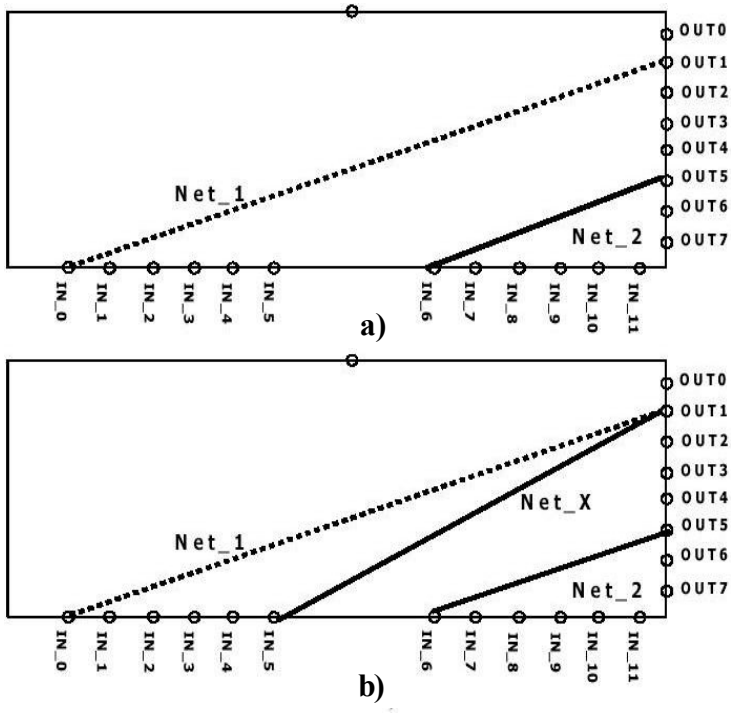


Ilustração 2.6.6.2: O SEU introduz uma conexão aberta cortando o roteamento da PIP NET_1 [35].

Na Ilustração 2.6.6.2.a o roteamento da PIP Net_1 é cortado, enquanto na Ilustração 2.6.6.2.b é inserida uma nova PIP Net_X conectando um novo nó de entrada com o nó de saída utilizado. Como resultado os CLBs são alimentados com o sinal previamente conduzido sobre a Net_1, tornando oscilatório.

- ***Ponte:***

Como mostra a Ilustração 2.6.6.3, um novo PIP, denominado Net_X é habilitado, enquanto a Net_1 é cortada, como é o caso do modelo **Aberto**. O novo PIP pode influenciar o comportamento do circuito implementado, já que a saída do bloco, originalmente conduzida pela rede cortada, é agora conduzida por um valor lógico desconhecido.

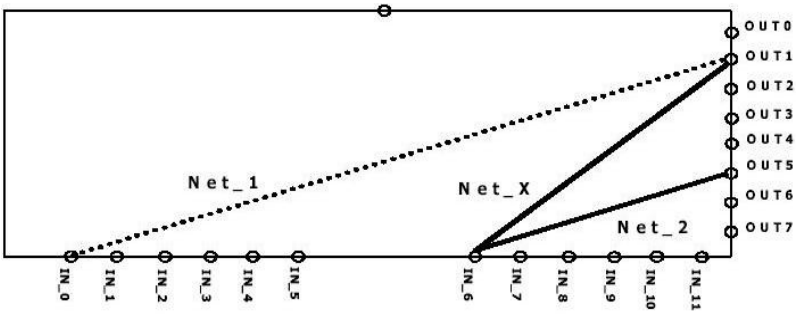


Ilustração 2.6.6.3: O SEU introduz um novo caminho entre os nós utilizados [35].

- ***Antena de Entrada:***

Um novo PIP, denominado Net_X, começa pela conexão de um nó não utilizado de entrada com um nó utilizado de saída como mostra a Ilustração 2.6.6.4. O novo PIP pode influenciar o comportamento do circuito implementado, já que a saída é conduzida por um valor lógico desconhecido.

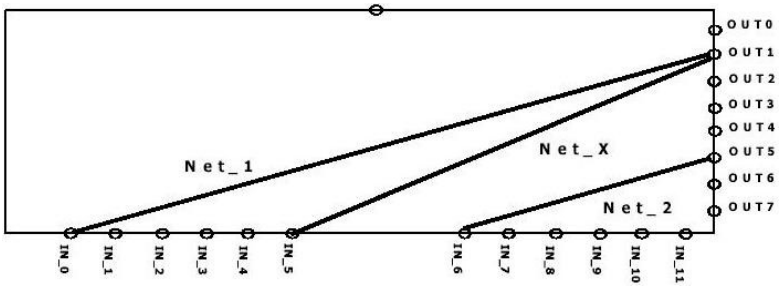


Ilustração 2.6.6.4: O SEU introduz um novo caminho entre um nó não utilizado de entrada e um nó utilizado de saída [35].

- ***Antena de saída:***

Um novo PIP, chamado Net_X, começa pela conexão de um nó utilizado de entrada com um nó não utilizado de saída como mostra a Ilustração 2.6.6.5. O novo PIP não influencia o comportamento do circuito implementado, já que os CLBs ou blocos de saída não são utilizados.

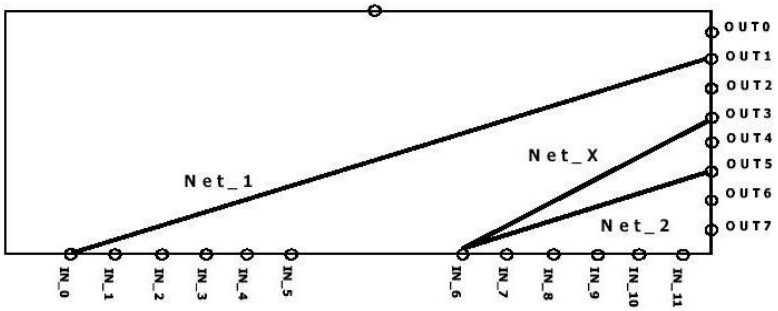


Ilustração 2.6.6.5: O SEU introduz um novo caminho entre um nó utilizado de entrada e um nó não utilizado de saída [35].

- ***Conflito:***

Um novo PIP denominado Net_X conecta um nó de entrada a um nó de saída, ambos já utilizados, como mostra a Ilustração 2.6.6.6. O novo PIP causa um conflito, resultando na propagação de valores desconhecidos para a saída da FPGA.

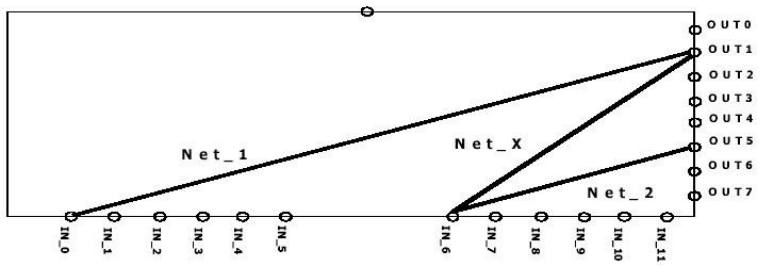


Ilustração 2.6.6.6: O SEU introduz um novo caminho entre nós utilizados [35].

Os efeitos do SEU podem não afetar as configurações dos PIPs, devido ao fato de modificar partes da configuração de memória do dispositivo não utilizados, e também por não estarem classificados em nenhuma das classes descritas acima. Porém, no trabalho apresentado por (VIOLANTE, 2003) em [35], foi demonstrado que as interconexões PIP da FPGA são mais sensíveis aos efeitos de SEUs que as alterações no conteúdo dos LUTs e que os efeitos das falhas são em predominância do tipo **Aberto** e **Conflito**.

3 INTRODUÇÃO AOS SISTEMAS EMBARCADOS

3.1 Sistemas de Tempo Real

Sistemas de tempo real caracterizam-se pela necessidade fundamental de manter um sincronismo constante com o processo, isto é, o sistema deve atuar de acordo com a dinâmica de estados do processo. A chave do sucesso em sistemas de tempo real é a oportunidade de execução de tarefas de processamento de dados que se comunicam para realizar um objetivo em comum [37].

Sistemas de tempo real não dependem só do resultado lógico de computação, mas também do tempo em que os resultados são produzidos, onde diversas tarefas são executadas e o escalonamento em função das restrições temporais é um grande problema. Tarefas recebem dados de entrada, executam um algoritmo e geram saídas. A tarefa está logicamente correta se gerar uma saída correta em função dos dados de entrada em um prazo temporal satisfatório [38].

Devido a isso pode-se afirmar que, o tempo é o recurso mais precioso utilizado por sistemas de tempo real, pois as tarefas executadas pelo processador devem ser concluídas antes de seu limite temporal (*deadline*) [39].

3.2 Desenvolvimento de Sistemas de Tempo Real Embarcados

Os Sistemas de tempo real embarcados tornam mais fácil a realização e a manutenção dos programas de aplicação de usuário e têm resolvido de forma aceitável os problemas de desenvolvimento de parte das aplicações, principalmente aquelas menos críticas. Porém, devido ao aumento da complexidade dos sistemas de tempo real embarcados e de requisitos mais exigentes quanto ao consumo de energia, à portabilidade, ao desempenho, à confiabilidade e ao tempo de projeto, cresce a necessidade por novas metodologias, ferramentas e paradigmas para a concepção desses sistemas.

Sistemas embarcados podem ser definidos como sistemas de processamento de informações embarcadas. Tais sistemas têm um grande número de características comuns, inclusive restrições de tempo real, requisitos de confiança e também eficiência. A tecnologia

de sistema embutidos é essencial para prover informações onipresentes, uma das metas da tecnologia da informação moderna [40].

O projeto de sistemas de tempo real possue funcionalidades implementadas em *software* e/ou em *hardware*, onde, eficiência e flexibilidade são duas propriedades importantes para aplicações embarcadas dependendo dos requisitos e das restrições impostas pela aplicação. Para eficiência, pode-se implementar mecanismos para despachar tarefas com pequeno tempo e um contador de tempo com alta resolução. Para flexibilidade, prover um bom sistema de escalonamento [41].

Sistemas de tempo real embarcado têm tantas possibilidades de utilização que são até encontrados em eletrodomésticos e brinquedos. Uma pequena amostragem dos domínios de tempo real e suas aplicações é demonstrada na Tabela 3.2.1.

Domínio	Exemplos
Aviação	Navegação
	Painéis
Multimídia	Jogos
	Simuladores
Medicina	Cirurgia por robô
	Cirurgia remota
	Imagens médicas
Sistemas industriais	Linhas de montagens robóticas
	Inspeção automatizada
Civil	Controle de elevador
	Sistemas automotivos

Tabela 3.2.1: Alguns exemplos típicos dos domínios de sistemas de tempo real [42].

Os sistemas de tempo real embarcados possuem um grande espaço de projeto arquitetural onde existem diversas opções que podem ser exploradas, sempre levando em consideração os requisitos da aplicação. Uma solução que vem ganhando popularidade é uso de sistemas inteiros integrados em uma única pastilha, também conhecidos como *Systems-on-Chip* (SoC). Os SoCs tem ganhado popularidade devido ao crescimento da necessidade de mais desempenho em diversos domínios de aplicação como as aplicações de multimídia, dispositivos portáteis, entre outros. Os sistemas computacionais presentes em um SoC, podem ser dos mais variados tipos como microcontroladores, microprocessadores, circuitos integrados específicos para a aplicação (*Application Specific Integrated Circuits* ou ASIC) e conversores A/Ds e D/As.

3.3 Classificação dos Algoritmos de Escalonamento

O escalonamento de tarefas é designação de tarefas para o processador e recursos relacionados a um sistema operacional. Em sistemas de tempo real, todas as tarefas são definidas por suas especificações de retardo, prazo final, tipo (periódicos ou recursos esporádicos), e recursos exigidos. As tarefas esporádicas são tarefas associadas a processamento dirigido a evento, como respostas para entradas de usuário. As tarefas periódicas são períodos de tarefas em intervalos regulares [43].

Algoritmos de escalonamento podem ser classificados de acordo com vários critérios. Ilustração 3.3.1 demonstra as classificações possíveis.

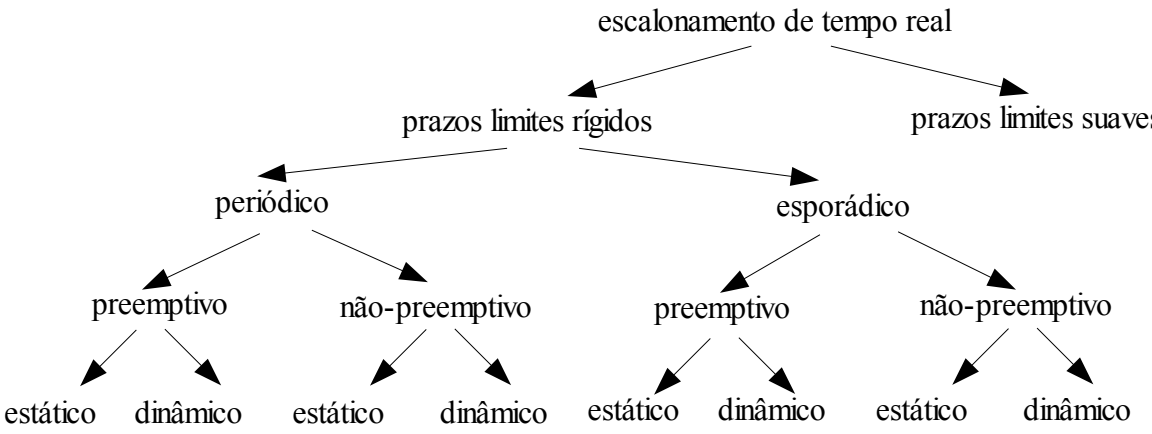


Ilustração 3.3.1: Classes de algoritmos de escalonamento [40].

A seguir a descrição detalhada dos elementos da Ilustração 3.3.1:

- Prazos limites rígido e suave: Escalonamento de prazo limite suave é frequentemente baseado em sistemas operacionais padrão. Por exemplo, o serviço de prioridades de tarefas e chamadas de sistema podem ser suficientes para atender seus requisitos.
- Escalonamento periódico e esporádico: As tarefas que devem ser executadas em "p" unidades de tempo são chamadas tarefas periódicas, e "p" é chamado seu período. As tarefas que não são periódicas são chamadas esporádicas.
- Escalonamento preemptivo e não-preemptivo: escalonamento não-preemptivo é baseado na execução de tarefas até seu fim. Como a resposta a um resultado para eventos externos pode ser bastante longa se alguma tarefa tem um grande tempo de execução, escalonamento preemptivo deve ser utilizado.

- Escalonamento estático e dinâmico: no escalonamento dinâmico as decisões são tomadas em tempo de execução e são bastante flexíveis, mas agregam custos (*overhead*) em tempo de execução. Em escalonamento estático as decisões são tomadas em tempo de projeto, podendo planejar o começo dos tempos das tarefas e gerar a tabela destes tempos para um simples escalonador.

3.4 Escalonamento Preemptivo

Escalonamento Preemptivo é o método mais comum utilizado no escalonamento de tarefas e é a maior vantagem em se utilizar RTOS, pois uma tarefa é executada até o seu término ou até que uma tarefa de maior prioridade preemptive a de menor prioridade [14,42].

As tarefas sob RTOS podem estar prontas ou não prontas. O RTOS mantém uma lista de tarefas que estão prontas e suas prioridades para execução. Uma tarefa pronta é adicionada a lista de tarefas e executada em seqüência. Quando uma tarefa ficar não pronta, é removida da lista.

3.5 Evento-Dirigido ao Escalonamento

Uma tarefa pode ser adicionada ou removida de uma lista de tarefas baseando-se em circunstâncias, tais como evento-dirigido ao escalonamento. Juntamente com escalonamento preemptivo, é o método mais utilizado em muitos RTOS. O escalonamento preemptivo representa um modelo mais próximo do mundo real, pois em inúmeros casos podemos citar exemplos em que determinadas tarefas deixam de ser momentaneamente atendidas para dar lugar a outras de maior prioridade [14].

Em sistema de evento-dirigido com preempção, um evento como o de interrupção ou uma tarefa podem determinar que alguma outra tarefa precisa ser ativada. Uma tarefa pode fazer isto, por exemplo, configurando um semáforo ou colocando dados em uma mensagem. A tarefa, que estava previamente na lista de tarefas para ser ativada pelo RTOS quando este evento acontecer, é ativada se tem uma prioridade mais alta que a tarefa corrente.

3.6 Multiprocessamento

É o processo de escalonar tarefas que pareçam operar simultaneamente. Todas as funções das tarefas ativação/desativação, escalonamento e prioridades são parte da função multiprocessamento. Nenhuma destas operações exigidas para multiprocessamento são implementadas em um simples programa de código seqüencial [5].

Multiprocessamento maximiza a utilização da CPU e também promove a construção modular das aplicações. Um dos aspectos mais importantes de multitarefa é que permite ao programador administrar complexidades inerentes em aplicativos de tempo real. Geralmente é mais fácil de projetar e manter aplicações de programas se utilizar multiprocessamento [14].

3.6.1 Taxa Monotônica (*Rate Monotonic* ou RM)

Para garantir o multiprocessamento com diversas tarefas, é importante realizar a análise de escalonabilidade em escalonamento de prioridade fixa (que é o caso do RTOS utilizado nesta dissertação) utilizando taxa monotônica.

A taxa monotônica é a atribuição de prioridades no processo de execução de tarefas onde uma tarefa com menor período recebe prioridade mais alta. É ideal quando o prazo final da tarefa for igual ao seu período [44]. A seguinte condição deve ser atendida para o escalonamento correto de uma tarefa, consistindo no conjunto de "n" tarefas periódicas P1,...,Pn, ver equação (3.6.1.1):

$$U = \Sigma_i^n \frac{C_i}{P_i} \leq n*(2^{\frac{1}{n}}-1) \qquad (3.6.1.1)$$

Onde "U" representa a utilização total do conjunto "n" de tarefas "i", C_i o caso de pior tempo de execução (*Worst Case Execution Time* - WCET) e "P_i" o período da tarefa "i" (somente para tarefas periódicas ver seção 3.3 na página 45).

Esta condição é satisfeita se a utilização total do conjunto de tarefas ($(\Sigma_i^n \frac{C_i}{P_i})$) for menor que o limite fixado de utilização ($n*(2^{\frac{1}{n}}-1)$), o escalonamento é garantido para o conjunto de tarefas através da atribuição de prioridade de taxa monotônica.

3.7 Mudança de Contexto

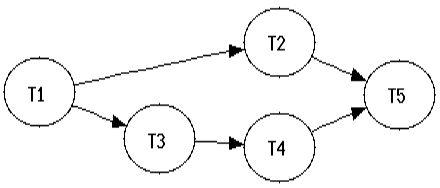
Mudança de contexto é o processo de salvar e restaurar dados suficientes para uma tarefa de tempo real e poder retornar depois de ser interrompida. Geralmente o contexto é salvo na pilha da estrutura de dados. A mudança de contexto contribui para tempo de resposta e por isso deve ser minimizado [42]. Uma quantia mínima de informações devem ser salvas para assegurar a restauração de qualquer tarefa depois de ser interrompida, tais como:

- Conteúdo dos registradores de uso geral;
- Conteúdo do contador do programa;
- Conteúdo dos registradores do co-processador (se presente);
- Registradores da página de memória;
- Posições de E/S das imagens mapeadas de memória.

3.8 Grafo de Tarefas

Com a utilização de fluxogramas, as ferramentas simplificam o processo de criação e manutenção de programas. Um problema em usar fluxogramas para documentação do código é a dificuldade de mostrar os efeitos das interrupções. Embora os fluxogramas tendem a não acompanhar as mudanças no código (a mudança no código implica um novo fluxograma), são uma boa escolha para mostrar a execução de códigos seqüenciais [14].

A relação mais óbvia entre tarefas é sua dependência causal: Muitos processos podem ser executados somente quando outras tarefas terminaram. Esta dependência é tipicamente demonstrada em grafos de dependência. A Ilustração 3.8.1 exibe à esquerda um grafo de dependência para um conjunto de processos ou tarefas e à direita o código utilizado para gerar este grafo no Sistema Operacional Linux.




```
echo "digraph G {
rankdir="LR";T1->T2;T2->T5;
T1->T3;T3->T4;T4->T5;}" |
dot -Tpng > ex_piccoli.png
```

Ilustração 3.8.1: Grafo de dependência [40].

Informações de temporização: As tarefas podem ter tempos de chegada, prazos limites, períodos e tempos de execução. A fim de tomar isto em conta em escalonamento de tarefas, estas informações são representadas de forma útil em grafos de tarefa. Possíveis intervalos de execuções são demonstrados na Ilustração 3.8.2. Assumindo-se que as tarefas T1 a T3 são independentes, embora combinações mais complexas de temporização e relações de dependência possam existir [40].

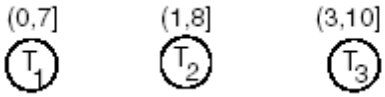


Ilustração 3.8.2: Grafo de tarefas incluindo informações de temporização [40].

Os sistemas de tempo real geralmente são modelados como uma coleção de tarefas independentes, onde cada tarefa gera uma seqüência de trabalho, cada um é caracterizado por tempo pronto, requisitos de execução, e tempo limite. A análise de escalonabilidade de tal sistema ocupa-se em determinar se é possível atribuir para cada tarefa um tempo de processamento equiparado ao seu requisito de execução, entre seu tempo pronto e seu prazo final onde a qualidade de resultado de uma tarefa depende somente do tempo despendido pela tarefa para produzir o resultado [45,46].

Tarefas de processamento de tempo real caracterizam-se por parâmetros determinísticos tais como: tempo de computação “e” (quantidade de processamento), tempo pronto “s” (tarefa disponível para processamento), e prazo final “ [u,l]” [47,48,49]. Restrições adicionais podem ser adicionadas às tarefas, como a freqüência máxima da tarefa que requisita serviço periódico ou uma relação de precedência definida para um conjunto de tarefas [50].

Na fórmula (3.8.1) tem-se um conjunto de variáveis, onde a variável “s”, corresponde ao tempo do começo da tarefa “i”, enquanto a variável “e” corresponde ao tempo de execução real, medida para a tarefa “i” [49].

$$\{s_1, e_1, s_2, e_2, \cdots, s_n, e_n\} \in \mathbb{R} \qquad (3.8.1)$$

Na fórmula (3.8.2), tem-se um conjunto de constantes, onde a constante "l_i", corresponde ao tempo de execução mínima da tarefa "i", e "u_i", corresponde ao tempo de execução máximo da tarefa "i".

$$\{u_1, l_1, u_2, l_2, \cdots, u_n, l_n\} \subseteq \mathbb{R}^+ \qquad (3.8.2)$$

Logo:

$$(e_i \in [l_i, u_i]) \qquad (3.8.3)$$

3.9 Grafos de Fluxo de Controle

Um aplicativo de análise de cobertura de tarefas é normalmente associado ao uso de controle e modelos de fluxo de dados para representar elementos e dados de programas estruturais [51]. Os elementos de lógica mais considerados na cobertura são baseados no fluxo de controle de uma unidade de código. Por exemplo:

i. Declarações do programa;
ii. Decisões/desvio (influência no fluxo de controle do programa);
iii. Condições (expressões de avaliação falso/verdadeiro)
iv. Combinações de decisões e condições;
v. Caminhos (seqüências de nós no grafo de fluxo).

Todos os programas estruturados podem ser construídos de três maneiras básicas: seqüenciais (atribuições de instruções), decisão (declarações de *if/then/else*) e interativo (instruções de *while*, laços de *for*). As representação gráficas para estes três princípios são mostradas na Ilustração 3.9.1.

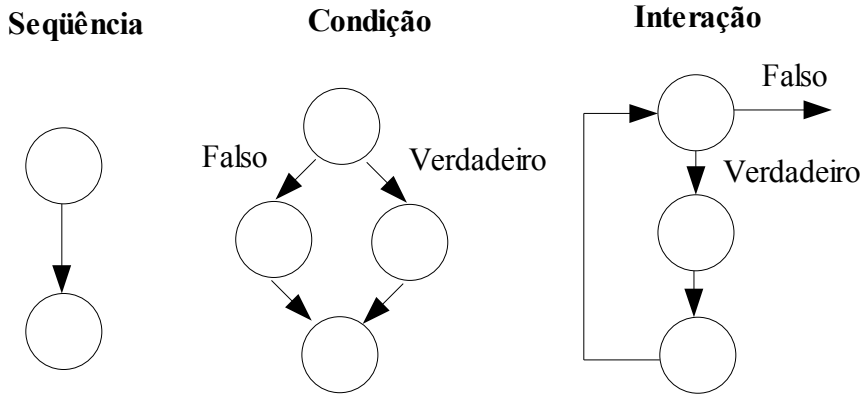


Ilustração 3.9.1: Principais representações de programa [51].

Existem ferramentas comerciais que geram grafos de fluxo de controle do código. O desenvolvedor de *software* pode utilizar estas ferramentas para gera grafos de fluxo de controle especialmente em partes complexas do código do programa.

Uma representação do fluxo de controle para o *software* submetido a testes facilita a concepção de casos de teste em código aberto, pois são claramente mostrados a cobertura dos elementos lógicos necessários para os critérios de escolha dos casos de teste [51].

3.10 Software Robusto

A robustez é definida como o grau que um sistema opera corretamente na presença de exceções de entradas ou condições de tensões ambientais. A robustez pode ser visualizada como uma indicação da capacidade do Sistema Operacional para resistir/reagir a falhas induzidas pelos aplicativos, ou falhas originárias na camada de *hardware* ou nos *drivers* dos dispositivos [52].

Quando entradas inválidas são detectadas, como mostrado na Ilustração 3.10.1, várias opções podem ser tomadas pelo *software* robusto [53]:

- Solicitar uma nova entrada (a origem de entrada pode ser um operador humano);
- Usar o último valor aceitável para a variável de entrada(s) em questão;
- Usar um valor predefinido para a entrada.

Depois da detecção e tolerâncias iniciais da entrada inválida, o *software* robusto cria um indicador (*flag)* de exceção indicando a necessidade para outro programa tratar a condição de exceção.

Uma vantagem do *software* robusto é que ele permite a proteção contra problemas predefinidos, desde que, as possíveis entradas com erros sejam descobertas cedo já na etapa de processo de desenvolvimento e teste. Uma desvantagem é que ele não abrange falhas induzidas por entradas erradas que não sejam especificadas.

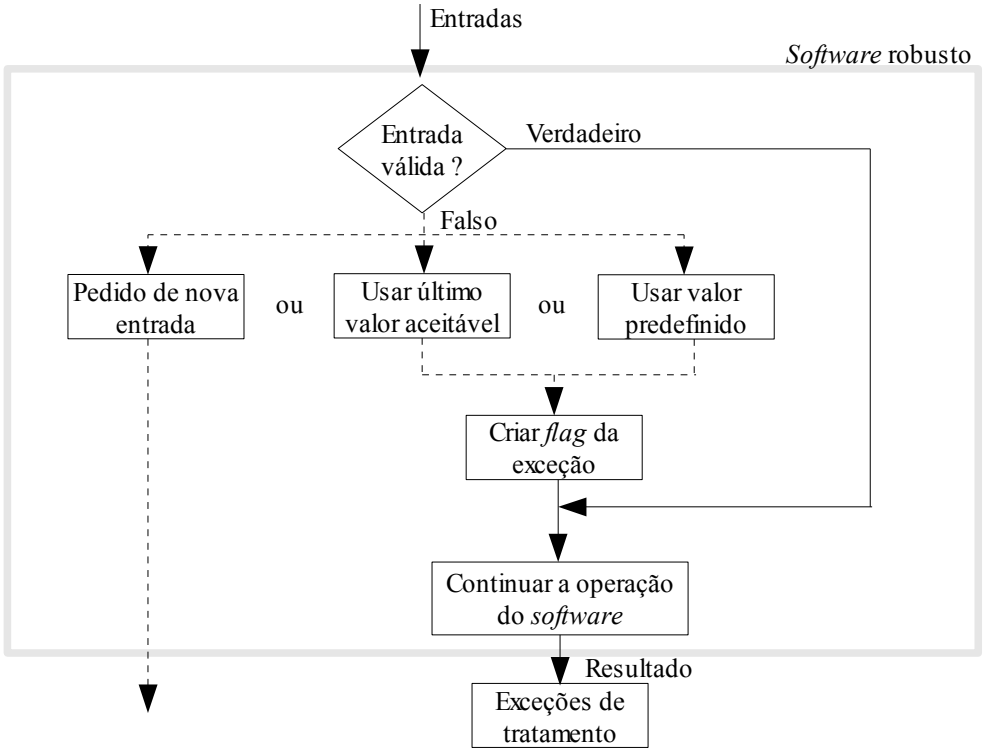


Ilustração 3.10.1: Operação robusta de *software* [53].

4 INJEÇÃO DE FALHAS

4.1 Introdução

O uso de técnicas de tolerância a falhas em sistemas com característica tempo real exige que as mesmas sejam validadas. Na validação por injeção de falhas, são testados os mecanismos de tolerância a falhas, possibilitando a observação do comportamento do sistema frente aos estímulos provocados pelas falhas injetadas. A injeção falhas pode ser de forma controlada, permitindo a aceleração da ocorrência das falhas permitindo diminuir a latência da falha e do erro e aumentando a probabilidade de uma falha causar um defeito [54].

Falhas de *hardware*, transientes ou permanentes, que afetam um computador isolado podem se propagar pela rede em um sistema distribuído afetando vários outros nós desta rede. Nesses casos para evitar se deter no detalhe de um nodo, usa-se uma abstração de maior nível que classifica as falhas em sistemas distribuídos como falhas de comunicação onde um componente não recebe um serviço solicitado na maneira especificada [55].

As falhas de *software* ocorrem normalmente por erros de especificação de projeto ou de codificação. Muitas destas falhas podem ficar latentes e só ocorrerem durante a operação do sistema, especialmente quando sob pesadas ou não usuais cargas de trabalho ou sob determinados intervalos de tempo. Os erros de natureza não-determinísticas são os mais difíceis de se eliminar por verificação, validação ou teste, e, normalmente, os sistemas os contêm durante todo o seu ciclo de vida.

Considerando-se que as falhas podem ter as mais diversas origens, a injeção de falhas se constitui numa importante forma de avaliar e validar os mecanismos de tolerância a falhas em sistemas de computadores, pois oferece vantagens pela habilidade de fazer medidas focadas em problemas específicos de interesse [54].

4.2 Objetivos da Injeção de Falhas

Com injeção de falhas, as falhas podem ser aplicadas durante a execução de uma aplicação alvo, assim um Monitor fornece dados sobre o comportamento da aplicação em presença das mesmas. A análise desses dados indica se a aplicação atende à especificação, ou seja, se realmente mascara ou recupera-se das falhas a que se propôs na fase de projeto [56].

Neste sentido, pode-se afirmar que a técnica de injeção acelera a ocorrência das falhas em um determinado sistema. Com isso, ao invés de esperar pela ocorrência espontânea das falhas, pode-se introduzí-las intencionalmente, controlando, por exemplo, o tipo, a localização e a duração das memas.

No caso de previsão de falhas, as medidas correspondem a ocorrência de estados de sistema, a duração que este sistema permanece neste estado e a freqüência/latência de alterações de estado. As medidas com relação a previsão de falhas consistem de medidas estatísticas caracterizando a Dependabilidade do sistema sob teste.

A Ilustração 4.2.1 mostra as transições de estado que descrevem o comportamento esperado de um sistema em resposta à injeção de uma falha. Uma falha injetada pode ou não ser ativada. Quando ativada, a falha pode ser mascarada, detectada ou não detectada. Falhas mascaradas pelos mecanismos de tolerância a falhas do sistema alcançam o estado bom, enquanto falhas não detectadas alcançam o estado ruim. Falhas detectadas podem ser recuperadas ou não. Quando recuperadas, as falhas atingem o estado bom, caso contrário atingem o estado ruim.

A seguir são descritas as definições de cada ação de falha, de acordo a Ilustração 4.2.1 [23]:

- Não ativada: ao ser injetada a falha, nenhuma ativação é observada em um determinado tempo de observação (volta para o estado ocioso);
- Ativada: a ativação da falha é observada. O tempo transcorrido entre a injeção e a ativação é registrado como dormência;
- Mascarada: apesar da ativação da falha, o sistema reage como se não houvesse erro (transição direta para o estado bom);
- Detectada: após a ativação da falha, os mecanismos de detecção de erros detectam um erro como conseqüência da falha injetada. O tempo decorrido entre a ativação e a detecção é registrado como latência;
- Não detectada: após a ativação da falha, o sistema exibe algum mal funcionamento sem ter detectado qualquer erro (transição direta para o estado ruim);

- Recuperada: se as ações de recuperação realizadas após a detecção da falha, tiverem sucesso então a transição para o estado bom ocorre;
- Não recuperada: quando as ações de recuperação não têm sucesso dentro do tempo de observação é diagnosticada uma falha do sistema (transição para o estado ruim).

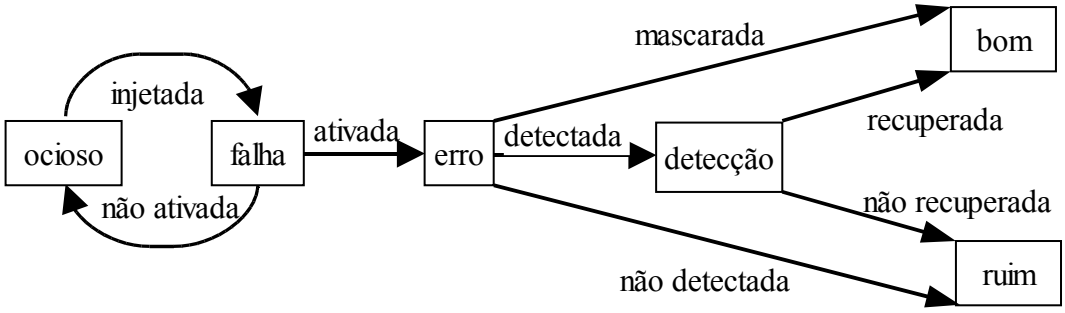


Ilustração 4.2.1: Grafo da falha [23].

4.3 Arquitetura de um Ambiente de Injeção de Falhas

Uma arquitetura genérica para um ambiente de injeção de falhas é demonstrado na Ilustração 4.3.1 [56]. Um ambiente de injeção de falhas consiste basicamente em:

- Controlador: responsável por coordenar todo o ambiente, normalmente é um computador que gerencia o processo de testes;
- Gerador de carga de trabalho: responsável pelo fornecimento de comandos, os quais devem ser executados pelo sistema alvo, conforme consta na biblioteca de carga de trabalho;
- Monitor: observa o sistema em teste, executa instruções quando necessário e faz a interface entre o controlador e o sistema sob teste, disparando a coleta de dados sempre que necessário;
- Injetor de falhas: utilizado via *software* ou *hardware* para injeção das falhas, emulando a presença de falhas no sistema alvo;
- Sistema alvo: sistema sob teste onde as falhas serão injetadas e monitoradas;
- Coletor de dados: recolhe os dados sobre o sistema;
- Analisador de dados: responsável por processar e analisar os dados coletados, podendo trabalhar *off-line* (não precisa trabalhar em tempo real).

