UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

MARCELO DE SOUZA MORAES

STEP: planejamento, geração e seleção de auto-teste *on-line* para processadores embarcados

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Marcelo Soares Lubaszewski
Orientador

Profª. Drª. Érika Fernandes Cota
Co-orientadora

Porto Alegre, maio de 2006.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Moraes, Marcelo de Souza

STEP: planejamento, geração e seleção de auto-teste on-line para processadores embarcados / Marcelo de Souza Moraes. – Porto Alegre: PPGC da UFRGS, 2006.

144f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Instituto de Informática, Porto Alegre, BR-RS, 2006. Orientador: Marcelo Soares Lubaszewski; Co-orientadora: Érika Fernandes Cota.

1. Projeto de auto-teste. 2. Teste on-line. 3. Auto-teste baseado em software. 4. Teste de processadores. 5. Processadores embarcados. 6. Sistemas de tempo-real. I. Lubaszewski, Marcelo Soares. II. Cota, Érika Fernandes. III. Título.

“Feliz o homem que se dedica à sabedoria, que reflete com inteligência, que medita no coração sobre seus caminhos e com a mente penetra em seus segredos.”

– ECLESIÁSTICO 14, 20-21

AGRADECIMENTOS

Agradeço aos meus pais, irmãos e familiares pelo apoio incondicional em todos os momentos e, também, aos meus sempre presentes amigos, que me motivaram nas horas difíceis e comemoraram comigo cada pequena conquista. Agradeço a todos os professores que me auxiliaram neste caminho pela busca de conhecimento, e ao professor e orientador Luba pelo apoio, incentivo e confiança em mim depositada. Em especial, agradeço à professora e co-orientadora Érika, que não apenas me orientou mas também trabalhou comigo, sugeriu, criticou, torceu e esteve sempre presente e disposta, contribuindo enormemente para o sucesso deste trabalho. Finalmente, agradeço a Deus por ter me presenteado com o dom da vida, e pela dádiva de ter chegado até aqui.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

A-VC	Address Visible Component
A-VHSC	Address Visible Hidden Subcomponent
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Set Processor
ATE	Automatic Test Equipment
ATPG	Automatic Test Pattern Generator
BIST	Built-In Self-Test
CA	Cellular Automata
CI	Circuito Integrado
CMOS	Complementary Metal Oxide Semiconductor
CUT	Circuit Under Test
D-VC	Data Visible Component
D-VHSC	Data Visible Hidden Subcomponent
DFT	Design For Testability
DSP	Digital Signal Processor
E/S	Entrada e Saída
EDA	Electronic Design Automation
EDF	Earliest Deadline First
FPGA	Field Programmable Gate Array
GME	Grupo de Microeletrônica
HC	Hidden Component
HBST	Hardware-Based Self-Test
ILP	Instruction Level Parallelism
IRST	Instruction Randomization Self-Test
ISA	Instruction Set Architecture
JVM	Java Virtual Machine

LFSR	Linear Feedback Shift Register
LSE	Laboratório de Sistemas Embarcados
M-VC	Mixed Visible Component
MISR	Multiple-Input Shift Register
NTC	Nontestable Component
PC	Program Counter
PVC	Partially Visible Component
PVHSC	Partially Visible Hidden Subcomponent
RAM	Random Access Memory
RAW	Read After Write
RM	Rate Monotonic
ROM	Read-Only Memory
RTSJ	Real-Time Specification for Java
RTL	Register-Transfer Level
SASHIMI	System As Software and Hardware In Microcontrollers
SBST	Software-Based Self-Test
SEEP	Sistemas Eletrônicos Embarcados baseados em Plataforma
SoC	System-on-Chip
STEP	Self-Test for Embedded Processors
STEP-FJ	Self-Test for Femtojava Embedded Processors
TMR	Triple Modular Redundancy
TPG	Test Pattern Generator
UF	Unidade Funcional
ULA	Unidade Lógica e Aritmética
VC	Visible Component
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration

LISTA DE SÍMBOLOS

AP	índice referente à aplicação
AR	índice referente ao analisador de respostas de teste
b	bit
BT	índice referente à biblioteca de auto-teste
C	número de componentes visados pelo programa de auto-teste
CE	índice referente ao consumo de energia
CR	custo relativo
EB	índice referente ao espaço de busca
EDF	índice referente ao algoritmo de escalonamento EDF
ESC	índice referente ao escalonador
F	conjunto de fatores de custo
FC	função de custo
i	índice genérico para programa de auto-teste
iadd	bytecode Java para a operação de adição
iand	bytecode Java para a operação lógica AND
imul	bytecode Java para a operação de multiplicação
ineg	bytecode Java para a operação de negação
ior	bytecode Java para a operação lógica OR
ishl	bytecode Java para deslocamento aritmético para a esquerda
ishr	bytecode Java para deslocamento aritmético para a direita
isub	bytecode Java para a operação de subtração
iushr	bytecode Java para deslocamento lógico para a direita
ixor	bytecode Java para a operação lógica XOR
j	índice genérico para fator de custo
k	índice genérico para componente do processador
L	tamanho da memória ROM do analisador de respostas
M	número de respostas de teste

N	número de rotinas do programa de auto-teste
NUP	nível de utilização do processador
P	período
PT	período de teste
R	restrição do sistema
RAM	índice referente à memória RAM
RM	índice referente ao algoritmo de escalonamento RM
ROM	índice referente à memória ROM
S	número de abordagens diferentes de rotinas de teste
t	índice genérico para tarefa da aplicação
T	número de tarefas da aplicação
Tam	tamanho
TE	tempo de execução ou índice referente ao tempo de execução
TUP	taxa de utilização do processador
u	custo da união das rotinas de teste
v	custo da rotina de teste
V	custo do programa de auto-teste ou da aplicação
w	peso do fator de custo

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Sistemas embarcados baseados em processadores têm sido largamente aplicados em áreas críticas no que diz respeito à segurança de seres humanos e do meio ambiente. Em tais aplicações, que compreendem desde o controle de freio de carros a missões espaciais, pode ser necessária a execução confiável de todas as funcionalidades do sistema durante longos períodos e em ambientes desconhecidos, hostis ou instáveis. Mesmo em aplicações não críticas, nas quais a confiabilidade do sistema não é um requisito primordial, o usuário final deseja que seu produto apresente comportamento estável e livre de erros. Daí vem a importância de se considerar o auto-teste *on-line* no projeto dos sistemas embarcados atuais.

Entretanto, a crescente complexidade de tais sistemas somada às fortes restrições a que eles estão sujeitos torna o projeto do auto-teste um problema cada vez mais desafiador. Em aplicações de tempo-real a dificuldade é ainda maior, uma vez que, além dos cuidados com as restrições do sistema alvo, deve-se levar em conta o atendimento dos requisitos temporais da aplicação. Entre as técnicas de auto-teste *on-line* atualmente pesquisadas, uma tem se destacado pela eficácia obtida a um baixo custo de projeto e sem grande impacto no atendimento dos requisitos e restrições do sistema: o auto-teste baseado em software (SBST – *Software-Based Self-Test*).

Neste trabalho, é proposta uma metodologia para o projeto e aplicação de auto-teste *on-line* para processadores embarcados, considerando-se também aplicações de tempo-real. Tal metodologia, denominada STEP (*Self-Test for Embedded Processors*), tem como base a técnica SBST e prevê o planejamento, a geração e a seleção de rotinas de teste para o processador alvo. O método proposto garante a execução periódica do auto-teste, com o menor período permitido pela aplicação de tempo-real, e assegura o atendimento de todas as restrições do sistema embarcado. Além disso, a solução fornecida pelo método alcança uma boa qualidade de teste enquanto auxilia a redução de custos do sistema final.

Como estudo de caso, a metodologia proposta é aplicada a diferentes arquiteturas de processadores Java e os resultados obtidos comprovam a eficiência da mesma. Por fim, é apresentada uma ferramenta que implementa a metodologia STEP, automatizando, assim, o projeto e a aplicação de auto-teste *on-line* para os processadores estudados.

Palavras-Chave: projeto de auto-teste, teste *on-line*, auto-teste baseado em software, teste de processadores, processadores embarcados, sistemas de tempo-real.

STEP: Planning, Generation and Selection of On-Line Self-Test for Embedded Processors

ABSTRACT

Processor-based embedded systems have been widely used in safety-critical applications. In such applications, which include from cars break control to spatial missions, the whole system operation must be reliable during long periods even within unknown, hostile and unstable environments. In non-critical applications, system reliability is not a prime requirement, but the final user requires an error free product, with stable behavior. Hence, one can realize the importance of on-line self-testing in current embedded systems.

Self-testing is becoming an important challenge due to the increasing complexity of the systems allied to their strong constraints. In real-time applications this problem becomes even more complex, since, besides meeting systems constraints, one must take into consideration the application timing requirements. Among all on-line self-testing techniques studied, Software-Based Self-Test (SBST) has been distinguished by its effectiveness, low-cost and small impact on system constraints and requirements.

This work proposes a methodology for the design and implementation of on-line self-test in embedded processors, considering real-time applications. Such a methodology, called STEP (Self-Test for Embedded Processors), is based on SBST technique and encloses planning, generation and selection of test routines for the target processor. The proposed method guarantees periodical self-test execution, at the smallest period allowed by the real-time application, and ensures that all embedded system constraints are met. Furthermore, provided solution achieves high test quality while helping in the optimization of the costs of the final system.

The proposed methodology is applied to different architectures of Java processors to demonstrate its efficiency. Finally, this work presents a tool that automates the design and implementation of on-line self-test in the studied processors by implementing the STEP methodology.

Keywords: self-test design, on-line testing, software-based self-test, processors testing, embedded processors, real-time systems.

1 INTRODUÇÃO

Este capítulo introdutório pretende contextualizar o presente trabalho, apresentando a motivação para o seu desenvolvimento, seu objetivo e suas contribuições. Ao final, é descrita a forma como o trabalho está organizado.

1.1 Motivação e Objetivos

Ao longo do seu processo de produção e da sua vida útil, um circuito integrado (CI) é submetido a diferentes tipos de teste. Dentre os mais utilizados, destaca-se o teste estrutural baseado em falhas (LUBASZEWSKI; COTA; KRUG, 2002), o qual permite a obtenção de uma medida quantitativa da efetividade do teste (cobertura de falhas). Esse método de teste consiste na aplicação de estímulos de entrada (denominados vetores ou padrões de teste) seguida pela observação dos valores das saídas do circuito (respostas de teste). A cobertura de falhas refere-se ao percentual de falhas do circuito que podem ser detectadas pelo conjunto de padrões de teste gerados. A obtenção de uma alta cobertura de falhas depende, dentre outras coisas, da testabilidade do circuito que, por sua vez, refere-se à facilidade ou à dificuldade de controle (controlabilidade) e observação (observabilidade) dos diversos pontos de falha.

A primeira dificuldade enfrentada pelo projetista de teste durante o projeto de um sistema digital consiste na geração dos padrões de teste. Embora essa tarefa seja realizada uma única vez para cada circuito projetado, ela consome um tempo considerável do projeto do sistema. Mesmo com o auxílio de ferramentas ATPG (*Automatic Test Pattern Generator*) para a geração automática de padrões de teste, a complexidade dos circuitos torna cada vez mais difícil a obtenção de um conjunto de teste reduzido que forneça boa cobertura de falhas estruturais. Isso ocorre porque tais circuitos contêm diversos caminhos de difícil acesso, tanto para o controle quanto para observação, muitas vezes devido à dificuldade de se inicializar determinados flip-flops. Para a maioria dos sistemas atuais, nem mesmo com um grande número de vetores é possível atingir uma cobertura de falhas aceitável sem a inserção de modificações de projeto visando aumento de testabilidade.

Técnicas de DFT (*Design For Testability*) (BUSHNELL; AGRAWAL, 2000), tanto *ad-hoc* quanto estruturadas, têm sido bastante utilizadas com o objetivo de aumentar a testabilidade do circuito integrado reduzindo, assim, o número de padrões de teste necessários. Métodos *ad-hoc* de DFT consistem em boas práticas de projeto adquiridas com a experiência, como, por exemplo, tornar todos os flip-flops inicializáveis e evitar portas lógicas com um grande número de sinais de *fan-in*. Entretanto, tais técnicas requerem a inspeção manual do circuito, tarefa difícil de ser realizada considerando-se o tamanho dos CIs atuais. Além disso, mesmo que os métodos *ad-hoc* sejam postos em

prática, na maioria dos casos eles não resolvem todos os problemas de testabilidade do circuito.

Técnicas estruturadas de DFT, apesar de mais invasivas, são mais eficazes em se tratando de testabilidade. Entre os métodos mais comumente usados estão a inserção de pontos de teste e de cadeias *scan*. Pontos de teste são caminhos adicionados ao circuito ligando, diretamente, regiões de difícil controle ou observação a entradas ou saídas primárias do CI, respectivamente. Similarmente, cadeias *scan* são caminhos adicionados ao circuito, que interligam um conjunto de flip-flops formando um registrador de deslocamento. Dessa forma, quando em modo de teste, os flip-flops podem ser facilmente controlados ou observados através de entradas ou saídas seriais no CI, respectivamente. Tanto a inserção de pontos de teste quanto a inserção de cadeias *scan* requer a adição de pinos extras ao circuito, para entrada e para saída de dados de teste.

Ao contrário da tarefa de geração de vetores de teste, as etapas de aplicação dos vetores, e captura e análise das respostas de teste são repetidas para cada CI fabricado. Assim, é importante que o tempo gasto nessas etapas seja bastante reduzido. Atualmente, testadores externos, ou ATEs (*Automatic Test Equipments*) (BUSHNELL; AGRAWAL, 2000), são usados para a realização dessas tarefas. No entanto, seu uso vem se tornando praticamente inviável com a rápida evolução dos circuitos integrados. Em primeiro lugar pela memória limitada desses equipamentos, o que requer que os padrões de teste sejam carregados em lotes, aumentando consideravelmente o tempo total de teste. Além disso, os testadores operam, em geral, a uma velocidade inferior à do circuito em teste (CUT - *Circuit Under Test*), impossibilitando a detecção de falhas que afetam o desempenho, e não o valor da saída. Por fim, o custo dos ATEs é, muitas vezes, excessivamente alto em relação ao benefício proporcionado, visto que tais equipamentos são usados somente nos testes de produção.

Após diversas etapas de teste ao longo do processo de produção, o sistema digital é finalmente posto em funcionamento, em seu ambiente natural de operação. Lá ele está suscetível a falhas operacionais causadas por diversos fatores. Entre eles incluem-se a ação do tempo e fatores externos, tais como temperaturas excessivas, vibrações, campos eletromagnéticos, partículas induzidas, etc. (GIZOPOULOS; PASCHALIS; ZORIAN, 2004). Assim sendo, mesmo depois de verificada a ausência de falhas nos testes de produção, o sistema deve ser submetido a testes periódicos para garantir seu correto funcionamento em campo.

Uma solução que tem sido adotada para substituir os testadores externos nos testes de produção, possibilitando, ainda, o teste em campo (*on-line*), são os circuitos auto-testáveis. Auto-teste é definido como a capacidade de um circuito integrado de testar a si próprio, isto é, excitar pontos de falha em potencial e propagar seus efeitos para locais observáveis de fora do *chip*. As tarefas de aplicação dos padrões de teste e captura das respostas de teste são ambas realizadas por recursos internos ao circuito e não por equipamento externo como no teste baseado em ATE. Entretanto, o projeto de auto-teste para um circuito digital demanda um estudo cuidadoso das técnicas disponíveis que aponte para aquela que melhor se adapta às características, aos requisitos e às restrições do sistema alvo.

Todos esses problemas relacionados ao teste de circuitos digitais tornam-se ainda mais relevantes quando o sistema alvo em questão é um sistema embarcado. Isso porque esse tipo de sistema computacional envolve restrições mais complexas que a computação de propósitos gerais, e sem comprometer o desempenho final do sistema

(CARRO; WAGNER, 2003). Por exemplo, as questões da portabilidade e do limite do consumo de potência sem perda de desempenho, a baixa disponibilidade de memória, a necessidade de segurança e confiabilidade, a possibilidade de funcionamento em uma rede maior, e o curto tempo de projeto (WOLF, 2001). Considerando-se ainda o vasto espaço de projeto arquitetural a ser explorado, tem-se o desafio que constitui não apenas desenvolver, mas também testar um sistema eletrônico embarcado.

E os desafios são ainda maiores quando a aplicação alvo do sistema embarcado é uma aplicação de tempo-real. Na computação de tempo-real, a exatidão do sistema depende não apenas do resultado lógico da computação mas também do tempo no qual os resultados são produzidos (STANKOVIC, 1988). Exemplos de sistemas de tempo-real incluem o controle de experimentos de laboratório, plantas nucleares, plantas de controle de processos, sistemas de controle de vôo, robótica, etc. Nesses casos, além das restrições inerentes ao sistema embarcado, é preciso levar em consideração o atendimento dos requisitos temporais do sistema, dos quais depende seu correto funcionamento. Tais requisitos adicionam dificuldades extras tanto no projeto quanto no teste do sistema.

Sistemas embarcados (de tempo-real ou não) baseados em processadores têm sido largamente aplicados em áreas críticas no que diz respeito à segurança de seres humanos e do meio-ambiente. Em tais aplicações, que compreendem desde o controle de freio de carros a missões espaciais, pode ser necessária a execução confiável de todas as funcionalidades do sistema durante longos períodos e em ambientes desconhecidos, hostis ou instáveis. Mesmo em aplicações não críticas (como telefones celulares, tocadores de mp3, máquinas de lavar, etc.), nas quais a confiabilidade do sistema não é um requisito primordial, o usuário final deseja que seu produto apresente comportamento estável e livre de erros. Soma-se a isso a crescente complexidade de tais sistemas, que contribui com o aumento de sua suscetibilidade à ocorrência de falhas (tanto de projeto quanto adquiridas), e tem-se a importância do auto-teste no projeto dos sistemas embarcados atuais.

Sistemas embarcados mais complexos implicam o aumento do número de transistores no circuito integrado e, conseqüentemente, o aumento da probabilidade de ocorrência de falhas. Além disso, a evolução contínua das tecnologias de fabricação de circuitos digitais possibilita a redução no tamanho dos transistores, tornando-os muito mais suscetíveis aos efeitos da radiação (HUGHES; BENEDETTO, 2003), e reduzindo a precisão e, com isso, a confiabilidade do processo de fabricação (AGARWAL; BLAAUW; ZOLOTOV, 2003).

Assim, considerando-se as restrições inerentes aos sistemas embarcados, os fortes requisitos temporais presentes em aplicações de tempo-real, a necessidade de teste *on-line* desses sistemas e a sua crescente complexidade (que, em geral, aumenta sua suscetibilidade e reduz sua testabilidade), torna-se visível o desafio que constitui o projeto de auto-teste *on-line* para sistemas embarcados e, em especial, para os processadores neles inseridos. Entre as técnicas de auto-teste *on-line* atualmente pesquisadas, uma tem se destacado pela eficácia obtida a um baixo custo de projeto e sem grande impacto no atendimento dos requisitos e restrições do sistema: o auto-teste baseado em software (SBST – *Software-Based Self-Test*).

Nesse contexto, o presente trabalho tem como objetivo principal a definição de uma metodologia que permita a automatização do projeto e implantação de auto-teste *on-line* periódico para processadores embarcados, com base na técnica SBST. Tal metodologia

deve levar em conta, não apenas a otimização absoluta do programa de auto-teste (abordagem essa que tem sido o foco das pesquisas atuais na aplicação de SBST), mas também as restrições do sistema embarcado e o atendimento dos requisitos de teste e de tempo-real, se for o caso. Como forma de validação da metodologia proposta, a mesma é aplicada aos processadores que compõem a plataforma Femtojava (ITO; CARRO; JACOBI, 2001; BECK; CARRO, 2003, 2004; KRAPF; CARRO, 2003), cujo projeto foi especificamente voltado para aplicações embarcadas. Nesse estudo de caso, considera-se, ainda, a execução de uma aplicação de tempo-real.

1.2 Contribuições

As principais contribuições deste trabalho são:

- Definição de uma classificação precisa para os diversos métodos de auto-teste em sistemas digitais
- Proposta de uma metodologia para o projeto de auto-teste *on-line* periódico para processadores embarcados, levando em conta seus requisitos e restrições
- Projeto de um módulo de hardware para análise *on-line* das respostas de teste
- Avaliação do impacto da aplicação de SBST *on-line* em sistemas embarcados e/ou de tempo-real
- Desenvolvimento de dois métodos eficientes para a seleção de programas de auto-teste, a partir de rotinas de teste em software, levando em consideração as restrições e requisitos do sistema embarcado alvo
- Definição de rotinas de teste em linguagem de alto-nível para processadores baseados em pilha
- Desenvolvimento de uma ferramenta para a automatização do projeto e aplicação de auto-teste *on-line* periódico para os processadores da Família Femtojava.

1.3 Organização do Trabalho

O presente trabalho é constituído de três partes. A primeira consiste num estudo acerca de diferentes técnicas de auto-teste *on-line* para processadores, com ênfase no auto-teste baseado em software. Na segunda parte, é definida a metodologia para automatização do projeto e aplicação de auto-teste *on-line* periódico para processadores embarcados, com base na técnica SBST. Por fim, os resultados obtidos da aplicação da metodologia proposta aos processadores da família Femtojava comprovam a eficácia da mesma.

O conteúdo deste trabalho é organizado como segue. O Capítulo 2 apresenta algumas definições importantes, servindo de auxílio para os leitores não familiarizados com os conceitos e nomenclaturas das áreas de tolerância a falhas e teste de sistemas digitais. O Capítulo 3 dá inicio à primeira parte do trabalho resumindo os principais mecanismos de auto-teste para circuitos digitais e classificando-os de acordo com características específicas. Esse capítulo também é dedicado ao teste *on-line* de processadores, apresentando as características e limitações das principais classes de métodos e as restrições para sua aplicação em sistemas embarcados e sistemas de tempo-real. Encerrando o estudo sobre auto-teste *on-line* para processadores, o Capítulo

4 apresenta, em detalhes, a técnica de auto-teste baseado em software, visto que ela constitui o foco deste trabalho.

A segunda parte do trabalho é composta unicamente pelo Capítulo 5, no qual é proposta uma metodologia para o projeto e a aplicação de auto-teste *on-line* periódico para processadores embarcados, com o uso da técnica de auto-teste baseado em software. A terceira e última parte é constituida pelo Capítulo 6 que, inicialmente, apresenta os processadores que compõem a família Femtojava e, a seguir, descreve a aplicação da metodologia proposta a esses processadores. Por fim, esse mesmo capítulo propõe um método para automatização do projeto e aplicação de auto-teste *on-line* periódico para os processadores Femtojava, apresentando uma ferramenta desenvolvida com esse objetivo e os resultados obtidos com o seu uso. Conclusões gerais e trabalhos futuros são apresentados no Capítulo 7.

2 CONCEITOS BÁSICOS

Ao longo dos próximos capítulos, diversos conceitos do escopo de teste de sistemas digitais e de tolerância a falhas serão repetidamente referenciados. Este capítulo é dedicado aos leitores pouco familiarizados com essas áreas. Nas próximas seções são apresentadas as definições de defeito, falha, erro, mal-funcionamento, latência, modelo de falhas, teste funcional, teste estrutural, e teste *at-speed*. Em cada seção, os conceitos mais estreitamente relacionados ao conteúdo deste trabalho são destacados. Sua perfeita compreensão é importante para o pleno entendimento dos capítulos que seguem.

2.1 Defeito, Falha, Erro e Mal-Funcionamento

Os termos *defeito* (*defect*), *falha* (*fault*), *erro* (*error*) e *mal-funcionamento*[1] (*failure*) são, por diversas vezes, utilizados de modo confuso na literatura relativa a teste de sistemas digitais. Em alguns casos, seus significados diferem daqueles atribuídos pelos pesquisadores de tolerância a falhas. Além disso, não existe, ainda hoje, uma padronização desses termos para a língua portuguesa, que unifique os conceitos das áreas de teste, e de tolerância a falhas. Neste trabalho, eles serão usados de acordo com as definições dadas a seguir (CLARK; PRADHAN, 1995; BUSHNELL; AGRAWAL, 2000).

Um *defeito* em um sistema eletrônico é a diferença não intencional entre o dispositivo físico real e o projeto pretendido. Exemplos de defeitos são a falta de janelas de contato, transistores parasitas, eletromigração, degradação de contatos, etc. Defeitos podem ocorrer tanto durante a fabricação do dispositivo, devido a imperfeições no processo, quanto durante o seu uso, por ação do tempo ou por fatores externos, como temperaturas excessivas, vibrações e outros. Sua ocorrência repetida indica a necessidade de melhorias no processo de fabricação ou no projeto do dispositivo. A abstração de um defeito no nível de funções (modelos físicos ou lógicos do dispositivo) é denominada *falha*. A diferença entre defeito e falha é bastante sutil. De uma maneira simples, eles podem ser definidos como imperfeições no hardware e na função desejada, respectivamente.

Falhas que ocorrem após o processo de fabricação, durante a atividade do sistema, são ditas *operacionais* e categorizadas principalmente por sua duração (CLARK; PRADHAN, 1995). *Falhas permanentes* são aquelas que continuam infinitamente

[1] O termo *failure* pertence ao escopo de tolerância a falhas. Duas traduções para ele podem ser encontradas na literatura desta área: *falha* e *defeito*. O termo *mal-funcionamento* é uma tradução livre deste autor, que pretende evitar conflito com as traduções de *fault* e *defect*, mais fortemente estabelecidas na literatura de teste de sistemas digitais.

ativas no mesmo local, e refletem defeitos irreversíveis causados por dano, fadiga ou imperfeições na fabricação. Uma vez que esse tipo de falha tenha ocorrido em um componente, este só pode ser restaurado por substituição ou, se possível, reparo. *Falhas intermitentes*, por sua vez, aparecem repetidamente no mesmo local e são ativadas por mudanças no ambiente. Essas falhas só podem causar erros quando estão ativas (não dormentes) e, em muitos casos, precedem à ocorrência de falhas permanentes. Já as *falhas transientes* aparecem de modo irregular em vários pontos do circuito e têm curta duração. Tais falhas podem ser causadas por flutuações de voltagem, interferência eletromagnética ou radiação.

Um valor de saída errôneo produzido por um circuito defeituoso (com falha) é denominado *erro*. Em outras palavras, um erro é o efeito de uma falha ativa. Uma única falha pode originar vários erros, logo, é mais fácil tratar uma única falha do que os diversos erros causados por ela. Finalmente, erros não tratados ou não contidos causam o *mal-funcionamento* do sistema. Ou seja, o mal-funcionamento consiste no efeito, perceptível ao usuário final, de um erro não contido, que por sua vez foi originado por uma falha ativa.

Como exemplo, consideremos um sistema defeituoso com uma porta AND de duas entradas *a* e *b*, e uma saída *c*. Nesse exemplo, o defeito é caracterizado pela falta de uma janela de contato em um dos transistores da porta AND. Uma possível falha relacionada a esse defeito é o sinal *b* estar colado ao (*stuck-at*) valor lógico zero. Assim, supondo que $a = 1$ e $b = 1$, a falha produziria um erro na saída do circuito (saída fornecida $c = 0$; saída correta $c = 1$). É importante observar que o erro não é permanente, embora a falha o seja, uma vez que se uma das entradas for zero, a saída fornecida será correta. Se o erro não for tratado, o sistema poderia apresentar um mal-funcionamento caracterizado, por exemplo, por um problema na abertura de um arquivo.

Mecanismos de teste podem visar tanto à detecção de falhas quanto à detecção de erros. O presente trabalho se insere no contexto da detecção de falhas, uma vez que tem como base a técnica SBST, a qual, como será visto adiante, é uma abordagem baseada na detecção de falhas. Ademais, por ser aplicado periodicamente, o teste aqui implementado possibilita a detecção de falhas intermitentes (desde que tenham duração considerável) e permanentes.

2.2 Latência

Os conceitos relacionados à latência são importantes para a identificação das técnicas de teste mais adequadas para determinados requisitos de sistema. As definições aqui apresentadas são uma junção dos conceitos relativos à literatura de teste de sistemas digitais e de tolerância a falhas (CLARK; PRADHAN, 1995; PFLANZ; VIERHAUS, 2001).

Conforme descrito na seção anterior, quando uma falha causa uma mudança incorreta no valor de saída do circuito, ocorre um erro. O tempo compreendido entre a ocorrência da falha e o surgimento do primeiro erro é denominado *latência de falha*. Uma vez que a falha pode permanecer ativa no mesmo ponto, múltiplos erros podem ser originados de um único ponto de falha, propagando-se pelo sistema. Caso um erro não seja contido, o tempo decorrido entre sua ativação e a primeira manifestação de mal-funcionamento do sistema é chamado *latência de erro*.

Se um circuito possui mecanismos de teste baseados na detecção de erros, tais mecanismos irão detectar um erro após um período de tempo denominado *latência de detecção de erro*. Em sistemas tolerantes a falhas, os mecanismos apropriados devem tratar ou conter o erro dentro do intervalo de tempo compreendido entre sua detecção e a manifestação de mal-funcionamento do sistema. A esse período de tempo dá-se o nome *latência para tratamento do erro*. Ou seja, para que um sistema provido de mecanismos de detecção de erros seja tolerante a falhas, a soma da latência de detecção de erro com a latência para tratamento do erro deve ser menor do que a latência de erro. Caso contrário, o mal-funcionamento será percebido pelo usuário.

Em sistemas que implementam técnicas de teste baseadas na detecção de falhas, a detecção de uma falha pode ocorrer durante a latência de falha, durante a latência de erro, ou mesmo após a manifestação do mal-funcionamento do sistema, desde que a falha esteja ativa. O tempo transcorrido entre a ativação da falha e sua detecção é denominado *latência de detecção de falha*. Dessa forma, sistemas tolerantes a falhas providos de mecanismos de detecção de falhas devem garantir que a latência de detecção de falha seja menor do que a latência de falha. Caso contrário, se a latência de detecção de falha for menor do que a soma da latência de falha com a latência de erro, o sistema deve ser capaz de identificar os erros gerados pela falha detectada, e tratá-los adequadamente antes que o mal-funcionamento seja ativado.

A adaptação (PFLANZ; VIERHAUS, 2001) mostrada na Figura 2.1 representa um diagrama de tempo ilustrando as definições de latência apresentadas.

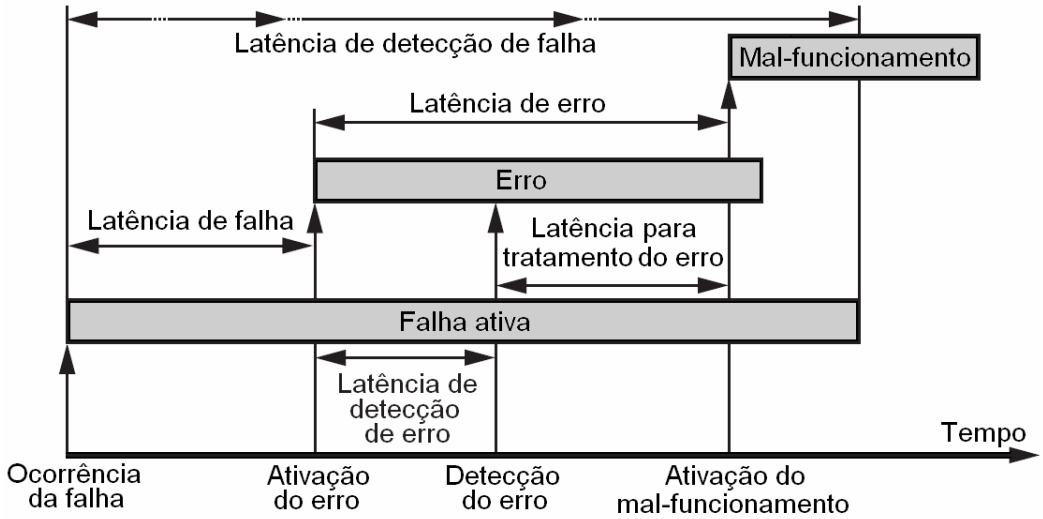

Figura 2.1: Diagrama de tempo com latências de falha, de erro e de detecção

2.3 Modelo de Falhas

A eficiência do teste depende fortemente da precisão e do realismo com que as falhas são modeladas. O modelo de falhas, por sua vez, representa as mudanças que a falha causa nos sinais do circuito (LALA, 1997). Nas últimas décadas, vários modelos de falhas foram propostos na tentativa de representar corretamente defeitos físicos (WADSACK, 1978; GALAY; CROUZET; VERNIAULT, 1980; COURTOIS, 1981; RAJSUMAN, 1992) e falhas transientes (LAGUNA; TREECE, 1986).

A modelagem de falhas está intimamente relacionada à modelagem do circuito (BUSHNELL; AGRAWAL, 2000). Na hierarquia de projeto, níveis referem-se a graus de abstração. Por exemplo, o *nível comportamental*, também denominado *alto-nível*, apresenta poucos detalhes de implementação. Logo, modelos de falhas nesse nível podem não ter correlação óbvia com defeitos de fabricação ou adquiridos. Modelos de falhas de alto-nível são mais importantes em verificação de projeto baseada em simulação do que em teste. Exceções são os modelos de falhas funcionais em memórias semicondutoras, uma vez que a função da memória é simples e sua estrutura é regular.

O *nível de transferência entre registradores* (RTL - *Register-Transfer Level*) ou *nível lógico* consiste de um *netlist* de portas lógicas. As falhas *stuck-at* (Seção 2.3.1), desse nível, constituem o mais popular modelo de falhas em teste digital. O nível lógico também inclui, entre outros, os modelos de falhas de *atraso* (Seção 2.3.2) e de *bridging* (Seção 2.3.3). Já os *níveis de transistores* ou mais baixos, chamados *níveis de componente*, incluem falhas do tipo *stuck-open*, também conhecidas como falhas *dependentes de tecnologia*. Falhas do nível de componente são modeladas principalmente no teste de circuitos analógicos.

Finalmente, existem os modelos de falhas que não se enquadram em nenhuma das hierarquias de projeto. Um exemplo típico são as falhas de *corrente quiescente* (I_{DDQ}), relevantes para a tecnologia CMOS (*Complementary Metal Oxide Semiconductor*), visto que causam um aumento de várias ordens de magnitude no valor da corrente de fuga em portas lógicas CMOS. A utilidade desses modelos vem do fato de que eles podem representar alguns defeitos físicos não representados por qualquer outro modelo. Por isso eles são denominados modelos *realísticos*.

Os modelos de falhas mais utilizados atualmente no teste de circuitos digitais pertencem ao nível lógico: falhas *stuck-at*, falhas de atraso e falhas de *bridging*. Nas próximas seções esses modelos são brevemente descritos.

