UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA

H-ALG

Um Algoritmo Hierárquico para a Geração de Teste para Circuitos Combinacionais

Autor: José de Mendonça Furtado Neto

Este exemplar corresponde à redação final da tese defendida por José Mendonça Furtado Neto e aprovada pela Comissão Julgadora em 25 07 / 90.

Orientador

Membros da Banca:
Presidente: Prof. Dr. Mario Lúcio Côrtes
Membro: Prof. Dr. Furio Damiani
Membro: Prof. Dr. Ivanil Sebastião Bonatti
Suplente: Dr. José Roberto Amazonas

Tese apresentada à Faculdade de Engenharia como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica.

UNICAMP – Junho de 1990

Índice

1 Introdução 4
2 Fundamentos 5
2.1 Testes, Erros, Falhas e Modelos de Falhas 5
2.2 Falhas tipo fixa-em nas Portas Básicas 10
2.2.1 Portas AND e NAND 10
2.2.2 Portas OR e NOR 11
2.2.3 NOT 12
2.2.4 XOR 13
2.3 Métodos para a geração de teste 14
2.3.1 Diferença Booleana 14
2.3.2 Complexidade dos algoritmos 16
2.3.3 Algoritmos 18
2.3.4 Outros métodos 25
2.3.5 Hierárquicos 26
2.4 Projeto visando testabilidade 28
3 O algoritmo incremental: I-ALG 30
3.1 Descrição geral 30
3.2 Tabelas das células e bloco mestre 32
3.3 Biblioteca de primitiva 33
3.4 Operações básicas 34
3.4.1 Geração da tabela de teste $T[M_{k+1}]$ 35
3.4.2 Geração da tabela de propagação $P[M_{k+1}]$ 42
3.4.3 Procedimentos para a seleção de C_j 44
3.5 Simplificações 46
3.5.1 Otimização tipo-OU 46
3.5.2 Fanout look-ahead 47
3.5.3 "Freelines" e "Headlines" 48
3.6 Resultados 49
3.6.1 Descrição plana e semi-hierárquica 51
3.6.2 Ordem de processamento das células 52
3.6.3 Simplificação tipo "fanout look-ahead" 54
3.6.4 Conclusões 55
4 O algoritmo hierárquico H-ALG 56
4.1 Considerações iniciais 56

4.2 Descrição geral 57
4.3 Tabelas e biblioteca 58
4.4 Operações básicas 59
4.4.1 Geração da tabela de teste $T[M_{k+1}]$ 59
4.4.2 Geração da tabela de aplicação $A[M_{k+1}]$ 63
4.4.3 Procedimentos para a seleção de C_j 64
4.5 Simplificações 64
4.5.1 Corte de vetores 64
4.6 Resultados 65
4.6.1 Comparação I-ALG x H-ALG 66
4.6.2 ULA descrita com hierarquia 68
4.6.3 Corte de vetores 69
4.6.4 Circuitos propostos em [Brglez 85] 73
4.6.5 Resultados do H-ALG x HILO 73
5 Conclusões 77
6 Trabalho futuro 79
7 Agradecimentos 79
8 Bibliografia e referências 80
ANEXO 1 – Implementação do H-ALG 84

Figuras

Figura 1: Exemplo de falhas em um circuito MOS. 6
Figura 2: Relação entre diagramas elétrico e lógico. 7
Figura 3: Excitação e propagação de falha. 8
Figura 4: Exemplo de falhas equivalentes. 9
Figura 5: teste para o AND2. 11
Figura 6: teste para o NAND2. 11
Figura 7: teste para o OR2. 12
Figura 8: teste para o NOR2. 12
Figura 9: teste para o NOT. 12
Figura 10: XOR implementado com portas NAND e NOT. 13
Figura 11: Tabela de falhas (a) e de teste do XOR (b). 13
Figura 12: Circuito com redundância 15
Figura 13: Circuito com derivação reconvergente. 20
Figura 14: Árvore de procura para o circuito da fig. 13 21
Figura 15: Circuito com falha redundante. 22
Figura 16: Grafo de procura para o circuito da figura 15. 23
Figura 17: Circuito exemplo (a) e grafo de procura do FAN (b). 24
Figura 18: Derivação reconvergente no centro do circuito. 25
Figura 19: Tabela comparativa entre alguns métodos e o H-ALG. 28
Figura 20: Exemplo de circuito usando LSSD. 29
Figura 21: Iterações do algoritmo. 30
Figura 22: Interface célula-bloco mestre. 31
Figura 23: Exemplo das tabelas da primitiva AND3. 34
Figura 24: "D-intersection" [Roth 66]. 35
Figura 25: Propagação de C_j através de M_k. 36
Figura 26: Aplicação de M_k via C_j. 36
Figura 27: Circuito exemplo para as operações do I-ALG. 37
Figura 28: Operação de propagação. 37
Figura 29: Propagação de cada um dos vetores. 38
Figura 30: Tabela de propagação $T_p[C_j]$. 38
Figura 31: Operação Aplicação. 39
Figura 32: Aplicação de cada um dos vetores. 40
Figura 33: Tabela de aplicação $T_A[M_k]$. 41
Figura 34: Tabela de teste de M_{k+1} ($T[M_{k+1}]$). 41

Figura 35: Combinação de $P[M_k]$ e $F_e[C_j]$. 42
Figura 36: O vetor 14 de $P[M_{k+1}]$. 43
Figura 37: Tabela $P[M_{k+1}]$. 44
Figura 38: Bloco mestre e suas saídas. 44
Figura 39: Simplificação de uma tabela de propagação. 46
Figura 40: Simplificação de uma tabela de teste. 47
Figura 41: Circuito passivel de simplificação por "fanout look-ahead". 47
Figura 42: Exemplo de simplificação por "fanout look-ahead". 48
Figura 43: Exemplo de "headline" e "freelines". 48
Figura 44: Unidade Lógica e Aritmética (ULA) - 74181. 50
Figura 45: Modificação que torna ULA 100% testável. 51
Figura 46: Comparação ULA plana (A) x semi-hierárquica (B) 52
Figura 47: Ordem de processamento das células (curva C, fig 48). 53
Figura 48: Comparação ULA semi-hierárquica com ordens de processamento diferentes 53
Figura 49: Situação da interface circuito - bloco mestre após a sétima iteração .54
Figura 50: Uso da simplificação tipo "fanout look-ahead" na ULA 55
Figura 51: Descrição do algoritmo. 57
Figura 52: Interface célula-bloco mestre. 60
Figura 53: Exemplo de fusão. 60
Figura 54: Tabelas de teste e aplicação da OR2. 61
Figura 55: Fusão de G_3 com G_5 e G_6. 61
Figura 56: Fusão de G_2 com G_4. 61
Figura 57: Aplicação dos grupos não fundidos. 62
Figura 58: Tabela de teste final ($T[M_{k+1}]$). 62
Figura 59: Exemplo de geração da tabela de aplicação ($A[M_{k+1}]$) 63
Figura 60: Unidade Lógica Aritmética - ULA (74181). 66
Figura 61: Comparação de desempenho entre os protótipos do H-ALG (A) e I-ALG (B). 67
Figura 62: Ordem de processamento das portas para a ULA hierárquica. 68
Figura 63: Comparação da ULA "flat" (A) e hierárquica (B). 69
Figura 64: Comparação entre ULA hierárquica com cortes (A) e sem cortes (B). 70
Figura 65: Número de grupos com a mesma quantidade de vetores a cada iteração (ULA hierárquica, sem cortes). 71
Figura 66: Número de grupos e vetores separados por faixas (ULA hierárquica, sem cortes) 72
Figura 67: Teste gerado pelo HILO 74
Figura 68: Teste gerado pelo H-ALG 75
Figura 69: Teste gerado pelo H-ALG (continuação) 76
Figura 70: Teste gerado pelo H-ALG (final) 77

Figura 71: Representação do interrelacionamento das estruturas de dados 84
Figura 72: Listagem das estruturas usadas no H-ALG 87

Para Olívia e Rafael

SUMÁRIO

Este trabalho propõe um novo algoritmo para a geração de padrão de teste para circuitos digitais combinacionais, descritos de forma hierárquica, o **H-ALG**. O algoritmo não faz uso de "backtracking" (retrocesso automático [Wagner 88]) e detecta todas as falhas detectáveis. Baseia-se na idéia de se resolver o teste para pequenas células e depois combiná-las de uma forma incremental [Côrtes 88]. As células podem ser portas lógicas simples, portas complexas, redes de transistores de passagem, PLA's, ROM's, ou qualquer outro tipo de lógica combinacional. Não se limita a tratar falhas do tipo "stuck-at" e não requer simulação de falhas.

O algoritmo admite um número ilimitado de níveis hierárquicos, sendo que o esforço para a geração do teste de uma determinada célula só é aplicado uma vez. Nas demais instâncias desta mesma célula o teste é apenas chamado da biblioteca, onde fora armazenado.

O algoritmo foi implementado em linguagem "C". Neste trabalho também são apresentados alguns resultados obtidos a partir desta versão.

1 Introdução

Os avanços alcançados no desenvolvimento da tecnologia de processamento de circuitos digitais (VLSI) têm causado um forte impacto na área de testes. Com o aumento da densidade dos circuitos e o cada vez mais difícil acesso aos seus nós, o custo do teste vem se tornando parte substancial do custo do produto. Para manter este valor dentro de níveis aceitáveis, foram desenvolvidas diversas técnicas de projeto visando a testabilidade ("design for testability") dos circuitos. O recurso usado por estes métodos é tornar, do ponto de vista do teste, circuitos sequenciais em combinacionais. permitindo, desta forma o uso de alguns métodos e algoritmos de geração de padrões de teste para circuitos combinacionais [Roth 66], [Goel 81], [Fujiwara 83], entre outros. Embora muito progresso venha sendo obtido, o esforço computacional para este tipo de atividade continua muito grande. A busca por soluções cada vez mais eficientes, prossegue.

Um meio natural de lidar com a crescente complexidade dos circuitos, é adotar técnicas que permitam o particionamento das tarefas. Métodos hierárquicos de projeto vêm sendo aperfeiçoados e seu uso é cada vez mais frequente. Técnicas como o "gate array" e o "standard cells", baseados em bibliotecas de células mais e mais sofisticadas, confirmam este direcionamento. Como resultado do uso destas técnicas, são produzidas descrições hierárquicas dos circuitos, que apresentam algumas vantagens, como: uso de menor área de memória; descrições mais facilmente inteligíveis; etc. Uma vez de posse de uma descrição deste tipo, é absolutamente indesejável e contraproducente que se tenha que planificá-la ("flatten out") antes de gerar os padrões de teste. Bastante esforço vem sendo empregado na busca de soluções que evitem este desperdício: [Breuer 85], [Brahme 87], [Chandra 87], [Krisnamurthy 87], [Murray 88], [Schulz 88], [Bhattacharya 89], [Calhoun 89], [Sarfert 89], por exemplo.

Neste trabalho são apresentados dois novos algoritmos para a geração de teste para circuitos combinacionais. O I-ALG [Côrtes 88] (Algoritmo Incremental) e o H-ALG [Mendonça 89] (Algoritmo Hierárquico), ambos foram desenvolvidos como parte deste trabalho.

O algoritmo I-ALG percorre o circuito desde suas saídas em direção às entradas, processando célula a célula as informações de teste de cada uma delas. Estas informações estão armazenadas em uma biblioteca pré-construida e são totalmente independentes da tecnologia a ser usada na fabricação do circuito. Admite qualquer modelamento de falha, uma vez que os testes podem ser definidos como se deseje.

Diferentemente dos principais algoritmos para geração de padrões de teste publicados ([Roth 66], [Goel 81] e [Fujiwara 83]), o I-ALG não é orientado pelas falhas a serem detectadas. Pode-se dizer que ele é dirigido pela conectividade do circuito. Por ser uma solução completamente diferente das citadas, o I-ALG não esbarra no maior problema com que se deparam aqueles algoritmos: o "backtracking". Naturalmente o algoritmo ora proposto apresenta algumas

limitações, principalmente no que diz respeito a memória necessária para o tratamento das tabelas internas. Neste trabalho são propostas diversas técnicas que visam minimizar este problema.

O algoritmo hierárquico, H-ALG [Mendonça 89] é baseado no I-ALG, tendo com principal característica sua capacidade de lidar com circuitos descritos com hierarquia. Trata-se de uma proposta original que não tem as limitações quanto ao número de níveis de hierarquia, apresentadas pelos trabalhos citados acima. O H-ALG admite qualquer quantidade de níveis hierárquicos. Enquanto tratando um circuito, ao se deparar com uma célula não definida na sua biblioteca, o algoritmo se chama recursivamente e resolve o teste desta célula, a partir da descrição de sua conectividade. Uma vez determinado o teste da célula, o controle retorna ao ponto de onde foi chamada a recursão, e o algoritmo prossegue tratando o circuito no nível hierárquico anterior.

A partir de medidas feitas em simulações realizadas com o protótipo desenvolvido, é possível concluir que embora ainda haja muito a se desenvolver na linha proposta, desde já o caminho se mostra promissor.

2 Fundamentos

Nesta seção são tratados alguns itens básicos que irão auxiliar no entendimento deste trabalho. Em 2.1 são revistas definições básicas com o objetivo de fazer com que os termos usados no restante do texto sejam perfeitamente compreendidos. As falhas na portas básicas são tratadas em 2.2. A seguir, em 2.3 é feito um estudo dos mais conhecidos métodos computacionais publicados na literatura. Finalmente em 2.4 são estudados alguns métodos de orientação ao projeto visando a testabilidade do circuito.

2.1 Testes, Erros, Falhas e Modelos de Falhas

Testar um circuito consiste em aplicar a suas entradas uma sequência de sinais e observar nas suas saídas as respostas obtidas, comparando-as com as respostas esperadas para o circuito correto. Na presença de qualquer discrepância na saída, diz-se que ocorreu um erro. O erro resulta de uma falha física no circuito. A geração de teste para um circuito pode visar atender dois diferentes objetivos:

1. Teste de protótipo – O objetivo deste teste é detectar a existência da falha, localizá-la no circuito e diagnosticar a causa, de modo a facilitar sua correção.

2. Teste de produção ("go-no-go") – Neste, basta detectar a ocorrência de um erro para que a amostra seja considerada ruim. Este teste é. normalmente, um sub-conjunto do anterior.

Para se gerar o teste de um circuito é conveniente se fazer uso de modelos lógicos das falhas (modelo de falhas). Tais modelos refletem mudanças no comportamento lógico do circuito, devido a presença de defeitos físicos.

Um determinado modelo de falha, usado na geração de testes é simplesmente uma abstração conveniente que reflete no nível lógico as falhas físicas que realmente ocorrem no circuito. Deve-se notar que as falhas consideradas pelo modelo de falha podem não corresponder univocamente a falhas reais. Por outro lado, o modelo deve ser tal que garanta, num certo nível de confiança, que a detecção das falhas nele modeladas implica na detecção da maioria das falhas físicas reais. A maior parte das falhas físicas de um componente pode ser representada por entradas ou saídas de portas lógicas simples (AND, OR, NOT, etc) fixa-em-zero (**f-e-0**) ou fixa-em-um (**f-e-1**), ("stuck-at") [Wadsack 78] e [Agrawal 88]. Entretanto, nem todas as falhas podem ser modeladas desta forma. A figura 1 apresenta o diagrama elétrico de uma porta MOS onde estão assinalados dois possíveis "curtos" (1 e 2) e dois possíveis "abertos" (3 e 4). O curto 1 e o aberto 3 podem ser modelados respectivamente como o fio E **f-e-1** e como o fio E ou o fio F ou ambos, **f-e-0**. Por outro lado, o curto 2 e o aberto 4 não podem ser modelados desta forma já que na realidade provocam uma modificação na função exercida pela porta.

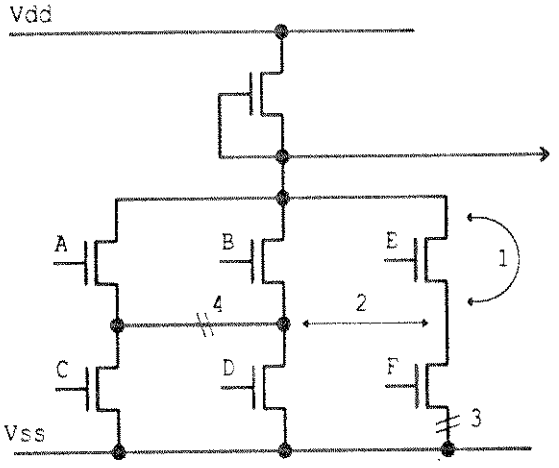

Figura 1: Exemplo de falhas em um circuito MOS.

Outro fator a ser considerado é o nível de representação do circuito, acompanhe pelo exemplo da figura 2, onde o mesmo circuito está representado pelos seus diagramas elétrico e lógico. Em cada qual estão assinaladas as falhas que não são possíveis de serem representadas ou mesmo ocorrerem, no outro. Por exemplo, o curto 2 que é fisicamente possível, não pode ser representado no diagrama lógico e o curto 1 no diagrama lógico não tem sentido físico.

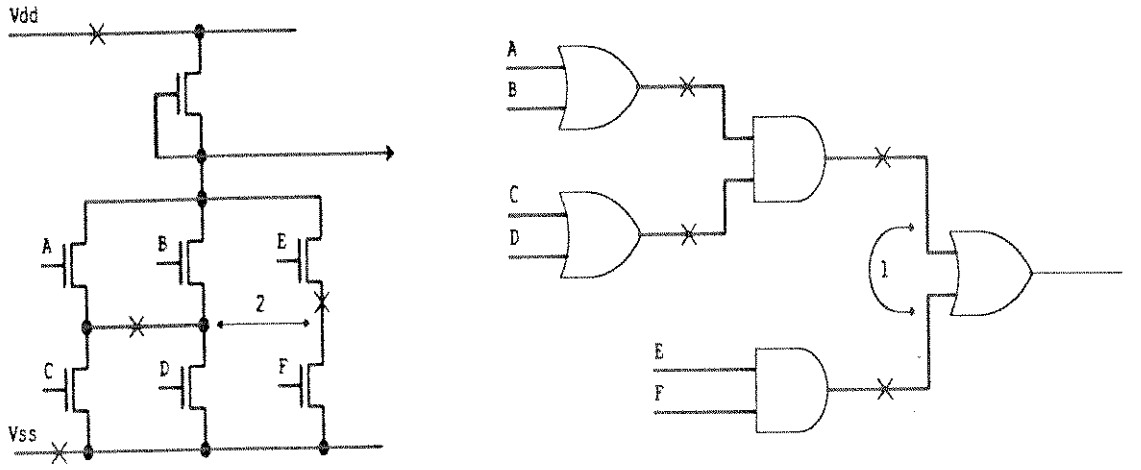

Figura 2: Relação entre diagramas elétrico e lógico.

A disseminação do uso das tecnologias MOS (especialmente CMOS) e a redução das dimensões mínimas dos dispositivos, fizeram com que se aprofundasse o estudo ([Agrawal 88] e [Tsui 87] trazem ampla lista de referências) sobre alguns tipos de mecanismos de falha não importantes nas tecnologias usadas anteriormente (TTL e ECL entre outras). Defeitos na estrutura cristalina do substrato, defeitos no óxido, etc. Estes problemas exigiram novos modelamentos que possibilitassem sua representação no nível lógico. Apesar de bastante questionado, o modelo de falha fixa-em continua sendo amplamente utilizado, como será visto adiante.

Supondo que o modelo adotado seja o de falha fixa-em, vamos verificar o grau de complexidade do problema de se levantar as possíveis falhas de um ciruito. Consideremos um circuito contendo **k** fios. Cada um deles, em um determinado instante pode estar em uma das três condições:

- sem falha
- com falha **f-e-0**
- com falha **f-e-1**

Se considerarmos todas as possíveis combinações destes **k** fios, podendo estar cada um em uma das três condições, teremos 3^k possíveis circuitos. Destas, apenas uma corresponde ao circuito correto, sem falha. Este modelo, que admite a ocorrência de mais de uma falha simultânea no circuito, é denominada modelo de falhas múltiplas. Por ser impraticável gerar testes para todas as imagináveis condições em que o circuito apresentar falha, é que se adotou o conceito de falha simples. Assim, ao se procurar um teste assume-se que existe **uma e apenas uma** falha no circuito.

Um modelo de falha confiável e aceito universalmente é o que trata falha simples tipo fixa-em ("single stuck-at") para circuitos descritos a nível de porta. Na prática, a detecção deste tipo de falha acaba por detectar um grande número das possíveis falhas múltiplas [Hughes 86]. Este modelo além de ter um nível de confiança adquirido com o seu emprego através de anos [Krisnamurthy 87] permite a enumeração das falhas. Para uma determinada porta lógica de **n**-entradas é possível assinalar um número específico de falhas. Mais ainda, é possível definir claramente as falhas e seus efeitos no comportamento da porta. Isso permite que se implemente algoritmos computacionais para o cálculo do teste, que tratem o circuito descrito no nível lógico.

As falhas físicas são modeladas mantendo-se um único fio do circuito em um valor constante (0 ou 1) independente de como o circuito em operação estimula este fio. Para testar se um determinado fio está **f-e-0** (**f-e-1**), é necessário encontrar pelo menos uma combinação das entradas primárias (terminais de entrada fisicamente acessíveis) que faça com que aquele fio seja igual a 1 (0). Este passo é definido como o de excitar (ou provocar) a falha. A combinação de sinais nas entradas (vetor de teste) também deve fazer com que a falha se propague até uma saída primária (terminal de saída fisicamente acessível), onde possa ser observada, vide figura 3. Este procedimento é definido como propagação da falha. Uma falha nestas condições é dita detectável. Falhas não detectáveis são aquelas para as quais não exista vetor que realize a excitação e a propagação simultaneamente. Em um circuito combinacional, falhas não detectáveis são resultantes de redundâncias.

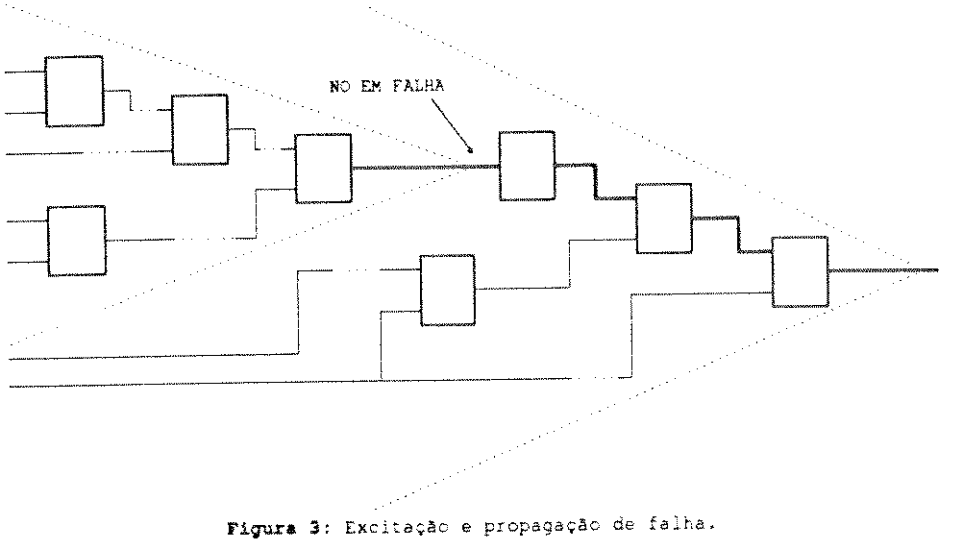

Figura 3: Excitação e propagação de falha.

Se for considerado um circuito com **k** fios e o modelo de falha simples tipo fixa-em, o número de falhas possíveis de ocorrer em todo o circuito (conjunto de falhas) é **2k**. Define-se cobertura de falhas como a fração das possíveis falhas consideradas pelo modelo usado, que é detectada pelo teste. A experiência mostra que circuitos testados para uma alta cobertura de falhas do tipo fixa-em (99%, ou mais), apresentam uma alta confiabilidade no campo [Bhattacharya 89] e [Williams 81].

Ao se realizar o levantamento das possíveis falhas (lista de falhas) em um circuito, poderá ser notado que não é possível distinguir certas falhas de determinadas outras. Se for tentado gerar duas tabelas verdade para o circuito da figura 4, sendo uma que descreva o comportamento do circuito quando o fio **E** estiver em falha (f-e-0, por exemplo) e outra que descreva o comportamento quando o fio **A** estiver **f-e-0**, poderá ser notado que estas tabelas são idênticas. Em casos semelhantes a este, nos quais não é possível distinguir uma falha da outra, diz-se que as falhas são equivalentes.

Mais precisamente: em um circuito combinacional que executa a função **f**, o conjunto de testes que detectam a falha **a** é definido pela equação[1] $T_a = f \oplus f_a$ ([Breuer 76]), e o conjunto de testes que detectam a falha **b** é definido por $T_b = f \oplus f_b$. O conjunto dos testes que permitem distinguir **a** de **b** é definido por $f_a \oplus f_b$. Se $f_a = f_b$, não existe teste que distingua **a** de **b** e estas falhas são ditas equivalentes.

Outro conceito importante e que ajuda a limitar o número de falhas a serem consideradas na geração de testes, é o de falhas dominantes. Suponha uma porta AND para a qual se deseja testar uma das entradas para falhas tipo **f-e-1**. Sempre que isso for feito a saída da porta também estará sendo testada para **f-e-1**. O inverso não é verdade; testar a saída para **f-e-1** não necessariamente testa qualquer das entradas para **f-e-1**. Diz-se que a falha **f-e-1** na saída domina as falhas **f-e-1** nas entradas. De forma geral, uma falha **a** domina uma falha **b** se $T_b \subseteq T_a$. Pode-se concluir, a partir desta definição, que se uma falha **a** domina uma falha **b**, então qualquer teste que detecte **b**, detecta também **a**.

Pelo óbvio motivo de que o tempo de processamento dos programas geradores de teste é diretamente afetado pelo número de falhas a serem tratadas, é conveniente a redução da lista de falhas através do uso dos conceitos de equivalência e dominância. O processo que faz uso destes conceitos visando reduzir a lista de falhas é chamado de "fault collapsing" ([Breuer 76], [Miczo 86]). Nos itens 2.2.1 e 2.2.2 este processo é detalhado.

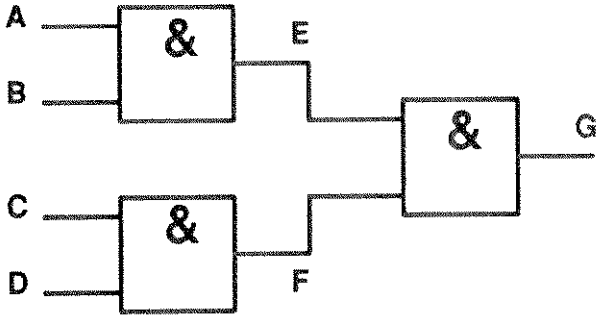

Figura 4: Exemplo de falhas equivalentes.

1. Onde ⊕ é o operador "ou-exclusivo".

2.2 Falhas tipo fixa-em nas Portas Básicas

A seguir são listados os vetores de teste para as portas AND, OR, NOT (inversor) e XOR (OU-exclusivo). O conjunto dos vetores de teste de uma determinanda porta é semelhante ao conjunto dos "primitive D-cube of a fault" ([Breuer 76]) de todas as falhas da portas em questão. Os conjuntos de vetores para as demais portas básicas podem ser deduzidos a partir destes, e são apenas apresentados.

2.2.1 Portas AND e NAND

Para qualquer porta de n-entradas pode-se enumerar 2(n+1) falhas simples do tipo **f-e**. Em uma porta AND todas as falhas tipo **f-e-0** são equivalentes. Não há teste capaz de distinguir qual dos terminais está em falha. Da mesma forma, para a porta NAND o conjunto das falhas **f-e-0** nas entradas e **f-e-1** na saída também são equivalentes. Assim, apenas n+2 falhas precisam ser consideradas para uma porta de n-entradas, a saber: tipo **f-e-1** nas entradas, tipo **f-e-0** e **f-e-1** na saída. O processo que leva a esta redução é chamado "equivalence fault collapsing", [Breuer 76].

Mais uma redução é possível se for considerado o conceito de dominância. Em uma porta AND a falha **f-e-1** na saída domina qualquer falha **f-e-1** nas entradas. Ou seja, um teste que detecte **f-e-1** em qualquer entrada também detecta **f-e-1** na saída. De modo similar para a porta NAND onde a falha **f-e-0** na saída domina qualquer falha **f-e-1** nas entradas. Este processo é chamado de "dominance fault collapsing" ([Breuer 76]). Conclui-se portanto que se as entradas destas portas forem testadas para falha tipo **f-e-1**, suas saídas terão sido testadas para **f-e-1**, no caso do AND e **f-e-0**, no caso do NAND. Portanto, não é preciso testar explicitamente as falhas nas saídas, a menos que se deseje separá-las das demais (interessante quando se deseja localizar a falha, vide seção 2.1). O "fault collapsing" reduz para n+1 o conjunto de falhas a testar.

Uma porta AND é modelada para falhas simples tipo **f-e-1** nas entradas e **f-e-0** na saída. O teste para uma das entradas **f-e-1** consiste em se aplicar um nível **0** na entrada em teste e nível **1** nas demais. Desta forma a entrada sob teste controla a porta. Ou seja, ela sozinha determina o valor da saída. Se não há falha, a saída é **0**, e na presença de falha a saída vai para **1**.