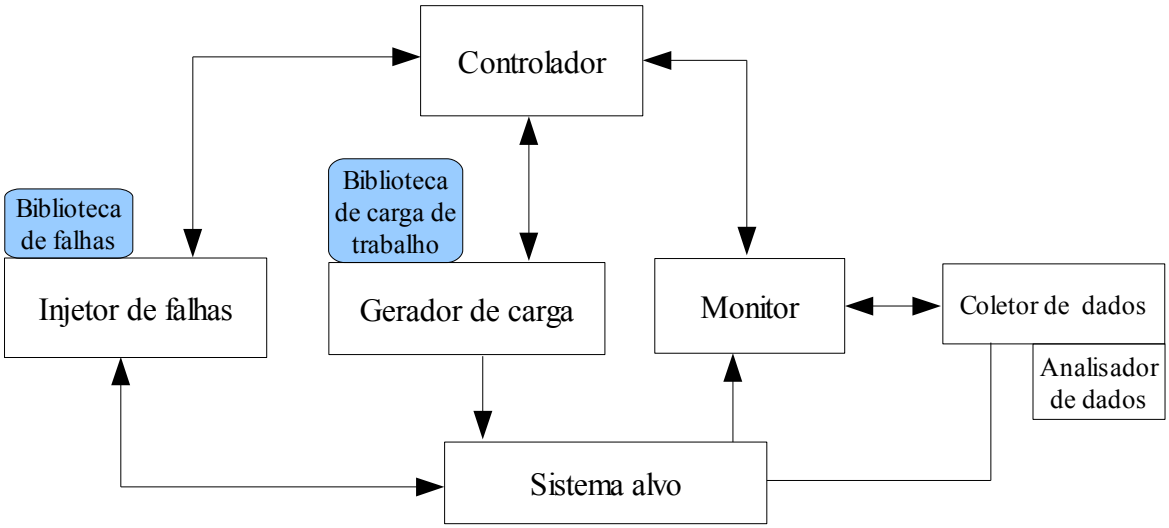


Ilustração 4.3.1: Arquitetura de um Ambiente de Injeção de Falhas [56].

O injetor de falhas, que pode ser implementado por *hardware* ou por *software*, pode suportar diferentes tipos, localizações e tempos de falhas.

4.4 Atributos das Técnicas de Injeção de Falhas.

A técnicas de injeção de falhas são baseada em cinco atributos descrevendo as características de cada uma [57,58,59], a saber: controlabilidade (com respeito ao espaço e tempo), portabilidade, repetibilidade, alcance físico, e a possibilidade de medida de tempo (por exemplo, detecção de latência de erro):

1. Controlabilidade: Pode ser dividida em duas variáveis: tempo e espaço. No domínio do espaço, é relatada a habilidade de controlar o local onde as falhas são injetadas. O domínio do tempo corresponde ao controle do instante do tempo no qual as falhas são injetadas;
2. Portabilidade: O quanto é necessário modificar um injetor de falhas para que o mesmo possa injetar falhas em um novo sistema em teste. Quanto maior for a portabilidade de um ambiente de injeção de falhas, menor é a necessidade de alterações na realização de testes em um novo sistema;
3. Repetibilidade: Refere-se à habilidade de reproduzir resultados estatísticos ou idênticos para um dado ambiente de testes. Uma alta reprodutibilidade permite resultados iguais ou semelhantes, gerados em um mesmo ambiente. Isto é necessário para assegurar a credibilidade dos testes realizados;

4. Alcance físico: Habilidade de acessar determinadas posições de um sistema para gerar possíveis falhas. Exemplos de locais no sistema em teste que podem gerar falhas: memória, pinos, registradores, etc;
5. Medida de tempo: A velocidade da aquisição das informações associada aos eventos monitorados (por exemplo: medida de detecção de latência de erro) é um atributo importante nas experiências de injeção de falhas.

4.5 Classificação da Injeção de Falhas

Técnicas de injeção de falhas têm sido amplamente utilizadas para avaliar as características de Dependabilidade dos sistemas ou simplesmente para validar determinados mecanismos de manipulação de falhas. A injeção de falhas pode ser implementada por simulação ou em sistemas reais, podendo esta última ser classificada em injeção de falhas em *hardware* e injeção de falhas por *software*.

4.5.1 Injeção de Falhas por Simulação

Na injeção de falhas aplicada a modelos de simulação, falhas/erros são introduzidos em um modelo do sistema. Uma vantagem desta abordagem é poder ser aplicada durante a fase de desenvolvimento, o que facilita a detecção de erros de projeto [60].

Tem como característica a habilidade para controlar e monitorar falhas injetadas, através da inserção de falhas em tempo de execução, controlando o local, tempo, duração e a rapidez no acesso aos resultados. A facilidade na sincronização das falhas com o estado do sistema e a visualização da propagação de falha nas atividades das diversas unidades funcionais, tornam a injeção de falhas aplicada a modelos de simulação uma alternativa bastante atrativa [60].

4.5.2 Injeção de Falhas em Hardware

Devido a complexidades envolvidas, os testes por injeção de falhas em sistemas baseados em simulação não podem ser totalmente validados, necessitando assim uma abordagem bastante comum de injeção de falhas que consiste na injeção de falhas físicas no *hardware* do sistema real [61]. Esta abordagem de gerar falhas de *hardware* reais, apresenta vantagem, pois torna-se mais próxima de um modelo de falhas reais.

Os métodos podem ser caracterizados de maneira global: métodos com contato e sem contato [61]. Diversos métodos têm sido utilizados, tais como:

1. Injeção de falha no nível do pino: Caracteriza-se pela injeção de falhas diretamente nos pinos do circuito integrado. Pode prover controle total da injeção de falhas, porém surge dificuldade na sincronização da injeção da falha com a atividade do sistema em gerar os padrões de erros causados por falhas no interior dos circuitos integrados Esta abordagem tem duas implementações distintas para gerar as variações de tensão ou corrente nos pinos [57,58,59]:
   1.1. Forçando: A falha é aplicada diretamente por meio de múltiplas ponteiras nos pinos do circuito integrado conectadas na linha de alimentação, modificando a tensão do circuito. Deve-se ter cuidado, pois variações grandes de tensão podem danificar o circuito em teste. Esta técnica foi explorada em base nas normas IEC 61000-4-29 e IEC 61000-17 [11] por (Vargas, 2005) em [62,63].
   1.2. Inserção: O circuito integrado em teste é removido do sistema e inserido em uma caixa onde o chaveamento transistorizado asseguram o isolamento adequado do circuito integrado em teste do sistema.
2. Injeção de falha embutida (*built-in fault injection*): Esta abordagem envolve incorporar a injeção de falhas no sistema, permitindo assim, boa controlabilidade no domínio do espaço e tempo [64]. Um módulo I-IP (*Infrastructure Intellectual Property*) pode ser desenvolvido e acoplado ao sistema para prover esta técnica que com sucesso acrescentou a eficiência e qualidade do teste do sistema em grandes computadores.
3. Distúrbios externos: nesta linha de estudo, há dois tipos de técnicas: Interferência eletromagnética e radiação.
   3.1. Interferência Eletromagnética (*Electro-Magnetic Interference* ou EMI): Uma classe importante de defeitos no computador são causados por interferência

eletromagnética (EMI). Tais perturbações são comuns em motores de carros, trens e em plantas industriais [57,58].

O injetor de falhas não tem contato físico direto com o sistema em teste, a injeção de falhas é realizada através de ondas eletromagnéticas que ocasionam mudanças de corrente dentro do sistema alvo em teste.

Como exemplo de injeção de falhas, podemos citar a pesquisa realizada em [63,65], onde injetou-se falhas em um sistema embarcado utilizando uma célula GTEM (*GigaHertz Transverse Eletromagnetic*), que pode ser encontrada em laboratórios para certificação de EMC (*Eletromagnetic Compatibility*) de produtos eletrônicos. A ferramenta de injeção de falhas demonstra bastante controle em função do sinal portador de frequência, da modulação da frequência do sinal, da amplitude e da potência.

3.2. Radiação (*Heavy-Ion Radiation* ou HIR):

Uma característica importante que difere das demais técnicas, é que na injeção de falhas por radiação pode ser inserida nos circuitos VLSI. Assim, as falhas injetadas geram grande variedade de padrões de erro [57,58]. A injeção de falha deve ser realizada através do uso de um acelerador de partículas, em câmara a vácuo com o encapsulamento do circuito integrado removido, pois os íons são facilmente atenuados pelo ar [21].

A radiação através do Californium-252 pode ser utilizada para injetar (SEUs), isto é, *bit-flips* em posições internas dos circuitos integrados. As partículas de íons são atenuados pelas moléculas do ar e outros materiais. Por isso, a radiação no circuito sob interferência deve ser realizada em uma câmara a vácuo (Ilustração 4.5.2.1). Este circuito deve ter seu encapsulamento superior retirado para facilitar a radiação, e a distância entre o mesmo e a fonte de Cf-252 deve ter aproximadamente 35mm [57].

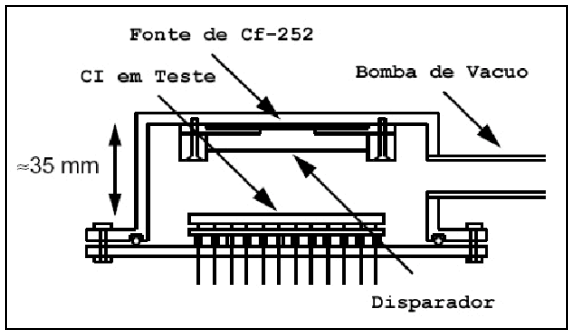


Ilustração 4.5.2.1: Câmara a vácuo utilizada para injeção de HIR [57].

Este método apresenta desvantagens, tais como: necessidade de proteção contra a radiação para cada um dos componentes da equipe de trabalho, dificuldade para encontrar o local onde a falha foi gerada, risco alto de danificação do sistema em teste, e um grande dispêndio de tempo na execução dos testes.

4.5.3 Injeção de Falhas por Software

Uma abordagem é emular a injeção de falhas por *hardware* com injeção de falhas por *software***,** corrompendo o estado de execução do programa, de forma que o comportamento do sistema apresente uma falha interna em tempo de execução. Deste modo, o objetivo da injeção de falhas por *software* é modificar o estado do *hardware/software*, que está sob controle do *software*, levando o sistema a se comportar como se falhas de *hardware* estivessem presentes [66].

Injeção de falhas por *software* consiste, usualmente, na interrupção da execução da aplicação e na execução do código do injetor de falhas. O injetor de falhas permite emular falhas em diferentes partes do sistema, como por exemplo alteração do conteúdo de registradores e posição de memória, alteração da área de código e sinalizadores (*flags*) [66].

Em sistemas com característica de tempo real, onde as aplicações por eles controladas devem ocorrer em instantes de tempo relativos a uma base de tempo externa ao sistema, a validação por injeção de falhas se torna mais crítica. As restrições temporais impostas pelos sistemas aliadas a sobrecarga provocada pelo injetor de falhas justificam este fato. Neste sentido, a execução do injetor de falhas interfere na característica de temporização do sistema, prejudicando o teste em funções de tempo críticas [61].

No contexto de ativação da injeção de falhas por *software*, pode-se distinguir duas categorias de injetores de falhas por *software*: durante tempo de compilação e durante tempo de execução (*runtime*) [56]:

a) Durante tempo de compilação:

   Com injeção de falhas durante tempo de compilação as instruções da aplicação alvo são modificadas antes da imagem do programa ser carregada e executada para emular o efeito de falhas transientes em *hardware*/*software*. As modificações são realizadas estaticamente podendo-se utilizar instruções alternativas responsáveis pela injeção de falhas. Com isso, não há interferência na carga de execução, uma vez que o erro somente torna-se ativo quando as instruções ou os dados alterados são alcançados [56].

b) Durante tempo de execução:

   Mecanismos utilizados para ativar a injeção de falhas:

   - Intervalo: um temporizador atinge um tempo predeterminado, ativando a injeção de falhas.
   - Exceção: neste caso, uma exceção de *hardware* ou um sinal *trap* de controle de *software* para ativar a injeção de falhas.
   - Inserção de código: instruções são adicionadas ao programa, modificando suas instruções de modo que a injeção de falha será ativada quando estas instruções são acessadas durante a execução do programa.

5 TÉCNICAS DE DETECÇÃO DE ERROS VIA SOFTWARE

5.1 Introdução

Em SoCs, durante seu período de funcionamento, falhas transientes e permanentes podem causar a execução incorreta da seqüência de instruções do processador, ou seja, falhas de controle de fluxo. Assim, para evitar que estas falhas sejam propagadas e gerem saídas incorretas, aconselha-se a utilização de técnicas específicas capazes de monitorarem em tempo real os dados manipulados e o fluxo de execução do programa.

Os processadores de VLSI de propósito geral estão sendo muito utilizados em aplicações críticas, em que a habilidade de descobrir falhas transientes é essencial. Muitos destes sistemas contêm um grande número de processadores e trabalham em ambientes relativamente severos. Nestes sistemas, falhas transientes ou erros devido a falhas intermitentes podem ser bastante numerosas quando comparadas a falhas permanentes. Conseqüentemente, existe muito interesse no desenvolvimento de testes concorrentes, ou monitoração, técnicas para detecção de erros [67].

Componentes comerciais de microprocessadores *Off-The-Shelf* (COTS) são extensamente utilizados em *System-On-Chip* (SoC) em projetos de sistemas embarcados. Neste caso, certas técnicas como as que utilizam redundância de *hardware* podem ser de altos custos [68].

Neste contexto, a fim de garantir eficientemente a detecção das falhas que alterem o fluxo de controle em SoCs, surgem as metodologias de detecção de erros de hardware implementadas via software (*Software Implemented Hardware Fault Tolerance* - SIHFT). Técnicas SIHFT são classificadas como um tipo de redundância de software e representam uma alternativa extremamente viável devido ao baixo custo associado a sua implementação.

As soluções baseadas em *software* apresentam vantagens como:

- Tem boa portabiliade, ou seja, podem ser implementadas em múltiplas plataformas, pois são aplicadas em alto nível;
- Representam soluções implementadas puramente em software.

A maioria das soluções via *software*, definidas para detecção de erros de fluxo de controle baseiam-se fundamentalmente na divisão do código do programa em blocos básicos e na utilização da notação, abaixo descrita, para representá-lo [68]:

P = {V,E}: representa o grafo do programa;

V = {*vi*, *i* = 1, 2, .., n}: representa o conjunto de blocos básicos;

E = {*ei*, *ej*, ..., *ek*}: representa as linhas de união entre os blocos básicos (conjunto de desvios);

vi: representa o bloco básico *i*;

ei: representa a linha de união entre os blocos básicos (desvios);

bri,j: desvio entre *vi* e *vj*;

suc(*vi*): conjunto de nós sucessores do nó vi;

pred(*vi*): conjunto de nós predecessores do nó vi.

Assim, um programa P = {V,E}, onde V = {*v1*, *v2*, *v3*, ..., *vn*} e E = {*e1*, *e2*, e3, ..., *em*}.

Cada nó *vi* representa um bloco básico e está associado a um conjunto de sucessores e predecessores, representados por *suc*(*vi*) e *pred*(*vi*) e cada linha *ei* representa um desvio *bri,j*, ou seja um desvio entre *vi* e *vj*.

Nas seções seguintes, serão descritas diferentes metodologias capazes de detectarem falhas de fluxo de controle propostas na literatura, com o objetivo de detectar erros provinientes de pertubações externas como citado na seção 2.5 na página 45.

5.2 Técnica BSSC (*Block Signature Self-Checking*) e ECI (*Error Capturing Instructions*)

Na técnica BSSC, um programa é dividido em blocos básicos e para cada um é atribuído uma assinatura. A verificação é realizada durante a execução do programa através do cálculo de assinaturas nas entradas e saídas de cada básico básico do código [69].

Na outra técnica, baseada em captura de instruções de erro (*Error Capturing Instructions* - ECIs), são inseridas instruções nas posições de memória principal que a CPU não utiliza durante uma execução normal. Deste modo a execução de um código de ECI é uma indicação que um erro de fluxo de controle ocorreu. Tanto as técnicas BSSC como ECI são implementados somente em *software*.

Todas as técnicas implementadas em *software* e apresentadas nesta dissertação para detecção de erros de fluxo de controle, incluindo as técnicas BSSC e ECI, são baseadas na hipótese de que a CPU continua a execução do programa mesmo depois de uma falha transiente ocorrer. No caso de uma parada forçada na execução do programa, seja pelo desvio do contador de programa (PC) para uma posição não utilizada pelo processador ou um laço infinito, estas técnicas podem ser complementadas com a utilização de um WDT para detectar erros de fluxo de controle.

Para realizar a detecção de erros no fluxo de controle do código, os blocos básicos são modificados como mostra a Ilustração 5.2.1. Uma instrução de chamada de entrada é inserida no início de cada bloco básico, uma instrução de chamada de saída e uma assinatura embarcada são inseridas ao final do bloco básico. A primeira instrução de chamada ativa uma rotina de entrada que armazena a assinatura do bloco em uma variável estática, em uma posição específica da memória. O retorno da rotina de entrada, isto é, o endereço da primeira instrução no bloco básico, é usado como assinatura. Depois de retornar da rotina de entrada, as instruções no bloco básico são executadas, sendo que a última instrução de chamada deste bloco ativa uma rotina de saída.

A rotina de saída compara a assinatura embarcada com a assinatura armazenada pela rotina de entrada na variável estática. Se houver diferença entre a assinatura embarcada com a assinatura na variável estática, um erro é detectado, a instrução de saída ativa uma rotina de exceção para iniciar um tratamento de recuperação. Se as assinaturas são iguais, a execução do código continua na partição de instrução seguinte, logo depois do bloco básico.

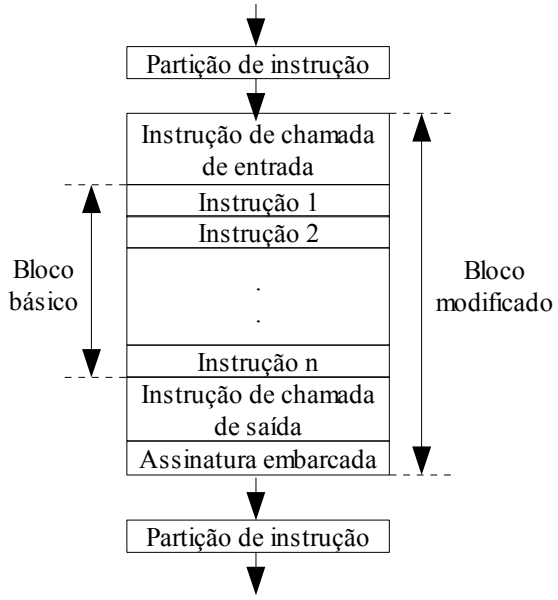


Ilustração 5.2.1: O mecanismo da técnica BSSC [69].

Quanto à cobertura de falhas, esta é capaz de detectar todos os erros de fluxo de controle simples tais como um desvio para dentro de um BFI ou um desvio para um BFI ilegal, ou seja, um desvio para um BFI que não pertence ao grupo de sucessores do BFI atual.

5.3 Técnica CCA (*Control Flow Checking by Assertions*)

Esta técnica consiste na divisão do código do programa em intervalos livres de desvio (BFIs) e na inserção de dois identificadores (identificador do intervalo livre de desvio - BID e identificador de fluxo de controle - CFID) a cada BFIs para verificação durante o período de execução do código [70].

Enquanto o identificador do intervalo livre de desvio (BID) armazena um valor único que identifica o BFI, o identificador de fluxo de controle (CFID) representa os desvios permitidos, indicando se os desvios de fluxo de controle estão ocorrendo na seqüência correta.

Para identificação da execução do código na ordem correta, o CFID é armazenado em dois elementos na fila. A fila é inicializada com o CFID do primeiro BFI. Na entrada do BFI, o CFID do próximo BFI é colocado na fila e na saída do mesmo o CFID é retirado da fila e verificado se o próximo elemento é o correto.

A Ilustração 5.3.1 mostra a estrutura "*if-then-else*" acrescida das instruções de verificação da técnica CCA.

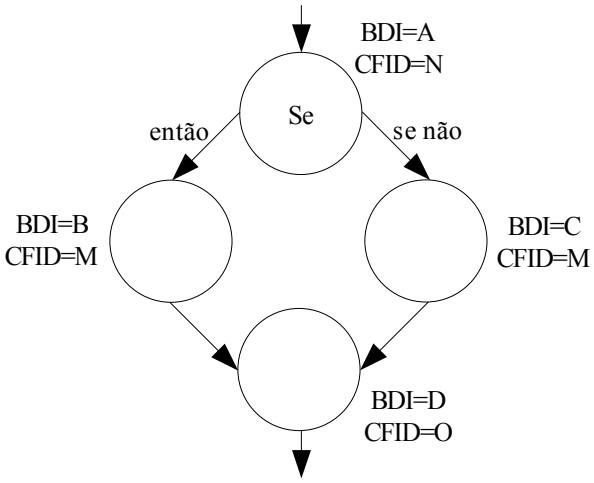


Ilustração 5.3.1: Instruções e verificação dos IDs
para estrutura *if-then-else* com CCA [70].

Quanto à cobertura de falhas, esta é capaz de detectar todos os erros fluxo de controle simples e a maioria dos erros múltiplos de fluxo de controle tais como um desvio para dentro de um BFI ou um desvio para um BFI ilegal, ou seja, um desvio para um BFI que não pertence ao grupo de sucessores do BFI atual.

5.4 Técnica ECCA (*Enhanced Control Flow using Assertions*)

Esta técnica é um aperfeiçoamento da técnica CCA e tem como objetivo a detecção de erros no fluxo de controle para sistemas de tempo real, por exigirem baixo custo (*overhead*) e baixa latência de detecção de erros [71,72]. Similarmente à CCA , a técnica ECCA consiste em inserir instruções de teste e atualização no código da aplicação a fim de dar subsídios para alguma aplicação posterior de tolerânica a falhas, consequentemente tornando mais confiável e robusto.

Para a realização da técnica, o programa é dividido em um conjunto de intervalos livres de desvio (BFI´s) e atribuído um número primo único, denominado identificador de intervalos livres de desvio (BID). Deve ser inserido para cada BFI uma instrução de teste (conforme a fórmula (5.4.1)) e no fim uma instrução de atualização (conforme a fórmula (5.4.2)).

$$id \leftarrow \frac{BID}{\overline{(id \bmod BID)}.(id \bmod 2)} \quad (5.4.1)$$

$$id \leftarrow NEXT + \overline{\overline{(id - BID)}} \qquad (5.4.2)$$

Onde: *id* representa a variável global atualizada durante o tempo de execução do algoritmo na entrada e na saída do BFI, ou seja, monitora o fluxo de execução do algoritmo; BID representa a assinatura de cada BFI armazenando um número primo único para cada BFI gerado em tempo de pré-processamento; NEXT representa o somatório de todos os BID dos BFI que podem ser sucessores do BFI atual gerado em tempo de pré-processamento.

Assim, a estrutura de um programa escrito em linguagem C é vista na Ilustração 5.4.1 após ser pré-processado pela técnica ECCA.

```
/* Inicio do código original*/
... BFI1 ...
se foo ... BFI2 ...
se não ... BFI3 ...
... BFI4 ...
/* Fim do código orignal */
```

Ilustração 5.4.1: Representação do código original [71].

A Ilustração 5.4.1 apresenta a estrutura de um programa em C dividido em BFIs. A Ilustração 5.4.2 representa o mesmo programa acrescido das instruções de controle definidas pela técnica, ou seja, após o pré-processamento. Finalmente a Ilustração 5.4.3 mostra o diagrama de bloco do mesmo programa.

```
/* Início do BFI */
id = <BID> / ((!(id%<BID>)) * (id%2));
… corpo do BFI …
id = <NEXT> + !!(id - <BID>);
/* Fim do BFI */
```

```
        /* Início do código pré-processado*/
... BFI1...   /* <BID> é 3 */
id = 35+!!(id-3);  /* NEXT = BFI 2*BFI 3 = 5*7= 35*/
se foo {
          id = 5 / ((!(id%5)*(id%2));  /* <BID> é 5 */
          ... BFI2 ...
          id = 11+!!(id-5);
} se não {
          id = 7/((!(id%7))*(id%2));  /* <BID> é 7 */
          … BFI3 …
          id = 11+!!(id-7);
}
id = 11/((!(id%11))*(id%2));  /* <BID> é 11 */
        ... BFI4 ...
        /* Fim do código pré-processado */
```

Ilustração 5.4.2: Representação do código tolerante a falhas de acordo com a técnica ECCA [71].

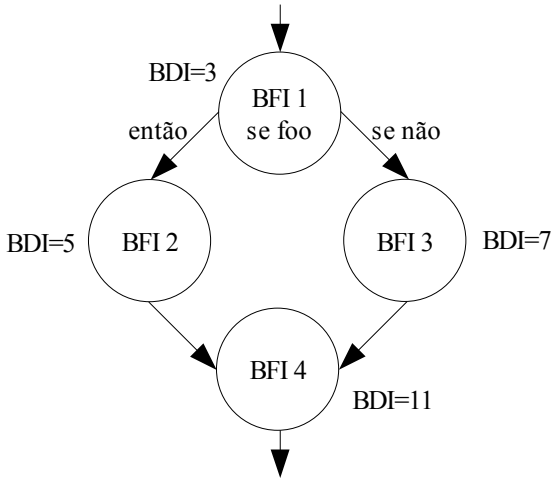


Ilustração 5.4.3: Diagrama de bloco do código tolerante a falhas de acordo com a técnica ECCA [71].

Quanto à cobertura de falhas, esta técnica apresenta a mesma capacidade de detecção da técnica CCA, ou seja, é capaz de detectar todos os erros de controle de fluxo simples e a maioria dos múltiplos. Entretanto, ECCA apresenta algumas vantagens em relação a CCA, tais como:

- ECCA apresenta um menor *overhead* em termos de espaço e tempo, pois são inseridas apenas duas linhas de código por BFI;
- ECCA utiliza apenas uma variável ao invés de duas filas e uma variável;

5.5 Técnica CFCSS (*Control Flow Checking by Software Signatures*)

A técnica de CFCSS utiliza basicamente a terminologia mencionada anteriormente e baseia-se fundamentalmente na divisão do programa em blocos básicos, na atribuição de assinaturas para cada um deles e no monitoramento das assinaturas em tempo de execução [73].

O seguintes passos são necessários para o monitoramento e verificação em tempo real do fluxo de controle do programa:

1. Identificar todos os blocos básicos (*vi*) construindo o grafo que representa o fluxo de execução do programa e numerar todos os nós do grafo;
2. Atribuir um valor único de assinatura (*si*) de 1 a N para cada bloco básico(*vi)* em tempo de compilação;
3. Calcular em tempo de compilação a assinatura diferença (*di*) definida com base nas assinaturas do(s) predecessor(es) de *vi* (*pred(vi)*) e na assinatura de *vi*. Este cálculo é realizado através da fórmula: $di = ss \oplus sd$, onde *ss* representa a assinatura do nó fonte (predecessor) e *sd* representam a assinatura do nó destino (no caso o próprio nó *vi*). Assim, dado o bloco básico *vi* e seu conjunto de predecessores igual a *pred(vi)={vj},* a assinatura *di* é calculada a partir da fórmula: $di = sj \oplus si;$
4. Quando um determinado nó *vi* possui mais de um predecessor é necessário calcular em tempo de compilação a assinatura de ajuste (D) [73].

Além dos passos acima descritos, uma assinatura *G* deve ser calculada em tempo de execução e armazenada em uma variável dedicada denominada registrador de assinatura global. Assim, toda vez que o controle é transferido de um bloco básico para outro a assinatura *G* é atualizada através da função de assinatura *f* dada pela equação (5.5.1).

$$f = f(G, di) = G \oplus dd \text{ , onde } dd = ss \oplus sd \qquad (5.5.1)$$

A Ilustração 5.5.1 mostra um exemplo de um desvio legal que ocorre do nó *v1* para *v2*. Observe que antes do desvio, $G = G1 = s1$ e após o desvio *G* é atualizado a partir do seguinte cálculo: $f = f(G1, d2) = G1 \oplus d2 \text{ , onde } d2 = s1 \oplus s2$ e assim $G2 = s1 \oplus (s1 \oplus s2) = s2$, ou seja, para o nó v2, $G2 = s2$.

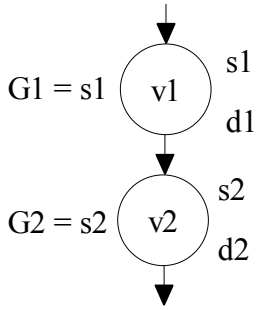


Ilustração 5.5.1: Exemplo de um desvio legal de *v1* para *v2* [73].

A Ilustração 5.5.2 mostra um exemplo de desvio ilegal de *v1* para *v4*. Observe que antes do desvio, $G = G1 = s1$ e após o desvio, *G* é atualizado a partir do seguinte cálculo: $f = f(G1, d4) = G1 \oplus d4$, *onde* $d4 = s3 \oplus s4$ e assim $G4 = s1 \oplus (s3 \oplus s4) \neq s2$, ou seja, o erro de fluxo de controle será detectado pois, *G4* é diferente de *s4*.

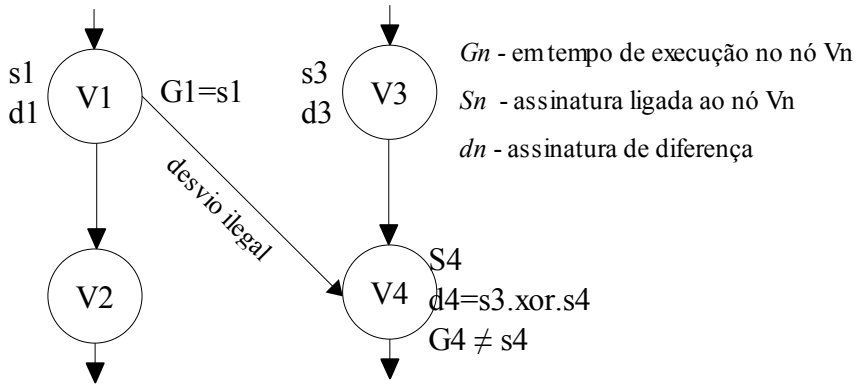


Ilustração 5.5.2: Exemplo de um desvio ilegal de *v1* para *v4* [73].

Assim, no topo de cada bloco básico devem ser inseridas as instruções de verificação abaixo definidas:

1. Função assinatura: G = (G $\oplus dk$);
2. Instrução de verificação de desvio: "Br (G $\oplus sk$) erro".

A Ilustração 5.5.3 mostra respectivamente a representação gráfica das instruções, do bloco básico acrescido das instruções de verificação e a representação utilizada para um nó em um grafo de fluxo de execução.

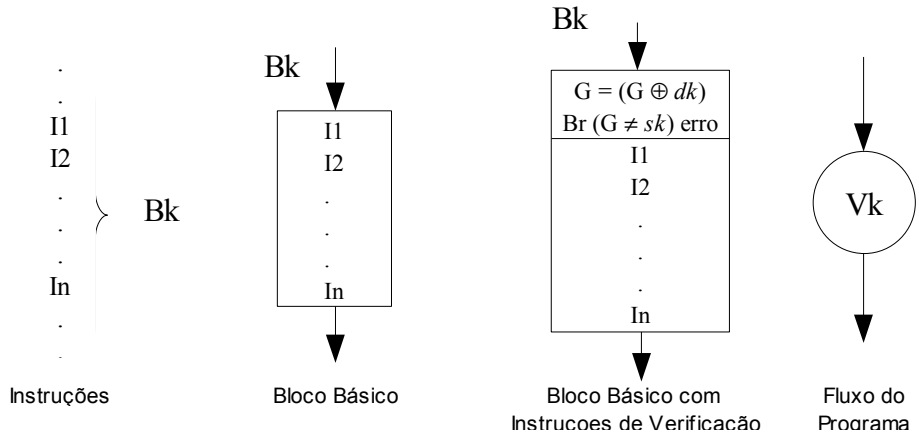


Ilustração 5.5.3: Representação gráfica [73].

Entretanto, quando um determinado bloco básico possui mais de um predecessor (grafos com nós convergentes) é necessário inserir no código uma assinatura extra denotada como D. A Ilustração 5.5.4 mostra claramente o problema que surge quando um determinado bloco básico possui mais de um predecessor. Observe que os nós *v1* e *v3* desviam para o nó *v5*. Assim *d5* seria um resultado de *s1 ⊕ s5*. Caso o desvio seja *br1,5* o G5 = G1 *⊕ d5 = s1 ⊕ s1 ⊕ s5 = s5* , ou seja, o G5 será igual a s5 e conseqüentemente nenhum erro será observado. Entretanto, ocorrerá um erro se o desvio for *br3,5,* pois o *G5 = G3 ⊕ d5 => s3 ⊕ s1 ⊕ s5 ⊕ s5* devido a *s3 ⊕ s1*.

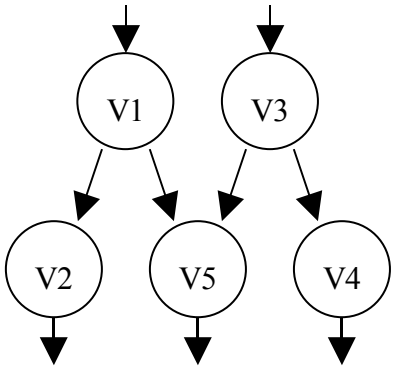


Ilustração 5.5.4: Exemplo de um bloco básico com mais de um predecessor [73].

Neste contexto, a fim de solucionar o problema acima ilustrado, surge a assinatura D, que deve ser inserida após a instrução que realiza o cálculo de atualização de G. A Ilustração 5.5.5 mostra um exemplo de utilização da assinatura D. Observe que no bloco básico B5 é adicionada a instrução *G = G ⊕ D* após a instrução que calcula a atualização da assinatura G,

dada por $G = G \oplus d5$. D é determinado a partir dos nós fontes *v1* e *v3* , ou seja, $D = s1 \oplus s3$. Inicialmente, $d5 = s1 \oplus s5$ e D no bloco B1 apresenta o valor 0000. Após o desvio *br1,5*, $G5 = G \oplus d5$ e $G5 = G \oplus D = s5 \oplus 0000 = s5$ e após o desvio br3,5, $G5 = G3 \oplus d5 = s3 \oplus (s1 \oplus s5)$ e $G = G5 \oplus D = s3 \oplus (s1 \oplus s5) \oplus (s1 \oplus s3) = s5$.

