PONTIFICIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE ENGENHARIA

PROGRAMA DE POS-GRADUAÇÃO DE ENGENHARIA ELÉTRICA

EXPLORANDO UMA SOLUÇÃO HÍBRIDA: HARDWARE + SOFTWARE PARA A DETECÇÃO DE FALHAS TEMPO REAL EM SYSTEMS-ON-CHIP (SoCs)

LETICIA MARIA VEIRAS BOLZANI

PORTO ALEGRE

2005

PONTIFICIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE ENGENHARIA
PROGRAMA DE POS-GRADUAÇÃO DE ENGENHARIA ELÉTRICA

EXPLORANDO UMA SOLUÇÃO HÍBRIDA: HARDWARE + SOFTWARE PARA A DETECÇÃO DE FALHAS EM SYSTEMS-ON-CHIP (SoCs)

LETICIA MARIA VEIRAS BOLZANI

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
2005

EXPLORANDO UMA SOLUÇÃO HÍBRIDA: HARDWARE + SOFTWARE PARA A DETECÇÃO DE FALHAS EM SYSTEMS-ON-CHIP (SoCs)

CANDIDATA: LETICIA MARIA VEIRAS BOLZANI

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. Renato Ribas – UFRGS

_____

Prof. Dr. Rubem Fagundes – PUCRS

Agradecimentos

Ao meu orientador, Fabian Vargas, exemplo de dedicação e competência, que sempre me incentivou, apoiou e principalmente acreditou em meu trabalho.

Ao professor Daniel Barros Júnior, agradeço pelo apoio e orientação que recebi durante todos os dias de trabalho.

A todos os integrantes do Grupo SiSC, em especial aos colegas Dárcio Prestes e Diogo Brum, que contribuíram para a realização deste trabalho e proporcionaram um ambiente de trabalho agradável e motivador.

Aos meus queridos pais e irmãos, que sempre me incentivaram e me apoiaram incondicionalmente em todas as minhas escolhas, agradeço pelo amor e carinho.

Aos funcionários dos laboratórios e as secretárias do PPGEE agradeço por seu profissionalismo e competência.

A PUCRS e ao Departamento de Engenharia Elétrica pela disponibilidade dos recursos necessários para a realização desta dissertação de mestrado.

Ao Programa Nacional de Microeletrônica – CNPq, pelo apoio financeiro dado ao programa de pesquisa e pós-graduação.

Leticia Maria Veiras Bolzani

RESUMO

Nos últimos anos, o crescente aumento do número de aplicações críticas baseadas em sistemas eletrônicos, intensificou a pesquisa sobre técnicas de tolerância à falhas. Durante o período de funcionamento destes sistemas, a probabilidade de ocorrerem falhas transientes e permanentes devido à presença de interferências dos mais variados tipos é bastante grande. Dentre as falhas mais freqüentes, salientam-se as falhas que corrompem os dados e as falhas que alteram o fluxo de controle do processador que executa a aplicação. Assim, a utilização de técnicas capazes de detectarem estes tipos de falhas evita que as mesmas se propaguem pelo sistema e acabem gerando saídas incorretas. Basicamente, estas técnicas são classificadas em dois grandes grupos: soluções baseadas em software e soluções baseadas em hardware.

Neste contexto, o objetivo principal deste trabalho é especificar e implementar uma solução híbrida, parte em software e parte em hardware, capaz de detectar em tempo de execução eventuais falhas em dados e no fluxo de controle do algoritmo. Esta solução baseia-se nas técnicas propostas em (REBAUDENGO, 2004) e (GOLOUBEVA, 2003) e implementa parte de suas regras de transformação de código via software e parte via hardware. Assim, informações redundantes são agregadas ao código da aplicação e testes de consistência são implementados via hardware. Em resumo, este trabalho propõe o desenvolvimento de um núcleo I-IP (*infrastructure intellectual property*), tal como um *watchdog*, para executar os testes de consistência concorrentemente à execução da aplicação.

Para isto, três versões diferentes do I-IP foram implementadas em linguagem de descrição de hardware (VHDL) e avaliadas através de experimentos de injeção de falhas. A primeira versão implementada provê a detecção de falhas em dados e, como todo protótipo, este também apresenta algumas restrições e limitações. A segunda versão também detecta falhas em dados, entretanto, supera todos os problemas da versão anterior. A terceira versão do I-IP agrega à versão anterior a capacidade de detectar falhas de fluxo de controle. Finalmente, após a implementação das versões anteriores, foi especificada uma quarta versão que agrega confiabilidade e robustez ao I-IP desenvolvido através da utilização de algumas técnicas de tolerância a falhas e da especificação de um auto-teste funcional.

Os resultados obtidos a partir da avaliação das versões do I-IP garantem que a metodologia proposta neste trabalho é bastante eficiente, pois apresenta uma alta cobertura de falhas e supera os principais problemas presentes nas soluções baseadas em software propostas na literatura, ou seja, degradação de desempenho e maior consumo de memória.

Finalmente, cabe mencionar que esta dissertação é o resultado parcial de atividades que fazem parte do escopo do Projeto Alfa (#AML/B7-311-97/0666/II-0086-FI) mantido entre os Grupos SiSC – PUCRS (Brasil) e CAD – Politecnico di Torino (Itália) no período de 2002-2005.

Palavras chaves: aplicações críticas, técnicas de tolerância a falhas, falhas em dados, falhas de fluxo de controle, soluções baseadas em software, soluções baseadas em hardware, solução híbrida.

ABSTRACT

The always increasing number of computer-based safety-critical applications has intensified the research over fault tolerance techniques. While those systems are working, the probability of both permanent and transient faults happens due to the presence of all sort of interference. The common faults are those which affect data and/or modify the expected program execution flow. Thus, the use of techniques allowing detecting these type of faults presents them from propagating to system output. Basically, these techniques are categorized in two groups: software-based approaches and hardware-based approaches.

Considering the above introduced, the goal of this work is to specify and to implement a hybrid approach, which combines software-based techniques and hardware-based ones, capable to detect run time data and algorithm control flow faults. It is settled around the techniques proposed in (REBAUDENGO, 2004) and (GOLOUBEVA, 2003). Nevertheless, the proposed approach implements part of its code-transformation rules via software and hardware. These redundant information is added to the software portion and consistency checks are implemented via hardware. Summary, we propose the development of an I-IP (infrastructure intellectual property) core, such as watchdog, to correctly execute the consistency checks concurrently to the application execution.

In this work, three different versions of the I-IP were implemented in VHDL and analyzed by means of fault injection experiments. The first implemented version allows data fault detection and, as any prototype, has its limitations. The second version also detects data faults, but eliminates the problems of the former version. The third I-IP version adds the capability of detecting control flow faults to the previous versions of the I-IP. Finally, after implementing these three versions, a fourth version was specified. It adds dependability and robustness to the I-IP by using Built-in Self-Test (BIST) techniques.

The results obtained from evaluating the different I-IP core versions guarantee that the hybrid approach is efficient, because it features high fault coverage and surpasses the main problems present in software-based techniques proposed in the literature, such as, performance degradation and code/data memory overhead.

Finally, this work is a partial result of a joint research project carried by the SiSC Group – PUCRS and CAD – Politecnico di Torino, under the scope of the Alfa Project (##AML/B7-311-97/0666/II-0086-FI, from 2002 to 2005).

Key-words: safety-critical applications, fault tolerance techniques, data faults, control flow faults, software-based approaches, hardware-based approaches, hybrid approach.

SUMÁRIO

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

LISTA DE ABREVIATURAS

MOSFET – Metal Oxide Silicon Field Effect Transistor

NFETS – Negative Field Effect Transistor

PFETS – Positive Field Effect Transistor

NMOS – Negative Metal Oxide Silicon

VLSI – Very Large Scale Integration

PCB – Printer Circuit Board

ASIC – Application Specific Integrated Circuit

FPGA – Field Programmable Array

CUT – Circuit Under Test

HDL – Hardware Description Language

FC – Fault Coverage

CI – Circuit Integrated

DFT – Design for Testability

CAD – Computer Aided Desgin

ATE – Automatic Test Equipament

BIST – Build in Self-Test

ATPG –Automatic Test Pattern Generation

ETA – Electronic Design Automation

LSSD – Level-Sensitive Scan Design

SRL – Shift-Register Latch

TAP – Port Access Port

TDI – Test Data Input

TDO – Test Data output

TCK – Test Clock

TMS – Test Mode Select

TPG – Test Pattern Generation

ORA – Output Response Analyzer

LFSR – Linear Feedback Shift Register

CA – Cellular Automata

FSM – Finite State Machine

ROM – Read Only Memory

RAM – Randon Access Memory

SAR – Signature Analysis Register

MISR – Multiple-Input Signature Register

TMR – Triple Modular Redundancy

NMR – N- Modular Redundancy

BFI – Branch Free Intervals

BID – Branch-Free Interval IDentifier

CFID – Control Flow IDentifier

CFCSS – Control flow checking signature by software

ECCA – Enhanced Control Flow using Assertions

CCA – Control Flow Checking by Assertions

YACCA – Yet Another Control-Flown Checking using Assertions

GSR – Global Shift Register

SoC – System-on-Chip

COTS - Commercial-of-the-shelf

SIHFT - Software Implemented Hardware Fault Tolerance

I-IP - Infrastructure Intellectual Property

Observação: Várias abreviaturas serão mantidas em inglês, pois para um leitor habituado com os temas abordados se torna muito mais agradável a leitura do texto.

PARTE I – FUNDAMENTOS

1. INTRODUÇÃO

1.1 Motivação

O número de aplicações críticas baseadas em sistemas eletrônicos cresceu significativamente nos últimos anos. Este crescimento intensificou a pesquisa relacionada às técnicas de tolerância a falhas, 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 tolerância a falhas podem ser divididas sinteticamente em soluções baseadas em software e soluções baseadas em hardware.

Várias metodologias baseadas em software e baseadas em hardware, com o intuito de agregarem confiabilidade a aplicações críticas, já foram proposta na literatura. Evidentemente, que estas metodologias apresentam uma série de vantagens e desvantagens relacionadas ao custo, ao desempenho e ao *overhead* de área agregado aos sistemas. Por isto, quando se deseja agregar confiabilidade a um determinado sistema é necessário levar em consideração todos estes aspectos.

Neste contexto, visando manter as vantagens oferecidas por cada uma das duas soluções e minimizar as penalidades decorrentes de suas utilizações, este trabalho propõe uma solução híbrida, parte implementada em software e parte implementada em hardware, para prover a detecção de falhas transientes e permanentes.

1.2 Visão Geral dos Objetivos

O principal objetivo deste trabalho é especificar, implementar e avaliar uma solução híbrida capaz de detectar falhas em dados e falhas de fluxo de controle em aplicações críticas durante seu funcionamento. A nova metodologia proposta baseia-se nas técnicas de tolerância a falhas implementadas via software propostas em (CHEYNET, 2000) e (GOLOUBEVA, 2003).

Esta solução propõe que as técnicas mencionadas no parágrafo anterior sejam implementadas parte em software e parte em hardware. Neste sentido, a duplicação das variáveis

e operações e a inserção de instruções redundantes necessárias para o controle do fluxo do programa são implementadas em software enquanto os testes de consistência são implementados em hardware através de um I-IP (*infrastructure intellectual property*).

Outro objetivo consiste em agregar confiabilidade e segurança ao I-IP desenvolvido neste trabalho através da especificação de um auto-teste funcional para o mesmo.

1.3 Apresentação dos Capítulos

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

Parte I – Fundamentos (Capítulos 1 ao 6):

- Capítulo 2: apresenta uma breve introdução relacionada aos principais conceitos que envolvem a teoria de teste;
- Capítulo 3: explora o conceito e as principais abordagens de projeto visando o teste (*Design for Testability* – DFT);
- Capítulo 4: apresenta detalhadamente os principais conceitos sobre auto-teste embutido (*Built-in Self-Test* – BIST). Este tipo de abordagem é utilizado para a realização de testes *off-line*, sejam eles funcionais ou estruturais, em circuitos e sistemas eletrônicos;
- Capítulo 5: apresenta o conceito e as quatro técnicas básicas para o projeto de sistemas tolerantes a falhas através do uso de redundância. Este tipo de abordagem é utilizado para a realização de testes *on-line* de circuitos e sistemas digitais em geral;
- Capítulo 6: aborda as principais metodologias de tolerância a falhas de hardware implementadas via software (*Software Implemented Hardware Fault Tolerance* - SIHFT). Estas são, mais especificamente, um tipo de redundância de software, propostas para a detecção de falhas em dados e de falhas no fluxo de controle de sistemas baseados em SoCs. Salienta-se que estas metodologias constituem o foco principal deste trabalho.

Parte II – Metodologia (Capítulo 7):

- Capítulo 7: apresenta detalhadamente a solução híbrida proposta neste trabalho de dissertação para detectar falhas em dados e falhas no fluxo de controle. A metodologia proposta baseia-se em soluções implementadas puramente em software, descritas detalhadamente no capítulo 6, e define que parte das regras de transformação de código seja implementada via software e parte via hardware. Neste contexto, o código da aplicação modificado a partir das regras de transformação de código e as três versões do núcleo I-IP implementadas durante este trabalho serão apresentados. Finalmente, será apresentada a especificação de uma quarta versão do núcleo I-IP que agrega algumas técnicas de tolerância a falhas, descritas no capítulo 4, a fim de aumentar sua confiabilidade e robustez.

Parte III – Resultados e Conclusões (Capítulos 8 e 9):

- Capítulo 8: visando avaliar a metodologia proposta no capítulo anterior, o capítulo 8 apresenta os resultados obtidos a partir de experimentos de injeção de falhas em cada uma das versões do I-IP implementado;
- Capítulo 9: apresenta as conclusões relacionadas a cada uma das versões do I-IP implementado, as conclusões relacionadas ao desenvolvimento deste trabalho de dissertação e finalmente sugere alguns trabalhos para serem desenvolvidos no futuro.

2. INTRODUÇÃO A TEORIA DE TESTE

2.1 Introdução

Nas últimas décadas, o mercado de sistemas eletrônicos evoluiu muito e conseqüentemente, passou a exigir que os circuitos fossem cada vez mais rápidos, complexos e fundamentalmente confiáveis. A confiabilidade, qualidade e segurança exigidas pelo mercado geraram a necessidade de agregar técnicas e mecanismos capazes de proverem eficientemente a detecção e eventualmente correção de defeitos. Esses defeitos podem ocorrer durante as fases de especificação, manufatura e vida útil dos sistemas.

Neste contexto, o teste, apontado como um processo extremamente crítico e importante, passa a fazer parte de todo o processo de desenvolvimento dos sistemas. Assim, o desenvolvimento de um determinado produto envolve basicamente três fases distintas e o teste desempenha um papel extremamente importante em cada uma delas. Durante a primeira fase, denominada fase de projeto, deve-se realizar a verificação da especificação a fim de detectar e identificar erros inseridos durante esta etapa garantindo assim, que o produto desempenhará corretamente suas funções. Na segunda fase, denominada fase de manufatura, deve-se realizar testes que detectam todo e qualquer tipo de defeito introduzido durante o período de fabricação dos circuitos. Finalmente, durante o período de operação do sistema deve-se realizar testes a fim de detectarem eventuais falhas que possam ocorrer durante o período de funcionamento. A detecção de falhas durante o funcionamento do sistema impede sua propagação no sistema e consequentemente evita que saídas erradas sejam geradas.

Neste contexto, o principal objetivo deste capítulo é apresentar uma visão geral sobre teste e os principais conceitos e definições necessários para o total e completo entendimento das técnicas de tolerância à falhas.

2.2 Visão geral

A evolução da tecnologia de sistemas eletrônicos impulsionou o desenvolvimento da tecnologia de teste que se baseia em três áreas distintas, mas inter-relacionadas: o teste de hardware, o teste de software e a teoria de teste que juntas desenvolvem uma base sólida para o teste. Basicamente o teste pode ser classificado a partir de uma série de conceitos que serão definidos a partir de agora.

2.2.1 Teste Off-line versus Teste On-line

O teste é dito *off*-line quando o sistema deve interromper sua execução normal e dedicar-se única e exclusivamente à execução do procedimento de teste. Entretanto, quando o procedimento de teste é executado como uma tarefa comum às tarefas gerais do sistema o teste é dito ser *on-line*. Claramente, os testes realizados durante a fase de manufatura são considerados *off-line* e os testes realizados durante seu período de funcionamento são considerados *on-line*. Salienta-se que os testes *on-line* são indicados para sistemas que exigem alta confiabilidade e segurança, tais quais, sistemas automotivos, aéreos, médicos entre outros.

2.2.2 Teste Funcional versus Teste Estrutural

Geralmente, o objetivo do teste é verificar se um determinado projeto está de acordo com suas especificações, ou seja, verificar se o projeto desempenha corretamente e eficientemente todas as funções a ele atribuídas. Este tipo de teste é denominado teste funcional.

Entretanto quando o objetivo do teste concentra-se na verificação física da implementação do circuito a fim de analisar se a mesma corresponde ao diagrama esquemático inicialmente proposto, o teste é denominado estrutural.

2.2.3 Defeitos e Modelos de Falhas

O teste de circuitos ou sistemas eletrônicos é realizado com o intuito de detectar falhas eventualmente presentes. Entretanto, para a realização do teste é necessário definir modelos de falhas baseados em falhas reais definidas a partir de mecanismos físicos e *layouts* reais. Segundo Paul H. Bardell (BARDEL, 1987) um modelo de falha especifica a série de defeitos físicos que podem ser detectados através de um procedimento de teste. Um bom modelo de falha, segundo Charles E. Stroud (STROUD, 2002), deve ser computacionalmente eficiente em relação ao dispositivo de simulação e refletir fielmente o comportamento dos defeitos que podem ocorrer durante o processo de projeto e manufatura bem como o comportamento das falhas que podem ocorrer durante a operação do sistema. Estes modelos são utilizados na emulação de falhas e defeitos durante a etapa de simulação do projeto.

Assim, nos últimos anos surgiram vários modelos de falhas baseados nos principais defeitos físicos encontrados nos circuitos.

A) Modelo de falha Gate-Level Stuck-at

Este modelo define que as portas de entrada e saída podem estar *stuck-at-0* (sa0) ou *stuck-at-1* (sa1), ou seja, independente do valor que é setado para a porta, ela apresenta sempre um valor fixo em 0 ou em 1. A figura 2.1 apresenta a notação utilizada para representar falhas *stuck-at* e a figura 2.2 o comportamento das mesmas.

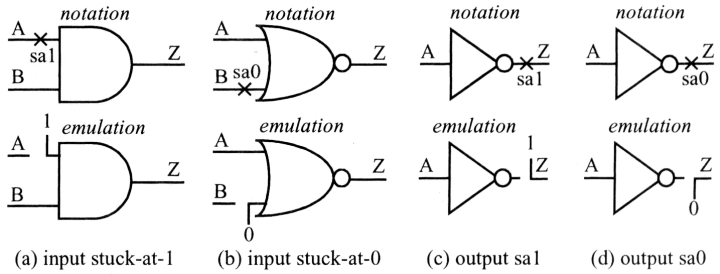
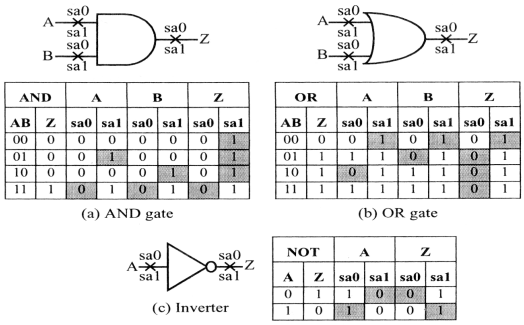


Figura 2.1 Notação do Modelo de Falha *Stuck-At*. (STROUD, 2002)



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

Figura 2.2 Comportamento do Modelo de Falha *Stuck-At*. (STROUD, 2002)

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

A) Modelo de falha Transistor-Level Stuck

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

A figura 2.3 ilustra o modelo de falha *Transistor-Level Stuck* e a figura 2.4 o seu comportamento.

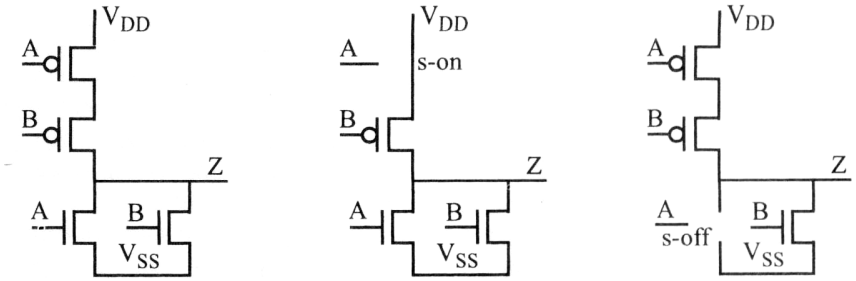


Figura 2.3 Modelo de Falha Transistor-Level Stuck. (STROUD, 2002)

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

Figura 2.4 Comportamento do Modelo de Falha Transistor-Level Stuck. (STROUD, 2002)

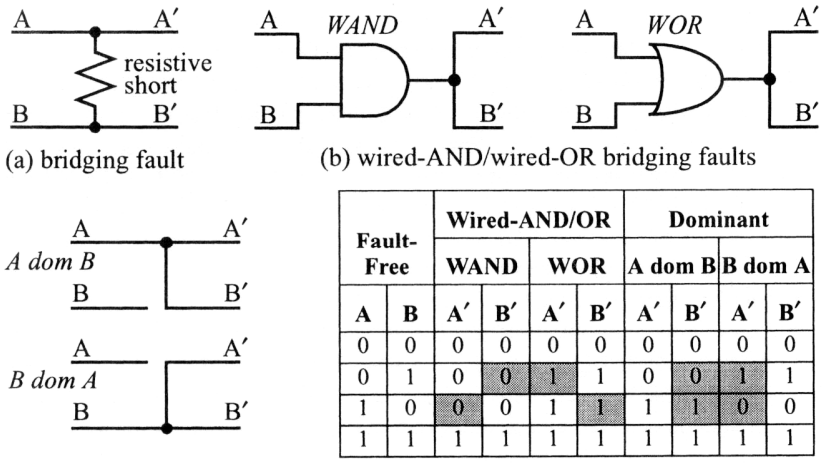
Salienta-se que as falhas *stuck-on* (*s-on*) podem ser emuladas através de um curto circuito entre o *source* e o *drain* do transistor e as falhas *stuck-off* (*s-off*) desconectando-se o transistor do circuito. Alternativamente, falhas *stuck-on* podem ser emuladas desconectando a porta MOSFET do 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* - MOSFET) a lógica 0 para transistores NFET e a lógica 1 para transistores PFET, assim o transistor nunca conduzirá.

A partir da análise da figura 2.2 e da figura 2.4 observa-se que um mesmo conjunto mínimo de vetores de teste é capaz de detectar simultaneamente falhas *stuck-on*, *stuck-off* e *stuck-at*.

B) Modelo de falha Bridging

Este modelo inclui um importante conjunto de defeitos físicos, tais como curtos entre trilhas e rompimento de trilhas (trilhas abertas). Basicamente estes tipos de defeitos resultam de um *over-etching* ou *under-etching* durante o processo de fabricação do VLSI ou PCB.

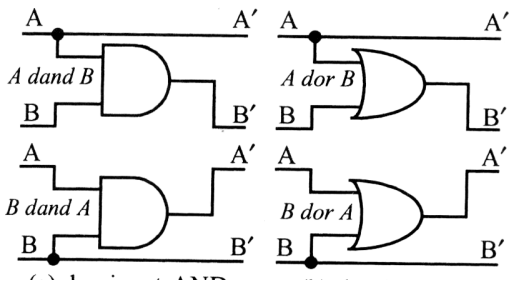
A figura 2.5 ilustra o modelamento de falhas *bridging* para dois modelos específicos denominados *wired-AND/wired_OR bridging* e *dominant bridging*.



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

Figura 2.5 Modelos de Falhas *wired-AND/wired-OR bridging* e *dominant bridging*. (STROUD, 2002)

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 figura 2.6 ilustra este modelo.



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

Figura 2.6 Modelo de Falha *dominant-AND/OR bridging*. (STROUD, 2002)

Observa-se que apesar das falhas *transistor-level* e *bridging* refletirem mais fielmente o comportamento dos defeitos presentes em circuitos, sua emulação e avaliação em simuladores é computacionalmente mais complexa em relação as tradicionais falhas *stuck-at*.

C) 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 na freqüência de operação requerida. Este tipo de falha origina-se a partir de um *over* ou *under etching* durante o processo de fabricação que origina MOSFETs com canais muito mais estreitos ou longos do que os pretendidos.

Assim, o teste de *Delay* concentra-se em encontrar e expor todo e qualquer defeito que possa existir no dispositivo fabricado. 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 falho.

Tipicamente, o teste de *Delay* consiste na aplicação seqüencial de dois vetores tal 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.

D) Modelo de falha simples versus múltiplos

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 abaixo 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.

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

2.2.4 Tipos de Falhas

Charles E. Stroud (STROUD, 2002) apresenta o conceito de falhas equivalentes, que consiste em um conjunto de falhas distintas com o mesmo comportamento falho. O reconhecimento de falhas equivalentes em um determinado circuito é bastante importante pois a detecção de uma delas garante conseqüentemente a detecção de todas as suas falhas equivalentes. A utilização do conceito acima descrito diminui significativamente o tempo de simulação devido fundamentalmente à redução do número de falhas a serem emuladas.

Charles E. Stroud (STROUD, 2002) ainda apresenta o conceito de *fault collapsing* que consiste em remover do conjunto de falhas a serem emuladas as falhas equivalentes. Charles E. Stroud (STROUD, 2002) salienta uma redução de aproximadamente 50% no número total de falhas a serem emuladas para um modelo de falha *gate-level collapsing* em relação a um modelo *uncollapsing*. Já no que diz respeito ao modelo de falha *transistor-level* a redução gira em torno de 25%. Para se obter um conjunto reduzido de falhas salienta-se a técnica de amostragem estatística de falhas que se baseia no princípio de amostrar randomicamente o conjunto total de falhas e assim gerar o conjunto reduzido.

Paul H. Bardell (BARDEL, 1987) também sugere que falhas podem ser classificadas como simples ou múltiplas. Falhas simples são aquelas que afetam um componente em particular e normalmente ocorrem durante a vida útil do sistema. Por outro lado, falhas múltiplas afetam vários componentes e geralmente são oriundas de defeitos de fabricação.

Além disto, Paul H. Bardell (BARDEL, 1987) sugere que as falhas podem ser seqüenciais ou combinacionais. Curtos ou rompimentos de trilhas entre condutores geram dois tipos de comportamentos falhos. Quando as linhas que estão em curto formarem um caminho de *feedback*, que por sua vez, cria um novo estado em que o circuito pode operar, a falha é denominada seqüencial. Entretanto, se este caminho de *feedback* não for formado a falha é denominada combinacional.

Além das classificações acima descritas, Ricardo Augusto da Luz Reis (REIS, 2000), sugere ainda que falhas podem ser permanentes, transitórias e intermitentes. Falhas permanentes podem ser geradas durante o processo de fabricação bem como durante a vida útil do circuito. Elas podem ser representadas fisicamente por curtos ou interconexões abertas, que consistem em uma condição falha permanente para o circuito. Já as falhas transitórias aparecem durante a vida

útil do circuito e são geradas por fenômenos aleatórios tais como interferência eletromagnética. E por fim, as falhas intermitentes são caracterizadas pela ocorrência temporária e repetida do defeito a partir de alguma variação nas condições externas ao circuito, como por exemplo, ocorrência de vibrações, variações da temperatura etc.

2.2.5 Teste e Simulação de Falhas

A partir da análise da figura 2.7 é possível observar que o teste consiste basicamente na aplicação de um conjunto de estímulos de entrada no circuito em teste (*circuit under test* – CUT) e na comparação, das respostas obtidas durante o teste, com um conjunto de respostas esperadas a fim de determinar se o circuito está livre ou não de falhas. Salienta-se que, durante o processo de verificação do projeto, os estímulos de entrada aplicados são vetores de teste definidos com o intuito de verificar se o projeto está de acordo com as especificações e desempenha corretamente suas funções. Os vetores são aplicados na linguagem de descrição de hardware (*hardware description language* - HDL) do circuito no ambiente de simulação a fim de produzir as correspondentes respostas (saídas). Além disso, os mesmos vetores também são freqüentemente utilizados como vetores funcionais durante o teste de manufatura.

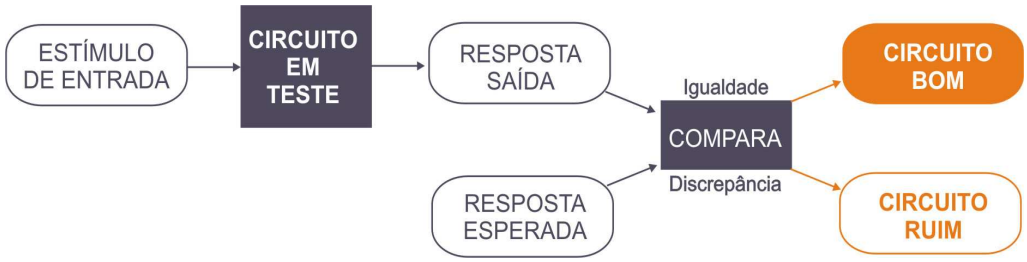


Figura 2.7 Fluxo básico do teste. (STROUD, 2002)

Outra etapa extremamente importante é a simulação de falhas que basicamente segue o fluxo da figura 2.7. A simulação apresenta apenas uma diferença em relação ao esquema acima descrito, além dos vetores de teste, uma lista de falhas é emulada no circuito em teste (*circuit under test* – CUT) e executada no ambiente de simulação de falha. Assim, o simulador emula cada uma das falhas presentes na lista e aplica os vetores de teste. Subseqüentemente, cada

resposta obtida é comparada com a resposta associada às respostas obtidas durante a simulação do circuito livre de falha. Segundo Charles E. Stroud (STROUD, 2002) uma falha é considerada detectada se a resposta gerada a partir de um determinado vetor de teste é diferente da esperada, ou seja, o circuito falho produziu uma resposta errada em relação as respostas do circuito livre de falha. Entretanto, se nenhum resultado diferente do esperado é obtido, a falha é considerada não detectada.

Assim, simulação de falha consiste basicamente na simulação do circuito na presença de falhas de um determinado modelo e na comparação destes resultados com os resultados obtidos durante a simulação do circuito sem a presença de falhas. O principal objetivo é verificar se estas falhas são ou não detectadas através da aplicação de determinados estímulos de entrada. Em resumo, o processo de simulação consiste em: (1) simulação do circuito sem falhas; (2) redução da lista de falhas, pela remoção de falhas que apresentem comportamento equivalente e falhas que dominam outras falhas; (3) injeção de falhas na descrição do circuito; (4) simulação do circuito com falhas; (5) comparação dos resultados da simulação e diante de uma determinada discordância, remover a falha da lista inicial de falhas. A figura 2.8 apresenta o esquemático de um procedimento de simulação.

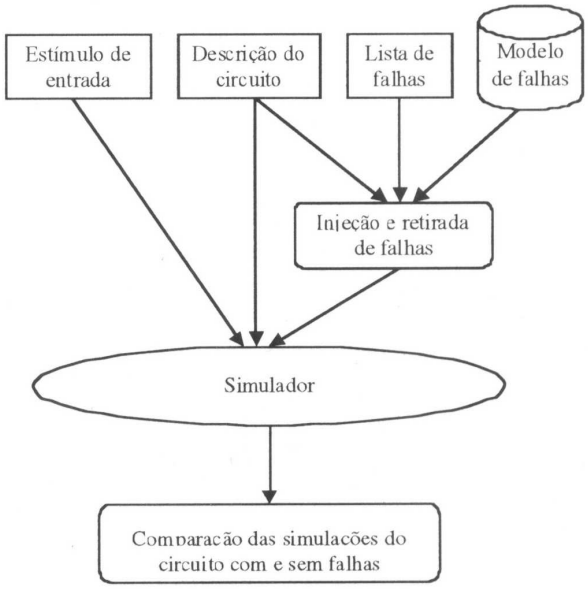


Figura 2.8 Esquemático de um procedimento de simulação de falhas genérico. (REIS, 2000).

A técnica de simulação de falhas pode ser utilizada para: (1) avaliação do teste, ou seja, através da simulação podemos obter o percentual de falhas detectadas pelos estímulos de entrada

aplicados; (2) eliminação de falhas da lista original na geração automática do teste, verificando quais falhas do modelo são detectadas pelo estímulo de teste calculado, entre outras.

Após o término da simulação, ou seja, após a emulação de todas as falhas, é possível avaliar a cobertura de falha obtida a partir da aplicação dos vetores de teste. Assim, segundo Charles E. Stroud (STROUD, 2002) cobertura de falha (*fault coverage* – FC) é a medida quantitativa da eficácia do conjunto de vetores de teste em detectarem as falhas e é dada pela equação (2.1):

$$FC = \frac{D}{T} , \qquad (2.1)$$

Onde: D é o número de falhas detectadas e T o número total de falhas presentes na lista.

Quanto ao tempo de simulação, pode-se afirmar que ele depende de vários fatores tais como o tamanho do circuito, o número de vetores de teste e de falhas a serem simulados e o tipo de simulador que será utilizado. Normalmente, este tempo é bastante significativo e representa a maior fatia de tempo dedicado às etapas de projeto e desenvolvimento de teste. Assim, a fim de reduzir o tempo de simulação, surge a técnica denominada *Trip-on-first-failure*, que consiste em descartar da simulação a falha detectada e seguir simulando somente as outras falhas ainda não detectadas.

No que diz respeito ao tipo de simulador, ele pode ser serial ou paralelo. No simulador paralelo o tempo de simulação pode ser consideravelmente reduzido em relação ao serial que, por sua vez, emula somente uma falha de cada vez.

Assim, é possível concluir que o número de falhas a serem simuladas é função do tamanho do CUT e do modelo de falhas utilizado e influencia diretamente o tempo exigido para a simulação do circuito.

Após definir o modelo de falhas apropriado a um determinado circuito e realizar a simulação de falhas, a geração dos vetores de teste é um processo extremamente importante. Entretanto, encontrar um conjunto de estímulos de entrada eficaz e que garanta a máxima cobertura de falhas é um processo bastante difícil. A figura 2.9 ilustra um procedimento genérico de geração de vetores de teste.

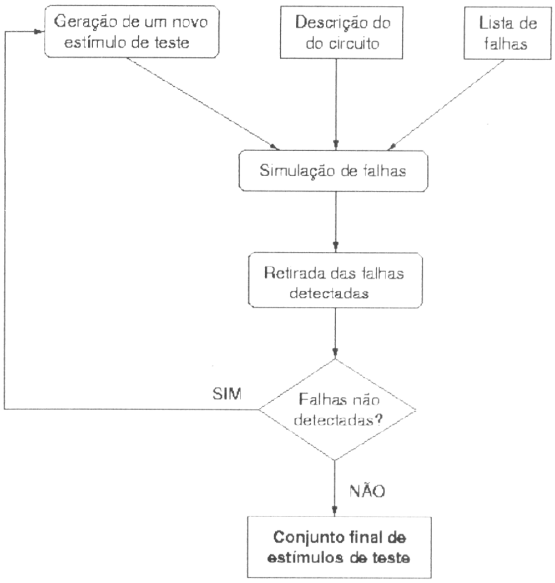


Figura 2.9 Procedimento genérico de geração de teste.(REIS, 2000)

Assim, a geração de vetores de teste pode ser feita de várias formas, são elas: exaustiva, pseudo-exaustiva, randômica, pseudo-randômica, etc. Salienta-se que no capítulo 4, intitulado Auto-Teste Embutido, serão descritas as principais metodologias utilizadas para a geração de vetores de teste.

3. PROJETO VISANDO O TESTE

3.1 Introdução

Até início dos anos 80, o teste de circuitos digitais era considerado um processo totalmente isolado das fases de projeto e manufatura. Entretanto, após este período a área de teste sofreu alterações bastante significativas e revolucionárias baseadas fundamentalmente nas mudanças e evoluções que ocorreram na tecnologia de desenvolvimento dos circuitos integrados (*circuits integrated* - CIs). Dentre as principais mudanças, salientam-se, o aumento em sua densidade e a alteração no número e nos tipos de falhas que podem afetá-los.

O aumento na densidade dos CIs acarreta em uma redução bastante significativa na capacidade de se controlar e observar determinados pontos internos específicos, com isto, os esforços exigidos por parte dos projetistas em busca de circuitos livres de falhas tornam-se cada vez maiores. Além disto, o aumento na densidade dos CIs, gera um aumento significativo no custo de desenvolvimento e aplicação do teste e uma crescente dificuldade em relação a uma completa cobertura de falhas.

Quanto ao número e tipo de falhas que podem afetar os CIs, eles dependem diretamente do tipo de dispositivo e da tecnologia utilizada para fabricá-lo. Assim, a validação e a avaliação da qualidade do teste torna-se um processo extremamente difícil e computacionalmente complexo. Neste contexto, Janusz Rajski (RAJSKI, 1998) sugere a utilização do conceito de cobertura de falhas como uma forma indireta de medir a qualidade do teste. Segundo Janusz Rajski (RAJSKI, 1998), cobertura de falha é a razão entre o número de falhas detectadas e o número total de falhas existentes no domínio assumido.