Um teste no qual em todas as entradas é aplicado nível **1**, testará a saída para falha tipo **f-e-0**. Este teste detecta ainda falhas tipo **f-e-0** nas entradas sendo, entretanto, impossível explicitar qual delas está em falha.

A convenção usada para a listagem das falhas detectadas é A/1 e A/0, quando o fio A apresenta falha **f-e-1** ou **f-e-0**, respectivamente.

A	B	saída		falhas detectadas
		correta	c/falha	
0	1	0	1	A/1, C/1
1	0	0	1	B/1, C/1
1	1	1	0	A/0, B/0, C/0

Figura 5: teste para o AND2.

De modo similar pode-se deduzir os testes necessários para se detectar as falhas tipo **fixa-em** nas portas NAND.

A	B	saída		falhas detectadas
		correta	c/falha	
0	1	1	0	A/1, C/0
1	0	1	0	B/1, C/0
1	1	0	1	A/0, B/0, C/1

Figura 6: teste para o NAND2.

2.2.2 Portas OR e NOR

Em uma porta OR todas as falhas tipo **f-e-1** são equivalentes. Não há teste capaz de distinguir qual dos terminais está em falha. Da mesma forma, para a porta NOR o conjunto das falhas **f-e-1** nas entradas e **f-e-0** na saída também são equivalentes. Assim, apenas n+2 falhas precisam ser consideradas para uma porta de n-entradas, a saber: tipo **f-e-0** nas entradas, tipo **f-e-0** e **f-e-1** na saída. Como nas portas AND e NAND, também aqui esta redução é feita através do "equivalence fault collapsing".

Mais uma redução é possível se for considerado o conceito de dominância. Em uma porta OR a falha **f-e-0** na saída domina qualquer falha **f-e-0** nas entradas. Ou seja, um teste que detecte **f-e-0** em qualquer entrada também detecta **f-e-0** na saída. De modo similar para a porta NOR onde a falha **f-e-1** na saída domina qualquer falha **f-e-0** nas entradas. Este processo é chamado de "dominance fault collapsing". Conclui-se portanto que se as entradas destas portas forem testadas para falha tipo **f-e-0**, suas saídas terão sido testadas para **f-e-0**, no caso do OR e **f-e-1**, no caso do NOR. Portanto, não é preciso testar explicitamente as falhas nas saídas, a menos que se deseje separá-las das demais (interessante quando se deseja localizar a falha, vide seção 2.1). O "fault collapsing" reduz para n+1 o conjunto de falhas a testar.

Uma porta OR é modelada para falhas simples tipo **f-e-0** nas entradas, e falha tipo **f-e-1** na saída. No caso das entradas, a que está sob teste é levada a nível **1** e as demais a nível **0**. A entrada

sob teste controla a porta que no caso de não haver falha apresenta a saída em **1** , e no caso de falha a saída será **0**. Estes testes também detectam falha tipo **f-e-0** na saída. Um teste que aplique **0** em todas as entradas detecta falhas tipo **f-e-1** na saída e em todas as entradas, sem que seja possível explicitar qual das entradas está em falha.

A	B	saída		falhas detectadas
		correta	c/falha	
0	1	1	0	B/0, C/0
1	0	1	0	A/0, C/0
0	0	0	1	A/1, B/1, C/1

Figura 7: teste para o OR2.

De modo similar pode-se deduzir os testes necessários para se detectar as falhas tipo **fixa-em** nas portas NOR:

A	B	saída		falhas detectadas
		correta	c/falha	
0	1	0	1	B/0, C/1
1	0	0	1	A/0, C/1
0	0	1	0	A/1, B/1, C/0

Figura 8: teste para o NOR2.

2.2.3 NOT

Qualquer falha na entrada de um NOT é equivalente à falha de polaridade invertida na sua saída. Não há teste que distingua uma falha da outra. Por outro lado as falhas nas entradas dominam as (e são dominadas pelas) falhas de polaridade inversa na saída. Para testar o NOT basta, portanto, testar sua saída para falhas tipo **f-e-0** e **f-e-1**.

A	saída		falhas detectadas
	correta	c/falha	
0	1	0	A/1, C/0
1	0	1	A/0, C/1

Figura 9: teste para o NOT.

2.2.4 XOR

A porta XOR embora seja tratada como primitiva, é na realidade um circuito composto por portas básicas interligadas (fig. 10). Existem vários tipos de implementação desta função, inclusive alguns dependentes da tecnologia usada na fabricação do circuito. O comum entre todas as implementações é a existência de "fanout" interno à célula.

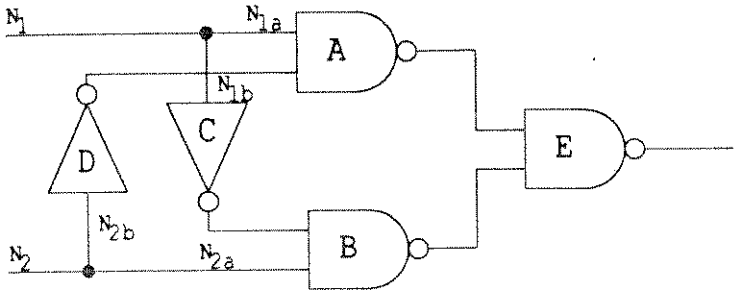

Figura 10: XOR implementado com portas NAND e NOT.

O ou-exclusivo é uma função não "unate" ([Miczo 86]) pois as duas variáveis de entrada aparecem em sua forma verdadeira e negada. Neste tipo de função não existe relação de dominância de falha na saída em relação a falha nas entradas. Como pode ser visto na sua tabela de teste, também não há relação de equivalência entre suas possíveis falhas.

Na figura 11 pode-se ver a tabela dos fios do circuito da fig. 10, contra as quatro possíveis combinações de sinais (vetores) na entrada do circuito. A tabela mostra as falhas que um determinado vetor detecta (1 = **f-e-1** e 0 = **f-e-0**). Embora os vetores **11**, **01** e **10** sejam suficientes para testar os terminais da porta, note que todos os quatro vetores são essenciais, ou seja, para se detectar todas as falhas, inclusive as internas, no XOR é necessário aplicar as quatro combinações em suas entradas.

vetores	fios								
	N_{1a}	N_{1b}	N_{2a}	N_{2b}	A	B	C	D	E
00	1		1		0	0			1
01		1	0			1	0		0
10	0			1	1			0	0
11		0		0	0	0	1	1	1

A	B	saída		falhas detectadas
		correta	c/falha	
0	0	0	1	A/1, B/1, C/1
0	1	1	0	A/1, B/0, C/0
1	0	1	0	A/0, B/1, C/0
1	1	0	1	A/0, B/0, C/1

Figura 11: Tabela de falhas (a) e de teste do XOR (b).

Embora o ou-exclusivo já esteja definido na biblioteca do H-ALG e aqui o tratemos como uma porta básica, nada impede que o usuário defina em seu circuito uma qualquer outra implementação. Tanto com hierarquia como sem, e deixe de usá-la a partir da biblioteca.

2.3 Métodos para a geração de teste

Datam do final da década de 50 as primeiras propostas de métodos computacionais para auxílio à geração de teste para circuitos digitais. O método da diferença booleana é uma destas propostas e embora seja pouco eficiente, é aplicável ao estudo em pequenos circuitos. No item 2.3.1 é feita uma revisão deste método.

Um grande desafio para os projetistas de circuitos de muito alta escala (VLSI) é o gerenciamento da sua crescente complexidade. Um método natural para lidar com este problema é adotar uma técnica que permita o máximo de ordem em cada fase do projeto, além de particionar o trabalho de uma forma hierárquica. Tais metodologias permitem que o projetista mantenha um completo domínio do problema no nível em que está sendo tratado. Existem diversos estudos publicados sobre o tema: [Mead 80], [Ayres 80], [Trimberger 81], [Treleaven 82], entre outros.

Outro grande desafio, na medida em que a complexidade dos circuitos cresce é o seu forte impacto sobre a geração de teste. Metodologias de projeto visando a testabilidade do circuito (vide seção 2.4) passaram a ser estudadas, chegando-se mesmo ao ponto de alterarem por completo o antigo **jeito** de se projetar circuitos. Hoje é ponto pacífico que alguma, ou mesmo algumas, destas metodologias deve ser adotada em qualquer projeto de porte. As que se tornaram mais usadas estão publicadas em [Eichelberger 77], [Williams 80], [Daehn 81], [Fujiwara 81] e são estudadas em 2.4

2.3.1 Diferença Booleana

Qualquer que seja o circuito combinacional, pode-se considerar que cada uma de suas saídas executa uma função lógica a partir das variáveis de entrada do circuito. Seja por exemplo, **f** (x_1, x_2, ..., x_n) uma destas funções. A diferença booleana de **f** com respeito a uma das entradas [2] (x_i, por exemplo), é definida como:

$$\frac{df(x)}{dx_i} = f(x_1, ..., x_i, ..., x_n) \oplus f(x_1, ..., \overline{x_i}, ..., x_n)$$

Como se trata de lógica binária, pode-se representar também:

$$\frac{df(x)}{dx_i} = f(x_1, ..., 1, ..., x_n) \oplus f(x_1, ..., 0, ..., x_n)$$

Ou ainda:

2. A diferença booleana também pode ser definida com relação a um nó interno qualquer.

$$\frac{df(x)}{dx_i} = f_i(1) \oplus f_i(0)$$

A diferença booleana representa todas as condições para as quais o valor da função **f** é sensível a mudanças no valor da variável x_i. Desta forma pode-se definir a função $T_{i/0}$ que represente o conjunto de todos os testes para a falha x_i **f-e-0**.

$$T_{i/0} = x_i \cdot \frac{df(x)}{dx_i} \tag{1}$$

Uma vez que $T_{i/0}$ também é sensível a x_i, uma alteração (falha) no valor de x_i fará com que o valor de $T_{i/0}$ também se altere e possa ser observado na saída do circuito. Da mesma forma o conjunto de testes que detecta a falha x_i **f-e-1** é definido por:

$$T_{i/1} = \overline{x_i} \cdot \frac{df(x)}{dx_i} \tag{2}$$

Este método gera todos os testes que detectam uma determinada falha, sendo porém limitado às falhas do tipo fixa-em. Sua grande desvantagem está na dificuldade de manipular as equações de circuitos grandes. O esforço computacional (tempo de máquina e demanda de memória) torna o método impraticável, porém tem utilidade para análise manual de pequenos circuitos.

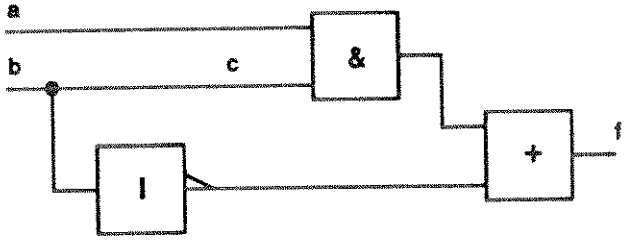

Figura 12: Circuito com redundância

Seja por exemplo, o circuito da figura 12, onde: $f = \overline{b} + ab$. Para derivar o teste de **a** f-e-1, basta usar a equação 2:

$$T_{a/1} = \overline{a} \cdot \frac{df}{da}$$

$$\frac{df}{da} = f_a(1) \oplus f_a(0) = 1 \oplus \overline{b} = b$$

$$T_{a/1} = \overline{a}b$$

Assim o teste para a falha **a** f-e-1 é: <ab> = <01>. Neste caso, apenas um vetor satisfaz a equação mas em caso geral, um conjunto de vetores pode satifazê-la, indicando que várias combinações de sinais nas entradas detectam esta mesma falha.

Da mesma forma, para derivar o teste para o nó interno, **c** f-e-1:

$$T_{c/1} = \overline{c} \cdot \frac{df}{dc}$$

$$\frac{df}{dc} = f_c(1) \oplus f_c(0) = (a + \overline{b}) \oplus \overline{b} = ab$$

$$T_{c/1} = \overline{b} \cdot ab = 0$$

Este resultado mostra que não existe teste para a falha **c** f-e-1, que é, portanto, não detectável. Isto ocorre devido ao fato do circuito ser redundante, pois: $f = \overline{b} + ab = \overline{b} + a$.

Sempre que há redundância no circuito, este pode ser simplificado. No exemplo, pode-se: desconectar o nó **c** e fazer a entrada da porta AND igual a 1, que é equivalente a eliminar a porta AND.

2.3.2 Complexidade dos algoritmos

Ibarra e Sahni [Ibarra 75] em um estudo teórico mostraram que a geração de teste para circuitos combinacionais pertence á classe dos problemas chamados de **NP-completos** (NP, para: Não-Polinomial). Isto significa que não é possível existir um algoritmo para a geração de teste cuja

complexidade seja polinomial com relação ao tempo. A complexidade não-polinomial (exponencial) neste caso se refere ao esforço necessário, no pior caso, para a geração de teste de um circuito. Na prática, os algoritmos conhecidos conseguem resultados que apresentam uma taxa de crescimento do tempo de processamento melhor que o previsto naquele estudo, por estabelecerem limites para o número de "backtrackings" (como será visto a seguir), com o consequente sacrifício da cobertura, e por fazerem uso extensivo de heurística nas técnicas de procura da solução.

O problema de geração de padrão de teste (GPT), pode ser visto como um problema de procura em espaço finito [Goel 81]. Para um circuito com N entradas primárias existem 2^N possíveis combinações de sinais nas entradas. Estas 2^N combinações representam todos os pontos que compõem o **espaço de procura**. Em geral, somente uma pequena parcela destas combinações preenche o requisito imposto, qual seja, ser um padrão de teste para a falha em questão. Desta forma, a geração de padrão de teste pode ser vista como o problema de encontrar um ponto no espaço de procura que corresponda a um padrão de teste e, consequentemente, a uma solução para o problema da procura.

Os algoritmos para GPT usualmente buscam a solução construindo uma árvore de decisão e nela aplicam procedimentos de procura via " **backtracking**" [Goel 81] e [Fujiwara 86]. A árvore de decisão pode ser dividida em: uma **área de solução**; e em várias **áreas de não-solução**. É importante notar que, se durante o processo de procura o algoritmo entra em uma destas áreas de não-solução, nenhuma solução pode ser encontrada pelo simples assinalamento de valores às entradas primárias do circuito. Neste caso, só se consegue deixar a área de não-solução fazendo o "backtracking", isto é, rejeitando-se assinalamentos préviamente assumidos e escolhendo-se um caminho alternativo que ainda não tenha sido percorrido.

O principal problema com os algoritmos de GPT é que, em geral eles não estão aptos a identificar por completo as áreas de não-solução mas apenas partes delas. Assim, quando um algoritmo de GPT entra em uma das áreas de **não-solução já identificadas**, ele reconhece que nenhuma solução será possível a partir deste ponto e imediatamente recorre a um "backtracking". No caso de entrar em áreas de não-solução não identificadas, uma grande quantidade de "backtracking" e de tempo de CPU é despendida na tentativa de se encontrar a solução que não existe. Em vista destes problemas, tem-se aplicado um grande esforço na busca de melhorias para os algoritmos de GPT, visando principalmente:

- minimizar as áreas de não-solução não identificadas
- evitar todas as áreas de não-solução durante o processo de procura

Estas metas podem ser atingidas por técnicas especiais de tratamento do problema de procura em árvores e, ainda, pelo uso extensivo de heurística como forma de identificar precocemente as não-soluções. A literatura contém diversos exemplos de esforços neste área:

- otimização do processo de poda da árvore de procura [Pearl 84], [Schulz 88].
- redução do número de "backtrackings" [Goel 81], [Fujiwara 83], [Abramovici 86], [Schulz 88], entre outros.
- reconhecimento de conflitos o mais cedo possível [Goel 81], [Fujiwara 83], [Schulz 88].

2.3.3 Algoritmos

Os trabalhos mais significativos nesta área são:

- o algoritmo D [Roth 66],
- o PODEM [Goel 81] e
- o FAN [Fujiwara 83].

O trabalho de Roth é fundamental. Foi o primeiro que apresentou um algoritmo completo para o problema de gerar padrão de teste para circuitos combinacionais. Algoritmo completo no sentido de que ele encontra pelo menos um teste para cada falha detectável.

O PODEM (" Path- Oriented DEcision Making") e o FAN (" FANout-oriented test generation algorithm"), são trabalhos relativamente recentes e ambos baseados no algoritmo D (D-ALG). Estes trabalhos resultam de um grande esforço na pesquisa de melhoras para o tempo de processamento do algoritmo D que se mostrou particularmente ineficiente para circuitos em árvore com muitas células tipo XOR. Com o uso de heurística para orientar o processo de escolha das soluções, tanto o PODEM quanto o FAN apresentaram sensíveis melhoras na eficiência do D-ALG.

A seguir é feito um breve resumo de cada um destes algoritmos.

Algoritmo D

Este algoritmo se desenvolve combinando dados conhecidos sobre as células primitivas do circuito com as informações sobre sua interconexão. O método é baseado na interseção dos **cubos-d**, uma notação para o cálculo de diferenças. Para descrever as diferenças entre o circuito

bom e o circuito em falha, Roth introduziu os símbolos **D** e $\overline{\mathbf{D}}$. O D-ALG lida com cinco valores para cada fio do circuito:

- 0 – zero binário tanto no circuito bom quanto no com falha,
- 1 – um binário tanto no circuito bom quanto no com falha,
- X – indeterminado (nenhum dos outros quatro),
- D – um binário no circuito bom, e zero no circuito com falha,
- $\overline{\mathrm{D}}$ – zero binário no circuito bom, e um no circuito com falha.

A seguir será examinado o algoritmo D como uma procura num espaço finito. O grafo de procura contém todos os nós do circuito, organizados em árvore. O algoritmo procura em todo o espaço uma solução para cada falha considerada. A procura prossegue até que uma solução seja encontrada ou todo o espaço tenha sido percorrido. A ordem na qual o D-ALG assinala valores para um nó depende da sua localização no grafo e do objetivo perseguido no instante do assinalamento. Por não existir uma estratégia global para atacar o problema, esta ordem depende da implementação do programa.

Como no espaço de procura existem nós que dependem completamente de outros nós do próprio espaço, à medida em que vão sendo assinalados valores para estes últimos, podem surgir conflitos que invalidem aqueles assinalamentos. Pode acontecer que conflitos-escondidos (difíceis de detectar) demorem a aparecer e como consequência uma grande perda de tempo com os assinalamentos subsequentes é inevitável.

Estes conflitos podem ser detectados pela **implicação**. Neste processo os efeitos de cada assinalamento são propagados para frente e para trás no circuito, em um processo semelhante a uma simulação lógica. Se surgirem valores inconsistentes em algum nó (0 e 1 simultaneamente, por exemplo), então existe conflito que precisa ser resolvido.

O primeiro passo do algoritmo é a construção do grafo de procura, após a qual, todos os nós são inicializados em **X**. A seguir é escolhida uma falha a ser testada. Assinala-se **D** ao nó no caso de falha **f-e-0** ou $\overline{\mathbf{D}}$ no caso de falha **f-e-1**. Inicia-se então o processo de **provocar a falha** ("fault sensitizing"). Trata-se da busca através do grafo, de assinalamentos que resultem em uma combinação de sinais nos nós de entrada primária do circuito, tais que o valor assinalado ao nó a ser testado seja preservado e não haja conflito nos demais. Diz-se que o valor de cada nó foi **justificado**. O próximo passo é **propagar** o valor assinalado ao nó a ser testado, para pelo menos uma das saídas primárias do circuito, onde possa ser observado.

Até este ponto do processo o algoritmo guarda uma lista das células que têm uma ou mais entradas em **D** e a saída em **X**. Esta lista é chamada de **fronteira-D** ("D-frontier"). Nesta altura, basta percorrer cada um destes nós e assinalar valores aos demais nós, até que a falha se propague para uma saída primária (propagação da falha). Uma vez atingido este objetivo, o teste para a falha a ser testada foi gerado. É armazenado e reinicia-se o ciclo para outra falha até que todas as falhas possíveis do circuito tenham sido processadas.

Durante todo o decorrer do algoritmo, qualquer tipo de conflito que ocorra, leva-o a efetuar um retrocesso automático [Wagner 88] ("backtracking"). Este processo consiste em andar para trás no grafo de procura para resolver conflitos, tentando assinalamentos alternativos aos feitos anteriormente. Se o conflito não é removido o processo continua até que o espaço de procura esteja esgotado. Neste caso a falha é redundante e não existe teste para ela.

A seguir é apresentado um exemplo simples do processamento do D-ALG. Para o circuito da figura 13 será determinado o teste para o nó **T f-e-0**.

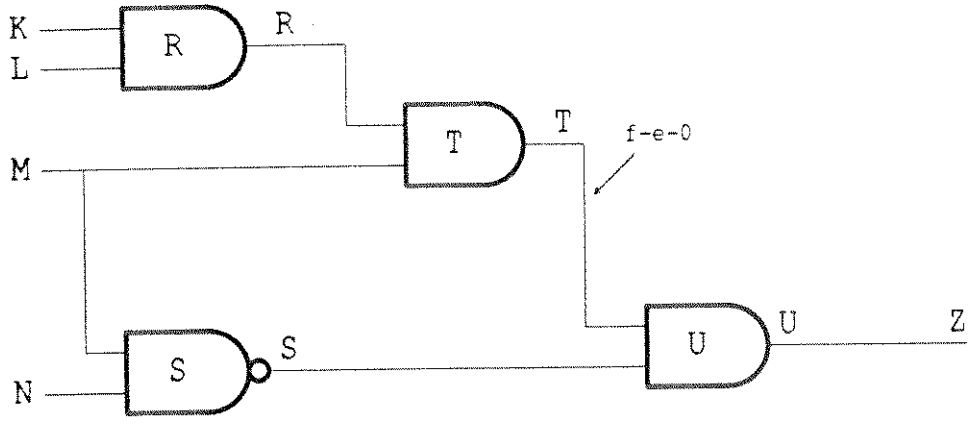

Figura 13: Circuito com derivação reconvergente.

primeiro passo: fazer T=D e assinalar valores aos demais nós de forma que seja provocada a falha. Para que a condição seja satisfeita é necessário que M=1 e R=1. Para satisfazer R=1, é preciso que K=1 e L=1 (fig. 14).

segundo passo: propagar o **D** do nó **T** para a saída **Z** do circuito. Para tanto é preciso fazer S=1. Para satisfazer S=1, há duas opções. Supondo que o D-ALG escolha M=0, haverá um conflito com o valor M=1 já assinalado no passo anterior. Uma vez ocorrido o conflito a solução é o "backtracking". O D-ALG volta na árvore (fig. 14) e verifica que existe a opção N=0 (para fazer S=1),

assinala este valor e dá por terminada a geração do teste para T **f-e-0.**

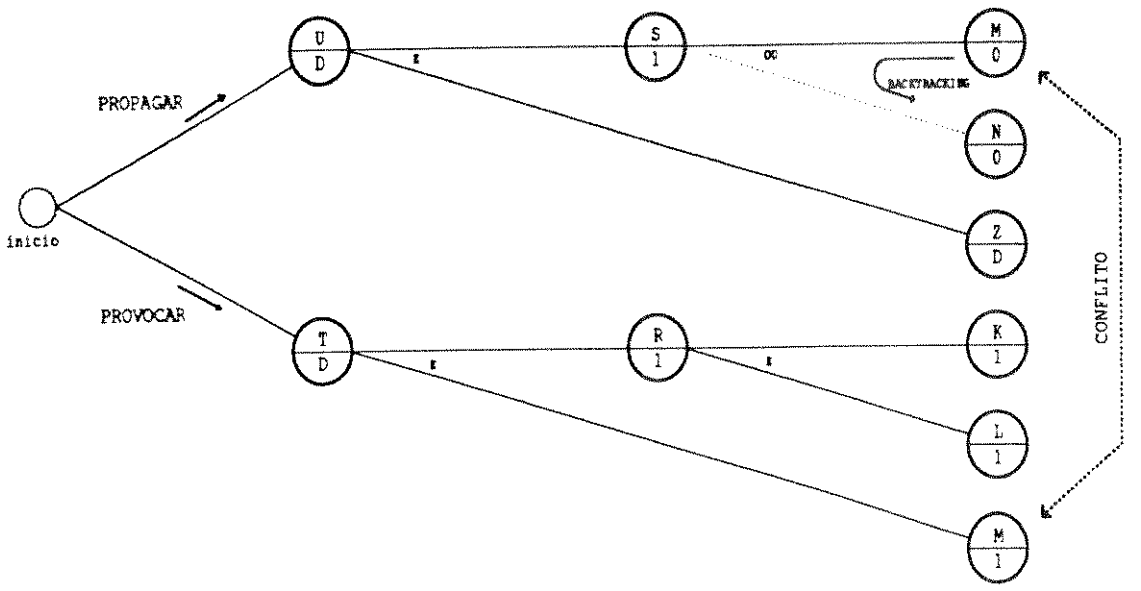

Figura 14: Árvore de procura para o circuito da fig. 13

Este foi um exemplo simples para ilustrar o processo de "backtracking". Quase sempre será necessário recorrer a esse recurso para encontrar o teste para uma falha. As exceções são os circuitos sem derivações ("fanout-free"). Poucos circuitos práticos têm esta característica. Ainda neste mesmo exemplo, havia apenas um caminho do nó em teste (T) para a saída do circuito. Vários circuitos apresentam mais de um caminho entre o nó em falha e as saídas primárias. Nestes casos, se houver conflitos, todas as possíveis combinações de dois caminhos, três caminhos, etc, devem ser tentadas.

O resultado é uma explosão na complexidade do algoritmo. Alguns trabalhos tentaram reduzir a complexidade ([Armstrong 66]) usando o método de propagação por caminho único ("single-path sensitization"). Embora reduza significantemente o espaço de procura, a propriedade algoritmica deixa de existir, na medida em que o processo não irá encontrar um teste que exija propagação por caminhos múltiplos ([Schneider 67]).

Para resumir, o algoritmo D sempre encontra um teste para uma falha, se o teste existe, mas ele tem algumas desvantagens:

1. não propõe nenhuma estratégia para a ordem do processamento.

2. os conflitos-escondidos resultantes de dependências levam a um esforço desnecessário antes de sua detecção.

3. é algoritmicamente muito complexo por examinar todas as possíveis combinações dos caminhos.

PODEM

Em [Goel 81] seu autor observou que para circuitos combinacionais todos os nós internos são função de alguma combinação dos entradas primárias e portanto basta que se trabalhe com estas entradas. Assim quando se escolher uma entrada primária no espaço de procura não haverão conflitos-escondidos entre esses nós, que podem ser assinalados independentemente. De qualquer forma, suponha que já se tenha um conjunto de entradas assinaladas e ao assinalar uma outra, seu valor cause um conflito. Só existe uma alternativa para sua solução: complementar o valor escolhido. Caso o conflito continue ocorrendo, é necessário apelar para o "backtracking".

Para exemplificar, seja o circuito da figura 15, para o qual se deseja gerar o teste para **S f-e-0**. Para isso é necessário provocar a falha, fazendo S=1. Inicialmente, o PODEM irá identificar um valor a ser assinalado a uma das entradas primárias do circuito. Ele faz isso percorrendo o circuito a partir do nó S, em direção às entradas, construindo o grafo de procura. Este processo é chamado de "backtracing". A figura 16 mostra o grafo de procura montado para este exemplo.

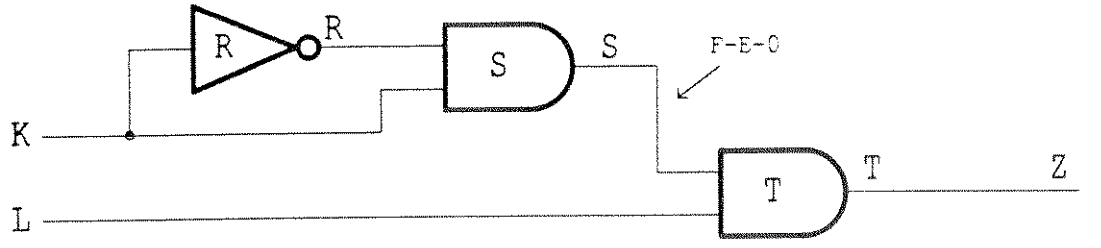

Figura 15: Circuito com falha redundante.

Suponha que a seguir o algoritmo prossiga assinalando K=1. A implicação deste assinalamento é que S=0 (pois R=0, fig. 15), o que conflita com nossa meta de fazer S=1 (para testar **S f-e-0**). O PODEM precisa então fazer um "backtracking" e tentar assinalar K=0. Também haverá conflito (duplo) e desta forma podemos concluir que não há teste para a falha em questão, porque ela é redundante.