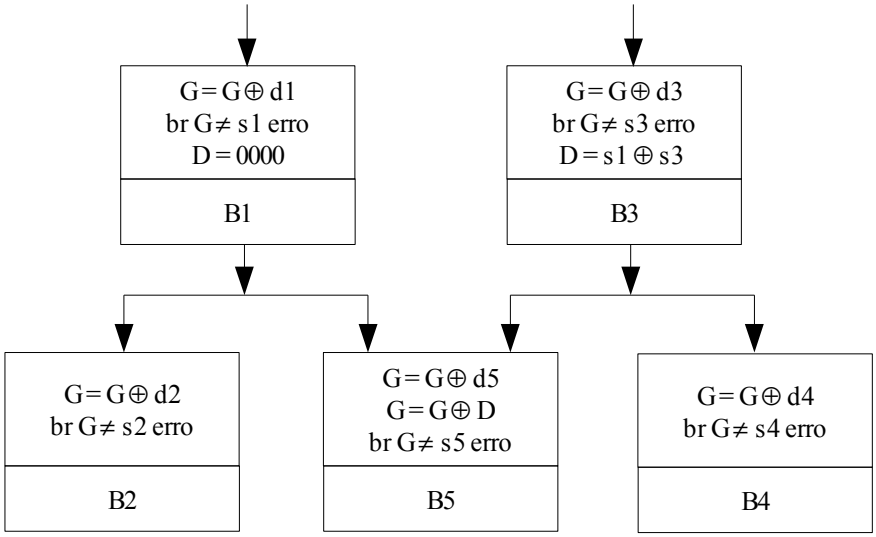


Ilustração 5.5.5: Exemplo de utilização da assinatura D utilizada para solucionar o problema de nós convergentes [73].

Neste contexto, surge o algoritmo a seguir, denominado algoritmo A, para nortear a implementação da técnica de CFCSS em um determinado programa.

Algoritmo A:

Passo 1: Identificar todos os blocos básicos, o grafo de fluxo do programa e o número de nós do grafo;

Passo 2: Atribuir um *si* para cada vi, em que $si \neq sj$ se $i \neq j$, i,*j*=1,2,...,N;

Passo 3: Para cada *vj*, *j* = 1,2,...,N;

3.1. Para *vj* cujo *pred(vj)* é somente um nó *vi*, então $dj = si \oplus sj$;

3.2. Para *vj* cujo *pred(vj)* é um conjunto de nós *vi*,1, *vi*,2,...,*vi,M-então*, vj é determinado por um dos nós como $dj = si,1 \oplus sj$. Para *vi,m*, *m*=1,2,...,M, inserir uma instrução Di, $m = si,1 \oplus si$,m em *vi,m;*

3.3. Inserir uma instrução $G = G \oplus dj$ no começo de *vj;*

3.4. Se *vj* é um nó com mais de um predecessor, então inserir a instrução G = G ⊕ D após *G = G ⊕ dj* no nó *vj;*

3.5. Inserir uma instrução "*br(G ⊕ sj) erro*" após as instruções dos dois últimos passos;

Passo 4: Fim Algoritmo.

Quanto à cobertura de falhas, a técnica de CFCSS é capaz de detectar:

1. Um desvio ilegal feito para a função de assinatura - primeira linha do nó;
2. Um desvio ilegal feito para a instrução "*br(G ⊕ s) erro*" - segunda linha do nó;
3. Um desvio ilegal para o corpo do nó;
4. Um desvio inserido dentro do próprio nó, isto é, dentro do próprio bloco básico;
5. A exclusão de uma instrução de desvio incondicional do nó.

Uma pequena modificação no algoritmo é sugerida [73], a fim de diminuir a degradação no desempenho em função da agregação desta técnica. Pois a inclusão desta técnica aumenta de 2 a 4 instruções em um bloco, sendo este bloco a exemplo de 8 instruções, causaria um aumento de memória utilizada de 25% a 43%. Ao invés de serem inseridas as instruções de verificação em todos os blocos básicos, pode-se eleger alguns blocos para receberem estas instruções a fim de minimizar os custos de memória utilizados. Porém, esta redução eleva a latência para detecção de erros de fluxo de controle e conseqüentemente pode aumentar o número de falhas não detectadas pela técnica. Em resumo, o sistema pode gerar saídas erradas até que uma instrução "*br(G ⊕ sj) erro*" seja executada pelo processador.

5.6 Técnica YACCA (*Yet Another Control-Flow Checking using Assertions*)

O princípio fundamental desta técnica é o mesmo das técnicas anteriores, ou seja, esta técnica tem como objetivo inserir um conjunto de assinaturas que são geradas durante a compilação do código e uma assinatura atualizada durante a execução para verificar a execução correta do fluxo de programa. O conjunto de assinaturas geradas durante a compilação são os identificadores dos blocos básicos e a assinatura gerada durante a execução do programa é armazenada em uma variável dedicada que está associada ao valor do bloco básico atual [2,3].

Deve-se notar que esta técnica, após diversos testes [2], apresenta as seguintes vantagens:

1. A assinatura é calculada sem depender de qualquer ajuste na assinatura como é o caso da técnica CFCSS;
2. Apresenta cobertura para erros de fluxo de controle se múltiplos nós compartilham múltiplos nós com seus nós de destino;
3. Apresenta menor custo de memória que as técnicas estudadas anteriormente (CFCSS e ECCA);
4. Apresenta maior cobertura, sendo uma alternativa mais robusta que as técnicas apresentadas anteriormente;

A aplicação da técnica baseia-se nos seguintes passos:

1. Definir o grafo de fluxo de execução para dividir o programa em blocos básicos;
2. Atribuir um valor único de assinatura (*si*) de 1 a N para cada bloco básico(*vi)* em tempo de compilação;
3. Inserir em cada bloco básico (*vi*) as seguintes instruções:
   3.1. Uma **instrução de teste** que controla a assinatura do bloco básico anterior (predecessor) e verifica se o desvio ocorrido é válido de acordo com o grafo do programa, ou seja, verifica se *vj* pertence ao grupo de *pred*(*vi*);
   3.2. Uma **instrução de atualização** que calcula a variável ***code*** com o novo valor referente ao bloco atual.

Assim, as duas instruções inseridas nos blocos básicos do programa: instrução de teste e instrução do cálculo da assinatura (***code***), são baseadas em regras e pertencem respectivamente ao conjunto de teste e ao conjunto das assinaturas.

- **Regras para definir o conjunto de teste:**

Durante a execução do programa, quando ocorre uma transição do bloco básico *vj* para *vi* é necessário verificar se esta transição é legal, ou seja, se *brj,i* pertence ao conjunto *Ei*.

Para que este controle seja realizado, a técnica sugere a criação de uma variável denominada $PREVIOUS_i$ que contém o produto de todas as assinaturas dos nós predecessores de *vi* (conforme a equação (5.6.1)) e a inserção da instrução de teste no início de cada bloco básico (conforme a equação (5.6.2)).

$$\boldsymbol{PREVIOUS_i} = \prod B_j, \forall v_j \; com \; desvio \; br_{j,i} \in E_i \qquad (5.6.1)$$

Onde: *Bj* corresponde à assinatura(s) do(s) nó(s) predecessor(es) de *vi*; *Ei* é o conjunto que contém os desvios legais para vi; *brj,i* representa o desvio de *vj* para *vi*.

$$if \; (\; \boldsymbol{PREVIOUS_i} \; \% \; CODE \;) \; \boldsymbol{erro}() \qquad (5.6.2)$$

Devido à complexidade da instrução acima, equação (5.6.2), seu processamento torna-se bastante crítico, por isto, são sugeridas duas soluções alternativas representadas pelas equações (5.6.3) e (5.6.4).

$$if \; ((\boldsymbol{CODE} \; != \; Ba) \wedge (\boldsymbol{CODE} \; != \; Bb) \wedge (\cdots) \wedge (\boldsymbol{CODE} \; != \; Bn)) \; \boldsymbol{error}() \qquad (5.6.3)$$

Onde: Ei = {$br_{a,i}$; $br_{b,i}$; ...; $br_{n,i}$}

$$\boldsymbol{ERR_{CODE}} \models ((\boldsymbol{CODE} \; != \; Ba) \wedge (\boldsymbol{CODE} \; != \; Bb) \wedge (\cdots) \wedge (\boldsymbol{CODE} \; != \; Bn)) \qquad (5.6.4)$$

Onde: Ei = {$br_{a,i}$; $br_{b,i}$; ...; $br_{n,i}$}

- **Regras para definir o conjunto de atualização das assinaturas:**

Dado um determinado bloco básico *vi* a variável ***code*** será igual a *Bi*, onde *Bi* corresponde à assinatura de *vi*.

A fórmula genérica para o cálculo de ***code*** é dada pela equação (5.6.5).

$$\boldsymbol{code} = (\boldsymbol{code} \wedge M1) \oplus M2 \qquad (5.6.5)$$

Onde, *M1* representa uma constante calculada a partir das assinaturas dos nós que formam o conjunto dos *pred(vi)*. Já *M2* representa uma constante gerada a partir da assinatura do nó atual e dos nós que formam o conjunto dos *pred(vi)*.

Os exemplos abaixo demonstram claramente o cálculo da variável ***code***.

Exemplo 1: Dado *vi*, seu conjunto de predecessores é:

$pred(vi) = \{vj\}$

$M1 = -1$ e $M2 = Bj \oplus Bi$

Assim $\boldsymbol{code} = \boldsymbol{code} \oplus (Bj \oplus Bi)$

Exemplo 2: Dado *vi*, seu conjunto de predecessores é:

$pred(vi) = \{vj, vk\}$

$M1 = (Bj \overline{\oplus} Bk)$ e $M2 = (Bj \,\&\, (Bj \overline{\oplus} Bk) \oplus Bi)$

E assim,

$$\boldsymbol{code} = ((\boldsymbol{code} \,\&\, (Bj \overline{\oplus} Bk)) \oplus (Bj \,\&\, (Bj \overline{\oplus} Bk) \oplus Bi))$$

A Ilustração 5.6.1 mostra uma aplicação modificada a partir das regras de transformação de código definidas pela técnica YACCA.

```
ERR_CODE |= (code != 0);
code = code ^ 1; /* code = 1; a expressão "^" é um "xor" */
x0 = 1;
y = 5;
i = 0;
while( i < 5 ) {
        ERR_CODE |= (code != 1) && (code != 3);
        code = (code & 5) ^ 3; /* code = 2 ; onde "&" é um "and" */
        z = x+i*y;
        i = i+1;
        ERR_CODE |= (code != 2);
        code = code ^ 1; /* code = 3 */
}
ERR_CODE |= (code != 1) && (code != 3);
i = 2*z;
```

Ilustração 5.6.1: Código modificado a partir da técnica YACCA.

Quanto à cobertura de falhas, esta técnica é capaz de detectar os seguintes tipos de erros conforme a Ilustração 5.6.2:

a) Um desvio de vi para o bloco básico vj que, por sua vez, não pertence ao conjunto de suc(vi);
b) Um desvio de vi para o início do bloco básico vj que, por sua vez, pertence ao conjunto de suc(vi);
c) Um desvio de vi para algum lugar do bloco (vj) que pertence ao conjunto de suc(vi).

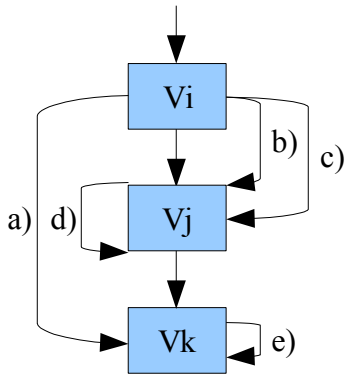


Ilustração 5.6.2: Cobertura de falhas da técnica YACCA.

Note que os desvios “d)” e “e)” na Ilustração 5.6.2 não são detectados por esta técnica porque não alteram a assinatura do bloco básico.

5.7 Resumo das Técnicas de Detecção de Erro via *Software*

Todas as técnicas apresentadas nos tópicos anteriores foram resumidas e apresentadas nos campos da Tabela 5.7.1:

Técnica	Método	Cobertura	Comentários
Block Signature Self Checking (BSSC) [69]	O programa é dividido em blocos básicos, e a cada bloco é atribuída uma assinatura (o endereço da primeira instrução no bloco básico) em uma variável. Uma instrução de chamada no fim do bloco busca a assinatura da variável e compara com a assinatura de referência. Caso haja discordância um sinal de erro é gerado.	Manipulação de lista encadeada, Classificação rápida (*quicksort*) , manipulação de matriz. Cobertura: 15-23% (BSSC), 32-35% (ECI) e 13-27%(WDT) [69].	A técnica baseia-se em endereços absolutos para o começo do bloco básico. Isto impede a utilização em códigos realocáveis.
Control Flow Checking by Assertions (CCA) [70]	Consiste na divisão do código do programa em intervalos livres de desvio (BFIs), na inserção de dois identificadores (identificador do intervalo livre de desvio - BID e identificador de fluxo de controle CFID) para cada um dos BFIs e na verificação dos identificadores durante o período de execução do código.	Cobertura: 37.2% [70]	Custo alto, são adicionados múltiplas linhas de código para empilhar, desempilhar para examinar uma fila, configurar uma variável, e testar a variável, tornando ineficiente.
Enhanced Control-flow Checking with Assertions (ECCA) [71]	São inseridas instruções em alto nível de programação nos pontos de entrada (instrução de teste) e saída (instruções de atualização) dos intervalos livres de desvio (BFI). No caso de erro no fluxo de controle, o calculo do identificador do intervalo livre de desvio BID causará um erro “dividir-por-zero”.	Cobertura: Teste 1: 22.6-72.9%, Expresso: 27.8-70% 0.22li (*benchmark*): 18.4-55.6% [71]	Diminuem o *overhead* relacionado ao custo e ao desempenho da técnica CCA. Não detecta falhas que não cruzam pelo limite do bloco, isto é, falhas que causam um desvio no mesmo bloco.

Técnica	Método	Cobertura	Comentários
Control Flow Checking by Software Signatures (CFCSS) [73]	Cada blocos básico (vi) possui um valor único de assinatura (si). Em tempo de compilação é gerada uma assinatura diferença (di) entre os predecessores de (vi) e a assinatura de (vi). Uma assinatura G deve ser calculada em tempo de execução em função de (di). A assinatura G é diferente da assinatura do bloco básico (vj) no caso de erro no fluxo de controle.	LZW: 30.8% FFT: 34.4% Mult. Matriz: 41.0% Class. Rápida: 28.8% Insert sort: 37.2% Hannoi: 34.6% Shuffle: 40.0% [73]	Uma limitação do CFCSS é que não cobre erros de fluxo de controle se múltiplos nós compartilham múltiplos nós com seus nós de destino.
Yet Another Control-Flown Checking using Assertions (YACCA) [2,3]	Cada bloco básico (vi) contém uma instrução de teste (controla a assinatura do precessor) e uma instrução de atualização (calcula a variável *code*). No caso de erro no fluxo de controle, a instrução de teste irá sinalizar o erro.	Mult. Matriz: 56.0% Ellipt. Filter: 54.5% Kalman Filter: 22.2% LZW: 21.1% [2,3]	A técnica de YACCA demonstrou melhor cobertura de falhas comparado com as técnicas de detecção de erro via *software* estudadas neste trabalho.

Tabela 5.7.1: Resumo das técnicas de detecção de erro por *Software*.

6 TÉCNICAS DE DETECÇÃO DE ERROS VIA *HARDWARE*

6.1 Introdução

A utilização do WDP para testes concorrentes (*on-line*) pode ser comparado com a utilização para testes funcionais (*off-line*) de microprocessadores. Em ambos os casos, a verificação é feita em nível mais alto (funcional) que o nível de circuito (transistores, portas lógicas ou registradores). Além disso, o WDP pode ser adicionado a qualquer sistema, sem grandes alterações no mesmo [74].

Os métodos existentes para monitoração de assinatura via *hardware* podem ser classificados em três categorias básicas, segundo a forma com que as assinaturas são armazenadas [75,74]:

- Na primeira categoria uma lista de assinaturas é elaborada durante a fase de compilação e armazenada separadamente do programa. Esta abordagem tem duas desvantagens principais:
  1. Custo (*overhead*) de memória para armazenar um grande volume de assinaturas;
  2. Execução mais lenta devido à busca das assinaturas.
- Na segunda categoria, assinaturas são embarcadas no fluxo das instruções. Esta abordagem exige mecanismos para distinguir as assinaturas de instruções regulares. A velocidade de execução do programa diminui devido à espera da CPU durante o processamento das assinaturas.
- Na terceira categoria, as assinaturas são embarcadas no fluxo do programa em linguagem de alto nível e são enviadas explicitamente para o WDP na forma de mensagens de dados. Como desvantagem tem-se a latência de comunicação no processo de escrita no *buffer* compartilhado. A organização do *hardware* (baseado na passagem de mensagem) é mostrado na Ilustração 6.1.1. O processador principal escreve no *buffer* compartilhado para o WDP acessar e ler (o *buffer* pode ser, por exemplo, um registrador interno do WDP). Ambos os processadores têm também suas memórias locais além do *buffer* compartilhado.

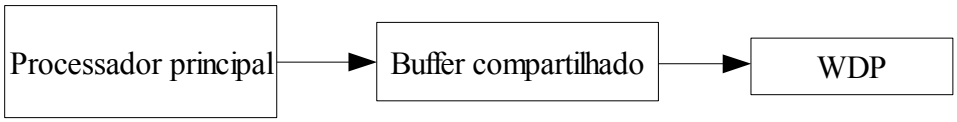


Ilustração 6.1.1: Organização do hardware para verificação [74].

O desempenho das técnicas de monitoração da assinatura podem ser caracterizada por cinco propriedades: cobertura de detecção de erro, memória suplementar, perda do desempenho do processador, latência de detecção de erro e complexidade do monitor [76,77,78]. Destacam-se como as mais importantes:

1. Latência de detecção de erro ou simplesmente latência: é o tempo médio levado para descobrir um erro depois de ter ocorrido, geralmente é medido em termos de número de instruções.
2. Cobertura de erro: é a relação do número de erros que se pode detectar pelo número total de erros possíveis.

Um projeto ideal deveria otimizar cada uma das cinco propriedades. Porém, como elas não são independentes uma das outras, otimizando uma propriedade pode ocasionar a degradação de outra. Por exemplo, as técnicas que reduzem custo (*overhead*) de memória também acrescentam latência, pois a distância entre as assinaturas de referência expandem-se. Então, é importante entender os requisitos do sistema a fim de avaliar a perda da qualidade ou aspecto de algum critério em troca do ganho de outra qualidade ou aspecto. Implicando em uma decisão feita com a compreensão completa dos aspectos positivos e negativos da escolha em questão.

A arquitetura do WDP deve ter também como alvo a portabilidade da implementação, permitindo uma adaptação fácil para diferentes microprocessadores existentes e diferentes arquiteturas de sistema. A característica escolhida deve satisfazer os seguintes requisitos:

1. O WDP não deve ser complexo;
2. Deve dispor de uma boa cobertura de falhas;
3. Não deve exigir grandes mudanças no projeto do processador supervisionado;
4. Não deve resultar em grandes custos (*overhead*) ao sistema monitorado.

O controle de fluxo do programa pode oferecer a melhor cobertura para detecção de erros e a análise de assinatura durante a execução do programa, é considerado como a técnica mais simples para a implementação de detecção de erros por controle de fluxo [7]. Segue a idéia básica destas técnicas:

1) O código do programa é logicamente dividido em blocos básicos no período de compilação;
2) Cada bloco básico tem uma seqüência válida e inalterável de instruções (por exemplo um bloco sem instruções de desvio);
3) As assinaturas são calculadas em função das instruções de cada bloco básico;
4) Durante a execução do programa, a assinatura de cada bloco básico é calculada por um WDP

em *hardwar*e. Assim, determinados erros podem ser detectados comparando a assinatura gerada durante a execução do programa com a assinatura gerada na fase de compilação.

Uma representação típica da técnica que utiliza um WDP é mostrado na Ilustração 6.1.2. A detecção de erros por meio de WDP é dividido em duas fases: na primeira fase (fase de configuração) o WDP é inicializado com algumas informações sobre o processador ou processo a ser verificado. Durante a segunda fase (verificação), o WDP monitora o processador e concomitantemente coleta informações relevantes [74,7]. A detecção de erro na execução do programa é baseada na comparação entre as informações coletadas concomitantemente durante a execução com as informações dispostas durante a fase de configuração.

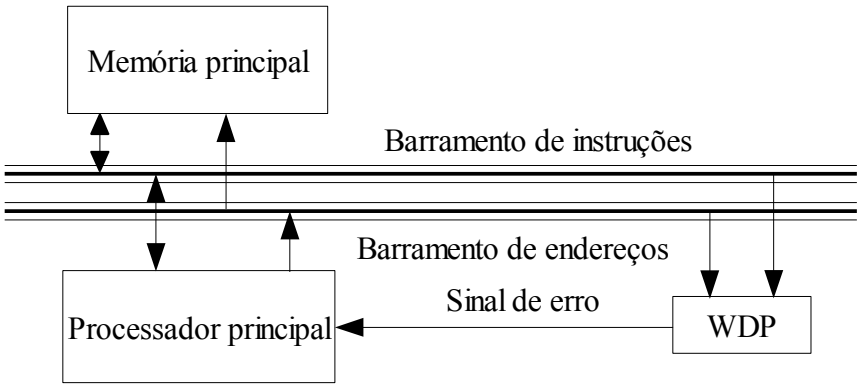


Ilustração 6.1.2: Detecção de erros utilizando WDP [74].

As técnicas de verificação por WDP em que as assinaturas são arbitrariamente associadas aos nós são denominadas de **assinatura atribuída** para verificação de controle de fluxo (*assigned-signature checking*) e as técnicas em que as assinaturas são derivadas dos nós são denominadas de **assinatura derivada** para verificação de controle de fluxo (*derived-signature checking*) [79]. Ambas são utilizadas nas técnicas de detecção de erros via *Hardware*, porém nas técnicas de detecção de erros via *Software,* as assinaturas são do tipo **assinatura atribuída**.

Na maioria dos métodos utilizados nos WDP como mostra a Ilustração 6.1.2, a seqüência de busca das instrução no barramento do sistema é supervisionado (método baseado em **assinatura derivada** para verificação de controle de fluxo) utilizando algum tipo de compactação de informações [10]. Porém, na arquitetura de computadores moderna, a observabilidade do barramento do sistema é criticamente reduzido pela utilização de memórias cache e fila de busca de pré-instrução.

As denominadas **assinaturas atribuídas** são calculadas e inseridas no programa fonte para a verificação do controle de fluxo do programa em tempo de compilação, porém para minimizar a perda de desempenho na utilização de inúmeras assinaturas deste tipo, desenvolveu-se duas maneiras de reduzir sua utilização no código [8]:

- **Redução estática**: baseia-se na diminuição de utilização de assinaturas através da unificação de assinaturas pela análise dos vértices no grafo de fluxo de controle (*Control Flow Graph* - CFG). Deste modo, as instruções que fazem a transferência do valor da assinatura no código do programa para o WDP unem múltiplas assinaturas em um único respectivo vértice com uma única assinatura, desde que este vértice tenha apenas um ponto de entrada e um ponto de saída sem instruções de desvios (similar a um bloco básico discutido em seções anteriores). O fator de **redução estática** é controlada e definida pelo usuário através da quantidade de assinaturas unidas. Altos fatores de redução resultam em menos verificações, acrescentando latência de detecção de erro e reduzindo a probabilidade de detecção, por outro lado resulta em um menor tempo de execução do programa e o custo de código extra em área de memória armazenada. A **redução estática** pode ser utilizada visando remover a verificação de assinaturas em pequenos blocos básicos no CFG.
- **Redução dinâmica**: a remoção da verificação de assinatura em blocos cíclicos (laços no código do programa do tipo *for*, *while*, *do*, etc..) no CFG não são possíveis, pois o programa pode ser executado em laço durante grandes períodos de tempo sem qualquer verificação. Conseqüentemente, cada laço pode conter pelo menos uma assinatura para verificação. Porém, pode-se ter uma grande perda do desempenho no tempo de execução do código, devido ao alto tráfego de dados no barramento do processador, devido a transferência de muitas assinaturas em pequenos laços. Neste caso, a **redução dinâmica** pode proporcionar eficiência evitando este tipo de efeito. Em vez de transferir a assinatura, apenas uma variável é incrementada. Se a contagem exceder um valor definido pelo usuário (fator de redução dinâmico), o contador é restaurado e a assinatura é transferida para o WDP. Para utilizar a **redução dinâmica**, pode-se fazer a análise do fluxo do programa através da utilização das ferramentas do Linux como o *gcc* e *gcov.* Assim, pode-se observar os pontos críticos do código, onde o programa executa em laços diversas vezes. Como pode-se observar na Ilustração 6.1.3, à esquerda encontra-se o programa original e a direita a análise da quantidade de execução de cada linha do código após aplicar-se os comandos do Sistema Operacional Linux (no centro na figura): *gcc* e *gcov*.

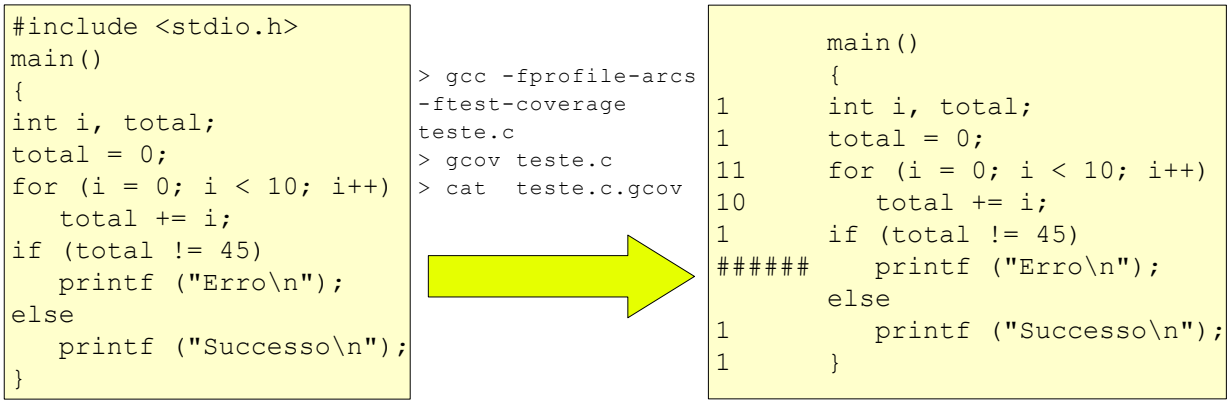


Ilustração 6.1.3: Análise do fluxo do programa para redução dinâmica.

Hoje em dia as denominadas **assinaturas atribuídas** para verificação de controle de fluxo, são quase que utilizadas exclusivamente em computadores *Off-The-Shelf* (COTS). Neste método um pré-processador extrai o grafo de fluxo de controle do programa (CFG) do código fonte em linguagem de programação de alto nível.

6.2 Técnica WDT (*Watchdog Timer*)

Enquanto o microprocessador executa instruções de memória, se uma descarga elétrica acontece, causando falhas nos dados trafegados no barramento, o processador pode executar um *byte* alterado. Uma falha no programa também pode resultar em um estouro de pilha. Em outro caso, em presença de falhas, o processador pode executar um código em algum ponto impossível de predizer, normalmente resultando em parada do sistema [4].

Se isto acontecer em computadores pessoais, pode-se utilizar o teclado do PC ou desligar a energia do computador para reiniciar o sistema. Particularmente em *software* de sistema embarcado, o WDT é literalmente um "cão de guarda", utilizado para fazer a mesma tarefa que o usuário realiza em máquinas pessoais. Se uma falha acontece no sistema, é uma tarefa do WDT devolver o sistema novamente em operação, devido a isso o WDT é obrigatório em qualquer aplicativo de alta confiabilidade [5].

Como exemplo, o laço infinito principal de um programa leva, em média, 25 milisegundos para executar; e o pior caso de tempo de execução de 35 milisegundos. Um dispositivo WDT está conectado com alta-prioridade de interrupção do sistema ou uma interrupção não mascarável para reinciar o mesmo. Depois do WDT ser ativado, o dispositivo espera 50 milisegundos (uma pequena margem além

do pior caso de tempo de execução) e então ativa o sinal de restauração (*reset*) do processador, causando a reinicialização do contador de programa (PC). O único meio de prevenir este sinal enviado pelo WDT ao processador, é através de um pulso no WDT a fim de reinicializar o seu contador de tempo interno, causando novamente a contagem de 50 milisegundos.

O exemplo da Ilustração 6.2.1 serve como sugestão para supervisionar a execução de um programa, antes de sinalizar ao WDT. Um sinalizador (*flag*) deve ser configurado em vários pontos no código, indicando a conclusão bem sucedida deste bloco de código. Logo, antes de sinalizar o WDT para reinicializar o seu *timer*, todas os *flags* são checados. Se todos os *flags* estiverem configurados, o temporizador (*timer*) pode ser reinicializado, caso contrário o modo de falha é registrado e algum tratamento de correção deve ser executado [5].

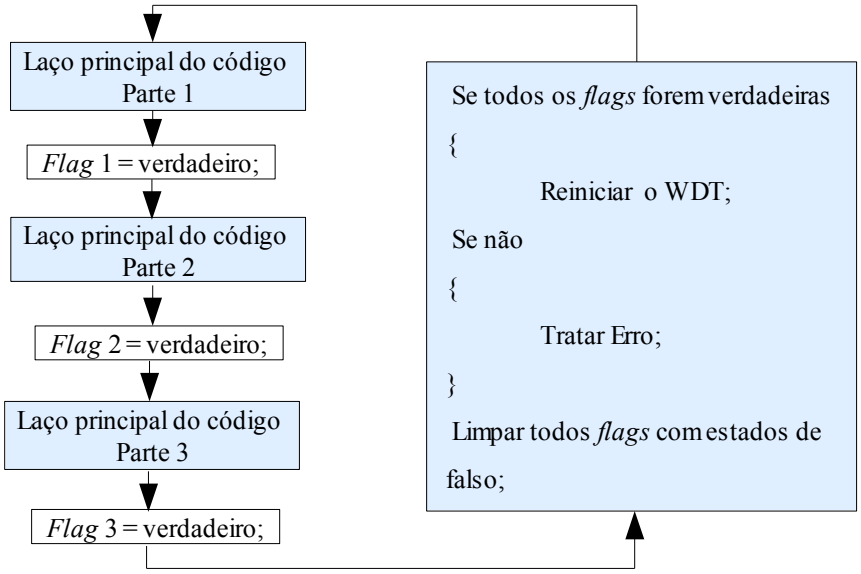


Ilustração 6.2.1: Esquema de funcionamento de um WDT [5].

6.3 Técnica *Concurrent Process Monitoring with No Reference Signatures*

Nesta técnica, uma função de assinatura conhecida, é aplicada ao fluxo de instrução em fase de compilação e quando o acúmulo das assinaturas formam um código *m-out-of-n*, as instruções correspondentes são sinalizadas. A verificação de erro é feita em tempo de execução, monitorando as assinaturas acumuladas nas posições sinalizadas para determinar se elas formam códigos *m-out-of-n* [75].

O código *m-out-of-n* consiste numa codificação em que uma palavra de “n” bits deve sempre existir ao final de palavras de “m” bits iguais a "1". Todas as palavras geradas que não possuem o número correto de "1", são detectadas pelo verificador e é sinalizado um erro.

A assinatura de um código seqüencial (trecho de código sem desvio) é derivada de uma função geradora de assinatura aplicada sucessivamente sobre o *opcode* até formar um código *m-out-of-n* para um especifico “*m*” e “*n*”. A posição na memória que corresponde ao código *m-out-of-n* é sinalizada como ponto de verificação para uma comparação posterior. Durante a execução, a assinatura gerada na posição marcada é checada para determinar se forma um código *m-out-of-n*, caso negativo, um sinal de erro é indicado.

Esta abordagem de verificação de assinatura não exige embarcar as assinaturas de referência em tempo de compilação, resultando assim em uma economia de memória e também do tempo de execução. A abordagem do código *m-out-of-n* oferece alta cobertura de erro e latência controlável [75].

6.4 Técnica CSM (*Continuous Signature Monitoring*)

O código do programa é dividido em blocos básicos durante a fase de compilação e é calculado uma assinatura de referência e inserida uma instrução de assinatura no final para cada bloco básico. A instrução de assinatura tem um campo que contém um identificador *opcode*, e um campo que contém a assinatura. O *opcode* pode ser um identificador do co-processador já incluso no conjunto de instruções do processador, ou pode ser adicionado especificamente para o conjunto de instruções. Em cada instrução de assinatura, o processador executa instruções de *NOP (No OPeration)* enquanto o Monitor compara em tempo de execução as assinaturas de referência, acusando um erro se elas não forem iguais [76].

O custo de memória e desempenho na utilização da assinatura de CSM é reduzido segundo alguns critérios, a exemplo, a redução no número de assinaturas embarcadas no programa seguindo os seguintes conceitos:

1) Desvios são de uma via (desvio incondicional) e de duas vias (desvio condicional) (sendo que múltiplos desvios podem ser decompostos em desvios condicionais);
2) O grafo de fluxo de programa (CFG) pode ser determinado durante a compilação do código e não é alterado durante a execução;
3) Um bloco tem somente um nó de entrada;
4) A memória do Monitor armazena somente o registro da assinatura.

Teorema: Se um programa tem "n" desvios condicionais e suas assinaturas intermediárias "v-bit" estão aleatoriamente distribuídas, então "(n + 1)v-bits" devem ser embarcados no programa para cobertura de erro de fluxo de controle de $1-2^{-v}$.

6.5 Técnica WDDP (*Watchdog Direct Processing*)

A técnica do WDDP consiste no monitoramento direto dos endereços acessados pelo processador principal. O programa é uma aplicação transparente ao WDDP que contém as informações necessárias para detectar os nós de referência do grafo de fluxo de controle (CFG) [78]. O WDDP tem dois objetivos principais:

- O primeiro consiste em calcular uma assinatura na seqüência de instruções executada;
- O segundo é a detecção dos nós executados pelo processador principal e os endereços de paradas (*breaks*) na seqüência de execução do programa. Em uma sequência legal de execução do programa, uma parada (*break*) deve resultar em um endereço de nó. Em caso de desvio, conhecendo o endereço do próximo nó (a ser executado) associado ao nó verificado, a assinatura é atualizada. Fundamentalmente a monitoração do WDDP pode ser resumida como segue:
  1. Se uma parada na execução do programa em um endereço de memória ocorreu antes do endereço do nó a ser executado, gera um sinal de erro;
  2. Quando o endereço de memória de um nó é executado, o WDDP verifica a assinatura corrente e atualiza o endereço do próximo nó. O endereço de parada é determinado pelo processador principal e em caso de desvio, é verificado o endereço de destino. Em caso de erro, gera uma sinal.

