TESE
930

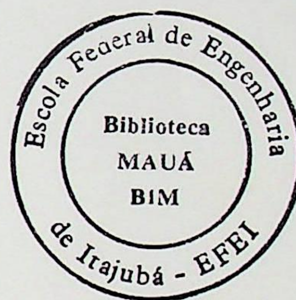
ESCOLA FEDERAL DE ENGENHARIA DE ITAJUBÁ

DISSERTAÇÃO DE MESTRADO

Correlação entre Vetores de Teste para Falhas Stuck-at e Stuck-open em Circuitos Digitais com Auto Teste Incorporado.

Autor: Marcos Rogério Cândido

Orientador: Prof. Tales Cleber Pimenta, Phd.



Dissertação de Mestrado apresentada à Escola Federal de Engenharia de Itajubá para obtenção do título de Mestre em Engenharia Elétrica.

ITAJUBÁ

1997

CLASS. 621.3.049.77 (043.2)
CUTT C217c
TOMBO. 930

Dedicatória

À Deus, pela força e coragem nos momentos mais difíceis.
Aos meus pais, pela eterna dedicação, confiança e incentivo.

AGRADECIMENTOS

Gostaria neste momento, expressar o mais profundo agradecimento ao Prof. Tales Cleber Pimenta, que durante todo o transcorrer deste trabalho sempre demonstrou o seu irrestrito apoio e confiança, fazendo da sua amizade e dedicação um dos maiores incentivos para o término deste trabalho.

ÍNDICE

RESUMO

CAPÍTULO I - Introdução

I.1 - Metodologia de Geração de Teste 04

CAPÍTULO II - Técnicas de Teste Incorporado

II.1 - Introdução 07

II.1.1 - Aspectos do BIST 08

II.2 - Estrutura BIST

II.2.1 - Implementação de BIST Pseudoexaustivo com LFSR 10

II.2.2 - Shift Register 11

II.2.3 - Shift Register com Realimentação Linear 13

II.2.4 - Polinômios Primitivos 15

II.3 - Modelamento de Falhas 19

II.3.1 - Introdução 19

II.3.2 - Falhas Equivalentes 23

II.3.2.1 - Detecção de Falhas *Stuck* em Transistores 24

CAPÍTULO III - Avaliação dos Testes
III.1 - Métodos de Simulação de Falhas 28
III.1.1 - Comparação dos Métodos de Simulação de Falhas 33
III.2 - Métodos de Simulação 34
III.2.1 - Verificação dos Testes Baseado em Códigos Lineares 38
III.3 - Análise Probabilística 41
III.4 - Análise Estatística 43

CAPÍTILO IV - CONCLUSÃO E PROPOSTAS DE EXTENSÃO 50

Referências Bibliográficas 52

Anexos
Anexo A 54
Anexo B 75
Anexo C 88
Anexo D 94

TABELA DE FIGURAS

Figura I.1 - Configuração Geral dos Testes 1

Figura I.2 - Configuração de um BIST 2

Figura II.1 - Hierarquia BIST 9

Figura II.2 - Shift Register (a) Básico, (b) com realimentação simples (cíclico) 10

Figura II.3 - Shift Register com 3 estágios 11

Figura II.4 - LFSR de n-estágios. 12

Figura II.5 - LFSR (a) com f(x) Primitivo, (b) com f(x) não-primitivo. 15

Figura II.6 - Operação de um LFSR de 4 estágios, baseado no polinômio $X^4 + X + 1$. 16

Figura II.7 - Circuito com falha stuck-at. 19

Figura II.8 - Porta NAND com falha stuck-open. 20

Figura II.9 - Exemplo de falha collapsing 22

Figura II.10 - Porta NAND nMOS com duas falhas 24

Figura II.11 - Porta NOR nMOS com quatro falhas. 26

Figura III.1 - Lista de falhas na simulação dedutiva. 28

Figura III.2 - Exemplo de simulação de falha concorrente. 31

Figura III.3 - Probabilidade de detecção versus vetores de teste 32

Figura III.4 - Estrutura básica dos simuladores 35

Figura III.5 - Circuito c17.isc. 36

Figura III.6 - Nível de defeito em função da cobertura de falha. 41

Figura III.7 - Curva de cobertura de falhas (SOPRANOXFSIM). 46

LISTA DE ABREVIAÇÕES

BIST	Built-In Self Test
BIT	Built-In Test
CUT	Circuit Under Test
DFT	Design for Testability
LFSR	Linear Feedback Shift Register
ROM	Read Only Memory
VLSI	Very Large Scale Integration

RESUMO

O custo do teste de circuitos integrados digitais complexos, tem alcançado valores muito elevados, podendo atingir cerca de 30% do custo final do dispositivo. Como algumas falhas podiam ser detectadas por uma combinação adequada de valores, outras exigiam seqüências específicas para serem detectadas. Assim, faz-se simulações específicas de testabilidade dos circuitos para cada uma destas falhas. Este trabalho, procurou estabelecer estatisticamente, uma correlação entre elas, de forma tal que, conhecendo-se o número de falhas combinacionais detectadas, pode-se estimar (com razoável precisão) o número de falhas seqüenciais detectadas e vice-versa.

Esta análise pode ser feita para vários vetores de teste, onde se pode relacionar o mais adequado para o circuito em teste. Isto permite testes mais rápidos, evitando-se a necessidade, e os custos computacionais, de uma simulação para cada caso.

I - INTRODUÇÃO

Os avanços na tecnologia VLSI (*Very Large Scale Integration*) oferecem Circuitos Integrados (CI) com baixo consumo de potência, alta velocidade, boa confiabilidade e elevada performance. Essas vantagens estão aliadas a habilidade de integrar milhares de dispositivos e conexões em um *chip*. Se os *chips* não puderem ser testados economicamente, essas vantagens podem ser superadas pelo preço final do CI.

Os testes aplicados ao *chip* devem verificar todas as possíveis falhas, que possam impedir o perfeito funcionamento do dispositivo. Em outras palavras, os testes devem verificar se o circuito foi devidamente fabricado e se executa todas as funções projetadas.

O rápido crescimento na densidade e da complexidade dos circuitos, tem tornado os testes cada vez mais dificeis e caros. Os testes tornaram-se uma parcela considerável no preço final dos circuitos VLSI, sendo que atualmente este valor chega aproximadamente a 30% do custo do dispositivo [Levitt 92].

O crescimento na densidade e na complexidade dos circuitos VLSI, ajudaram no desenvolvimento de alguns circuitos adicionais, dedicados ao testes dentro dos *chips*. Este conceito é conhecido como Teste Incorporado - BIT (*Built-in Test*), ou BIST (*Built-in Self Test*). O BIST pode executar no chip, a geração e verificação dos testes, com uma cobertura de falhas adequada.

A configuração geral de teste para um circuito digital (placa ou *chip*) é apresentada na Figura I.1.

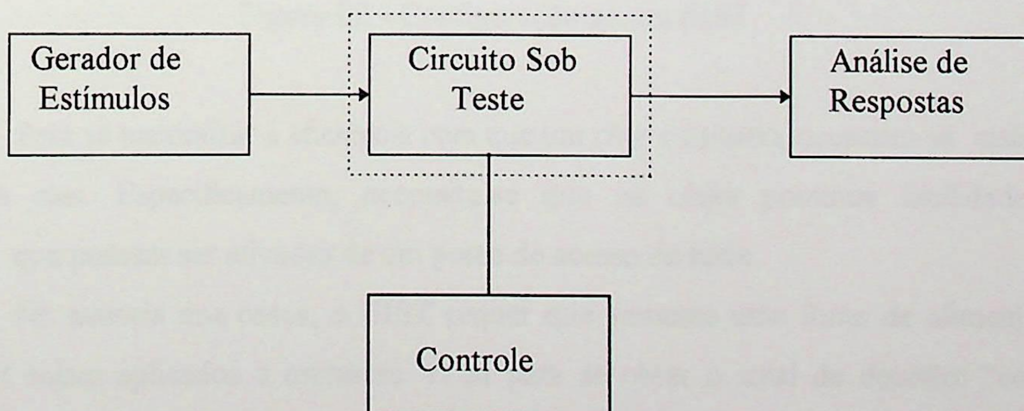


Figura I.1 - Configuração Geral dos Testes.

A organização geral do BIST é mostrado na Figura I.2, onde a área tracejada representa o CI. Pode-se observar que neste caso o circuito integrado incorpora o controle de testes, a geração dos vetores de teste e a análise das respostas.

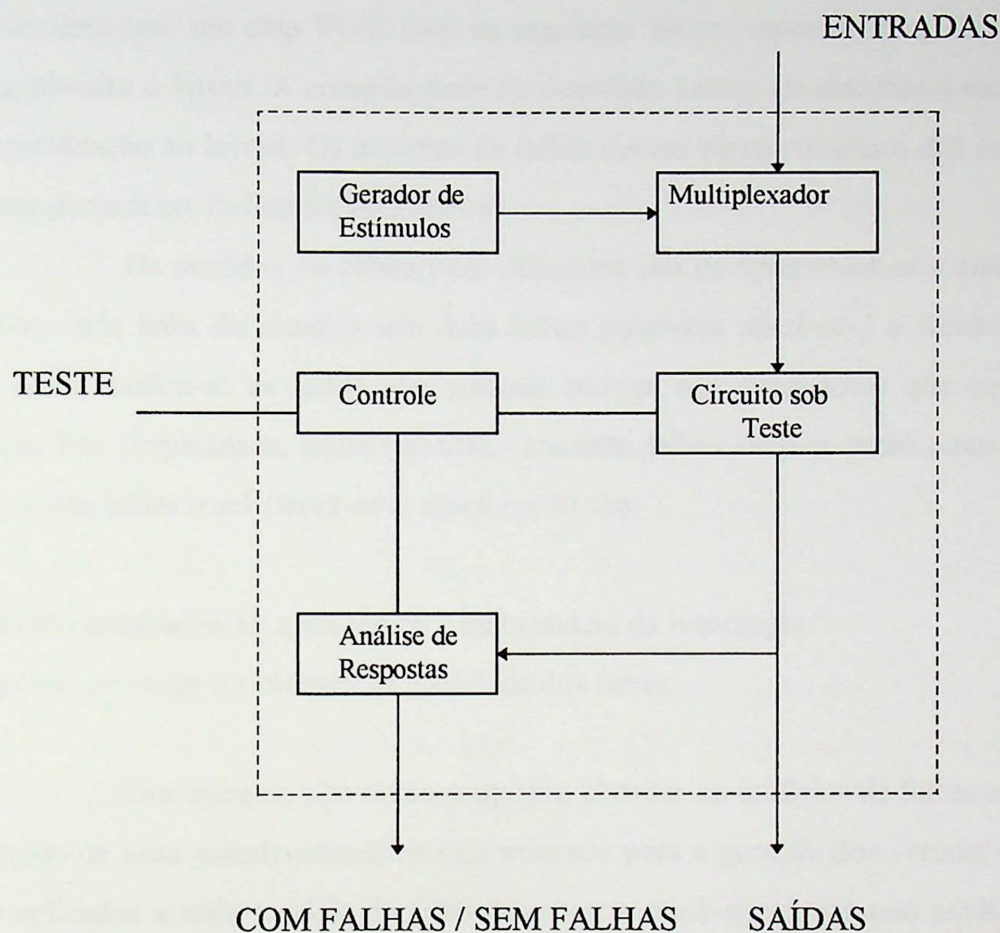


Figura I.2 - Configuração de um BIST.

Para se maximizar a eficiência com que um *chip* é testado, necessita-se mais do que se ter acesso a eles. Especificamente, necessita-se que os *chips* possuam facilidade de teste incorporado, que possam ser ativadas de um porto de acesso de teste.

Na maioria dos casos, o BIST requer que somente uma fonte de alimentação e um sinal de *clock* sejam aplicados à estrutura VLSI para se obter o sinal de decisão: "com ou sem falhas". Isto se torna muito importante, já que os equipamentos baseados em testes são muito caros, podendo chegar a milhões de dólares.

A colocação de circuitos de teste dentro dos CIs não é suficiente para resolver os problemas de teste. A densidade dos circuitos VLSI e o seu número de pinos, tornam os testes extremamente longos.

Os modelos práticos de falha dependem da descrição do *chip*, da tecnologia e, em alguns casos, da fase particular do ciclo de vida do chip onde a análise é conduzida. Tipicamente pode-se descrever um chip VLSI com os seguintes níveis: especificação, comportamento, função, lógica, circuito e *layout*. A complexidade da descrição cresce em detalhes à medida que se desloca da especificação ao layout. Os modelos de falhas devem simular o efeito dos erros de forma que os mesmos possam ser facilmente detectados.

Os modelos de falhas mais utilizados são os tipos *stuck-at e stuck-open*. No modelo *stuck-at*, cada linha do circuito tem duas falhas possíveis: *stuck-at-1* e *stuck-at-0*. Já no modelo *stuck-open*, analisa-se as falhas que possam ocorrer nos transistores que compõem a malha do circuito. Por simplicidade, neste trabalho, somente falhas simples serão consideradas. Os maiores avanços das falhas *stuck* (*stuck-at* & *stuck-open*) são:

a) elas são modeladas ao nível lógico e independem da tecnologia;
b) elas têm provado ser efetivas na qualidade dos testes.

Este trabalho têm como propósito abordar os modelos de falhas *stuck-at* e *stuck-open* O método de teste pseudo-exaustivo será utilizado para a geração dos vetores de teste.Tais vetores serão aplicados a cada modelo de falha (*stuck-at* e *stuck-open*), visando avaliar o comportamento dos mesmos.

O restante deste capítulo é dedicado à introdução da terminologia e dos conceitos de testabilidade, além da revisão de técnicas de projetos para testabilidade (no qual é baseado os conceitos para o BIST).

I.1 - METODOLOGIA DE GERAÇÃO DE TESTE

O testador de circuitos integrados de alta velocidade, incluindo todos seus circuitos é astronomicamente caro. A redução do tempo de teste de um simples componente pode ter um grande impacto no custo final do dispositivo. A utilização de modelos de falhas adequados é o primeiro passo para se obter circuitos com custo reduzido . Como a natureza e o número de falhas que podem ocorrer dependem do tipo do dispositivo (*chip*, placa e assim por diante) e da tecnologia (CMOS, bipolar, GaAs) então, a avaliação da qualidade de um teste pode ser uma tarefa complicada. Em geral, requisitos de qualidade tais como 95% de cobertura de todas as falhas para *chips* VLSI e 100% de cobertura de todas as falhas de interconexões de uma placa de circuito impresso, são baseados em considerações práticas.

Vários são os fatores necessários para se obter um alto nível de qualidade a um preço cada vez menor. Os custo são constituídos de: custo com o programa de geração automática, custo de desenvolvimento (ferramentas CAD, geração dos vetores, programação dos testes), e custo do projeto de testabilidade [Pitt84, Ambl86]. Em um futuro bem próximo, os projetos de testabilidade se tornarão o fator preponderante na equação de economia dos testes.

O fator decisivo na escolha do BIST pelos projetistas é a relação custo-benefício. Deve-se avaliar de forma global tal relação, pois mesmo que pequenas reduções de custo sejam notadas ao nível do *chip*, BIST pode ser muito importante ao longo do ciclo de vida do dispositivo [Dear91].

A Tabela I.1 apresenta o resumo da análise do BIST no custo do teste de *chips*, placas e sistemas [Amazonas96].Considerar:

+ : aumento de custo;

- : redução de custos

+/- : aumento e redução de custos equilibrados

	Projeto, desenvolvimento de teste	Fabricação	Teste	Teste e manutenção	Diagnóstico e Reparo	Interrupção de serviço
CHIPS	+/-	+	-			
PLACAS	+/-	+	-		-	
SISTEMAS	+/-	+	-	-	-	-

Tabela I.1 - Custos do BIST.

O principal ponto da Tabela I.1 é o benefício significativo que BIST oferece ao nível do sistema. Assim, mesmo com benefícios consideravelmente menores ao nível de placas e de *chips*, BIST coloca-se como uma das melhores técnicas de projetos voltadas para a testabilidade.

Os testes de circuitos podem ser realizados *off-line* ou *on-line*. Tais testes são realizados para se averiguar o perfeito funcionamento da estrutura do circuito, sendo geralmente realizados pelos fabricantes.

Considerando-se os conceitos de base do BIST, deve-se observar sua arquitetura básica e sua aplicação hierárquica. A seguir são focalizados dois componentes específicos de BIST: geração de vetores e análise das respostas.

A arquitetura básica BIST requer a adição de três blocos de *hardware* a um circuito digital: um gerador de padrões (vetores de teste), um analisador de resposta e um controlador de teste. Exemplos de geradores de padrões são uma ROM (*Read-Only Memory*) com vetores armazenados, um contador e um registrador de deslocamento com realimentação linear (Linear Feedback Shift Register - LFSR). Um analisador de resposta típico é um comparador com respostas armazenadas ou um LFSR usado como analisador de assinatura. Um bloco de controle é necessário para ativar o teste e analisar as respostas. Em geral, entretanto, várias das funções relativas ao teste podem ser executadas por um circuito gerenciador (controlador) de teste.

Em testes armazenados, um conjunto de determinados vetores de teste, gerados a partir de algum padrão de geração, pode ser armazenado em uma memória ROM (*Read-Only Memory*). Infelizmente as memórias ROM se tornaram pequenas para o tamanho e complexidade dos circuitos, onde seu tamanho é proporcional ao número de vetores de teste.

Testes exaustivos implicam na aplicação de todos 2^n testes, onde n é o número de entradas do circuito sob teste CUT (*Circuit Under Test*). Aplicando-se todos os possíveis vetores de teste ao CUT, pode-se garantir que todas as falhas serão detectadas. A desvantagem deste método é o tempo para se testar circuitos muito complexos. Como exemplo, considere um circuito com 64 entradas, com os teste aplicados a uma média de 1 nanosegundo, seriam necessários aproximadamente 585 anos para testar o circuito por completo. Utilizando as mesmas considerações, um circuito com 32 entradas, requer apenas 9 segundos [Elder59].

Testes aleatórios são aplicados por sub-conjuntos de teste exaustivo (de forma aleatória). Na verdade o processo poderia ser chamado de pseudo-aleatório, pois a seqüência de teste é previsível e repetitiva. O número de testes nesse sub-conjunto pode ser obtido através de medidas probabilísticas do circuito (baseado em métodos analíticos).

Este trabalho explora a existência dos métodos de teste pseudo-exaustivo, para fazer a análise das falhas *stuck-at* e *stuck-open*. Buscando dessa forma, encontrar a correlação entre estes modelos de falhas, de tal sorte que, a partir de uma determinada falha (por exemplo *stuck-at*), se possa com segurança avaliar o comportamento do dispositivo que sob teste, a presença de outra falha (*stuck-open*), trazendo assim, economia no preço final do dispositivo.

No capítulo II é feita uma revisão das técnicas utilizadas em teste incorporado. Os modelos de falhas, bem como a aplicabilidade dos mesmos.

No capítulo III, apresenta-se os resultados alcançados na análise dos modelos de falhas *stuck-at* e *stuck-open*, bem como o comportamento de tais modelos na presença dos de circuito padronizados (ISCAS85).

O capítulo IV traz as conclusões e propostas de extensão.

II - TÉCNICAS DE TESTE INCORPORADO

Pode-se definir *Built-in Self Test* (Auto-teste Incorporado) como a capacidade que uma rede tem de aplicar padrões de teste, compactar os resultados obtidos de modo a permitir que sejam observados quando o teste for completado e comparados com a resposta sem falhas. O crescente aumento da complexidade dos projetos dos CI's dedicados de alta velocidade e a difusão do uso de componentes de montagem de superfícies - SMD (*Surface Mounted Devices*) têm contribuído para a aceitação desta metodologia, pois ela minimiza os requisitos exigidos dos equipamentos de teste, permitindo que o teste possa ser realizado através de conectores ligados ao sistema [Agra93a] [Agra93b].

II .1 - INTRODUÇÃO

O teste e o diagnóstico devem ser rápidos e eficazes para que um sistema digital desempenhe as funções para as quais foi projetado. Para se conseguir tal propósito emprega-se o auto-teste, especificando-se ainda o teste como uma função do sistema.

Sistemas digitais envolvem uma hierarquia de componentes: chips, placas, gabinetes e assim por diante. No nível mais elevado que pode envolver todo o sistema, a operação é controlada por *software*. Tal teste pode ter uma baixa resolução de diagnósticos, pois ele deve testar componentes projetados sem requisitos específicos de testabilidade além disso, um bom programa de teste pode ser muito longo, lento e muito caro para ser desenvolvido.

Uma alternativa que tem se tornado cada vez mais interessante é o auto-teste incorporado (*built-in self-test*) ,isto é, um auto-teste (*self-test*) implementado pelo próprio *hardware* (*built-in*) . O *Built-in self-test* é universalmente designado por BIST.

O BIST é uma técnica na qual o teste (geração e aplicação do teste) é conseguido através de características incorporadas ao *hardware*.

II.1.1 - Aspectos do BIST

A incorporação do teste ao *hardware,* além de tornar o circuito hierárquico, também traz o benefício do aumento da velocidade e da eficiência. O custo-benefício que pode parecer não muito significativo ao nível do *chip*, é enorme ao nível do sistema. Em geral, BIST oferece soluções para os maiores problemas de teste.

Considere o teste de um *chip* inserido em uma placa que é uma parte do sistema. Para testar o chip, o sistema envia um sinal de controle para a placa, que por sua vez, ativa o auto-teste no *chip* e envia o resultado de volta ao sistema. Assim, BIST provê um teste eficiente dos componentes utilizados e interconexões, reduzindo a carga ao nível do teste do sistema, que precisa somente verificar a sinergia funcional entre componentes.

Padrões BIST exaustivos eliminam técnicas de geração de teste e possuem alta cobertura de falha. Para se testar um bloco de lógica combinatória de n entradas devem ser aplicadas todas as 2^n possíveis combinações de entrada. Mesmo com elevadas velocidades de relógio, o tempo necessário para aplicar os padrões pode tornar o BIST, com padrões exaustivos, impraticável para um circuito com n maior do que 25. Desta forma, deve-se particionar ou segmentar a lógica em blocos menores, talvez com superposição, com número de entradas menor que n. Cada bloco é, então, testado exaustivamente. Esta técnica é chamada de BIST pseudo-exaustivo [Amaz88],[McCI84].

II.2. - ESTRUTURA BIST

Considere uma aplicação hierárquica do conceito BIST. O sistema consiste de várias placas de circuito impresso. Cada placa pode conter vários *chips* VLSI. A Figura II.1 mostra tal sistema [Agra93a], [Agra93b]. Nesta figura, CUT indica o circuito sob teste (*Circuit Under Test).*

Ao nível do sistema o gerenciador de teste ativa simultaneamente o auto-teste em todas as placas. O gerenciador de teste de um *chip* é responsável pela execução do auto-teste no chip e depois pela transmissão do resultado (com ou sem falha) ao gerenciador de teste da placa que contém o *chip.* Os resultados dos testes de todos os *chips* são acumulados no gerenciador de teste do sistema. Usando esses resultados, o gerenciador de teste do sistema pode isolar *chips* e placas com falha.

A eficácia deste procedimento de diagnóstico depende de quão completo é o auto-teste implementado nos *chips*. Assim, a cobertura de falha é um dos pontos principais em projetos BIST. Outros aspectos importantes são a área adicional e seu impacto no rendimento de fabricação, pinos extras necessários para o teste e perda de desempenho [Glos89], [Chal89].