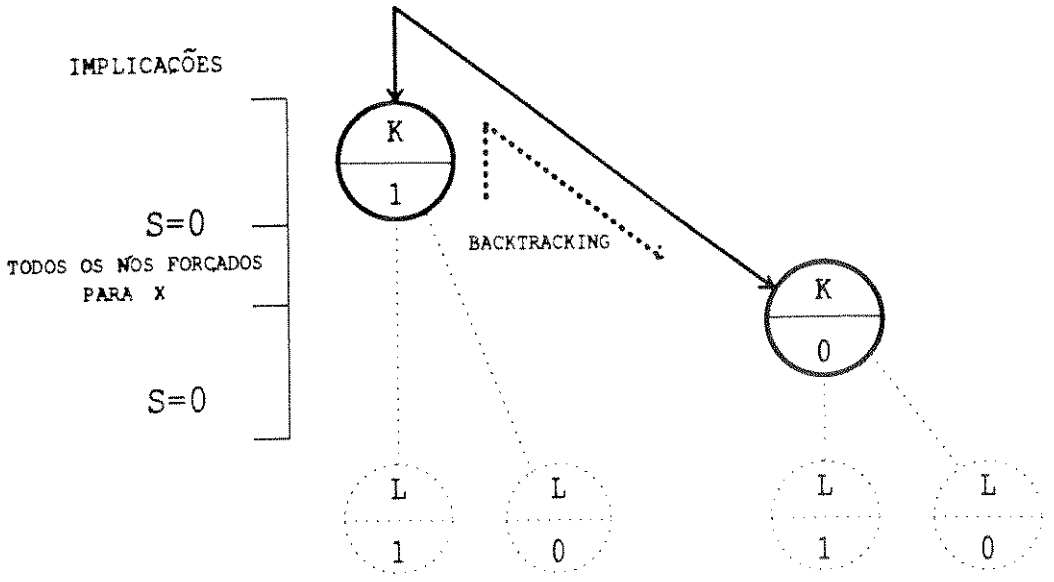

Figura 16: Grafo de procura para o circuito da figura 15.

Caso o algoritmo já houvesse assinalado valores para outras entradas antes de detectar este conflito, a conclusão seria que este assinalamento não seria um teste para a falha **S f-e-0**. O PODEM recorreria a "backtrackings" até que o grafo de procura tivesse sido todo percorrido sem que o teste fosse encontrado.

Este algoritmo irá eventualmente tentar todas as possíveis combinações nas entradas, explicitamente ou implicitamente, até encontrar um teste. Ele tenta explicitamente todas as combinações realmente assinaladas e implicitamente, aquelas rejeitadas devido a conflitos duplos. As combinações não tentadas são efetivamente cortadas do espaço de busca (linhas pontilhadas na fig. 16) e nunca serão tentadas. Por essa razão o PODEM é usualmente mais eficiente que o D-ALG, especialmente para circuitos com grande número de XORs, tais com árvores de paridade e circuitos corretores de erros.

FAN

Este algoritmo é assim chamado porque trata os nós com derivações ("fanout") como um caso especial, na tentativa de reduzir o número de "backtrackings" no grafo de procura. O espaço do grafo usado para procura pelo FAN é composto de nós com derivações e as chamadas "headlines". Para defini-las é preciso antes definir o conceito de linhas livres ("freelines") ([Fujiwara 83] e [Kirkland 88]). As linhas livres são saídas de células tais que nenhum dos nós antecessores no circuito seja uma derivação reconvergente. As "headlines" são linhas livres conectadas a uma célula que faça parte de uma derivação reconvergente.

Na figura 13, por exemplo os nós **K**, **L**, **R**, **M** e **N** são linhas livres mas somente **R**, **M** e **N** são "headlines", uma vez que **K** e **L** são antecessoras de **R**. Outra maneira de definir uma "headline" é considerá-la a raíz de uma árvore que forma um subgrafo no circuito. Sendo as folhas da árvore, entradas primárias.

Aos nós com derivações devem ser assinalados valores consistentes com todos os caminhos livres desde o nó até o ponto de reconvergência. Às "headlines", por outro lado, pode-se assinalar valores arbitrariamente porque esses nós são independentes de qualquer um dos já assinalados no circuito. Assim quando o FAN executando um "backtracking" encontra uma "headline" ele pára. Isto é, o algoritmo não precisa completar o "backtrace" até as entradas primárias (como o PODEM). Esta é uma das razões pelas quais o desempenho do FAN supera o do PODEM.

Na figura 17 pode-se acompanhar como o FAN gera um teste para o nó T **f-e-0**, e comparar com o PODEM. Ambos começam da mesma maneira, assinalando T=D e M=1 e iniciando o "backtrace". Quando atinge R=1 (fig. 17(b)) o FAN pára porque sabe que trata-se de uma "headline" e portanto, pode ser mais tarde aplicada sem conflitos. Neste ponto o PODEM prosseguiria até atingir as entradas primárias (**K** e **L**) e assinalar valores a esses nós de tal forma que R=1. Evitando este esforço, o FAN prossegue assinalando N=0, fazendo com que os nós do circuito assumam os valores anotados na figura 17(b). Neste ponto (como a saída primária Z=D) o algoritmo reconhece que encontrou um teste e tudo o que resta a fazer é aplicar o valor desejado à "headline" **R**. Isso é feito da mesma forma que no PODEM. A diferença resulta do fato de que este esforço só é despendido após a certeza de que um teste foi encontrado.

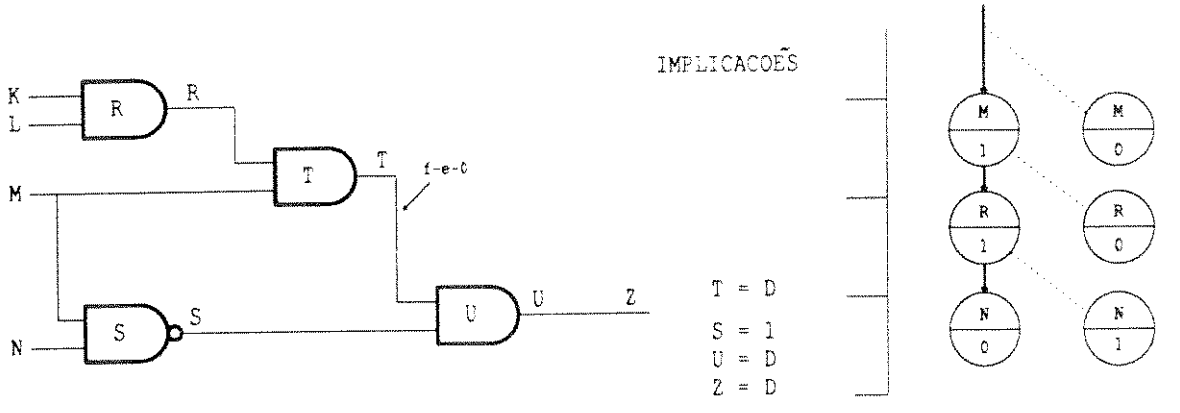

Figura 17: Circuito exemplo (a) e grafo de procura do FAN (b).

Outra técnica usada pelo FAN é o "backtrace" múltiplo que na maioria dos circuitos acaba por diminuir o número de "backtrackings" no grafo de procura. Na maior parte dos casos a

quantidade de esforço despendido para cada "backtrace" no circuito é menor que com outros algoritmos.

Para exemplificar esta técnica, suponha o circuito da figura 18, onde há nós com derivações reconvergentes no centro do circuito ("embedded"). Para uma falha em **Y**, o PODEM executa o "backtrace" de **Y** para **U**, assume um valor para **U** e prossegue em direção a **R** e depois em direção às entradas primárias. Só então um possível conflito seria detectado e uma grande quantidade de "backtrackings" poderia ter que ser executada.

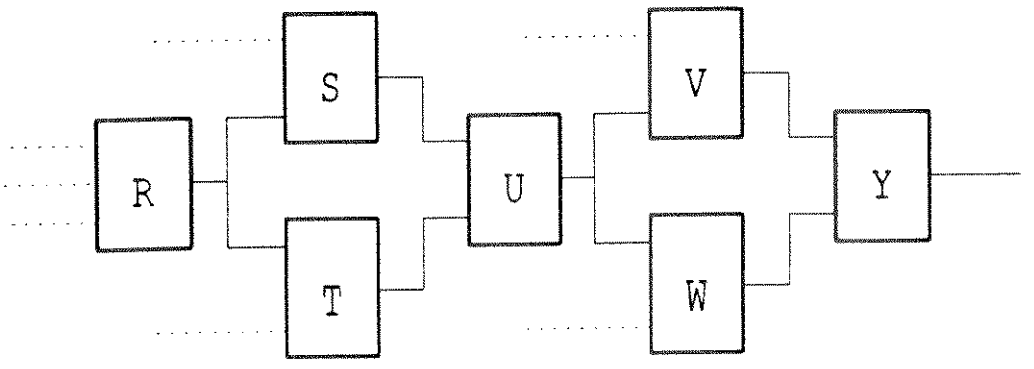

Figura 18: Derivação reconvergente no centro do circuito.

O FAN resolve este problema fazendo com que cada nó com derivação seja inicialmente, explicitamente marcado. Assim ao encontrar **U**, o algoritmo pára o "backtrace", volta ao nó sendo tratado (**Y**, no exemplo) e inicia um novo "backtrace" por outro caminho. Só após ter tratado todos os possíveis caminhos do ponto de reconvergência (Y) até o nó com derivação (U) é que o FAN assume um valor para **U**. Caso exista algum conflito, o valor é trocado, com muito menos esforço que no PODEM.

2.3.4 Outros métodos

Nesta seção são citados resumidamente alguns métodos não algoritmicos para a geração de teste. Para um estudo aprofundado do tema o leitor deve consultar [Agrawal 88] e [Tsui 87], livros recentes contendo uma extensa bibliografia sobre o tema.

Teste Exaustivo

Em alguns casos específicos, um teste de alta qualidade pode ser obtido a baixo custo. Por exemplo, para circuitos com pequeno número de entradas primárias, a aplicação de todos os possíveis vetores é factível e irá garantir uma cobertura de falhas de 100%. Esses vetores podem ser gerados facilmente tanto por software como por hardware. Esta técnica pode ser estendida a

circuitos mais complexos através do particionamento deste em sub-circuitos menores. O tratamento computacional do particionamento de circuitos é muito complexo e ainda não resolvido a contento. Na realidade o esforço computacional não é reduzido mas apenas transferido da geração do teste para o particionamento do circuito.

Sub-circuitos podem ser testados exaustivamente em circuitos onde se use o "built-in-test". Neste tipo de solução, geradores de teste implementados em hardware na forma de deslocadores realimentados ("linear feedback shift-registers") são incluidos no circuito. É comum neste ambiente o uso de análise de assinatura ("signature analysis") como forma de compactar o espaço necessário ao armazenamento das respostas corretas (circuito sem falha), contra as quais serão comparadas as respostas obtidas do circuito sob teste.

Teste Pseudo-Aleatórios

Neste método os vetores são gerados a partir de uma máquina cíclica, pseudo-aleatória. Após a geração de cada vetor ou de alguns deles, é feita uma simulação de falhas para avaliação da cobertura alcançada. Trata-se de um método computacionalmente simples que tem mostrado-se eficiente ([McCluskey 86]), embora o tamanho do teste (número de vetores) tenda a ser maior que o obtido com os métodos algoritmicos.

Sua grande potencialidade está no fato de que a máquina cíclica que gera os vetores pode ser implementada tanto em programas ("software") como em circuitos. Desta forma a solução encontrada pode ser facilmente transposta para um ambiente de "built-in-test".

2.3.5 Hierárquicos

Há vários trabalhos publicados que reconhecem a importância e necessidade do tratamneto de células hierarquizadas na geração de testes para circuitos digitais [Johansson 83], [Trischler 84], [Breuer 85], entre outros. Mais recentemente vêm sendo publicados trabalhos que realmente propõem soluções para o problema. Nesta seção é feita uma breve revisão em três destes textos e uma comparação destes com o H-ALG. Os métodos revistos foram escolhidos, dois deles por se aplicarem a sistemas de ATPG já em uso e um por propor o uso de inteligência artificial.

Em [Sarfert 89] é apresentada a solução implementada no sistema SOCRATES [Schultz 88]. Este sistema já fazia uso do FAN para a solução de circuitos planificados ("flat") e continua a usá-lo para circuitos descritos com hierarquia. Nele é feito o levantamento de todas as possíveis falhas no circuito e o "fault collapsing". A seguir, é gerado o teste para cada falha. Sempre que tem que gerar teste para uma falha interna a uma macro, o sistema faz a expansão desta para a sua representação no nível de portas. As macros são pré-definidas e as informações necessárias para

seu processamento são armazenadas em uma biblioteca. Os circuitos só devem fazer uso destas macros e assim, apenas um nível de hierarquia é admitido.

Outro método proposto foi publicado em [Calhoun 89] e aplica-se ao sistema de auxílio ao projeto OASIS ([Brglez 89]). Denominado MODEM ("module-oriented decision making"), este método usa o algoritmo PODEM para gerar os testes em duas passadas pelo circuito. Na primeira são tratadas apenas as falhas no nível hierárquico superior (nível de módulos). A seguir é feita a simulação de falhas visando obter a lista das falhas internas aos módulos que não são testadas pelos vetores gerados na primeira fase. Na segunda passada pelo circuito, são tratadas as falhas restantes, sendo o módulo onde se encontra a falha, expandido para sua descrição a nível de porta (um nível de hierárquia).

O uso de inteligência artificial na geração de padrões de teste tem sido estudado e [Krishnamurthy 87] é um exemplo. Nele o autor propõe a separação do controle do processo de geração de teste, da base de conhecimento. O algoritmo de busca (no caso o D-ALG) é codificado como a parte de controle e a semântica dos módulos como a parte de conhecimento deste sistema. Esta base de conhecimento é um conjunto de máquinas de estado que descrevem adequadamente o comportamento dos módulos. Nesta "biblioteca" as máquinas são escritas em alguma linguagem computacional. Construir a base de dados de alguns módulos pode ser uma tarefa muito complexa. Como solução o autor propõe o uso de modelos incompletos que podem ser aperfeiçoados ao longo do tempo. Naturalmente o uso destes modelos incompletos pode levar a não detecção de alguma falha detectável.

A tabela da figura 19 apresenta uma comparação entre os três métodos citados nesta seção e o H-ALG. Como pode ser visto, apenas o método aqui proposto admite hierarquia em qualquer profundidade e biblioteca que cresce não apenas pela inclusão de modelos gerados manualmente, mas também através dos modelos gerados pelo próprio sistema, permitindo a reutilização dos testes gerados para as macros.

	Macros pré definidas ?	níveis hierárquicos permitidos	Algoritmo usado	Solução para hierarquia
Socrates	sim	2	FAN	expande a célula cuja falha está sendo tratada
MODEM	sim	2	PODEM ou FAN	expande a célula cuja falha está sendo tratada
Intelig. artificial	sim	2	D	o módulo é descrito através de uma máquina que fornece as informações sobre propagação e aplicação
HALG	sim, além das gerads pelo próprio método	sem limite	IALG	soluciona os diversos níveis hierárquicos igualmente

Figura 19: Tabela comparativa entre alguns métodos e o H-ALG.

Embora seja difícil fazer uma comparação entre os tempos de processamento dos métodos acima, como poderá ser visto em 4.6, o protótipo do H-ALG ainda apresenta um resultado que se pode considerar pobre, em relação aos demais. Porém, suas reais características hierárquicas, sem dúvida superam estes métodos

Outros trabalhos publicados nesta área e não revisados aqui, podem ser pesquisados pelo leitor interessado, em: [Somenzi 85], [Chandra 87], [Brahme 87] e [Murray 88].

2.4 Projeto visando testabilidade

A literatura nesta área é farta. O já citado [Agrawal 88], contém um cuidadoso levantamento bibliográfico a respeito. Nesta seção são citados de maneira superficial os métodos mais difundidos atualmente.

O principal objetivo das técnicas desenvolvidas para orientar o projeto de circuitos digitais visando sua testabilidade é minimizar o esforço necessário à geração e execução dos testes e tornar os circuitos testáveis. Os circuitos sequenciais são particularmente visados por estes métodos devido a reconhecida dificuldade que apresentam para serem testados. Muitas vezes a não aplicação de uma ou mesmo de uma combinação destas técnicas torna o circuito impossível de ser testado, levando às consequentes perdas.

A proposta que maior impacto exerce ainda hoje é o scan-test publicado em [Eichelberger 77] e denominado naquele trabalho de Level Sensitive Scan Design, LSSD. O fundamental deste método é permitir que qualquer circuito sequencial possa, para efeito de teste, ser subdividido em diversas partes (módulos) estritamente combinacionais. Todos os elementos de

armazenamento, no modo teste, são interligados formando cadeias de deslocamento que se iniciam em uma entrada primária e terminam em uma saída primária. O teste de circuitos implementados segundo este método se dá em duas fases: o teste das cadeias de deslocamento, seguido do teste dos módulos combinacionais. Estes módulos são particionados de tal forma que se proporcione o acesso às suas entradas através de entradas primárias do circuito ou de saídas da cadeia de deslocamento. Da mesma forma, suas saídas ou são acessadas através de saídas primárias do circuito ou de entradas da cadeia de deslocamento (fig. 20).

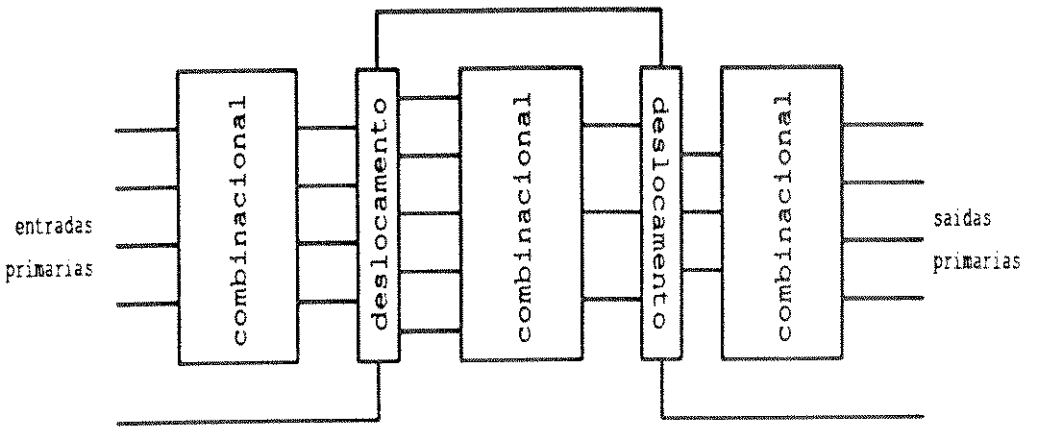

Figura 20: Exemplo de circuito usando LSSD.

O chamado "boundary-scan" [vanEerdevijk 86] é uma extensão da proposta do "scan-test" para aplicação a nível de placa. Este método permite eliminar a antiga cama-de-pregos usada para acessar pontos internos à placa durante os testes. A idéia é incluir uma série de flip-flops, internos a cada integrado, adjacentes aos seus pinos, de forma que os sinais entre os CI's possam ser controlados e observados da mesma forma que na proposta original. Se o método for usado em todos os CI's que compõem a placa, o "scan-path" resultante pode ser usado para testar a maioria dos defeitos introduzidos no processo de montagem da placa [IEEE 89].

O uso da metodologia "scan-test" permite que a geração de padrão de teste para qualquer circuito, possa ser feita por algoritmos que tratem apenas circuitos combinacionais. Como pode ser visto no item 2.3.3, mesmo esta "simplificação" não implica em que a geração de padrão de teste seja um desafio simples.

3 O algoritmo incremental: I-ALG

Nesta seção é descrito o I-ALG ([Côrtes 88]). Em 3.1, são revistos os fundamentos nos quais se baseia o algoritmo. A seguir, em 3.2 são definidas as diversas tabelas usadas no decorrer do processamento do algoritmo. A biblioteca das primitivas tratadas pelo I-ALG está no item 3.3. Em 3.4 são explicadas as operações básicas do algoritmo, e a seguir, em 3.5 verificamos as simplificações que podem ser efetuadas. Finalmente, em 3.6 são apresentados alguns resultados obtidos em testes executados com o protótipo do I-ALG.

3.1 Descrição geral

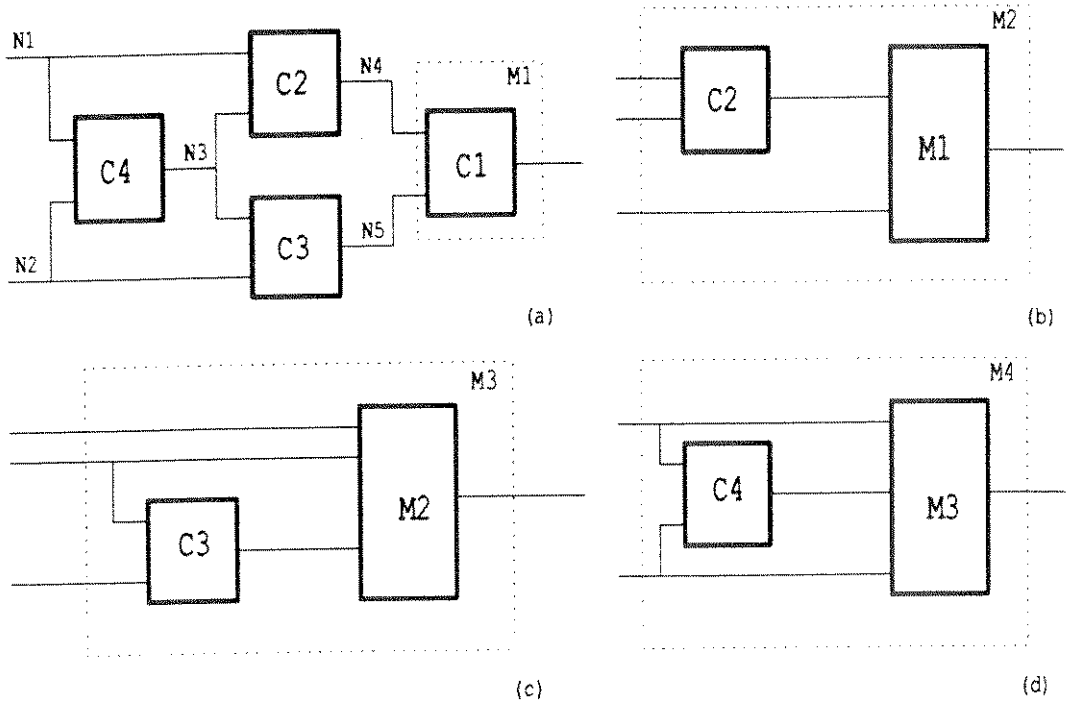

Figura 21: Iterações do algoritmo.

O algoritmo incremental é baseado na idéia de se resolver o teste de pequenas células e ir combinando os resultados de uma forma incremental, agregando-se uma nova célula a cada passo. A figura 21 ilustra a aplicação do algoritmo para um circuito composto de 4 células: C_1, C_2, C_3, C_4. No passo inicial a célula C_1 é escolhida para compor o primeiro **bloco mestre** M_1 (fig. 21(a)). A seguir, uma nova célula C_2 (por exemplo) é escolhida para ser incorporada a M_1, formando assim o novo bloco mestre M_2 (fig.21 (b)). Assim, sucessivamente, a cada iteração nova célula é incorporada ao bloco mestre atual, formando um novo bloco a ser utilizado na próxima iteração. Os passos se repetem até que todas as células tenham sido incorporadas e o

bloco mestre seja o próprio circuito em questão. Desta forma, as condições de teste para o circuito como um todo, são levantadas a partir de pequenos incrementos feitos célula a célula.

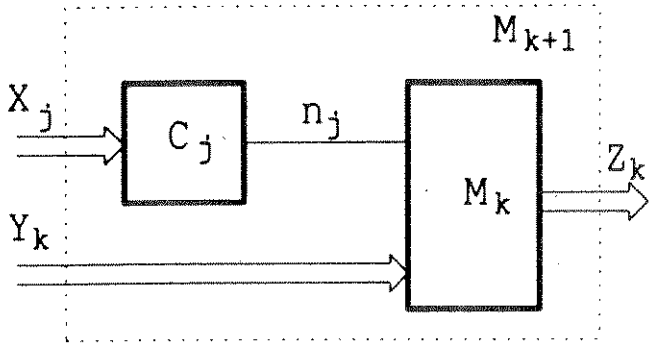

Figura 22: Interface célula-bloco mestre.

Caso seja congelado o processamento em uma dada iteração ($k^{ésima}$, por exemplo), será possível observar que a interface entre a célula e o bloco mestre (fig. 22), consiste da célula C_j, do bloco mestre M_k, do **nó de interface** n_j, além dos fios de entrada X_j e Y_k e dos fios de saída Z_k. A incorporação das células se faz das saídas do circuito caminhando em direção às entradas, assim, por construção, as linhas Z_k são **saídas primárias** através das quais pode-se observar o comportamento do circuito.

Abstraindo o restante do circuito, pode-se supor que de alguma forma seja possível controlar os fios X_j e Y_k. Assumindo que isso seja verdade, o teste da célula C_j pode ser **aplicado** aos fios X_j e ser observado em n_j, nó de interface. Como não se tem acesso externo ao nó de interface, é necessário observá-lo nas saídas primárias Z_k. Para tanto, é necessário controlar Y_k de forma que o fio n_j seja **propagado** através do bloco mestre M_k. Por outro lado, para testar o bloco M_k é preciso aplicar os vetores obtidos na iteração anterior, às suas entradas (Y_k e n_j). Como foi assumido anteriormente que é possível de alguma forma controlar Y_k, o problema para testar M_k se resume a controlar n_j. Isso é possível de ser feito através dos fios X_j uma vez que a função de transferência de C_j é conhecida.

Resumindo, para se testar o bloco mestre M_{k+1}, é necessário:

1. conhecer que vetores precisam ser aplicados às entradas da célula C_j e do bloco M_k para que cada qual seja testado.

2. saber que combinação de sinais deve ser aplicada às entradas Y_k do bloco M_k de forma que o nó de interface seja propagado através de M_k, para que possa ser observado nas saídas primárias (Z_k).

3. saber como controlar n_j (nó de interface) a partir das entradas X_j de C_j.

Como estas condições são satisfeitas no I-ALG é assunto da seção 3.2. Antes porém é necessário verificar se a hipótese assumida inicialmente é factível. Qual seja, controlar X_j e Y_k, para qualquer célula interna ao circuito. A verificação é simples: Como a incorporação de novas células se faz das saídas do circuito em direção às entradas primárias (as quais se pode controlar) basta constatar que cada um dos fios X_j e Y_k ou é saída de outra célula que irá ser incorporada ao bloco mestre ou é uma entrada primária. No último caso, é óbvio, já se tem controle do fio. Quanto ao caso do fio ser saída de uma célula, basta imaginar que quando esta célula for incorporada ao bloco mestre outros X_j (suas entradas) tomarão o lugar do nó de interface. Repetindo-se este raciocínio para todas as demais células que compõem o circuito, facilmente se poderá concluir que, no limite, na última célula a ser incorporada, tanto X_j como Y_k serão as próprias entradas primárias do circuito. E, ainda mais, o bloco mestre será o próprio circuito.

No caso de circuitos sem derivações ("fanout-free") sempre será possível satisfazer todas as condições listadas anteriormente. Nos circuitos com derivações é provável que em algumas situações seja impossível controlar algum dos fios X_j ou Y_k. Neste caso surgirão contradições no decorrer do processamento que serão resolvidas pelas funções **propagação**, **aplicação** e **combinação**, como será visto mais adiante.

3.2 Tabelas das células e bloco mestre

Nesta seção é mostrado como são organizadas as informações necessárias para a execução do algoritmo.

O algoritmo incremental trabalha basicamente manipulando tabelas que se referem às células e ao bloco mestre. As que se referem às células são pré-definidas e fazem parte de uma biblioteca as do bloco mestre vão sendo calculadas no decorrer do processamento. Todas têm tantas colunas quantas forem as entradas de cada célula, mais uma referente à saída. Ou seja, para células com N entradas: N+1 colunas (veja fig. 23). As linhas das tabelas contêm conjuntos de valores para as entradas e a saída das células (**vetores**). O tipo e finalidade de cada tabela é descrito na seção 3.3. Nestas tabelas os valores lógicos assumidos pelos fios são representados da forma que se segue:

0- para fios em nível lógico zero.

1- para fios em nível lógico um.

X- para situações onde o nível lógico do fio não interessa à operação (" don't care").

D- em substituição aos valores **0** e **1** nos fios que estejam propagando erros. Tal como em [Roth 66], significa **1** para o caso do fio correto e **0** no caso do fio não correto.

- $\overline{\mathbf{D}}$- em substituição aos valores **0** e **1** nos fios que estejam propagando falhas. Tal como em [Roth 66], significa **0** para o caso do fio correto e **1** no caso do fio não correto.

As tabelas associadas às células são pré-calculadas para todas as primitivas e são guardadas em uma biblioteca. Estas tabelas são de três tipos:

- Tabela de teste – $T[C_j]$ é o conjunto de padrões que testam C_j assumindo que se tenha acesso às suas entradas X_j e à sua saída n_j. $T[C_j]$ é o conjunto dos "primitive D-cube" [Roth 66] para todas as falhas de C_j.

- Tabela de aplicação – $A[C_j]$ mostra todas as maneiras de se controlar n_j para 0's e 1's, a partir das entradas da célula. $A[C_j]$ é similar ao "primitive D-cube" [Roth 66] ou ao "singular cover" [Fujiwara 86].

- Tabela de função estendida – $F_e[C_j]$ mostra todas as possíveis maneiras de se obter **0**, **1**, **D** ou $\overline{\mathbf{D}}$ na saída da primitiva.