Salienta-se que os testes em nível de placa de circuito impresso e em nível de sistema são normalmente bastante difíceis e complexos e por isto, é absolutamente imprescindível a realização de testes durante todas as fases de desenvolvimento do projeto, ou seja, durante as fases de projeto, manufatura e vida util.

Janusz Rajski (RAJSKI, 1998), menciona alguns problemas encontrados durante a realização de testes em nível de chip, são eles:

- Grande e crescente razão entre *logic-to-pin*, ou seja, existe uma relação de desequilíbrio entre o número de portas de entradas e saídas e número dos dispositivos semicondutores que devem ser acessados através delas;
- O crescente aumento na complexidade dos circuitos devido a tecnologias submicron que acarretam em alta densidade e velocidade;
- Significativo aumento do tempo requerido durante o processo de geração de padrões de teste e de aplicação do teste;
- Um volume excessivo de dados de teste que devem ser armazenados pelo equipamento de teste;
- Equipamentos de teste externos executam testes em baixas velocidades;
- Projetistas desconhecem a estrutura a nível de portas lógicas;
- Falta de métodos e métricas capazes de avaliarem completamente os esquemas de testes utilizados;

Em vista do exposto, observa-se que a utilização de paradigmas convencionais de teste baseados na utilização de testadores externos é cada vez mais inviável devido fundamentalmente ao seu custo, que por sua vez, é diretamente proporcional a complexidade dos CIs.

Assim, testabilidade passa a ser um objetivo primordial dentre as etapas de desenvolvimento de projetos. Segundo Janusz Rajski (RAJSKI, 1998) testabilidade reflete a habilidade dos vetores de teste gerados detectarem e possivelmente localizarem as falhas geradas durante as etapas de desenvolvimento dos CIs.

Neste contexto, surge o conceito de Projeto Visando o Teste (*Design for Testability*- DFT).

3.2 Técnicas de DFT

Segundo Paul H. Bardel (BARDEL, 1987), técnicas de DFT englobam uma série de métodos e técnicas de projeto que visam gerar um projeto testável.

Segundo Charles Stroud (STROUD, 2002), técnicas de DFT são capazes de aumentarem a testabilidade de um circuito através da inclusão de elementos extras que aumentam a controlabilidade e observabilidade do CUT.

Assim, técnicas de DFT baseiam-se na idéia de preocupar-se com o teste desde o início do projeto do CI. Neste contexto, as técnicas de DFT são utilizadas com a finalidade de facilitarem sobremaneira o teste de CIs e visam aumentar a testabilidade dos mesmos através da inserção de elementos de hardware extras. Salienta-se que aumentar a testabilidade de um determinado circuito significa aumentar sua controlabilidade e sua observabilidade. Os conceitos de controlabilidade e observabilidade serão discutidos e apresentados posteriormente.

Por isto, a utilização destas técnicas durante o desenvolvimento do projeto diminui e muitas vezes soluciona vários dos problemas relacionados a testabilidade de CIs. Apesar de normalmente aumentarem o tempo dedicado a fase de projeto do CI, as técnicas de DFT reduzem sobremaneira o tempo de desenvolvimento de teste devido fundamentalmente a utilização de ferramentas automáticas de síntese para gerarem o hardware extra necessário. Além das vantagens anteriormente citadas, o uso destas técnicas gera circuitos facilmente testáveis, reduz significativamente o custo associado ao desenvolvimento do teste (cerca de 40%), aumenta a cobertura de falhas e conseqüentemente aumenta a qualidade do teste e finalmente possibilitam a identificação de problemas no processo de manufatura e/ou fabricação, o que conseqüentemente reduz o número de defeitos dos circuitos.

Evidentemente, a utilização de técnicas de DFT agrega algumas penalidades que devem ser consideradas como critério para a seleção do método que será implementado. Dentre as principais penalidades agregadas aos CIs salientam-se o *overhead* de área gerado a partir da inserção de circuitos extras e a queda no desempenho. Por isto, é extremamente importante levar em consideração as seguintes características do circuito alvo da técnica: (1) número de pinos; (2) dissipação de potência do circuito; (3) tempo de aplicação do teste; (4) cobertura de falhas obtida a partir de modelos de falhas específicos; (5) suporte a ferramentas (*computer aided desgin* – CAD); (6) custo associado a equipamentos de teste automáticos (*automatic test equipament* – ATE); etc.

A escolha da metodologia de projeto que será utilizada pelo projetista depende de um grande número de fatores, dentre os quais salienta-se fundamentalmente o conhecimento que o projetista tem a cerca da implementação de seu projeto. Paul H. Bardel (BARDEL, 1987) define uma série de propriedades gerais, baseadas em experiências práticas, capazes de facilitarem a fase de teste, são elas: (1) existência ou não de lógica redundante e assíncrona; (2) isolamento ou não do clock em relação ao circuito; (3) facilidade na inicialização de circuitos seqüenciais; (4)

facilidade no diagnóstico e (5) apresenta um pequeno aumento no número de portas lógicas ou pinos em relação ao projeto normal.

Neste contexto, serão apresentadas as principais técnicas de DFT mencionadas na literatura. As técnicas *AdHoc*, as técnicas de projeto *Scan*, incluindo uma breve descrição a cerca da técnica *Boundary Scan* e as técnicas de Auto-Teste Embutido (*Built-in Self-test* - BIST). Salienta-se que na prática, diferentes técnicas podem ser utilizadas em diferentes porções do CUT ou múltiplas técnicas podem ser combinadas a fim de aumentar a eficiência do DFT.

Entretanto, antes da descrição das metodologias de DFT será apresentado e brevemente discutido os conceitos de controlabilidade e observabilidade.

3.2.1 Controlabilidade versus Observabilidade

Conforme mencionado anteriormente, testabilidade é definida como sendo a combinação de dois atributos, são eles: controlabilidade e observabilidade.

Segundo Janusz Rajski (RAJSKI, 1998), controlabilidade é a medida da dificuldade de definir para um ponto interno do circuito um valor que seja capaz de gerar uma falha, ou seja, segundo L. M. Cortês (CORTÊS, 1991) controlabilidade é definida como a facilidade para estabelecer valores para os sinais internos do circuito.

Segundo Janusz Rajski (RAJSKI, 1998), observabilidade é a medida da dificuldade de se propagar o sinal de falha de um ponto interno do circuito para uma saída primária, ou seja, segundo L. M Cortês (CORTÊS, 1991) é definida como a facilidade para se observar valores presentes nos nós internos do circuito.

Assim, o aumento da testabilidade de circuitos integrados está intimamente associado a controlabilidade e observabilidade.

Segundo Janusz Rajski (RAJSKI, 1998) a essência do DFT é aplicar o mínimo de mudanças no projeto do circuito original de forma que a controlabilidade e observabilidade do mesmo sejam melhoradas. Um conjunto de definições freqüentemente utilizado para controlabilidade e observabilidade de cada nó em um circuito inclui três valores que representam o grau relativo de dificuldade de:

- Alcançar 1 no nó (controlabilidade 1);

- Alcançar 0 no nó (controlabilidade 0);
- Direcionar os efeitos das falhas do nó para um saída primária.

As medidas acima tem sido utilizadas para empregar testes pseudo-randômicos ou determinísticos. No último caso, todas as medidas podem ser empregadas para guiar uma técnica de geração de padrão automática. No BIST o padrão pseudo-randômico é bastante utilizado. Assim, redefinindo os conceitos de controlabilidade e observabilidade:

- Controlabilidade 1 (0) de um nó é a probabilidade que um vetor de entrada aplicado randomicamente setará o nó para o valor 1 (0),
- Observabilidade de um nó é a probabilidade que um vetor de entrada aplicado randomicamente sensibilizará um ou mais caminhos da linha para uma saída primária.

Então, um circuito terá uma pequena controlabilidade e/ou observabilidade caso um único vetor de teste ou uma grande seqüência de teste é requerido para estabelecer o estado deste nó e então propagar este estado para as saídas do circuito.

O principal objetivo das técnicas de DFT é aumentar a capacidade de controlar e observar diretamente os estados das variáveis existentes nos circuitos.

3.2.2 Técnicas Ad Hoc

Nos últimos anos, várias técnicas de projeto tem sido utilizadas a fim de evitarem eventuais problemas durante a fase de teste. Neste contexto, as técnicas Ad Hoc aparecem como uma das principais alternativas utilizadas pelos projetistas.

Segundo Janusz Rajski (RAJSKI, 1998), técnicas Ad Hoc representam regras que devem ser seguidas durante o desenvolvimento do projeto a fim de aumentar a controlabilidade e observabilidade do circuito.

Segundo Charles E. Stroud (STROUD, 2002), técnicas Ad Hoc são definidos como um conjunto de bons métodos de projeto aplicados manualmente baseados única e exclusivamente no julgamento e na avaliação do projetista.

Entretanto, salienta-se que estas técnicas não geram nenhuma metodologia sistemática (algoritmo) que aumente a testabilidade do circuito e conseqüentemente não resolvem

completamente o problema de teste. Entretanto as iniciativas tomadas durante a fase de projeto de forma não estruturada e não sistematizada melhoram sobremaneira a testabilidade do circuito, ou seja, aumentam sua controlabilidade e observabilidade. Assim, dentre as principais iniciativas utilizadas salientam-se:

Pontos de teste: Pontos de teste podem ser adicionados a fim de facilitar a geração de uma falha (ponto de controle) e sua observação (ponto de observação). A figura 3.1 mostra um circuito onde foi injetado 0 e outro onde foi injetado 1 onde duas portas lógicas extras são utilizadas para armazenarem a controlabilidade 0 e 1 de uma linha de conexão do sub-circuito C1 e C2.

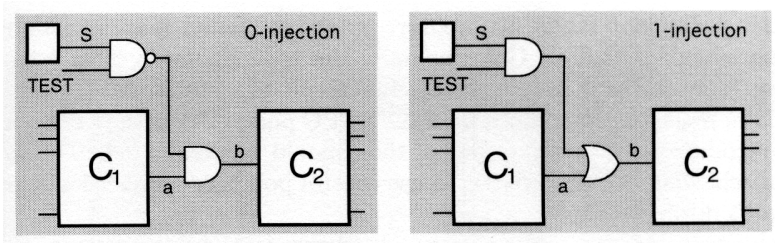


Figura 3.1 Pontos de Controle 0 e 1. (RAJSKY, 1998).

Clocks gerados internamente, monoestável multivibrators, e Osciladores: A fim de eliminar a necessidade de sincronizar o testador e pulsos internos com o circuito, esses dispositivos devem ser desabilitados durante o teste. Além disso, o teste pode ser executado na mesma velocidade do circuito.

Inicialização: Um circuito seqüencial deve ser setado com um estado conhecido antes de iniciar o teste. Isto pode ser obtido através de uma seqüência de inicialização pré-definida. Contudo, como a seqüência é usualmente criada pelo projetista, é improvável mostrá-la facilmente a fim de que seja recriada por um software gerador de padrões de teste automático (*automatic test pattern generation* – ATPG), ou seja, usada em um ambiente de auto teste embutido (*Built-in Self-Test* - BIST. Assim, é recomendado utilizar botões de *reset* ou um conjunto de entradas para os *flip-flops*.

Redundância lógica: A menos que seja adicionada intencionalmente para eliminar riscos ou para aumentar a confiabilidade, redundância lógica é um efeito altamente indesejado

que deve ser completamente evitado. A presença de redundância faz com que ferramentas de ATPG desperdicem tempo gerando testes inexistentes para as falhas redundantes. Contudo, falhas redundantes podem invalidar testes para falhas não redundantes. Infelizmente, a redundância é freqüentemente introduzida sem nenhum cuidado e, portanto extremamente difíceis de identificar e remover.

Caminhos de feedback globais: Quando se utiliza ATPG, os caminhos de *feedback* podem introduzir longos caminhos entre as portas lógicas e isto deve ser eliminado. Uma forma de evitar esta situação é utilizar pontos de controle ou alguma lógica que seja capaz de romper/cortar esses caminhos durante o teste.

Grandes contadores e registradores de deslocamento: Deve-se evitar a utilização de grandes contadores, pois estes exigem um número muito grande de ciclos de *clock* para alterarem os bits mais significativos. Uma solução bastante viável e utilizada é inserir alguns pontos de controle de forma que o contador (ou o registrador de deslocamento - *shift register*) seja particionado em pequenas unidades.

Blocos de memória e outras estruturas embutidas: Basicamente, devem-se isolar os blocos de memória do restante do circuito a fim de facilitar o teste através da utilização de esquemas de testes desenvolvidos especificamente para estas estruturas.

Grandes circuitos combinacionais: Devido à grande complexidade dos geradores de teste e simuladores de falha necessários para realizarem o teste de grandes circuitos combinacionais aconselha-se, o particionamento dos mesmos, em estruturas menores e menos complexas a fim de aumentar sua testabilidade. A figura 3.2 ilustra claramente um exemplo de particionamento de um circuito.

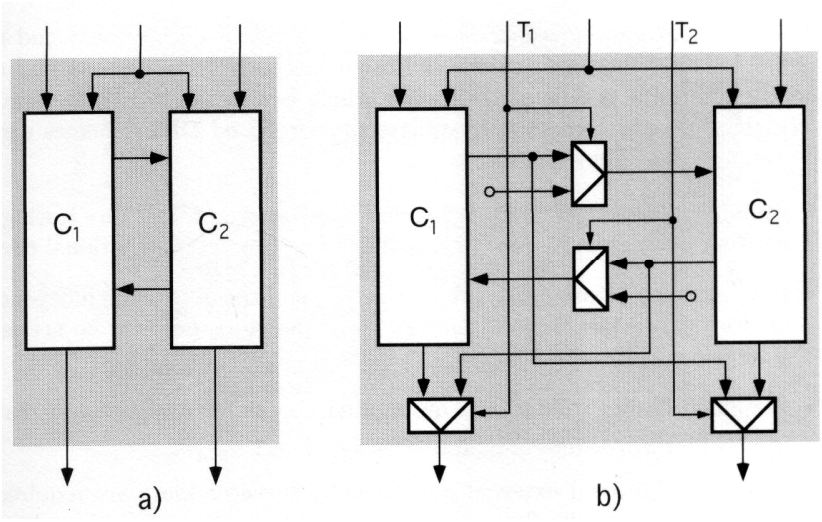


Figura 3.2 Particionamento de um Circuito. (RAJSKI, 1998)

3.2.3 Técnicas Estruturais

As técnicas estruturais representam uma concepção contrária à abordagem *Ad Hoc*, ou seja, representam um enfoque sistemático para o teste de circuitos mais complexos. Assim, essas técnicas superam as limitações observadas na abordagem *Ad Hoc* e viabilizam a automatização da fase de teste através de ferramentas de Automação de Projeto Eletrônico (*Electronic Design Automation* – EDA). Basicamente, as técnicas estruturais baseiam-se na idéia de controlar e observar diretamente todos ou a grande maioria das variáveis de estado (elementos de memória) presentes no circuito. Assim, um circuito seqüencial pode ser manipulado como se fosse um circuito combinacional e conseqüentemente ser avaliado de acordo com modelos de falhas combinacionais.

Dentre as várias técnicas estruturais propostas na literatura, a mais popular e difundida é a *Scan Design*.

A Técnica SCAN Design

A técnica *Scan Design* permite testar qualquer circuito seqüencial como um circuito combinacional, pois assegura total controlabilidade e observabilidade de todos os elementos de memória do circuito. Basicamente, esta técnica assume que durante o período de teste, todos os

registradores (*flip-flops* e *latches*) do circuito seqüencial estão conectados dentro de um ou mais registradores de deslocamento ou caminhos de *scan*. Assim, o circuito terá dois modos de operação, são eles:

- Modo Normal – neste modo de operação, os elementos de memória desempenham suas funções regulares, ou seja, é como se o circuito não tivesse sofrido nenhuma alteração;
- Modo Teste/*Scan* – neste modo de operação, todos os elementos de memória conectados dentro de um registrador de deslocamento são utilizados para *shift in* (ou *scan in*) e saída de dados de teste.

Assim, durante o processo de teste, a técnica de *Scan Design* executa a seguinte seqüência de operações:

1. Entra no modo de teste, ou seja, todos os elementos de memória formam um registrador de deslocamento;
2. Carrega os valores do padrão de teste nos *flip-flops*;
3. Seleciona os valores correspondentes nas entradas primárias;
4. Entra no modo normal;
5. Após os valores lógicos terem sido estabilizados, os valores de saída são verificados e captura subseqüentemente uma resposta de teste dentro dos *flip-flops*;
6. Entra no modo de teste, gera a saída e compara-as com os valores corretos de reposta. O próximo vetor de entrada pode ser carregado nos *flip-flops* neste mesmo instante;
7. Repete as operações de 2 a 6 para sucessivos vetores de teste.

Dentre as principais vantagens relacionadas ao esquema acima descrito, salienta-se fundamentalmente:

- Simplificação dos processos de geração e validação de padrões de teste – esta simplificação deve-se ao fato de que o teste da rede gerada a partir da técnica de *Scan* Design será exatamente igual ao teste realizado em circuitos combinacionais;
- Simplificação na análise do tempo – tendo em vista que a rede é totalmente independente as características do *clock*, seu funcionamento necessita apenas que um pulso seja ativado durante um período suficiente;

- Simplificação do processo de validação do projeto – a verificação das regras de projeto é realizada com uma certa facilidade devido à automatização do processo;
- Adição de poucos pinos extras – a técnica exige que sejam adicionados cerca de 3 pinos extras;
- Relativa facilidade na etapa de verificação do projeto – os caminhos gerados a partir da técnica permitem acesso direto a muitos nós internos do circuito.

Evidentemente, a técnica de *Scan Design* apresenta algumas limitações e agrega penalidades ao projeto. Dentre as principais limitações e penalidades podemos citar:

- Introdução de hardware adicional;
- Degradação do desempenho;
- Aumento no tempo dedicado ao teste.

A figura 3.3 mostra a arquitetura básica de um projeto *Scan*, onde é possível observar os três pinos adicionados ao projeto (*scan-in*, *scan-out* e *test mode*), a área acrescida devida ao hardware adicional e o *overhead* de desempenho gerado devido aos multiplexadores.

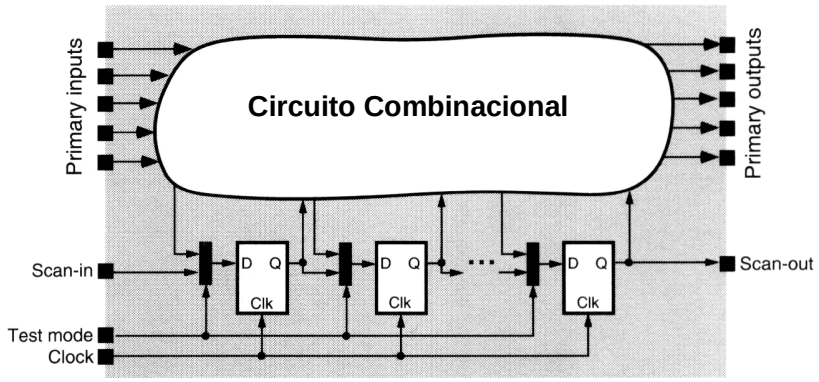


Figura 3.3 Arquitetura Básica de um Projeto Scan. (RAJSKI, 1998)

Segundo Janusz Rajski (RAJSKI, 1998), existem vários tipos *Scan Design*, dentre os quais salientam-se fundamentalmente o *Scan-Path*, o *Scan-Test*, o *Random-Access Scan* e o *Level-Sensitive Scan Design* - LSSD. Entretanto, segundo Paul H. Bardell (BARDELL, 1987) a técnica

mais difundida e bem documentada é a LSSD. Por isto, a técnica LSSD será brevemente definida e ilustrada.

A técnica Level-Sensitive Scan Design - LSSD

A técnica *Level-Sensitive Scan Design* (LSSD) utiliza elementos de memória que, por sua vez, são implementados como *latches* cujo conteúdo não pode ser alterado por nenhuma das entradas existentes se o *clock* estiver desligado. Quando se acrescenta a um determinado *latch* um latch extra que, por sua vez possui uma entrada de *clock* separada forma-se um *latch* registrador de deslocamento (*shift-register latch* - SRL).

A figura 3.4 mostra um modelo de circuito seqüencial LSSD onde se pode observar a separação da lógica combinacional dos elementos de armazenamento da estrutura LSSD. Basicamente, as entradas e saídas primárias e o circuito combinacional não sofre nenhuma mudança em sua estrutura. Entretanto, as estruturas de armazenamento são modificadas conforme acima descrito a fim de formarem os SRL. O acesso ao teste nos SRLs é feito através da entrada primária *scan-in* e da saída primária *scan-out*.

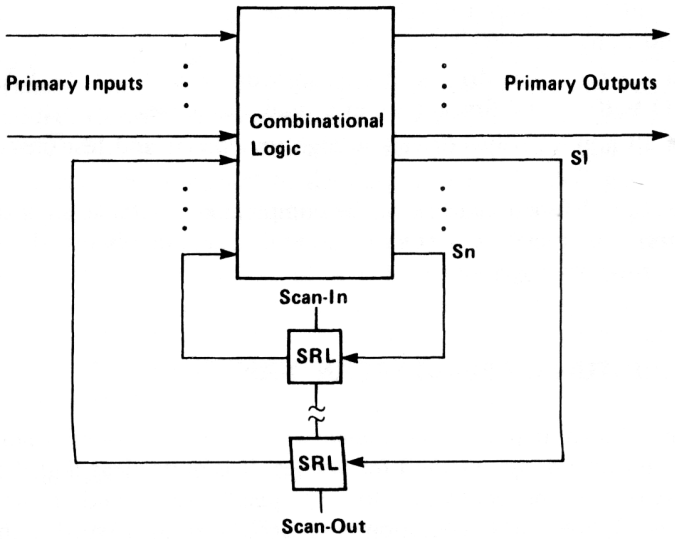


Figura 3.4 Modelo de Circuito Seqüencial LSSD. (BARDEL, 1987)

Já a figura 3.5 mostra a forma geral de um *polarity-hold* SRL. Durante a operação normal do sistema, os *clocks shift* A e *shift* B estão desligados e o SRL possui apenas dois sinais de

entrada, o *system date* D e o *system clock* C, que ativa a função de memória *latch*. Quando o *clock* C está ativo, o estado interno de L1 é carregado com o valor da entrada D.

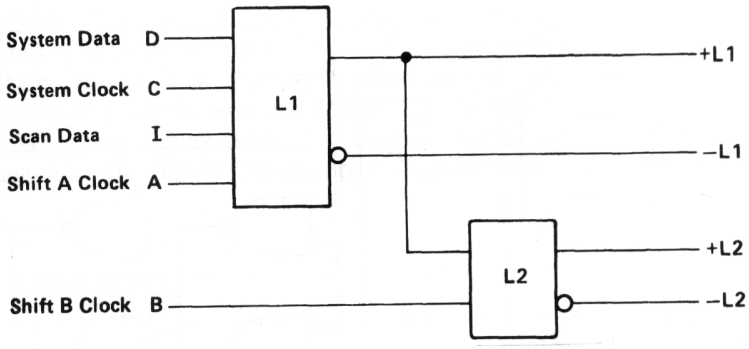


Figura 3.5 Forma Geral de um Circuito SRL polarity-hold. (BARDEL 1987)

E finalmente, a figura 3.6 mostra a interconexão de SRLs, que por sua vez, é feita conectando-se a saída +L2 a entrada do próximo SRL e todas as entradas aos dois *clocks* em paralelo.

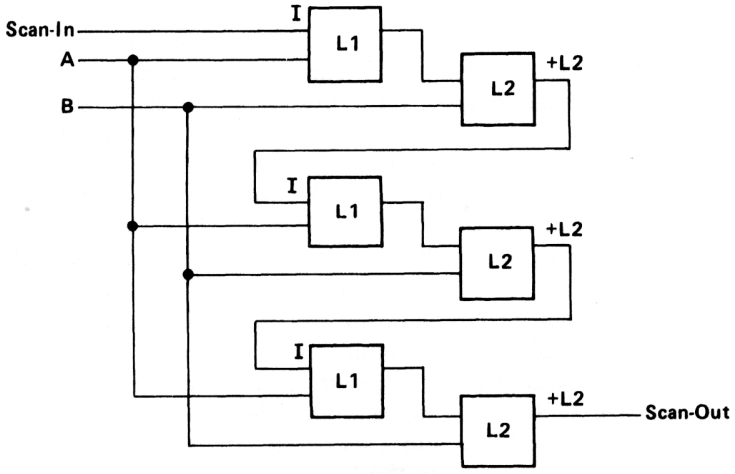


Figura 3.6 Interconexão de SRLs (BARDEL,1987)

Assim, existem basicamente duas estruturas que utilizam o esquema LSSD, são elas: (1) projeto *single-latch* e (2) projeto *double-latch* ilustradas respectivamente nas figuras 3.7 e 3.8.

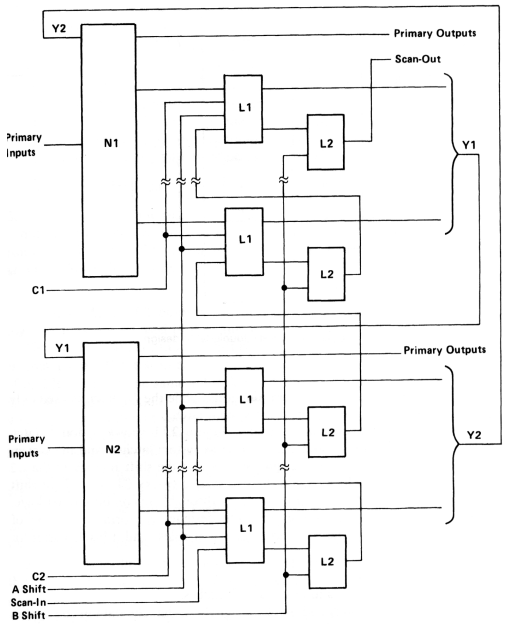


Figura 3.7 Projeto de um LSSD *single-latch*. (BARDEL 1987)

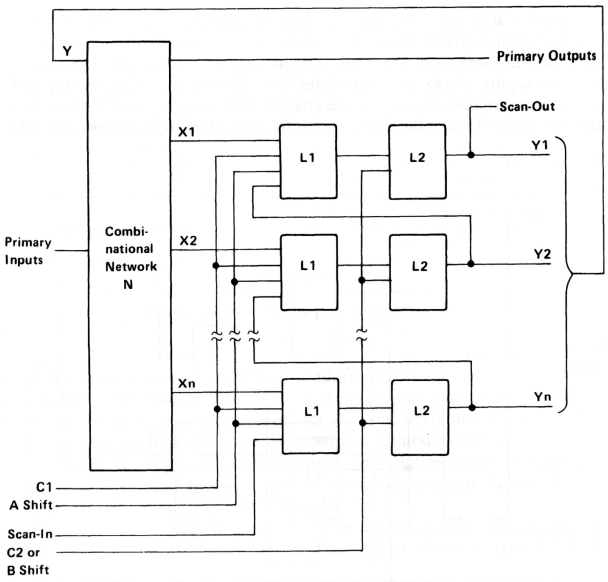


Figura 3.8 Projeto de um LSSD *double-lathc*. (BARDEL 1987)

Quando diante de um projeto estrutural, o projetista deve levar em consideração uma série de regras e normas de projeto. Especificamente, diante da técnica LSSD as quatro regras abaixo citadas devem ser obedecidas:

- Todo armazenamento interno é feito em um *clocked latches* DC;
- Os *latches* são controlados em ciclos de *clock* distintos (*clocks nonoverlapping*);
- Um determinado *latch* X pode alimentar o *latch* Y se e somente se o *clock* que alimenta X não é o mesmo que alimenta Y e se os *clocks* são distintos;
- Todos os *latches* estão contidos em um *latch* registrador de deslocamento e estes estarão todos interconectados em um ou mais registradores de deslocamento.

A Técnica Boundary Scan

Em 1990 o IEEE adotou um novo padrão de teste denominado IEEE Standard 1149.1 (IEEE *Standard Test Access Port and Boundary-Scan Architecture*). A técnica *Boundary Scan* é agregada em nível de placa de circuito impresso e consiste na extensão dos *scan path* internos a placa de circuito impresso para sua interface e na utilização de um protocolo que possui várias funções de teste que devem ser executadas.

A sua arquitetura é composta de um registrador de instruções, registradores de dados de teste, uma porta de acesso à infra-estrutura de teste (*test access port* - TAP) e seu controlador. A porta TAP é composta de 4 pinos, são eles: 1. Entrada de dados de teste (*Test Data Input* - TDI); 2. Saída dos dados de teste (*Test Data Output* - TDO); 3. Seleção do modo de teste (*Test Mode Select* - TMS) e 4. *Clock* do teste (*Test Clock* - TCK). O controlador TAP é uma máquina de estados finitos composta de 16 estados, responsáveis pela seleção de registradores e de operações de teste, pela captura, pelo deslocamento (com ou sem pausa) e atualização de instruções e dados de teste, pelo *reset* da lógica de teste e pela eventual execução do procedimento de auto-teste integrado.

Configurando-se corretamente os modos de operação dos multiplexadores das células *boundary scan* podem ser realizados três tipos diferentes de testes, são eles:

- Teste externo – baseado no controle das inter-conexões das placas;
- Teste interno – baseado no controle das entradas e observação das saídas dos circuitos integrados;
- Teste de amostragem – baseado na observação das entradas e saídas dos circuitos integrados.

A figura 3.9 mostra o caminho *boundary scan* ideal em uma placa.

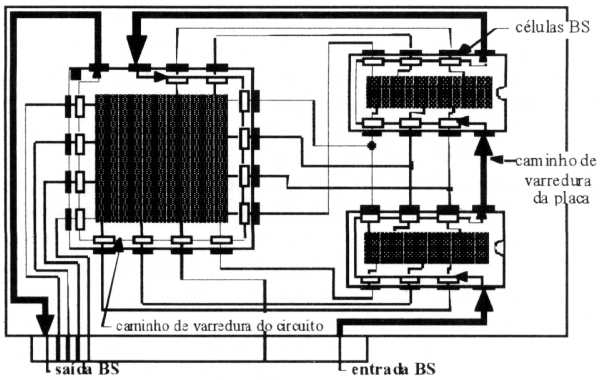


Figura 3.9 Caminho *Boundary-scan*. (REIS, 2000).

E finalmente, a figura 3.10 mostra o padrão de teste para portas de acesso denominado IEEE 1149.1.

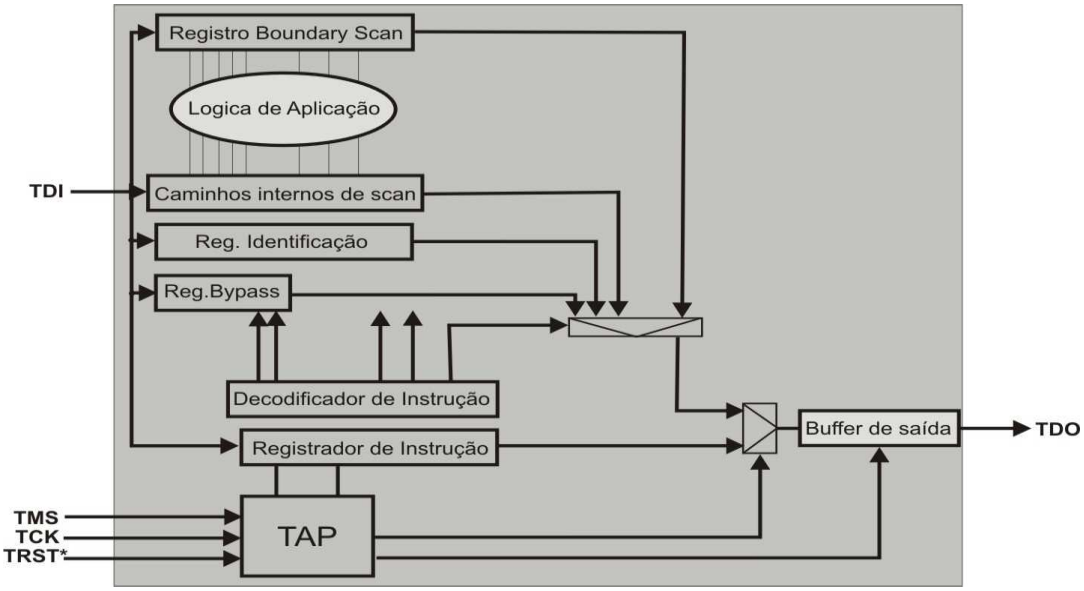


Figura 3.10 Padrão de teste IEEE 1149.1. (RAJSKI, 1998).

Em vista do exposto, a técnica acima descrita além de detectar falhas em nível de placa, módulo ou sistema é capaz de gerar facilmente um ambiente de diagnóstico e localização de falha.

4. AUTO-TESTE EMBUTIDO

4.1 Introdução

Testar um circuito integrado consiste basicamente em aplicar determinados estímulos em suas entradas e, após o processamento, comparar as respostas obtidas com o conjunto de repostas esperadas. Tradicionalmente, o teste de circuitos integrados era realizado através de ATEs. Entretanto, diante do aumento da densidade dos CIs, a utilização dos mesmos tornou-se impraticável e inviável. Esta inviabilidade deve-se fundamentalmente a 5 fatores, são eles: (1) a dificuldade na geração dos padrões de teste; (2) a grande probabilidade do número de vetores de teste atingir proporções impraticáveis e conseqüentemente impossíveis de serem armazenadas e manipuladas pelo HW do testador; (3) a necessidade de muito tempo para aplicar todos os vetores de teste; (4) ao custo associado aos ATEs e (5) a impossibilidade de utilizar ATEs quando os circuitos encontram-se inseridos em um sistema.

Neste contexto, uma solução bastante viável e cada vez mais utilizada é o Auto-Teste Embutido (*Built-in Self-Test* - BIST). No BIST circuitos extras são adicionados ao CI para desempenharem novas funcionalidades, ou seja, para gerarem os vetores de teste de acordo com um determinado padrão de teste, avaliarem as respostas obtidas e por fim, controlarem o teste. Salienta-se que no decorrer deste capítulo serão apresentadas e definidas as diferentes metodologias para a geração de padrões de teste e de compactação das respostas do teste.

Assim, BIST baseia-se na idéia de projetar um circuito que seja capaz de se auto-testar a fim de verificar a presença de erros, ou seja, o próprio circuito é capaz de verificar se ele está bom ou ruim.

Segundo Charles E. Stroud (STROUD, 2002), BIST é qualquer método de testar um CI que utilize circuitos especiais projetados dentro do CI. Este circuito desempenha funções de teste e gera sinais de controle para indicarem se as partes do CI cobertas pelos circuitos BIST estão trabalhando propriamente.

Indubitavelmente, o BIST revolucionou o teste de CIs, pois apresenta uma série de vantagens em relação a metodologias de testes convencionais anteriormente mencionadas e definidas no capítulo anterior. Dentre as principais vantagens, salientam-se:

- Redução no custo associado ao teste de manufatura - esta redução no custo está associada a diminuição do tempo necessário para aplicação do teste e na quantidade de dados de teste que devem ser armazenados e a eliminação do alto custo relacionado a utilização dos ATEs.
- Redução no ciclo e no custo de desenvolvimento e manutenção do sistema – esta redução é atribuída à propriedade de testabilidade vertical que o BIST possui, ou seja, o BIST pode ser aplicado em diferentes níveis hierárquicos (*chips*, placas e sistemas).
- Aumento na velocidade de execução do teste – este aumento deve-se ao fato de que o circuito BIST executa seu teste com o mesmo *clock* do CUT.

Evidentemente, o BIST também apresenta algumas penalidades, dentre as quais salientam-se:

- Aumento da área de silício, ou seja, *overhead* de área;
- Possível degradação no desempenho devido à presença de multiplexadores necessários para aplicação dos padrões de teste no circuito;
- Aumento da rigidez durante o projeto.

A tabela 4.1 abaixo apresenta um resumo das principais vantagens e desvantagens do BIST.

Vantagens	**Desvantagens**
Testabilidade vertical (*wafer* para sistema)	Overhead de área
Alta resolução de diagnóstico	Penalidades de desempenho
Teste em alta velocidade	Tempo e esforço de projeto adicional
Redução na necessidade de testadores externos	Risco adicional para o projeto
Redução no tempo e esforço dedicado ao desenvolvimento do teste	
Teste de exaustão mais econômico	
Redução no tempo e custo do teste de manufatura	
Redução do *time-to-market*	

Tabela 4.1 Resumo de vantagens e desvantagens do BIST. (STROUD, 2002)

Basicamente, espera-se que o circuito BIST agregado ao CI gere uma elevada cobertura de falhas, seja executado em um curto intervalo de tempo, possua um pequeno volume de dados de teste e seja compatível com a metodologia de DFT assumida.

4.2 Arquitetura Básica do BIST

A figura 4.1 mostra o diagrama de bloco da arquitetura básica de um circuito de BIST agregado ao CUT. Basicamente, a arquitetura do BIST inclui duas funções essenciais e duas funções adicionais que facilitam a execução do auto-teste em nível de sistema. Dentre as funções essenciais salienta-se o gerador de padrão de teste (*test pattern generation* - TPG), que gera os padrões de teste a serem aplicados no CUT, e o analisador das respostas da saída (*output response analyzer* – ORA), que compacta as respostas obtidas a partir da aplicação do teste no CUT e indica se o circuito apresenta ou não uma falha. Em relação as outras duas funções adicionais, inclui-se um circuito para controlar o BIST e um circuito para isolar as entradas. Salienta-se que além das quatro funções acima definidas, o BIST pode eventualmente exigir a inserção de pinos extras de entrada e saída para ativarem sua execução.

