JOSÉ ARTUR QUILICI GONZALEZ

**ALGORITMOS DE OTIMIZAÇÃO DE PLANOS DE TESTE
DE UNIDADES FUNCIONAIS PARA CIRCUITOS BIST**

Dissertação apresentada à Escola
Politécnica da Universidade de São Paulo
para obtenção do título de Mestre em
Engenharia

Área de Concentração:
Engenharia Eletrônica

Orientador:
Wang Jiang Chau

São Paulo
2001

Gonzalez, José Artur Quilici.

Algoritmos de otimização de planos de teste de unidades funcionais para circuitos BIST. São Paulo, 2001.

117p.

Dissertação (Mestrado) – Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Eletrônica.

1. Circuitos Integrados Digitais – Teste. 2. CAD - Teste. I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Eletrônica II. t

À memória de meus pais, Antonieta e Arthur, incansáveis trabalhadores, que empreenderam um esforço admirável para garantir o estudo de seus filhos.

AGRADECIMENTOS

Ao amigo e orientador, Prof. Dr. Wang Jiang Chau, pela sua didática ao me transmitir parte de seus conhecimentos e pela sua postura inteligentemente compreensiva.

Ao Prof. Dr. Marius Strum, com quem freqüentemente dialogo, ora sobre alguns dos temas tratados nesta Dissertação e ora sobre temas mais abstratos.

Ao Dr. Ricardo Pires, que me indicou o aplicativo LEDA e me ensinou os primeiros passos para a sua utilização.

À CAPES, pela concessão da Bolsa de Estudo e à COSEAS, por meio da Assistente Social Fátima, que me apoiou com a Bolsa Moradia do CRUSP.

Aos meus irmãos e irmãs, aos meus amigos e amigas, pelo companheirismo, sem o qual a vida não teria muita motivação.

SUMÁRIO

RESUMO	6
ABSTRACT	7
1 INTRODUÇÃO	8
1.1 Teste Externo.....	9
1.2 Autoteste Incorporado ou BIST.....	10
1.3 Síntese Visando Testabilidade e Síntese Estrutural para Teste.....	11
1.4 Funcionamento no Modo Teste BIST.....	12
1.5 Objetivos e Contribuição.....	14
1.6 Organização desta Dissertação.....	16
2 TRABALHOS RELACIONADOS	17
2.1 Objetivo.....	17
2.2 HLS e Testabilidade.....	18
2.3 Built-In Self-Test.....	27
2.4 Liner Feedback Shift Register.....	31
2.5 Cellular Automata.....	34
2.6 Por que BIST Associado à Síntese de Alto Nível?.....	35
2.7 Implementação de Síntese para Testabilidade.....	37
2.7.1 Projeto Arquitetural Simultâneo à Síntese Estrutural para Testabilidade.....	38
2.7.2 Síntese Estrutural para Testabilidade Incorporada Após o Projeto Arquitetural.....	40
2.8 Resumo.....	44
3 ANÁLISE, PROJETO E IMPLEMENTAÇÃO DO SISTEMA	45
3.1 Os Algoritmos e o Ambiente de Teste	45
3.2 Fluxograma dos Algoritmos para Síntese Estrutural BIST.....	47
3.3 Fluxograma da Métrica Série.....	50
3.4 Fluxograma da Métrica Paralela.....	60
3.5 Os Algoritmos e Sua Implementação.....	74
3.6 Análise, Projeto e Implementação dos Programas BIST_S e BIST_P.....	79
3.6.1 Metodologia Baseada em Objetos.....	79
3.7 Resumo.....	80
4 APRESENTAÇÃO E ANÁLISE DOS RESULTADOS EXPERIMENTAIS	82
4.1 Resultados com o Algoritmo Série.....	82
4.2 Resultados com o Algoritmo Paralelo.....	92
4.3 Resumo.....	99
5 CONCLUSÃO	100
5.1 Considerações Finais.....	100
5.2 Futuros Desenvolvimentos.....	102
6 REFERÊNCIA BIBLIOGRÁFICA	103
APÊNDICE 1 – GLOSSÁRIO	108
APÊNDICE 2 – Resultados Obtidos pela Aplicação da Metodologia Baseada em Objetos	110

RESUMO

Grandes saltos tecnológicos viabilizaram a integração de circuitos digitais de alta complexidade, com centenas de pinos e milhões de transistores. Sistemáticamente, dispositivos eletromecânicos estão sendo substituídos por Circuitos Integrados (CIs) que contêm sistemas inteiros, ampliando o uso generalizada da eletrônica. Com o aumento da complexidade e quantidade de CIs produzidos, a tarefa de detectar de forma rápida e eficiente aqueles *chips* com problemas assumiu grande importância.

Como a Testabilidade [McClu 86] de um CI afeta sua qualidade, um circuito que não é completamente testável, para um determinado modelo de falha, tem menos valor que outro inteiramente testável [De Mi 94]. Em estudos sobre confiabilidade, desempenho, custos e Testabilidade de circuitos VLSI verificou-se que o custo associado ao processo de testes de circuitos VLSI estava praticamente estabilizado, enquanto outros componentes do preço final do *chip* caíam. Por estas razões, a Testabilidade foi incorporada ao projeto desde suas concepções iniciais, apresentando geralmente resultados com menor Sobreárea e mínimo impacto no desempenho, quando comparado a CIs produzidos sem considerações relacionadas a testes.

Uma técnica de teste que dispensa o uso de Equipamento Automático de Teste, conhecida como Autoteste Incorporado (em inglês, Built-In Self-Test – BIST), consiste em adaptar partes do próprio CI para gerar Vetores de Teste, comprimir e analisar os resultados. A técnica BIST tem sido empregada com sucesso em ambientes de projeto de Síntese de Alto Nível (High Level Synthesis, HLS), que tende a reduzir o tempo de projeto de um ASIC, auxiliando a determinação da arquitetura RTL [Stru et al. 99]. Nesta dissertação, considera-se uma forma específica de BIST, o *structural off-line* BIST [Abr et al. 90], em que o Autoteste se dá quando o CI digital é retirado de sua operação normal e colocado em modo teste, para que um Plano de Teste para Unidades Funcionais seja executado (na fase de manufatura e/ou de serviço em campo).

O Plano de Teste, baseado em uma descrição estrutural do CI, é gerado por um algoritmo concebido para detectar os registradores que devem ser reconfigurados em Geradores de Padrões de Teste (em inglês, Test Pattern Generators, TPGs) e Analisadores de Assinatura (Signature Analyzers, SAs). O critério de seleção dos registradores baseia-se numa Função Custo, que avalia globalmente o grau de compartilhamento de cada registrador candidato, e a contribuição que a sua eventual escolha causaria no tempo final de teste. Os Registradores de Teste, reconfigurados em Autômatos Celulares (Cellular Automata, CAs), devem operar segundo um “*Rule Number*”, neste caso, Regra 90 ou 150.

A tarefa do Autoteste paralelo é dividida em duas etapas: na primeira, a cada uma das Unidades Funcionais é associado o melhor momento possível para início de sua sessão de teste, resultando na construção gradativa de uma Matriz de Estado de Teste, e na segunda, com a Matriz de Estado de Teste já totalmente definida, é feita uma análise global para minimizar o número de candidatos a registrador de teste.

O resultado final é um Plano de Teste Otimizado definindo as regras dos TPGs (TPG90 e/ou TPG150), o custo em termos de Sobreárea, o início da geração dos Vetores de Teste, sua duração e a Cobertura de Falha mínima.

ABSTRACT

Great technological developments have made possible the integration of digital circuits of high complexity, with hundreds of pins and million of transistors. Electromechanical devices are being systematically substituted for Integrated Circuits (ICs) that contain complete systems, extending the generalized use of electronics. With the increasing complexity and amount of ICs, the task of detecting faulty chips in a fast and efficient way has assumed great importance.

As testability [McClu 86] affects the quality of an IC, a circuit that is not fully testable for one determined fault model has less value than another which is entirely testable [De Mi 94]. It was verified in studies on reliability, performance, costs and testability of VLSI circuits that the cost associated with tests was practically stabilized, while other components of the final price fell. For these reasons, testability was incorporated in the design since its initial conception, producing generally minor area overhead and minimum impact on the performance, when compared to ICs produced without test considerations.

One test technique that eliminates the needs for use of Automatic Test Equipment, known as BIST - Built-In Self-Test, consists of adapting parts of the IC to generate Test Vectors, compress and analyze the results. The BIST technique has been used successfully in design of HLS - High Level Synthesis - environments, which tends to reduce the ASIC's design time, easing the task of determining the RTL architecture [Stru et al. 99]. In this dissertation a specific form of BIST, the structural off-line BIST [Abr et al. 90], is considered. To start the self-test it is necessary to halt the normal operation of the digital IC and put it in test mode, so that a Test Plan for Functional Units is executed (in the phase of manufacture and/or in the field).

The Test Plan based on the structure of the IC is generated by an algorithm conceived to detect the registers which will be reconfigured in TPGs - Test Pattern Generators, and SAs - Signature Analyzers. The criterion for selection of the registers is based on a Cost Function, which globally evaluates the degree of sharing of each register candidate, and the contribution that its eventual choice would cause in the final test time. The Test Registers reconfigured in CAs - Cellular Automata - must operate according to a "Rule Number", in this case Rule 90 or 150.

The task of the parallel self-test is divided into two stages: in the first one, to each Functional Unit is associated the best possible moment for its test session beginning, resulting in the gradual construction of a Test Status Matrix, and in the second, with the Test Status Matrix completely defined, a global analysis is made to minimize the number of register candidates.

The final result is an Optimized Test Plan, which defines the rules of the TPGs (TPG90 and/or TPG150), the area overhead cost, the beginning of the Test Vector Generation, its duration and the minimum Fault Coverage.

1 INTRODUÇÃO

Várias conseqüências podem derivar de uma falha elétrica num CI, como prejuízos econômicos, eventualmente desconfiças em relação a determinado fabricante ou até mesmo, nos casos mais extremos, riscos a muitas vidas (por ex., com o funcionamento incorreto de equipamentos médicos). Por isso, a detecção de falhas em CIs defeituosos tornou-se uma preocupação constante ao longo de seu ciclo de produção, ressaltando a importância de testes.

Dada a atual complexidade dos CIs, com circuitos cada vez maiores, porém com uma relação decrescente entre o número de conexões externas e o número de transistores, a Testabilidade incorporou-se ao projeto, pois realizar testes sem um planejamento minucioso encarece muito o produto.

Para equacionar problemas desta natureza, várias técnicas de teste têm sido propostas, às vezes com a ajuda de um Equipamento Automático de Teste, que aplicam Vetores de Teste e analisam os resultados, porém com um ônus muito grande devido ao preço destes equipamentos e com taxa de aplicação de teste lenta, causada pelas conexões *off-chip* com o equipamento [Raj 92].

Outras vezes, partes do próprio CI são adaptadas para a geração de Vetores de Teste, compressão e análise de resultados, ocorrendo, então, um Autoteste¹ do CI, i.e., BIST (do inglês, Built-In Self-Test).

¹ Aqui, o termo Autoteste é usado em sentido amplo [Fuj 86], não significando a capacidade total de um circuito checar todas as suas partes. Essa estratégia de teste tem obtido grande aceitação, porque as áreas do circuito não controladas pela técnica BIST, geralmente são pequenas e acessíveis através de pinos. Por ex., o microprocessador PowerPC 750, um processador RISC de terceira geração, 300 MHz, possui 6,35 milhões de transistores (67mm^2), dos quais 5,58 milhões são testados pela técnica BIST, resultando numa Cobertura de Falha de 100%, com Sobreárea estimada de 5% a 8%. Os transistores restantes são testados funcionalmente através do conjunto de instruções, elevando a Cobertura Total para 99,75% [Pyr et al. 98].

1.1 Teste Externo

Duas formas de teste externo de um CI, utilizando a técnica de *Design for Testability (DFT)*, se tornaram muito comum entre os fabricantes: Teste Randômico e Teste Determinístico.

No Teste Randômico [Fuj 86], também conhecido como Teste Comparativo [Abr et al. 90], são aplicados Vetores de Teste gerados randomicamente em dois CIs, um deles sabidamente isento de falhas, conhecido como *Gold Circuit* e outro a ser testado. Se as respostas dos dois coincidirem, então o CI sob teste está aprovado.

Fig. 1.1 – Esquema para o Teste Randômico [Fuj 86]

Embora o Teste Randômico da forma esquematizada pela Fig. 1.1 seja bastante rápido, a dependência de um *Gold Circuit* e a dificuldade para determinar a Cobertura de Falha, geralmente obtida numa Simulação de Falha, acabam se tornando desvantagens consideráveis.

No Teste Determinístico, fortemente dependente de *software*, todos os Vetores de Teste precisam ser conhecidos assim como as respostas esperadas. Independentemente do processo de geração dos Vetores de Teste, que podem ser gerados manualmente ou automaticamente através de programas de teste, a sequência em que eles são injetados permanece inalterada porque os Valores de Referência,

previamente armazenados, pressupõem uma determinada ordem dos Vetores de Teste, como ilustra a Fig. 1.2.

Fig. 1.2 – Esquema para o Teste Determinístico [Fuj 86]

1.2 Autoteste Incorporado ou BIST

O Autoteste Incorporado ou BIST atenua duas grandes dificuldades do teste externo: o acesso às partes internas do CI e o uso de Equipamentos Automáticos de Teste. Outra grande vantagem da incorporação de mecanismos de teste dentro do CI é que o teste pode ser feito como uma atividade especialmente preparada para tal, neste caso sendo tratado como BIST *off-line*, ou durante as operações para as quais ele foi projetado, neste caso sendo tratado como BIST *on-line*.

No contexto deste trabalho, como uma das possíveis alternativas ao teste externo, ilustrado pela Fig.1.3 [Ama et al. 97a], BIST deve ser entendido como *structural off-line* BIST [Abr et al. 90], i.e., o Autoteste do CI digital deve ocorrer quando ele não estiver desempenhando suas funções normais, para que um Plano de Teste, baseado numa descrição estrutural do CI, seja executado (ao final da fase de manufatura e/ou de serviço em campo).

A técnica para redução da Sobreárea, dentro da estratégia geral BIST para lógica combinatória (Unidades Funcionais, na Fig.1.3), é a de reconfiguração parcial de registradores em Geradores de Padrões de Teste (TPGs) para geração de padrões de teste ou em Analisadores de Assinatura (SAs) para compactação das respostas da lógica.

Fig. 1.3 – Estrutura BIST com Reconfiguração Parcial de Registradores em TPG ou SA

1.3 Síntese Visando Testabilidade e Síntese Estrutural para Teste

O aprimoramento da técnica de Síntese de Alto Nível (*High-Level Synthesis, HLS*) [Gaj et al. 92] ou Síntese Comportamental [Stru et al. 99] reflete a necessidade de abordar a implementação de circuitos complexos com um nível de abstração mais elevado, para que as exigências de CIs menores, mais rápidos, com melhor Testabilidade e menor tempo de projeto possam ser atendidas.

A Síntese Arquitetural deve ser entendida como a construção macroscópica da estrutura de um circuito digital, partindo-se de um modelo comportamental que possa ser capturado por um Grafo de Fluxo de Dados (*Data-Flow Graph, DFG*) [De Mi 94]. Tendo a Testabilidade como uma de suas restrições, ainda assim a estrutura com *data path* obtida pode ou não trazer incorporada uma Estrutura de Teste. Em caso negativo, a resolução final do problema do teste é transferida para a fase em que a microarquitetura do circuito está definida, caracterizando-se a Síntese Estrutural para Teste.

Na Síntese Estrutural para Teste, as Estruturas de Teste são introduzidas imediatamente após a HLS, no nível RTL, enquanto que na Síntese Visando Testabilidade as Estruturas de Teste são definidas durante a HLS. Em qualquer um dos casos, as principais restrições de projeto são área, velocidade e Testabilidade.

A solução para o problema do teste discutida aqui se enquadra na Síntese Estrutural para Teste, orientada por um Sistema de Síntese não comercial [De Mi 94] definida em [Ama et al. 97a] e detalhada em [Ama et al. 97b] e [Ama et al. 98].

1.4 Funcionamento no Modo Teste BIST

O domínio de aplicação dos algoritmos discutidos nesta dissertação abrange CIs digitais com *data paths* claramente definidos por uma arquitetura do tipo RTL (*Register Transfer Logic*) gerada por um processo de HLS. Esses CIs devem ter basicamente dois modos de funcionamento: Modo Normal e Modo Teste. Se o CI for colocado no Modo Teste, o procedimento de teste será feito da forma descrita a seguir.

Observando a Fig. 1.3, quando o sinal Run_BIST, externamente controlado pelo pino TST, é acionado, o Controlador de Teste reconfigura alguns dos registradores em TPGs ou SAs, e lhes transfere um padrão de inicialização (ou *seed*), armazenados em uma ROM (ou em uma Máquina de Estados Finitos). Neste momento, o pino FIM é levado ao nível lógico LOW, e o pino GO, ao nível HIGH.

A cada sinal de *clock* transmitido pelo Controlador de Teste, os TPGs geram Vetores de Teste Pseudo-Aleatórios, segundo Regras de Computação previamente definidas, enquanto que os SAs compactam as respostas das UFs.

Após um número pré-determinado de sinal de *clock*, as respostas compactadas são mandadas para o Controlador de Teste através de uma Linha de Scan, que as compara com padrões de resposta armazenados na ROM. Se alguma das UFs se revelar

defeituosa, o pino GO deve sinalizar esse fato mudando de estado e voltando ao normal (i.e., HIGH, LOW, HIGH), para que outras UFs possam ser testadas.

Ao final do Teste BIST, o pino FIM sinaliza o fim dos testes (HIGH) e se o pino GO não acusou nenhuma falha, então o circuito sob teste está aprovado. Caso o pino GO tenha mudado de estado pelo menos uma vez durante o teste das UFs, ele retornará ao estado LOW, indicando que o circuito está reprovado.

Neste modelo de Autoteste, como nem todos os Vetores de Saída das UFs são enviados ao comparador, mas apenas uma versão compactada deles, então o método de teste adotado pode ser entendido como Autoteste compactado, *off-line* e estrutural (baseado num modelo de falha estrutural e não num funcional, ainda que o ambiente de análise seja de alto nível). A geração dos Vetores de Teste é pseudo-randômica (as seqüências apresentam várias características das seqüências randômicas, porém são periódicas e determinísticas), através da reconfiguração do registrador normal em Registrador Linear de Deslocamento Realimentado (*Linear Feedback Shift Register*, LFSR) ou Autômatos Celulares (Cellular Automata – CA) formando uma arquitetura distribuída de teste.

A Cobertura de Falha de um circuito geralmente depende do número de Vetores de Teste aplicados, e esse número de vetores é determinado pelas regras de configuração, ou seja, como os registradores individuais dos CAs ou LFSRs vão ser conectados entre si. Como diferentes regras têm diferentes custos de Sobreárea, uma das restrições da Síntese Estrutural para Teste é o da utilização de Registradores de Teste com menor custo e no menor número possível.

1.5 Objetivos e Contribuição

O objetivo deste trabalho é a Análise, Projeto e Implementação de Algoritmos que propõem modificações no Projeto de um CI genérico, durante a HLS, imediatamente após a elaboração de sua arquitetura em RTL, para que o CI possa realizar o Autoteste descrito na seção 1.4, a um Custo de *Hardware* e Custo de Tempo otimizados, com alta Cobertura de Falhas.

Este trabalho baseia-se na Síntese Estrutural para Teste proposta por [Ama et al. 97a], que parte de uma Descrição Estrutural da arquitetura de um CI e visa obter um teste com determinada Cobertura de Falhas, segundo o modelo de Falha Lógica, *stuck-at*. Mais especificamente, este trabalho visa solucionar heurísticamente o problema da inserção de registradores de teste, sob a técnica BIST, num circuito com *data path* e UFs, com uso de bibliotecas previamente compiladas para cada tipo de UF. Como o problema da escolha do conjunto mínimo de registradores de teste é NP-completo, i.e., não há garantias de que para todos os casos seja possível encontrar esse conjunto num tempo polinomial, pretende-se aplicar a solução heurística em *benchmarks* clássicos objetivando demonstrar que a dificuldade computacional do problema pode ser drasticamente reduzida.

Os principais aspectos deste trabalho podem ser resumidos nos seguintes itens:

- O algoritmo utiliza informações técnicas pré-compiladas, como o número de Vetores de Teste necessários para se atingir determinada Cobertura de Falha, minimizando as Sessões de Teste e o Custo de *Hardware* para a implementação dos Geradores de Teste e Analisadores de Assinatura.
- A Implementação Computacional da Métrica para Testabilidade proposta em [Ama et al. 97a], [Ama et al. 97b] e [Ama et al. 98] detalha a idéia geral de tratamento do problema NP-completo de inserção dos Registradores de Teste através de Funções

Custos. Numa primeira fase, selecciona-se um grupo de registradores com possibilidades de utilização no Modo Teste, para na fase seguinte avaliar os custos de cada combinação global factível, eliminando-se as situações de compartilhamento de um mesmo registrador para duas funções simultâneas proibidas, reduzindo-se drasticamente o número de casos considerados. Na implementação para o Teste Série das UFs, o problema da inserção dos Registradores de Teste é menos complexo, porque o grau de compartilhamento de um registrador constitui um critério de seleção simples, diferentemente da implementação para o Teste Paralelo das UFs, em que se avalia um conjunto de registradores em meio a inúmeras combinações possíveis, exigindo intrincados critérios de escolha.

- Otimização da Geração Pseudo-Randômica dos Vetores de Teste pela adoção de implementações diversas de registradores de teste, incluindo-se aí, a implementação especial de Cellular Automata proposta por [Car et al. 98a].
- A análise dos resultados experimentais mostra que a Solução Heurística proposta para o problema NP-completo de inserção dos Registradores de Teste produz bons resultados para a maioria dos *benchmarks* considerados.