As tabelas associadas ao bloco mestre M_k são computadas a cada iteração e são de dois tipos:

- Tabela de teste – $T[M_k]$ mostra como testar M_k, assumindo que tenhamos acesso a n_j, Y_k e às saídas Z_k (fig. 22).

- Tabela de propagação – $P[M_k]$ mostra todas as possíveis maneiras de se propagar erros das entradas de M_k (Y_k en_j) para as saídas Z_k (fig. 22). Esta tabela ("transparency cube" em [Ladjadj 86]) é uma generalização do "propagation D-cube" ([Roth 66]) no sentido de que permite a propagação simultânea de mais de um **D** através do bloco mestre.

3.3 Biblioteca de primitiva

Nesta biblioteca estão armazenadas as tabelas de **teste**, **aplicação** e **função estendida** de cada primitiva tratada pelo I-ALG. Nesta seção é apresentado um exemplo da estrutura de cada uma destas tabelas (fig.23) para a primitiva AND3 (porta E de 3 entradas).

teste

A	B	C	S	conjunto de falhas
1	1	1	D	A/0, B/0, C/0, S/0
0	1	1	$\overline{D}$	A/1, S/1
1	0	1	$\overline{D}$	B/1, S/1
1	1	0	$\overline{D}$	C/1, S/1

aplicação

A	B	C	S
1	1	1	1
0	X	X	0
X	0	X	0
X	X	0	0

função estendida

A	B	C	S
1	1	1	1
0	X	X	0
X	0	X	0
X	X	0	0
D	1	1	D
1	D	1	D
1	1	D	D
D	D	1	D
D	1	D	D
1	D	D	D
D	$\overline{D}$	X	0
D	X	$\overline{D}$	0
X	D	$\overline{D}$	0

Figura 23: Exemplo das tabelas da primitiva AND3.

T[AND3] é a **tabela de teste** para a primitiva. Dela fazem parte além dos vetores de teste propriamente ditos, o conjunto das falhas que cada vetor detecta. No exemplo só nos ativemos em detectar falhas tipo fixo-em mas, em geral, qualquer tabela de teste para qualquer modelo de falha pode ser fornecida ao algoritmo. Esta característica é uma vantagem adicional do I-ALG que permite que ele possa ser utilizado com modelos de falhas tais como "stuck-on" e "bridging faults" [Wadsack 78], além do clássico fixo-em.

A **tabela de aplicação** A[AND3], contém informações sobre todas as opções de como fazer a saída da primitiva assumir valores **0** ou **1**. Finalmente, a **tabela de função estendida**, mostra como obter **0**, **1**, **D** ou $\overline{\mathbf{D}}$ na saída da primitiva. Note (fig. 23) que não existem linhas que definam $\overline{\mathbf{D}}$ nas saídas, uma vez que basta inverter todos os **D** da linha, em $\overline{\mathbf{D}}$ e vice-versa para se obter estas condições. No exemplo da figura 23 a tabela de aplicação está isolada apenas para facilidade de raciocínio. Como pode ser observado, ela na realidade é uma parte da tabela de função estendida (primeiros 4 vetores, no exemplo), e portanto não ocupa espaço extra na memória.

3.4 Operações básicas

Como já foi visto anteriormente o objetivo básico de cada iteração do algoritmo é calcular as tabelas de teste e propagação do bloco mestre a serem utilizadas na iteração seguinte. Quando da inicialização do bloco mestre, na primeira iteração, sua tabela de teste é definida como cópia da tabela de teste da primeira célula a ser processada. A tabela de propagação do bloco mestre por sua vez, é definida como uma cópia da tabela de função estendida desta mesma célula. Com exceção deste primeiro passo onde o bloco mestre é inicializado, em todos os demais são

efetuadas três operações básicas que são detalhadas nos sub-itens a seguir, todas elas se baseiam na interseção de dois vetores de teste.

Antes porém, é necessário definir a interseção de dois vetores, **a** $\cap$ **b** ("D-intersection", em [Roth 66]):

Esta operação define como duas condições especificadas pelos vetores **a** e **b**, podem ser simultaneamente satisfeitas. A operação é efetuada comparando-se membro a membro os vetores, levando-se em consideração a tabela da figura 24. Nela são usados quatro novos símbolos assim definidos:

Φ – caso algum membro do vetor interseção assuma este valor, diz-se que a interseção é vazia.

Ψ – caso algum membro do vetor interseção assuma este valor, diz-se que a interseção é não definida.

λ e μ – no caso em que nenhum membro do vetor interseção resulte em Φ ou Ψ, considera-se três hipoteses:

- se ocorrerem ambos (λ e μ) simultaneamente, a interseção é não definida.
- se ocorrerem apenas μ's, deve-se para estes membros, fazer $D \cap D = D$, $\overline{D} \cap \overline{D} = \overline{D}$, e a interseção está definida.
- se ocorrerem apenas λ's, deve-se inverter em **b** cada D por $\overline{D}$ e vice-versa. Recaindo-se na caso anterior, ao se refazer a operação.

$\cap$	0	1	X	D	$\overline{D}$
0	0	Φ	0	Ψ	Ψ
1	Φ	1	1	Ψ	Ψ
X	0	1	X	D	$\overline{D}$
D	Ψ	Ψ	D	μ	λ
$\overline{D}$	Ψ	Ψ	$\overline{D}$	λ	μ

Figura 24: "D-intersection" [Roth 66].

3.4.1 Geração da tabela de teste $T[M_{k+1}]$

Gerar a tabela de teste do bloco mestre a cada iteração é um trabalho dividido em duas etapas:

1. **fazer com que o teste da célula C_j, que juntamente com M_k vai formar o bloco mestre M_{k+1}, seja propagado através de M_k** de tal forma que possa ser observado na saídas primárias do circuito (fig. 25).

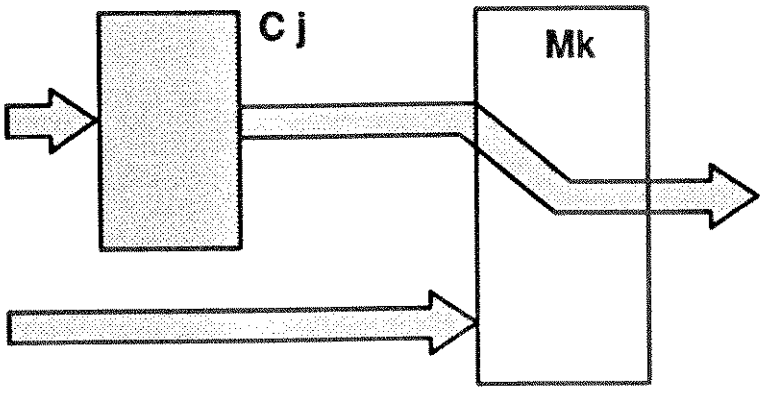

Figura 25: Propagação de C_j através de M_k.

2. na outra etapa é necessário fazer com que o teste de M_k, calculado na iteração anterior, seja **aplicado** através das entradas do novo bloco mestre M_{k+1} (fig. 26).

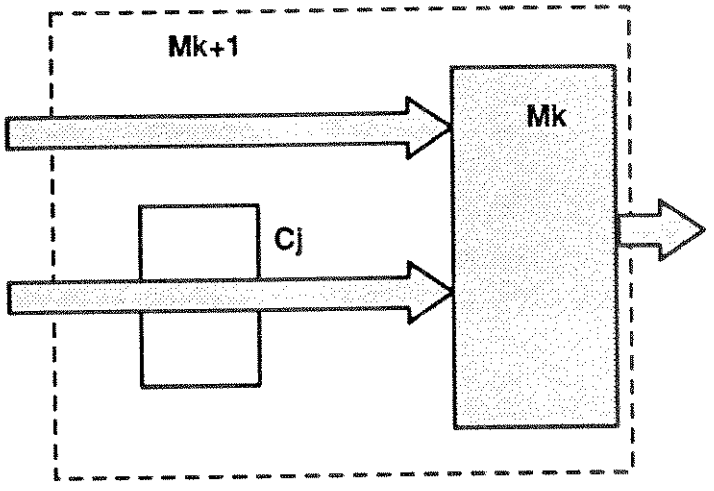

Figura 26: Aplicação de M_k via C_j.

A seguir são detalhadas estas operações através de exemplo baseado no circuito da figura 27..

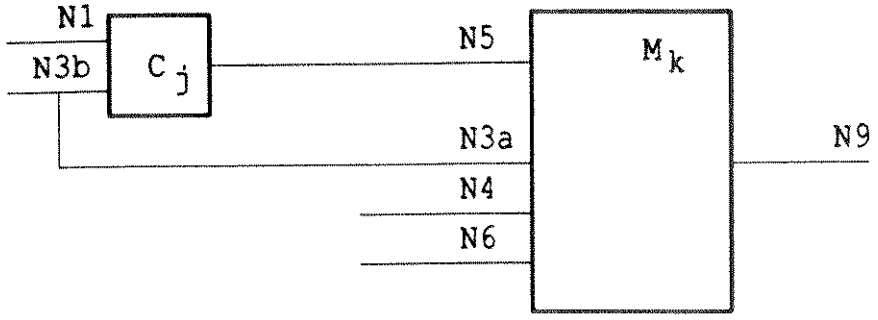

Figura 27: Circuito exemplo para as operações do I-ALG.

Propagação

O objetivo desta operação é calcular a parte da tabela de teste do novo bloco mestre (M_{k+1}) que se refere ao teste da célula propagado através do bloco mestre atual (M_k). A operação é efetuada entre a tabela de teste da célula e a de propagação do bloco mestre.

$$T_p[C_j] = \textbf{Propagação}\ (T[C_j], P[M_k])$$

Pela figura 28 podemos acompanhar como se processa a operação. $T[C_j]$ é a tabela de teste da célula sendo incorporada (E de 2 entradas). Seja $P[M_k]$ a tabela de propagação de M_k, calculada na iteração anterior. N_5 é o nó de interface entre a célula e o bloco mestre. Note que o nó N_3 tem uma derivação ("fanout") sendo, para facilitar o entendimento, tratado como se fossem dois fios N_{3a} e N_{3b}. Mais adiante veremos como tratar eventuais incompatibilidades entre os valores encontrados para N_{3a} e N_{3b} na tabela $T_p[C_j]$.

$T[C_j]$

	N_1	N_{3b}	N_5	falhas detectadas
1	1	0	$\overline{D}$	N_{3b} , N_5 , f-e-1
2	0	1	$\overline{D}$	N_1 , N_5 , f-e-1
3	1	1	D	N_1 , N_{3b} , N_5 , f-e-0

$P[M_k]$

	N_{3a}	N_4	N_5	N_6	N_9
4	0	D	1	0	D
5	1	1	D	X	D
6	1	0	D	1	D
7	0	0	D	D	D

Figura 28: Operação de propagação.

Sendo N_5 o nó de interface, devemos verificar como os erros (**D** ou $\overline{\mathbf{D}}$) que se apresentem neste nó se propagam para N_9, saída do bloco mestre. Na tabela $P[M_k]$ vemos na coluna N_5 que os vetores **5**, **6**, **7** satisfazem esta condição. Como porém, nos limitamos a detectar falhas simples (no caso, em C_j) o vetor **7** deve ser descartado pois supõe um erro em N_6 que é

impossível de ser causado por falha simples em C_j. Desta forma, apenas as linhas **5** e **6** são consideradas.

Vetor 1 propagado através de 5 ⇒ vetor 8 (T_{g1})

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
1	1		0		$\overline{D}$		
5		1		1	D	X	D
8	1	1	0	1	$\overline{D}$	X	$\overline{D}$

Vetor 1 propagado através de 6 ⇒ vetor 9 (T_{g1})

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
1	1		0		$\overline{D}$		
6		1		0	D	1	D
9	1	1	0	0	$\overline{D}$	1	$\overline{D}$

Vetor 2 propagado através de 5 ⇒ vetor 10 (T_{g2})

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
2	0		1		$\overline{D}$		
5		1		1	D	X	D
10	0	1	1	1	$\overline{D}$	X	$\overline{D}$

Vetor 2 propagado através de 6 ⇒ vetor 11 (T_{g2})

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
2	0		1		$\overline{D}$		
6		1		0	D	1	D
11	0	1	1	0	$\overline{D}$	1	$\overline{D}$

Vetor 3 propagado através de 5 ⇒ vetor 12 (T_{g3})

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
3	1		1		D		
5		1		1	D	X	D
12	1	1	1	1	D	X	D

Vetor 3 propagado através de 6 ⇒ vetor 13 (T_{g3})

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
3	1		1		D		
6		1		0	D	1	D
13	1	1	1	0	D	1	D

Figura 29: Propagação de cada um dos vetores.

A operação é feita para cada um dos vetores de teste de C_j, com os dois possíveis vetores de propagação de M_k (fig. 29). Isto faz com que a partir de cada vetor de C_j seja gerado um conjunto de vetores que denominamos grupo de teste. Cada grupo contém vetores que testam o mesmo conjunto de falhas, associado à linha de $T[C_j]$ que o gerou. Assim por exemplo, o grupo T_{g1} detecta as mesmas falhas que o vetor **1** detecta {N_{3b} e N_5, **f-e-1**}. Cada um dos vetores do grupo, entretanto, contém informações diferentes sobre as possibilidades de se propagar estas falhas até a saída primária N_9. Devem portanto ser conservados até o fim do processamento do algoritmo ou até que uma destas opções se mostre impossível de ser satisfeita (item 3.1).

$T_p[C_j]$

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9	grupo
8	1	1	0	1	$\overline{D}$	X	$\overline{D}$	$T_{g1} = \phi$
9	1	1	0	0	$\overline{D}$	1	$\overline{D}$	
10	0	1	1	1	$\overline{D}$	X	$\overline{D}$	T_{g2}
11	0	1	1	0	$\overline{D}$	1	$\overline{D}$	
12	1	1	1	1	D	X	D	T_{g3}
13	1	1	1	0	D	1	D	

Figura 30: Tabela de propagação $T_p[C_j]$.

Como citado anteriormente, embora N_{3a} e N_{3b} sejam tratados separadamente, eles são na realidade, o mesmo fio N_3 e não podem portanto, ter valores conflitantes. No exemplo acima (figura 29) os dois vetores que compõem o grupo de teste T_{g1} têm valores conflitantes para N_{3a} (

1) e N_{3b} (**0**) e por isso, na prática, não existem (fig. 30). Assim o próprio grupo de teste T_{g1} é vazio e as falhas que ele detectaria são potenciais candidatas a serem falhas não detectáveis, uma vez que não se propagam através de M_k. Note porém, que a falha N_5 **f-e-1** também é detectada pelo teste **2** de $T[C_j]$, que gerou T_{g2} que por sua vez não apresenta conflito em N_3 e portanto fará parte da tabela $T_p[C_j]$. Ocorre entretanto que a falha N_{3b} **f-e-1**, só era detectada pelo vetor **1** de $T[C_j]$, sendo portanto não detectável já que T_{g1} é vazio e nenhum outro vetor de teste detecta esta falha.

Aplicação

O objetivo desta operação é gerar a parte da tabela de teste do bloco mestre M_{k+1} que se refere ao teste de M_k aplicado através das novas entradas de M_{k+1}. A operação é efetuada entre a tabela de teste de M_k e a de aplicação da célula.

$$T_A[M_k] = \textbf{Aplicação}\ (A[C_j], T[M_k])$$

Pela figura 31 pode-se acompanhar como se processa a operação. $A[C_j]$, tabela de aplicação da célula a ser incorporada (E de 2 entradas), mostra as possíveis maneiras de aplicar **0**s e **1**s na saída da célula. $T[M_k]$ é a tabela de teste do bloco mestre M_k, calculada na iteração anterior, note que no exemplo, é composta de dois grupos de teste T_{g4} e T_{g5}. Cada qual detecta um conjunto de falhas internas ao bloco M_k. N_3 é tratado da mesma maneira que na propagação.

$A[C_j]$

	N_1	N_{3b}	N_5
1	1	1	1
2	0	X	0
3	X	0	0

$T[M_k]$

	N_{3a}	N_4	N_5	N_6	N_9
4	0	1	0	X	D
5	0	0	X	0	D
6	1	X	0	0	$\overline{D}$
7	X	1	1	0	$\overline{D}$

Figura 31: Operação Aplicação.

Sendo N_5 o nó de interface, é preciso inicialmente verificar que valor cada vetor de $T[M_k]$ exige que seja aplicado neste nó de forma a testar falhas em M_k. A seguir, consulta-se $A[C_j]$ para verificar como é possível fazer N_5 assumir o valor exigido por $T[M_k]$ e opera-se os vetores das duas tabelas que satisfazem a estas condições. Este processo é feito para cada grupo de teste por vez. Assim, no exemplo da fig. 31, o vetor **4** exige N_5=0. Consultando $A[C_j]$ pode-se verificar que existem duas opções de se gerar **0** em N_5 (vetores **2** e **3**). A aplicação destes vetores (fig. 32) gera dois vetores (**8** e **9**) em $T_A[M_k]$. Da mesma forma verifica-se que para o vetor **5** de $T[M_k]$, N_5=X, ou seja, qualquer que seja o valor aplicado a este nó, o teste será satisfeito. Isto significa que não há necessidade de aplicar qualquer vetor para gerar $T_A[M_k]$, basta apenas transportar o vetor **5** para esta tabela (agora como vetor **10**), preenchendo as colunas relativas às entradas da célula C_j com X. O próximo passo é verificar a consistência dos valores de N_3 (fio

com derivação). Neste exemplo não há contradições e no caso dos vetores **8** e **10**, o nó N_3 deve ter valor **0** já que N_{3a} assim o exige e para N_{3b} não importa se **0** ou **1**.

Vetor 4 aplicado por 2 ⇒ vetor 8 (T_{g4})

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
4		0		1	0	X	D
2	0		X		0		
8	0	0	X	1	0	X	D

Vetor 4 aplicado por 3 ⇒ vetor 9 (T_{g4})

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
4		0		1	0	X	D
3	X		0		0		
9	X	0	0	1	0	X	D

Vetor 5 aplicado ⇒ vetor 10 (T_{g4})

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
5		0		0	X	0	D
10	**X**	0	**X**	0	X	0	D

Vetor 6 aplicado por 2 ⇒ vetor 11 (T_{g5})

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
6		1		X	0	0	$\overline{D}$
2	0		X		0		
11	0	1	X	X	0	0	$\overline{D}$

Vetor 6 aplicado por 3 ⇒ vetor 12 (T_{g5})

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
6		1		X	0	0	$\overline{D}$
3	X		0		0		
12	X	1	0	X	0	0	$\overline{D}$

Vetor 7 aplicado por 1 ⇒ vetor 13 (T_{g5})

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
7		X		1	1	0	$\overline{D}$
1	1		1		1		
13	1	X	1	1	1	0	$\overline{D}$

Figura 32: Aplicação de cada um dos vetores.

Seguindo o mesmo procedimento para o grupo de teste T_{g5}, verifica-se que o vetor **6** exige N_5=0, gerando os vetores **11** e **12** em $T_A[M_k]$. Já o vetor **7** exige N_5=1. Em $A[C_j]$ apenas o vetor **1** atende esta condição, fazendo gerar o vetor **13** em $T_A[M_k]$. A verificação da consistência dos valores de N_3 mostra que o vetor

12 é impossível de ser satisfeito já que N_{3a}=1 e N_{3b}=0, devendo assim ser eliminado (fig. 33). Veja que os grupos de teste prosseguem sendo denominados T_{g4} e T_{g5} uma vez que detectam as mesmas falhas (internas a M_k) que detectavam antes da operação.

$T_a[M_k]$

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9	grupos
8	0	0	X	1	0	X	D	T_{g4}
9	X	0	0	1	0	X	D	
10	X	0	X	0	X	0	D	
11	0	1	X	X	0	0	$\overline{D}$	T_{g5}
12	X	1	0	X	0	0	$\overline{D}$	
13	1	X	1	1	1	0	$\overline{D}$	

Figura 33: Tabela de aplicação $T_A[M_k]$.

Montagem de T[M $_{k+1}$]

Como foi visto anteriormente, a tabela de teste do novo bloco mestre é dada por:

$$T[M_{k+1}] = T_p[C_j] \cup T_A[M_k]$$

Assim, basta unir as tabelas calculadas pelas funções propagação e aplicação, formando a tabela desejada (fig. 34).

	N_1	N_3	N_4	N_5	N_6	N_9	origem
10	0	1	1	$\overline{D}$	X	$\overline{D}$	da propagação
11	0	1	0	$\overline{D}$	1	$\overline{D}$	
12	1	1	1	D	X	D	
13	1	1	0	D	1	D	
8	0	0	1	0	X	D	da aplicação
9	X	0	1	0	X	D	
10	X	0	0	X	0	D	
11	0	1	X	0	0	$\overline{D}$	
13	1	1	1	1	0	$\overline{D}$	

Figura 34: Tabela de teste de M_{k+1} ($T[M_{k+1}]$).

Terminado este passo, pode-se realizar uma minimização desta tabela, assunto do item 3.5. É importante verificar que na última iteração do algoritmo, a tabela gerada será a tabela de teste do circuito objeto do cálculo. Como cada grupo de teste contém vetores que na realidade são opções de teste para o mesmo conjunto de falhas, basta que se escolha um de cada grupo para compor o teste do circuito.

3.4.2 Geração da tabela de propagação $P[M_{k+1}]$

Esta tabela define todas as maneiras de se propagar erros através do bloco mestre que está sendo criado (M_{k+1}). Ela é gerada a partir da operação **combinação**, efetuada entre as tabelas $P[M_k]$ e $F_e[C_j]$. Onde $P[M_k]$ é a tabela de propagação do bloco mestre M_k, calculada na iteração anterior e $F_e[C_j]$ é a tabela de função estendida da célula a ser incorporada.

$$P[M_{k+1}] = \textbf{combinação}\ (P[M_k], F_e[C_j])$$

Pela figura 35 pode-se acompanhar como é processada a operação. As tabelas se referem ao circuito da figura 27, onde N_5 é o nó de interface. O objetivo da operação é, através das entradas de C_j satisfazer o nível lógico necessário ao nó N_5, conforme definido na tabela $P[M_k]$. Neste exemplo, o vetor **8** desta tabela pede N_5=0. Em $F_e[C_j]$, existem três opções para que a saída da célula seja **0**: vetores **2**, **3** e **7**. A **combinação** de cada um deles com o vetor **8** de $P[M_k]$ gera os vetores **12**, **13** e **14**.

$F_e[C_j]$

	N_1	N_{3b}	N_5
1	1	1	1
2	0	X	0
3	X	0	0
4	1	D	D
5	D	1	D
6	D	D	D
7	D	$\overline{D}$	0

$P[M_k]$

	N_{3a}	N_4	N_5	N_6	N_9
8	D	1	0	1	D
9	0	D	1	0	D
10	1	0	$\overline{D}$	1	D
11	1	1	D	X	D

$P[M_{k+1}]$

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
12	0	D	X	1	0	1	D
13	X	D	0	1	0	1	D
14	D	D	$\overline{D}$	1	0	1	D
15	1	0	1	D	1	0	D
16	1	1	$\overline{D}$	0	$\overline{D}$	1	D
17	$\overline{D}$	1	1	0	$\overline{D}$	1	D
18	$\overline{D}$	1	$\overline{D}$	0	$\overline{D}$	1	D
19	1	1	D	1	D	X	D
20	D	1	1	1	D	X	D
21	D	1	D	1	D	X	D

Figura 35: Combinação de $P[M_k]$ e $F_e[C_j]$.

O próximo vetor em $P[M_k]$ (**9**), pede N_5=1. Em $F_e[C_j]$ apenas o vetor **1** satisfaz esta exigência. A combinação destes dois vetores gera **15** em $P[M_{k+1}]$. O terceiro vetor de $P[M_k]$ a ser processado (**10**) tem N_5= $\overline{\mathbf{D}}$. Os vetores de $F_e[C_j]$ que satisfazem são **4**, **5** e **6**. Note que embora $F_e[C_j]$ não tenha explicitamente as condições para N_5= $\overline{\mathbf{D}}$, basta inverter todos os **D** nos vetores para que sejam satisfeitas (vide 3.3). A operação com o vetor **10** gera os vetores **16**,

17 e **18** em $P[M_{k+1}]$. Por sua vez, a combinação do vetor **11** de $P[M_k]$ (N_5=D) é feita novamente com os vetores **4**, **5** e **6** de $F_e[C_j]$, gerando **19**, **20** e **21**.

Finalmente, basta verificar os valores em N_{3a} e N_{3b} e eliminar os vetores inconsistentes. Desta forma os vetores **13**, **15**, **16**, **18**, **19** e **21** são eliminados.

Note o vetor **14** (fig. 36). Nele N_{3a}= **D** e N_{3b}= $\overline{\mathbf{D}}$, o que não constitui uma inconsistência, uma vez que se forem invertidos os **D** (vide 3.3) provenientes do vetor **7** de $F_e[C_j]$, N_3 assumirá valor **D** e N_1, valor $\overline{\mathbf{D}}$, sem que isso altere a condição inicial a ser satisfeita:

1. nó de interface, N_5=0 e

2. N_{3a}=D

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
14	D	D	$\overline{D}$	1	0	1	D

$\Rightarrow$

	N_1	N_{3a}	N_{3b}	N_4	N_5	N_6	N_9
14	$\overline{D}$	D	D	1	0	1	D

Figura 36: O vetor 14 de $P[M_{k+1}]$.

Outro fato a se notar no vetor **14** é que ele apresenta **D** em duas entradas (N_1 e N_3), isso significa que há possibilidade de propagação através de caminhos múltiplos (" multiple path sensitization"). Como está sendo usado o modelo de falha simples, convém verificar a real possibilidade de, em função da topologia do restante do circuito, esta falha poder causar dois erros (**D**) em fios distintos. Isso pode acontecer em casos onde exista uma derivação reconvergente.

Caso seja possível verificar de antemão, se a topologia do circuito realmente possibilita ou não a existência desses dois erros em C_j (provenientes de falha simples) pode-se, em caso negativo, eliminar desde agora este vetor da tabela. A conveniência disso é óbvia pois este vetor ao ser eliminado deixará de ser objeto de operações posteriores, evitando tanto o gasto em tempo de processamento quanto em área de memória para tabelas. A Matriz de Dependência ([McCluskey 82]), mostra a inter-dependência entre os fios e é um meio conveniente para se guardar tais informações. No I-ALG ela é gerada na inicialização, antes do algoritmo propriamente dito.

Desta forma, obtem-se a tabela $P[M_{k+1}]$ final, constituida pelos vetores **12**, **14**, **17** e **20**, figura 37.

	N_1	N_3	N_4	N_5	N_6	N_9
12	0	D	1	0	1	$\overline{D}$
14	$\overline{D}$	D	1	0	1	D
17	$\overline{D}$	1	0	$\overline{D}$	1	D
20	D	1	1	D	X	D

Figura 37: Tabela $P[M_{k+1}]$.

3.4.3 Procedimentos para a seleção de C_j

As operações descritas nos itens anteriores pressupõem que as saídas do bloco mestre sejam sempre saídas primárias do circuito. Trata-se de uma simplificação que não traz nenhuma limitação à generalidade do algoritmo e ao mesmo tempo, evita a situação de uma saída do bloco mestre alimentar a entrada de uma célula. Isso obrigaria o I-ALG a conhecer e calcular as tabelas de aplicação e de função estendida do bloco mestre. A primeira, para poder gerar a parte da tabela de teste de M_{k+1} relativa ao teste da célula aplicado ao bloco e a segunda para permitir o cálculo da tabela de propagação do novo bloco mestre.

Uma consequência desta simplificação é que o critério para seleção da próxima célula a ser incluida no bloco mestre tem que ser tal que a saída da célula escolhida somente esteja conectada ao bloco mestre M_k, ou que ela seja uma saída primária do circuito (fig. 38).

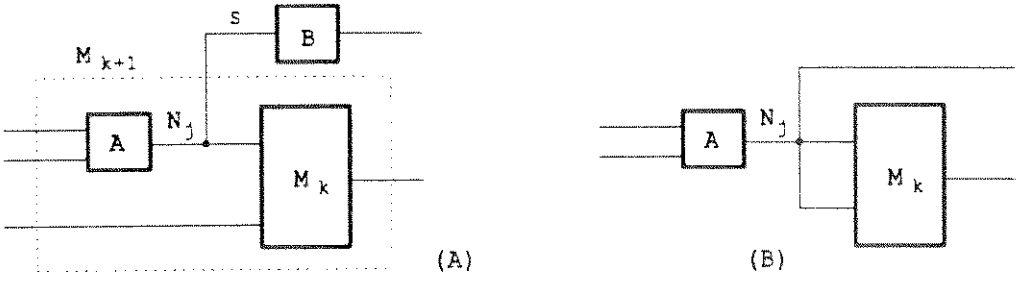

Figura 38: Bloco mestre e suas saídas.

Veja na figura 38(a) que se a célula **A** for processada antes da célula **B**, o bloco mestre M_{k+1} teria um sinal de saída **s** que não é uma saída primária do circuito, contrariando o convencionado acima. Caso a derivação em N_j fosse diretamente para uma saída primária, a célula **A** satisfaria a condição e poderia ser a próxima a ser tratada (caso (b), figura 38). Esta regra pode ser enunciada da seguinte forma:

- Uma célula cujo nó de interface com o bloco mestre tenha derivações, só pode ser considerada como **candidata a próxima célula a ser tratada** se cada uma destas derivações for entrada do bloco mestre, ou uma saída primária. (fig. 38(b)).