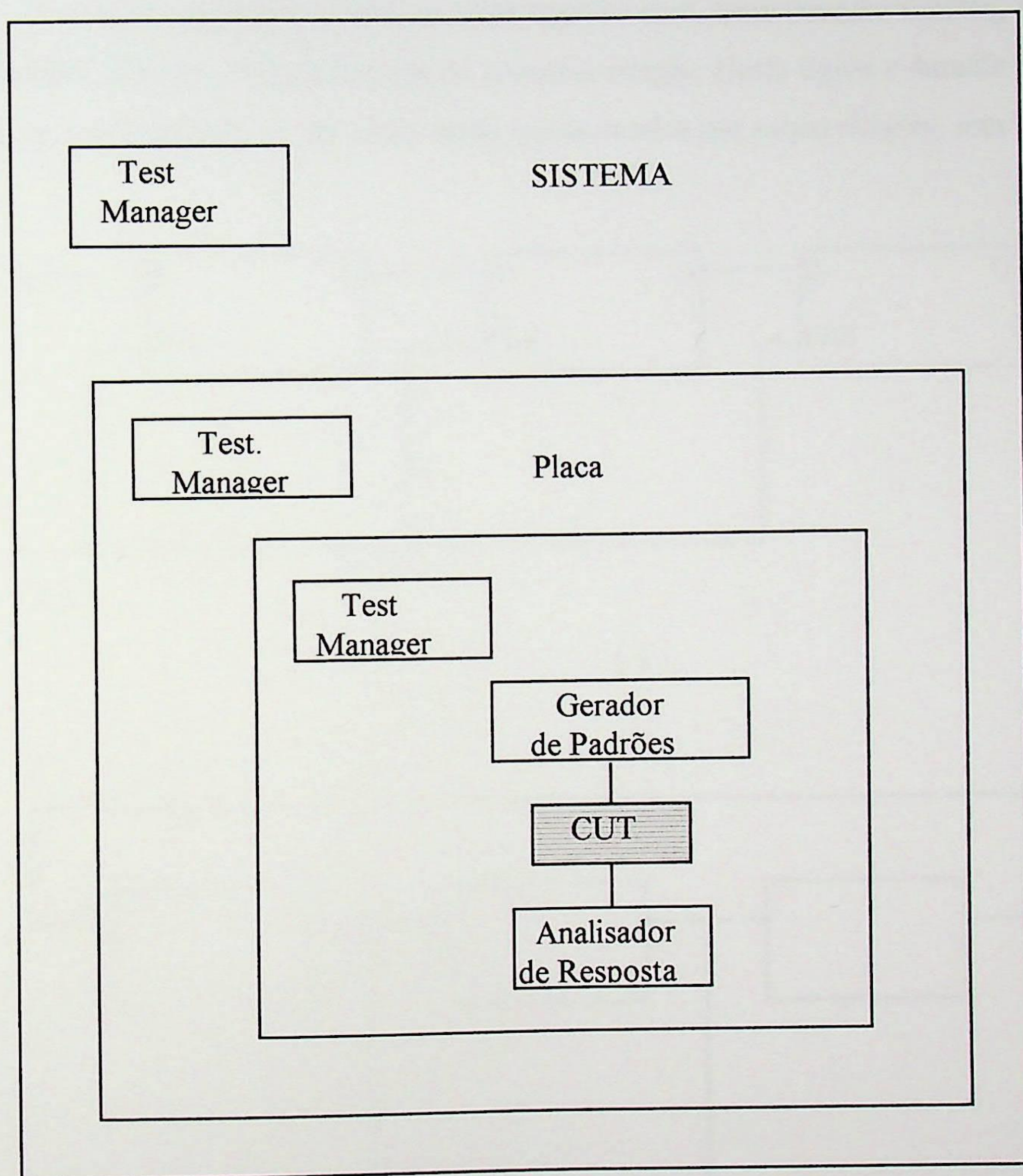


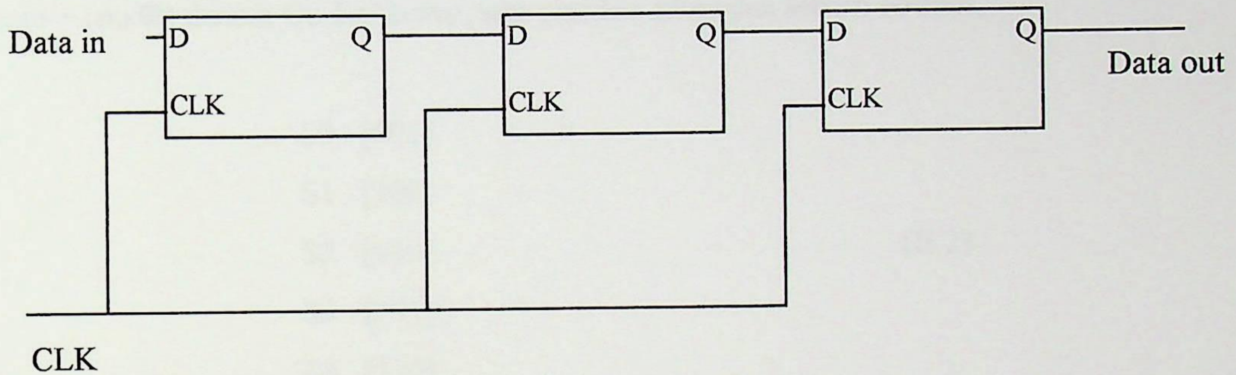
Figura II.1 - Hierarquia BIST

Escola Federal de Engenharia
Biblioteca
MAUÁ
BIM
de Itajubá - EFEI

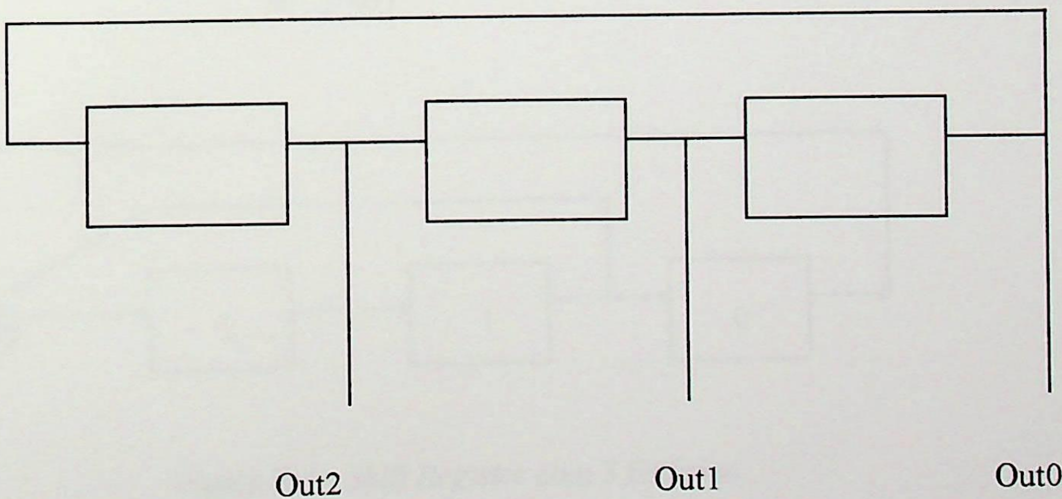
II.2.2 - *Shift Register*

Um *shift register* é um conjunto de *flip-flops*, ligados de forma que as saídas de um *flip-flop* (Q e Q') alimentem as entradas (D) do *flip-flop* seguinte. A Figura II.2.a mostra um *shift register* de 3 estágios, onde cada estágio é um elemento de armazenagem.

A Figura II.2.b mostra um *shift register* com realimentação simples, onde a saída do último estágio é alimentada pela entrada do primeiro estágio. Nesta figura e durante o decorrer deste trabalho, por simplicidade, os *flip-flops* serão representados por caixas simples, sem linhas de *clock*.



(a)



(b)

Figura.II.2 -*Shift Register* (a) Básico, e (b) com realimentação simples (cíclico).

Tomando-se [001] como estado inicial no *shift register* da Figura II.2.b, o estado seguinte será [100] depois do pulso de *clock*, então [010], e voltará para o estado inicial [001], ou seja:

S0 [001]
S1 [100] (II.1)
S2 [010]
S0 [001]

No entanto, se as conexões do *shift register* fossem conforme as da Figura II.3, onde o símbolo + (ou ⊕) denota Ou-Exclusivo, sete estados diferentes seriam obtidos,

S0 [001]
S1 [100]
S2 [010] (II.2)
S3 [101]
S4 [110]
S5 [111]
S6 [011]
S0 [001]

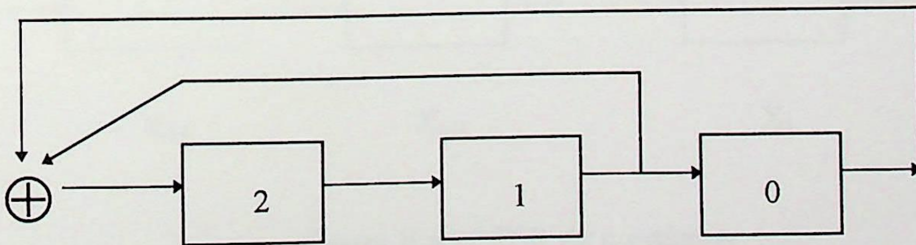


Figura II.3 - *Shift Register* com 3 Estágios.

II.2.3 - *Shift Register* com Realimentação Linear

Um circuito digital é dito linear se o mesmo for composto por unidades de armazenamento (*latches*), somadores módulo-2 (XOR's) e multiplicadores de módulo-2 (operação *shift*). Os circuitos digitais lineares têm a propriedade de que sua resposta às combinações lineares das entradas preservam o princípio da superposição.

A Figura II.4 mostra a configuração padrão de um LFSR. O estado presente do n-estágio (no tempo t) é $[x_o\ (t), x_1\ (t), ..., x_{n-1}(t)]$.

O próximo estado do LFSR (no tempo t+1) é $[x_o\ (t+1), x_1\ (t+1), ..., x_{n-1}\ (t+1)]$.

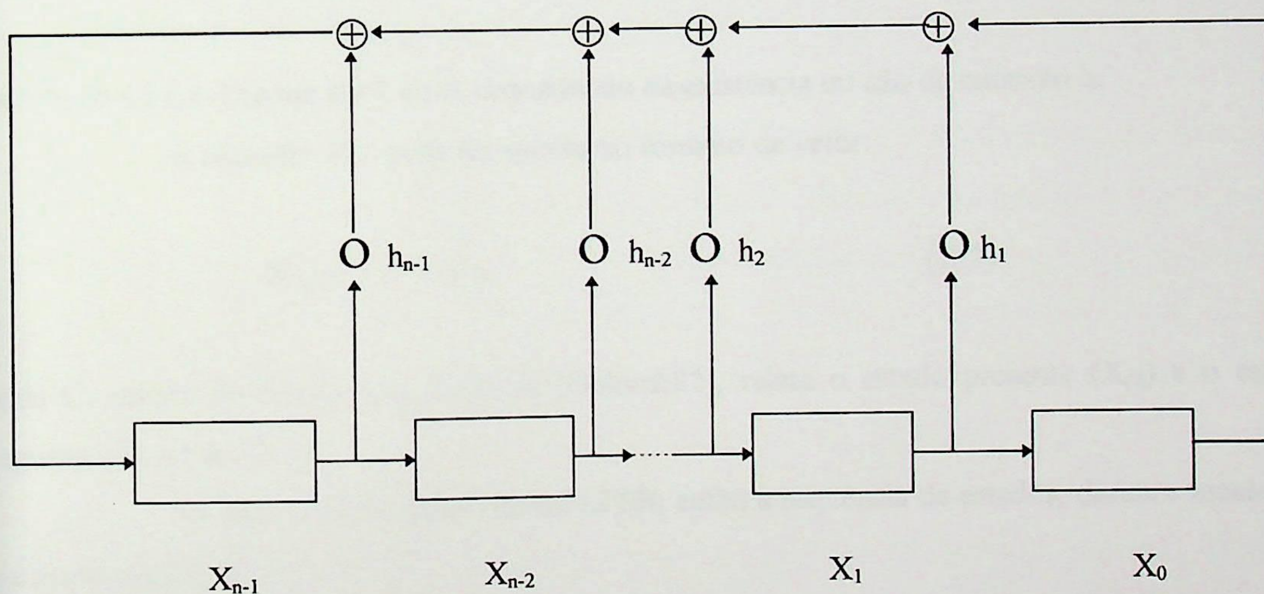


Figura II.4 - LFSR de n-estágios.

A relação entre os estágios atual e seguinte do LFSR é representado por:

$$\begin{bmatrix} x_0\,(t+1) \\ x_1\,(t+1) \\ x_2\,(t+1) \\ \vdots \\ x_{n-2}\,(t+1) \\ x_{n-1}\,(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & \cdots & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & & 0 & 1 \\ 1 & h1 & & h2 & h_{n-1} \end{bmatrix} * \begin{bmatrix} x_0\,(t) \\ x_1\,(t) \\ x_2\,(t) \\ \vdots \\ x_{n-2}\,(t) \\ x_{n-1}\,(t) \end{bmatrix}$$

onde h_i $(0 \leq i \leq n-1)$ pode ser 1 ou 0, dependendo da existência ou não da conexão h_i.

A equação II.1 pode ser escrita no formato de vetor:

$$X_{(t+1)} = C * X_{(t)} \qquad \text{(II.2)}$$

onde C, matriz de comparação (n-by-n) [Golomb82], relata o estado presente ($X_{(t)}$) e o estado seguinte ($X_{(t+1)}$) do LFSR.

Se X_0 é o estado inicial de um LFSR, então a seqüência de estados, durante sucessivos pulsos de relógio seria:

$$X_0,\ C*X_0,\ C^2*X_0,\ C^3*X_0 \qquad \text{(II.3)}$$

O polinômio característico f(x) de um LFSR é definido como o determinante de C - Ix, ou :

$$f\,(x) = |C - Ix| \qquad \text{(II.4)}$$

onde I é a matriz identidade.

A equação acima pode ser escrita como:

$$f(x) = 1 + h_1x + h2x^2 + \ldots\ldots\ldots\ldots + h_{n-}1xn^{-1} + x^h \qquad (II.5)$$

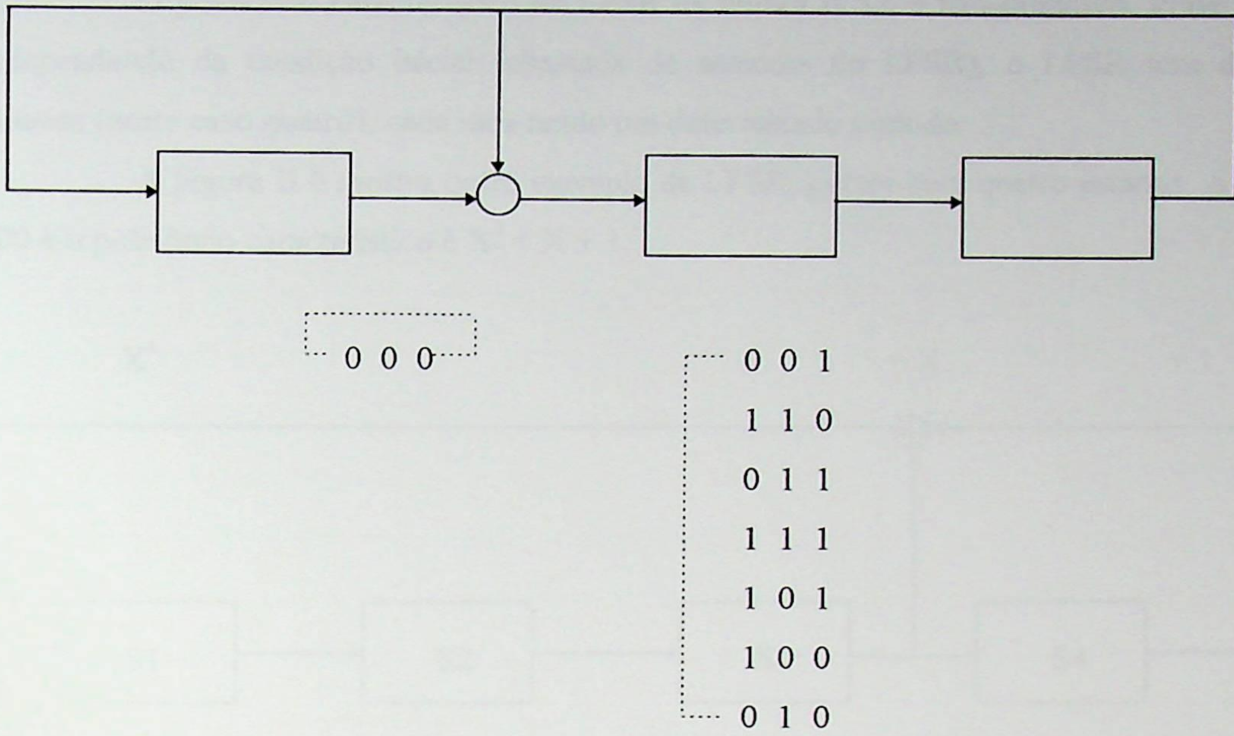
Sendo dado um LFSR, o seu polinômio característico, pode ser obtido por inspeção. Da mesma forma, a partir do polinômio característico, o LFSR correspondente pode ser facilmente obtido, mostrando a versatilidade desta equação.

O menor T e tal que $C^T = I$, é o período (ou máximo tamanho do ciclo) de um LFSR. Pode ser provado [Bardell87] que o período T é o menor inteiro pelo qual f (x) divide o polinômio $1 + x^T$.

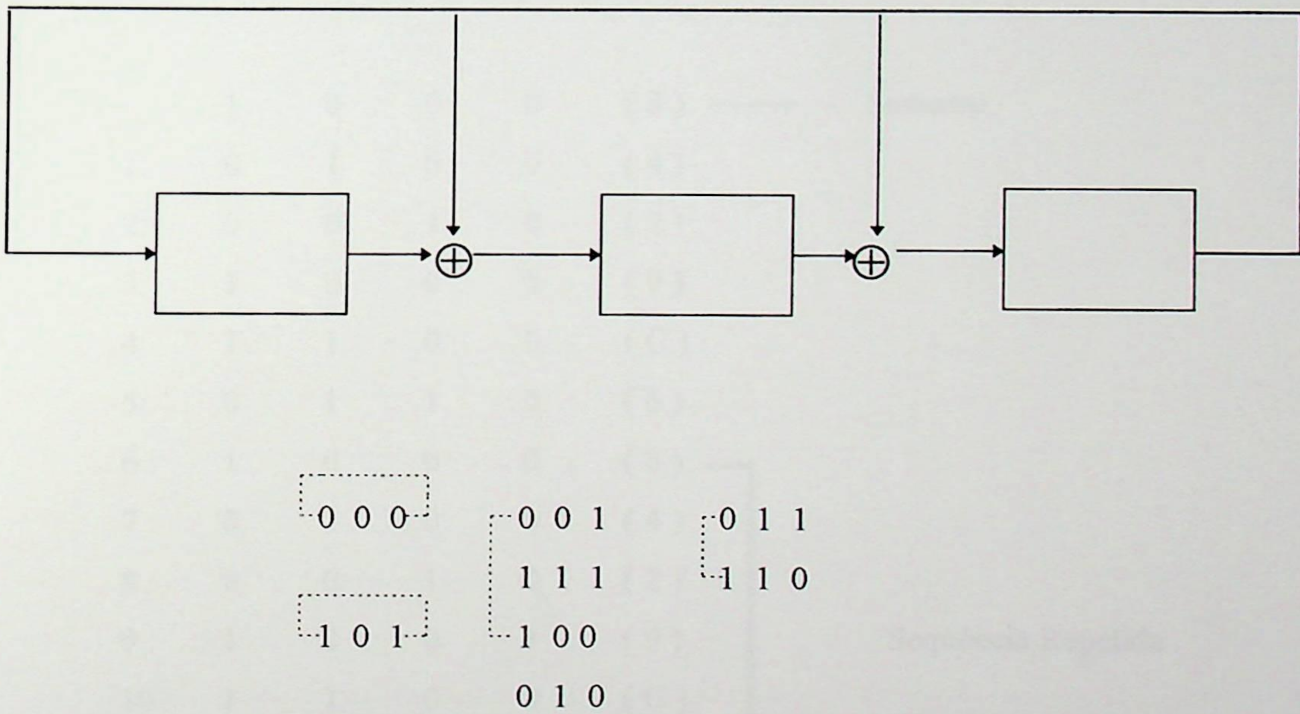
II.2.4 - Polinômios Primitivos

Se $T = 2^n - 1$, então a seqüência gerada pelo LFSR é chamada de seqüência máxima, ou seqüência m. O polinômio característico de seqüência máxima é conhecido como polinômio primitivo. Somente polinômios primitivos podem gerar a seqüência m, que corresponde a todos 2^n vetores de teste, exceto o vetor zero. Para polinômios primitivos, o menor inteiro pelo qual f(x) divide $1 + x^T$ é $T = 2^n - 1$. Em outras palavras, um polinômio f(x) de grau n é primitivo se divide $1 + X^T$.

Para clarear a diferença entre polinômios primitivos e não-primitivos, considere a Figura II.5, que mostra um LFSR implementando um polinômio primitivo de grau três. Pode ser visto que, exceto para o estado 0, o LFSR gera todos os outros estados, e o período é $2^3 - 1$ ou 7. A Tabela II.2, mostra o exemplo de alguns polinômios característicos.



(a)

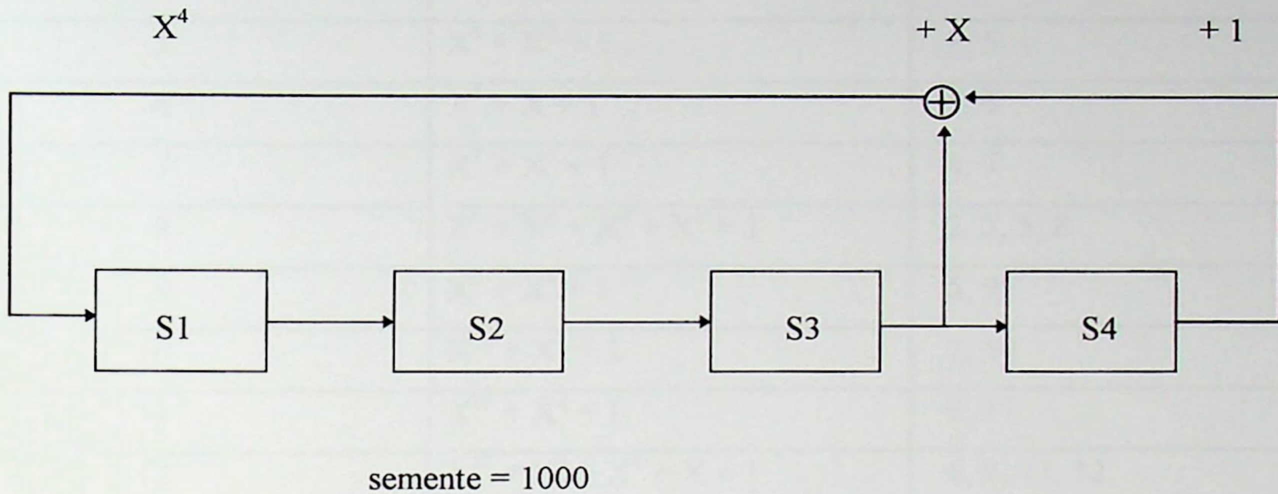


(b)

Figura II.5 - LFSR (a) com f (x) Primitivo e (b) com f(x) Não-Primitivo.

O polinômio característico do LFSR da Figura II.5.b é não-primitivo. Pode ser visto que dependendo da condição inicial (chamada de semente do LFSR), o LFSR terá diferentes seqüências (neste caso quatro), cada uma tendo um determinado período.

A Figura II.6 mostra outro exemplo de LFSR, porém com quatro estados. A semente é 1000 e o polinômio característico é $X^4 + X + 1$.