O WDDP contém uma instrução para cada nó na aplicação do programa. Cada instrução inclui três campos: o *opcode* (tipo de nó), o endereço associado a instrução no código do programa e uma informação de referência. Sete tipos de nós diferentes (*opcodes*) são definidos para esta técnica:

- I0: nó de inicialização;
- II: nó de destino;
- I2: nó seqüencial, desvio incondicional;
- I3: nó seqüencial, desvio condicional;

- I4: nó seqüencial, desvio incondicional para sub-rotina;
- I5: nó seqüencial, desvio condicional para sub-rotina;
- I6: nó seqüencial, retorno da sub-rotina.

Depois de carregar o contador de programa (PC) do WDDP, o WDDP espera por uma parada na seqüência de endereço do processador principal. No caso de um desvio incondicional, um erro é sinalizado caso não exista nenhuma parada depois de um certo número de ciclos de *clock*. Depois deste desvio, o endereço executado pelo processador principal é comparado com o valor do endereço de destino adicionado a um valor de deslocamento (dependendo da característica do processador principal). Finalmente o contador de programa (PC) do WDDP é incrementado por "1". Em cada nó, três ou quatro informações são necessárias:

1. Tipo do nó;
2. O endereço da instrução associada ao nó para verificação da execução do programa;
3. Valor de referência da assinatura do nó;
4. Endereço do nó de destino armazenado no WDDP (somente no caso de nó seqüencial).

A arquitetura do WDDP foi definida, com ênfase especial na sua modularidade:

- Contém um **módulo de interface do microprocessador** que é responsável pela detecção das paradas na seqüência de endereço;
- Contém um **módulo de interface de memória** que é responsável pelo acesso à memória pelo WDDP, onde estão as informações necessárias para detectar os nós de referência do grafo de fluxo de controle (CFG);
- Contém um **módulo de compactação** que é o gerador de assinatura. Quando o processador principal executa uma instrução de leitura, este módulo calcula e compacta o valor do barramento de instruções com a assinatura corrente e armazena o resultado em um registrador;
- Contém um **módulo de execução** que executa o conjunto de instruções do WDDP. A funcionalidade deste módulo é independente do microprocessador verificado. Quando **módulo de interface do microprocessador** indicar que um nó é executado, o **módulo de execução** ativa certas ações apropriadas (de acordo com o tipo de nó) e busca sua próxima instrução. Este módulo é composto por uma estrutura simples de microprocessador, pois contém um contador de

programa (PC), um registrador de instruções, um ponteiro da pilha, um comparador, um incrementador/decrementador e alguma lógica de controle.

6.6 Técnica ASIS (*Asynchronous Signatured Instruction Streams*)

A técnica *Asynchronous Signatured Instruction Streams* (ASIS) é capaz de monitorar continuamente o controle de fluxo de instruções e concomitantemente múltiplos processadores em um sistema [67].

Durante a compilação do código fonte, as informações redundantes são geradas caracterizando o controle de fluxo de programa da aplicação. Ao passo que, durante a execução do programa, as informações redundantes são utilizadas pelo Monitor (Verificador) para verificar a execução correta do fluxo de controle da aplicação. Em vez de embarcar as assinaturas na sequência da instrução, as assinaturas são organizadas em um grafo que caracteriza o fluxo de controle do programa da aplicação, e enviadas por uma função embarcada no código.

Uma assinatura é gerada para cada bloco-D (bloco dinâmico) no código da aplicação. As assinaturas para todos os blocos-D são organizadas em um grafo caracterizando o comportamento de desvio dinâmico do programa aplicativo. Um bloco-D consiste em um conjunto de blocos básicos com o primeiro bloco sendo o bloco de entrada e o último o bloco de saída (único bloco cujo ponto de saída tem mais de um possível destino). Existe exatamente um caminho de programa desde o início ao fim de um bloco-D em que o fluxo de controle pode ser mudado mas não desviado.

A Ilustração 6.6.1 mostra a implementação do esquemático em *hardware* do ASIS. Deve ser adicionado um módulo denominado de gerador de assinatura de *hardware* (*Hardware Signature Generator* - HSG) para cada processador a ser monitorado. O HSG monitora continuamente o barramento de instruções do processador, codifica as instruções buscadas da memória do programa, podendo detectar o início e o final de cada bloco-D. No fim de cada bloco-D a assinatura calculada pelo HSG é enviada para uma fila de assinatura e o registrador de assinatura (utilizado pelo HSG no cálculo da assinatura) é restaurado. Assim o HSG reinicia novamente o processo de gerar assinatura para o próximo bloco-D.

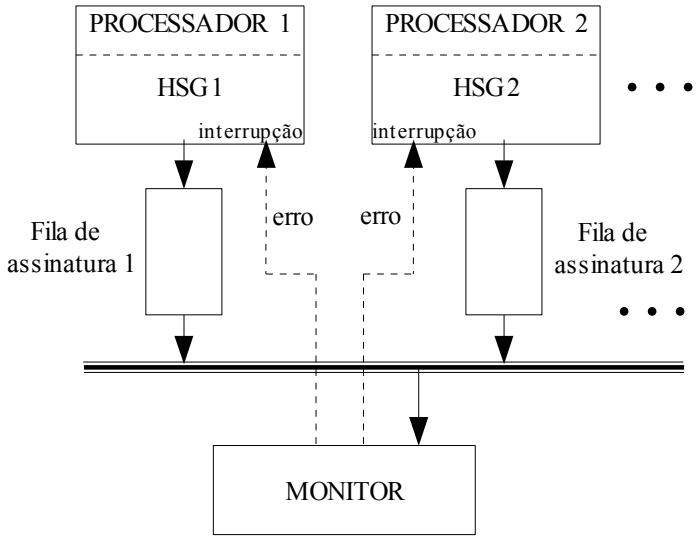


Ilustração 6.6.1: Implementação do esquemático em *hardware* do ASIS [67].

A técnica ASIS contém um processador para verificação (denominado Monitor), e um ou mais processadores monitorados. Durante a compilação do código, em linguagem de máquina, as informações redundantes são geradas para caracterizar o controle de fluxo do programa aplicativo. Durante a execução do programa, as informações redundantes são utilizadas pelo Monitor para checar a execução correta do controle de fluxo do programa. Em vez de embarcar as assinaturas no fluxo de instruções, as assinaturas são organizadas em um grafo que caracteriza o controle do fluxo do programa aplicativo. Este grafo de assinatura é armazenado na memória do Monitor e utilizado como referência para verificar a seqüência correta durante a execução das assinaturas geradas. Note que este é um dos maiores problemas desta técnica: o grafo que representa o fluxo de controle da aplicação pode ocupar um grande espaço de memória, dependendo da complexidade da aplicação monitorada.

O Monitor conhece a posição correta do nó corrente, pois possui armazenado em sua memória o grafo de assinatura. A próxima assinatura é buscada da fila de assinatura e comparada pelo Monitor com as assinaturas de todos os nós sucessores do nó corrente. Se a assinatura buscada da fila se equivale a uma assinatura de um nó sucessor então aquele nó se torna o nó corrente. Caso contrário, se uma assinatura inválida foi gerada durante a execução do código, ocorre uma indicação de erro enviando um sinal de interrupção para o processador correspondente.

6.7 Técnica OSLC (*On-line Signature Learning and Checking*)

Na técnica *On-line Signature Learning and Checking* (OSLC) a identificação de cada bloco básico e a geração da assinatura de referência são realizadas durante a execução normal da aplicação do programa em uma fase denominada fase de aprendizagem de assinatura (*signature learning phase*) [6,7].

A fase de aprendizagem de assinatura é realizada somente durante a execução do programa, ou seja, na fase de teste final do *software*. Para isso é necessário que cada bloco básico do programa seja executado pelo menos uma vez durante o teste final. O sistema detecta automaticamente a primeira execução de cada bloco e gera as assinaturas de referência. Cada assinatura é enviada para um computador externo (ou processador) junto com o endereço da última instrução do bloco. Uma vez que o teste do programa é finalizado, as assinaturas coletadas pelo computador externo representam o conjunto de assinaturas válidas para o programa em teste.

A configuração típica de *hardware* do sistema OSLC é apresentada na Ilustração 6.7.1. Um circuito pequeno denominado Gerador de Assinatura (*Signature Generator* - SG) é adicionado a cada processador específico (*Application Processor* – AP). Este circuito detecta o início e o fim de cada bloco básico do programa executado pelo processador específico e envia a assinatura calculada para um WDP Monitor (Verificador - *Checker*) onde a execução correta das assinaturas são verificadas. O Monitor tem acesso assíncrono em relação aos APs, conseqüentemente pode ser empregado para um sistema contendo vários processadores específicos (AP 1 até AP N) [6,7,80].

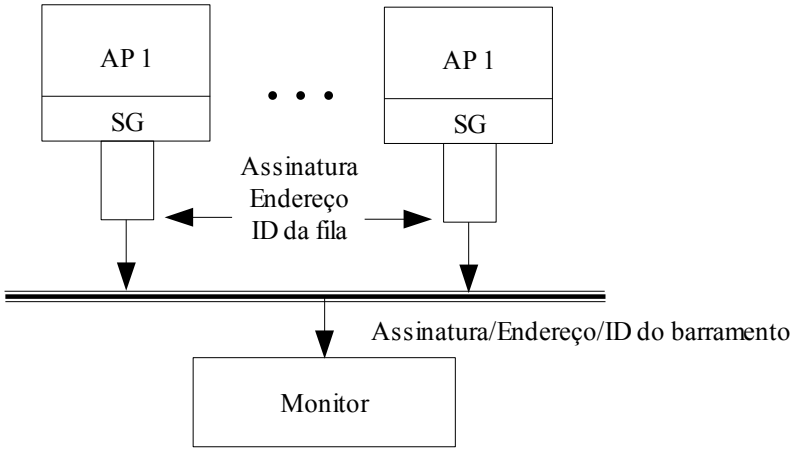


Ilustração 6.7.1: Configuração geral do OSLC [6,7,80].

Toda vez que o Monitor recebe uma instrução contendo um conjunto de campos PI (Identificador do processador)+Endereço+Assinatura (veja Ilustração 6.7.2), primeiramente utiliza as informações nos campos PI+Endereço para encontrar o segmento identificável (*Segment Identifier* - SI) da assinatura. Posteriormente o SI informa ao Monitor que o segmento de memória em que a assinatura deve ser verificada. Se a assinatura está entre as assinaturas armazenadas neste segmento, a assinatura é considerada correta, caso contrário, significa que um erro ocorreu no AP correspondente.

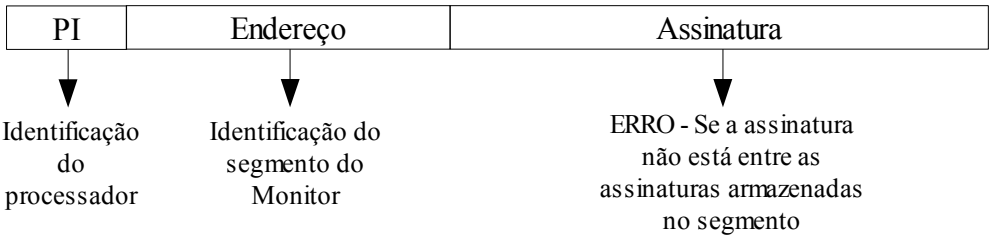


Ilustração 6.7.2: Formato das informações recebidas pelo Monitor [7].

Na técnica OSLC as assinaturas são armazenadas na memória local do Monitor, sendo este baseado nos seguintes princípios (Ilustração 6.7.3):

- O programa monitorado está logicamente dividido em pequenas seções de tal forma que existe somente um número pequeno de N assinaturas para cada seção de programa (isto é, para cada bloco básico). O número de N assinaturas por seção deve ser pequeno comparado ao número total de assinaturas possíveis para um dado tamanho de assinaturas a serem armazenados;

  Cada seção de programa abrange tantas instruções de programa necessárias para se ter N assinaturas;
- Para cada seção de programa monitorada existe um segmento correspondente na memória do Monitor onde todas as assinaturas de programa são armazenadas;
- O WDP (Monitor) pode identificar o segmento onde uma assinatura pertence, toda vez que o Gerador de Assinatura envia uma assinatura contendo o endereço da última instrução do bloco correspondente;
- O Gerador de Assinatura envia continuamente uma assinatura para o Monitor contendo o endereço da última instrução do bloco correspondente, então o WDP pode identificar o segmento onde aquela assinatura pertence;

Uma assinatura é considerada correta pelo Monitor caso esteja presente entre as assinaturas armazenadas em seu segmento. Como o número de N assinaturas por seção é pequeno, esta verificação pode ser feita rapidamente utilizando uma simples pesquisa binária.

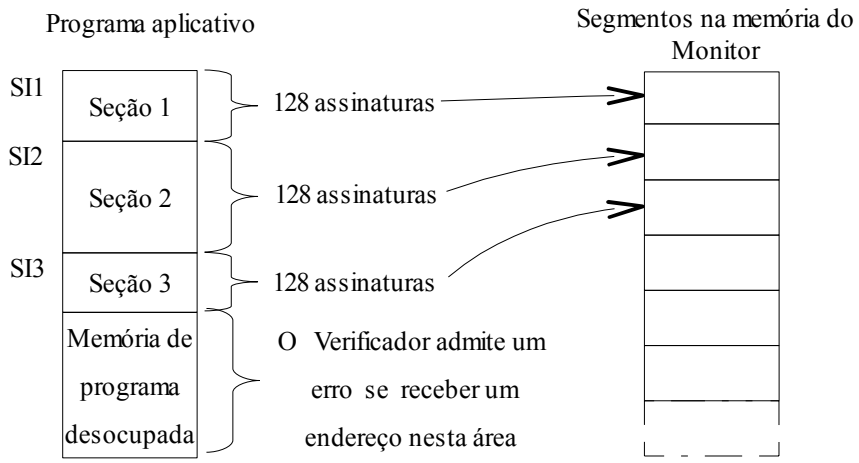


Ilustração 6.7.3: O Princípio do Monitor [6,7].

A idéia básica da verificação da assinatura está fundamentada no fato de que a probabilidade de uma assinatura errada ser igual para quaisquer das outras assinaturas na mesma seção, é muito baixa.

6.8 Técnica SEIS (*Signature Encoded Instruction Stream*)

Esta técnica é conveniente em ambiente de multiprocessamento e multitarefa, pois um único WDP monitora múltiplos processadores através assinaturas enviadas ao barramento pela aplicação. Essas assinaturas são embarcadas no programa em linguagem de alto nível, contendo um campo de identificação do processador em questão [8,9,10].

Durante a execução do programa as assinaturas são enviadas explicitamente ao WDP identificando a posição do programa. O WDP certifica a execução correta do programa através da verificação da seqüência da assinatura recebida. Esta seqüência será aceita como correta se corresponder a um fluxo de programa existente. No caso de uma instrução de desvio condicional, é verificado se a instrução de destino pertence a um conjunto de sucessores.

As instruções, inseridas em linhas do código em alto nível de programação na fase de pré-processamento, são enviadas explicitamente ao WDP contendo campos com informações relevantes. A declaração dos campos não identificam somente os vértices do controle de fluxo

de programa, mas também os vértices dos sucessores válidos. Desta maneira, não há necessidade de um banco de dados de assinaturas de referência, pois a seqüência correta de execução é verificada nos campos que indicam o vértice atual e o seu predecessor.

A estrutura da assinatura enviada ao WDP é mostrada na Ilustração 6.8.1 e dividida nos seguintes grupos:

- **Verificação no nível de declaração**: É composto por Rótulos que contém um algoritmo codificador para representar o grafo e sub-grafo de fluxo de controle do programa. As assinaturas nestes campos identificam a posição do programa e os sucessores válidos e contém uma única assinatura de referência para o nó no grafo de controle de fluxo (*Control Flow Graph* - CFG);
- **Verificação do procedimento**: Contém campos que permitem o WDP verificar o retorno ou chamada de um procedimento, e assim, as assinaturas são empilhadas ou desempilhadas na área de memória do WDP;
- **Verificação no nível de processo**: Contém campos que permitem o monitoramento pelo WDP das aplicações em um determinado processador, a verificação do escalonamento e a interação entre diferentes processadores. Por exemplo, pode monitorar o tempo de transferência de uma assinatura para detectar se o sistema entrou em um *laço infinito* e não responde a qualquer instrução (*hung*).

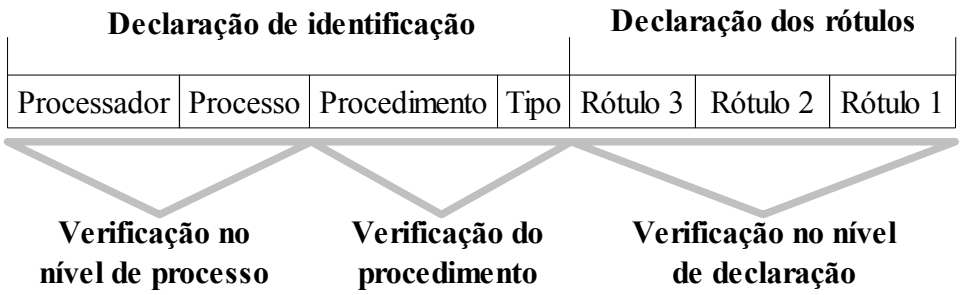


Ilustração 6.8.1: Estrutura da assinatura.

Os primeiros e últimos campos da **Verificação do procedimento** são marcados por dois indicadores (*flags*): Começo do Procedimento (*Start of Procedure* - SOP) e Fim do Procedimento (*End of Procedure* - EOP) [81]. O indicador SOP significa que o WDP tem que empilhar a assinatura de referência atual na memória e um campo da **Declaração dos rótulos** é determinado como a primeira referência do procedimento chamado. No caso do indicador EOP um campo da **Declaração dos rótulos** deve ser validado e a próxima referência tem que ser desempilhada (a referência do procedimento chamado).

Os procedimentos do programa são numerados e seus identificadores são embarcados nas assinaturas, junto com campo da **Declaração dos rótulos**. Um campo da **Verificação no nível de processo** pode ser alterado somente se um indicador de SOP for configurado.

A inicialização interna do WDP e a inicialização das tabelas de tradução de endereços na MMU são acionadas sempre que ocorre a criação ou escalonamento de um novo processo sob o controle do Sistema Operacional modificado.

6.9 Técnica *Watchdog* Co-Processador

Nesta abordagem, o *Watchdog* (WD) é um co-processador escalonável e extensível para multiprocessamento em ambientes que utilizam memória *cache*. O esquema desenvolvido não afeta significativamente a arquitetura do sistema, pois a utilização do WD não exige qualquer mudança no projeto do barramento, do sistema de memória, da arquitetura da memória *cache*, ou do projeto da CPU [79].

A Ilustração 6.9.1 mostra a configuração do *Watchdog* co-processador. A comunicação com WD é estabelecida utilizando instruções de co-processamento. Esta abordagem permite ao WD co-processador receber de herança todas as funções de transferência de dados de um co-processador e especificar suas próprias operações de compactação de dados. Para verificação do fluxo de controle, a compactação pode ser baseado com LFSR (*Linear Feedback Shift Register*) ou cálculo baseado no somador de *checksum*.

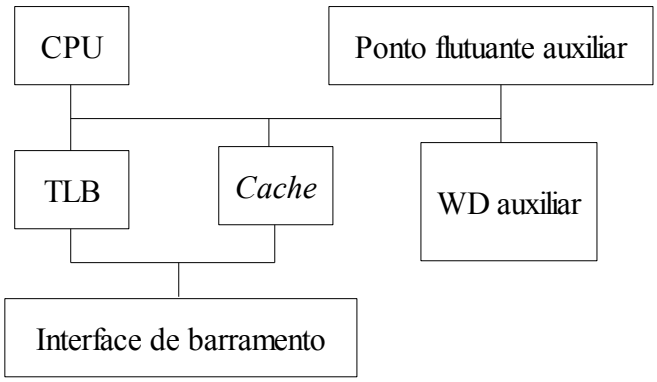


Ilustração 6.9.1: *Watchdog* co-processador [79].

A Ilustração 6.9.2 representa um bloco de instruções livres de desvio e mostra o uso da técnica *extended-precision checksum* para compactar as instruções no fluxo de controle verificado. O *extended-precision checksum* do bloco de instrução é a soma total da ***instrução_1*** até a ***instrução_s*** ou a soma total de alguma transformação sobre estas instruções (alterações devido a presença de falhas). As instruções explícitas do WD co-processador primeiramente transmitem o *extended-precision checksum* que representa a soma total da ***instrução_1*** até a ***instrução_s*** do bloco básico de instruções.

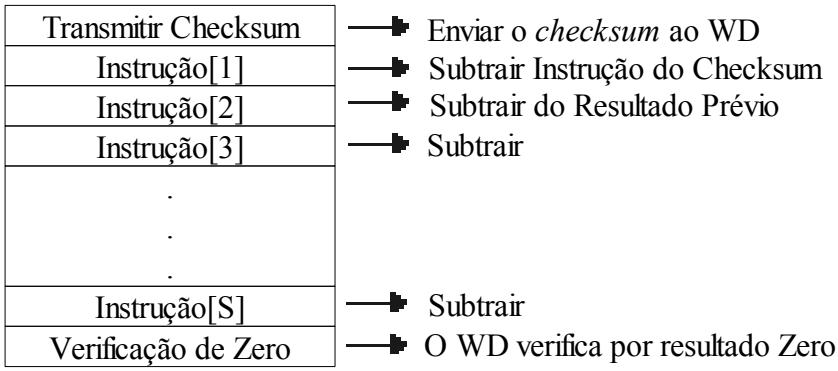


Ilustração 6.9.2: Verificação de controle de fluxo por *extended-precision* [79].

Uma única instrução de escrita do processador pode ser utilizada para transmitir a assinatura compactada ao WD, já que as instruções são compactadas em intervalos de n-bits para r-bits (r < n) e o *extended-precision checksum* é calculado sobre as instruções compactadas. Depois do WD receber o valor do *extended-precision checksum*, ele começa a subtrair instruções como mostra a Ilustração 6.9.2. No fim do bloco, o WD verifica por um resultado zero. A verificação de zero pode ser disparada no início de uma instrução de desvio. Portanto:

1. Um erro é detectado se o resultado do WD for negativo antes de receber um verificador de zero.
2. Um erro é detectado se o resultado do WD não for zero quando receber verificador de zero.
3. Um erro é detectado se o WD receber um indicador de zero antes do *checksum* ser transmitido.

6.10 Resumo das Técnicas de Detecção de Erro via *Hardware*

Todas as técnicas apresentadas nos tópicos anteriores foram resumidas e apresentadas nos campos da Tabela 6.10.1:

Técnica	Método	Cobertura	Comentários
Concurrent Process Monitoring with No Reference Signatures [75]	Uma função de assinatura é aplicada no fluxo de instruções em tempo de compilação. Quando as assinatura formarem o código *m-out-of-n* ou executar uma instrução de desvio, a instrução é sinalizada. Em tempo de execução o WDP detecta a instrução sinalizada e o código gerado *m-out-of-n*.	Não existe implementação e a cobertura é estimada em função do m e do n do código *m-out-of-n:* A cobertura é máxima se m=n : $1-\frac{(\frac{n}{m})-1}{2^n-1}$ [75]	Melhor na detecção de erros de *bit-flip* que no controle de fluxo.
Continuous Signature Monitoring (CSM) [76]	O código é dividido em blocos básicos e é gerada uma assinatura de referência para cada bloco e inserida uma instrução (*opcode*) no final do bloco. Em cada instrução de assinatura, o processador executa instruções de NOP enquanto o Monitor compara com a assinaturas de referência, acusando um erro se elas não forem iguais.	Cobertura: Estimada em $1-2^{-v}=99.9999\,\%$ para um processador de *v* = 32-bit [76].	Apresenta baixa latência para detecção de erros de *bit-flip*, mas alta latência para detecção de erros de fluxo de controle.
Watchdog Direct Processing (WDDP) [78]	O WDDT calcula a assinatura da seqüência executada de instruções nos blocos básicos, se houver uma instrução de parada, o WDDT verifica se o nó pertence a um endereço correto. No caso de desvio, sabendo o endereço do próximo nó a ser executado, é verificado e a assinatura é restaurada.	Não avaliado.	Não se aplica em processadores com *caches* de instrução interna, pois pequenos laços podem ser completamente executados dentro da *cache* sem o alcance do WDDP.

Técnica	Método	Cobertura	Comentários
Asynchronous Signatured Instruction Streams (ASIS) [67]	A assinatura é embarcada no fluxo de instruções da aplicação no nível de linguagem de máquina.	Não foram realizados testes devido a complexidade de modificar o *assembler* e o *linker*. Recursos de *hardware* necessário: WDP e HSG	Todas as chamadas para sub-rotinas devem retornar ao mesmo lugar aonde a chamada se originou. Além disso, não é permitido interrupções de troca de contexto. Um grande espaço de memória pode ser necessário, justo ao Monitor, para armazenar o grafo que representa a aplicação monitorada.
On-line Signature Learning and Checking (OSLC) [6,7]	O código é dividido em seções, cada uma com vários intervalos livres de desvio. Ao receber a saída de um bloco-D pelo gerador de assinatura, o Verificador procura se a assinatura encontra-se nos blocos da mesma seção.	Gerador de número pseudo-randômico, procura de *string*, manipulação de bits, classificação rápida, gerador de número primo no Z-80 com cobertura de 86.3% [6,7].	É necessária a sincronização entre o processador, o gerador de assinatura, e o Monitor.
Signature Encoded Instruction Stream (SEIS) [8,9, 10].	As assinaturas são enviadas ao WDP para verificar a execução correta da seqüência da assinatura recebida, para que a seqüência corresponda a um fluxo de programa existente.	Whetstone, dhrystone, Unpack, multigrid 5; Cobertura: 20-50%. [8,9, 10]	Não permite que o código compartilhe processos diferentes ou preemptivos.
WD co-processador [79]	No início de cada bloco básico, o WD recebe o valor do *extended-precision checksum*. Após, executa subtrações do número de instruções até o final do bloco básico, então, o WD verifica por um resultado zero.	Não avaliado.	É necessária a sincronização entre o processador e o WD, e acesso ao barramento de dados.

Tabela 6.10.1: Resumo das técnicas de detecção de erro por *Hardware*.

PARTE II - METODOLOGIA

7 IMPLEMENTAÇÕES DO WDP-IP

7.1 Introdução

De acordo com os capítulos anteriores, é possível concluir que as técnicas de detecção de erros podem ser divididas sinteticamente em dois grandes grupos: técnicas baseadas em *hardware* e técnicas baseadas em *software*. Dentre as soluções baseadas em *hardware*, descritas detalhadamente no capítulo 6, salienta-se o uso do WDP que consiste em agregar um processador extra para monitorar constantemente a atividade realizada pelo processador principal do sistema, e ativar um procedimento de tratamento de erro caso o seu comportamento seja diferente do esperado.

O WDP desenvolvido nesta dissertação, denominado WDP-IP tem como aplicação sistemas embarcado de tempo real. Estes sistemas, cujas características dependem do cumprimento de requisitos temporais e lógicos e onde as conseqüências do não cumprimento desses mesmos requisitos podem causar prejuízos, como por exemplo na segurança de pessoas. Nesta perspectiva, em conjunto com o WDP-IP (*Watchdog Processor - Infrastructure Intellectual Property*), foi utilizado o Sistemas Operacional de Tempo Real (*Real-Time Operating Systems* - RTOS) uC/OS-II no ambiente multitarefa executando aplicações preemptivas e interconectadas por semáforo, na qual, várias tarefas críticas devem ser processadas simultaneamente. O WDP-IP, implementado tanto em *hardware* quanto em *software* deve verificar se as tarefas críticas são processadas dentro de um tempo a elas alocado.

Os sistemas operacionais em geral tem seu próprio mecanismo de detecção de erro no caso de espera prolongada por algum recurso que pode ser um semáforo, uma mensagem um tempo em estado ocioso, etc. Já o WDP-IP implementa uma nova concepção no controle das tarefas executadas sob o escalonamento do sistema operacional, pois monitora cada tarefa pelo tempo em que ficou alocado na CPU e o tempo em que esta tarefa ficou ociosa esperando ser executada na CPU.

7.2 O Sistema Operacional de Tempo Real uC/OS-II

Uma pergunta comum no momento da especificação de requisitos de sistemas de tempo real é: “Uma solução de tempo real deve utilizar um sistema comercial ou deve elaborar ou adaptar um?”. A resposta depende da situação, RTOS comerciais são freqüentemente escolhidos, pois geralmente provêm serviços robustos, são fáceis de usar, e podem ser portáveis à plataforma requerida [42].

A seleção do RTOS uC/OS-II foi motivado pela perspectiva de que este sistema é extensamente utilizado há vários anos, em aplicações críticas industriais para garantir um comportamento correto e seguro e estudado pelo meio científico através de técnicas de injeção de falhas [82]. É possível encontrar diversos aplicativos e sistemas como robôs industriais, controle de motor, instrumentos médicos, etc., que utilizam este sistema operacional.

O RTOS uC/OS-II é uma aplicação do programa que controla a execução de diversos programas, fazendo o escalonamento entre as tarefas para a computação de um único microprocessador. É um RTOS de código pequeno (pode ocupar de 10KB a 100KB dependendo da configuração e aplicação utilizada), porém muito poderoso [14]. Foi desenvolvido por Jean J. Labrosse. O código fonte é fornecido junto com o livro “*MicroC/OS-II The Real-Time Kernel*” [14], pode ser utilizado para aplicações não-comerciais e finalidades educacionais.

7.2.1 Características do RTOS uC/OS-II

A seguir são citadas as principais características do RTOS uC/OS-II [14]:

a) Portabilidade: uC/OS-II é escrito em C-ANSI, com uma parte do código específico ao microprocessador, geralmente escrito em *assembly*. Pode ser portado para centenas de processadores desde que tenham uma CPU, ponteiro de pilha para que os registradores possam ser empilhados e desempilhados. Pode rodar em processadores de 8-bit, 16-bit, 32-bit ou mesmo nos microcontroladores de 64-bit e DSP’s;

b) Embarcável: foi desenvolvido para aplicações embarcadas, isto significa que com as ferramentas apropriadas (compilador, assemblador, ligador e carregador), o uC/OS-II pode ser parte de um produto;

c) Escalonável: é possível usar somente os serviços do uC/OS-II requisitados pela aplicação, permitindo reduzir-se a quantidade de memória (RAM e ROM) necessária;

d) Preemptivo: o uC/OS-II é baseado em prioridades (segundo a Ilustração 7.2.1.1), a tarefa de prioridade mais elevada assume o controle da CPU;

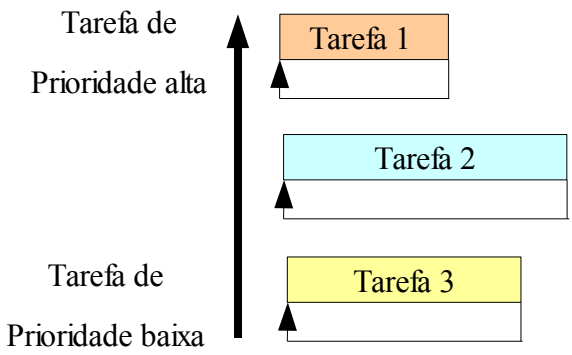


Ilustração 7.2.1.1: Sistema Preemptivo [14].

e) Multiprocessamento: o uC/OS-II pode controlar até 64 tarefas, entretanto, a versão atual do software reserva oito destas tarefas para o uso do sistema. Cada tarefa tem uma prioridade original atribuída, significa que uC/OS-II não pode fazer escalonamento circular (*round robin*);

f) Determinístico: O tempo de execução de todas as funções e serviços do uC/OS-II são determinísticos. Pode-se saber exatamente o tempo que o uC/OS-II executará uma função ou um serviço;

g) Pilhas de Tarefa: Cada tarefa requer sua própria pilha na memória, o uC/OS-II permite que cada tarefa tenha um tamanho diferente nesta pilha. Isto permite reduzir a quantidade de RAM necessária na aplicação desenvolvida;

h) Serviços: possui diversos serviços, tais como caixas postais (*mailboxes*), filas, semáforos, tamanhos de memória de partição fixa e funções de tempo relacionados, entre outros;

i) Gerenciamento de Interrupções: As interrupções podem suspender a execução de uma tarefa. Se uma tarefa de prioridade mais elevada aguarda por execução, assumirá o controle da CPU. As interrupções podem ser empilhadas até 255 níveis profundamente;

j) Robusto e confiável: o uC/OS-II é baseado no uC/OS, que foi utilizado em centenas de aplicações comerciais desde 1992, tais como: aplicações industriais, robôs, controlador de motores, instrumentos médicos, etc.. É 99% compatível com os padrões de código da associação de confiabilidade de software da indústria de motores MISRA (*Motor Industry Software Reliability Association*). Possui certificado para aplicações aeroespaciais RTCA / EUROCAE DO-178B Nível A, onde uma falha pode causar uma perda do avião e uma catástrofe. Foi aprovado para uso em dispositivos médicos FDA Classe III onde um falha pode resultar na perda da vida do paciente ou do caso clínico [14].

7.2.2 Arquitetura do RTOS uC/OS-II

O RTOS fornece uma plataforma virtual de alto nível ao programador escondendo os detalhes do *hardware* e facilitando o desenvolvimento de programas que utilizam os recursos do sistema. A arquitetura do uC/OS-II é distribuída basicamente em camadas *software* e *hardware* conforme mostra a Ilustração 7.2.2.1:

- A camada de *hardware* deve prover os recursos básicos para operação do RTOS uC/OS-II que são um microprocessador com CPU e ponteiros de pilha com registradores que possam ser empilhados e desempilhados para trocas de contexto e interrupção por um temporizador.

- Na camada de *Software*, podemos identificar 4 segmentos:

a) Porte: Se encontra a parte de código dependente do microprocessador, incluindo macros que definem os protótipos de funções, tamanho de declaração das variáveis, funções em *assembler* específicas para troca de contexto, entrada em modo supervisor, etc. Nestes arquivos de porte, foram introduzidas chamadas para o WDP-IP informando a Tarefa que se encontra em execução;
b) Configuração: Se encontram arquivos da parte de código específico para a aplicação (no arquivo app_cfg.h), configuração dos recursos do RTOS (no arquivo os_cfg.h), rotinas para interfaceamento com os recursos da placa (arquivos bsp.c e bsp.h);

c) O uC/OS-II: é o RTOS referido neste trabalho, cujo código é independente do microprocessador utilizado;

d) Software de aplicação: é a aplicação desenvolvida pelo usuário. Neste trabalho, foram utilizadas três Tarefas. Sempre no início da execução do RTOS em uma sub-rotina chamada "*begin hook*", as interrupções são desabilitadas, e em uma subrotina "*end hook*" onde o WDP-IP é configurado.