Para verificar se essa condição é sempre possível de ser satisfeita, qualquer que seja o circuito combinacional, será usado um nó qualquer de entrada do bloco mestre. Nele há duas condições a se considerar:

1. O nó não tem derivação – neste caso a condição está satisfeita automaticamente e a célula cuja saída é este nó pode ser a próxima a ser tratada.

2. O nó tem derivações – é necessário tratar cada derivação até que todas satisfaçam a condição de ser ou entrada do bloco mestre ou saída primária. Para verificar a possibilidade desta condição ser sempre satisfeita, basta considerar uma qualquer derivação de um nó e verificar que só existem três hipóteses, ou:

    - ser uma saída primária, ou
    - entrar no bloco mestre, ou
    - entrar em uma célula.

As duas primeiras opções já satisfazem (a célula poderia ser a próxima). É preciso então, se preocupar apenas com o caso de derivação que entra em uma outra célula.

Para verificar esta condição, basta imaginar que caso se caminhe pelo circuito a partir deste nó, em direção às saídas, passando de célula em célula, será fácil assumir que ou se chegará a uma saída primária ou a uma entrada do bloco mestre (que vem crescendo das saídas em direção às entradas). Neste caso a condição está satisfeita e a próxima célula a ser tomada, pode ser a última desta cadeia.

Ou seja, em qualquer dos três casos foi verificado que é possível encontrar uma célula que satisfaça a regra mencionada anteriormente. Antes porém, de considerar concluida esta demonstração, devemos verificar dois casos particulares extremos: os nós de saídas e entradas primárias. No primeiro caso se não há derivação, a condição está automaticamente satisfeita. Caso haja derivação ela deve ser tratada como um nó comum (como visto acima). Quanto às entradas primárias, nada há a considerar pois não existe célula cuja saída esteja ligada a um destes nós.

Na implementação do protótipo do I-ALG feito no CPqD-TELEBRÁS, a determinação da ordem de processamento para os circuitos teste foi feita manualmente. Como será visto na seção 3.6, esta ordem influi no tempo de processamento do algoritmo. Isso indica que será conveniente introduzir alguma heurística que auxilie na determinação da ordem de inclusão das células.

3.5 Simplificações

Na seção 3.4 foi visto como calcular as tabelas de **teste** e **propagação** do bloco mestre M_{k+1}. Durante os processos lá descritos, algumas simplificações que visavam reduzir o número de vetores nas tabelas, foram feitas:

- eliminação de linhas com valores conflitantes em nós com derivação
- eliminação de linhas com propagações múltiplas inexistentes na topologia do circuito.

Nesta seção veremos outro tipo de simplificação que denominamos otimização tipo-OU, além de dois métodos de simplificação associados à topologia do circuito.

3.5.1 Otimização tipo-OU

	N_1	N_2	N_3	N_4
1	1	*D*	0	*D*
2	*D*	0	X	*D*
3	*D*	0	0	*D*
4	0	1	*D*	*D*
5	X	1	*D*	*D*

$\Rightarrow$

	N_1	N_2	N_3	N_4
1	1	*D*	0	*D*
2	*D*	0	X	*D*
5	X	1	*D*	*D*

Figura 39: Simplificação de uma tabela de propagação.

Como visto anteriormente as linhas (vetores) da tabela de propagação $P[M_{k+1}]$ mostram alternativas de como propagar erros através do bloco mestre. Por se tratarem de opções, pode-se dizer que cada linha tem com as demais uma relação tipo **OU** lógico. Para exemplificar, suponha que a tabela da figura 39 seja a de propagação do bloco mestre gerada em uma determinada iteração do algoritmo. Comparando cada linha com as demais, verifica-se qua a linha **3** é um caso particular da linha **2**, mais genérica que aquela por admitir N_3=X. Ora, se ambas satisfazem ao objetivo (neste caso, de fazer o erro em N_1 se propagar até N_4) é natural que o vetor **3** seja abandonado, sem nenhum prejuizo para a solução desejada e com os óbvios ganhos em tempo de processamento e área de memória. Do mesmo modo a linha **5** contém a **4**, que pode ser abandonada. De forma geral,

se a linha $i \subseteq$ linha j, elimina-se a linha i

Como foi visto em 3.4.1 parte da tabela de teste $T[M_{k+1}]$ do bloco mestre é gerada pela propagação do teste da célula C_j através do bloco mestre M_k. Essa operação é feita com o uso da tabela de propagação e gera grupos de teste. Cada grupo testa as mesmas falhas e cada uma de suas linhas (vetores de teste) é uma opção para o teste destas falhas. Da mesma forma como na tabela de propagação, na de teste há, dentro de cada grupo, um relacionamento lógico tipo **OU** entre suas linhas. É possível portanto, simplificar grupos de teste eliminando aqueles vetores que são contidos por outros. No exemplo da figura 40 a linha **2** é caso particular da linha **1** e ambas pertencem ao mesmo grupo de teste T_{g1}. Pode-se desta forma, abandonar a linha **2**, sem perda de qualquer informação, uma vez que na realidade, havia uma duplicação de informações.

	N_1	N_2	N_3	N_4
1	1	0	X	D
2	1	0	0	D
3	1	1	0	$\overline{D}$
4	X	1	0	$\overline{D}$
5	0	1	1	$\overline{D}$

$\Rightarrow$

	N_1	N_2	N_3	N_4
1	1	0	X	D
3	1	1	0	$\overline{D}$

Figura 40: Simplificação de uma tabela de teste.

3.5.2 Fanout look-ahead

Esse método de simplificação procura antecipar a detecção de incompatibilidades entre os vetores da tabela de teste do bloco mestre e aqueles das tabelas de aplicação das células. Ao fim de cada iteração é verificado se existem no circuito conexões como a em negrito da figura 41. É o caso de fio com derivação que além de ser entrada do bloco mestre é também entrada de uma célula cuja saída é entrada do bloco mestre.

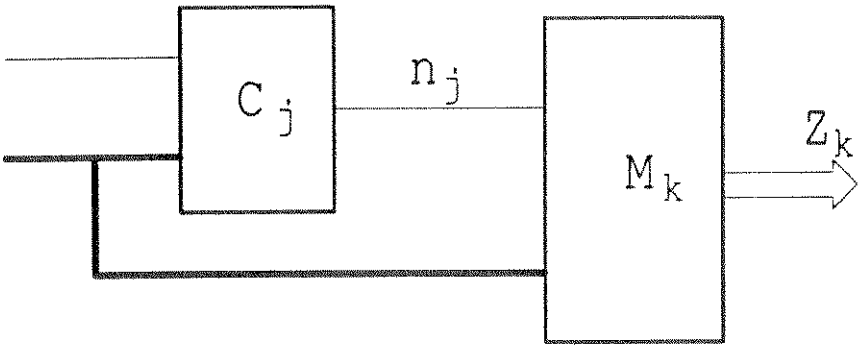

Figura 41: Circuito passivel de simplificação por "fanout look-ahead".

Mesmo que a célula em questão não seja a próxima a ser inserida no bloco, desde já pode-se procurar e eliminar vetores do bloco mestre cuja aplicação através desta célula venha a se mostrar impossível. Seja o exemplo da figura 42 onde está a tabela de aplicação da porta AND2 e um suposto trecho da tabela de teste do bloco mestre. N_4 é o nó de interface e N_3 o fio com

derivação. O vetor **1** de $T[M_k]$, resultante das iterações anteriores, exige N_3=0 e N_4=1. Ao se consultar a tabela de aplicação do AND será possível verificar que a única forma de fazer N_4=1, exige que as entradas sejam iguais a **1** (o fio N_3, inclusive). Desta forma, deve-se abandonar o vetor **1**, desde já.

aplicação AND2

N_1	N_3	N_4
1	1	1
0	X	0
X	0	0

$T[M_k]$

	N_3	N_4	N_5	N_6	N_9
1	0	1	0	X	D
2	0	0	X	0	D
3	1	X	0	0	$\overline{D}$
4	X	0	1	0	D

Figura 42: Exemplo de simplificação por "fanout look-ahead".

Esse procedimento evita que vetores que mais adiante virão a ser eliminados, fiquem sobrecarregando tanto o tempo de processamento quanto a memória. O ganho é mais significativo, nestes termos, nos casos em que a célula em questão não é a próxima a ser incluida no bloco mestre. Na seção 3.6 é apresentada uma análise da economia a partir de um exemplo prático.

3.5.3 "Freelines" e "Headlines"

Este método de simplificação é baseado no trabalho de Fujiwara sobre o algoritmo FAN ([Fujiwara 83] e item 2.3.3), onde foi observado que para as "headlines" pode-se aplicar valores arbitrariamente, sem conflitos, já que estes nós só dependem de antecessores do tipo "freeline". Isto quer dizer que a partir de entradas primárias consegue-se aplicar ao nó "headline" os valores **0** e **1**, independentemente de qualquer outra condição.

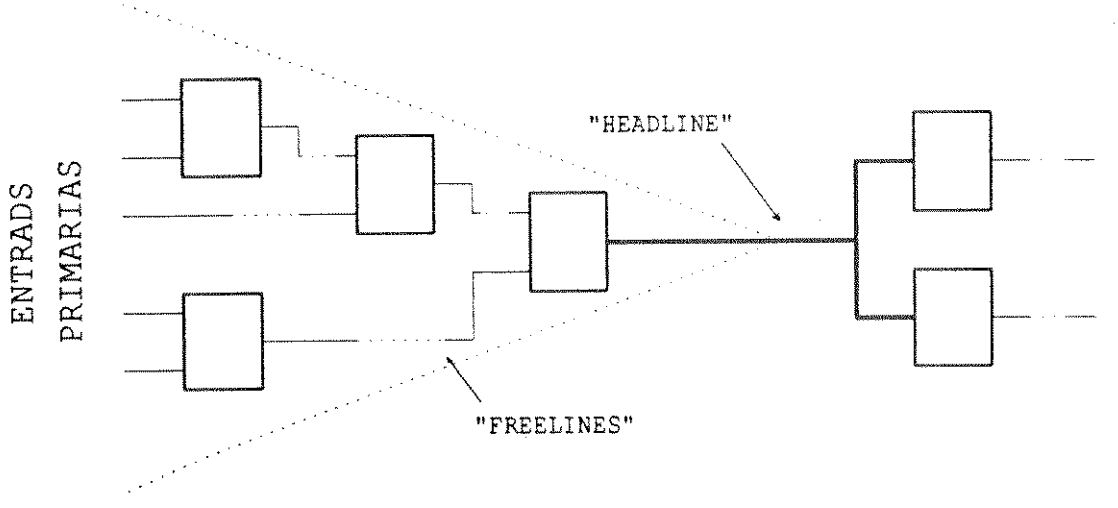

Figura 43: Exemplo de "headline" e "freelines".

Assim, se todas as "headlines" do circuito forem consideradas como entradas primárias, pode-se evitar percorrer o cone de circuito do qual a "headline" é o vértice. Ao fim do processamento, calcula-se para cada cone a aplicação de **0** e **1** em seu vértice e substitui-se em cada vetor de teste do circuito o valor exigido na "headline", pelos valores calculados para os nós de entrada do cone (fig. 43). A economia é evidente por evitar que sejam carregadas tabelas contendo opções, quando é possível *a priori*, escolher qualquer uma destas já que todas são possiveis pelo simples fato de não haver "fanout" no cone. Circuitos em forma de árvore sem derivações, são especialmente problemáticos para o I-ALG, por levarem a um crescimento desmedido das tabelas.

3.6 Resultados

Nesta seção são apresentados alguns resultados obtidos a partir de experimentos realizados com o protótipo do I-ALG. Esta versão foi implementada em linguagem C (aproximadamente 1000 linhas de código), pelo autor e pelo companheiro Sérgio Augusto de Oliveira Andrade.

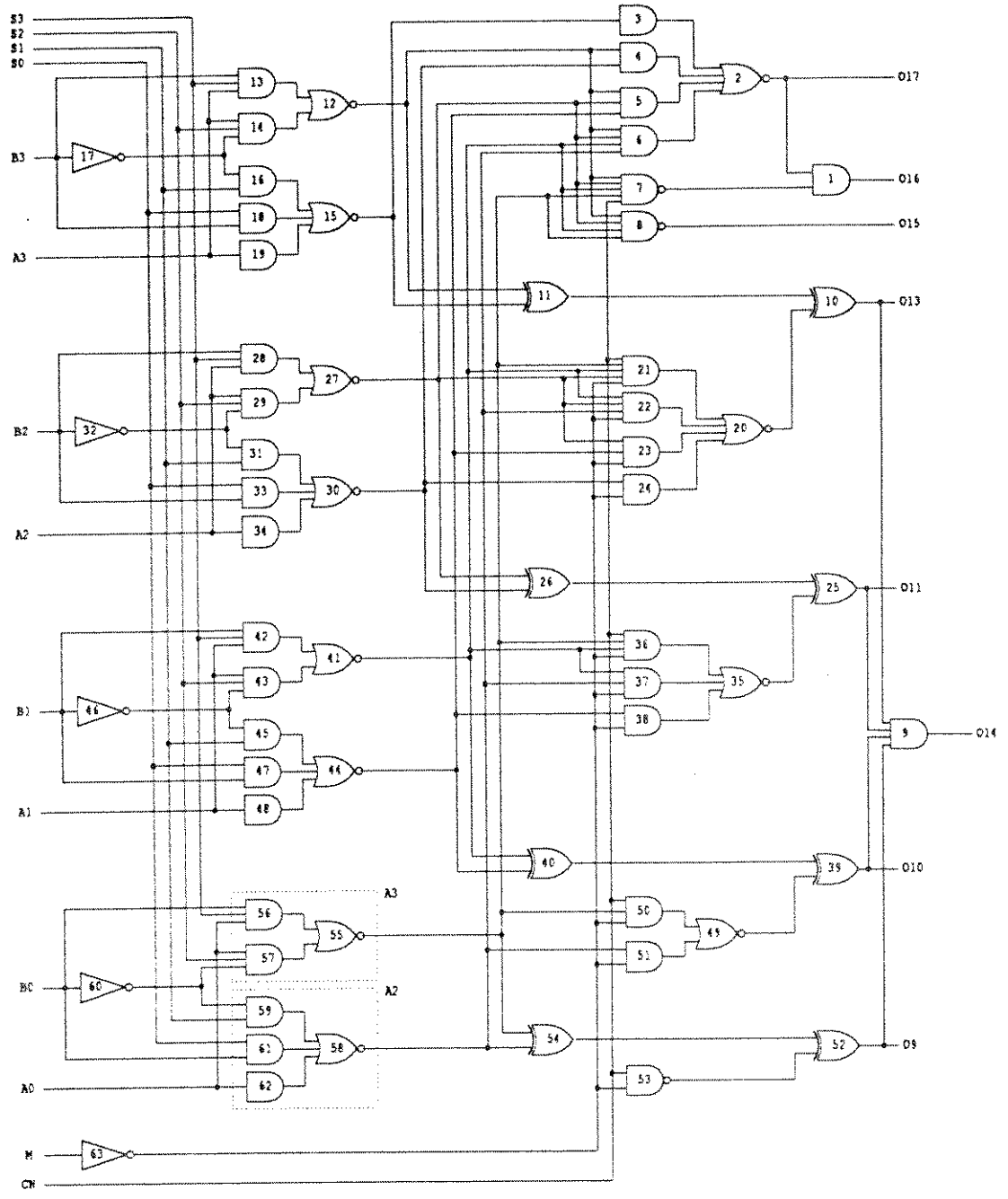

Figura 44: Unidade Lógica e Aritmética (ULA) - 74181.

O circuito usado como base para os testes é o CI comercial 74181, uma unidade lógica e aritmética (ULA) de 4 bits, contendo 63 portas ([TEXAS 76]). Este circuito (fig. 44) tem quatro falhas redundantes, internas aos XORs G11, G26, G40 e G54 que podem ser removidas substituindo-se estas portas pelo circuito da figura 45, fazendo com que a ULA seja 100% testável.

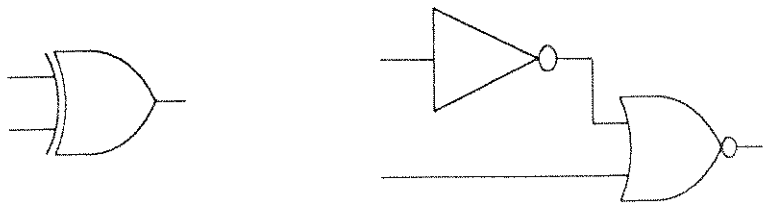

Figura 45: Modificação que torna ULA 100% testável.

Os experimentos visaram verificar a influência de alguns parâmetros no comportamento do sistema. Os parâmetros nos quais se basearam as observações, foram:

1. uso de descrição semi-hierárquica,
2. ordem de processamento das células,
3. uso do "fanout look-ahead" como método de simplificação.

3.6.1 Descrição plana e semi-hierárquica

Embora já fosse esperado que o uso de descrições hierárquicas dos circuitos no processo de geração de teste trouxesse ganhos reais em termos de tempo de processamento, o I-ALG não aceita este tipo de descrição. Para efeito de estudo desta influência introduziu-se o conceito de semi-hierarquia, uma forma que permitiu avaliar os ganhos potenciais do uso da hierarquia, usando o sistema que não a admitia. Como semi-hierarquia entende-se a definição de um determinado sub-circuito e a adição de suas tabelas (teste, aplicação e função estendida) à biblioteca de primitivas, de forma que o algoritmo possa passar a tratá-lo como uma célula básica. No exemplo foram definidos dois sub-circuitos, A2 e A3 (fig. 44), que no 74181 são repetidos quatro vezes. Assim, quando daqui para a frente for considerado o circuito como semi-hierárquico, deve-se lembrar que se trata desta ULA, com as portas básicas que compõem aqueles sub-circuitos, substituidas por uma descrição deste conjunto como se fora uma primitiva.

A comparação foi feita entre o processamento do 74181 descrito "flat" (com suas 63 portas) e o processamento deste circuito descrito com as duas células semi-hierárquicas repetidas 4 vezes cada. A ordem de processamento utilizada foi definida à mão e é dada pela própria numeração das células na fig.44.

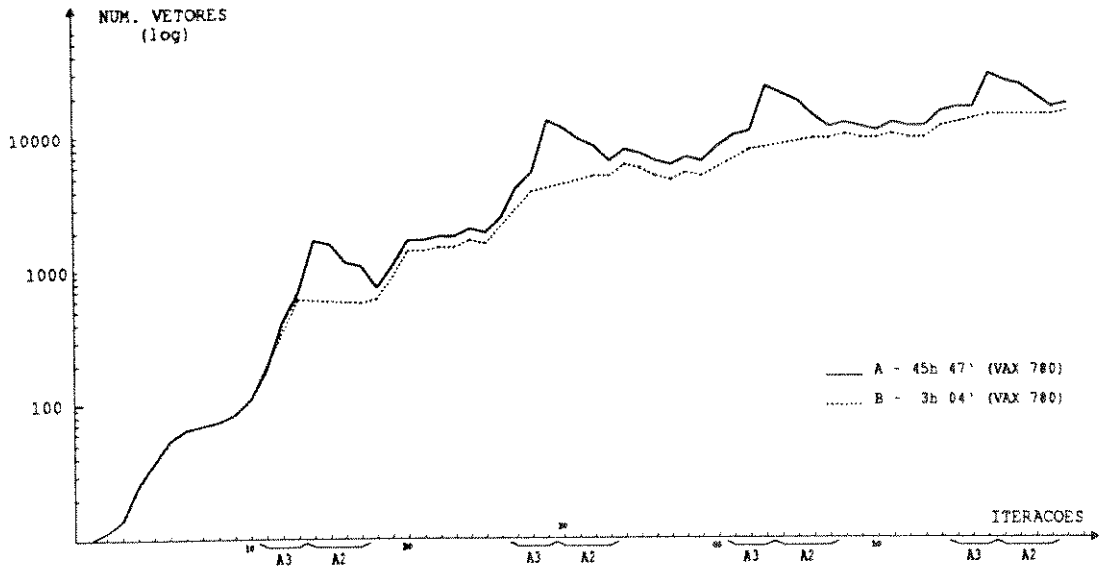

Figura 46: Comparação ULA plana (A) x semi-hierárquica (B)

A figura 46 é um gráfico do número de linhas da tabela de teste do bloco mestre a cada iteração do algoritmo (ou seja, a cada célula tratada). Este número foi computado ao final de cada iteração. A curva A se refere ao experimento com a descrição plana da ULA e a curva B se refere à descrição semi-hieráquica. Como o número de iterações nos dois exemplos é diferente e para possibilitar a sua comparação, o espaço correspondente às iterações dos sub-circuitos (A2 e A3) na curva B tem a largura igual ao número de portas que compõe cada qual.

Como pode ser observado, o tratamento semi-hierárquico reduz significativamente o número de linhas na tabela. Estas linhas deixam de ser tratadas nas n iterações correspondentes às células básicas que compõem a semi-hierárquica, passando a serem processadas apenas uma vez. O esforço computacional é claramente diminuido, e o resultado é um tempo de processamento para o circuito B de aproximadamente 6% do tempo observado para o circuito A. Um ganho significativo que aponta na direção de que soluções hierárquicas trazem ganhos efetivos neste parâmetro.

3.6.2 Ordem de processamento das células

A importância deste item nos algoritmos de GPT é conhecida. Nos baseados no D-ALG, a forma de implementação influi nas decisões tomadas cada vez que o algoritmo se depara com possíveis alternativas. O resultado são tempos melhores ou piores em função de se encontrar soluções mais rapidamente ou não.

G09	G50	G21	G07	G32
G10	G35	G26	G55	G46
G25	G38	G01	G58	G60
G39	G37	G02	G41	G63
G52	G36	G03	G27	
G53	G20	G11	G12	
G49	G24	G05	G44	
G51	G23	G06	G15	
G54	G22	G08	G17	

Figura 47: Ordem de processamento das células (curva C, fig 48).

No I-ALG é de se esperar que também exista uma relação entra a ordem de tratamento das células e o consequente tempo de processamento. Neste experimento comparou-se o resultado obtido com o processamento do circuito B, do item anterior, com o obtido para o mesmo circuito processado em outra ordem, definida na figura 47 (curva C). Ambas as ordens adotadas foram definidas à mão e existe uma diferença básica entre elas: A primeira (circuito B) tenta definir as células de modo a alcançar o mais cedo possível as entradas primárias. Já a segunda ordem adotada (circuito C), foi definida de modo que o processamento das células fosse como uma varredura em paralelo das saídas em direção às entradas.

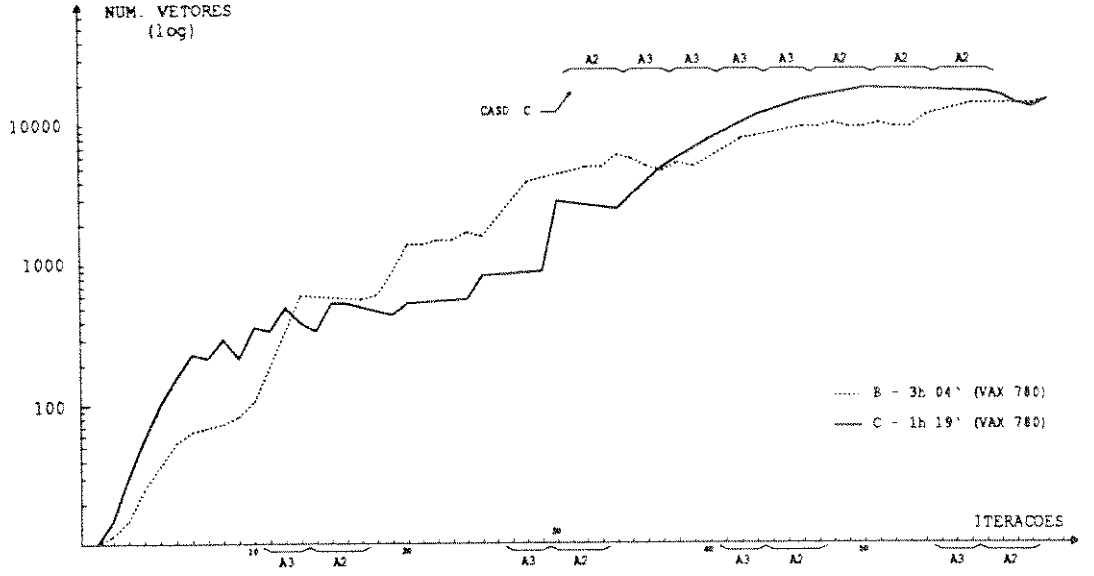

Figura 48: Comparação ULA semi-hierárquica com ordens de processamento diferentes

O resultado pode ser visto na figura 48. Como as duas descrições são semi-hierárquicas, há o mesmo número de iterações para os dois exemplos. Note que inicialmente a opção C leva a um número maior de linhas da tabela de teste do bloco mestre a serem processadas. A partir de um certo ponto porém a situação se inverte. Como a escala das ordenadas é logarítmica, a diferença entre as curvas passa a ser de aproximadamente uma ordem de grandeza. Note que por muitas

iterações o circuito C trata menos linhas de tabelas que o circuito B. O resultado final em termos de tempo de processamento é que a opção C é cerca de 3 vezes mais rápida que a B.

As duas alternativas de ordem de processamento escolhidas, tiveram como único objetivo auxiliar a verificar se era correto esperar que a ordem de processamento influisse significativamente nos resultados. Com a resposta positiva obtida, fica patente a necessidade de se realizar alguns estudos para a definição de métodos heurísticos que orientem um processo de definição da ordem de processamento das células.

3.6.3 Simplificação tipo "fanout look-ahead"

O método de simplificação descrito na seção 3.5.2 foi exercitado também com o uso da ULA. Foi usada a descrição semi-hierárquica que gerou a curva C da figura 48.

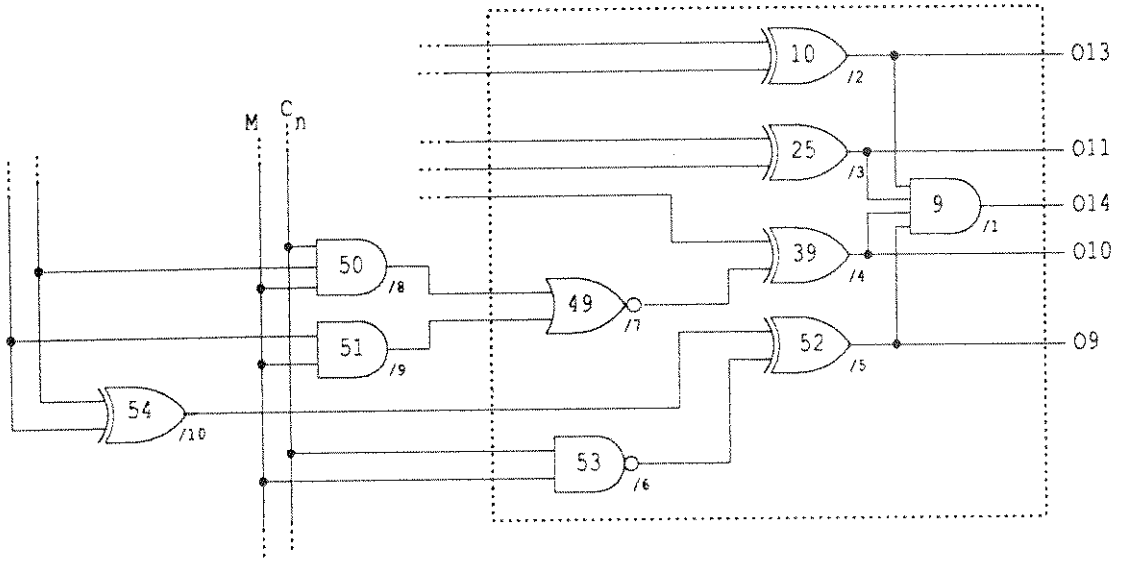

Figura 49: Situação da interface circuito - bloco mestre após a sétima iteração

Na figura 49 pode ser vista a siutuação da interface do bloco mestre com o restante do circuito, após o processamento da porta 49, a sétima a ser tratada. Note que a situação dos fios CN e M é aquela propícia a este tipo de simplificação: ambos são entradas do bloco mestre e de células (50 e 51) que acionam diretamente o bloco.

O resultado da aplicação do "fanout look-ahead" pode ser acompanhado pela figura 50. A curva C refere-se ao experimento sem a simplificação e a curva D ao experimento com a simplificação. Note que a aplicação do método antecipa a eliminação de vetores da tabela de teste após a tratamento da porta 49 (trecho assinalado). Na curva C estes vetores continuam sendo

processados por mais algumas iterações e vão sendo pouco a pouco eliminados, até que após a décima iteração (porta 50), as duas curvas voltam a se igualar. Neste ponto não existe nenhum fio em situação que permita a simplificação. Logo a seguir, porém, o processo volta a se repetir duas vezes (após o tratamento da célula 35 e após a 20) levando a novos trechos onde a curva D contém menos vetores na tabela de teste do que a curva C.