	1	0	0	0	(8)	⟶ - Semente
1:	0	1	0	0	(4)	
2:	0	0	1	0	(2)	
3:	1	0	0	1	(9)	
4:	1	1	0	0	(C)	
5:	0	1	1	0	(6)	
6 :	1	0	0	0	(8)	
7 :	0	1	0	0	(4)	
8 :	0	0	1	0	(2)	
9 :	1	0	0	1	(9)	Seqüência Repetida (linhas 6 a 12)
10 :	1	1	0	0	(C)	
11 :	0	1	1	0	(6)	
12 :	1	0	0	0	(8)	

Figura II.6 - Operação de um LFSR de 4 estágios, baseado no polinômio $X^4 + X + 1$.

GRAU	POLONÔMIO PRIMITIVO MÍNIMO	REALIMENTAÇAO DO LATCH (da esquerda para a direita)
2	$X^2 + X + 1$	1, 2
3	$X^3 + X + 1$	1, 3
4	$X^4 + X + 1$	3, 4
5	$X^5 + X^2 + 1$	3, 5
6	$X^6 + X + 1$	5, 6
7	$X^7 + X^3 + 1$	4, 7
8	$X^8 + X^6 + X^5 + X^3 + 1$	2, 3, 5, 8
9	$X^9 + X^4 + 1$	5, 9
10	$X^{10} + X^3 + 1$	7, 10
11	$X^{11} + X^2 + 1$	9, 11
12	$X^{12} + X^6 + X^4 + X + 1$	6, 8, 11, 12
13	$X^{13} + X^4 + X^3 + X + 1$	9, 10, 12, 13
14	$X^{14} + X^{10} + X^6 + X + 1$	4, 8, 13, 14
15	$X^{15} + X + 1$	14, 15
16	$X^{16} + X^{12} + X^3 + X + 1$	4, 13, 15, 16
17	$X^{17} + X^3 + 1$	14, 17
18	$X^{18} + X^7 + 1$	11, 18
19	$X^{19} + X^5 + X^2 + X + 1$	14, 17, 18, 19
20	$X^{20} + X^3 + 1$	17, 20
21	$X^{21} + X^2 + 1$	19, 21
22	$X^{22} + X + 1$	21, 22
23	$X^{23} + X^5 + 1$	18, 23
24	$X^{24} + X^7 + X^2 + X + 1$	17, 22, 23, 24
25	$X^{25} + X^3 + 1$	22, 25

Tabela II.2 - Alguns Polinômios Primitivos de Graus 2 a 25.

II.3 - MODELAMENTO DE FALHAS

Da mesma forma que outras análises, a análise de falhas requer modelamento (ou abstração). Modelos de falha servem a dois propósitos. Primeiro, eles ajudam a gerar os testes, e, segundo, ajudam a avaliar a qualidade dos testes, definido em termos da cobertura das falhas. Um modelo bom é aquele que é simples de ser analisado e ao mesmo tempo represente o comportamento das falhas fisicas do circuito.

Circuitos digitais são comumente descritos como a interconexão de portas lógicas. Pode-se considerar uma série de falhas possíveis, por exemplo: perda de interconexões, curto-circuito entre interconexões, etc. A maioria dessas falhas, apesar de serem fisicamente reais, também são complexas para se modelar. O modelo que tem sido comumente usado são as falhas *stuck*. [ChanM70], [FrieM71], [Suss73] e [Bren76].

II.3.1 - Introdução

Sistemas de testabilidade digitais trabalham com detecção de falhas de *hardware*, que são na realidade estados impróprios de um dado sistema, resultado de falha de componentes, erros operacionais, ou mesmo projeto incorreto [Avizienis75]. Um erro é um sinal de saída incorreto causado por uma falha, quando padrões de teste são aplicados nas entradas. A detecção de falhas envolve a aplicação de padrões de teste, que produzem erros causados por algum conjunto de específico de falhas.

As falhas podem ser divididas em permanentes e transientes. Falhas permanentes representam os defeitos fisicos que são estáveis e contínuos, e que geralmente causam danos fisicos irreversíveis ao sistema. Falhas transientes, por outro lado, estão presentes ocasionalmente, e representam instabilidade temporária de *hardware*, ou de condições adversas. Infelizmente, não há condição segura de se detectar as falhas transientes, já que elas podem desaparecer quando os testes são aplicados. Estas falhas não serão tratadas neste trabalho.

Um dos modelos de falha mais usados é o modelo *stuck-at,* sendo um dos modelos mais efetivos e poderosos em uso. Este modelo assume que as falhas afetam somente as interconexões das portas lógicas. Cada linha pode ser *stuck-at* 1 (s-a-1) ou *stuck-at* 0 (s-a-0), se a linha assumir sempre os valores 1 ou 0, respectivamente.

As falhas *stuck* não são as mais simples de se analisar, mas têm se mostrado muito efetivas na representação do comportamento faltoso dos dispositivos reais. A simplicidade das falhas *stuck* é derivado do seu comportamento lógico, por isso são também chamadas de falhas lógicas. Uma das primeiras discussões sobre falhas *stuck* foi feita por Poage [Poage63].

A Figura II.7 mostra um circuito com *stuck-at-1* na saída da porta G1. O efeito desta falha (neste circuito), e stuck-at-1 na saída da porta G3 não podem ser distinguidas na saída do circuito. Qualquer teste que detecte uma das falhas, irá detectar a outra. Tais falhas podem ser agrupadas, reduzindo o número total de falhas a serem detectadas.

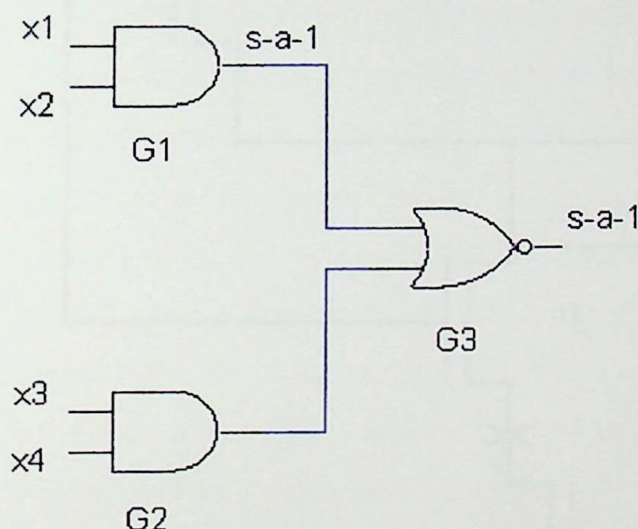


Figura II.7 - Circuito com falha *stuck-at*.

Uma falha *stuck-open* em uma lógica CMOS, é aquela que leva o sistema a um estado de "não-condução", ou estado de alto impedância, podendo em alguns casos reter o estado lógico anterior na saída do dispositivo. Como exemplo, considere a porta *NAND* mostrada na Figura II.8, com um circuito aberto na conexão do dreno do transistor n2. A condição de entrada x1x2 = 11, produz a inicialização de n2 e a instabilidade da célula de saída. A capacitância de carga irá armazenar o valor da saída, que será função da capacitância e da corrente de carga do circuito. O

circuito aberto produz o efeito de memória no circuito combinacional, além de impedir a comutação do valor da saída. Por esta razão que as falhas *stuck-open,* são conhecidas como falhas de "memória" [Soden 89]. Dessa forma, para as falhas *stuck-open,* a seqüência que os vetores de teste são aplicados é muito importante. Dois vetores são necessário para se detectar estas falhas [Craig 85]. O primeiro vetor, chamado de vetor de inicialização , inicia os estados do circuito, enquanto que o segundo testa a presença da falha.

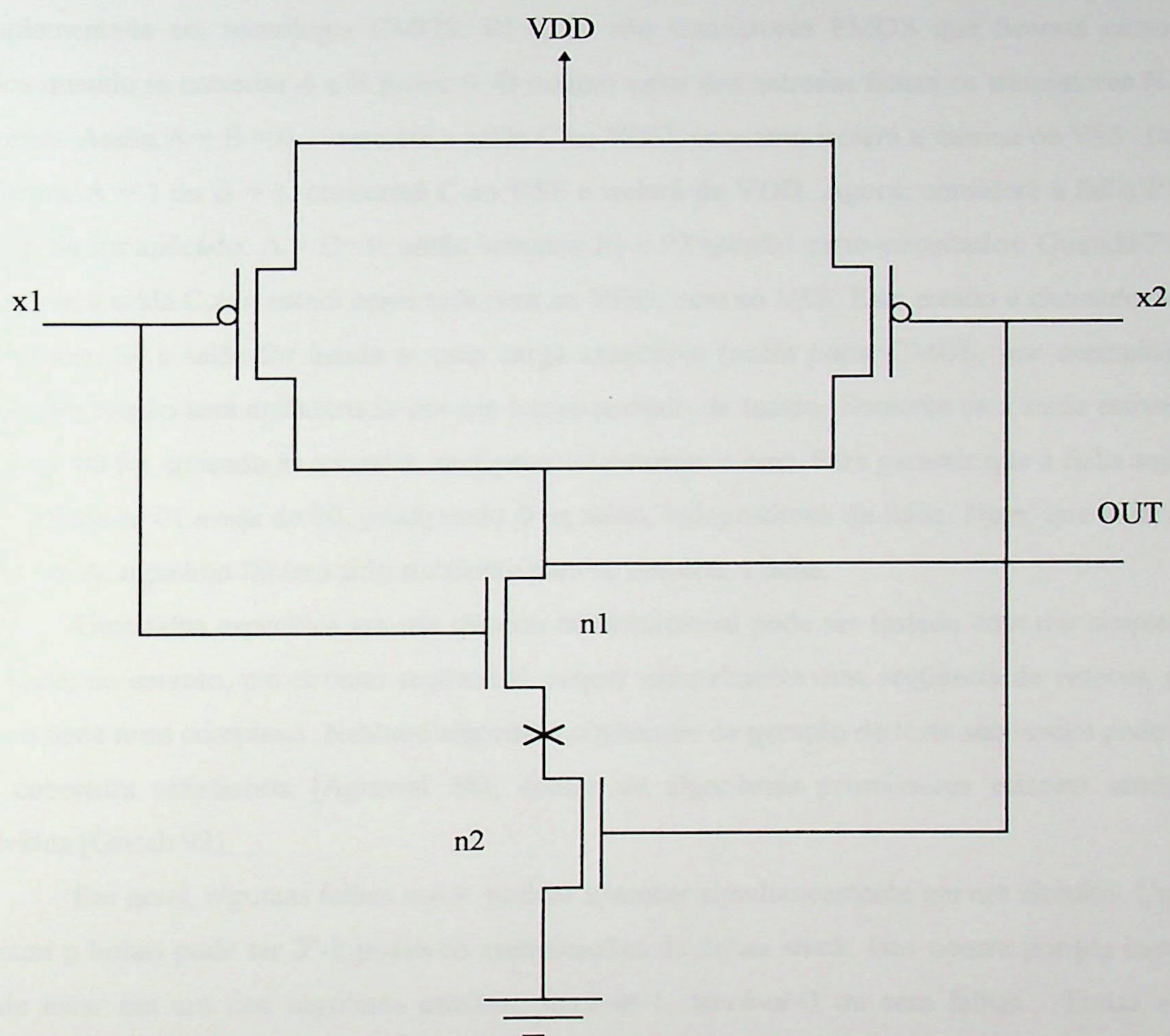


Figura II.8 - Porta *NAND* com falha *stuck-open*.

Os atuais dispositivos VLSI são compostos por transistores MOS, que se comportam como chaves ideais. Em 1978, Wadsack [Wads78] descobriu que circuitos digitais implementados com tecnologia CMOS poderiam apresentar modelos de falhas diferentes das falhas *stuck-at*. O pior neste caso é que os testes descritos para falhas *stuck-at*, poderiam *a priori*, ser ineficientes para detectar tais falhas, denominada como falhas não-clássicas.

Para ilustrar tal conceito considere os transistores MOS como chaves ideais, as falhas não-clássicas correspondem a um transistor em estado ***stuck-open***. A Figura II.9 mostra uma porta NOR implementada em tecnologia CMOS. P1 e P2 são transistores PMOS que ficaram curto-circuitados quando as entradas A e B forem 0. O mesmo valor das entradas fazem os transistores N1 e N2 abrirem. Assim A = B = 0, conectará a saída C ao VDD, enquanto isolará a mesma de VSS. Da mesma forma, A = 1 ou B = 1, conectará C ao VSS e isolará de VDD. Agora, considere a falha P1 *stuck-open*. Se for aplicado A = B= 0, então somente P1 e P2 estarão curto-circuitados. Quando P1 é *stuck-open*, a saída C não estará conectada nem ao VDD, nem ao VSS. Este estado é chamado de alta impedância. Se a saída for ligada a uma carga capacitiva (outra porta CMOS, por exemplo), então qualquer tensão será armazenada por um longo período de tempo. Somente se a saída estiver em 0 quando 00 for aplicado às entradas, será possível detectar o erro. Para garantir que a falha seja detectada, aplica-se 01 antes de 00, produzindo 0 na saída, independente da falha. Note, que para a falha s-a-1 em A, o padrão 00 terá sido suficiente para se detectar a falha.

Uma falha específica em um circuito combinacional pode ser testada com um simples vetor de teste, no entanto, um circuito seqüencial requer normalmente uma seqüência de vetores, o que torna o teste mais complexo. Nenhum algoritmo conhecido de geração de teste seqüencial pode oferecer cobertura satisfatória [Agrawal 88], apesar de algoritmos promissores estarem sendo desenvolvidos [Ghosh 92].

Em geral, algumas falhas *stuck* podem aparecer simultaneamente em um circuito. Um circuito com n linhas pode ter 3^n-1 possíveis combinações de linhas *stuck*. Isto ocorre porque cada linha pode estar em um dos seguintes estados: *stuck-at*-1, *stuck-at*-0 ou sem falhas. Todas as combinações exceto aquela em que todas estão sem falhas, são contadas como falhas. É fácil imaginar que mesmo com pequenos valores de n , o número das múltiplas falhas *stuck* será muito grande, embora, na prática analisa-se somente falhas *stuck* simples.

III.3.2 - Falhas Equivalentes

Duas falhas são ditas equivalentes se o seu efeito é indistinguível nas saídas do circuito, o que significa dizer que qualquer teste que detecte uma das falhas, poderá detectar a outra. Chamamos de falha *collapsing*, a seleção de uma falha representativa de cada classe de falhas equivalentes. Computacionalmente, é um problema não atrativo na sua forma geral. Um bom exemplo, são as falhas *collapsing* associadas com entradas e saídas de portas lógicas . Pode-se observar que uma entrada com *stuck*-at-0 (é prática comum escrever *stuck-at*-i como s-a-i) é equivalente à saída com s-a-0; não há maneira de se distinguir as duas, se incidiram na entrada ou na saída do circuito. Para a geração de teste, no entanto, consideremos n+2 das 2n+2 falhas associadas com uma porta NAND: s-a-1 de cada entrada e s-a-1 e s-a-0 da saída. Similarmente, em uma porta OR com n - entradas, testa-se s-a-0 em cada entrada e s-a-1 e s-a-0 na saída.

Os três valores usados na simulação são: 0, 1 e X ≡ (*don't care* - indiferente). Considere o circuito da Figura II.9. O valor 0 na linha A (independente das outras entradas da porta NAND) força 1 na linha E , e 1 na linha G . Pode-se facilmente verificar que, colocando a condição *don't care* nas outras entradas, que o valor das linhas E e G ainda são únicas. Dessa forma, as faltas A *stuck-at*-0, e H stuck-at-1, são equivalentes.

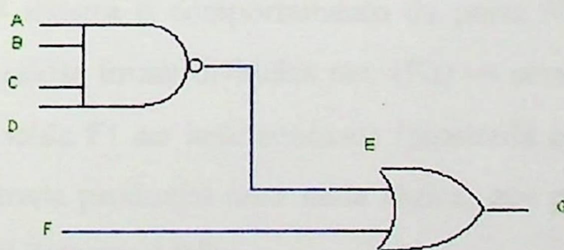


Figura II.9 - Exemplo de falha *collapsing*.

Considere duas falhas f1 e f2. Suponha que todos os testes para f1 também detectem f2, mas apenas alguns testes para f2 detectem f1. Então f1 é dita dominante sobre f2 [Poage63].

Falhas *collapsing* equivalentes e dominantes, podem ser usadas para reduzir o número de falhas que devem ser consideradas em uma determinada análise. Por exemplo, na porta NAND anterior há duas falhas *stuck* na linha de saída que poderiam ser classificadas como *collapsing*.

Em circuitos seqüenciais, falhas *collapsing* são geralmente acompanhadas de vários estágios. Outros detalhes sobre falhas *collapsing* podem ser encontrados em [McCI71] e [Sche72]. Em outro trabalho, um algoritmo para redução do conjunto de falhas foi mostrado por Goel [Goel73] e Cha [Cha79].

II.3.2.1 - Detecção de Falhas *Stuck* em Transistores

Quando se considera falhas em tecnologia MOS, pode-se notar que alguma dessas falhas não podem ser modeladas pelas falhas *stuck-at*. Como exemplo, considere a Figura II.10, onde a porta *NAND* - MOS tem duas falhas provocadas por curto-circuito (mostrado através das linhas tracejadas). Se o curto-circuito 1 está presente, então quando houver 111 nas entradas, a saída poderá não ser 0, mas sim uma tensão indeterminada representada como /. Assim, através desta falha e das combinações das entradas, a saída não será permanentemente *stuck-at*-0 ou *stuck-at*-1. Este exemplo pode conduzir a idéia de fracasso do modelo de falha *stuck-at*. No entanto, se considera-se que os testes gerados para falhas do tipo *stuck-at*, irão detectar as falhas em transistores, atestando então que este modelo apresenta aplicabilidade prática [Abraham 84].

A Tabela II.3 mostra o comportamento da porta *NAND* nMOS quando o teste de falhas *stuck* é aplicado. As saídas foram divididas em: (Fo) → sem falhas e F1 e F2 → com falhas (curto-circuito). Apesar da saída F1 ser indeterminada (mostrada como /), pode-se observar que a próxima combinação de entrada produzirá uma saída lógica, que por sua vez é diferente da saída correta, sendo assim possível detectar a falha.

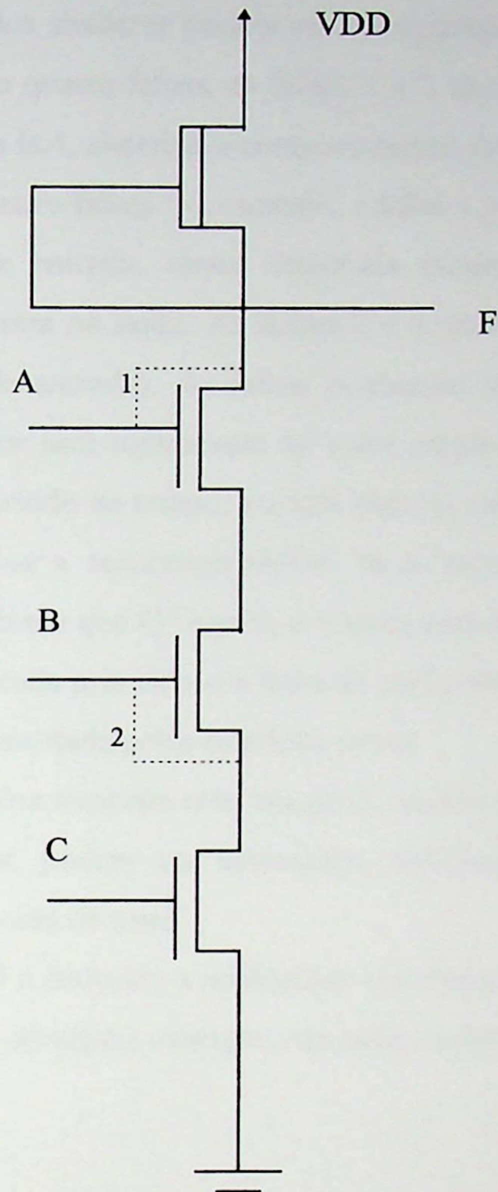


Figura II.10 - Porta *NAND* nMOS com Duas Falhas.

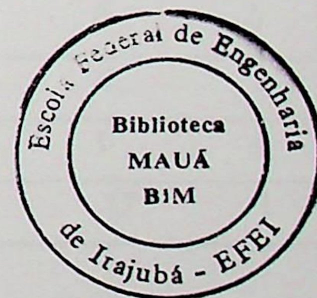
ENTRADAS			SAÍDAS		
A	B	C	Fo	F1	F2
1	1	1	0	/	1
0	1	1	1	0	1
1	0	1	1	1	1
1	1	0	1	1	1

Tabela II.3 - Testes para a porta NAND nMOS.

Resultados similares podem ser conseguidos com a porta *NOR* nMOS. A Figura II.11 mostra tal porta com quatro falhas, as falhas 1 e 2 são curtos-circuitos, e as falhas 3 e 4 são circuitos abertos. A Tabela II.4, sintetiza o comportamento da porta quando os testes de falhas *stuck* são aplicados com estas quatro falhas. Novamente, a falha 1 produz uma saída indeterminada para as primeiras combinações de entrada, sendo detectada eventualmente (pelo último teste) quando produzir uma lógica incorreta na saída. As falhas 2 e 4 podem ser vistas como equivalentes (pelo menos para o conjunto de entrada). As falhas produzem uma alta impedância na saída com a combinação 000. Este valor será equivalente ao valor anterior da saída (mostrado como Q"), pelo menos por um pequeno período de tempo, até que alguma carga residual possa alterar a saída. Este fato se justifica por se utilizar a tecnologia nMOS. Se os testes forem aplicados na ordem mostrada, esta falha será detectada, desde que Q" seja 0, e a saída correta seja 1. É interessante notar que se a combinação 000 fosse aplicada primeiro, e a linha de saída estivesse com o valor 1 armazenado, esta falha particular não seria detectada pelos referidos testes.

Estes exemplos mostram que, em geral, muitas falhas que não podem ser modeladas a partir do modelo *stuck-at*, podem ser detectadas, aplicando-se ao circuito MOS determinado conjunto específico de vetores de teste.

O capítulo III é dedicado a análise dos resultados com os modelos de falhas *stuck-open* e *stuck-at*, suas relações, e principais vantagens de cada modelo.



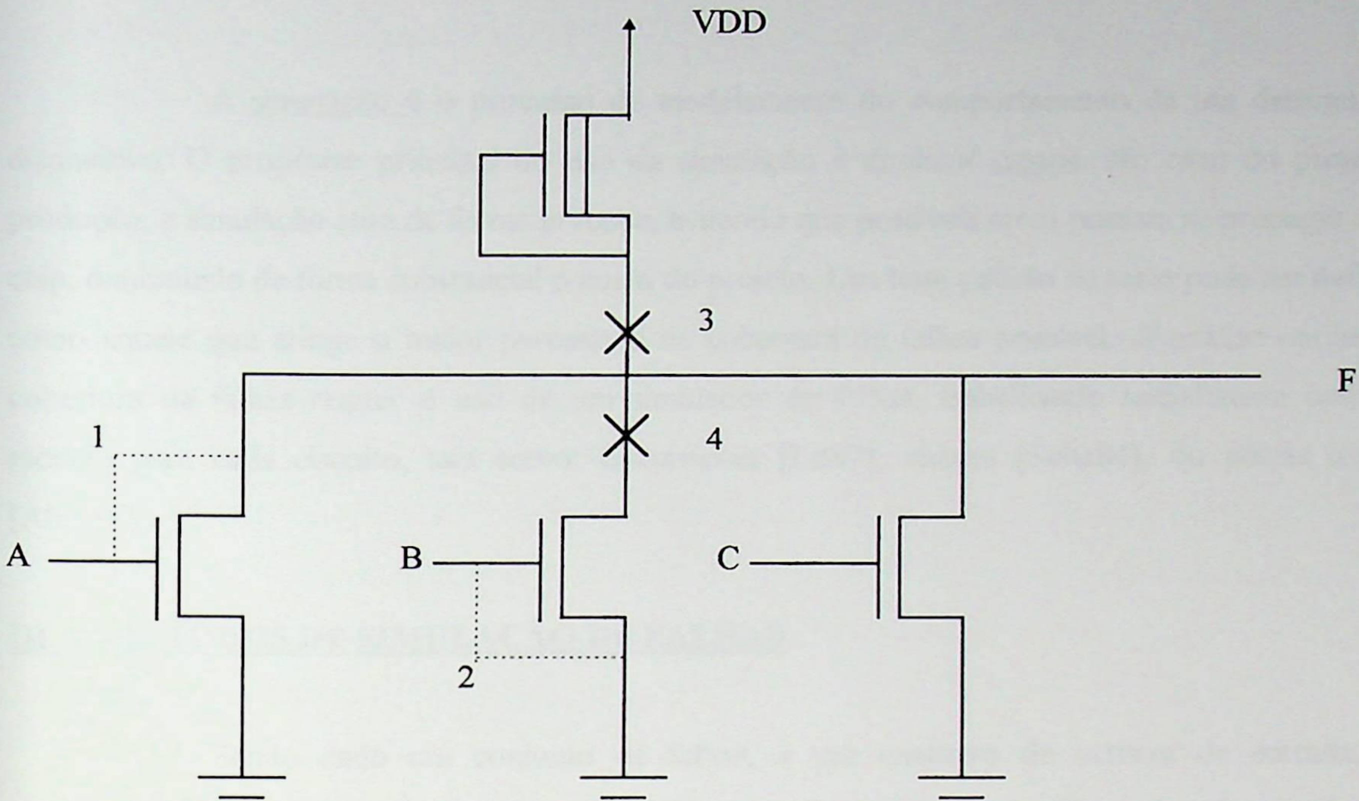


Figura II.11 - Porta NOR nMOS com quatro falhas.