2.3.1 Falhas *Stuck-at*

Um circuito pode ser modelado como uma interconexão de portas lógicas, chamada *netlist*. Assume-se que uma falha *stuck-at* afeta apenas a interconexão entre portas. Cada linha de conexão pode ter dois tipos de falhas: *stuck-at-1* (abreviada como *s-a-1*) e *stuck-at-0* (representada por *s-a-0*). Assim, uma linha com uma falha *stuck-at-1* estará colada em um, ou seja, terá sempre o valor lógico 1, independentemente do valor de saída correto da porta direcionada a ela. Da mesma forma, uma linha com uma falha *stuck-at-0* estará colada em zero, ou seja, terá sempre o valor lógico 0, mesmo quando o valor correto for 1. O modelo *stuck-at*, também chamado de modelo de falhas *clássico*, oferece uma boa representação para os tipos de defeitos mais comuns em muitas tecnologias, por exemplo, curtos-circuitos e circuitos abertos (LALA, 1997).

Em geral, várias falhas *stuck-at* podem estar simultaneamente presentes no circuito. Cada linha de interconexão pode estar em um dos três estados: *s-a-0*, *s-a-1*, ou livre de falhas. Todas as possíveis combinações, com exceção daquela que tem todas as linhas no estado livre de falhas, são contadas como falhas. Assim, mesmo um circuito de tamanho moderado terá um número imenso de falhas *stuck-at* múltiplas. Por isso, é prática comum modelar apenas falhas *stuck-at* simples. Nesse modelo, apenas uma linha do circuito pode conter uma falha *stuck-at* a cada momento. Logo, considera-se que um circuito de n linhas de interconexão pode ter no máximo $2n$ falhas *stuck-at* simples.

2.3.2 Falhas de Atraso

Pequenos defeitos no circuito, como curto-circuito parcial ou circuito parcialmente aberto, têm maior probabilidade de acontecer devido a variações estatísticas no processo de fabricação (LALA, 1997). Tais defeitos resultam em falhas no atendimento das especificações temporais do circuito sem quaisquer alterações em sua função lógica. Um pequeno defeito pode atrasar a transição de um sinal de 0 para 1, ou vice-versa. Ocorrências desse tipo são modeladas por falhas de *atraso* (*delay faults*) que, em outras palavras, afetam o desempenho do circuito fazendo com que o seu atraso combinacional exceda o período de relógio. Especificamente, falhas de atraso podem ser falhas de transição (*transition*), falhas de atraso de porta (*gate-delay*), falhas de atraso de linha (*line-delay*), falhas de atraso de segmento (*segment-delay*) e falhas de atraso de caminho (*path-delay*) (BUSHNELL; AGRAWAL, 2000).

2.3.2.1 Falhas de Transição

Nesse modelo assume-se que, no circuito sem falhas, todas as portas lógicas e interconexões têm algum atraso nominal, e que o atraso de uma única porta ou interconexão foi alterado. Tal alteração é caracterizada por um incremento no atraso nominal da porta ou interconexão, grande o suficiente para impedir que uma transição no valor do sinal alcance qualquer saída primária do circuito dentro do período de relógio, mesmo que a transição se propague pelo caminho mais curto. Falhas de transição podem ser de subida (*slow-to-rise*, transição de 0 para 1) ou de descida (*slow-to-fall*, transição de 1 para 0).

2.3.2.2 Falhas de Atraso de Porta

Falhas de atraso de porta são uma especialização das falhas de transição. Elas causam o aumento do atraso da entrada para a saída de uma única porta lógica, enquanto todas as outras mantêm seu valor de atraso nominal. O aumento no atraso da porta com falha é denominado o *tamanho* da falha de atraso de porta. Uma vez que falhas de atraso de porta podem ser de subida e de descida, o número total de falhas em um circuito é igual a duas vezes o número de portas lógicas que ele contém.

2.3.2.3 Falhas de Atraso de Linha

Essas falhas modelam os atrasos de subida e de descida de uma dada linha de sinal. Ao contrário da falha de transição, onde a transição pode ser propagada por qualquer caminho, um teste para falhas de atraso de linha deve propagar a transição através do mais longo caminho sensibilizado. Em caso de falhas simples, o número de falhas do circuito fica limitado a duas vezes o número de linhas.

2.3.2.4 Falhas de Atraso de Segmento

Nesse modelo, assume-se que um segmento de tamanho L equivale a uma cadeia de L portas lógicas. Tal segmento pode estar contido em um ou mais caminhos da entrada para a saída do circuito. Uma falha de atraso de segmento aumenta o atraso de um segmento de modo que todos os caminhos que o contêm terão uma falha de atraso de caminho. Se L é a máxima profundidade combinacional do circuito, então a falha de atraso de segmento se torna uma falha de atraso de caminho. Quando $L = 1$, a falha de atraso de segmento se torna idêntica a uma falha de atraso de porta. Duas falhas, correspondendo aos dois tipos de transição (subida e descida), são modeladas para cada segmento.

2.3.2.5 Falhas de Atraso de Caminho

Esse tipo de falha faz com que o atraso de propagação cumulativo de um caminho combinacional aumente, ultrapassando algum limite de tempo especificado. O caminho combinacional começa numa entrada primária do circuito ou em um flip-flop, contém uma cadeia de portas lógicas, e termina numa saída primária do circuito ou em outro flip-flop. O limite de tempo especificado pode ser o período do relógio ou o período do ciclo de teste. O atraso de propagação é definido para a propagação de uma transição de sinal através do caminho. Portanto, para cada caminho combinacional existem duas falhas de atraso de caminho, correspondendo às transições de subida e de descida.

2.3.3 Falhas de *Bridging*

Podendo ser modeladas tanto no nível lógico quanto no nível de transistores, as falhas de *bridging* representam um curto-circuito entre um grupo de sinais. O valor lógico da rede em curto pode ser modelado como 1-dominante (*wired*-OR), 0-dominante (*wired*-AND), ou indeterminado, dependendo da tecnologia na qual o circuito foi fabricado.

Falhas de *bridging* podem ser categorizadas em três grupos: *input bridging*, *feedback bridging* e *nonfeedback bridging* (LALA, 1997). Uma falha de *input bridging* corresponde a um curto-circuito entre certo número de sinais de entrada primária, enquanto uma falha de *feedback bridging* ocorre se há um curto entre um sinal de saída e um sinal de entrada. Uma falha de *nonfeedback bridging* identifica uma falha de *bridging* que não pertence a nenhuma das categorias anteriores. Falhas de *input bridging* e *nonfeedback bridging* são combinacionais, e sua cobertura pelo teste de falhas *stuck-at* é normalmente muito alta (BUSHNELL; AGRAWAL, 2000). Esse não é o caso de falhas de *feedback bridging*, que produzem estados de memória onde deveria haver lógica combinacional.

Os modelos de falhas mais adequados para o teste *on-line* de processadores são aqueles pertencentes ao nível lógico, ou RTL. No desenvolvimento deste trabalho, considerou-se apenas o modelo de falhas *stuck-at* simples. Entretanto, conforme descrito na seção anterior, o teste para falhas *stuck-at* detecta grande parte das falhas de *input bridging* e *nonfeedback bridging*. Além disso, como será visto adiante, o método usado na metodologia desenvolvida realiza o teste na mesma freqüência de operação normal do circuito. Por isso, algumas falhas de atraso também são cobertas. Dessa forma, pode-se dizer que o método desenvolvido neste trabalho abrange o modelo de falhas *stuck-at* e, potencialmente, os modelos de falhas de atraso e de *bridging*.

2.4 Teste *At-Speed*

No teste *at-speed*, os vetores de teste são aplicados e as respostas observadas na mesma freqüência de operação normal do circuito (BUSHNELL; AGRAWAL, 2000). Esse tipo de teste é essencial para os atuais *chips* VLSI (*Very Large Scale Integration*), os quais comandam sistemas extremamente rápidos e são, em geral, produzidos através dos processos mais avançados, os quais estão cada vez mais próximos dos limites impostos pela atual tecnologia de fabricação de circuitos integrados.

A utilização de vetores de teste para falhas *sutck-at* em testes *at-speed* é uma estratégia freqüentemente aplicada, embora não seja a mais adequada. Apesar de essa

estratégia possibilitar a detecção de algumas falhas de atraso, ela é, em geral, incapaz de atingir uma cobertura de falhas muito alta para esse modelo de falhas. Nos casos em que uma alta cobertura de falhas de atraso se faz necessária, deve-se incluir, no teste *at-speed*, padrões de teste para falhas de atraso de caminho, ao menos para os caminhos críticos.

A metodologia de teste desenvolvida no presente trabalho realiza teste *at-speed*, uma vez que o próprio processador é responsável pela aplicação dos padrões de teste e captura das respostas. Logo, o teste é executado na freqüência de operação do circuito, e não em uma freqüência inferior, como no caso do teste baseado em ATE. Assim, mesmo não visando à detecção de falhas de atraso, o método aqui aplicado também pode detectar algumas dessas falhas como efeito colateral do teste para falhas *stuck-at*.

2.5 Teste Funcional e Teste Estrutural

Os tipos de teste de sistemas digitais têm objetivos bem definidos e distintos. “Em geral, o teste deve verificar se o projeto está de acordo com as especificações, se o circuito tem um comportamento funcional correto (teste funcional) ou se a implementação física do circuito espelha seu esquemático (teste estrutural)” (LUBASZEWSKI; COTA; KRUG, 2002, p.168).

O *teste funcional* pretende verificar se todas as funções atribuídas ao circuito digital estão sendo executadas corretamente (GIZOPOULOS; PASCHALIS; ZORIAN, 2004). Além disso, ele deve identificar problemas de desempenho (teste *at-speed*), como atrasos excessivos, por exemplo (LUBASZEWSKI; COTA; KRUG, 2002). No caso de processadores, o teste funcional visa cobrir as diferentes funções implementadas pelo conjunto de instruções da arquitetura (ISA - *Instruction Set Architecture*). Portanto, o teste funcional de processadores necessita apenas de informações sobre o ISA (e nenhum outro modelo de mais baixo nível, como um *netlist*) para o desenvolvimento de um conjunto de padrões de teste.

O principal problema do teste funcional consiste no fato dele não estar diretamente relacionado à real testabilidade estrutural do circuito, que diz respeito a defeitos físicos. A testabilidade estrutural que um teste funcional atinge depende fortemente do conjunto de dados (operandos, no caso de processadores) usados para testar as funções do circuito. Na maioria dos casos, operandos pseudo-aleatórios são empregados no teste funcional, levando a conjuntos de teste com tempo de aplicação excessivamente longo e incapazes de atingir alta cobertura de falhas estruturais. O uso de teste funcional geralmente se faz necessário para verificação de projeto. Seqüências de teste previamente desenvolvidas para essa finalidade podem ser reusadas para testes posteriores, reduzindo o custo de desenvolvimento do teste.

Por outro lado, o *teste estrutural* visa um modelo de falhas estruturais específico e, por isso, pode prover uma medida quantitativa de efetividade do teste (cobertura de falhas). Ferramentas ATPG podem ser usadas para geração automática de seqüências de teste com possível suporte a técnicas de DFT, como cadeias *scan* ou inserção de pontos de teste. Para a geração de teste estrutural, quase sempre se faz necessário um modelo do circuito no nível de portas lógicas. Se essa informação estiver disponível, uma alta cobertura de falhas pode ser obtida para o modelo de falhas estruturais visado com um pequeno conjunto de teste e curto tempo de aplicação (GIZOPOULOS; PASCHALIS; ZORIAN, 2004). Nos casos em que o teste estrutural não pode ser aplicado na

freqüência de operação do sistema (teste *at-speed*), o teste funcional também deve ser realizado no circuito, embora isso leve a um tempo de teste mais longo.

A metodologia desenvolvida neste trabalho mescla o teste funcional e o teste estrutural. O teste é estrutural no sentido em que é baseado em falhas, e visa a um modelo de falhas específico. Por esse motivo faz-se necessário o conhecimento da organização do processador alvo, em nível lógico, para a geração dos padrões de teste estruturais. Por outro lado, a aplicação dos padrões de teste e captura das respostas de teste é funcional, uma vez que ambas utilizam-se unicamente do conjunto de instruções do processador. Com isso obtém-se o melhor das duas abordagens, pois é possível atingir uma cobertura de falhas estruturais alta, para um modelo de falhas específico, através da aplicação funcional dos vetores de teste, o que permite que o teste seja *at-speed*.

2.6 Resumo e Conclusões

Neste capítulo foram apresentados conceitos relativos a teste de sistemas digitais e a tolerância a falhas, os quais serão utilizados ao longo do trabalho.

Viu-se que os termos defeito, falha, erro e mal-funcionamento têm significados distintos. Defeito é uma imperfeição física do hardware produzido em relação ao que foi projetado, enquanto a falha representa a abstração de um defeito em nível de funções. Falhas podem ser permanentes, intermitentes ou transientes, de acordo com seu tempo de duração. Erro é o efeito causado por uma falha ativa, na saída do circuito. Erros não contidos produzem efeitos perceptíveis ao usuário do sistema, aos quais se dá o nome mal-funcionamento.

Foram apresentados também os conceitos de latência de falha, de erro e de detecção. A latência de falha indica o tempo compreendido entre a ocorrência de uma falha e o aparecimento do primeiro erro causado por ela. A latência de erro corresponde ao tempo decorrido entre o surgimento do erro e o mal-funcionamento do sistema. As latências de detecção de falha e de erro referem-se aos períodos compreendidos entre a ocorrência de uma falha e de um erro, respectivamente, e sua detecção.

Em seguida, foi demonstrada a importância de um modelo de falhas realístico para o sucesso do teste. O modelo de falhas representa as alterações que uma determinada falha causa nos sinais do circuito. Existem modelos de falhas para cada nível da hierarquia de projeto. Ao nível comportamental pertencem os modelos de falhas funcionais em memórias semicondutores. O nível lógico inclui os modelos de falhas *stuck-at*, de atraso e de *bridging*, os mais utilizados em teste digital. No nível de componentes há os modelos de falhas dependentes de tecnologia, como as falhas *stuck-open*, mais adequadas para teste analógico. Por fim, existem os modelos de falhas que não pertencem a nenhum nível da hierarquia de projeto, como as falhas de corrente quiescente.

Outro conceito apresentado foi o teste *at-speed*, o qual visa principalmente à detecção de falhas relativas a desempenho, como falhas de atraso. Nele, os vetores de teste são aplicados e as respostas observadas na mesma freqüência de operação normal do circuito. Uma estratégia bastante utilizada para esse tipo de teste é a aplicação de padrões de teste para falhas *stuck-at* na mesma freqüência de operação do sistema. Nesse caso, para que se consiga um bom resultado de teste, é indicado que se incluam,

também, vetores de teste para falhas de atraso de caminho, no mínimo para os caminhos críticos.

Finalmente, teste estrutural e teste funcional foram descritos e diferenciados. O teste funcional tem por objetivo verificar se todas as funções do circuito estão sendo executadas corretamente e no tempo determinado. Em processadores, por exemplo, o teste funcional deve validar o conjunto de instruções da arquitetura. Entretanto, esse tipo de teste não pode prover uma medida quantitativa da sua efetividade, ao contrário do teste estrutural, que visa à detecção de falhas estruturais de um modelo específico. Por isso a efetividade do teste estrutural, que nem sempre pode ser executado *at-speed*, é verificada através da sua cobertura de falhas.

Com base nos conceitos aqui apresentados e em algumas características, já fornecidas, do trabalho descrito nos próximos capítulos, é possível extrair algumas conclusões preliminares. A primeira mostra que a metodologia de teste desenvolvida nesta dissertação visa à detecção de falhas, tanto permanentes quanto intermitentes. Sendo o teste aplicado periodicamente, é importante que o período de teste (intervalo de tempo entre duas execuções consecutivas do teste) seja o menor possível no intuito de reduzir a latência de detecção de falha. Com isso, mecanismos de tolerância a falhas que venham a ser adicionados terão mais tempo para impedir a ocorrência e/ou a propagação de erros, evitando, assim, o mal-funcionamento do sistema.

Além disso, conclui-se que o método de teste adotado neste trabalho mistura as estratégias estrutural e funcional. Tal peculiaridade possibilita a obtenção de uma alta cobertura de falhas para o modelo visado (característica do teste estrutural), e permite, potencialmente, a detecção de falhas que afetam o desempenho do circuito, pela execução de teste *at-speed* (característica do teste funcional). Além do modelo visado (falhas *stuck-at*) e das falhas que afetam o desempenho (falhas de atraso), o uso de vetores de teste para falhas *stuck-at* pode implicar também a detecção de algumas falhas de *bridging*.

3 TESTE DE CIRCUITOS DIGITAIS

O teste de circuitos integrados envolve diversas etapas realizadas em diferentes lugares por diferentes pessoas. Seu objetivo é identificar e isolar dispositivos falhos, ou mesmo substituir partes defeituosas.

Quando um novo *chip* é projetado, e antes que seja enviado para produção, a primeira etapa de teste deve verificar se o projeto está correto, ou seja, se atende a todas as especificações, e se o procedimento de teste é valido. Essa etapa geralmente requer o envolvimento do engenheiro de projeto e é realizada no próprio centro de desenvolvimento, ao invés da fábrica. A ela dá-se o nome *teste de verificação*, ou *caracterização* (BUSHNELL; AGRAWAL, 2000). A caracterização inclui, entre outros tipos de teste, testes funcionais. Com base nos seus resultados, tanto o projeto quanto o procedimento de teste podem sofrer modificações.

O sucesso no teste de verificação indica o início da produção. Cada *chip* fabricado é submetido, ainda na fábrica, a *testes de manufatura* ou *produção*. Nessa etapa são realizados testes estruturais visando à obtenção de uma alta cobertura para as falhas modeladas. Não há preocupação em cobrir todas as possíveis funções e padrões de dados do circuito, mas verifica-se se alguns parâmetros do CUT estão consistentes com as especificações do dispositivo sob condições normais de operação. Essa é uma etapa de alto custo uma vez que todos os dispositivos fabricados precisam ser testados. *Chips* defeituosos são descartados. Em alguns casos, os circuitos descartados passam por uma etapa de diagnóstico para identificação e localização da origem da falha.

Quando os *chips* produzidos são recebidos pelo cliente, eles podem ser novamente testados para garantir sua qualidade. Esse teste, conhecido como *inspeção de chegada* ou *teste de aceitação*, pode ser conduzido tanto pelo usuário quanto por alguma empresa de teste independente. Quando o cliente é um fabricante de sistemas, o teste de aceitação é realizado antes de o dispositivo ser integrado ao sistema. A inspeção de chegada é feita em uma amostra aleatória de *chips*, cujo tamanho depende da qualidade do dispositivo e de requisitos do sistema. O objetivo principal desse teste é evitar que dispositivos defeituosos sejam integrados ao sistema, onde o custo do diagnóstico é muito superior ao custo da inspeção de chegada.

Os testes de verificação, de produção e de aceitação são denominados testes fora de funcionamento (*off-line*), uma vez que são executados quando o dispositivo não está em operação (LUBASZEWSKI; COTA; KRUG, 2002). Com exceção da etapa de caracterização, na qual se utilizam ferramentas de simulação e de verificação, esses testes são usualmente realizados através de equipamentos testadores, também chamados testadores externos ou ATEs. O propósito básico de um testador é direcionar as entradas e monitorar as saídas de um circuito em teste.

O equipamento testador é um instrumento usado para aplicar padrões de teste a um CUT, analisar as respostas fornecidas, e marcar o circuito como bom ou ruim (BUSHNELL; AGRAWAL, 2000). Um ATE contém uma ou mais cabeças de teste (*test heads*), onde pode ser acomodado tanto um *wafer* (pastilha de silício oriunda do processo de fabricação e que contém vários circuitos) quanto um circuito já encapsulado. A freqüência máxima de operação do testador indica se o teste pode ser realizado *at-speed* e, juntamente com número de padrões de teste, define o tempo de teste. Para circuitos muito complexos, que necessitam de um grande número de vetores de teste, os vetores precisam ser carregados e as respostas de teste descarregadas em lotes, visto que, em geral, a memória dos ATEs não é grande o suficiente (GIZOPOULOS; PASCHALIS; ZORIAN, 2004). Esse fator contribui bastante para o aumento no tempo de teste. Considerando o alto custo de uso desses equipamentos e a grande quantidade de circuitos que precisam ser testados, o tempo de teste é um fator fundamental no projeto do teste de um circuito integrado.

Após as três etapas de teste *off-line*, o circuito pode ser integrado ao sistema onde irá operar. Porém, para sistemas que requerem alta segurança, como sistemas de aviação, automotivos, espaciais, plantas nucleares e outros, faz-se necessária uma verificação constante para garantir o correto funcionamento do circuito durante toda sua vida útil. A esse tipo de teste, em que a detecção de falhas é feita com o sistema já operativo visando validar suas operações, dá-se o nome de teste em campo, ou *on-line*. O teste *on-line* é necessário porque mesmo um CI que tenha sido corretamente fabricado, extensivamente testado durante sua produção e considerado livre de defeitos pode adquirir falhas, mais tarde, devido a diversos fatores que aparecem em campo. Tais fatores incluem, além da ação do tempo, fatores externos como temperaturas excessivas, vibrações, campos eletromagnéticos, partículas induzidas, etc. (GIZOPOULOS; PASCHALIS; ZORIAN, 2004).

O uso de ATEs para os testes de produção é a abordagem tradicional seguida pela maioria dos grandes fabricantes de circuitos integrados. Entretanto, pequenos volumes de produção não justificam o alto custo dos equipamentos testadores. Esse compromisso de custo-benefício, somado a outros fatores, vem abrindo espaço para o uso de técnicas de auto-teste. Além de ter um custo muito reduzido nos testes de produção, o uso de auto-teste possibilita a realização de teste *on-line*, tarefa essa essencial para sistemas críticos e impraticável com ATEs.

3.1 Métodos de Auto-Teste

Auto-teste é definido como a capacidade de um circuito integrado de testar a si próprio, isto é, excitar pontos de falha em potencial e propagar seus efeitos para locais observáveis fora do *chip* (GIZOPOULOS; PASCHALIS; ZORIAN, 2004). As tarefas de aplicação dos padrões de teste e captura das respostas de teste são ambas realizadas por recursos internos ao circuito e não por equipamento externo como no teste baseado em ATE. Desse modo, os recursos usados para realização de tais tarefas devem ser capazes de testar a si próprios, ou seja, o circuito extra usado para o auto-teste (caso exista) deve também ser testável. Esse requisito, em geral, adiciona dificuldades extras às metodologias de auto-teste.

O uso de auto-teste proporciona uma série de vantagens sobre a utilização de testadores externos. Algumas delas são listadas abaixo (GIZOPOULOS; PASCHALIS; ZORIAN, 2004):

- os custos relativos à compra, uso e manutenção do ATE são praticamente eliminados quando técnicas de auto-teste são aplicadas;
- mecanismos de auto-teste têm acesso muito mais fácil a nodos internos do circuito do que testadores externos, implicando uma maior capacidade de detecção de falhas. Além disso, eles geralmente obtêm coberturas de falhas mais altas, para um dado modelo de falhas, porque permitem a aplicação de um número maior de vetores de teste em tempo aceitável;
- técnicas de auto-teste possibilitam a detecção de falhas que se manifestam apenas na velocidade real de operação do circuito (teste *at-speed*), tais como falhas de atraso. A superioridade do auto-teste nesse ponto faz com que ele seja, muitas vezes, a única opção, porque devido aos rápidos avanços na tecnologia de fabricação de *chips* um grande número de falhas só pode ser detectado por teste *at-speed*;
- não há perda de rendimento (*yield loss* - circuitos bons descartados) devido à imprecisão nas medições do ATE porque o circuito testa a si próprio. Essa é uma grande preocupação na atual produção de *chips*, que já sofre sérias perdas devido a imperfeições no processo de fabricação. A perda de rendimento devido a *overtesting* (teste para falhas que não afetam a operação normal do circuito) pode continuar existindo no auto-teste se o método adotado testar o circuito em um modo de operação diferente do normal (Seção 3.1.2);
- os recursos de hardware adicionados ao circuito para realização do auto-teste podem ser reusados em estágios posteriores do ciclo de vida do *chip* (possibilitando tanto o teste *off-line* quanto o teste *on-line*) e na execução de diferentes tarefas (tanto tarefas de teste quanto tarefas operacionais);
- o uso de auto-teste nos atuais CIs oferece uma redução significativa no tempo de aplicação do teste em relação ao uso de ATEs. Tanto porque o auto-teste é realizado na mesma freqüência de operação do circuito, quanto por evitar a necessidade de múltiplas cargas de vetores de teste (como acontece com os ATEs).

Considerando as vantagens apresentadas, é possível entender porque as pesquisas na área de teste de sistemas digitais estão cada vez mais voltadas para mecanismos de auto-teste. Entretanto, ainda não foi estabelecida uma classificação precisa para os métodos de auto-teste. Muitas vezes, os termos utilizados por um autor para descrever determinada técnica contradizem a classificação usada por outro autor. Assim, nas próximas seções serão apresentadas algumas técnicas de auto-teste seguindo uma classificação precisa e coerente com seus objetivos, características e aplicações.

Os métodos de auto-teste, da mesma forma que o teste em geral, são classificados de acordo com o estado do sistema durante sua execução. Quando aplicado em campo, com o sistema já operativo, o auto-teste é dito *on-line*. Caso o sistema necessite ser desligado para a realização do teste ou este seja executado durante o processo de fabricação do *chip*, diz-se que o teste é *off-line*.

Ao contrário do teste *off-line*, os mecanismos de teste *on-line* realizam o teste enquanto o sistema está em atividade, mesmo que para isso seja necessária a interrupção temporária da execução da aplicação. Assim, eles têm a vantagem de detectar falhas que se manifestam apenas na freqüência e no ambiente natural de operação do sistema

embarcado. Além disso, essa abordagem implica, em geral, uma redução de desempenho menor que a do teste *off-line* por dispensar a desativação total do sistema, motivo pelo qual tem sido muito utilizada.

As estratégias de auto-teste *on-line* são usualmente classificadas em dois grupos: concorrentes e não concorrentes. A Figura 3.1 mostra um diagrama de classificação das técnicas de auto-teste. Nas próximas seções cada categoria será abordada individualmente.

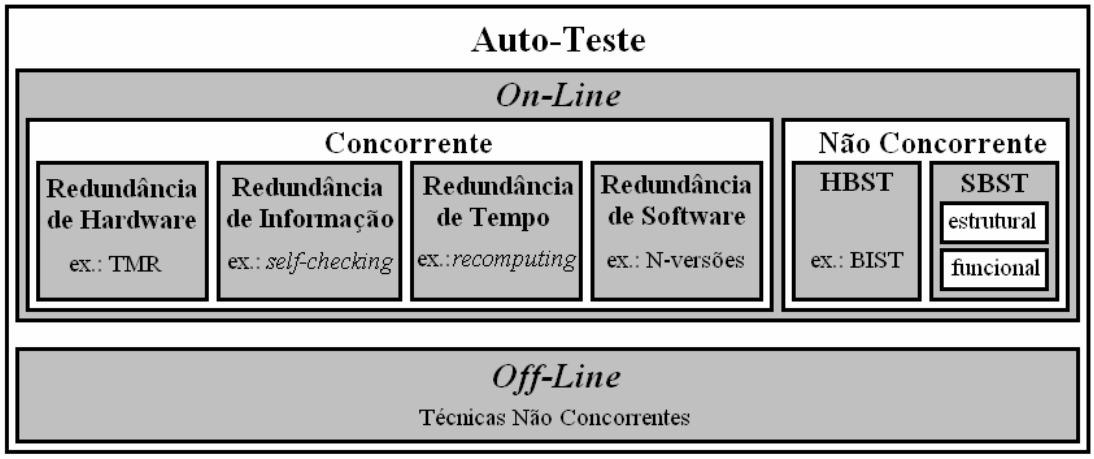

Figura 3.1: Diagrama de classificação dos métodos de auto-teste

3.1.1 Auto-Teste *On-Line* Concorrente

Nos mecanismos de teste *on-line* concorrente, a execução do teste ocorre em paralelo com a execução da aplicação, ou seja, o sistema mantém-se operando normalmente durante o teste. Técnicas concorrentes visam à detecção de erros, uma vez que verificam constantemente os resultados da aplicação que está sendo executada pelo CUT. A detecção de erros independe do tipo de falha operacional (permanente, intermitente ou transiente) que os originou. Por serem executadas em paralelo com a aplicação, essas técnicas têm baixa latência de detecção de erro, o que dificulta ou impede sua propagação. Essa característica dos métodos concorrentes torna-os bastante adequados para sistemas críticos tolerantes a falhas.

As estratégias de teste *on-line* concorrente podem ser classificadas em quatro categorias: redundância de hardware, redundância de informação, redundância de tempo e redundância de software (PASCHALIS; GIZOPOULOS, 2005).

Duplicação com comparação e *TMR* (*Triple Modular Redundancy*) (PRASAD, 1989) são exemplos de mecanismos de auto-teste *on-line* concorrente baseados em *redundância de hardware*. Na primeira abordagem, o circuito em teste é duplicado e um bloco comparador verifica se as respostas fornecidas por ambas as cópias, a cada ciclo de relógio, são idênticas, indicando a presença ou não de um erro. Em caso de divergência não é possível fornecer, apenas com essa técnica, a resposta correta. Com o TMR isso é possível graças à triplicação do CUT e à presença de um bloco votador, que fornece como resposta correta aquela dada por no mínimo duas das três réplicas. Uma alternativa para economia de área consiste na replicação apenas dos blocos mais críticos, ao invés do circuito todo. Nesse caso, é necessário um comparador/votador para cada circuito replicado. Ainda assim, a área adicionada pelo conjunto de

comparadores/votadores é compensada pela área economizada com o tamanho reduzido das réplicas.

As abordagens de teste concorrente que utilizam *redundância de informação* incluem, além de diversos esquemas de codificação, a técnica *self-checking* que, baseada em códigos de detecção ou correção de erros, verifica continuamente as respostas fornecidas pelo circuito em teste (PFLANZ; VIERHAUS, 2001). A idéia chave dessa técnica é comparar a codificação do dado de saída de determinado módulo do CUT com o código esperado. Em módulos de armazenamento, como memórias e registradores, o código esperado é calculado na entrada do módulo e armazenado junto com o dado. Em módulos que operam sobre dados, como somadores, multiplicadores e outros, o código esperado é pré-calculado a partir dos dados de entrada. Dentre os códigos mais utilizados destacam-se o bit de paridade, o código Hamming, os códigos residuais e o código Berger (ABRAMOVICI; BREUER; FRIEDMAN, 1994).

Tanto as estratégias que implementam redundância de informação quanto aquelas que usam redundância de hardware têm baixa latência de detecção de erro, mas implicam grande incremento de hardware (NICOLAIDIS; ZORIAN, 1998). A baixa latência de detecção de erro se deve à verificação ciclo a ciclo das saídas fornecidas. Mas para isso, há o incremento de hardware causado tanto pelos blocos comparadores e votadores, quanto pela lógica adicionada para geração, armazenamento e comparação dos códigos de detecção de erros. É importante lembrar que, em ambas as abordagens, a lógica adicionada para o auto-teste também deve ser testada de modo a garantir a total confiabilidade do circuito.

As técnicas que utilizam *redundância de tempo* baseiam-se em recomputação, a exemplo da *pseudo-duplicação* (KIM; TAKAHASHI; HA, 1998). Nessa abordagem, os dados são processados duas vezes, pelo mesmo circuito, mas através de caminhos diferentes. Por fim, estratégias como *N-versões* e *monitoramento de assinatura* (OH; MCCLUSKEY, 2002) são exemplos do uso de *redundância de software*. Tais estratégias utilizam diferentes implementações de software para a execução de uma mesma tarefa. Embora essas duas categorias de teste *on-line* concorrente não impliquem grande incremento de área, elas apresentam maior latência de detecção de erro (quando comparadas às redundâncias de hardware e de informação) e causam uma grande redução de desempenho.

3.1.2 Auto-Teste *On-Line* Não Concorrente

Na outra categoria de auto-teste *on-line* estão as estratégias não concorrentes. Tais estratégias visam à detecção de falhas e, por isso, podem fornecer uma medida de efetividade do teste. Na abordagem não concorrente, a execução da aplicação é suspensa durante a realização do teste, seja pela mudança de modo de operação normal para modo teste, seja pela substituição do programa em execução pelo programa de teste.

Por não serem executadas em paralelo com a aplicação, as estratégias não concorrentes requerem testes periódicos (motivo pelo qual esse tipo de auto-teste também é chamado *on-line* periódico), ou que estes sejam realizados quando o sistema está ocioso. Desse modo, falhas intermitentes e transientes somente são detectadas se estiverem ativas durante a execução do teste. Mesmo as falhas permanentes não são detectadas assim que ocorrem, mas apenas quando o teste for realizado. Essa característica implica o aumento da latência de detecção de falha, possibilitando a ocorrência e propagação de erros e, eventualmente, o mal-funcionamento do sistema.

Logo, tais mecanismos são mais indicados para sistemas pouco críticos ou que implementam, adicionalmente, alguma técnica concorrente em blocos mais críticos.

Nas estratégias de auto-teste *on-line* não concorrente, os padrões de teste podem ser gerados previamente ou durante a execução do teste (GIZOPOULOS; PASCHALIS; ZORIAN, 2004). Na primeira abordagem, padrões previamente gerados são armazenados em uma área de memória reservada no *chip* (RAM – *Random Access Memory*, ou ROM – *Read-Only Memory*) e aplicados durante o teste (abordagem essa denominada *auto-teste com padrões armazenados*). Na segunda abordagem, os padrões de teste são gerados durante a execução do teste por hardware dedicado ou por rotinas de software específicas (a essa abordagem dá-se o nome *auto-teste com padrões gerados on-chip*). Além disso, as respostas de teste capturadas podem ser tanto armazenadas em áreas de memória reservadas, quanto compactadas de modo a gerar uma ou mais assinaturas de teste. Neste último caso, que visa reduzir o tamanho de memória necessário, circuitos ou rotinas de software dedicados são utilizados para a compactação das respostas de teste e geração das assinaturas.

Nas técnicas que implementam auto-teste *on-line* não concorrente, a análise que verifica se o circuito possui ou não possui falhas pode ser realizada tanto dentro do *chip* quanto fora dele. No caso extremo em que as comparações com as respostas de teste esperadas são feitas internamente, um único bit de erro é enviado para fora do circuito, indicando a presença ou não de falhas. O extremo oposto é o caso em que todas as respostas de teste são extraídas do *chip* para avaliação externa (sem compactação). No caso médio (mais comum), algumas poucas assinaturas de teste são coletadas do *chip* e, ao final da execução do auto-teste, avaliadas externamente.