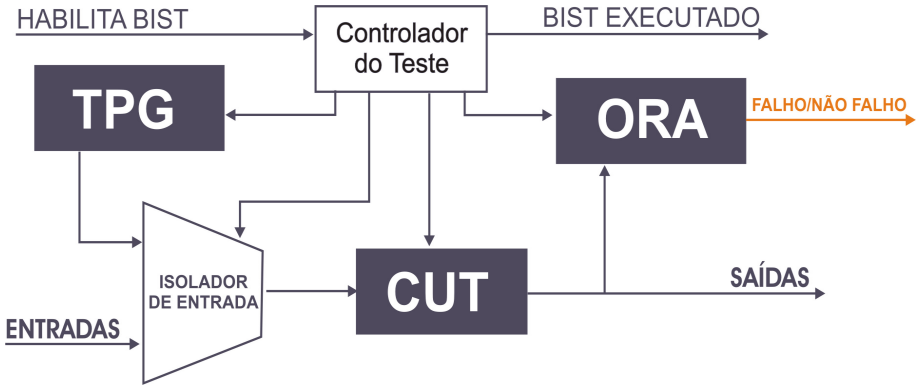


Figura 4.1 Arquitetura básica do BIST. (STROUD, 2002)

Segundo Paul H. Bardell (BARDELL, 1987), o BIST pode ser classificado como concorrente ou não-concorrente. Quando classificado como concorrente, o teste é *on-line* e utiliza algum tipo de redundância. Entretanto, quando classificado como não-concorrente, o teste é off-line e basicamente verifica a integridade estrutural e funcional do circuito. Em resumo, um BIST concorrente agrega mecanismos de detecção e correção de erros, circuitos totalmente auto-testáveis, auto-verificação e outros. Salienta-se que estas técnicas serão detalhadamente descritas no capítulo 5. Finalmente, um BIST não-concorrente exige que sejam agregados ao circuito um gerador de padrões de teste (TPG), um analisador das respostas de saída (ORA) e um controlador

de teste. Salienta-se que no decorrer deste capítulo serão descritos detalhadamente os diferentes tipos de TPGs e ORAs.

Além da classificação acima descrita, segundo (STROUD, 2002) o circuito do BIST pode ser centralizado ou distribuído. O esquema centralizado, também chamado de compartilhado, é bastante eficiente em termos de *overhead* de área, mas requer múltiplas execuções da seqüência de teste. Assim, cada CUT deve ser roteado independentemente, através de um circuito adicional (multiplexador), para o ORA. Entretanto, no esquema distribuído, cada CUT possui seu TPG e seu ORA. Apesar da independência dos circuitos relacionados à implementação do BIST permitir a execução paralela do teste, ela também aumenta significativamente o *overhead* de área. A figura 4.2 ilustra as arquiteturas centralizada e distribuída.

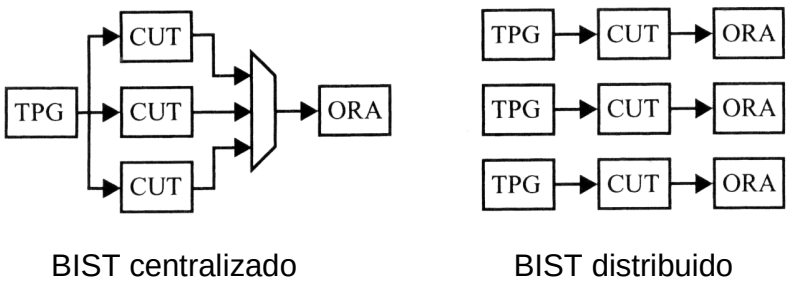


Figura 4.2 Arquiteturas BIST centralizada e distribuída. (STROUD, 2002)

A arquitetura BIST também pode ser classificada em *embedded* ou *separate*. Quando as funções TPG e ORA são implementadas utilizando-se os *flip-flops* e registradores existentes no CUT, a arquitetura é referenciada como *embedded*. Entretanto, quando o circuito que executa essas funções é adicionado ao CUT, ela recebe o nome de *separate*. As metodologias embedded são também denominadas como *intrusive* ou *invasive* e as *separate* com *non-intrusive* ou *non-invasive.*

E por fim, as arquiteturas BIST podem ser classificadas de acordo com a aplicação dos padrões de teste no CUT. Quando um novo padrão de teste é aplicado a cada ciclo de *clock*, ela é referenciada como um sistema BIST do tipo *test-per-clock* ou paralelo. Entretanto, quando um mecanismo de *scan* é utilizado para aplicar os padrões de teste e retirar as respostas de saída do CUT, o BIST é classificado como *test-per-scan* ou serial. A figura 4.3 ilustra claramente as duas arquiteturas acima definidas.

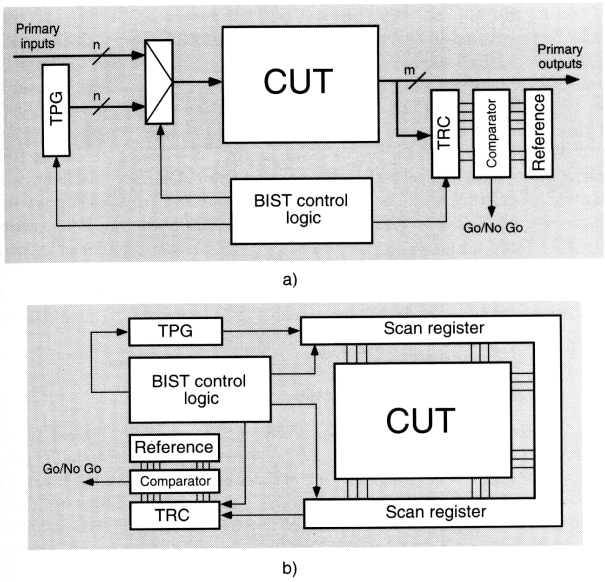


Figura 4.3 Arquiteturas BIST do tipo *test-per-clock* e *test-per-scan*. (RAJSKI, 1998)

4.3 Geradores de Padrões de Teste

Dentre os principais componentes que compõem a arquitetura BIST, salienta-se o gerador de padrões de teste. Esta importância justifica-se, pois os vetores de teste gerados pelo TPG influenciam diretamente na cobertura de falhas obtida para um determinado modelo de falha. Por isto, a geração de vetores de teste para aplicações BIST deve estar de acordo com duas exigências básicas: (1) devem ser capazes de obter uma alta cobertura de falhas e (2) o hardware necessário para sua implementação deve ser o menor possível e o mais facilmente agregado ao chip.

Dentro da literatura existe uma grande variedade de padrões de teste, os quais serão apresentados e definidos a seguir:

A) Padrão de teste exaustivo:

Este padrão caracteriza-se por produzir todas as 2^n combinações possíveis de entradas para um determinado circuito que apresenta n entradas. O padrão exaustivo garante, sem a necessidade de simulação de falhas, a detecção de todas as falhas *stuck-at* em nível de portas lógicas e falhas *wired*-AND/OR e *dominant bridging*. Entretanto, isto não ocorre diante de falhas de transistor e de *delay* pois, estes tipos de falhas exigem um ordenamento específico dos vetores e a repetição

de alguns deles. Salienta-se que esta metodologia é inviável diante de circuitos que possuem um número relativamente grande de entradas em termos de tempo de execução do teste. Quanto a estrutura de hardware utilizada para implementar este padrão de teste, utiliza-se normalmente um contador binário de *n*-bits, onde n representa o número de entradas do circuito alvo.

B) Padrão de teste pseudo-exaustivo:

O teste pseudo-exaustivo baseia-se no padrão de teste anteriormente definido e consiste basicamente em particionar um determinado circuito de n entradas em sub-circuitos com k entradas e testá-los exaustivamente. Esta modificação garante a mesma cobertura de falhas do padrão anterior e reduz significativamente o tempo de aplicação do teste. A redução observada justifica-se devido a diminuição do número de entradas, ou seja, necessariamente k é menor que n e conseqüentemente o número de vetores de teste 2^k . Salienta-se que o desempenho desta metodologia depende diretamente da técnica de partição ou segmentação. Quanto a estrutura de hardware utilizada para implementar este padrão de teste, normalmente utilizam-se contadores binários, registradores de deslocamento de *feedback* linear (*linear feedback shift register* – LFSR) ou ainda (*celullar automata* – CA).

C) Padrão de teste determinístico:

Este padrão de teste, também conhecido como padrão de teste armazenado, é utilizado para detectar falhas ou defeitos estruturais específicas de um determinado CUT. Quanto ao hardware utilizado para sua implementação, ele baseia-se na utilização de uma memória somente de leitura (*read only memory* – ROM) endereçada através de um contador.

D) Padrão de teste algorítmico:

Este padrão é bastante semelhante ao padrão anterior sendo utilizado no BIST de estruturas regulares como memórias de acesso randômico (*randon access memory* – RAM). Entretanto, neste padrão de teste utiliza-se máquinas de estados finitos (*finite status machine* – FSM) na sua implementação.

E) Padrão de teste randômico:

Este padrão é normalmente utilizado para teste funcional externo de microprocessadores. Entretanto, os padrões gerados a partir do mesmo são pouco utilizados em aplicações BIST devido à incapacidade de repeti-los, ou seja, a cobertura de falhas será sempre diferente de uma execução para outra da seqüência de vetores de teste.

F) Padrão de teste pseudo-randômico:

Este padrão de teste é o mais utilizado para a geração de vetores de teste em aplicações BIST. Quanto aos padrões produzidos, eles apresentam propriedades semelhantes ao esquema randômico, mas com uma grande vantagem, os padrões podem ser repetidos. Quanto ao hardware utilizado para implementá-lo, normalmente utiliza-se LFSR ou CA.

4.4 As Estruturas de Hardware

Assim, os padrões de teste acima descritos são produzidos a partir de algumas estruturas de hardware pré-definidas, tais como:

A)Contadores:

Apesar de ser uma estrutura pouco utilizada como TPG, um contador é capaz de testar completamente um circuito combinacional de n entradas, pois geram todos os 2^n possíveis padrões de teste. Entretanto, quando utilizados, ajudam a diminuir a área de *overhead* do BIST, pois normalmente já estão presentes nos circuitos desempenhando funções normais dos sistemas.

B) Máquinas de estados finitos:

A estrutura FSM é normalmente utilizada juntamente com contadores para gerar padrões de teste algorítmicos. A figura 4.4 mostra um exemplo de um TPG que utiliza um FSM em conjunto com um contador para testar uma RAM de acordo com o algoritmo *March*.

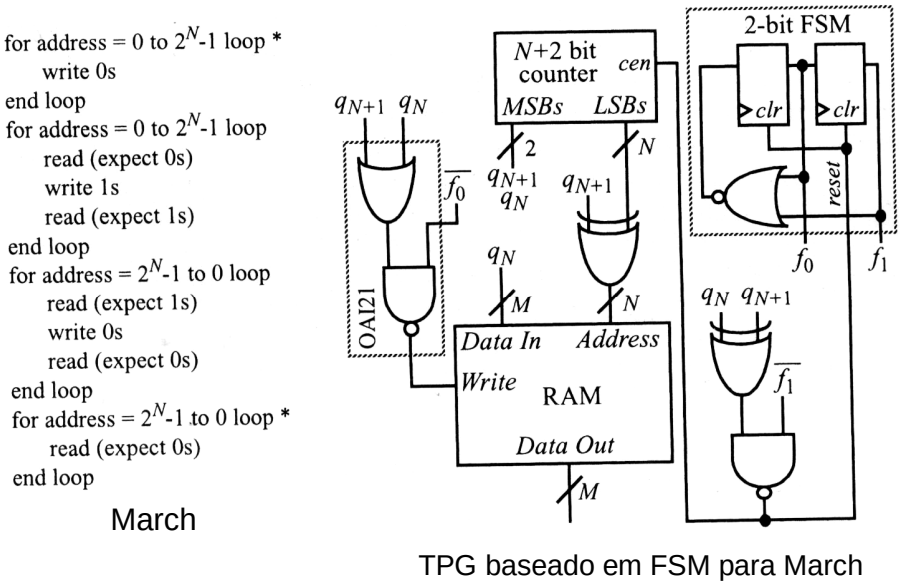


Figura 4.4 Exemplo de um TPG baseado em um FSM. (STROUD, 2002)

C) Linear Feedback Shift Register:

O LFSR é a estrutura de hardware mais utilizada para a implementação de TPG em aplicações BIST e consiste em uma rede de *flip-flops* e portas lógicas do tipo OU-exclusivo. Basicamente, existem dois tipos de implementações: (1) interna, utilizada em aplicações de alto desempenho que exigem uma alta freqüência de operação máxima e (2) externa utilizada em aplicações que priorizam a uniformidade dos registradores de deslocamento. A figura 4.5 mostra as duas implementações acima descritas.

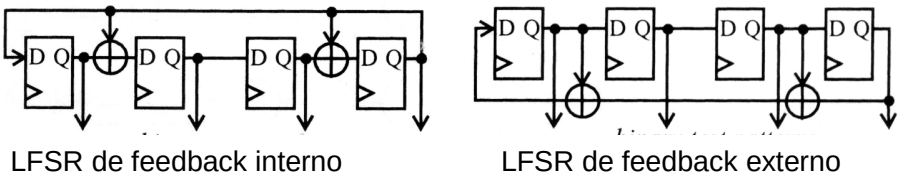
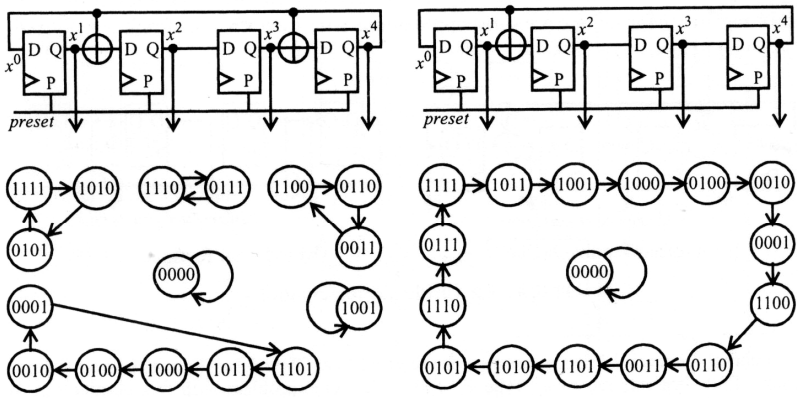


Figura 4.5 Implementações de LFSRs. (STROUD, 2002)

A seqüência de teste produzida pelo LFSR é intimamente relacionada com a disposição das portas OU-exclusivo na rede de *feedback*. A figura 4.6 mostra duas seqüências de teste geradas a

partir da implementação de dois LFSRs. Observe que na figura 4.6a, são produzidos 6 padrões de teste diferentes antes da repetição dos mesmos.



$P(x) = x^4 + x^3 + x + 1$ $P(x) = x^4 + x + 1$

Figura 4.6 Seqüência de padrões de teste produzidos pelos LFSR.

Assim, a seqüência mais longa de padrões de teste, antes de sua repetição, para um LFSR de *n*-bits será $2^n - 1$ padrões. Quanto a sua construção, as portas tipo OU-exclusivo são posicionadas de acordo com o polinômio característico - P(x), o número de *flip-flops* resultam do grau do polinômio onde cada coeficiente não zero representa uma porta OU-exclusivo na rede de *flip-flops*. Esta regra não é válida para os coeficientes x^n e x^0 que sempre são diferentes de zero mas não representam a agregação de uma porta OU-exclusivo.

Existe um tipo específico de polinômio, denominado polinômio primitivo que quando utilizado como base para a construção do LFSR garante a geração de uma seqüência de teste com comprimento máximo. A figura 4.7 mostra a seqüência de teste gerada por um LFSR externo baseado em um polinômio primitivo.

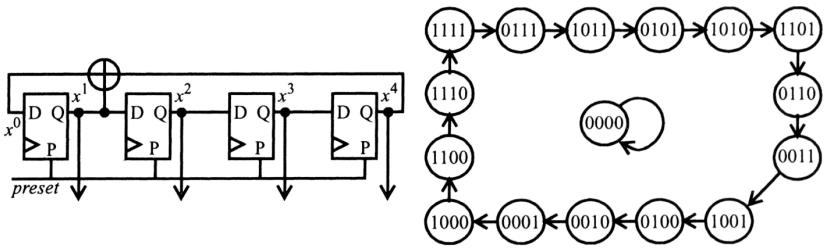


Figura 4.7 Seqüência de teste gerada por um LFSR externo com polinômio primitivo.
(STROUD, 2002)

Assim, um polinômio característico ideal é aquele que possui poucos coeficientes não zeros. A tabela 4.2 mostra uma lista de polinômios primitivos.

Grau(n)	Polinomio	Grau(n)	Polinomio
2, 3, 4, 6, 7, 15, 22, 60, 63	$x^n + x + 1$	12	$x^n + x^6 + x^4 + x^3 + 1$
5, 11, 21, 29, 35	$x^n + x^2 + 1$	33	$x^n + x^{13} + 1$
8, 19, 38, 43	$x^n + x^6 + x^5 + x + 1$	34	$x^n + x^{15} + x^{14} + x + 1$
9, 39	$x^n + x^4 + 1$	36	$x^n + x^{11} + 1$
10, 17, 20, 25, 28, 31, 41, 52	$x^n + x^3 + 1$	37	$x^n + x^{12} + x^{10} + x^2 + 1$
13, 24, 45, 64	$x^n + x^4 + x^3 + x + 1$	40	$x^n + x^{21} + x^{19} + x^2 + 1$
14, 16	$x^n + x^5 + x^4 + x^3 + 1$	42	$x^n + x^{23} + x^{22} + x + 1$
18, 57	$x^n + x^7 + 1$	46	$x^n + x^{21} + x^{20} + x + 1$
23, 47	$x^n + x^5 + 1$	54	$x^n + x^{37} + x^{36} + x + 1$
26, 27	$x^n + x^{12} + x^{11} + x + 1$	55	$x^n + x^{24} + 1$
30, 51, 53, 61, 70	$x^n + x^{16} + x^{15} + x + 1$	58	$x^n + x^{19} + 1$
32, 48	$x^n + x^{28} + x^{27} + x + 1$	65	$x^n + x^{18} + 1$
44, 50	$x^n + x^{27} + x^{26} + x + 1$	69	$x^n + x^{29} + x^{27} + x^2 + 1$
49, 68	$x^n + x^9 + 1$	71	$x^n + x^6 + 1$
56, 59	$x^n + x^{22} + x^{21} + x + 1$	72	$x^n + x^{53} + x^{47} + x^6 + 1$
66, 67, 74	$x^n + x^{10} + x^9 + x + 1$	73	$x^n + x^{25} + 1$

Tabela 4.2 Lista de polinômios primitivos. (STROUD, 2002)

D) Cellular Automata:

Assim como o LFSR, o *Cellular Automata* - CA produz uma seqüência de padrões de teste pseudo-randômicos. A grande vantagem de CA em relação à estrutura descrita anteriormente é que o efeito dos bits deslocando não é observado. A figura 4.8 ilustra claramente a diferença entre a seqüência de teste gerada a partir destas duas estruturas de hardware. Os quadros cinzas representam lógica 1 e os brancos lógica 0.

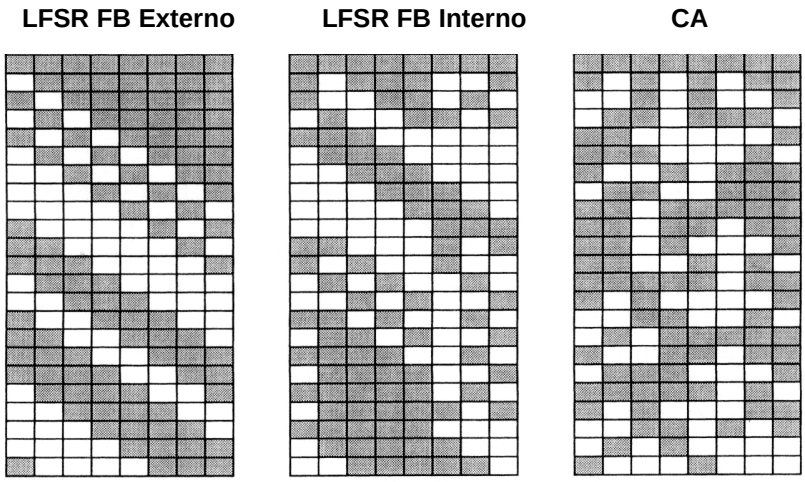


Figura 4.8 Comparação entre os padrões de teste gerados por LFSR e CA. (STROUD, 2002)

Quanto a sua construção, um registrador CA é baseado na relação lógica dos dois *flip-flops* vizinhos. A relação lógica existente entre os *flip-flops* é referenciada como "regras" e dentre elas, as mais utilizadas são as regras 90 e 150. A tabela 4.3 mostra as regras 90 e 150 para implementação de CA.

		7	6	5	4	3	2	1	0
	$x_{i-1}(t)\ x_i(t)\ x_{i+1}(t)$	111	110	101	100	011	010	001	000
regra 90	$x_i(t+1)$	0	1	0	1	1	0	1	0
	valor = 90		2^6		2^4	2^3		2^1	
regra 150	$x_i(t+1)$	1	0	0	1	0	1	1	0
	valor = 150	2^7			2^4		2^2	2^1	

Tabela 4.3 Regras 90 e 150. (STROUD, 2002)

Salienta-se que o nome da regra é obtido a partir do valor decimal do código binário gerado para o próximo estado do flip-flop *xi* com base no estado corrente e nos dois estados correntes dos dois vizinhos mais próximos, ou seja, *xi*+1 e *xi*-1.

A figura 4.9 ilustra claramente a implementação de um CA de 6-bits.

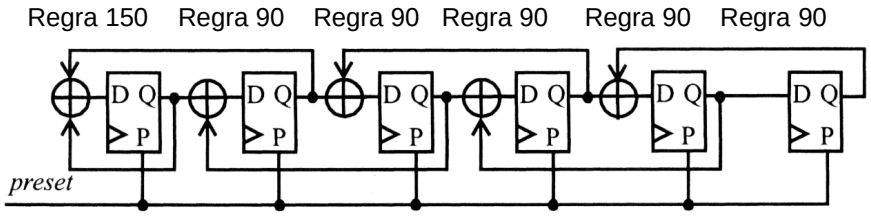


Figura 4.9 Exemplo de implementação de um CA de 6-bits. (STROUD, 2002)

4.5 Analisador das Respostas da Saída

A comparação bit-a-bit das respostas do CUT não é uma solução viável, pois exigiria do hardware uma grande capacidade de armazenamento. Por isto, existem alguns esquemas de compactação de respostas que basicamente geram uma assinatura do CUT e posteriormente a comparam com um padrão correto pré-armazenado. Neste contexto, surge o que chamamos de analisador das respostas da saída (*output response analyser* – ORA) que basicamente compacta as saídas do CUT, geradas a partir da aplicação dos padrões de teste gerados pelo TPG e indica o status *pass/fail* no final a execução do BIST. Assim, o objetivo do compactador das respostas de teste é reduzir o volume de dados gerados durante o procedimento de teste. Esses dados são compactados em uma assinatura que apresenta a capacidade de indicar se um circuito está ou não livre de falhas. Evidentemente, que todos os esquemas de compactação tendem a perder informação e por isso geram o que chamamos de *mascaramento de falha*. Segundo Paul H. Bardel (BARDELL, 1987), mascaramento de falha é medido pela probabilidade de um circuito falho produzir a mesma assinatura de um circuito livre de falha. Salienta-se que muitas vezes esta probabilidade é utilizada como critério de eficiência para ORAs.

Portanto, segundo Janusz Rajski (RAJSKI, 1998), um compactador ideal deve possuir as seguintes propriedades:

- O algoritmo de compactação deve ser facilmente implementado e agregado ao circuito BIST;
- A implementação não deve apresentar limites quanto ao tempo de teste;
- O algoritmo de compactação deve ser capaz de gerar um algoritmo de compressão dos dados de teste que minimizem significativamente o tamanho da assinatura;
- A técnica de compactação não deve perder informação referente às falhas.

Existem vários tipos de circuitos ORAs, os quais são classificados de acordo com a metodologia utilizada para realizar a compactação dos dados de teste. Assim, as principais metodologias de compactação presentes na literatura são:

A) Concentradores:

Esta técnica compacta os dados de múltiplas saídas de um CUT em um único dado e conseqüentemente reduz o número de saídas que devem ser monitoradas durante a seqüência de teste. Apesar de não ser considerada uma típica técnica ORA, os concentradores são muito utilizados em conjunto com outras metodologias ORAs a fim de reduzir o *overhead* de área gerada pelo BIST. Quanto à estrutura de hardware, esta técnica é implementada a partir da construção de uma árvore de portas lógicas do tipo OU-exclusivo. A figura 4.10 mostra um exemplo de um concentrador de 8-bits.

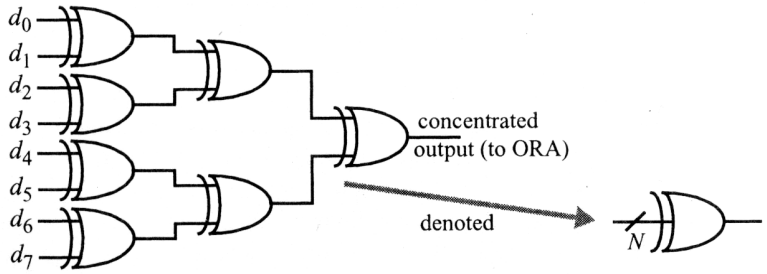


Figura 4.10 Exemplo de um concentrador de 8-bits.

Salienta-se que o uso de concentradores em aplicações de BIST pode mascarar falhas de um determinado CUT quando os erros afetarem um número par de bits de saída dos circuito.

B) Comparadores:

A figura 4.11 exemplifica o uso desta metodologia, para um projeto BIST, onde as respostas esperadas são armazenadas em uma ROM e posteriormente comparadas com as respostas geradas pelo CUT.

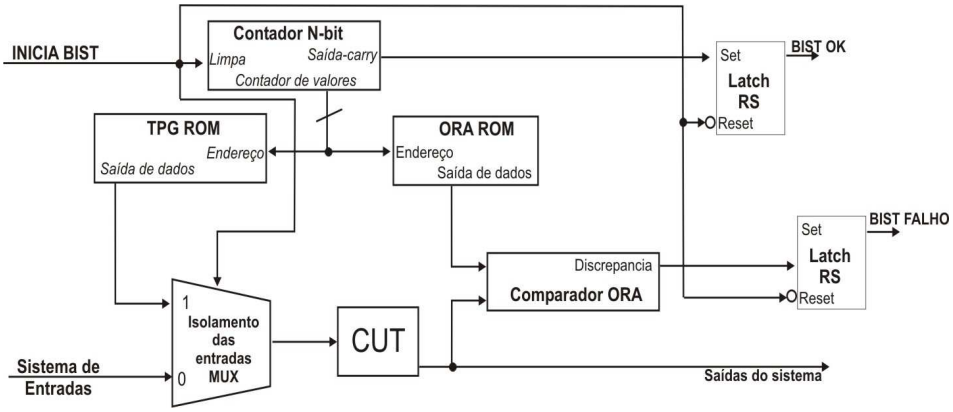


Figura 4.11 Exemplo de uma arquitetura BIST com comparador. (STROUD, 2002)

Salienta-se que devido ao elevado *overhead* de área agregado ao circuito e à exigência de um monitoramento contínuo da execução dos vetores de teste, esta técnica não é muito utilizada. Entretanto, a incorporação de um RS *latch* para armazenar eventuais discrepâncias durante a execução da seqüência de teste, exclui a necessidade do monitoramento contínuo.

Dentre as aplicações, salientam-se CUTs compostos por muitos circuitos idênticos. Neste caso, deve-se agregar um comparador para verificar as respostas obtidas a partir da aplicação dos vetores de teste e identificar eventuais discrepâncias.

C) Contador:

Esta técnica compacta a saída de um determinado CUT através da contagem do número de 1´s ou 0´s, ou seja, uma assinatura é gerada a partir do número de transições de 1 para 0 ou de 0 para 1. Assim, no final da execução do BIST, a assinatura obtida em tempo de execução será comparada com o valor esperado para o CUT. Quanto à estrutura de hardware, utiliza-se um contador para cada saída. Uma implementação híbrida que reduz o número de contadores que devem ser adicionados ao CUT é sugerida na figura 4.12. Nesta implementação, utiliza-se apenas um contador pois um concentrador é agregado às saídas do CUT.

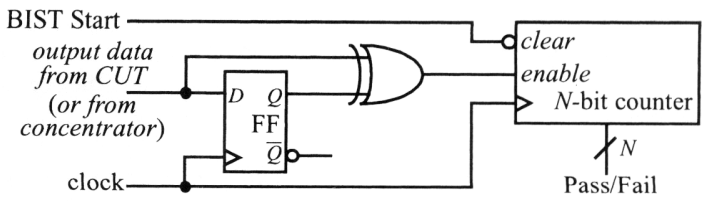


Figura 4.12 Contador de transições. (STROUD, 2002)

D) Análise de Assinatura:

A análise de assinatura consiste basicamente em dividir o polinômio de dados, ou seja, a saída do CUT, pelo polinômio característico do LFSR. O resultado obtido a partir desta divisão será a assinatura utilizada para determinar se o circuito está ou não livre de falhas. Assim, para que se possa utilizar o LFSR como dispositivo de compactação, ele deve ser modificado a fim de aceitar uma entrada externa e atuar como divisor polinomial. Assim, o circuito receberá continuamente um dado de entrada, no caso a saída do CUT. A figura 4.13a ilustra a modificação, previamente descrita, que deve ser realizada na implementação do LFSR. Além da implementação anterior, uma alternativa é mostrada na figura 4.13b. A seqüência de entrada, representada por um polinômio, é dividida pelo polinômio característico do LFSR. Assim, a divisão procede da seguinte forma: a seqüência quociente aparece na saída do LFSR e o resto é mantido no LFSR. Quando o teste termina, o LFSR conterá a assinatura do CUT.

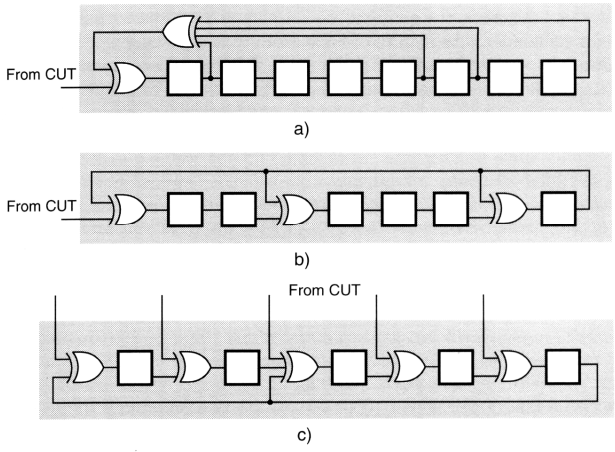


Figura 4.13 LFSR e MISR como compactadores. (RAJSKI, 1998)

Este processo é claramente ilustrado na figura 4.14 onde um registrador de análise de assinatura (*signature analysis register* – SAR) é construído a partir de um LFSR interno com um polinômio característico da Tabela 4.3.

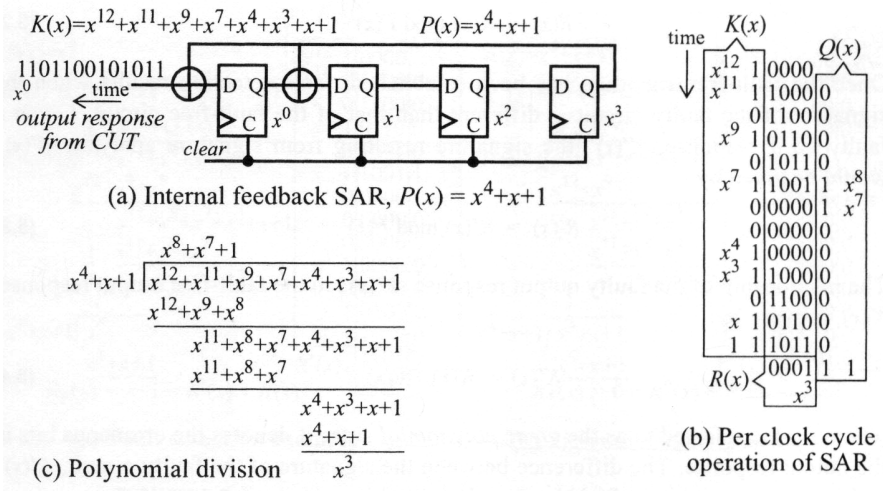


Figura 4.14 Exemplo de análise de assinatura. (STROUD, 2002)

Assim, o processo de divisão polinomial do SAR é dado pela equação abaixo:

$k(x) = Q(x)P(x) + R(x)$, também denotada como $R(x) = K(x) \bmod P(x)$ (4.1)

Onde: *K(x)* representa a saída do CUT, *Q(x)* o quociente obtido a partir da divisão de *K(x)* pelo polinômio característico, *P(x)* o polinômio característico do LFSR e *R(x)* o resto da divisão, ou seja, a assinatura obtida ao final da seqüência de teste.

Quando ocorre uma falha, ou seja, a resposta do CUT (*K´(x)*) estiver errada, a assinatura resultante será *R´(x)* e será:

$R'(x) = K'(x) \bmod P(x)$ (4.2)

Assim, a relação entre a resposta falha (*K´(x)*) e a resposta do circuito livre de falha (*K(x)*) é dada por:

$K'(x) = K(x) + E(x)$ (4.3)

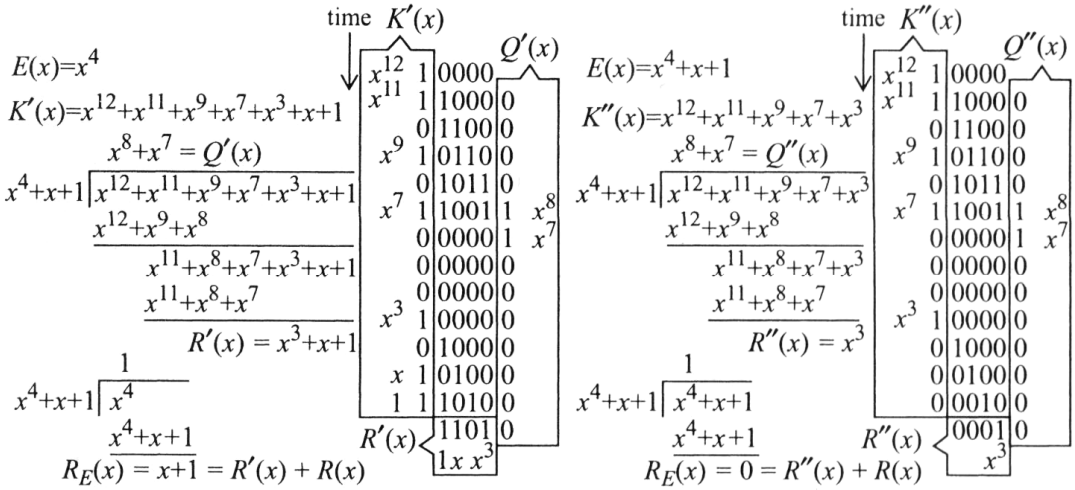
Onde: *E(x)* representa o erro polinomial que denota os bits errados na resposta do teste.

Assim, a diferença entre a assinatura do circuito falho (*R´(x)*) e a assinatura do circuito livre de falha (*R(x)*) é dada por:

$R_E(x) = R(x) + R'(x) = E(x) \bmod P(x)$ (4.4)

Onde: $R_E(x)$ representa a assinatura do erro polinomial *E(x)*. Assim, quando $R_E(x)$ apresenta um valor diferente de zero e múltiplo de *P(x)* significa que a falha do CUT foi

detectada. Entretanto, quando $R_E(x)$ é igual a zero, a falha não foi detectada e conseqüentemente ocorreu o que se denomina *signature aliasing*. A figura 4.15a ilustra uma falha detectada, onde um único bit de erro ($E(x) = x^4$) foi introduzido na resposta $K'(x)$ do CUT. Já a figura 4.15b ilustra um exemplo de *signature aliasing*.



Exemplo de detecção de falha — Exemplo de *aliasing* de assinatura

Figura 4.15 Exemplo dos processos de detecção de falha e de mascaramento de falha (*aliasing*). (STROUD, 2002)

Uma variação do compactador baseado em LFSR é dada por um LFSR de múltiplas entradas denominado registrador de assinatura de múltiplas entradas (*multiple-input signature register* – MISR). Esta estrutura é bastante utilizada para testar circuitos de múltiplas saídas em paralelo e caracteriza-se pela adição de portas lógicas do tipo OU-exclusivo nos *flip-flops*.

Durante a simulação, quando uma falha é detectada, ela deve ser adicionada à lista de falhas detectadas e não deve mais ser simulada, ou seja, o simulador deve parar e reiniciar a simulação a partir da próxima falha da lista. Este procedimento, denominado *fault dropping*, evita o desperdício de tempo durante o processo de simulação.

Funções ORAs removem a necessidade de um monitoramento contínuo das respostas do CUT, ou seja, exigem apenas que a indicação *pass/fail* seja monitorada no final da execução da seqüência de teste. Entretanto, este procedimento pode gerar *mascaramento de falhas* e conseqüentemente afetar diretamente a cobertura de falha do CUT. Assim, para se atingir uma

alta cobertura de falhas, toda a seqüência de teste deve ser executada antes da observação e avaliação da indicação *pass/fail*. Evidentemente, que este procedimento torna-se extremamente custoso em termos de tempo. Assim, uma alternativa bastante viável para solucionar este problema é monitorar periodicamente o status *pass/fail* durante a simulação e permitir *fault dropping*.