Vale ressaltar que o trabalho original proposto por [Ama et al. 97b] e [Ama et al. 98] consiste de descrição dos procedimentos básicos dos Algoritmos de Geração de Plano de Teste e de Estudos de Caso realizados manualmente. Nesta dissertação, procurou-se completá-lo através da proposição de novos algoritmos, implementação em código de todos os procedimentos, e análise de outros exemplos maiores.

1.6 Organização desta Dissertação

O Capítulo 1 apresenta os objetivos desta dissertação e uma introdução ao problema do teste e Autoteste de CIs. É apresentado também um tipo de BIST *off-line* com algumas inovações, discutidas nos próximos capítulos, como o uso de dados técnicos pré-compilados e uma Métrica de Teste Eficiente para tratar do problema da inserção de Registradores de Teste.

No Capítulo 2, são discutidas a HLS e a Testabilidade, com atenção especial ao BIST e suas formas comuns de gerar padrões de teste através de *Linear Feedback Shift Registers* e *Cellular Automata*. Depois são apresentadas duas formas de incorporação das estruturas de teste no CI: Projeto Arquitetural Simultâneo à Síntese Estrutural para Testabilidade e Síntese Estrutural para Testabilidade Incorporada após o Projeto Arquitetural.

No Capítulo 3, são apresentados os Algoritmos Série e Paralelo, com seus Fluxogramas das Métricas Série e Paralela. A implementação desses algoritmos é discutida com bastantes detalhes, ressaltando a Solução Heurística proposta para o problema NP-Completo da inserção de registradores de teste.

O Capítulo 4 traz os resultados experimentais obtidos com a aplicação da implementação em C++ dos Algoritmos Série e Paralelo. É feita uma análise desses resultados e é mostrado que quando a Testabilidade é incorporada como restrição de projeto desde as primeiras fases, é menor o Custo de *Hardware* pago pela Testabilidade.

No Capítulo 5 são tiradas as conclusões com base nos objetivos propostos e nos resultados obtidos, e enumerados alguns possíveis desdobramentos para a continuidade deste trabalho.

No Apêndice 1 encontra-se um Glossário das principais siglas usadas e no Apêndice 2, os resultados obtidos pela aplicação da Metodologia Baseada em Objetos.

2 TRABALHOS RELACIONADOS

2.1 Objetivo

Para facilitar o entendimento e a classificação de inúmeros trabalhos que apresentam soluções ao problema de teste de CIs, foram selecionadas algumas alternativas representativas de propostas já implementadas e com resultados positivos no âmbito da HLS. Este capítulo começa com uma visão geral das principais abordagens direcionadas à Testabilidade, faz uma recapitulação sucinta da alternativa BIST, discute a aplicação de LFSRs e CAs como mecanismos de teste, para finalmente agrupar os enfoques de reconhecidos autores versados na área de teste.

Duas abordagens de pesquisa se delineiam claramente no que se refere à forma como a Testabilidade é incorporada à metodologia de projeto em HLS para circuitos com *data path*:

1. Projeto Arquitetural Simultâneo com Síntese Estrutural para Testabilidade;
2. Testabilidade Incorporada após a Síntese do Projeto Arquitetural².

Com o constante aumento do poder de processamento dos computadores, a abordagem 1 parece estar prevalecendo sobre a 2. Na abordagem 2, para que a estrutura sintetizada resulte em uma implementação eficiente, é necessário que se leve em conta a interação entre HLS e a Testabilidade do circuito, ou seja, a Síntese do Projeto Arquitetural deve evitar estruturas que dificultem a posterior incorporação de mecanismos de teste.

² Como não há uniformidade na terminologia da literatura especializada relativa à síntese de alto nível para testabilidade, a nomenclatura de [Geb e Elm 92] parece se ajustar melhor aos trabalhos cuja abrangência na HLS envolva mais o domínio estrutural do que o comportamental.

2.2 HLS e Testabilidade

Em geral, os algoritmos de HLS procuram otimizar a área (pela redução do número de UFs, de registradores etc.) e o atraso nos caminhos críticos de uma arquitetura. Ocorre, porém, que uma arquitetura pode ter esses parâmetros otimizados e apresentar baixa Testabilidade. Nesses casos, a implementação de uma metodologia DFT, como o *Scan*, na descrição estrutural do circuito (durante ou após a Síntese Lógica) costuma apresentar um alto custo de Sobreárea, além da possibilidade de degradação de seu desempenho em modo normal.

Para contornar dificuldades dessa natureza, a Testabilidade passou a ser um dos objetivos da HLS, trazendo algumas vantagens, entre as quais, um espaço de solução maior, por ser a Testabilidade considerada num nível mais alto, a alternativa de geração dos Vetores de Teste durante a síntese [Bha e Jha 94] etc.

Há uma aceitação bem generalizada, atualmente, de que a Testabilidade quando adequadamente incorporada à HLS não chega a comprometer os outros objetivos, pois a uma descrição comportamental podem corresponder várias estruturas equivalentes com diferentes formas de teste, sem prejudicar o desempenho do circuito.

[Lee 97] sintetizou essas idéias através da representação da Fig.2.1, em que três eixos ortogonais do espaço hipotético de projeto estão associados à área, ao atraso e à Testabilidade. A figura ilustra a idéia de que numa arquitetura em que a estrutura para teste tenha sido incorporada durante a HLS, a Testabilidade é alta e o impacto sobre a área e o desempenho do CI é pequeno.

A alternativa de incorporação de estrutura de teste após a HLS apresenta duas variantes, qual sejam, HLS sem preocupação com a Testabilidade e HLS voltada para a incorporação posterior de estrutura de teste. A primeira variante está praticamente abandonada, dada a atual complexidade dos CIs, enquanto que a segunda pode

apresentar alguns pontos atrativos em determinadas situações, como se verá durante a explanação específica a essa variante.

Fig. 2.1 – Espaço hipotético tridimensional de um projeto em HLS

Para implementar um algoritmo baseado em processos heurísticos que gerasse arquiteturas com melhor Testabilidade, [Lee 97] apresentou o sistema PHITS (Princeton HI-level Test Synthesis), que se vale da liberdade de escolha apresentada pelas operações de *scheduling* e alocação durante a síntese de *data path* e cria uma arquitetura testável a um baixo custo de Sobreárea.

Basicamente o PHITS procura aumentar a controlabilidade e a observabilidade do circuito, com o objetivo de facilitar a posterior geração automática de padrões de teste determinístico, tanto para a estratégia de *scan* parcial³ como a sem linha de *scan*. Para isto, busca-se reduzir a profundidade e o *loop* sequenciais, e maximizar o

³ Em alguns casos, a decomposição completa de um circuito sequencial em partes combinatórias, ligadas por uma linha de *scan*, não é possível, ou indesejada, seja pela presença de *flip-flops* assíncronos ou pelo custo proibitivo da Sobreárea. Uma solução é criar a linha de *scan* parcial ou incompleta. Veja [Fuj 86].

compartilhamento condicional de registradores durante a alocação. Esses pontos serão aqui discutidos por representarem aspectos comuns à maioria dos sistemas de HLS que agregam a Testabilidade como um de seus objetivos.

Um circuito com alta controlabilidade e observabilidade apresenta suficientes caminhos para ativar suas possíveis falhas e propagar seus efeitos até um ponto de teste, o que em termos práticos resulta em alta Testabilidade. A análise de Testabilidade [Ham et al. 94] pode servir como guia do processo de HLS, identificando os elementos potencialmente difíceis de testar. Como a observabilidade é fortemente dependente da controlabilidade, os problemas de controlabilidade geralmente são resolvidos primeiro⁴.

[Lee 97] propôs o aprimoramento do algoritmo de alocação de registradores para garantir a Testabilidade, por meio da redução da profundidade sequencial do circuito. Um dos algoritmos mais utilizados, talvez pela sua simplicidade e eficácia, para a alocação do número mínimo de registradores durante a HLS é o algoritmo *greedy* (normalmente traduzido como “guloso”, porém “ávido” parece mais adequado) do canto esquerdo [Gaj et al. 92], em que uma tarefa é dividida em ciclos de forma a facilitar a adoção da melhor solução parcial a cada um dos ciclos. Em geral, esse algoritmo apresenta bons resultados com altas probabilidades, mas há inúmeros casos em que o conjunto das soluções locais ótimas não coincide com a melhor solução global⁵.

Quando aplicado na busca do conjunto mínimo de registradores de teste, uma limitação específica desse algoritmo revela-se no fato de ele não levar em conta a Testabilidade do circuito. Por isso [Lee 97] incorporou algumas modificações que, sem comprometer o desempenho do algoritmo, buscam um circuito com um número mínimo

⁴ Por exemplo, garantindo a existência de caminhos de controle (e observação) para cada variável criada durante a alocação de registradores. Veja [Ham et al. 94].

⁵ [Aho et al. 83] apresenta um contra-exemplo de algoritmo para fazer troco com o menor número possível de moedas, sendo que as de maior valor devem ser escolhidas primeiro. Se os valores disponíveis

de registradores, porém com controlabilidade e observabilidade aprimoradas. Como os detalhes dessa implementação fogem ao escopo deste trabalho, apenas um exemplo ilustrativo desses resultados está apresentado na Fig. 2.2.

Fig. 2.2: (a) Exemplo de um quadro com a duração das variáveis; (b) Uma possível alocação de registradores com o algoritmo do canto esquerdo; (c) Outra possível alocação de registradores levando-se em conta a Testabilidade.

A Fig. 2.2 (a) mostra uma tabela com a duração de oito variáveis, sendo $V_I = \{a, b\}$ o conjunto das entradas primárias, $V_M = \{c, d, e, f\}$ o conjunto das variáveis intermediárias e $V_O = \{g, h\}$ o conjunto das saídas primárias.

Como os registradores R_3 e R_4 da Fig. 2.2 (b) não estão ligados a nenhuma entrada ou saída, eles só podem ser acessados através de outros registradores, não sendo, portanto, diretamente controláveis ou observáveis, dificultando a detecção de falhas testadas através de R_3 e R_4 . A Fig. 2.2 (c) ilustra uma solução em que, partindo-se da hipótese de que R_1 e R_2 sejam observáveis através de algum caminho seqüencial, mesmo sem a atribuição a eles das variáveis g e h , então estas podem ser associadas aos registradores R_3 e R_4 , respectivamente, tornando-os observáveis.

das moedas forem de 1, 5 e 11 centavo(s), então para um troco de 15 centavos o algoritmo seleciona 1 moeda de 11 centavos e 4 de 1 centavo, quando 3 de 5 centavos bastariam.

Com isso, todos os registradores passam a ser controláveis e/ou observáveis, melhorando a Testabilidade.

Contudo, a alocação de registradores isoladamente não garante uma alta Testabilidade pois a alocação de UFs (somadores, ALUS etc.) bem como a de interconexões influenciam na profundidade seqüencial de um circuito, i.e., influenciam no número de níveis de registradores no caminho de um sinal, entre as entradas primárias e saídas primárias. (Pode-se mostrar que a extensão da seqüência de testes é linear em termos de profundidade seqüencial (Miczo 86 apud [Lee 97]).)

Para ilustrar esse fato, [Lee 97] apresenta um exemplo, denominado **ex1** e reproduzido na Fig. 2.3, em que, para as três operações mostradas, são alocados dois somadores e quatro registradores, com a seguinte distribuição de variáveis:

$$\mathbf{R} = \{(a, c, g), (b, f), (d), (e)\}.$$

(Supõe-se que cada registrador seja controlável ou observável.)

Há duas possíveis alocações de módulos ou alocações de UFs:

$$\mathbf{M}_1 = \{ (+_{1,3}), (+_2) \} \text{ ou } \mathbf{M}_2 = \{ (+_{2,3}), (+_1) \}.$$

Fig. 2.3: (a) Scheduled Data Flow Graph do ex1; (b) Quadro com a duração das variáveis

Inicialmente, a alocação $\mathbf{M}_1$ parece ser preferível, pois como as operações $+_1$ e $+_3$ recebem um de seus operandos (a e c , respectivamente) do registrador comum R_1 , e também guardam seus resultados (c e g , respectivamente) em R_1 , há uma economia de interconexões.

Nessa implementação, mostrada na Fig. 2.4 (a), tanto os registradores R_1 e R_2 quanto o somador $+_{1,3}$ são fáceis de testar graças à controlabilidade das entradas a , b e à observabilidade da saída g . O mesmo não se pode dizer dos registradores R_3 e R_4 e do somador $+_2$, cuja saída apresenta baixa observabilidade. O efeito de um erro nele precisa se propagar pelo registrador R_2 para ser observado em g , como ilustra o *Data Path Circuit Graph* (DPCG) da Fig. 2.4(b).

Num DPCG, cada nó representa um registrador e cada arco representa uma UF ou Módulo Funcional se conectando a dois registradores. Acrescenta-se também um arco da entrada primária até o nó correspondente ao registrador de entrada, assim como um outro partindo do nó correspondente ao registrador de saída até a saída primária. Essa representação abstrata da arquitetura de um circuito, utilizada por Lee, é similar ao S-graph proposto por Cheng e Agrawal em [Chen e Agra 90].

Fig. 2.4: (a) Uma implementação de ex1 sem levar em conta a Testabilidade; (b) O respectivo DPCG.

O DPCG da Fig. 2.4(b) ilustra claramente uma situação típica em que, se a profundidade seqüencial de um circuito puder ser diminuída (no caso R3 e R4), a Testabilidade será melhorada.

A questão é saber se isso é factível a um baixo custo de Sobreárea. Neste caso específico, e em geral também, a resposta é sim, como ilustra a Fig. 2.5 uma possível alternativa:

Fig. 2.5: (a) Outra implementação de ex1 levando-se em conta a Testabilidade; (b) O respectivo DPCG.

Com uma pequena alteração na associação de variáveis a registradores, o somador +₂ e os registradores R₃ e R₄ que fornecem os operandos para +₂ são agora facilmente observáveis, embora com um aumento de interconexões e um mux precise ter 3 entradas em vez de 2.

Outra forma de contornar a complexidade gerada por uma eventual profundidade seqüencial alta é aproveitar as propriedades de transparência de UFs, isto é, a de permitir que palavras a uma entrada da UF sejam observadas sem mudanças em sua saída. Como exemplo, pode-se citar o multiplicador, cuja condição de transparência é dada pelo valor 1 em uma de suas entradas. A propriedade de transparência é bastante utilizada em técnicas de *scan* parcial e pode ser verificada também em circuitos que

adotam o conceito de *orthogonal scan path*, apresentado por [Nor e MacClu 96] em 1996. Trata-se de uma linha de *scan* ortogonal à tradicional, voltada a circuitos com *data path*, conforme mostra a Fig. 2.6. A *orthogonal scan path* pode ser vista como várias linhas de *scan* paralelas (quase sempre em número igual ao número de *bits* dos registradores) e em grande parte coincidentes com o *data path*. Os registradores passam então a ser carregados em paralelo e não mais em série, como na linha de *scan* tradicional.

Normalmente as linhas de *scan* são implementadas com a HLS praticamente pronta, porém os autores obtiveram resultados positivos mostrando que o conhecimento da função do circuito poderia ser explorado vantajosamente durante a HLS para inserir linhas de *scan* com Sobreárea menor (em alguns casos, a apenas um terço da Sobreárea resultante da linha de *scan* tradicional).

Fig. 2.6 (a) Scan Tradicional

(b) – Scan Ortogonal

Como a linha de *scan* ortogonal coincide com o *data path*, os dados –ou Vetores de Teste– não precisam mais ser injetados ou retirados *bit a bit* (como no “*test per scan*”) e sim *word a word* (como no “*test per clock*”), que geralmente traz como consequência uma diminuição do Tempo de Teste e a possibilidade de detecção de falhas por atraso. Uma das desvantagens desse método está na necessidade de alteração de certas UFs para torná-las transparentes à passagem de dados (por ex., a criação de

um *identity value* -zero aritmético nesse caso- na entrada esquerda do somador da Fig. 2.7).

Fig. 2.7 (a) Circuito com Data Path

(b) Grafo de Conectividade

A Fig. 2.7 mostra um circuito com *data path* e o correspondente Grafo de Conectividade ressaltando a linha de *scan* ortogonal, em que todos os registradores podem ser inicializados através de parte da *data path*. As linhas contínuas no Grafo de Conectividade mostram os dois caminhos para o fluxo de dados no *scan* ortogonal do circuito com *data path*, por onde seriam transferidos os resultados do teste, não considerado aqui. Num dos caminhos, os dados são injetados pela entrada primária **C**, passam pelo registrador 3, pelo somador, pelo registrador 1 e são coletados na saída primária **Y** ($C \Rightarrow 3 \overset{+}{\Rightarrow} 1 \Rightarrow Y$); no outro, o ponto de partida é a entrada primária **B**, passando pelo multiplicador, pelo registrador 2 até chegar à saída **Z** ($B \overset{x}{\Rightarrow} 2 \Rightarrow Z$). Com o auxílio do Grafo de Conectividade, [Nor e McClu 96] definem a *linha de scan ortogonal* como sendo um caminho no Grafo de Conectividade que se inicia em um nó de entrada primária, inclui um subconjunto de nós representando registradores e UFs, e termina em um nó de saída primária.

Finalmente o PHITS procura eliminar os *loops* seqüenciais, i.e., situações em que há um caminho que, partindo de um registrador, passa por uma ou mais UF(s),

passa por um ou mais registrador(es) e volta ao registrador inicial⁶. Como os *loops* sequenciais num circuito são os principais responsáveis pela dificuldade na geração do teste sequencial, eles devem ser evitados.

Apesar de os conceitos apresentados nesta seção terem sido idealizados para a HLS para teste de circuitos com metodologias convencionais para ATPG, muitos deles podem ser aplicados em soluções mais estruturadas, como o *Built-In Self Test*. A seguir, uma pequena exposição sobre a técnica BIST é apresentada.

2.3 Built-In Self-Test

Com o aumento da densidade dos circuitos VLSI, a técnica da linha de *scan*, empregada para tornar circuitos sequenciais em combinatórios durante os testes, tornou-se ineficiente e lenta, especialmente nos casos de *single-cycle testing* [Tsu 87], em que o processo de injeção dos *bits* de entrada (*scan-in*) e retirada dos *bits* de resposta (*scan-out*) implica uma morosidade proibitiva. Além disso, “...assume-se que os vetores de teste serão aplicados ao circuito por um equipamento de teste capaz de armazenar os padrões de teste e as correspondentes respostas corretas. Tais equipamentos são caros.” [McClu 86]. De certa forma, a técnica BIST contorna esses problemas através da incorporação de estruturas de teste ao próprio circuito. Estas podem ser na forma de vetores armazenados ou geradores de padrões de teste sequenciais (contadores) ou pseudo-randômicos (são atualmente os mais utilizados).

Na técnica de geração pseudo-randômica, normalmente existem registradores de teste que funcionam como geradores de padrões pseudo-randômicos (PRPG – Pseudo-Random Pattern Generator) e registradores de assinatura com múltiplas entradas (MISR – Multiple Input Signature Register), como mostra a Fig. 2.8c. Enquanto os PRPGs são

⁶ [Car e Papa 95] usam o termo *feedback indireto* em vez de *loop sequencial*.

responsáveis pela geração da sequência de padrões pseudo-randômicos, os MISRs coletam as respostas e reduzem o número de *bits* de saída, através de um processo de compactação, produzindo uma assinatura⁷ diferenciada para cada circuito. Para maior eficiência da técnica BIST, são construídos registradores para operar ora como Geradores de Padrões e ora como Analisador de Assinatura Paralelo, constituindo-se um *Built-In Logic Block Observer* (BILBO) [McClu 86]. Os circuitos sequenciais podem ser particionados, durante o teste, em um conjunto de circuitos lógicos combinacionais, sendo que um BILBO em determinado momento é o PRPG de um destes circuitos e em outro, é o MISR de outro bloco lógico.

A técnica de BIST pseudo-randômico, segundo [Avra 91], assume que todos os registradores inseridos no *data path* possam, em princípio, ser reconfigurados em 4 formas distintas: Registrador Normal (Reg), Registrador Serial de Deslocamento (Shift)(Scan), PRPG e MISR. A Fig. 2.8 ilustra 4 situações em que as mencionadas reconfigurações dos registradores são associadas a um circuito com Lógica Combinatória (LC):

Fig. 2.8 – Técnica de BIST Pseudo-Randômica: (a) operação normal; (b) inicialização; (c) BIST; (d) retirada dos resultados.

⁷ Durante o processo de compactação, a resposta de um circuito defeituoso pode produzir uma assinatura idêntica à de um circuito sem falha. A este fenômeno dá-se o nome de *aliasing* e ele é um dos responsáveis pela perda da cobertura efetiva de falhas.

No modelo BIST *off-line*, o circuito é retirado de seu modo de operação normal (*a*) e colocado em modo teste (*b*, *c* e *d*). Antes do início dos testes, os registradores precisam ser colocados em determinado estado, seja através de uma operação de *reset* ou através da “inserção de uma semente” por uma linha de *scan* serial (*b*) (ou ortogonal), quando então o BIST propriamente dito (*c*) se inicia com a aplicação dos Vetores de Teste (PRPG) ao circuito lógico combinatório, e as respostas são compactadas no registrador MISR. Para que os resultados obtidos sejam comparados com os esperados, os registradores precisam novamente ser reconfigurados como registradores de deslocamento (*d*) e a assinatura retirada através da linha de *scan*.

Os procedimentos de injeção dos Vetores de Teste (*scan-in*) e retirada das respostas (*scan-out*) tornam o processo lento, pois apenas um *bit* é injetado ou retirado por *clock* (seja para BIST ou não). O conceito de *orthogonal scan path* apresentado por [Nor e MacClu 96] pode ser empregado, com pequenas adaptações, ao BIST. Em um circuito comum, os Vetores de Teste, em vez de introduzidos pelas entradas primárias, poderiam ser gerados e compactados pelos registradores (atuando como CAs). Com isso, todos os elementos contidos na linha de *scan* ortogonal, inclusive suas interconexões [Nor 97], podem ser testados isoladamente ou em grupos.