Os mecanismos de auto-teste não concorrente podem ser baseados em hardware ou em software, de acordo com o meio usado para geração e aplicação dos padrões de teste e para captura e compactação das respostas de teste. Tais possibilidades são analisadas a seguir.

3.1.2.1 Auto-Teste Baseado em Hardware

Também denominado HBST (*Hardware-Based Self-Test*) (GIZOPOULOS; PASCHALIS; ZORIAN, 2004), o auto-teste baseado em hardware é ativado pela mudança do modo de operação do CUT, do modo normal para modo teste. Nesse momento, a execução da aplicação é suspensa e padrões de teste são gerados *on-chip* por circuitos específicos, denominados TPGs – *Test Pattern Generators* (no caso de auto-teste com padrões gerados *on-chip*), ou, alternativamente, padrões previamente gerados são buscados na memória (auto-teste com padrões armazenados). Após a aplicação dos padrões ao CUT, as respostas de teste obtidas podem ser individualmente armazenadas em áreas reservadas de memória, ou compactadas de modo a gerarem assinaturas de teste. Em caso de teste de processadores, alguns componentes do próprio processador (somadores, subtratores, deslocadores, multiplicadores, etc.) podem ser reusados para a geração dos padrões e compactação das respostas de teste.

As estratégias de HBST podem ser baseadas tanto em padrões de teste *determinísticos*, quanto em padrões *pseudo-aleatórios*. Em geral, no caso determinístico, o número total de padrões de teste é algumas ordens de grandeza menor, mas a geração *on-chip* de padrões pseudo-aleatórios é mais fácil e realizada por circuitos menores em comparação com o caso determinístico (GIZOPOULOS; PASCHALIS; ZORIAN, 2004). Por esse motivo, padrões de teste determinísticos são

mais adequados para auto-teste com padrões armazenados, e podem ser tanto gerados por ferramentas ATPG, quanto previamente computados ou conhecidos para módulos específicos do circuito em teste.

Diversos tipos de circuitos aritméticos, por exemplo, podem ser amplamente testados com pequenos conjuntos de padrões previamente conhecidos. Por outro lado, técnicas de auto-teste pseudo-aleatório usam geradores de padrões pseudo-aleatórios *on-chip*, como um LFSR (*Linear Feedback Shift Register*), ou um CA (*Cellular Automata*) (ABRAMOVICI; BREUER; FRIEDMAN, 1994). Uma combinação das duas abordagens (alguns padrões de teste determinísticos e outros pseudo-aleatórios) constitui uma estratégia bastante utilizada para aumentar a eficiência do auto-teste baseado em hardware.

O maior exemplo de auto-teste baseado em hardware, o qual tem se tornado prática comum na indústria de circuitos integrados, é o BIST (*Built-In Self-Test*) (BUSHNELL; AGRAWAL, 2000). Essa técnica utiliza geradores de padrões de teste e analisadores de resposta *on-chip*. Durante o teste, os padrões gerados pelo TPG são inseridos no CUT e as respostas fornecidas são compactadas e armazenadas para posterior comparação. Em geral, LFSRs são usados para a geração dos padrões de teste e MISRs (*Multiple-Input Shift Registers*) fazem a compactação das respostas de teste, gerando as assinaturas de teste. Uma das principais dificuldades no uso de BIST é encontrar a melhor solução para o número e a distribuição dos TPGs e dos analisadores de resposta, de modo a atender as restrições do sistema em termos de área, consumo de energia, tempo de teste e cobertura de falhas.

Embora o HBST tenha mostrado sucesso como uma tecnologia de teste para diferentes tipos de circuitos digitais, de pequeno e médio porte, ele não é considerado a estratégia ideal e definitiva para o teste de todos os tipos de arquiteturas. A seguir, são apresentados alguns fatores que devem ser levados em consideração pelo projetista de teste quando o auto-teste baseado em hardware é a metodologia de teste pretendida (GIZOPOULOS; PASCHALIS; ZORIAN, 2004):

- *Incremento de hardware*: é a quantidade ou percentual de hardware adicionado ao circuito, devido ao HBST, que é aceitável para um projeto particular. Uma dificuldade usualmente encontrada pelo projetista de teste e que diz respeito a esse fator é a escolha pelo uso de padrões armazenados ou padrões gerados *on-chip*. A decisão final depende fortemente do número de padrões de teste necessário. Se o conjunto de teste consiste de poucos padrões, eles podem ser armazenados numa pequena memória interna, reduzindo, assim, o incremento de hardware. Caso contrário, uma solução mais efetiva consiste no uso de um pequeno e eficiente TPG, o qual implica um incremento de hardware muito menor, quando comparado à alternativa anterior.

- *Degradação de desempenho*: é o impacto em termos de desempenho que a metodologia de auto-teste pode causar no circuito. Em muitos casos, circuitos otimizados para alto desempenho podem não admitir quaisquer modificações em seus caminhos críticos. Nesses casos, o auto-teste baseado em hardware deve ser cuidadosamente aplicado de modo a respeitar esse requisito e implicar um impacto mínimo (ou nulo) no desempenho do circuito durante sua operação normal. Entretanto, esse é um grande desafio uma vez que os caminhos críticos usualmente contêm partes difíceis de testar, as quais necessitam de modificações para aumentar sua testabilidade (DFT).

- *Dissipação de potência*: é a potência adicional que é permitida ao circuito dissipar devido ao acréscimo de hardware para o teste. A dissipação de potência é um fator crítico em sistemas nos quais o custo para inclusão de mecanismos eficientes de resfriamento e dissipação de calor é muito alto.
- *Consumo de energia*: é a energia adicional que o circuito pode consumir durante o teste. É um fator crítico quando o circuito é auto-testado em um modo de operação diferente do normal, uma vez que caminhos não ativados no modo de operação normal do circuito são geralmente ativados durante períodos de teste. O consumo de energia é um problema sério em sistemas operados por bateria quando os *chips* são testados em campo usando os mecanismos de auto-teste existentes. Nesses casos, o consumo excessivo de energia para o auto-teste reduz o tempo de vida efetivo da bateria do sistema.

As limitações apresentadas acima levaram ao desenvolvimento de uma metodologia de auto-teste de baixo custo, mas também de alta qualidade. Tal metodologia é apresentada na próxima seção e tratada com maiores detalhes no Capítulo 4, uma vez que constitui a base do presente trabalho.

3.1.2.2 Auto-Teste Baseado em Software

Há situações em que o incremento de hardware e a degradação de desempenho causados pelo uso de técnicas de auto-teste baseado em hardware são inadmissíveis e vão além das restrições de projeto. Tais situações motivaram o desenvolvimento de uma abordagem de auto-teste alternativa, e com grande potencial para aplicação em sistemas embarcados. O auto-teste baseado em software (SBST - *Software-Based Self-Test*), também chamado de auto-teste baseado em processador, pode ser usado no teste de componentes de um processador, do processador como um todo, ou de diferentes núcleos (*cores*) de um SoC (*System-on-Chip*) baseado em processador (KEATING; BRICAUD, 2002).

No SBST, as tarefas de geração e aplicação dos padrões de teste, e captura e compactação das respostas de teste são realizadas por rotinas de software executadas pelo próprio processador, ao invés de serem designadas a módulos de hardware dedicados, como no HBST. Portanto, processadores podem ser reusados como uma infra-estrutura de teste já existente tanto para testes de produção, quanto para testes *on-line* periódicos. O auto-teste baseado em software é uma abordagem naturalmente não intrusiva uma vez que o próprio processador controla o fluxo dos dados de teste em seu interior por meio do seu conjunto de instruções, sem a necessidade de hardware adicional.

No SBST, o processador executa uma rotina ou conjunto de rotinas de software previamente desenvolvidas e armazenadas na memória de programa. Assim como no HBST, os padrões de teste podem ser gerados *on-line* pelas rotinas de software ou gerados previamente e armazenados na memória do sistema. Na seqüência, o processador (por meio das rotinas de auto-teste) aplica cada um dos padrões no componente em teste, coleta as respostas fornecidas pelo componente e armazena-as na memória de dados. De modo similar à abordagem anterior, as respostas de teste podem ser armazenadas individualmente ou compactadas de modo a comporem assinaturas de teste.

A abordagem de auto-teste baseado em software classifica-se em funcional e estrutural. Tal abordagem será melhor discutida no Capítulo 4, incluindo os detalhes dessa classificação.

Conforme visto anteriormente, mecanismos de auto-teste também podem ser utilizados para execução de teste *off-line*, por exemplo, nos testes de produção. Nesses casos, só podem ser usadas estratégias não concorrentes. A explicação para isso é simples. Estratégias concorrentes constituem, basicamente, teste funcional, enquanto testes de produção devem ser estruturais, como os métodos não concorrentes. Além disso, técnicas de auto-teste concorrente são baseadas em erros, o que requer a execução de uma aplicação. Como nos testes *off-line* não há aplicação sendo executada, as técnicas concorrentes não podem ser aplicadas. Por outro lado, técnicas de auto-teste não concorrente, como BIST e SBST, visam à detecção de falhas e são independentes da aplicação. Por esse motivo, são métodos eficientes tanto para teste *on-line*, quanto para teste *off-line*.

3.2 Teste *On-Line* de Processadores

Até o presente momento, falou-se de teste de sistemas digitais em termos genéricos, sem a especificação do tipo de circuito visado. Uma vez que o foco deste trabalho é o teste *on-line* de processadores embarcados, esta seção pretende dar uma visão geral do teste *on-line* de processadores, com ênfase em aplicações para sistemas embarcados e sistemas de tempo-real.

É importante salientar que embora sejam circuitos digitais e, por isso, possam ser testados com quaisquer dos métodos já apresentados, processadores apresentam algumas características particulares que favorecem a aplicação de determinado método de teste *on-line*. Por outro lado, quando são partes integrantes de sistemas embarcados e/ou de tempo-real, a aplicação de determinadas abordagens de teste torna-se impraticável devido às sérias restrições desses sistemas. Tais características e restrições são abordadas na seqüência.

Processadores têm representado, há bastante tempo, um papel importante no desenvolvimento de circuitos digitais sendo, constantemente, os elementos centrais em todos os tipos de aplicações. Hoje, os processadores são ainda mais importantes devido ao seu crescente uso em sistemas embarcados. Em arquiteturas SoC, os processadores são, em geral, os circuitos responsáveis pela execução dos algoritmos mais críticos e pela coordenação da comunicação entre os diversos núcleos do sistema. Em alguns casos, eles são também responsáveis pela execução de auto-teste, depuração e diagnóstico de todo o sistema.

Como conseqüência, a criticidade e a importância do teste do processador, tanto na produção como em campo, é equivalente à criticidade e à importância da sua própria existência em um sistema ou em um SoC (GIZOPOULOS; PASCHALIS; ZORIAN, 2004). Quando uma falha aparece em um processador, por exemplo, em um dos seus registradores, então todos os programas que utilizam esse registrador específico (talvez todos os programas a serem executados) irão apresentar mal-funcionamento. Embora a falha exista apenas dentro do processador, é muito provável que o sistema inteiro fique completamente inutilizável porque toda a funcionalidade executada pelo processador irá fornecer saídas incorretas.

Outros componentes de um sistema ou núcleos de um SoC não são tão críticos quanto um processador em relação ao correto funcionamento do sistema. Por exemplo, se uma palavra de memória contém uma falha, apenas escritas e leituras naquele local específico serão errôneos, ou seja, apenas alguns poucos programas (que utilizam a palavra de memória com falha) irão apresentar mal-funcionamento. Embora cause um comportamento incorreto, uma falha no módulo de memória pode não produzir resultados tão catastróficos no sistema quanto uma falha no processador. O mesmo raciocínio é válido para outros componentes, como o controlador de um dispositivo periférico, por exemplo. Se uma falha ocorre em tal controlador, então o sistema pode ter problemas de acesso ao dispositivo em questão, mas manterá todas as outras funcionalidades corretas.

O teste de processadores é uma tarefa essencial porque se deve garantir que o processador esteja livre de falhas para que ele possa ser usado como veículo de teste para os demais módulos do sistema. Ambos os métodos de auto-teste *on-line* não concorrente, tanto baseado em hardware quanto baseado em software, podem utilizar estruturas do processador para o seu próprio teste, bem como para o teste dos componentes que o cercam. E, embora a importância de todos os tipos, tamanhos e arquiteturas de processadores seja a mesma para propósitos de teste, isso não significa que todos serão testados com a mesma dificuldade.

3.2.1 Classes de Teste *On-Line*

Desde o aparecimento do primeiro microprocessador, pesquisas intensas têm sido feitas no campo de teste de processadores. Uma variedade de metodologias genéricas, bem como diversas soluções *ad hoc* podem ser encontradas na literatura (COURTOIS, 1981; HENSHAW, 1986; PFLANZ; VIERHAUS, 2001; XENOULIS et al., 2003). Cada abordagem tem diferentes objetivos e limitações, dependendo da aplicação para a qual o processador é utilizado e das restrições que devem ser respeitadas. Algumas das técnicas propostas, que foram desenvolvidas tendo como única finalidade o teste de produção, hoje são vistas como soluções também para teste *on-line* (TIMOC et al., 1983; KARPOVSKY; VAN METER, 1984; ADHAM; GUPTA, 1996; RADECKA; RAJSKI; TYSZER, 1997).

Assim como para qualquer outro sistema digital, não existe uma solução ótima única de teste *on-line* para as diferentes arquiteturas de processadores. Em muitos casos, mais de uma estratégia devem ser combinadas para prover a solução mais eficiente e adequada para uma dada configuração de sistema. A estratégia de teste a ser aplicada em um processador depende da sua arquitetura e conjunto de instruções específicos, das características do SoC ao qual ele pertence, e das limitações de projeto do sistema e dos custos do teste.

A separação de uma variedade de técnicas de teste *on-line* de processadores em classes com diferentes objetivos provê um método sistemático para a seleção da técnica apropriada para determinada configuração de sistema. Isso pode ser feito através da combinação dos benefícios de cada classe com os requisitos da aplicação. Com esse objetivo, esta seção irá discutir as diferentes classes às quais uma metodologia de teste *on-line* de processadores pode pertencer (GIZOPOULOS; PASCHALIS; ZORIAN, 2004). Suas características serão apresentadas, bem como as vantagens e desvantagens da sua aplicação.

3.2.1.1 Teste Baseado em DFT vs. Teste Não Intrusivo

Técnicas de DFT (*ad hoc*, cadeias *scan*, BIST e outras) podem ser utilizadas tanto para aumentar a testabilidade do processador quanto para possibilitar auto-teste. O BIST, por exemplo, é uma metodologia clássica de auto-teste baseada na geração de padrões de teste pseudo-aleatórios. Sua aplicação em processadores é facilitada pelo reuso de estruturas do processador, reduzindo, assim, o incremento de hardware. Já o teste baseado em *scan* ou em outras técnicas estruturadas de DFT (como a inserção de pontos de teste) requer grandes modificações de projeto. No caso de processadores, tais técnicas não são muito indicadas simplesmente porque os projetistas são bastante relutantes em adotar alterações significantes no projeto devidas ao teste. Tais alterações têm, em geral, grande impacto na área, no consumo de energia e no desempenho do processador.

Por outro lado, técnicas não intrusivas de teste *on-line* não requerem quaisquer modificações no projeto do processador. Uma vez que não implicam incremento de hardware, de dissipação de potência e nem queda de desempenho, o uso dessas técnicas é mais facilmente aceito pelos projetistas. A principal questão relacionada às técnicas de teste não intrusivas é se elas são capazes de alcançar uma cobertura de falhas suficiente, sendo que elas se restringem a falhas que podem ser detectadas durante a operação normal do processador (algumas vezes chamadas de *falhas funcionalmente detectáveis*).

3.2.1.2 Teste Funcional vs. Teste Estrutural

Os conceitos de teste funcional e teste estrutural já foram discutidos no Capítulo 2 (Seção 2.5). Aqui, vale relembrar suas particularidades quando aplicados a processadores. O teste funcional tem por objetivo verificar se todas as operações do processador estão sendo executadas corretamente e no tempo previsto. Em outras palavras, ele visa assegurar que todas as instruções definidas no ISA seguem as especificações, tanto em termos de funcionalidade, quanto em termos de atraso. Por isso, o teste funcional tem grande relação com a verificação do projeto.

Em contrapartida, o teste estrutural está estreitamente relacionado à precisão do processo de fabricação. Ele pretende verificar se a implementação física do processador condiz com o seu esquemático. Desse modo, a obtenção de uma alta cobertura de falhas no teste estrutural depende da disponibilidade de um modelo lógico do processador, ao contrário do teste funcional, cuja testabilidade estrutural não é, em geral, muito alta, pois depende fortemente do conjunto de operandos utilizado no teste. Na maioria das vezes, operandos pseudo-aleatórios são usados, o que implica um tempo de execução de teste muito alto, e uma baixa cobertura de falhas.

3.2.1.3 Teste para Falhas Combinacionais vs. Teste para Falhas Seqüenciais

Falhas combinacionais alteram o comportamento do circuito de modo que partes combinacionais desse circuito continuam comportando-se como combinacionais, mas com uma função diferente da correta. Já as falhas seqüenciais fazem com que partes combinacionais do circuito comportem-se como seqüenciais, ou seja, saídas dependem tanto das entradas atuais quanto de entradas anteriores (GIZOPOULOS; PASCHALIS; ZORIAN, 2004). O teste de processadores pode focar a detecção de falhas que pertencem tanto a modelos combinacionais, como o padrão industrial *stuck-at* simples, quanto a modelos seqüenciais, isto é, modelos cujas falhas levam a um comportamento seqüencial e que requerem dois vetores de teste para a detecção de uma única falha.

Modelos de falhas de atraso, como falhas de atraso de caminho, são os modelos de falhas seqüenciais mais utilizados.

O tempo de geração de padrões de teste para modelos de falhas seqüenciais é muito maior do que para falhas combinacionais. Por isso, as falhas de atraso visadas devem ser adequadamente selecionadas no intuito de reduzir o tempo de ATPG. Ferramentas de EDA (*Electronic Design Automation*) para modelos de falhas combinacionais (principalmente para o modelo de falhas *stuck-at*) têm sido usadas há bastante tempo. Por outro lado, ferramentas para modelos de falhas seqüenciais são menos maduras, mas estão em constante aperfeiçoamento em termos de desempenho e de eficiência.

Ambos os testes, para falhas combinacionais e falhas seqüenciais, pertencem à classe de teste estrutural uma vez que ambos requerem um modelo lógico do processador para a geração do teste e cálculo da cobertura de falhas. Obviamente, estratégias de teste de processadores que visam à detecção de falhas seqüenciais oferecem maior cobertura de defeitos e maior qualidade de teste, mas precisam ser executadas *at-speed*. Por outro lado, métodos de teste para falhas combinacionais, como falhas *stuck-at*, requerem muito menos padrões de teste e programas de teste com tempo de execução muito menor do que o teste para falhas de atraso ou outras falhas seqüenciais. Além disso não requerem teste *at-speed*.

3.2.1.4 Teste Pseudo-Aleatório vs. Teste Determinístico

A cobertura de falhas que uma técnica para teste de processador pode atingir depende do número, tipo e natureza dos padrões de teste aplicados. O teste pseudo-aleatório para processadores pode ser baseado em *seqüências de instruções pseudo-aleatórias*, *operandos pseudo-aleatórios* ou uma *combinação* dos dois (BECK et al., 2005; CORNO et al., 2001). Como em qualquer outro circuito, o teste pseudo-aleatório em processadores tem a desvantagem de requerer seqüências de teste excessivamente longas para alcançar uma cobertura de falhas aceitável. Isso é particularmente verdadeiro para alguns componentes do processador que são resistentes a padrões aleatórios. Seqüências de instruções aleatórias são geralmente muito improváveis de atingir uma alta cobertura de falhas.

A despeito dessas dificuldades, o teste pseudo-aleatório de processadores tem sido extensivamente estudado e aplicado porque é uma metodologia simples que requer mínimo esforço de engenharia: nenhum algoritmo ou ferramenta especial para a geração do teste é necessário. Além disso, o desenvolvimento de seqüências ou programas de teste pseudo-aleatórios não requer um modelo do processador em nível lógico, embora o cálculo da cobertura de falhas dependa da existência de tal modelo. Um dos geradores pseudo-aleatórios mais utilizados é o LFSR (*Linear Feedback Shift Register*), o qual determina a seqüência de padrões com base em um *polinômio característico*, que descreve as conexões dos seus elementos de memória, e um valor inicial, denominado *semente*. A seleção de uma semente e de um polinômio característico apropriados pode levar a uma redução significante no número de padrões de teste necessários para a obtenção da cobertura de falhas desejada. Porém, tal escolha consome, em geral, grande quantidade de tempo.

Em contrapartida, o teste determinístico para processadores pode ser baseado tanto em seqüências de teste geradas por ATPG, quanto por padrões de teste previamente calculados para o processador ou seus componentes. Por exemplo, para maioria dos módulos funcionais de um processador (como ULAs, multiplicadores, divisores,

deslocadores, etc.) existem conjuntos de teste previamente e cuidadosamente desenvolvidos que garantem alta cobertura de falhas. Tais conjuntos podem ser aplicados aos componentes do processador através de suas instruções. Uma alta cobertura de falhas (tanto para modelos de falhas combinacionais quanto seqüenciais) pode ser também alcançada com uma boa ferramenta ATPG, desde que o *netlist* do processador esteja disponível. O sucesso dessa estratégia depende da qualidade da ferramenta EDA (o ATPG) adotada. Portanto, a grande vantagem do teste determinístico (pré-computado ou baseado em ATPG) é a alta cobertura de falhas obtida com pequenas seqüências de teste.

A combinação do teste pseudo-aleatório com teste determinístico é também uma situação bastante comum. Componentes do processador que podem ser eficientemente testados com seqüências pseudo-aleatórias o são, enquanto para os outros se pode utilizar tanto padrões pré-calculados quanto conjuntos de teste gerados por ATPG. Uma prática também comum para aperfeiçoar o teste pseudo-aleatório é a inclusão de alguns poucos padrões determinísticos nas seqüências pseudo-aleatórias, reduzindo, assim, o tamanho das seqüências e melhorando a capacidade de detecção. Uma observação final sobre o teste pseudo-aleatório é que ele pode ser reusável ou programável, no sentido em que um único gerador (um LFSR, por exemplo) pode ser alimentado com diferentes sementes ou mesmo reconfigurado para um diferente polinômio e, desse modo, ele pode ser usado para testar partes distintas do processador (GIZOPOULOS; PASCHALIS; ZORIAN, 2004).

3.2.1.5 Teste de Microprocessador vs. Teste de DSP

Elementos de processamento embarcados podem aparecer tanto na forma de arquiteturas clássicas de Von Neumann, com uma única memória para armazenamento de instruções e de dados, quanto na forma de arquiteturas Harvard, com memórias separadas para dados e instruções. Arquiteturas Harvard são amplamente utilizadas em processadores digitais de sinais (DSPs – *Digital Signal Processors*), onde a manipulação de dados e a largura de banda de transferência de dados é maior do que em processadores de propósito geral. Por outro lado, processadores de propósito geral usualmente empregam mecanismos para aperfeiçoamento de desempenho (como predição de salto), os quais não são comumente usados em DSPs devido à natureza probabilística desses mecanismos.

Microprocessadores e processadores embarcados de propósito geral têm, usualmente, estruturas de controle complexas em comparação com DSPs, os quais têm, em geral, unidade de controle simples mas módulos de processamento de dados mais complexos e largos. Finalmente, ambos os tipos de arquitetura podem incluir estruturas de pipeline relativamente simples ou mais complexas, para aumento de desempenho. Embora a classificação em teste de microprocessador ou teste de DSP não pertença à estratégia de teste em si, é importante que todas essas diferenças sejam levadas em consideração na escolha do método de teste mais adequado para cada tipo de arquitetura.

A Tabela 3.1 resume as vantagens e desvantagem de cada uma das classes de teste apresentadas. Com base nas informações contidas nessa tabela, nas características e objetivos dos diversos métodos de teste on-line para processadores (GIZOPOULOS; PASCHALIS; ZORIAN, 2004. p.64-79) e nos requisitos e limitações da aplicação alvo,

discutidos nas próximas seções, pode-se definir a estratégia de teste mais adequada. Assim, faz-se importante, a partir desse ponto, o conhecimento das propriedades e dos objetivos da técnica de auto-teste baseado em software (SBST), uma vez que ela constitui a base da metodologia desenvolvida neste trabalho. Os motivos que levaram à escolha dessa estratégia de teste são discutidos ao longo das próximas seções, e um estudo detalhado da técnica é apresentado no Capítulo 4.

Tabela 3.1: Vantagens e desvantagens das classes de teste

	Vantagens	***Desvantagens***
Teste Baseado em DFT	• Alta cobertura de falhas mais facilmente obtida • Uso extensivo de ferramentas de EDA	• Considerável incremento de hardware e de consumo de energia • Considerável queda de desempenho
Teste Não Intrusivo	• Nenhum incremento de hardware e de consumo de energia • Nenhuma queda de desempenho	• Uso limitado de ferramentas de EDA • Alta cobertura de falhas mais dificilmente obtida
Teste Funcional	• Sem necessidade de detalhes baixo-nível • Reuso de padrões de verificação funcional • Baixo custo de desenvolvimento do teste	• Sem relação com falhas estruturais • Baixa cobertura de defeitos • Operandos pseudo-aleatórios • Longas seqüências de teste • Longos programas de teste
Teste Estrutural	• Uso de ferramentas de EDA • Alta cobertura de falhas • Pequenas seqüências de teste • Rápidos programas de teste	• Necessidade do modelo lógico do processador • Mais alto custo de desenvolvimento do teste
Teste para Falhas Combinacionais	• Pequenos conjuntos ou programas de teste • Curto tempo de aplicação do teste • Curto tempo de geração do teste • Ferramentas de EDA consolidadas	• Necessidade do modelo lógico do processador • Menor cobertura de defeitos
Teste para Falhas Seqüenciais	• Mais alta qualidade do teste • Maior cobertura de defeitos	• Grandes conjuntos ou programas de teste • Longo tempo de aplicação do teste • Necessidade do modelo lógico do processador • Longo tempo de geração do teste • Ferramentas de EDA menos consolidadas
Teste Pseudo-Aleatório	• Fácil desenvolvimento das seqüências de teste • Sem necessidade de detalhes em nível lógico	• Longas seqüências de teste • Alta cobertura de falhas mais dificilmente obtida
Teste Determinístico	• Alta cobertura de falhas mais facilmente obtida • Pequenas seqüências de teste	• Necessidade de detalhes em nível lógico • Necessidade de ATPG para obtenção de resultados satisfatórios

Fonte: GIZOPOULOS; PASCHALIS; ZORIAN, 2004. p.57-62

O SBST é um método de teste não intrusivo porque não implica qualquer modificação no projeto do circuito para realização do teste. Ele pode realizar tanto teste

funcional quanto teste estrutural, dependendo das estratégias utilizadas para geração das instruções e dos dados de teste. Tais estratégias podem ser pseudo-aleatórias ou determinísticas. Em caso de teste estrutural, o SBST pode ser desenvolvido visando o teste de falhas combinacionais ou seqüenciais (de acordo com o modelo de falhas escolhido), desde que sejam funcionalmente detectáveis. Por fim, o auto-teste baseado em software pode ser aplicado em microprocessadores ou em DSPs, tanto para o teste do próprio processador quanto para o teste dos componentes ao seu redor.

3.2.2 Teste *On-Line* de Processadores em Sistemas Embarcados

O auto-teste *on-line* de um processador embarcado antes que ele possa ser utilizado para execução do teste de outros componentes do sistema é uma tarefa bastante desafiadora. Diversas técnicas de projeto são aplicadas aos componentes do processador com o principal objetivo de atingir o melhor desempenho possível sob limitações adicionais de projeto, como tamanho do circuito, dissipação de potência, entre outras. Por exemplo, em muitos casos, o desempenho máximo do processador é buscado sob a restrição de que o tamanho do circuito não exceda um limite especificado (provavelmente impostos pelas limitações e custos de encapsulamento do *chip*). Em outros casos, o fator que limita o desempenho alcançável no projeto de um processador é a potência máxima que pode ser dissipada pelo circuito. Tal limitação é usualmente dada pelo custo do resfriador e dos mecanismos de remoção de calor, ou pelo consumo de energia.

As técnicas para o auto-teste de processadores embarcados são aplicadas com sérias dificuldades e restrições devido às arquiteturas otimizadas desses processadores, as quais admitem, na melhor hipótese, apenas alterações marginais no circuito e impacto marginal no desempenho e no consumo de energia. A seguir, são apresentados e discutidos os requisitos que uma técnica de auto-teste deve satisfazer quando aplicada a um processador embarcado (GIZOPOULOS; PASCHALIS; ZORIAN, 2004). Os mesmos requisitos são válidos não apenas para o auto-teste do processador, mas também quando o processador é usado para testar os demais componentes de um SoC.

- Processadores embarcados são, em geral, projetos bastante otimizados em termos de desempenho. Assim, estratégias adequadas ao teste *on-line* de sistemas embarcados devem evitar qualquer redução nesse fator crítico. A degradação de desempenho imposta por técnicas de DFT, estruturadas ou *ad hoc*, é praticamente inaceitável na maioria dos casos. Quando métodos não concorrentes de teste *on-line* são aplicados, deve-se buscar um tempo de aplicação do teste bastante reduzido para evitar a degradação do desempenho total do sistema.
- Processadores embarcados são cuidadosamente otimizados no que diz respeito a seu tamanho e número de células lógicas de modo a ocupar a menor área de silício possível quando integrados a um SoC. Por isso, qualquer incremento de área devido ao teste deve ser evitado ou, no mínimo, detalhadamente estudado antes de ser aceito. Novamente as técnicas de DFT são contra-indicadas para o auto-teste de processadores embarcados, uma vez que implicam considerável incremento de hardware.
- Outro fator de grande peso no projeto de processadores embarcados é a potência dissipada pelo circuito, uma vez que está diretamente relacionado ao consumo de energia e ao custo dos mecanismos de resfriamento e remoção de calor.

Sabendo-se que incremento de hardware implica o aumento de potência dissipada, o uso de técnicas de DFT é mais uma vez desaconselhado.

- O consumo de energia é um parâmetro crítico em processadores embarcados projetados para aplicações *low-power*, usadas em dispositivos portáteis operados por bateria. Assim, além da preocupação com a redução da potência dissipada, é importante que o tempo de execução do teste, em métodos não concorrentes, seja o menor possível, de modo que a energia consumida pelo teste não tenha um efeito significante na carga disponível na bateria do sistema.
- O tamanho das memórias, tanto de programa quanto de dados, também é um fator importante no projeto de sistemas embarcados, porque tem impacto direto no tamanho e no custo do sistema. Assim, se a técnica utilizada para o teste *on-line* do processador fizer uso de programas de teste e/ou padrões de teste armazenados em memória (como no auto-teste baseado em software), estes devem ter o menor tamanho possível. Além disso, deve-se tentar reduzir o número de respostas de teste armazenadas em memória, tanto fazendo uso de mecanismos de compactação, quanto reduzindo o número de padrões de teste aplicados.
- Há também os importantes requisitos de qualidade do teste e cobertura de falhas. A cobertura de falhas modeladas alcançada com a estratégia de teste *on-line* deve ser a maior possível primeiramente para o processador embarcado, e depois para os demais núcleos do SoC. Um processador embarcado deve ser testado até um alto nível de cobertura de falhas para aumentar a confiança de que ele irá operar corretamente. Esse é um requisito fundamental porque, conforme visto anteriormente, uma falha no processador pode levar ao mal-funcionamento de quase todos os programas em execução, tornando o sistema inutilizável.
- Por fim, o custo de desenvolvimento do método de teste também deve ser considerado. Com o rápido avanço nas tecnologias de projeto e fabricação de sistemas eletrônicos, o tempo para lançamento de um produto (*time-to-market*) é cada vez mais curto. Todo o projeto de um sistema embarcado, incluindo o projeto do teste, deve ser rápido e utilizar o mínimo de recursos para reduzir custos. Logo, a automatização do projeto do teste é extremamente importante para a redução do custo de desenvolvimento de um sistema embarcado.

A partir dos requisitos discutidos acima, verifica-se que a técnica de auto-teste baseado em software é bastante adequada para aplicação em processadores embarcados. Sendo uma estratégia não intrusiva de teste *on-line*, o SBST não implica incremento de hardware, nem adiciona atraso nos caminhos críticos do circuito. Assim, a aplicação dessa técnica não tem qualquer impacto na área, na potência dissipada, e na freqüência de operação do processador. O impacto causado no desempenho total do sistema será mínimo desde que o tempo de execução do teste seja muito pequeno em relação aos processos da aplicação em execução. Essa característica é discutida na Seção 3.2.3.

Com relação ao impacto no consumo de energia devido à aplicação de SBST, este é dado principalmente pelo tempo de execução do programa de teste e pela quantidade de acessos à memória. O consumo de energia é diretamente proporcional a esse dois fatores. Obviamente, o impacto no tamanho das memórias de instruções e dados está diretamente relacionado ao tamanho do programa de teste e ao número de padrões e respostas de teste armazenados em memória, respectivamente. O Capítulo 4 apresenta

diversas soluções de programas de teste que favorecem um ou outro requisito. A escolha da solução mais adequada para dados requisitos de sistema é um dos pontos abordados na metodologia desenvolvida (Capítulo 5).

Os requisitos de qualidade de teste e cobertura de falhas na técnica SBST são dados pela seqüência de instruções e pelos dados de teste utilizados. Esses fatores também estão relacionados com o tipo de teste visado (funcional ou estrutural) e com o tipo de geração de padrões de teste (pseudo-aleatória ou determinística). É claro que a qualidade do teste e a cobertura de falhas têm impacto direto no tempo de teste e no tamanho das memórias. Assim sendo, todos esses fatores devem ser considerados no desenvolvimento do programa de teste mais adequado para dada aplicação alvo, conforme mostrado no Capítulo 5.

Finalmente, o custo de desenvolvimento da técnica SBST para dada arquitetura é determinado pela metodologia usada no desenvolvimento do programa de teste. Uma redução significativa no tempo de projeto do teste já é obtida por ser desnecessária qualquer modificação no hardware. O Capítulo 6 apresenta uma ferramenta que automatiza a metodologia de teste desenvolvida neste trabalho, para os processadores da família Femtojava, possibilitando uma grande redução no custo de desenvolvimento do teste.

3.2.3 Teste *On-Line* de Processadores em Sistemas de Tempo-Real

Na computação de tempo-real, a exatidão do sistema depende não apenas do resultado lógico da computação, mas também do tempo no qual os resultados são produzidos (STANKOVIC, 1988).