Contudo, a cobertura de falhas será precisa somente se esta técnica também for utilizada durante o teste de manufatura. No caso de análise de assinatura, a observação de assinaturas intermediárias reduz significativamente a probabilidade de mascaramento de falha.

4.6 Considerações sobre regras de projetos BIST

As soluções BIST, conforme anteriormente mencionado, podem ser classificadas como *on-line* ou *off-line*. Em soluções *on-line*, o teste é executado em paralelo com a operação normal do CUT e por isto são classificadas como concorrentes. Já em soluções *off-line*, o CUT é colocado em um modo de teste e não executa nenhuma de suas funções normais e portanto são classificadas como não-concorrentes.

A grande maioria das soluções BIST utilizadas são *off-line*. Assim, a arquitetura básica de uma solução BIST *off-line* é formada pelos seguintes elementos:

- Gerador de padrão de teste (TPG);
- Compactador das respostas de teste (ORA);
- Módulo de formatação dos dados de teste;
- Controlador do BIST.

Normalmente, esses blocos são adicionados ao CUT como uma lógica extra e segundo (RAJSKI, 1998) podem ser classificados como *ex-situ* ou *in-situ*. Assim, quando a lógica adicionada não contribui para a realização das funções do circuito, o BIST é classificado como *ex-situ*. Entretanto, quando a lógica adicionada contribui para a realização de algumas ou todas as funções do circuito, o BIST é dito *in-situ*.

Além disto, o BIST pode ser centralizado ou distribuído, de acordo com a estrutura do circuito alvo e do particionamento realizado nos componentes. Assim, a identificação dos módulos é realizada de acordo com:

- Tipo do módulo estrutural (RAM, ROM, etc);
- Alcance do módulo de *clock*;
- Hierarquia do circuito introduzida por um estilo particular de projeto.

Os módulos obtidos a partir do particionamento do circuito são também afetados pelo processo de implementação do BIST. Neste contexto, surgem alguns problemas de potência e ruído durante as execuções de BIST paralelos:

- Dissipação de potência nos módulos;
- Adjacência física entre módulos e protocolos de comunicação;
- Tipos de módulos;
- Comprimento do teste esperado dos módulos sucessivos;
- Especificações de entrada e saída.

Uma alternativa bastante viável e interessante, capaz de reduzir a redundância, o *overhead* de área e o *overhead* de desempenho, é agrupar módulos similares e compartilhar determinados recursos utilizados durante o BIST. O agrupamento de módulos de lógica aleatória pode reduzir significativamente o número de nós necessários para controlar e observar o teste. Contudo, este método pode aumentar bastante o tempo de aplicação de teste.

5. TÉCNICAS DE REDUNDÂNCIA

5.1 Introdução

Este capítulo irá abordar as diferentes técnicas de redundância através da apresentação de seus principais conceitos, metodologias e aplicações. A redundância de recursos em sistemas é capaz de agregar aos mesmos, funcionalidades extras, ou seja, a redundância pode torná-los tolerante a falhas. Assim, o conceito de redundância implica na adição de informações, recursos (hardware e software) ou tempo além dos que seriam necessários para a operação normal de um determinado sistema. A redundância pode assumir quatro formas, são elas: redundância de hardware, de informação, de tempo e de software. Conforme anteriormente mencionado, as técnicas de redundância, são utilizadas em testes concorrentes, ou seja, em testes on-line que são executados concorrentemente a aplicação normal do sistema. Salienta-se que, apesar de aumentarem a confiabilidade de sistemas, a redundância causa impactos bastante significativos no que diz respeito à performance, ao tamanho, ao peso, ao poder de consumo entre outros.

5.2 Redundância de Hardware

Atualmente, a redundância de hardware é a abordagem mais utilizada em um sistema de computação. Isso se deve principalmente a constante queda no custo do hardware, ou seja, os componentes semicondutores estão cada vez menores e mais baratos. Existem três metodologias básicas dentro da redundância de hardware, são elas: passiva, ativa e híbrida.

5.2.1 Redundância de Hardware Passiva

A técnica passiva de redundância de hardware baseia-se no conceito de *mascaramento de falhas*. Fundamentalmente, a técnica procura evitar que a ocorrência de falhas resultem em erros no sistema, ou seja, a técnica não detecta falhas, simplesmente provê o mascaramento das

mesmas. A grande maioria das metodologias passivas utiliza o conceito de “maioria de votação” para prover o mascaramento das falhas e não exige nenhum tipo de ação por parte do sistema ou operador.

Dentre as principais metodologias de redundância de hardware passiva, encontra-se a Redundância Modular Tripla (*Triple Modolar Redundancy* - TMR) ilustrada na figura 5.1a Basicamente, a TMR consiste em triplicar o hardware e executar voto pela maioria (*majority vote*) para determinar a saída do sistema. Assim, quando ocorre uma falha em um dos módulos, os módulos restantes, livres de falhas, mascaram os resultados do módulo falho através do voto pela maioria. Este conceito pode ser utilizado em vários dispositivos de hardware, tais como processadores e memórias.

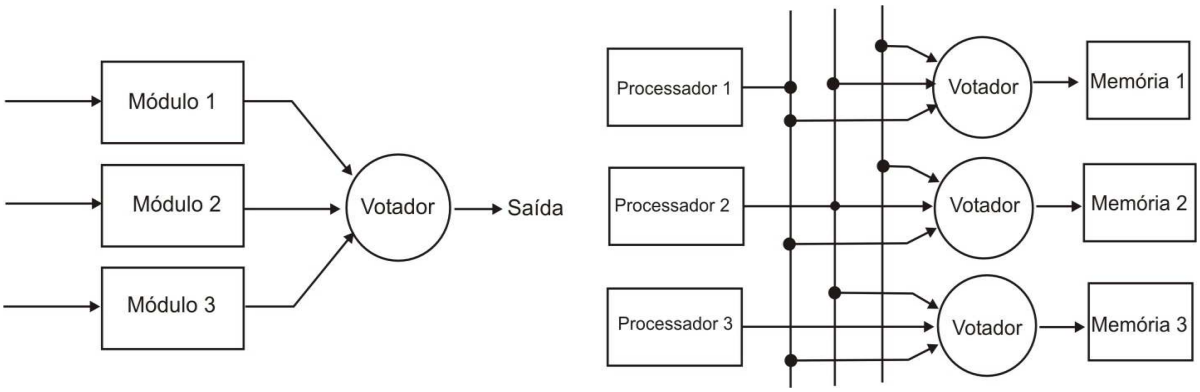


Figura 5.1 Redundância de hardware passiva: a) TMR e b) TMR com votadores triplicados. (PRADHAN, 1996)

Entretanto, o principal problema encontrado nesta abordagem é possibilidade do votador falhar e assim levar o sistema a uma falha geral. Segundo D. K. Pradhan (PRADHAN, 1996), uma falha em um único ponto (*Single-point-of-failure*) ocorre quando ocorre uma falha geral no sistema devido a apenas um componente dentro do mesmo. Para solucionar o problema acima mencionado, várias técnicas já foram propostas na literatura. Uma destas técnicas sugere a triplicação do elemento votador, assim cada um dos membros recebe dados de um votador diferente que, por sua vez, recebe como entrada o conteúdo de três processadores separados, como mostra a figura 5.1b Caso um dos processadores falhe a memória continuará recebendo o valor correto, pois seu eleitor corrigirá o valor corrompido. Este tipo de sistema é normalmente

chamado de órgão restaurador, pois produz três saídas corretas mesmo se uma das entradas estiver errada.

O votador dentro de um sistema com *n* redundância modular (*n modular redundancy* – NMR) pode ser utilizado em vários pontos do sistema. A escolha do ponto em que o votador será inserido deve considerar os critérios a seguir considerados. Primeiramente, deve-se decidir se o votador será hardware ou em software. Evidentemente que esta escolha deve levar em consideração os seguintes aspectos: um votador implementado em software aproveita a capacidade computacional disponível no processador para executar o processo de votação com um mínimo de hardware extra e permite através da simples modificação do software a alteração da forma com que o voto é utilizado; entretanto, a utilização de um votador implementado em hardware permite que os processos sejam executados dedicadamente e mais rapidamente.

Portanto, a decisão entre utilizar hardware ou software para implementação do votador baseia-se fundamentalmente nos seguintes aspectos:

- disponibilidade de um processador para executar o processo de voto;
- a velocidade em que o processo de voto deve ser executado;
- as limitações de espaço, poder e peso;
- o número de votadores diferentes que devem ser gerados;
- a flexibilidade do votador em relação a futuras alterações.

5.2.2 Redundância de Hardware Ativa

A Redundância de Hardware Ativa baseia-se no princípio de obter tolerância a falhas a partir da detecção, localização e recuperação (*recovery*) de falhas, ou seja, técnicas ativas exigem que o sistema execute recuperação do sistema para tolerar falhas. Em muitos sistemas, as falhas podem ser detectadas simplesmente observando-se os erros que elas produzem. Este método é indicado para aplicações onde resultados errados durante um breve intervalo de tempo, reconfiguração e recuperação do status operacional do sistema são considerados procedimentos aceitáveis.

A figura 5.2 mostra o diagrama de funcionamento da redundância de hardware ativa onde é possível observar a ocorrência de falhas durante a operação normal do sistema. Neste contexto, os termos falha (*fault*), erro (*error*) e falho (*failure*) definem eventos distintos dentro de um sistema. Segundo D. K. Pradhan (PRADHAN, 1996) falha define uma discrepância qualquer; erro ocorre após o período de latência da falha, e este pode ser detectado ou não. E finalmente, um sistema falho é o resultado de um erro que não foi detectado. Caso o erro seja detectado, sua origem deve ser localizada e o componente falho removido da operação. Assim, um componente próximo e disponível deve ser habilitado para que o sistema volte a seu estado operacional. Cabe salientar que os processos de localização de falhas, restrição da falha e restauração são normalmente referenciadas simplesmente como reconfiguração. Com isto, fica claro que métodos ativos de tolerância à falhas exigem a capacidade de detecção e localização de falhas.

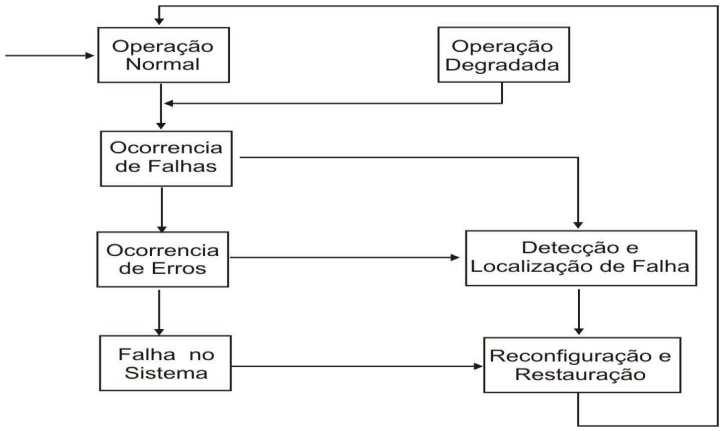


Figura 5.2 Operação básica de um método ativo de tolerância a falhas.(PRADHAN, 1996)

Dentre os mecanismos de detecção de falha, temos a Duplicação com Comparação (*duplication with comparison*) que se baseia na construção de duas peças idênticas de hardware que executam as mesmas computações em paralelo e comparam os resultados. Caso haja um desacordo entre os resultados uma mensagem de erro será gerada. Entretanto, apesar de ser capaz de detectar falhas, este método não consegue determinar em qual dos dois módulos ocorreu a falha. Além disso, esta metodologia apresenta os seguintes problemas:

- Dependendo da área de aplicação, o comparador pode não ser capaz de executar a comparação exata;
- Falhas no comparador podem gerar uma indicação de erro quando este não existe;

- O comparador pode falhar e com isso falhas eventuais nos módulos duplicados jamais serão detectadas.

No entanto, existem algumas técnicas capazes de superarem estes problemas. Em sistemas que possuem o microprocessador duplicado, se pode implementar o processo de comparação em software a fim de que o mesmo seja executado pelos microprocessadores. A figura 5.3. mostra um esquema da técnica de duplicação com comparação implementada em software. Neste esquema, cada processador tem sua própria memória para armazenar programas e dados e uma memória compartilhada entre os processadores utilizada para transferir resultados de um processador para o outro a fim de compará-los. Assim, este método detecta eventuais falhas na memória compartilhada e nos processadores.

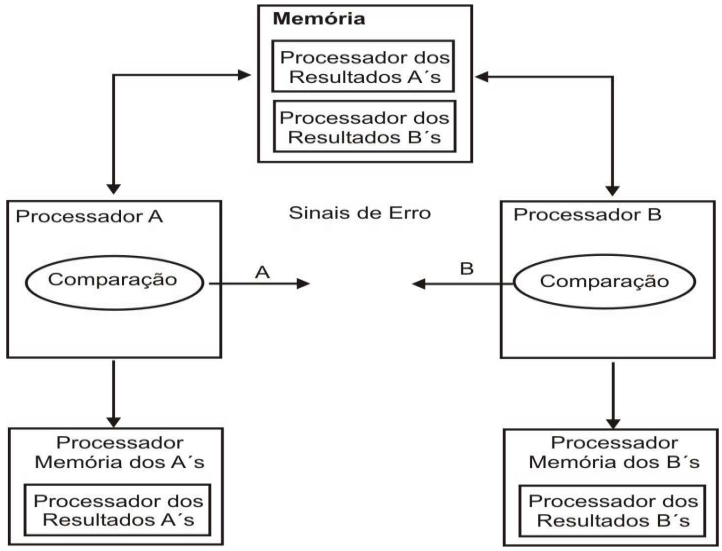


Figura 5.3 Implementação de um esquema da duplicação com comparação em software.(PRADHAN, 1996)

Existem várias técnicas que podem ser utilizadas para comparar processadores, dentre as quais podemos citar a comparação bit a bit de cada palavra digital.

Além do método acima apresentado, existe o *standby replacement* ou *standby sparing*. Neste método um módulo é operacional e os outros servem como *standbys* ou *spares*. Quando uma falha é detectada e localizada, o módulo falho deve ser removido e substituído por *spares*. Assim, a reconfiguração do sistema - que pode ser realizada logo após a falha ter ocorrido - pode ser vista como um interruptor cuja saída é selecionada de um dos módulos, após examinar os

erros associados a cada módulo, que abastecem a entrada do interruptor. Caso os módulos sejam livres de falhas a seleção é feita com base na prioridade fixada; caso contrário, os módulos errados são descartados.

E por fim, salienta-se a redundância de hardware híbrida, que combina características dos métodos passivo e ativo, ou seja, utiliza o conceito de mascaramento de falhas para evitar a propagação de resultados errados, juntamente com recursos de detecção, localização e restauração de falhas. Este tipo de redundância é utilizado somente em aplicações que exigem integridade rápida das computações.

A tabela 5.1 mostra um de maneira bastante clara um resumo comparativo entre os três tipos de redundância de hardware.

Característica Avaliada	**Técnica PASSIVA**	**Técnica ATIVA**	**Técnica HIBRIDA**
Utilizam:	Mascaramento de Falhas.	Técnicas de detecção, localização e reconfiguração.	Mascaramento de Falhas juntamente com reconfiguração.
São indicadas em aplicações que:	Computação crítica que não toleram resultados errados momentâneos.	Em aplicações que toleram temporariamente saídas erradas.	Computação crítica com altos índices de confiabilidade que não toleram resultados errados momentâneos.
Custo:	Pequeno	Substancial	Grande

Tabela 5.1 Comparação dos tipos de Redundância de Hardware.

5.3 Redundância de Tempo

A redundância de tempo é uma opção bastante interessante capaz de gerar sistemas tolerantes a falhas específicas. Ela surgiu com o intuito de minimizar os custos associados a redundância de hardware, a partir da utilização de tempo adicional. Então, a redundância de tempo é obtida a partir da repetição de computações com o objetivo de detectar eventuais falhas, ou seja, executar uma mesma computação duas ou mais vezes e comparar seus resultados a fim de verificar se existe alguma discordância entre eles. Caso seja detectado algum erro, as

computações devem ser executadas novamente a fim de verificar se a discordância desapareceu ou não. Neste caso, as falhas detectadas são do tipo transientes ou temporárias.

Esta técnica é indicada em aplicações cujo tempo pode ser prontamente disponibilizado e quando se deseja minimizar o impacto gerado pela aplicação de metodologias de redundância de hardware (entidades físicas geram impactos de peso, tamanho, custos, dentre outros), ou seja, em sistemas que podem tolerar tempo adicional mais facilmente do que hardware adicional. O uso de redundância de tempo é bastante eficiente para detectar erros causados por falhas transientes.
Entretanto, o principal problema das técnicas de redundância de tempo é assegurar a disponibilidade do mesmo dado para cada tempo de execução da computação redundante. Cabe salientar, que caso uma falha transiente ocorra, dados do sistema podem ser totalmente corrompidos dificultando a repetição da computação.

Além de ser usada na detecção de falhas transientes, atualmente indica-se o uso de técnicas de redundância de tempo na detecção de falhas permanentes. Este conceito é mostrado na figura 5.4 onde durante a primeira computação, os operandos são utilizados como apresentados e os resultados são armazenados em um registrador. Antes de iniciar a segunda computação os operandos são codificados utilizando uma função de codificação. Depois das operações terem sido executadas nos dados de codificação, os resultados são decodificados e comparados com aqueles obtidos durante a primeira operação. Existem alguns exemplos de funções de codificação como o operador de complementação (*complementation operator*) ou lógica alternada (*alternating logic*) e também troca aritmética (*arithmetic shift*).

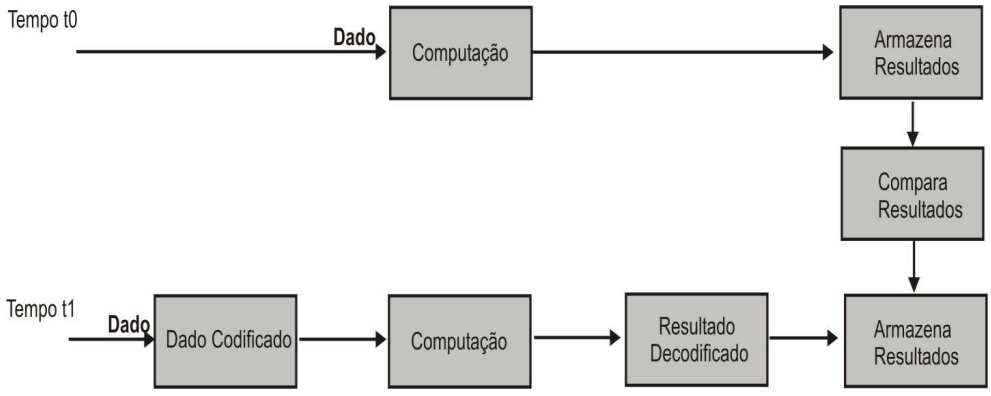


Figura 5.4 Detecção de falha permanente usando redundância de tempo.(PRADHAN, 1996)

O operador de complementação é comumente utilizado para a detecção de falhas na transmissão de dados digitais e a detecção de falhas em circuitos digitais. Suponha, por exemplo, que seja necessário detectar eventuais erros que podem ocorrer em um barramento a partir do uso de métodos de redundância de tempo. Em um tempo *t0*, o dado é transmitido, em um tempo *t1* transmitimos o complemento codificado do primeiro do dado. Se o barramento está *stuck* em 1 ou 0, os dois dados transmitidos serão iguais e não um o complemento do outro então, a falha é detectada. Deste modo a transmissão livre de erros deve ser uma seqüência alternada de 0 e 1.

5.4 Redundância de Informação

A Redundância de informação consiste na adição de informação redundante com o objetivo de detectar, mascarar ou possivelmente tolerar falhas. Antes de detalhar esta técnica de redundância, será apresentado e definido alguns de seus principais conceitos.

Segundo Pradhan (PRADHAN, 1996), a redundância de informação envolve os conceitos abaixo definidos:

Código (*Code*) – é a maneira de representar uma determinada informação, ou dados, a partir de um conjunto de regras.

Palavra de Código (*Codeword*) – é uma coleção de símbolos, frequentemente chamada de dígitos se os símbolos forem números, usado para representar um pedaço em particular de dados baseados em um código específico.

Código Binário (*Binary Code*) – é um código que utiliza somente os dígitos 0 e 1 para formar uma palavra de código.

Operação de Codificação (*Encoding Operation*) – é o processo para determinar a palavra de código correspondente para um elemento de dados em particular, ou seja, é o processo que transforma um elemento de dados original em uma palavra de código obedecendo as regras do código.

Operação de Decodificação (*Decoding Operation*) – é o processo de restauração da palavra de código para o dado original, ou seja, a partir da palavra de código este processo encontra o dado original.

Código de Detecção de Erro *(Error Detecting Code*) – é a capacidade de distinguir um palavra de código válida de uma palavra de código corrompida ou inválida.

Código de Detecção de Erro Duplo (*Double-error-detecting-code*) – um código é dito código de detecção de erro duplo quando é capaz de corrigir 2 bits errados, e assim por diante.

Um código pode ser caracterizado a partir de dois conceitos fundamentais, a distância *Hamming* e a separabilidade (*separability*). A distância *Hamming* é dada pelo número de posições que diferem duas palavras entre si, por exemplo, em 0000 e 0001 a distancia *Hamming* é 1. Assim a distância de um código é a menor distância entre dois *codeword* válidos quaisquer. Separabilidade de um código é obtida quando a informação original é anexada com a nova informação para formar a palavra do código. Com isto o processo de decodificação consiste simplesmente em remover o código adicional ou bits extras, denominados bits de código ou bits de verificação, e manter a informação original. Quando um código não possui esta propriedade é necessário utilizar procedimentos de decodificação mais complexos.

5.4.1 Código de Paridade

O Código de Paridade é a maneira mais simples para se proteger um código. O Código de Paridade de Bit Único (*Single-bit parity code*) é uma variação do código de paridade que basicamente exige a adição de um bit extra na palavra binária resultando, assim em uma palavra de código com um número par ou ímpar de 1s. Caso o número total de 1s na palavra de código for impar, o código é denominado paridade ímpar (*odd parity*) e, se o número de 1s na palavra de código for par, o código será paridade par (*even parity*). Portanto, se em uma palavra de código com paridade ímpar um de seus bits for alterado sua paridade passará a ser par, e vice-versa, conseqüentemente é possível detectar erros de bit único checando o número de 1s da palavra de código. Esta abordagem é bastante utilizada em memórias de sistemas de computadores.

Um dos problemas mais significativos encontrados na técnica de paridade de bit único é a sua inabilidade em detectar alguns erros de bits múltiplos.

O esquema básico de paridade pode ser modificado a fim de torná-la capaz de detectar erros adicionais. Pradhan (PRADHAN, 1996) sugere 5 modificações para o mesmo:

A) Paridade por palavra *(Bit-per-word parity*): é anexado em cada palavra um bit de paridade.

B) Paridade por byte (*Bit-per-byte parity*): o dado original é separado em duas porções iguais e um bit de paridade é associado a cada uma delas.

A principal desvantagem das técnicas acima, bit por palavra e bit por byte, é o fato de não possuírem a capacidade de detectar erros em mais de um bit.

C) Paridade por múltiplos chips (*Bit-per-multiple-chip*): é necessário um bit de cada chip da memória associado com um único bit de paridade. Apesar desta técnica detectar a falha de um chip, ela não é capaz de localizá-lo.

D) Paridade intercalada (*Interlaced parity*): apresenta um conceito muito semelhante à paridade por múltiplos chips com apenas uma diferença, em paridade intercalada os grupos de paridade são formados sem considerar a organização física da memória, ou seja, em paridade intercalada os bits de informação são divididos em grupos de tamanhos iguais e um bit de paridade é associado a cada grupo.

E) Paridade sobreposta (*overlapping parity*): os grupos de paridade são formados por bits que aparecem em mais de um grupo de paridade, ao contrário das técnicas acima descritas onde cada bit é formado por apenas um grupo de paridade.

5.4.2 Códigos Aritméticos

Os códigos aritméticos são bastante utilizados para a verificação de operações aritméticas e consistem basicamente em codificar os dados fornecidos para a operação aritmética antes de seu processamento. Após a execução das operações, a palavra de código resultante será verificada a fim de assegurar sua validade. Caso as palavras de códigos não sejam válidas o erro será sinalizado.

Para que um código aritmético funcione corretamente, ele deve ser invariável para um conjunto de operações aritméticas. Assim, um código aritmético A, tem a propriedade que A(b*c)= A(b)*A(c), onde b e c são operandos, * é alguma operação aritmética e A(b) e A(c) são palavras de código aritméticos para os operandos b e c.

Dentre os principais tipos de códigos aritméticos, salientam-se: códigos NA, código de resíduo e código de resíduo inverso.

5.4.3 Código de Hamming

O código *Hamming* é formado a partir da divisão dos bits da informação em grupos e especificação de um bit de paridade para cada grupo. No código *Hamming* o dado original é codificado gerando um conjunto chamado C*g*, de bits de verificação de paridade. Quando for necessário a verificação da informação para suas possíveis correções, o processo de codificação é repetido e um conjunto chamado C*c* de bits de verificação de paridade é regenerado. Se C*g* e C*c* concordam, a informação é assumida ser correta. Se, contudo eles discordam a informação está incorreta e deve ser corrigida. Este tipo de código é provavelmente o mais utilizado para a proteção de dados armazenados em memórias. Isto se deve fundamentalmente ao baixo custo associado a sua implementação e ao curto intervalo de tempo requerido para a execução do processo de correção (a codificação e decodificação acrescentam *delays* relativamente pequenos).

O código *Hamming* permite a localização do bit errado e a correção de erros em bits únicos (*single-bit errors*). Infelizmente, erros em mais de um bit serão erroneamente corrigidos usando o código *Hamming*. Para superar o problema de correções errôneas e prover um código que pode corrigir erros de bits únicos e identificar erros de bits duplos existe uma modificação do código *Hamming*. Neste caso, associa-se um bit de paridade ao código de *Hamming*.

Salienta-se que além dos códigos aritméticos anteriormente mencionados, existe ainda o código M de N, o código Berger e o *checksum*, que apesar de serem largamente utilizados e comentados na literatura não serão detalhados, pois não fazem parte do escopo principal desta dissertação. O leitor interessado poderá obter maiores informações em Pradhan (PRADHAN, 1996).

5.5 Redundância de Software

A redundância de software é uma técnica que pode ser utilizada em muitas aplicações e consiste em utilizar software para implementar técnicas de tolerância à falhas e detectar falhas.

Este tipo de redundância requer uma quantidade mínima de redundância de hardware e pode assumir várias formas.

5.5.1 Verificação de Consistência

A verificação de consistência utiliza as características da informação para verificar se as mesmas estão corretas. Esta técnica é implementada na maioria das vezes via software podendo, também, ser implementada via hardware.

Como exemplos de aplicações onde a verificação de consistência se apresenta como uma excelente solução, salientam-se: (1) Sistema de controle que armazena os valores lidos a partir de um sensor e verifica se os mesmos pertencem ao conjunto de valores aceitáveis. (2) Sistema bancário que verifica se a quantidade de dinheiro solicitada por um determinado cliente em um terminal excede ou não seu limite de crédito; (3) Sistema de detecção de instruções inválidas em computadores onde a verificação de consistência pode ser executado em hardware. Muitos computadores utilizam uma quantidade de *n*-bits para representar 2k possíveis códigos de instruções onde $2k < 2n$, ou seja, $2n - 2k$ códigos de instruções são inválidos. Neste contexto cada instrução é verificada e se um código inválido for encontrado o processador pára, a fim de evitar a execução de operações erradas.

Apesar das vantagens acima mencionadas, a verificação de consistência apresenta alguns pontos críticos que devem ser considerados: (1) Dificuldade para estabelecer o nível de desvio permitido antes de sinalizar um erro e (2) Dificuldade no que diz respeito à precisão do modelo, caso sejam obtidos bons resultados.

5.5.2 Verificação de Capacidade

Verificação de capacidade é executada para verificar se um sistema possui a capacidade esperada ou se todos os seus recursos estão disponíveis. Pradhan (PRADHAN, 1996) sugere algumas aplicações para esta abordagem, são elas: (1) Um simples teste de memória onde o processador escreve padrões específicos em determinadas posições da memória e depois lê estas posições a fim de verificar se os dados foram corretamente armazenados e recuperados; (2)

Testes da ULA que consiste na execução periódica de instruções específicas (adição, multiplicação, operações lógicas e transferência de dados) de dados específicos e na comparação dos resultados em relação aos armazenados na ROM e (3) Verificar se todos os processadores, em um sistema formado por múltiplos processadores, são capazes de se comunicarem entre si através da comunicação entre suas memórias.

5.5.3 Programação N – Auto-Testável

As técnicas apresentadas até aqui utilizam software extra ou redundante para detectarem falhas que possam ocorrer no hardware. A partir de agora serão apresentados métodos que podem ser utilizados para detectar ou tolerar falhas que possam ocorrer no software de um determinado sistema.

Ao contrário do hardware, o software não pode parar e suas falhas resultam de erros no projeto ou na codificação. Por isso, é importante salientar que a duplicação e comparação de um procedimento não garante a detecção de falhas no software, pois os módulos duplicados são idênticos e assim, apresentam a mesma codificação.

Neste contexto surgiu a programação N-auto-testável (*N Self-checking programing*). Ela consiste essencialmente em escrever N versões de um programa e especificar um conjunto de testes de aceitação para cada um deles. Neste método cada programa é executado simultaneamente, de forma concorrente na mesma máquina; ou de forma paralela em um sistema distribuído. Quando uma falha é detectada na versão 1 do programa de aplicação pelos testes de aceitação, a versão 2 passa a ter suas saídas validadas, caso estas tenham sido aprovadas pelos testes de aceitação. A programação N-auto-testável detecta em cada versão do programa falhas independentes e conseqüentemente pode tolerar N-1 falhas. A figura 5.5 ilustra claramente o método de programação N- Auto- testável.

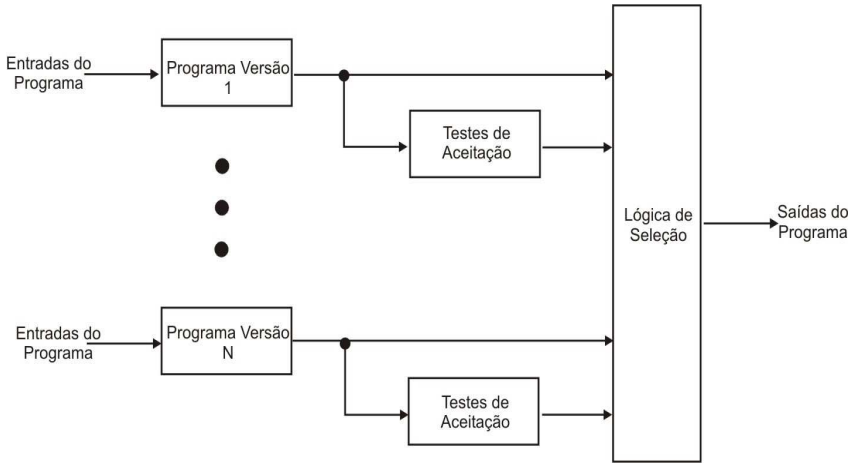


Figura 5.5 O método de programação N-auto-testável para tolerância à falhas de software.(PRADHAN, 1996)

5.5.4 Programação a N-versões

O conceito básico desta técnica é projetar e codificar o módulo de software N vezes, e votar nos N resultados produzidos por esses módulos. Cada um dos N módulos é projetado e codificado por um grupo separado de programadores que recebe o mesmo conjunto de especificações. Espera-se com isto, que os erros introduzidos nos módulos não sejam os mesmos e que as falhas que possam vir a ocorrer sejam distintas e não ocorram necessariamente em todos os módulos. Assim, os resultados gerados pelos módulos serão diferentes. Assumindo que as falhas são independentes, este método pode tolerar (N-1)/2 falhas.

As principais dificuldades encontradas neste método são: (1) Projetistas e programadores de software podem cometer os mesmos erros; (2) As N versões obtidas são desenvolvidas a partir de uma especificação comum, por isso não é possível garantir tolerância de erros na especificação; (3) Se as *n*-versões forem executadas concorrentemente na mesma máquina, a degradação no tempo de execução do código da aplicação pode tornar-se inaceitável.

Para superar os problemas associados com a técnica de programação *n*-versões, os projetistas de software utilizam rígidas técnicas de projeto, pois um software projetado corretamente dispensa o uso de técnicas de tolerância a falhas. A figura 5.6 mostra o conceito de programação *n*-versão.

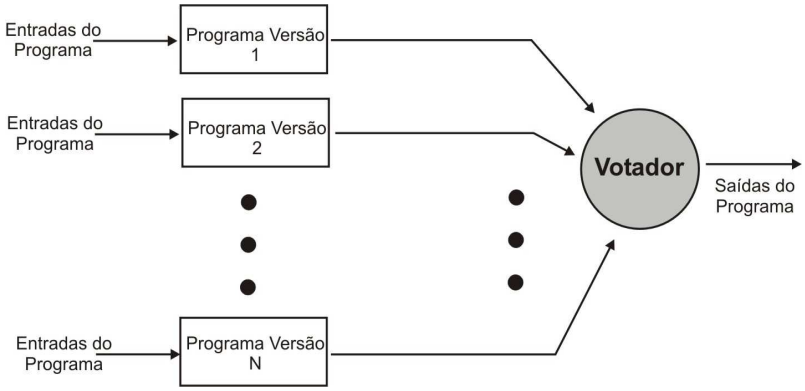


Figura 5.6 O conceito de programação N-versão.(PRADHAN, 1996)

5.5.5 Blocos de Recuperação

É uma técnica análoga ao método de redundância de hardware *cold* sparing, ou seja, N versões de um programa são desenvolvidas e um único conjunto de testes de aceitação é utilizado. Uma versão é designada como *versão primária* e as demais são os *spares* ou versões secundárias. Os testes de aceitação são sempre aplicados à versão primária, a menos que a mesma não passe pelos testes. Se isto ocorrer a primeira versão secundária é inicializada e testada e assim sucessivamente. Caso nenhuma das versões testadas passe pelo teste de aceitação o sistema irá falhar. Este método pode tolerar até N-1 falhas. A figura 5.7 ilustra a metodologia de blocos de recuperação.

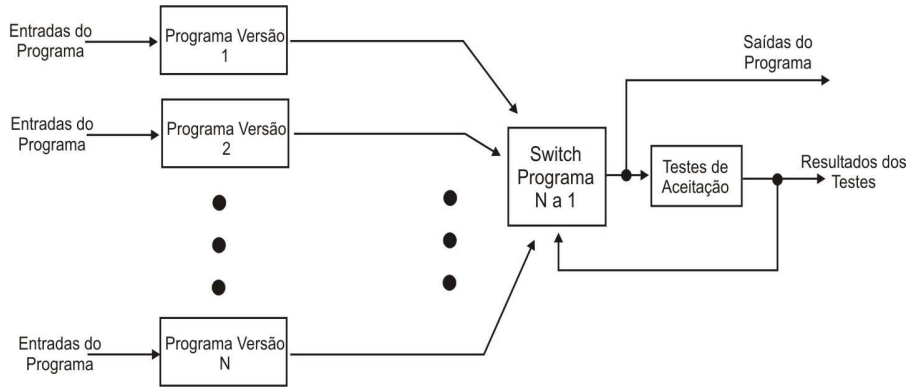


Figura 5.7 O método recuperação de bloco para tolerância a falhas em software.(PRADHAN, 1996)

6. TÉCNICAS DE TOLERÂNCIA A FALHAS VIA SOFTWARE

Introdução

Em SoCs, durante seu período de funcionamento, falhas transientes e permanentes podem causar discrepâncias nos valores dos dados manipulados e a execução incorreta da seqüência de instruções. Quando isto ocorre surge o que chamamos respectivamente de *falhas em dados* e *falhas de fluxo de controle*. 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.

As técnicas de bloco de recuperação, programação a *n*-versões e programação *n*-auto-testável, definidas no capítulo anterior, também são capazes de detectarem estes tipos de falhas, entretanto não garantam a detecção de todas as suas ocorrências. Além disso, as técnicas de redundância de software, apresentam uma maior latência para a detecção das falhas, dado que a quantidade de testes de aceitação é limitada e são normalmente inseridos somente no fim do código da aplicação ou em rotinas internas previamente definidas.

Neste contexto, a fim de garantir eficientemente a detecção das falhas em dados e no fluxo de controle em SoCs, surgem as metodologias de tolerância a falhas 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.

Técnicas de Detecção de Erros em Dados

As falhas em dados provocam alterações indesejadas no conteúdo das variáveis. Assim para evitar que essas falhas se propaguem no sistema, surgem várias metodologias de tolerância a falhas capazes de detectá-las. A seguir serão apresentadas duas técnicas propostas na literatura para a detecção de falhas em dados.

Técnica ED^{4}I

Error Detection by Diverse Data and Duplicated Instructions - ED^{4}I, proposta por Oh. Nahmsuk (NAHMSUK, 2002) é uma técnica SIHFT que provê a detecção de falhas permanentes e temporárias executando dois programas "diferentes", que apresentam a mesma funcionalidade e diferentes conjuntos de dados, e comparam as suas saídas. Basicamente, ED^{4}I determina que cada número *x*, do programa original seja mapeado em novo número *x´* de tal forma que os resultados das duas versões coincidam. O mapeamento é feito através da equação (6.1).

$$x´ = k.x \quad (6.1)$$

Onde: *k* determina a probabilidade de detecção de falha e a integridade dos dados do sistema.