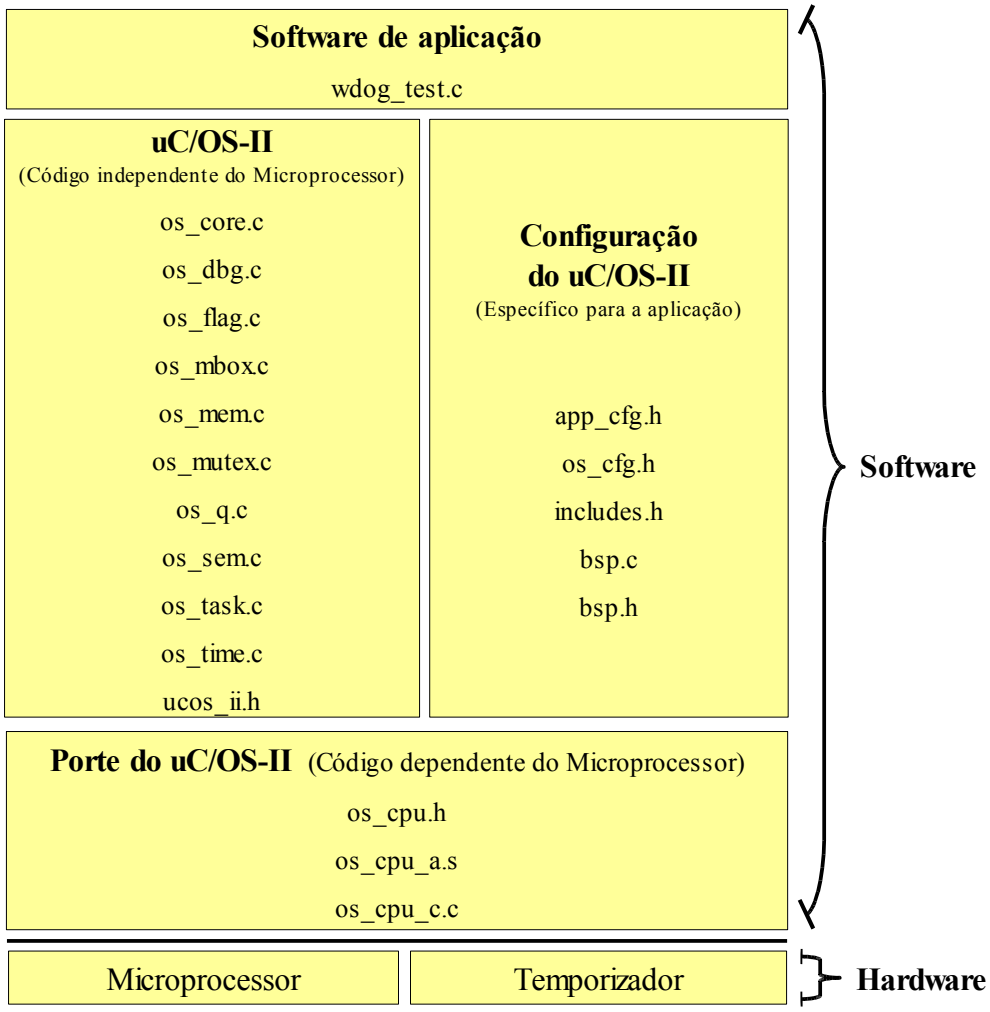


Ilustração 7.2.2.1: Arquitetura e arquivos do RTOS uC/OS-II utilizado [14].

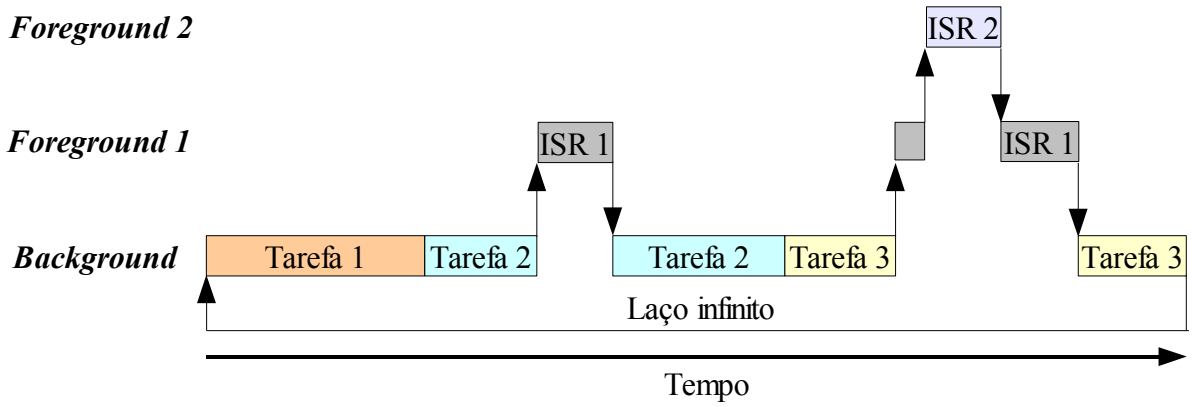
7.2.3 Escalonamento do RTOS uC/OS-II

O processo de escalonamento de tarefas permite, estruturar uma aplicação em um conjunto de tarefas menores e dedicadas, que partilham o mesmo processador permitindo ao programador lidar com situações complexas que são inerentes em aplicações de tempo real.

Uma tarefa consiste em um programa único que supõe estar usando a CPU sozinho. A realização de uma aplicação de tempo real envolve a divisão do trabalho que será feito por tarefas que serão responsáveis por uma porção do problema [14].

A Ilustração 7.2.3.1 mostra o sistema denominado Primeiro/Segundo plano (*foreground/background*). Uma aplicação consiste em Tarefas de laço infinito (demonstrada na Ilustração 7.2.3.3) que requerem ao RTOS realizar/executar as operações que se deseja. As Tarefas são executadas seqüencialmente em segundo plano (*background*) e são interrompidas por rotinas de serviço de interrupções (*Interrupt Service Routines* - ISRs) que lidam com eventos assíncronos no primeiro plano (*foreground*).

As operações críticas são executadas pelo ISRs de modo a garantir que estas serão executadas o mais rápido possível "*best effort*". Devido a este fato, ISRs são tendencialmente mais demoradas do que deveriam ser.



Background - seqüência temporal principal, onde roda as aplicações de usuário.
Foreground - seqüência temporal secundária, onde o sistema operacional faz escalonamento de tarefas e também ocorrem interrupções do sistema.

Ilustração 7.2.3.1: Sistema Primeiro/Segundo plano *(Foreground/Background*) [14].

A informação para uma Tarefa em *background*, que pode ser acessada por uma ISR, só será processada quando a rotina *background* estiver ápta para ser executada. Neste caso a latência depende de quanto tempo o laço em *background* demora a ser executado.

Na Ilustração 7.2.3.2, um exemplo de aplicações do tipo *foreground/background*:

```
/* Background */
void main (void)
{
  Inicialização;
  Sempre {
    Leitura de entradas analógicas;
    Leitura de entradas discretas;
    Monitoração de funções;
    Controle de funções;
    Atualizar saídas analógicas;
    Atualizar saídas discretas;
    Varredura do teclado;
    Atualizar display;
    Outros...
  }
}
```

```
/* Foreground */
ISR (void)
{
    Evento assíncrono;
}
```

Ilustração 7.2.3.2: Aplicações do tipo *foreground/background* [14].

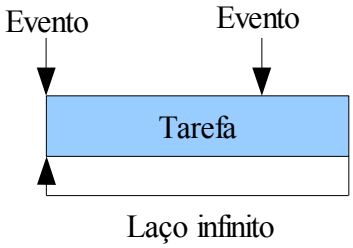


Ilustração 7.2.3.3: Execução de cada Tarefa [14].

O escalonador é a parte do RTOS responsável por decidir a sequência de execução das tarefas. O núcleo do uC/OS-II é baseado numa estrutura de ordem de prioridade: cada Tarefa tem uma prioridade associada de acordo com a sua importância, descrita na Ilustração 7.2.1.1. Deste modo, o controle da CPU será atribuído à tarefa com prioridade mais elevada que está pronta para ser executada. Como o uC/OS-II é um RTOS com recursos de escalonamento preemptivo, um evento (interrupção ou função de escalonamento) transforma uma tarefa de prioridade mais elevada pronta para ser executada, a tarefa atual é imediatamente suspensa e a CPU é cedida para a execução desta tarefa de maior prioridade.

O processo que consiste em escalonar e distribuir o tempo da CPU entre várias tarefas de uma única CPU é realizado de forma seqüencial, a Ilustração 7.2.3.4 mostra os estados permitidos que as tarefas podem assumir segundo o RTOS uC/OS-II:

a) Executando no processador (*running*): a CPU está executando as instruções que compõem esta tarefa. Como o sistema não é multi-processado, há no máximo uma tarefa sendo executada a cada instante;

b) Pronta (*ready*): quando o processador está executando uma outra tarefa, mas se uma tarefa em estado de pronta de maior prioridade ficar disponível, isto é, pronta para execução, a CPU poderá executar esta nova tarefa (pode haver um número qualquer de tarefas prontas). Em outras palavras, uma tarefa em estado de pronta pode executar se existir atualmente outras tarefas de prioridade menor em execução ou em estado de pronta;

c) Bloqueada (*waitting*): quando a tarefa estiver ociosa, esperando por algum evento externo. Pode haver um número qualquer de tarefas neste estado A tarefa pode estar esperando uma operação de E/S, algum recurso compartilhado entre as demais tarefas, um evento temporário, etc.;

d) Latente (*dormant*) corresponde a uma tarefa que reside na memória, mas não está disponível no ambiente;

e) Interrompida (ISR): quando uma interrupção ocorreu e a CPU está atendendo aos serviços de interrupção.

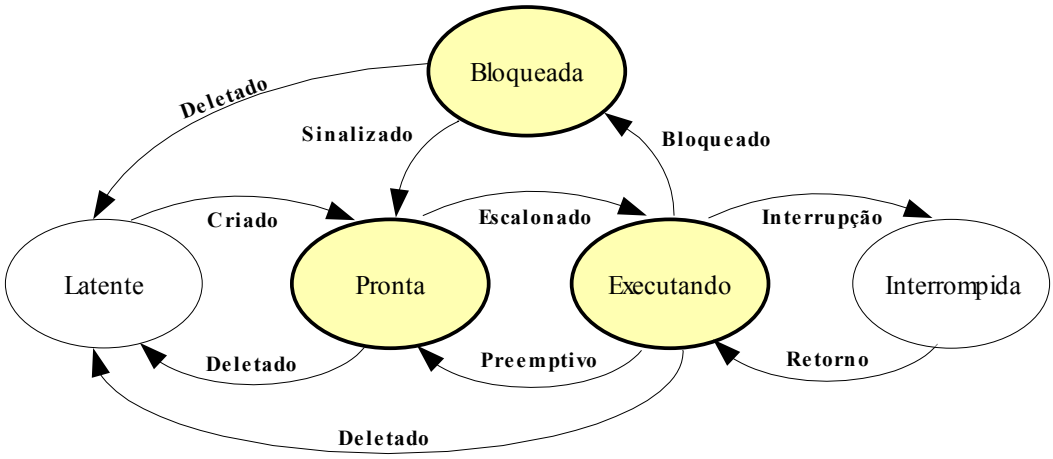


Ilustração 7.2.3.4: Estados de uma tarefa no RTOS uC/OS-II [14].

7.3 Abordagem Geral da Arquitetura Proposta

A Ilustração 7.3.1 apresenta de modo geral a arquitetura e implementação proposta para o WDT-IP [15,16,17], assumindo que o sistema alvo está executando, sob o controle do RTOS uC/OS-II, em ambiente multitarefa. O WDP-IP possui duas implementações diferentes a saber:

- WDP-IP/HW - implementado em *hardware* e descrito na linguagem VHDL;
- WDP-IP/SW - implementado em *software* e descrito na linguagem C ANSI.

O WDP-IP apresenta detecção de erro de fluxo de controle (provinientes de pertubações externas como citado na seção 2.5 na página 45) em tarefas na aplicação do usuário e no código do RTOS, através do controle do tempo de execução de cada um. Compreende-se que as falhas no sistema alvo que afetam o fluxo de controle são aquelas que obrigatoriamente mudam o tempo (acrescentando ou reduzindo) das tarefas monitoradas em durante a execução do programa.

Deste modo, a fim de detectar a ocorrência de erros do fluxo de controle, a função do WDP-IP é monitorar o número de ciclos de *clock* exigidos para executar as tarefas da aplicação.

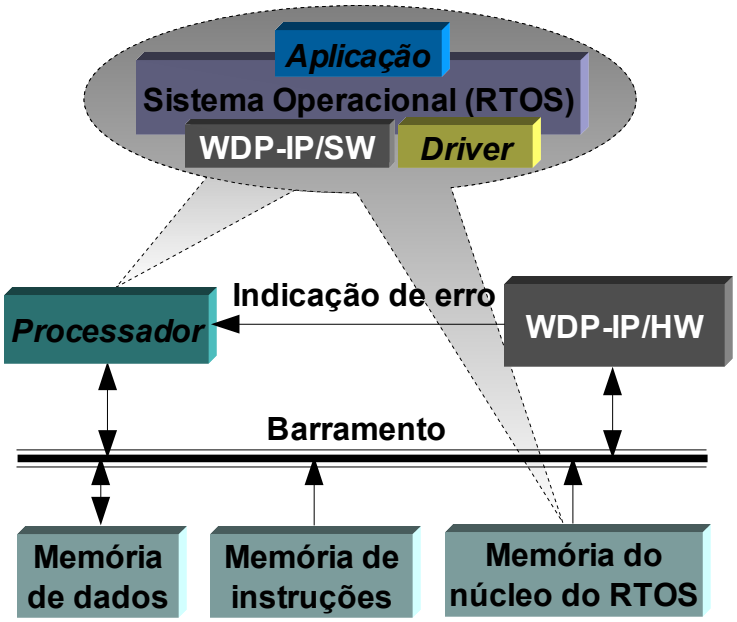


Ilustração 7.3.1: Abordagem geral da arquitetura do WDP-IP proposta [15,16,17].

7.4 Desenvolvimento do *Hardware* Implementado

A Ilustração 7.4.1 representa a estrutura em diagrama de blocos do SoC em *hardware* implementado. A descrição de cada bloco é abordada nas seções seguintes.

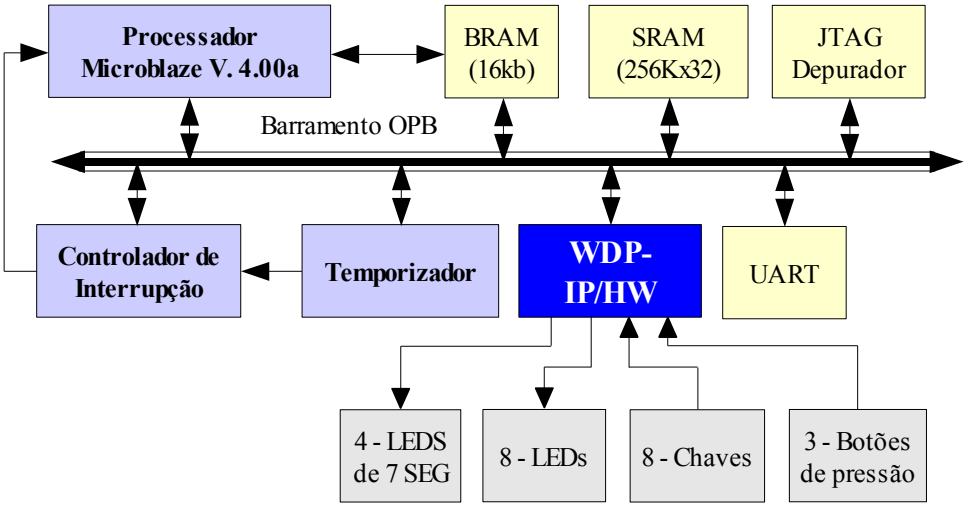


Ilustração 7.4.1: Diagrama em blocos do SoC, salientando-se as conexões do processador, WDP-IP/HW e memórias através do barramento OPB.

7.4.1 Bloco BRAM

A plataforma de desenvolvimento Xilinx Spartan3 FPGA [18] utilizado nesta pesquisa, contém um recurso muito útil que são os blocos de memória (*Block Rams* - BRAM) 216-*Kbit*. A BRAM têm duas portas totalmente independentes de acesso (*dual port*), entretanto, pode-se ler e escrever na mesma posição que serão fornecidos os dados velhos ou recentemente escritos.

Os dados e instruções são armazenados normalmente na BRAM utilizando o controlador de BRAM para o barramento OPB (*On-chip Peripheral Bus*), ao se carregar o *bitstream* com o programa compilado. Porém, como este espaço de memória não é suficiente para carregar a aplicação utilizada nesta dissertação, esta área não foi utilizada.

7.4.2 Bloco JTAG Depurador

O bloco JTAG Depurador é utilizado para conectar o Aplicativo depurador (*debugger)* XMD/GDB. Pode servir como uma ferramenta para carregar na SRAM, o programa de aplicação e na elaboração das aplicações desenvolvidas pelo programador de *software*.

7.4.3 Bloco Temporizador

O bloco Temporizador consiste em um contador que decrementa ou incrementa na velocidade do *clock* do processador. O processador pode ler o contador e o Temporizador pode gerar um pulso de interrupção de saída quando o contador passar por zero [4].

Todos os RTOS requerem uma interrupção periódica do sistema ou um "*tick tack*" (*clock tick*) para executar seus serviços. Por exemplo, as tarefas do usuário podem aguardar por um determinado tempo "*tick tacks*" do sistema chamando uma função do RTOS uC/OS-II do tipo OSTimeDly(). Adicionalmente, o uC/OS-II fornece uma facilidade de intervalo (*time-out*) para impedir que as tarefas esperem por um determinado recurso indisponível. E, finalmente, durante cada "*tick tack*" a ISR (*Interrupt Service Routines*) do RTOS uC/OS-II chama o escalonador para determinar se uma tarefa de prioridade mais elevada está pronta para funcionar. A taxa do "*tick tack*" do sistema é dirigida a aplicação, mas encontra-se geralmente entre 100 ms a 5 ms [14]. Aumentando a freqüência do "*tick tack*" do sistema aumentará custos gerais do RTOS e reduzirá o tempo disponível da tarefa. Tipicamente, o caso de pior tempo de execução WCET deve ser realizada na aplicação para determinar a melhor taxa de "*tick tack*" do sistema.

O temporizador para os experimentos com o WDP-IP/HW, foi utilizado em conjunto com o bloco de interrupção para gerar uma interrupção periódica no processador. Este temporizador foi ajustado para gerar uma interrupção a cada 10 ms.

7.4.4 Bloco UART

O bloco UART é usado para conectar o PC via porta serial com a placa de desenvolvimento onde encontra-se o processador Microblaze e as técnicas embarcadas na FPGA, permitindo assim a visualização dos dados enviados da aplicação, sob o controle do RTOS, pela serial na tela do PC. O bloco UART foi ajustado na taxa de 57600 *baud rate*, 8 bits de dados, nenhuma paridade, um bit de paridade, e nenhum controle de fluxo de *hardware*.

7.4.5 Bloco Controlador de interrupção

É um controlador de interrupção simples e parametrizável que está conectado no barramento OPB. O controlador de interrupção esta conectado na saída da interrupção do Temporizador com a entrada da interrupção do processador.

7.4.6 Bloco WDP-IP/HW

O WDP-IP/HW é a implementação do WDP-IP em *hardware* cuja função é a de monitorar a aplicação a partir dos tempos de execução das Tarefas do usuário ou do núcleo do uC/OS-II no modo supervisor. Indicar qual Tarefa não cumpriu, seus prazos de tempo requeridos caso o tempo de execução ultrapassar um valor determinado ou quando este tempo não foi atingido, ou se durante algumas execuções do sistema a Tarefa não foi executada.

No ambiente de sistema embarcado de tempo real, as Tarefas tem um comportamento e tempo de execução determinístico, o que possibilita o programador informar (através de um comando que será descrito na seção 7.6 na página 113) ao WDP-IP o tempo de execução na aplicação de cada Tarefa sob controle de detecção de erros, caso não seja informado pelo programador, em uma fase de auto-aprendizado, o WDP-IP calcula automaticamente este tempo após ter executado pelo menos uma vez cada Tarefa.

O bloco WDP-IP/HW controla os seguintes sub-blocos para fins de depuração:

1) Sub-bloco 8-LEDs (8 diodos luminosos de 0 a 7): Exibe a Tarefa e/ou se o kernel do RTOS apresentou erro devido a disturbios extenos. O LED-0 ascende se

apresentou erro durante a eecução do RTOS, o LED-1 se for detectado erro durante a execução na Tarefa 1, o LED-2 na Tarefa 2 e assim por diante;

2) Sub-bloco 4-LEDs de 7-SEG (4 diodos luminosos de 7 segmentos): Mostra se ocorreu erro, a tarefa atual em execução e sua contagem. O LED-1 exibe “C” caso ocorreu erro detectado por curto tempo de execução da Tarefa, exibe “L” caso erro detectado por longo tempo de execução da tarefa e “O” quando a tarefa ficar ociosa. O LED-2 exibe a tarefa atual em execução, os LED-3 e LED-4 exibem os 8 bits menos significativos da contagem atual da tarefa em execução;
3) Sub-bloco 8-Chaves: Servem para mudar a velocidade de amostragem dos LEDs de 7 segmentos, no mínimo é uma amostra a cada 1 ms e o máximo é a cada 255 ms;
4) Sub-bloco 3 Botões de pressão: O Botão-0 (BTN0 – na placa de desenvolvimento) quando pressionado, reinicia o WDP-IP/HW, zerando todos os registradores e a tabela de memória interna. O Botão-1 (BTN1) quando pressionado, limpa o registrador que acusa erro e toda a coluna da tabela de memória interna em que consta os valores utilizado pelo contador.

7.5 Mapeamento do Barramento

O processador MicroBlaze possui até 4 Giga bytes de espaço de endereçamento, que podem ser usados, por exemplo, para conectar dispositivos de E/S que residem na barramento do processador. Para o desenvolvimento deste projeto, os dispositivos utilizados estão mapeados como mostra a Tabela 7.5.1:

Módulo	Faixa de endereço
BRAM	0x00000000-0x00003FFF
CONTROLADOR DE INTERRUPÇÃO	0x41200000-0x4120FFFF
JTAG DEBUG	0x41400000-0x4140FFFF
SRAM_256Kx32	0x20100000-0x201FFFFF
TEMPORIZADOR	0x41c00000-0x41c0FFFF
UART	0x40600000-0x4060FFFF
WDP-IP/HW	0x76400000-0x7640FFFF

Tabela 7.5.1: Mapeamento do barramento.

7.6 Funções Inseridas no uC/OS-II para a Comunicação com o WDP-IP

A descrição das funções inseridas no núcleo do uC/OS-II (conforme comentado na seção 7.2.2 na página 103) que permitem a comunicação com o WDP-IP via barramento estão descritas a seguir:

1) Função **ip_adq()**: Inserida em uma Tarefa no código pelo programador (tanto em modo usuário quanto supervisor) para receber o estado do WDP-IP. O tipo de estado solicitado é definido pela função **ip_cmd** que será descrito a seguir;
2) Função **ip_cmd(*flag*)**: Essa função tem como objetivo enviar comandos ao WDP-IP. Através de diferentes combinações no ***flag*** passado, pode-se efetuar diversos requisitos. Para ficar mais fácil a utilização, foram definidas diversas constantes e incorporadas no núcleo do RTOS. Estas constantes atribuem um determinado valor para o ***flag*** passado na função e foram nomeadas como segue:
    a) O ***flag*** igual a **(p_wdst)**: Permite reiniciar o WDP-IP;
    b) O ***flag*** igual a **(p_maxc|Tarefa)**: O WDP-IP carrega um registrador interno com o valor máximo do contador da **Tarefa** informada para que a função **ip_adq()** busque posteriormente;
    c) O ***flag*** igual a **(p_cont|Tarefa)**: O WDP-IP carrega um registrador interno com o valor da contagem atual da **Tarefa** informada para que a função **ip_adq()** busque posteriormente;
    d) O ***flag*** igual a **(p_rst|Tarefa)** : Permite zerar e iniciar o contador da **Tarefa** informada na aplicação desenvolvida pelo usuário;
    e) O ***flag*** igual a **(p_falha)**: O WDP-IP carrega um registrador interno informando a Tarefa que falhou para que a função **ip_adq()** busque posteriormente;
    f) O ***flag*** igual a **(p_ksrt)**: Semelhante a variável **(p_rst|Tarefa)** utilizada pelo usuário, essa variável é utilizada pelo núcleo do RTOS para zerar e iniciar o contador de modo supervisor;
    g) O ***flag*** igual a **(p_kstp)**: Utilizado pelo núcleo do RTOS para parar o contador de modo supervisor;
    h) O ***flag*** igual a **(p_kact|Tarefa)**: Utilizado pelo núcleo do RTOS para informar ao WDP-IP a **Tarefa** atual em execução;

3) Função **ip_wrt(Tarefa,Valor)**: O WDP-IP calcula automaticamente o tempo de execução (**Valor**) de cada **Tarefa**, desde que pelo menos esta **Tarefa** seja executada integralmente ao menos uma vez e que essa função não seja utilizada previamente para informar o tempo de execução da referida **Tarefa**. Se o usuário necessitar determinar ou alterar o **Valor** de tempo de execução de uma determinada **Tarefa**, com essa função é possível passar como parâmetros o valor máximo de contagem (**Valor**) da Tarefa informada (**Tarefa**);
4) Função **ip_idle(Valor)**: É passado ao WDP-IP o **Valor** do número de vezes em que é executado qualquer **Tarefa** em um dado intervalo de tempo. Se alguma **Tarefa** não foi executada dentro desse número de vezes, o WDP-IP acusa erro na **Tarefa** que ficou em estado ocioso.
5) Função **ip_sw**: essa função é específica e somente necessária em RTOS preemptivo, pois serve para informar ao WDP-IP que o RTOS esta em execução na CPU e qual tarefa a ser executada irá preemptar. Esta função utiliza os recursos da função **ip_cmd** com o *flag* igual a (**p_kact|Tarefa**) para determinar a tarefa atual, o *flag* igual a **(p_ksrt)** para informar ao WDP-IP que o RTOS esta utilizando a CPU e o *flag* igual a **(p_kstp)** para informar ao WDP-IP que o RTOS deixou a CPU disponível.

7.7 Esquema de Comunicação Fundamental para o WDP-IP

A Ilustração 7.7.1 mostra as funções básicas para comunicação do processador com o WDP-IP, exemplificando uma seqüência de execução das Tarefas.

Sempre no início de execução de cada Tarefa, é informado ao WDP-IP a Tarefa atual em execução que iniciou pelo comando **ip_cmd(p_rst|Tarefa)**, assim o WDP-IP zera o valor na contagem na memória indexada desta Tarefa e verifica também se o tempo de execução desta Tarefa foi aceitável, ou seja não extrapolou e nem foi muito rápido. Caso o WDP-IP não tenha o parâmetro de contagem desta Tarefa (não foi informado pelo usuário) ele assume que a contagem atual é o valor padrão em ciclos de *clock* desta Tarefa.

Sempre que é realizada a troca de contexto pelo RTOS, a função **ip_sw** é ativada para informar ao WDP-IP a Tarefa que será executada. Assim, o WDP-IP procura na tabela indexada as informações da referida Tarefa.

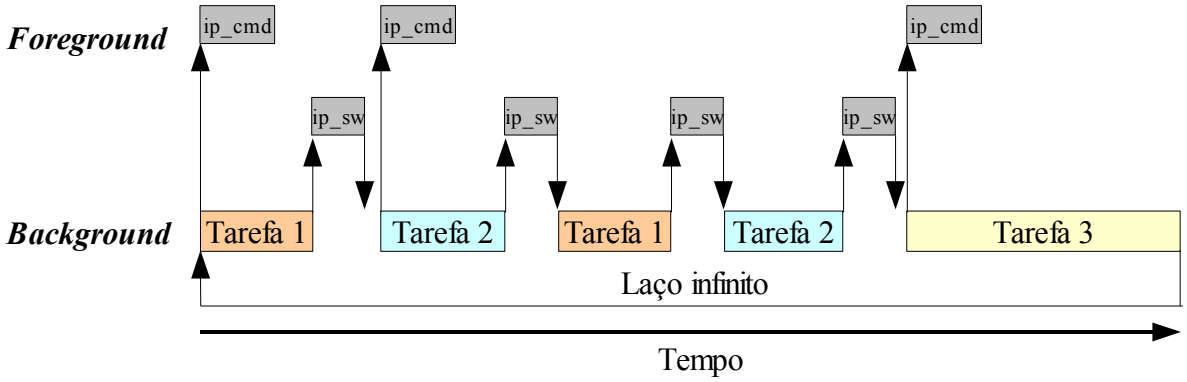


Ilustração 7.7.1: Funções básicas para comunicação com o WDP-IP.

7.8 Arquitetura do WDP-IP implementado em *Hardware* (WDP-IP/HW)

O WDP-IP/HW foi implementado em VHDL, encontra-se conectado ao barramento OPB e é acessado pelos endereços 0x76400000 para o registrador **reg0** e 0x76400002 para o registrador **reg1**, ambos de 16 bits. O registrador **reg0** recebe os comandos enviados via *software* pela Tarefa atual em execução e pelo núcleo do RTOS, enquanto que o registrador **reg1** recebe valores para serem carregados em sua tabela interna de memória. O WDP-IP/HW retorna também valores para depuração via *software* pelo endereço 0x76400002 (**reg1**), aos quais são solicitados via comando recebido no registrador **reg0**.

A seguir, na Ilustração 7.8.1, está representada a estrutura da comunicação entre o hardware WDP-IP/HW e o processador, aplicação via *driver* implementado no núcleo do RTOS uC/OS-II com suas respectivas funções e endereços utilizados.

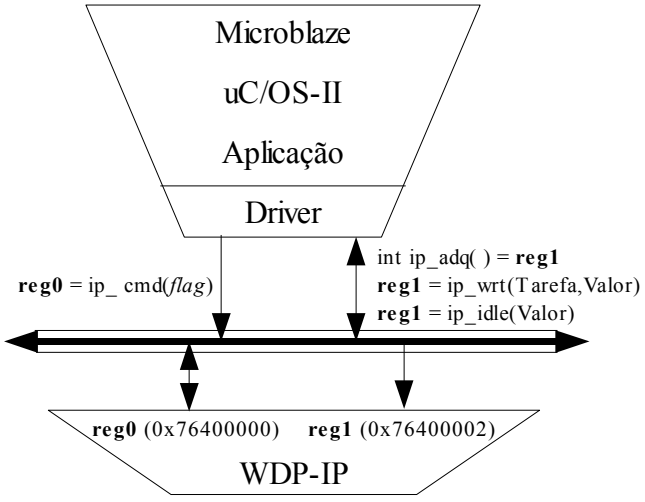


Ilustração 7.8.1: Comunicação processador e o WDP-IP/HW, com *driver* implementado no núcleo do uC/OS-II.

As funções inseridas no núcleo do uC/OS-II e os registradores respectivos usados que permitem a comunicação com o WDP-IP/HW via barramento são:

- Para o Registrador **reg0**: A função **p_cmd** carrega esse registrador, escrevendo assim as instruções a serem executadas pelo WDP-IP/HW;
- Para o Registrador **reg1**: São utilizados para escrita as funções **ip_ wrt** e **ip_idle** e para leitura a função **ip_adq**.

A Ilustração 7.8.2 exibe a arquitetura do WDP-IP implementado e *hardware* (WDP-IP/HW) em forma de diagrama de blocos.

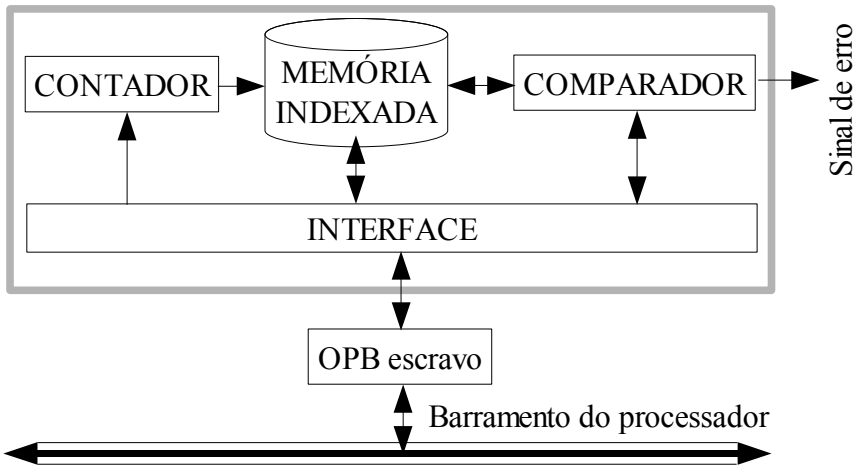


Ilustração 7.8.2: Arquitetura do WDP-IP/HW.

O princípio básico resumido do funcionamento interno de cada módulo do WDP-IP/HW é descrido a seguir:

a) O módulo **OPB escravo** representa para o programador, um endereço de acesso, na qual está conectado ao barramento com o mestre (processador Microblaze), com isso, pode-se comunicar escrevendo ou lendo dados nos registradores internos do WDP-IP/HW (nesta plataforma de teste foram utilizados 16 bits, mas podem ser de 8, 16 ou 32 bits). Esta comunicação é estabelecida por funções inseridas no programa do usuário com alterações no núcleo do RTOS conforme foi descrito na seção 7.6 na página 113.

b) O módulo **MEMÓRIA INDEXADA** é composto por campos denominados índice, ocioso, contador, contagem e paridade (demonstrados na Tabela 7.8.1). Os campos da tabela estão descritos como segue:

i) Campo **Índice**: está referenciado com o processo atual que está em execução na CPU. Esse campo é numerado com valor igual a "0" (zero) para o núcleo do RTOS na primeira linha da **MEMÓRIA INDEXADA**, "1" para a primeira Tarefa na segunda linha do respectivo módulo, "2" para a segunda Tarefa na terceira linha do respectivo módulo, e assim por diante.

ii) Campo **Ocioso**: é uma sinalização em que é configurada sempre que uma tarefa fica por um determinado tempo sem ser executada. O WDP-IP sinaliza erro pelo fato de alguma Tarefa estar ociosa. Este tempo é passado pela função **ip_idle(Valor)** na inicialização do RTOS e é determinada pelo usuário.

iii) Campo **Contagem**: representa a contagem atual da Tarefa, é medido em *clocks* do barramento OPB. Esta contagem pode ser dividida para equivaler ao *clock tick* do RTOS.

iv) Campo **Total**: representa a contagem total de execução da Tarefa, ou seja, seu valor máximo de contagem. Este valor é passado pelo usuário ou calculado automaticamente pelo WDP-IP.

v) O campo **Paridade** (ainda não implementado) serve para o WDP-IP realizar um auto-teste e informar ao programa do usuário se está tudo bem na resposta de um *ping*.