Os sistemas computacionais modernos são compostos por um conjunto de tarefas. Uma tarefa é um programa executável que constitui a unidade básica de trabalho dos sistemas de tempo compartilhado (SILBERSCHATZ, 2002). Assim, o objetivo dos sistemas de tempo-real é atender os requisitos temporais individuais de cada tarefa da aplicação. Mais do que ser rápido, a propriedade fundamental de um sistema de tempo-real é a previsibilidade, ou seja, seu comportamento funcional e temporal deve ser tão determinístico quanto o necessário para satisfazer as especificações do sistema. Computação rápida é útil para atender especificações temporais estritas mas, sozinha, não garante previsibilidade.

Enquanto especificação e verificação dizem respeito à integridade do sistema em termos de funcionalidade, a teoria do escalonamento (LIU; LAYLAND, 1973) remete ao problema do atendimento de requisitos temporais. A satisfação dos requisitos temporais de um sistema de tempo-real demanda o escalonamento dos seus recursos de acordo com algoritmos bem estabelecidos, de modo que o comportamento temporal do sistema seja compreensível e previsível. Uma medida primária de algoritmos de escalonamento é o seu nível permitido de utilização do processador, abaixo do qual os *deadlines* de todas as tarefas podem ser atendidos.

Para efeito de exemplificação, vamos considerar que uma aplicação de tempo-real é composta por um conjunto de tarefas periódicas independentes, e que o período de cada tarefa é igual ao seu *deadline*. O período de uma tarefa é o intervalo de tempo compreendido entre duas ativações consecutivas dessa tarefa, enquanto seu *deadline* indica o tempo máximo dentro do qual sua execução deve ser completada (BUTTAZZO, 2002). A taxa de utilização do processador por uma tarefa é dada pela razão entre o período (*deadline*) da tarefa, e o seu tempo de execução. O período de

cada tarefa é determinado de acordo com a especificação do sistema e o tempo de execução pode ser definido por meio de simulação ou por análise de pior caso. Assim, a soma das taxas de utilização do processador por cada tarefa da aplicação determina o nível de utilização do processador, que deve ser igual ou menor do que o nível permitido pelo algoritmo de escalonamento, de modo a garantir o atendimento de todos os *deadlines*.

A forte interação entre um sistema de tempo-real e o ambiente implica sérias limitações de tempo e confiabilidade. A menos que o sistema possa prover serviços aceitáveis para o seu ambiente, ele perderá sua função, tornado-se falho ou inexistente. Daí vem a importância do teste *on-line* para sistemas de tempo-real. Na escolha de uma técnica adequada, deve-se considerar que os métodos concorrentes de teste *on-line* não alteram a previsibilidade do sistema, uma vez que executam em paralelo com a aplicação. Por outro lado, métodos não concorrentes de teste *on-line* têm impacto direto na previsibilidade do sistema, pois implicam a suspensão periódica da execução da aplicação. Por isso, esses métodos requerem uma análise cuidadosa de sua aplicabilidade em sistemas de tempo-real. Vale salientar que não está sendo considerado o impacto dos mecanismos de tolerância a falhas (para tratamento e correção de erros), mas apenas dos métodos de detecção (teste *on-line*).

A qualidade de um teste *on-line* não concorrente está relacionada não só com a cobertura de falhas alcançada, mas também com a periodicidade do teste. Quanto menor o período de teste, menor será a latência de detecção de falhas e maiores serão as chances de tratamento e correção de erros. Sabendo-se que um método de teste não concorrente compete com a aplicação pelo uso do processador (pois suspende sua execução), ele deve ser considerado como uma tarefa a mais, e escalonado juntamente com as tarefas da aplicação de tempo-real. Assim sendo, para assegurar a qualidade do teste, reduzindo seu período, e ainda manter o atendimento dos *deadlines* de todas as tarefas da aplicação, deve-se buscar a redução do tempo de execução do teste. Com isso, pretende-se garantir que a taxa de utilização do processador pela tarefa de teste seja pequena o suficiente para manter o nível de utilização do processador abaixo do valor permitido pelo algoritmo de escalonamento.

Quando a técnica de SBST é aplicada a um sistema de tempo-real, a premissa acima reforça a necessidade de redução do tempo de execução do programa de teste, e ainda introduz um outro fator a ser analisado: o período de teste. O tempo de execução do programa de teste está intimamente relacionado ao número de padrões de teste e à estratégia de geração desses padrões. O período de teste deve ser definido com base nas tarefas de tempo-real, de modo a ser o menor valor que permita o atendimento dos *deadlines* da aplicação, para o algoritmo de escalonamento utilizado. A metodologia desenvolvida neste trabalho, e apresentada no Capítulo 5, considera ambos os fatores (tempo de execução e período de teste) na aplicação de teste *on-line* periódico para processadores embarcados em sistemas de tempo-real.

3.3 Resumo e Conclusões

Neste capítulo foram revisados os tipos de teste pelos quais os circuitos integrados passam, desde o projeto até o fim da sua vida útil. Viu-se que, primeiramente, o projeto do circuito passa pela etapa de caracterização, composta basicamente por testes funcionais que visam verificar se o projeto atende às especificações. Depois de produzidos, todos os *chips* passam pelo teste de produção que contém testes estruturais,

além da verificação de consistência de alguns parâmetros do dispositivo. Após a entrega dos circuitos ao cliente e antes de serem integrados ao sistema final, uma amostra dos *chips* passa pelo teste de aceitação, cujo objetivo é evitar que dispositivos defeituosos sejam integrados ao sistema, onde o custo de diagnóstico é muito alto.

Essas três primeiras etapas de teste de circuitos integrados são realizadas *off-line*, ou seja, quando o sistema ainda não está em operação. Porém, para sistemas críticos, é necessária a verificação contínua do correto funcionamento do circuito durante toda sua vida útil. O teste *on-line* tem por objetivo validação das operações do CI, através da detecção de falhas ou de erros com o sistema já operativo. Esse tipo de teste é necessário porque mesmo tendo sido exaustivamente testado e validado durante sua fabricação, quando em campo, o circuito está sujeito às falhas causadas pela ação do tempo e por fatores externos.

Foi visto também, que o uso de testadores externos para os testes de produção ainda é a estratégia mais utilizada pelos fabricantes de circuitos integrados. Porém, o alto custo desses equipamentos e a crescente complexidade dos *chips* vêm tornando o seu uso impraticável. Circuitos auto-testáveis são uma solução eficiente tanto para substituir os ATEs nos testes de produção quanto para realizar testes *on-line*. Esses circuitos têm a capacidade de testar a si próprios, gerando e aplicando padrões de teste e capturando e compactando as respostas de teste, utilizando apenas recursos internos. Mecanismos de auto-teste apresentam uma série de vantagens sobre os testadores externos.

Métodos de auto-teste podem ser utilizados tanto para teste *on-line*, quanto para teste *off-line*. As técnicas de auto-teste *on-line* classificam-se em concorrentes e não concorrentes. Técnicas concorrentes visam à detecção de erros e são executadas durante a operação normal do sistema, em paralelo com a aplicação. Mecanismos de auto-teste *on-line* concorrente podem ser baseados em redundância de hardware, redundância de informação, redundância de tempo ou redundância de software. Os métodos baseados em redundância de hardware e de informação têm menor latência de detecção de erro, mas implicam maior incremento de hardware. Por outro lado, técnicas que implementam redundância de tempo e de software causam pequeno impacto na área do circuito, mas causam grande impacto no desempenho e têm maior latência de detecção de erro.

Os mecanismos de auto-teste *on-line* não concorrente visam à detecção de falhas. Uma vez que o teste não é executado em paralelo com a aplicação, a execução desta deve ser suspensa durante o teste, que é realizado periodicamente. Métodos não concorrentes de auto-teste podem ser baseados em hardware ou em software. No auto-teste baseado em hardware, a aplicação dos vetores de teste e a captura das respostas é feita por circuitos específicos internos ao *chip*, implicando incremento de hardware. No auto-teste baseado em software, essas duas tarefas são realizadas por rotinas de software, o que requer a presença de um processador. Em ambas as abordagens, os padrões de teste podem ser previamente gerados e armazenados em memória, ou gerados *on-chip*. As respostas de teste podem ou não serem compactadas.

Por fim, foi demonstrado que apenas os mecanismos de auto-teste não concorrente podem ser usados também para teste *off-line*. Tanto porque eles realizam teste estrutural, quanto porque eles independem da aplicação.

Do conteúdo exposto até este ponto podem-se extrair algumas conclusões. A primeira delas é que o uso de circuitos auto-testáveis tem se mostrado uma estratégia promissora, porque, entre outras vantagens, pode substituir o uso de ATEs nos testes de

produção e ainda possibilitar o teste *on-line*. Além disso, conclui-se que a escolha da técnica de auto-teste mais adequada depende fortemente de requisitos do sistema, como confiabilidade, área, desempenho, etc.

Técnicas de auto-teste concorrente são mais adequadas para sistemas críticos em termos de confiabilidade, visto que têm baixa latência de detecção de erro. Essa característica facilita a tarefa dos mecanismos de tolerância a falhas. Por outro lado, métodos concorrentes implicam, em geral, considerável incremento de hardware e alguma degradação de desempenho. Além disso, não podem ser utilizados nos testes de produção, exigindo o uso de ATEs e aumentado o custo de produção dos CIs.

Já as técnicas de auto-teste não concorrente são indicadas para sistemas pouco críticos, uma vez que realizam testes periódicos, implicando uma considerável latência de detecção de falha. No entanto, esses métodos causam menor degradação de desempenho e ainda podem ser utilizados no teste *off-line*, dispensando a necessidade de ATEs. Dentre os métodos não concorrentes, o auto-teste baseado em hardware apresenta algumas desvantagens em relação ao auto-teste baseado em software. Entre elas incluem-se o incremento de hardware, a degradação de desempenho e o aumento no consumo de energia. Com base nesses fatores é possível concluir que, para sistemas embarcados (baseados em processadores) pouco críticos, mas com necessidade de teste periódico, o auto-teste baseado em software (SBST) é uma solução bastante adequada.

O teste *on-line* de processadores foi o foco no restante deste capítulo. Sua importância foi discutida e seus métodos foram analisados. A importância do teste de processadores vem do fato de eles serem, muitas vezes, os componentes principais do sistema que integram. Uma falha no processador pode inutilizar todo o sistema, uma vez que ele é, em geral, responsável pela execução dos algoritmos mais críticos. O mesmo não se pode dizer dos demais componentes, pois desses dependem apenas funcionalidades específicas e isoladas. Ademais, um processador pode ser responsável pelas tarefas de auto-teste, depuração e diagnóstico de si próprio, e dos componentes ao seu redor. Por isso o teste *on-line* desses circuitos é indispensável.

Viu-se também que, sendo circuitos digitais, os processadores podem ser testados com qualquer método de teste digital. Porém, sabendo-se que as diversas estratégias de teste têm características e objetivos distintos, a escolha da estratégia mais adequada para determinada aplicação deve ser baseada nas expectativas do teste e nas restrições do sistema alvo. Além disso, alguns métodos de teste são favorecidos pelo simples fato de que o CUT é um processador, possibilitando a reutilização de recursos. Os métodos de teste para processadores são classificados de acordo com seus objetivos e limitações. As diversas classes, suas características, suas vantagens e suas desvantagens foram apresentadas neste capítulo.

Técnicas de teste *on-line* podem:

- ser baseadas em DFT, quando modificam o projeto do circuito para adição de estruturas de teste; ou não intrusivas, quando se utilizam apenas das estruturas já existentes no processador
- realizar teste funcional, quando pretendem verificar se todas as instruções do processador estão sendo corretamente executadas; ou teste estrutural, quando pretendem verificar se o circuito real corresponde ao que foi projetado
- detectar falhas combinacionais, que alteram a função lógica do circuito; ou falhas seqüenciais, que alteram suas características temporais

- ser pseudo-aleatórias, quando os padrões de teste são gerados aleatoriamente; ou determinísticas, quando os padrões são gerados com base em um modelo lógico do circuito
- ser otimizadas para testar microprocessadores, ou para testar DSPs.

Quando o processador a ser testado faz parte de um sistema embarcado, a escolha da estratégia de teste deve considerar todas as restrições adicionais que esse tipo de sistema normalmente impõe. Tais restrições, discutidas neste capítulo, são: desempenho, área, potência dissipada, consumo de energia, tamanho de memória, qualidade do teste, cobertura de falhas e custo de desenvolvimento. No caso de sistemas de tempo-real, devido à necessidade de previsibilidade e confiabilidade desses sistemas, dois novos requisitos devem ser adicionados: tempo de execução e período do teste.

A estratégia SBST, usada como base para a metodologia desenvolvida neste trabalho, constitui um método não intrusivo de teste. Quanto às demais classificações, o SBST não apresenta restrições, podendo ser desenvolvido para satisfazer ambas as possibilidades de cada classe. Com isso, é possível concluir que a técnica SBST é uma solução adequada para a implantação de auto-teste *on-line* periódico em processadores embarcados. Além de evitar quaisquer modificações DFT, o auto-teste baseado em software é um método bastante flexível, possibilitando a realização de diferentes tipos de teste, a detecção de diferentes tipos de falhas, e a aplicação em diferentes arquiteturas.

A flexibilidade da técnica SBST permite, ainda, que os requisitos e as restrições de projeto sejam atendidos tanto para sistemas embarcados, quanto para sistemas de tempo real. Tal flexibilidade é possível graças às diversas estratégias para o desenvolvimento do programa de teste. Várias dessas estratégias são apresentadas no Capítulo 4, e a utilização dessas estratégias para o atendimento dos requisitos do sistema (embarcado e/ou de tempo-real) é uma das contribuições deste trabalho, proposta no Capítulo 5. Adicionalmente, uma alternativa de automatização do projeto do teste, para a redução no custo de desenvolvimento do sistema, é apresentada no Capítulo 6.

4 AUTO-TESTE BASEADO EM SOFTWARE

Auto-teste baseado em software (SBST) é uma metodologia alternativa ao auto-teste baseado em hardware (HBST) que realiza o teste do processador (ou dos componentes ao seu redor) usando suas próprias instruções (CHEN; DEY, 2001). Enquanto o HBST precisa ser aplicado em um modo de operação não funcional, o SBST pode ser aplicado no modo de operação normal do processador, sem a necessidade de quaisquer alterações de projeto nem a adição de estruturas de hardware.

O princípio básico do SBST consiste na geração de um programa de auto-teste eficiente que alcance alta cobertura de falhas, em caso de teste estrutural, ou que cubra todas as funções do sistema, em caso de teste funcional. O conceito dessa metodologia é ilustrado na Figura 4.1. O programa de auto-teste é armazenadas na memória de instruções, e os dados necessários para sua execução, bem como as respostas obtidas e esperadas, são armazenados na memória de dados (considera-se aqui uma arquitetura Harvard, na qual instruções e dados são armazenados em memórias separadas).

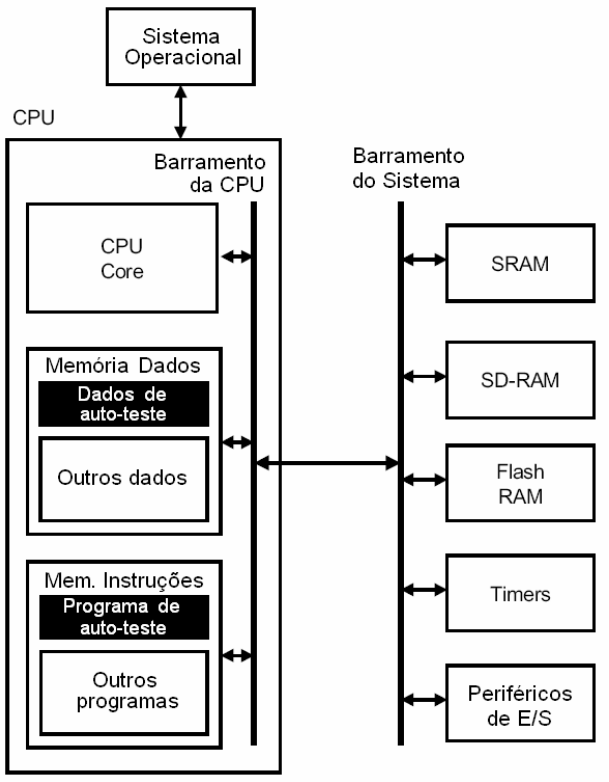

Figura 4.1: Conceito do auto-teste baseado em software
(XENOULIS et al., 2003, p.150)

O auto-teste baseado em software apresenta um série de vantagens sobre os demais métodos de teste (tanto de produção quanto *on-line*). A seguir são descritas algumas características do SBST que evidenciam tais vantagens, principalmente em relação ao HBST, e comprovam seu aspecto de baixo custo (GIZOPOULOS; PASCHALIS; ZORIAN, 2004).

- SBST não requer qualquer modificação no projeto, na estrutura, ou no conjunto de instruções do processador, uma vez que é uma abordagem não intrusiva de teste. Assim, sua aplicação evita os problemas, já mencionados, comuns em técnicas de HBST, como degradação de desempenho, aumento de área, e aumento de dissipação de potência.
- SBST é uma metodologia de baixo custo porque pode ser aplicada tanto em testes de produção quanto em testes *on-line* periódicos, sem o aumento de área, atraso, ou dissipação de potência no processador.
- SBST é uma metodologia de baixo custo porque não depende de testadores externos muito caros, para teste de produção. Um ATE de baixo custo, baixa freqüência e poucos pinos pode ser perfeitamente utilizado pelo SBST durante o teste de produção de um processador ou um SoC. Ele precisa simplesmente carregar o programa e os dados de auto-teste na memória *on-chip* do processador (se já não estiverem permanentemente armazenados em uma memória flash ou ROM), e extrair as respostas ou assinaturas do teste para avaliação externa (se necessário). Se o programa de teste for suficientemente pequeno, o tempo total de aplicação do teste será minimamente afetado pelos tempos de carga e extração realizados à baixa freqüência do testador.
- SBST realiza teste *at-speed*, uma vez que todos os padrões são aplicados na freqüência real de operação do circuito durante sua operação normal. Em testes de produção, o teste será *at-speed* mesmo que o ATE utilizado seja de baixa freqüência. Todos os defeitos físicos funcionalmente detectáveis podem ser detectados, independente de como eles alteram a funcionalidade do circuito (falhas combinacionais) ou seu comportamento temporal (falhas seqüenciais). Portanto a qualidade do teste resultante é muito alta.
- SBST é uma metodologia de teste de baixa potência porque o processador nunca é excitado por padrões de teste que jamais aparecem durante sua operação normal. O auto-teste baseado em software aplica apenas padrões de teste que podem ser propagados com instruções normais do processador, e não durante um modo especial de teste (como modo *scan*). Portanto, a potência média dissipada durante a execução do programa de auto-teste não difere da média de potência dissipada pelo processador durante sua operação normal. Se o tempo de execução do teste por período for curto o suficiente, então a potência total dissipada durante o auto-teste terá impacto mínimo na potência do *chip*. Isso é particularmente importante no auto-teste *on-line* periódico. A dissipação excessiva de potência durante períodos de teste é um sério problema em sistemas portáteis, operados por bateria, pois aumenta o consumo de energia. Esse problema é evitado com o auto-teste baseado em software.
- SBST é uma estratégia de teste bastante flexível e programável para sistemas complexos, uma vez que simples modificações no código do programa de teste são suficientes para estender a capacidade de teste (do processador e dos demais

componentes do SoC) para novos componentes adicionados ao sistema. Da mesma forma simples, pode-se alterar o modelo de falhas visado para um outro que necessite mais padrões de teste (buscando uma maior cobertura de defeitos), ou mesmo mudar o propósito do teste, para teste estrutural, teste funcional ou diagnóstico.

Todas essas características reforçam a idéia de que o auto-teste baseado em software é uma ótima estratégia para o teste *on-line* de processadores, e que se adequa perfeitamente à aplicação em sistemas embarcados (conforme os requisitos discutidos no Capítulo 3).

Quando o auto-teste baseado em software é utilizado para o teste *on-line* de processadores ou SoCs, o escalonador do sistema operacional é responsável por selecionar, periodicamente, o programa de teste para execução. Sendo executado periodicamente, o SBST possibilita a detecção de falhas permanentes e falhas intermitentes de duração considerável (em relação ao período do teste). Falhas transientes somente poderão ser detectadas se ocorrerem durante a execução do teste. Além disso, o SBST para testes periódicos é um processo que compete com os processos do usuário por recursos do sistema, ciclos do processador e memória.

Em sistemas de tempo-real, o programa de teste deve ser cuidadosamente desenvolvido de modo a não interferir no atendimento dos *deadlines* das tarefas da aplicação. No caso ideal, o tempo de aplicação do teste deve ser pequeno o suficiente para evitar trocas de contexto durante a execução do programa de teste. Conquanto esta seja uma situação muito difícil de ser alcançada, pois depende do algoritmo de escalonamento e pode requerer um tempo de teste extremamente curto, uma solução para o problema do atendimento de *deadlines* em sistemas de tempo-real é proposta no Capítulo 5.

A execução do programa de teste resulta no teste do próprio processador e, opcionalmente, no teste dos demais componentes do sistema. A cada período de teste, as respostas do teste armazenadas na memória devem ser comparadas com as respostas esperadas, indicando, assim, a presença ou ausência de falhas. Um módulo dedicado, externo ao processador, pode ser usado para a análise das respostas de teste (ver Seção 5.3). O uso de um módulo externo evita que o próprio processador, possivelmente falho, seja responsável pela comparação das respostas obtidas com as respostas esperadas, o que poderia resultar na anulação do efeito de uma falha por uma outra falha, ou pela mesma. Nesse caso, uma ou ambas as falhas não seriam detectadas.

4.1 SBST Funcional

O teste baseado em software tem uma longa história como técnica *ad hoc* para o teste de processadores. Sistemas de computadores são regularmente equipados com programas de software para realizar teste em campo. Tais testes são tipicamente usados para verificar a funcionalidade do sistema, não para detectar defeitos de fabricação ou adquiridos, ou seja, realizam teste funcional.

Apesar de longo histórico, o método de auto-teste baseado em software só foi formalmente proposto como uma alternativa para o auto-teste baseado em hardware em 1998 (SHEN; ABRAHAM, 1998). Nessa proposta, foi apresentada uma abordagem para geração de testes funcionais e sua aplicação para teste de produção e validação de projeto. Nenhum modelo de falhas estrutural é considerado nessa abordagem e as únicas

informações necessárias para o método de geração de teste são o conjunto de instruções do processador e as operações que o processador executa como resposta para cada instrução. Para o teste funcional de cada instrução, o método gera uma seqüência de instruções que enumera todas as possíveis combinações de operações, e sistematicamente seleciona operandos. Também, seqüências aleatórias são geradas para grupos de instruções, para exercitar as instruções e propagar seus efeitos.

A abordagem para a composição do programa de teste que utiliza principalmente seqüências aleatórias de instruções e operandos, e que visa prioritariamente verificar a funcionalidade do processador, é denominada *SBST funcional*. Após a proposta de Shen e Abraham, outros métodos de auto-teste utilizando seqüências aleatórias de instruções e operandos foram apresentados. Entre eles destaca-se o IRST (*Instruction Randomization Self-Test*) (BATCHER; PAPACHRISTOU, 1999). Nesse método, o auto-teste é realizado através de instruções do processador aleatoriamente selecionadas por um circuito dedicado a esse propósito, projetado fora do processador. O IRST não causa queda de desempenho do processador e o hardware extra é relativamente pequeno se comparado ao tamanho do processador.

Em 2002 foi proposta uma outra abordagem, denominada FRITS (*Functional Random Instruction Testing*), a qual aplica seqüências de instruções aleatórias e tenta reduzir ainda mais o custo de desenvolvimento do teste funcional de microprocessadores (PARVATHALA; MANEPARAMBIL; LINDSAY, 2002). Nessa abordagem, são propostas modificações DFT para possibilitar a aplicação da metodologia de auto-teste funcional para teste de produção, usando testadores de baixo custo e com poucos pinos. Além disso, a automatização da geração de programas de auto-teste é também considerada. A característica básica do FRITS, que constitui a principal diferença em relação às técnicas clássicas de auto-teste funcional de processadores, é que um conjunto de rotinas FRITS básicas (chamadas *kernels*) é carregado para a memória do processador sendo, então, responsável pela geração de vários programas que consistem de seqüências de instruções aleatórias e são usados para testar partes do processador.

A abordagem SBST funcional tem baixo custo de projeto porque se baseia na geração aleatória de instruções e operandos, sendo portanto, independente da estrutura lógica do processador. Entretanto, a principal limitação das técnicas apresentadas, assim como qualquer outra técnica de auto-teste funcional baseada na geração aleatória de conjuntos de teste, é que um nível aceitável de cobertura de falhas somente pode ser alcançado com a aplicação de longas seqüências de instruções e inúmeros diferentes operandos. Por isso, novas abordagens para a composição do programa de auto-teste foram desenvolvidas, e são discutidas a seguir.

4.2 SBST Estrutural

Na abordagem *SBST estrutural*, o programa de auto-teste é composto por um conjunto de rotinas de teste especificamente desenvolvidas para a detecção de falhas estruturais em cada um dos principais componentes do processador. Sendo assim, o conhecimento da organização do processador, no mínimo em nível RTL, se faz necessário. É importante ressaltar que, embora essa estratégia realize teste estrutural, uma vez que visa à detecção de falhas estruturais através de um modelo RTL do circuito, a aplicação dos padrões de teste continua sendo funcional, pois é feita por meio do conjunto de instruções funcionais do processador. Isso significa que somente as

falhas funcionalmente detectáveis podem ser detectadas pela abordagem SBST (tanto funcional quanto estrutural). Com base nessa abordagem, diversos métodos para a definição de rotinas de teste para componentes específicos do processador têm sido propostos.

Em 2001, foi proposta uma abordagem de SBST estrutural baseada no uso de assinaturas de auto-teste (CHEN; DEY, 2001). Assinaturas de auto-teste provêem uma forma compacta de carregar para a memória do sistema padrões de teste previamente preparados ou aleatórios. As assinaturas de auto-teste são expandidas em padrões de teste, através de rotinas de software, os quais são aplicados aos componentes do processador. As respostas de teste são coletadas (individualmente ou na forma de uma assinatura de teste) para posterior avaliação. Os conjuntos de teste para os componentes podem ser tanto previamente gerados por ATPG e inseridos em seqüências pseudo-aleatórias, quanto gerados por geradores pseudo-aleatórios implementados em software (LFSRs). O método leva em consideração as restrições impostas pelo conjunto de instruções do processador para a geração dos padrões de teste a partir de LFSRs.

Também a partir de 2001, outros autores propuseram programas de auto-teste baseados em pequenos conjuntos de padrões de teste determinísticos (PASCHALIS et al., 2001; KRANITIS et al., 2003; 2003a). Se as informações do processador estão disponíveis em nível lógico, padrões determinísticos podem ser gerados por uma ferramenta ATPG. A aplicação dos padrões gerados pode ser implementada de duas formas, com os padrões de teste buscados na memória de dados, ou aplicados como operandos imediatos das instruções. Nesses trabalhos também foram propostos métodos de auto-teste estrutural de alto-nível, nos quais as próprias rotinas de auto-teste geram os padrões de teste através de funções regulares independentes da implementação dos componentes no nível lógico. Os vetores de teste são baseados apenas no conhecimento do ISA do processador, sem necessidade de sua descrição RTL.

O SBST estrutural é um método muito mais flexível do que o SBST funcional, uma vez que permite a seleção dos componentes do processador que serão testados e o desenvolvimento de rotinas de teste exclusivamente para eles. Cada rotina pode ser implementada com uma técnica diferente, possibilitando uma melhor adequação aos requisitos do sistema, tanto em termos de teste quanto em termos da aplicação. O SBST estrutural pode, ainda, abranger diferentes modelos de falhas, pois possibilita o desenvolvimento de rotinas de teste para cada modelo de falhas visado. Além disso, apesar de ter um custo de desenvolvimento maior e, algumas vezes, depender da descrição RTL do processador, a estratégia SBST estrutural necessita de um número bem menor de padrões de teste para atingir alta cobertura de falhas, pois não é baseado na geração aleatória de seqüências de instruções. Assim, obtém-se um tempo de teste bastante reduzido em relação à estratégia SBST funcional, tornando a abordagem estrutural ideal para testes periódicos.

Pelos motivos descritos acima (flexibilidade, alta cobertura de falhas e curto tempo de teste), a abordagem SBST estrutural foi escolhida como base para o desenvolvimento da metodologia de teste apresentada neste trabalho. Assim, para permitir um melhor entendimento dessa técnica, a próxima seção trata, com detalhes, o problema da seleção dos componentes de um processador a serem testados e da geração de rotinas de teste para cada um deles, tendo em vista o teste *on-line* periódico.

4.3 Metodologia para o Projeto de SBST Estrutural

Foi dito anteriormente que o auto-teste baseado em software pode ser usado para testar somente alguns componentes de um processador, para testar o processador como um todo, ou para testar os demais núcleos de um SoC baseado em processador. Este trabalho tem por objetivo a obtenção de uma alta cobertura de falhas para todo o processador. Porém, visto que a abordagem SBST estrutural é mais adequada para o teste *on-line* periódico e que esta se baseia no desenvolvimento de rotinas de auto-teste para componentes específicos do processador, a estratégia usada deve se concentrar no teste desses componentes visando à obtenção de alta cobertura de falhas para o processador como um todo.

A metodologia apresentada a seguir (PASCHALIS; GIZOPOULOS, 2005) descreve as etapas do projeto de SBST estrutural para o teste *on-line* de processadores. Nela são definidos os componentes do processador que devem ser priorizados para a obtenção de uma cobertura de falhas alta. Além disso, são sugeridas algumas abordagens para o desenvolvimento das rotinas de teste que irão compor o programa de auto-teste. Vale salientar que esta metodologia foi utilizada como base para o desenvolvimento do presente trabalho, que é mais abrangente no sentido em que é voltado para o teste *on-line* periódico de processadores embarcados, possivelmente em sistemas de tempo-real. O conteúdo desta seção constitui uma das etapas da metodologia desenvolvida neste trabalho, a qual é inteiramente descrita no próximo capítulo.

O projeto de SBST estrutural aqui descrito consiste de três fases, mostradas na Figura 4.2 e resumidamente descritas a seguir.

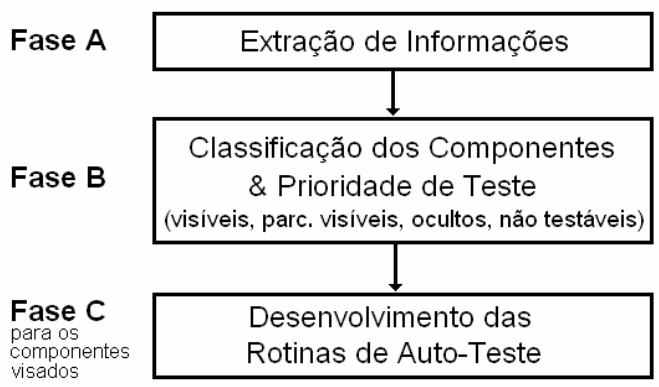

Figura 4.2: Fases da metodologia SBST estrutural
(PASCHALIS; GIZOPOULOS, 2005, p.92)

Fase A: identificação dos componentes do processador (com seus respectivos multiplexadores) e suas operações, bem como das instruções que excitam tais operações e das instruções (ou seqüências de instruções) para controle e observação dos registradores do processador.

Fase B: classificação dos componentes do processador de acordo com suas propriedades no que diz respeito ao ponto de vista do programador de linguagem Assembly (linguagem de máquina), e priorização dos componentes para desenvolvimento do teste. Esse esquema de classificação é importante para a escolha dos componentes visados no teste. Também pode ser usado para a seleção sistemática de uma estratégia de geração de padrões de teste (TPG – *Test Patterns Generation*)

conveniente, bem como para a transformação sistemática dos padrões de teste em rotinas de auto-teste.

Fase C: somente para os componentes visados. Desenvolvimento das rotinas de auto-teste com base em estilos de código específicos para as três principais estratégias de TPG. Indicação da estratégia de TPG mais efetiva para cada classe de componentes, focando o teste *on-line* periódico.

As três fases da metodologia são explicadas com maiores detalhes nas próximas seções.

4.3.1 Fase A – Extração de Informações

O principal objetivo dessa primeira fase do projeto de auto-teste baseado em software é a obtenção das informações do processador necessárias para o desenvolvimento das rotinas de auto-teste. A Figura 4.3 mostra um fluxograma das etapas que compõem esta fase da metodologia.

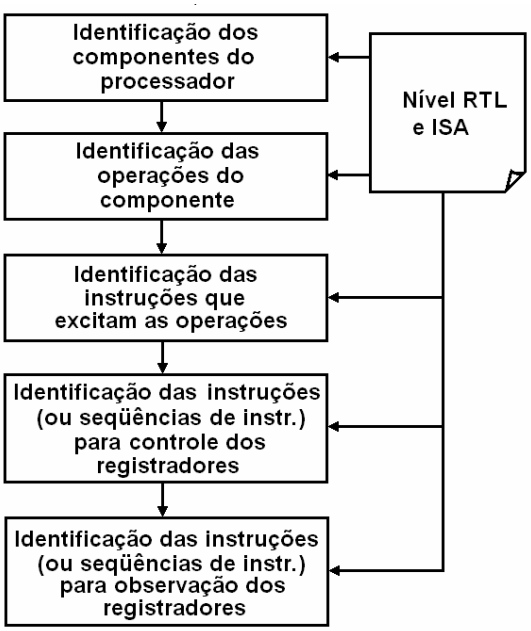

Figura 4.3: Fase A da metodologia SBST estrutural
(KRANITIS et al., 2003a, p.433)

O ponto de partida da estratégia SBST estrutural são os formatos das instruções do processador, derivados do ISA, e as descrições RTL das micro-operações do processador, derivadas da sua descrição RTL. A partir daí, são identificados os componentes do processador, com suas respectivas entradas e saídas, e as micro-operações realizadas por cada componente no nível RTL. Nesse estágio, também se deve identificar possíveis multiplexadores nas entradas ou saídas de cada componente. Então, mapeiam-se as entradas e saídas de cada componente (incluindo seus multiplexadores) para registradores temporários internos, no caso de uma arquitetura multiciclo, ou para registradores de pipeline, no caso de uma arquitetura pipeline.

O passo seguinte dessa fase consiste na identificação das instruções que executam uma operação específica e excitam um determinado componente com seus multiplexadores e registradores correspondentes, se existirem. Finalmente, são identificadas instruções para controlar os valores das entradas dos componentes ou

registradores correspondentes (instruções para controlabilidade); e instruções que assegurem a propagação das saídas dos componentes ou registradores correspondentes para saídas primárias do processador (instruções para observabilidade). Ambos os processos de controlabilidade e observabilidade podem ser realizados por uma única instrução, ou por uma seqüência de instruções do processador.