A Testabilidade de um circuito pode ser equacionada a partir de sua topologia, com os blocos combinatórios e os registradores já fixados, independentemente de terem ou não sido observadas considerações sobre teste durante a síntese. Nesse tipo de *Testabilidade estrutural*, ainda é possível otimizar os resultados através da definição do caminho de teste, da sequência de teste, da escolha adequada dos registradores de teste, da detecção e eliminação de obstáculos que inviabilizam ou dificultam o teste etc.

Um problema muito comum detectado durante a *síntese arquitetural para Testabilidade* é o do registrador auto-adjacente [De Mi 94], i.e., quando tanto a entrada

quanto a saída de um bloco lógico combinatório se conectam ao mesmo registrador. [De Mi 94] fornece alguns exemplos, mostrados na Fig. 2.9.

Fig. 2.9 – (a) Estrutura auto-testável; (b) Registrador auto-adjacente; (c) Registrador compartilhado.

O circuito da Fig. 2.9 (a) ilustra o caso de uma UF auto-testável. Para que a UF da Fig. 2.9 (b) seja auto-testável, o registrador auto-adjacente precisaria, simultaneamente, desempenhar as funções de TPG e SA, típicas de um CBILBO (*Concurrent Built-in Logic Block Observer*). Embora o uso de um CBILBO em lugar de dois BILBOs (*Built-in Logic Block Observer*) [Kön et al. 84] possa simplificar a parte de controle de teste, a Sobreárea de um CBILBO em relação a um BILBO gira em torno de 3,5 vezes [Str e Wun 98], tornando sua aplicação pouco atrativa para alguns casos.

De qualquer forma, a reconfiguração dos registradores comuns em registradores de teste está motivando muita pesquisa, de um lado pelos bons resultados práticos alcançados e publicados nas revistas especializadas da IEEE (Cobertura de Falhas entre 90 e 100%, Sobreárea em torno de 10% e sempre diminuindo, Tempo de Teste inferior aos gastos com Equipamentos Automáticos de Teste, possibilidade de detecção de

Falhas por Atraso etc.) e por outro lado pela incessante tentativa de reduzir os pequenos impactos no desempenho provocados por essa dupla função.

2.4 Linear Feedback Shift Registers

Para a geração randômica de padrões de teste, nos casos em que as seqüências de números não precisem ser estritamente randômicas [McClu 86], ou como Analisador de Assinatura, os *Linear Feedback Shift Registers* (LFSRs) [Abra et al. 90] [Fuj 86] desenvolvem estas funções com um baixo custo de Sobreárea.

[Eps 93] apresenta um exemplo de como testar um circuito combinatório com 4 entradas e uma saída, mostrando que o uso de LFSRs é atrativo pois consistem basicamente de *Flip-Flops* (FFs) já presentes no circuito, mais algumas portas lógicas (XORs) extras. O TPG é constituído de 4 FFs, enquanto que outros 3 FFs formam o SA, como ilustra a Fig. 2.10.

Fig. 2.10 Circuito combinatório sob teste

O comportamento na saída do circuito (19) é representado pela expressão 2.1:

$$f = x_1 x_3 (\overline{x_2 x_4}) \vee \bar{x}_2 \bar{x}_4 (x_1 \vee x_3) \quad (2.1)$$

O polinômio primitivo⁸ do LFSR TPG é $1 + x + x^4$, gerando 15 estados de teste como mostra a Fig. 2.11 abaixo:

a₁	a₂	a₃	a₄	f		b₁	B₂	B₃
1	0	0	0	0	→	0	0	0
0	0	0	1	1	→	0	0	1
0	0	1	0	1	→	0	1	1
0	1	0	0	0	→	1	1	1
1	0	0	1	0	→	1	1	0
0	0	1	1	1	→	1	0	1
0	1	1	0	0	→	0	1	1
1	1	0	1	0	→	1	1	1
1	0	1	0	0	→	1	1	0
0	1	0	1	0	→	1	0	0
1	0	1	1	1	→	0	0	0
0	1	1	1	1	→	0	0	1
1	1	1	1	0	→	0	1	0
1	1	1	0	0	→	1	0	1
1	1	0	0	0	→	0	1	1

Fig. 2.11 – Resultado da Compactação de Dados para o circuito da Fig. 2.10.

O resultado dos 15 Vetores de Teste aplicados ao circuito são injetados no LFSR SA, composto por 3 FFs, descrito por $\mathbf{Db}_3 = \mathbf{b}_1 \oplus \mathbf{b}_2 \oplus \mathbf{f}$, sendo que inicialmente assume-se $b_1 = b_2 = b_3 = 0$. Os 3 FFs funcionam como Compactador de Dados, já que estes são compactados ou comprimidos nos dígitos b_1 , b_2 e b_3 . A sequência 3-tupla resultante após 15 simulações do circuito ($b_1 = 0$, $b_2 = 1$ e $b_3 = 1$) passa a ser a *assinatura* para a compactação dos dados. Um erro poderá ser detectado se a assinatura obtida for diferente do padrão esperado.

⁸ Também conhecidos como polinômios irredutíveis, são aqueles impossíveis de se fatorar em polinômios não triviais sobre o mesmo campo. Por ex., no campo finito GF(2), $x^2 + x + 1$ é irredutível, enquanto que $x^2 + 1$ não, pois $(x + 1)(x + 1) = x^2 + 2x + 1 \equiv x^2 + 1 \pmod{2}$. Veja [Nag 51] apud <http://mathworld.wolfram.com/IrreduciblePolynomial.html>

Suponha, por exemplo, que a linha 13 do circuito combinatório esteja *presa-em*

1. Consequentemente, quando o vetor de teste $a_1 = 1$, $a_2 = 1$, $a_3 = 1$ e $a_4 = 1$ for aplicado ao circuito, a resposta em **f** será 1 em vez de 0 e a assinatura dos dados compactados será a sequência 3-tupla 110, diferentemente do esperado que seria 011, como mostra a Fig. 2.12.

a₁	a₂	a₃	a₄	f com a linha 13 s-a-1		b₁	b₂	b₃
1	0	0	0	0	→	0	0	0
0	0	0	1	1	→	0	0	1
0	0	1	0	1	→	0	1	1
0	1	0	0	0	→	1	1	1
1	0	0	1	0	→	1	1	0
0	0	1	1	1	→	1	0	1
0	1	1	0	0	→	0	1	1
1	1	0	1	0	→	1	1	1
1	0	1	0	0	→	1	1	0
0	1	0	1	0	→	1	0	0
1	0	1	1	1	→	0	0	0
0	1	1	1	1	→	0	0	1
1	1	1	1	1	→	0	1	1
1	1	1	0	0	→	1	1	1
1	1	0	0	0	→	1	1	0

Fig. 2.12 – Resultado da Compactação de Dados com a linha 13 do circuito presa-em-1.

Como já foi mencionado, há casos em que um circuito contendo uma falha altera sua saída e ainda assim a assinatura resultante permanece idêntica ao padrão esperado. Para compreender melhor esse fenômeno da *aliasing*, que pode ser uma consequência da compactação dos dados, [Wil et al. 88] fizeram uma discussão detalhada. Nesse artigo, os autores modelaram um LFSR por um processo de Markov e chegaram à conclusão de que o uso de polinômios primitivos em vez de não primitivos, geralmente, minimiza as probabilidades de *aliasing*.

2.5 Cellular Automata

Os padrões gerados por um LFSR de n -saídas não são geralmente do tipo randômicos de n -dígitos devido à sua natureza de deslocamento. No exemplo apresentado, os 15 Vetores de Teste estão fortemente correlacionados por um deslocamento, mas este problema está minimizado porque todos os $2^n - 1$ vetores foram injetados no circuito. No entanto, na medida em que n cresce, a geração dos $2^n - 1$ estados pode se tornar proibitiva. Nos casos em que apenas parte dos $2^n - 1$ estados serão gerados é conveniente um comportamento mais randômico, ou pseudo-randômico, como aquele apresentado por um encadeamento de células (FFs), cujos estados evoluem de acordo com um conjunto de regras baseado no estado das células vizinhas, conhecido como Cellular Automata (CA) [Hor 89a], porque certos tipos de falhas como por atraso (delay) e de circuitos abertos (open faults) são resistentes a Vetores de Teste correlacionados. Existem CAs com cadeia de células de diferentes regras, sendo que a regra de cada célula depende do algoritmo usado para computar seu *próximo estado*, baseando-se na informação obtida das células anterior e posterior.

CAs híbridos, por ex., com combinação de regras computacionais 90 e 150, ($\mathbf{a_i(t+1)} = \mathbf{a_{i-1}(t)} \oplus \mathbf{a_{i+1}(t)}$ e $\mathbf{a_i(t+1)} = \mathbf{a_{i-1}(t)} \oplus \mathbf{a_i(t)} \oplus \mathbf{a_{i+1}(t)}$, respectivamente) [Hor et al 89b], geram Vetores de Teste com excelentes padrões de distribuição, com probabilidade de *aliasing* de $\mathbf{P_a} = \mathbf{2^{-n}}$ [Dam et al. 90], sendo n o tamanho do compactador ou da assinatura. Para o caso comum de $n=16$, esta probabilidade é, aproximadamente, $\mathbf{P_a} \cong 10^{-6}$, valor este geralmente aceito como muito bom.⁹

Além de produzir seqüências mais randômicas, os CAs apresentam a vantagem de exigir que apenas as células adjacentes se comuniquem, tornando-o

⁹ No entanto, como [Hor 89a] nos alerta, a análise da *aliasing* em um CA é um problema bem menos tratável do que a análise equivalente ao LFSR.

modulável. Opções técnicas com características equivalentes às dos CAs têm firmado o BIST como sinônimo de efetividade, velocidade e pequena Sobreárea [Ser et al. 90] e [Ser 94]. Os CAs trazem consigo alguns problemas, também: a área de cada célula do CA é maior que a correspondente do LFSR, a extensão de seu ciclo geralmente é menor do que o correspondente LFSR etc.

Um problema comum aos CAs de ciclo máximo é a necessidade de intercalação de células com diferentes regras (e, conseqüentemente, áreas). Em certos casos, uma composição com homogeneidade alta torna-se vantajosa. [Car et al. 98b] apresentaram uma metodologia de busca na qual as estruturas obtidas são *quase-uniformes*, i.e., geram seqüências com ciclo máximo a um custo de área próximo ao de uma estrutura uniforme, ou pelo menos o mais próximo possível. As várias combinações de estruturas do CA desejado são ordenadas e, por meio de algoritmos eficientes, os correspondentes polinômios característicos são determinados e submetidos a um teste indicador da condição de polinômio primitivo ou não. Como os resultados concretos apresentados indicaram um grau de uniformidade muito grande, os CAs construídos de acordo com essa técnica podem ser tratados, por abuso de linguagem, como sendo CA regra 90 ou CA regra 150.

2.6 Por que BIST associado à Síntese de Alto Nível ?

Uma grande vantagem¹⁰ da técnica BIST é a substituição da geração determinística de teste [Raj 95] por seqüências pseudo-randômicas, cuja implementação normalmente causa pequena Sobreárea na arquitetura original, já que os registradores podem ser reconfigurados em *Cellular Automata*, ou *Linear Feedback Shift Register*, para funcionarem como TPG ou SA, durante a fase de teste. Opções técnicas com

características equivalentes às dos CAs têm firmado o BIST como sinônimo de efetividade, velocidade e baixo custo [Ser et al. 90] e [Ser 94].

Quando à HLS associa-se a técnica BIST, tem-se uma poderosa plataforma capaz de gerar CIs com arquiteturas em que a incorporação de Autoteste torna-se simplesmente mais uma de suas opções, pois para uma certa descrição comportamental podem existir diferentes estruturas sintetizadas, todas desempenhando a mesma função. As técnicas levantadas por [Lee 97], e já comentadas, são de aplicação bem ampla na HLS e produzem resultados igualmente bons para o caso específico de HLS agregada à técnica BIST com TPGs e SAs.

Fig. 2.13 – Fases da HLS no ambiente MACH.
[Flot et al. 95]

A Testabilidade pode ser abordada tanto durante ou após a HLS (normalmente na Alocação, Fig. 2.13). Em qualquer dos casos, o nível de abstração do projeto é perfeitamente adequado à metodologia BIST, e segue a tendência de mudança do nível de localização de falhas do transistor para o sistema [Ohl 96].

Porém, decisões de longo alcance tomadas durante a HLS voltadas à Testabilidade, na fase em que a estrutura do circuito ainda está sendo determinada, causam generalizações com efeitos nem sempre previsíveis, pois a modelagem da Testabilidade em HLS baseia-se numa abstração das informações do nível de portas, uma vez que o modelo de falha aceito está justamente neste nível de portas lógicas¹¹.

¹⁰ Comumente caracteriza-se a dispensa dos Equipamentos Automáticos de Teste como a maior das vantagens do BIST.

¹¹ Como bem mencionado em [Abra et al. 90], o modelo de falha funcional baseado em variável falha é limitado pela necessidade de se definir explicitamente o comportamento da falha, e sua aplicação está restrita a um pequeno número de falhas específicas.

Por exemplo, nos circuitos com *fanout* reconvergente, podem existir falhas funcionalmente equivalentes que não sejam estruturalmente equivalentes, introduzindo uma dificuldade computacional extra, pois a determinação de relações de equivalência funcional é um problema NP-completo.

Ainda que, em geral, a consideração com a incorporação do teste depois da HLS resulte em Sobreárea maior, a Síntese Estrutural para Teste pós HLS pode também produzir bons resultados, se considerações genéricas com o teste (Observabilidade, Controlabilidade, Profundidade Seqüencial, Self-loop etc.) tiverem sido respeitadas na HLS. Para uma mesma arquitetura sintetizada podem existir diferentes soluções de teste, todas obtendo a mesma Cobertura de Falhas a um custo diferenciado.

2.7 Implementações de Síntese para Testabilidade

Uma vez feita essas considerações mais genéricas, serão apresentadas agora algumas soluções mais específicas empregando a técnica BIST (*off-line*) associada à HLS, nas duas formas mais comuns, i.e., incorporação de estruturas de teste durante a HLS e incorporação com a arquitetura já sintetizada. Para um circuito genérico, a incorporação do teste durante a HLS parece ser um bom passo inicial, pois eventualmente os resultados podem ser mais que satisfatórios. Se, no entanto, as expectativas não forem satisfeitas, a alternativa de incorporar o teste com a arquitetura inicial sintetizada torna-se convidativa, pois nesse nível a intervenção sobre os pontos problemáticos tem um alcance mais específico. Sempre que houver a necessidade de uma solução iterativa, o ciclo de tarefas nesse caso é menor.

2.7.1 Projeto Arquitetural Simultâneo à Síntese Estrutural para Testabilidade

[Geb e Elm 92] propõem uma abordagem para a síntese de arquiteturas testáveis chamada CATREE2 (*Computer Aided TREE*), com uma metodologia de teste específica incorporada à HLS, de modo que qualquer decisão inicial é tomada considerando sempre o modo de funcionamento normal e teste do circuito. Nesse sistema, as bibliotecas contêm os tipos de UFs à disposição (definidos como uma lista de operações que elas podem realizar), o tempo de propagação (da entrada de dados até a saída de dados), o atraso entre sucessivas entradas de dados e a área da UF. Assumindo que cada UF tenha sido pré-caracterizada para teste, a sua respectiva biblioteca contém informações sobre Testabilidade (BIST), como o polinômio e o estado inicial do TPG associado à UF, a extensão da sequência pseudo-aleatória e o total de falhas detectadas. Por exemplo, um conjunto para teste com 5000 vetores pode atingir Cobertura de Falhas de 90%, enquanto que outro com 1700 vetores atinge 80% etc.

O diagrama de blocos desse Projeto Arquitetural Simultâneo com alocação para a Síntese Estrutural para Testabilidade e algoritmos de *binding* está ilustrado pela Fig.2.14 abaixo:

Fig.2.14 – CATREE2: Projeto Integrado com uma Metodologia para Teste

Embora um CI funcione no modo normal ou teste de forma independente, [Ora e Har 97] argumentam que, para otimizar o desempenho do CI nos dois modos, deve

haver um fluxo de informação entre um Plano de Teste preliminar e o processo de HLS. Assim, a partir de um CDFG - *Control Data Flow Graph*, faz-se uma avaliação do impacto do *scheduling* e *binding* sobre as opções de teste, procurando sempre criar alternativas que resultem num tempo total de teste menor. Na fase específica de Síntese Estrutural para Testabilidade, alguns registradores podem ser incorporados aos Caminhos de Teste como registradores intermediários, de forma a implementar-se um *pipelining* durante o teste de qualquer UF, aumentando o paralelismo potencial entre elas. Esses autores dedicam uma atenção especial para a maximização do paralelismo potencial das UFs durante o *scheduling* e *binding*.

Um dos problemas da otimização concomitante do desempenho dos modos teste e normal é que a otimização do modo teste pode acontecer à custa de uma Sobreárea desnecessária no CI, uma vez que a minimização dos conflitos de teste direciona a HLS, tentando evitar que vias de dados distintas compartilhem a mesma unidade de *hardware* (por ex., dois componentes cujos resultados são observados por um único registrador de teste não podem ser observados simultaneamente). Há na abordagem de [Ora e Har 97] uma hipótese implicitamente aceita de que o tempo necessário para testar diferentes UFs não seja considerado nessa fase. Na prática, no entanto, pode ocorrer de o número de ciclos necessários para testar, sequencialmente, duas UFs ser menor que o necessário para testar uma terceira UF. Nestes casos, não se justificaria o teste simultâneo das três unidades.

A HLS associada com a Síntese Estrutural para Testabilidade pode evitar circuitos com *self-loops*, i.e., circuitos em que um único registrador deve gerar Vetores de Teste e compactar as respostas. [Avra 91] fez um estudo comparativo, cujos resultados iniciais indicam que a incorporação da Testabilidade durante o processo de

HLS de circuitos com *Data Paths* autotestáveis resulta em um custo global de projeto menor.

2.7.2 Síntese Estrutural para Testabilidade Incorporada após o Projeto Arquitetural

Em [Car e Papa 95], a métrica para Testabilidade é computada por meio de um Modelo da Cadeia de Markov, que, partindo da descrição RTL do circuito (i.e., com a respectiva arquitetura já sintetizada), atribui valores para a distribuição de probabilidade de estado de cada registrador do circuito. Este Modelo de Markov particiona o circuito em pequenas partes, com cada uma contendo a informação necessária para analisar um único registrador, para então modelar cada registrador separadamente na medida em que o registrador muda de um estado para outro.

Uma vantagem desse particionamento fino é a obtenção de dados suficientes para executar uma refinada análise do circuito RTL, feita com duas métricas que medem a efetividade com que um registrador isolado no circuito gera um vetor de teste, sua aleatoriedade (baseada em entropia) e a esperada cobertura de estados possíveis do registrador. Com isso, em alguns casos, pode-se dispensar o uso dos TPG nas duas entradas de uma UF, ao se constatar que um registrador comum associado à técnica de BIST circular pode gerar Vetores de Teste suficientemente randômicos.

A Fig.2.15 ilustra o diagrama geral de blocos desta abordagem, em que as estruturas de teste são definidas com a arquitetura correspondente à descrição funcional do circuito já pronta.

Fig.2.15 – Testabilidade incorporada após a Síntese do Projeto

Numa linha de argumentação um pouco distinta, em [Ama et al. 97a] parte-se de uma arquitetura otimizada por um Sistema de HLS, no caso, o MACH da Université de Montpellier, França, e minimiza-se uma Função Custo de Teste, que avalia Testabilidade, Área e Desempenho. Introduzindo-se, durante a elaboração do Plano de Teste das UFs, informações detalhadas, previamente coletadas¹², como a Cobertura de Falhas e a duração, em números de ciclos, da sessão de teste para cada tipo de UF, reduz-se o tempo de computação gasto para a solução de problemas relacionados aos teste do CI.

Os autores optaram por uma metodologia de teste que vem ganhando importância nos últimos anos, como mostram alguns estudos acerca do impacto positivo que informações de teste pré-compiladas causam no processo de geração de testes. [Mur e Hay 90] implementaram um sistema empregando *Test Packages*, cujos resultados sugerem uma melhora na velocidade de geração de testes de três ordens de grandeza sobre as técnicas convencionais para circuitos compostos por *data paths*. Pela metodologia de [Ama et al. 97a], a Cobertura de Falha mínima das UFs já é conhecida

¹² Existem evidências bem embasadas sugerindo que o teste de cada UF precisa ser minuciosamente planejado, se se deseja otimizar o resultado. [Pom e Red 98] demonstraram que dependendo do estado inicial do TPG, a Cobertura de Falha e o Tempo de Teste podem variar consideravelmente, especialmente quando há falhas difíceis de ser detectadas por um número prático de vetores de teste pseudo-aleatórios.

antes da simulação de falhas do circuito, restando então escolher os registradores e as interconexões necessárias ao teste, cujo custo computacional é bem menor.

Os procedimentos descritos nesse artigo se referem, essencialmente, à Síntese Estrutural, mas a intenção dos autores é incorporar esses resultados preliminares no sistema de HLS, para que os circuitos sintetizados já tragam uma metodologia de teste (BIST) incorporada.

A Fig.2.16 ilustra, genericamente, os diversos procedimentos executados na Síntese Comportamental (do Projeto Arquitetural) e Estrutural.

Fig.2.16 – Ambiente de Síntese de Alto Nível BIST

A Análise BIST verifica se o circuito está estruturado numa forma adequada para incorporar o Teste BIST, i.e., se a cada UF existem pelo menos dois registradores de entrada e um de saída disponíveis, se não há *self-loop* etc. A Geração de Teste das UFs determina quais registradores e quais segmentos do *data path* serão usados como Caminho de Teste para determinada UF. Os segmentos do *data path* e registradores que não estiverem incluídos em nenhum Caminho de Teste de UF serão testados pelos procedimentos Geração de Teste de Conexões e de Registradores, respectivamente.

Recentemente, [Stro e Wund 98] propuseram um forma de inserção de registradores de teste para um circuito considerado no nível de portas lógicas. O circuito é representado por um grafo direcional G (Fig. 2.17), com cada ciclo representado dentro de um caminho denominado Componentes Fortemente Conectados (SCC – Strongly Connected Component) de G .