Índice	Ocioso	Contagem	Total	Paridade
0	B"0"	X"0000"	X"0000"	B"0"
⋮	⋮	⋮	⋮	⋮
N	B"0"	X"0000"	X"0000"	B"0"

Tabela 7.8.1: Memória indexada do WDP-IP.

c) O módulo **CONTADOR**, através do *clock do* barramento, faz o incremento no campo **Contagem** na tabela de **MEMÓRIA INDEXADA** da Tarefa atual em execução. Pode-se ter um divisor de *clock* para contagem que apresente tempo maior e/ou produza a contagem do *clock tick* do RTOS.

d) O módulo **COMPARADOR**, sinaliza com um erro caso a comparação do valor do campo **Contagem** é maior ou menor que o valor do campo **Total** na tabela de **MEMÓRIA INDEXADA** (uma margem de comparação é determinada pelo programador). Um erro também é sinalizado caso algum *bit* no campo **Ocioso** na tabela de **MEMÓRIA INDEXADA** foi configurado. Este erro pode ser exibido nos LEDs de 7 Segmentos, nos LEDs ou pelo próprio programa do usuário quando solicitado pela função **ip_adq**. A Tarefa que apresentou erro e o tipo do erro (se a Tarefa ultrapassou o tempo de execução, a Tarefa executou muito rápido ou a Tarefa ficou ociosa) também podem ser observados.

e) O módulo **INTERFACE**, é responsável pela comunicação e controle entre o WDP-IP/HW e o barramento de dados. Possui dois registradores que podem ser acessados para escrita ou leitura pelo processador. O primeiro registrador **reg0** é usado pelo processador para escrita de instruções a serem executadas pelo WDP-IP/HW, enquanto o segundo registrador **reg1** é usado pelo processador tanto para escrita quanto para leitura. No caso de escrita no **reg1**, o WDP-IP/HW recebe algum valor para ser carregado em sua memória interna, e quando o valor de **reg1** é buscado pelo processador, retorna valores para depuração. Os Pinos de conexão da interface com o ambiente externo do WDP-IP/HW Tabela 7.8.2 são:

   i) **Clock do Barramento** – Pino do WDP-IP/HW para conectar no *clock* do barramento**;**

   ii) **Reinício** – Quando ativado, reinicia e apaga os valores dos registradores do WDP-IP/HW;

   iii) **Barramento de dados** – Conexão ao barramento de dados, pode ser escalonável, na aplicação utilizada foi configurada para a largura de 16 bits;

   iv) **Habilita leitura** - sinal de 1 bit para habilitar leitura do registradores **reg1** do WDP-IP/HW.

   v) **Habilita escrita** - sinal de 2 bits para habilitar escrita dos registradores **reg0** e **reg1** do WDP-IP/HW;

   vi) **Erro –** Pino de saída que indica erro detectado, pode ser utilizado para interrupção do processador e tratamento, ou simplesmente sinalizar em um LED o erro detectado.

NOME	DIR	# BITS
Clock do Barramento	**E**	**1**
Reinício	**E**	**1**
Erro	**S**	**1**
Barramento de dados	**E/S**	**16**
Habilita leitura	**E**	**1**
Habilita escrita	**E**	**2**

Tabela 7.8.2: Pinos de conexão da interface com o ambiente externo do WDP-IP/HW.

7.8.1 Arquitetura do WDP-IP implementado em *Software* (WDP-IP/SW)

O funcionamento é o mesmo da versão do WDP-IP/HW, só que o módulo de comunicação com o barramento não existe, pois os comandos de comunicação entre aplicação e/ou núcleo do uC/OS-II com o WDP-IP/SW são realizados diretamente pelas funções incrementadas no núcleo do RTOS uC/OS-II descritas na seção 7.6 da página 113, utilizados no WDP-IP/HW somente para leitura e escrita nos registradores, neste esquema, são utilizados para desempenhar os recursos do WDP-IP/SW.

A Ilustração 7.8.1.1 exibe a arquitetura do WDP-IP implementado e *software* (WDP-IP/SW) em forma de diagrama de blocos.

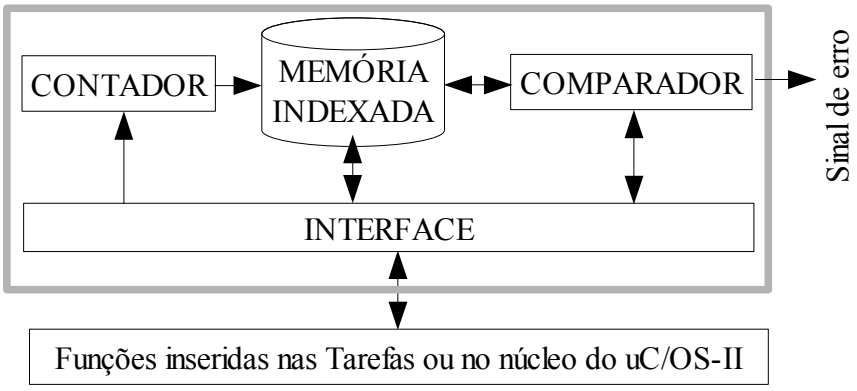


Ilustração 7.8.1.1: Arquitetura em software do WDP-IP/SW.

O principio básico resumido do funcionamento interno de cada módulo do WDP-IP/SW é descrido a seguir:

a) O módulo **MEMÓRIA INDEXADA** é composto por campos denominados índice, ocioso, contador, contagem e paridade (demonstrados na Tabela 7.8.1). Esta memória foi inserida no RTOS como uma estrutura de dados, onde cada linha na tabela de memória é numerada e indexada para cada processo executado pela CPU. Os campos desta memória já foram descritos na seção 7.8 da página 115;
b) O módulo **CONTADOR** faz o incremento no campo **Contagem** na tabela de **MEMÓRIA INDEXADA** da Tarefa atual em execução através das interrupções do *clock tick* do RTOS;
c) O módulo **COMPARADOR** sinaliza com um erro caso a comparação do valor do campo **Contagem** é maior ou menor que o valor do campo **Total** na tabela de **MEMÓRIA INDEXADA** (uma margem de comparação é determinada pelo programador). Um erro também é sinalizado caso algum *bit* no campo **Ocioso** na tabela de **MEMÓRIA INDEXADA** foi configurado;
d) O módulo **INTERFACE** foi incorporado nas funções **ip_cmd, ip_idle, ip_wrt** e **ip_sw** para acessar e controlar os dados na estrutura da **MEMÓRIA INDEXADA** embarcadas no núcleo do RTOS.

7.9 Funcionamento do WDP-IP Durante a Execução da Aplicação

O funcionamento do WDP-IP durante a execução da aplicação em um ambiente preemptivo é exibido na Ilustração 7.9.1. Um parâmetro **Cmax** é definido pela função **ip_idle** para determinar o número máximo de ciclos de *clock* que o processador poderá executar entre sucessivas fatias de tarefas e deverá ter o valor máximo do **CC laço n**. Por exemplo, se existem três tarefas para executar na CPU, o controle de tempo compartilhado deverá ser CCmax igual a “CC11 + CC21 + CC31” (onde CCnf representa o número máximo de ciclos de *clock* executados pelo processador para a tarefa “n” de fatia “f”).

O **Cmaxn,** que é o tempo de execução máximo para cada Tarefa, é determinado pela soma total das fatias “f” de cada tarefa “n” e armazenado na memória indexada do WDP-IP. É determinado pelo usuário ou calculado pelo WDP-IP em uma fase de auto-aprendizado.

Esta combinação entre o **Cmax** e o **Cmaxn** aliado ao fato do WDP-IP monitorar um valor mínimo e máximo para estes tempo, permite diminuir a latência de detecção de erros.

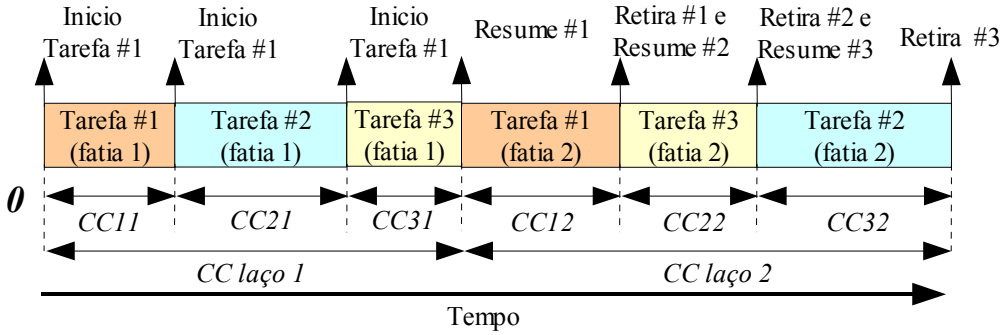


Ilustração 7.9.1: Funcionamento do WDP-IP durante a execução da aplicação com RTOS em ambiente preemptivo.

7.10 Técnica WDP-IP+ (uma melhoria do WDP-IP)

Em vista de controlador o fluxo de controle do programa, o WDP-IP foi modificado para monitorar a parte da aplicação que está em execução, através da inserção de uma instrução específica dentro do código fonte. Essa instrução alimenta o WDP-IP+ com a informação para cada vez que o fluxo de controle entra em um bloco básico do programa, permitindo a verificação da seqüência de execução do programa. Caso o fluxo de execução do programa seja diferente do esperado, um sinal é ativado a fim de indicar a ocorrência de um erro de fluxo de controle. Para realizar o monitoramento do fluxo de execução do programa, utilizou-se a técnica de YACCA (ver seção 5.6 na página 73) com uma assinatura gerada em tempo de execução “Bi” para cada bloco “vi” em execução. A função criada e embarcada no RTOS esta descrita a seguir:

- Função **ip_yacca(Bi , Bj , Bk):** Esta função informa ao WDP-IP+ a assinatura gerada em tempo de execução “Bi”, a assinatura “Bj” e “Bk” associada com o bloco “vj” e “vk” respectivamente, que são predecessor de “Bi”. Caso não exista o predecessor “Bk”, deve-se satisfazer as igualdades “Bk” = “Bj” ou “Bk” = “Bi”.

O WDP-IP+ ao receber o valor da função **ip_yacca** (contendo os campos **Bi**, **Bj** e **Bk)** enviado pelo processador e embarcada no código da aplicação, ele calcula imediatamente durante a execução do programa o algoritmo **code** (ver a fórmula (5.6.5) na página 75), armazena seu valor em um campo da **MEMÓRIA INDEXADA** e imediatamente verifica se está correto o controle de fluxo do programa comparando o valor **code** com as assinaturas **Bj** e **Bk** (ver as fórmulas (5.6.3) e (5.6.4) na página 75). Em outras palavras, o WDP-IP+ ao receber os campos da função **ip_yacca**, realiza ao mesmo tempo a operação de atualização e teste descrita na seção 5.6.

Foi inserido mais um campo na **MEMÓRIA INDEXADA** na Tabela 7.8.1 para armazenar o último valor do algoritmo **code** calculado da Tarefa atual em execução (ver Tabela 7.10.1).

Índice	Ocioso	Contagem	Total	Paridade	code
0	B"0"	X"0000"	X"0000"	B"0"	X"0"
⋮	⋮	⋮	⋮	⋮	⋮
N	B"0"	X"0000"	X"0000"	B"0"	X"0"

Novo

Tabela 7.10.1: Memória indexada do WDP-IP+.

A arquitetura do WDP-IP já descrita em seções anteriores (ver seção a 7.8 na página 115 para a arquitetura do WDP-IP em *hardware* e a seção 7.8.1 na página 119 para a arquitetura do WDP-IP em *Software*) foram modificadas de modo que o WDP-IP+ possa calcular o algoritmo **code** no módulo **CALCULA CODE** e verificar o controle de fluxo de programa no módulo **COMPARADOR** como mostra a Ilustração 7.10.1.

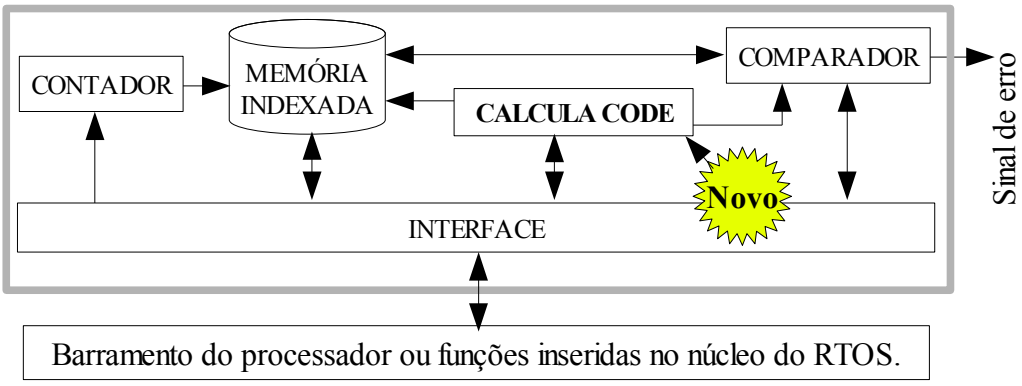


Ilustração 7.10.1: Arquitetura geral do WDP-IP+, versão melhorada do WDP-IP.

Para o WDP-IP+ implementado em *hardware,* nesse novo esquema, um novo registrador denominado **reg2** foi adicionado para receber os valores enviados pela função **ip_yacca(Bi,Bj,Bk)** (demonstrada na Ilustração 7.10.2).

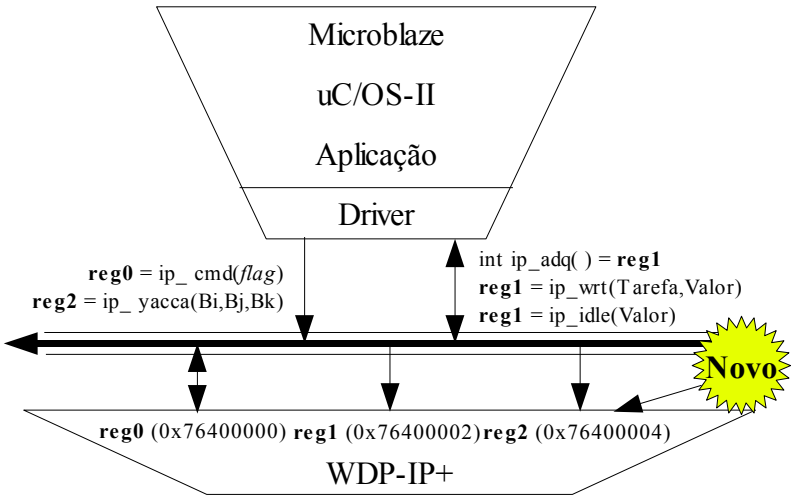


Ilustração 7.10.2: Comunicação com o WDP-IP+ em *Hardware*.

7.11 Aplicação Utilizada para os Testes

Em todos os testes realizados, a CPU executou três tarefas sob o controle do RTOS uCOS-II em ambiente multitarefa (conforme a Ilustração 7.11.1). As tarefas executadas seguiram sempre a mesma estrutura de comunicação e sincronismo, conforme segue:

a) Tarefa 1: Ao iniciar, ela aloca e cria as Tarefas 2 e 3 e se mantém em um laço infinito capturando os estados do WDP-IP e exibindo na saída padrão de vídeo configurado para o SoC. Esses estados obtidos são de contagem das Tarefas, valor máximo do contador de cada Tarefa, e se algum método detectou alguma falha ativada em alguma Tarefa. Para isso foram inseridas no código as funções **ip_cmd** para requerer o estado solicitado do WDP-IP e **ip_adq** para receber do WDP-IP o estado solicitado. A Tarefa 1 é do tipo multitarefa, preemptiva, possui prioridade mais elevada que as outras e executada em intervalos de um segundo, respeitando a taxa monotônica estudada na seção 3.6.1 na página 47;

b) Tarefa 2: Essa tarefa se encontra em laço infinito, porém ela inicia a execução do código somente se receber um semáforo da Tarefa 3. No final da execução do código, antes de voltar ao início devido ao laço infinito, ela sinaliza através de semáforo a Tarefa 3 que terminou de executar seu código e volta para o início esperando receber a confirmação da Tarefa 3 para continuar de onde parou.

c) Tarefa 3: Essa tarefa se encontra em laço infinito, porém ela inicia a execução do código somente se receber um semáforo da Tarefa 2. No final da execução do código, antes de voltar ao início devido ao laço infinito, ela sinaliza através de semáforo a Tarefa 2 que terminou de executar seu código e volta para o início esperando receber a confirmação da Tarefa 2 para continuar de onde parou.

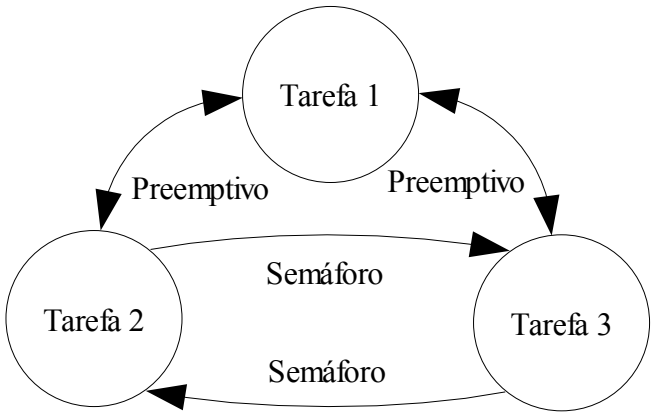


Ilustração 7.11.1: Esquema de escalonamento entre as tarefas.

Resumindo, ao iniciar o ambiente de multitarefa, o RTOS cria a Tarefa 1 que executa a cada um segundo e captura através do WDP-IP diversos estados como o valor de contagem atual de cada Tarefa, o valor máximo de contagem de cada Tarefa e a Tarefa que apresentou erro. As Tarefas 2 e 3 estão sincronizadas por semáforo e executam um algoritmo *benchmark* que será escolhido para os teste de validação da técnica WDP-IP nas próximas seções.

7.12 Custos da Implementação do WDP-IP (*Overheads*)

A Tabela 7.12.1 resume o custo de área exigida para mapear e rotear o WDP-IP/HW na FPGA em relação à área ocupada pelo processador *MicroBlaze*. Também é mostrado nesta tabela o custo de código extra para armazenar na memória a implementação do WDP-IP/SW (puramente em *software*) no núcleo do RTOS, e o custo da modificação no núcleo do RTOS para comunicação do processador *MicroBlaze* com o WDP-IP/HW. Finalmente, a Tabela 7.12.1 também demonstra a degradação no desempenho do processador ao implementar as estruturas de detecção de erros embarcadas no núcleo do RTOS do WDP-IP/SW em relação ao código original sem as tais implementações. Pode-se observar que devido ao número muito pequeno de instruções usadas pelo WDP-IP/HW, a degradação no desempenho é praticamente desprezível. Os custos da Área (Blocos de Lógica Configurável) podem ser reduzidos através da diminuição do tamanho da memória CAM, devido ao fato desta ser sintetizada na FPGA na forma de *flip-flops*.

Custos				
Área (Blocos de Lógica Configurável)	Código extra na Memória (Bytes)		Degradação no desempenho (ms)	
	WDP-IP/SW	WDP-IP/HW	WDP-IP/SW	WDP-IP/HW
12.47% [2]	6.30%	0.77%	1.45%	1 a 3 instruções de comunicação. [1]

[1] Cada instrução representa um operação de leitura ou escrita. A operação da escrita no OPB requer 3 ciclos de pulso de clock, enquanto quando a operação de leitura requerer 4 [18];
[2] Pode ser reduzido até 5,2% (diminuindo o tamanho da memória CAM).

Tabela 7.12.1: Custos extras para a implementação do WDP-IP.

7.13 Teste EMI irradiado

Na realização dos testes de interferência EMI, utilizou-se uma célula GTEM [83] (vide Ilustração 7.13.1) na injeção de falhas, no primeiro momento nas partes internas da FPGA e no segundo momento em toda placa de circuito impresso. Os equipamentos e procedimentos utilizados na execução dos testes são descritos a seguir.

Ilustração 7.13.1: Célula GTEM ETS LINDGREN 5402 [83].

7.13.1 Célula GTEM (*Gigahertz Transverse Electromagnetic*)

A Célula GTEM utilizada para os testes nesta dissertação foi a GTEM ETS LINDGREN 5402 Ilustração 7.13.1. Essa Célula é um instrumento de teste de precisão para validar compatibilidade eletromagnética (*Electromagnetic Compatibility* – EMC), onde, um equipamento não causa interferência em outros equipamentos, é imune às emissões de outros equipamentos e não causa interferência em si próprio.

A utilização planejada para a Célula GTEM (ver Tabela 7.13.1.1) são nos testes de imunidade irradiada (*Radiated Immunity* – RI) e emissão irradiada (*radiated emissions* – RE):

- Imunidade irradiada (RI): determina a frequência e energia do sinal RF para gerar a energia radiada no GTEM que podem resultar em um defeito ou operação anormal do

dispositivo sob teste (*Device Under Test* - DUT) [83];

- Emissão irradiada (RE): medida no ponto de alimentação da célula GTEM, produzidos pela energia radiada do DUT na faixa de frequência equivalente para a antena monopólo para cada medida do eixo ortogonal (X,Y e Z). As medidas são então convertidas por meio de um fator de antena do GTEM em um valor de intensidade de campo [83].

Faixa de freqüência	Impedância de entrada	Máxima potência de entrada	Típico VSWR
Testes de RE (9kHz-5GHz)	50 Ohms	50 Watts	$\leq 1.75:1$
Testes de RI (DC-20GHz)[1]			$\leq 1.3:1$
[1] Baixa entrada de VSWR para f < 20 gigahertz permitidas [83].			

Tabela 7.13.1.1: Especificação elétrica da Célula GTEM ETS LINDGREN 5402.

A medida do fator de onda estacionária (*Voltage Standing Wave Radio* – VSWR) é muito importante, pois aponta mudanças nos parâmetros da GTEM. Para que toda potência transmitida chegue a antena, a impedância de cabos e conectores deve ser a mesma (casamento de impedância), do contrário teremos parte do sinal transmitido sendo refletido na linha no ponto onde não há esse casamento.

A geometria do GTEM, como mostra a Ilustração 7.13.1.1, o DUT deve ficar no centro do volume da Célula. Devido a carga de 50 Ohms, o equipamento RF deve ter um casamento de mesma impedância para evitar o fator de onda estacionária.

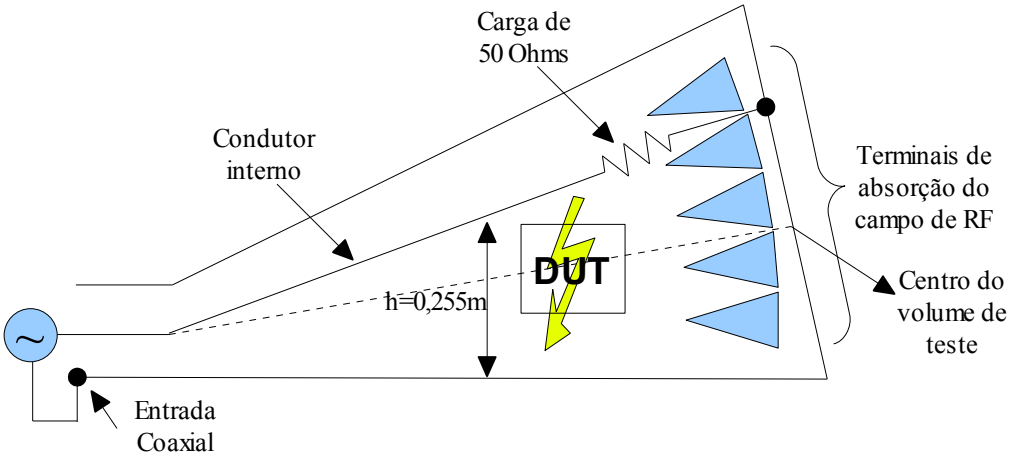


Ilustração 7.13.1.1: Geometria da Célula GTEM [83].

A Célula GTEM ETS LINDGREN 5402 tem total conformidade com as normas EN 61000-4-3 e ANSI C63.4 (ver apêndice “B” na página 159).

7.13.1 Plataforma de Testes

A norma IEC 62132-2 [11] descreve uma técnica para quantificar a imunidade do dispositivo sob teste (*Device Under Test* - DUT) em um campo de radiação eletromagnética utilizando uma Célula GTEM. A placa do circuito deve ser projetada de tal forma que apenas o circuito integrado sob teste fique dentro da Célula, enquanto as trilhas e conexões ficam do lado de fora da Célula, do outro lado da placa. A Ilustração 7.13.1.1 mostra o lado frontal e a Ilustração 7.13.1.2 o verso da placa desenvolvida no grupo SiSC (o esquemático desta placa se encontra no apêndice “A” na página 157). O lado frontal da placa contém o FPGA XC3S200-VQ100 da Xilinx [18] (mapeado com o microprocessador Microblaze) e no verso 1 Mega Byte de memória (duas memórias SRAM de 256Kbx32) e um *clock* de 49.152 MHz.

A interferêncial eletromagnética irradiada ocorre através do ganho de antena [84] nas trilhas da placa e dos pinos da FPGA decorrentes de sua área efetiva (relacionada ao tamanho físico da trilha e seu formato). Pode-se calcular seu ganho através da seguinte relação:

$$G_{ANTENA} = \left(\frac{4 \Pi A}{\lambda^2}\right) = \left(\frac{4 \Pi f^2 A}{c^2}\right) \rightarrow (dB - decibels) \qquad \textbf{(7.13.1.1)}$$

Onde:

- G = ganho da antena (dB);
- A = área efetiva (m2);
- f = freqüência da portadora;
- c = velocidade da luz (m/s2);
- λ = comprimento de onda da portadora (m).

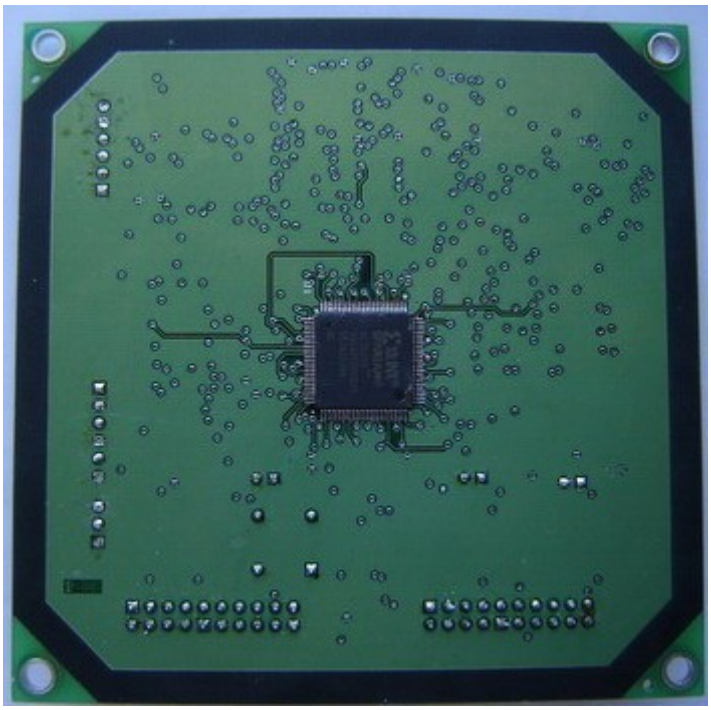


Ilustração 7.13.1.1: Vista de frente da placa desenvolvida.

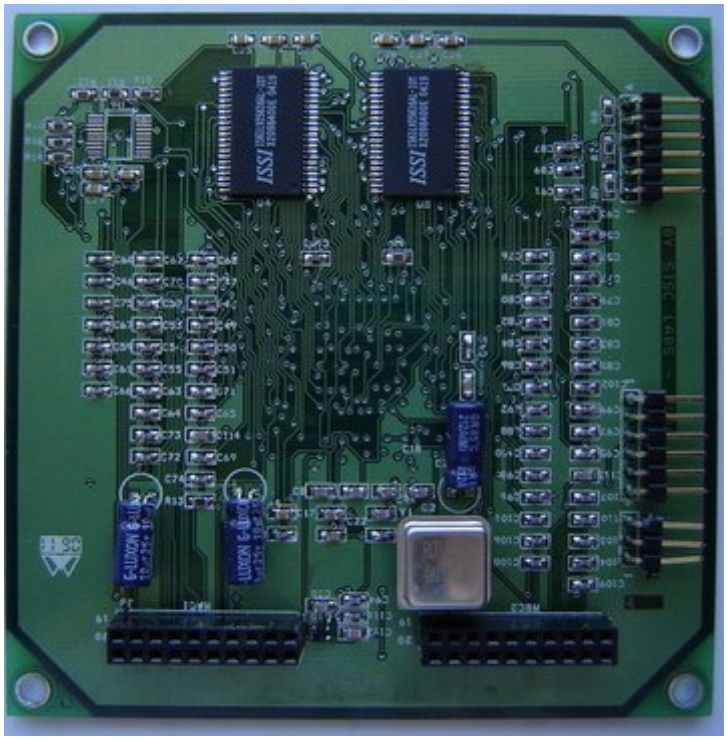


Ilustração 7.13.1.2: Vista do verso da placa desenvolvida.

Na primeira parte da norma IEC 62132 [11] (ver apêndice “A” na página 159) diversas classes são descritas (de “A” a “D”) para classificar o mau funcionamento do DUT durante a exposição a EMI irradiado. Note que nos testes experimentais desta dissertação, para efeito da cobertura de falhas da técnica proposta, as classes avaliadas foram “B”(erros visuais que apenas alteram as informações coletadas) e “C”(erros visuais que alteram as informações coletadas e causaram uma parada forçada no sistema). Já a classe “A”, por não apresentar erros no sistema (erros no *bitstream* na FPGA, na verificação de CRC na área de memória, nos dados computados pela aplicação e nas informações transmitidas para o PC) , não foi computada.

Os equipamentos e procedimentos utilizados para injetar falhas conforme a norma 62132-2 [11] são resumidos na Ilustração 7.13.1.3. O gerador de sinal RF é utilizado para selecionar a freqüência do sinal portador, a freqüência e amplitude do sinal modulador. No próximo passo, o sinal portador já alterado pela modulante é amplificado (Amplificador de RF), o sinal passa por um acoplador que lhe dá um ganho de 40dBm e que está associado ao medidor de potência e uma carga, o sinal então atinge a Célula GTEM. O DUT sob teste está conectado diretamente ao PC, permitindo assim a configuração da FPGA e o monitoramento das aplicações.

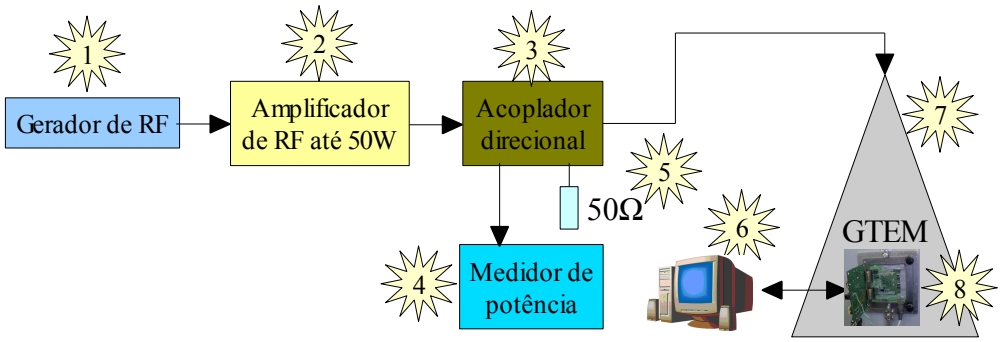


Ilustração 7.13.1.3: Plataforma de testes para EMI irradiado pela norma 62132-2 [11].

Como mostra a Ilustração 7.13.1.3 os equipamentos utilizados são descritos como segue:

1. Gerador de freqüência FLUKE 6061A;
2. Amplificador de RF ar 50W1000B;
3. Acoplador direcional AR DC3002;
4. Power Meter Agilent E4419B;
5. Carga de 50 Ohms;

6. Computador conectando o cabo JTAG pela paralela para fazer a configuração da FPGA e o cabo serial para receber informações enviadas pelo microprocessador;
7. Célula GTEM ETS LINDGREN 5402;
8. Dispositivo sob teste (*Device Under Test* – DUT).

O campo eletromagnético desejado, no qual o DUT é exposto dentro da célula GTEM, é gerado combinando as seguintes variáveis: a freqüência do sinal portador, a freqüência, a amplitude e a potência do sinal de modulação.

O valor do campo elétrico dentro da caixa GTEM ETS 5402 para um campo irradiado pode ser calculado pela fórmula (7.13.1.2) sabendo-se a potência necessária aplicada na caixa GTEM para obter um campo elétrico desejado e constante em uma faixa de freqüência e as dimensões da caixa e impedância.

$$P_{GTEM} = 10 * \log_{10}\left(\frac{(E.h)^2 * 1000}{Z_o}\right) \rightarrow (dBm\ decibels\ relativo\ a\ um\ miliwatt) \qquad \textbf{(7.13.1.2)}$$

Onde "E" é o campo eletromagnético em Volts por metro (V/m), "h" a distância dentro da Célula GTEM e "Zo" sua impedância. As seguintes variáveis tem valores fixos: $h = 0{,}255\,m$ e $Z_o = 50\,\Omega$.

O ambiente de testes descrito acima é utilizado para certificar a compatibilidade eletromagnética (EMC) de dispositivos eletrônicos. Esses testes são realizados seguindo normas internacionais, como por exemplo, as ditadas pela IEC. Tipicamente, essas normas visam padronizar procedimentos para verificar o comportamento do DUT sob um campo eletromagnético entre 1 e 10V/m (exceto para sistemas embarcados na indústria automotiva). No caso desta dissertação, o ambiente descrito foi utilizado como um poderoso processo para injeção de falhas no DUT. Para realização desse procedimento, o campo eletromagnético foi aumentado para uma escala de 30 a 200 V/m, a fim de levar o DUT à falhar. A Ilustração 7.13.1.4 mostra o ambiente completo, dos testes realizados no INTI.

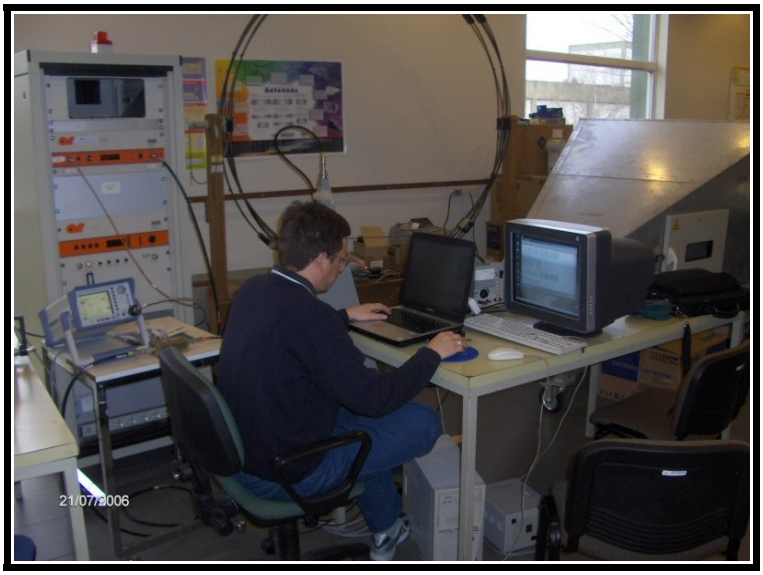


Ilustração 7.13.1.4: Ambiente de testes no INTI.