4.3.2 Fase B – Classificação dos Componentes e Prioridade de Teste

Com base nas informações obtidas na fase A, os componentes do processador são classificados nas quatro categorias a seguir (PASCHALIS; GIZOPOULOS, 2005), definidas a partir do ponto de vista do programador de linguagem Assembly:

- *Componentes Visíveis* (VC – *Visible Components*): são os componentes do processador cujas entradas e saídas são visíveis ao programador de linguagem Assembly. Para esses componentes há, no mínimo, uma instrução ou seqüência de instruções que controla suas entradas ou registradores correspondentes e, no mínimo, uma instrução ou seqüência de instruções que assegura a propagação de suas saídas ou dos registradores correspondentes para saídas primárias do processador. Os VCs são divididos, ainda, em três sub-classes de acordo com o tipo de suas entradas e saídas (dados ou endereços), conforme segue.
  - *Componentes Visíveis de Dados* (D-VC – *Data Visible Components*}: as entradas desses componentes recebem dados que podem estar armazenados: 1) nos campos de uma instrução, com modo de endereçamento imediato; 2) no banco de registradores, com modo de endereçamento de registradores; 3) na memória de dados; ou 4) em registradores de dados controláveis (isto é, registradores de dados diretamente conectados ao banco de registradores ou à memória de dados). As saídas desses componentes produzem dados que podem ser armazenados: 1) no banco de registradores; 2) na memória de dados; ou 3) em registradores de dados observáveis (isto é, registradores de dados diretamente conectados ao banco de registradores ou à memória de dados). Tais componentes são *componentes de processamento de dados* (ULAs, deslocadores, multiplicadores, divisores, etc.) e *componentes de armazenamento de dados* (banco de registradores e registradores especiais ou temporários de dados). A esta classe também pertencem os multiplexadores das entradas ou saídas desses componentes.
  - *Componentes Visíveis de Endereço* (A-VC – *Address Visible Components*): as entradas e saídas desses componentes são endereços de memória. Os valores desses endereços dependem das posições de memória onde instruções e dados estão armazenados e onde dados serão armazenados. Assim, tais componentes tornam-se visíveis pelo conveniente armazenamento de instruções e dados na memória do sistema. Os A-VCs geralmente aparecem dentro da *unidade de busca de instruções* (por exemplo, o PC – *Program Counter*) e do *controlador da memória de dados* (por exemplo, o MAR – *Memory Address Register*). A esta classe também pertencem os registradores especiais ou temporários de endereço, bem como os multiplexadores das entradas ou saídas desses componentes.
  - *Componentes Visíveis Mistos (endereços-dados)* (M-VC – *Mixed Visible Components*): esses componentes têm entradas e/ou saídas de ambos os tipos (dados ou endereços) e tornam-se visíveis pelas mesmas formas mencionadas

acima. A esta subclasse pertence, por exemplo, o somador usado para incrementar o conteúdo do PC.

- *Componentes Parcialmente Visíveis* (PVC – *Partially Visible Components*): esses são os componentes do processador que geram sinais de controle. Uma vez que os sinais de controle que saem desses componentes afetam a operação dos VCs, esses componentes podem ser considerados parcialmente visíveis ao programador de linguagem Assembly. Tais componentes são a unidade de controle do processador e pequenos controladores distribuídos, geralmente implementados como *máquinas de estados finitos*.
- *Componentes Ocultos* (HC – *Hidden Components*): são os componentes usualmente adicionados à arquitetura do processador para aumentar seu desempenho, mas que não são visíveis ao programador de linguagem Assembly. Esses componentes consistem de sub-componentes, os quais são classificados nas três subclasses a seguir:
  - *Sub-componentes Ocultos Visíveis de Dados* (D-VHSC – *Data Visible Hidden Subcomponents*): esses sub-componentes recebem e produzem dados, da mesma forma que os D-VCs. Eles são os campos de dados e os campos imediatos dos registradores de pipeline, e seus respectivos multiplexadores.
  - *Sub-componentes Ocultos Visíveis de Endereços* (A-VHSC – *Address Visible Hidden Subcomponents*): esses sub-componentes recebem endereços de memória, como os A-VCs. Eles são os campos de endereço dos registradores de pipeline e seus respectivos multiplexadores.
  - *Sub-componentes Ocultos Parcialmente Visíveis* (PVHSC – *Partially Visible Hidden Subcomponents*): estes sub-componentes geram sinais de controle que afetam a operação dos sub-componentes visíveis dentro de um componente oculto. Eles são as unidades de controle do pipeline (por exemplo, unidade de *forwarding*, unidade de detecção de dependências, etc.) e o mecanismo de predição de desvio.
- *Componentes Não-testáveis* (NTC – *Nontestable Components*): são os componentes do processador que não podem ser testados unicamente pela execução de programas de auto-teste desenvolvidos pelo programador de linguagem Assembly. Tais componentes incluem, por exemplo, registradores, multiplexadores e a lógica de controle para tratamento de interrupções e exceções.

A partir da classificação dos componentes apresentada acima, foi definida a prioridade de teste de cada classe e sub-classe em função da sua testabilidade e da área ocupada por seus componentes no processador (PASCHALIS; GIZOPOULOS, 2005). Tal prioridade é ilustrada na Figura 4.4 e enumerada a seguir, em ordem decrescente.

1. **D-VCs**: esses componentes possuem a mais alta testabilidade e constituem grande parte da área do processador. Por isso têm a mais alta prioridade e são ideais para testes periódicos, fornecendo, em muitos casos, uma alta cobertura de falhas para todo o processador.
2. **D-VHSCs**: esses sub-componentes têm alta prioridade de teste porque possuem alta testabilidade, sendo adequados para teste periódico. Eles são suficientemente testados pela grande quantidade de dados que recebem durante o teste dos D-VCs.

Figura 4.4: Prioridade de teste dos componentes do processador
(PASCHALIS; GIZOPOULOS, 2005, p.93)

3. **PVCs**: esses componentes possuem testabilidade média-alta. Eles ocupam uma pequena parte da área do processador e geralmente são adequados para teste *on-line* periódico. Para testar tais componentes adotam-se, em geral, testes funcionais simples de alto nível, como a aplicação de todos os *opcodes* de instruções ainda não aplicados, no caso da unidade de controle, bem como a aplicação de instruções que alcancem a maior cobertura de código RTL possível, no caso de outras máquinas de estados finitos específicas.
4. **PVHSCs**: esses sub-componentes possuem testabilidade média-alta e são, em geral, adequados para teste *on-line* periódico, podendo ser testados da mesma forma que os PVCs.
5. **M-VCs e A-VCs**: esses componentes têm testabilidade média-baixa, devido às limitações inerentes do espaço de endereçamento, e não são adequados para teste periódico uma vez que seu teste requer várias referências distribuídas à memória, aumentando consideravelmente o tempo de execução da rotina de teste. Além disso, eles ocupam uma área muito pequena do processador e são parcialmente testados pelos endereçamentos ao programa e aos dados de teste durante o teste dos D-VCs.
6. **A-VHSCs**: esses sub-componentes têm as mesmas características dos A-VCs e M-VCs, sendo parcialmente testados como efeito colateral do teste dos D-VCs.
7. **NTCs**: esses componentes não podem ser considerados para aplicação de SBST.

Componentes da mesma classe, ou subclasse, são priorizados por seu tamanho (quanto maior sua área, mais alta é a sua prioridade). Como regra geral, deve-se prosseguir para o próximo componente da mesma classe (ou subclasse) ou para a próxima classe (ou subclasse) apenas no caso em que a cobertura de falhas ainda não é aceitável.

Os D-VCs e D-VHSCs são os componentes mais importantes a serem testados, e cujo teste é ideal para a aplicação de SBST periódico. Eles dominam a área em vários processadores, simples ou complexos, comumente usados em sistemas embarcados de baixo custo. Portanto, na prática, o teste dos componentes ou sub-componentes visíveis de dados geralmente levam a uma cobertura de falhas aceitável uma vez que os demais componentes do processador são parcialmente testados como um efeito colateral (PASCHALIS; GIZOPOULOS, 2005). Essa cobertura de falhas pode ser ligeiramente

aumentada, se necessário, com o teste dos PVCs e PVHSCs, que ocupam uma pequena área do processador, adotando-se testes funcionais.

A Figura 4.5 representa o fluxograma da segunda fase do projeto de auto-teste baseado em software estrutural. A próxima fase do projeto concentra-se no desenvolvimento das rotinas de auto-teste para a classe de componentes mais crítica em termos de teste, os componentes visíveis de dados (D-VCs).

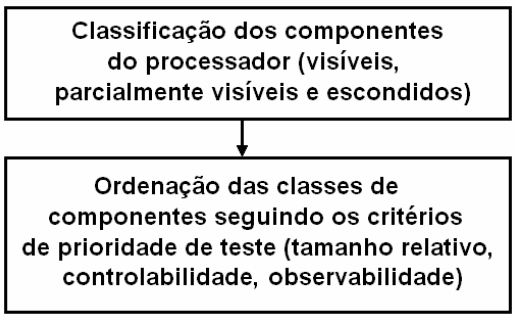

Figura 4.5: Fase B da metodologia SBST estrutural
(KRANITIS et al., 2003a, p.434)

4.3.3 Fase C – Desenvolvimento das Rotinas de Auto-Teste

O desenvolvimento das rotinas de auto-teste inicia com a geração dos padrões de teste (TPG – *Test Pattern Generation*) para os componentes alvo, e continua com a transformação destes padrões em rotinas de software. Esses passos podem ser executados usando diferentes abordagens, que variam de acordo com a estratégia de TPG adotada e o estilo do código usado na implementação das rotinas de auto-teste. As próximas seções apresentam algumas possíveis abordagens para a geração de código das rotinas de teste (PASCHALIS; GIZOPOULOS, 2005). As abordagens são classificadas de acordo com a estratégia utilizada para geração dos padrões de teste, conforme ilustrado na Figura 4.6.

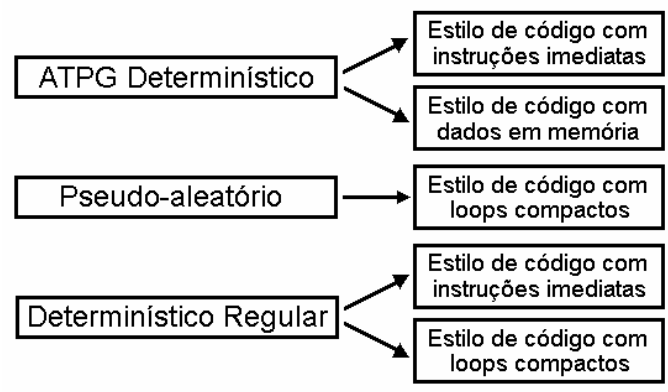

Figura 4.6: Estratégias de TPG e seus estilos de código
(PASCHALIS; GIZOPOULOS, 2005, p.94)

4.3.3.1 TPG Determinístico baseado em ATPG

A primeira estratégia de geração de padrões de teste é baseada em ATPG determinístico e é geralmente aplicada a D-VCs combinacionais. A ferramenta de ATPG utilizada deve levar em consideração as limitações impostas pelo conjunto de

instruções do processador. Essa é uma estratégia de baixo nível, visto que, em geral, ferramentas ATPG necessitam do conhecimento da estrutura do circuito no nível de portas lógicas para a geração dos vetores de teste.

A partir dos padrões gerados por ATPG, duas abordagens para a implementação de rotinas de auto-teste eficientes são possíveis. Na primeira, os padrões de teste são aplicados como operandos imediatos das instruções, ou seja, estão contidos na própria rotina. Na segunda, os padrões são armazenados na memória de dados e a rotina de auto-teste, formada por um laço, busca-os e aplica-os ao componente em teste.

Para efeito de exemplificação, a Figura 4.7 mostra o estilo de código para uma rotina de auto-teste que aplica *n* padrões de teste a um (sub)componente visível de dados, com duas entradas *X* e *Y*, através de instruções com modo de endereçamento imediato. O código dessa figura, bem como os demais nesta seção, utiliza a linguagem Assembly da arquitetura MIPS. A instrução de nome *funcao*, que endereça registradores, executa uma operação específica e excita o D-VC correspondente com seus respectivos multiplexadores e registradores, caso existam. Os padrões de teste são carregados em registradores pela pseudo-instrução *load immediate* (*li*), a qual o montador decompõe nas instruções *lui* e *ori* sem transferir dados da memória. Após a aplicação dos padrões de teste, as respostas podem ser compactadas por uma rotina de compactação, que geralmente implementa um MISR, de forma a evitar a transferência e o armazenamento de muitos dados na memória. Ao final, a assinatura do teste é armazenada na memória de dados no endereço *endereco_assinatura+deslocamento*.

```
li $s0, padrao_X_1;                  # padrão X1
li $s1, padrao_Y_1;                  # padrão Y1
funcao $s2 $s0, $s1;                 # aplicação do teste
jal endereco_rotina_compactacao;     # compactação da resposta
.......
li $s0, padrao_X_n;                  # padrão Xn
li $s1, padrao_Y_n;                  # padrão Yn
funcao $s2 $s0, $s1;                 # aplicação do teste
jal endereco_rotina_compactacao;     # compactação da resposta
li $s3, endereco_assinatura;
sw $s2, (deslocamento) ($s3);        # armazenamento da assinatura
```

Figura 4.7: Estilo de código baseado em ATPG com instruções imediatas (PASCHALIS; GIZOPOULOS, 2005, p.94)

Essa abordagem de rotina de auto-teste possui as seguintes características:

- O tamanho do código é linearmente proporcional ao número de padrões de teste. Se o número de padrões de teste é grande, o tamanho da memória de instruções ocupada também será.
- O tamanho da memória de dados necessária é linearmente proporcional ao número de respostas de teste. Nos casos em que há compactação, esse tamanho é mínimo.
- Código sem dependências de dados não resolvidas (assumindo-se que o processador suporta *forwarding*), com instruções executadas seqüencialmente, o que tira vantagem da localidade espacial na *cache* de instruções.
- Alta taxa de *cache miss* de instruções atenuada pela localidade espacial e linearmente dependente do número de padrões de teste.

- A ausência de leituras na memória de dados e o uso de apenas uma escrita nessa memória resultam em *cache miss* de dados igual a zero.

Alternativamente, um exemplo de estilo de código da rotina de auto-teste que aplica *n* padrões de teste a um (sub)componente visível de dados, com duas entradas *X* e *Y*, por meio de um laço que busca os padrões na memória de dados, é mostrado na Figura 4.8. Assume-se que o primeiro padrão de teste está armazenado na memória de dados no endereço *endereco_primeiro_padrao*, e também que o número de padrões de teste para ambas as entradas é *numero_de_padroes*. Cada padrão é primeiro carregado para o banco de registradores, depois aplicado ao CUT, e sua resposta é então compactada. Após a aplicação de todos os padrões, a assinatura do teste é armazenada na memória de dados.

```
li $s3, endereco_primeiro_padrao;
addi $s4, $zero, numero_de_padroes;
add $t0, $zero, $zero;
loop_padroes_teste:
     lw $so, 0($s3);                        # padrão X
     addiu $s3, $s3, 0x0004;                # assumindo dados de 32 bits
     lw $s1, 0($s3);                        # padrão Y
     addiu $s3, $s3, 0x0004;
     funcao $s2 $s0, $s1;                   # aplicação do teste
     jal endereco_rotina_compactacao;       # compactação da resposta
     addiu $t0, $t0, 0x0001;
     bne $s4, $t0, loop_padroes_teste;      # armazenamento da assinatura
li $s5, endereco_assinatura;
sw $s2, (deslocamento) ($s5);
```

Figura 4.8: Estilo de código baseado em ATPG com dados em memória (PASCHALIS; GIZOPOULOS, 2005, p.94)

Essa abordagem de rotina de auto-teste possui as seguintes características:

- O tamanho do código é pequeno e independente do número de padrões de teste. Logo, a quantidade de memória de instruções necessária é pequena.
- O tamanho da memória de dados necessária é linearmente proporcional ao número de padrões e respostas de teste.
- Código sem dependências de dados não resolvidas (assumindo-se que o processador suporta *forwarding*), com um laço compacto, o que tira vantagem da localidade temporal na *cache* de instruções.
- Baixa taxa de *cache miss* de instruções atenuada pela localidade espacial e linearmente dependente do número de padrões de teste.
- Alta taxa de *cache miss* de dados atenuada pela localidade espacial e linearmente dependente do número de padrões de teste.

Ambas as alternativas de rotinas de auto-teste podem ser usadas na prática para teste *on-line* periódico uma vez que elas têm tempo de execução pequeno, assumindo-se que o número de padrões de teste gerados ATPG é pequeno. A primeira tem maior tamanho de memória de instruções (pois nela estão contidos os padrões de teste), enquanto a segunda tem maior tamanho de memória de dados (para armazenamento dos padrões de teste).

4.3.3.2 TPG Pseudo-aleatório

A segunda estratégia de TPG é baseada na geração de padrões pseudo-aleatórios e é usualmente aplicada a D-VCs combinacionais de estrutura irregular. Os padrões de teste pseudo-aleatórios são gerados por uma rotina de software, baseada em laço, que implementa um LFSR, levando em consideração as limitações impostas pelo conjunto de instruções do processador (CHEN; DEY, 2001). Essa também é uma estratégia de baixo nível, pois requer o conhecimento da estrutura do processador em nível lógico. Além disso, a geração pseudo-aleatória requer um número muito grande de padrões de teste para atingir uma cobertura de falhas aceitável, aumentando consideravelmente o tempo de teste. Essa característica pode tornar a abordagem baseada em LFSR inadequada para o teste *on-line* periódico.

A Figura 4.9 mostra um exemplo de estilo de código para uma rotina de auto-teste que gera e aplica padrões de teste pseudo-aleatórios em um (sub)componente visível de dados, com duas entradas *X* e *Y*, por meio de um código baseado em laço. A instrução de nome *funcao*, que endereça registradores, executa uma operação específica e excita o D-VC correspondente, como nos exemplos anteriores. Assume-se que o número de padrões de teste para ambas as entradas é dado por *numero_de_padroes*, e que *semente* e *polinomio* são usados pela implementação em software do LFSR. Todos os padrões de teste gerados pseudo-aleatoriamente são aplicados ao CUT e suas respostas são compactadas. Por fim, a assinatura de teste é armazenada na memória de dados.

```
li $s3, semente;
li $s4, polinomio;
addi $s5, $zero, numero_de_padroes;
add $t0, $zero, $zero;
loop_padroes_teste:
     # geracao LFSR para $s0;              # geração do padrão X
     # geracao LFSR para $s1;              # geração do padrão Y
     funcao $s2 $s0, $s1;                  # aplicação do teste
     addiu $t0, $t0, 0x0001;
     jal endereco_rotina_compactacao;      # compactação da resposta
     bne $s5, $t0, loop_padroes_teste;
li $s6, endereco_assinatura;
sw $s2, (deslocamento) ($s6);              # armazenamento da assinatura
```

Figura 4.9: Estilo de código baseado em laço com padrões pseudo-aleatórios (PASCHALIS; GIZOPOULOS, 2005, p.95)

Essa abordagem de rotina de auto-teste possui as seguintes características:

- O tamanho do código é pequeno e independente do número de padrões de teste. Logo, a quantidade de memória de instruções necessária é pequena.
- O tamanho da memória de dados necessária é linearmente proporcional ao número de respostas de teste. Nos casos em que há compactação, esse tamanho é mínimo.
- Código sem dependências de dados não resolvidas (assumindo-se que o processador suporta *forwarding*), com um laço compacto, o que tira vantagem da localidade temporal na *cache* de instruções.
- Baixa taxa de *cache miss* de instruções atenuada pela localidade espacial e linearmente dependente do número de padrões de teste.

- A ausência de leituras na memória de dados e o uso de apenas uma escrita nessa memória resultam em *cache miss* de dados igual a zero.

4.3.3.3 TPG Determinístico Regular

A terceira estratégia de TPG é baseada na geração de padrões determinísticos regulares. Ela explora a regularidade inerente aos D-VCs mais críticos em termos de teste, como unidades lógicas e aritméticas, deslocadores, comparadores, multiplexadores, registradores, e banco de registradores, os quais usualmente constituem a grande maioria dos componentes do processador. Em muitos casos, o teste somente desses componentes fornece uma cobertura de falhas aceitável para todo o processador.

Essa estratégia de TPG é uma abordagem de alto nível, uma vez que os padrões de teste gerados são independentes da implementação dos componentes em nível de portas lógicas e constituem conjuntos de teste de tamanho constante ou linear. Os padrões de teste determinísticos regulares são transformados em rotinas de auto-teste de duas formas alternativas: 1) os vetores de teste são transformados em instruções com modo de endereçamento imediato nos casos em que sua quantidade é pequena o suficiente ou 2) os padrões de teste são gerados e aplicados por uma rotina de auto-teste baseada em laço a partir de um valor inicial, um valor final, e uma função específica que faz a transição regular de um valor para outro (PASCHALIS et al., 2001; KRANITIS et al., 2003a).

Nos casos em que os padrões de teste determinísticos regulares não podem ser facilmente gerados por uma rotina que implementa um laço simples, pode-se utilizar instruções do processador com modo de endereçamento imediato para aplicar os padrões previamente determinados, de modo similar ao mostrado na Figura 4.7. Esse é o caso do teste do banco de registradores, onde podem ser aplicados padrões determinísticos produzidos por algoritmos específicos para o teste de elementos de memória, como o algoritmo March (LALA, 1997). Além disso, todos os registradores do banco devem receber no mínimo dois padrões. Para evitar armazenamentos na memória de dados durante o teste, este pode ser feito em duas fases. Na primeira, testa-se uma metade do banco de registradores usando-se a outra metade para armazenamento temporário (por exemplo, para compactação). Na segunda fase, procede-se de maneira contrária.

Nos outros casos, pode-se utilizar o estilo de código exemplificado na Figura 4.10, no qual a rotina de auto-teste gera um pequeno número de padrões de teste determinísticos regulares e aplica-os ao respectivo (sub)componente visível de dados, de duas entradas *X* e *Y*. Assume-se que uma instrução de nome *funcao*, que endereça registradores, excita o D-VC pertinente, como nos exemplos anteriores. Considera-se também que, para cada valor de entrada *X*, todos os valores de entrada *Y* são aplicados ao componente e, a cada iteração, a resposta de teste é compactada. Em alguns casos, o valor final é igual ao valor inicial, e algumas instruções podem ser eliminadas. Ao final, a assinatura de teste é armazenada na memória de dados.

```
li $s0, valor_inicial_X;                    # inicia padrão X
li $s3, valor inicial_Y;
li $s4, valor_final_X;
li $s5, valor_final_Y;
add $s1, $s3, $zero;                        # inicia padrão Y
loop_padroes_teste:
     funcao $s2 $s0, $s1;                   # aplicação do teste
     # gera proximo padrao Y em $s1;        # geração do próximo padrão Y
     jal endereco_rotina_compactacao;       # compactação da resposta
     bne $s5, $s1, loop_padroes_teste;
     add $s1, $s3, $zero;                   # reinicia padrão Y
     # gera proximo padrao X em $s0;        # geração do próximo padrão X
     bne $s4, $s0, loop_padroes_teste;
li $s6, endereco_assinatura;
sw $s2, (deslocamento) ($s6);               # armazenamento da assinatura
```

Figura 4.10: Estilo de código baseado em laço com padrões determinísticos regulares (PASCHALIS; GIZOPOULOS, 2005, p.96)

Essa abordagem de rotina de auto-teste possui as seguintes características:

- O tamanho do código é pequeno e independente do número de padrões de teste. Logo, a quantidade de memória de instruções necessária é pequena.
- O tamanho da memória de dados necessária é linearmente proporcional ao número de respostas de teste. Nos casos em que há compactação, esse tamanho é mínimo.
- Código sem dependências de dados não resolvidas (assumindo-se que o processador suporta *forwarding*), com um laço compacto, o que tira vantagem da localidade temporal na *cache* de instruções.
- Baixa taxa de *cache miss* de instruções atenuada pela localidade espacial e linearmente dependente do número de padrões de teste.
- A ausência de leituras na memória de dados e o uso de apenas uma escrita nessa memória resultam em *cache miss* de dados igual a zero.

4.3.3.4 Aplicabilidade das Estratégias de TPG

A Tabela 4.1 resume, conceitualmente, as características das abordagens para o desenvolvimento de rotinas de auto-teste, em termos de ocupação de memória, taxa de *cache miss* e número de padrões de teste.

Tabela 4.1: Características das rotinas de auto-teste

Rotinas de Auto-Teste	Ocupação da Mem. de Instruções	Ocupação da Mem. de Dados	Taxa de *Cache Miss* de Instruções	Taxa de *Cache Miss* de Dados	Quantidade de Padrões de Teste
ATPG Imed.	ALTA	BAIXA	ALTA	BAIXA	BAIXA
ATPG Mem.	BAIXA	ALTA	BAIXA	ALTA	BAIXA
LFSR	BAIXA	BAIXA	BAIXA	BAIXA	ALTA
Det. Regular	BAIXA	BAIXA	BAIXA	BAIXA	BAIXA

Fonte: XENOULIS et al., 2003. p.153

A abordagem baseada em ATPG leva à utilização de grande quantidade de memória, bem como a uma alta taxa de *cache miss* , ou de dados ou de instruções. A construção dessa rotina de auto-teste é afetada pela irregularidade dos padrões de teste gerados por

ATPG. Já a abordagem pseudo-aleatória mantém baixa a ocupação das memórias e a as taxas de *cache miss* (tanto de dados quanto de instruções), mas geralmente requer um grande número de padrões de teste para garantir alta cobertura de falhas, aumentando o tempo de execução do teste. Por outro lado, a abordagem regular determinística alcança alta qualidade de teste com um pequeno número de padrões não gerados por ATPG, com baixas taxas de *cache miss*, e utilizando poucas palavras de memória. No entanto, essa abordagem não pode ser aplicada para qualquer componente do processador, sendo aplicável apenas para aqueles de estrutura regular.

Paschalis e Gizopoulos (PASCHALIS; GIZOPOULOS, 2005) definiram a aplicabilidade das estratégias de TPG e estilos de código para o teste *on-line* dos D-VCs de um processador com base nos seguintes fatores:

- A máxima cobertura de falhas possível que a estratégia de TPG pode alcançar para esse componente.
- O mínimo tempo de execução possível da rotina de auto-teste derivada da estratégia de TPG e do estilo de código.
- O mínimo tamanho possível de código e dados de teste da rotina de auto-teste derivada da estratégia de TPG e do estilo de código.

A partir desses fatores, os mesmos autores propuseram as seguintes regras para a aplicação de cada uma das três abordagens de TPG apresentadas:

- A estratégia de TPG baseada em padrões determinísticos regulares é indicada para D-VCs combinacionais ou seqüenciais com regularidade inerente, quando o tamanho do conjunto de teste é pequeno e leva a um tamanho de código e tempo de execução de teste aceitáveis.
- A estratégia de TPG baseada em ATPG é adequada para D-VCs combinacionais, quando o tamanho do conjunto de teste é pequeno e leva a um tamanho de código de teste aceitável (usa-se o estilo de código com instruções imediatas) ou de dados de teste aceitável (usa-se o estilo de código com busca em memória), e o tempo de execução do teste é menor do que na estratégia baseada em padrões determinísticos regulares.
- A estratégia de TPG baseada em padrões pseudo-aleatórios é indicada para D-VCs combinacionais de estrutura irregular, quando o tamanho do conjunto de teste (geralmente grande) leva a um tamanho de código de teste menor e a um tempo de execução de teste comparável às demais estratégias.

É importante ressaltar que Paschalis e Gizopoulos não consideram a aplicação de SBST *on-line* periódico para sistemas embarcados e/ou de tempo-real. Por isso, as regras propostas por eles para a seleção das estratégias de TPG para componentes específicos do processador não consideram as fortes restrições presentes nesses sistemas. Tais restrições, discutidas no Capítulo 3, são consideradas para a composição do programa de teste na metodologia desenvolvida neste trabalho e apresentada no próximo capítulo. Resultados comparativos, apresentados no Capítulo 6, da seleção de rotinas de auto-teste com base nas regras de Paschalis e Gizopoulos e com base na metodologia desenvolvida atestam a importância de se levar em consideração tais restrições do sistema durante a composição do programa de auto-teste.

4.4 Resumo e Conclusões

O auto-teste baseado em software foi o assunto principal deste capítulo. O SBST consiste no desenvolvimento de um programa de teste eficiente que alcance alta cobertura de falhas. O programa de auto-teste é armazenado na memória de instruções enquanto os padrões e as respostas de teste são armazenados na memória de dados. Ao ser executado, o programa de auto-teste aplica os padrões aos componentes do processador, ou aos demais núcleos do SoC se for o caso, e as respostas obtidas são armazenadas na memória do sistema.

A técnica SBST apresenta uma série de vantagens sobre outros métodos de teste *on-line*, principalmente sobre o HBST. A que mais se destaca é o fato do SBST ser um método de teste não intrusivo, uma vez que utiliza as próprias instruções do processador, dispensando quaisquer alterações DFT na sua estrutura ou no seu conjunto de instruções. Além disso, o SBST é uma estratégia de baixo custo, de baixa potência, bastante flexível e facilmente programável. Somando-se a isso a possibilidade de realização de teste *at-speed*, conclui-se que a técnica SBST é bastante adequada para o teste *on-line* de processadores embarcados.

O auto-teste baseado em software pode ser tanto funcional quanto estrutural. Na abordagem funcional são usados operandos e/ou seqüências de instruções aleatórias para testar todas as funcionalidades do processador. Apesar da vantagem de ser uma abordagem de alto nível, a cobertura de falhas alcançada é, em geral, muito baixa e o número de padrões de teste é alto. Já na abordagem estrutural, o programa de auto-teste é desenvolvido visando o teste de componentes específicos do processador. Assim, ele é composto por um conjunto de rotinas de teste, uma para cada componente visado. Essa estratégia requer um modelo do processador, no mínimo, em nível RTL, mas obtém alta cobertura de falhas com tempo de execução de teste aceitável, sendo, por isso, mais adequada para o teste *on-line* periódico.

Este capítulo também descreveu uma metodologia para o projeto de SBST estrutural para o teste *on-line* de processadores. Tal metodologia é composta de três fases: extração de informações, classificação dos componentes e prioridade de teste, e desenvolvimento das rotinas de auto-teste. A primeira fase consiste na identificação dos componentes do processador e das operações que eles realizam, bem como das instruções que executam tais operações e das instruções para controle e observação de cada componente.

Na segunda fase os componentes identificados são classificados a partir do ponto de vista do programador de linguagem Assembly, que tem relação direta com a testabilidade do componente. Eles podem ser: componentes visíveis (de dados, de endereços ou mistos), componentes parcialmente visíveis, componentes ocultos, ou componentes não testáveis. Essa fase da metodologia também define a prioridade de teste dos componentes, sendo mais prioritários os componentes visíveis de dados. Além de ocuparem grande parte da área do processador, os D-VCs têm alta testabilidade e o seu teste fornece, em geral, alta cobertura de falhas para todo o processador. Por esses motivos os D-VCs são bastante adequados para o teste *on-line* periódico.

Por fim, na terceira fase são desenvolvidas as rotinas de auto-teste. Ela inicia com a geração dos padrões de teste para os componentes alvo e termina com a transformação desses padrões em rotinas de software. A metodologia apresenta três abordagens para a geração dos padrões de teste e diferentes estilos de código para a transformação dos padrões em rotinas de auto-teste. As características de cada abordagem foram discutidas

e a aplicabilidade de cada uma foi apontada com base na sua cobertura de falhas, tempo de execução do teste e ocupação de memória de dados e instruções. A partir desses fatores, a abordagem regular determinística foi apontada como a mais adequada para componentes de estrutura regular, os quais são, na maioria, componentes visíveis de dados, ou seja, os de maior prioridade de teste.

Com base no conteúdo apresentado neste capítulo, pode-se concluir que a técnica de auto-teste baseado em software estrutural é bastante adequada para o teste *on-line* periódico de processadores embarcados. Por ser uma abordagem não intrusiva e muito flexível, ela foi escolhida como meio para a realização dos objetivos definidos no Capítulo 1. A metodologia aqui descrita foi usada como base para o desenvolvimento deste trabalho, cujos métodos são definidos e os resultados apresentados nos próximos capítulos.

Observa-se também que as rotinas de auto-teste devem ser desenvolvidas, em primeiro lugar, para os componentes visíveis de dados, visto que eles têm maior prioridade de teste. Uma vez que diversas abordagens de rotinas de auto-teste são possíveis para o teste de tais componentes, é fundamental a seleção das rotinas mais adequadas para a aplicação de teste *on-line* periódico em processadores embarcados. Uma das contribuições mais importantes deste trabalho são os métodos para a seleção de rotinas para compor o programa de auto-teste que, ao contrário da estratégia proposta por Paschalis e Gizopoulos apresentada neste capítulo, têm como foco principal as restrições e os requisitos do sistema ao qual pertence o processador alvo. A importância de tais métodos é comprovada por resultados comparativos no Capítulo 6.

5 A METODOLOGIA “STEP”

Este capítulo descreve uma metodologia para projeto e aplicação de auto-teste *on-line* periódico para processadores embarcados, a qual foi desenvolvida ao longo deste trabalho. O método aqui proposto foi batizado de STEP (*Self-Test for Embedded Processors*), e pretende ser genérico o suficiente para que possa ser aplicado a qualquer processador embarcado. Como um primeiro esforço para sua validação, a metodologia STEP foi aplicada aos processadores da família Femtojava. As particularidades de tal metodologia e os resultados obtidos quando da sua aplicação aos processadores Femtojava são discutidos no Capítulo 6.

A metodologia para o projeto de auto-teste *on-line* de processadores embarcados descrita a seguir se utiliza da técnica de auto-teste baseado em software (SBST). Desse modo, conforme justificado em capítulos anteriores, ela tem como base a metodologia para o projeto de SBST estrutural proposta por Paschalis e Gizopoulos, e apresentada no Capítulo 4 (Seção 4.3). Entretanto, a metodologia aqui proposta é mais específica e, talvez por isso, mais completa do que aquela. É mais específica porque foca o auto-teste de processadores embarcados, possivelmente em sistemas de tempo-real, considerando, portanto, os requisitos e restrições inerentes a esses sistemas. E mais completa porque inclui fatores que a outra desconsidera, como uma seleção efetiva das rotinas de teste para composição do programa de teste, ou que a outra assume, mas não trata, como a análise *on-line* das respostas de teste.