Fig. 2.17 – (a) Teste com BILBO

(b) Teste Otimizado com CBILBO

Partindo da hipótese de que uma célula CBILBO ocupa uma área 3,5 vezes maior do que a de uma célula BILBO, eles chegaram a resultados mostrando que as

estruturas BIST baseadas na utilização de CBILBO têm um custo de Sobreárea menor que aquelas que usam BILBO.

Para obter esses resultados surpreendentes, eles desenvolveram um algoritmo que explora o princípio de que para uma solução (geral) de custo mínimo de inserção em **G** basta minimizar o custo de cada SCC de **G**. Embora o artigo forneça evidências de que a aplicação desse algoritmo produza soluções melhores com CBILBO do que com BILBO, não parece ter sido feita uma comparação de custos com outros algoritmos tradicionais que utilizem apenas BILBO.

A literatura especializada relatando os avanços na área de teste no estado da arte reportam que, independentemente de a Síntese Estrutural para Testabilidade ter sido incorporada após ou durante o Projeto Arquitetural de circuitos complexos (VLSI), a Testabilidade precisa ser considerada desde os primeiros estágios do projeto do CI.

2.8 Resumo

Neste capítulo, foram abordados três objetivos básicos da HLS: otimização da Área, dos Atrasos em caminhos críticos e da Testabilidade. Foi ressaltada a importância da Controlabilidade e da Observabilidade do circuito como parâmetros genéricos para a Testabilidade, independentemente da forma como a Estrutura de Teste é incorporada à Arquitetura gerada pela HLS.

Alguns trabalhos tratando da Síntese para Testabilidade foram apresentados e classificados em dois grupos:

1. Projeto Arquitetural Simultâneo à Síntese Estrutural para Testabilidade
2. Síntese Estrutural para Testabilidade Incorporada após o Projeto Arquitetural

Em todos os trabalhos do estado da arte verificados, a Testabilidade foi levada em consideração desde os primeiros estágios do projeto do CI.

3 ANÁLISE, PROJETO E IMPLEMENTAÇÃO DO SISTEMA

Este capítulo trata da Análise, do Projeto e da Implementação do Sistema BIST_S, baseado no Algoritmo Série para Otimização de Planos de Teste de UFs proposto em [Ama et al. 97a] e [Ama et al. 97b], e da Análise, do Projeto e da Implementação do Sistema BIST_P, baseado no Algoritmo Paralelo para Otimização de Planos de Teste de UFs proposto em [Ama et al. 98].

Os Algoritmos Série e Paralelo podem ser vistos como uma resposta dos autores ao seguinte Problema do Autoteste:

“Dada uma arquitetura sintetizada ou não por um sistema de HLS, determine quais as modificações estruturais necessárias para tornar o circuito capaz de testar suas UFs, de acordo com um esquema BIST off-line, distribuído, nas seguintes situações:

- a. Menor custo de Sobreárea de Hardware possível, ou*
- b. Menor Tempo de Teste possível.”*

Durante o Projeto e a Implementação do BIST_P, verificou-se a conveniência de refinar o Algoritmo Paralelo para que o usuário pudesse determinar um compromisso, através da escolha do paralelismo máximo, entre Custo de Sobreárea e Tempo de Teste.

3.1 Os Algoritmos e o Ambiente de Teste

Os algoritmos de otimização tratados aqui seguem a estratégia de BIST *off-line* apresentados nas seções 1.2 e 1.4 para *data paths* de arquiteturas gerados por Sistemas de Síntese de Alto Nível. O objetivo proposto pelos idealizadores deste esquema de teste é o de geração de uma Estrutura de Teste e um Plano de Testes otimizados para todos os elementos do *data path*, ou seja, as UFs, os registradores, os Muxes e as

interconexões [Ama et al. 97a]. Os Algoritmos Série e Paralelo tratam da geração e aplicação de testes de UFs.

Essencialmente eles se diferenciam pelo fato de que no Algoritmo Série todas as UFs são testadas sequencialmente, fazendo com que um TPG e/ou um SA possa(m) ser compartilhado(s) por várias UFs, reduzindo-se a Sobreárea (de *Hardware*), mas aumentando o Tempo de Teste. O Algoritmo Paralelo busca o máximo paralelismo permitido pela arquitetura definida (ou não) pelo Sistema de HLS, diminuindo-se assim a probabilidade de um registrador TPG ou SA ser associado a várias UFs, porém tornando o teste mais rápido.

Em qualquer das opções, o algoritmo deve realizar uma Síntese Estrutural para Testabilidade e gerar um Plano de Teste baseado na opção *test per clock*, que tem vantagens com respeito a Tempo de Teste, detecção de falha por atraso, Cobertura de Falhas etc., mas que frequentemente requer uma Sobreárea maior do que a opção *test per scan* [Str e Wun 98].

A separação entre a Síntese do Projeto Arquitetural e do Estrutural tem como uma desvantagens a duplicação da Base de Dados [Geb e Elm 92], e a conseqüente falta de uniformidade de parâmetros estimativos que se vê durante uma análise de Testabilidade. Ademais, uma arquitetura otimizada sem considerações com a Testabilidade costuma conter *self-loops*, que diminuem o número de registradores, mas dificultam o Autoteste. Como os custos de manufatura estão em queda, diferentemente dos Custos de Teste, então não é recomendável economizar no número de registradores a ponto de penalizar a Testabilidade. O busca do ponto ideal de compromisso entre esse dois fatores tem servido de motivação para constantes aperfeiçoamentos na técnica BIST.

Uma grande vantagem desta alternativa reside na facilidade com que se implementam soluções iterativas para os problemas de teste, sem necessidade de refazer toda a HLS. Suponha que para um determinado ASIC, é encontrada, para uma cobertura mínima de falha admissível, uma solução com um custo de Sobreárea abaixo daquele valor tolerável pelo projeto; então, uma nova solução com Cobertura de Falha maior poderá ser buscada, repetindo-se apenas a Síntese Estrutural para Testabilidade.

3.2 Fluxograma dos Algoritmos para Síntese Estrutural BIST

Na estratégia proposta por [Ama et al. 97a], [Ama et al. 97b] e [Ama et al. 98], os TPGs e SAs são os próprios registradores, reconfigurados na forma de *Cellular Automata*, especificamente como QULCA [Car et al. 98a], gerando vetores pseudo-aleatórios com extensão de ciclo máximo, com visto na seção 2.5. Como existem diversas regras para montar CAs e QULCAs de mesmo tamanho de palavra, fica proporcionado um universo de busca para soluções melhores em termos de Cobertura de Falhas, duração da aplicação de testes, ocupação de área, etc.

Os algoritmos Série e Paralelo objetivam gerar um Plano de Teste e uma *Netlist* especificando quais registradores se transformarão em TPG e/ou SA e a seqüência em que as UFs devem ser testadas. Isto depois de uma seleção feita heurísticamente¹³, de acordo com uma Métrica Série-Paralela. Como mostrado na Fig. 3.1, os Algoritmos Série e Paralelo possuem as mesmas etapas computacionais básicas e se diferenciam pela Métrica.

¹³ “Um algoritmo que produza rapidamente soluções boas, mas não necessariamente ótimas, é dito heurístico”. Veja [Aho et al. 83].

Fig. 3.1 – Fluxograma da Geração de Plano de Teste

Os algoritmos aqui descritos fornecem uma solução para o problema da Inserção de Registradores de Teste, baseados em informações pré-computadas, específicas a cada tipo de UF, que permitem a criação de regras preferenciais para teste. Assim, se a Cobertura de Falhas exigida for, por exemplo, 0.9, a este valor estará associado um número mínimo de Vetores de Teste, uma regra específica para os CAs (ou QULCAS) responsáveis pela geração dos Vetores de Teste, a Sobreárea associada a eles etc.

O desenvolvimento dessas bibliotecas técnicas foge ao escopo desta dissertação; elas estão sendo implementadas num outro projeto do Laboratório de Microeletrônica da EPUSP. Como os Algoritmos Série e Paralelo devem encontrar uma solução otimizada heurísticamente, dada uma base de dados qualquer, e como as bibliotecas definitivas ainda não estão prontas, as bibliotecas utilizadas nas aplicações aqui

descritas são projeções e estimativas iniciais, que simulam aproximadamente uma biblioteca real.

O primeiro conjunto de dados que está sendo adotado para este trabalho serve para avaliar o Custo Unitário Relativo de *Hardware* associado à transformação de um registrador em CA, a partir de uma Biblioteca de *Hardware*, com todos os valores dos custos de TPG, SA e BILBO, como mostra a Tab. 3.1 [Ama et al. 97a].

Tab. 3.1 – Exemplo de Arquivo B_TPG_SA com o Custo Relativo de *Hardware*

Tipo	Custo Relativo
normal	1.00
TPG90	1.40
TPG150	1.80
TPG90/150	2.04
SA90	1.80
SA150	2.19
SA/TPG90	2.04
SA/TPG150	2.44
SA/TPG90/150	2.87

O Algoritmo busca um segundo conjunto de dados, a biblioteca de dados pré-compilada, com os parâmetros de teste relativos a cada tipo de UF, como o mostrado na Tab. 3.2.

Tab. 3.2 – Exemplo de Arquivo B_UF com Cobertura de Falha e Tempo Gasto

Tipo	TPG	N.º de Vet. de Teste	Cob. de Falhas	C_{time}
fu 1	TPG90	3000	0.3	0.0320
fu 1	TPG90	5000	0.8	0.0153
fu 1	TPG90	10000	0.95	0.0076
fu 1	TPG150	20000	0.96	0.0122
fu 2	TPG90	8000	0.95	0.0061
fu 2	TPG150	6000	0.95	0.0046
fu 3	TPG90	10000	0.90	0.0153
fu 3	TPG150	12000	0.95	0.0092

C_{time} indica o Tempo de Teste, proporcional ao número de vetores necessários para atingir uma certa Cobertura de Falhas. No cálculo de C_{time} foi usada uma Função Custo fortemente influenciada pela Taxa de Cobertura de Falha, como mostra a expressão 3.1.

$$C_{time} = \frac{v_i}{2^n} \cdot (1 - FC_i) \quad (3.1)$$

Na expressão (3.1), v_i representa o número de Vetores de Teste, n representa o número total de *bits* de entrada (nos exemplos apresentados, $n=16$) e FC_i , a Cobertura de Falha. Alternativamente, poderia ter sido usada uma função custo mais sensível ao número de vetores, como mostra a expressão (3.2).

$$C_{time} = \frac{v_i}{2^n} \cdot \frac{1}{FC_i} \quad (3.2)$$

3.3 Fluxograma da Métrica Série

Como mostra a Fig. 3.2, a Métrica Série proposta em [Ama et al. 97b] faz, inicialmente, uma leitura de dados de entrada e, em seguida, o levantamento de todos os registradores associados a cada UF, que podem ser transformados em TPG ou SA. O fluxograma segue através de dois grandes blocos: a Otimização de Registradores e a Atribuição de Regras.

Na Otimização de Registradores, tabelas de relacionamento de entrada e saída são construídas para se descobrir quais registradores oferecem mais alternativas de conexões com as UFs, de forma que o impacto das transformações dos registradores seja o menor possível. A alocação dos registradores a serem transformados é feita iterativamente, um a um, definidos através de um Fator de Ganho.

A etapa final de Atribuição de Regras consiste na avaliação de mudanças nas regras dos CAs e de como isto afeta o *hardware*. A análise é baseada no cálculo do

Custo de *Hardware* de cada uma das soluções encontradas com as mudanças de regras a partir da Tabela 3.1 e 3.2.

Fig. 3.2 – Fluxograma da Métrica Série

A resolução algorítmica ao Problema do Autoteste (Série), com um grupo de instruções ordenadas para encontrar o conjunto de Registradores de Teste, a um Custo Computacional Otimizado, é mostrada na Fig. 3.3.

```

1-{
2-    If (Arquitetura Gerada pelo Mach){
3-        Lê Base de Dados do Mach;
4-        Cria Netlist;
5-
6-    }
7-    else{
8-        Lê Netlist;
9-

```

```

10-    }
11-
12-
13-    Cria a Lista de Registradores Candidatos;
14-
15-
16-
17-
18-
19-    // Verifica Se o Circuito Original Precisa de Alterações;
20-    Faz Análise de Testabilidade;
21-    While (UF Sem Registrador)
22-        Acrescenta Mux e Registrador ou Conexões aos Existentes;
23-
24-    Atualiza a Lista de Registradores Candidatos;
25-
26-    While (UF Sem TPG ou SA){
27-        Constrói as Tabelas de Entrada e Saída;
28-        Calcula o Ganho de Todos Registradores;
29-        Escolhe o Registrador de Teste R com Maior Ganho;
30-        Coloque R em Todas as Saídas e Entradas Possíveis;
31-    }
32-
33-    Lê os Dados sobre a Cobertura de Falha Desejada;
34-    Lê os Dados sobre o Percentual de Custo Adicional de Tempo Aceito;
35-
36-    While (UF Sem Regras de Teste){
37-        Atribui as Regras Preferenciais aos Registradores de Teste;
38-        Calcula o Custo Preferencial;
39-        do{
40-            do{
41-                Segue = 0;
42-                Atribui Regra Não-Preferencial a um dos Registradores;
43-                Calcula o Custo Não-Preferencial;
44-                If (Custo Não-Preferencial < Custo Preferencial){
45-                    Solução Preferencial = Solução Não Preferencial;
46-                    Segue = 1;
47-                }
48-            } while (Registrador Sem Mudança de Regra e Segue == 0);
49-        } while (Nova Solução Preferencial);
50-    }
51-    Imprime Plano de Teste da Última Solução Preferencial;
52-}

```

Fig. 3.3 – Algoritmo Série

A linha 2 do Algoritmo Série corresponde à pergunta que o usuário deve responder, indicando a origem dos dados de entrada. O formato de entrada de dados pode ser o arquivo de saída do HLS Sintetizador Mach ou uma *Netlist* com formato compatível.

Fig. 3.4 – Circuito Data3UF com Data Path

Tendo a arquitetura com *data path* sido analisada, uma lista de todos os Registradores Candidatos (linha 13) é gerada, isto é, uma lista com todos os registradores associados a cada UF. Exemplo desta lista está na Figura 3.5 para o *data path* da Fig. 3.4. Pode-se observar que para cada UF, todos os registradores candidatos de entradas e saída são discriminados e que quando houver apenas um candidato ele é representado por um retângulo na figura.

$$\begin{aligned}
 \text{UF}_1: \{R_1, R_2\} & \quad \boxed{R_3} \quad || \quad \{R_2, R_3, R_4, R_6, R_7\} \\
 \text{UF}_2: \{R_3, R_4, R_5\} & \quad \boxed{R_6} \quad || \quad \{R_2, R_7\} \\
 \text{UF}_3: \{R_6, R_7\} & \quad \{R_3, R_8\} \quad || \quad \{R_2, R_3, R_4, R_6, R_7, R_9\}
 \end{aligned}$$

Fig. 3.5 – Lista com a alocação de Registradores para o Circuito Data3UF

Esta lista é utilizada na Análise de Testabilidade (linha 20) do circuito para descobrir, por ex., se todas as UFs têm registradores candidatos para funcionar como TPGs e SAs. Nos casos de *self-loop*, como o ilustrado pela Fig. 3.6 (a), o algoritmo oferece duas alternativas: acrescentar um novo registrador ou uma ligação a um já

existente (Fig. 3.6 (b)). Em qualquer uma das alternativas, é preciso acrescentar um multiplexador de teste.

Fig. 3.6 - (a) FU não testável. (b) FU testável.

Depois da Análise de Testabilidade e eventuais mudanças (linha 21), a Lista de Registradores Candidatos é atualizada (linha 24). Uma depuração é feita nas listas de candidatos: sempre que uma UF apresentar apenas um registrador como possibilidade de transformá-lo em SA, esse registrador é retirado da lista de candidatos a TPG, e considerado como candidato único, e sempre que houver um único registrador candidato numa das entradas da UF, ele é retirado da lista de SA e classificado como candidato único a TPG, evitando-se assim o *self-loop*.

Por exemplo, a lista de alocação da Fig. 3.4 passa a ser o da Figura 3.7, uma vez que o registrador R_3 , que aparece inicialmente como candidato a registrador de saída para a UF_1 , deve ser retirado, pois ele é o único candidato para a entrada 2 da UF_1 .

$$\begin{aligned}
 UF_1: \{R_1, R_2\} \quad \boxed{R_3} \quad || \quad \{R_2, R_4, R_6, R_7\} \\
 UF_2: \{R_3, R_4, R_5\} \quad \boxed{R_6} \quad || \quad \{R_2, R_7\} \\
 UF_3: \{R_6, R_7\} \quad \{R_3, R_8\} \quad || \quad \{R_2, R_3, R_4, R_6, R_7, R_9\}
 \end{aligned}$$

Fig. 3.7 – Alocação de Registradores para o Circuito Data3UF

As Tabelas de Entrada e Saída (linha 27) são geradas com o objetivo de retratar o compartilhamento de registradores entre UFs. Caso um registrador compartilhado seja transformado em TPG ou SA haverá uma redução de custo em área ao se comparar com o caso de dois registradores separados serem transformados, um para cada UF. Por exemplo, para o circuito Data3UF, da Fig. 3.4, as tabelas iniciais de entrada e saída são as mostradas na Fig. 3.8:

IN	UF ₁	UF ₂	UF ₃	OUT	UF ₁	UF ₂	UF ₃	I/O	UF ₁	UF ₂	UF ₃
UF ₁		R ₃	R ₃	UF ₁		R ₂ R ₇	R ₂ R ₄	UF ₁		R ₂	R ₂ R ₃
UF ₂			R ₃ R ₆				R ₆ R ₇	UF ₂	R ₄ R ₆	-	R ₃
UF ₃				UF ₂			R ₂ R ₇				R ₄ R ₆
				UF ₃				UF ₃	R ₆ R ₇	R ₇	

$$\text{IN}(3)=3; \text{IN}(6)=1;$$

OUT(2, 7)=3;

```
OUT(4, 6)=1;
```

$$\text{I/O}(2, 3, 4, 7)=2;$$

I/O(6)=3;

$$\text{SL}(2, 3, 6, 7)=1;$$

Fig. 3.8 – Tabelas de Entrada e Saída para o circuito Data3UF

Com as tabelas de relacionamento entre entradas e saídas, o Ganho dos Registradores (linha 28) pode ser facilmente calculado como a combinação do número de ocorrência individual de cada registrador, dois a dois. O fator Ganho dos Registradores, $\mathbf{GR}_i$, ('i' indicando o número do registrador) expressa essas características em termos numéricos, de acordo com a fórmula (3.3), abaixo.

$$\mathbf{GR}(\mathbf{i}) = \alpha.\mathbf{IN}(\mathbf{i}) + \beta.\mathbf{OUT}(\mathbf{i}) + \delta.\mathbf{IO}(\mathbf{i}) - \gamma\mathbf{SL}(\mathbf{i}), \quad (3.3)$$

sendo IN, OUT, IO e SL, respectivamente, representações do número de registradores compartilhados na entrada ou saída das UFs, entrada/saída de diferentes UFs e *self-loops* de todas as UFs do circuito. O peso negativo associado a γ expressa o fato de que registradores envolvidos em *self-loops* não são desejáveis. Nos exemplos subseqüentes, $\alpha=2$, $\beta=3$, $\delta=1$ e $\gamma=1$ são adotados, indicando pesos associados às relações. Os pesos são

determinados pelo grau de redução ou acréscimo de *hardware* devido à existência de compartilhamento. Por exemplo, transformar registradores comuns em SAs é mais caro que em TPGs, portanto o seu compartilhamento à saída (β) proporciona um melhor ganho que à entrada(α). Os registradores com ganhos maiores são os prováveis candidatos para a transformação TPG ou SA.

A Tabela de Entradas e Saída é dinâmica, com o seu conteúdo modificado sempre que certas decisões de escolha de um registrador causam uma exclusão nas opções de atribuição dos outros registradores ainda candidatos.

Exemplificando, a alocação inicial da Fig. 3.7 para o circuito Data3UF é modificada calculando-se o registrador com o maior ganho. Pelos resultados baseados na Fig. 3.8, chega-se à conclusão que o registrador de maior ganho é o R_2 . A sua escolha gera uma outra lista de alocação mostrada na Figura 3.9.

Fig. 3.9 – Alocação de Registradores para o Circuito Data3UF

Com o ganho associado a cada registrador, a escolha final dos registradores candidatos (linhas 29 e 30) é feita com base nessa ordem hierárquica, geralmente ao longo de vários passos, já que somente o registrador com maior ganho é analisado a cada passo (claro que a escolha, por ex., de um registrador para TPG significa retirá-lo da lista de candidatos a SA (para uma mesma UF), e isso pode implicar a indicação simultânea de outro registrador para SA, se restar apenas ele).

O circuito Data3FU exigiu três passos para a escolha final dos registradores, cujo resultado é mostrado na Fig. 3.10.

UF ₁ :	R₁	R₃		R₂
UF ₂ :	R₃	R₆		R₂
UF ₃ :	R₆	R₃		R₂

Fig.3.10 - Listas Finais de Alocação de Registradores do circuito Data3FU.

A partir destas Listas Finais de Entrada e Saída, inicia-se um processo de “Otimização dos Registradores” para atribuição das regras de teste aos CAs, visando encontrar a alternativa com menor Custo de *Hardware* (linha 36 e seguintes). Para isso, usa-se uma abordagem *greedy*, inicialmente atribuindo-se regras preferenciais (i.e., de menor custo) a cada um dos TPGs e SAs, com valores de Custo de *Hardware* do Plano de Teste obtidos de bibliotecas de UFs, e introduzidos na Função Custo (3.4),

$$\text{Custo_Hardware} = \sum_i \text{Custo_UF}_i (\text{TPG}+\text{SA}) \quad (3.4)$$

O Custo de *Hardware* procura refletir o conjunto de modificações introduzidas no circuito para torná-lo autotestável, e é a soma dos valores parciais relativos a cada UF, cujo teste envolve a transformação de até dois registradores em TPG e um em SA. Por ex., para a UF₃ da Fig.3.4, os TPGs poderiam ser R₆ e R₈, além de R₉ para SA. Mas, supondo-se que R₆ também foi o registrador escolhido para TPG da UF₂, então, efetivamente apenas o custo de transformação de R₈ e R₉ devem ser computados (a menos que UF₂ e UF₃ sejam testadas com diferentes regras, ou seja, R₆ funcionaria ora como TPG90 e ora como TPG150).