Parte III - RESULTADOS E CONCLUSÕES

8 RESULTADOS

8.1 Introdução

O método proposto nesta dissertação foi testado e avaliado em relação à capacidade de detecção de erros na injeção de falhas, tanto injetada em áreas de memória quanto irradiada em laboratório. A implementação do WDP-IP atua na monitoração da execução da aplicação do usuário que são alguns programas *benchmarks* em conjunto com o RTOS uC/OS-II modificados.

Os experimentos de injeção de falhas foram executados a fim de verificar a capacidade do método proposto de detectar falhas transientes.

As falhas consideradas foram de três tipos:

a) Aquelas que causam a inversão de um (ou mais) *bit* entre os 32 que compõem uma palavra de instrução do processador;
b) Alteração em uma palavra de instrução que provoca o desvio incondicional para um endereço aleatório da memória de programa do processador;
c) Falhas que alteram a configuração (bitstream) da FPGA.

Quanto à quantidade de testes para avaliar a efetividade do método, foi considerado que quanto mais casos de testes são gerados e quanto maior o número de detecção de erros pelo método proposto, mais ele se torna efetivo [85]. Baseado nesta lógica, a medida de efetividade do teste (*Test Case Effectiveness* - TCE) é definida como a relação de erros detectados pelos casos de teste (Ntc) para um número total de erros (Ntot) ocorridos na função durante o ciclo de teste:

$$TCE = \frac{Ntc * 100}{Ntot}$$ [85].

No contexto deste trabalho, duas perguntas surgem. A primeira: É possível ter uma cobertura de falhas razoável baseada na monitoração do tempo de execução da aplicação (isto é, contando os ciclos de *clock* exigidos pela CPU para executar uma dada tarefa)? E a segunda: O quanto efetivo é a técnica baseada no WDP-IP (isto é, em *hardware*) em comparação com outras técnicas de detecção de erros embarcadas em *software*.

No trabalho proposto, considera-se falhas transientes e falhas permanentes. Neste último caso, assume-se que falhas permanentes são provocadas por mudanças no *bitstream* de configuração do *hardware* da FPGA que contém o sistema embarcado (microprocessador e alguns módulos). Em ambos os casos, assume-se que falhas transientes (na lógica do usuário) ou falhas permanentes (no *hardware* de configuração da FPGA) são causadas por interferência eletromagnética (EMI) irradiada, diretamente no *chip* da FPGA ou acoplada em componentes de placa (filtros, cabos, conectores, etc) e conduzidas na forma de ruído até os pinos de entrada da FPGA.

8.2 Teste de Injeção de Falhas via *Software* (Memória da Placa)

Uma placa de desenvolvimento Xilinx MicroBlaze Spartan3 [18] foi utilizada para executar a aplicação que consistia de três tarefas de usuário sob o escalonamento do RTOS uC/OS-II (ver Ilustração 7.11.1): A Tarefa 1 dedicada para ler e verificar periodicamente o estado interno do WDP; a Tarefa 2 um gerador de números primos aleatório e a Tarefa 3 um algoritmo *bubblesort* de classificação de matrizes. Dois procedimentos de injeção de falhas no sistema foram utilizados:

a) As falhas eram injetadas aleatoriamente substituindo uma instrução de programa por um desvio (*jump*) para dentro da própria aplicação;

b) Um *bit-flip* injetado aleatoriamente no fluxo de instrução da aplicação.

O WDP-IP implementado tanto em *software* quanto em *hardware* foram testados conforme segue nas próximas seções.

8.2.1 Teste do WDP-IP/HW

Antes de cada teste efetuado, o sistema foi reiniciado para certificar-se de que nenhuma falha estivesse residente na memória. Foram gerados 100 testes de injeções de falhas, inserindo em uma posição aleatória de memória uma instrução de desvio para uma posição aleatória de memória dentro da área onde a aplicação se encontrava armazenada.

O resultado dos testes é exibido na Ilustração 8.2.1.1 onde 69% dos erros foram detectados, enquanto 31% passaram despercebidos pela técnica.

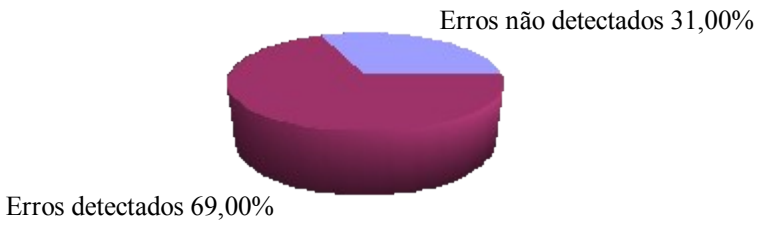


Ilustração 8.2.1.1: Detecção de desvio aleatório do WDP-IP implementado em *hardware*.

Da mesma forma que o teste anterior, antes de cada teste efetuado, o sistema foi restaurado para certificar-se de que nenhuma falha estivesse residente na memória. Foram gerados 200 testes de injeções de falhas inserindo um único *bit-flip* em uma posição aleatória da memória dentro da área onde a aplicação se encontrava armazenada.

O resultado dos testes é exibido na Ilustração 8.2.1.2 onde 59% dos erros foram detectados, enquanto 41% passaram despercebidos pela técnica.

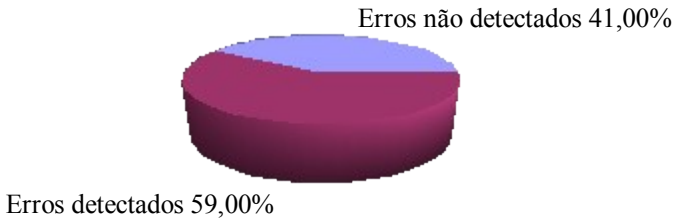


Ilustração 8.2.1.2: Detecção de *bit-flip* aleatório do WDP-IP implementado em *hardware*.

8.2.2 Teste do WDP-IP (WDP-IP/HW e WDP-IP/SW)

Neste teste foram utilizados em conjunto para a detecção de erros a técnica do WDP-IP implementada tanto em *software* (WDP-IP/SW) quanto em *hardware* (WDP-IP/HW). Da mesma forma que os testes anteriores, antes de cada teste efetuado, o sistema foi reiniciado para certificar-se de que nenhuma falha estivesse residente na memória. Foram gerados 100 testes de injeções de falhas inserindo em uma posição aleatória de memória uma instrução de

desvio para uma posição aleatória de memória dentro da área onde a aplicação se encontrava armazenada. O resultado dos testes é exibido na Ilustração 8.2.2.1 onde 88% dos erros foram detectados, enquanto 12% passaram despercebidos pelas técnicas.

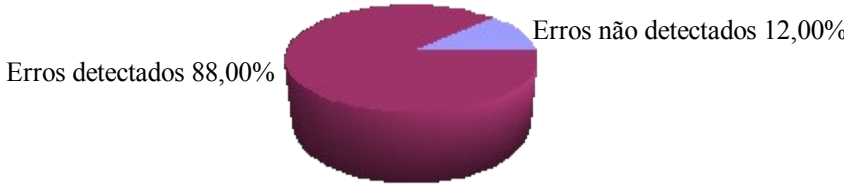


Ilustração 8.2.2.1: Detecção de desvio aleatório do WDP-IP implementado em *hardware* e *software*.

As duas técnicas de implementação do WDP-IP (em *hardware* e *software*) detectaram 88% de erros nesse teste (de acordo com a Ilustração 8.2.2.1), porém 80% foram detectados pela técnica do WDP-IP/HW e 20% detectados pela técnica do WDP-IP/SW, como mostra a Ilustração 8.2.2.2, ou seja a técnica do WDP-IP/SW contribuiu somente com 8% de efetividade no processo de detecção para a técnica do WDP-IP/HW.

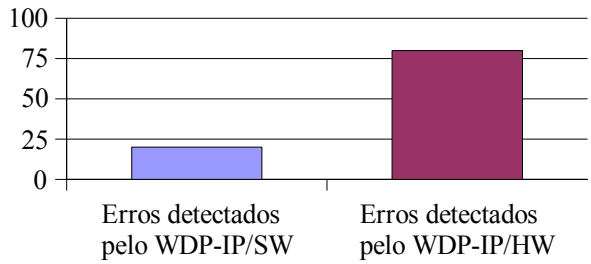


Ilustração 8.2.2.2: Percentual de detecção de desvio aleatório do WDP-IP implementado em *hardware* e *software*.

Da mesma forma que o teste anterior, antes de cada teste efetuado, o sistema foi reinicializado para certificar-se de que nenhuma falha estivesse residente na memória. Foram gerados 200 testes de injeções de falhas inserindo um único *bit-flip* em uma posição aleatória da memória dentro da área onde a aplicação se encontrava armazenada.

O resultado dos testes é exibido na Ilustração 8.2.2.3 onde 59% dos erros foram detectados, enquanto 41% passaram despercebidos pelas técnicas.

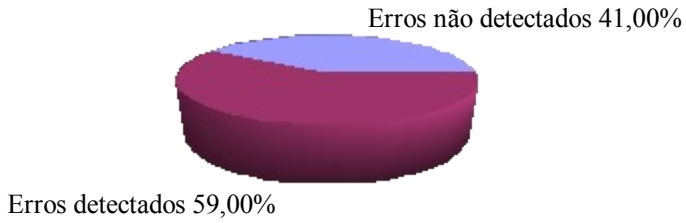


Ilustração 8.2.2.3: Detecção de *bit-flip* aleatório do WDP-IP implementado em *hardware* e *software*.

As duas técnicas de implementação do WDPI-IP (em *hardware* e *software*) detectaram 59% de erros nesse teste (conforme a Ilustração 8.2.2.3), porém 50% foram detectados pela técnica do WDP-IP/HW e 32% detectados pela técnica do WDP-IP/SW, como mostra a Ilustração 8.2.2.4, ou seja a técnica do WDP-IP/SW contribuiu somente com 9% de efetividade no processo de detecção para a técnica do WDP-IP/HW.

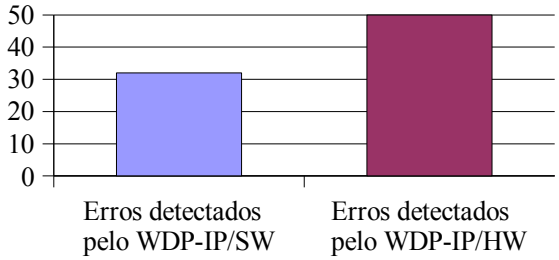


Ilustração 8.2.2.4: Percentual de detecção de *bit-flip* aleatório do WDP-IP implementado em *hardware* e *software*.

8.3 Teste de Injeção de Falhas via *Hardware* (Teste EMI Irradiado)

Os testes se dividiram basicamente em duas etapas, na primeira etapa (Ilustração 8.3.1) o DUT foi colocado seguindo os padrões convencionados para a Célula GTEM, onde somente a FPGA estaria exposta as irradiações. Nos testes realizados nessa etapa não foi detectado nenhuma falha ativada no sistema, para freqüências que variaram de 1MHz a 3GHz e campo elétrico que variou de 10V/m a 210V/m.

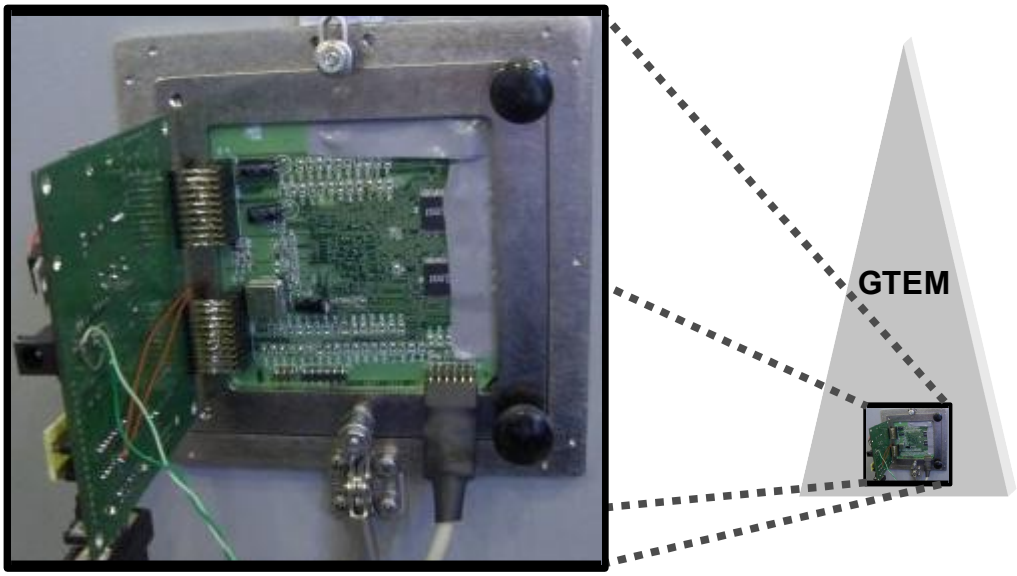


Ilustração 8.3.1: DUT seguindo padrões de teste.

Na segunda etapa, como podemos ver na Ilustração 8.3.2, o DUT foi inserido dentro da Célula GTEM e os cabos foram isolados e colocados tubos de ferrite para não haver interferência na comunicação com o computador.

Ilustração 8.3.2: DUT dentro da Célula.

A injeção de falhas pode ser realizada com sucesso para os testes realizados na segunda etapa. A seqüência de testes utilizada para validar as técnicas de detecção é exibido no diagrama de blocos da Ilustração 8.3.3.

O gerador de freqüência então é ligado e o amplificador é acionado para inicialmente ampliar o sinal modulado ao mínimo (Ilustração 7.13.1.3). Uma frequência portadora é selecionada e acionada a modulação, o valor de amplificação é aumentado lentamente até que o sistema apresente erros na saída ou o valor de amplificação atinja o patamar de 45Watts(46.5 dBm) produzindo um campo de 210V/m dentro da Célula GTEM.

Primeiramente é feita a seleção da aplicação a ser executada para ser compilada. O *Bitstream* é carregado na FPGA contendo a configuração do *Hardware* contendo o SoCs com o processador *Microblaze* e a técnica embarcada em *Hardware* do WDP-IP/HW. Posteriormente o processador *Microblaze* carrega na área de memória externa (1MByte) a aplicação que consiste no RTOS contendo um programa para teste com a técnica YACCA, o WDP-IP/SW e o WDP-IP+ em *Software*.

O estado da saída da aplicação é enviada pela serial ao computador e exibida na tela através do programa *Hyperterminal*, a exemplo no sistema operacional *Windows*, e monitorada visualmente. Caso apresente erros no programa ou caso a aplicação não tenha executado até o final é marcado na tabela de testes. Caso alguma técnica tenha detectado algum erro, também é marcado na tabela de testes, a técnica utilizando o WDP-IP/HW sinaliza em um LED caso tenha detectado falhas ativadas.

O *readback* da FPGA é realizado para verificar se houve modificações na configuração do *harware* armazenado nas células da FPGA. Caso apresente diferença na comparação entre o *Bitstream* armazenado na FPGA e o *Bitstream* original, é marcado na tabela de testes. O *Bitstream* é carregado novamente permitindo a execução do próximo passo.

O processador *Microblaze* carrega em uma área livre de memória uma aplicação que calcula o CRC-CCITT, CRC-16 e o CRC-32 [86,87] da área de memória utilizada para armazenar a aplicação contendo o RTOS uC/OS-II, as tarefas de usuário, e as técnicas embarcadas: WDP-IP/SW e WDP-IP+ em *Software*. Este programa então é executado e anotado na tabela de testes caso alguma diferença entre o CRC calculado e o esperado (calculado antes da injeção de falha) tenha ocorrido.

A inicialização dos testes é novamente executada, outras freqüências e campos elétricos são utilizados.

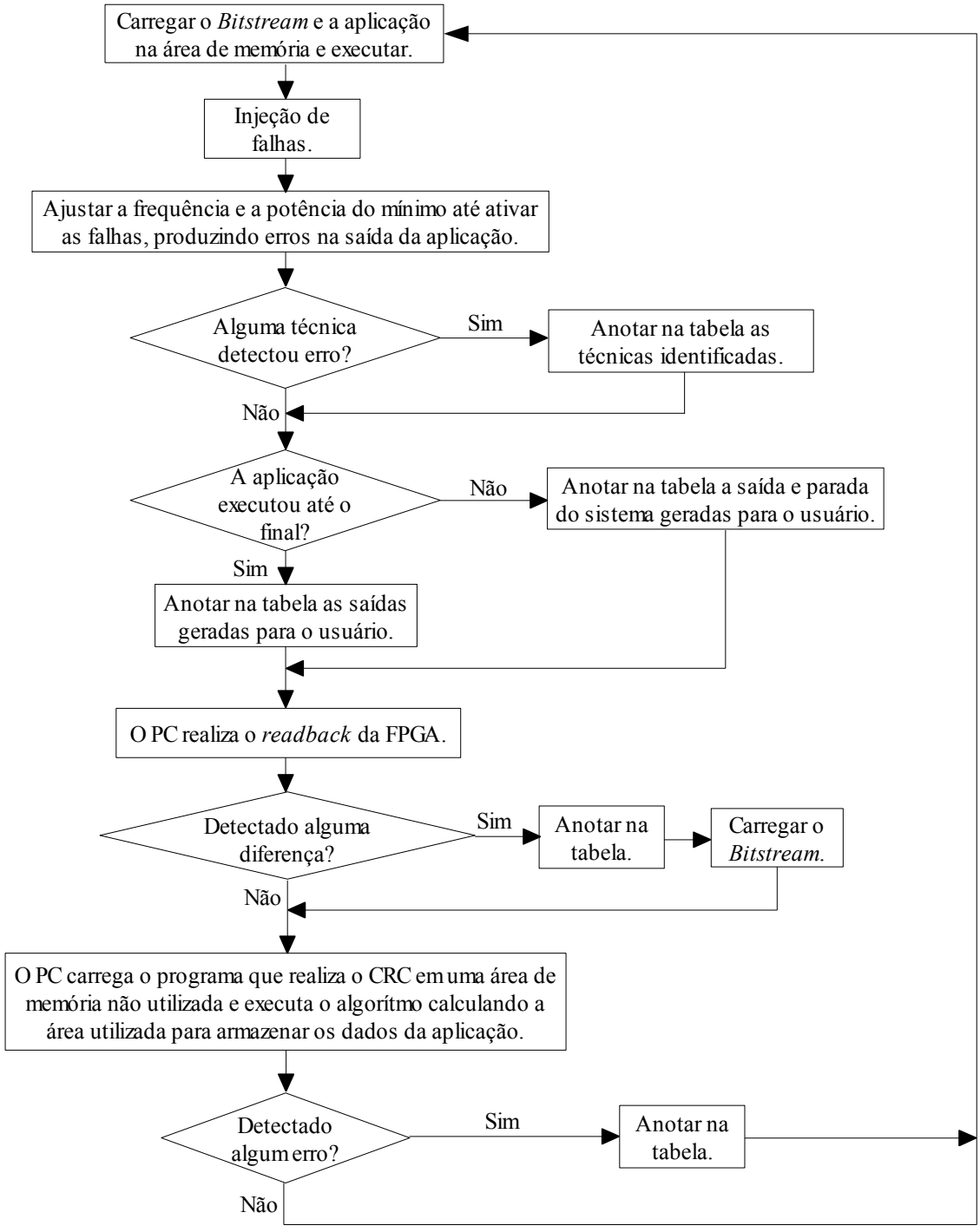


Ilustração 8.3.3: Diagrama de blocos do procedimento de teste realizado para validar as técnicas de detecção de erros propostas.

O DUT foi exposto a diversos campos eletromagnéticos entre 30V/m e 210V/m na faixa de freqüência modulada de 1MHz a 1GHz, conforme descrito anteriormente. As aplicações utilizadas foram *bubblesort* (ordenador de matrizes), gerador de números primos e um filtro IIR.

8.3.1 Classificação dos Tipos de Falhas Gerados

Quanto ao tipos de erros gerados podemos considerar:

a) Erro na configuração (*bitstream*) do FPGA;
b) Erro na área de memória utilizada para armazenar o código;
c) Outras falhas.

Conforme será visto nas Ilustrações (8.3.2.2, 8.3.3.2 e 8.3.4.2), neste contexto, podemos considerar:

a) Que falhas que afetaram o *bitstream* de configuração do FPGA são do tipo "permanentes";
b) Que falhas que afetaram a área de memória do programa são do tipo "transientes";
c) Que falhas intituladas "outras falhas" são aquelas que provocam a inversão de um ou mais *bits* (ex.: PC, ponteiro de pilha, instruções de registradores), ou na interferência na comunicação entre o FPGA e o PC (*test host*), entre outras.

O estado da saída da aplicação é enviada para a porta serial e exibida na tela do computador. Essas saídas foram marcadas na tabela quando notou-se erros na execução do programa e classificou-se entre erros visuais que apenas alteram as informações coletadas ou que além de alterar essas informações, causaram uma parada forçada no sistema conforme será mostrado nas Ilustrações (8.3.2.3, 8.3.3.3 e 8.3.4.3).

8.3.2 Teste Utilizando Ordenador de Matrizes

A Ilustração 8.3.2.1 apresenta um resumo da capacidade de detecção de erros das técnicas utilizadas durante o funcionamento do sistema no interior da célula GTEM, para teste de imunidade irradiada. Nota-se uma pequena taxa de detecção de erro na utilização de semáforos pelo RTOS uC/OS-II e uma significante diferença entre o WDP-IP/HW frente as demais técnicas. O WDP-IP+ implementado em *software* (o qual combina as técnicas

YACCA e WDP-IP/SW) apresentou pouco acréscimo na taxa de detecção frente à versão do WDP-IP/SW. Já a técnica de detecção YACCA apresentou menor taxa de detecção em relação às demais técnicas, salvo a detecção por violação dos semáforos, nativos do próprio RTOS uC/OS-II.

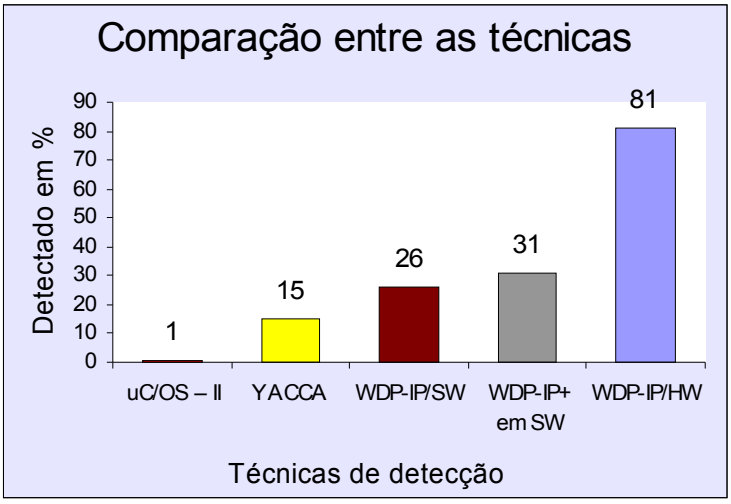


Ilustração 8.3.2.1: Comparação entre as técnicas de detecção utilizando o algoritmo ordenador de matrizes.

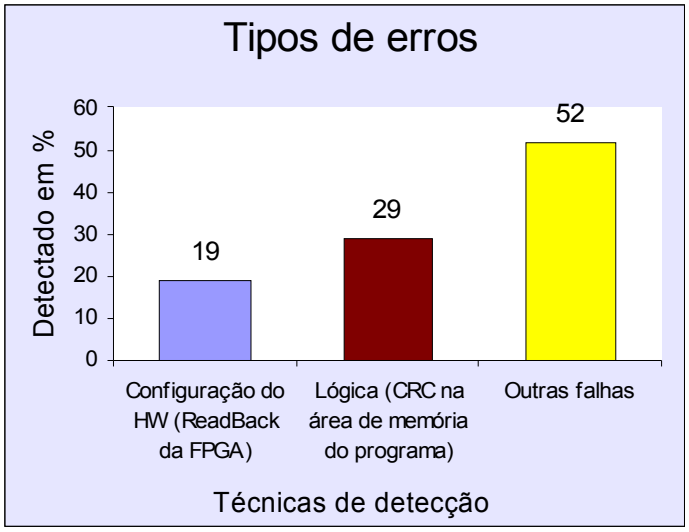


Ilustração 8.3.2.2: Tipos de erros detectados utilizando o algoritmo ordenador de matrizes.

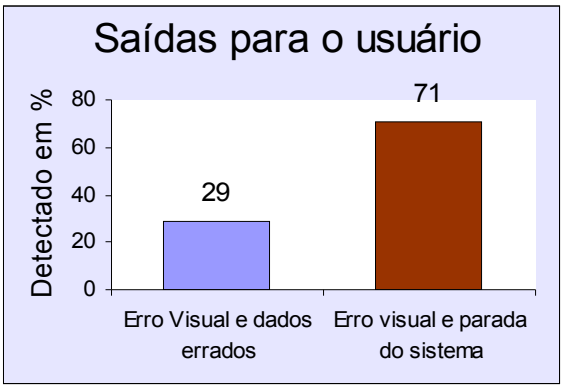


Ilustração 8.3.2.3: Saídas do programa em execução utilizando o algoritmo ordenador de matrizes.

8.3.3 Teste Utilizando Gerador de Números Primos

A Ilustração 8.3.3.1 apresenta um resumo da capacidade de detecção de erros das técnicas utilizadas durante o funcionamento do sistema no interior da célula GTEM, para teste de imunidade irradiada. Nota-se uma pequena taxa de detecção de erro na utilização de semáforos pelo RTOS uC/OS-II e uma significante diferença entre o WDP-IP/HW frente as demais técnicas. O WDP-IP+ implementado em *software* (o qual combina as técnicas YACCA e WDP-IP/SW) apresentou pouco acréscimo na taxa de detecção frente a versão do WDP-IP/SW. Já a técnica de detecção YACCA ficou aquém na taxa de detecção em relação as demais técnicas, salvo a detecção por violação dos semáforos, nativos do próprio RTOS uC/OS-II.

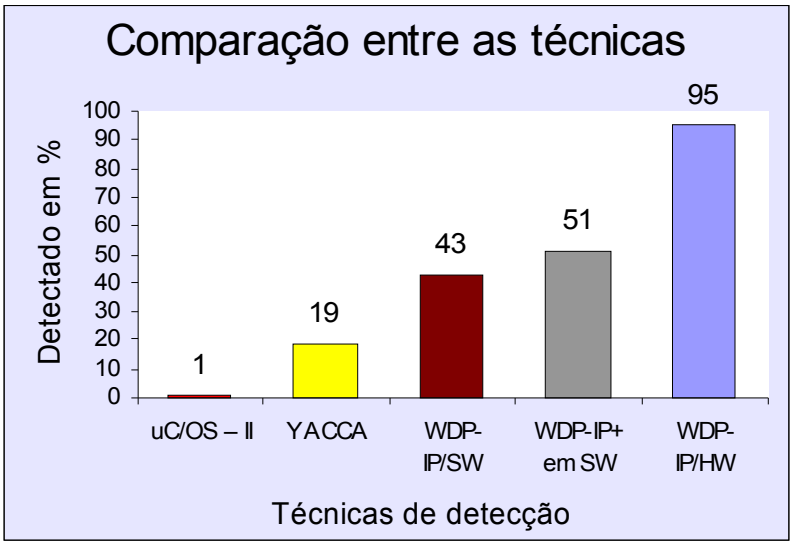


Ilustração 8.3.3.1: Comparação entre as técnicas de detecção utilizando o algoritmo gerador de números primos.

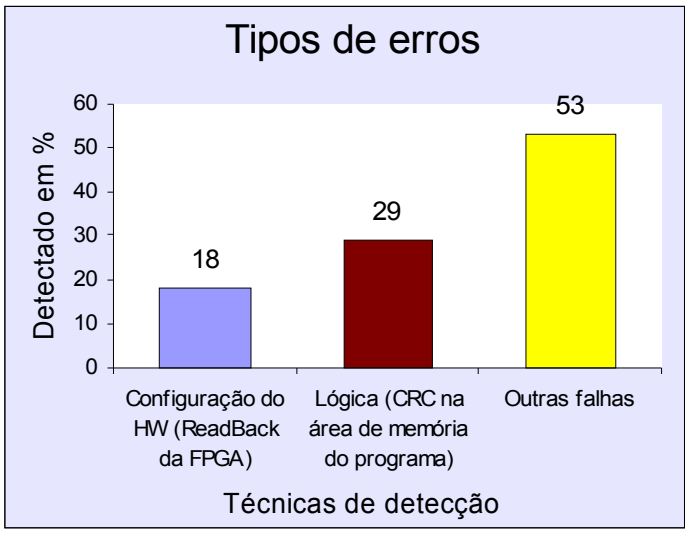


Ilustração 8.3.3.2: Tipos de erros detectados utilizando o algoritmo gerador de números primos.

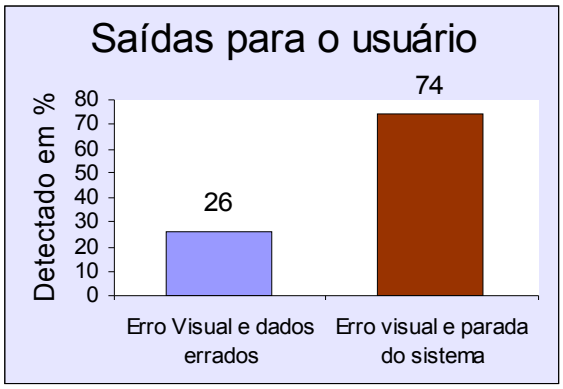


Ilustração 8.3.3.3: Saídas do programa em execução utilizando o algoritmo gerador de números primos.

8.3.4 Teste Utilizando Filtro IIR

A Ilustração 8.3.4.1 apresenta um resumo da capacidade de detecção de erros das técnicas utilizadas durante o funcionamento do sistema no interior da célula GTEM, para teste de imunidade irradiada. Nota-se uma pequena taxa de detecção de erro na utilização de semáforos pelo RTOS uC/OS-II e uma significante diferença entre o WDP-IP/HW frente as demais técnicas. O WDP-IP+ implementado em *software* (o qual combina as técnicas YACCA e WDP-IP/SW) apresentou pouco acréscimo na taxa de detecção frente a versão do WDP-IP/SW. Já a técnica de detecção YACCA apresentou menor taxa de detecção em relação as demais técnicas, salvo a detecção por violação dos semáforos, nativos do próprio RTOS uC/OS-II.

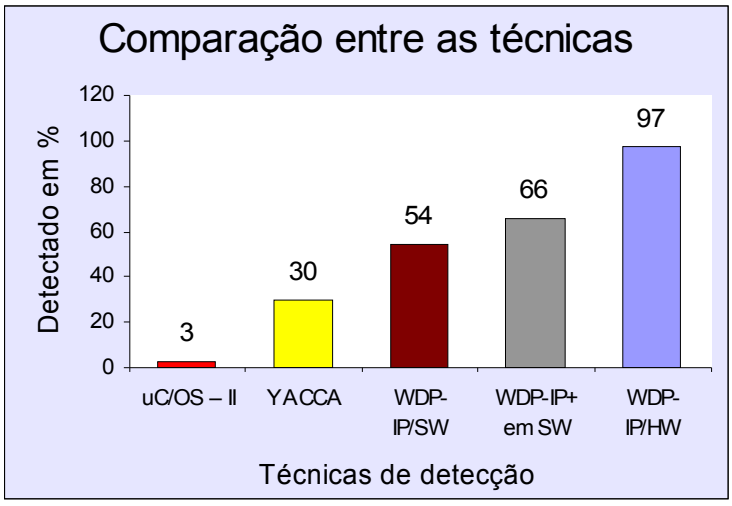


Ilustração 8.3.4.1: Comparação entre as técnicas de detecção utilizando o algoritmo filtro IIR.

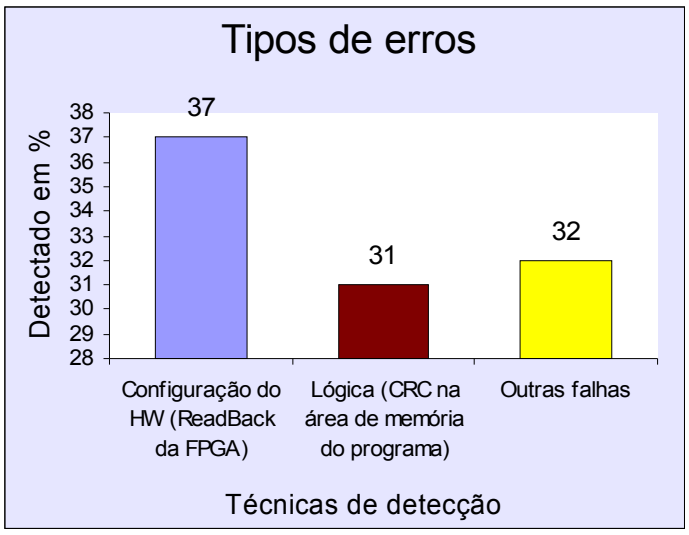


Ilustração 8.3.4.2: Tipos de erros detectados utilizando o algoritmo filtro IIR.

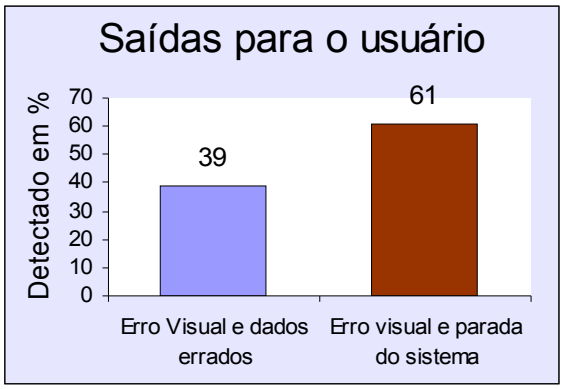


Ilustração 8.3.4.3: Saídas do programa em execução utilizando o algoritmo filtro IIR.