5.1 Fluxo de Projeto

Na metodologia STEP, o projeto do auto-teste *on-line* do processador embarcado ocorre em paralelo com o projeto do próprio sistema embarcado. Uma vez que ambos os projetos estão estreitamente relacionados, existe uma forte interação entre eles. A Figura 5.1 ilustra o fluxo de projeto do SEEP – *Sistemas Eletrônicos Embarcados baseados em Plataforma* (LSE, 2006), cujo teste pode ser planejado e aplicado por meio da metodologia STEP. O SEEP é um projeto desenvolvido pelo Grupo de Microeletrônica (GME) desta universidade, no Laboratório de Sistemas Embarcados (LSE). Seu objetivo é o desenvolvimento de sistemas eletrônicos embarcados baseados em plataforma, com ênfase na plataforma Femtojava, a qual foi utilizada no estudo de caso apresentado no próximo capítulo.

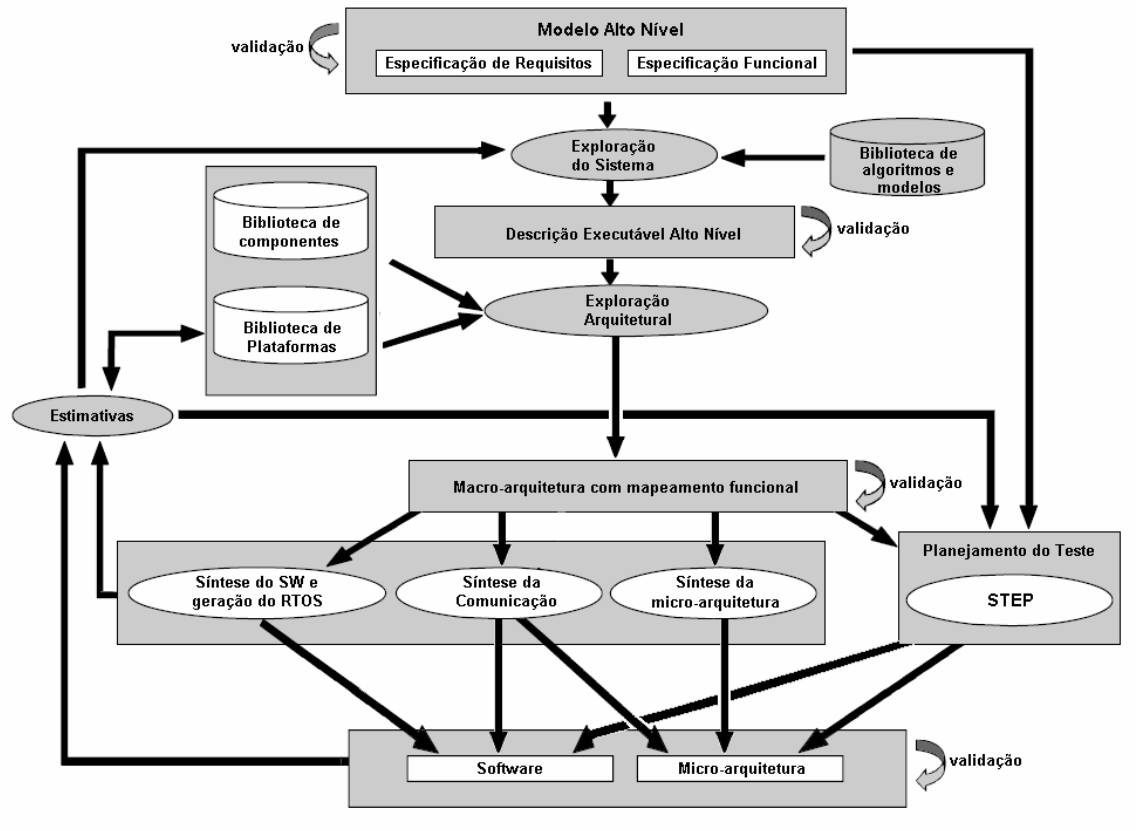

Figura 5.1: Fluxo de projeto de sistemas embarcados no SEEP (LSE, 2006)

A Figura 5.2 ilustra, de maneira bem simples, o fluxo do projeto de auto-teste *on-line* para processadores embarcados na metodologia STEP e as interações entre esse fluxo e o projeto do próprio sistema embarcado (fluxo SEEP).

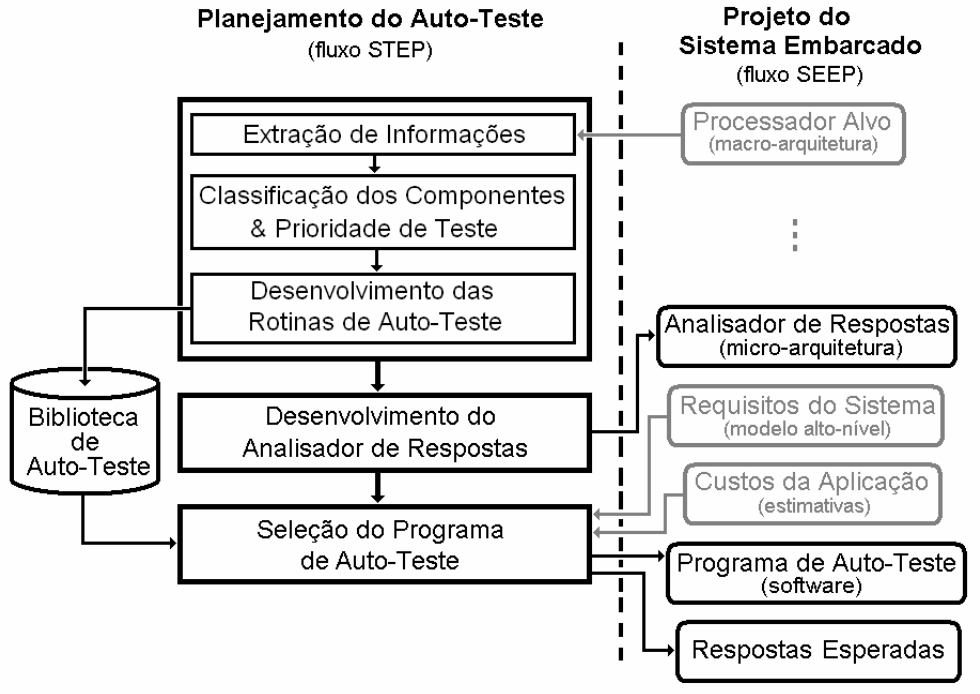

Figura 5.2: Fluxo de projeto de auto-teste *on-line* na metodologia STEP

No projeto do sistema embarcado, a partir do momento em que é definido o processador alvo (macro-arquitetura), e desde que se tenha um modelo RTL do mesmo, o projeto do auto-teste pode ser iniciado. A primeira etapa consiste no desenvolvimento da biblioteca de auto-teste. Para tanto, utilizam-se as três fases da metodologia de projeto de SBST estrutural, descritas no capítulo anterior. Na primeira fase são obtidas informações sobre o processador alvo. Na segunda, os componentes identificados no processador são classificados e recebem uma prioridade de teste. Na terceira fase, padrões de teste são gerados de diferentes modos para os componentes visados e são transformados em diversas rotinas de auto-teste. Todas essas rotinas de auto-teste, com diferentes implementações para os mesmos componentes, constituem a biblioteca de auto-teste.

Seguindo o fluxo de projeto, após a criação da biblioteca de auto-teste deve-se planejar e desenvolver o analisador de respostas. Esse módulo realiza, sem interromper a aplicação, a comparação entre as respostas obtidas durante o teste e as respostas esperadas. Embora seja independente do processador alvo, o analisador de respostas precisa ter acesso à memória do sistema, ou do processador, onde estão armazenadas as respostas do teste. Nesse ponto do projeto as respostas esperadas ainda não são conhecidas, pois o programa de teste não foi selecionado. Porém, já é possível definir se tais respostas serão armazenadas na memória do sistema ou em uma memória ROM interna ao analisador. Assim, o analisador é desenvolvido com base na opção escolhida e já pode ser integrado ao sistema embarcado.

A última etapa do projeto de auto-teste para o processador alvo é também a mais importante, uma vez que o atendimento dos requisitos e restrições do sistema embarcado depende da seleção adequada das rotinas de teste para comporem o programa de auto-teste. Essa etapa vasculha a biblioteca de auto-teste à procura da melhor combinação de rotinas, uma para cada componente visado, que possibilite o atendimento das restrições do sistema e dos requisitos da aplicação alvo. Ao final, são obtidas as rotinas selecionadas para a composição do programa de auto-teste com seus respectivos dados, os quais serão armazenados na memória de instruções e dados do processador, respectivamente, ou na memória do sistema. Também são fornecidas as respostas esperadas para o teste selecionado, as quais serão armazenadas na memória ROM do analisador de respostas ou, opcionalmente, na memória do sistema. Por fim, nessa etapa são obtidas informações adicionais, tais como a ocupação de memória pelo programa de teste e o menor período de teste possível, em sistemas de tempo-real.

Nas próximas seções deste capítulo, cada uma das etapas do fluxo e seus respectivos produtos são detalhadamente descritos.

5.2 Biblioteca de Auto-Teste

O desenvolvimento das rotinas de auto-teste já foi descrito na Seção 4.3.3. Entretanto, o objetivo agora é a criação de uma biblioteca de teste que contenha um número razoável de rotinas diferentes para cada um dos componentes visados. Além das rotinas de auto-teste, a biblioteca deve conter as respostas de teste esperadas para cada uma das rotinas. É importante lembrar que quanto maior for a biblioteca de auto-teste, mais combinações de rotinas serão possíveis e, portanto, maiores serão as chances de se encontrar um programa de teste que satisfaça todos os requisitos e restrições do sistema e ainda permita a redução de custos. Esta seção aborda, principalmente, a escolha dos

componentes para os quais é necessário incluir rotinas de teste na biblioteca, e que abordagens utilizar no seu desenvolvimento.

O programa de auto-teste é composto por um conjunto de rotinas de teste, uma para cada componente visado. Logo, a biblioteca de teste deve conter rotinas para todos os componentes visados, preferencialmente, mais de uma para cada componente. A escolha desses componentes é o primeiro ponto importante a ser discutido. Na metodologia para o projeto de SBST estrutural, Paschalis e Gizopoulos apontaram os componentes visíveis de dados (D-VCs) como aqueles de maior prioridade para o teste. Desse modo, a biblioteca de auto-teste precisa conter, necessariamente, rotinas para cada um dos D-VCs identificados no processador alvo. São eles: componentes de processamento de dados (ULAs, deslocadores, multiplicadores, divisores, etc.) e componentes de armazenamento de dados (banco de registradores e registradores especiais ou temporários de dados).

Ainda segundo aquela metodologia, o teste dos D-VCs pode fornecer um boa cobertura de falhas para todo o processador. Porém, uma cobertura de falhas aceitável dificilmente será alcançada sem o teste adequado da unidade do controle do processador. Isso porque o teste dos D-VCs, em geral, não utiliza todo o conjunto de instruções da arquitetura. Logo, é muito importante que a biblioteca de auto-teste também contenha rotinas para o teste da unidade de controle. Tal constatação vai ao encontro da prioridade de teste definida por Paschalis e Gizopoulos. Rotinas de teste para outros componentes podem ser adicionadas mais tarde, nos casos em que a cobertura de falhas ainda estiver abaixo do esperado.

Outro ponto a ser discutido são as abordagens para o desenvolvimento das rotinas de teste. Para os D-VCs podem ser usadas as abordagens apresentadas na Seção 4.3.3 (*ATPG Imediato*, *ATPG Memória*, *LFSR* e *Determinístico Regular*) que se baseiam nas três estratégias de TPG discutidas. É recomendável que, para cada componente, sejam desenvolvidas rotinas usando todas as abordagens possíveis. Para a unidade de controle podem-se utilizar seqüências aleatórias ou determinísticas de instruções, ou mesmo uma combinação das duas, desde que todos os *opcodes* das instruções sejam aplicados. Além disso, para cada uma das rotinas da biblioteca, é indicado que se tenha uma versão com código de compactação e outra sem. O código de compactação gera uma assinatura de teste, o que reduz o número de respostas de teste, mas aumenta o tempo de execução da rotina.

A última questão relativa à biblioteca de auto-teste diz respeito ao nível em que as rotinas devem ser descritas. Paschalis e Gizopoulos, em sua metodologia para o projeto de SBST estrutural, propõem que as rotinas sejam descritas na linguagem Assembly do processador. Sem dúvida, essa é melhor forma de garantir que os padrões de teste sejam aplicados da maneira que se espera, e pelas instruções planejadas. Quando descritas em alto nível e compiladas, o código resultante da compilação pode ser bem diferente daquilo que era esperado. Isso porque os compiladores, em geral, realizam otimizações de código que alteram a estrutura inicial do programa. Por exemplo, o próprio compilador pode realizar operações aritméticas que eram esperadas serem realizadas pelo processador usando instruções com modo de endereçamento imediato.

Mesmo com esses problemas, por vezes pode ser apropriado descrever as rotinas de auto-teste utilizando uma linguagem de alto nível qualquer. Por exemplo, quando se tem uma família de processadores que contêm os mesmos componentes visíveis de dados, mas que apresentam arquiteturas diferentes, ou seja, têm instruções diferentes ou as

mesmas instruções são organizadas na memória de formas diferentes. Nesse caso, é mais eficiente descrever todas as rotinas da biblioteca em uma linguagem de alto nível, como Java, e compilar apenas aquelas que forem selecionadas para a arquitetura específica. Esse é o caso dos processadores da família Femtojava, cujo projeto do auto-teste é apresentado no Capítulo 6. No entanto, deve-se tomar cuidado para que o resultado da compilação seja equivalente à rotina planejada. Para isso é preciso conhecer o funcionamento do compilador e/ou ser possível verificar o código executável gerado. Além disso, pode ser necessário desativar algumas otimizações de compilação.

5.3 Analisador de Respostas de Teste

A análise das respostas de teste ocorre após o término de cada execução do programa de auto-teste. Nesse momento, as respostas obtidas durante o teste, armazenadas na memória RAM do processador (ou do sistema), são comparadas com as respostas esperadas, armazenadas na memória RAM do processador (ou do sistema) ou em uma memória ROM interna ao analisador de respostas. Esta seção descreve o projeto de um módulo dedicado, externo ao processador, para a análise das respostas de teste. Conforme dito anteriormente, o uso de um módulo externo evita que o próprio processador, possivelmente falho, seja responsável pela análise das respostas, o que poderia resultar na anulação do efeito de uma falha por uma outra falha, ou pela mesma. Nesse caso, uma ou ambas as falhas não seriam detectadas.

Em sistemas embarcados, o analisador de respostas de teste pode ser um núcleo do sistema que tem acesso à memória onde estão armazenadas as respostas obtidas. Embora a presença de um módulo adicional implique certo incremento de área e de dissipação de potência, ela dispensa a necessidade de qualquer modificação no projeto do processador, ou dos demais componentes do sistema. Além disso, o circuito do analisador é tão simples que o seu impacto em termos de área e potência é mínimo. Suas únicas funções consistem em acessar a memória RAM para leitura e escrita, e comparar dois valores (resposta esperada e resposta obtida), fornecendo como saída um único bit contendo o resultado da comparação (uma diferença nos valores indica a presença de uma falha).

Enquanto a necessidade de acesso para leitura da memória RAM, por parte do analisador de respostas, tem motivo óbvio, a necessidade de escrita pode não ser tão clara. Para entender tal necessidade é preciso saber que a existência de uma falha no processador pode causar quatro efeitos distintos: a escrita de um valor errado no endereço de memória correto, a escrita de um valor correto no endereço de memória errado, a escrita de um valor errado no endereço de memória errado, e a interrupção ou a execução incorreta do programa de teste. É preciso lembrar também, que o analisador de respostas faz leituras em endereços específicos da memória RAM, onde ele espera encontrar as respostas do teste.

Assim sendo, é possível que o surgimento de uma falha no processador cause a escrita da resposta de teste em um endereço de memória incorreto, ou mesmo em nenhum endereço. Nesse caso, sabendo que o programa de teste é executado periodicamente, o analisador de respostas lerá o valor correto escrito durante a execução anterior do programa de teste, e não a resposta da execução corrente. Para evitar que tal situação ocorra, ou seja, que uma resposta correta do teste anterior mascare o efeito de uma falha recente, o analisador precisa apagar todas as respostas de teste lidas, antes

que uma nova execução do programa de teste aconteça. Para tanto ele deve escrever um determinado valor de inicialização em todos os endereços lidos.

As respostas de teste esperadas, as quais serão usadas para comparação com as respostas obtidas, podem estar armazenadas tanto na memória RAM do processador (ou do sistema), junto com as respostas obtidas, quanto em uma memória ROM interna ao analisador. Nos casos em que o programa de teste não realiza teste da memória RAM, é importante a existência de um mecanismo de teste auxiliar para esse módulo (por exemplo, um BIST de memória) de forma a assegurar a integridade dos dados que serão usados pelo programa de teste, das respostas obtidas no teste, e das respostas esperadas (se for o caso). Mas vale ressaltar que o próprio programa de auto-teste do SBST pode ser adaptado para efetuar o teste *on-line* da memória RAM do processador e do sistema, assim como dos demais componentes.

Quando da aplicação de auto-teste baseado em software para teste *on-line* de sistemas embarcados, é fundamental a existência de um mecanismo de auto-teste para o analisador de respostas. Para garantir a qualidade do teste, é essencial que a realização do auto-teste do processador (assim como dos demais componentes do sistema), através do programa de teste, e a análise das respostas de teste sejam tarefas completamente independentes. Visto que o analisador de respostas é um circuito simples e pequeno, a inclusão de um método de auto-teste para esse módulo terá um impacto mínimo na área e na potência dissipada do sistema todo. Quaisquer métodos de teste *on-line* podem ser utilizados, como, por exemplo, TMR, *self-checking* ou BIST.

A Figura 5.3 exemplifica a aplicação de SBST *on-line* para vários núcleos de um SoC embarcado. Nesse exemplo, o processador responsável pelo auto-teste possui memórias internas para instruções e dados, onde estão armazenados, respectivamente, o programa e os dados para o teste de todos os componentes do sistema. As respostas obtidas do teste, bem como as respostas esperadas, são armazenadas na memória RAM do sistema, compartilhada por todos os núcleos. O analisador de respostas acessa a memória RAM, através do barramento do sistema, para leitura das respostas obtidas e esperadas, e para escrita do valor de inicialização. Além disso, ele implementa TMR em seu comparador. A saída do analisador sinaliza a detecção de uma falha no sistema. O SoC mostrado possui ainda um processador DSP e periféricos de entrada e saída. A ilustração não está representada em escala.

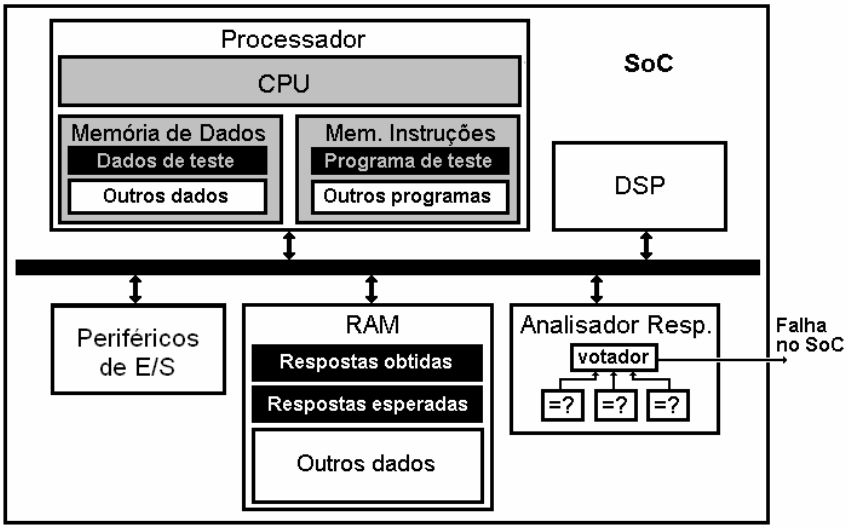

Figura 5.3: Exemplo de aplicação de SBST *on-line* para um SoC

Outro exemplo da aplicação de SBST *on-line* em um sistema embarcado é mostrado na Figura 5.4. Porém, somente o processador é testado nesse caso. Novamente o programa e os dados de auto-teste são armazenados, respectivamente, nas memórias de programa e de dados do processador. Mas, ao contrário do exemplo anterior, as respostas obtidas durante o teste também são armazenadas na memória de dados do processador. Aqui, o analisador de respostas contém uma memória ROM interna com as respostas esperadas. Ele tem acesso direto à memória de dados do processador (sem passar pelo barramento do sistema), de onde lê as respostas obtidas e onde escreve os valores de inicialização. Novamente o analisador implementa TMR em seu comparador. Sua saída sinaliza uma falha apenas no processador, pois não existe teste dos demais núcleos do sistema.

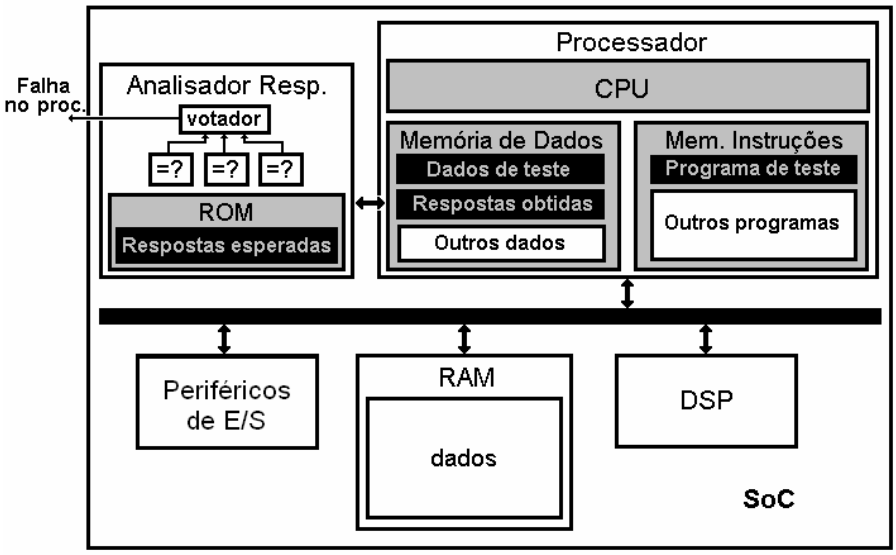

Figura 5.4: Exemplo de aplicação de SBST *on-line* para um processador em um SoC

O projeto do analisador de respostas apresentado nos próximos parágrafos baseia-se no exemplo da Figura 5.4, onde apenas o processador é testado (visto que este trabalho limita-se ao teste de processadores). Considera-se também que as respostas de teste obtidas são armazenadas na memória de dados do processador, e que as respostas esperadas são armazenadas em uma memória ROM, interna ao analisador.

Sabendo-se que as respostas obtidas de cada rotina de auto-teste podem não estar armazenadas em endereços consecutivos da memória do sistema, o analisador de respostas precisa conhecer o endereço base onde cada rotina armazena seus resultados. Além disso, sabendo-se que cada rotina pode ter um número diferente de respostas de teste, o analisador de respostas também precisa conhecer o número de respostas que cada rotina produz. O analisador precisa saber, ainda, quantas rotinas de teste o programa de auto-teste contém, e qual é o valor usado para inicialização das posições de memória onde são armazenadas as respostas obtidas, após a análise de cada resposta.

Para que o projeto do analisador de respostas tenha continuidade, é necessário definir, neste momento, onde os dados acima descritos e as respostas esperadas serão armazenados: numa memória ROM interna ao analisador ou em uma memória RAM (do processador ou do sistema). O projeto do analisador aqui exemplificado utiliza uma ROM interna para armazenamento das respostas esperadas e demais dados necessários. Embora as respostas esperadas sejam conhecidas somente após a seleção do programa de auto-teste (portanto, a memória ROM do analisador só poderá ser escrita após essa

etapa) é preciso definir de que forma os dados serão nela armazenados para que o projeto do analisador possa prosseguir. A Figura 5.5 apresenta uma possível forma de organização dos dados e respostas de teste esperadas, na memória ROM do analisador de respostas.

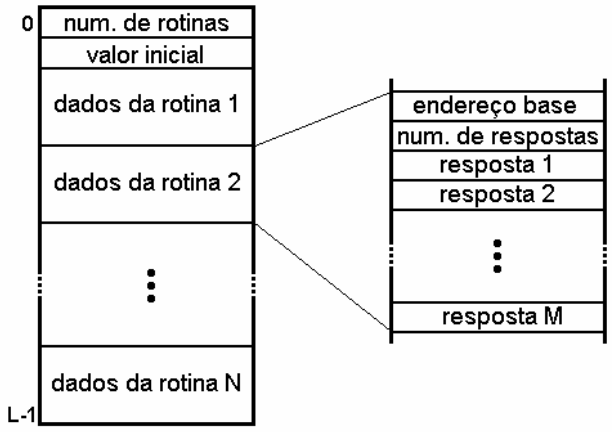

Figura 5.5: Organização dos dados na memória ROM do analisador

Neste exemplo, o endereço zero da memória contém o número *N* de rotinas que compõem o programa de auto-teste. Na segunda posição da memória ROM, é armazenado o valor de inicialização da memória de dados do processador, ou seja, o valor que é escrito nas posições de memória de onde foram lidas as respostas obtidas durante o teste. Tal valor é escrito após cada comparação de respostas. As posições seguintes armazenam os dados e as respostas esperadas para cada uma das *N* rotinas de teste. No espaço reservado para cada rotina, mostrado no lado direito da ilustração, são armazenados os seguintes dados, em ordem: o endereço base na memória RAM do processador para a leitura das respostas obtidas, o número *M* de respostas de teste que a rotina em questão produz, e as *M* respostas esperadas para esta rotina.

A largura da memória ROM é dada ou pelo tamanho do endereço base, ou pelo tamanho das respostas de teste, o que for maior. No caso de respostas não compactadas, o tamanho das respostas é, em geral, igual à largura de dados do processador. Quando compactadas, a assinatura de teste gerada pode ser maior que a largura de dados do processador. Já o tamanho *L* da memória ROM é dado pela equação:

$$L = 2 + \sum_{N-1}^{i=0} (M_i + 2) \qquad (5.1)$$

onde M_i indica o número de respostas de teste da rotina *i*, e *N* é o número de rotinas do programa de auto-teste.

Definida a organização dos dados na memória ROM, pode-se começar o projeto do analisador de respostas. Conforme já foi discutido, ele precisa executar três operações básicas: ler e escrever em posições específicas da memória RAM, e comparar dois valores. As operações de leitura e escrita requerem um somador para o cálculo do endereço alvo na memória de dados do processador, dado pela soma do endereço base da rotina com o número da resposta de teste corrente. A operação de comparação da resposta obtida com a resposta esperada requer um comparador que sinalize um bit, indicando se os valores são iguais ou diferentes. Além dessas estruturas, alguns registradores e incrementadores são necessários, bem como uma unidade de controle.

A Figura 5.6 ilustra, de maneira simplificada, a estrutura interna de uma possível implementação para o analisador de respostas. No exemplo proposto, a memória ROM é lida seqüencialmente e as informações necessárias são armazenadas em registradores específicos. Para tanto, existe um registrador, denominado ROM_COUNTER, que contém o endereço de leitura corrente da memória ROM, o qual é incrementado de um em um.

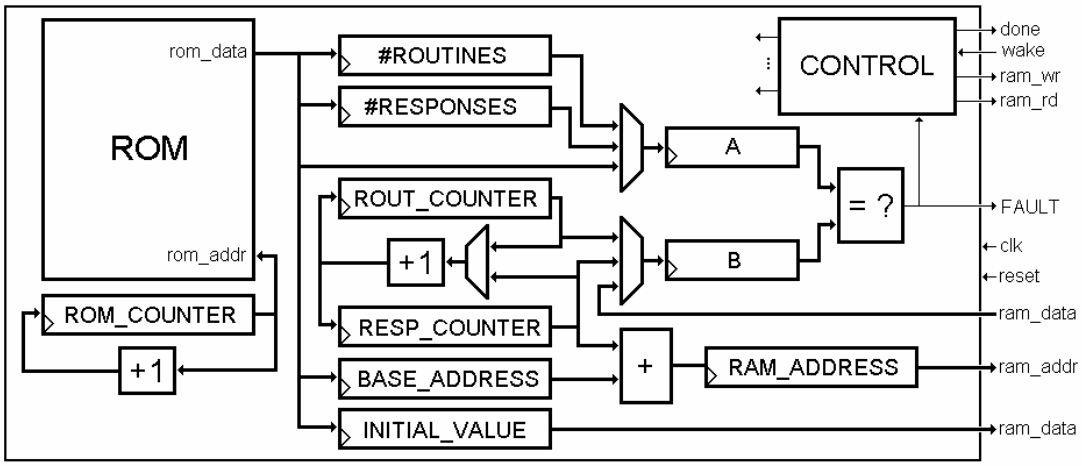

Figura 5.6: Estrutura interna do analisador de respostas

A primeira informação lida, referente ao número de rotinas do programa de auto-teste, é armazenada no registrador #ROUTINES. O valor de inicialização da memória RAM, contido na segunda posição da ROM, é lido e armazenado no registrador INITIAL_VALUE. A partir daí, os valores lidos da memória ROM e armazenados nos respectivos registradores referem-se à rotina corrente. O endereço base e o número de respostas de teste da rotina corrente são armazenados, respectivamente, nos registradores BASE_ADDRESS e #RESPONSES. Existem também um contador de rotinas (ROUT_COUNTER) e um contador de respostas (RESP_COUNTER), os quais compartilham um único incrementador. O conteúdo de RESP_COUNTER somado ao conteúdo de BASE_ADDRESS fornece o endereço para leitura ou escrita na memória de dados do processador, que é armazenado em RAM_ADDRESS. A e B são registradores temporários para os valores a serem comparados.

Todas as operações do analisador são gerenciadas pela unidade de controle (CONTROL) através dos sinais de habilitação e reset dos registradores, de seleção dos multiplexadores, e de escrita e leitura na RAM do processador. Durante a inicialização do sistema, o analisador de respostas é posto em estado de espera (*sleep*) até que receba um sinal *wake*, indicando que o programa de teste terminou sua execução e as respostas já podem ser comparadas. O fluxograma da Figura 5.7 ilustra as principais operações realizadas pelo analisador de respostas proposto. Um algoritmo detalhando todas essas operações é apresentado no Apêndice A.

Foi discutida anteriormente a importância de prover o analisador de respostas com mecanismos de auto-teste. Nota-se, porém, que o exemplo ilustrado na Figura 5.6 não contém nenhuma estrutura para esse propósito. De fato, diversos métodos de auto-teste *on-line* podem ser aplicados a esse módulo sem causar grande impacto no atendimento das restrições do sistema (tanto em termos de área quanto em termos potência ou desempenho). Uma solução bastante simples, mas pouco otimizada, é o uso de redundância de hardware (duplicação com comparação ou TMR) em todo o módulo

analisador. Soluções mais eficientes podem combinar diferentes técnicas, utilizando, por exemplo, redundância de hardware e/ou de tempo para os componentes combinacionais do circuito, e redundância de hardware e/ou de informação (*self-checking*) para os componentes de armazenamento.

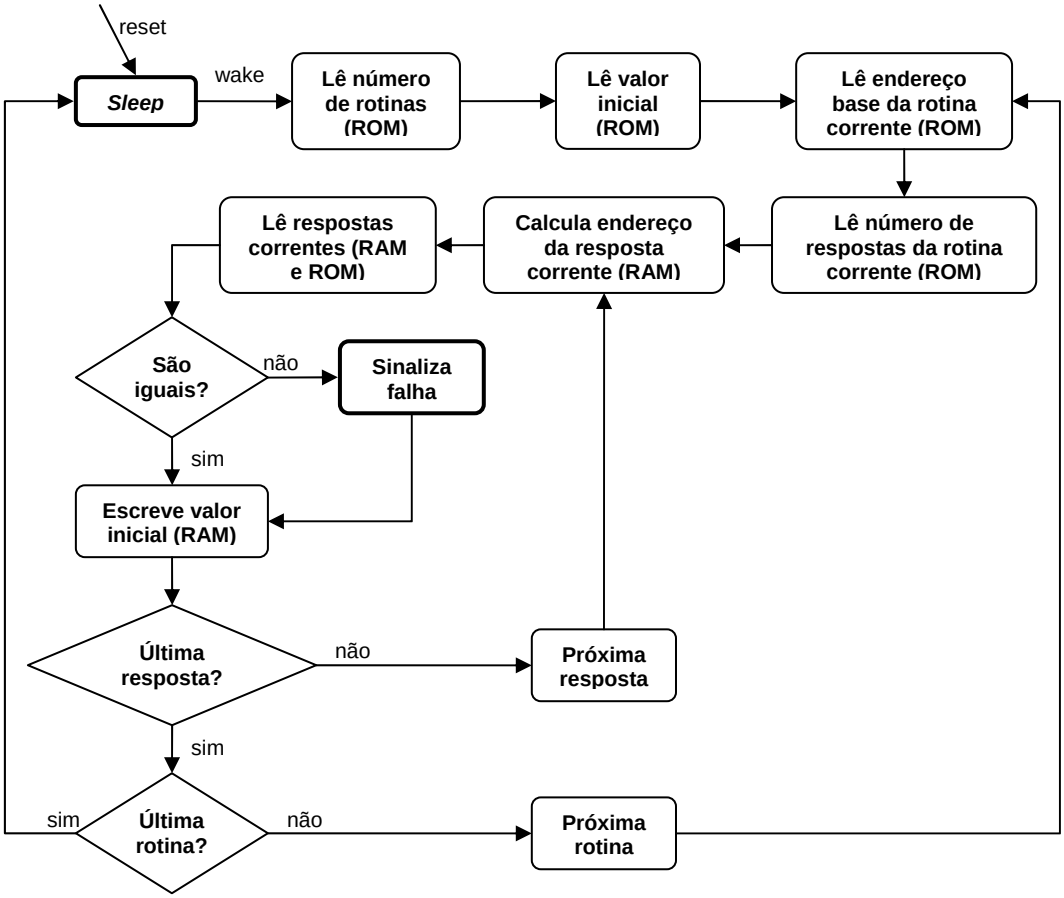

Figura 5.7: Fluxograma de execução do analisador de respostas

Na próxima seção, durante a seleção do programa de auto-teste, será considerada a aplicação de SBST *on-line* periódico para processadores embarcados em sistemas de tempo-real. Nesses casos, por motivos que serão explicados mais tarde, é preciso conhecer o tempo de execução do analisador de respostas. É claro que esse tempo é uma função do número de respostas de teste, visto que até este ponto tal valor é desconhecido. Assim, assumindo-se que a freqüência de operação do analisador de respostas é muito maior do que a freqüência da memória RAM do processador, pode-se considerar que o tempo de execução do analisador é dado pelos tempos de leitura e escrita na RAM. Desse modo, sabendo-se que são feitas uma leitura e uma escrita na memória RAM para cada resposta de teste, pode-se assumir que o tempo de execução do analisador de respostas é igual a duas vezes o número de respostas de teste, em ciclos de memória do processador. Ou seja,

$$TE_{AR} = 2 * \sum_{N-1}^{i=0} M_i \qquad (5.2)$$

onde TE_{AR} é o tempo de execução do analisador de respostas, expresso em número de ciclos da memória RAM do processador, e o resultado do somatório corresponde ao

número de respostas de teste do programa de auto-teste, sendo N o número de rotinas e M_i o número de respostas de teste da rotina i.