ENTRADAS			SAÍDAS				
A	B	C	Fo	F1	F2	F3	F4
1	0	0	0	/	0	0	0
0	1	0	0	0	1	0	1
0	0	1	0	0	0	0	0
0	0	0	1	0	1	Q”	1

Tabela II.4 - Testes para a porta NOR nMOS.

III - AVALIAÇÃO DOS TESTES

A simulação é o processo de modelamento do comportamento de um determinado dispositivo. O propósito principal do uso da simulação é diminuir custos. No caso do projeto e produção, a simulação atua de forma precoce, evitando que possíveis erros possam se propagar até o chip, diminuindo de forma substancial o custo do projeto. Um bom padrão de teste pode ser definido como aquele que atinge o maior percentual de cobertura de falhas possível. A análise correta da cobertura de falhas requer o uso de um simulador de falhas, trabalhando inicialmente com um modelo para cada circuito, tais como: transistores [Lo87], chaves [Scha84], ou portas lógicas [Abra86].

III.1 - MÉTODOS DE SIMULAÇÃO DE FALHAS

Sendo dado um conjunto de falhas, e um conjunto de vetores de entrada, um simulador deve descobrir quais falhas são detectadas por tais vetores. Em todos os simuladores comerciais é feito um esforço para reduzir o tempo computacional necessário na detecção de falhas, usando para tal a simulação de mais de uma falha em um dado passo, para um determinado vetor de teste. Esta técnica e conhecida como simulação paralela [Thom75], em que uma palavra de w *bits* é associada com cada linha do circuito em análise. Durante a passagem através do simulador, cada *bit* em uma posição particular seria associado com o circuito que tem uma falha específica. Depois da simulação, o *bit* armazena o valor na linha, associada com o circuito. Antes do começo de um passo, um conjunto de (w - 1) palavras não simuladas é escolhido. O último *bit* é usado para simular o circuito sem falhas. Portas lógicas nas quais as entradas/saídas são diretamente afetadas por uma determinada falha são sinalizadas. Quando uma porta sinalizada é simulada, o efeito da falha é sinalizado através de *bits* apropriados de uma palavra, representando suas entradas e saídas. No final do teste, a palavra de cada saída primária pode ser examinada para se determinar qual das falhas simuladas seria detectada naquela saída. Isto é feito através da comparação da resposta de cada circuito sob faltas e do mesmo livre das mesmas. Através de operações lógicas em computadores, as diferentes falhas (w - 1) são simuladas em paralelo em cada passo deste processo, dando assim o nome a tal método. Um total de F falhas seria simulada em F/(w - 1) passos.

O simulador dedutivo resolve o problema do número de passos de simulação [Arms72]. É necessário somente um passo para que se realize a simulação, independente do número de falhas simuladas. A idéia básica, é associar com cada linha uma lista falhas , através da simulação do vetor de entrada. A simulação da porta lógica requer a dedução da lista de falhas da saída, através da lista de falhas da entrada. Esta falha será exemplificada por uma porta AND, mostrada na Figura III.1. Assuma que esta porta esteja inserida no circuito sob teste. A lista de falha associada com as entradas *a* e *b* são mostradas, tais falhas já deveriam ter sido armazenadas em algum passo do algoritmo. Os valores apresentados do sinal, são para circuitos sem falhas. Considere o efeito da falha f_1, aparecendo em L_a, mas não em L_b, na saída da porta. O valor do sinal na linha *a* mudará de 1 para 0, no entanto, na linha *b* o valor ficará inalterado. Dessa forma, a saída ficará em 0, de tal forma que f_1 não pode estar na lista de falha L_c. Novamente, nenhuma falha em L_a pode ocorrer em L_c, desde que a saída não mude em função da falha. Tal falha mudará o valor nas linhas *b* e *c* e, em conseqüência, deverá ser incluída em L_c.

Assim, a falha "c *stuck at* 1" (ou c_1) deve estar na lista de falha da saída. Dessa forma, obtêm-se a expressão para a lista de falha da saída, mostrado na figura.

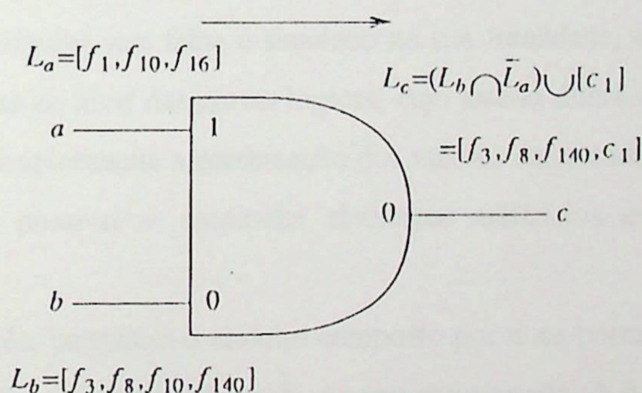


Figura III.1 - Lista de falhas na simulação dedutiva.

Comparado com a simulação de falha paralela, as penalidades pagas para que se tenha somente um passo na simulação dedutiva, são:

1) variação dinâmica de armazenagem para as listas de falhas associadas com cada linha;
2) processamento complexo das portas, requerendo um conjunto de instruções nas listas de falhas das linhas.

Nota-se que o processamento das listas de falhas da saída é essencialmente dinâmico. A expressão para a lista de saída, não é apenas uma função do tipo de porta, dependendo também do valor do sinal das portas de entrada. A lista de falha da saída pode mudar mesmo que a entrada e a saída se mantenham inalteradas. Isto acontece porque a lista de falha da linha pode mudar, mesmo quando seu valor não mudar. Por exemplo, quando um valor na linha *a* muda para 0, o valor da linha *c* não é afetado, porém L_c poderá ser alterado. Este "evento da lista de falha" pode ser propagado através de todas as portas, para o qual a linha *c* é uma entrada. Esta característica indesejada do método dedutivo, pode ser resolvido através do método de simulação concorrente [Ulri74].

A idéia básica sobre a simulação de falhas concorrente é bastante simples. Tipicamente, a falha altera o valor de pequenos sinais no circuito como um todo. Assim a maioria, senão todas as informações para a simulação de uma falha, estão contidas no circuito sem falhas. Na simulação concorrente, o circuito sem falha é simulado na sua totalidade, enquanto que o circuito com falha é simulado apenas ao nível das portas lógicas, cujo estado difere do estado perfeito. Para portas lógicas, o estado é simplesmente a combinação dos valores da entrada e da saída, no entanto, no método concorrente é possível se manipular elementos arbitrários com estado previamente armazenados.

Como exemplo, considere o circuito composto por duas portas na Figura III.2.a, com os sinais 1, 0, e 1 sendo aplicados às entradas a, b, e c respectivamente. A figura mostra o valor real da simulação para tais vetores, bem como ligado a cada porta, a lista das falhas, nas quais os estados diferem do valor correto. Por exemplo, quando a falha "b *stuck at* 1" (b_1 na figura) ocorre, altera os estados de ambas as portas, aparecendo dessa forma nas listas. A falha "a *stuck at* 0" (mostrado como a_0) não causa alteração no estado da porta G2, não aparecendo dessa forma na lista de falhas dessa porta.

Suponha que a entrada tenha mudado de 1 para 0 (Figura III.4.b). Desde que a entrada G1 tenha sido alterada, então deverá ser realizado outro processo de simulação, com novos valores. Os resultados dos valores reais e das portas com falhas são ilustrados na figura. A nova lista para G1 foi obtida da lista de falhas original, seguindo as seguintes etapas:

1) A mudança na entrada é refletida no circuito sem falha, tanto quanto as portas com falhas de G1, o novo estado de cada porta é determinado através de nova simulação.
2) Se existirem algumas portas com falhas, apresentam o estado inicial idêntico ao correto, então apaga-se tais falhas da lista original.
3) Se forem encontradas novas portas com falhas, nas quais os estados poderiam ser diferentes dos estados sem falhas, tais falhas serão anexadas à lista.
4) Finalmente, se o estado da porta de saída sem falha, mudar como resultado desses passos, em "evento correto" será incluído nas portas alimentadas pela saída G1. Similarmente, se o estado da porta de saída não mudar, um "evento incorreto" será incluído nas portas alimentadas pela saída de G1. No exemplo, nenhum "evento correto" será armazenado, pois a saída do circuito sem falha não foi alterado. Entretanto, para a falha b_1 a saída muda de 1 para 0, então o evento de falha correspondente será armazenado em G2.

O evento de falha em G2 será então processado. Tal processo envolve a procura por uma entrada para este evento na lista de G2, alterando-o apropriadamente, e simulando a porta com o novo valor. Neste caso, após a simulação, os estados com falha coincidem com os estados sem falhas, dessa forma, a entrada para b_1 é apagada da lista de G2. Note que, a simulação não causou uma mudança na saída da porta com falha, não sendo necessário armazenar outros eventos.

O termo concorrente é derivado do fato de que cada porta com falha, carrega informação para a simulação independente da falha associada. A habilidade da simulação concorrente de avaliar independentemente portas com falha, a torna o modelo para implementações nos aceleradores de *hardware* [Blan84].

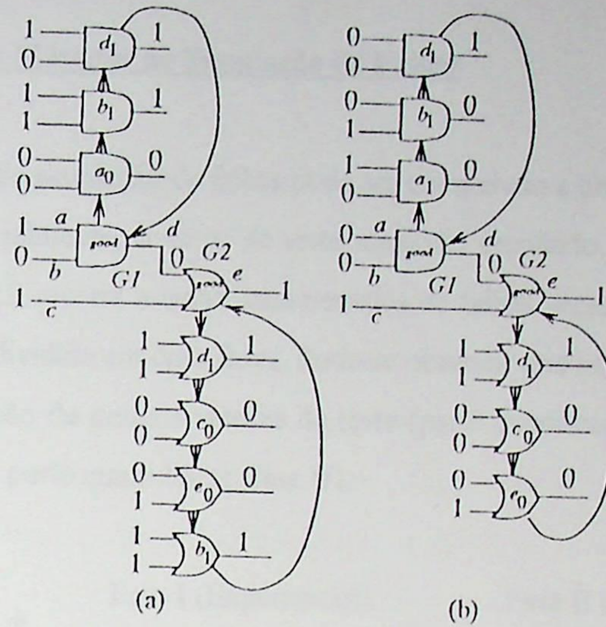


Figura III.2 - Exemplo de simulação de falha concorrente.

As simulações paralela, dedutiva e concorrente pode ser usada tanto em circuitos combinacionais, quanto em circuitos seqüenciais.

O simulador de falhas de uma rede digital, nada mais é do que o modelamento do comportamento da rede na presença de falhas que possam causar defeitos físicos, ou influenciar o bom funcionamento dos circuitos. Do ponto de vista dos produtos LSI/VLSI, a simulação se tornou extremamente necessária, simplesmente porque não pode haver nenhuma falha no dispositivo a ser fabricado.

A geração dos testes para a detecção da presença de falhas precisa ser avaliada em função da sua efetividade. Esta avaliação pode ser feita pelos simuladores de falha, o qual depende do correto modelamento das falhas. Dessa forma, o modelamento de falhas estabelece o básico para os simuladores de falhas e geração de testes.

III.1.1 - Comparação dos Métodos de Simulação de Falhas

O processo de simulação de falhas pode ser comparado a um tiro em um pequeno alvo que vai se afunilando. Inicialmente, o vetor de teste aleatório escolhido, provavelmente detectará muitas falhas. A Figura III.3 mostra a curva característica de falhas versus o número de vetores de teste. Esta curva pode ser dividida em duas fases. Pode-se observar que na fase I a maioria das falhas são detectadas com aplicação de poucos vetores de teste (parte exponencial). Existe uma transição gradual da fase I, para uma parte quase linear (fase II).

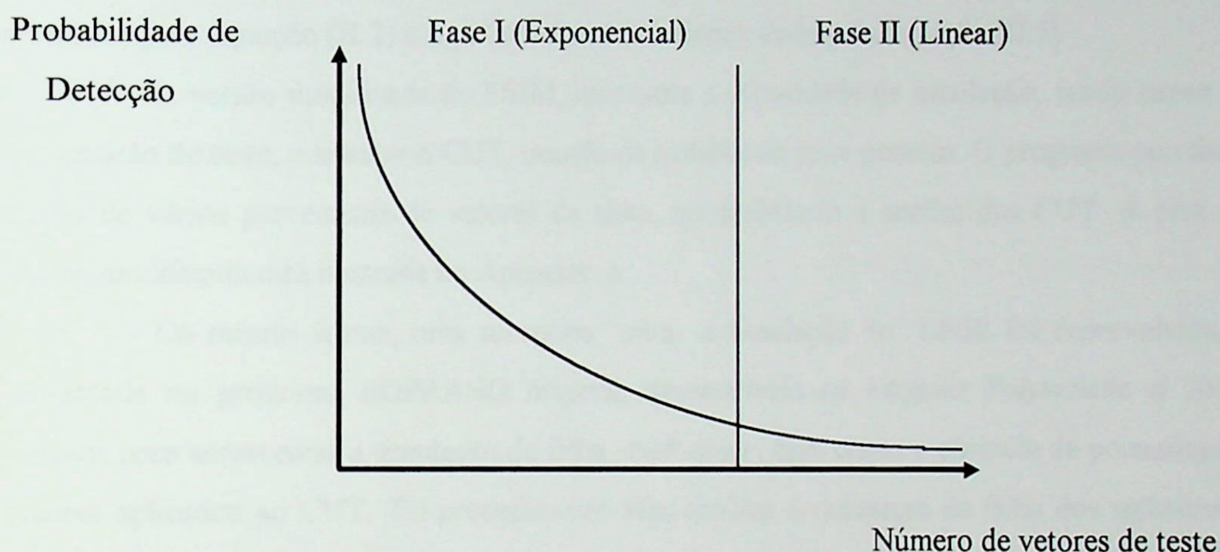


Figura III.3 - Probabilidade de detecção versus vetores de teste.

Tomando a curva acima como exemplo, foi estimado que o custo de uma rede com G portas cresce proporcionalmente para $\underline{G^3}$ e $\underline{G^2}$ quando são utilizadas simulações paralelas e dedutivas, respectivamente [Goel 80]. Um simulador de falha concorrente a nível de transistores leva cerca de 1527 segundos (aproximadamente 25 minutos), para simular um circuito com 9478 transistores [Lo87]. Um conjunto de 2241 vetores foi usado nesta simulação. Com certeza a simulação de grandes circuitos não é uma proposição prática, mesmo no caso de circuitos com pequenas densidades. Uma alternativa atrativa é o uso de técnicas de amostragem estatística.

III.2 - MÉTODOS DE SIMULAÇÃO

Uma subrotina para a simulação de LFSR foi desenvolvida (em linguagem C), e implementada no programa FSIM original , desenvolvido na *Virginia Polytechnic & State University*. O programa original é capaz de ler padrões de teste de um arquivo de entrada. As modificações visam controlar também a percentagem de vetores aplicados a cada circuito ISCAS85

A subrotina implementada usa o modelo de LFSR dado na Figura II.4 (capítulo II), representado pela equação (II.2) e o polinômio característico dado pela equação (II.5).

A versão modificada do FSIM, incorpora a capacidade de simulação, sendo capaz de gerar o padrão de teste, e simular o CUT, usando os padrões de teste gerados. O programa permite a aplicação de vários percentuais de vetores de teste, possibilitando a análise dos CUT. A lista do programa modificada está ilustrada no Apêndice A .

Da mesma forma, uma subrotina para a simulação do LFSR foi desenvolvida, e implementada no programa SOPRANO original, desenvolvida na *Virginia Polytechnic & State University*, para acrescentar a simulação de falha *stuck-open* , bem como o controle da porcentagem de vetores aplicados ao CUT. Tal procedimento visa analisar a cobertura de falha dos *softwares* , podendo-se avaliar o desempenho e a confiabilidade, além de se procurar o percentual de correlação dos mesmos.

Em seguida será mostrado o procedimento de avaliação utilizado. O procedimento é dividido em 4 passos. O primeiro passo é a instalação dos parâmetros do LFSR. Neste passo, os parâmetros são armazenados. No segundo passo, os padrões de teste que serão aplicados ao (n,w) CUT é gerado pelo (n,k)LFSR. No terceiro passo, os pares de teste são aplicados ao circuito e o circuito é simulado, assumindo-se que o atraso de propagação das portas lógicas seja zero. O percentual de vetores de teste pode ser controlado, propiciando a análise da cobertura de falha com diferentes porções de vetores. O procedimento é descrito abaixo:

Passo 1: Carregamento dos parâmetros do circuito LFSR.

Determinação do tamanho máximo de w.

Determinação do grau do polinômio primitivo p(x), k.

Determinação do grau do polinômio g(x), n-k.

Passo 2: (Criação do circuito LFSR).

Criação do circuito LFSR f(x) = p(x).g(x).

Carregamento do valor inicial (semente) do LFSR.

Carregamento dos ciclos de *clock*.

Carregamento do tempo inicial.

Passo 3: (Este procedimento simula o circuito assumindo que as portas lógicas não tenham atraso de propagação).

Se as falhas forem detectadas, ir para o passo 4.

Se o ciclo de *clock* for excedido, ir para o passo 4.

Entrar com o percentual de vetores de teste.

Se os padrões de teste forem > *max_test_pattern*, então parar.

Realizar a simulação de falha *stuck-open* para o circuito.

Passo 4: (Este procedimento checa a condição de parada).

Se a cobertura de falha desejada, usando o número especificado de ciclos de *clock* for alcançada, parar.

Selecionar novos valores para p(x), g(x), ou ciclo de *clock* , ir para o passo 2.

A Figura III.4, mostra a característica dos *softwares* de simulação.

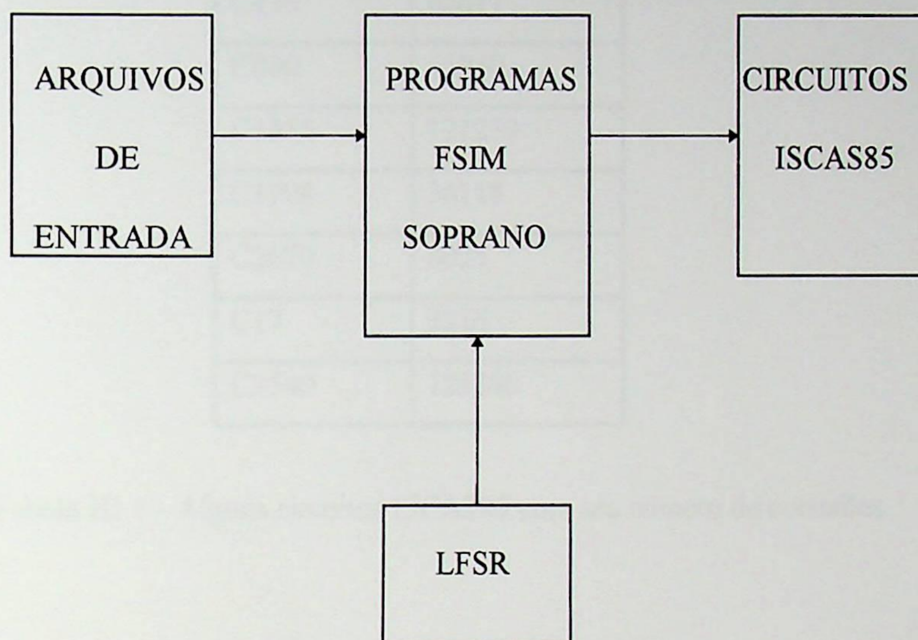


Figura III.4 - Estrutura Básica dos simuladores.

Os *softwares* FSIM e SOPRANO foram modificados e utilizados na simulação de circuitos padrões (ISCAS85) [Brglez85]. Tais circuitos foram utilizados na avaliação do desempenho das falhas *stuck-open* e *stuck-at*, pois necessita-se de um mesmo padrão desenvolver tal avaliação, garantindo a confiabilidade dos dados gerados pelos programas. A Tabela III.1 mostra o número de conexões de alguns circuitos ISCAS85, e a Figura III.5 apresenta o circuito C17.isc.

CIRCUITO	CONEXÕES
C432	4730
C499	47011
C880	13940
C1355	121259
C1908	36118
C2670	8025
C17	3216
C3540	125886

Tabela III.1 - Alguns circuitos ISCAS85 com seu número de conexões.

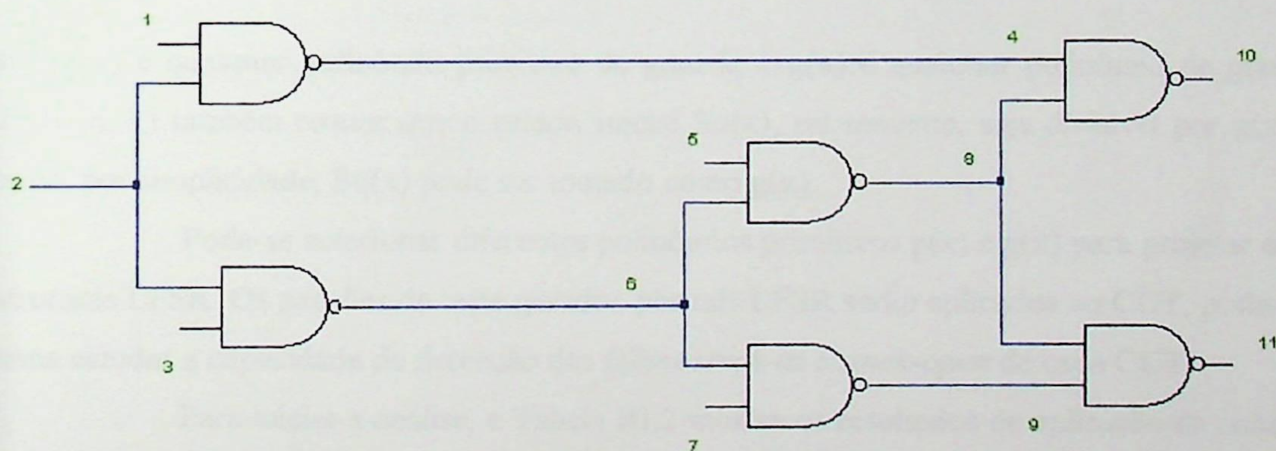


Figura III.5 - Circuito c17.isc.