Os dados necessários à tomada de decisão são retirados das tabelas 3.1 e 3.2. Inicialmente as regras preferenciais, i.e., as com menor custo(linha 37), que satisfazem a Taxa de Cobertura de Falha (linha 33) e o Porcentual de Custo Adicional de Tempo Aceito (eventualmente pode ser encontrada uma solução diferente da de referência, com um Custo de *Hardware* menor, porém com um acréscimo no Tempo de Teste)(linha 34) são atribuídas ao par de registradores de teste TPGs correspondentes à determinada UF (nos exemplos apresentados aqui, um SA terá sempre a regra 90).

Finalmente é preenchida uma lista com todas as UFs e suas respectivas regras, além de seu Custo de *Hardware* associado. Se a mudança das Regras Preferenciais de cada UF isoladamente produzir alguma nova Solução Preferencial(linha 44), isto é, o custo for menor e o eventual aumento de Tempo de Teste estiver dentro do tolerado, esta nova solução passa a ser a preferencial e, mais uma vez, vai ter todas as regras de teste novamente mudadas, para que se determine qual é a solução com menor custo. Caso contrário, o algoritmo termina.

Assim, para o circuito Data3UF, a solução inicial (que coincidentemente é também a final, pois foi a solução com menor custo) é:

```
TPG90 TPG150 TPG150  ( Preferencial! )

#TPG90: reg 1
#TPG150: reg 6
#TPG90/150: reg 3
#SA90: reg 2
#SA/TPG90:
#SA/TPG150:
#SA/TPG90/150:

Custo de Hardware: 16.04 ( Preferencial! )
Custo de Tempo: 0.0214
```

Obs. – Os registradores foram identificados tanto por *reg* como por *R*, enquanto que nos arquivos do MACH as UFs foram sempre identificadas por *fca*.

O Custo de *Hardware* de 16.04 foi obtido considerando-se os 9 registradores em funcionamento normal, um TPG90, um TPG150, um TPG90/150 e um SA ($9+1.4+1.8+2.04+1.8 = 16.04$).

A Tab. 3.3 mostra o Plano de Teste para o circuito Data3FU. As regras 90-150-150 referem-se à UFs 1, 2 e 3 respectivamente, que foram associadas aos tipos de UF 1, 2, 3 da Tab 3.2. Na continuidade do algoritmo, cada uma das UFs terá sua regra alterada, formando um Plano de Teste, e se com isso o Custo de *Hardware* baixar, esta disposição passa a ser a preferencial e o processo se inicia novamente.

Tab. 3.3 – Plano de Teste para o circuito Data3FU

	90-150-150	150-150-150	90-90-150	90-150-90
# TPG 90	reg 1		reg 1	reg 1
# TPG 150	reg 6	reg 1, 3, 6		
# TPG 90/150	reg 3		reg 3, 6	reg 3, 6
# SA 90	reg 2	reg 2	reg 2	reg 2
# SA/TPG 90				
# SA/TPG 150				
# SA/TPG 90/150				
Custo Hardware	16.04	16.20	16.28	16.28
Custo Tempo	0.0214	0.0260	0.0229	0.0275

A solução final para o circuito Data3FU, com Cobertura de Falha = 0.9, utiliza 4 registradores, do total de 9, para fazer o Autoteste, como mostra a Tab. 3.4.

Tab. 3.4 – Relação de CAs para o Teste Série do Circuito Data3FU

UF	REGRA	ENTRADA 1	ENTRADA 2	SAÍDA
fca 1	90	reg 1	reg 3	reg 2
fca 2	150	reg 3	reg 6	reg 2
fca 3	150	reg 6	reg 3	reg 2

3.4 Fluxograma da Métrica Paralela

O Algoritmo Paralelo proposto em [Ama et al. 98] caracteriza-se pela busca da compatibilidade entre as UFs, para que elas possam ser testadas no menor intervalo de tempo e com o menor Custo de *Hardware* possíveis. O processo de otimização é controlado pelo usuário que define o grau de paralelismo desejado, ou seja, quantas UFs quer que sejam testadas ao mesmo tempo (devem existir TPGs e SAs disponíveis para as UFs testadas simultaneamente).

Fig. 3.11 - Principais Passos do Algoritmo Paralelo.

O Fluxograma da Figura 3.11 apresenta os dois principais blocos do Algoritmo Paralelo, ou seja, a Otimização do Tempo de Teste e a Atribuição de Registradores. O primeiro conjunto de tarefas concentra-se na definição do escalonamento da aplicação de testes para cada uma das UFs dentro de um critério de consumo de *hardware* e de tempo total de teste dentro de um aproveitamento máximo do paralelismo desejado. Não há restrições para o início e término da sessão de teste para qualquer UF em relação à outra.

Encerrada a fase de Otimização, a tarefa de Alocação de Registradores é iniciada. Esta fase de busca por uma solução de mínima Área/Tempo de Teste existe porque o escalonamento para o teste das UFs, realizada na etapa anterior, é implementada de forma que os registradores de teste sejam definidos apenas quando eles são únicos ou quando se fizer imprescindível. De outra forma, existem várias atribuições de registradores para teste em aberto que devem ser decididas na parte final do algoritmo. O procedimento de Atribuição de Registradores para o conjunto das UFs procura sempre a solução com maior número de compartilhamento e menor custo, cujos Custos Relativos são retirados da Tab. 3.1.

A Figura 3.12 apresenta o Algoritmo da Métrica Paralela em pseudocódigo. Os primeiros passos do Algoritmo Paralelo (linhas 1 a 24) são idênticos aos do Algoritmo Série, ou seja, a leitura de um arquivo inicial, a criação da representação interna do *data path*, a Alocação de Registradores Candidatos a CAs para cada UF, a Análise de Testabilidade e a eventual adaptação do circuito para que todas as UFs tenham suficientes registradores candidatos.

Para que duas UFs sejam testadas concorrentemente, é necessário que os seus caminhos de teste sejam independentes, isto é, que as duas UFs apresentem diferentes conjuntos de TPGs, Muxes, SAs e interconexões. Exceção pode ser feita para

UFs no que tange ao compartilhamento de TPGs. Se elas necessariamente compartilharem a mesma entrada, então, quando testadas ao mesmo tempo, terão que usar a mesma regra, pois um TPG não pode simultaneamente gerar Vetores de Teste sob as regras 90 e 150. Trata-se de uma restrição imposta pelos idealizadores do algoritmo para tornar o problema mais real

```

1-{
2-    If (Arquitetura Gerada pelo Mach){
3-        Lê Base de Dados do Mach;
4-        Cria Netlist;
5-
6-    }
7-    else{
8-        Lê Netlist;
9-
10-    }
11-
12-
13-    Cria a Lista de Registradores Candidatos;
14-
15-    Cria o Grafo do Circuito Sem Muxes;
16-    While (UF Não Visitada)
17-        Cria Lista Com Todos Registradores Associados a Cada UF;
18-
19-    // Verifica Se o Circuito Original Precisa de Alterações;
20-    Faz Análise de Testabilidade;
21-    While (UF Sem Registrador)
22-        Acrescenta Mux e Registrador ou Conexões aos Existentes;
23-
24-    Atualiza a Lista de Registradores Candidatos;
25-
26-    BFS_MuxReg;
27-
28-    Cria o Grafo de Compatibilidade;
29-    Determina a Compatibilidade Máxima Potencial;
30-    Encontra a Compatibilidade Máxima Efetiva;
31-
32-    Cria a Matriz de Estado de Teste Com o Número Escolhido de Colunas;
33-
34-    Faz a Escolha Final dos Registradores de Teste Com Menor Custo;
35-
36-    Imprime Plano de Teste da Primeira Solução Com Menor Custo;
37-}

```

Fig. 3.12 – Algoritmo Paralelo

Quando duas ou mais UFs são testadas simultaneamente, pode ocorrer de elas terem diferentes registradores candidatos a SA, porém dependentes do mesmo mux, como por exemplo, as UF₁ e UF₄ do *data path* da Fig. 3.13, que têm na saída os

registradores R_4 e R_5 , porém com o Mux_2 no caminho. Por isso, para verificar se todas as UFs concorrentes têm caminhos de teste independentes (i.e., pelo menos um SA e respectivas conexões até a UF completamente independentes) não basta consultar a lista de registradores candidatos; é necessário conhecer também todos os muxes associados aos registradores. Estas informações são obtidas através de uma busca em extensão do tipo *Breadth First Search* da linha 26 (BFS_MuxReg).

Fig. 3.13 – Circuito Data5UF

A Alocação dos Registradores ou Muxes associados a cada UF serve como ponto de partida para um Grafo de Compatibilidade (linha 28), que indica, através de um arco, quando duas UFs podem ser testadas em paralelo. Os casos de *Compatibilidade Condicional* (i.e., quando duas UFs podem ser testadas simultaneamente desde que usem a mesma regra) são representados num Grafo de Compatibilidade com arcos associados a um índice 0, enquanto que os casos de *Compatibilidade Plena* (saídas e entradas totalmente independentes) por arcos com índice 1.

A Fig. 3.14 mostra o Grafo de Compatibilidade do Circuito Data5UF (Fig. 3.13) com 5 UFs, ressaltando a Compatibilidade Plena entre **fca 1** e **fca 3**, entre **fca 1** e **fca 5**, entre **fca 2** e **fca 4**, entre **fca 3** e **fca 4** e entre **fca 4** e **fca 5**; um caso de Compatibilidade Condicional entre **fca 2** e **fca 3**, além da Incompatibilidade entre **fca 1** e **fca 2**, entre **fca 1** e **fca 4**, entre **fca 2** e **fca 5** e entre **fca 3** e **fca 5**.

Fig. 3.14 – Grafo de Compatibilidade para o Circuito Data5UF

Embora o Grafo de Compatibilidade mostre apenas as compatibilidades entre duas UFs, ele revela também a informação da Compatibilidade Máxima Potencial (linha 29) que o conjunto pode atingir, pois para que um conjunto tenha compatibilidade N_C são necessárias N_C UFs com compatibilidade (N_C-1) (porém a existência de N_C UFs com compatibilidade (N_C-1) não garante que o conjunto atinja a compatibilidade N_C).

Por ex., o grafo da Fig. 3.14 mostra a existência de pelo menos 3 UFs com compatibilidade 2, o que implica que a Compatibilidade Máxima do conjunto será

menor ou igual a 3. De fato, as **fcas 2, 3 e 4** poderão ser testadas simultaneamente, desde que as fcas **2 e 3** sejam testadas sob a mesma regra (pois necessariamente compartilham pelo menos um dos TPGs).

Levando-se em conta as restrições impostas pelo paralelismo, a Alocação Inicial de Registradores é feita da mesma forma que a Métrica Série. Note que a Fig. 3.7 mostra apenas a primeira fase da Alocação de Registradores, pois, durante a distribuição efetiva de registradores à cada UF, verifica-se que nos testes simultâneos de UFs, o número de alternativas geralmente diminui. Por exemplo, no circuito Data3UF da Fig. 3.4, a UF₂ tem como alternativa de TPG em uma de suas entradas apenas o registrador R₆. Ora, se essa UF for testada simultaneamente com as outras duas, então R₆ deixa de ser uma alternativa de saída.

O Grafo de Compatibilidade da Fig. 3.15 mostra que as 3 UFs poderiam ser testadas simultaneamente.

Fig. 3.15 – Grafo de Compatibilidade para o Circuito Data3UF

Uma vez que o Grafo de Compatibilidade apresenta a compatibilidade dois a dois, a comprovação de que a Compatibilidade Máxima Potencial pode ser efetivada será obtida depois que uma rotina de busca encontre a solução máxima possível, ou seja a Compatibilidade Máxima Efetiva (linha 30). Com isso, o usuário é solicitado a escolher um valor máximo de teste simultâneo de UFs, compreendido entre 1 e a Compatibilidade Máxima Efetiva.

Os idealizadores do algoritmo paralelo não deram solução satisfatória à obtenção da Compatibilidade Máxima Efetiva. Nesta Dissertação, foi proposta uma rotina de combinações entre o conjunto de registradores e UFs, até que se encontre o seu valor. Se na prática se verificar que a Compatibilidade Máxima Potencial não pode ser atingida, a rotina de combinações busca a solução com paralelismo imediatamente abaixo, e assim sucessivamente, até que se encontre a Compatibilidade Máxima Efetiva.

A rotina de combinações não precisa executar todas as possibilidades dadas por uma análise combinatória entre UFs, muxes e registradores, porque efetivamente a escolha de um registrador quase sempre implica a exclusão de muitos outros (por ex., a escolha de um registrador associado a um mux exclui todos os outros registradores dependentes do mesmo mux).

Para mapear estes Desdobramentos Sucessivos, são construídas tabelas, que expressam uma busca em extensão, hierarquizando as Combinações Efetivas. Para o Circuito Data5FU, da Fig. 3.13, essas Combinações Efetivas de Saídas estão ilustradas inicialmente nas Tab. 3.5, 3.6 e 3.7.

1º Passo – São chamadas de Saídas Primárias aquelas alternativas de saídas diretamente ligadas à uma UF, como ilustra a Tab. 3.5. As Saídas Primárias deste exemplo se desdobram em 6 possíveis Caminhos de Teste (1x2x3x1x1).

Tab. 3.5 – Lista de Saídas Primárias

LISTA DE SAÍDAS PRIMÁRIAS				
UF₁	UF₂	UF₃	UF₄	UF₅
Mux ₂	Mux ₁	R ₇	Mux ₂	R ₁
	R ₃	Mux ₁		
		Mux ₂		

2º Passo – A Tab. 3.6 mostra as 6 combinação possíveis obtidas pelo processo de Desdobramentos Sucessivos das Saídas Primárias.

Tab. 3.6 – Desdobramento 1

Lista de Saídas Combinadas 1						
Caminho	UF₁	UF₂	UF₃	UF₄	UF₅	Compatib.
1	Mux ₂	Mux ₁	R ₇	Mux ₂	R ₁	4
2	Mux ₂	Mux ₁	Mux ₁	Mux ₂	R ₁	3
3	Mux ₂	Mux ₁	Mux ₂	Mux ₂	R ₁	3
4	Mux ₂	R ₃	R ₇	Mux ₂	R ₁	4
5	Mux ₂	R ₃	Mux ₁	Mux ₂	R ₁	4
6	Mux ₂	R ₃	Mux ₂	Mux ₂	R ₁	3

3º Passo – A análise da Lista de Saídas Combinadas 1 mostra que as saídas 1, 4 e 5 são as de maior compatibilidade, e que além disso, elas podem ser desdobradas em novas combinações. Por ex., pela Tab. 3.6 verifica-se que o Caminho1 da Lista de Saídas Combinadas oferece a possibilidade inicial de teste simultâneo das UFs (1, 2, 3 e 5) e UFs (2, 3, 4 e 5). Quando o mesmo raciocínio é aplicado nas Saídas Combinadas 4 e 5, resulta que estas 3 saídas são desdobradas em 6, como mostra a Tab. 3.7.

Tab. 3.7 – Desdobramento 2

Lista de Saídas Combinadas 2						
Caminho	UF₁	UF₂	UF₃	UF₄	UF₅	Compatib.
1A	Mux ₂	Mux ₁	R ₇	-----	R ₁	4
1B	-----	Mux ₁	R ₇	Mux ₂	R ₁	4
4A	Mux ₂	R ₃	R ₇	-----	R ₁	4
4B	-----	R ₃	R ₇	Mux ₂	R ₁	4
5A	Mux ₂	R ₃	Mux ₁	-----	R ₁	4
5B	-----	R ₃	Mux ₁	Mux ₂	R ₁	4

Como o Mux₂ apresenta duas alternativas de saída (R₄ e R₅), o processo de Desdobramentos Sucessivos continua, produzindo 12 Caminhos de Teste, como mostram as Tab. 3.8, 3.9, 3.10, 3.11, 3.12 e 3.13.

Tab. 3.8 – Desdobramento 3A

Lista de Saídas Combinadas 3A						
Caminho	UF₁	UF₂	UF₃	UF₄	UF₅	Compatib.
	Mux ₂	Mux ₁	R ₇	-----	R ₁	
1	R ₄	R ₂	R ₇	-----	R ₁	4
2	R ₅	R ₂	R ₇	-----	R ₁	4

Tab. 3.9 – Desdobramento 3B

Lista de Saídas Combinadas 3B						
Caminho	UF₁	UF₂	UF₃	UF₄	UF₅	Compatib.
	-----	Mux ₁	R ₇	Mux ₂	R ₁	
3	-----	R ₂	R ₇	R ₄	R ₁	4
4	-----	R ₂	R ₇	R ₅	R ₁	4

Tab. 3.10 – Desdobramento 3C

Lista de Saídas Combinadas 3C						
Caminho	UF₁	UF₂	UF₃	UF₄	UF₅	Compatib.
	Mux ₂	R ₃	R ₇	-----	R ₁	
5	R ₄	R ₃	R ₇	-----	R ₁	4
6	R ₅	R ₃	R ₇	-----	R ₁	4

Tab. 3.11 – Desdobramento 3D

Lista de Saídas Combinadas 3D						
Caminho	UF₁	UF₂	UF₃	UF₄	UF₅	Compatib.
	-----	R ₃	R ₇	Mux ₂	R ₁	
7	-----	R ₃	R ₇	R ₄	R ₁	4
8	-----	R ₃	R ₇	R ₅	R ₁	4

Tab. 3.12 – Desdobramento 3E

Lista de Saídas Combinadas 3E						
Caminho	UF₁	UF₂	UF₃	UF₄	UF₅	Compatib.
	Mux ₂	R ₃	Mux ₁	-----	R ₁	
9	R ₄	R ₃	R ₂	-----	R ₁	4
10	R ₅	R ₃	R ₂	-----	R ₁	4

Tab. 3.13 – Desdobramento 3F

Lista de Saídas Combinadas 3F						
Caminho	UF₁	UF₂	UF₃	UF₄	UF₅	Compatib.
	-----	R ₃	Mux ₁	Mux ₂	R ₁	
11	-----	R ₃	R ₂	R ₄	R ₁	4
12	-----	R ₃	R ₂	R ₅	R ₁	4

As Tab. 3.8, 3.9, 3.10, 3.11, 3.12 e 3.13 mostram que o número máximo de Caminhos de Saídas Independentes de Teste para a arquitetura do Circuito Data5UF é 4, ou seja, o número de saídas limita a Compatibilidade Máxima em 4.

Por um processo semelhante, a rotina de combinações tenta encontrar registradores de entrada para cada uma das soluções de saída encontradas. Quando não existir solução para uma certa compatibilidade, é feita uma nova busca, através de Desdobramentos Sucessivos, para a compatibilidade imediatamente abaixo. Por ex., para este circuito, verifica-se que a Compatibilidade Máxima Efetiva é 3, como mostra a Tab. 3.14, baseada num Desdobramento da Solução de Saída 3.

Tab. 3.14 – Desdobramento 3X

Lista de Entradas e Saídas Combinadas 3X						
	UF₁	UF₂	UF₃	UF₄	UF₅	Compatib.
Saída	-----	Mux ₁	R ₇	Mux ₂	-----	
3	-----	R ₂	R ₇	R ₄	-----	3
Entrada						
X	-----	R ₁ ou R ₅	R ₅ ou R ₁	R ₈ ou R ₉	-----	3

Como o processo de Desdobramentos Sucessivos monta apenas as Combinações Efetivas, partindo sempre dos Caminhos de Teste de Saída (em geral, numericamente muito menores que os de entrada), o resultado é um número de soluções efetivamente computadas drasticamente inferior ao número de combinações teoricamente possíveis. Tão logo é encontrada uma Solução Efetiva, o processo de Desdobramentos Sucessivos é suspenso.

Definido o grau de paralelismo desejado pelo usuário, inicia-se a construção da Matriz de Estado de Teste (linha 32). A ordem em que as UFs serão testadas é determinada pelo preenchimento dessa Matriz de Estado de Teste, que essencialmente consiste de solução parcial, apresentando os tempos de início de teste das UFs e os comprimentos de teste. Usa-se uma técnica semelhante à de *list scheduling* [Gaj et al. 92], e vai se completando de forma construtiva. A seqüência em que cada uma das UFs será analisada é codificada numa *lista de prioridade* de todas UFs para que disputas decorrentes da limitação de recursos sejam resolvidas de acordo com a posição da UF nessa lista.

Para a Matriz de Estado de Teste, uma lista de prioridades classifica as UFs de acordo com o grau de compatibilidade de uma UF em relação ao conjunto, ou de acordo com o número de vetores necessários para testá-la, dependendo da opção feita pelo usuário. As colunas, onde são “fixadas” as UFs formando um núcleo, representam o paralelismo máximo entre UFs, e as linhas (nesse caso não equidistantes como nos passos de controle da *list scheduling* normal) delimitam os estados de teste, caracterizados pelo início ou fim do teste de uma UF. Como a Matriz de Estado implementa o *Scheduling* das Sessões de Teste, os limites dos Estados de Teste são determinados dinamicamente.

A Fig. 3.16 mostra a Matriz de Estado para o Circuito Data3UF (com a opção arbitrária de se testar apenas duas UFs simultaneamente), em que se observa claramente diferentes tempos de duração para cada Estado de Teste.

Fig. 3.16 – Matriz de Estado Para o Circuito Data3UF

O Paralelismo Máximo é igual ao número de colunas que a Matriz de Estado vai ter. Da lista de prioridades é retirada uma UF e fixada numa das colunas, na posição em que a Função Custo da expressão (3.5) apresentar o menor valor.

$$F_COST = COST_HW - G_TIME - G_ASS \quad (3.5)$$

A alocação da primeira UF no núcleo é trivial: ou o núcleo se inicia com a solução de paralelismo inicial já encontrada (como uma opção por Tempo de Teste mais longo), ou uma nova tentativa é feita, baseada inteiramente nos custos de cada possível posição de uma UF dentro da Matriz

No cálculo da Função Custo são considerados três fatores distintos: o Custo de *Hardware* ($COST_HW$), o Ganho de Tempo (G_TIME) e o Ganho de Assimetria (G_ASS).