Outra informação necessária durante a seleção do programa de auto-teste diz respeito ao modo como o analisador de respostas acessa a memória RAM do processador. Essa memória pode ser *single port* (uma única porta de acesso) ou *dual port* (duas portas de acesso). Quando a memória RAM do processador é *dual port*, a análise das respostas de teste independe da tarefa que está sendo executada no processador, visto que não há conflitos de acesso à memória. A única restrição é que, no período corrente, a execução do programa de auto-teste tenha terminado, e que haja tempo suficiente para a análise de todas as respostas de teste.

Se a memória RAM do processador é *single port*, processador e analisador de respostas compartilham a mesma porta de acesso. Nesse caso, nenhuma tarefa da aplicação pode ser executada durante a análise das respostas de teste, com o intuito de evitar conflitos de acesso à memória. Uma rotina de gerenciamento do analisador de respostas, que não faz acessos à memória RAM, deve ser incluída na tarefa de teste após as rotinas de auto-teste de cada componente do processador. Assim, para efeito de tempo, a execução dessa rotina equivale à execução do analisador de respostas. Com isso, evita-se a ocorrência de conflitos de acesso à memória. A importância dessa informação será explicada na próxima seção, no final do processo de seleção do programa de auto-teste.

5.4 Seleção do Programa de Auto-Teste

A terceira e última etapa da metodologia para o projeto de auto-teste *on-line* periódico para processadores embarcados consiste na seleção das rotinas de teste, dentre aquelas disponíveis na biblioteca de auto-teste, para composição do programa de auto-teste. O uso de um método eficiente para a seleção das rotinas de teste garante o atendimento dos requisitos e restrições do sistema embarcado e/ou de tempo-real, ao contrário das regras de aplicabilidade propostas por Paschalis e Gizopoulos (PASCHALIS; GIZOPOULOS, 2005) e apresentadas na Seção 4.3.3.4. A importância de tal método de seleção será reforçada no Capítulo 6, quando resultados práticos de sua aplicação serão comparados ao uso das regras de Paschalis e Gizopoulos.

No Capítulo 4 foram referenciadas algumas pesquisas relacionadas ao uso de SBST para o auto-teste de processadores. O objetivo principal de tais trabalhos é a otimização absoluta das rotinas de auto-teste em termos de custo de desenvolvimento, ocupação de memória, tempo de execução do teste, e cobertura de falhas resultante para os componentes visados. Entretanto, nenhum desses trabalhos visa a seleção efetiva de diferentes abordagens de rotinas de teste para a composição do programa de auto-teste. Nem mesmo endereçam o impacto da aplicação de SBST *on-line* em sistemas embarcados e/ou de tempo-real.

Tais sistemas podem apresentar sérias limitações para a aplicação de auto-teste *on-line* periódico devido às fortes restrições de projeto discutidas nas Seções 3.2.2 e 3.2.3 (pequena memória disponível no sistema; capacidade da bateria do sistema, que limita a energia que pode ser consumida; curto tempo disponível para o teste, devido às características de tempo-real do sistema; requisitos do teste, como cobertura de falhas e período de teste). Assim, a definição de um programa de teste adequado, considerando as restrições citadas, é um desafio adicional no projeto do auto-teste *on-line* para

processadores embarcados. Daí a importância de um método eficiente para a seleção do programa de auto-teste.

Nesta seção são propostos dois métodos para a seleção do programa de auto-teste: um pseudo-exaustivo, para bibliotecas de auto-teste pequenas; outro heurístico, para bibliotecas grandes. Ambos os métodos permitem a exploração do espaço de teste através da seleção de um conjunto de rotinas que implementam diferentes abordagens de teste (discutidas na Seção 4.3.3). Os métodos propostos asseguram não apenas a execução periódica do programa de teste, mas também o atendimento dos requisitos e restrições do sistema em termos de tamanho de memória, de energia consumida, e dos *deadlines* das tarefas de tempo-real (se for o caso).

Ambos os métodos, pseudo-exaustivo e heurístico, baseiam a seleção do programa de auto-teste em dois critérios principais: o conjunto de restrições do sistema e uma função de custo. O conjunto de restrições do sistema pode incluir limitações no tamanho da memória de dados, no tamanho da memória de programa, no consumo de energia total e no período de teste. Para os métodos de seleção, qualquer desses fatores de custo pode ser limitado por restrições do sistema, ou pode não ter restrições. A função de custo deve ser fornecida pelo projetista do sistema, e também pode incluir quaisquer dos quatro fatores de custo citados. Nessa função, um peso é associado a cada fator de custo, representando a importância de tal fator no projeto do sistema.

O espaço de busca dos métodos de seleção consiste no conjunto de todos os programas de auto-teste candidatos, ou seja, todas as possíveis combinações de rotinas de teste que formam programas de auto-teste (entenda-se uma rotina para cada componente visado do processador). O objetivo dos métodos é encontrar, no espaço de busca, um programa de auto-teste que satisfaça todas as restrições do sistema, e que apresente um baixo valor para a função de custo. Desse modo, diz-se que foi encontrada a solução ótima quando o programa de auto-teste selecionado é aquele, dentre todos os candidatos, com o menor valor para a função de custo e que permite o atendimento de todas as restrições do sistema.

Para atingir seu objetivo, ambos os métodos necessitam de informações sobre as rotinas existentes na biblioteca de auto-teste, sobre as tarefas da aplicação (seja ela de tempo-real ou não), e sobre os requisitos do sistema. Uma vez que os custos da aplicação devem ser previamente conhecidos, pressupõe-se que não há criação dinâmica de tarefas. Antes do início do processo de seleção, a execução de todas as rotinas de auto-teste deve ser simulada no processador alvo, para obtenção de seus custos em termos de ocupação de memória de dados e de programa, de consumo de energia, e de tempo de execução. De modo similar, as tarefas da aplicação devem ser analisadas para a obtenção de seus tempos de execução no pior caso, de sua ocupação de memória, e de seu consumo de energia. É importante ressaltar que fatores de custo não limitados por requisitos do sistema, nem incluídos na função de custo, não precisam ser fornecidos.

Em aplicações de tempo-real, devido à necessidade de determinismo temporal do sistema, informações adicionais precisam ser fornecidas. Por exemplo, é preciso conhecer o nível de utilização do processador pelas tarefas da aplicação. Se todas as tarefas são periódicas, essa informação pode ser facilmente obtida através do tempo de execução e período de cada tarefa (ver Seção 3.2.3). O período e o *deadline* das tarefas são definidos na especificação da aplicação. Outro dado necessário é o tempo de execução do analisador de respostas, dado em função do número de respostas de teste, para o cálculo do nível de utilização do processador pela tarefa de auto-teste. Por fim, é

preciso conhecer a política de escalonamento utilizada pelo sistema, e o nível máximo de utilização do processador que tal política admite. Todos esses dados são necessários tanto para a verificação do requisito de período de teste, quanto para o cálculo do menor período de teste que garante o atendimento dos *deadlines* da aplicação. Este último é um dos dados de saída dos métodos, e será explicado em breve.

Ao final do processo de seleção, ambos os métodos fornecem o conjunto de rotinas de teste selecionado (para composição do programa de auto-teste) com suas respectivas respostas de teste esperadas (para escrita da memória ROM do analisador de respostas), os custos totais do sistema para os fatores considerados, e o menor período de teste admitido pela aplicação de tempo-real (se for o caso).

A seguir, os métodos pseudo-exaustivo e heurístico para a seleção de programas de auto-teste são propostos. Aqui, eles são descritos de forma bastante genérica para permitir que cada projetista adeque o processo de seleção às necessidades de seu projeto. No Capítulo 6, os detalhes de implementação de ambos os métodos são exemplificados , nesse caso, para se adequarem às necessidades do projeto de auto-teste para os processadores Femtojava.

5.4.1 Método Pseudo-Exaustivo

O método de seleção aqui proposto (MORAES; COTA; CARRO; WAGNER; LUBASZEWSKI, 2005-a; MORAES; COTA; CARRO; WAGNER; LUBASZEWSKI, 2005-b) é dito pseudo-exaustivo porque realiza uma busca exaustiva pelo programa de auto-teste que apresenta o menor valor para a função de custo. Porém, antes dessa busca exaustiva, ele elimina do espaço de busca os candidatos que não permitem o atendimento das restrições do sistema. Esse tipo de pesquisa pode oferecer uma boa redução do espaço de busca dependendo dos valores limite para os fatores de custo restringidos. Note-se, porém, que a eliminação de soluções implica que o método percorra todas as possibilidades ao menos uma vez, para verificação do primeiro requisito analisado.

O processo de seleção do programa de auto-teste pelo método pseudo-exaustivo é ilustrado na Figura 5.8. Como já foi mencionado, o método recebe como entradas os custos de cada uma das rotinas da biblioteca de auto-teste e de cada tarefa da aplicação, bem como as restrições do sistema e a função de custo que deve ser minimizada. Além desses dados, para aplicações de tempo-real, também devem ser fornecidos o tempo de execução do analisador de respostas e o nível máximo de utilização do processador permitido pela política de escalonamento implementada no sistema embarcado alvo. O processo de seleção ocorre em sete passos (*steps*), descritos a seguir.

O primeiro passo do método de seleção, denominado *Cálculo de Custos*, é responsável pela construção do espaço de busca. Ele é um passo exaustivo porque identifica cada uma das soluções montando todas as possíveis combinação de rotinas de auto-teste e calculando seus custos para todos os fatores fornecidos. Tais fatores podem incluir tamanho de memória de dados (RAM), tamanho de memória de programa (ROM), tempo de execução e energia consumida. Cada combinação é um programa de teste candidato, contendo exatamente uma rotina de auto-teste para cada componente visado do processador. Os custos de cada candidato são calculados a partir dos custos individuais das rotinas combinadas para formá-lo, e são armazenados para uso nos próximos passos.

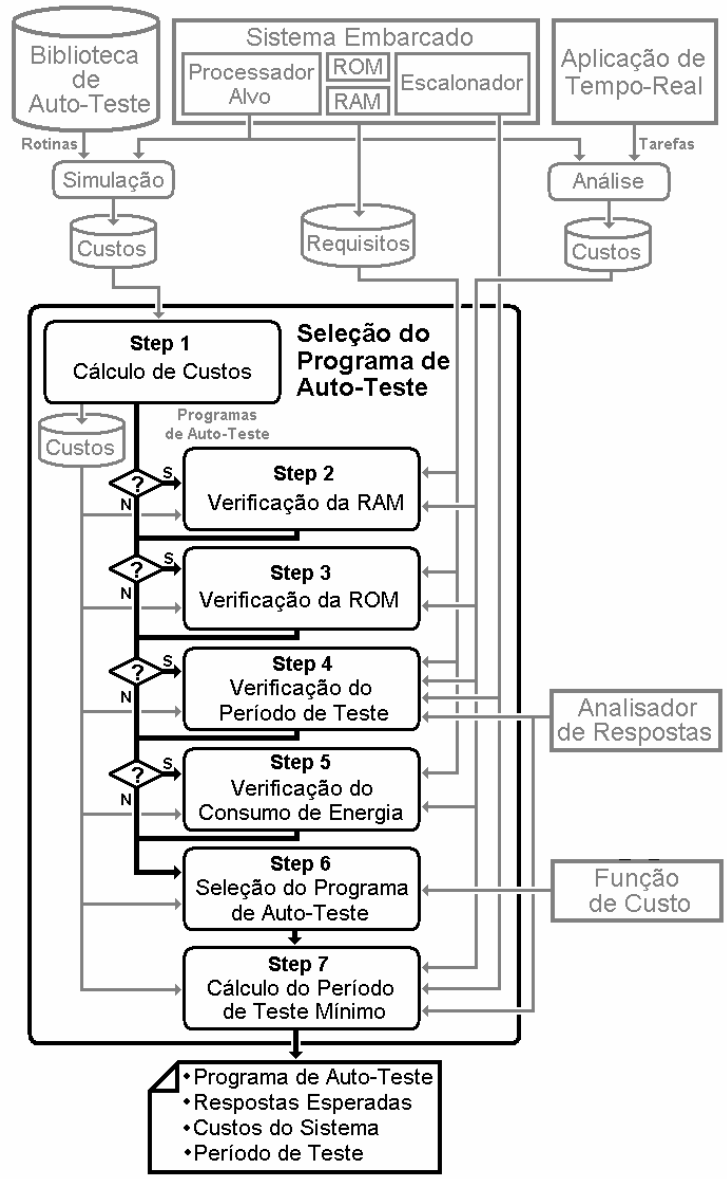

Figura 5.8: Fluxo de seleção do programa de teste pelo método pseudo-exaustivo

Os passos de dois a cinco são condicionais, e sua execução depende da existência de restrições de sistema para cada um dos respectivos fatores de custo. As tomadas de decisão, indicadas por interrogações na Figura 5.8, representam a verificação da existência de um valor limite para um dado fator de custo. A existência de uma restrição para dado fator leva à execução do passo correspondente à verificação de atendimento da referida restrição. A inexistência de tal restrição causa a não execução do passo correspondente.

O segundo passo, *Verificação da RAM*, é executado sempre que o sistema apresenta uma restrição no tamanho da memória de dados. Nesse caso, todos os programas de teste candidatos são analisados para a pré-seleção daqueles que cabem na memória RAM do sistema, junto com os dados da aplicação. Esse passo, quando executado, realiza a primeira redução do espaço de busca, eliminando as soluções que não atendem o requisito de memória RAM. Apenas os candidatos restantes passam para o próximo passo.

O passo de *Verificação da ROM* é executado sempre que o sistema é restrito em relação ao tamanho da memória de programa. As soluções aprovadas na etapa anterior passam, então, por uma nova seleção. Desta vez são eliminados os candidatos que não cabem na memória ROM do sistema, junto com as tarefas da aplicação. Novamente apenas as soluções restantes são encaminhadas para o próximo passo.

O passo de número quatro, *Verificação do Período de Teste*, é válido apenas para aplicações de tempo-real, e quando há um requisito de sistema que impõe um limite máximo para o período de teste. Nesses casos, as soluções aprovadas na etapa anterior passam pelo teste de aceitação do escalonador. O objetivo desse teste é selecionar os programas de auto-teste que podem ser executados com um período menor ou igual ao valor limite, sem impedir o atendimento de todos os *deadlines* das tarefas da aplicação. O teste de aceitação do escalonador verifica se o nível de utilização do processador pelo auto-teste (programa de auto-teste e análise das respostas de teste) somado ao nível de utilização do processador pela aplicação é menor ou igual ao valor permitido pelo algoritmo de escalonamento implementado no sistema. As soluções reprovadas são eliminadas do espaço de busca, enquanto as demais passam para a próxima etapa.

O quinto passo, *Verificação do Consumo de Energia*, é a última etapa condicional do método. Sua execução depende da existência de uma restrição no consumo de energia de sistema. Nesse passo, verificam-se quais das soluções previamente selecionadas têm um consumo de energia tal que, quando somado à energia consumida pela aplicação, resulte em um valor menor ou igual ao limite admitido pelo sistema. Nesse momento é feita a última redução no espaço de busca. As soluções restantes atendem todas as restrições do sistema.

Pode acontecer de, em algum dos passos anteriores, o espaço de busca ser reduzido a um conjunto vazio. Isso significa que nenhum dos programas de auto-teste candidatos permite o atendimento de todas as restrições do sistema. O processo de seleção é então interrompido, informando que não existe solução possível. Nesse caso, o ideal é permitir o relaxamento de um ou mais dos requisitos do sistema e executar o método novamente, até que uma solução seja encontrada. Se o sistema possuir restrições muito fortes, as quais não podem ser alteradas, deve-se então modificar a aplicação e/ou a biblioteca de auto-teste, na tentativa de reduzir seus custos.

Feitas todas as possíveis reduções no espaço de busca e restando nele candidatos, é tarefa do sexto passo encontrar a melhor solução. A *Seleção de Programa de Auto-Teste* é realizada com base nos valores de cada candidato para a função de custo fornecida. Para tanto, a função de custo é calculada para cada uma das soluções restantes e aquela que resultar no menor valor será a escolhida. Apenas o programa de auto-teste selecionado passa para a próxima etapa. Note-se que a solução encontrada é sempre a solução ótima, uma vez que este passo é exaustivo para o espaço de busca final, o qual inclui apenas os candidatos que satisfazem todas as restrições do sistema.

Finalmente, o sétimo e último passo do processo de seleção realiza o *Cálculo do Período de Teste Mínimo*. Este passo, assim como o quarto, é válido apenas para aplicações de tempo-real e seu objetivo é fornecer um dado adicional de grande relevância para o projetista do sistema. Ele calcula o menor período de teste, para o programa de auto-teste selecionado, que permite o atendimento de todos os *deadlines* das tarefas da aplicação. Com isso, pode-se garantir que o teste será executado tão freqüentemente quanto o permitido pela aplicação e pela política de escalonamento. Essa etapa novamente faz uso do teste de aceitação do escalonador.

Aqui vale uma consideração acerca de como são calculados o período e o *deadline* da tarefa de teste em função do tipo de memória RAM do processador. Já foram discutidas na seção anterior as implicações do uso de uma memória *single port* ou *dual port*. Se a memória RAM do processador é *single port*, a análise das respostas de teste faz parte da tarefa de teste. Então, para o cálculo do período de teste (que é igual ao *deadline*) utiliza-se o teste de aceitação do escalonador, considerando-se que o tempo de execução do teste equivale à soma do tempo de execução do programa de auto-teste com o tempo de execução do analisador de respostas.

Por outro lado, quando a memória RAM é *dual port*, a análise das respostas não faz parte da tarefa de teste. Nesse caso, o *deadline* da tarefa de teste deve ser menor do que o seu período, para que haja tempo suficiente para a análise das repostas de teste após o término da execução do programa de auto-teste e antes do próximo período de teste. Assim, o *deadline* da tarefa de teste é calculado por meio do teste de aceitação do escalonador, considerando-se apenas o tempo de execução do programa de auto-teste. Para o cálculo do período de teste, basta adicionar ao *deadline* o tempo de execução do analisador de respostas.

Ao final do sétimo passo, o método informa a rotina de teste selecionada para cada um dos componentes alvo do processador, com suas respectivas respostas de teste esperadas. A partir dessas informações são construídos o programa de auto-teste e a memória ROM do analisador de respostas. Além disso, são fornecidos os custos totais do sistema (teste mais aplicação) para os fatores considerados. Tais custos foram calculados nos passos de dois a cinco, durante a verificação das restrições do sistema. Finalmente, é informado o período de teste mínimo, calculado no passo sete. As operações realizadas em cada passo do método de seleção, em especial nos passos quatro e sete, poderão ser melhor compreendidas no Capítulo 6 (Seção 6.4), quando será apresentado um exemplo de implementação para esse método.

É importante lembrar que o método pseudo-exaustivo, aqui proposto para seleção do programa de auto-teste, é recomendado para bibliotecas de auto-teste pequenas. Isso se deve, primeiramente, à redução no espaço de busca que pode ser, em alguns casos, insignificante ou nula considerando-se que a existência de restrições do sistema não é obrigatória. Mesmo que haja tais restrições, elas podem não implicar a eliminação de muitos candidatos. Nesses casos, a função de custo deverá ser calculada para um grande número de soluções, o que pode ser inaceitável para determinada capacidade de processamento. Em segundo lugar, deve-se ter em mente que, mesmo com uma grande redução no espaço de busca, os primeiros passos do processo são executados sobre todas as possíveis soluções. Esses passos incluem o cálculo de custos e a verificação de, no mínimo, um requisito de sistema.

Na tentativa de solucionar esses problemas foi desenvolvido um segundo método para seleção do programa de auto-teste, baseado em uma busca heurística. Tal método é descrito na próxima seção.

5.4.2 Método Heurístico

O número de soluções possíveis para os métodos de seleção do programa de auto-teste é uma função exponencial do número de componentes do processador para os quais foram desenvolvidas rotinas de teste. Assumindo-se que a biblioteca de auto-teste contém o mesmo número S de diferentes rotinas para cada um dos C componentes

visados, então o tamanho dessa biblioteca (o número de rotinas que ela contém) é dado pela equação

$$Tam_{BT} = S * C \qquad (5.3)$$

enquanto o tamanho do espaço de busca (número de soluções) que ela origina é dado por

$$Tam_{EB} = S^{C} \qquad (5.4)$$

Tomemos como exemplo o projeto de um sistema embarcado cujo processador a ser testado tem uma arquitetura VLIW (*Very Long Instruction Word*) (FISHER, 1984), como o DSP TMS320C62x™ (TEXAS INSTRUMENTS PRODUCTS, 2006), da Texas Instruments. Esse processador possui oito unidades funcionais independentes, e dois bancos de registradores. Se, para o projeto do auto-teste desse processador, forem desenvolvidas rotinas de teste apenas para os D-VCs (unidades funcionais e bancos de registradores), teremos *C=10*. Se, para cada componente, forem utilizadas quatro das abordagens para o desenvolvimento de rotinas de teste apresentadas na Seção 4.3.3, teremos *S=4*. Nesse caso, a biblioteca de auto-teste terá 40 rotinas e o espaço de busca dos métodos de seleção terá 4^{10} (ou 2^{20}) soluções. O cálculo de custos para todas essas soluções demandaria uma grande quantidade de processamento.

Se formos um pouco além, podemos considerar que, para possibilitar uma maior exploração do espaço de teste, cada rotina da biblioteca de auto-teste foi desenvolvida em duas versões: uma com código de compactação e outra sem. Nesse caso, em que *S=8*, a biblioteca de auto-teste conteria 80 rotinas e o espaço de busca teria 8^{10} (ou 2^{30}) soluções. E se, para aumentar a cobertura de falhas total do processador, fosse necessário testar outros componentes, seguindo a prioridade de teste definida na Seção 4.3.2, esse número cresceria com um fator exponencial. Com um espaço de busca de tal proporção, uma avaliação exaustiva de custos e o cálculo da função de custo, mesmo que para metade das soluções, seria impraticável em um computador com uma capacidade de processamento razoável.

Para lidar com tais situações extremas, foi proposto um método de busca heurístico para a seleção do program de auto-teste (MORAES; COTA; CARRO; LUBASZEWSKI, 2005). O método heurístico é iterativo, sendo cada iteração denominada passo (*step*). Portanto, uma dada execução do método pode ter um número qualquer de passos, diferente do método anterior que consiste em exatamente sete passos, embora alguns possam ser ignorados em uma dada execução. Esse método é mais eficiente para bibliotecas de auto-teste grandes porque, ao contrário do método pseudo-exaustivo, o cálculo de custos é feito sob demanda, apenas para o programa de auto-teste pré-selecionado. Essa pré-seleção é feita com base apenas na função de custo fornecida pelo projetista e nos custos individuais das rotinas de teste da biblioteca.

O processo de seleção de um programa de auto-teste pelo método heurístico é ilustrado na Figura 5.9. Novamente, é preciso conhecer os custos individuais de cada uma das rotinas de teste, bem como os custos das tarefas da aplicação e as restrições do sistema. Também são necessários a função de custo fornecida pelo projetista e, para aplicações de tempo-real, o tempo de execução do analisador de respostas e o nível máximo de utilização do processador permitido pela política de escalonamento implementada no sistema. O processo de seleção pelo método heurístico é composto por um laço principal no qual são executadas quatro tarefas: *Pré-Seleção do programa de Auto-Teste*, *Verificação de Requisito*, *Eliminação de Rotina* e *Ajuste de Pesos*.

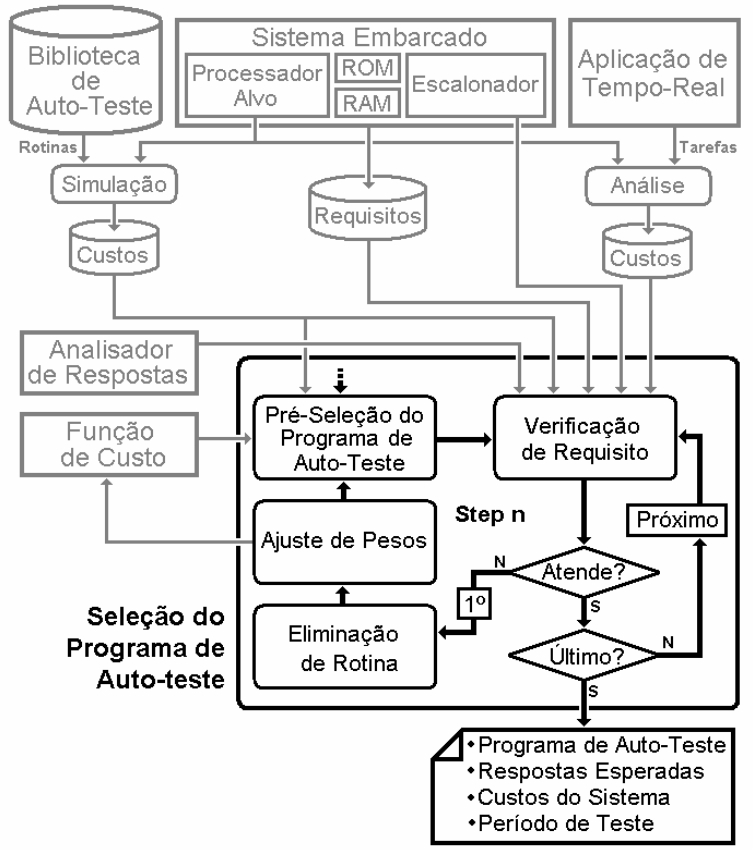

Figura 5.9: Fluxo de seleção do programa de teste pelo método heurístico

A primeira tarefa a ser executada é a *Pré-Seleção do Programa de Auto-Teste*, que segue o critério do menor valor para a função de custo. Porém, diferentemente do método pseudo-exaustivo, a função de custo é calculada para cada rotina da biblioteca (*S*C* rotinas), e não para cada programa de auto-teste candidato (S^C candidatos). Na verdade, o conceito de programa de auto-teste candidato não existe neste método, uma vez que as soluções são construídas uma por vez, e sob demanda. Depois de calculada a função de custo para cada uma das rotinas de teste, é identificada, para cada componente visado do processador, a rotina que tem o menor valor calculado. As rotinas identificadas compõem, então, o programa de auto-teste pré-selecionado, ou programa de auto-teste corrente. Note-se, porém, que mesmo atendendo todas as restrições do sistema o programa de auto-teste pré-selecionado não é, necessariamente, a solução ótima. Isso ocorre porque a minimização da função de custo é local (para as rotinas), e não global (para os programas de teste). De qualquer forma ela se aproxima da solução ótima.

O programa de auto-teste pré-selecionado passa então para a etapa de *Verificação de Requisito*. Apenas uma das restrições do sistema é verificada a cada execução dessa etapa, em uma ordem pré-estabelecida. A próxima restrição será verificada apenas na próxima execução dessa etapa, que ocorrerá, para o mesmo programa de auto-teste corrente, se a restrição atual for atendida. Esse processo é ilustrado no laço secundário, à direita na Figura 5.9. A verificação de cada requisito para o programa de auto-teste corrente ocorre de maneira similar àquela realizada nos passos de dois a cinco do método pseudo-exaustivo. A única diferença acontece no requisito de período de teste. Aqui, primeiro é calculado o período de teste mínimo e depois esse valor é comparado com o valor limite aceito para o sistema. O mesmo ocorre para os demais requisitos,

como no outro método de seleção: primeiro calcula-se o custo total do sistema para o fator atual, e depois se compara esse valor com a restrição do sistema.

No cálculo de período de teste mínimo, assim como no método pseudo-exaustivo, deve-se considerar o tipo de memória RAM que o sistema utiliza: *single port* ou *dual port*. Para memória *single port*, o *deadline* da tarefa de teste é igual ao seu período, em cujo cálculo considera-se como tempo de teste o tempo de execução do programa de auto-teste somado ao tempo de execução do analisador de respostas. Para memória *dual port*, o *deadline* é calculado tendo-se como tempo de teste apenas o tempo de execução do programa de auto-teste. Nesse caso, o período de teste é dado pela soma do *deadline* com o tempo de execução do analisador de respostas.

Caso o programa de auto-teste pré-selecionado não atenda o requisito atual, o contador de requisitos volta para o primeiro da ordem pré-estabelecida e o processo de seleção segue para a etapa de *Eliminação de Rotina*. Uma vez que o programa de auto-teste corrente não é uma solução possível, o objetivo dessa etapa é evitar que esse mesmo programa seja novamente pré-selecionado. Pretende-se também, com essa etapa, otimizar o processo de seleção reduzindo o número de iterações necessárias. Para tanto, uma das rotinas do programa de auto-teste corrente é *virtualmente* eliminada da biblioteca de auto-teste. A rotina eliminada é aquela que, dentre as rotinas para o mesmo componente existentes na biblioteca, tem o maior custo para o fator cuja restrição não foi atendida. Note-se que essa etapa pode, eventualmente, eliminar do espaço de busca a solução ótima (que combinaria a rotina eliminada com outras que não as atualmente selecionadas). Entretanto, esse é um risco pequeno em relação ao benefício de se ter o espaço de busca bastante reduzido, reduzindo também o número de passos para a seleção do programa de auto-teste.

Após a eliminação virtual de uma das rotinas da biblioteca de auto-teste, o processo de seleção segue para a quarta e última etapa do laço. O *Ajuste de Pesos* visa acelerar a escolha de um programa de auto-teste que atenda todas as restrições do sistema. Para tanto, é feito um reajuste dos pesos de cada fator na função de custo. Assim, o peso do fator de custo cuja restrição não foi atendida é incrementado por um valor predefinido. Para manter o equilíbrio da função de custo, o aumento no valor de um peso deve ser descontado dos demais. Por isso o peso dos outros fatores é decrementado por frações iguais do valor do incremento. É importante que o valor predefinido para o ajuste de pesos seja grande o suficiente para acelerar o processo de seleção, mas também pequeno o suficiente para evitar uma discrepância muito grande da função de custo final em relação à inicial.

Feito o ajuste de pesos, um novo programa de auto-teste é pré-selecionado tendo como critério a minimização da nova função de custo. Então, todo o processo é repetido até que se encontre uma solução que atenda a todos os requisitos do sistema, ou até que não restem mais soluções possíveis no espaço de busca. No primeiro caso, o método fornece o conjunto de rotinas de teste para a composição do programa de auto-teste e suas respectivas respostas esperadas, para a escrita da memória ROM do analisador de respostas. Também são informados os custos totais do sistema para os fatores considerados e o período de teste mínimo, todos calculados na etapa de *Verificação de Requisito*. Vale ressaltar que, em alguns casos testados na aplicação deste método, a solução encontrada é ótima, e nos demais é próxima a ela.

No caso do processo de seleção terminar pela falta de rotinas na biblioteca de auto-teste para um dos componentes do processador, o método informa que não foi capaz de

encontrar uma solução possível. Note-se que isso não significa que não haja soluções possíveis, uma vez que rotinas foram eliminadas da biblioteca sem que todas as combinações com ela fossem testadas. Nesse caso, recomenda-se a execução do método pseudo-exaustivo para se ter certeza de que não existem soluções que atendem todas as restrições do sistema. Se a inexistência de soluções for confirmada, deve-se proceder com o relaxamento das restrições ou com a otimização de rotinas da biblioteca de auto-teste, antes de uma nova tentativa com um dos dois métodos de busca.

Todas as questões relativas à implementação do processo heurístico de seleção do programa de auto-teste serão exemplificadas no Capítulo 6 (Seção 6.4), quando será descrita a implementação do método para o projeto de auto-teste *on-line* periódico para os processadores da família Femtojava. Assim, um melhor entendimento de cada etapa do processo poderá ser obtido lá.

Conhecidos os dois métodos para a seleção do programa de auto-teste é preciso estabelecer critérios para a utilização de um ou outro. Como se viu, o método pseudo-exaustivo fornece a solução ótima sempre que ela existir, mas realiza algumas operações exaustivamente, ou seja, para todas as soluções do espaço de busca. Por outro lado, o método heurístico jamais opera sobre todo o espaço de busca, mas nem sempre fornece uma solução (mesmo que exista), e quando fornece pode não ser a solução ótima.

Conforme foi dito anteriormente, o método pseudo-exaustivo é indicado para bibliotecas de auto-teste pequenas, enquanto o método heurístico é indicado para bibliotecas grandes. Porém, os conceitos de *pequeno* e *grande* são muito vagos. Na verdade, a escolha do método não depende diretamente do tamanho da biblioteca de auto-teste, e sim do tamanho do espaço de busca que ela origina. Outros fatores importantes são a capacidade de processamento da máquina na qual o método será executado e o tempo disponível para sua execução. Assim, deve-se ter em mente que o método pseudo-exaustivo calcula os custos de todas as soluções do espaço de busca e que, no pior caso, pode também verificar os requisitos e calcular a função de custo para todas as soluções. Por outro lado, a operação mais complexa do método heurístico é o cálculo da função de custo para todas as rotinas da biblioteca de auto-teste.

Os tamanhos da biblioteca de auto-teste e do espaço de busca são definidos, respectivamente, pelas Equações 5.3 e 5.4 já apresentadas. Tais equações mostram que o número de soluções cresce exponencialmente com o número de componentes do processador a serem testados, enquanto o tamanho da biblioteca cresce linearmente com esse fator. Portanto, a escolha entre a utilização de um ou outro método de seleção é uma decisão de projeto que deve ser baseada em todas essas informações.

A seleção do programa de auto-teste encerra a metodologia de projeto de auto-teste *on-line* periódico para processadores embarcados. Com as informações fornecidas nessa etapa, pode-se escrever a tarefa de teste (programa de auto-teste) na memória de programa do processador e escrever as respostas de teste esperadas na memória ROM do analisador de respostas. Então o analisador de respostas pode ser integrado ao sistema embarcado. No caso de uma aplicação de tempo-real, o período e o *deadline* da tarefa de teste, já conhecidos, podem ser fornecidos ao escalonador do sistema. Com isso, o sistema embarcado está pronto para realizar o auto-teste *on-line* periódico de seu processador, respeitando todos os requisitos e restrições impostos pelo projeto, e alcançando uma boa qualidade de teste.