III.2.1 - Verificação dos Testes Baseado em Códigos Lineares

Baseado no comportamento dos LFSR, apresentado no capítulo II, considere que (n, w) seja um circuito com n entradas, onde w é o número máximo de entradas que uma saída pode depender. Para testar um CUT com (n, w), um LFSR de n-estágios e de período $T = 2^k - 1$ é necessário, de tal forma que, qualquer combinação de w possa conter todos os 2^w padrões de teste distintos. O LFSR (n, k) deve ter o menor inteiro k ($w \leq k \leq n$) dado por:

$$W \leq \left\lfloor \frac{k}{n - k + 1} \right\rfloor + \left\lfloor \frac{k}{n - k + 1} \right\rfloor$$

O polinômio característico de um LFSR (n, k) é dado por:

$$f(x) = g(x) \cdot p(x)$$

onde p(x) é qualquer polinômio primitivo de grau k, e g(x) é qualquer polinômio de grau n-k. O LFSR (n, k) também requer que o estado inicial So(x), ou semente, seja divisível por g(x). Dessa forma, por simplicidade, So(x) pode ser tomado como g(x).

Pode-se selecionar diferentes polinômios primitivos p(x) e g(x) para projetar diferentes estruturas LFSR. Os padrões de teste gerados por tais LFSR serão aplicados ao CUT, pode-se dessa forma estudar a capacidade de detecção das falhas *stuck-at* e *stuck-open* de cada CUT.

Para iniciar a análise, a Tabela III.2 mostra os resultados de aplicação da cobertura de falhas *stuck -at* para alguns circuitos ISCAS85. O Apêndice B mostra o exemplo de alguns vetores de teste utilizados neste trabalho.

Circuito	n	f (x)	So (x)	cobertura (%)
c355.isc	41	$X^{41} + X^3 + 1$	X^4+X^2+1	79.24
c1908.isc	33	$X^{33} +X^{13} + 1$	X^4+X^2+1	63.79
c2670.isc	157	$X^{157}+ X^{27} +$ $X^{26} + X + 1$	$X^{10}+X^9+X^8+X^7+X^6+X^5+X^4+X^3+$ X^2+X+1	54.45
c499.isc	41	$X^{41} + X^3 + 1$	X^4+X^2+1	89.46

Tabela III.2 - Cobertura de falhas para circuitos ISCAS85.

As Tabelas III.3 e III.4 mostram os resultados da simulação do circuito c17.isc em relação as falhas *stuck-open,* que mostra o percentual de cobertura das falhas, bem como o tempo necessário para a execução do teste, pode-se verificar que a escolha adequada do polinômio, pode ser o decisivo no que diz respeito ao esforço computacional necessário para a cobertura das falhas, refletindo no custo final do teste . A Tabela III.5, mostra os LFSR's com melhor cobertura de falha para circuitos ISCAS85. A cobertura de falha é comparada com os padrões de teste gerados pelo SOPRANO. Pode-se notar que o *software* reduz o número de pares de vetores de teste necessários para a cobertura de falhas. A condição de partida para as falhas *stuck-open* (semente - capítulo II) é mostrada na Tabela III.6, onde encontramos o exemplo de vetores de inicialização. Pode-se observar que quando mais circuitos LFSR forem simulados, melhores serão os resultados, e quanto maior for o número de ciclos de *clock* aplicados, maior será a cobertura de falhas.

g (x)	p (x) = $x^4 + x + 1$	
	cobertura de falhas (%)	Tempo CPU (segundos)
g (x) = x	88.889	0.213
g (x) = x + 1	88.889	0.183

Tabela III.3 - Resultados experimentais para o circuito c17.isc.

g (x) = So	f (x) = $x^5 + x^3 + x^2 + x + 1$	
	cobertura de falhas (%)	Tempo CPU (segundos)
g (x) = 1	100.00	0.083

Tabela III.4 - Resultados experimentais par o circuito c17.isc.

Nome do circuito	n	Software	Soprano	LFSR (*Hardware*)
		Número de vetores de teste	Cobertura de falhas (%)	Cobertura de falhas (%)
c17.isc	5	13	100	100
c432.isc	36	144	96.12	98.78
c880.isc	60	202	100	97.03
c1355.isc	41	337	98.38	97.01
c1908.isc	33	401	99.52	95.12

Tabela III.5 - Resultados experimentais para diversos circuitos ISCAS85.

Falhas	f1 (a, b, c)		f2 (d, e, f)		f3, f4 (g, h, i, j)		f5, f6 (k, l)	
Vetor de inicialização (T1)	x1,	x2	x1,	x2	x1,	x2	x1,	x2
	1	1	1	1	0	0	1	1
					0	1		
					1	0		
Vetor de Teste (T2)	0	1	1	0	1	1	0	0
							0	1
							1	0

Tabela III.6 - Exemplo de vetores de inicialização para falhas *stuck-open*.

III.3 - ANÁLISE PROBABILÍSTICA

Seja Pn a probabilidade de ocorrência de uma determinada falha n em um dado *chip*. A probabilidade de não ocorrência desta falha pode ser representado por Y [Willians 86], então:

$$Y = (1-Pn)^n \qquad \text{(III.1)}$$

Seja A o caso de ocorrência de nenhuma falha *stuck-open* ou *stuck-at* no *chip*. Seja B o caso em que não exista falha em uma determinada porção m dos testes. Assim:

$$P(B) = (1-Pn)^m \qquad \text{(III.2)}$$

Para se determinar a probabilidade de que A aconteça, sabendo-se que os testes das m falhas foram realizados com sucesso, pode-se usar a seguinte relação:

$$P(A/B) = \frac{P(A \cap B)}{P(B)} \qquad \text{(III.3)}$$

A probabilidade de A/B, é a probabilidade de não se encontrar uma falha na porção m do conjunto dos vetores de teste. Dessa forma, temos:

$$P(A \cap B) = (1-Pn)^n \qquad \text{(III.4)}$$

A probabilidade que um *chip* defeituoso seja encontrado é 1 menos a probabilidade que o *chip* seja bom. Seja ND o nível de defeito, então a equação para ND será:

$$ND = 1 - P(A/B)$$
$$= 1 - \frac{P(A \cap B)}{P(B)}$$
$$= 1 - \frac{(1-Pn)^n}{(1-Pn)^m}$$
$$= 1 - (1-Pn)^{n-m} \qquad \text{(III.5)}$$

Substituindo o valor de Y da equação (III.1) em (III.5), tem-se:

$$ND = 1 - Y^{(n-m)/n} \quad = \quad 1 - Y^{1-(m/n)} \qquad \text{(III.6)}$$

Se T representa a cobertura de falhas para um dado conjunto de vetores (m/n) testes, então o nível de defeito é dado por:

$$ND = 1 - Y^{(1-T)} \qquad \text{(III.7)}$$

Esta equação é mostrada na Figura III.6, onde pode-se encontrar vários valores de Y. Se Y for 0.25, e não foi feito absolutamente nenhum teste, tem-se por definição uma média de 25% de dispositivos bons e 75% de dispositivos ruins.

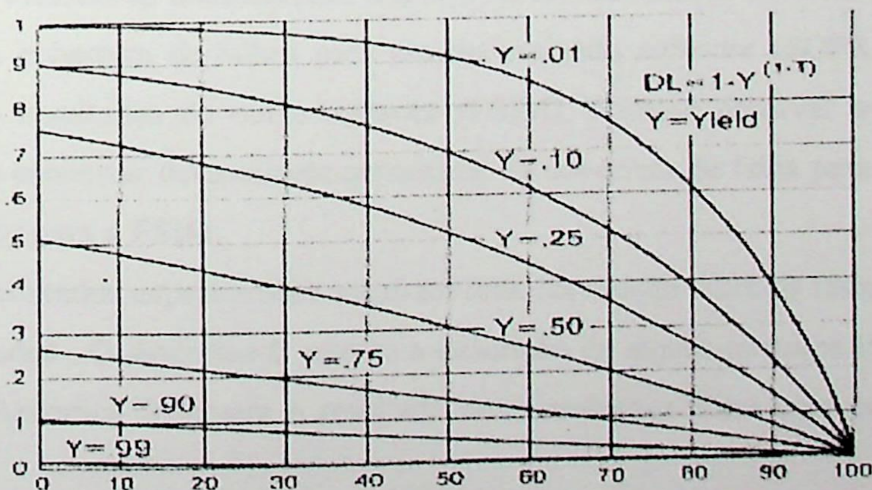


Figura III.6 - Nível de defeito em função da cobertura de falha.

Para melhorar o nível de defeito, trazê-lo a um patamar cada vez mais baixo, deve-se ter uma cobertura de falha que seja a maior possível. Portanto, aplica-se os conceitos acima aos circuitos ISCAS85, quando for encontrado o maior valor de cobertura de falha *stuck-open*, pode-se estimar o percentual de cobertura para as falhas *stuck-at*, e vice-versa, otimizando o processo de teste, trazendo maior versatilidade, com conseqüente economia de tempo e dinheiro.

III.4 - ANÁLISE ESTATÍSTICA

Para grandes circuitos, a repetição da simulação de falhas para diferentes valores de f (x) pode ser um processo demorado. Para esses casos, ao invés da simulação do CUT para todas as falhas, o processo de simulação de falhas pode ser modificado para que somente um determinado percentual **c** do total de falhas possa ser considerado.

Foi mostrado (experimentalmente) por [Bardell87] e [Agrawal88], que o número de vetores de teste tende a ser linear, dependendo do número de falhas. Cada f (x) tem uma relação de dependência diferente para cada falha a ser simulada.

As modificações feitas nos programas originais da *Virginia Polytechnic & State University*, permite controlar os vetores de teste de tal sorte que se possa ter diferentes percentuais de cobertura para as falhas *stuck-open* e *stuck-at* (FSIM X SOPRANO). Dessa forma pode-se comparar os resultados experimentais encontrados para cada modelo. As Tabelas III.8, III.9, III.10, III.11, mostram os resultados da cobertura de falhas para o circuito c2670.isc. Para tais simulações foram utilizadas as mesmas as modificações nos *softwares*. Os vetores de teste foram selecionados de tal sorte que, a cobertura de falhas para um determinado *software* (SOPRANO) pudesse ser comparada com os resultados do outro *software* (FSIM). Pode-se observar que a simulação foi direcionada para se encontrar determinado percentual de cobertura de falha para o SOPRANO, e o correspondente valor para o FSIM.

Os resultados experimentais mostram uma correlação entre os testes realizados entre os *softwares* analisados . O Apêndice C mostra a descrição de alguns circuitos ISCAS85, utilizados neste trabalho. O Apêndice D mostra o resultado das simulações feitas com os diversos circuitos ISCAS85.

Cobertura de Falhas (%)	
SOPRANO	FSIM
25.66	52.08
28.01	54.21
24.90	51.14
28.02	53.95
25.66	52.63
26.79	52.91
24.46	49.72
27.04	52.63
29.03	55.75
27.77	54.01

Tabela III.8 - Aplicação de 1% dos vetores de teste ao c2670.isc.

Cobertura de Falhas (%)	
SOPRANO	FSIM
50.93	75.71
49.40	74.62
49.37	73.24
49.06	74.55
52.39	76.19
51.36	75.99
52.70	75.86
50.93	76.52
51.72	75.57
49.92	75.36

Tabela III.9 - Aplicação de 3.5% dos vetores de teste ao c2670.isc.

Cobertura de Falhas (%)	
SOPRANO	FSIM
75.83	88.78
76.96	89.44
78.28	90.73
74.73	87.73
76.17	88.46
74.12	87.44
78.80	90.95
75.95	89.73
78.15	90.93
77.05	88.53

Tabela III.10 - Aplicação de 15% dos vetores de testes ao c2670.isc.

Cobertura de Falhas (%)	
SOPRANO	FSIM
94.24	95.74
94.24	95.74
94.24	95.74
94.24	95.74
94.24	95.74
94.24	95.74
94.24	95.74
94.24	95.74
94.24	95.74

Tabela III.11 - Aplicação de 100% dos vetores de teste ao c2670.isc.

Utilizando os resultados experimentais para o circuito c2670isc, pode-se calcular o fator de correlação entre os modelos SOPRANO e FSIM.

O fator de correlação r [Baten 38] é definido como:

$$r = \frac{\sum (x-x')*(y-y')}{\left(\sum (x-x')^2 * \sum (y-y')^2 \right)^{1/2}} \qquad (III.1)$$

onde x' representa a média de todos os elementos da coluna x. Pode-se encontrar $-1 \leq r \leq 1$, onde 1 representa a relação linear positiva, e -1 representa a relação negativa. Na relação positiva os valores de um conjunto de dados seguem uma linha de projeção, baseado em outro conjunto de dados. A mesma análise é verdadeira para a relação negativa, na qual os valores seguiram uma linha de projeção negativa. Se r é zero não existe nenhuma relação entre os dois conjuntos de dados.

Os cálculos dos valores de r estão mostrados na Tabela III.12. Para se realizar estes cálculos utilizamos a Tabela III.8.

Circuito c.2670.isc					
SOPRANO	FSIM	(x-x')	$(x-x')^2$	(y-y')	$(y-y')^2$
25.66	52.08	-0.74	0.54	-0.62	0.38
28.01	54.21	1.61	2.59	1.51	2.28
24.90	51.14	-1.49	2.24	-1.55	2.41
28.02	53.95	0.62	0.38	1.25	1.56
25.61	52.63	-1.88	3.55	-0.06	0.003
26.79	52.91	0.39	0.15	-1.22	1.50
24.46	49.72	-1.93	3.74	-2.97	8.83
27.04	52.60	0.64	0.40	-0.06	0.003
29.03	55.73	1.37	1.87	1.577	2.48
27.77	54.01	1.63	2.65	3.05	9.32
x'= 26.4	y'= 52.7	--------	--------	--------	--------

Tabela III.12 - Valores para cálculo do coeficiente de correlação r.

Levando os valores da Tabela III.12 na relação III.1, temos:

$$r = \frac{19{,}54}{(18.13*28.76)} \cong 0{,}88$$

O fator de correlação de 0.88 para o circuito c2670.isc, representa uma relação linear positiva muito boa para a cobertura de falhas dos conjuntos SOPRANO e FSIM. Dessa forma, a partir da análise das falhas *stuck-open* de um determinado circuito pode-se estimar o comportamento deste circuito para as falhas *stuck-at*, trazendo dessa forma uma economia no preço final do dispositivo sob teste. O mesmo procedimento de cálculo foi aplicado às Tabelas III.9, III.10 e III.11. Os resultados encontrados são mostrados na Figura III.7, nesta figura pode-se encontrar o correspondente valor de cobertura de falhas *stuck-at* para um dado circuito, a partir do valor das falhas *stuck-open*. Tal procedimento traz uma economia no esforço computacional, necessário para o processo de simulação, otimizando o processo de simulação.

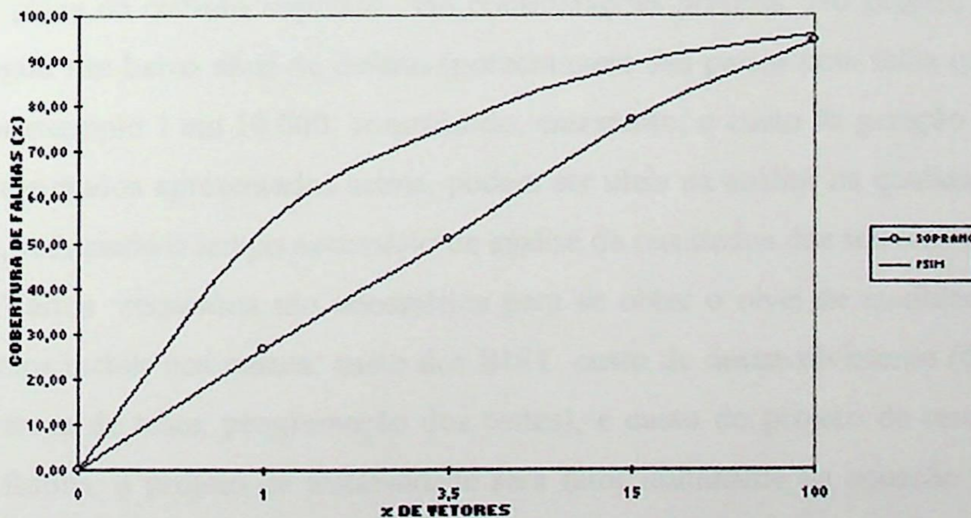


Figura III.7 - Curva de Cobertura de Falhas (SOPRANO X FSIM).

A análise do modelo de falha do dispositivo sob teste, traz informações que serão úteis no aprimoramento do processo VLSI [Flow86]. Dessa forma, dispositivos faltosos normalmente apontam para padrões de falhas repetitivas. A investigação de tais falhas podem apontar os pontos fracos (sensibilidade do processo de fabricação) do projeto. Tais informações são usadas na busca lógica e nas estratégias dos projetos de *layout* [Lin78, Farn85]. Portanto, a estimativa dos efeitos de uma falha *stuck-open*, a partir da análise das falhas *stuck-at*, pode afetar de forma decisiva, o processo de fabricação do dispositivo final.

A qualidade de um produto depende da eficácia de seus testes. A eficácia de um teste é muito freqüentemente medida como a cobertura de falhas simples do tipo (*stuck-open, stuck-at*). Assim, os testes são avaliados de acordo com sua eficácia para detectar linhas que se comportam como se estivessem curto-circuitadas com o terra, ou com a fonte de alimentação, ou como um problema intrínseco do dispositivo. Uma vez que a natureza e o número de falhas que podem ocorrer dependem do tipo do dispositivo (*chip*, placa e assim por diante) e da tecnologia (CMOS, bipolar, GaAs) então, a avaliação da qualidade do teste pode ser uma tarefa complicada. Em geral, requisitos de qualidade tais como 95% de cobertura de falha para *chips* VLSI e 100% de cobertura de todas as falhas de uma placa de circuito impresso, são considerações práticas. No projeto dos dispositivos, tenta-se conseguir um baixo nível de defeito (porcentagem das partes com falha que são aprovadas pelo teste), por exemplo 1 em 10.000, controlando, entretanto, o custo da geração do teste e de sua aplicação. Os resultados apresentados acima, podem ser úteis na análise da qualidade, para sistemas muito grandes, reduzindo o tempo necessário de análise de resultados dos testes.

Vários requisitos são necessários para se obter o nível de qualidade com um custo mínimo. Podemos incluir nos custos: custo dos BIST, custo de desenvolvimento (ferramentas CAD, geração de vetores de teste, programação dos testes), e custo do projeto de testabilidade [Pitt84, Ambl86]. No futuro, o projeto de testabilidade será fator dominante na equação de economia dos testes. Os resultados alcançados neste trabalho, podem ser usados como fator de redução ainda maior de tais custos.

Quando a complexidade dos sistemas VLSI aumenta, pergunta-se se o problema de teste pode ser particionado. A resposta, infelizmente, é não. Considere, por exemplo, dois dispositivos conectados em cascata. Não existe, freqüentemente, uma maneira simples de se derivar testes para a cascata a partir dos testes para os componentes individuais. Outra possibilidade é utilizar uma abordagem hierárquica. Os complexos problemas de automação de projetos, síntese e *layout* (projeto físico) são freqüentemente resolvidos através de procedimentos hierárquicos. O problema de teste, entretanto, não é fácil de ser resolvido com métodos hierárquicos convencionais. Pode-se concluir que, o problema da complexidade dos testes para grandes circuitos, pode ser minimizado com a ajuda dos resultados encontrados neste trabalho, pois pode-se estimar o comportamento do circuito para determinada falha, diminuindo novamente o tempo necessário para a realização dos testes.

A confiabilidade representa um importante fator, na análise de desempenho dos dispositivos. Inicialmente, por simplicidade, considere do Grau de Falha λ sendo constante e independente do tempo (distribuição exponencial). Considere que os componentes analisados, estão em operação normal, dentro do limite do seu limite de vida útil.

Assume-se que 90% dos circuito estejam funcionando depois de 5 anos. Através de distribuição exponencial, a função densidade de falha f(t) e o grau de falha são tratados por [Billinton83]

$$f(t) = \lambda e^{-\lambda t}$$

onde t é o tempo (dado em anos). Então, após 5 anos, e considerando 90% do circuito sem falhas,

$$0.9\lambda = \lambda e^{-\lambda 5}$$

obtendo $\lambda = 0.021$ [falhas/ano].

MTTF (*Mean Time To Failure*) é a média de tempo que um sistema leva para falhar. MTTF é dado por $1/\lambda$, ou MTTF = 47.46 [anos]. Novamente, os resultados encontrados, podem ser extrapolados, de tal sorte que, em relação a confiabilidade do dispositivo, a estimativa de cobertura das falhas pode trazer, um aumento deste fator, já que a análise inicial de desempenho pode ser feita com maior critério, utilizando-se o tempo do processo de simulação de falha estimado.

O capítulo V, mostra a conclusão deste trabalho, bem como algumas proposições para futuras melhorias.

IV - CONCLUSÕES E PROPOSTAS DE EXTENSÃO

Os testes de circuitos integrados se tornam cada vez mais caros, atingindo cerca de 30% do custo final do dispositivo [Levitt 92]. A alta densidade e a complexidade dos circuitos integrados, não fazem os testes se tornarem somente caros, mas também complexos e muito demorados. Existe a necessidade de se entender como que as falhas afetam os sistemas, cada vez mais complexos.

Como o número de circuitos dentro dos dispositivos a serem testados tem crescido, os custos da geração dos testes crescem na mesma proporção. O esforço computacional é proporcional ao quadrado das portas do dispositivo [Goel 80]. Existem três maneiras de se prover meios para realizar: 1) usar um computador mais veloz; 2) trabalhar com um algoritmo mais rápido, ou mais simples; 3) projeto do circuito, de tal sorte que, a geração dos testes possa ser acompanhado com maior facilidade.

BIST torna os testes menos complexos, pois o equipamento externo pode ser simplificado, e os testadores são mais simples e mais baratos. Embora o teste possa ser simplificado, o esforço computacional ainda é muito alto. Métodos de testes Pseudo-exaustivo, são usados no projeto de circuitos BIST. Testes Pseudo-exaustivos reduzem o esforço computacional, embora o tempo exigido ainda seja alto.

Na tecnologia VLSI atual, os modelos de falhas *stuck* (*stuck-at* - *stuck-open*) se mostram como um padrão para os modelos de falhas. Este trabalho foi dedicado a investigação desses modelos de falhas, buscando as relações que pudessem existir entre os modelos.

No capítulo I procurou-se mostrar o universo no qual se insere o trabalho desenvolvido. Apresentou-se uma breve revisão sobre conceitos básicos de testabilidade, necessários ao entendimento do conteúdo do trabalho.

No capítulo II é feita uma revisão das técnicas utilizadas em teste incorporado. É Procurou-se apresentar uma compilação razoavelmente completa das alternativas para geração de feito um apanhado das técnicas possíveis para a implementação de BIST, particularmente aquelas baseadas em registradores de deslocamento com realimentação linear (LFSR). Essas estruturas são analisadas tanto para a geração de vetores pseudoexaustivos e de vetores pseudoaleatórios.

Neste capítulo também são apresentados os conceitos de modelos de falhas, com suas vantagens e limitações. Os conceitos dos modelos de falhas *stuck-at* e *stuck-open* foram mostrados.

Na detecção de erros dos testes, o diagnóstico das falhas precisam ser implementadas para localizar a falha que esta causando o erro. Neste trabalho, os métodos para o diagnóstico de falhas foram utilizados na simulação de falhas em circuitos padrões, com a finalidade de se fazer a comparação entre os mesmos.

No capítulo III, apresentou-se os resultados das análises das falhas *stuck-at e stuck-open*. Para realizar as simulações foram feitas mudanças na estrutura dos *softwares* FSIM e SOPRANO, desenvolvido na *Virginia Polytechnic & State University* . Tais modificações foram necessárias para se poder controlar o percentual de vetores de teste aplicados ao circuitos sob teste (ISCAS85).