Regras de transformação do algoritmo

Antes de apresentar as regras de transformação definidas em ED^{4}I, algumas definições e terminologias pertinentes serão apresentadas.

Segundo Oh. Nahmsuk (NAHMSUK, 2002) um bloco básico é uma seqüência de instruções consecutivas em que o fluxo de controle entra no início e sai no fim sem encontrar nenhum desvio até o final.

Abaixo segue a notação utilizada em ED^{4}I:

V = {v1, v2, .. , vn}: conjunto dos vértices que representam os blocos básicos;

E = {(i,j)| (i,j) é um desvio de vi para vj}: conjunto dos limites que representam os possíveis desvios entre os blocos básicos;

Assim um programa pode ser representado por um grafo P = {V, E}.

A figura 6.1 apresenta o um determinado programa seguido de seu grafo de fluxo. Neste caso, este programa possui quatro blocos básicos, *v1*, *v2*, *v3* e *v4* e o conjunto dos desvios possíveis é definido por E = {(1,2), (2.3), (3,2),(2,4)}.

As regras definidas em ED^{4}I transformam *P*, programa original em *P´*. *P´* resulta da multiplicação de todas as variáveis e constantes do programa original por um fator k e portanto se *x* é *k* vezes maior que *y*, *x* é *k* múltiplo de *y*. Assim esta técnica determina que sejam executadas as transformações abaixo descritas:

- **Transformação da expressão**: este passo transforma as expressões dentro *P* para novas expressões em P´, de tal forma que cada variável ou constante de *P´* é o valor de *P* multiplicado por *k*. Como os valores de P´ são diferentes dos valores originais, quando comparamos os dois valores em um instrução condicional, a relação de desigualdade pode necessitar de uma alteração. Por exemplo, dada a instrução condicional *if (i < 5)* em *P*, ela necessita ser alterada para *if (i > - 10)* em *P´* quando *k = -2*. De outra forma, o fluxo de controle determinado pelas instruções condicionais em *P´* devem ser diferentes do fluxo de controle em *P*, e portanto o resultado da computação do programa *P´* não será *k* múltiplo do código original.
- **Transformação das condições de desvio**: este passo ajusta a relação desigual na instrução condicional em *P´* tal que o fluxo de controle em *P* é *P´*são idênticos.

Assim, um programa multiplicado por um fator *k* é um programa com um novo grafo denotado por *Pg´[{V´, E´}* que por sua vez é isomórfico a *Pg.* Dado que *S* e *S´* representam o conjunto das variáveis de *P* e *P´* respectivamente e *n* o número de vértices (blocos básicos) executados, então *S(n)* e *S´(n)* são definidos como :

- *S(n):* o conjunto de variáveis em *S* após *n* blocos básicos serem executados;
- *S´(n):* o conjunto de variáveis em *S´* após *n* blocos básicos serem executados.

A partir da análise da figura 6.1 é possível definir o conjunto de variáveis, ou seja, *S = {i, x, y, z}*. Assim, após o programa ser iniciado e um bloco básico ser executado, *n = 1* e *S(1) = {i = 0, x = 1, y = 5, z = 0}* porque as quatro instruções do primeiro bloco são executadas. Após *v2* e *v3* serem executadas, *n = 3* e *S(3) = {i = 1, x = 1, y = 5, z = 1}*.

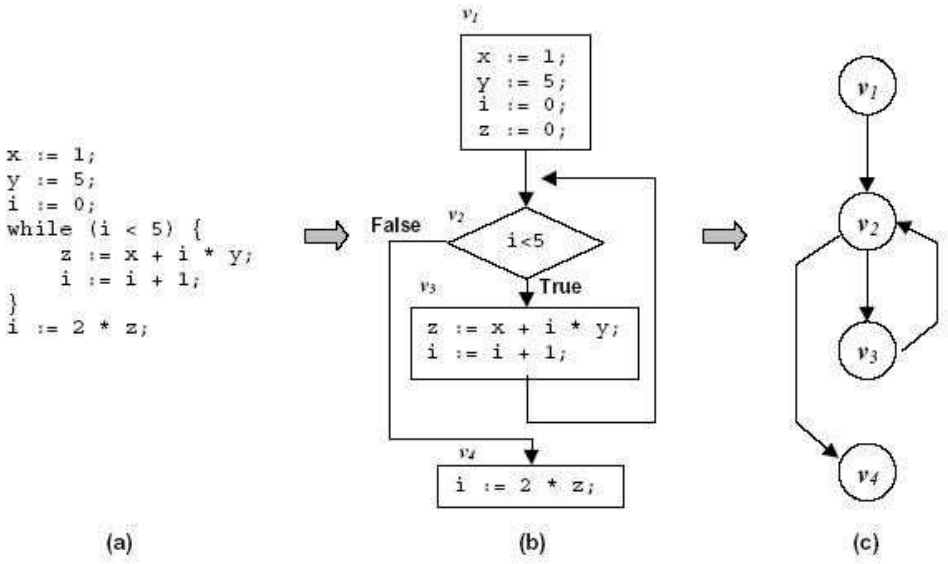


Figura 6.1 (a) um programa; (b) o grafo de fluxo e (c) o grafo do programa Pg. (NAHMSUK, 2002)

Em resumo, a transformação do programa deve satisfazer as condições abaixo: *Pg* e *Pg´*são isomórficos;

(1) $k.S(n) = S´(n)$, para $\forall\ n > 0$

Onde: *k.S(n)* é obtido multiplicando todos os elementos em *S(n)* por *k.*

A condição (1) define que o fluxo de controle nos dois programas deve ser idêntico e a condição (2) define que todas as variáveis de *P´* são sempre múltiplas de k do programa original *P*.

A figura 6.2 e 6.3 mostra o programa da figura 6.1 transformado a partir da utilização de um fator $k = -2$.

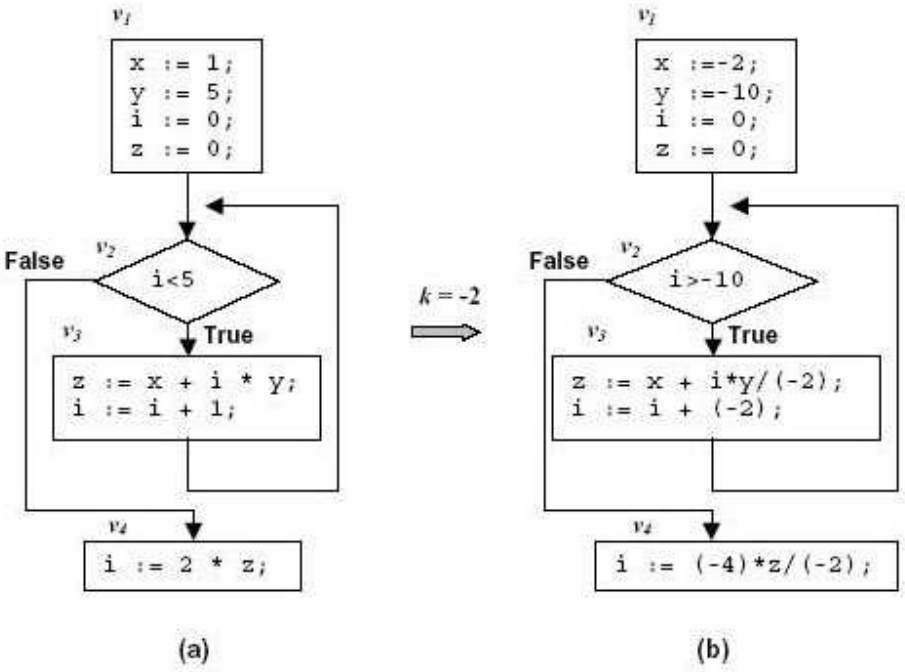


Figura 6.2 (a) grafo do programa original e (b) grafo do programa transformado com *k = -2*. (NAHMSUK, 2002)

```
x := 1;
y := 5;
i := 0;
while (i < 5) {
    z := x + i * y;
    i := i + 1;
}
i := 2 * z;
```

(a)

k = -2

```
x := -2;
y := -10;
i := 0;
while (i > -10) {
    z := x + i * y /(-2);
    i := i + (-2);
}
i := (-4) * z / (-2);
```

(b)

Figura 6.3 (a) o programa original e (b) o programa transformado com *k = -2*. (NAHMSUK, 2002)

Salienta-se que o fator *k* é determinado levando-se em consideração e conseqüentemente evitando um eventual *overflow* durante a execução do programa transformado.

Quanto à cobertura de falhas, esta técnica é capaz de detectar falhas temporárias através da comparação dos resultados obtidos a partir da execução das duas versões do código e falhas permanentes. Além disso, esta técnica pode detectar falhas permanentes que afetam o caminho dos dados em unidades funcionais através da execução de dois programas com dados diferentes utilizando diferentes partes das unidades funcionais e comparando os resultados.

Técnica proposta por M. Rebaudengo

M. Rebaudengo (REBAUDENGO, 2004) propõe uma solução via software para a detecção e correção de falhas transientes afetando dados manipulados pelo programa. Para isto, esta técnica apresenta várias regras de transformação que são aplicadas ao código fonte do programa alvo. Para aplicar esta técnica em um determinado programa, deve-se seguir as seguintes regras:

- Cada variável *x* deve ser duplicada: *x0* e *x1* representarão as duas cópias. Cada operação de escrita executada em x deve ser executada também em *x0* e *x1*. Assim, dois conjuntos de variáveis são gerados: o conjunto 0 e o conjunto 1.
- Após cada operação de leitura em x, as duas cópias *x0* e *x1* devem ser verificadas, e se alguma inconsistência é detectada, uma rotina de correção de erro deve ser ativada.

A figura 6.4 mostra à esquerda o código original de um determinado programa escrito em C ANSI e à direita o mesmo código acrescido das modificações acima descritas.

Código original	Código modificado

`a = b;`	`a0 = b0;`
	`a1 = b1;`
	`if (b0 != b1)`
	`  error();`

`a = b + c;`	`a0 = b0 + c0;`
	`a1 = b1 + c1;`
	`if ((b0!=b1) || (c0!=c1))`
	`  error();`

Figura 6.4 Código original e código tolerante a falhas. (REBAUDENGO, 1999)

Esta técnica detecta falhas que possam vir a ocorrer durante o processo de manipulação das variáveis pelo processador ou durante o período em que estas estiverem armazenadas na memória.

Técnicas de Detecção de Erros de Fluxo de Controle

Assim como as falhas em dados, as falhas de fluxo de controle podem ser geradas a partir de ruídos externos e consistem na execução das instruções em uma seqüência incorreta, ou seja, a execução do programa não segue o fluxo correto de execução.

Salienta-se que a grande 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.

Oh Nahmsuk (NAHMSUK, 2002) sugere a seguinte notação:

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 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*.

A seguir serão descritas diferentes metodologias capazes de detectarem falhas de fluxo de controle propostas na literatura.

Técnica CCA (Control Flow Checking by Assertions)

A técnica CCA proposta em G. A. Kanawati (KANAWATI, 1996) e Z. Alkhalif (ALKHALIF, 1999) é uma solução em software que provê a detecção de erros de fluxo de controle em algoritmos. Ela consiste na divisão do código do programa em intervalos livres de desvio (*branch free intervals* - BFIs), na inserção de dois identificadores para cada um dos BFIs e na verificação dos identificadores durante o período de execução do código.

O primeiro identificador é denominado identificador do intervalo livre de desvio (*branch-free interval IDdentifier* - BID) e armazena um valor único que identifica o BFI. Já o segundo, denominado identificador de fluxo de controle (*control flow IDentifier* - CFID), representa os desvios permitidos, ou seja, indica se os desvios de fluxo de controle estão ocorrendo na seqüência correta. O CFID é armazenado em duas filas de elementos que são inicializadas 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. A fila que monitora o valor CFID provê a redução da média de latência de detecção da falha.

A figura 6.5 mostra a estrutura IF-THEN-ELSE acrescida das instruções de verificação da técnica CCA.

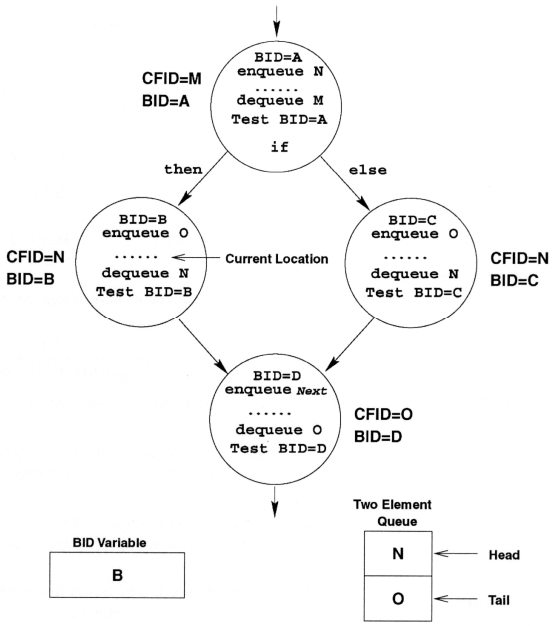


Figura 6.5 Instruções e verificação dos IDs para estrutura IF-THEN-ELSE com CCA. (ALKHALIFA, 1999)

Quanto à cobertura de falhas, a técnica acima descrita é capaz de detectar todos os erros de fluxo de controle simples e a maioria dos erros múltiplos. Basicamente, as falhas podem gerar:

- um desvio de dentro de um BFI;.
- um desvio para dentro de um BFI;.
- um desvio para um BFI ilegal, ou seja, um desvio para um BFI que não pertence ao grupo de sucessores do BFI atual.

Técnica ECCA (Enhanced Control Flow using Assertions)

Segundo Z. Alkhalifa (ALKHALIFA, 1997), a técnica ECCA é uma solução em software capaz de detectar erros no fluxo de controle de algoritmos que se baseia fundamentalmente em inserir instruções de teste e atualização no código da aplicação a fim de torná-lo tolerante a falhas e conseqüentemente mais confiável e robusto.

A aplicação da técnica ECCA é feita a partir da realização dos seguintes passos:

1. dividir o programa em um conjunto de intervalos livres de desvio (*branch free intervals* - BFI´s)

2. atribuir um número primo único, denominado identificador de intervalos livres de desvio (*branch free intervals IDentifier* – BID), para cada BFI.

3. inserir no início do BFI a instrução (6.2) e no fim a instrução (6.3).

$$id \leftarrow \frac{BID}{\overline{\overline{(id \bmod BID)}}.(id \bmod 2)} \tag{6.2}$$

$$id \leftarrow NEXT + \overline{\overline{(id - BID)}} \tag{6.3}$$

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, ou seja, BID armazena 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, um programa escrito em linguagem C possui a seguinte estrutura após ser pré-processado pela técnica ECCA.

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

A figura 6.6 apresenta a estrutura de um programa em C dividido em BFIs. A figura 6.6 representa o mesmo programa acrescido das instruções de controle definidas pela técnica, ou seja, após o pré-processamento. Finalmente a figura 6.7 mostra o diagrama de bloco do mesmo programa.

Figura 6.6 Representação do código original. (ALKHALIFA, 1997)

```
/* Início do código pré-processado*/
...
/* <BID> é 3 */
... BFI1...
id = 35+!!(id-3);
/* Observe que NEXT vale 35 e resulta da multiplicação do BID = 5 do BFI 2 com o
BID = 7 do BFI 3. */
if foo
{
/* <BID> é 5 */
                    id = 5 / ((!(id%5)*(id%2));
                    ... BFI2 ...
                    id = 11+!!(id-5);
}
else
/* <BID> é 7 */
                     id = 7/((!(id%7))*(id%2));
                    … BFI3 …
                     id = 11+!!(id-7);
}
/* <BID> é 11 */
id = 11/((!(id%11))*(id%2));
... BFI4 ...
/* Fim do código pré-processado */
```

Figura 6.7 Representação do código tolerante a falhas de acordo com a técnica ECCA. (ALKHALIFA, 1997)

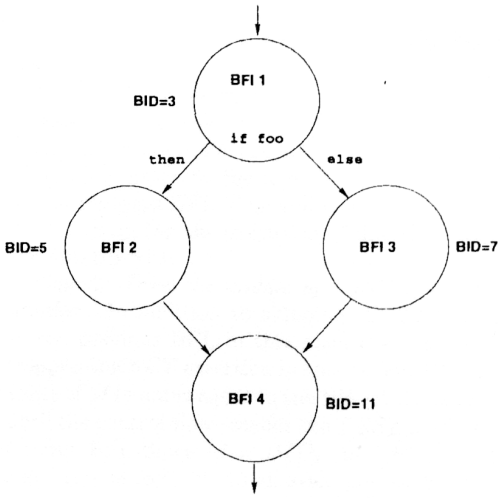


Figura 6.8 Diagrama de bloco do código tolerante a falhas de acordo com a técnica ECCA. (ALKHALIFA, 1997)

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, são elas:

- 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;

A figura 6.9 mostra a redução de erros múltiplos de fluxo de controle onde, dado três BFI´s A, B e C; um erro de fluxo de controle múltiplo pode ocorrer somente se ocorrer um desvio ilegal de dentro do BFI A para dentro do BFI B, e então do BFI B ocorrer um desvio para dentro do BFI C. Diante desta situação, o erro será classificado como um erro de A para C, ou seja, um erro de fluxo de controle simples.

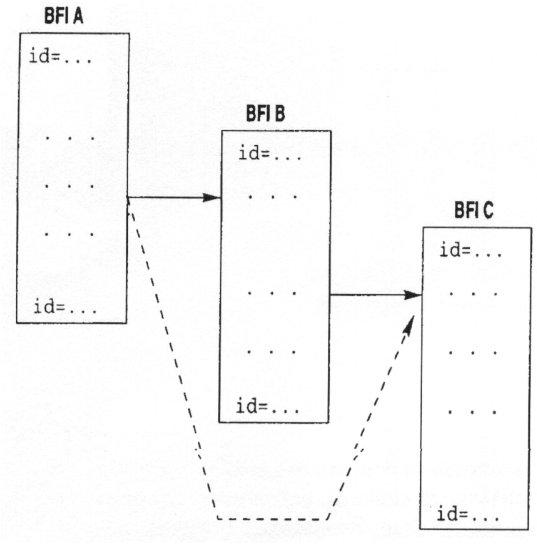


Figura 6.9 Redução de erros de fluxo de controle múltiplos.

Assim, comparando as técnicas acima descritas com as soluções baseadas em hardware anteriormente propostos, as técnicas CCA e ECCA apresentam as seguintes vantagens:

- São portáveis, ou seja, podem ser implementadas para múltiplas plataformas, pois são aplicadas em alto nível;
- Representam soluções implementadas puramente em software;
- Diminuem o overhead relacionado ao custo e ao desempenho.

Técnica CFCSS (Control Flow Checking by Software Signatures)

A técnica de CFCSS, proposta por E. McCluskey (MCCLUSKEY, 2002) é uma solução via software que provê a detecção de erros de fluxo de controle. Assim como as técnicas anteriormente descritas, a CFCSS 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.

Em relação à notação adotada, a técnica de CFCSS utiliza basicamente a terminologia mencionada anteriormente. Assim, a técnica determina que o programa alvo seja dividido em blocos básicos e que a partir desta divisão seja construído um grafo que representa o fluxo de execução do mesmo. A figura 6.10 mostra um programa e seu respectivo grafo de execução.

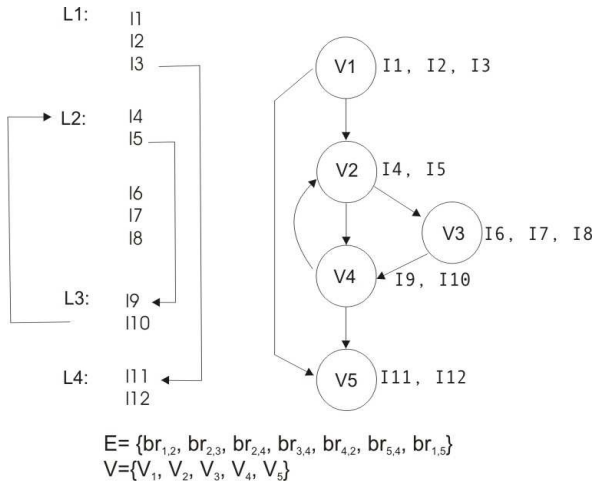


Figura 6.10 Seqüência de instruções e seu respectivo grafo. (MCCLUSKEY, 2002)

Descrição detalhada da técnica de CFCSS

A técnica de CFCSS verifica em tempo real o fluxo de controle do programa através do monitoramento e verificação das assinaturas, sendo definida pelos seguintes passos:

1. Dividir o programa em blocos básicos (*vi*);
2. Construir o grafo que representa o fluxo de execução do programa;
3. Atribuir aleatoriamente um valor único de assinatura (*si*) para cada bloco básico em tempo de compilação para representar o bloco básico *vi*;

4. 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 seguinte 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 formula: $di = sj \oplus si$.
5. Quando necessário, calcular em tempo de compilação a assinatura de ajuste (D). Esta assinatura é utilizada quando um determinado nó *vi* possui mais de um predecessor.

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 (*Global Signature Register* - GSR). 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 (6.4).

$$f = f(G, di) = G \oplus dd \text{, onde } dd = ss \oplus sd \qquad (6.4)$$

A figura 6.11 ilustra 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: $f = f(G1, d2) = G1 \oplus d2$, onde $d2 = s1 \oplus s2$ e assim $G2 = s1 \oplus (s1 \oplus s2) = s2$, ou seja, para o nó *v2*, G2 = *s2*.

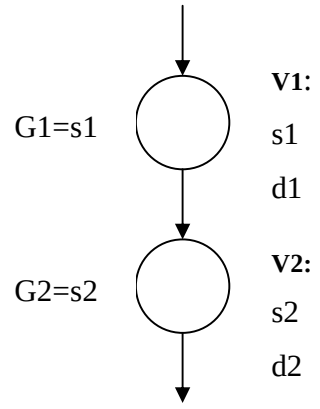


Figura 6.11 Exemplo de um desvio legal de *v1* para *v2*. (MCCLUSKEY, 2002)

A figura 6.12 ilustra 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 $G4 = s1 \oplus (s3 \oplus s4) \neq s4$, ou seja, o erro de fluxo de controle será detectado pois, G4 é diferente de *s4*.

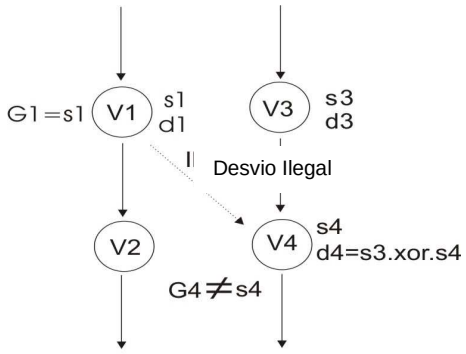


Figura 6.12 Exemplo de um desvio ilegal de *v1* para *v4*. (MCCLUSKEY, 2002)

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

1. Função assinatura: G = (G ⊕ *dk*);

2. Instrução de verificação de desvio: “Br (G ≠ *sk*) error”.

A figura 6.13 ilustra respectivamente a representação gráfica das instruções, do bloco básico, do bloco básico acrescido das instruções de verificação e a representação utilizada para um nó em um grafo de fluxo de execução.

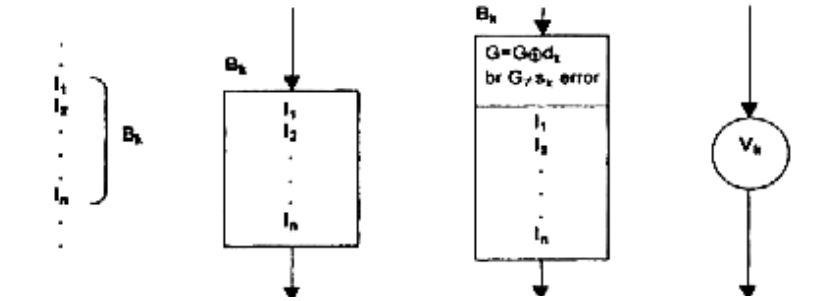


Figura 6.13 Representação gráfica. (MCCLUSKEY, 2002)

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

figura 6.14 ilustra 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*.

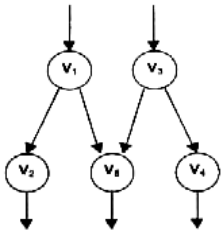


Figura 6.14. Exemplo de um bloco básico com mais de um predecessor. (MCCLUSKEY, 2002)

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 figura 6.15 ilustra um exemplo de utilização da assinatura D. Observe que no bloco básico B5 é adicionada a instrução $G = G \oplus D$ 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, $D = s1 \oplus s3$. Assim inicialmente, $d5 = s1 \oplus s5$ e D no bloco B1 apresenta o valor 0000. Assim, 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 $G = G5 \oplus D = s3 \oplus (s1 \oplus s5) \oplus (s1 \oplus s3) = s5$.

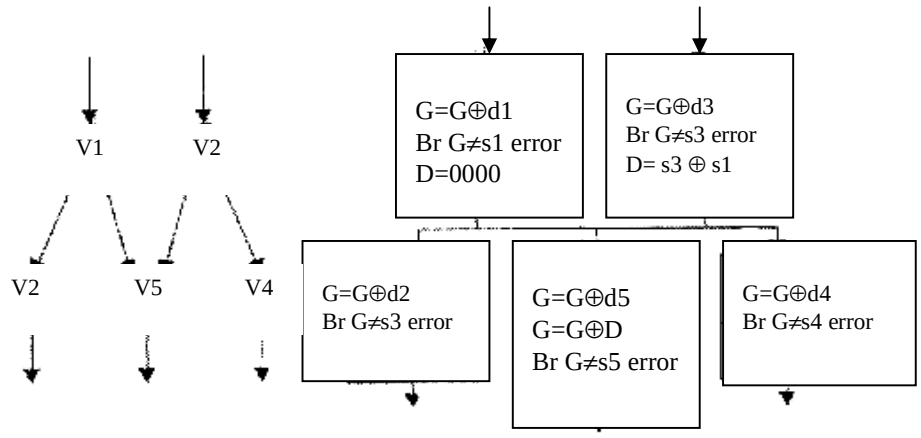


Figura 6.15 Exemplo de utilização da assinatura D utilizada para solucionar o problema de nós convergentes. (MCCLUSKEY, 2002)

Neste contexto, surge o algoritmo abaixo, 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,...,*viM*, *dj* 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 $\oplus$ D após $G = G \oplus dj$ no nó *vj*.

3.5. Inserir uma instrução "*br(G $\neq$ sj) error*" 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 $\neq$ s)error*" – 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.

E. McCluskey (MCCLUSKEY, 2002) sugere uma pequena modificação no algoritmo A, a fim de diminuir a degradação no desempenho em função da agregação desta técnica. A alteração deve ser feita no item 3.5, ou seja, ao invés de inserirmos a instrução "br(G $\neq$ *sj*) error" em todos os blocos básicos deve-se eleger alguns blocos para receberem a instrução de verificação. Porém, esta solução aumenta a latência para detecção de falhas 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) error*" seja executada pelo processador, ou mesmo entrar em *crash* e pendurar.

Técnica YACCA (Yet Another Control-Flown Checking using Assertions)

A técnica YACCA, proposta por O. Goloubeva (GOLOUBEVA, 2003) consiste em uma solução via software capaz de detectar erros de fluxo de controle através do monitoramento de assinaturas inseridas nos programas. A técnica define um conjunto de assinaturas que são geradas em tempo de compilação e uma assinatura atualizada em tempo de execução. O conjunto de assinaturas geradas em tempo de compilação são os identificadores dos blocos básicos e a assinatura gerada em tempo de execução é armazenada em uma variável dedicada e está associada ao valor do bloco básico atual.

A aplicação da técnica baseia-se nos seguintes passos:

- Dividir o programa em blocos básicos e definir seu grafo de fluxo de execução;
- Associar, em tempo de compilação, a cada bloco básico uma assinatura única;
- Inserir em cada bloco básico (*vi*) as seguintes instruções:
  - 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*);
  - 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 PREVIOUSi que contém o produto de todas as assinaturas dos nós predecessores de *vi*, equação (6.5), e a inserção da instrução de teste no início de cada bloco básico, equação (6.6).

PREVIOUSi = Π *Bj*, ∀*vj* com desvio *brj,i* ∈ *Ei* (6.5)

Onde: *Bj* corresponde a 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 (PREVIOUSi % CODE) error() (6.6)

Devido à complexidade da instrução acima, equação (6.4), seu processamento torna-se bastante crítico, por isto, são sugeridas duas soluções alternativas representadas pelas equações (6.7) e (6.8).

if ((CODE!= Ba) && (CODE!= Bb) && (...) && (CODE!= Bn)) error() (6.7)

Onde: Ei = {bra,i; brb,i; ...; brn,i}

ERR_CODE |= ((CODE!= Ba) && (CODE!= Bb) && (...) && (CODE!= Bn)) (6.8)

Onde: Ei = {bra,i; brb,i; ...; brn,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 (6.7).

code = (*code* & M1) ⊕ M2 (6.7)

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* ⊕ *Bi*

Assim *code* = *code* ⊕ (*Bj* ⊕ *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, $code = (code \,\&\, (Bj \overline{\oplus} Bk)) \oplus (Bj \,\&\, (Bj \overline{\oplus} Bk) \oplus Bi))$

A figura 6.16 mostra uma aplicação modificado a partir das regras de transformação de código definidas pela técnica YACCA.

```
ERR_CODE |= (code != 0);
code = code ^ 1; /* code = 1 */
x0 = 1;
y = 5;
i = 0;
while( i < 5 ) {
        ERR_CODE |= (code != 1) && (code != 3);
        code = (code & 5) ^ 3; /* code = 2 */
        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;
```

Figura 6.16 Código modificado a partir da técnica YACCA.

Quanto à cobertura de falhas, esta técnica é capaz de detectar os seguintes tipos de erros de controle:

- um desvio de *vi* para o bloco básico *vj* que, por sua vez, não pertence ao conjunto de *suc*(*vi*);
- um desvio de *vi* para o início do bloco básico *vj* que, por sua vez, pertence ao conjunto de *suc*(*vi*);
- um desvio de *vi* para algum lugar do bloco *vj* que pertence ao conjunto de *suc*(*vi*).

PARTE II – METODOLOGIA

7. DESCRIÇÃO DAS IMPLEMENTAÇÕES DO I-IP

Introdução

De acordo com os capítulos anteriores, é possível concluir que as técnicas de tolerância a falhas podem ser divididas sinteticamente em dois grandes grupos: técnicas baseadas em hardware e técncias baseadas em software. Dentre as soluções baseadas em hardware, descritas detalhadamente no capítulo 4, salienta-se o uso do *watchdog processor* que consiste em agregar um processador extra para monitorar constantemente a atividade realizada pelo microprocessador e ativar um procedimento de tratamento de erro caso o seu comportamento seja diferente do esperado.

Quanto às soluções baseadas em software, descritas detalhadamente no capítulo 6, salienta-se que elas representam uma solução bastante viável para aplicações críticas, pois não exigem nenhuma modificação no hardware e conseqüentemente minimizam significativamente o custo associado a sua implementação. Os setores automotivo e biomédico são exemplos clássicos de áreas que utilizam essas soluções. Assim, quando métodos como (CHEYNET, 2000) e (NAHMSUK, 2002) são explorados, processadores *commercial-of-the-shelf* (COTS) são normalmente utilizados para executarem as versões especiais do software, ou seja, versões tolerantes a falhas. Entretanto, apesar de serem bastante eficientes na detecção de falhas transientes e permanentes, técnicas baseadas em software podem introduzir significativa degradação de desempenho e considerável *overhead* na memória de código e dados.

Assim, núcleos IP são considerados atualmente uma extremamente eficiente para agregar tolerância a falhas e confiabilidade a sistemas em *chips* (*system-on-chips* – SoCs). Esses núcleos IP podem ser implementados à parte dos módulos que executam as funções gerais do sistema (memórias, processadores, etc) ou ainda, adicionados aos SoCs para agregarem e ou garantirem algumas funções extras (auxílio no *debbuging*, suporte de teste, etc). Normalmente, estes núcleos são chamados de *Infrastructure IP-cores* (I-IPs).

Neste contexto, este trabalho investiga a possibilidade de utilizar um I-IP para agregar segurança e confiabilidade aos SoCs utilizados em aplicações críticas e principalmente, busca especificar um método que possa ser facilmente inserido no fluxo de controle do SoC, ou seja,

que exija pequenas mudanças no hardware e simplifique significativamente o software relacionado ao método de detecção implementado puramente em software. Os resultados desta investigação apontam para uma solução híbrida que combina técnicas baseadas em software com técnicas baseadas em hardware. Assim, a solução proposta neste trabalho, combina os benefícios de ambas as técnicas, ou seja, um baixo custo de desenvolvimento e baixo *overhead* relacionado ao desempenho. Outra característica bastante importante está relacionada à flexibilidade do método, ou seja, uma eventual mudança na aplicação não exige que o hardware seja modificado. Especificamente neste trabalho, a solução híbrida proposta baseia-se nas técnicas de tolerância à falhas propostas em (REBAUDENGO, 2004) e (GOLOUBEVA, 2003).

No capitulo atual serão apresentadas e discutidas as soluções híbridas propostas para a detecção de falhas em dados e falhas de fluxo de controle e, detalhadas as versões do I-IP implementadas para monitorar o microcontrolador 8051 da Intel durante seu período de funcionamento. Em resumo, a *primeira versão* implementada do I-IP é um protótipo que provê apenas a detecção de falhas em dados e, como todo protótipo, este também apresenta algumas restrições e particularidades. A *segunda versão* também é capaz de detectar falhas em dados, entretanto supera as restrições da versão anterior através de um remodelamento da arquitetura básica do I-IP e da inserção de um módulo de memória. A *terceira versão* do I-IP agrega a *segunda versão*, a capacidade de detectar falhas de fluxo de controle. E por fim, a *quarta versão* agrega algumas técnicas de tolerância a falhas ao I-IP desenvolvido.

Entretanto, antes da apresentação detalhada da solução híbrida proposta, serão discutidas as principais limitações das técnicas que serão utilizadas como base para a realização deste trabalho.

Limitações das soluções baseadas em software

Assim como as técnicas baseadas em hardware, as soluções propostas em (REBAUDENGO, 2004) e (GOLOUBEVA, 2003), também agregam algumas penalidades. Basicamente, estas soluções reduzem significativamente o desempenho, agregam *overhead* de memória e provêem uma cobertura de falhas limitada. Neste sentido, estes aspectos serão descritos e analisados detalhadamente a seguir.

A) Degradação do desempenho:

A queda no desempenho pode ser avaliada a partir da análise do *overhead* do tamanho do código do programa original e relação ao programa tolerante a falhas.

Assim, dado um programa P dividido em *n* blocos básico e representado por P = {B1, B2,..., Bn}, seu tamanho é definido a partir da equação (7.1).

$$size(P) = \sum_{i=1}^{n} (OI_i + BI_i) \tag{7.1}$$

Onde: OIi representa o número de instruções operativas dentro do bloco básico Bi e BIi representa as instruções de desvio.

A aplicação das regras de transformação de código definidas em (CHEYNET, 2000) e (GOLOUBEVA, 2003) modificam o tamanho do programa tolerante a falhas HP, conforme a equação (7.2).

$$Size(HP) = \sum_{i=1}^{n} (2 \cdot OI_i + BI_i + DC_i + CC_i + TI_i + SI_i) \tag{7.2}$$

Onde: DCi representa as instruções extras introduzidas para a verificação dos dados; TIi representa as instruções extras introduzidas pelas instruções de teste e SIi as instruções introduzidas pelas instruções de atualização.

Assim, RCS é definido como a razão entre o tamanho do código original e o código tolerante a falhas. Neste contexto, dois extremos são considerados nas equações (7.3) e (7.4).

$$\lim_{Average(OIi) \to \infty} (RCS) = 2 \tag{7.3}$$

$$\lim_{Average(OIi) \to 1} (RCS) = N \text{ com M >= 2} \tag{7.4}$$

Em vista do exposto, é possível concluir que a queda no desempenho se deve ao *overhead* agregado ao tamanho do código, pois RCS será sempre maior que 2.

B) Overhead de memória:

O *overhead* de memória no programa original, definido pela equação (7.5), é calculado a partir da área da memória de código e da memória dos dados.

$$Size(P) = Size(Code) + Size(Data) \quad (7.5)$$

Entretanto, no programa tolerante a falhas, o *overhead* de memória é dado pela equação (7.6).

$$Size(HP) = RCS * Size(Code) + RDS * Size(Data), \text{ onde RDS} \approx 2. \quad (7.6)$$

Assim, avaliação do *overhead* de memória é feita com base em RCD, que corresponde à razão entre o tamanho do código e dos dados no programa original e RM, que é definido como a razão entre o programa original e o programa tolerante a falhas. RM depende de RCD e assim, dois extremos são considerados nas equações (7.7) e (7.8).

$$\lim_{RCD \to \infty} (RM) = 2 \quad (7.7)$$

$$\lim_{Average(OIi) \to \infty} (RM) = RCS \quad (7.8)$$

C) Cobertura de falhas limitada:

As instruções extras inseridas para a verificação dos dados e do fluxo de execução, chamadas de instruções de teste, introduzem desvios adicionais e conseqüentemente blocos básicos extras. Essas instruções, caso sejam corrompidas por uma falha, podem causar desvios ilegais que podem não ser detectados caso o desvio ocorra dentro do mesmo bloco básico. Esta situação é extremamente crítica quão maior é o tamanho do bloco básico.