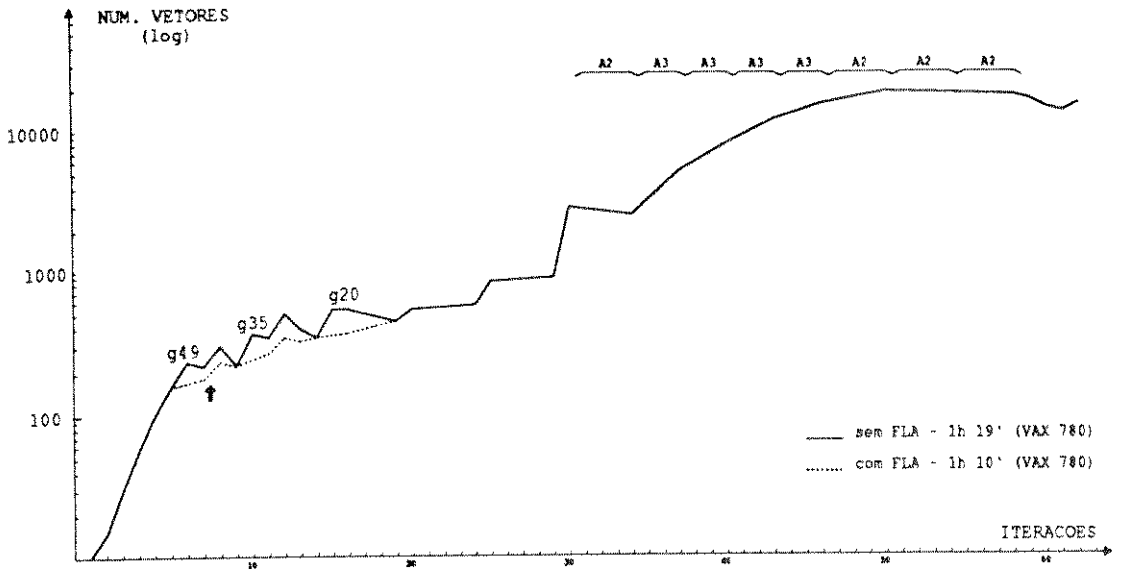

Figura 50: Uso da simplificação tipo "fanout look-ahead" na ULA

O ganho percentual em termos de vetores pré-eliminados nestas três situações foi de 28,3%, 32,7% e 31,3%, respectivamente. O impacto no tempo de processamento foi uma redução de aproximadamente 12% no experimento D, em relação ao experimento C.

3.6.4 Conclusões

Neste experimentos foi possível observar uma proporcionalidade entre o número de linhas das tabelas processadas e o tempo de CPU necessário ao processamento. Na busca de melhor desempenho global, a observação destes experimentos sugere que seja aplicado esforço no sentido de:

- reduzir ao máximo o tamanho das tabelas como meio de minimizar o tempo e a memória alocada
- simplificar as operações que manuseiam as tabelas
- buscar uma solução realmente hierárquica para o problema

4 O algoritmo hierárquico H-ALG

Nesta seção é descrito o H-ALG ([Mendonça 89]). Em 4.1 são feitas algumas considerações sobre o algoritmo. A seguir em 4.2 é feita uma descrição dos passos do algoritmo. As tabelas usadas e a estrutura da biblioteca das primitivas estão descritas em 4.3. Em 4.4 são explicadas as operações básicas do algoritmo, e a seguir, em 4.5 são expostas as simplificações que podem ser efetuadas. Finalmente em 4.6 são apresentados os resultados obtidos a partir de experimentos realizados com os protótipos do I-ALG e H-ALG.

4.1 Considerações iniciais

O H-ALG ([Mendonça 89]) é uma extensão do I-ALG (seção 3). Ele trata circuitos descritos de forma hierárquica, suportando um número indefinido de níveis. Como visto em 2.3.5, este é um campo pouco explorado e as soluções até agora publicadas não garantem a detecção de todas as falhas detectáveis, e nenhuma trata níveis hierárquicos múltiplos.

O crescente número de trabalhos nesta área se justifica pela cada vez maior complexidade dos circuitos digitais. Isto tem impulsionado o uso de metodologias de projeto tipo "bottom-up" e/ou "top-down", nas quais a descrição hierárquica de circuitos é um fundamento. Desta forma tem sido colocado em xeque todo o esforço despendido até aqui na pesquisa de métodos que otimizem os algoritmos baseados no D-ALG (vide item 2.3.3). Esses algoritmos funcionam perfeitamente com circuitos planificados (não hierárquicos). mas esbarram em sérios problemas ao lidar com circuitos hierárquicos (vide 2.3.5).

O H-ALG não faz uso do "backtracking" que, como foi visto em 2.3.3, é um dos grandes senões dos métodos baseados no D-ALG. Quando tratando com circuitos descritos de forma hierárquica estes métodos ou planificam o circuito (ou as partes que estejam sendo tratadas no momento) ou simplesmente ignoram as falhas em nós de nível hierárquico inferior, esperando que elas sejam detectadas pelos vetores gerados para as falhas do nível hierárquico mais alto.

Além de não usar o "backtracking" o H-ALG tem algumas outras características muito interessantes:

- não requer simulação de falhas. Sua estrutura de dados carrega a informação sobre as falhas detectadas.
- pode ser utilizado para aplicar testes pré-calculados a macros internas ("embedded") ao circuito.
- é independente do modelo de falhas

- tolera descrições hierárquicas sem limite de profundidade.

4.2 Descrição geral

O H-ALG usa o mesmo princípio de funcionamento que o I-ALG (vide seção 3). Iniciando por uma saída primária, antes de processar qualquer célula o algoritmo verifica se existem, na biblioteca, dados a seu respeito. Caso as tabelas não estejam definidas, o algoritmo é chamado de forma recursiva e inicia o tratamento desta célula (sub-circuito) a partir da descrição de sua conectividade. Após terminada a execução do algoritmo neste nível, as tabelas de teste e aplicação (vide 4.3) deste sub-circuito são armazenadas na biblioteca e o algoritmo retorna ao nível hierárquico superior. Neste nível o sub-circuito é tratado como sendo uma célula que, agora, já tem seu teste e aplicação definidos.

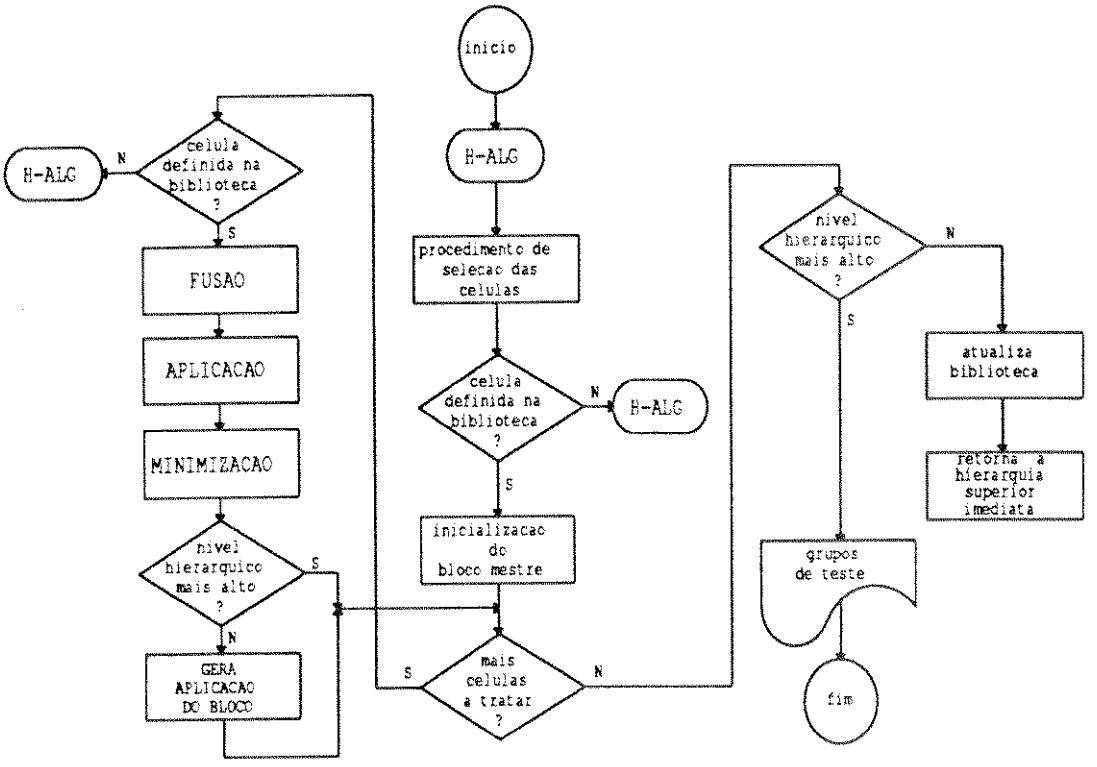

Figura 51: Descrição do algoritmo.

Como será detalhado no item 4.4 o H-ALG difere do I-ALG quanto às operações executadas a cada iteração. No item 4.4.1 é introduzido o conceito de **fusão**, uma operação que simplifica, tanto o processamento a cada iteração, quanto a representação (tabelas) do bloco mestre e de cada célula.

4.3 Tabelas e biblioteca

O H-ALG da mesma forma que o I-ALG, trabalha com tabelas que representam o comportamento da célula e do bloco mestre. As tabelas relativas às células ou foram pré-calculadas e estão disponíveis na biblioteca ou, no caso de uma célula hierárquica, são computadas no decorrer da execução do algoritmo. Nestas tabelas os valores lógicos assumidos pelos fios são representados da forma que se segue:

0- para fios em nível lógico zero.

1- para fios em nível lógico um.

X- para situações onde o nível lógico do fio não interessa à operação (" don't care").

No H-ALG as tabelas da célula são de dois tipos:

- Tabela de teste – $T[C_j]$ é o conjunto de padrões que testam C_j assumindo que se tenha acesso às entradas X_j e à sua saída n_j. $T[C_j]$ é o conjunto dos "primitive D-cube" [Roth 66] para todas as falhas de C_j.

- Tabela de aplicação – $A[C_j]$ mostra todas as maneiras de se controlar n_j para 0's e 1's, a partir das entradas da célula. $A[C_j]$ é similar ao "primitive D-cube" [Roth 66] ou ao "singular cover" [Fujiwara 86], do algoritmo D (D-ALG).

As tabelas associadas ao bloco mestre M_k são computadas a cada iteração para serem usadas na iteração seguinte.

- Tabela de teste – $T[M_k]$ mostra como testar M_k, assumindo que tenhamos acesso a n_j, Y_k e às saídas Z_k (fig. 22).

- Tabela de aplicação – $A[M_k]$ mostra todas as maneiras de se controlar a saída do bloco mestre para 0's e 1's, a partir de suas entradas.

A tabela de aplicação do bloco mestre só é calculada nas ocasiões em que o algoritmo esteja processando uma célula hierárquica (sub-circuito). É necessária para que o H-ALG ao retornar ao nível hierárquico superior monte as informações sobre esse sub-circuito (célula) na biblioteca.

A tabela de teste do bloco mestre é inicializada como cópia da tabela de teste da primeira célula a ser tratada. A cada nova célula incorporada, a tabela do bloco mestre é atualizada pelas operações **fusão e aplicação**, definidas no item 4.4.1. Esta tabela, após serem incorporadas

todas as células, é o próprio conjunto de vetores que testam o circuito. O conceito de grupo de teste, definido em 3.4.1, também se aplica no H-ALG.

4.4 Operações básicas

No H-ALG, devido a sua característica hierárquica, é fundamental que uma célula e o bloco mestre sejam o mais similares possível no que diz respeito à representação de suas tabelas. Ao terminar o processamento de um determinado nível hierárquico inferior, é necessário que as informações sobre teste e propagação deste sub-circuito (o próprio bloco mestre final) sejam encapsuladas em formato compatível com o de uma célula. Estes dados serão utilizados pelo algoritmo no nível hierárquico superior exatamente como se fossem de uma célula.

O objetivo de cada iteração do algoritmo é calcular a tabela de teste do bloco mestre a ser utilizada na iteração seguinte. Quando tratando sub-circuitos, em níveis hierárquicos inferiores, o H-ALG também calcula a tabela de aplicação do bloco mestre. Nesta seção são detalhadas as operações efetuadas a cada iteração.

4.4.1 Geração da tabela de teste $T[M_{k+1}]$

A geração da tabela de teste do bloco mestre, para ser utilizada na iteração seguinte ($T[M_{k+1}]$), é feita em duas etapas:

fusão – entre os grupos de teste do bloco e os da célula que detectem o mesmo tipo de falha no nó de interface.

aplicação – via nó de interface dos grupos tratados pela fusão e que por qualquer motivo não fundiram, além dos demais grupos que não detectam falha no nó de interface.

A seguir são detalhadas estas operações.

Fusão

O conceito de fusão entre os vetores de teste da célula e os do bloco mestre se baseou na observação de que as falhas na saída da célula a ser incorporada são equivalentes (seção 2.1) às falhas na entrada do bloco mestre que é acionada por aquela célula. Trata-se afinal do mesmo fio, o nó de interface.

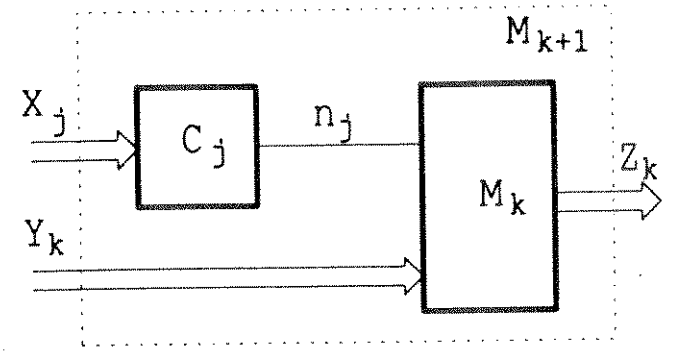

Figura 52: Interface célula-bloco mestre.

Outra observação importante é que os vetores da tabela de teste do bloco, que detectam falhas no nó de interface, já contêm as informações sobre como propagar o erro (resultante da falha) até uma saída primária (Z_k), onde possa ser observado. Por outro lado todos os vetores de teste da célula testam sua saída (o mesmo nó de interface) ou para **f-e-0** ou **f-e-1**. O que a operação fusão faz é a interseção entre os vetores de teste da célula e os do bloco mestre que testam o nó de interface para o **mesmo tipo** de falha. Desta forma será gerado um conjunto de grupos de teste que:

- testa todas as falhas do bloco mestre que inicialmente já eram detectadas pelos seus grupos de teste, e
- testa todas as falhas da célula, testadas pelos seus grupos de teste.

A seguir será visto um exemplo da operação, baseado no circuito da figura 53.

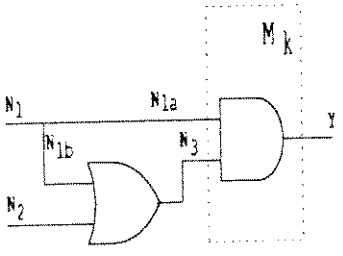

Tabela de Teste do Bloco Mestre: $T[M_k]$

N_{1a}	N_{1b}	N_2	N_3	Y	conjunto de falhas detectadas	grupos
0			1	0	$\{N_{1a}$, Y, s-a-1$\}$	G_1
1			0	0	$\{N_3$, Y, s-a-1$\}$	G_2
1			1	1	$\{N_{1a}$, N_3 , Y, s-a-0$\}$	G_3

Figura 53: Exemplo de fusão.

A inicialização da tabela de teste do bloco mestre é feita através da cópia da tabela da primeira célula a ser processada. Pelo exemplo a seguir pode-se acompanhar o desenvolvimento da operação. Seja o circuito da figura 53 e sua tabela de teste do bloco mestre ($T[M_k]$). Na figura 54 estão as tabelas de teste e aplicação da porta OR2, próxima célula a ser tratada.

Tabela de Teste de C_j : $T[C_j]$

N_{1a}	N_{1b}	N_2	N_3	Y	conjunto de falhas detectadas	grupos
	0	0	0		$\{N_{1b}$, N_2 , N_3 , s-a-1$\}$	G_4
	0	1	1		$\{N_2$, N_3 , s-a-0$\}$	G_5
	1	0	1		$\{N_{1b}$, N_3 , s-a-0$\}$	G_6

Tabela de Aplicação de C_j : $A[C_j]$

N_{1a}	N_{1b}	N_2	N_3	Y	vetores
	0	0	0		A_1
	X	1	1		A_2
	1	X	1		A_3

Figura 54: Tabelas de teste e aplicação da OR2.

A fusão se inicia pela verificação de quais grupos de teste do bloco (fig. 53) detectam falha **f-e-0** no nó de interface (N_3). No caso, apenas G_3 sastisfaz. A seguir verifica-se quais grupos da célula detectam o mesmo tipo de falha neste nó. Na tabela de teste da fig. 54, G_5 e G_6 satisfazem. A fusão entre G_3 e os grupos da célula pode ser vista na figura 55.

	N_{1a}	N_{1b}	N_2	N_3	Y	conjunto de falhas detectadas
G_3	1			1	1	$\{N_{1a}$, N_3 , Y, s-a-0$\}$
G_5		0	1	1		$\{N_2$, N_3 , s-a-0$\}$
G_7	1	0	1	1	1	

	N_{1a}	N_{1b}	N_2	N_3	Y	conjunto de falhas detectadas
G_3	1			1	1	$\{N_{1a}$, N_3 , Y, s-a-0$\}$
G_6		1	0	1		$\{N_{1b}$, N_3 , s-a-0$\}$
G_8	1	1	0	1	1	$\{N_{1a}$, N_{1b} , N_3 , Y s-a-0$\}$

Figura 55: Fusão de G_3 com G_5 e G_6.

O próximo passo é efetuar a fusão entre os grupos do bloco mestre que detectam falha tipo **f-e-1** em N_3 (G_2 apenas), e os grupos da célula que detectam o mesmo tipo de falha: G_4. O resultado pode ser visto na figura 56.

	N_{1a}	N_{1b}	N_2	N_3	Y	conjunto de falhas detectadas
G_2	1			0	0	$\{N_3$, Y, s-a-1$\}$
G_4		0	0	0		$\{N_{1b}$, N_2 , N_3 , s-a-1$\}$
G_9	1	0	0	0	0	

Figura 56: Fusão de G_2 com G_4.

Terminada a operação propriamente dita, basta verificar as inconsistências entre valores de derivações de um mesmo fio. Neste caso o fio com derivação é N_1 que nos grupos G_7 e G_9 apresenta conflito. Estes grupos devem ser eliminados. Pode-se notar que dos grupos de bloco mestre que detectam falhas em N_3, apenas G_3 criou descendência (G_8). Seu conjunto de falhas detectadas é o mesmo de G_3 acrescido das falhas detectadas pelo grupo da célula.

Aplicação

Até aqui só foram tratados os grupos do bloco mestre que testam o nó de interface. Naturalmente existirão outros, que testam as demais entradas do bloco. Estes vetores devem ser aplicados através da célula, para que passem a fazer parte da tabela de teste da próxima iteração. Também devem ser aplicados os grupos que detectam falhas no nó de interface mas que por qualquer motivo (conflito devido a nó com derivação, por exemplo) não foram operados com sucesso na fusão. Estes grupos detectam falhas que agora são internas ao bloco mestre e pode ser que sejam imprescindíveis para esse fim, por isso não devem ser descartados no momento.

	N_{1a}	N_{1b}	N_2	N_3	Y	conjunto de falhas detectadas
V_{10a}	0	X	1	1	0	$\{N_{1a}$, Y, s-a-1$\}$
V_{10b}	0	1	X	1	0	
G_{11}	1	0	0	0	0	

Figura 57: Aplicação dos grupos não fundidos.

No exemplo, os grupos G_1 e G_2 do bloco mestre não fundiram, o primeiro por não detectar falha no nó de interface e G_2 devido a conflitos em N_1. Na figura 57 pode ser visto o resultado da aplicação destes grupos através da célula OR2. O grupo G_1 (fig. 53), para o qual N_3=1, é aplicado por A_2 e A_3 (fig. 54), formando o grupo G_{10} (composto pelos vetores V_{10a} e V_{10b}). G_2, é aplicado apenas por A_1 (N_3=0), gerando o grupo G_{11}. Devido a conflito entre os valores do nó N_1, o vetor V_{10b} e o grupo G_{11} são eliminados.

Montagem de T[M_{k+1}]

A tabela de teste final, que será utilizada na iteração seguinte, é composta pela união dos grupos resultantes da fusão com os resultantes da aplicação. Assim, T[M_{k+1}] pode ser vista na figura 58.

	N_{1a}	N_{1b}	N_2	N_3	Y	conjunto de falhas detectadas
G_8	1	1	0	1	1	$\{N_{1a}$, N_{1b} , N_3 , Y s-a-0$\}$
G_{10}	0	X	1	1	0	$\{N_{1a}$, Y, s-a-1$\}$

Figura 58: Tabela de teste final (T[M_{k+1}]).

4.4.2 Geração da tabela de aplicação $A[M_{k+1}]$

O objetivo desta operação é gerar a tabela de aplicação do bloco mestre, a partir das tabelas de aplicação das células que o constituem. Esta operação é executada para todos os sub-circuitos de forma que ao retornar ao nível hierárquico superior a tabela gerada seja colocada na biblioteca de celulas para uso posterior.

A tabela de aplicação do bloco mestre é inicializada através da cópia da tabela de aplicação da primeira célula a ser processada. Nos demais passos é executada a aplicação desta tabela através da nova célula a ser incorporada. Pela figura 59, pode-se acompanhar a operação para o circuito exemplo (fig. 53). A tabela $A[M_k]$ é, no caso, uma cópia da aplicação do AND2 e $A[C_j]$ é a tabela de aplicação do NOR2.

$A[M_k]$ (aplicação AND2)

	N_{1a}	N_3	S
1	0	X	0
2	X	0	0
3	1	1	1

$A[C_j]$ (aplicação NOR2)

	N_{1b}	N_2	N_3
4	1	X	0
5	X	1	0
6	0	0	1

$A[M_{k+1}]$ (aplicação do novo bloco mestre)

	N_{1a}	N_{1b}	N_2	N_3	S	origem
7	0	X	X	X	0	vetor 1
8	X	1	X	0	0	vetor 2 aplicado por 4
9	X	X	1	0	0	vetor 2 aplicado por 5
10	1	0	0	1	1	vetor 3 aplicado por 6

Figura 59: Exemplo de geração da tabela de aplicação ($A[M_{k+1}]$)

A operação tem início pela verificação do valor que o nó de interface (no exemplo, N_3) exige na tabela do bloco mestre $A[M_k]$. No caso do vetor **1**, este valor é **X**, ou seja, este vetor para fazer S=0, independe do valor aplicado a esta entrada do bloco. Por outro lado, é óbvio que qualquer valor lógico que se colocar nas entradas da célula satisfará o **X** em sua saída (N_3). Logo, N_2 e N_{1b} são definidos como **X** (vetor **7** em $A[M_{k+1}]$). A seguir, toma-se o segundo vetor de $A[M_k]$, e nele N_3=0. Na tabela da célula verifica-se que os vetores **4** e **5** satisfazem esta condição. A operação do vetor **2** com o **4** e de **2** com **5**, gera os vetores **8** e **9**, respectivamente.

Finalmente, toma-se o vetor **3** que exige N_3=1. O único vetor de $A[C_j]$ que satisfaz é o **6**. A operação de **3** com **6** gera o vetor **10**. O último passo é eliminar aqueles vetores nos quais haja conflito de valor em fios com derivação. No exemplo o vetor **10** é o único a ser eliminado. Note que isso significa que é impossível, através das entradas N_1 e N_2 do bloco mestre M_{k+1}, fazer a saída do circuito igual a **1**.

4.4.3 Procedimentos para a seleção de C_j

As considerações feitas na seção 3.4.3 são exatamente as mesmas para o H-ALG. É importante observar que o H-ALG só trata células que tenham apenas uma saída. Esta restrição se aplica, naturalmente, aos sub-circuitos inferiores, uma vez que nos níveis superiores eles serão tratados como se fossem uma célula.

No protótipo do H-ALG foi implementado com êxito, o algoritmo para seleção da ordem de processamento das células descrito na seção 3.4.3.

4.5 Simplificações

O H-ALG admite os mesmos tipos de simplificações que o I-ALG, a menos de algumas considerações feitas a seguir:

- otimização tipo-OU – se aplica sem restrições.
- "fanout-look-ahead" – se aplica sem restrições.
- "freelines" e "headlines" – só se aplica ao nível hierárquico mais alto. O teste resultante desta simplificação não é completo, no sentido de portar todas as opções. Isso deve ser evitado em níveis hierárquicos inferiores, onde é preciso gerar um teste completo para uso no nível superior e/ou inserção na biblioteca.

Nesta seção é tratado um novo tipo de simplificação implementado no protótipo do H-ALG: o corte de vetores.

4.5.1 Corte de vetores

Em testes realizados no protótipo do H-ALG com alguns circuitos, pôde-se verificar que mesmo com cada grupo de teste sendo acidentalmente limitado a apenas um vetor, a cobertura de falhas obtida ainda era significativa. Isto significa que mesmo abandonando-se os demais vetores que comporiam cada grupo, o resultado alcançado continuava cobrindo todas as falhas.

Porém, se limitar o número de vetores em cada grupo por um lado significa abandonar alternativas de propagação, por outro, implica em um grande ganho no tempo de processamento (como será visto adiante).

A partir desta observação foram realizados vários experimentos onde foi limitado o número máximo permitido de vetores por grupo de teste. Quando um determinado grupo, criado a partir

da fusão ou aplicação, tinha uma quantidade de vetores que passava de um número pré-determinado, os demais vetores eram simplesmente abandonados.

Para um determinado circuito, iniciava-se os experimentos limitando-se o número de vetores por grupo em, por exemplo, 50. Repetia-se a geração de teste reduzindo-se sucessivamente este limite, até que o conjunto de vetores gerado deixasse de detectar alguma falha. A experimentação com o integrado 74181 (ULA), mostrou que 9 vetores é o limite inferior para o qual todas suas falhas são detectadas.

Naturalmente, a forma como o corte dos vetores é feito deve influenciar este resultado. Nestes experimentos o método utilizado, por simplicidade, foi o de se manter os primeiros **n** (onde **n** é o número pré-definido) vetores do grupo de teste e eliminar os restantes. Aparentemente o método ideal parece ser a escolha aleatória dos vetores a serem eliminados. A comparação entre estas propostas fica para trabalho futuro. Maiores detalhes dos experimentos realizados podem ser vistos na seção 4.6.

4.6 Resultados

Nesta seção são apresentados alguns resultados obtidos nas experimentações realizadas com o protótipo do H-ALG desenvolvido no CPqD - TELEBRAS. A implementação deste protótipo foi feita, pelo autor deste trabalho, em linguagem "C", com aproximadamente 8000 linhas de código.

Visando melhorar o desempenho do programa, investiu-se um grande esforço em definir uma nova estrutura de dados, em relação à usada no protótipo do I-ALG. Esta estrutura foi desenvolvida em três grandes partes: a representação da conectividade do circuito; a biblioteca com informações sobre as células primitivas; e as tabelas de teste e aplicação. A tabela de teste é, na realidade, representada como sendo um conjunto de trechos do que seria a tabela propriamente. Cada um destes trechos correspondendo a um grupo de teste (seção 3.4.1). Aproveitando a conhecida potencialidade da linguagem adotada, em lidar com apontadores, as três partes da estrutura se referenciam de tal forma que, por exemplo, é possível de um nó do circuito saber que grupos de teste detectam suas falhas, sendo a inversa também verdadeira.

Esta solução pesa, do ponto de vista de memória necessária para a própria estrutura mas, como será visto adiante, foi fundamental para que fosse possível obter resultados mais aceitaveis em termos de tempo de processamento.