Os padrões de testes são aplicados ao CUT, até que todas as falhas possam ser detectadas. No processo de simulação de falhas, para cada padrão de teste gerado pelo LFSR, os CUT são simulados, e as falhas encontradas são marcadas. Este processo para, quando todas as possíveis falhas foram detectadas.

Este procedimento é repetido para diferentes LFSR (implementado diferentes polinômios característicos). O tempo necessário para simulação é sensivelmente reduzido, mesmo que cada LFSR possam gerar, diferentes seqüências de padrões de teste, cada qual requerendo um tempo de teste para detectar todas as falhas.

Os resultados são analisados, buscando-se detectar a correlação entre os modelos, verificou-se que o fator de correlação representou uma boa relação positiva linear. Podendo-se então concluir que quando se detectar a cobertura de falha para um determinado modelo, basta aplicar o fator de correlação, e estimar o comportamento do dispositivo para o outro modelo. Tal procedimento pode trazer uma economia substancial ao custo do projeto dos dispositivo.

Várias extensões podem ser sugeridas ao trabalho apresentado. Dentre elas destacam-se:

a) apesar da pesquisa bibliográfica, o texto poderia ser enriquecido com uma discussão sobre outros modelos de falhas e o correspondente impacto sobre as técnicas de autoteste;
b) implementação de Métodos de Testes Reduzido, para aplicação em CUT's, analisando o comportamento dos modelos de teste para o novo método.

REFERÊNCIAS BIBLIOGRÁFICAS

[Abra86] M. Abramovici, J.J Kulikowski, P.R. Menon, and D.T. Miller, "SMART and FAST: Test Generation for VLSI Scan-Design Circuits," *IEEE Design & Test of Computers*, Vol.3, pp.43-54, agosto 1986.

[Agra93a] Vishwani D. Agrawal, Charles R. Kime, Kewal K. Saluja, "A Tutorial on Built-In Self Test, Part 1: Principles," *IEEE Design & Test of Computers*, março 1993.

[Agra93b] Vishwani D. Agrawal, Charles R. Kime, Kewal K. Saluja, "A Tutorial on Built-In Self - Test, Part 2: Principles," *IEEE Design & Test of Computers*, junho 1993.

[Agrawal88] V.D. Agrawal and S.C. Seth, "Tutorial Test Generation for VLSI chips," *IEEE Computer Society Press*, Washington, 1988.

[Amaz88] José Roberto de A. Amazonas, "Estudo, Implementação e Aplicação de Técnicas Pseudo-Exaustivas ao Teste de Circuitos Intregrados Digitais," Tese de Doutorado apresentada à Escola Politécnica da USP, 1988.

[Amaz96] José Roberto de A. Amazonas, "Uma Contribuição ao Estudo de Auto-Teste Incorporado e Desenvolvimento de uma Nova Técnica de Análise de Assinatura," Tese de Livre-Docência apresentada à Escola Politécnica da USP, 1996.

[Ambl86] A.P. Ambler, M. Paraskeva, D.F. Burrows, W.L. Knight, and I. D. Dear, "Economically Viable Automatic Insertion of Self-Test Features for Custom VLSI," *Proc. Int. Test Conf.*, Washington, D.C., pp.232-243, setembro 1986.

[Bardell87] P. H. Bardell, W. H. McAnney, and J. Savir, "Built-in Test for VLSI Pseudorandom Techniques," *Wiley-Interscience*, 1987.

[Brglez85] F. Brglez and H. Fujiwara, "A Neutral List of 10 Combination Benchmark Circuits on a Target Translator in Fortran," *IEEE Intel. Symp. on Circuits and Systems*, pp.695-698, junho 1985.

[Baten38] Willian D. Baten, Elementary Mathmatical Statistic, John Wiley & Sons, NY, 1938.

[Blan84] T. Blank, "A Survey of Hardware Accelerators Used in Computer-Aided Design," *IEEE Design & Test of Computers*, pp. 21-39, agosto 1984.

[Cha79] C. W. Cha, "Multiple Fault Diagnosis in Combination Networks," *Proc. 16th Des. Auto. Conf.*, San Diego, CA, Vol. CAD-6, pp.232-240, março 1987.

[ChanM70] H. Y. Chang, E. G. Manning, and G. Metze, "Fault Diagnosis of Digital Systems," *Viley-Interscience*, New York, 1970.

[Chal89] P. R. Chalasani, S. Bhawmik, A. Acharya, P. Palchan "Design of Testable VLSI Circuits with Minimun Area Overhead," *IEEE Transaction on Computers*, outubro 1989.

[Dear91] I. D. Dear, C. Dislis, A. P. Ambler, J. Dick, "Economic Effects in Design and Test," *IEEE Design & Test of Computers*, dezembro 1991.

[Elder59] R. D. Elder, "Test Routines Based on Symbolic Logical Statements," *Journal of the ACM*, pp.33-36, janeiro 1959.

[FrieM71] A.D. Friedman, "Fault Detection in Redundant Circuits," *IEET-EC*, pp. 99-100.

[Glos89] Clay S. Gloster, Franc Brglez, "Boundary Scan With Built-in Self-Test," *IEEE Design & Test of Computers*, fevereiro 1989.

[Ghosh92] A. Glosh, S. Devadas and A. R. Newton, "*Sequential Logic Testing and Verification*", Kluwer Academic Publishing, Boston 1992.

[Goel73] P. Goel, "The Feed Forward Logic Model in the Testing of Large Scale Integrated Logic Circuits," PhD. Dissertation, Carnegie-Mellon University, Pittsburg, PA, setembro 1973.

[Goel80] P. Goel, "Test Generation Costs Analysis and Projections, " *Proc. 17th Design Automation Conference*, Minneapolis, MN, pp. 77-84, junho 1980.

[Kuba84] J. Kuban, W. Bruce, "Self-Testing yhe Motorola MC6804P2," *IEEE Design & Test of Computers*, pp. 33-41, maio 1984.

[Lo87] C. Y. Lo, H. N. Nham, and A. K. Bose, "Algorithms for an Advanced Fault Simulations System in MOTIS," *IEEE Trans. CAD, Vol. CAD-6*, pp. 232-240, março 1987.

[Levitt92] M. E. Levitt, "ASIC Testing Upgraded," *IEE Spectrum*, vol.29, pp.26-29, maio 1992.

[McCI84] E. J. McCluskey, "Verification Testing - A Pseudoexhaustive Test Technique," *IEEE Transactions on Computers*, Vol. C-33, pp. 541-546, No.6, junho 1984.

[Poage63] J. F. Poage, "Derivation of Optimun Tests to Detect Faults in Combinational Circuits," *Proc. 1963 Symp. on Math*. Theory of Automata, pp.183-528.

[Pitt84] J. S. Pittman and W. C. Bruce, "Test Logic Economic Consideration in a Commercial VLSI Chip Enviroment," *Proc. Int. Test Conf., Philadelphia, PA*, pp. 31-39, outubro 1984.

[Suss73] A. K. Susskind, "Diagnostics for Logic Networks," *IEE Spectrum*, pp. 40-47.

[Sche72] D.R. Schertz and G. Metze, "A New Representation for Faults in Combinational Circuits by Maximizing the Probability os Fault Detection," *IEEE Computers*, Vol. C-29, pp. 410-416, maio 1980.

[Thom75] E. W. Thompson and S. A. Szygenda, "Digital Logic Simulation in a Time-Based, Table-Driven Environment, Part 2, Parallel Fault Simulation," *Computer*, Vol.8, pp. 38-44, março 1975.

[Willians 86] T. W. Willians, W. Daehn, M. Gruetzner and C. W. Starke, "Comparison of Alising Erros for Primitive and Non-Primitive Polynomials," *Int. Test Conference*, pp. 282-288, 1986.

[Wang86a] L. T. Wang, E. J. McCluskey, "Condensed Linear Feedback Shift Register (LFSR) Testing," *IEEE Transactions on Computers*, pp. 1145-1150, dezembro 1983.

[Wang86b] L. T. Wang, E. J. McCluskey, "Circuits for Pseudo-exhaustive Teste Generation," *Proceeding IEEE International Test Conference*, setembro 1986.

[Wang88a] L. T. Wang, E. J. McCluskey, "Linear Feedback Shift Register Using Cyclic Codes, "*IEEE Transactions on Computer*," outubro 1988.

[Wang88b] L. T. Wang, E. J. McCluskey, "Circuit for Pseudo-exhaustive Test Pattern Generation," *IEEE Transaction on Computer Aided of Integraded Circuits and Systems*, outubro 1988.

APÊNDICE A

```
/*************************************************************************

        Copyright (C) 1991,
        Virginia Polytechnic Institute & State University


        This program was originally written by Mr. Hyung K. Lee
        under the supervision of Dr. Dong S. Ha, in the Bradley
        Department of Electrical Engineering, VPI&SU, in 1991.

        This program is released for research use only. This program,
        or any derivative thereof, may not be reproduced nor used
        for any commercial product without the written permission
        of the authors.

        For detailed information, please contact to

        Dr. Dong S. Ha
        Bradley Department of Electrical Engineering
        Virginia Polytechnic Institute & State University
        Blacksburg, VA 24061

        Ph.: (703) 231-4942
        Fax: (703) 231-3362
        E-Mail: ha@hope.etg.ee.vt.edu

*************************************************************************/

/***************************** HISTORY ************************************

            soprano: version 1.0

            Original: H. K. Lee, 8/15/1990
            Updated: H. K. Lee, 8/15/1991

            soprano: version 1.1

            Changed Parser and added on-line manual: H. K. Lee, 10/5/1992

            Now, soprano accepts the circuit written in the netlist format
            of ISCAS89 benchmark circuits as well as the netlist format of
            ISCAS85 benchmark circuits.

*************************************************************************/

/*-------------------------------------------------------------------
            soprano.c
            Main program of soprano.
            An automatic test pattern generator for single
            stuck-open faults in CMOS combinational circuits.
            Assumes zero gate delays.
-------------------------------------------------------------------*/
```

```
#include <stdio.h>
#include <time.h>
#include "atpg.h"
#include "truthtable.h"

#define DONE (-1)
#define CHECKPOINTMODE 1
#define DEFAULTMODE 3
#define is_random_mode(mode) (mode=='y')

#define output0 output

#define checkbit(word,nth) ((word&bitmask[nth])!=ALL0)
#define setbit(word,nth) (word|=bitmask[nth])
#define resetbit(word,nth) (word&=(~bitmask[nth]))

/* external variables */
extern void setfanoutstem();
extern void set_unique_path();
extern void pinit_simulation();
extern void GetPRandompattern(),GetRandompattern();
extern void pfault_free_simulation(),pfirst_fault_free_simulation();
extern void initialize_FGL();
extern void update_all1();
extern char *strcpy(), *strcat();
extern void print_log_topic();
extern void gettime();
extern void printinputs(), printoutputs(), printfault();
extern void set_testability();
extern void fault_drop();
extern long random();

/* variables for main program */
level test_vectors[MAXTEST][MAXPI+1];

/* default parameters setting */
char name1[100]="",name2[100]="",name3[100]="",commandfile[100];
char namecct[100]="";
char inputmode='d';               /* default mode */
char rptmode='y';          /* RPT mode ON */
char logmode='n';                 /* LOG off */
char helpmode='q';          /* On-line help mode */
char learnmode='n';               /* unused */
char cctmode=ISCAS89;         /* input circuit format */
int iseed=0;                      /* initial random seed */
int randomlimit=2;                /* condition for RPT stopping */
int maxbacktrack=10;              /* maximum backtracking of FAN */
level LFSR[MAXPI];

/*      main
        Main program of SOPRANO.
*/
```

```
void main(argc,argv)
int argc;
char *argv[];
{
  char tecla[10];
  int numbergate,numberpi,numberpo,noff=0,ndom;
  int numberfault,nrestoredfault,maxdetect;
  int maxdpi;
  FAULTTYPE *pcurrentfault,*f;
  GATEPTR gut,*stem;
  int nstem;
  int i,j,k,n,iteration;
  status state,fault_selection_mode;
  int nbacktrack;
  int ntried=0;
  int ndetect=0, nredundant=0, noverbacktrack=0, ntest=0;
  int ntest1=0, ntest2=0, ntest3=0,ntest4=0;
  int ndetect1=0, ndetect2=0, ndetect3=0,ndetect4=0;
  int SOPdetect=0;
  int tbacktrack=0;
  level SOPtest[MAXTEST*8],SOPcompact[MAXTEST*8];
  int numberofSOPtest;
  double minutes,seconds,starttime,inittime,runtime1,runtime2;
  double fantime1,simtime1,simtime2,simtime3,simtime4;
  int numbercheckfault;
  char c;
  level ran;

  boolean first=TRUE;
  int maxbits=BITSIZE;
  int fault_profile[MAXTEST*8];
  boolean dropbit=FALSE;
        float porcentagem;

/*******************************************************************
 *                                                   *
 *       step 1: preprocess ---                            *
 *              input and output file interface                *
 *                                                   *
 *******************************************************************/

  if(argc==1) {
    helpmode='d';
  } else for(i=1;i<argc;i++) {
       if(argv[i][0]=='-') {
         if((i=option_set(argv[i][1],argv,i,argc))<0) {
           helpmode='d'; break;
       }}
       else strcpy(name1,argv[i]);
     }
  if(inputmode=='f') file_mode();

  if(helpmode!='q') { help(helpmode); exit(0); }
```

```
if((circuit = fopen(name1,"r")) == NULL) {
  fprintf(stderr,"Fatal error: no such file exists %s\n",name1);
  exit(0);
}

strcpy(namecct,name1);

i=0; j=0;
if(name2[0]=='\0') {
  while((c=name1[i++])!='\0') {
    if(c=='/') j=0;
    else if(c=='.') break;
    else name2[j++]=c;
  }
  name2[j]='\0';
  strcat(name2,".test");
}

i=0; j=0;
if(name3[0]=='\0') {
  while((c=name1[i++])!='\0') {
    if(c=='/') j=0;
    else if(c=='.') break;
    else name3[j++]=c;
  }
  name3[j]='\0';
  strcat(name3,".log");
}

if((test = fopen(name2,"w")) == NULL) {
  fprintf(stderr,"Fatal error: %s file open error\n",name2);
  exit(0);
}

if(logmode=='y')
  if((logfile = fopen(name3,"w")) == NULL) {
        fprintf(stderr,"Fatal error: %s file open error\n",name3);
        exit(0);
  }
if(logmode=='y') print_log_topic(logfile,name1);

iseed=Seed(iseed);

gettime(&minutes,&seconds,&runtime1);
starttime=runtime1;

       printf("entre com a porcentagem :");
       scanf("%f",&porcentagem);
/*****************************************************************
 *                                              *
 *       step 1: preprocess ---                        *
 *             construction of data structures              *
 *                                              *
 *****************************************************************/
```

```
if(cctmode==ISCAS89) {
  if(read_circuit(namecct,&numbergate,&numberpi,&numberpo,&noff) <0) {
    fprintf(stderr,"Fatal error: Invalid circuit file.\n");
    exit(0);
  }
} else if(!circin(&numbergate,&numberpi,&numberpo)) {
  fprintf(stderr,"Fatal error: Invalid circuit file\n");
  exit(0);
}
fclose(circuit);

if(numbergate<=0 || numberpi<=0 || numberpo<=0) {
  fprintf(stderr,"Fatal error: Invalid circuit file\n");
  exit(0);
}

if(cctmode==ISCAS89) {
  ALLOCATE(stack.list,GATEPTR,numbergate+10);
  clear(stack);
  if(levelize(net,numbergate,numberpi,numberpo,noff,stack.list) <0) {
    fprintf(stderr,"Fatal error: Invalid circuit file.\n");
    exit(0);
  }
  if(noff > 0) {
    fprintf(stderr,"Error: Invalid type DFF is defined.\n");
    exit(0);
  }
} else stack.list=NULL;

maxdpi=set_cct_parameters(numbergate,numberpi);
set_testability(numbergate);
nstem=0;
```