Em vista do exposto, a fim de superar os problemas acima mencionados, este trabalho propõe uma solução híbrida para a detecção de falhas em dados e falhas de fluxo de controle.

Técnica híbrida proposta para a detecção de falhas de dados

Introdução

A metodologia proposta neste trabalho explora regras de transformação de código baseadas na técnica proposta em (REBAUDENGO, 1999) e (CHEYNET, 2000) para introduzir automaticamente informação e operações redundantes na parte implementada em software e um I-IP para executar os testes de consistência na informação redundante, na parte implementada em hardware. Em resumo, o I-IP realizará um monitoramento constante do barramento do processador a fim de verificar se a informação redundante é consistente, ou seja, se as cópias de uma determinada variável possuem ou não os mesmos valores. Caso seja encontrada uma discrepância entre os valores, um sinal de erro deve ser ativado.

Descrição detalhada da técnica

Esta seção descreve as etapas de definição e de implementação das porções que implementarão as partes de software e de hardware nas quatro versões apresentadas e detalhadas a partir da seção 7.5. Dado um algoritmo qualquer, a solução baseada em software exige a execução dos seguintes passos:

1. Todas as variáveis existentes no algoritmo devem ser duplicadas;
2. Todas as operações presentes no algoritmo devem ser duplicadas;
3. Testes de consistência devem ser executados, durante a execução do programa, após a leitura das duas cópias das variáveis. Caso uma discrepância seja observada, um sinal de erro deve ser ativado.

Salienta-se que o último passo consome um tempo de processamento bastante significativo, pois deve ser executado após a leitura das cópias de cada uma das variáveis duplicadas. Assim, com o intuito de reduzir o tempo consumido durante os testes, este trabalho define que o último passo seja implementada em hardware e os passos 1 e 2 em software.

Parte implementada em software:

Conforme anteriormente mencionado, os passos 1 e 2, acima descritos, devem ser aplicados ao código fonte da aplicação que deve ser protegida. Assim, a figura 7.1 ilustra claramente o código resultante após a aplicação das regras de transformação de código ao programa original.

Código original	Código modificado
int a, b;	int a1, a2, b1, b2;
...	...
a = b + 5;	a1 = b1 + 5;
	a2 = b2 + 5;

Figura 7.1.Comparação entre código original e código tolerante a falhas.

Parte implementada em hardware:

Quanto à parte implementada em hardware, passo 3 anteriormente descrito, um I-IP *ad hoc* foi desenvolvido para monitorar o barramento do processador embutido no SoC e, verificar continuamente se as variáveis lidas são consistentes. Assim, quando uma discrepância entre os valores das duas cópias de uma determinada variável é encontrada, um sinal é ativado e um procedimento de tratamento de erro é ativado.

Técnica híbrida proposta para a detecção de falhas de fluxo de controle

Introdução

Até o momento, várias técnicas capazes de detectarem erros que afetam o fluxo de controle de um programa já foram propostas na literatura (ALKHALIFA, 1999), (KANAWATI, 1996), (NAHMSUK, 2002), etc. Esses erros ocorrem quando o processador busca e executa uma instrução incorreta e isto modifica o fluxo de controle esperado. Normalmente, estes erros resultam de falhas no processador ou na memória que armazena o programa. Assim como as técnicas que provêem a detecção de erros em dados, essas soluções podem ser implementadas via

hardware ou software. Evidentemente que quando implementadas por hardware, essas técnicas exigem que seja agregado ou modificado o hardware existente e conseqüentemente comprometem sua portabilidade além de implicarem em um custo de implementação mais elevado. Entretanto, quando implementadas por software, essas técnicas podem ser aplicadas diretamente no código fonte, não exigem hardware extra, portanto resultam em um menor custo, e garantem total portabilidade para diferentes plataformas.

De acordo com o capítulo 6, as técnicas baseadas em software consistem em dividir o programa em blocos básicos, associar aleatoriamente assinaturas para cada um dos blocos e por fim, durante a execução do programa, comparar a assinatura pré-computada com a assinatura gerada em tempo de execução. Entretanto, soluções baseadas em software geram um *overhead* de tempo bastante significativo e aumentam consideravelmente o tamanho do código devido às instruções adicionadas.

Neste contexto, uma solução híbrida, capaz de detectar erros de fluxo de controle, baseada na técnica descrita em (GOLOUBEVA, 2003) é proposta.

Descrição detalhada da técnica

A solução híbrida proposta para detecção de erros de fluxo de controle explora uma versão modificada do código fonte do programa em combinação com um I-IP. Para isto, o fluxo de controle do programa é monitorado através da inserção de instruções específicas dentro do seu código fonte a fim de indicar ao I-IP a parte da aplicação que está em execução. Essas instruções enviam informações para o I-IP cada vez que o fluxo de controle entra ou sai de um bloco básico do programa e finalmente o I-IP verifica se a evolução do programa corresponde ao comportamento esperado. 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, utiliza-se uma assinatura gerada em tempo de execução, denominada *Bi* para representar o bloco *vi* que está em execução.

Antes da descrição detalhada da solução híbrida para detecção de erros de fluxo de controle, será brevemente descrita a solução baseada puramente em software. Assim, dado um

algoritmo, o método exige que as instruções abaixo sejam introduzidas dentro de cada um dos blocos básicos:

- Instrução de teste: esta instrução controla a assinatura do bloco básico predecessor e verifica, de acordo com o grafo do programa, se ele pertence ao conjunto *pred(vi)*;
- Instrução de atualização: esta instrução atualiza a assinatura, ou seja, calcula em tempo de execução o valor *Bi* correspondente ao bloco básico *vi*. Este cálculo depende basicamente da assinatura do nó atual. Assim, dado um bloco básico vi, a formula geral é dada por: *code = (code ^ M1 ⊕ M2)*, onde M1 representa um valor constante que depende das assinaturas dos nós que pertencem ao conjunto *pred(vi)* e *M2* representa um valor constante dependente da assinatura atual, ou seja, de *vi* e das assinaturas dos nós que pertencem ao conjunto *pred(vi).*

Entretanto, a solução proposta sugere que parte da técnica acima descrita seja implementada parte em software e parte em hardware.

Parte implementada em software:

A técnica híbrida proposta para a detecção de erros de fluxo de controle determina que parte da técnica seja implementada em software e parte hardware. As modificações implementadas em software são efetuadas no código fonte do programa. Assim, dado o código fonte do programa cada bloco básico *vi* deve ser modificado conforme as instruções abaixo:

- Para cada bloco *vj* ∈ *pred(vi)*, uma instrução *IIPteste(Bj)* deve ser agregada ao bloco *vi*. Conforme já mencionado, esta instrução envia a assinatura *Bj* associada com o bloco *vj* para o Pandora. Assim, após receber o valor, Pandora verifica se *Bj* é diferente do valor corrente da assinatura do programa nela armazenado. Caso seja identificada alguma discrepância entre os dois valores, um *flag* de erro é setado com TRUE.
- A instrução IIPset(*Bi*) é adicionada ao final do bloco vi. Esta instrução envia ao Pandora, o novo valor da assinatura do bloco básico. Antes de receber o valor, caso o flag de erro seja igual a TRUE um sinal de erro é setado; caso contrário, Pandora

calcula os novos valores de M1 e M2 para o bloco básico *vi* a partir dos valores *Bj* e *Bi* recebidos e assim atualizar a assinatura do programa.

A figura 7.2 apresenta um código fonte de um programa modificado a partir das regras de transformação de código acima descritas.

```
01: while(k1<DIM)
02: {
03:     IIPtest( BB1 );
04:     IIPtest( BB2 );
05:     A1 = matrixA1[i1][k1];
06:     B1 = matrixB1[k1][j1];
07:     C1 += A1*B1;
08:     matrixC1[i1][j1] = C1;
09:     k1++;
10:     IIPset( BB2 );
11: }
```

Figura 7.2 Programa modificado a partir das regras da solução híbrida.

A partir da análise da figura 7.2, observa-se que a solução híbrida proposta, exige pequenas modificações no código original do programa.

Parte implementada em hardware:

Quanto a parte implementada em hardware, um I-IP , denominado *Pandora*, foi desenvolvido para monitorar o funcionamento do processador embutido no SoC e verificar se a execução do programa segue o fluxo esperado. Finalmente, quando um erro de fluxo de controle for detectado, um sinal será ativado e um procedimento de tratamento de erro será executado.

Descrição da primeira versão do I-IP

Introdução

Conforme anteriormente mencionado, a primeira versão do I-IP trabalha em conjunto com o microcontrolador 8051 da Intel, monitorando seu barramento a fim de verificar a consistência das cópias das variáveis lidas durante a execução do programa. Essas variáveis, são armazenadas

na memória RAM do SoC e, sendo assim o I-IP deve monitorar todos os ciclos de leitura da RAM. A figura 7.3 mostra o diagrama de bloco do núcleo do microcontrolador 8051 da Intel utilizado nesta implementação.

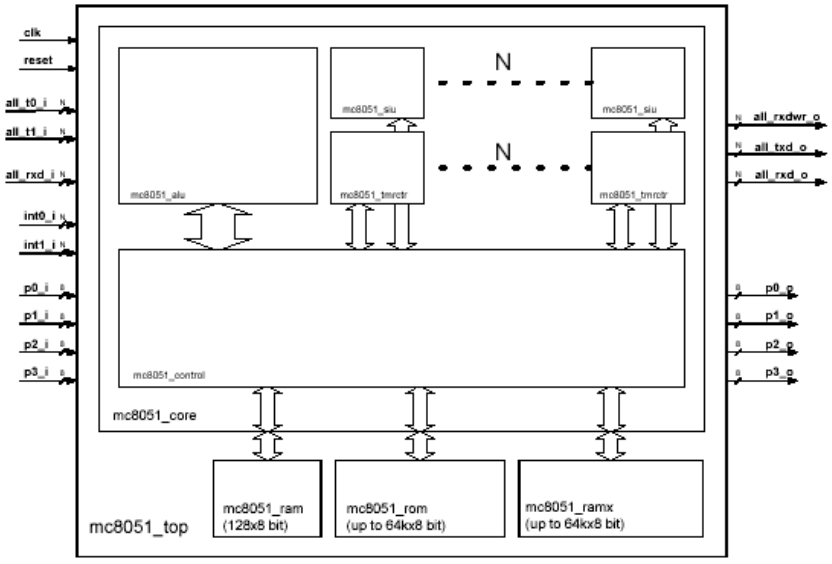


Figura 7.3 Diagrama de bloco do núcleo do microcontrolador 8051 da Intel.

Esta primeira versão, assim como as demais, utiliza um SoC com um processador COTS e dois módulos de memória COTS, memória de código (*code memory* – CM) e memória de dados (*data memory* – DM). A CM armazena o código binário do software que deve ser executado pelo processador e a DM os dados manipulados pelo software. Salienta-se que nenhum mecanismo de detecção ou correção de erros foi agregado aos módulos de memória e que o processador não possui memória cache. Todos os detalhes internos sobre o processador e os módulos de memória não são levados em consideração, pois se parte do pressuposto que eles não são acessíveis. Convencionalmente, os detalhes a respeito da interface do barramento que conecta o processador com as memórias são conhecidos. Também, neste trabalho não são consideradas aplicações que utilizam memórias de alocação dinâmica. A figura 7.4 mostra o diagrama de bloco do SoC acrescido do I-IP que será implementado neste trabalho.

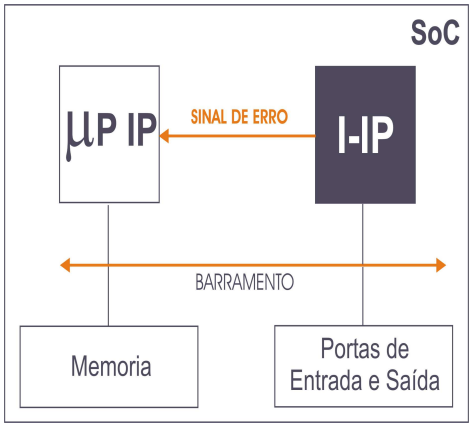


Figura 7.4 Diagrama de bloco do SoC acrescido do I-IP.

Arquitetura básica do I-IP

Nesta primeira versão, o I-IP foi particionado em dois módulos: lógica de interface com o barramento (*bus interface logic*) e a lógica de verificação de consistência (*consistency check logic)*. O primeiro módulo implementa a interface necessária para acessar o barramento do processador, no caso o microcontrolador 8051 da Intel. Assim, diante de um ciclo de leitura, este módulo busca e armazena o valor da variável presente no barramento. Após armazenar os valores das variáveis duplicadas, a lógica de interface do barramento envia-os à lógica de verificação de consistência. Na seqüência, o segundo módulo implementa os testes de consistência necessários para verificarem se o dado armazenado na memória foi afetado por uma determinada falha. Esta verificação ocorre somente após o primeiro módulo capturar no barramento os valores, ou seja, as duas cópias de uma determinada variável lida e enviá-los para a lógica de verificação de consistência. Caso os valores sejam diferentes, um sinal de erro é ativado. A figura 7.5 mostra o diagrama de bloco da primeira versão do I-IP.

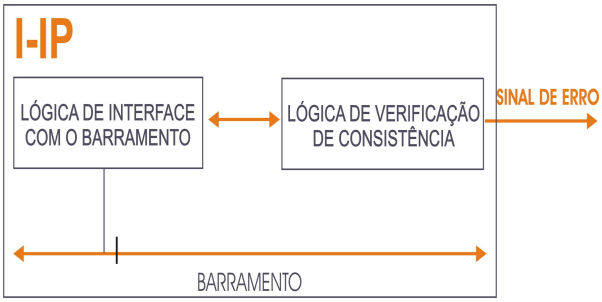


Figura 7.5 Diagrama de bloco da arquitetura do I-IP.

Descrição do funcionamento da primeira versão do I-IP

A primeira versão implementada do I-IP funciona basicamente conforme os seguintes passos:

- Passo 1: O I-IP deve monitorar constantemente a execução do programa a fim de buscar as instruções que fazem acesso à memória RAM do 8051, a fim de obter a primeira cópia da variável.
- Passo 2: Quando ocorrer um ciclo de leitura, o I-IP deve buscar no barramento do microcontrolador a primeira cópia da variável duplicada e armazená-la em um buffer temporário (*dado_1*);
- Passo 3: Após o armazenamento do *dado_1*, o I-IP deve esperar pela próxima instrução que faz acesso a memória RAM a fim de obter a segunda cópia da variável duplicada;
- Passo 4: Quando ocorrer outro ciclo de leitura, o I-IP deve buscar no barramento do microcontrolador a segunda cópia da variável duplicada e armazená-la em outro buffer temporário (*dado_2*);
- Passo 5: Assim, após obter os dois valores da variável duplicada, ou seja, *dado_1* e *dado_2*, eles devem ser comparados e caso sejam diferentes um sinal deve ser ativado a fim de indicar a ocorrência de um erro.

Considerações à cerca da primeira versão do I-IP

Esta primeira versão do I-IP apresenta algumas limitações, pois é capaz de monitorar somente um pequeno conjunto de instruções que lêem o conteúdo da memória RAM e, além disso, considera que as leituras das variáveis duplicadas são executadas consecutivamente e que estas estão armazenadas na RAM em endereços consecutivos. Assim, dado *a1* e *a2*, pressupõe-se que a leitura de *a2* será realizada logo após a leitura de *a1* e entre uma leitura e outra, nenhum outro ciclo de leitura acessará a memória RAM.

A tabela 7.1 mostra a lista de todas as instruções do 8051 que lêem o conteúdo da memória RAM, ou seja, lêem os dados armazenados na memória. A coluna “código binário” apresenta a

representação binária das instruções e a coluna símbolo à respectiva representação simbólica utilizada pelo núcleo VHDL do 8051. Já a coluna mais à direita, apresenta os respectivos *mneumônios assembler* das instruções do microcontrolador 8051 da Intel.

Código binário	Símbolo do núcleo	Mnemônio
00100101	**ADD_A_D**	**ADD A,diretto**
0010011	**ADD_A_ATRI**	**ADD A,@Ri**
00110101	**ADDC_A_D**	**ADDC A,diretto**
0011011	ADDC_A_ATRI	ADDC A,@Ri
10010101	**SUBB_A_D**	**SUBB A,diretto**
1001011	SUBB_A_ATRI	SUBB A,@Ri
01010101	**ANL_A_D**	**ANL A,diretto**
0101011	ANL_A_ATRI	ANL A,@Ri
01000101	**ORL_A_D**	**ORL A,diretto**
0100011	ORL_A_ATRI	ORL A,@Ri
01100101	**XRL_A_D**	**XRL A,diretto**
0110011	XRL_A_ATRI	XRL A,@Ri
11100101	**MOV_A_D**	**MOV A,diretto**
1110011	**MOV_A_ATRI**	**MOV A,@Ri**
10101	MOV_R_D	MOV Rn,diretto
10000101	MOV_D_D	MOV diretto1,diretto2
1010011	MOV_ATRI_D	MOV @Ri, diretto

Tabela 7.1. Lista das instruções do 8051 que fazem acesso à memória de dados.

Dentre as instruções apresentadas na tabela 7.1, somente as instruções em *negrito* são monitoradas nesta primeira versão do I-IP.

A figura 7.6 mostra o diagrama de bloco do 8051 acrescido do módulo do I-IP.

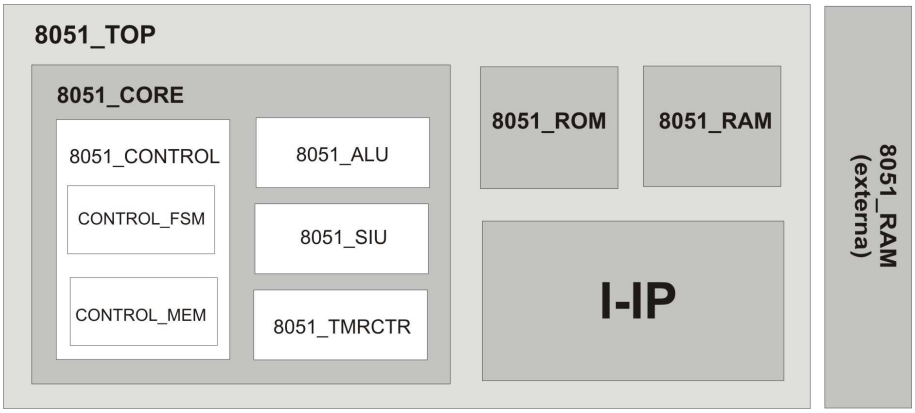


Figura 7.6 Diagrama de bloco do 8051 acrescido do I-IP.

E, por fim a figura 7.7 mostra o diagrama de bloco detalhado da primeira versão do I-IP. Salienta-se que os sinais de entrada do I-IP são capturados a partir do barramento do núcleo do microcontrolador 8051 utilizado.

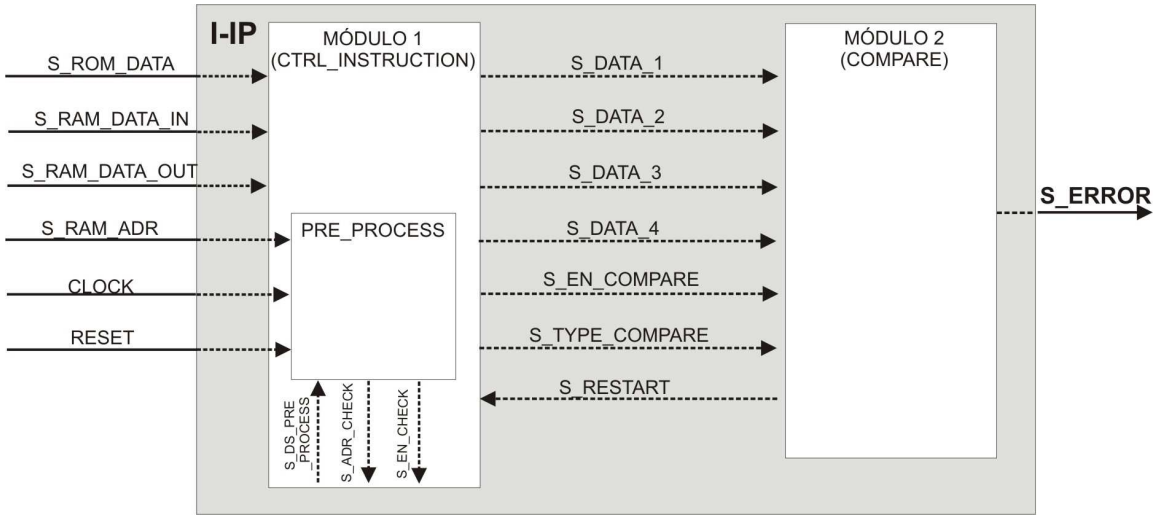


Figura 7.7 Diagrama de bloco detalhado da primeira versão do I-IP.

A partir da análise do diagrama de bloco da figura 7.7 conclui-se que o I-IP monitora seguintes sinais:

- s_rom_data: sinal que armazena o código da instrução que está sendo executada pelo 8051.
- s_ram_data_out: sinal que armazena o dado que sai da memória RAM, ou seja, o dado lido.
- s_ram_data_in: sinal que armazena o dado que entra na memória RAM, ou seja, o dado que será escrito.
- s_ram_adr: sinal que armazena o endereço do dado lido ou escrito na memória RAM.

Este monitoramento só é possível, pois o núcleo do 8051 utilizado está descrito em linguagem de descrição de hardware (***v****ery hight speed integrated circuit* ***h****ardware* ***d****escription* ***l****anguage* – VHDL). A primeira versão do I-IP foi escrita em linguagem VHDL e possui cerca de 655 linhas de código.

Descrição da segunda versão do I-IP

Introdução

Assim como a primeira versão do I-IP, esta também trabalha em conjunto com microcontrolador 8051 e monitora seu barramento durante os ciclos de leitura da memória de dados. Esta implementação recebeu o nome de *Cerberus* e executa as mesmas funções desempenhadas pela versão anterior. Entretanto, o Cerberus supera totalmente as limitações e restrições da versão anterior, descritas na seção 7.5.4, ou seja, esta solução não exige que os ciclos de leitura das cópias das variáveis duplicadas sejam consecutivos.

Arquitetura básica do Cerberus

A implementação do *Cerberus*, parte do pressuposto que as variáveis utilizadas pelo programa em execução no microcontrolador 8051 estão ordenadas na memória de dados conforme mostra a figura 7.8. Basicamente, o segmento de dados foi dividido em duas partes, onde a parte superior da memória armazena a primeira réplica das variáveis duplicadas e a parte inferior a segunda réplica.

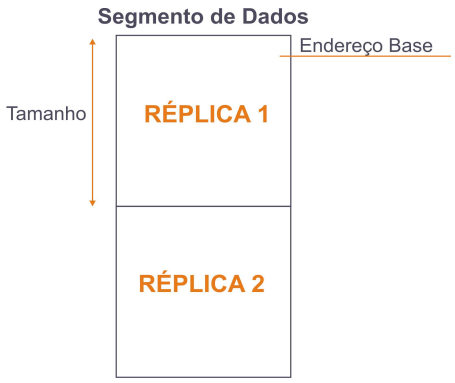


Figura 7.8 Divisão dos dados armazenados na memória.

Este ordenamento facilita sobremaneira a identificação das réplicas das variáveis a partir do endereço, da base e do tamanho. O endereço indica a posição da variável armazenada na memória

de dados, a base indica o início do segmento de dados e finalmente, o tamanho indica a dimensão de cada parte que forma a memória de dados.

O *Cerberus* foi particionado em três módulos distintos: lógica de interface com o barramento (*bus interface logic*), lógica de verificação de consistência (*consistency check logic*) e memória CAM (*CAM*[1] *memory*). Os dois primeiros módulos desempenham as mesmas funções da primeira versão. Assim, a diferença básica entre o *Cerberus* e a versão anterior é a presença de um bloco de memória. Esta memória é indexada pelo endereço da variável lida e armazena a primeira réplica de cada variável duplicada lida a partir da memória RAM. A figura 7.9 mostra o diagrama de bloco do *Cerberus*.

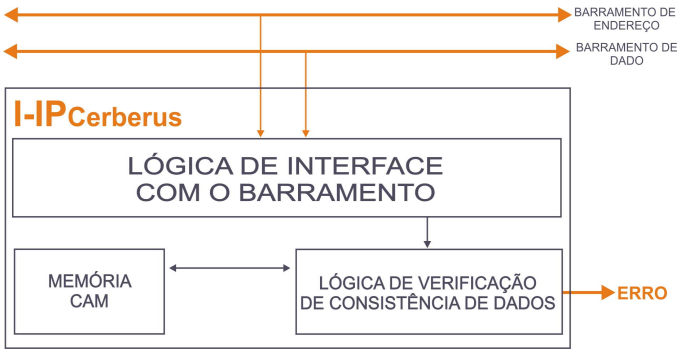


Figura 7.9 Diagrama de bloco da arquitetura do *Cerberus*.

Descrição do funcionamento do Cerberus

O *Cerberus* funciona basicamente conforme os seguintes passos:

- Passo 1: O *Cerberus* deve monitorar constantemente a execução do programa a fim de buscar as instruções que fazem lêem a memória RAM do 8051;
- Passo 2: Quando ocorrer um ciclo de leitura, ele deve buscar no barramento do microcontrolador a variável lida e seu respectivo endereço e armazená-los em *buffers* temporários;

[1] *Content-Addressble Memory* - CAM, este tipo de memória é endereçavel pelo conteúdo e executa uma determinada busca em apenas um ciclo de clock.

- Passo 3: Após o armazenamento dos dados acima descritos, o *Cerberus* deve analisar o endereço da variável a fim de verificar se ela é a primeira ou a segunda réplica.

  Passo 3.1: Caso a variável lida representa a primeira réplica, o *Cerberus* deve armazená-la na memória CAM e continuar o monitoramento do barramento do processador (passo 2);

  Passo 3.2: Entretanto, caso a variável representa a segunda réplica, o *Cerberus* deve buscar na memória CAM a sua correspondente cópia e armazená-la em um buffer temporário;
- Passo 4: Assim, após obter os dois valores da variável duplicada, o *Cerberus* deve compará-los e caso sejam diferentes, um sinal de ser ativado a fim de indicar a ocorrência de um erro.

Salienta-se que após a comparação das réplicas de uma determinada variável, a mesma deve ser removida da memória CAM.

Considerações à cerca do Cerberus

Esta segunda implementação do I-IP supera as restrições e limitações da versão anterior, pois leva em consideração todas as instruções do 8051 que lêem a memória de dados e não pressupõe que as variáveis duplicadas estão armazenadas na RAM em endereços consecutivos. Além disso, o *Cerberus* pode ser facilmente adaptado a diferentes processadores, pois a memória CAM e a lógica de verificação de consistência são paramétricas podendo ser facilmente adaptadas a diferentes tamanhos de endereços e de dados. Portanto, somente a lógica de interface com o barramento é que necessita ser reescrita e readaptada aos ciclos de leitura e escrita do novo processador alvo.

Assim como a primeira versão do I-IP, o *Cerberus* também foi escrito em VHDL e possui aproximadamente 700 linhas de código.

Descrição da terceira versão do I-IP

Introdução

A terceira versão do I-IP consiste na união do *Cerberus*, I-IP implementado na seção anterior, com o *Pandora*, porção de hardware que implementa a solução híbrida proposta para a detecção de erros no fluxo de controle. Nas seções seguintes serão detalhados a arquitetura e o funcionamento do I-IP proposto para a detecção de erros de fluxo de controle e, em seguida será apresentada detalhadamente a terceira versão do I-IP que consiste na integração de *Cerberus* e *Pandora*.

Arquitetura básica do Pandora

A exemplo do I-IP *Cerberus*, a arquitetura do I-IP proposto para detectar erros de fluxo de controle foi particionada em dois módulos: lógica de interface com o barramento (*bus interface logic)* e lógica de verificação de consistência (*consistency check logic)*. O primeiro módulo implementa a interface necessária para realizar a comunicação com o barramento do processador, no caso o microcontrolador 8051 da Intel. O segundo módulo, implementa o teste e o conjunto de instruções necessárias para verificar se algum erro de fluxo de controle afetou a seqüência de execução do programa. Caso isto tenha ocorrido, o I-IP deve informar ao sistema, através de um sinal, que um erro foi detectado. A figura 7.10 mostra a arquitetura do *Pandora*.

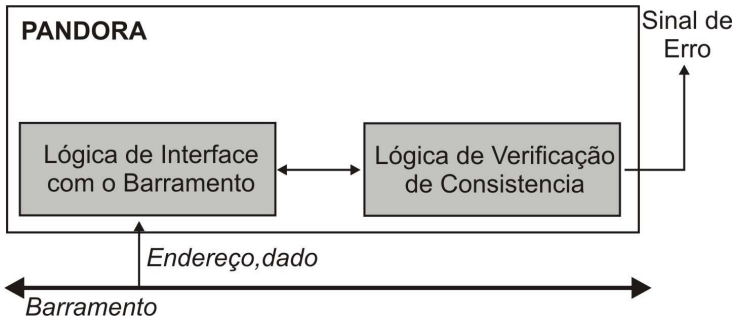


Figura 7.10 Arquitetura do I-IP, Pandora.

Descrição do funcionamento do Pandora

Inicialmente, as instruções *IIPtest()* e *IIPset()* devem ser inseridas no código fonte da aplicação respectivamente no início e no fim de cada bloco básico e, *Pandora* deve monitorar a execução da aplicação pelo micrcocontrolador 8051 da Intel em busca das mesmas. Assim, quando uma instrução do tipo *IIPtest(Bj)* é executada, *Pandora* deve armazenar o valor de *Bj* associado ao bloco básico *vj*. Após receber *Bj*, *Pandora* deve verificar se este valor difere do valor da assinatura atual, nele armazenado, e caso o valor seja diferente, deve atribuir o valor TRUE ao *flag* de erro. Finalmente, quando uma instrução *IIPset (Bi)* é executada, Pandora deve armazenar *Bi*, que corresponde a nova assinatura do bloco básico. Assim após receber o valor de *Bi*, Pandora deve verificar o conteúdo do *flag* de erro. Caso o *flag* de erro contenha o valor TRUE, um sinal de erro deve ser ativado. Caso contrário *Pandora* deve calcular os valores das máscaras M1 e M2 com base nos valores recebidos *Bj* e *Bi* e atualizar a assinatura do programa.

Considerações à cerca do Pandora

O I-IP implementado para detectar erros de fluxo de controle, assim como o *Cerberus*, trabalha em conjunto com o microcontrolador 8051 da Intel. *Pandora* também foi escrito em VHDL e possui cerca de 1500 linhas de código.

A terceira versão: Cerberus + Pandora

Em síntese, a terceira versão do I-IP é o resultado da união dos I-IPs anteriormente descritos, *Cerberus* e *Pandora*. Assim, este novo componente implementa a porção em hardware das metodologias híbridas propostas para a detecção de falhas em dados e para a detecção de falhas de fluxo de controle.

Durante a execução da aplicação tolerante a falhas, o I-IP deve verificar, durante os ciclos de leitura, a consistência das cópias das variáveis e caso uma discrepância seja encontrada o I-IP deve ativar um sinal de erro. Este particionamento simplifica significativamente o código, pois

além de reduzir seu tamanho e aumentar seu desempenho, ele remove as instruções condicionais que eram agregadas ao código na solução implementada via software. Além disso, o I-IP deve monitorar o fluxo de execução da aplicação com base nos dados recebidos a partir das instruções *IIPtest()* e *IIPset()* inseridas no código fonte da aplicação. Essas instruções enviam informações ao I-IP cada vez que o fluxo de execução entra ou sai de um bloco básico. Assim cada vez que o I-IP recebe uma informação atualizada sobre a aplicação, ele verifica se a evolução do programa é compatível com o comportamento esperado, e ativa um procedimento de manipulação de erro quando um erro é detectado.

A tabela 7.2 resume o particionamento definido na solução proposta neste trabalho.

Regras de transformação de código	**Implementação**
Duplicação das variáveis	Software
Duplicação das operações	Software
Verificação da consistência dos dados	Hardware
Verificação do fluxo de controle	Hardware

Tabela 7.2 Particionamento da solução híbrida.

Quanto a arquitetura da terceira versão do I-IP, o hardware desenvolvido para realizar a verificação dos dados consiste em um circuito extra que armazena e verifica a consistência das variáveis duplicadas armazenadas na RAM após seu ciclo de leitura. Finalmente, o hardware desenvolvido para realizar a verificação do fluxo de execução da aplicação também consiste em um circuito extra que calcula a assinatura em tempo de execução e compara-a com as assinaturas previamente armazenadas. A figura 7.11 mostra a arquitetura básica do I-IP implementado para detectar falhas em dados e falhas de fluxo de controle. Salienta-se que para zerar o conteúdo da memória CAM, um sinal de comunicação entre a lógica de consistência de dados e a de consistência de fluxo de controle, foi agregado a fim de indicar o término da execução de um determinado bloco básico.

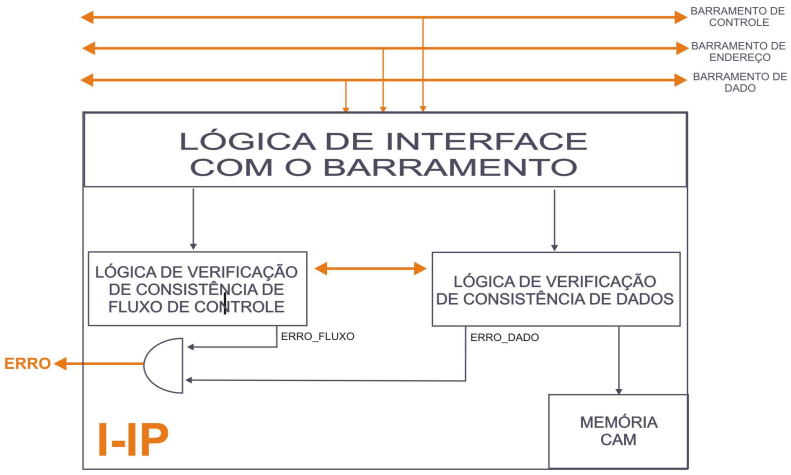


Figura 7.11 Arquitetura da terceira versão do I-IP.

Descrição da quarta versão: I-IP tolerante a falhas

Introdução

A literatura propõe uma infinidade de técnicas de tolerância a falhas que podem ser utilizadas para garantirem a confiabilidade e a segurança de sistemas. Basicamente, estas soluções podem ser classificadas em *on-line* e *off-line*. Segundo Janusz Rajski (RAJSKI, 1998) quando classificadas como *on-line* elas ainda podem ser concorrentes ou não-concorrentes. Neste contexto, a versão final do I-IP agrega alguns mecanismos de tolerância à falhas capazes de garantirem a confiabilidade dos resultados produzidos a partir do monitoramento da execução do programa pelo microcontrolador 8051 da Intel. Para isto, foram especificadas duas soluções diferentes: um teste *off-line* e um teste *on-line*.

Especificação de um teste *off-line*

Conforme anteriormente definido, um teste é dito *off-line* quando o sistema deve interromper sua execução normal e dedicar-se única e exclusivamente a execução do procedimento de teste. Neste sentido, várias metodologias de teste *off-line*, descritas no capítulo 4, podem ser exploradas com o intuito de verificar se o I-IP está funcionando corretamente.

Primeira abordagem de teste off-line:

Para ilustrar mais detalhadamente o auto-teste funcional especificado para o I-IP será utilizada a segunda versão do I-IP, ou seja, *Cerberus*. Assim, esta primeira abordagem consiste em testar funcionalmente o I-IP como uma caixa preta, ou seja, gerar as entradas necessárias para o funcionamento do mesmo e a partir delas observar as respostas obtidas após o processamento das mesmas pelo *Cerberus*. A figura 7.12 mostra o diagrama de bloco genérico do BIST off-line funcional. A figura 7.13 ilustra a metodologia de teste off-line através de um diagrama de bloco mais detalhado.

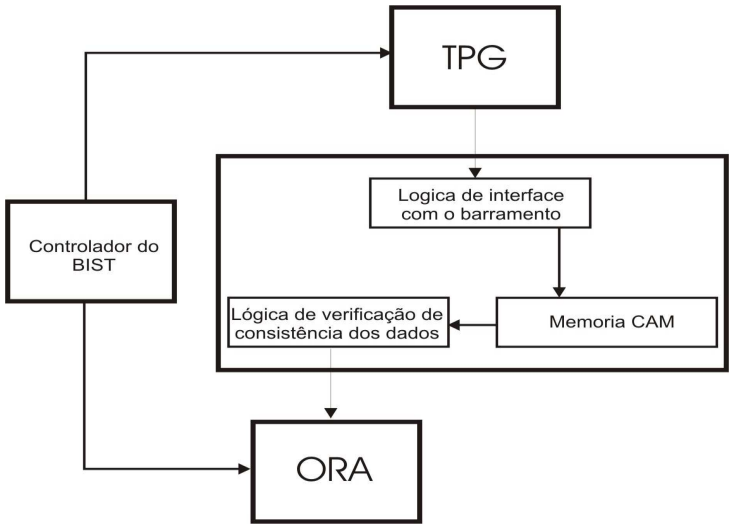


Figura 7.12 Diagrama de bloco genérico do BIST off-line funcional.