$COST_HW$ (Custo de *Hardware*) considera as mudanças de área resultantes da transformação de registradores normais em CAs ou do compartilhamento dos CAs existentes por duas ou mais UFs, testadas sob a mesma regra, como mostra a expressão (3.6)

$$COST_HW = \frac{A_TPG - t.(l+r).\Delta A_TPG}{N} \quad (3.6)$$

A_TPG , cujos valores relativos são retirados da Tab. 3.1, representa o aumento de área devido à transformação de 3 registradores normais em registradores de teste, ΔA_TPG , a redução de área devido a um eventual compartilhamento de registrador à esquerda e/ou à direita (l e/ou r igual a 1, neste caso), t assume valores 1 ou 0, dependendo do compartilhamento ou não da mesma regra pelos registradores e N é o número total de registradores do circuito.

Suponha, por exemplo, que os registradores R_7 e R_1 estejam sendo considerados como TPGs para a UF_3 do Circuito Data5UF da Fig. 3.13, sob regra 90, e que R_1 já tenha sido definido como TPG para a UF_2 , sob regra 90. Neste caso, $COST_HW$ seria:

$$COST_HW = \frac{(2 * 1.4 + 1.8) - 1 * (0 + 1) * 1.4}{9} = 0.355$$

G_TIME (Ganho de Tempo) corresponde ao ganho resultante do posicionamento de uma UF na Matriz de Teste que pode estender ou não a duração geral do teste (C_t_L), comparado com a Sessão de Teste se ela fosse implementada em série ($C_t_core + C_t_FU$), como mostra a expressão (3.7).

$$G_TIME = \frac{C_t_core + C_t_FU_ref - C_t_L}{C_t_ref} \quad (3.7)$$

C_t_core expressa o Tempo de Teste num determinado estágio do *scheduling* da coluna de teste com menor custo possível, $C_t_FU_ref$, o Tempo de Teste da UF sendo posicionada na Matriz de Teste, C_t_L , o valor da coluna mais longa e C_t_ref , o Tempo de Teste da UF com a mais longa Sessão de Teste Preferencial, tomada como referência.

Suponha que na Matriz de Estados de Teste da Fig. 3.16, uma UF₅ esteja sendo avaliada na coluna 1, com um número de Vetores de Teste ($C_t_FU_ref$) igual a 1600 e que C_t_ref também seja 1600. Neste caso, G_TIME seria:

$$G_TIME = \frac{1400 + 1600 - 2000}{1600} = 0.625$$

G_ASS (Ganho de Assimetria) reflete o impacto das sessões de teste com diferentes durações, quando posicionadas, em paralelo, nas colunas permitidas. Com isso, avalia-se o paralelismo restante em cada Estado de Teste individual, depois da eventual inclusão da UF no *core* (núcleo), como mostra a expressão (3.8).

$$G_ASS = \sum_{st} \frac{C_time(st).Min(Comp_FU(st), tr_left(st))}{C_t_ref.Max_parallel(st)} \quad (3.8)$$

$C_time(st)$ é a duração de cada Estado de Teste, $Comp_FU(st)$ e $tr_left(st)$ refletem o número de UFs ou colunas à disposição depois de fixada em determinada posição a UF em análise, e $Max_parallel(st)$ é o paralelismo máximo.

Suponha o exemplo da Matriz de Estados de Teste da Fig. 3.16, citado anteriormente. Para este caso, o G_ASS para os Estados de Teste 1, 2 e 3 seria 0, já que não restaria nenhuma coluna livre quando a UF₅ fosse posicionada na coluna 1. Para o Estado 4, que se iniciaria em 2000 e terminaria em 3000, o G_ASS seria:

$$G_ASS = \sum_{st} \frac{1000 * Min(1,1)}{1600 * 2} = 0.3125$$

Baseado na Função Custo (F_COST), a UF de custo menor será agregada ao núcleo atual formando um novo núcleo. Antes de consolidar a solução, é feita uma verificação de Compatibilidade Efetiva se tiver ocorrido aumento de paralelismo. Aí

reside a importância de se ter a Compatibilidade Efetiva e não a Potencial pois, com a última, a rotina poderia levar à uma infinidade de soluções intermediárias em que a compatibilidade efetiva não existiria, tornando a rotina extremamente ineficiente.

Uma vez conhecida a disposição final de todas as UFs, é iniciado um procedimento de Atribuição de Registradores (linha 34) para o conjunto das UFs, procurando sempre a solução com maior número de compartilhamento e menor custo, cujos Custos Relativos são retirados da Tab. 3.1. O procedimento final de alocação é semelhante à alocação final do algoritmo série.

3.5 Os Algoritmos e Sua Implementação

A resolução algorítmica ao Problema do Autoteste (Série e Paralelo), apresentada na Fig. 3.3 e Fig. 3.12, teve o seu código implementado em C++, com os compiladores da GNU. Os dados necessários à tomada de decisão são retirados das tabelas de dois arquivos. Primeiro, é feita uma consulta ao arquivo B_TPG_SA para avaliar o Custo Unitário Relativo de *Hardware* da Tab. 3.1 e, depois, o programa busca nas bibliotecas de dados pré-compilados, B_UF, os parâmetros de teste relativos a cada tipo de UF, como o mostrado na Tab. 3.2.

O formato de entrada de dados para o programa computacional (BIST_S_P.c), que implementa os Algoritmos Série e Paralelo, pode ser o arquivo de saída do HLS Sintetizador Mach ou uma *Netlist* com o formato mostrado na Fig.3.17. A primeira coluna da *Netlist* indica o elemento inicial da conexão, a segunda coluna indica o outro extremo da conexão e o número na terceira coluna, o terminal de entrada do elemento da segunda coluna. Por ex., a primeira linha da *Netlist* da Fig. 3.17 representa uma conexão partindo da fca 1 e chegando ao terminal 1 do mux 5.

fca 1	mux 5 1
fca 1	mux 6 1
fca 2	mux 6 2
fca 3	mux 5 2
fca 3	mux 6 3
fca 3	reg 9 1
mux 1	fca 1 1
mux 2	fca 2 1
mux 3	fca 3 1
mux 4	fca 3 2
mux 5	reg 3 1
mux 5	reg 4 1
mux 5	reg 6 1
mux 6	reg 2 1
mux 6	reg 7 1
reg 1	mux 1 1
reg 2	mux 1 2
reg 3	fca 1 2
reg 3	mux 2 1
reg 3	mux 4 1
reg 4	mux 2 2
reg 5	mux 2 3
reg 6	fca 2 2
reg 6	mux 3 1
reg 7	mux 3 2
reg 8	mux 4 2

Fig. 3.17 - Netlist do Circuito Data3UF (da Fig. 3.4)

Como é de se prever, uma *netlist* tem um número variável de itens. Por esta razão, a adoção de uma estrutura de dados de tamanho fixo para implementar listas de elementos pode se mostrar inadequada, seja por ter um tamanho menor que o necessário ou por alocar um espaço maior do que o efetivamente utilizado. Uma forma de contornar esse inconveniente é através de estruturas dinâmicas de dados, que crescem ou diminuem durante o tempo de execução, tais como as *Linked Lists*.

Após a leitura da *netlist*, uma Lista Dinâmica do Circuito é implementada na forma de uma lista linear, i.e., uma coleção linear de classe de objetos auto-referenciados [Dei e Dei 94], formando um seqüência de itens do tipo (list_item) [Mehl S/D] , como ilustra a Fig. 3.18, referente ao circuito Data3UF.

```
fca 1 fca 2 fca 3 mux 1 mux 2 mux 3 mux 4 mux 5 mux 6 reg 1 reg 2 reg  
3 reg 4 reg 5 reg 6 reg 7 reg 8 reg 9
```

Fig. 3.18 – Lista Linear do circuito Data3UF (da Fig. 3.4)

Este tipo de lista pode ser implementada de forma bem compacta dentro de pacotes como o LEDA (Library of Efficient Data types and Algorithms) [Mehl S/D], que, além de trazerem inúmeros comandos úteis (ordenamento, busca por conteúdo, por ponteiro, por posição, *merge* de listas etc.), incorporam outras formas de manipulação de tipos complexos de dados, como *graphs*, *dictionaries*, *queues*, *graphics* etc.

Nos Algoritmos, uma lista de todos os componentes do circuito fornece os elementos para a implementação de um *directed graph* que modele o *Data Path* de um circuito genérico, criando um *DataGraph*, no qual cada item da Lista de Componentes é um nó e os ramos entre os nós correspondem às ligações do circuito (especificadas na *Netlist*), como mostra a Fig. 3.19.

Fig. 3.19 – DataGraph para o circuito Data3UF

O Grafo apresentado na tela do computador pode ser modelado, arrastado, receber novos nós, conexões etc., além de muitos recursos gráficos que auxiliem a análise do circuito através da Teoria dos Grafos.

Este mesmo *graph* é representado analiticamente através de um arquivo específico do LEDA, com o formato mostrado na Fig. 3.20, mais adequado para o processamento dos dados.

```
[0](fca 1) : [0]--(1)-->[7][0]--(1)-->[8]
[1](fca 2) : [1]--(2)-->[8]
[2](fca 3) : [2]--(2)-->[7][2]--(3)-->[8][2]--(1)-->[17]
[3](mux 1) : [3]--(1)-->[0]
[4](mux 2) : [4]--(1)-->[1]
[5](mux 3) : [5]--(1)-->[2]
[6](mux 4) : [6]--(2)-->[2]
[7](mux 5) : [7]--(1)-->[11][7]--(1)-->[12][7]--(1)-->[14]
[8](mux 6) : [8]--(1)-->[10][8]--(1)-->[15]
[9](reg 1) : [9]--(1)-->[3]
[10](reg 2) : [10]--(2)-->[3]
```

```

[11](reg 3) : [11]--(2)-->[0][11]--(1)-->[4][11]--(1)-->[6]
[12](reg 4) : [12]--(2)-->[4]
[13](reg 5) : [13]--(3)-->[4]
[14](reg 6) : [14]--(2)-->[1][14]--(1)-->[5]
[15](reg 7) : [15]--(2)-->[5]
[16](reg 8) : [16]--(2)-->[6]
[17](reg 9) :

```

Fig. 3.20 – Arquivo Específico do LEDA para o *DataGraph* da Fig. 3.19

No arquivo LEDA da Fig. 3.20, cada um dos 18 nós, numerados de [0] a [17], representa um dos componentes do circuito Data3UF. Por ex., o nó [0] representa a UF fca 1, que possui duas ligações de saída: uma para o nó [7] (mux 5) através do ramo 1, cujo número coincide com o terminal de entrada do mux 5, e outra para o nó [8] (mux 6), através do ramo igualmente numerado 1 porque a conexão vai até o terminal 1 do mux 6.

Com base no *DataGraph*, a Lista de todos os Registradores Candidatos é gerada. Com a facilidade do pacote LEDA, por meio de um comando simples, todos os Muxes são retirados do Circuito Original, de tal forma que os Registradores se comuniquem diretamente com as UFs, e é criada a Lista Dinâmica de todos os Registradores associados a cada UF.

As Tabelas de Entrada e Saída foram implementadas de maneira computacionalmente mais eficaz que as representações visuais apresentadas na Fig. 3.8, através de *dictionary arrays*, para dados parametrizados do tipo $d_array<I,E>$, com I representando os registradores, ordenados linearmente, e E o número de ocorrência deles como candidatos, formando um mapeamento injetor. Dessa forma, o Ganho dos Registradores pode ser facilmente calculado como a combinação do número de ocorrência individual de cada registrador, dois a dois.

Como particularidade do Algoritmo Paralelo, o Grafo de Compatibilidade é montado como uma lista de nós V , um par de nós $(v, w) \in V \times V$ é associado a cada

ramo $e \in E$, sendo v denominado *origem* de e e w *alvo* de e , com v e w sendo chamados de *pontos extremos* de e . Como o Grafo de Compatibilidade é um grafo não direcionado, a cada nó v está associada uma lista de todos os ramos incidentes em v , chamada de ramos adjacentes, com a seguinte relação: $adj_edges(v) = \{ e \in E \mid v \in \{origem(e), alvo(e)\} \}$ e ele não pode conter *self-loop*, i.e., não há nele um ramo cuja origem coincida com o alvo [Mehl S/D]. A lista de ramos E corresponde ao conjunto de compatibilidades duas a duas das UFs.

3.6 Análise, Projeto e Implementação dos Programas BIST_S e BIST_P

3.6.1 Metodologia Baseada em Objetos

O uso de linguagens baseadas em objetos tem produzido resultados muito bons para projetos de grande porte, com extensos programas contendo milhares de linha de comando e certa complexidade nas tarefas realizadas, exigindo que as fases de análise, projeto e implementação não sejam processos disjuntos [Gray 94] como ocorrem freqüentemente nos projetos *top-down* convencionais, que enfatizam a funcionalidade do sistema.

Como os programas BIST_S e BIST_P estão inseridos num escopo bem mais amplo do que aquele descrito aqui, futuramente novos objetos devem ser acrescentados e/ou modificados. Para casos como este, o projeto baseado em objetos pode significar uma melhor separação de seus componentes do que nos projetos *top-down*, já que nestes, a alteração de uma função costuma gerar um efeito cascata, forçando a mudança de outras funções [Joh et al. 95], diferentemente dos projetos baseado em objetos, que fundem numa mesma entidade os dados e suas correspondentes operações.

Embora autores como [Rum 94] busquem uma unificação dos métodos baseados em objetos, pela Técnica de Modelagem de Objetos ([Rum et al. 91] e [Rum et al. 97]),

a metodologia para o desenvolvimento de um modelo do domínio da aplicação a partir da abstração do mundo real e a geração de um projeto é dividida em três etapas:

- **Análise de Requisitos**, vista como um entendimento do que o sistema deverá fazer e com quais atores. Mais concretamente, esta etapa de Modelamento da Aplicação se baseou na leitura dos *papers* [Ama et al. 97a], [Ama et al. 97b] e [Ama et al. 98] para extrair as principais propriedades do modelo e na elaboração do Modelo de Objetos, do Modelo Dinâmico e do Modelo Funcional.
- **Projeto do Sistema e dos Objetos**, entendido como um detalhamento das estruturas de dados e dos algoritmos pertencentes a cada classe. Essencialmente consistiu no refinamento dos três modelos que, baseados na estrutura lógica do Modelo de Análise, delimitaram a Organização Física dos Programas BIST_S e BIST_P no Domínio de Computação.
- **Implementação**, entendida como a codificação em uma linguagem de programação das Classes de Objetos e seus Relacionamentos, num Sistema Flexível. Concretamente, os programas BIST_S e BIST_P, foram escritos em C++, com a ajuda das bibliotecas LEDA [Mehl S/D], em Estações de Trabalho Sun.

OBS – Está no Apêndice 2 a seção 3.6.2: Resultados Obtidos pela Aplicação da Metodologia Baseada em Objetos.

3.7 Resumo

Este capítulo discorre sobre a forma como foram implementados os Algoritmos Série e Paralelo. No Algoritmo Série, todas as UFs são testadas sequencialmente, aumentando as possibilidades de que um CA seja compartilhado por várias UFs, mas aumentando o Tempo de Teste, enquanto que no Algoritmo Paralelo, procura-se

explorar o paralelismo máximo da arquitetura do CI, diminuindo a probabilidade de associação de um CA a várias UF, porém tornando o teste mais rápido.

A solução heurística para o problema NP-completo da inserção de Registradores de Teste consiste de duas fases.

Na primeira, todos os registradores que necessariamente funcionarão como SAs são retirados da lista de candidatos à TPG, assim como aqueles que necessariamente serão TPGs são retirados da lista de candidatos a SA.

Na segunda fase, para o BIST_S, são calculados os ganhos dos registradores, i.e., o grau de compartilhamento dos registradores, e aqueles com maior ganho são escolhidos para SA ou TPG. Para o BIST_P, as Saídas Primárias das UFs determinam os Caminhos de Teste Independentes e por meio de um processo de Desdobramentos Sucessivos somente as combinações de registradores que resultem em Soluções Efetivas de Saída são determinadas, representando uma sensível diminuição no custo computacional. O mesmo processo é aplicado na busca do conjunto de registradores de entrada candidatos. Uma Função Custo, que avalia qual solução acarreta o aumento mínimo de *hardware* na escolha da seqüência de testes das UFs, decide qual das alternativas é a mais atraente.

Foram discutidas a Análise, o Projeto e a Implementação dos Algoritmos BIST_S e BIST_P, dentro de um contexto de Programação Baseada em Objetos, na linguagem C++, ressaltando os bons resultados desta técnica para sistemas de considerável complexidade.

4 APRESENTAÇÃO E ANÁLISE DOS RESULTADOS EXPERIMENTAIS

Vários exemplos, geralmente conhecidos *benchmarks* da literatura para HLS, são apresentados neste Capítulo para testar os programas BIST_S e BIST_P, apresentados no Capítulo 3. Alguns destes exemplos são mostrados aqui de forma detalhada para que se compreenda mais concretamente como os programas levaram aos resultados finais. A maioria destas arquiteturas foi sintetizada no programa MACH, da Université de Montpellier, na França, enquanto que outras foram obtidas diretamente das fontes citadas.

4.1 Resultados com o Algoritmo Série

O circuito DiffEq da Fig. 4.1 [Ama et al. 87b] é um bom estudo de caso para ilustrar o funcionamento do programa BIST_S, porque, apesar de pequeno, várias dificuldades típicas estão contidas nele. Trata-se de uma das possíveis implementações do *benchmark diffeq*, com dois multiplicadores (UF_1 e UF_3) e um somador (UF_2).

Fig. 4.1 – Circuito DiffEq

Inicialmente o BIST_S detecta que nem todas UFs têm registradores candidatos, mostrando a mensagem abaixo na tela do computador:

```
fca14 1 precisa de um Registrador na Entrada (1)
fca 3 precisa de um Registrador na Entrada (1)
```

e oferece duas alternativas:

Nem Todas Unidades Funcionais Possuem Registradores Suficientes
para o Teste!

Escolha Uma das Alternativas:

```
< 1 > - Acrescentar Novos Registradores para Teste!
< 2 > - Realocar os Registradores Existentes!
```

? 1

A opção 1 foi deliberadamente escolhida para mostrar como o BIST_S acrescenta registradores e muxes de teste (neste caso para as UFs 1 e 3), pois o registrador R₅ não pode desempenhar ao mesmo tempo a função de TPG e SA da UF₁; já a UF₃ simplesmente não possui nenhum registrador acoplado a uma de suas entradas.

A fig. 4.2 mostra o arquivo interno do aplicativo LEDA com os acréscimos de dois pares de componentes: o registrador R₇ conectando-se ao mux₁₀ e o registrador R₈ ao mux₁₁.

Grafo Adaptado Temporariamente ao BIST:

```
[0](fca 1) : [0]--(1)-->[23]
[1](fca 2) : [1]--(2)-->[3][1]--(3)-->[4][1]--(2)-->[5]
[2](fca 3) : [2]--(1)-->[24]
[3](mux 1) : [3]--(1)-->[19]
[4](mux 2) : [4]--(1)-->[21]
[5](mux 3) : [5]--(1)-->[20]
[6](mux 4) : [6]--(1)-->[25]
[7](mux 5) : [7]--(2)-->[0]
[8](mux 6) : [8]--(1)-->[1]
[9](mux 7) : [9]--(2)-->[1]
[10](mux 8) : [10]--(1)-->[27]
[11](mux 9) : [11]--(2)-->[2]
[12](por 1) : [12]--(2)-->[6]
[13](por 2) : [13]--(2)-->[9]
[14](por 3) : [14]--(3)-->[9]
[15](por 4) : [15]--(1)-->[10]
```

¹⁴ fca 1 e fca 3 referem-se à UF₁ e à UF₃

```

[16](por 5) : [16]--(2)-->[10]
[17](por 6) :
[18](por 7) : [18]--(1)-->[3][18]--(2)-->[4][18]--(1)-->[5]
[19](reg 1) : [19]--(2)-->[7][19]--(1)-->[8]
[20](reg 2) : [20]--(1)-->[4][20]--(2)-->[8][20]--(2)-->[11][20]--(1)-->[17]
[21](reg 3) : [21]--(4)-->[8][21]--(1)-->[11]
[22](reg 4) : [22]--(3)-->[8]
[23](reg 5) : [23]--(1)-->[6][23]--(1)-->[9]
[24](reg 6) : [24]--(1)-->[7]
[25](mux 10) : [25]--(1)-->[0]
[26](reg 7) : [26]--(2)-->[25]
[27](mux 11) : [27]--(1)-->[2]
[28](reg 8) : [28]--(2)-->[27]

```

Fig. 4.2 – Arquivo Interno do LEDA com as Alterações para o DiffEq.

A Fig. 4.3 mostra o circuito DiffEq com as modificações implementadas, sendo este novo circuito o ponto de partida para o BIST_S.

Fig. 4.3 – Circuito DiffEq Adaptado

O espaço inicial de soluções para este circuito, mostrado na Fig. 4.4, já apresenta algumas escolhas inevitáveis de registradores candidatos, pois quando existir apenas um

registrador para uma determinada posição, este registrador é “fixado” neste local, representado por um retângulo na figura.

$$\begin{aligned} \text{UF}_1: & \boxed{R_7} \quad \{ R_1, R_6 \} \parallel \boxed{R_5} \\ \text{UF}_2: & \{ R_1, R_2, R_3, R_4 \} \quad \boxed{R_5} \parallel \{ R_1, R_2, R_3 \} \\ \text{UF}_3: & \boxed{R_8} \quad \{ R_2, R_3 \} \parallel \boxed{R_6} \end{aligned}$$

Fig. 4.4 – Alocação de Registradores para o Circuito DiffEq no Passo 0

Nesta fase do algoritmo, referida como Passo 0, a Lista de Registradores Candidatos é composta pelos seguintes registradores:

$$\text{Lista de Registradores Candidatos} = \{ R_1, R_2, R_3, R_4, R_6 \}$$

Como os registradores R_5 , R_7 e R_8 não disputam uma posição com outros registradores, eles estão fora da Lista de Registradores Candidatos.