9 CONCLUSÃO

9.1 Conclusão do Teste de Injeção de Falhas via *Software*

O comparativo entre os testes da cobertura de detecção de erros é mostrado na Ilustração 9.1.1 para a injeção de falhas de *bit-flip* e na Ilustração 9.1.2 para a injeção de falhas de desvio. Como pode-se observar, a técnica WDP-IP/HW tem maior cobertura de detecção de erro que a implementação pura em *software* do WDP-IP/SW. Isto pode ser explicado pelo fato de que as falhas podem atingir a operação do RTOS quando em modo supervisor, fazendo com que a própria operação da aplicação e a técnica embarcada do WDP-IP/SW se corrompam. Já com o WDP-IP/HW, como uma entidade independente externa, esta degradação já não ocorre.

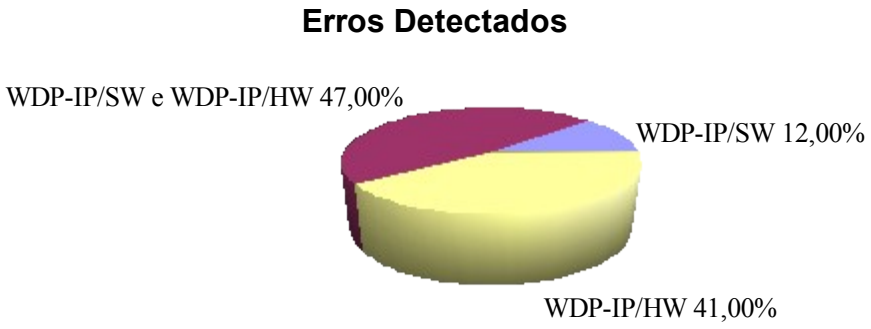


Ilustração 9.1.1: Comparação entre as técnicas WDP-IP implementados em *hardware* e em *software* para a injeção de falhas aleatórias de *bit-flip*.

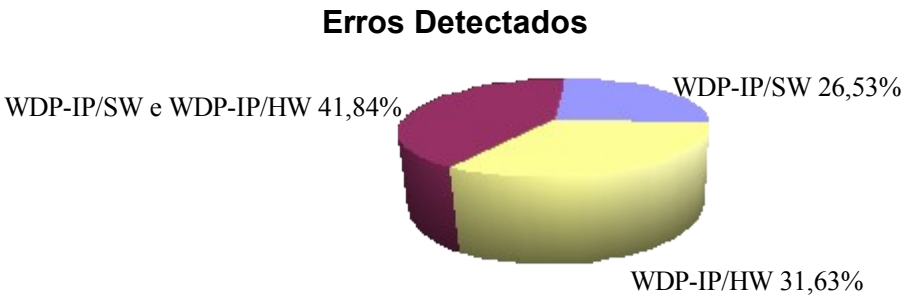


Ilustração 9.1.2: Comparação entre as técnicas WDP-IP implementados em *hardware* e em *software* para a injeção de falhas aleatórias de desvio.

9.2 Conclusões do Teste de Injeção de Falhas via *Hardware*

Analisando detalhadamente a tabela de testes, em todos os caso em que o WDP-IP/SW detectou erro, o WDP-IP/HW também detectou. O WDP-IP/HW foi capaz de detectar os mais variados erros, tanto erros observados pela assinatura gerada pelo CRC na área de memória da aplicação quanto alterações no *Bitstream* de configuração da FPGA.

As técnicas utilizando o algoritmo YACCA e WDP-IP+ em *software* foram capazes de detectar erros em falhas permanentes e transientes. A maioria dos erros detectados foram em falhas transientes (erros detectado pelo CRC e outras falhas) enquanto poucos erros foram detectados em falhas permanentes (erro no *Bitstream* da configuração da FPGA).

O RTOS uC/OS-II detectou somente erros quando alocava recursos de semáforo no momento em que falhas não identificadas (outras falhas) foram geradas no processo de injeções de falhas.

9.3 Conclusões Gerais

A capacidade de detecção de erros foi avaliada a partir de vários experimentos de injeção de falhas, levando em consideração falhas reais que afetam os elementos de memória ou que afetaram a configuração da FPGA. Demais falhas como inversão de um registrador, comunicação no barramento, entre outras, foram agrupadas em outras falhas.

Ficou evidente nos testes a eficácia da técnica embarcada em *hardware.* Em todos os casos a capacidade de detecção foi de longe superior as técnicas embarcadas em *software*, principalmente porque a maioria das falhas culminaram na instabilidade do sistema levando-o a parar de executar.

Comparando todas as técnicas no quesito de detecção de erros podemos classificar as técnicas WDP-IP, uma vez embarcadas no núcleo do RTOS, como as mais hábeis, fáceis de utilização frente outras técnicas já estudadas.

Assim, as principais vantagens do método proposto nesta dissertação podem ser salientadas nos seguintes pontos:

- O método detecta eficientemente um grande número de falhas transientes e permanentes, que podem ser localizadas na configuração da FPGA, na comunicação entre os módulos, em algum registrador do processador ou na área de memória que

armazena o código ou os dados;

- É uma solução bastante apropriada quando se consideram SoCs, uma vez que esta não exige qualquer modificação nos núcleos do processador e da memória, e somente a inserção de um WDP-IP/HW no barramento do processador;
- O WDP-IP/HW é completamente independente do código executado pelo processador, quando mudanças são introduzidas no código, o WDP-IP/HW não precisa ser alterado;
- O custo por área do WDP-IP/HW é relativamente reduzido e se considerar que com uma pequena alteração pode controlar inúmeros processadores, se torna relativamente menor ainda;
- Os custos de memória (*overheads*) e de desempenho da versão implementada em *hardware* são significativamente menores que os adicionados pelas soluções implementadas puramente em *software*;
- Pode identificar a tarefa que apresentou erro, somado a isso, a versão melhorada (denomidada WDP-IP+) permite identificar o bloco básico da tarefa que apresentou erro. Essas informações são importantes para a realização de futuros tratamentos;
- Tem grande portabilidade para diversas arquiteturas, haja visto que o acesso ao WDP-IP/HW é similar a uma memória SRAM e a implementação da parte em *software* pode ser descrita em linguam C ANSI.

9.4 Trabalhos Futuros

- Realizar mais testes para as técnicas propostas;

- Implementar mais um campo no módulo MEMÓRIA INDEXADA para que o WDP-IP trabalhe em ambiente multiprocessado (vários processadores em paralelo ou distribuídos) como visto na técnica SEIS [8,9,10];
- Atribuir recursos de recuperação do sistema.

10 REFERÊNCIAS BIBLIOGRÁFICAS

[1] Teixeira J P, Teixeira I C, Santos M B, Andina J J R, Piccoli L B, Rodriguez I, Marcial, Vargas F. Aumento de Tolerância Dinâmica de Circuitos Electrónicos Integrados Digitais a Variações de Tensão de Alimentação e de Temperatura. Patente. Lisboa, Potugal. 2006;v. único.

[2] Goloubeva O, Rebaudengo M, Reorda M S, Violante M. Soft-error Detection Using Control Flow Assertions. IEEE Int. Sym. on Defect and Fault Tolerance in VLSI Sys. Washington, DC, USA. 2003;581-588.

[3] Goloubeva O, Rebaudengo M, Reorda M S, Violante M. Improved Software-Based Processor Control-Flow Errors Detection Technique. Ann. Reliab. and Maint. Symp. Alexandria, USA. 2005;583-589.

[4] Ball S R. Embedded Microprocessor Systems Real World Design. 2nd Edition.United States of America:Butterworth-Heinemann; 2000. p. 50-128.

[5] Berger A S. Embedded Systems Design: An Introduction to Processes, Tools and Techniques. Berkeley:CMP Books; 2002. p. 67-208.

[6] Madeira H, Silva J G. On-Line Signature Learning and Checking: Experimental Evaluation. IEEE Adv. Comp. Technology. Tucson, Arizona, USA. 1991;642-646.

[7] Madeira H, Camoes J, Silva J G. Signature Verification: A New Concept for Building Simple and Effective Watchdog Processors. 6th Mediterranean Electrotechnical Conf. LJubljana, Slovenia. 1991;2:1188-1191.

[8] Majzik I, Pataricza A, Cin M, Hohl W, Hönig J, Sieh V. Hierarchical Checking of Multiprocessors Using Watchdog Processors. Proc. European Dependable Comp. Conf. on Dependable Comp. London, UK. 1994;852:386-403.

[9] Majzik I, Hohl W, Pataricza A, Sieh V. Multiprocessor Checking using Watchdog Processors. Comp. Sys. Science and Eng. Budapest, Hung. 1996;5:301-310.

[10] Pataricza A, Majzik I, Hohl W, Hoenig J. Watchdog Processors in Parallel Systems. Microprocessing and Microprogramming. Amsterdam, The Netherlands. 1993;39(2-5):69-74.

[11] IEC. IEC standards. Int. Electrotechnical Commission. [periódico online]. 2006 [capturado 2006 Jul 21]; Disponível em: http://www.iec.ch.

[12] Veríssimo P, Lemos R. Confiança no Funcionamento: Proposta para uma Terminologia em Português. INESC Technical Report. [periódico online]. 1989 [capturado 2006 Jul 01]; Disponível em: www.navigators.di.fc.ul.pt/docs/abstracts/terminology.html.

[13] Laprie J C. Dependable Computing and Fault Tolerance: Concepts and Terminology. 15th IEEE Int. Sym. on Fault Tolerant Comp. Ann Arbor, Michigan, USA. 1985;2-11.

[14] Labrosse J J. MicroC/OS-II The Real-Time Kernel. 2nd Edition.Berkeley:CMP Books; 2002.

[15] Vargas F, Piccoli L, Alecrim A A, Moraes M, Gama M. Time-Sensitive Control-Flow Checking Monitoring for Multitask SoCs. IEEE East-West Design & Test Workshop. Sochi, Russia. 2006;v. único.

[16] Vargas F, Piccoli L, Alecrim A A, Moraes M, Gama M. Time-Sensitive Control-Flow Checking Monitoring for Multitask SoCs: on EMI-Based Case-Study. IEEE Int. Conf. on Field Prog. Technology. Bangkok, Thailand. 2006;v. único.

[17] Vargas F, Piccoli L B, Alecrim A A, Moraes M, Gama M. Hybrid Solutions for Leveraging SoC Robustness in EMI-Exposed Environments. Proc. Design & Test of Sys. on Chip for EMC Workshop. Barcelona, Spain. 2006;v. único.

[18] Xilinx, Inc. FPGA. Xilinx. [periódico online]. 2006 [capturado 2006 Jul 21]; Disponível em: http://www.xilinx.com/.

[19] Johnson B W. The Electrical Engineering Handbook. 2nd Edition.Boca Raton:CRC Press; 1997.
[20] Avizienis A. Toward Systematic Design of Fault-Tolerant Systems. IEEE Comp. Los Alamitos, CA, USA. 1997;30(4):51-58.
[21] Clark J A, Pradhan D K. Fault Injection A Method for Validating Computer-System Dependability. IEEE Comp. Washington, DC, USA. 1995;26(6):47-56.
[22] Miremadi G, Torin J. Evaluating Processor Behavior and Three Error Detection Mechanisms Using Physical Fault-Injection. IEEE Trans. on Reliability. 1995;44(3):441-453.
[23] Vardanega T, David P, Chane J-F, Mader W, Messaros R, Arlat Jean. On the Development of Fault-Tolerant On-Board Control Software and its Evaluation by Fault Injection. 25th IEEE Int. Sym. on Fault Tolerant Comp. Sys. Washington, DC, USA. 1995;510-515.
[24] Pradhan D K. Fault-Tolerant Computer System Design. United States of America:Prentice-Hall; 1995. p. 4-6.
[25] Pandya M, Malek M. Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Task. IEEE Trans. on Comp. Washington, DC, USA. 1998;47(10):1102-1112.
[26] Weaver C, Austin T. A Fault Tolerant Approach to Microprocessor Design. Proc. Int. Conf. on Dependable Sys. Goteborg, Sweden. 2001;411-420.
[27] Thiagrajan S. Survey of Fault-Tolerant Techniques in Modern Micro-Processors. University of Wisconsin-Madison [periódico online]. 2006 [capturado 2006 Jul 01]; Disponível em: homepages.cae.wisc.edu/~ece753/INFO.html.
[28] Carreira J, Silva J G. Why do Some (weird) People Inject Faults?. ACM SIGSOFT Software Eng. Notes. New York, NY, USA. 1998;23(1):42-43.
[29] Benso A, Carlo S , Natale G, Prinetto P. A Watchdog Processor to Detect Data and Control Flow Errors. 9th IEEE On-Line Testing Sym. Kos Island, Greece. 2003;144.
[30] Mukherjee S S, Kontz M, Reinhardt S K. Detailed Design and Evaluation of Redundant Multithreading Alternatives. Proc. 29th Annual Int. Sym. of Comp. Arch. Washington, DC, USA. 2002;99-110.
[31] Rebaudengo M, Reorda M S, Torchiano M, Violante M. Soft-error Detection through Software Fault-Tolerance techniques. IEEE Int. Sym. on Defect and Fault Tolerance in VLSI Sys. Washington, DC, USA. 1999;210-218.
[32] Stroud C E. A Designer's Guide to Built-In Self Test. United States of America:Kluwer Academic Publishers; 2002. p. 15-27.
[33] Cortner J M. Digital Test Engineering. United States of America:Wiley-Interscience; 1987. p. 1-27.
[34] Ceschia M, Violante M, Reorda M S, Paccagnella A, Bernardi P, Rebaudengo M, et al. Identification and Classification of Single-event Upsets in the Configuration Memory of SRAM-based FPGAS. IEEE Trans. on Nuclear Science. 2003;50(6):2088-2094.
[35] Violante M, Ceschia M, Reorda M S , Paccagnella A, Bernardi P, Rebaudengo M, et al. Analyzing SEU Effects in SRAM-based FPGAs. IEEE Int. On-Line Testing Sym. Kos Island, Greece. 2003;119-123.
[36] Bellato M, Bernardi P, Bortolato D, Candelori A, Ceschia M, Paccagn A. Evaluating the Effects of SEUs Affecting the Configuration Memory of an SRAM-Based FPGA. Design, Automation and Test in Europe. Washington, DC, USA. 2004;188-193.
[37] Malcolm N, Zhao W. The Timed-Token Protocol for Real-Time Communications. IEEE Comp. Los Alamitos, CA, USA. 1994;27(1):35-41.
[38] Ramamritham K, Stankovic J A. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. IEEE Proceedings. 1994;82(1):55-67.

[39] Shin K G , Ramanathan P. A New Discipline of Computer Science and Engineering. IEEE Proceedings. Australian. 1994;82(1):6-24.
[40] Marwedel P. Embedded System Design. Netherlands:Springer; 2006. p. 67-208.
[41] Wang Y-C, Lin K-J. Implementing A General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel. IEEE Real-Time Sys. Sym. Washington, DC, USA. 1999;246-255.
[42] Laplante P A. Real-Time Systems Design and Analysis. 3nd Edition.United States of America:Wiley-IEEE Press; 2004. p. 2-105.
[43] Ngolah C F, Wang Y, Tan X. The Real-Time Task Scheduling Algorithm of RTOS+. IEEE Canadian Journal of Electrical and Comp. Eng. Canada. 2004;29(4):237-243.
[44] Lee C G, Hahn J, Seo Y, Min S L, Ha R, Hong S, Park C Y, Lee M, Kim C S. Analysis of Cache-related Preemption Delay in Fixed-priority Preemptive Scheduling. IEEE Trans. on Comp. Washington, DC, USA. 1998;47(6):700-713.
[45] S. Chakraborty, T. Erlebach, S. K unzli, L. Thiele. Schedulability of Event-Driven Code Blocks in Real-Time Embedded Systems. Proc. 39th Design Automation Con. New Orleans, USA. 2002;616-621.
[46] Feng W, Liu J W S. Algorithms for Scheduling Real-Time Tasks with Input Error and End-to-End Deadlines. IEEE Trans. on Software Eng. Piscataway, NJ, USA. 1997;23(2):93-106.
[47] Jackson L, Rouskas G. Deterministic Preemptive Scheduling of Real Time Tasks. IEEE Comp. Magazine. Raleigh, NC, USA. 2002;35(5):72-79.
[48] Bertossi A A, Mancini L V, Rossini F. Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time System. IEEE Trans. on Parallel and Distributed Sys. Piscataway, NJ, USA. 1999;10(1):934-945.
[49] Gerber R, Pugh W, Saksena M. Parametric Dispatching of Hard Real-Time Tasks. IEEE Trans. on Comp. Washington, DC, USA. 1995;44(3):471-479.
[50] Dertouzos M L, Mok A K. Multiprocessor Online Scheduling of Hard-Real-Time Tasks. IEEE Trans. on Software Eng. Piscataway, NJ, USA. 1989;15(12):1497-1506.
[51] Burnstein I. Practical Software Testing: A Process-oriented Approach. Chicago:Springer-Verlag; 2003. p. 20-281.
[52] Pardo J, Campelo J C, Serrano J J. Robustness Study of an Embedded Operating System for Industrial Applications. 28th Annual Int. Comp. Software and App. Conf. Washington, DC, USA. 2004;2:64-65.
[53] Pullum L L. Software Fault Tolerance Techniques and Implementation. British:Artech House; 2001. p. 27-29.
[54] Chillarege R, Bowen N. Understanding Large-System Failures: A Fault-Injection Experiment. IEEE 19th Int. Sym. on Fault Tolerant Comp. Sys. Los Alamitos, CA, USA. 1989;356-363.
[55] Cristian F, Aghili H, Strong R. Clock Synchronization in the Presence of Omissions and Performance Faults, and Processor Joins. Proc. 16th IEEE Int. Sym. on Fault Tolerant Comp. Sys. Vienna. 1986;218-223.
[56] Hsueh M, Tsai T, Iyer R K. Fault Injection Techniques and Tools. IEEE Comp. Los Alamitos, CA, USA. 1997;30(4):75-82.
[57] Karlsson J, Folkesson P, Arlat J, Crouzet Y, Leber G, Reisinger J. Comparison and Integration of Three Diverse Physical Fault Injection Techniques. 2th Predictably Dependable Computing Systems Houston, Texas, USA. 1994;615-642.
[58] Karlsson J, Folkesson P, Arlat J, Crouzet Y, Leber G. Integration and Comparison of Three Physical Fault Injection Techniques. Pred. Depend. Comp. Sys. Vienna, Austria. 1995;309-329.

[59] Karlsson J, Folkesson P, Arlat J, Crouzet Y, Leber G, Reisinger J. Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture. 5th IFIP Work. Conf.on Depend. Comp. for Critic. App. Urbana-Champaign, USA. 1998;267-287.

[60] Choi G S, Iyer R K. FOCUS: An Experimental Environment for Fault Sensitivity Analysis. IEEE Trans. on Comp. Washington, DC, USA. 1992;41(12):1515-1526.

[61] Karlsson J, Lidén P, Dahlgren P, Johansson R, Gunneflo U. Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms. IEEE Micro Magazine. Los Alamitos, CA, USA. 1994;14(1):8-23.

[62] Vargas F, Lopes D C, Gatti E, Prestes D P, Lupi D, Rhod E, et al. EMI-Based Fault Injection. Proc. 6th IEEE Latin American Test Workshop. Salvador - BA, Brazil. 2005;91-96.

[63] Vargas F, Lopes D C, Gatti E, Prestes D P, Lupi D. On the Proposition of an EMI-Based Fault Injection Approach. 11th IEEE Int. On-Line Test. Symp. Saint-Raphael, France. 2005;207-208.

[64] Steininger A, Rahbaran B, Handl T. Built-in Fault Injectors - The Logical Continuation of BIST?. Proc. Workshop on Intelligent Sol. in Emb. Sys. Vienna, Austria. 2003;187-196.

[65] Vargas F, Benfica J, Farina A, Bezerra E, Gatti E, Garcia L, et al. Observing SRAM-Based FPGA Robustness in EMI-Exposed Environments. 7th IEEE L.A. Test Work. Buenos Aires, Argentina. 2006;201-206.

[66] Kanawati G A, Kanawati N A, Abraham J A. FERRARI: A Flexible Software-Based Fault and Error Injection System. IEEE Trans. on Comp. Washington, DC, USA. 1995;44(2):248-260.

[67] Eifert J B, Shen J P. Processor Monitoring Using Asynchronous Signatured Instruction Streams. 14th IEEE Int. Sym. on Fault Tolerant Comp. Washington, DC, USA. 1984;394-399.

[68] Oh N, Mitra S, McCluskey E J. ED4I: Error Detection by Diverse Data and Duplicated Instructions. IEEE Trans. on Comp. Washington, DC, USA. 2002;51(2):180-199.

[69] Miremadi G, Karlsson J, Gunneflo U, Torin J. Two Software Techniques for On-Line Error Detection. 22th IEEE Int. Sym. on Fault Tolerant Comp. Boston, MA, USA. 1992;328-335.

[70] Kanawati G A, Nair V S S, Krishnamurthy N, Abraham J A. Evaluation of Integrated System-Level Checks for On-Line Error Detection. Proc. IEEE Int. Comp. Performance and Dependability Sym. Washington, DC, USA. 1996;292-301.

[71] Alkhalifa Z, Nair V S S, Krishnamurthy N, Abraham J A. Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. IEEE Trans. on Parallel and Distributed Sys. Piscataway, NJ, USA. 1999;10(6):627-641.

[72] Alkhalifa Z, Nair V S S. Design of a Portable Control-Flow Checking Technique. Proc. 2nd High-Assurance Sys. Eng. Workshop. Washington, DC, USA. 1997;120-123.

[73] Oh N, Shirvani P P, McCluskey E J. Control-Flow Checking by Software Signatures. IEEE Trans. on Reliability. Lafayette, Colorado, USA. 2002;51(2):111-122.

[74] Mahmood A, McCluskey E J. Concurrent Error Detection Using Watchdog Processors-A Survey. IEEE Trans. on Comp. Washington, DC, USA. 1998;37(2):160-174.

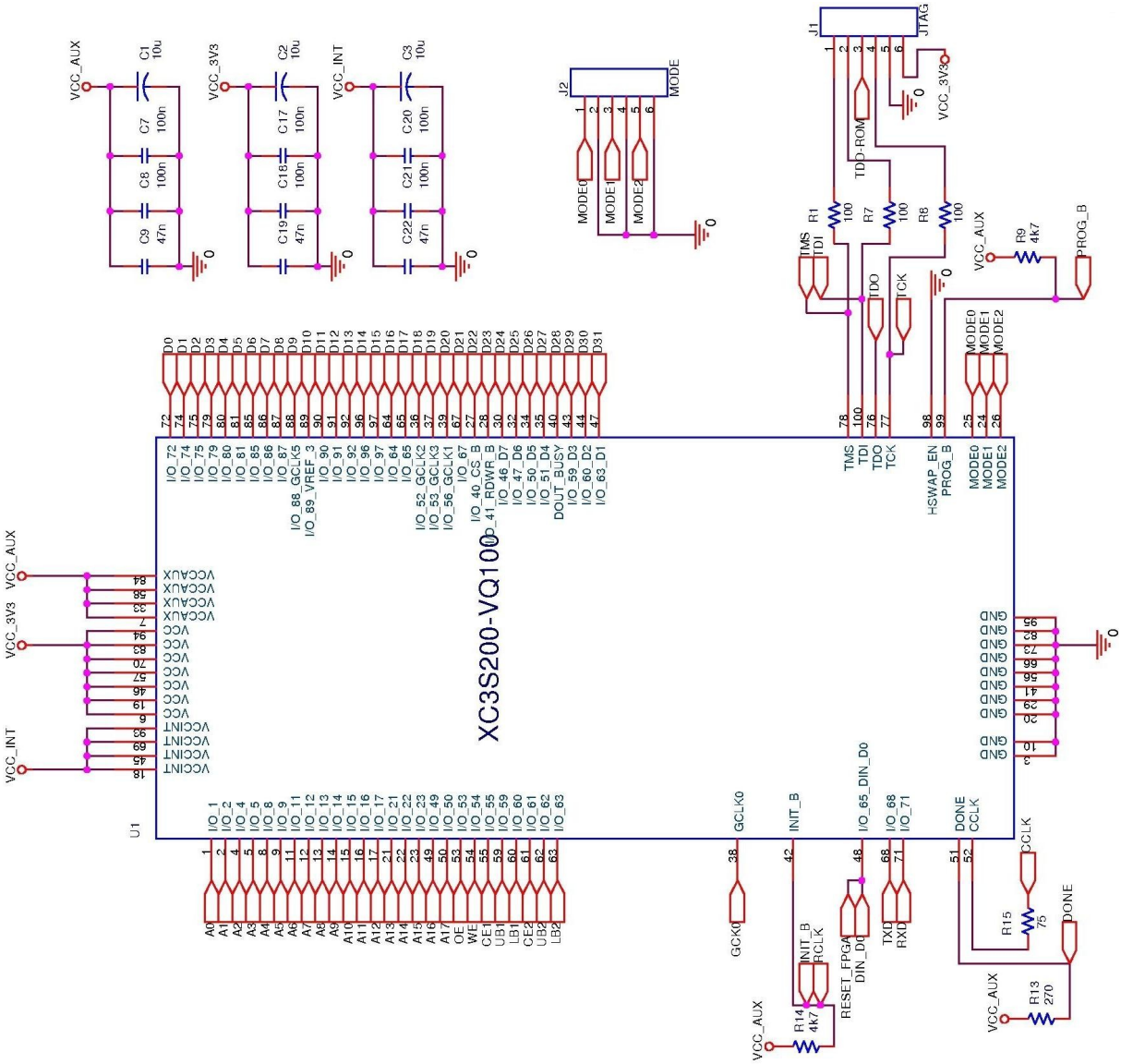
[75] Upadhyaya S, Ramamurthy B. Concurrent Process Monitoring with No Reference Signatures. IEEE Trans. on Comp. Washington, DC, USA. 1994;43(4):475-480.

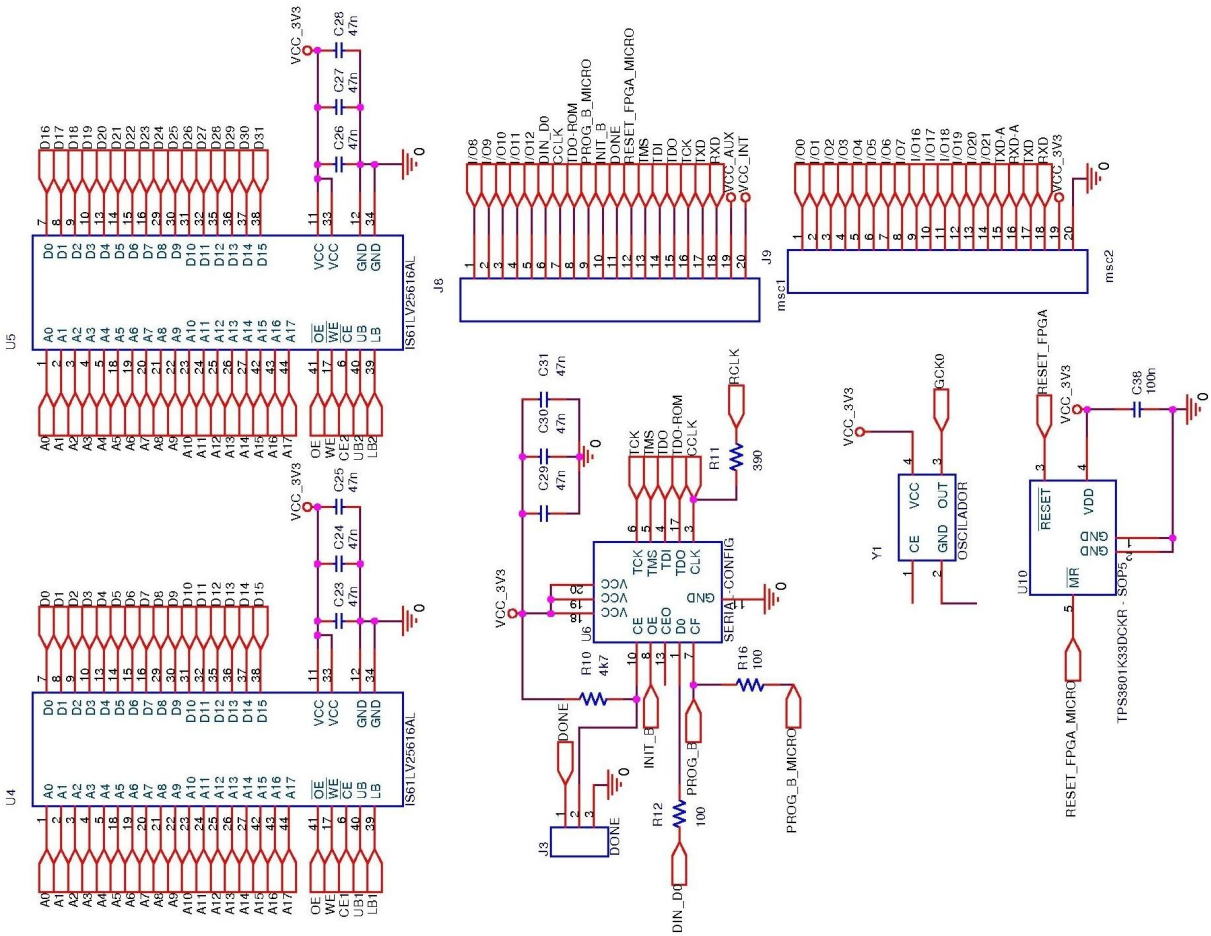
[76] Wilken K, Shen J P. Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Errors. IEEE Trans. on Comp. Aided Design and Sys. Washington, DC, USA. 1990;9(6):629-641.

[77] Warter N, Hwu W. A Software Based Approach to Achieving Optimal Performance for Signature Control Flow Checking. IEEE Trans. on Comp. Chapel Hill, NC, USA. 1994;43(2):129-140.
[78] Michel T, Leveugle R, Saucier G. A New Approach to Control Flow Checking without Program Modification. 21th IEEE Trans. on Comp. New York, NY, USA. 1993;42(11):1372 - 1381.
[79] Saxena N R, McCluskey W K. Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums. IEEE Trans. on Comp. Washington, DC, USA. 1990;39(4):554-559.
[80] Tomas S, Shen J. A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems. Proc. Int. Conf. on Comp. Design: VLSI in Comp and Processors. Amsterdam, The Netherlands. 1985;531-539.
[81] Hnig J, Sieh V. Software-Based Concurrent Control Flow Checking. Submitted 24th Int. Fault-Tolerant Comp. Sym. Austin, Texas, USA. 1993.
[82] Pardo J, Campelo J C, Serrano J J. Robustness Study of an Embedded Operating System for Industrial Applications. Proc. 29th IEEE Annual Int. Comp. Software and App. Conf. Washington, DC, USA. 2004;2:64-65.
[83] ESCO. 5300 Gigahertz Transverse Electromagnetic GTEM. ESCO Technologies. [periódico online]. 2006 [capturado 2006 Jul 21]; Disponível em: http://www.emctest.com/Manuals.cfm.
[84] Klaus Finkenzeller. RFID Handbook: Fundamentals and Applications inContactless Smart Cards and Identification. 2nd Edition.:John Wiley & Sons; 2003.
[85] Chernak Y. Validating and Improving Test-Case Effectiveness. IEEE Software. Los Alamitos, CA, USA. 2001;28(1):81-86.
[86] Weisstein E W. CRC Concise Encyclopedia of Mathematics. 2nd Edition.United States of America:Chapman & Hall/CRC; 2002.p. 642.
[87] Williams R N. A Painless Guide to CRC Error Detection Algorithms. Rocksoft [periódico online]. 1993 [capturado 2006 Jul 21]; Disponível em: http://www.ross.net/crc/download/crc_v3.txt.
[88] ANSI. ANSI standards. American National Std. Inst. [periódico online]. 2006 [capturado 2006 Jul 21]; Disponível em: http://www.ansi.org/.

11 APÊNDICES

A) Diagrama Esquemático do Dispositivo Sob Teste (DUT)




U5
IS61LV25616AL
U4
IS61LV25616AL
VCC_3V3
C28 47n
C27 47n
C26 47n
C25 47n
C24 47n
C23 47n
C31 47n
C30 47n
C29 47n
J6
J9
msc1
msc2
I/O8
I/O9
I/O10
I/O11
I/O12
DIN_D0
CCLK
TDO-ROM
PROG_B_MICRO
INIT_B
DONE
RESET_FPGA_MICRO
TMS
TDI
TDO
TCK
TXD
RXD
VCC_AUX
VCC_INT
I/O0
I/O1
I/O2
I/O3
I/O4
I/O5
I/O6
I/O7
I/O16
I/O17
I/O18
I/O19
I/O20
I/O21
TXD-A
RXD-A
TXD
RXD
VCC_3V3
U6
SERIAL-CONFIG
R10 4k7
R11 390
R12 100
R16 100
J3
DONE
DIN_D0
PROG_B_MICRO
RCLK
Y1
OSCILADOR
GCK0
U10
TPS3801K33DCKR - SOP5
RESET_FPGA
RESET_FPGA_MICRO
C38 100n

B) Resumos das Normas

A medida de imunidade eletromagnética depende de parâmetros específicos do sistema e da aplicação. A imunidade eletromagnética de um DUT pode ser caracterizada por perturbações de RF conduzida ou radiada. Na primeira parte da IEC 62132[11] sugere critérios para determinar se um DUT está trabalhando corretamente ou não durante a exposição a EMI irradiado através de diversas classes (de "A" a "D"):

- Classe "A": desempenho normal dentro dos limites especificados do sistema;
- Classe "B": degradação nas informações de saída do sistema, ocasionando mudanças de estado ou perda de dados;
- Classe "C": degradação do sistema ocorre e permanece assim até que o usuário reinicie o mesmo;
- Classe "D": Defeito no sistema devido a dano no circuito integrado.

Outras normas também são atendidas pelo teste na GTEM (ver seção 7.13.1 na página 128) a exemplo:

- **Norma IEC 61000-4-3** [11] (Técnicas de teste e medidas de imunidade a campo eletromagnético radiado): A faixa de freqüência é de 80 MHz a 1 GHz. O DUT é disposto nos seus quatro lados com polarização horizontal e vertical ao sinal AM senoidal modulado de amplitude de 80% e 1KHz. O DUT é aplicado a uma distância entre 1 a 3 m da antena com amplitude de 1V/m, 3V/m e 10 V/m;
- **Norma ANSI C63.4** [88] (Técnicas de teste e medidas de pertubações em RF): A faixa de freqüência é de 9KHz a 40GHz, para dispositivos de baixa voltagem (menores que 600 Volts DC ou AC) e aplicado a uma distância de 3, 10 ou 30 m da antena.

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE
DO SUL FACULDADE DE ENGENHARIA

PROGRAMA DE PÓS-GRADUAÇÃO DE ENGENHARIA
ELÉTRICA - PPGEE

DISSERTAÇÃO DE MESTRADO

SOLUÇÕES HÍBRIDAS DE HARDWARE/SOFTWARE PARA A DETECÇÃO DE ERROS EM SYSTEMS-ON-CHIP (SoC) DE TEMPO REAL

LEONARDO BISCH PICCOLI

PORTO ALEGRE
Agosto, 2006