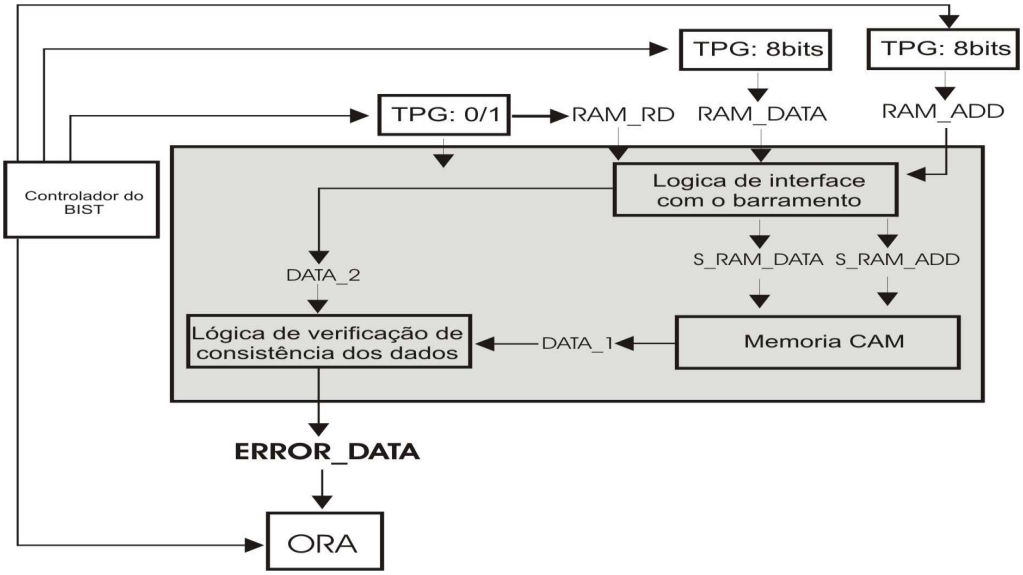


Figura 7.13 Diagrama de bloco detalhado do BIST off-line funcional.

Segunda abordagem de teste off-line:

Esta segunda abordagem consiste em testar em paralelo cada um dos módulos funcionais que compõe a arquitetura do I-IP. Figura 7.14 mostra o diagrama de bloco detalhado da metodologia de teste funcional em paralelo.

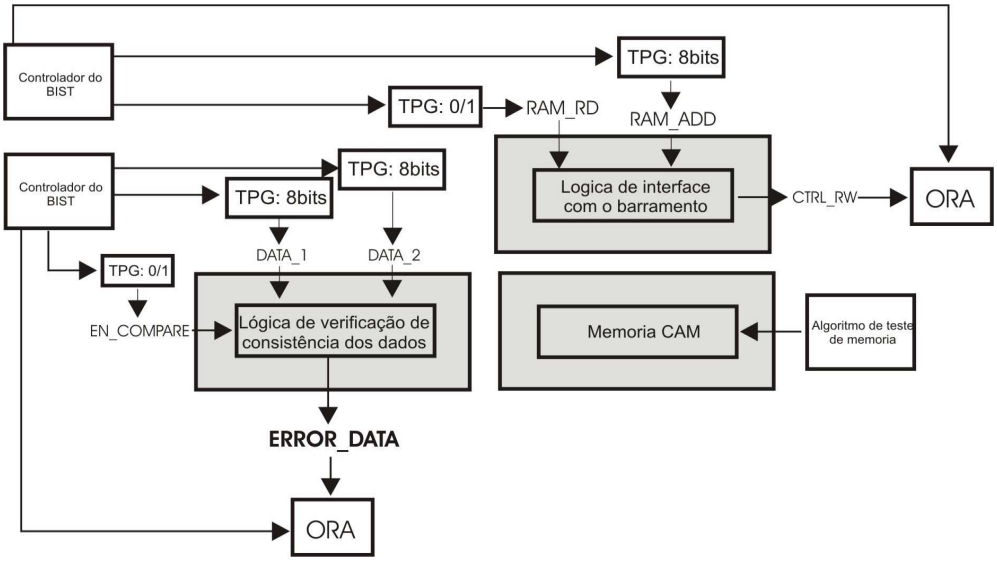


Figura 7.14 Diagrama de bloco de um BIST off-line paralelo.

Especificação de um teste *on-line*

Outra solução para verificar se o I-IP está funcionando corretamente, em tempo de execução, é agregar algumas redundâncias, ou seja, agregar alguma técnica de redundância nos módulos que apresentem funções críticas, por exemplo, o módulo que verifica a consistência dos dados. A figura 7.15 ilustra claramente a metodologia de teste *on-line* sugerida para proteger o I-IP desenvolvido neste trabalho de dissertação.

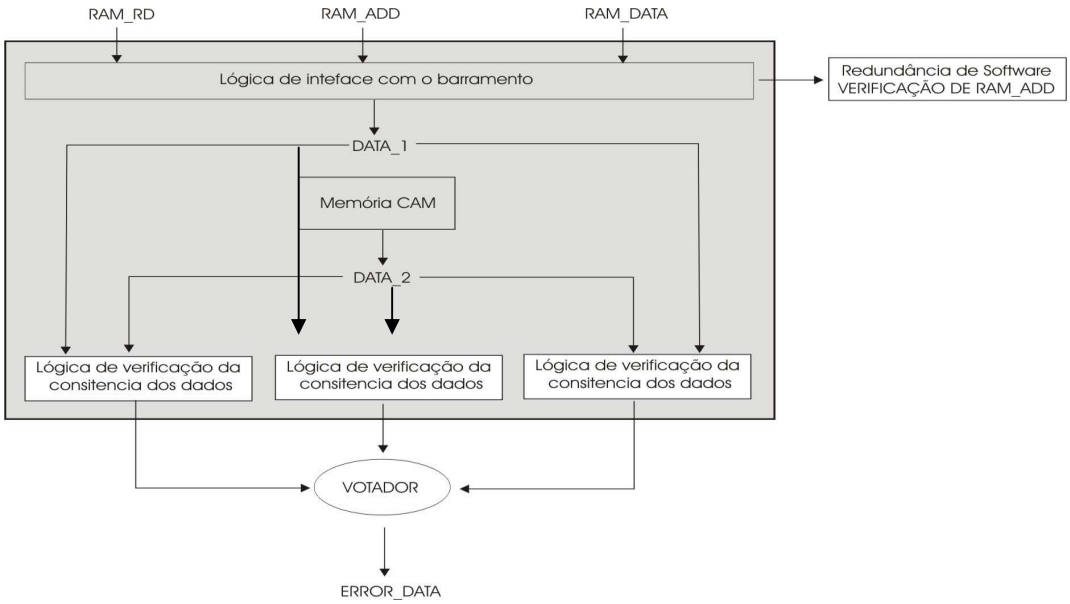


Figura 7.15 Diagrama de bloco de um esquema de BIST *on-line*.

Parte III – RESULTADOS E CONCLUSÕES

8. RESULTADOS

Introdução

A avaliação da capacidade do método proposto nesta dissertação foi realizada a partir da implementação de um I-IP que monitora o barramento do microcontrolador 8051 da Intel durante a execução de uma determinada aplicação. As aplicações executadas pelo microcontrolador 8051 da Intel consistem em alguns programas *benchmarks* modificados a partir das regras de transformação de código definidas no capítulo anterior na seção 7.

Os experimentos de injeção de falhas foram executados a fim de verificar a capacidade do método proposto de detectar falhas transientes que afetam a informação armazenada. Salienta-se que somente na terceira versão do I-IP foi agregada a capacidade de detectar além de erros em dados, também erros de fluxo de controle.

Finalmente, quanto ao tipo de falha, o SoC deve ser tolerante a falhas do tipo *Single Event Upset* (SEU). Este tipo de falha é resultado da incidência de partículas de alta energia no substrato, ionizando-o e gerando uma perda de elétrons e lacunas no substrato, em áreas próximas as células que armazenam informação útil. O resultado dista perturbação é a inversão de um ou mais bits que representam esta informação, corrompendo-a. Este tipo de falha caracteriza-se por ser um evento aleatório e por isto podem ocorrer em tempos imprevisíveis. Neste trabalho, para modelar o efeito dos SEUs explorou-se o modelo de falha *transient bit-flip*, que consiste na modificação do conteúdo armazenado em uma determinada célula de memória durante a execução de um programa.

Assim, a partir da próxima seção serão apresentados e discutidos os resultados obtidos a partir dos experimentos de injeção de falhas.

Análise da primeira versão do I-IP

Introdução

A primeira versão do I-IP foi avaliada a partir da análise de vários experimentos de injeção de falhas *transient bit-flip* na memória de dados (RAM) do SoC. Quanto à aplicação executada

pelo microcontrolador 8051 a Intel, foram levados em consideração os dois *benchmarks* abaixo descritos:

- *Matrix*: algoritmo de multiplicação de duas matrizes 3x3.
- *Bubble Sort*: algoritmo de ordenamento de um *array* de 10 bytes.

Durante os experimentos de injeção de falhas, foram considerados três implementações diferentes dos algoritmos acima mencionados.

- Original: representa os *benchmarks* originais, ou seja, em V1 nenhuma técnica de tolerância a falhas foi agregada aos algoritmos.
- Software: representa uma implementação tolerante a falhas de acordo com método proposto em (REBAUDENGO, 2004).
- Híbrida: representa a solução híbrida proposta nesta dissertação, ou seja, uma implementação parte em software e parte em hardware. A parte em software representa o algoritmo modificado a partir das regras de transformação de código anteriormente descritas e o I-IP a parte de hardware.

Capacidade de detecção de falhas da primeira versão do I-IP

Após a implementação dos algoritmos acima descritos, foram realizados vários ciclos de injeção de falhas e cerca de 10.000 falhas foram injetadas nos segmentos de dados e nos registradores do microcontrolador 8051 da Intel acessíveis aos usuários durante a execução da implementação original.

Salienta-se que o número de falhas injetadas é proporcional ao *overhead* de área e de desempenho introduzido pelas técnicas de tolerância à falhas. Este critério foi adotado, pois os programas tolerantes a falhas são maiores que os originais, exigem mais tempo para serem executados e conseqüentemente são mais sensíveis aos SEUs.

A localização e o tempo de injeção de falhas foram selecionados randomicamente. Os efeitos das falhas foram classificados em:

- Silêncio: a falha não modifica a execução do programa, ou seja, é como se o programa estivesse sendo executado sem a injeção de falhas.

- *Time-out*: a falha modifica a execução do programa de tal modo que o processador entra em um laço infinito e conseqüentemente nenhum resultado pode ser gerado.
- Detectado: a falha modifica a execução do programa, mas a técnica de tolerância a falhas implementada consegue detectá-la.
- Resposta Errada: a falha modifica a execução do programa e gera saídas diferentes das esperadas. Essas falhas são extremamente críticas e devem ser minimizadas pela técnica de tolerância a falhas agregada.

A partir da análise da tabela 8.1, pode-se concluir que o método híbrido proposto nesta dissertação é bastante eficiente na detecção de falhas.

Programa	Implementação	S [#]	T [#]	D [#]	WA [#]
Bubble Sort	V1	9,261	0	0	739
	V2	30,585	0	10,060	0
	V3	61,122	0	13,746	0
Matrix	V1	9,486	0	0	514
	V2	30,599	0	4,933	11
	V3	46,106	0	9,206	44

Tabela 8.1. Resultados dos experimentos de injeção de falhas.

Overhead da primeira versão do I-IP

Basicamente, o *overhead* de área foi estimado a partir da síntese do codigo fonte do I-IP, escrito em VHDL. Assim, a primeira versão do I-IP, que possui cerca de 655 linhas de código, apresenta um overhead de área de aproximadamente 5,1% em relação ao microcontrolador 8051.

E por fim, no que diz respeito aos *overheads* de memória e desempenho, eles foram estimados a partir da análise da quantidade de memória ocupada pelos programas tolerantes a falhas, obtidos a partir da aplicação das regras de transformação de código propostas neste trabalho, em relação aos programas gerados a partir da técnica proposta em (REBAUDENGO, 2004). Salienta-se que, o programa original foi utilizado como padrão de referência para realizar estas estimativas.

A tabela 8.2 ilustra o *overhead* de área e de desempenho. O *overhead* de área é medido a partir da análise do número de bytes nos segmentos de código e dados do programa e o de desempenho a partir do número de ciclos de *clock* por execução completa do programa.

Programa	Implementação	Ocupação		Duração	
		[# bytes]	[%]	[# clock]	[%]
Bubble Sort	Original	9261		75	
	Software	61122	659.99	267	356.00
	Hibrida	30585	330.26	145	193.33
Matrix	Original	9486		12	
	Software	46106	486.04	32	266.67
	Hibrida	30599	322.57	20	166.67

Tabela 8.2. Ocupação da memória e tempo de execução do programa.

Análise da segunda versão do I-IP

Introdução

A segunda versão do I-IP, denominado Cerberus, foi avaliada a partir da análise de vários experimentos de injeção de falhas *transient bit-flip* na memória de dados (RAM) do SoC. Quanto à aplicação executada pelo microcontrolador 8051 a Intel, foram levados em consideração os três *benchmarks* abaixo descritos:

- *Matrix*: algoritmo de multiplicação de duas matrizes 3x3.
- *Ellipf*: algoritmo que aplica um filtro *Elliptic* sobre um conjunto de seis amostras.
- FIR: algoritmo que aplica um filtro *finite impulse response* de 8-*tap* sobre um conjunto de 16 amostras.

Durante os experimentos de injeção de falhas, foram considerados três implementações diferentes dos algoritmos: original, software e híbrida, ja definidas na seção 8.2.1.

Capacidade de detecção de falhas da segunda versão do I-IP

Após a implementação dos algoritmos acima descritos, foram realizados vários ciclos de injeção de falhas e cerca de 10.000 falhas foram injetadas nos segmentos de dados e nos registradores do microcontrolador acessíveis aos usuários na implementação plana.

Salienta-se que o número de falhas injetadas é proporcional ao overhead de área e de desempenho introduzido nas implementações software e híbrida. Este critério foi adotado, pois os programas tolerantes a falhas são maiores que os originais, exigem mais tempo para serem executados e conseqüentemente são mais sensíveis aos SEUs. A localização e o tempo de injeção de falhas foram selecionados randomicamente. Os efeitos das falhas também foram classificados como silêncio, *time-out*, detectado e resposta errada.

A tabela 8.3 resume os resultados obtidos a partir dos experimentos de injeção de falhls.

Programa	**Implementação**	**Silêncio**		**Time Out**		**Detectado**		**Resposta Errada**	
		[#]	**[%]**	**[#]**	**[%]**	**[#]**	**[%]**	**[#]**	**[%]**
Matrix	Original	8702	87.02	21	0.21	0	0.00	1277	12.77
	Software	49518	72.07	144	0.21	18620	27.10	426	0.62
	Híbrida	23464	69.80	131	0.39	10007	29.77	13	0.04
Ellipf	Original	9536	95.36	7	0.07	0	0.00	457	4.57
	Software	67567	75.36	81	0.09	22011	24.55	0	0.00
	Híbrida	26885	80.05	0	0.00	6697	19.94	3	0.01
FIR	Original	8568	85.68	6	0.06	0	0.00	1426	14.26
	Software	51393	66.68	39	0.05	25635	33.26	8	0.01
	Híbrida	23449	68.02	45	0.13	10973	31.83	7	0.02

Tabela 8.3. Resultados dos experimentos de injeção de falhas.

Overhead da segunda versão do I-IP

A implementação desta versão do I-IP também agrega os mesmos tipos de overhead da versão anterior. O *overhead* de área está relacionado a implementação do *Cerberus*, ou seja, hardware agregado. O overhead de memória ocorre devido a duplicação dos dados e das operações presentes no código do programa. E, por fim o overhead de desempenho está relacionada à duplicação das operações.

A tabela 8.4 mostra o *overhead* de área e de desempenho. O *overhead* de área é medido a partir da análise do número de *bytes* nos segmentos de código e dados do programa e o de desempenho a partir do número de ciclos de *clock* por execução completa do programa.

Programa	Versão	Ocupação		Duração	
		[# bytes]	[%]	[# clock]	[%]
Matrix	Original	246		29715	
	Software	720	292.68	102084	343.54
	Híbrida	438	178.05	49945	168.08
Ellipf	Original	359		16268	
	Software	1755	488.86	72929	448.30
	Híbrida	667	185.79	27318	167.92
FIR	Original	228		43434	
	Software	793	347.81	167381	385.37
	Híbrida	544	238.60	74867	172.37

Tabela 8.4. Ocupação da memória e tempo de execução do programa.

Análise da terceira versão do I-IP

Introdução

As duas primeiras versões do I-IP implementam uma solução híbrida capaz de detectar erros em dados. Já a terceira versão do I-IP agrega ao *Cerberus* a capacidade de prover a detecção de erros de fluxo de controle através de uma solução híbrida implementada parte em hardware e parte em software. Assim como nas versões anteriormente desenvolvidas, a eficiência desta versão do I-IP foi avaliada a partir da análise de um protótipo que trabalha em conjunto com o microcontrolador 8051 da Intel.

A capacidade de detecção de falha do método proposto foi investigada considerando os seguintes tipos de falhas: A) SEUs que afetam o conteúdo do segmento de código; B) SEUs que afetam a porção de memória de dados; C) SEUs que afetam os registradores do microcontrolador.

Quando foi considerado o primeiro tipo de falha, ou seja, as falhas que afetam o conteúdo do segmento de código, o conjunto de instruções do microcontrolador foi dividido em duas categorias: (1) instruções funcionais e (2) instruções de desvio. Conseqüentemente, como o bit

modificado na área de código pertence a um *opcode*, as seguintes categorias podem ser introduzidas:

- functional_to_branch: o bit altera o *opcode* transforma a instrução funcional em uma instrução de desvio;
- branch_to_functional – o bit altera o *opcode* e transforma uma instrução de desvio em uma instrução funcional;
- functional_to_functional – o *opcode* de uma instrução funcional é transformado em outra instrução funcional com o mesmo número de operandos ou com um número de operandos diferentes.

Caso o bit altere o operando de uma determinada instrução, os seguintes comportamentos foram investigados:

- wrong_memory_access: o operando modificado é o endereço de uma variável;
- wrong_immediate_value: o operando modificado é um valor imediato;
- wrong_branch_offset: o operando modificado é o rótulo de uma instrução de desvio.

Considerando falhas que afetam a área de dados, os efeitos das falhas foram classificados em:

- wrong_elaboration: o valor lido da memória está errado;
- wrong_branch_condition: uma condição de desvio é executada para um valor incorreto.

Finalmente, considerando as falhas únicas que afetam o conteúdo dos registradores do processador, os efeitos das falhas foram classificados em:

- wrong_general_purpose_value: um registrador de uso geral armazena um valor incorreto;
- wrong_configuration_value: o processador é configurado incorretamente.

Neste contexto, os experimentos de injeção de falhas foram executados levando-se em consideração quatro *benchmarks* inspirados no suíte EEMBC Automotive/Industrial, descritos a seguir:

- *Matrix Multiplication* (MTX): este algoritmo calcula o produto de duas matrizes 5x5;

- *Fifth Order Elliptical Wave Filter* (ELPF): este algoritmo aplica um filtro *Elliptic* sobre um conjunto de 6 amostras;
- *Lempel-Ziv-Welch Data Compression Algorithm* (LZM): este algoritmo comprime dados.
- *Viterbi Algorithm* (V): este algoritmo implementa o algoritmo de Viterbi.

Assim como nas implementações anteriores do I-IP, cada um dos *benchmarks* foi codificado nas versões original, *software* e híbrida e o impacto das falhas de fluxo de controle foram classificadas em silêncio, *time-out*, detectado e resposta errada.

Cobertura de falhas da terceira versão do I-IP

Assim como nas versões anteriores, a capacidade de detecção de falha desta versão do I-IP, foi avaliada a partir da execução de vários experimentos de injeção de falhas. Assim, foram injetados 30,000 *bit flips* selecionados aleatoriamente na área da memória que armazena o código de cada programa *benchmark*. As tabelas 8.5, 8.6 e 8.7 resumem os resultados obtidos a partir dos experimentos de injeção de falhas no segmento de código, no segmento de dados e por fim nos registradores do processador respectivamente.

Programa	**Versão**	**Silêncio**		**Time Out**		**Detectado**		**Resposta Errada**	
		[#]	[%]	[#]	[%]	[#]	[%]	[#]	[%]
MTX	Original	20607	68,69	3864	12,88	0	0	5529	18,43
	Software	18798	62,66	2121	7,07	8208	27,36	873	2,91
	Híbrida	15567	51,89	3225	10,75	11202	3,7,34	3	0,01
ELPF	Original	16071	53,57	3339	11,13	0	0	10590	35,30
	Software	18015	60,05	5583	18,61	5115	17,05	1287	4,29
	Híbrida	14448	48,16	2751	9,17	12750	42,50	51	0,17
LZW	Original	6852	22,84	5469	18,23	0	0	17679	58,93
	Software	10260	34,20	9405	31,35	9420	31,40	915	3,05
	Híbrida	8703	29,01	7836	26,12	13377	44,59	84	0,28
V	Original	8178	27,26	6093	20,31	0	0	15729	52,43
	Software	10884	36,28	10302	34,34	7518	25,06	1296	4,32
	Híbrida	10743	35,81	5136	17,12	13734	45,78	387	1,29

Tabela 8.5 Resultados obtidos a partir de experimentos de injeção de falhas afetando a memória que armazena o código.

Programa	Versão	Silêncio		Time Out		Detectado		Resposta Errada	
		[#]	[%]	[#]	[%]	[#]	[%]	[#]	[%]
MTX	Original	26808	89,36	63	0,21	0	0	3129	10,43
	Software	24930	83,10	57	0,19	5013	16,71	0	0
	Híbrida	25053	83,51	0	0	4947	16,49	0	0
ELPF	Original	28623	95,41	33	0,11	0	0	1344	4,48
	Software	22764	75,88	30	0,10	7206	24,02	0	0
	Híbrida	24339	81,13	0	0	5661	18,87	0	0
LZW	Original	23889	79,63	450	1,50	0	0	5661	18,87
	Software	18642	62,14	273	0,91	1185	36,95	0	0
	Híbrida	17958	59,86	0	0	12042	40,14	0	0
V	Original	19137	63,79	810	2,70	0	0	10053	33,51
	Software	17067	56,89	450	1,50	12483	41,97	0	0
	Híbrida	17433	58,11	0	0	12567	41,90	0	0

Tabela 8.6 Resultados obtidos a partir de experimentos de injeção de falhas afetando a memória que armazena os dados.

Programa	Versão	Silêncio		Time Out		Detectado		Resposta Errada	
		[#]	[%]	[#]	[%]	[#]	[%]	[#]	[%]
MTX	Original	27777	92,59	927	3,09	0	0	1296	4,32
	Software	27987	93,29	189	0,63	1734	5,78	90	0,30
	Híbrida	27039	90,13	651	2,17	2265	7,55	45	0,15
ELPF	Original	27789	92,63	948	3,16	0	0	1263	4,21
	Software	28035	93,45	93	0,31	1641	5,47	231	0,77
	Híbrida	26817	89,39	807	2,69	2307	7,69	69	0,23
LZW	Original	26763	89,21	705	2,35	0	0	2532	8,44
	Software	26871	89,57	303	1,01	2565	8,55	261	0,87
	Híbrida	27390	91,30	384	1,28	1836	6,12	390	1,30
V	Original	27396	91,32	1437	4,79	0	0	1167	3,89
	Software	27618	92,06	933	3,11	1236	4,12	213	0,71
	Híbrida	26907	89,69	939	3,13	2067	6,89	87	0,29

Tabela 8.7 Resultados dos experimentos de injeção de falhas afetando os elementos de memória dentro do processador.

Overhead da terceira versão do I-IP

A solução híbrida proposta para detectar falhas em dados e falhas de fluxo de controle agrega três tipos de *overheads*:

- Área – *overhead* relacionado a inserção do I-IP no SoC;

- Memória – *overhead* relacionado a duplicação das variáveis e operações e a inserção das instruções *IIPtest()* e *IIPset()*.
- Desempenho – *overhead* relacionado as instruções extras que devem ser executadas.

A quantificação da área agregada pelo I-IP desenvolvido foi realizada a partir da síntese de seu código escrito em VHDL através da ferramenta comercial *Synopsys Design Analyzer* e uma biblioteca desenvolvida pelas nossas instruções. O I-IP foi configurado para interagir com o barramento de memória interna do núcleo do microcontrolador 8051 e possui uma memória CAM capaz de armazenar 16 entradas. A tabela 8.8 abaixo detalha a ocupação em nível de portas lógicas do I-IP.

Componente lógico	**[#] de portas equivalentes**
Interface com o barramento	251
Verificação da consistência do fluxo de controle	741
Verificação da consistência dos dados	1,348
Memória CAM	1,736
TOTAL	4,076

Tabela 8.8 Número de portas lógicas dos módulos do I-IP

Os *overheads* de memória e desempenho foram mensurados a partir da comparação entre a ocupação de memória dos programas tolerante a falhas de acordo com a solução híbrida proposta neste trabalho e a ocupação de memória dos programas tolerante a falhas de acordo com (REBAUDENGO, 2004) e (GOLOUBEVA, 2003). Além disso, a área ocupada e a duração dos programa originais foram considerados. A tabela 8.9 abaixo resume o *overhead* de memória e de desempenho. A ocupação da memória foi representada pelo número de bytes nos segmentos de dados e de código enquanto a duração foi representada pelo número de ciclos de *clock* para executar o programa.

Programa	Versão	Tempo de Execução (CC)		Tamanho do Código (B)		Tamanho dos Dados (D)	
		[#]	[%]	[#]	[%]	[#]	[%]
MTX	Plana	13055	-	329	-	16	-
	Software	42584	226,19	1315	299,7	34	112,50
	Híbrida	27930	113,94	683	107,6	34	112,50
ELPF	Plana	12349	-	384	-	48	-
	Software	46545	276,91	1527	297,66	100	108,33
	Híbrida	21946	77,71	645	67,97	100	108,33
LZW	Plana	19209	-	232	-	35	-
	Software	92003	378,96	1898	718,10	72	105,71
	Híbrida	38878	102,39	859	270,26	72	105,71
V	Plana	286364	-	436	-	85	-
	Software	1.398423	388,34	2032	366,06	172	102,35
	Híbrida	598410	108,97	1323	203,44	172	102,35

Tabela 8.9 Resumo do *overhead* de memória e desempenho.

9. CONCLUSÕES E TRABALHOS FUTUROS

Conclusões específicas para cada uma das versões do I-IP

As conclusões abaixo mencionadas foram formalizadas a partir da análise dos resultados obtidos nos experimentos de injeção de falhas apresentados no capítulo anterior.

Conclusões relativas à primeira versão do I-IP

A partir da análise da tabela 8.1 é possível concluir que a grande maioria das falhas injetadas nos dados, cerca de 99,02%, foi detectada. Salienta-se que provavelmente as falhas não detectadas (0,08%) representam modificações no fluxo de execução do algoritmo, ou seja, erros de fluxo de controle e, portanto, a detecção das mesmas exige que seja agregado ao I-IP técnicas de controle de fluxo de algoritmo.

Conforme anteriormente mencionado, a primeira versão do I-IP possui cerca de 655 linhas de código e agrega basicamente três tipos de *overhead*: de área, de memória e de desempenho. Esta primeira versão do I-IP agrega aproximadamente um overhead de área de 5,1% em relação à área de ocupação do núcleo do microcontrolador 8051 da Intel. No que diz respeito ao overhead de memória e de desempenho, podemos concluir a partir da tabela 8.2 que a solução proposta neste trabalho reduz cerca de 3 vezes o overhead de memória e cerca de 4 vezes o overhead de desempenho em relação à técnica implementada puramente em software.

Em vista do exposto, é possível concluir que a técnica híbrida para detecção de erros em dados, proposta nesta dissertação, representa uma solução viável e eficiente. A partir dos experimentos de injeção de falhas é possível evidenciar e constatar que este método é capaz de prover eficientemente a detecção de erros em dados, que o *overhead* de área agregado é bastante pequeno e que o desempenho sofre uma queda pouco significativa.

Conclusões relativas à segunda versão do I-IP

A partir da análise da tabela 8.3 é possível observar uma redução bastante significativamente do número de respostas erradas (WA*)* em relação ao algoritmo original. A capacidade de detecção de falha do *Cerberus* é comparável a capacidade de detecção do método implementado puramente em software, e algumas vezes é ligeiramente melhor. Isto ocorre, porque o método híbrido é capaz de detectar algumas falhas que modificam o fluxo de execução do código.

Quanto ao *overhead* de área, o *Cerberus* foi sintetizado com a ferramenta *Synopsys Design Analyzer* e uma biblioteca de tecnologia desenvolvida no Politécnico de Torino. Conforme anteriormente mencionado, o *Cerberus* foi configurado para monitorar o barramento da memória interna do microcontrolador 8051 da Intel, sendo que sua memória CAM foi configurada para armazenar 16 entradas e possui aproximadamente 1.636 portas lógicas e 288 *flip-flops* correspondendo a 700 linhas de código. Por outro lado, o núcleo do 8051 utilizado possui cerca de 10.723 portas lógicas e 19.757 *flip-flops*.

No que diz respeito aos *overheads* de memória e de desempenho, podemos concluir a partir da tabela 8.4 que a solução proposta neste trabalho reduz, em média, cerca de 2 vezes os *overheads* de memória e de desempenho em relação a técnica implementada puramente em *software*.

Em vista do exposto, é possível concluir que o *Cerberus* representa uma solução viável e eficiente. A partir dos experimentos de injeção de falhas é possível evidenciar e constatar que este método é capaz de prover eficientemente a detecção de erros em dados, que o *overhead* de área agregado é bastante pequeno e que o desempenho sofre uma queda pouco significativa.

Conclusões relativas à terceira versão do I-IP

A partir da análise da tabela 8.5 é possível concluir que o método híbrido, em relação ao método implementado via software, diminui significativamente o número de falhas que conduzem a repostas erradas (*wrong answer*). Diante de *Bit flips* que afetam o *opcode* das instruções e eventualmente geram uma resposta errada devido principalmente a falhas do tipo functional_to_functional ou modificações branch_to_functional a solução híbrida é mais eficiente que a solução implementada via software. Além disso, é possível observar a partir desta tabela que *bit flips* que afetam os operandos raramente produzem respostas erradas e são eficientemente

detectados por ambas as técnicas. Adicionalmente, o método implementado via software produz, a partir das instruções de teste, desvios adicionais no grafo de fluxo do programa para verificar continuamente o valor do *flag* de erro. Além disso, o compilador C pode transformar algumas instruções implementadas para a verificação da consistência dos dados e do fluxo de controle em novas instruções em nível de *assembly* contendo desvios adicionais. Neste caso, os novos desvios não estarão protegidos com as funções IIPtest() e IIPset() e conseqüentemente o método implementado via software pode não detectará estas falhas. Convencionalmente, quando se utiliza a solução híbrida, nenhuma instrução de desvio é introduzida no código da aplicação e isto diminui o número de situações com respostas erradas (*wrong answer*) e time out.

A partir da análise da tabela 8.6 é possível observar que assim como na versão implementada puramente em software, o método híbrido também não produz nenhuma situação de resposta errada. Entretanto, quando os efeitos *time out* são consideradas, a solução híbrida é mais eficiente devido fundamentalmente ao diferente comportamento relacionado as falhas da categoria wrong_branch_condition.

Finalmente a partir da tabela 8.7 é possível observar que quando consideramos as respostas erradas, o comportamento dos *benchmarks* modificados a partir da solução baseada em software é extremamente diferente do comportamento dos *benchmarks* modificados a partir da solução híbrida.

Uma análise mais detalhada dos resultados permite verificar que especificamente no algoritmo LZW a solução híbrida é menos eficiente que a solução implementada puramente em software. Isto ocorre, pois o programa utiliza algumas variáveis armazenadas nos registradores e conseqüentemente as falhas que eventualmente as afetem não são detectadas pela solução híbrida.

A partir da síntese do I-IP com a ferramenta *Synopsys Design Analyser* é possível concluir que o I-IP agrega ao SoC um *overhead* de área de cerca de 5%. Salienta-se que este valor foi calculado levando-se em consideração a área do microcontrolador 8051 e suas memórias.

No que diz respeito ao *overhead* de desempenho, podemos concluir a partir da tabela 8.4 que a solução proposta neste trabalho reduz pela metade este *overhead* em relação a solução implementada via software. Finalmente no que diz respeito ao overhead de memória, é possível observar que o tamanho da memória de dados exigida por ambas as soluções é similar. Entretanto, a memória exigida para o armazenamento do código da solução híbrida é a metade da memória exigida pela solução implementada puramente em software.

Em vista do exposto, a partir da análise dos *overheads* e da capacidade de detecção de falha, é possível concluir que o método híbrido aumenta significativamente a confiabilidade de SoCs, ao custo de um pequeno *overhead* de área e memória e de uma degradação de desempenho pouco significativa.

Conclusões gerais

Este trabalho teve como objetivo principal especificar, implementar e avaliar uma solução híbrida baseada nas técnicas propostas em (REBAUDENGO, 2004) e (GOLOUBEVA, 2003), capaz de detectar falhas em dados e falhas de fluxo de controle em aplicações críticas durante seu funcionamento.

Para cumprir este objetivo, as técnicas propostas em (REBAUDENGO, 2004) e (GOLOUBEVA, 2003) foram implementadas parte em software e parte em hardware. Este particionamento híbrido levou em consideração aspectos críticos, tais como degradação do desempenho, *overhead* de área e de memória e a capacidade de detecção de falhas.

As modificações realizadas no código fonte da aplicação agregam redundância de software através da duplicação das variáveis e operações, e da inserção de instruções necessárias para o monitoramento do fluxo de execução algoritmo. O I-IP desenvolvido para implementar a parte em hardware evoluiu através de três implementações diferentes. A primeira versão do I-IP provê apenas a detecção de falhas em dados e, como todo protótipo, este também apresenta algumas restrições e particularidades que serão detalhadamente descritas. A segunda versão também é capaz de detectar apenas falhas em dados, entretanto supera as restrições da versão anterior através de um remodelamento da arquitetura básica do I-IP e da inserção de um módulo de memória. Por fim, a terceira versão do I-IP, agrega a capacidade de detectar falhas de fluxo de controle à segunda versão.

A capacidade de detecção de falhas foi avaliada a partir de vários experimentos de injeção de falhas, levando em consideração falhas reais que afetam os elementos de memória. Neste caso, o modelo de falhas assumido foi o *Single Event Upset* (SEU) para falhas transientes.

Assim, as principais vantagens do método híbrido proposto neste trabalho podem ser salientadas nos seguintes pontos:

- 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 I-IP no barramento do processador.
- O I-IP é completamente independente do código executado pelo processador, quando mudanças são introduzidas no código, o I-IP não precisa ser alterado.
- O método detecta eficientemente um grande número de falhas transientes, que podem ser localizadas em algum registrador do processador ou na área de memória que armazena o código ou os dados; salienta-se o fato que esta solução pode detectar simultaneamente falhas afetando os dados e o código.
- O custo por área do I-IP é relativamente reduzido.
- Os *overheads* de memória e de desempenho são significativamente menores que os adicionados pelas soluções implementadas puramente em software.

Trabalhos futuros

- Implementar em VHDL a quarta versão do I-IP, tolerante a falhas, especificada neste trabalho;
- Avaliar a eficiência e o *overhead* do I-IP tolerante a falhas a partir de procedimentos de injeção de falhas;
- Colocar o I-IP em um FPGA e testá-lo em um sistema real, ou seja, com o próprio microcontrolador.

REFERÊNCIAS BIBLIOGRÁFICAS

(CHEYNET, 2000) P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, M. Violante. "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors", IEEE Transaction on Nuclear Science, Vol. 47, No. 6, December 2000, pp. 2231-2236

(MCCLUSKEY, 2002) N. Oh, P. Shirvani, Edward .J. McCluskey. "Control-Flow Checking by Software Signatures", IEEE Transactions on Reliability, Vol. 51, No. 2, March 2002, pp. 111-122

(CIVERA, 2002) P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante. "An FPGA-based approach for speeding-up Fault Injection campaigns on safety-critical circuits", Journal of Electronic Testing: Theory and Applications, Vol. 18, No. 3, June 2002, pp. 261-271

(GOLOUBEVA, 2003) O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante. "Soft-error Detection Using Control Flow Assertions", IEEE Int.l Symp. on Defect and Fault Tolerance in VLSI Systems, 2003, pp. 581-588

(REBAUDENGO, 1999) M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante. "Soft-error Detection through Software Fault-Tolerance Techniques", IEEE Int.l Symp. on Defect and Fault Tolerance in VLSI Systems, 1999.

(REBAUDENGO, 2004) M. Rebaudengo, M. Sonza Reorda, M. Violante. "A New Approach to Software-Implemented Fault Tolerance", Journal of Electronic Testing: Theory and Applications, 2004.

(ALKHALIFA, 1997) Z. Alkhalifa, V.S.S. Nair, "Design of a Portable Control-Flow Checking Technique", IEEE High-Assurance Systems Engineering Workshop, 1997.

(KANAWATI, 1996) G. A. Kanawati, V.S.S. Nair, N. Krishanmurthy, J. A. Agraham. “Evaluation of Integrated System-Level Checks for On-Line Error Detection”, IEEE Computer Performance and Dependability Symposium, 1996, pp. 292-301.