As correspondentes Tabelas de Entrada e Saída, mostradas na Fig. 4.5, revelam que não há registradores de saída compartilhados por mais de uma UF (por isso, a tabela OUT está vazia), que há 3 registradores formando um potencial *self-loop* (na realidade, aqui é possível destinar alguns deles para a entrada e outro para a saída) e que há candidatos a BILBO, já que funcionam ora como registradores de entrada e ora de saída.

IN	UF ₁	UF ₂	UF ₃	OUT	UF ₁	UF ₂	UF ₃	I/O	UF ₁	UF ₂	UF ₃
UF ₁		R ₁		UF ₁				UF ₁		R ₁	R ₆
UF ₂			R ₂ R ₃	UF ₂				UF ₂	R ₅		
UF ₃				UF ₃				UF ₃		R ₂	
										R ₃	

IN(1, 2, 3) = 1;

I/O(1, 2, 3, 5, 6) = 1;

SL(1, 2, 3) = 1;

Fig. 4.5 – Tabelas de Entrada e Saída para o Circuito DiffEq no Passo 0

O Ganho dos Registradores da Lista de Registradores Candidatos foi calculado de acordo com a expressão (3.3), resultando em:

$$\begin{array}{ll} \text{GR}(1) = 2 & \text{GR}(4) = 0 \\ \text{GR}(2) = 2 & \text{GR}(6) = 1 \\ \text{GR}(3) = 2 & \end{array}$$

refletindo o fato de que os candidatos a registrador de saída têm um peso maior que os demais, constituindo-se numa prioridade sobre os de entrada, e que a adoção de BILBOs deve ser minimizada, pois os candidatos que estão simultaneamente nas entradas e saídas terão ganhos menores.

Como o custo de um SA é maior do que o de um TPG, os registradores de saída devem ser escolhidos antes que os de entrada, e como o primeiro registrador da Lista de Registradores Candidatos com maior ganho é R_1 , o resultado é que este registrador é o escolhido para a saída da UF_2 , como se vê pela Fig. 4.6. (Nessa altura, como R_1 estava na lista de candidatos de entrada da UF_2 , ele é automaticamente retirado dela.)

$$\begin{array}{lll} \text{UF}_1: \boxed{R_7} & \{ R_1, R_6 \} \parallel & \boxed{R_5} \\ \text{UF}_2: \{ R_2, R_3, R_4 \} & \boxed{R_5} \parallel & \boxed{R_1} \\ \text{UF}_3: \boxed{R_8} & \{ R_2, R_3 \} \parallel & \boxed{R_6} \end{array}$$

Fig. 4.6 – Alocação de Registradores para o Circuito DiffEq no Passo 1

A Lista de Registradores Candidatos é atualizada, mas coincidentemente permanece inalterada, porque embora R_1 tenha sido “fixado” na saída de UF_2 , e consequentemente retirado dos candidatos a TPG para UF_2 , ele ainda é um candidato a TPG para UF_1 .

$$\text{Lista de Registradores Candidatos} = \{ R_1, R_2, R_3, R_4, R_6 \}$$

As Tabelas de Entrada e Saída também são atualizadas e como a cada passo um registrador é fixado numa posição, as Tabelas de Entrada e Saída vão progressivamente ficando vazias. Depois de encontrado todos os SAs para cada uma das UFs, inicia-se o processo de escolha dos TPG baseado no ganho dos Registradores. A Fig. 4.7 mostra a configuração final para o circuito DiffEq.

Fig. 4.7 – Alocação Final de Registradores para o Circuito DiffEq

Com os Registradores de Teste definidos, resta agora determinar a combinação de Regras de Teste (90 ou 150) que apresente o menor custo. Como no arquivo B_Atrib consta que os tipos de UF para o circuito DiffEq são

```
fca 1 fu 1
fca 2 fu 2
fca 3 fu 1
```

o programa BIST_S atribuiu as regras de teste 90 para as UFs 1 e 3, e 150 para a UF₂, pois estas são as regras preferenciais encontradas na Tab. 3.2, para uma Cobertura de Falha de 0.9 (especificada na linha de comando da tela durante a execução do programa). Assim, a Solução Preferencial Inicial para UF₁, UF₂ e UF₃ passa a ser TPG90, TPG150 e TPG90, com um Custo¹⁵ de *Hardware* Inicial de 19.12, baseado na Tab. 3.1. Para tentar baixar este custo, cada uma das UFs terá sua Regra de Teste mudada isoladamente no conjunto.

¹⁵ Os valores relativos dos Custos de Hardware são os mais importante aqui, embora eles não se baseiem no impacto global da área do CI causado pelas mudanças. Estimativas desta natureza exigiriam tempo, pessoas e ferramentas de *software* não disponíveis no LME-EPUSP no momento.

Solução 1: TPG90 TPG150 TPG90 (Preferencial!)

```
#TPG90: reg 7 reg 8
#TPG150:
#TPG90/150: reg 2
#SA90: reg 6
#SA/TPG90: reg 1
#SA/TPG150: reg 5
#SA/TPG90/150:
```

Custo de Hardware: 19.12
Custo de Tempo: 0.0198

Solução 2: TPG150 TPG150 TPG90

```
#TPG90: reg 8
#TPG150: reg 7
#TPG90/150: reg 2
#SA90: reg 6
#SA/TPG90:
#SA/TPG150: reg 1 reg 5
#SA/TPG90/150:
```

Custo de Hardware: 19.92
Custo de Tempo: 0.0244

Solução 3: TPG90 TPG90 TPG90

```
#TPG90: reg 2 reg 7 reg 8
#TPG150:
#TPG90/150:
#SA90: reg 6
#SA/TPG90: reg 1 reg 5
#SA/TPG150:
#SA/TPG90/150:
```

Custo de Hardware: 18.08
Custo de Tempo: 0.0213

Solução 4: TPG90 TPG150 TPG150

```
#TPG90: reg 7
#TPG150: reg 2 reg 8
#TPG90/150:
#SA90: reg 6
#SA/TPG90: reg 1
#SA/TPG150: reg 5
#SA/TPG90/150:
```

Custo de Hardware: 19.28
Custo de Tempo: 0.0244

Como a Solução 3 apresentou um Custo de *Hardware* menor, sem ultrapassar o
Custo de Tempo de 40% da Solução Preferencial Inicial (também especificado na Linha

de Comando da Tela durante a execução), ela passa a ser a nova Solução Preferencial. Na Solução 3, o reg 2 precisa ser configurado apenas como TPG90 e não mais em TPG 90 e 150, assim como o reg 5 agora assume a regra 90 tanto quando funciona como TPG como SA.

Solução 5: TPG90 TPG90 TPG90 (Preferencial!)

```
#TPG90: reg 2 reg 7 reg 8
#TPG150:
#TPG90/150:
#SA90: reg 6
#SA/TPG90: reg 1 reg 5
#SA/TPG150:
#SA/TPG90/150:

Custo de Hardware: 18.08
Custo de Tempo: 0.0213
```

Solução 6: TPG150 TPG90 TPG90

```
#TPG90: reg 2 reg 8
#TPG150: reg 7
#TPG90/150:
#SA90: reg 6
#SA/TPG90: reg 5
#SA/TPG150: reg 1
#SA/TPG90/150:

Custo de Hardware: 18.88
Custo de Tempo: 0.0259
```

Solução 7: TPG90 TPG150 TPG90

```
#TPG90: reg 7 reg 8
#TPG150:
#TPG90/150: reg 2
#SA90: reg 6
#SA/TPG90: reg 1
#SA/TPG150: reg 5
#SA/TPG90/150:

Custo de Hardware: 19.12
Custo de Tempo: 0.0198
```

Solução 8: TPG90 TPG90 TPG150

```
#TPG90: reg 7
#TPG150: reg 8
#TPG90/150: reg 2
#SA90: reg 6
#SA/TPG90: reg 1 reg 5
#SA/TPG150:
#SA/TPG90/150:

Custo de Hardware: 19.12
Custo de Tempo: 0.0259
```

Neste novo ciclo de mudança de regras, com Custo de *Hardware* baseado na Tab. 3.1, não apareceu nenhuma solução com custo menor que a Solução 5. Portanto, ela é a solução final.

Uma forma mais precisa de avaliar o impacto das mudanças causadas pela Estrutura de Teste sobre o Custo de *Hardware* Absoluto, em que apenas os Registradores de Teste são considerados, poderia ser calculada da seguinte forma para a Solução 5:

$$\begin{aligned}\text{Custo de Hardware Absoluto} &= 3*\text{TPG90} + 1*\text{SA90} + 2*\text{SA/TPG90} \\ &= 3*0.40 + 1*.8 + 2*1.04 = 4.08\end{aligned}$$

Este Custo de *Hardware* Absoluto reflete o fato de que 6 registradores vão ser utilizados durante o Autoteste, e isso implica uma alteração cujo custo é 4.08. (O acréscimo percentual no Custo de Hardware refere-se apenas aos registradores e não a todos os elementos de hardware do circuito).

A memória e o Tempo de CPU usados em uma estação Sun Sparc 10 estão especificados na Fig. 4.8.

Memória Usada:

STD_MEMORY_MGR

size	used	free	blocks	bytes
12	184	497	1	8172
20	54	354	1	8160
32	1	253	1	8128
36	0	226	1	8136
40	73	130	1	8120
44	106	79	1	8140
52	58	98	1	8112
72	66	46	1	8064
96	130	38	2	16128
156	1	50	1	7956
time: 24.75 sec			space: 87.03 kb	

Fig. 4.8 – Tempo de CPU e Memória Usada para o Circuito DiffEq

A busca de uma solução paralela para este mesmo circuito não apresenta grande interesse, pois como mostra o Grafo de Compatibilidade da Fig. 4.9, não há qualquer compatibilidade entre as 3 UFs, obrigando uma solução série.

fca 2

fca 1

fca 3

Fig. 4.9 – Grafo de Compatibilidade do Circuito DiffEq

A ausência de interconexões entre as 3 UFs significa que não há Caminho de Teste independentes entre elas, i.e., elas compartilham os registradores de saída.

4.2 Resultados com o Algoritmo Paralelo

Para analisar os algoritmos na Geração de Planos de Teste Concorrente de UFs, o Circuito Data5UF, da Fig. 3.13, foi adotado com suas UFs na condição de paralelismo 1, 2 e 3 (máximo possível). Para o paralelismo 1, i.e., a Solução Série, duas alternativas foram implementadas. Na primeira, obtida com o programa BIST_S, privilegiou-se o Custo de *Hardware* em detrimento do Custo de Tempo de Teste. Na segunda, obtida com o programa BIST_P, buscou-se uma alternativa em que o Tempo de Teste fosse o menor possível. Ambas estão mostradas na Fig. 4.10 e Tab.4.1.

Tabela 4.1: Plano de Otimização do Teste Série para o Circuito da Fig. 3.13

Solução com Custo Mínimo de <i>Hardware</i>	Solução com Custo Mínimo de Tempo
Custo <i>Hardware</i> Absoluto = 4.48	Custo <i>Hardware</i> Absoluto = 6.59
Custo de Tempo = 6900	Custo de Tempo = 6000

Fig. 4.10 – Diferentes soluções para o Teste Série do Circuito Data5UF

A solução encontrada pelo BIST_S foi a melhor entre outras 30, enquanto que o BIST_P pesquisou 1152 soluções, com um custo computacional de tempo e espaço 3 vezes maior. Este comportamento, a exemplo de outros testes não apresentados aqui, evidencia que o BIST_P não apresenta um bom desempenho para a Solução Série, quando comparado com o desempenho do BIST_S.

Com o paralelismo 2, na melhor solução entre 48 encontradas, o Tempo de Teste foi reduzido em 48%, a um Custo de *Hardware* 41% maior (6.32) em relação à solução do BIST_S, já que agora mais registradores precisam ser reconfigurados em CAs, como mostra a Fig. 4.11. A Matriz de Estados de Teste apresenta a ordem em que as UFs devem ser testadas, num total de 3600 vetores, e a Tabela de Alocação de Registradores de Teste indica quais registradores serão transformados para teste.

Fig. 4.11 – Solução de Teste para o Circuito Data5UF com Paralelismo 2

Quando a opção de paralelismo 3 é implementada, entre 46 alternativas encontradas, o Custo de Tempo cai para 57% do Teste Série (veja Fig. 4.12), enquanto que o Custo de *Hardware* Absoluto é 41% maior (6.32) que o Teste Série.

Fig. 4.12 – Solução de Teste para o Circuito Data5UF com Paralelismo 3

Neste exemplo, o Custo de *Hardware* para a solução com Paralelismo 3 foi o mesmo que a solução com Paralelismo 2, mostrando que em muitas situações é possível otimizar o Custo de Tempo com pequena ou nenhuma Penalidade ao Custo de *Hardware* [Gon et al. 00].

Outras soluções podem ser encontradas pelo BIST_P, dependendo das alternativas escolhidas durante a execução do programa, mas de um modo geral, o que se observa é que o BIST_P se norteia pela busca de uma Solução Série ou Paralela com o menor Tempo de Teste, enquanto que o BIST_S se norteia pela busca de uma Solução Série com o menor Custo de *Hardware*.

O BIST_P foi igualmente aplicado a duas variantes do circuito Tseng, sintetizadas pelo Sistema de HLS MACH. Na primeira arquitetura não foram levadas em conta considerações com a Testabilidade (Tsengn), enquanto que na segunda (Tsengt), a Testabilidade foi considerada desde as primeiras fases da HLS, resultando num circuito com Custo de *Hardware* um pouco maior, porém com a Controlabilidade e Observabilidade aumentadas[Ham et al 94].

Embora o número de UFs e registradores tenha permanecido inalterado quando a Testabilidade foi considerada como uma de suas restrições, o mesmo não ocorreu com os muxes. Como mostram as Fig. 4.13 e Fig. 4.16, a arquitetura resultante passou de 5 muxes com 2 entradas, 1 mux com 3 entradas e 1 mux com 4 entradas, para 7 muxes com 2 entradas, 2 muxes com 3 entradas e 1 mux com 4 entradas.

A Fig. 4.13 mostra uma possível implementação do circuito Tsengn, com 4 UFs, sintetizada pelo Sistema de HLS MACH.

A numeração das UFs (1, 2, 5 e 6) foi mantida da forma como apresentada no pacote original do MACH.

Fig. 4.13 – Circuito Tsengn

Embora o Grafo de Compatibilidade do Circuito Tsengn (Fig. 4.14) mostre a possibilidade de até três UFs serem testadas simultaneamente, o BIST_P não encontra registradores para este Paralelismo e a Compatibilidade Máxima Efetiva é fixada em 2.

Fig. 4.14 – Grafo de Compatibilidade do Circuito Tsengn

As UFs 5 e 6 não são compatíveis porque R_2 é a única possibilidade de TPG para UF_5 e ao mesmo tempo é a única saída da UF_6 . As UFs 1, 2 e 5 não podem ser testadas simultaneamente porque R_2 e R_5 precisam obrigatoriamente funcionar como TPGs da

UF₅, reduzindo, assim, os candidatos a SA das 3 UFs para R₃, R₄ e R₁. Mas, ou R₃ ou R₄ precisa funcionar como TPG da UF₂, eliminando a possibilidade de Paralelismo 3.

A Fig. 4.15 mostra que o Paralelismo 2 inicial de UF₂ e UF₅ não pode ser mantido nos Estados de Teste subseqüentes, pois não há compatibilidade entre as UFs 1 e 6. No resultado final, o Custo de *Hardware* é elevado de 4.56 (Solução Série) para 5.79, porém o Tempo de Teste é reduzido de 5400 para 3600.

Fig. 4.15 – Solução de Teste Paralelo para o Circuito Tsengn

Para o Circuito Tsengt, da Fig. 4.16, a Compatibilidade Máxima Efetiva continua sendo 2, mas neste caso o Paralelismo 2 não produz uma má solução, como se vê na Fig. 4.17, porque o paralelismo 3 poderia diminuir muito pouco o Tempo de Teste. Se a UF₂ fosse compatível com a UF₁ e UF₅, a redução no Tempo de Teste seria nula e se a UF₆ fosse testada junto com a UF₁ e UF₅ a redução seria muito pequena.

Quando o Custo de *Hardware* do Tsengt é comparado com o do Tsengn, verifica-se que houve uma redução de 5.79 para 3.68, que o Tempo de Teste caiu de 3600 para 2800 e que em relação à Solução Série há uma redução de 48% no Tempo de Teste (de 5400 para 2800), com um aumento de 28% no Custo de *Hardware* (2.88 para

a Solução Série), mostrando que, em geral, quando a Testabilidade é considerada durante a HLS, os resultados são significativamente melhores.

Fig. 4.16 – Circuito Tsengt

A Fig. 4.17 mostra a Matriz de Estado de Teste do Circuito Tsengt, com um alto grau de aproveitamento de espaço, pois apenas no Estado 2 a incompatibilidade entre UFs se manifesta.

Fig. 4.17 – Solução de Teste Paralelo para o Circuito Tsengt

Outros exemplos gerados pelo Sistema HLS MACH foram submetidos ao BIST_P e ao BIST_S, cujos resultados estão resumidos na Tab. 4.2. Na primeira coluna estão as identificações dos *benchmarks*, na segunda o número de UFs do circuito e na

terceira o número total de registradores para o funcionamento normal. Pela quarta coluna é possível saber quantas UFs serão testadas pela Regra Preferencial, que tem um custo menor, e quantas usam a Regra Não Preferencial, que apesar de ter um custo maior, acabam gerando um custo global menor, devido ao compartilhamento de TPGs.

Tabela 4.2: Exemplos e Regras

Exemplo	# UFs	# Registradores	Otimização do Plano de Teste Paralelo
			Regra / # UFs
Data5UF	5	9	90 / 3 – 90* / 1 – 150 / 1
Tsengt	4	5	90 / 2 – 90* / 2
DiffEq	4	10	90 / 2 – 90* / 2
ArFiltt	3	10	90 / 1 – 90* / 1 – 150 / 1

* Indica a Regra Não Preferencial

A Tab. 4.3 compara alguns dos resultados produzidos pelo BIST_S e BIST_P, mostrando que a redução no Custo de Tempo, em geral, é obtida com um aumento no Custo de *Hardware*. A coluna dedicada ao Custo de Tempo indica o número de Vetores de Teste necessário para satisfazer a Cobertura de Falhas exigida. A coluna do Custo de *Hardware* considera apenas os registradores que terão de ser reconfigurados em TPGs e/ou SAs, enquanto que a coluna ao lado especifica quantos desses registradores serão TPGs, SAs ou TPG/SAs.

Tabela 4.3: Resultados (Área e Tempo) para a Otimização Série e Paralela

Exemplo	Otimização do Plano de Teste Série			Otimização do Plano de Teste Paralelo		
	Custo de Tempo	Custo de <i>Hardware</i>	#TPG, #SA, #TPG/SA	Custo de Tempo	Custo de <i>Hardware</i>	#TPG, #SA, #TPG/SA
Data5UF	6000	6.59	4;1;2	3000	6.32	4;1;3
Tsengt	5400	2.88	2;0;2	2800	3.68	2;1;2
DiffEq	5300	4.08	4;0;2	1600	7.20	7;3;0
ArFiltt	4000	6.00	2;3;0	2200	6.00	3;3;0

4.3 Resumo

Neste capítulo foram apresentados alguns resultados experimentais obtidos com a aplicação do BIST_S e BIST_P a vários *benchmarks*, depois de feita uma adaptação simples ao circuito (quando necessária) para que a Estrutura de Teste BIST pudesse ser introduzida.

No BIST_S, a escolha do conjunto mínimo de Registradores de Teste é baseada no Ganho dos Registradores e, em seguida, são tentadas algumas mudanças nas Regras Preferenciais com vistas a um Custo de *Hardware* o menor possível.

No BIST_P, a partir de um Grafo de Compatibilidade é determinado o Grau de Paralelismo entre as UFs do circuito e construída uma Matriz de Estado de Teste, para que o Tempo de Teste seja minimizado, a um Custo de *Hardware* otimizado.

Finalmente foram feitas algumas comparações entre os resultados produzidos pelo BIST_S e BIST_P, mostrando que uma redução no Tempo de Teste geralmente implica um aumento no Custo de *Hardware*, mas que em alguns casos este aumento pode ser pequeno.

5 CONCLUSÃO

Nesta dissertação procurou-se argumentar que a complexidade atual dos CIs tem exigido que a Testabilidade seja considerada como uma restrição de projeto, pois se o Teste do CI não for levado em conta desde as primeiras etapas do projeto, os Custos de *Hardware*, de Tempo de Projeto e de Tempo de Teste podem torná-lo inviável.

As soluções iniciais quase sempre baseavam-se em Equipamentos Automáticos de Teste, que geravam e aplicavam os Vetores de Teste no CI para, em seguida, comparar as respostas obtidas com um padrão esperado. Com o desenvolvimento da técnica BIST, em que parte da Estrutura de Teste é incorporada ao próprio CI, de forma a permitir que as tarefas de geração, aplicação e comparação de resultados fossem feitas dentro do CI, as Taxas de Cobertura de Falhas se elevaram, novos tipos de falha passaram a ser detectadas e o Tempo de Teste reduziu-se sobremaneira.

5.1 Considerações Finais

Na Síntese Estrutural para Teste, a inserção da Estrutura de Teste ocorre quando o Sistema de HLS já definiu a arquitetura do CI, no nível RTL, com ou sem considerações sobre a Testabilidade. No modelo de Síntese Estrutural para Teste proposto por [Ama et al. 97a], objeto de estudo desta dissertação, utilizam-se bibliotecas previamente compiladas para cada tipo de UF, contendo informações tais como Número de Vetores de Teste, Taxa de Cobertura de Falhas, Custo de *Hardware* etc., tornando parte dos resultados da Simulação de Teste previsíveis desde sua fase inicial.