1

101

```
for(i=0;i<numbergate;i++)
  if(is_fanout(net[i]) || net[i]->po) nstem++;
stem=(GATEPTR *)malloc((unsigned)(sizeof(GATEPTR)*nstem));
setfanoutstem(numbergate,stem,nstem);

if((numbercheckfault=set_SOP_fault_list(numbergate,nstem,stem))<0) {
  fprintf(stderr,"Fatal error: error in setting fault list.\n");
  exit(0);
}
numberfault=fault_list.last+1;

for(i=0;i<numbergate;i++) {
  reset(net[i]->ichange);
  net[i]->freach=numbergate;
}
ndom=set_dominator(numbergate,maxdpi);
set_unique_path(numbergate,maxdpi);
```

```
/*****************************************************************
 *                                    *
 *     step 1: preprocess ---                    *
 *          initialization of circuit parameters      *
 *                                    *
*****************************************************************/

  /* initialization of system parameters */
  for(i=0;i<numbergate;i++) reset(net[i]->ichange);
  for(i=0;i<numberfault;i++) {
    fault_list.list[i]->covered=UNDETECTED;
    fault_list.list[i]->t1=(-1);
    fault_list.list[i]->t2=(-1);
  }
  initialize_FGL(numbergate);
  maxdetect=numberfault;

  all_one=ALL1;
  bitmask[0]=~(ALL1<<1);
  for(i=1;i<BITSIZE;i++) bitmask[i]=bitmask[i-1]<<1;

  gettime(&minutes,&seconds,&runtime2);
  inittime=runtime2-runtime1;
  runtime1=runtime2;
  simtime1=0.0;

/*****************************************************************
 *                                    *
 *     step 2: Random pattern testing session            *
 *          1. generate 32 random patterns             *
 *          2. fault free simulation                *
 *          3. fault simulation                  *
 *          4. initialization check
 *                                    *
*****************************************************************/

  if(is_random_mode(rptmode)) {
    ntried=0;
    iteration=0;
    while(iteration<randomlimit) {
         ntried++;
         if(ntried>1) first=FALSE;
         GetPRandompattern(numberpi,LFSR);
      for(i=0;i<numberpi;i++) net[i]->output1=net[i]->output0=LFSR[i];
         for(i=0;i<maxbits;i++) fault_profile[i]=0;

if((n=Dominant_Test_Detect(numbergate,maxdpi,numberpi,numberpo,first,maxbits,fault_profile,ntest))>
0) {
           iteration=0;
           SOPdetect+=check_initialization(fault_profile,maxbits,ntest);
           for(i=maxbits-1;i>=0;i--) {
              if(fault_profile[i]>0) {
                    ntest++;
```

```
                    if(ntest>MAXTEST) printf("WARNING: number of test exceeds the maximum
allowed memory size. %d\n",ntest);
                    for(j=0;j<numberpi;j++)
                       if(checkbit(LFSR[j],i)) test_vectors[ntest][j]=ONE;
                       else test_vectors[ntest][j]=ZERO;
              }
            }
            ndetect+=n;
            if(ndetect>=maxdetect) break;
        /*          if (((float)ndetect/(float)numberfault*100)>=porcentagem){
                              printf("ntest:%d\n",ntest);
                              printf("Numberfault:%d\n",numberfault);

                              printf("ndetect:%d\n",ndetect);
                              printf("porcentagem:%f\n",(float)ndetect/(float)numberfault*100);
                              printf("continuar?(S/N)");
                              scanf("%s",&tecla);
                              if (tecla[0]=='N') break;
                    } */
          }
          else iteration++;
    }
    ntest1=ntest;
    ndetect1=ndetect;
    gettime(&minutes,&seconds,&runtime2);
    simtime1=runtime2-runtime1;
    runtime1=runtime2;
  }

  fantime1=0;
  fault_selection_mode=CHECKPOINTMODE;
  first=TRUE;
  dropbit=FALSE;

/*******************************************************************
 *                                                      *
 *    step 3: Deterministic Test Patrtern Generation Session      *number
 *          (fan + one-bit fault simulation)                *
 *                                                      *
 *******************************************************************/

  while(fault_selection_mode!=DONE) {
    /* select any undetected and untried fault */
    pcurrentfault=NULL;
    switch(fault_selection_mode) {
      case CHECKPOINTMODE:
            for(i=fault_list.last;i>=0;i--)
               if(fault_list.list[i]->checkpoint==TRUE)
            if(is_undetected(fault_list.list[i])) {
                  pcurrentfault=fault_list.list[i];
                  break;
               }
            if(pcurrentfault==NULL) fault_selection_mode=DEFAULTMODE;
            break;
```

```
      case DEFAULTMODE:
            for(i=fault_list.last;i>=0;i--)
          if(is_undetected(fault_list.list[i])) {
                pcurrentfault=fault_list.list[i];
                break;
              }
            if(pcurrentfault==NULL) fault_selection_mode=DONE;
            break;
    }
    if(pcurrentfault==NULL) continue;

    gut=pcurrentfault->faulty_gate;
    gettime(&minutes,&seconds,&runtime2);
    fantime1-=runtime2;

    /* test pattern generation using fan */
    state=fan(numbergate,maxdpi,numberpi,numberpo,pcurrentfault,maxbacktrack,&nbacktrack);
    tbacktrack+=nbacktrack;

    gettime(&minutes,&seconds,&runtime2);
    fantime1+=runtime2;

    if(state==TEST_FOUND) {
          /* fault is detected, delete the detected fault from fault list */
          ntest++;
if(ntest>MAXTEST) printf("WARNING: number of test patterns exceeds the maximum allowed memory
space\n");
          pcurrentfault->covered=PROCESSED;
/*        pcurrentfault=DETECTED;
          pcurrentfault->t2=ntest;
          if(--(gut->nfault)==0){delete_FGL(gut->gid);fault_drop(numbergate);}*/

          /* assign random zero and ones to the unassigned bits */
          for(j=0;j<numberpi;j++) {
            if(net[j]->output==ZERO) {
              test_vectors[ntest][j]=ZERO;
              net[j]->output1=ALL0;
            }
            else if(net[j]->output==ONE) {
              test_vectors[ntest][j]=ONE;
              net[j]->output1=ALL1;
        }
            else {
              ran = (level)random();
              net[j]->output1=ran&01;
              test_vectors[ntest][j]=net[j]->output1;
            }
            reset(net[j]->ichange);
            net[j]->output0=net[j]->output1;
          }

          /* initialization for the fault simulation */
          for(j=numberpi;j<numbergate;j++) { reset(net[j]->ichange); }
          if(dropbit) {fault_drop(numbergate); reset(dropbit);}
```

```
          clear(stack);

          /* fault simulation */
          if(--ntest>0) first=FALSE;
          fault_profile[0]=0;

ndetect+=Dominant_Test_Detect(numbergate,maxdpi,numberpi,numberpo,first,1,fault_profile,ntest);

          if(!is_detected(pcurrentfault))
            fprintf(stderr,"Warning: error in test pattern generation.\n");

       SOPdetect+=check_initialization(fault_profile,1,ntest);
          if(++ntest>0) first=FALSE;
    }
    else if(state==NO_TEST) {
          /* redundant faults */
          pcurrentfault->covered=REDUNDANT;
          swap(fault_list,i,fault_list.last,f);
          delete_last(fault_list);
          if(--(gut->nfault)==0) {delete_FGL(gut->gid);set(dropbit);}
          nredundant++;
    }
    else {
          /* over backtracking */
          noverbacktrack++;
          pcurrentfault->covered=PROCESSED;
    }
  }

  ntest2=ntest;
  ndetect2=ndetect;

  gettime(&minutes,&seconds,&runtime2);
  simtime2=runtime2-runtime1-fantime1;
  runtime1=runtime2;

/************************************************************************
 *                                          *
 *    step 4: Organization of test patterns          *
 *          Minimal length superstring              *
 *                                          *
 ************************************************************************/

  numberofSOPtest=(int)((float)shortest_superstring(numberfault,SOPtest,ntest)*porcentagem/100);

  ntest3=numberofSOPtest;
  ndetect3=SOPdetect;

  gettime(&minutes,&seconds,&runtime2);
  simtime3=(runtime2-runtime1);
  runtime1=runtime2;

/************************************************************************
 *                                          *
```

```
*    step 5: Test Compaction session                              *
*         32-bit forward and reverse SOP fault simulation         *
*                                                                 *
*******************************************************************/

  /* forward fault simulation */
  if((nrestoredfault=restore_SOP_fault_list(numberfault,nstem,stem))<0) {
    fprintf(stderr,"Warning: error in restoreation of the fault list.\n");
    exit(0);
  }

ndetect4=SOP_fault_simulation(test_vectors,SOPtest,fault_profile,&numberofSOPtest,numbergate,numbe
rpi,numberpo,numberfault,maxdpi);

  /* reverse fault simulation */
  if((nrestoredfault=restore_SOP_fault_list(numberfault,nstem,stem))<0) {
    fprintf(stderr,"Warning: error in restoreation of the fault list.\n");
    exit(0);
  }

  k=0;
  fault_profile[numberofSOPtest]=0;
  for(i=numberofSOPtest-1;i>=0;i--)
    if(fault_profile[i]==0) {
          j=i;
          SOPcompact[k++]=SOPtest[j++];
          while(fault_profile[j]>0) SOPcompact[k++]=SOPtest[j++];
    }

ndetect4=SOP_fault_simulation(test_vectors,SOPcompact,fault_profile,&numberofSOPtest,numbergate,n
umberpi,numberpo,numberfault,maxdpi);

  ntest4=numberofSOPtest;
  gettime(&minutes,&seconds,&runtime2);
  simtime4=runtime2-runtime1;
  runtime1=runtime2;

/*******************************************************************
*                                                                 *
*    End of test pattern generation.                              *
*    Output test patterns and fault simulation result.            *
*                                                                 *
*******************************************************************/

  /* put the test patterns to the test pattern file */
  print_test_topic(test,numberpi,numberpo,namecct);

  for(i=0;i<ntest4;i++) {

    for(j=0;j<numberpi;j++)
          net[j]->output0 = net[j]->output1
          = test_vectors[SOPcompact[i]][j]&01;
```

```
    pfirst_fault_free_simulation(numbergate,numberpi);

    printio(test,numberpi,numberpo,0,i+1);

    if(logmode=='y') {
            fprintf(logfile,"test %4d: ",i);
            printinputs(logfile,numberpi,0);
            fprintf(logfile," ");
            printoutputs(logfile,numberpo,0);
            fprintf(logfile," %4d faults detected\n",i,fault_profile[i]);
    }
  }

  /* print out the results */
  print_atpg_head(stdout);
  print_atpg_result(stdout,namecct,numbergate,numberpi,numberpo,maxdpi,
               maxbacktrack,ntest3,ntest4,numberfault,ndetect4,
               nredundant,tbacktrack,inittime,
               simtime1+simtime2+simtime3+simtime4,
                      fantime1,runtime2-starttime,'n');
  if(logmode=='y') {
    fprintf(logfile,"\nEnd of test pattern generation.\n\n");
    print_atpg_head(logfile);
    print_atpg_result(logfile,namecct,numbergate,numberpi,numberpo,maxdpi,
               maxbacktrack,ntest3,ntest4,numberfault,ndetect4,
               nredundant,tbacktrack,inittime,
               simtime1+simtime2+simtime3+simtime4,
                      fantime1,runtime2-starttime,'y');   }

  fclose(test);
  if(logmode=='y') fclose(logfile);
}


int option_set(option,array,i,n)
char option,*array[];
int i,n;
{
  if(i+1 >= n) return((-1));

  switch(option) {
    case 'd': inputmode='d'; break;
    case 'I': cctmode=ISCAS85; break;
    case 'f': inputmode='f'; strcpy(commandfile,array[++i]); break;
    case 'r': sscanf(array[++i],"%d",&randomlimit);
            if(randomlimit==0) rptmode='n'; break;
    case 's': sscanf(array[++i],"%d",&iseed); break;
    case 'b': sscanf(array[++i],"%d",&maxbacktrack); break;
    case 't': strcpy(name2,array[++i]); break;
    case 'l': logmode='y'; strcpy(name3,array[++i]); break;
    case 'n': strcpy(name1,array[++i]); break;
    case 'h': helpmode=*(array[++i]); break;
    default:  i=(-1);
```

```
  }
  return(i);
}

#define is_white_space(c) (c==' ' || c=='\t' || c=='\n')

int file_mode()
{
  FILE *fp;
  char c;
  char array[30][100];
  int count=1,i;

  if((fp=fopen(commandfile,"r")) == NULL) return(FALSE);
  i=0;
  while((c=getc(fp))!=EOF)
    if(is_white_space(c)) {
      if(i>0) {array[count++][i]='\0'; i=0;}
    } else array[count][i++]=c;
  if(i>0) array[count++][i]='\0';

  fclose(fp);

  for(i=1;i<count;i++)
    if(array[i][0]=='-') {
      if(i+1>=count) return(FALSE);

      switch(array[i][1]) {
        case 'r': sscanf(array[++i],"%d",&randomlimit);
                  if(randomlimit==0) rptmode='n'; break;
        case 's': sscanf(array[++i],"%d",&iseed); break;
        case 'I': cctmode=ISCAS85; break;
        case 'b': sscanf(array[++i],"%d",&maxbacktrack); break;
        case 't': strcpy(name2,array[++i]); break;
        case 'l': logmode='y'; strcpy(name3,array[++i]); break;
        case 'n': strcpy(name1,array[++i]); break;
        case 'h': helpmode=*(array[++i]); break;
        default:  return(FALSE);
      }
    }
  return(TRUE);
}
```

```
/****************************************************************************

     Copyright (C) 1991,
     Virginia Polytechnic Institute & State University

     This program was originally written by Mr. Hyung K. Lee
     under the supervision of Dr. Dong S. Ha, in the Bradley
     Department of Electrical Engineering, VPI&SU, in 1991.

     This program is released for research use only. This program,
     or any derivative thereof, may not be reproduced nor used
     for any commercial product without the written permission
     of the authors.

     For detailed information, please contact to

     Dr. Dong S. Ha
     Bradley Department of Electrical Engineering
     Virginia Polytechnic Institute & State University
     Blacksburg, VA 24061

     Ph.: (703) 231-4942
     Fax: (703) 231-3362
     E-Mail: ha@hope.etg.ee.vt.edu

****************************************************************************/

/**************************** HISTORY **********************************

          fsim: version 1.0

          Original: H. K. Lee, 8/15/1990
          Updated: H. K. Lee, 8/15/1991

          fsim: version 1.1

          Changed Parser and added on-line manual: H. K. Lee, 10/5/1992

          Now, fsim accepts the circuit written in the netlist format
          of ISCAS89 benchmark circuits as well as the netlist format of
          ISCAS85 benchmark circuits.

****************************************************************************/

/*-----------------------------------------------------------------
          fsim.c
          Main program for FSIM.
          FSIM performs fault simulation for combinational circuits.
          Test patterns can be read from a user supplied file or
          generated randomly.
          Collapses the equivalent faults and fault coverage is
```

```
        computed based on the collapsed fault set.
-----------------------------------------------------------------------*/

#include <stdio.h>
#include "atpg.h"
#include <time.h>
#include <sys/types.h>
#include <strings.h>

#define DEFAULT_RLIMIT 224
#define is_random_mode(mode) (mode=='y')
#define is_fanout(gate) (gate->noutput>1)
#define output0    output

/* external functions */
extern void GetPRandompattern();
extern void pinit_simulation();
extern void pfault_free_simulation();
extern void printinputs(),print_log_topic(),printoutputs(),printfault();
char *strcpy(), *strcat();
extern void update_all(), update_all1();
extern void gettime();
extern void setfanoutstem();
extern void set_unique_path();
extern caddr_t sbrk();

/* default parameters setting */
char name1[100]="",name2[100]="",name3[100]="",commandfile[100];
char namecct[100]="";
char faultname[100]="";
char inputmode='d';                /* default mode */
char rptmode='y';          /* RPT mode ON */
char logmode='n';                  /* LOG off */
char helpmode='q';         /* On-line help mode */
char cctmode=ISCAS89;          /* input circuit format */
int iseed=0;                       /* initial random seed */
int randomlimit=DEFAULT_RLIMIT;        /* limit of random patterns */
int maxbit=BITSIZE;                /* number of bits for one operation */
char faultmode='d';
char compact;
int maxcompact;
int msize;
level LFSR[MAXPI];
FILE *faultfile;

/*
        main
        Main program of FSIM.
*/
main(argc,argv)
int argc; char *argv[];
{
  register int i, j;
  int nbit;
```

```
  int nog,npi,npo,maxdpi,nstem,ndominator,noff=0;
  int nof,ndetect,maxdetect,nredundant;
  int ntest;
  FAULTPTR f;
  GATEPTR *stem;
  int fault_profile[BITSIZE];
  double inittime,runtime;
  double runtime1,minutes,seconds;
  double coverage;
  char c;

  nbit=maxbit;
  update_flag=FALSE;
  update_flag2=FALSE;

/******************************************************************
 *                                                          *
 *        preprocess ---                                        *
 *        input and output file interface                            *
 *                                                          *
 ******************************************************************/

  if(argc==1) {
    helpmode='d';
  }
  else for(i=1;i<argc;i++) {
       if(argv[i][0]=='-') {
             if((i=option_set(argv[i][1],argv,i,argc))<0) {
            helpmode='d'; break;
            }}
            else strcpy(name1,argv[i]);
     }

  if(helpmode!='q') { help(helpmode); exit(0); }

  if((circuit = fopen(name1,"r")) == NULL) {
    fprintf(stderr,"Fatal error: no such file exists %s\n",name1);
    exit(0);
  }

  strcpy(namecct,name1);

  i=0; j=0;
  if(name3[0]=='\0') {
    while((c=name1[i++])!='\0') {
      if(c=='/') j=0;
      else if(c=='.') break;
      else name3[j++]=c;
    }
    name3[j]='\0';
    strcat(name3,".log");
  }

  if(rptmode=='n') if((test = fopen(name2,"r")) == NULL) {
```

```
    fprintf(stderr,"Fatal error: %s file open error\n",name2);
    exit(0);
  }

  if(logmode=='y')
    if((logfile = fopen(name3,"w")) == NULL) {
          fprintf(stderr,"Fatal error: %s file open error\n",name3);
          exit(0);
    }
  if(logmode=='y') print_log_topic(logfile,namecct);

  if(cctmode==ISCAS85 && faultmode=='f') {
    fprintf(stderr,"Fatal error in options:\n");
    fprintf(stderr,"The option -f can not combined with the option -I.\n");
    exit(0);
  }

  if(faultmode=='f')
    if((faultfile = fopen(faultname,"r")) == NULL) {
      fprintf(stderr,"Fatal error: %s file open error\n",faultname);
      exit(0);
    }

  if(rptmode=='y') iseed=Seed(iseed);

  gettime(&minutes,&seconds,&runtime1);

/*************************************************************
 *                                             *
 *            preprocess ---                        *
 *            construction of data structures            *
 *                                             *
 *************************************************************/

  if(cctmode==ISCAS89) {
    if(read_circuit(namecct,&nog,&npi,&npo,&noff) <0) {
      fprintf(stderr,"Fatal error: Invalid circuit file.\n");
      exit(0);
    }
  } else if(!circin(&nog,&npi,&npo)) {
    fprintf(stderr,"Fatal error: Invalid circuit file\n");
    exit(0);
  }
  fclose(circuit);

  if(nog<=0 || npi<=0 || npo<=0) {
    fprintf(stderr,"Fatal error: Invalid circuit file\n");
    exit(0);
  }

  if(cctmode==ISCAS89) {
    ALLOCATE(stack.list,GATEPTR,nog);
    clear(stack);
    if(levelize(net,nog,npi,npo,noff,stack.list) <0) {
```

```
      fprintf(stderr,"Fatal error: Invalid circuit file.\n");
      exit(0);
    }
    if(noff > 0) {
      fprintf(stderr,"Error: Invalid type DFF is defined.\n");
      exit(0);
    }
  } else stack.list=NULL;

  maxdpi=set_cct_parameters(nog,npi);
  if(!allocate_dynamic_buffers(nog)) {
    fprintf(stderr,"Fatal error: memory allocation error\n");
    exit(0);
  }

  nstem=0;
  for(i=0; i<nog; i++)
    if((net[i]->noutput>1) || (net[i]->po)) nstem++;
  stem=(GATEPTR *)malloc((unsigned)((sizeof(GATEPTR))*nstem));
  setfanoutstem(nog,stem,nstem);

  nof = (faultmode=='f') ?
       readfaults(faultfile,nog,nstem,stem) :
       set_all_fault_list(nog,nstem,stem);

  if(nof<0) {
    fprintf(stderr,"Fatal error: error in setting fault list\n");
    exit(0);
  }

  nof=fault_list.last+1;

  for(i=0;i<nog;i++) {
    reset(net[i]->ichange);
    net[i]->freach=nog;
  }
  ndominator=set_dominator(nog,maxdpi);
  set_unique_path(nog,maxdpi);

/*****************************************************************
 *                                                         *
 *        Initialization of circuit parameters          *
 *                                                         *
 *****************************************************************/

  for(i=0;i<nof;i++) {
    fault_list.list[i]->covered=UNDETECTED;
    fault_list.list[i]->observe=ALL0;
  }

  nredundant=check_redundant_faults(nog);

  pinit_simulation(nog,maxdpi,npi);
  ntest=0;
```

```
    ndetect=0;
    maxdetect=nof;

    all_one= (maxbit==BITSIZE) ? ALL1 : ~(ALL1<<maxbit);
    bitmask[0]=~(ALL1<<1);
    for(i=1;i<BITSIZE;i++) bitmask[i]=bitmask[i-1]<<1;

    gettime(&minutes,&seconds,&runtime1);
    inittime=runtime1;

/*****************************************************************
*                                          *
*          Main loop for the fault simulaiton.        *
*          1. Read test patterns.                *
*          2. Fault free simulation.               *
*          3. Fault Simulation.                          *
*          4. Update flags.                        *
*                                          *
*****************************************************************/

    while(ntest<randomlimit) {

      /* Get a test pattern */
      if(rptmode=='y') GetPRandompattern(npi,LFSR);
      else {
            if((nbit=pget_test(test,LFSR,npi,maxbit))==0) break;
            all_one=(nbit==BITSIZE) ? ALL1 : ~(ALL1<<nbit);
      }

      /* fault free simulation */
      for(i=0;i<npi;i++)
            net[i]->output1=net[i]->output0=LFSR[i];

      pfault_free_simulation();

      /* fault simulation */
      for(i=0;i<nbit;i++) fault_profile[i]=0;
      ndetect +=
            Fault1_Simulation(nog,maxdpi,npi,npo,nstem,stem,nbit,fault_profile);
      ntest+=nbit;

      if(logmode=='y') {
            ntest-=nbit;
            for(i=nbit-1;i>=0;i--) {
              fprintf(logfile,"test %4d: ",++ntest);
              printinputs(logfile,npi,i);
              fprintf(logfile," ");
              printoutputs(logfile,npo,i);
              fprintf(logfile," %4d faults detected",fault_profile[i]);

              fprintf(logfile,"\n");
            }
      }
```

Escola Federal de Engenharia de Itajubá - EFEI
Biblioteca
MAUÁ
BIM

```
    if(ndetect>=maxdetect) break;

    /* dynamic scheduling of network flags */
    for(i=0;i<=nsstack;i++) dynamic_stack[i]->cobserve=ALL0;
    if(update_flag) {
            if(logmode=='y') update_all1(npi);
            else update_all(npi);
            reset(update_flag);
    }
    else for(i=ndstack;i>nsstack;i--) dynamic_stack[i]->freach=0;
    ndstack=nsstack;
  }

  gettime(&minutes,&seconds,&runtime);
  coverage=(double)ndetect/(double)nof*100.00;
  msize = (int)sbrk(0)/1000;

  /* print out the results */
  print_sim_head(stdout);
  print_sim_result(stdout,namecct,nog,npi,npo,maxdpi,name2,
    ntest,nof,ndetect,inittime,runtime-inittime,runtime,'n');
  if(logmode=='y') {
    fprintf(logfile,"\nEnd of fault simulation.\n\n");
    print_sim_head(logfile);
    print_sim_result(logfile,namecct,nog,npi,npo,maxdpi,name2,
      ntest,nof,ndetect,inittime,runtime-inittime,runtime,'y');
    fclose(logfile);
  }

  if(rptmode=='n') fclose(test);
}


int option_set(option,array,i,n)
char option,*array[];
register int i,n;
{
  if(i+1 >= n) return((-1));

  switch(option) {
    case 'd': inputmode='d'; break;
    case 'I': cctmode=ISCAS85; break;
    case 'r': sscanf(array[++i],"%d",&randomlimit);
                  if(randomlimit==0) randomlimit=DEFAULT_RLIMIT; break;
    case 's': sscanf(array[++i],"%d",&iseed); break;
    case 't': rptmode='n'; randomlimit=INFINITY;
                  strcpy(name2,array[++i]); break;
    case 'l': logmode='y'; strcpy(name3,array[++i]); break;
    case 'n': strcpy(name1,array[++i]); break;
    case 'h': helpmode=*(array[++i]); break;
    case 'f': faultmode='f'; strcpy(faultname,array[++i]); break;
    default:  i=(-1);
  }
  return(i);
```

APÊNDICE B

* Name of circuit: c17
* Primary inputs :
1 2 3 6 7

* Primary outputs:
1 2

* Test patterns and fault free responses:

1: 01010 11
2: 00000 00
3: 11111 10
4: 00011 01
5: 00000 00
6: 01010 11
7: 11111 10

* Name of circuit: c1908
* Primary inputs :
1 4 7 10 13 16 19 22 25 28 31 34 37
40 43 46 49 53 56 60 63 66 69 72 76 79
82 85 88 91 94 99 104

* Primary outputs:
1 4 7 10 13 16 19 22 25 28 31 34 37
40 43 46 49 53 56 60 63 66 69 72 76

* Test patterns and fault free responses:

1: 110101100010010010100101110110111 1101100011001001100100111
2: 000000000000000000000000000000000 0000000000000000100000110
3: 001100101101111010001101101000110 0111110001010111101000111
4: 110101100010010010100101110110111 1101100011001001100100111
5: 001010000101101000101000000100010 0010010100000110111000101
6: 111100101011000000100101101000111 1111000001011101101000110
7: 000000000000000000000000000000000 0000000000000000100000110
8: 000001001101001101001001000110110 0000111010010100000101001
9: 000000000001010000000100110110111 0000000000000001101101010
10: 011100110000111010100110000010000 0111010001100011001000000
11: 010111000101100010000101111100101 0101110110000110100000110
12: 000000000000000000000000000000000 0000000000000000100000110
13: 001101001110110110000110000111000 0011101010011011100010010
14: 010011100100001010100001111010111 0100111111000000110011010
15: 111110110010000110001110010011010 1111000101101000111101000
16: 001101001110110110000110000111000 0011101010011011100010010
17: 011100110000111010100110000010000 0111010001100011001000000
18: 100100100000000010100101010000101 1001000000000000100000110
19: 100111011010011100001011010001110 1001011110111001010110000
20: 010011111000100100000000110110110 0100001111110110110110010
21: 000000000001010000000100110110111 0000000000000001101101010
22: 010101100110000100001000110110111 0101101011001100100110110
23: 111011111101101111000100110111110 1110111111110110011101010
24: 010101100110000100001000110110111 0101101011001100100110110
25: 001100010000010000010110110111110 0011000000100001010001000
26: 010101100110000100001000110110111 0101101011001100100110110
27: 101100001010001110000100000100010 1011011000011000001001011
28: 010101100110000100001000110110111 0101101011001100100110110
29: 111111101001110000000100110110100 1111000111010111000011010
30: 1111111110101000000001011000101100 1111000111111010000110100
31: 010101100110000100001000110110111 0101101011001100100110110
32: 100001110011101000101010000001010 1000010011101110001110111
33: 000110101000110000010000000011110 0001000101010011101101001
34: 000111010011100111010000000001110 0001001110101110011111111
35: 000110101000110000010000000011110 0001000101010011101101001
36: 000111001011101000011001010000110 0001010110011110001011000
37: 000110101000110000010000000011110 0001000101010011101101001
38: 010011001000100100010101100011110 0100001110010010010011000
39: 000110101000110000010000000011110 0001000101010011101101001
40: 0011000001101000000010100001000110 00111000000001010010101101
41: 000110101000110000010000000011110 0001000101010011101101001
42: 010011011011001010000111000000110 0100010110111100001001111

43: 01010110011000010000100011011011 01011010110011001001101100
44: 01000100000101100001101000111001 0 01000100100001011001101 01