(ALKHALIFA, 1999) Z Alkhalifa, VSS Nair, N Krishnamurthy, JA Abraham. “Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection”, IEEE Trans. On Parallel and Distributed Systems, 1999.

(NAHMUSK, 2002) Nahmsuk Oh, Subhasish Mitra and Edward J. McCluskey. “ED4I: Error Detection by Diverse Data and Duplicated Instructions”, IEEE Trans. on Computers, Vol 51, nº 2, Feb. 2002, pp. 180-199.

(MICHEL, 1991) T. Michel, R. Leveugle, G. Saucier. “A New Approach to Control Flow Checking Without Program Modification”, IEEE Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium.

(YAU, 1978) S. S. Yau, F. C. Chen, D. H. Yau. “An Approach to Real-Time Control Flow Checking”, Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International.

(AGRAWAL, 1993) V. D. Agrawal, C. R. Kime, K. K. Saluja. “A Tutorial on Built-in Self-Test”, IEEE Design & Test of Computers, março 1993. Pp. 73-82.

(CORTÊS, 1991) L. M. Cortês, J. F. Mendonça. “Introdução ao Teste de Circuitos Digitais”, livro texto da V Escola Brasileiro-Argentina de Informática. Nova Friburgo, RJ, fevereiro de 1991.

(PRADHAN, 1996) D. K. Pradhan. “Fault-Tolerant Computer System Design”, Prentice-Hall, 1996.

(WILKEN, 1990) K. Wilken, J. P. Shen. “Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1990.

(STROUD, 2002) E. Charles Stroud. “A designer's guide to built-in self-test”. Boston : Kluwer Academic, 2002.

(BARDELL, 1987) Paul H. Bardell. “Built in test for VLSI: pseudorandom techniques”. New York. 1987.

(CORTNER, 1987) J. Max Cortner,.“Digital test engineering”. New York, NY. 1987.

(RAJSKI, 1998) Janusz Rajski. “Arithmetic built-in self-test for embedded systems”. Upper Saddle River, NJ: Prentice Hall, 1998.

(REIS, 2000) R. A. L. Reis. “Concepção de Circuitos Integrados”, Editora Sabra Luzzato, 2000.

(SCHILDT, 1997) H. Schildt. “C: completo e total”. São Paulo : Makron Books, 1997.

APÊNDICES

A. O MICROCONTROLADOR 8051

Introdução

A família de microcontroladores de 8 bits MCS-51 foi criada pela Intel em 1980 como uma evolução da família MCS-48. Atualmente diversos fabricantes produzem este componente o que o torna um dos microcontroladores mais populares do mundo. O 8051 básico oferece os seguintes recursos:

- CPU de 8 bits otimizada para controle;
- Linhas de I/O bidirecionais e individualmente endereçáveis;
- UART *full duplex*;
- 5 Interrupções (2 externas, 2 dos *timers/counters* e 1 da porta serial);
- 64 KB de Memória de Programa;
- 64 KB de Memória de Dados;
- 111 Instruções;
- PC (*Program Counter*) de 16 bits.

Arquitetura básica

A figura A.1 apresenta o diagrama de bloco da arquitetura do microcontrolador 8051, onde é possível observar a unidade central de processamento (*Central processing unit* – CPU) e seus elementos básicos: RAM, ROM, periféricos e fonte de alimentação.

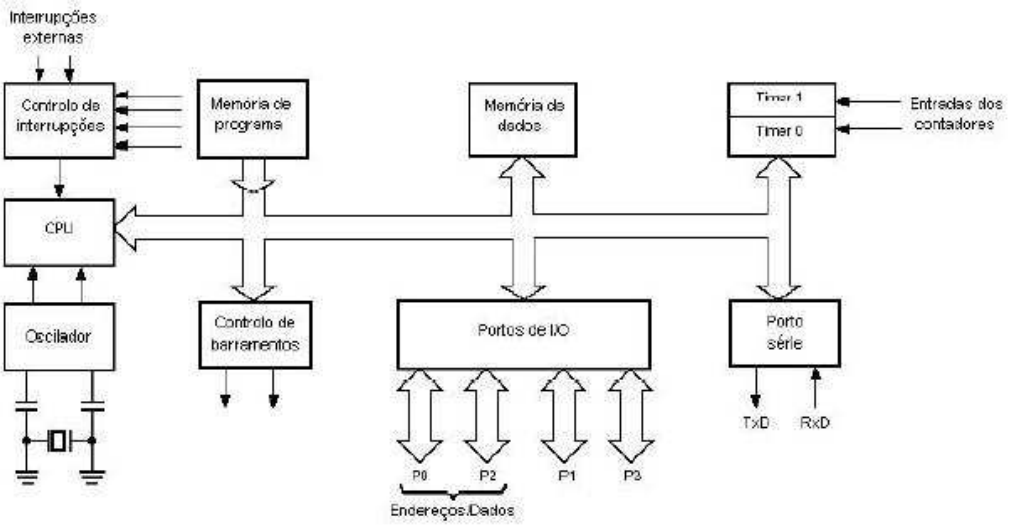


Figura A.1 Diagrama de bloco da arquitetura básica do microcontrolador 8051.

A seguir é feita uma breve descrição dos principais módulos que compõe o microcontrolador 8051.

A) Memória:

A organização da memória do 8051 segue a arquitetura *Harvard*, ou seja, o microcontrolador apresenta uma memória para os dados e outra para o programa. A memória de dados armazena temporariamente informações do uso próprio das instruções, enquanto estas devem ser armazenadas e a memória de programa armazena as instruções que devem ser executadas. A separação lógica destas memórias possibilita que os dados sejam acessados por endereços de 8 bits e conseqüentemente sejam manipulados e armazenados mais rapidamente por uma CPU de 8 bits. Salienta-se que também podem ser gerados endereços de 16 bits através da utilização do registrador DPTR.

Memória de dados:

A memória interna de dados do 8051 armazena exatamente 256 bytes e parte dela pode ser acessada *bit* a *bit*, facilitando sobremaneira a realização de operações *booleanas*. Algumas versões do 8051 possuem até 2kb de memória de programa interna, entretanto este valor não é o padrão. Quanto a utilização de memória externa, o micrcontrolador 8051 é capaz de acessar diretamente até 64kb e, neste caso a unidade central de processamento (*central processing unity* – CPU) gera os sinais de leitura (RD) e escrita (WR).

Memória de Programa:

Atualmente, os microcontroladores 8051 possuem memórias de programas do tipo flash, o que facilita sobremaneira a alteração do seu conteúdo e podem armazenar cerca de 32kb. Além da memória interna de programa, é possível agregar uma memória externa de até 64kb. O acesso a esta memória é feito através de um sinal *Strobe* ativado através do *Program Store Enable Signal* – PSEN.

B) Conjunto de instruções:

As instruções do microcontrolador 8051 podem ser agrupadas funcionalmente nos seguintes conjuntos:

- Instruções aritméticas (+, -, /, *);
- Instruções lógicas (AND, OR, EXOR, operando com byte);
- Instruções de transferência de dados (mover dado – MOV);
- Instruções *booleanas* (AND, OR, EXOR, operando co bit);
- Instruções de desvio/decisão.

Salienta-se que o conjunto de instruções do 8051 é otimizado para aplicações de controle e proporciona vários modos de endereçamento da RAM interna para facilitar operações em pequenas estruturas de dados. Existe um ótimo suporte a variáveis de 1 bit como um tipo separado de dado, permitindo manipulação direta em controle e sistemas lógicos que necessitam de processamento *booleano*.

Bibliografias:

(NICOLOSI, 2000) D. E. C. Nicolosi. “Microcontrolador 8051 Detalhado”, Editora Érica Ltda, 2000.

B. MINI-TUTORIAL DE VHDL

Introdução

Na década de 80, o Departamento de Defesa dos Estados Unidos e o IEEE patrocinaram o desenvolvimento de uma linguagem de descrição de hardware (*Hardware Description Language* – HDL) utilizada para desenvolver sistemas digitais. Esta linguagem tornou-se uma linguagem de padrão industrial para descrição de sistemas digitais. Além do *Very High Speed Integrated Circuit Hardware Description Language* -VHDL, existe o Verilog e a *Advanced Boolean Equation Language* – ABEL. Porém esta última linguagem é utilizada especificamente no projeto de dispositivos de lógica programável (*Programmable Logic Devices* – PLD).

As linguagens de descrição de hardware apresentam diferenças bastante significativas em relação as linguagens convencionais.A HDL é totalmente paralela, pois os comandos, que correspondem a portas lógicas, são executados em paralelos. Basicamente, a HDL emula o comportamento físico de um sistema digital.

Portanto, uma HDL descreve o que um sistema faz e representa o modelo do sistema de hardware que será executado em um software de simulação. Normalmente, o sistema descrito é implementado em um dispositivo programável (*Field Program Gate Array* – FPGA) a fim de possibilitar seu uso em campo.

Este tipo de descrição apresenta uma série de vantagens, tais como:

- Intercâmbio de projetos entre grupos de pesquisa sem a necessidade de alteração;
- Permite ao projetista considerar no seu projeto os *delays* comuns aos circuitos digitais;
- A linguagem é independente da tecnologia atual;
- Os projetos podem ser modificados facilmente;
- O custo de produção é significativamente reduzido em relação a um circuito convencional;
- O tempo de projeto e implementação é consideravelmente reduzido.

Em contrapartida, o VHDL não é capaz de gerar um hardware otimizado.

Níveis de representação e abstração

Um sistema digital pode ser representado através de diferentes níveis de abstração, ou seja, em nível comportamental, estrutural ou físico. A figura 1 mostra os três níveis de abstração de um sistema.

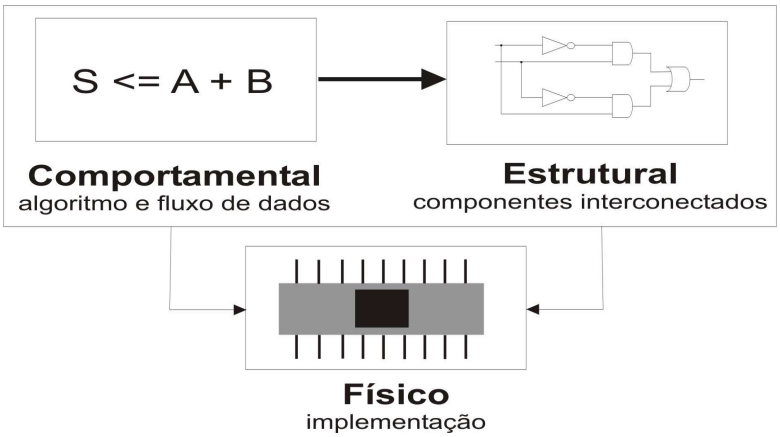


Figura B.1 Níveis de abstração de um sistema.

O nível comportamental descreve um sistema em nível de seus componentes e da interconexão existente entre eles. Basicamente, este tipo de descrição relaciona os sinais de entrada e saída e pode ser implementado através de um fluxo de dados ou através de um algoritmo.

O nível estrutural descreve um sistema através de uma coleção de portas lógicas e componentes interconectados para executar a função desejada. A figura 2 mostra uma representação em nível estrutural de um circuito.

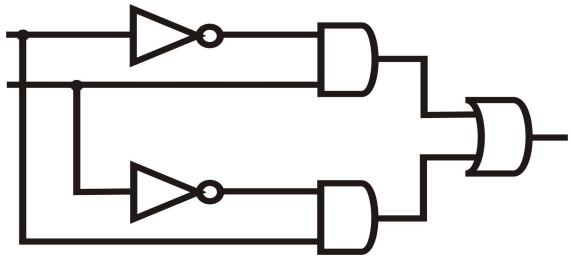


Figura B.2 Representação estrutural de um circuito.

Assim, através da utilização do VHDL é possível descrever um sistema digital em nível comportamental ou estrutural.

Estrutura básica de um programa VHDL

Um sistema escrito em VHDL é formado por uma entidade (*entity*) que pode conter outras entidades que são consideradas componentes da *top-level entity*. Assim, cada entidade é modelada a partir da *entity declaration* e da *architecture body*. A figura 3 mostra o diagrama de bloco de uma entidade.

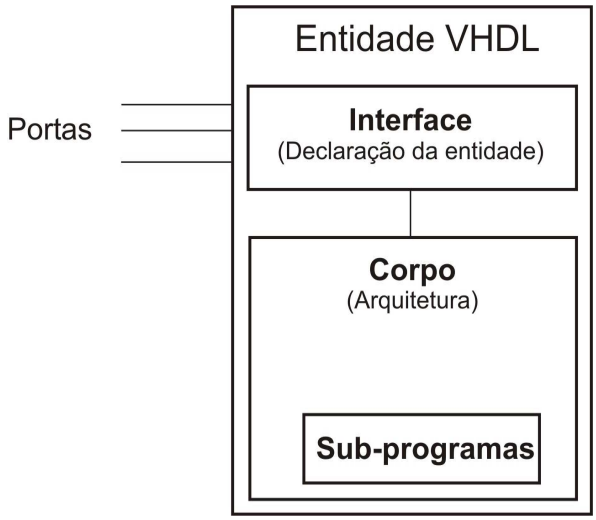


Figura B.3 Entidade VHDL formada de uma interface (*entity declaration*) e de um corpo (*architectural description*).

A estrutura de um programa escrito em VHDL possui quatro blocos básicos, são eles:

- *Package*: bloco onde são declaradas as constantes, os tipos de dados e os sub-programas;
- *Entity:* bloco onde são declarados os pinos de entrada e saída;
- *Architecture*: bloco onde a implementação do projeto é definida;
- *Configuration*: bloco onde as arquiteturas a serem utilizadas são definidas.

A figura B.4 resume a estrutura básica de um programa escrito em VHDL, a figura B.5 apresenta mais detalhadamente a estrutura básica de um programa escrito em VHDL e finalmente a figura B.6 mostra um pequeno código de um programa escrito em VHDL.

PACKAGE
ENTITY
ARCHITECTURE
CONFIGURATION

Figura B.4 Estrutura básica de um programa escrito em VHDL.

LIBRARY IEEE; **USE IEEE.STD_LOGIC_1164.all;** **USE IEEE.STD_LOGIC_UNSIGNED.all;**	**PACKAGE** **(BIBLIOTECAS)**
ENTITY exemplo **IS** **PORT (** < descrição dos pinos de entrada e saída > **);** **END** exemplo;	**ENTITY** **(PINOS DE I/O)**
ARCHITECTURE teste **OF** exemplo **IS** **BEGIN** **PROCESS(** <pinos de entrada e signal > **)** **BEGIN** < descrição do circuito integrado > **END PROCESS**; **END** teste;	**ARCHITECTURE** **(ARQUITETURA)**

Figura B.5 Estrutura detalhada de um programa VHDL.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;

Entity placa is
port  (
        a : in bit_vector( 6 downto 0);
        b : out bit_vector( 6 downto 0)
);
end placa;

architecture TTL of placa is
Signal pino_1 bit;
Signal pino_2 bit;
Signal pino_3 bit;
Signal pino_4 bit;

Begin
        CI_X : process( a )
        Begin
                <descrição do processo>
        end process CI_Y;

        CI_Y : process( a )
        Begin
                <descrição do processo>
        end process CI_Z;
end TTL;
```

Figura B.6 Exemplo de um programa escrito em VHDL.

Bibliografias:

J. V. der Spiegel. “VHDL Tutorial”.
www.seas.upenn.edu/~ee201/vhdl/vhdl_primer.html

A. R. Terroso. “Minicurso 2: Dispositivo Lógicos Programáveis (FPGA) e Linguagem de Descrição de Hardware (VHDL)”.
www.aterroso.com

(CHANG, 1999) C. K. Chang. “Digital systems design with VHDL and synthesis: an integrated approach”. Washington, D.C: IEEE Computer Society, 1999.

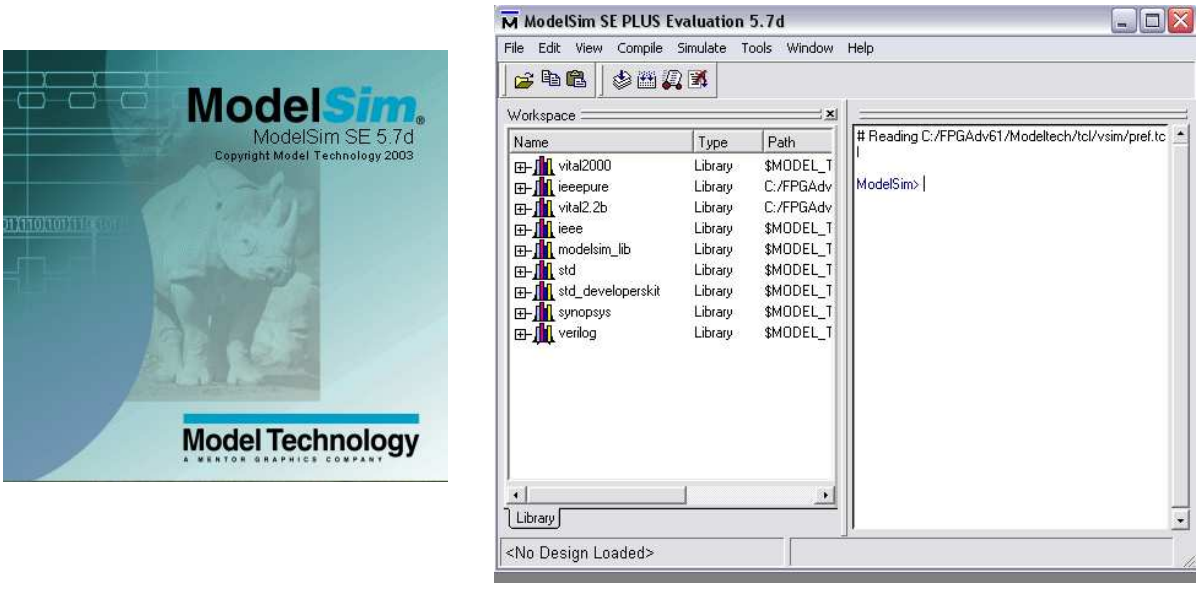
C. INTRODUÇÃO AO MODELSIM

Introdução

Neste trabalho, conforme anteriormente mencionado, o I-IP desenvolvido foi implementado em VHDL e simulado na ferramenta *Modelsim* da *Menthor Graphics*. Neste contexto, alguns dos principais comandos da ferramenta Modelsim serão apresentados brevemente a partir da próxima seção deste apêndice.

Iniciando o Modelsim

A figura C.1a e C.1b apresenta as telas iniciais do programa de simulação *Modelsim* da *Menthor Graphics*.



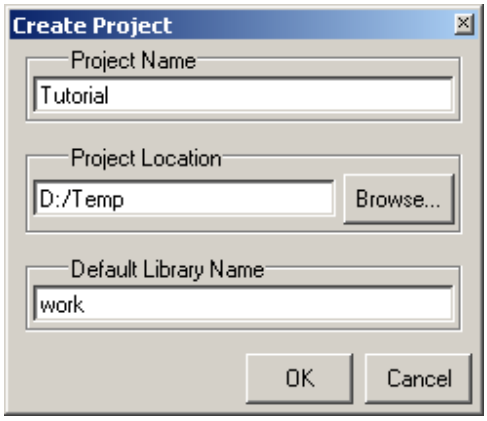
(a) (b)

Figura C.1 Telas iniciais do Modelsim

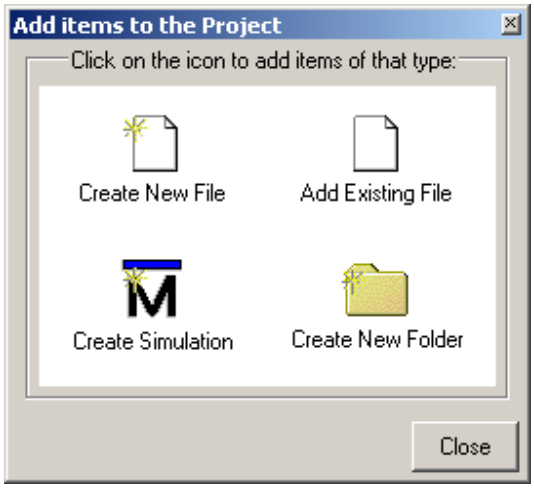
Descrição dos principais comandos

Começando um novo projeto:

File → New → Project



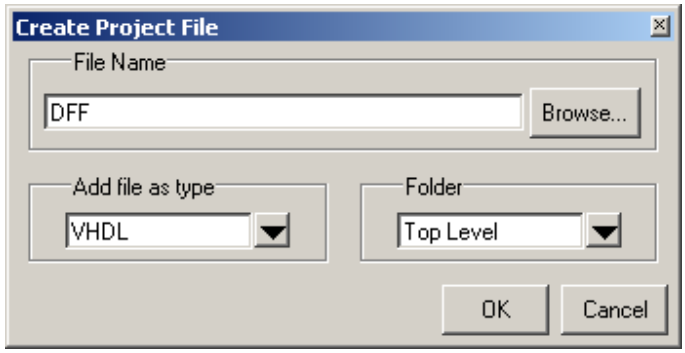
Selecione a opção desejada na caixa de diálogo **Add items to the Project**



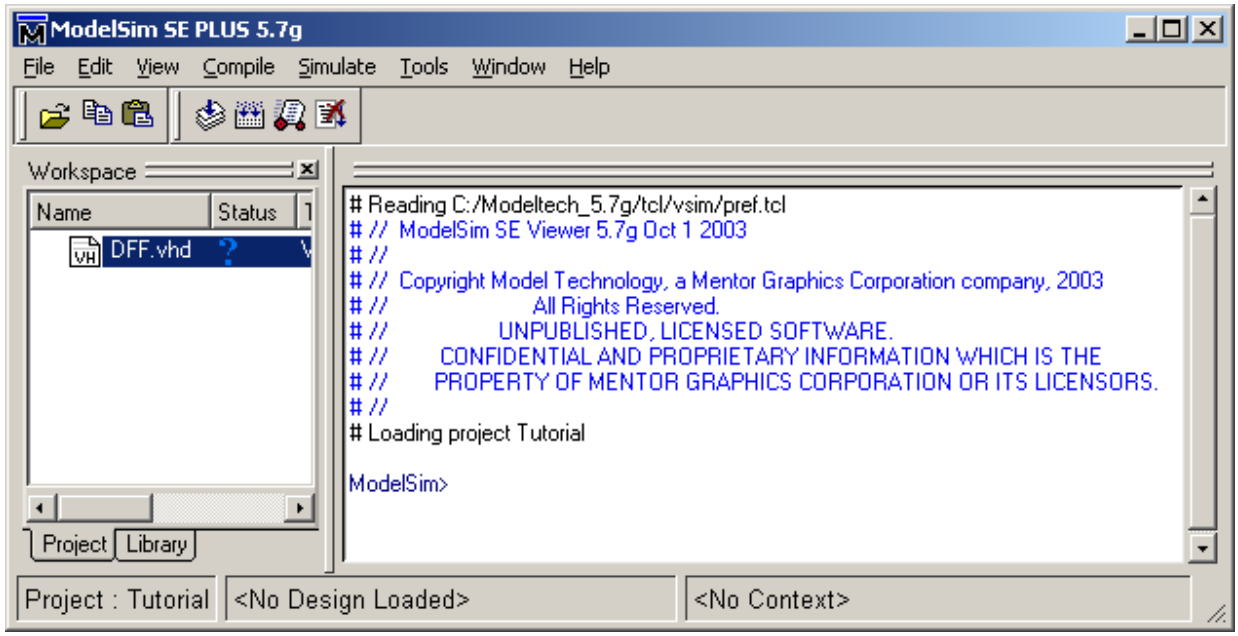
Criando um novo arquivo:

File → Add to Project → New File

Selecione as opções desejadas na caixa de diálogo “Create Project File”



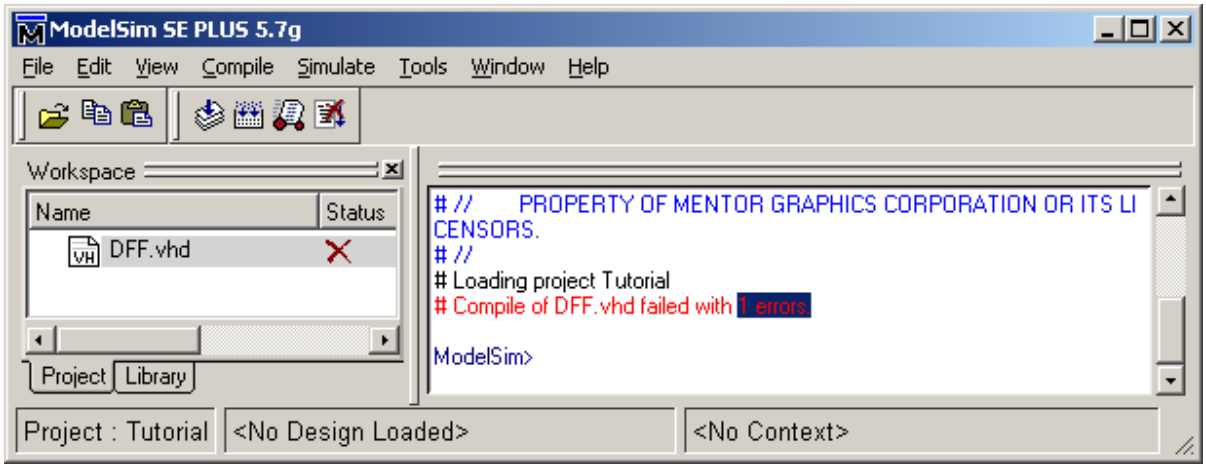
Janela de trabalho do Modelsim.



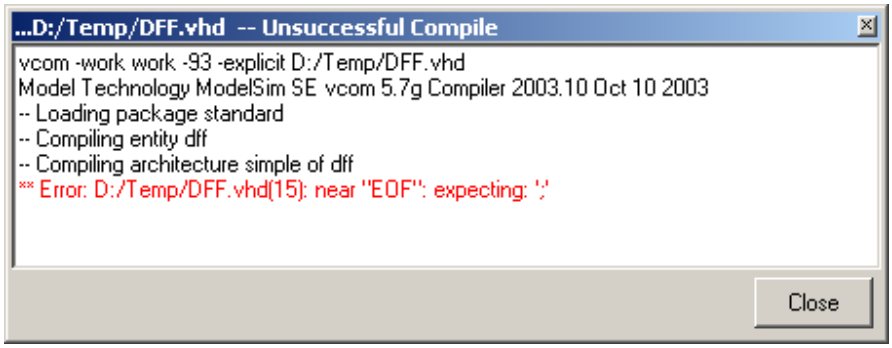
Compilando no Modelsim:

Compile → Compile All.

A janela abaixo mostra uma situação de erro de compilação.

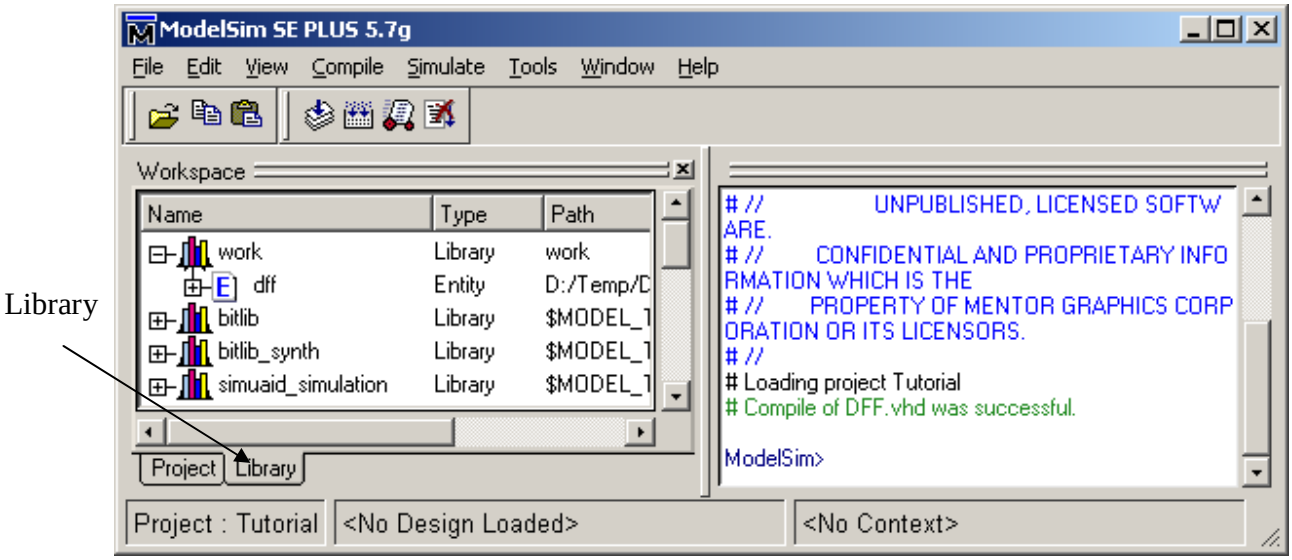


A janela abaixo mostra a mensagem de erro gerada após a tentativa de compilação.



Simulando no Modelsim:

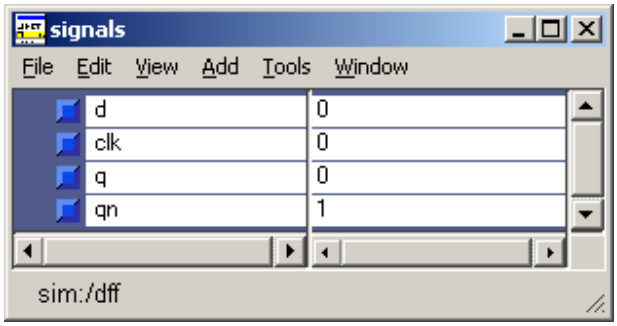
Após compilar o projeto, clique na guia *Library* indicada na janela abaixo;



Dê um duplo *click* na entidade que será simulada;

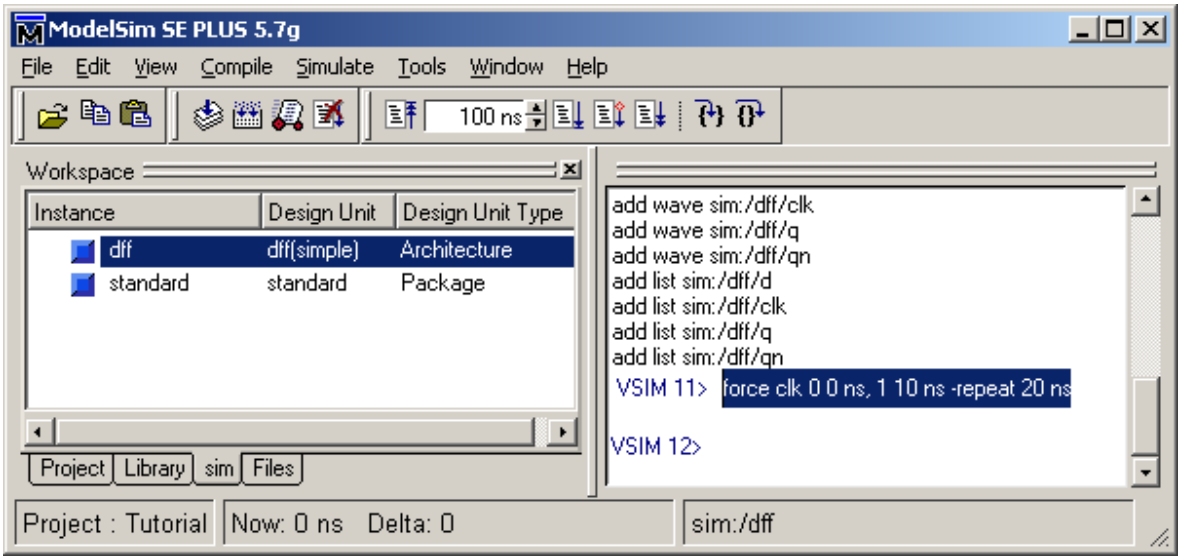
Clique em Wiew → All Windows;

A janela abaixo contem todos os sinais da entidade a ser simulada;

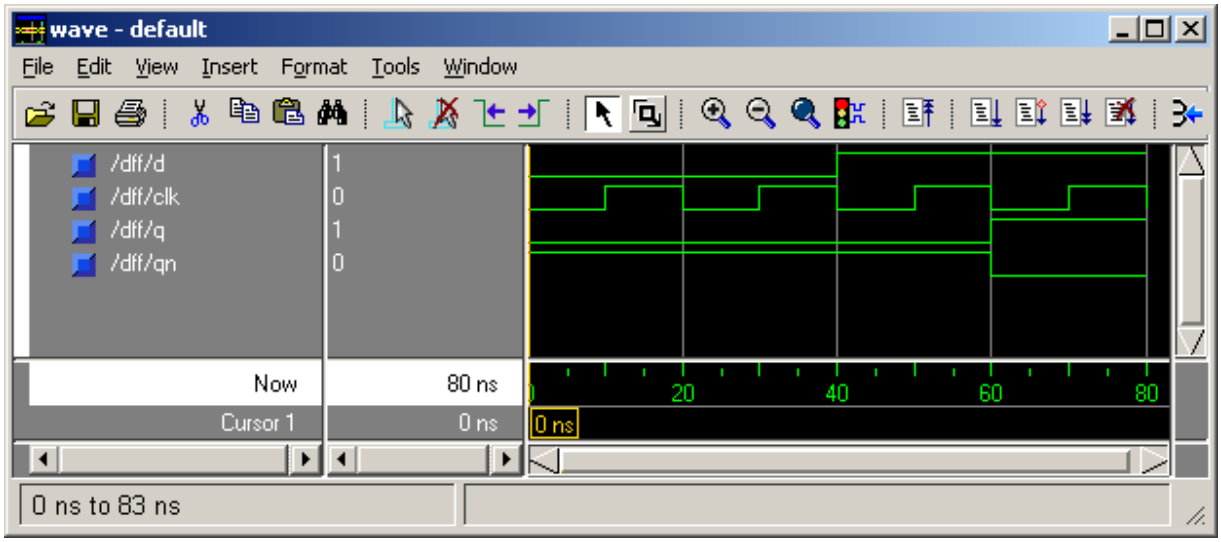


Clique em add → wave → Selected signals;

Especifique o tempo que deseja similar;



A janela abaixo mostra a onda gerada durante a simulação de uma determinada entidade.



Tutoriais disponíveis na Web:

http://www.ece.utexas.edu/projects/ee360m/spring04/ModelSim.doc

http://www.uweb.ucsb.edu/~arfaee/TTL/ModelSim.pdf

Datasheet Modelsim:

http://www.mentor.com/products/fpga_pld/simulation/modelsim_se/loader.cfm?url=/commonspot/security/getfile.cfm&pageid=26140

D. ARTIGOS PUBLICADOS

Neste item serão apresentados os artigos publicados nos congressos e revistas abaixo:

F. L. Vargas, D. Brum, D. P. Prestes, **L. Bolzani**, D. V. Lettinin. "*On the Mitigation of Conducted Electromagnetic Immunity by Means of SW-Based Fault Handling Mechanisms*", 4th IEEE Latin American Test Workshop - LATW'03, 2003, Natal, RN.

F. L. Vargas, D. Brum, D. P. Prestes, **L. Bolzani**, D. V. Lettinin., G. M. Rodrigues. "*On the Study of the Effectiveness of SW-Based Fault Handling Mechanisms to Cope with IC Conducted Electromagnetic Interference*", IX Workshop IBERCHIP, 2003, La Havana, Cuba.

F. L. Vargas, D. Brum, D. P. Prestes, **L. Bolzani**, E. L. Rhod, M. S. Reorda. "*SW-Based Fault Handling Mechanisms to Cope with EMI in Embedded Electronics: are they a good remedy? A Case Study on a COTS Microprocessor*", 9th Internacional On-Line Test Symposium - IOLTS'03, 2003, Kos Island, Greece.

F. Vargas, **L. Bolzani**, D. Becker, D. Prestes. "*Appending On-Line Fault Detection Mechanisms into Application Code to Handle EMI in Embedded Electronics: A Case Study*", Injection Ingeniería Electrónica, Automática y Comunicaciones del InstitutoSuperior Politécnico José Antonio Echeverría" (CUJAE), con Vol. XXV,No. 1, 2004, de Cuba.

L. Bolzani, M. Rebaudengo, M. Sonza Reorda, F. Vargas, M. Violante. "*An Infrastructure IP for Soft Error Detection*", 5th Latin American Test Workshop – LATW04, Cartagena, Colombia.

L. Bolzani, M. Rebaudengo, M. Sonza Reorda, F. Vargas, M. Violante. "*Hybrid Soft Error Detection by means of Infrastructure IP cores*", 10th Internacional On-Line Test Symposium – IOLTS´04, Ilha da Madeira.

P. Bernardi, **L. Bolzani**, M. Rebaudengo, M. Sonza Reorda, F. Vargas, M. Violante. "*Cerberus I-IP: an HW/SW approach to Control Flow Checking*", 2nd IEEE International Workshop on Infrastructure IP, I-IP2004, Charlotte, North Carolina, USA.

P. Bernardi, **L. Bolzani**, M. Rebaudengo, M. Sonza Reorda, F. Vargas, M. Violante. "*Pandora I-IP: an HW/SW approach to Control Flow Checking*", 6th IEEE Latin American Test Workshop - LATW'05, 2005, Salvador, BA.

P. Bernardi, **L. Bolzani**, M. Rebaudengo, M. Sonza Reorda, M. Violante, F. Vargas, "*A New Hybrid Fault Detection Technique for Systems-on-a-Chip*", artigo submetído ao IEEE Transactions on Computer.