4.6.1 Comparação I-ALG x H-ALG

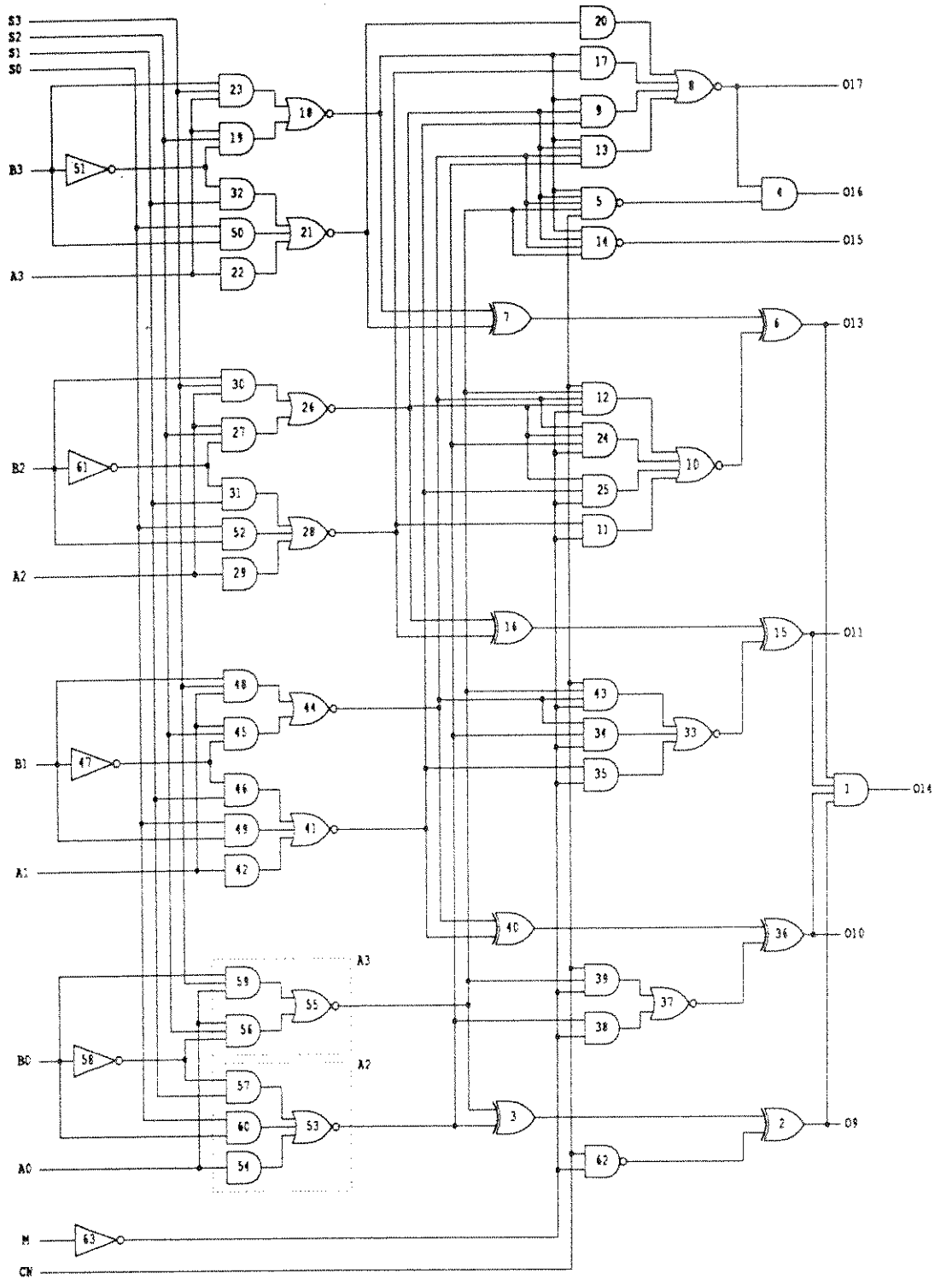

Figura 60: Unidade Lógica Aritmética - ULA (74181).

No primeiro experimento realizado foi processada o circuito integrado 74181, uma Unidade Lógica e Aritmética (vide fig. 60), descrita "flat", para que fosse possível comparar a diferença de tempo de processamento entre os dois protótipos. Para realizar esta comparação foi necessário refazer alguns experimentos já realizados anteriormente e apresentados na seção 3.6, devido a dois motivos básicos:

- Não era possível dispor do VAX 780 usado para aquelas comparações. As medidas passaram a ser feitas em um VAX 8800 (aproximadamente 6 vezes mais rápido).

- Como o ordem de processamento das células passou a ser definida automaticamente através do algoritmo descrito na seção 3.4.3, foi necessário redefinir para o I-ALG esta mesma ordem para que a comparação independesse deste parâmetro. Esta nova ordem de processamento esta definida na figura 60, pela própria numeração das portas.

Na figura 61 pode ser visto o gráfico do número total de vetores de teste a cada iteração do algoritmo. Nela é possível comparar os tamanhos destas tabelas no I-ALG (curva B) e no H-ALG (curva A), além do tempo de processamento de cada versão. Note que embora as tabelas de teste sejam praticamente equivalentes, a diferença no tempo de processamento é significativa. Isto se deve, entre outros fatores, à introdução da operação fusão no H-ALG com a consequente eliminação do processamento da tabela de propagação; e ainda à nova estrutura de dados.

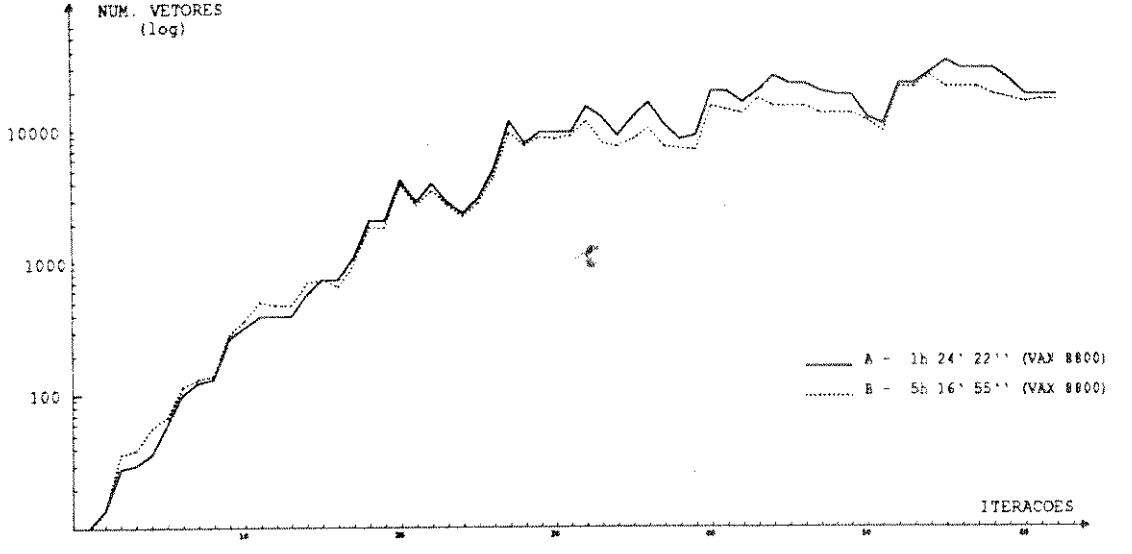

Figura 61: Comparação de desempenho entre os protótipos do H-ALG (A) e I-ALG (B).

Outro experimento realizado visou observar qual a influência desta nova ordem (automática) de tratamento das células, no tempo de processamento. A comparação foi feita entre o resultado do

I-ALG apresentado na figura 61 e o resultado de uma nova execução da ULA, no mesmo VAX 8800, descrita na ordem definida manualmente (figura 44). A ordem definida manualmente resultou em 3h 02'de processamento, contra as 5h 17', resultante de ordem gerada via algoritmo.

Este resultado é até certo ponto surpreendente dada a grande diferença obtida. O mais importante a notar é a influência da ordem de processamento das células, no tempo de máquina. É alentador observar que em se alterando, através de métodos heurísticos, o atual algoritmo de definição da ordem de processamento, de modo que seu resultado se aproxime do alcançado manualmente, será possível obter com o H-ALG resultados ainda melhores que os que são apresentados a seguir.

Convém citar ainda que o protótipo do H-ALG está bem munido de código para auxílio à depuração e instrumentação (contadores de tempos e quantidades de objetos). Com a eliminação desta parte do código em versões futuras, é de se esperar resultados ainda mais positivos quanto ao tempo de processamento.

4.6.2 ULA descrita com hierarquia

Este experimento objetivou comparar o resultado obtido com a descrição da ULA "flat", com o resultado da ULA descrita com hierarquia. A comparação reflete uma situação real onde um mesmo circuito descrito das duas formas (plana e hierárquica) é processado com o próprio algoritmo definindo a ordem do tratamento das células, em um caso e no outro. Na figura 62 está definida a ordem de processamento gerada pelo algoritmo para a ULA descrita com hierarquia. A numeração das portas se refere à figura 60.

```
G01   G39   G12   G05   G61
G06   G33   G16   G55   G47
G15   G35   G04   G53   G58
G36   G34   G08   G44   G63
G02   G43   G20   G26
G62   G10   G07   G18
G37   G11   G09   G41
G38   G25   G13   G21
G03   G24   G14   G51
```

Figura 62: Ordem de processamento das portas para a ULA hierárquica.

É importante observar que ao processar pela primeira vez as células descritas em hierarquia inferior, os vetores de teste tratados pelo algoritmo são apenas os que se referem ao sub-circuito. Embora todos os vetores que faziam parte da tabela de teste no nível superior prossigam ocupando a memória. Isso se reflete nas curvas da figura 63 como os "vales" da curva B (descrição hierárquica).

Um fator influi significativamente para que ocorra um ganho no tempo de processamento entre os dois experimentos, é que ao tratar as outras instâncias destas células o algoritmo realiza apenas uma iteração contra, neste exemplo, 3 ou 4 dependendo da célula, A2 ou A3. O resultado para o H-ALG fazendo uso de sua capacidade de lidar com circuitos descritos com hierarquia, é um tempo de processamento de cerca de 34% do tempo para processar o circuito descrito "flat" (fig. 63).

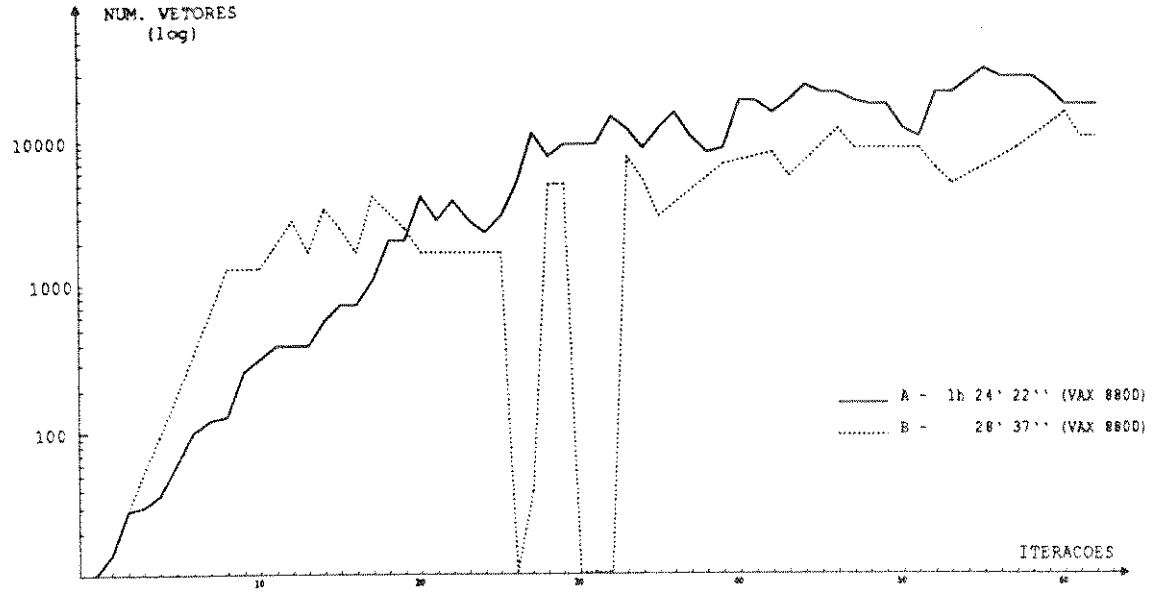

Figura 63: Comparação da ULA "flat" (A) e hierárquica (B).

4.6.3 Corte de vetores

Como foi citado na seção 4.5.1, a possibilidade de mais essa simplificação foi percebida por acaso. Por um erro no programa, todos os grupos ficaram limitados a apenas um vetor. Isso resultou, no caso da ULA, em 15 segundos de processamento. A consequência esperada era uma queda na cobertura de falhas. O que realmente ocorreu. Uma vez sanado o problema do programa, foram realizados vários experimentos que visavam determinar o número mínimo de vetores por grupo, necessário para que todas as falhas continuassem sendo detectadas. Para a ULA o número encontrado foi 9. O que pôde ser observado de mais importante foi o grande ganho no tempo de processamento: o exemplo com o corte definido em 9 vetores foi cerca de 20 vezes mais rápido que o processamento sem cortes (figura 64), sem perda na cobertura de falhas.

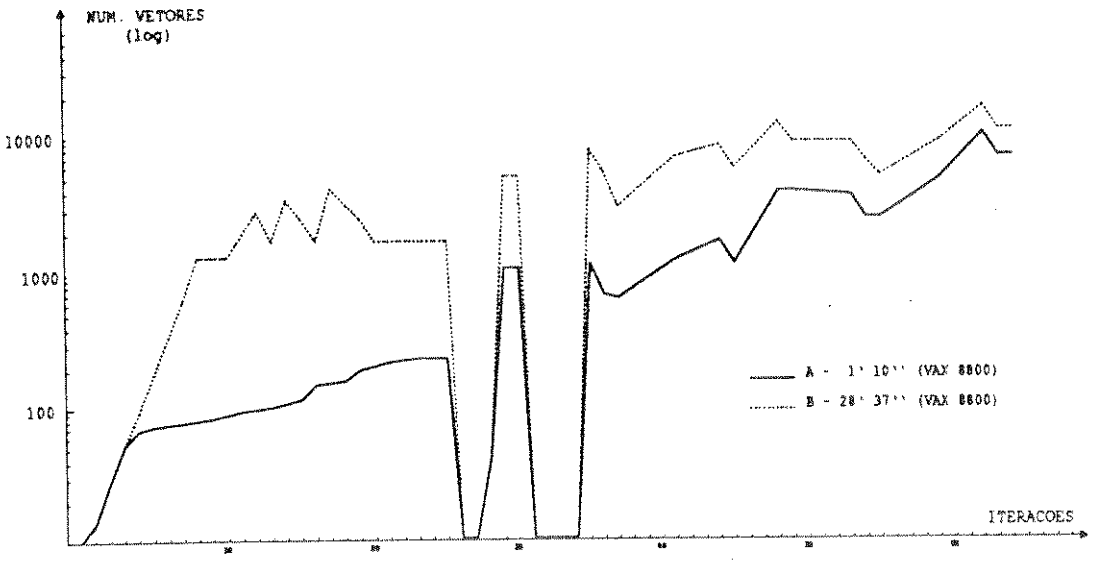

Figura 64: Comparação entre ULA hierárquica com cortes (A) e sem cortes (B).

Outro aspecto interessante que merece ser citado é o perfil dos grupos de teste quanto ao seu número de vetores. No estudo realizado usando-se o processamento da ULA descrita com hierarquia e sem corte de vetores foi levantado o perfil apresentado na figura 65. Nela as linhas se referem a cada iteração e cada coluna, à quantidade de grupos com o mesmo número de vetores. O levantamento foi feito para grupos com 1 vetor cada até 19, e grupos com 20 ou mais vetores.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	>20
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
35	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
38	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
42	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
46	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
50	0	8	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
58	0	8	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
58	0	20	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
78	4	8	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
90	4	8	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
86	4	24	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
106	8	8	12	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
114	8	12	4	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
119	8	12	4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6
119	8	12	7	4	0	0	0	0	0	0	0	0	0	0	0	1	0	0	5
119	8	13	6	4	0	0	0	0	0	0	0	0	0	0	1	0	0	0	5
123	9	13	5	4	0	0	0	0	0	0	0	0	1	0	0	0	0	0	5
128	9	13	5	4	0	0	0	0	0	0	0	0	1	0	0	0	0	0	5
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	8	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
239	133	39	14	40	10	0	0	0	11	0	0	0	0	3	0	0	0	0	8
239	133	39	14	40	10	0	0	0	11	0	0	0	0	3	0	0	0	0	8
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
266	145	34	33	18	13	4	1	0	10	0	1	0	0	0	0	0	0	0	9
298	30	38	6	24	1	0	0	0	1	0	0	0	0	1	0	0	0	0	6
302	38	30	6	24	1	0	0	0	1	0	0	0	0	1	0	0	0	0	6
241	209	33	56	23	1	13	9	3	0	1	0	0	2	0	1	0	0	0	7
459	295	19	56	19	17	8	12	10	10	0	0	0	0	0	1	0	0	1	8
558	63	51	17	31	13	9	7	3	0	0	0	0	0	0	1	1	0	0	7
1191	519	273	79	56	67	24	11	15	2	0	1	0	6	5	0	1	1	0	10
1191	519	273	79	56	67	24	15	11	2	0	1	0	6	5	0	1	1	0	10
1353	515	160	135	48	11	46	9	6	7	4	5	7	2	1	0	0	1	5	10
1387	264	132	56	24	6	13	0	5	0	5	5	3	0	2	0	0	1	0	9
1387	264	132	58	22	11	8	1	4	2	8	1	2	0	2	0	1	0	0	9
1347	639	229	225	28	36	38	22	6	12	14	7	5	2	0	3	2	2	0	12
4441	1382	504	283	184	35	33	48	13	19	8	7	7	6	2	5	3	0	0	19
4452	842	349	89	112	11	6	6	26	9	2	2	1	1	3	0	0	1	1	8
4452	842	349	89	112	11	6	6	26	9	2	2	1	1	3	0	0	1	1	8

Figura 65: Número de grupos com a mesma quantidade de vetores a cada iteração (ULA hierárquica, sem cortes).

A figura 66 mostra este mesmo estudo com as quantidades de grupos separadas por faixas segundo o número de vetores de cada um: 1 a 10 vetores; 11 a 20 vetores e mais de 20 vetores. Ao lado de cada uma destas colunas estão anotados os totais de vetores alocados por faixa, e o respectivo percentual em relação ao total de vetores resultantes da iteração.

Por estas tabelas (figura 65 e 66) pode-se verificar que a maioria esmagadora dos grupos está situada na faixa de 1 a 10 vetores por grupo. A faixa de grupos com número de vetores entre 11 e 20 tem um baixo percentual de grupos e vetores. O surpreendente é a terceira faixa, nela encontram-se pouquíssimos grupos porém, um percentual em geral muito alto de vetores. Ocorre de a **média** de vetores por grupo chegar a 800 !

total de vetores por iteração	1 A 10 vet/grupo: número grupos	total de vetores	%	11 A 20 vet/grupo: número grupos	total de vetores	%	mais de 20 vet/grupo: número grupos	total de vetores	%
5	5	5	100.00	0	0	0.00	0	0	0.00
14	9	14	100.00	0	0	0.00	0	0	0.00
28	13	28	100.00	0	0	0.00	0	0	0.00
52	17	52	100.00	0	0	0.00	0	0	0.00
96	16	16	16.67	5	80	83.33	0	0	0.00
180	20	20	11.11	0	0	0.00	5	160	88.89
344	24	24	6.98	0	0	0.00	5	320	93.02
668	28	28	4.19	0	0	0.00	5	640	95.81
1312	32	32	2.44	0	0	0.00	5	1280	97.56
1315	35	35	2.66	0	0	0.00	5	1280	97.34
1323	39	43	3.25	0	0	0.00	5	1280	96.75
1967	43	47	2.39	0	0	0.00	5	1920	97.61
2931	47	51	1.74	0	0	0.00	5	2880	98.26
1839	59	79	4.30	0	0	0.00	5	1760	95.70
3607	67	87	2.41	0	0	0.00	5	3520	97.59
2619	79	123	4.70	0	0	0.00	5	2496	95.30
1787	95	131	7.33	0	0	0.00	5	1656	92.67
4283	107	143	3.34	0	0	0.00	5	4140	96.66
3309	119	187	5.65	0	0	0.00	5	3122	94.35
2503	135	199	7.95	0	0	0.00	5	2304	92.05
1757	143	207	11.78	0	0	0.00	5	1550	88.22
1777	147	207	11.65	1	20	1.13	5	1550	87.23
1786	150	219	12.26	1	17	0.95	5	1550	86.79
1784	150	218	12.22	1	16	0.90	5	1550	86.88
1784	154	220	12.33	1	14	0.78	5	1550	86.88
1789	159	225	12.58	1	14	0.78	5	1550	86.64
3	3	3	100.00	0	0	0.00	0	0	0.00
10	6	10	100.00	0	0	0.00	0	0	0.00
42	18	42	100.00	0	0	0.00	0	0	0.00
5093	486	1048	20.58	3	45	0.88	8	4000	78.54
5093	486	1048	20.58	3	45	0.88	8	4000	78.54
4	4	4	100.00	0	0	0.00	0	0	0.00
7	5	7	100.00	0	0	0.00	0	0	0.00
12	7	12	100.00	0	0	0.00	0	0	0.00
12	7	12	100.00	0	0	0.00	0	0	0.00
7896	524	1094	13.86	3	52	0.66	7	6750	85.49
5391	398	632	11.72	1	15	0.28	6	4744	88.00
3053	402	628	20.57	1	15	0.49	6	2410	78.94
7124	588	1293	18.15	4	55	0.77	7	5776	81.08
8319	905	1869	22.47	2	35	0.42	8	6415	77.11
5983	752	1284	21.46	2	33	0.55	7	4666	77.99
12888	2237	4457	34.58	14	206	1.60	10	8225	63.82
9104	2237	4453	48.91	14	206	2.26	10	4445	48.82
9381	2290	4227	45.06	25	351	3.74	10	4803	51.20
6819	1887	2827	41.46	16	202	2.96	9	3790	55.58
5243	1889	2839	54.15	15	193	3.68	8	2211	42.17
9392	2582	5184	55.20	35	449	4.78	12	3759	40.02
16330	6942	11901	72.88	38	508	3.11	19	3921	24.01
11420	5902	8579	75.12	11	155	1.36	8	2686	23.52
11420	5902	8579	75.12	11	155	1.36	8	2686	23.52

Figura 66: Número de grupos e vetores separados por faixas (ULA hierárquica, sem cortes)

O corte de vetores como proposto no item 4.5.1 controla esses casos extremos e, na realidade, não afeta um número significativo de grupos. Não afetando, se feito dentro de certos limites, a cobertura de falhas. Pôde-se observar ainda que a diferença do tempo de processamento deste circuito exemplo, com cortes em 10, 20 e 30 vetores, não é significativa (1:10, 1:19 e 1:19 minutos, respectivamente). Isso justifica o uso do método com o corte em um número relativamente alto de vetores, de forma a minimizar o risco de não detecção de falha detectável.

Observou-se ainda, que a redução do número de vetores através do corte resulta também na redução do número de grupos, uma vez que alguns deles deixarão de ser gerados durante a fusão.

4.6.4 Circuitos propostos em [Brglez 85]

Em [Brglez 85] foram definidos alguns circuitos a serem utilizados como base para as comparações de desempenho entre sistemas de GPT. São circuitos que vão de 160 a 3512 portas. Todos eles descritos sem hierarquia.

O atual protótipo do H-ALG se mostrou ineficiente para a solução destes circuitos. O grande entrave foi o espaço de memória necessário ao armazenamento de sua estrutura de dados e das tabelas. Com as limitações impostas pelo processador utilizado, o número máximo de células processadas chegou a 390, ocupando uma área de aproximadamente 75 Mbytes da memória. A medida foi feita com o circuito DATA5 (880 portas), tendo sido especificado em 2 o número máximo de vetores nos grupos. O tempo de processamento chegou a 51'20".

Isso demonstra que o sistema proposto é capaz de solucionar circuitos de porte médio, mas necessita de alterações na sua representação de dados, que o tornem menos exigente de espaço de memória.

4.6.5 Resultados do H-ALG x HILO

A seguir são apresentados os padrões de teste obtidos como resultado do processamento do 74181 pelo H-ALG e pelo HILO.

O HILO é um pacote de software, fornecido pela GenRad, que inclui: simulador lógico, simulador de falhas e gerador de padrão de teste. Seu GPT usa o algoritmo do caminho crítico.

O padrão gerado pelo HILO detecta todas as possíveis falhas tipo fixa-em do circuito e é composto de 24 vetores (figura 67). O tempo de CPU, em VAX 8700, para seu processamento foi de 4 s.

s	s	s	s	a	a	a	a	b	b	b	b	m	c
0	1	2	3	0	1	2	3	0	1	2	3		n
0	1	1	0	1	1	1	0	1	1	1	1	0	1
1	0	1	1	0	0	0	1	0	1	1	0	0	0
1	0	0	1	1	1	1	0	0	0	0	0	0	1
1	0	0	1	1	0	0	1	1	0	1	1	0	0
1	0	1	1	1	1	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	1	1	1	0	0	0	1
1	0	1	1	1	1	0	0	0	0	0	1	1	1
0	1	1	1	0	0	0	1	1	0	0	0	0	0
0	1	1	1	1	1	1	0	0	0	0	0	0	0
0	1	1	0	1	0	0	1	0	1	1	1	0	0
0	1	0	1	1	1	1	1	1	1	0	1	0	0
0	1	1	0	0	1	0	1	0	0	0	1	0	1
0	0	1	0	1	0	1	1	0	1	0	0	1	1
0	0	1	0	1	0	1	1	1	0	0	1	0	0
0	0	1	0	0	1	1	1	1	1	0	0	1	0
0	0	1	0	1	0	1	1	0	0	1	1	0	1
0	0	1	0	1	1	1	1	0	0	0	0	0	0
0	0	1	0	0	0	1	1	0	0	0	1	1	1
1	0	1	0	0	1	0	1	0	0	1	1	0	1
1	0	1	0	0	0	1	1	0	0	0	1	0	1
1	0	1	0	1	0	0	1	0	1	1	1	0	1
1	0	1	0	0	0	0	0	0	1	1	1	1	0
1	0	0	0	1	1	1	1	0	0	0	1	0	0
1	0	0	0	0	0	0	0	1	1	1	1	1	1

Figura 67: Teste gerado pelo HILO

O processamento do mesmo circuito no H-ALG, sem que seja feito qualquer tipo de simplificação, resulta em 5946 grupos de teste. Como cada grupo de vetores de teste detecta um determinado conjunto de falhas, qualquer vetor deste grupo pode ser tomado como seu "representante" no teste final. A partir dos 5946 vetores tomados, é possível efetuar uma simplificação tipo E, que no caso resultou em 166 vetores (tabela 68).

A simplificação tipo E nada mais é que a busca, em todo o universo de vetores (no caso, 5946), de pares de vetores cuja interseção seja não vazia. Neste caso substitui-se o par pelo seu vetor interseção.