43: 01010110011000010000100011011011 1

97: 1110101001001001010010100000010110 111010110100001010000000

98: 0000000000000000110010010000010110 000000000000000000010000110

99: 0000101011000101100010001010 01101 0100101101010001100000100

100: 0110110101101011001001011111111010 01101111101010100000011001

101: 0000101011000101100010001010 01101 0100101101010001100000100

102: 1110101001010011100001011010 00100 1110111110100010000000 1011

103: 0000101011000101100010001010 01101 0100101101010001100000100

104: 1111101011111110100000011000 11110 1111110101011111010000100

105: 0000101011000101100010001010 01101 0100101101010001100000100

106: 1101111001111111100101010010 11110 1101111111001111011000100

107: 0000101011000101100010001010 01101 0100101101010001100000100

108: 1001100111000101100001001110 00110 1001101100110001010101111

109: 0000101011000101100010001010 01101 0100101101010001100000100

110: 1100110010011100000000100000 001010 1100000111010111110001011

111: 1001001000000000101001010100 00101 1001000000000000100000110

112: 1001110100010000101000110100 00101 1001000110100100100000110

113: 0011010011101101100001000000 000000 0011101010011011000010010

114: 0000000000101000000010011011 0111 000000000000000001101101010

115: 1110001111011010101001100100 00000 1110110001110110100000001

116: 0001000010001100100101100000 01000 00010000000010011000001000

117: 1111101100100001100011100100 11010 1111000101101000111101000

118: 0101100010101011011001011110 01110 0101011100011010001010001

119: 1111101100100001100011100100 11010 1111000101101000111101000

120: 1010010111111110101001011100 10111 1010110010111111110100011

121: 1111101100100001100011100100 11010 1111000101101000111101000

122: 1110000010101110101110001110 11110 1110010000011011001101001

123: 1111101100100001100011100100 11010 1111000101101000111101000

124: 1001100111000101100001001110 00110 1001101100110001010101111

125: 0000101011000101100010001010 01101 0100101101010001100000100

126: 0111101110100100100000011010 00010 0111000101111001011000101

127: 0011111011001010001000001000 01000 0011110111010010001101010

128: 0101110001011001111011000101 10010 0100101110000110111010011

129: 0110001000101000001111100000 01110 0110000000100101000100 1011

130: 1111101100100001100011100100 11010 1111000101101000111101000

131: 0110001000101000001111100000 01110 0110000000100101000100 1011

132: 0010001000011001101011011001 00010 0010001001100110101101000

133: 1110010110010010111101100101 10010 1110010010010100110100101

134: 0010100001011010001010000000 100010 0010010100000110111000101

135: 0100000011111100110100101011 01000 0100100000001111101010 1011

136: 0010100001011010001010000000 100010 0010010100000110111000101

137: 1010010111111110101001011100 10111 1010110010111111110100011

138: 0010100001011010001010000000 100010 0010010100000110111000101

139: 1011000010100011100001000000 100010 1011011000011000001001011

140: 0010001110100110101010101000 11110 0011010001111001111100110

141: 1010110001111010111000010011 11010 1010110110001110001110100

142: 0010001110100110101010101000 11110 0011010001111001111100110

143: 1111101011111110100000011000 11110 1111110101011111010000100

144: 0010001110100110101010101000 11110 0011010001111001111100110

145: 0001010011101010001001010111 10010 0001110010011010010110101

146: 0010001110100110101010101000 11110 0011010001111001111100110

147: 1100001100000100011011001010 01010 1100000001100001010010100

148: 0001101100010001000000011000 00010 0001001101110100111101100

149: 1100001010111011010111100110 11010 1100011001001110110001001

150: 0010001110100110101010101000 11110 0011010001111001111100110

151: 01001001100110011100010001001000 0100001100110110011001011
152: 11011110111110010011010101001110 1101101110111100111101111
153: 00011011000100010000000110000001 0 0001001101110100111101100
154: 01001100100010010001010110001111 0 0100001110010010010011000
155: 11110111101100101010010001111101 0 1111010011111100110111011
156: 11111010111111101000000110001111 0 1111110101011111010000100
157: 11110111101100101010010001111101 0 1111010011111100110111011
158: 10011001010010011000111000011011 0 1001101100100010010100100
159: 11110111101100101010010001111101 0 1111010011111100110111011
160: 00010100111010100010010101111001 0 0001110010011010010110101
161: 11110111101100101010010001111101 0 1111010011111100110111011
162: 01001101101100101000011100000011 0 0100010110111100001001111
163: 01101101001101100000010111011001 0 0110010110101101101011011
164: 01011001100001011010010011000001 0 0101001100110001001010101
165: 01101101001101100000010111011001 0 0110010110101101101011011
166: 00010100111010100010010101111001 0 0001110010011010010110101
167: 01101101001101100000010111011001 0 0110010110101101101011011
168: 11100111111000100010000000100111 10 1110110011111000010111110
169: 10101111000101000110111010001101 0 1010000101100101111010011
170: 01101101001101100000010111011001 0 0110010110101101101011011
171: 01100010001010000011111000000111 0 0110000000100101000100101 1
172: 00000101000100110100001010001111 0 0000011010100100001001100
173: 11001000001001110000111001011101 0 1100011100001001101010011
174: 01101101011010110010010111111101 0 0110111110101010000011001
175: 11001000001001110000111001011101 0 1100011100001001101010011
176: 11111010111111101000000110001111 0 1111110101011111010000100
177: 11001000001001110000111001011101 0 1100011100001001101010011
178: 01001100100010010001010110001111 0 0100001110010010010011000
179: 00100011101001101010101010001111 0 0011010001111001111100110
180: 10101111000101000110111010001101 0 1010000101100101111010011
181: 11001000001001110000111001011101 0 1100011100001001101010011
182: 00110001110110000000111110000111 0 0011100000110110000000011
183: 11111110100111000000001001101101 00 1111000111010111000011010
184: 01101101001101100000010111011001 0 0110010110101101101011011
185: 10000100110100000001001110000101 10 1000100010010100000100100
186: 11000010101110110101111001101101 0 1100011001001110110001001
187: 11010001100000000000001010001000 10 1101000100110000110110001
188: 11000010101110110101111001101101 0 1100011001001110110001001
189: 00010100111010100010010101111001 0 0001110010011010010110101
190: 01010011011011101000100110010010 1 0111110001101011100000110
191: 01100010000111011000110010010110 1 0100001001000111100000110
192: 11100101100100101111011001011001 0 1110010010010100110100101
193: 10011001010010011000111000011011 0 1001101100100010010100100
194: 11100101100100101111011001011001 0 1110010010010100110100101
195: 00010010110011011011011000011001 0 0001101001010011010100101
196: 11100101100100101111011001011001 0 1110010010010100110100101
197: 01110011000010111001010010110111 0 0111011001100010011010001
198: 10000110011100101000100010100001 0 1000110011001100101000010
199: 10111111001110101000001010110001 0 1011010111101110001011011
200: 10000110011100101000100010100001 0 1000110011001100101000010
201: 01101101011010110010010111111101 0 0110111110101010000011001
202: 10000110011100101000100010100001 0 1000110011001100101000010
203: 01000010111000001110010000100001 0 0100100001011000000011100
204: 11100101100100101111011001011001 0 1110010010010100110100101

205: 01111010111111010111111001010100000 0111101101011111010100001
206: 01010011011011101000100110010010 1 01111100011010111000001 10
207: 11000010101110110101111001101101 0 1100011001001110110001001
208: 01101001101111011011010010100011 1 0110001100111111110111011 1
209: 00101101101001100001011000100001 0 0010010110111000101001100
210: 11010111110011100011100000000100 10 1101110011110011001100100
211: 10000110011100101000100010100001 0 1000110011001100101000010
212: 00101001011011110100110000001001 0 0010111100101011010100010
213: 00100011101001000110010000100001 0 0010000001111001100101000
214: 10000100110100000010011100001011 0 1000100010010100000100100
215: 01000100000101100001101000111001 0 0100010010000101100110101
216: 11100011110110101010011001000000 0 1110110001110110100000001
217: 11110000001001100100011011100001 01 0111100000000001110000011 0
218: 11110010101100000001001011010001 11 1111000001011101101000110
219: 00110010110111101000110110100011 0 0111110001010111101000111
220: 01010110100110001000110110100011 1 0001000011010110101100110
221: 01010101100101111111111000000001 0 0101011010110101001111101
222: 00000000000000000000000000000000 0000000000000000100000110
223: 01011001010110111000011001011011 0 0101111110100110100110000
224: 00001111000011000001101101110101 0 0000000111100011010100110
225: 01011001010110111000011001011011 0 0101111110100110100110000
226: 10100101010000110000100010101101 0 1010111010100000001110101
227: 10010001101110110010011011001011 0 1001011000111100101101001
228: 01011001010110111000011001011011 0 0101111110100110100110000
229: 10011001010010011000111000011011 0 1001101100100010010100100
230: 10010001101110110010011011001011 0 1001011000111100101101001
231: 11001000000100111000011100101110 10 1100011100001001101010011
232: 00100010101000010010101010000111 0 0010001001011000011101101
233: 00101101101001100001011000100001 0 0010010110111000101001100
234: 10000000011110001110011010101011 0 1000100000000111000001001 0
235: 00101101101001100001011000100001 0 0010010110111000101001100
236: 01101101011010110010010111111101 0 0110111110101010000011001
237: 00101101101001100001011000100001 0 0010010110111000101001100
238: 11000100011100000000000101101101 0 1100100010001100001011001
239: 10010001101110110010011011001011 0 1001011000111100101101001
240: 11100001111111111010111100100001 0 1110111000111111011101111
241: 11110111101100101010010001111101 0 1111010011111100110111011
242: 00000000000000001100100100001011 0 0000000000000000010000110
243: 11110000001001100100011011100001 01 0111100000000001110000011 0
244: 00111001101011010100100111000111 1 0011001100111011100001110
245: 00101000010110100010100000001000 10 0010010100000110111000101
246: 10110100010000101000000010111000 10 1011110010000000001100110
247: 01101001101111011011010010100011 1 0110001100111111110111011 1
248: 10010001101110110010011011001011 0 1001011000111100101101001
249: 11011110111111001001101010100111 10 1101101111011110011110111
250: 01011001010110111000011001011011 0 0101111110100110100110000
251: 10110100010000101000000010111000 10 1011110010000000001100110
252: 01100010000111011000110010010110 1 0100001001000111100000110
253: 11000000101101010011100010101001 0 1100001000011101010111100
254: 01100010000111011000110010010110 1 0100001001000111100000110
255: 10110100010000101000000010111000 10 1011110010000000001100110
256: 01100010000111011000110010010110 1 0100001001000111100000110
257: 11111010111111101000000110001111 0 1111110101011111010000100
258: 01100010000111011000110010010110 1 0100001001000111100000110

259: 110001001001001010000000100110111 1100010010010100110111110

313: 111110111110110110110010010000010 1111101101110011101100000
314: 101000111010000000101110011011010 1010000001111000100110111
315: 001100011110101100011101100111110 0011111000111010000101011
316: 010110010101101110000110010110110 0101111110100110100110000
317: 000100101100110110110110000110010 0001101001010011010100101
318: 110000101011101101011110011011010 1100011001001110110001001
319: 010000110001000011100010111101111 0100000001100100110001110
320: 000000000000000000000000000000000 00000000000000000100000110
321: 011111100001000101111100010100000 0111001111000100000011111
322: 000110001001101110001010111110110 0001001100010110110110111
323: 1101000000010010000101110110001110 1101000000001011101111001
324: 1100111001011000000001000111010101 1100100111001110100000010
325: 111110111110110110110010010000010 1111101101110011101100000
326: 101101000100001010000001011100010 1011110010000000001100110
327: 000110001001101110001010111110110 0001001100010110110110111
328: 110000110000010001101100101001010 1100000001100001010010100
329: 011000100001110110001100100101101 0100001001000111100000110
330: 010110010101101110000110010110110 0101111110100110100110000
331: 000110001001101110001010111110110 0001001100010110110110111
332: 000100101100110110110110000110010 0001101001010011010100101
333: 000110001001101110001010111110110 0001001100010110110110111
334: 1111101010000100100011101111101100 1111000101010001000010110
335: 010110010101101110000110010110110 0101111110100110100110000
336: 101011110001010001101110100011010 1010000101100101111010011
337: 001010000101101000101000000010010 0010010100000110111000101
338: 0010000011100100101000000010100110 0010100000011001010101011
339: 000000000000000000000000000000000 00000000000000000100000110
340: 000000011111000010101110011010000 00001000000111100000100010
341: 1101000000010010000101110110001110 1101000000001011101111001
342: 100100011011101100100110110010110 1001011000111100101101001
343: 00100011101001101010101010100011110 0011010001111001111100110
344: 001000100001100110101101100100010 0010001001100110101101000
345: 000010101100010110001000101001101 0100101101010001100000100
346: 100011001001100110000110101001111 1100001110010110101001110
347: 1101011111001110001110000000010010 1101110011110011001100100
348: 101000111010000000101110011011010 1010000001111000100110111
349: 0100001011100000011100100001000010 0100100001011000000011100
350: 100001100111001010001000101000010 1000110011001100101000010
351: 011110111100000000000111111101010 0111100101110000001010111
352: 0001101010001100000100000000011110 0001000101010011101101001
353: 011011111001001100011011000010110 0110011111110100001010110
354: 110101100010010010100101110110111 1101100011001001100100111
355: 110101111100100110100011000001000 1101101011110010110000110
356: 010101101001100010001101101000111 0001000011010110101100110
357: 000000000000000000000000000000000 00000000000000000100000110
358: 0001100100000010000000010001010100 0001000100100001010110001
359: 110100011000000000000101000100010 1101000100110000110110001
360: 111000010001001110010000111000101 1110011000100100100000110
361: 001111010101101110001110011000101 0011111010100110100000110
362: 000000000000000000000000000000000 00000000000000000100000110
363: 010111001000010111101110100001000 0101001110010001011000100
364: 000000000000000000000000000000000 00000000000000000100000110
365: 1101111111101000101001001000010100 1101101111111010001010111
366: 001111010101101110001110011000101 0011111010100110100000110

367: 100000110110100000101010110001101 100010000110101010100000110
368: 000110110001000100000001100000010 0001001101110100111101100
369: 010101010000010000100110001001110 0101000010100001000110101
370: 001000111010010001100100001000010 0010000001111001100101000
371: 111111111010100000001011000101100 1111000111111010000110100
372: 010011100100001010100001111010111 0100111111000000110011010
373: 011100110000111010100110000010000 0111010001100011001000000
374: 011100110000111010100110100000000 0111010001100011101000000
375: 101011010111001110000110001101000 1010111110101100000100001
376: 010000110001000011100010111101111 0100000001100100110001110
377: 0100000001011111011100001000011001 01000100000011111100000100
378: 000100001100000110000111000101111 0001101000010000101010100
379: 100110011100010111111000100111111 1001101100110001111101100
380: 011000110110111110111110010110100 0110111001101011001010111
381: 100011001001101101010010000001011 1000011110010110101011111
382: 000110001001101110001010111110110 0001001100010110110110111
383: 100001110011101000101010000001010 1000010011101110001110111
384: 001101001110110110000110000111000 0011101010011011100010010
385: 0000000110000000000001101100011100 0000000000110000011011000
386: 011100110000111010100110100000000 0111010001100011101000000
387: 010000110001000011100010111101111 0100000001100100110001110
388: 100100000000000000111010011101001 10010000000000000100000100
389: 110110111101101101100000101110101 1101111101110110100000010
390: 101001111100111010000110101110011 1010110011110011100011110
391: 110100011000000000000101000100010 1101000100110000110110001
392: 001111010101101110001110011000101 0011111010100110100000110
393: 001000111010011010101010100011110 0011010001111001111100110
394: 010111000101100111101100010110010 0100101110000110111010011
395: 110100011000000000000101000100010 1101000100110000110110001
396: 100110011100010110000100111000110 1001101100110001010101111
397: 001100010000010000010110110111110 00110000000100001010001000
398: 101001111100111010000110101110011 1010110011110011100011110
399: 101000111010000000101110011011010 1010000001111000100110111
400: 101111011110100100000011000110110 1011101110111010011101101

APÊNDICE C

```
esmeralda% soprano c432.test c432.bench
entre com a porcentagem :30
*******************************************************
*                                                     *
*       Welcome to soprano (version 1.1)              *
*                                                     *
*            Copyright (C) 1991,                      *
*  Virginia Polytechnic Institute & State University  *
*                                                     *
*******************************************************

******  SUMMARY OF TEST PATTERN GENERATION RESULTS  ******
1. Circuit structure
  Name of the circuit                      : c432
  Number of primary inputs                 : 36
  Number of primary outputs                : 7
  Number of gates                          : 160
  Depth of the circuit                     : 18

2. ATPG parameters
  Test pattern generation mode             : RPT + DTPG + TC
  Limit of random patterns (packets)       : 2
  Backtrack limit                          : 10
  Initial random number generator seed     : 846323997

3. Test pattern generation results
  Number of test patterns before compaction : 76
  Number of test patterns after compaction  : 62
  Fault coverage                           : 72.179 %
  Number of collapsed faults               : 514
  Number of identified redundant faults    : 1
  Number of aborted faults                 : 142
  Number of backtracking in FAN            : 67
4. CPU time
  Initialization                           : 0.083 secs
  Fault simulation                         : 0.233 secs
  FAN                                      : 0.150 secs
  Total                                    : 0.467 secs
esmeralda% fsim -t c432.test c432.bench
*******************************************************
*                                                     *
*       Welcome to fsim (version 1.1)                 *
*                                                     *
*            Copyright (C) 1991,                      *
*  Virginia Polytechnic Institute & State University  *
*                                                     *
*******************************************************

******     SUMMARY OF FAULT SIMULATION RESULTS     ******
1. Circuit structure
  Name of the circuit                      : c432
  Number of gates                          : 196
  Number of primary inputs                 : 36
  Number of primary outputs                : 7
```

```
   Depth of the circuit                 : 18

 2. Simulation parameters
   Simulation mode                      : file (c432.test)

 3. Simulation results
   Number of test patterns applied      : 62
   Fault coverage                       : 88.168 %
   Number of collapsed faults           : 524
   Number of detected faults            : 462
   Number of undetected faults          : 62

 4. Memory used                         : 356 Kbytes

 5. CPU time
   Initialization                       : 0.117 secs
   Fault simulation                     : 0.050 secs
   Total                                : 0.167 secs
esmeralda% soprano c432.test c432.bench
entre com a porcentagem :10
*******************************************************
*                                                     *
*        Welcome to soprano (version 1.1)             *
*                                                     *
*             Copyright (C) 1991,                     *
*  Virginia Polytechnic Institute & State University  *
*                                                     *
*******************************************************

****** SUMMARY OF TEST PATTERN GENERATION RESULTS ******
1. Circuit structure
   Name of the circuit                  : c432
   Number of primary inputs             : 36
   Number of primary outputs            : 7
   Number of gates                      : 160
   Depth of the circuit                 : 18

2. ATPG parameters
   Test pattern generation mode         : RPT + DTPG + TC
   Limit of random patterns (packets)   : 2
   Backtrack limit                      : 10
   Initial random number generator seed : 846324388

3. Test pattern generation results
   Number of test patterns before compaction : 26
   Number of test patterns after compaction  : 22
   Fault coverage                       : 42.802 %
   Number of collapsed faults           : 514
   Number of identified redundant faults : 1
   Number of aborted faults             : 293
   Number of backtracking in FAN        : 67

4. CPU time
   Initialization                       : 0.100 secs
```

```
  Fault simulation                 : 0.167 secs
  FAN                              : 0.133 secs
  Total                            : 0.400 secs
esmeralda% fsim -t c432.test c432.bench
*******************************************************
*                                                     *
*         Welcome to fsim (version 1.1)               *
*                                                     *
*            Copyright (C) 1991,                      *
*  Virginia Polytechnic Institute & State University  *
*                                                     *
*******************************************************

******    SUMMARY OF FAULT SIMULATION RESULTS    ******
1. Circuit structure
  Name of the circuit              : c432
  Number of gates                  : 196
  Number of primary inputs         : 36
  Number of primary outputs        : 7
  Depth of the circuit             : 18

2. Simulation parameters
  Simulation mode                  : file (c432.test)

3. Simulation results
  Number of test patterns applied  : 22
  Fault coverage                   : 70.611 %
  Number of collapsed faults       : 524
  Number of detected faults        : 370
  Number of undetected faults      : 154

4. Memory used                     : 356 Kbytes

5. CPU time
  Initialization                   : 0.083 secs
  Fault simulation                 : 0.050 secs
  Total                            : 0.133 secs
esmeralda% soprano c432.test c432.bench
entre com a porcentagem :5
*******************************************************
*                                                     *
*        Welcome to soprano (version 1.1)             *
*                                                     *
*            Copyright (C) 1991,                      *
*  Virginia Polytechnic Institute & State University  *
*                                                     *
*******************************************************

******  SUMMARY OF TEST PATTERN GENERATION RESULTS  ******
1. Circuit structure
  Name of the circuit              : c432
  Number of primary inputs         : 36
  Number of primary outputs        : 7
  Number of gates                  : 160
```

```
  Depth of the circuit                 : 18

2. ATPG parameters
  Test pattern generation mode             : RPT + DTPG + TC
  Limit of random patterns (packets)       : 2
  Backtrack limit                      : 10
  Initial random number generator seed     : 846324483

3. Test pattern generation results
  Number of test patterns before compaction : 12
  Number of test patterns after compaction  : 10
  Fault coverage                       : 25.292 %
  Number of collapsed faults               : 514
  Number of identified redundant faults     : 1
  Number of aborted faults                 : 383
  Number of backtracking in FAN            : 67

4. CPU time
  Initialization                      : 0.100 secs
  Fault simulation                      : 0.167 secs
  FAN                                : 0.150 secs
  Total                              : 0.417 secs
esmeralda% fsim -t c432.test c432.bench
*******************************************************
*                                        *
*         Welcome to fsim (version 1.1)           *
*                                        *
*            Copyright (C) 1991,              *
*  Virginia Polytechnic Institute & State University  *
*                                        *
*******************************************************

******     SUMMARY OF FAULT SIMULATION RESULTS     ******
1. Circuit structure
  Name of the circuit                  : c432
  Number of gates                      : 196
  Number of primary inputs                 : 36
  Number of primary outputs                : 7
  Depth of the circuit                 : 18

2. Simulation parameters
  Simulation mode                      : file (c432.test)

3. Simulation results
  Number of test patterns applied          : 10
  Fault coverage                      : 54.962 %
  Number of collapsed faults               : 524
  Number of detected faults                : 288
  Number of undetected faults              : 236

4. Memory used                         : 356 Kbytes

5. CPU time
  Initialization                      : 0.100 secs
```

APÊNDICE D

Cobertura de	Falhas (%)
SOPRANO	FSIM
28.29	66.75
24.65	64.90
28.15	66.75
24.51	62.66
28.71	65.04
22.40	59.63
31.09	71.63
27.73	63.72
21.70	67.94
20.16	57.12

Aplicação de 1% dos vetores de teste ao c499.isc.

Cobertura de	Falhas (%)
SOPRANO	FSIM
48.45	82.05
50.28	80.60
52.52	83.37
55.32	83.77
48.03	78.49
55.88	84.56
54.62	83.11
52.52	83.90
52.94	84.03
56.30	84.03

Aplicação de 3.5% dos vetores de teste ao c499.isc.

Cobertura de	Falhas (%)
SOPRANO	FSIM
72.54	91.16
75.07	91.55
74.65	91.95
73.24	92.48
74.23	93.40
74.65	93.47
75.35	92.48
75.49	91.42
73.95	91.54
73.52	92.87

Aplicação de 15% dos vetores de teste ao c499.isc.

Cobertura de	Falhas (%)
SOPRANO	FSIM
98.31	98.81
98.59	98.94
98.73	98.94
98.03	98.54
98.59	98.54
98.17	98.81
98.31	98.81
98.59	98.94
98.73	98.94
98.03	98.94

Aplicação de 100% dos vetores de teste ao c499.isc.

Cobertura de	Falhas (%)
SOPRANO	FSIM
27.52	52.76
24.33	47.90
29.28	54.71
25.50	50.97
22.38	47.10
29.67	55.55
25.13	47.98
23.95	47.73
28.22	53.42
30.16	50.91

Aplicação de 1% dos vetores de teste ao c5315.isc.

Cobertura de	Falhas (%)
SOPRANO	FSIM
50.69	75.45
49.38	72.84
54.46	79.53
51.27	78.99
53.13	78.44
54.81	78.37
54.74	79.57
48.91	75.04
55.28	79.34
51.65	76.71

Aplicação de 3.5% dos vetores de teste ao c5315.isc.

Cobertura de	Falhas (%)
SOPRANO	FSIM
75.01	94.01
74.18	91.85
75.21	92.63
74.07	92.35
73.51	90.71
73.68	91.70
71.79	91.06
70.28	89.73
72.31	89.55
72.98	91.04

Aplicação de 15% dos vetores de teste ao c5315.isc.

Cobertura de	Falhas (%)
SOPRANO	FSIM
99.08	98.89
99.08	98.89
99.08	98.89
99.08	98.89
99.08	98.89
99.08	98.89
99.08	98.89
99.08	98.89
99.08	98.89
99.08	98.89

Aplicação de 100% dos vetores de teste ao c5315.isc.

Cobertura de	Falhas (%)
SOPRANO	FSIM
25.29	54.96
29.76	57.06
28.21	51.90
30.15	61.26
26.65	55.34
26.45	54.38
23.15	52.67
28.79	45.03
27.43	59.54
26.07	51.14

Aplicação de 1% dos vetores de teste ao c432.isc.

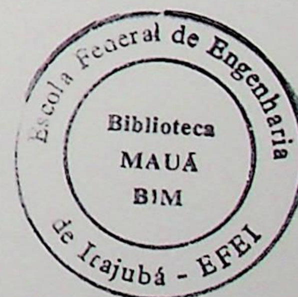
Cobertura de	Falhas (%)
SOPRANO	FSIM
51.55	79.38
51.36	79.38
51.94	76.90
57.19	79.58
49.80	76.52
52.52	85.87
54.47	81.10
50.00	77.09
53.50	81.48
48.83	71.56

Aplicação de 3.5% dos vetores de teste ao c432.isc.

Cobertura de	Falhas (%)
SOPRANO	FSIM
74.31	91.03
70.81	87.59
74.51	91.03
73.93	90.64
70.62	90.45
70.23	91.22
74.90	90.07
79.18	92.93
77.62	93.70
75.87	90.26

Aplicação de 15% dos vetores de teste ao c432.isc.

Cobertura de	Falhas (%)
SOPRANO	FSIM
97.08	99.23
97.47	99.23
96.49	99.23
96.88	99.23
96.30	99.23
97.27	99.23
96.69	99.23
96.10	99.23
97.59	99.23
96.81	99.23



Aplicação de 100% dos vetores de teste ao c432.isc.

EFEI / BIBLIOTECA

ESTE LIVRO DEVE SER DEVOLVIDO NA ÚLTIMA DATA CARIMBADA.

27/07/2004		

EFEI - BIBLIOTECA MAUÁ
8200930

NÃO DANIFIQUE ESTA ETIQUETA