Nesse contexto, duas formas de realização de Autoteste foram apresentadas: a Solução Série e a Paralela. Os respectivos algoritmos foram implementados como os programas BIST_S e BIST_P e aplicados a diversos exemplos de circuitos. Na Solução Série, busca-se um conjunto mínimo de Registradores de Teste com o Custo de

Hardware otimizado, baseado no Ganho dos Registradores, que, essencialmente, reflete o grau de compartilhamento de um registrador entre várias UFs. Na Solução Paralela, o conjunto mínimo de Registradores de Teste está associado a um Tempo de Teste otimizado, resultado da construção de uma Matriz de Teste com as diferentes Sessões de Teste de todas UFs.

O Algoritmo Paralelo oferece a possibilidade de escolha do Grau de Paralelismo 1 até um valor máximo admitido pela arquitetura do CI ou pelo consumo de potência, mas resultados com uma série de *benchmarks* indicaram que ele não é muito eficiente quando o usuário escolhe um valor bem abaixo do máximo permitido pela arquitetura, pois um número grande de soluções alternativas precisa ser analisado, tornando o custo computacional elevado. Para estes casos, o Algoritmo Série geralmente chega a uma Solução Heurística mais rapidamente, com um Custo de *Hardware* menor que o encontrado pelo Algoritmo Paralelo, porém com um Tempo de Teste maior.

A aplicação do Algoritmo Paralelo a vários circuitos mostrou que embora a determinação do nível de paralelismo seja condicionado pelo *trade-off* entre a Sobreárea de *Hardware* aceitável e a duração do Tempo de Teste, esta relação não se comporta linearmente, podendo-se obter melhoras significativas no Tempo de Teste com pequeno Custo Adicional de *Hardware*.

5.2 Futuros Desenvolvimentos

A qualidade dos resultados obtidos com o BIST_S e BIST_P depende em grande parte das respostas fornecidas pelo usuário durante o tempo de execução. Através de uma análise sistemática dos resultados de várias simulações seria possível estabelecer maneiras de automatizar a escolha dos melhores resultados.

Nem sempre a melhor solução obtida com o BIST_P é aquela com o Paralelismo Máximo e a simulação de todas as possibilidades é algo pouco viável até mesmo para circuitos com 10 ou 12 UFs. Nestas situações, a intuição e a experiência do projetista tornam-se muito importantes, mas isso exige que se faça uma análise minuciosa dos resultados preliminares. Como é razoável supor que o projetista esteja sempre interessado em saber qual é o melhor grau de paralelismo, seria interessante o desenvolvimento de uma rotina que apontasse o grau de paralelismo com maior possibilidade de fornecer a solução desejada.

Como já se explicou, o desenvolvimento do Controlador de Teste foge ao escopo desta dissertação, mas como argumenta acertadamente [Alv et al. 96], esta tarefa não é de grande complexidade, podendo ser realizada, por exemplo, através de uma Máquina de Estados Finita.

Embora o BIST_S e o BIST_P tenham sido implementados separadamente, os resultados obtidos sugerem a conveniência de fundir os dois algoritmos num Algoritmo Híbrido, de tal forma que, para um Grau de Paralelismo baixo, prevaleça a lógica do Algoritmo Série, deixando o Algoritmo Paralelo para os casos restantes. Com isso, as relações Área X Tempo de Teste X Cobertura de Falha estariam sendo exploradas de maneira mais eficiente pelo projetista.

6 REFERÊNCIAS BIBLIOGRÁFICAS

- [Abr et al. 90] ABRAMOVICI, M.; BREUER, M. A.; FRIEDMAN, A., D. **Digital systems testing and testable design**. Revising printing. Piscataway, NJ, IEEE PRESS Marketing, 1990.
- [Aho et al. 83] AHO, A. V.; HOPCROFT, J. E.; ULLMAN, J. D. **Data structures and algorithms**. Reading, MA, Addison-Wesley Publishing Company, 1983.
- [Alv et al. 96] ALVES, V. C.; ANTUNES, A. R.; MARZOUKI, M. Requirements and general framework for an efficient synthesis for testability methodology. **Proceedings of the XI Conference of The Brazilian Microelectronics Society**, p.135-140, 1996.
- [Ama et al. 97a] AMAZONAS, J. R. DE A.; STRUM, M.; NETO, J. V.; CHAU, W. J.; ROUZEYRE, B.; FOTTES, M-L. Built-in self-test high level synthesis. **XII Conference of the Brazilian Microelectronics Society (CD-ROM)**, p.31, 1997.
- [Ama et al. 97b] AMAZONAS, J. R. DE A.; STRUM, M.; NETO, J. V.; CHAU, W. J.; ROUZEYRE, B.; FOTTES, M-L. Data path functional units BIST plan optimization: a study-case. **Proceedings of the SBCCI97**, p.185-194, 1997.
- [Ama et al. 98] AMAZONAS, J. R. DE A.; STRUM, M.; CHAU, W. J. Exploring concurrency in data path functional units BIST plan optimization: a study-case. **Proceedings of the SBCCI98**, p.244-247, 1998.
- [Avra 91] AVRA, L. N. Allocation and assignment in high-level synthesis for self-testable data paths. **Proceedings of the International Test Conference 1991**, p.463-472, 1991.
- [Bha e Jha 94] BHATIA, S.; JHA, N. K. Behavioral synthesis for hierarchical testability of controller/data path circuits with conditional branches. **1994 IEEE International Conference on Computer Design: VLSI in Computers & Processors**, p. 91-96 , 1994.
- [Car et al. 98a] CARDOSO, P. S.; STRUM, M.; AMAZONAS, J. R.; WANG, J. C. Quasi-uniform cellular automata for BIST application. **Proceedings of the IV Workshop Iberchip**, p.392-401, 1998.
- [Car et al. 98b] CARDOSO, P. S.; STRUM, M.; AMAZONAS, J. R.; WANG, J. C. A methodology for minimum area cellular automata generation. **Proceedings of the XIII SBMicro**, p.51-59, 1998.
- [Car e Papa 95] CARLETTA, J.; PAPACHRISTOU, C. A method for testability analysis and BIST insertion at the RTL. **Proceedings of The European Design and Test Conference**, p.600, 1995.

- [Chen e Agra 90] CHEN, C.-H.; AGRAWAL, V.D. A partial scan method for sequential circuits with feedback. **IEEE Transaction Computer**, v. 39, p. 544-548, 1990.
- [Dam et al. 90] DAMIANI, M.; OLIVIO, P.; FAVALLI, M.; ERCOLANI, S. Aliasing in signature analyzer testing with multiple input shift registers. **IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems**, v. 9, n.12, p.1344-1353, 1990.
- [Dei e Dei 94] DEITEL, H. M.; DEITEL, P. J. **C++ how to program**. Englewood Cliffs, NJ, Prentice-Hall, 1994.
- [De Mi 94] MICHELI, G. **Synthesis and optimization of digital circuits**. New York, NY, McGraw-Hill, 1994.
- [Eps 93] EPSTEIN, G. **Multiple-valued logic design: an introduction**. Bristol, England, IOP Publishing Ltd., 1993.
- [Flot et al. 95] FLOTTES, M. L.; HAMMABAD, D.; ROUZEYRE, B. High-level synthesis for easy testability. **Proceedings of The European Design and Test Conference**, p.198-206, 1995.
- [Fuj 86] FUJIWARA, H. **Logic testing and design for testability**. Second printing. Cambridge, MA, The MIT Press, 1986.
- [Gaj et al. 92] GAJSKI, D. D.; DUTT, N. D.; WU, A. C. H.; LIN, S. Y. L. **High-level synthesis: chip and system design**. Norwell, MA, Kluwer Academic Publishers, 1992.
- [Geb e Elm 92] GEBOTYS, C.; ELMASRY, M. I. **Optimal VLSI architectural synthesis: area, performance and testability**. Norwell, MA, Kluwer Academic Publishers, 1992.
- [Gon et al. 00] GONZALEZ, J. A. Q.; AMAZONAS, J. R. A.; STRUM, M.; CHAU, W. J. Self test built-in plan for data-path functional units. **Digest of Papers of the 1st IEEE Latin-American Test Workshop**, p.9-14, 2000.
- [Gray 94] GRAY, N. A. B. **Programming with class – a practical introduction to object-oriented programming with C++**. New York, NY, John Wiley & Sons, 1994.
- [Ham et al 94] HAMMAD, D.; FLOTTES, M. L.; ROUZEYRE, B.; An analysis based approach for easy testable data-paths synthesis. **Report LIRMM 1994**, Université de Montpellier, France, 1994.
- [Hor 89a] HORTENSIUS, P.D. et al. Cellular automata-based pseudorandom number generators for built-in self-test. **IEEE Transactions on Computer-Aided Design**, v. 8, n.8, p.842-859, 1989.

- [Hor et al 89b] HORTENSIUS, P.D.; MCLEOD, R.D.; CARD, H. C. Parallel random number generation for VLSI systems using CA. **IEEE Transactions on Computers**, v. 38, n.10, p.1466-1473, 1989.
- [Joh et al. 95] JOHNSONBAUGH, R; KALIN, M. **Object-oriented programming in C++**. Englewood Cliffs, NJ, Prentice Hall, 1995.
- [Kön et al. 84] KÖNEMANN, B. et al. Built-in logic block observation techniques. In **Selected Reprints on Logic Design For Testability**, Compiled by CONSTANTIN C. TIMOC, IEEE Computer Society Press, p.262-266, 1984.
- [Lee 97] LEE, M. T.-C. **High-level test synthesis of digital VLSI circuits**. Norwood, MA, Artech House, 1997.
- [McClu 86] MCCLUSKEY, E. J. **Logic design principles with emphasis on testable semicustom circuits**. Englewood Cliffs, NJ, Prentice-Hall, 1986.
- [Mehl S/D] MEHLHORN, K. et al. **The LEDA user manual. Version 3.5.1**. Saarbrücken, Germany, S/D. (<http://www.mpi-sb.mpg.de/LEDA/leda.html>)
- [Mur e Hay 90] MURRAY, B. T.; HAYES, J. P. Hierarchical test generation using precomputed test for modules. **IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems**, v. 9, n.6, p.594-603, 1990.
- [Nag 51] NAGELL, T. **Introduction to number theory**. Wiley, New York, p. 160-164, 1951.
- [Nor e McClu 96] NORWOOD, R. B.; MCCLUSKEY, E. J. Orthogonal scan: low-overhead scan for data paths. **Proceeding of the International Test Conference (ITC 96)**, p.659-668, 1996.
- [Nor 97] NORWOOD, R. B. **Synthesis-for-scan: reducing scan overhead with high-level synthesis**. Stanford, USA, 1997 . Tese (Doutorado) – Departments of Electrical Engineering and Computer Science, Stanford University.
- [Ohl 96] OHLETZ, M. J. Fault modelling and simulation for the test of integrated analog and mixed-signal circuits. In: MACHADO, G. A. S. **Low-power HF microelectronics – a unified approach**. Lavenham Press Ltd., Lavenham, UK. Cap. 13, p. 475-510, 1996.
- [Ora e Har 97] ORAILOGLU, A.; HARRIS, I. G. Microarchitectural synthesis for rapid BIST testing. **IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems**, v.16, n.6, p.573-586, 1997.
- [Pom e Red 98] POMERANZ, I.; REDDY, S. M. On methods to match a test pattern generator to a circuit-under-test. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v.6, n.3, p.4332-4343, 1998.

- [Pyr et al. 98] PYRON, C.; PRADO, J.; GOLAB, J. Test strategy for the PowerPC 750 microprocessador. **IEEE Design & Test of Computers**, v.15, n.3, p.90-97, 1998.
- [Raj 92] RAJSUMAN, R. **Digital hardware testing: transistor level fault modeling and testing**. Norwood, MA, Artech House, 1992.
- [Raj 95] RAJSUMAN, R. **Iddq testing for CMOS VLSI**. Norwood, MA, Artech House, 1995.
- [Rum et al. 91] RUMBAUGH, J. et al. **Object-oriented modeling and design**. Englewood Cliffs, NJ, Prentice-Hall, 1991.
- [Rum 94] RUMBAUGH, J. **OMT - insights**. SIGS Books&Multimedia, NY, 1994.
- [Rum et al. 97] RUMBAUGH, J. et al. **Modelagem e projetos baseados em objetos**. Campus, RJ, Editora Campus, 1997.
- [Ser et al. 90] SERRA, M.; SLATER, T.; MUZIO, J. C.; MILLER, D. M. The analysis of one-dimensional linear cellular automata and their aliasing properties. **IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems**, v.9, n.7, p.767-778, 1990.
- [Ser 94] SERRA, M. Digital IC testing: an introduction. University of Victoria, Victoria, BC, Canada.
<http://csr.csc.uvic.ca/home/mserra/testing/draft.html>, 1994.
- [Str e Wun 98] STROELE, A. P.; WUNDERLICH, H.-J. Hardware-optimal test register insertion. **IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems**, v.17, n.6, p.573-586, 1998.
- [Stru et al. 99] STRUM, M.; CHAU, W. J.; NETO, J. V. V. Síntese de alto nível: plana e hierárquica. In **Sistemas Digitales – Elementos para un Diseño a Alto Nivel**, compilado por A. G. Roza, Bogotá, Colombia, Ediciones Uniandes, p. 181-279, 1999.
- [Tsu 87] TSUI, F. F. **LSI/VLSI testability design**. New York, NY, McGraw-Hill, 1987.
- [Wil et al. 88] WILLIAMS, T. W. et al. Bounds an analysis of aliasing errors in linear feedback shift registers. **IEEE Transactions on Computer Aided Design**, v. 17, n. 1, p.75-83, 1988.

APÊNDICE 1

GLOSSÁRIO

- ALAP** – As Late As Possible: O Mais Tarde Possível.
- ALU** – Arithmetic Logic Unit: Unidade Lógica Aritmética.
- ASAP** – As Soon As Possible: O Mais Cedo Possível.
- ASIC** – Application Specific Integrated Circuit: Circuito Integrado de Aplicação Específica.
- ATE** – Automatic Test Equipment: Equipamento Automático de Teste.
- ATPG** – Automatic Test Pattern Generation: Gerador Automático de Padrões de Teste.
- BILBO** – Built-In Logic Block Observer: Bloco Lógico Observador Interno.
- BIST** – Built-In Self-Test: Autoteste Incorporado.
- CA** - Cellular Automata: Autômatos Celulares.
- CAD** – Computer Aided Design: Projeto Auxiliado por Computador.
- CBILBO** - Concurrent Built-in Logic Block Observer: Bloco Lógico Observador Interno Simultâneo.
- CDFG** – Control/Data Flow Graphs: Grafo de Fluxo de Dados/Controle.
- DFT** – Design For Testability – Projeto Voltado à Testabilidade.
- DPCG** - Data Path Circuit Graph: Grafo do Circuito do Caminho de Dados.
- FF** – Flip-Flop: Flip-Flop.
- HLS** - High Level Synthesis: Síntese de Alto Nível.
- LFSR** – Linear Feedback Shift Register: Registrador Linear de Deslocamento Realimentado.
- MISR** – Multiple Input Signature Register: Registrador de Assinatura de Múltiplas Entradas.
- RTL** – Register Transfer Level: Transferência em Nível de Registrador.
- SA** – Signature Analyzer: Analisador de Assinatura.
- SCC** – Strongly Connected Component: Componente Fortemente Conectado.
- TPG** – Test Pattern Generator: Gerador de Padrão de Teste.
- UF** – Unidade Funcional.
- VLSI** – Very Large Scale Integration: Integração de Escala Muito Alta.

APÊNDICE 2

3.6.2 Resultados Obtidos pela Aplicação da Metodologia Baseada em Objetos

3.6.2.1 Análise

A leitura dos *papers* mencionados no item 3.6.1 – Análise de Requisitos - definiu os requisitos do problema, ressaltando o que deve ser feito, independentemente de como será feito. Por isso, nesta fase, ainda não houve decisões de implementação e, portanto, não se produziu uma proposta de solução ao problema. Mas ficou claro, por exemplo, que deveria haver consultas do sistema a bibliotecas com dados pré-compilados referentes à cada tipo de UF, que o usuário deveria informar o formato da codificação da arquitetura inicial do CI, que o sistema deveria gerar alguns resultados intermediários para controle da consistência das transformações nos dados e que finalmente o usuário receberia um Plano de Teste, pois este era o objetivo dos cálculos.

A decisão de usar a Teoria dos Grafos sinalizou a necessidade de se criar uma classe no Domínio da Aplicação que representasse um Circuito, e as leituras subsequentes dos *papers* com esta classe central já definida sugeriu a criação de outras classes interligadas por associações, que se reduziram a apenas quatro classes durante o refinamento do Modelo de Objetos.

3.6.2.1.1 Modelo de Objetos

A Fig. 3.21 mostra o Modelo de Objetos do BIST_S com sua representação do mundo real através de uma Estrutura Estática dos Objetos. Este modelo não é exatamente o modelo inicial da Análise; na realidade, ele já contém os refinamentos da fase de Projeto de Objetos, mas para evitar a repetição resultante da sobreposição de partes significativas destas duas fases, optou-se por apresentar apenas esta versão síntese dos resultados.

Fig. 3.21 - Modelo de Objetos do BIST_S com Atributos e Herança

O Modelo de Objetos do BIST_P, mostrado na Fig. 3.22, contém as mesmas classes do BIST_S, porém com algumas funções a mais.

A função `BFS_MuxReg` realiza uma procura do tipo *Breadth-First Search* no grafo do circuito para detectar não apenas possíveis incompatibilidades entre registradores, mas também entre muxes, i.e., um mux não pode ser solicitado por dois caminhos distintos ao mesmo tempo.

`Num_TestesParalelo` detecta qual o número máximo de UFs que podem ser testadas simultaneamente, enquanto que `Matriz_EstadoTeste` é a função responsável

pela criação da Matriz de Estado de Teste, que, ao final das contas, faz o *Scheduling* de Teste das UFs.

Fig. 3.22 - Modelo de Objetos do BIST_P com Atributos e Herança

3.6.2.1.2 Modelo Dinâmico

A Fig. 3.23 mostra o Diagrama de Estados do BIST_S, ressaltando o comportamento das partes do sistema dependentes do tempo, i.e., a seqüência de interações durante o processamento dos dados. Por este diagrama é possível ver os principais eventos externos ligando o sistema e o mundo exterior.

Fig. 3.23 – Diagrama de Estados para a Classe BIST_S

Como no teste paralelo várias UFs podem ser testadas simultaneamente, durante a distribuição dos registradores candidatos a SA é necessário levar em conta o mux associado a determinado registrador, pois pode ocorrer casos de um registrador ser candidato a SA mas não existir um caminho para a coleta das saídas da UF. A Fig. 3.24 mostra que há uma operação para encontrar os pares reg-mux.

Fig. 3.24 – Diagrama de Estados para a Classe BIST_P

3.6.2.1.3 Modelo Funcional

Na Fig. 3.25 vê-se o Diagrama de Fluxo de Dados do BIST_S, com os caminhos pelos quais passa o fluxo de dados durante sua transformação, sem no entanto preocupar-se sobre o momento em que se dão estas transformações. A Netlist Inicial, as Bibliotecas e o Plano de Teste são os principais valores de entrada e saída do sistema e devem se harmonizar com os eventos externos do Diagrama de Estados da Fig. 3.23.

Fig. 3.25 Diagrama de Fluxo de Dados para os Processos Otimizar Registradores e Atribuir Regras do BIST_S

Foram incorporados os processos de busca dos pares registrador/mux e de criação da Matriz de Estado de Teste no Diagrama de Fluxo de Dados do BIST_P, mostrado na Fig. 3.26.

Fig. 3.26 Diagrama de Fluxo de Dados para os Processos Otimizar Registradores e Matriz de Estado de Teste do BIST_P

Novamente é apresentado, na Fig. 3.27, um Diagrama de Fluxo de Dados do BIST_S, porém visto de um nível mais elevado

Fig. 3.27 Diagrama de Fluxo de Dados de Nível mais Elevado para o BIST_S

A Fig. 3.28 mostra o processamento dos dados do sistema com o mundo externo, tanto do BIST_S quanto do BIST_P, sem considerar a sequência em que eles ocorrem.

Fig. 3.28 – Valores de Entrada e Saída para o Sistema do BIST_S e do BIST_P

3.6.2.2 Projeto do Sistema e de Objetos

Partindo-se do Modelo de Análise, com suas classes iniciais, verificou-se a conveniência da adoção de Listas Dinâmicas como Objetos Internos para o depósito de dados entre as interações. Como o número de componentes de cada CI varia muito, a escolha de estruturas estáticas dificilmente faria a reserva do espaço de memória necessário ao processamento de dados.

3.6.2.3 Implementação

Os modelos do BIST_S foram implementados em C++ e as bibliotecas LEDA [Mehl S/D] divididos em três arquivos e um makefile, para tornar o sistema mais flexível.

3.6.2.3.1 Arquivos do programa BIST_S:

- **makefile** – contém 7 linhas de comando com instruções para compilar e fazer o *link* dos três arquivos com os códigos fontes. Quando somente um dos programas sofre alterações, o *makefile* evita que todos os outros programas não modificados precisem ser recompilados.
- **BIST_S_P.c** – contém 107 linhas de comando. É o Programa Principal e seleciona a ordem em que as Funções são executadas.
- **BIST_S_C.h** - contém 110 linhas de comando e define todas as Classes.
- **BIST_S_F.c** – contém 1472 linhas de comando com todas as Funções do BIST_S.

O BIST_P se diferencia do BIST_S principalmente pelo programa BIST_P_F.c, já que os outros três arquivos têm poucas diferenças significativas.

3.6.2.3.2 Arquivos do programa BIST_P:

- **makefile** – contém 7 linhas de comando.
- **BIST_P_P.c** – contém 103 linhas de comando.
- **BIST_P_C.h** – contém 127 linhas de comando.
- **BIST_P_F.c** – contém 3884 linhas de comando.