Outra simplificação possível, que não foi implementada no protótipo do programa, é a feita pelo método Quine-McCluskey. Esta, além de reduzir o número de vetores ao mínimo necessário, tem como efeito colateral o levantamento do dicionário de falhas do circuito.

s	s	s	s	a	a	a	a	b	b	b	b	m	c
0	1	2	3	0	1	2	3	0	1	2	3		n
x	0	1	x	1	x	x	x	0	x	x	x	1	1
0	0	1	0	0	1	x	x	x	1	x	x	1	1
0	0	0	1	0	1	x	x	x	0	x	x	1	1
0	0	1	x	0	1	x	x	x	0	x	x	1	1
0	0	x	1	0	1	x	x	x	1	x	x	1	1
x	0	1	0	1	1	x	x	1	0	x	x	0	1
0	0	0	1	1	1	x	x	0	1	x	x	0	1
0	1	0	1	1	0	0	x	1	1	1	x	0	x
1	0	1	1	x	0	1	x	1	0	0	x	0	x
0	1	x	1	1	0	1	x	1	1	1	x	0	x
0	0	1	0	0	0	1	x	x	0	1	x	1	1
0	0	0	1	0	0	1	x	x	0	0	x	1	1
1	0	0	x	0	0	0	x	0	0	1	x	1	1
1	0	0	x	0	0	0	x	0	0	0	x	1	1
0	0	1	x	0	0	1	x	x	0	0	x	1	1
0	0	x	1	0	0	1	x	x	0	1	x	1	1
1	0	1	0	0	1	0	x	0	1	0	x	0	0
1	0	0	1	0	1	0	x	0	0	0	x	0	0
0	1	1	0	0	1	0	x	1	1	1	x	0	0
0	1	0	1	0	1	0	x	1	0	1	x	0	0
1	0	1	0	0	1	1	x	0	1	0	x	0	0
0	1	0	1	0	1	1	x	1	0	1	x	0	0
1	0	1	0	0	1	0	x	0	1	1	x	0	0
1	0	0	1	0	1	0	x	0	0	1	x	0	0
0	1	1	0	0	1	0	x	1	1	0	x	0	0
0	1	0	1	0	1	0	x	1	0	0	x	0	0
1	0	1	0	1	1	0	x	1	1	0	x	0	1
0	1	1	0	1	1	0	x	1	1	1	x	0	1
1	0	0	1	1	1	0	x	0	0	0	x	0	1
0	1	0	1	1	1	0	x	0	0	1	x	0	1
x	0	1	0	1	1	1	x	1	1	0	x	0	1
0	x	0	1	1	1	1	x	0	0	1	x	0	1
1	x	1	0	1	1	0	x	1	1	1	x	0	1
x	1	1	0	1	1	0	x	1	1	0	x	0	1
x	1	0	1	1	1	0	x	0	0	0	x	0	1
0	0	0	1	0	0	0	1	x	x	x	0	1	1
0	0	1	0	0	0	0	1	x	x	x	1	1	1
1	0	1	0	1	0	1	1	0	0	1	0	0	x
1	0	1	0	0	1	1	1	0	1	1	0	0	0
x	0	1	0	1	1	1	1	1	1	1	0	0	1
0	0	0	1	1	1	1	1	0	0	0	1	0	1
0	0	0	1	0	0	1	1	x	x	1	0	0	1
0	0	1	1	0	0	1	1	1	1	1	0	0	1
0	0	1	1	0	1	1	x	x	0	1	x	0	1
0	0	x	1	0	1	1	x	x	1	1	x	0	1
0	1	x	1	0	1	1	x	0	1	1	x	0	0
0	1	x	1	0	0	1	x	0	0	1	x	0	0
0	0	0	1	1	1	1	x	1	x	1	x	0	1
0	1	x	1	1	0	1	x	1	0	1	x	0	1
1	0	0	1	x	0	1	1	1	0	0	0	0	x
0	1	0	1	x	0	1	1	x	1	0	0	x	x
1	0	0	1	0	1	1	1	0	0	0	0	0	0
0	1	0	1	0	1	1	1	1	0	0	0	0	0
0	1	x	1	0	0	1	0	1	1	1	0	0	1
x	0	0	1	1	1	1	1	0	0	0	0	0	1

Figura 68: Teste gerado pelo H-ALG

s	s	s	s	a	a	a	a	b	b	b	b	m	c
0	1	2	3	0	1	2	3	0	1	2	3		n
0	0	0	1	0	0	1	1	x	1	1	1	0	1
0	0	1	1	1	1	1	x	0	1	1	x	0	1
x	1	0	1	0	1	x	x	0	0	x	x	0	0
1	x	x	1	0	1	x	x	1	1	x	x	0	0
1	1	x	1	0	1	0	0	x	1	0	0	x	1
1	0	0	1	1	0	0	0	1	0	0	x	0	x
1	1	0	1	1	x	1	0	1	1	0	0	x	1
0	0	x	1	0	0	0	1	x	x	x	1	1	1
0	0	0	1	0	0	0	0	1	0	x	1	1	0
0	x	x	1	0	0	0	0	1	1	1	1	1	0
0	0	0	0	1	0	0	0	x	x	0	0	1	0
0	1	1	1	1	1	x	0	1	0	0	1	1	1
0	0	1	x	0	0	0	1	x	0	1	0	x	x
0	0	1	x	0	0	0	1	1	1	x	0	1	1
x	1	1	x	1	1	0	0	0	1	0	0	x	1
1	x	1	0	0	1	x	x	1	1	x	x	0	0
0	0	1	0	1	1	x	x	1	0	x	x	0	0
x	0	1	x	1	1	x	x	0	0	x	x	0	1
0	0	1	1	1	1	x	x	1	0	x	x	0	1
1	1	x	x	0	0	0	0	0	0	1	0	x	x
0	1	0	0	0	0	1	0	0	0	1	0	x	0
1	1	x	x	0	0	0	0	0	1	1	0	x	x
0	1	x	0	0	1	1	0	0	1	1	0	x	x
1	1	x	x	0	0	0	0	1	0	1	0	x	x
1	1	x	x	0	0	0	0	1	1	1	0	x	x
0	1	x	0	1	0	1	0	1	0	1	0	x	x
0	1	1	0	1	1	1	0	1	1	1	0	1	1
1	x	x	x	0	0	0	0	1	1	1	1	x	x
1	0	0	x	0	0	1	0	1	1	0	1	x	x
1	0	0	x	0	1	0	0	1	0	1	1	x	x
1	0	0	x	0	1	1	0	1	0	0	1	x	x
1	0	1	x	1	0	0	0	0	1	1	1	0	0
1	0	0	1	1	0	1	0	0	1	0	1	1	1
1	0	0	1	1	1	0	0	0	0	1	1	0	1
1	0	0	1	1	1	1	0	0	0	0	1	0	1
0	1	1	0	0	1	1	0	1	0	1	0	0	0
1	0	x	1	0	1	0	0	0	1	1	1	x	x
1	0	0	1	0	1	1	0	0	1	0	1	0	0
1	0	1	x	0	1	0	0	0	0	1	1	0	0
1	0	1	0	0	1	0	1	1	0	0	1	0	0
1	1	x	x	0	0	0	0	0	0	1	1	0	0
1	1	x	x	0	0	0	0	0	1	1	1	0	0
1	1	x	x	0	0	0	0	1	0	1	1	0	0
1	0	1	x	1	0	0	0	0	1	0	x	0	1
1	0	0	1	0	0	0	1	1	1	0	0	0	0
1	0	0	1	0	1	0	1	1	0	0	0	0	0
1	0	1	0	0	1	0	x	0	0	0	x	0	1
1	0	1	0	0	1	0	1	0	0	1	0	0	1
0	0	1	0	0	1	1	1	x	0	1	0	0	1
0	0	0	1	0	1	1	1	x	1	0	1	0	1
0	0	1	0	1	1	1	1	1	1	1	0	0	0
1	0	1	x	0	0	0	1	1	1	1	0	0	0
0	0	0	1	1	1	1	1	0	x	0	1	0	0
0	0	1	0	1	1	1	1	0	1	1	0	0	1
1	0	1	x	1	0	0	1	0	1	1	0	0	1
0	0	0	1	1	1	1	1	1	x	0	1	0	1

Figura 69: Teste gerado pelo H-ALG (continuação)

s	s	s	s	a	a	a	a	b	b	b	b	m	c
0	1	2	3	0	1	2	3	0	1	2	3		n
0	0	0	1	0	1	1	1	x	1	0	0	0	1
1	0	0	1	1	1	0	1	0	0	1	0	0	0
x	0	0	1	1	1	1	1	0	0	0	0	0	0
1	0	0	1	1	0	0	1	0	1	1	0	0	0
1	0	0	1	1	0	1	1	0	1	0	0	0	0
1	0	0	1	0	1	0	1	1	0	1	0	0	0
1	0	0	1	0	1	1	1	1	0	0	0	0	0
1	0	0	1	0	0	0	1	1	1	1	0	0	0
1	0	0	1	0	0	1	1	1	1	0	0	0	0
0	0	0	1	1	1	1	1	1	0	0	0	0	1
0	1	x	0	1	1	0	x	1	1	1	x	0	0
0	1	x	0	1	0	0	x	1	0	1	x	0	0
0	1	1	x	0	1	0	1	0	1	1	0	0	0
0	1	x	1	1	1	0	x	1	1	1	x	0	1
0	1	x	1	1	0	0	x	1	0	1	x	0	1
0	1	1	0	0	1	0	1	0	0	1	1	0	0
0	1	0	1	0	0	0	1	0	0	1	0	0	0
0	1	1	x	0	1	0	0	1	0	0	0	1	1
0	1	x	1	0	1	0	0	1	1	0	0	0	0
0	1	x	1	1	0	0	1	1	0	0	1	0	0
0	1	x	x	0	1	0	1	0	1	0	1	x	x
0	1	x	1	0	0	0	1	0	0	0	1	x	1
1	1	x	x	0	0	0	0	1	1	0	1	x	x
1	1	x	x	0	0	0	0	1	0	0	1	x	x
1	1	x	x	0	0	0	0	0	1	0	1	x	x
1	1	x	x	0	0	0	0	0	0	0	1	x	x
0	1	x	0	1	1	0	0	1	1	0	0	0	0
0	1	x	0	1	0	0	0	1	0	0	0	0	0
1	1	x	x	0	0	0	0	1	1	0	0	0	0
1	1	x	x	0	0	0	0	1	0	0	0	0	0
0	1	x	1	0	1	0	0	0	1	0	0	0	0
1	1	x	x	0	0	0	0	0	1	0	0	0	0
x	1	x	x	0	0	0	0	0	0	0	0	x	x
0	0	1	0	0	1	1	1	x	0	1	1	0	1
0	0	1	0	1	1	1	1	1	1	1	1	0	0
0	1	1	0	1	0	1	1	1	0	1	1	0	0
0	1	1	0	0	1	1	1	0	1	1	1	0	0
0	1	1	0	0	0	1	1	0	0	1	1	0	0
0	0	1	0	1	1	1	1	0	1	1	1	0	1
1	0	1	0	x	0	1	1	1	0	1	1	x	x
0	1	1	0	1	0	1	1	0	1	1	1	0	x
1	0	1	0	0	1	1	1	0	1	1	1	0	0
0	1	1	0	0	1	1	1	1	1	1	1	0	0
0	0	1	0	1	1	1	1	1	1	1	1	0	1
0	0	1	0	0	0	1	1	x	x	0	1	0	1
0	0	1	x	0	0	1	1	1	1	0	0	0	1
0	0	1	x	0	1	1	x	x	0	0	x	0	1
0	0	1	1	0	1	1	x	x	1	0	x	0	1
0	0	1	0	1	1	1	x	1	1	0	x	0	0
1	0	1	x	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	1	x	0	1	0	x	0	1
1	0	1	1	1	0	1	0	0	1	0	0	0	1
0	0	1	1	1	1	1	x	1	1	0	x	0	1
0	0	1	1	0	0	1	1	x	x	0	1	0	1
1	0	1	x	0	0	1	0	0	0	0	1	1	1

Figura 70: Teste gerado pelo H-ALG (final)

5 Conclusões

Dos experimentos realizados podemos concluir que o esforço na busca de uma solução realmente hierárquica para o problema de GPT foi plenamente compensado. Foram alcançados ganhos significativos no tempo de processamento, embora ainda haja muito a ser melhorado, tanto no que se refere ao problema da representação dos vetores de teste quanto a representação da estrutura de conectividade do circuito.

Métodos hierárquicos de projeto vêm sendo aperfeiçoados e seu uso é cada vez mais frequente. Técnicas como o "gate array" e o "standard cells", baseados em bibliotecas de células mais e mais sofisticadas, confirmam este direcionamento. Como resultado do uso destas técnicas, são produzidas descrições hierárquicas dos circuitos, que apresentam algumas vantagens, como: uso de menor área de memória; descrições mais facilmente inteligíveis; etc. Neste trabalho foram apresentados dois novos algoritmos para a geração de teste para circuitos combinacionais. O I-ALG [Côrtes 88] (Algoritmo Incremental) e o H-ALG [Mendonça 89] (Algoritmo Hierárquico), ambos foram desenvolvidos como parte deste trabalho.

O algoritmo I-ALG percorre o circuito desde suas saídas em direção às entradas, processando célula a célula as informações de teste de cada uma delas. Estas informações estão armazenadas em uma biblioteca pré-construida e são totalmente independentes da tecnologia a ser usada na fabricação do circuito. Admite qualquer modelamento de falha, uma vez que os testes podem ser definidos como se deseje.

Diferentemente dos principais algoritmos para geração de padrões de teste publicados ([Roth 66], [Goel 81] e [Fujiwara 83]), o I-ALG não é orientado pelas falhas a serem detectadas. Pode-se dizer que ele é dirigido pela conectividade do circuito. Por ser uma solução completamente diferente das citadas, o I-ALG não esbarra no maior problema com que se deparam aqueles algoritmos: o "backtracking".

O algoritmo hierárquico, H-ALG [Mendonça 89] é baseado no I-ALG, tendo com principal característica sua capacidade de lidar com circuitos descritos com hierarquia. O H-ALG admite qualquer quantidade de níveis hierárquicos. Enquanto tratando um circuito, ao se deparar com uma célula não definida na sua biblioteca, o algoritmo se chama recursivamente e resolve o teste desta célula, a partir da descrição de sua conectividade. Uma vez determinado o teste da célula, o controle retorna ao ponto de onde foi chamada a recursão, e o algoritmo prossegue tratando o circuito no nível hierárquico anterior.

Os tempos para circuitos de porte médio são perfeitamente admissíveis, embora distantes ainda dos referentes a outros métodos publicados. Não cabe porém apenas comparar-se números, o fundamental é que o método proposto cumpre sua finalidade: gera padrões de teste para circuitos combinacionais descritos hierarquicamente.

Seu protótipo encontra-se disponível, embora sem documentação completa.

6 Trabalho futuro

Nesta seção são propostos temas nos quais se recomenda investir esforço no sentido de aprimorar o sistema de geração de padrão de teste apresentado neste trabalho.

- Estrutura de dados:
  - reduzir a representação dos valores lógicos de um byte cada para 2bits cada.
  - aprimorar a estrutura de representação da conectividade do circuito.
- Trabalhar no sentido de solucionar circuitos do tamanho dos propostos em [Brglez 85].
- Aprofundar conhecimentos e observações no sentido de definir e realizar um algoritmo de geração da ordem de processamento que consiga melhor desempenho global (inteligência artificial ?).
- Aprofundar estudos sobre corte de vetores com outros circuitos.
- Implementar método aleatório de corte dos vetores.
- Fazer tratamento especial para regiões "fanout-free" internas ao circuito (verificar possibilidade de simplificações).
- Elaborar uma medida que permita avaliar a "importância" dos grupos, de forma a orientar um método de corte de grupos.

7 Agradecimentos

Aos companheiros Sérgio Augusto de O. Andrade, Nilton Oliveira Jr., e Djalma Salles pelo auxílio direto, apoio nas horas de desespero e suporte no desenvolvimento dos protótipos.

Aos companheiros Ariovaldo V. de Almeida, Davi C. Kerr, Geovane C. Magalhães, José Pedro Junior, Márcia Fiovilli G. Roscito, Valéria Bevilacqua e Vinicius J. Latorre, pelo suporte na confecção deste texto.

Cabe ainda agradecer ao Centro de Pesquisa e Desenvolvimento da TELEBRAS pela oportunidade de desenvolver este trabalho.

8 Bibliografia e referências

[Abramovici 86] M. Abramovici, J. J. Kulikowski, P. R. Menon, e D. T. Miller, "SMART and FAST: Test generation for VLSI scan-design circuits," *IEEE Design & Test*, pp. 43-54, ago. 1986.

[Agrawal 88] V. D. Agrawal e S. C. Seth, *Test Generation for VLSI Chips"*, Los Alamitos, CA; IEEE Computer Press, 1988.

[Armstrong 66] D. B. Armstrong, "On finding a nearly minimal set of fault detection tests for combinational logic nets", *IEEE Trans. on Electronic Computers*, 1966, pp. 66-73.

[Bennetts 75] R. G. Bennetts, D. C. Brittle, A. C. Prior e J. L. Washington, "A modular approach to test sequence generation for large digital networks", *Digital Processes*, 1975, pp. 3-23.

[Bhattacharya 89] D. Bhattacharya, B.T.Murray, J.P.Hayes, 'High-Level Test Generation for VLSI", *IEEE Computer*, vol. 22, No. 4, abr. 89, pp 16-24.

[Brahme 87] D. S. Brahme e J. A. Abraham, "Knowledged based test generation for VLSI circuits", *Proc. 1987 International Conf. on Computer-Aided Design*, nov. 87, pp. 292-295.

[Breuer 76] M. A. Breuer e A. D. Friedman, *Fault Diagnosis of Digital Systems*, Huntington, NY, R. Krieger Publishing, 1974.

[Breuer 85] M. A. Breuer e A. D. Friedman, "Functional level primitives in test generation", *IEEE Trans. on Computers*, mar. 85, pp. 223-235.

[Brglez 85] F. Brglez e H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN", *Proc. 1985 International Symposium On Circuits and Systems*, jun. 85, pp. 663-698.

[Brglez 89] F. Brglez, D. Bryan, J Calhoun, G. Kedem e R. Lisanke, "Automated Synthesis for Testability", *IEEE Trans. on Industrial Electronics*, mai. 89, pp. 263-277.

[Calhoun 89] J. D. Calhoun e F. Brglez, "A framework and method for hierarchical test generation", *Proc 1989 International Test Conference*, nov. 89, pp. 480-490.

[Chandra 87] S. J. Chandra e J. H. Patel, "A hierarchical approach to test vector generation", *Proc 24th Design Automation Conference*, jun. 87, pp. 495-501.

[Côrtes 88] I-ALG: Um algoritmo incremental para geração de padrão de teste, *Anais do III Congresso Bras. de Microeletrônica*, jun. 88, pp. 427-436.

[Daehn 81] W. Daehn e J. Mucha, "A hardware approach to self-testing of large programmable logic arrays", *IEEE Trans. Comp.*, vol. C-30, pp 829-833, nov. 1981.

[Eichelberger 77] E. B. Eichelberger e T. M. Williams, "A logic design structure for LSI testability", in *Proc. Design Auto. Conf.*, jun. 1977, pp. 462-468.

[Fujiwara 81] H. Fujiwara e K. Kinoshita, "A design of programmable logic arrays with universal tests", *IEEE Trans. Comp.*, vol. C-30, pp 823-828, nov. 1981.

[Fujiwara 83] H. Fujiwara, e T. Shimono, "On the acceleration of test generation algorithms," *IEEE Trans. Comp.*, vol. C-32, pp. 1137-1144, dez. 1983.

[Goel 81] P. Goel, "An implicit enumeration Algorithm to generate tests for combinational logic circuits," *IEEE Trans. Comp.*, vol. C-30, pp. 215-222, mar. 1981.

[Johansson 83] M. Johansson, "The GENESYS- algorithm for ATPG without fault simulation", *Proc. 1983 International Test Conference*, set. 87, pp. 538-546.

[IEEE 89] "Standard test access port and boundary-scan architecture," *Institute of Electrical and Electronics Engineers*, rascunho da norma P1149.1, jun. 89.

[Kirkland 88] T. Kirkland e M. R. Mercer, "Algorithms for automatic teste pattern generation," *IEEE Design & Test of Computers*, jun. 88, pp. 43-55.

[Krishnamurthy 87] B. Krishnamurthy, "Hierarchical Test Generation: Can AI Help ?", *Proc. Intl. Test Conf.*, 1987, pp 694-700.

[Ladjadj 86] M. Ladjadj, J. F. McDonald, D-H Ho e W. Murray Jr., Use of the Subscripted D-ALG in submodule testing with applications in cellular arrays," *Proc. Design Autom. Conf*, 1986, pp. 346-353.

[McCluskey 86] E. J. McCluskey, *Logic Design Principles, With Emphasis on Testable Semicustom Circuits*, Englewood Cliffs, NJ, Prentice-Hall, 1986.

[Mendonça 89] J.Mendonça F. Neto, M. L. Côrtes, S. A. O. Andrade, "Um Gerador hierárquico de padrão de teste", *Anais do IV Congresso Bras. de Microeletrônica*, jun. 89, pp. 441-449.

[Miczo 86] A. Miczo, *Digital Logic Testing and Simulation*, New York, Harper & Row, 1986.

[Murray 88] B. T. Murray e J. P. Hayes, "Hierarchical test generation using precomputed tests for modules", *Proc 1988 International Test Conference*, jun. 88, pp. 221-229.

[Pearl 84] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA, Addison-Wesley, 1984.

[Schneider 67] P. R. Schneider, "On the necessity to examine D-chains in diagnostic test generation – an example", *IBM J. Res. Dev.*,vol. 10, num. 1, 1967, pp. 114.

[Schulz 88] M. H. Schulz, E. Trischler, e T. M. Sarfert, "SOCRATES: A highly efficient automatic test pattern generation system," *IEEE Trans. Comp.-Aided Design*, vol. CAD-7, pp. 126-137, jan. 1988.

[Roth 66] J. P. Roth, "Diagnosis of automata failures: A calculus and a method", *IBM Journal of Reserch and Development*, vol.10, pp. 278-291.

[Sarfert 89] T. M. Sarfert, R. Markgraf, E. Trischler e M. H. Schulz, "Hierarchical test pattern generation based on high-level primitives", *Proc. 1989 International Test Conference*,
out. 89, pp. 470-479.

[Somenzi 85] F. Somenzi, S. Gai, M. Mezzalama, e P. Prinetto, "Testing strategy and technique for macro-based circuits", *IEEE Trans. Comp.*, vol. C-34, pp. 85-90, jan. 1985.

[Trischler 84] E. Trischler, "ATWIG, an automatic test pattern generator with inherent guidance", *Proc. 1984 International Test Conference*, Out. 84, pp. 80-87.

[VanEerdevijk 86] K. J. E. VanEerdewijk e F. P. M. Beenker, "A modular boundary scan implementation, *Philips Reserch Labs Report*, 1986.

[Wagner 88] F. R. Wagner, I. J-Pôrto, R.F. Weber, T. S. Weber, *Métodos de Validação de Sistemas Digitais*, UNICAMP, Campinas, SP, 1988.

[Wadsack 78] R. L. Wadsack, "Fault modeling and logic simulation of CMOS and MOS integrated circuits", *The Bell System Tech. Journal*, vol. 57, pp.1449-1474, mai.-jun. 1978.

[Williams 81] T. W. Williams e N. C. Brown, "Defect level as a function of fault coverage", {\em IEEE Trans on Computers}, vol. C-30, pp. 987-988, dez. 1981.

[Williams 83] T. W. Williams e K. P. Parker, "Design for testability – A survey", *Proceedings of the IEEE*, vol. 71, pp. 98-112, jan. 1983.

ANEXO 1 – Implementação do H-ALG

A seguir são apresentados alguns aspectos da implementação em linguagem "C", do H-ALG.

Estrutura de dados

A forma de representar no computador as variáveis envolvidas no problema de geração de padrão de teste, influi diretamente tanto no tempo de processamento quanto nas necessidades de memória. A busca do ponto de equilíbrio entre o menor tempo de processamento e baixa exigência de memória, não é uma tarefa fácil.

No caso do H-ALG, baseado na experiência obtida com a implementação do I-ALG, optou-se pela definição de uma estrutura de representação do circuito que otimizasse o tempo de processamento. Os principais compromissos assumidos foram:

1. evitar ao máximo a procura em filas,

2. Atender a necessidade de percorrer o circuito tanto na direção entradas saídas quanto das saídas para as entradas.

A estrutura adotada está apresentada de forma gráfica e resumida na figura 71, foi criada de forma a melhor aproveitar a facilidade de se manusear "ponteiros" na linguagem adotada.

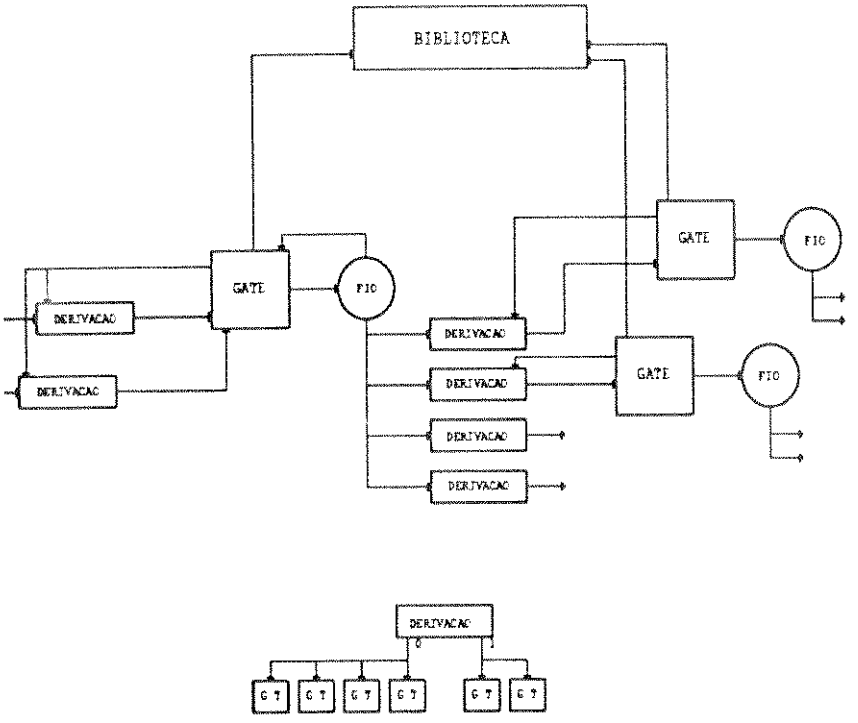

Figura 71: Representação do interrelacionamento das estruturas de dados

A representação dos vetores de teste é um problema crítico neste tipo de programa. Se por um lado a estrutura que representa cada circuito tem um tamanho fixo, por outro lado, os vetores de teste variam em quantidade e comprimento ao longo do processamento. Na versão atualmente disponível cada posição do vetor (correspondente ao estado lógico de um fio do circuito) é representada por um número inteiro, ocupando no caso do VAX um byte (8 bits). Como cada vetor tem tantas posições quantos são os fios do circuito, esta forma de representação não é econômica em termos de memória.

São três os valores lógicos que um fio pode assumir: **0**, **1** ou **x**. É possível portanto, representar seu estado em apenas dois bits da máquina, reduzindo-se em quatro vezes a área de memória necessária.

Outra forma de minimizar a necessidade de memória é reduzindo-se o comprimento dos vetores. Isto é possível já que o processamento do H-ALG é feito sempre entre uma célula e o bloco mestre, não importando o restante do circuito. Assim, o vetor pode ser representado de modo que seu comprimento varie em função do número de entradas do bloco mestre a cada iteração.

A implementação destas alterações na representação dos vetores está sendo estudada. Na figura 72 estão listadas as diversas estruturas usadas no programa.

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

struct Func {                                   /* Definicao da funcao na biblioteca */
   int  nInputs;                                /* numero de entradas da funcao */
   char *pName;                                 /* nome pelo qual e chamada no circuito */
   Logical Filled;                              /* se true => existe o teste da funcao na biblioteca */
   struct Group *pFirstGroup0,                  /* grupos de teste da funcao que detectam f-e-0 na sua
                                                                                        saida */
                *pFirstGroup1;                  /* grupos de teste da funcao que detectam f-e-1 na sua
                                                                                        saida */
   struct VectLink *pFirstApplic0,              /* vetores de aplicacao que fazem a saida da
                                                                                        funcao = 0 */
                   *pFirstApplic1;              /* vetores de aplicacao que fazem a saida da
                                                                                        funcao = 1 */
   struct Func *pNext;                          /* encadeador para formar a biblioteca */
};

struct Group {                                  /* grupo de teste */
   int Id;                                      /* identificacao para debug */
   Logical Colour;                              /* se true => ja foi tratado na iteracao corrente */
   struct Group *pNext,                         /* encadeador para formar lista de grupos */
                *pLast;                         /* encadeador para formar lista de grupos */
   struct VectLink *pFirstVectLink;             /* link para lista de vetores associados a este grupo */
   struct BranchLink *pFirstBranchLink0,        /* link para lista de derivacoes onde o grupo
                                                                                        detecta f-e-0 */
                     *pFirstBranchLink1;        /* link para lista de derivacoes onde o grupo
                                                                                        detecta f-e-1 */
   struct FaultTree *pFaultTree0,               /* link para arvore de nos do circuito onde o grupo
                                                                                        detecta f-e-0 */
                    *pFaultTree1;               /* link para arvore de nos do circuito onde o grupo
                                                                                        detecta f-e-1 */

};

struct VectLink {
   struct VectLink *pNext;                      /* encadeador para lista de vetores */
   char *pVect;                                 /* vetor de teste ou aplicacao */
};

struct Gate {                                   /* instancia de porta logica presente no circuito */
   int Status;                                  /* usado no algoritmo de selecao da ordem de
                                                                                        processamento */
   char *pName;                                 /* nome da instancia usado no arquivo de entrada */
   struct Func *pFunc;                          /* ponteiro para a funcao da porta */
   struct Node *pOutput;                        /* ponteiro para o no de saida da porta */
   struct Branch **pInput;                      /* derivacoes conectadas as entradas da porta */
};


struct Node {                                   /* nos do circuito */
   char *pName;                                 /* nome do no usado no arquivo de entrada */
   struct Node *pNext;                          /* encadeador para formar uma lista de nos */
   int Id;                                      /* identificador numerico */
   struct Gate *pDriver;                        /* ponteiro para a porta que aciona este no */
   struct Branch **pDriven;                     /* lista de ponteiros para as derivacoes deste no */
   int FanOut;                                  /* numero de derivacoes */
   int Status;                                  /* tipo do fio: INPUT, OUTPUT ou INTERNAL */
};


struct FaultTree {                              /* arvore de falhas */
   struct FaultTree *pSameLevel,                /* ponteiro para  "galho" do mesmo nivel */
                    *pBottom;                   /* ponteiro para  "galho de nivel inferior */
   char *Instance;                              /* nome do sinal no arquivo de entrada */
   struct Group *pGroup;                        /* ponteiro para o grupo onde esta pendurada
                                                                                        a arvore */

};

struct GroupLink {                              /* link derivacao -> grupo */
   struct GroupLink *pNext;                     /* encadeador para formar uma lista de links */
   struct BranchLink *pBranchGroupLink;         /* ponteiro para um dos BranchLink associados a um */
                                                        /* grupo que detecta falhas no Branch ao qual
                                                                        este link esta associado */
   struct Group *pGroup;                        /* ponteiro para o grupo ao qual esta associado o */
                                                /* BranchLink apontado pelo pBranchGroupLink */

};

struct Branch {                                 /* derivacao */
   struct GroupLink *pFirstGroupLink0,          /* lista dos grupos que detectam f-e-0 nesta
```

```
                                                                    derivacao */
                    *pFirstGroupLink1;      /* lista dos grupos que detectam f-e-1 nesta
                                                                    derivacao */
   struct Gate *pDriven;                    /* porta acionada por esta derivacao */
   struct Node *pDriver;                    /* no que aciona esta derivacao */
   struct Branch  *pNext;                   /* encadeador para formar uma lista de derivacoes */
};

struct BranchLink {                         /* link grupo -> derivacao */
   struct BranchLink *pNext,                /* encadeador para formar uma lista de derivacoes */
                     *pLast;                /* encadeador para formar uma lista de derivacoes */
   union {
      struct GroupLink *pGroupBranchLink;   /* link para grupo que detecta falha nesta
                                                                    derivacao */
                                            /* usado na estrutura de circuito */
      int Offset;                           /* identificador do numero da entrada da funcao */
                                            /* usado na biblioteca */
   }U;
};

struct Applic {                             /* vetor de aplicacao. Usado apenas na biblioteca */
   char *pVectApplic;                       /* vetor propriamente dito */
   struct Applic *pNext;                    /* encadeador para formar uma lista de vetores */
};


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

Figura 72: Listagem das estruturas usadas no H-ALG