5.5 Resumo e Conclusões

Neste capítulo foi proposta uma metodologia bastante genérica para o projeto de auto-teste *on-line* periódico para processadores embarcados. Tal proposta, denominada STEP, é baseada na metodologia para o projeto de SBST estrutural, de Paschalis e Gizopoulos (PASCHALIS; GIZOPOULOS, 2005), no que tange o desenvolvimento das rotinas de teste. Porém, foram acrescentados o projeto de um módulo para a análise *on-line* das respostas de teste e dois métodos para a seleção efetiva de rotinas para a composição do programa de auto-teste. O fluxo de projeto proposto inclui três etapas: desenvolvimento da biblioteca de auto-teste, desenvolvimento do analisador de respostas e seleção do programa de auto-teste.

A etapa de desenvolvimento da biblioteca de auto-teste é constituida pelas três fases da metodologia de Paschalis e Gizopoulos: extração de informações, classificação dos componentes e prioridade de teste, e desenvolvimento das rotinas de auto-teste. Porém, propõe-se que sejam desenvolvidas tantas rotinas de teste quanto possível para cada componente visível de dados (D-VC) do processador, bem como para a unidade de controle. Com o teste desses componentes é possível atingir, na maioria dos casos, uma cobertura de falhas aceitável para todo o processador. Quanto mais rotinas existirem na biblioteca de auto-teste, maiores serão as chance de haver um programa de auto-teste que satisfaça todos os requisitos e restrições do sistema, e que permita, ainda, a redução de custos.

A segunda etapa da metodologia proposta consiste no desenvolvimento do módulo responsável pela comparação das respostas obtidas durante o teste com as respostas esperadas, uma vez que tal comparação deve ser realizada *on-line*. O analisador de respostas deve ter acesso à memória onde serão armazenadas as respostas do teste, bem como à memória que contém as respostas esperadas. Ele executa três operações principais: leitura e escrita na memória RAM que contém as respostas do teste, e a comparação das respostas obtidas com as respostas esperadas. Sabendo-se que o analisador de respostas deve ser auto-testável, para evitar a sobreposição do efeito de falhas, é indispensável que esse módulo seja provido de algum mecanismo de auto-teste *on-line*. Uma possível implementação de um módulo analisador de respostas foi proposta neste capítulo.

A terceira e última etapa do projeto de auto-teste *on-line* periódico para processadores embarcados resume-se à seleção do programa de auto-teste, a partir das rotinas existentes na biblioteca de auto-teste. Para o processo de seleção foram propostos dois métodos: um pseudo-exaustivo e outro heurístico. O critério de seleção de ambos os métodos é o atendimento de todas as restrições do sistema e a minimização de uma função de custo, fornecida pelo projetista do sistema. Por isso, os métodos requerem informações sobre os custos das rotinas de teste e das tarefas da aplicação, bem como os requisitos do sistema e a função de custo a ser minimizada. Para aplicações de tempo-real é preciso conhecer, ainda, o tempo de execução do analisador de respostas em função do número de respostas de teste, e o nível máximo de utilização do processador permitido pela política de escalonamento implementada.

O método de seleção pseudo-exaustivo elimina do espaço de busca as soluções que não atendem todas as restrições do sistema. Sobre as soluções restantes é aplicada a função de custo, e aquela que apresentar o menor valor para a função é a solução escolhida. Esse método garante que o programa de auto-teste selecionado é a solução ótima, ou seja, atende todos os requisitos do sistema e tem o menor valor para a função

de custo. O processo de seleção é executado em sete passos. O primeiro passo constrói o espaço de busca, calculando também os custos de todas as possíveis soluções. Os passos de dois a cinco são condicionais, e responsáveis pela verificação das restrições de memória de dados, memória de programa, período de teste e consumo de energia, respectivamente, eliminando as soluções que não atendem tais restrições. O passo seis seleciona, dentre os restantes, o programa de auto-teste que apresenta o menor valor para a função de custo, e o último passo calcula o menor período de teste, para o programa de auto-teste selecionado, que permite o atendimento de todos os *deadlines* das tarefas da aplicação.

O método heurístico, ao contrário do anterior, tenta primeiro minimizar a função de custo, para depois verificar se a solução pré-selecionada atende todas as restrições do sistema. Esse método nem sempre encontra uma solução, e mesmo que encontre, ela nem sempre é ótima, embora se aproxime dela. O método heurístico é baseado em um laço (onde cada iteração é um passo) que contém quatro etapas principais. A primeira etapa pré-seleciona um programa de auto-teste composto pelas rotinas que apresentam o menor valor para a função de custo. Então, a segunda etapa verifica se a solução corrente atende a todos os requisitos do sistema. Caso um deles não seja atendido, uma das rotinas é virtualmente eliminada da biblioteca de auto-teste para que a mesma solução não seja novamente selecionada. Na última etapa do laço é feito um reajuste de pesos da função de custo para otimizar o processo de pré-seleção. Todas as etapas são repetidas até que seja encontrada uma solução que atenda todas as restrições do sistema, ou até que não restem mais soluções possíveis.

Ambos os métodos de seleção, ao encontrarem uma solução, fornecem o conjunto das rotinas de teste selecionadas e suas respectivas respostas de teste. Além disso, são fornecidos os custos totais do sistema para os fatores considerados e o menor período de teste calculado. Com esses dados pode-se finalizar o projeto do auto-teste. O programa de auto-teste é então escrito na memória de programa de processador e as respostas de teste esperadas são escritas na memória ROM do analisador de respostas, que já pode ser integrado ao sistema embarcado. Por fim, no caso de aplicações de tempo-real, o *deadline* e o período da tarefa de teste são fornecidos ao escalonador do sistema.

A partir do conteúdo exposto neste capítulo, podem-se extrair algumas conclusões preliminares. A mais importante é a necessidade de uma metodologia de projeto bem definida, que permita a redução de custos e de tempo no projeto de auto-teste *on-line* periódico para processadores embarcados. Tal metodologia deve ser capaz de verificar o atendimento de todas as restrições do sistema embarcado, bem como os requisitos de tempo-real, se for o caso. A metodologia aqui proposta tem essa capacidade e, além disso, abrange todos os fatores indispensáveis à aplicação de auto-teste *on-line* periódico em processadores embarcados.

Com relação à biblioteca de auto-teste, verificou-se a importância de se ter rotinas de teste otimizadas em termos de ocupação de memória, tempo de execução e consumo de energia, de modo a permitir a composição de programas de auto-teste que satisfaçam os requisitos e restrições do sistema. No tocante ao analisador de respostas, pode-se concluir que ele deve ser otimizado em termos de área, para reduzir o impacto na área total do sistema, e em termos de freqüência de operação, para permitir que seu tempo de execução seja limitado pelos tempos de leitura e escrita na memória RAM do processador.

Verificou-se também a importância da utilização de um método eficiente para a seleção do programa de auto-teste, e que leve em consideração o atendimento dos requisitos do sistema embarcado e, possivelmente, da aplicação de tempo-real. Ambos os métodos propostos satisfazem tais necessidades, garantindo a execução periódica do programa de auto-teste selecionado, com a maior freqüência permitida pela aplicação de tempo-real, e assegurando o atendimento das restrições do sistema. Por fim, foi demonstrado que o espaço de busca para os métodos de seleção cresce exponencialmente com o número de componentes do processador a serem testados. O método heurístico proposto possibilita uma seleção eficiente do programa de auto-teste, com um tempo de busca aceitável, mesmo em casos extremos.

6 ESTUDO DE CASO: AUTO-TESTE *ON-LINE* PARA OS PROCESSADORES FEMTOJAVA

Como um esforço inicial para a validação da metodologia STEP proposta no capítulo anterior, foi realizado um estudo de caso onde tal metodologia é aplicada aos processadores que compõem a família Femtojava. Com isso pretende-se verificar se o método proposto, embora bastante genérico, é completo o suficiente para abranger todas as etapas necessárias ao planejamento e implantação de auto-teste *on-line* para um processador de uma dada plataforma embarcada. O estudo aqui apresentado também visa demonstrar, através de resultados práticos, que a metodologia STEP atinge os objetivos aos quais se propõe. São eles: a garantia de execução periódica de um programa de auto-teste de alta qualidade para um processador embarcado, respeitando todas as restrições do sistema alvo e todos os requisitos da aplicação de tempo-real.

A preferência pela família Femtojava como plataforma alvo deste estudo de caso deveu-se à facilidade de acesso às informações necessárias para o trabalho (artigos, diagramas, descrições textuais, descrições em VHDL – *VHSIC Hardware Description Language* – etc.), uma vez que tais processadores foram desenvolvidos (e continuam em desenvolvimento) nesta Universidade. Embora os processadores Femtojava não sejam usados em aplicações comerciais, eles apresentam as mesmas características básicas necessárias para qualquer projeto de um sistema embarcado baseado em processador, incluindo portas de entrada e saída, controle de interrupção e técnicas para aumento de desempenho como pipeline, VLIW e outras.

Antes de descrever a aplicação da metodologia STEP aos processadores Femtojava e apresentar os resultados obtidos, é necessário um conhecimento mínimo sobre esses processadores que constituem a plataforma alvo. Assim, a próxima seção descreve brevemente cada um dos processadores Femtojava utilizados neste estudo de caso, salientado as características mais relevantes de sua arquitetura e organização no que diz respeito ao projeto de auto-teste *on-line* periódico com o método STEP.

6.1 A Família de Processadores Femtojava

Femtojava (ITO; CARRO; JACOBI, 2001) é um microcontrolador baseado em pilha que executa nativamente *bytecodes* Java. Ele foi projetado especificamente para o desenvolvimento de aplicações embarcadas em FPGA (*Field Programmable Gate Array*). O microcontrolador tem memórias de dados (RAM) e de programa (ROM), *single port*, em uma arquitetura Harvard, ou seja, com barramentos separados para dados e endereços. Além disso, possui um mecanismo para tratamento de interrupções com dois níveis de prioridade e portas de E/S mapeadas na memória de dados.

O processador Femtojava segue as especificações da Máquina Virtual Java (JVM – *Java Virtual Machine*) (VENNERS, 1998) e é capaz de executar um subconjunto das instruções definidas na JVM. Tais instruções, ou *bytecodes* Java, podem ter um, dois ou três bytes. O primeiro byte contém o *opcode* da instrução e os dois próximos, quando presentes, contêm dados. Portanto, um *bytecode* pode ter até dois bytes de operandos imediatos. Há instruções para operações aritméticas e lógicas, controle de fluxo, operações com a pilha, transferências entre a pilha e o repositório de variáveis locais e operações com *arrays* em memória. Adicionalmente, algumas instruções para entrada e saída (E/S) e operações de escalonamento, não previstas pela JVM, também foram implementadas.

Uma versão ASIP (*Application-Specific Instruction Set Processor*) do Femtojava pode ser automaticamente gerada por meio do ambiente de desenvolvimento SASHIMI – *System As Software and Hardware In Microcontrollers* (ITO; CARRO; JACOBI, 2001). Nesse ambiente, a aplicação é descrita em linguagem Java e a ferramenta automaticamente gera, em VHDL, uma versão customizada do processador com base na análise dos *bytecodes* Java da aplicação. A versão customizada inclui apenas as unidades funcionais e as instruções necessárias para a execução da aplicação descrita. A largura de dados do processador também é parametrizável, podendo ser de 8, 16 ou 32 bits. Portanto, os tamanhos dos blocos de dados e de controle do ASIP gerado são diretamente proporcionais ao número de diferentes *bytecodes* utilizados pelo software da aplicação.

Para prover um *framework* para aplicações de tempo-real, uma API (*Application Programming Interface*) baseada no RTSJ (*Real-Time Specification for Java*) foi integrada ao ambiente SASHIMI. Essa API (WEHRMEISTER; BECKER; PEREIRA, 2004) permite ao projetista programar seus algoritmos de escalonamento em linguagem Java. A estrutura de escalonamento desenvolvida consiste em uma tarefa adicional encarregada de alocar o processador para aquelas tarefas da aplicação que estão prontas para execução, com base na política de escalonamento implementada. Essa abordagem é a mesma utilizada em sistemas operacionais de tempo-real. O Femtojava suporta até oito tarefas estáticas (não há criação de tarefas em tempo de execução), periódicas e independentes.

Até a data desta dissertação, duas políticas de escalonamento amplamente utilizadas em sistemas de tempo-real haviam sido implementadas para o processador Femtojava. São elas: o algoritmo RM (*Rate Monotonic*), de prioridade fixa, e o algoritmo EDF (*Earliest Deadline First*), de prioridade dinâmica (LIU; LAYLAND, 1973). Assim, o desenvolvedor da aplicação deve selecionar, durante o projeto, a política de escalonamento mais adequada, e um escalonador customizado é então sintetizado juntamente com a aplicação e a versão ASIP do processador Femtojava.

O primeiro processador Femtojava projetado e desenvolvido tem uma organização multiciclo (ITO; CARRO; JACOBI, 2001). Nessa versão do microcontrolador, a execução de cada instrução é completada em determinado número de ciclos. De acordo com a instrução, esse número pode ser três, quatro, sete ou onze. A Figura 6.1 ilustra a microarquitetura do processador Femtojava Multiciclo. Nessa arquitetura, a pilha de operandos e o repositório de variáveis locais são implementados na memória de dados.

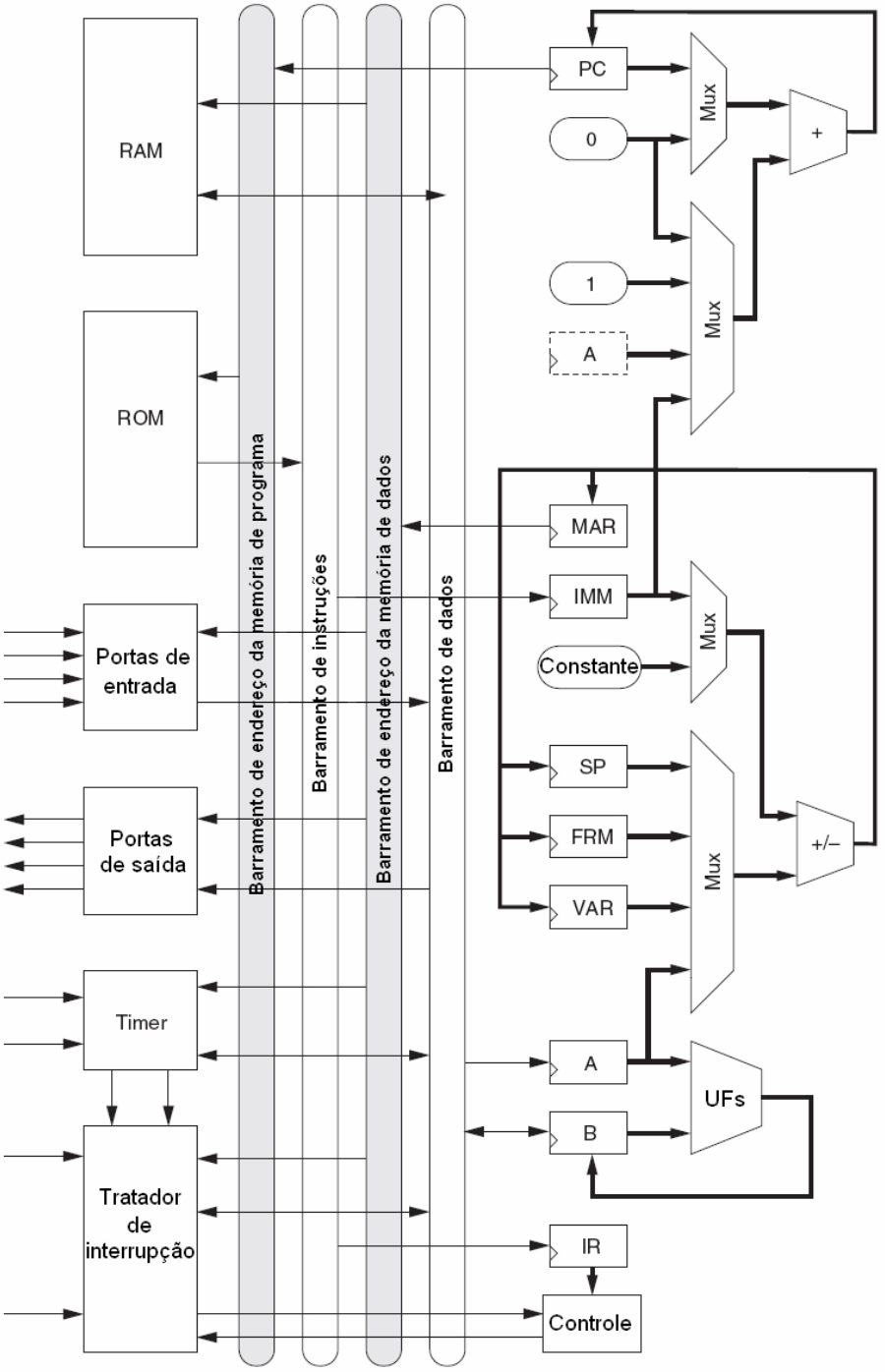

Figura 6.1: Microarquitetura do Femtojava Multiciclo
(ITO; CARRO; JACOBI, 2001, p.105)

O processador Femtojava Multiciclo possui três unidades funcionais implementadas: uma ULA, um multiplicador e um deslocador. Na Figura 6.1, essas unidades encontram-se dentro do bloco indicado por UFs (Unidades Funcionais). Todas as operações lógicas e aritméticas são realizadas sobre os valores do topo da pilha. Por isso, os registradores A e B armazenam sempre o conteúdo das duas posições mais altas da pilha. Em cada uma dessas operações, os operandos são desempilhados e o resultado é então empilhado.

A Unidade Lógica e Aritmética é responsável pela execução de seis operações. As operações aritméticas de adição e subtração em complemento de dois são ativadas, respectivamente, pelas instruções *iadd* e *isub*. As instruções *iand*, *ior* e *ixor* executam, respectivamente, as operações lógicas AND, OR e XOR. A ULA realiza ainda uma operação de negação (instrução *ineg*) sobre um único operando. O módulo multiplicador executa uma única operação, ativada pela instrução *imul*, cujo resultado tem a mesma largura dos operandos. Por fim, o deslocador realiza deslocamentos para a esquerda (aritmético) e para a direita (aritmético e lógico). Essas operações são realizadas, respectivamente, pelas instruções *ishl*, *ishr* e *iushr*.

Depois do Multiciclo, foi desenvolvida uma versão pipeline do microcontrolador (BECK; CARRO, 2003). Esse processador, também denominado Femtojava Low-Power, possui um pipeline de cinco estágios: busca de instruções, decodificação, busca de operandos, execução, e escrita de resultados. Uma das principais características da versão Low-Power é a implementação da pilha de operandos e do repositório de variáveis locais do método em um banco de registradores, ao invés da memória de dados, como no Femtojava Multiciclo.

O primeiro estágio do pipeline busca instruções na memória através de palavras de 32 bits, diferentemente do Femtojava Multiciclo, no qual as palavras da memória de programa são de 8 bits. Esse estágio possui uma fila de instruções com 9 registradores de um byte cada. Esse mecanismo permite a busca antecipada de *bytecodes*, evitando que o pipeline fique parado na busca de instruções. A primeira instrução da fila é enviada para o segundo estágio, a decodificação. O estágio de decodificação tem quatro funções: gerar a palavra de controle para a instrução corrente, informar o tamanho dessa instrução para a fila de instruções, analisar a dependência de dados das instruções e fazer a análise de *forwarding*.

O mecanismo de *forwarding* evita a inserção de bolhas no pipeline no caso de dependência de dados do tipo RAW (*Read After Write*). Para isso, um resultado que ainda não foi escrito no banco de registradores deve ser repassado diretamente para o estágio de busca de operandos. No Femtojava Low-Power, o valor repassado pode vir do estágio de execução e/ou do estágio de escrita de resultados. Sabendo-se que operandos lidos não são mais utilizados, uma vez que são desempilhados, os valores repassados via *forwarding* (resultados de instruções anteriores) nem mesmo são gravados no banco de registradores. É importante dar ênfase a essa característica da versão Low-Power, pois ela exige muito cuidado no desenvolvimento de rotinas de auto-teste para o banco de registradores, como será mostrado adiante.

No terceiro estágio os operandos são buscados para a execução. Ele é composto basicamente por um banco de registradores, onde podem ser feitas duas leituras independentes ou uma escrita a cada ciclo de relógio. A largura do banco é a mesma largura de dados do processador. O número de registradores é configurável, durante o projeto do sistema, no ambiente SASHIMI. A Figura 6.2 ilustra o estágio de busca de operandos. O banco de registradores (*Register File*) contém a pilha de operandos e o repositório de variáveis locais do método. Os registradores SP e VARS apontam para o topo da pilha e para o início do repositório, respectivamente.

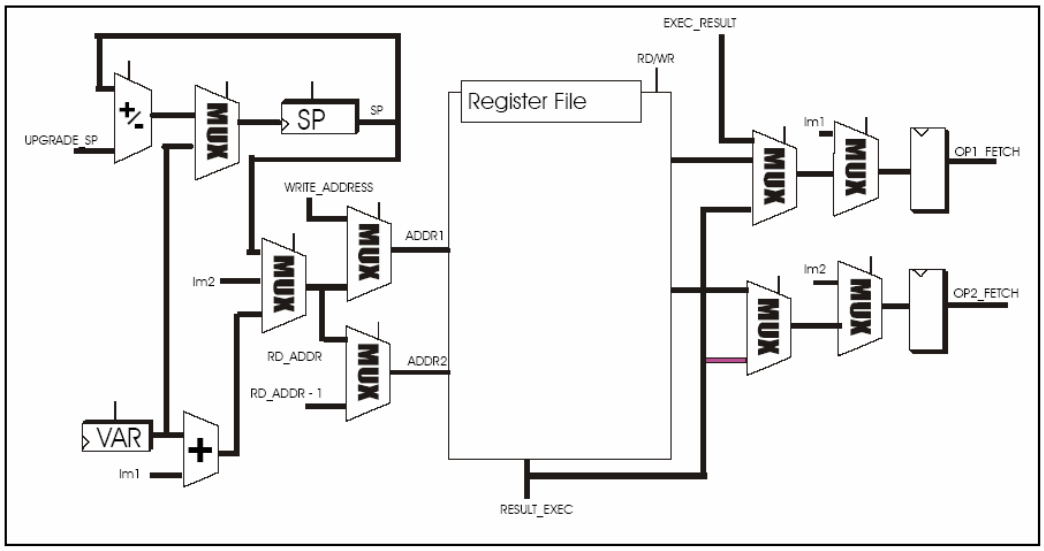

Figura 6.2: Estágio de busca de operandos do Femtojava Pipeline (BECK, 2004, p.50)

O estágio de execução contém unidades funcionais idênticas às do Femtojava Multiciclo: uma ULA, um multiplicador (MULT) e um deslocador (Shifter). Esse estágio é ilustrado na Figura 6.3. Além das unidades funcionais, estão presentes uma unidade de *load/store* (LD/ST), responsável pelo cálculo de endereços, e uma unidade de desvio (Branch), que verifica se um desvio deve ou não ocorrer. Não há predição de desvio, pois todos são previamente considerados como verdadeiros. Por fim, no último estágio do pipeline, o resultado do estágio de execução é escrito no banco de registradores (exceto no caso de *forwarding*).

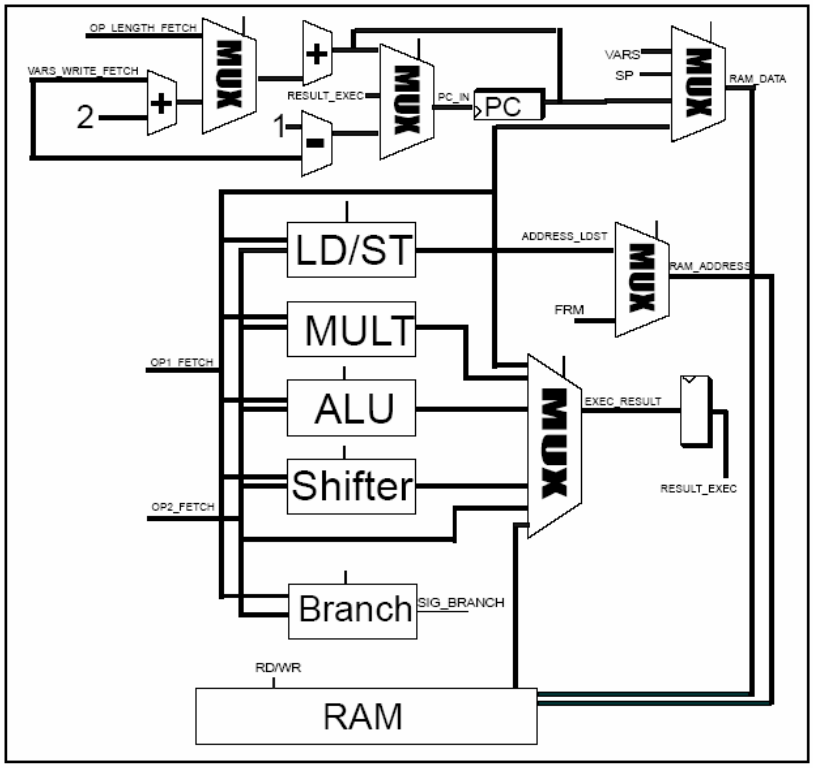

Figura 6.3: Estágio de execução do Femtojava Pipeline (BECK, 2004, p.51)

O processador Femtojava VLIW (BECK; CARRO, 2004) é uma extensão da versão Pipeline. Basicamente, ele tem suas unidades funcionais, banco de registradores e decodificador de instruções replicados. O número de réplicas é configurável em tempo de projeto, podendo ser igual a dois, quatro ou oito. Assim, o processador pode conter duas, quatro ou oito réplicas dos seus estágios de pipeline (fluxos de instruções). Cada fluxo tem sua própria pilha de operandos, enquanto o repositório de variáveis locais do método é compartilhado por todos os fluxos, e localiza-se no banco de registradores do fluxo principal. A comunicação entre os fluxos de instruções se dá pelo repositório de variáveis locais.

A busca por ILP (*Instruction Level Parallelism*) no programa Java é feita em nível de *bytecodes*. O algoritmo de análise do código separa as instruções que dependem de resultados de instruções anteriores em *blocos de operandos*. Todo o programa é separado nesses blocos, os quais podem ser executados em paralelo, respeitando as limitações das unidades funcionais. Por exemplo, se existe apenas um multiplicador, duas operações *imul* não podem ser executadas em paralelo. A existência de um algoritmo definido para análise de paralelismo é bastante relevante no desenvolvimento de um programa de auto-teste para a versão VLIW do Femtojava, como será discutido adiante.

O microcontrolador Femtojava DSP (KRAPF; CARRO, 2003) é uma adaptação da versão Multiciclo com otimizações para aplicações de processamento digital de sinais. As otimizações implementadas incluem a adição de uma instrução multiplica-acumula (*imac*), o controle eficiente de laços, a inclusão de um *buffer* circular e o endereçamento com bit reverso. Para a instrução *imac* foram utilizadas as unidades funcionais já existentes, bastando criar um caminho direto entre a saída do multiplicador e a entrada da ULA. Para o controle eficiente de laços, foram adicionados três registradores, um somador, um comparador simples e uma pequena lógica de controle. O *buffer* circular foi implementado na memória de dados, sendo necessárias algumas modificações no árbitro do barramento para o controle do *buffer*. Finalmente, o endereçamento com bit reverso foi implementado na ULA, por meio de uma série de modos de endereçamento com bit reverso simples. Todas essas modificações resultaram num incremento de 15,5% de área, em relação ao Femtojava Multiciclo.

Conhecidos os processadores que compõem a família Femtojava, algumas observações se fazem necessárias antes da apresentação desse estudo de caso. Em primeiro lugar, é preciso dizer que, devido às restrições de tempo para o desenvolvimento do presente trabalho, a metodologia STEP foi aplicada apenas para os processadores Femtojava Multiciclo e Femtojava Pipeline. Entretanto, para possibilitar um estudo mais amplo da metodologia proposta, as particularidades e limitações da sua aplicação para os processadores Femtojava VLIW e Femtojava DSP serão brevemente discutidas. Como será visto, as características dessas arquiteturas não inviabilizam a aplicação do método.

Foi dito no início desta seção que a família Femtojava foi especialmente projetada para prototipação em FPGAs. Entretanto, neste trabalho, considerou-se a síntese dos processadores para ASIC (*Application-Specific Integrated Circuit*). Isso porque o teste de um FPGA depende unicamente de sua estrutura interna, e não do hardware nele programado. Assim, os processadores foram sintetizados e mapeados, com a ferramenta Leonardo Spectrum® (Mentor Graphics), para a tecnologia ADK 0,35µm, e otimizando área. O *design kit* dessa tecnologia, o qual inclui a biblioteca ATPG necessária às ferramentas de teste utilizadas, é de acesso gratuito para universidades, possibilitando

seu uso neste trabalho. Vale ressaltar que a metodologia STEP independe da tecnologia de fabricação do circuito integrado alvo.

Também foi dito que a largura de dados do processador sintetizado, bem como as unidades funcionais nele presentes e as instruções por ele executadas, são parâmetros de projeto no ambiente SASHIMI. Assim , para garantir que os resultados obtidos sejam significativos, foram utilizadas versões de ambos os processadores com largura de dados de 16 bits, capazes de executar todas as instruções possíveis, e com as três unidades funcionais implementadas. No processador Femtojava Pipeline, onde o tamanho do banco de registradores também é parametrizável, foi utilizado um banco com 16 registradores, número este suficiente para uma grande variedade de aplicações.

Por fim, deve-se observar que o teste das memórias RAM e ROM dos processadores não faz parte do escopo desta dissertação. Entretanto, como foi visto em capítulos anteriores, o uso da técnica SBST pode facilmente ser estendido para cobrir o teste também desses módulos. Logo, a inclusão de rotinas para o teste das memórias na biblioteca de auto-teste possibilita o uso da mesma metodologia para essa finalidade.

Nas próximas seções, serão descritas as etapas da aplicação da metodologia STEP para o auto-teste *on-line* periódico dos processadores Femtojava. Os resultados obtidos serão analisados visando à validação do método e a indicação de melhorias necessárias.

6.2 A Biblioteca de Auto-Teste

A primeira etapa da metodologia STEP, conforme apresentado na Seção 5.2, consiste no desenvolvimento da biblioteca de auto-teste para os componentes visíveis de dados e para a unidade de controle do processador alvo. Nesse estudo de caso existem dois processadores alvo, Femtojava Multiciclo e Femtojava Pipeline, para os quais foi desenvolvida uma única biblioteca de auto-teste. Essa peculiaridade visa facilitar a automatização do projeto e aplicação de auto-teste para a Família Femtojava, uma vez que os processadores dessa família possuem alguns componentes em comum. A referida automatização é detalhada na Seção 6.5.

Para cada componente em comum entre o Femtojava Multiciclo e o Femtojava Pipeline, uma única rotina de teste foi desenvolvida com cada uma das abordagens utilizadas. Essas rotinas podem ser aplicadas tanto para um processador quanto para o outro. Porém, sabe-se que o tamanho das palavras da memória de programa desses processadores é diferente. Por isso, as rotinas da biblioteca de auto-teste para os processadores Femtojava foram implementadas como métodos da linguagem Java, de modo que elas possam ser facilmente sintetizadas para o processador alvo por meio do ambiente SASHIMI. Essa característica da biblioteca também contribuiu para a automatização do método STEP para a família Femtojava, como será visto adiante.

Foram identificados no processador Femtojava Multiciclo três D-VCs principais: a ULA, o multiplicador e o deslocador. Além desses, há os registradores temporários de dados A e B, que armazenam os valores do topo da pilha. Entretanto, uma vez que eles são utilizados em todas as operações das unidades funcionais, seu teste já está incluido nas rotinas de teste para os três D-VCs principais. Com relação ao Femtojava Pipeline, além dos mesmos três D-VCs, há ainda o banco de registradores. E da mesma forma que na versão Multiciclo, os registradores de dados são testados por meio das rotinas para os D-VCs principais. Assim, a biblioteca de auto-teste para a família Femtojava deve conter rotinas para o multiplicador, a ULA, o deslocador e o banco de

registradores. Também deve ter, no mínimo, uma rotina para a unidade de controle de cada um dos processadores.

É importante salientar que a biblioteca de auto-teste foi desenvolvida de tal modo que, para cada componente, qualquer uma das abordagens implementadas forneça a mesma cobertura de falhas (ou muito próxima). Essa característica foi alcançada pelo ajuste no número de padrões de teste em cada rotina. Com isso pretende-se realizar comparações justas entre as diferentes abordagens de desenvolvimento de rotinas de teste com o intuito de verificar se existe alguma que se destaque por seus reduzidos custos. Além disso, a utilização de rotinas que fornecem a mesma cobertura de falhas evita que haja uma discrepância muito grande entre a cobertura de falhas total do processador obtida com programas de auto-teste diferentes.

O desenvolvimento das rotinas de teste para cada um dos componentes alvo é descrito a seguir. As rotinas foram implementadas como métodos de classes Java onde os padrões de teste são representados por valores inteiros. Dados armazenados em memória (padrões e respostas de teste) são representados, como variáveis globais da classe, na forma de *arrays*. Nas abordagens que utilizam padrões de teste gerados por ATPG, foi utilizada a ferramenta Flextest® (Mentor Graphics) para geração dos padrões.

6.2.1 Multiplicador

Para o multiplicador foram desenvolvidas quatro rotinas, seguindo as abordagens apresentadas na Seção 4.3.3, que se baseiam em três diferentes estratégias de TPG.

A classe Java *Multiplier_ATPGimm*, resumida na Figura 6.4, contém a rotina de teste na qual padrões gerados por ATPG são aplicados ao multiplicador como operandos imediatos. Vale observar que nessa rotina, bem como em todas as outras que utilizam a abordagem com operando imediatos, os padrões de teste são primeiro atribuídos a variáveis locais do método, para depois serem operados. O objetivo disso é evitar que o próprio compilador Java realize as operações (nesse caso, as multiplicações), ao invés do componente alvo.

```
class Multiplier_ATPGimm {
  public static int[] results = {0, 0, ..., 0};

  public static void Testing() {
    int operand1;
    int operand2;

    operand1 = PAD_X_1;
    operand2 = PAD_Y_1;
    results[0] = operand1 * operand2;
    operand1 = PAD_X_2;
    operand2 = PAD_Y_2;
    results[1] = operand1 * operand2;
    ...
    operand1 = PAD_X_29;
    operand2 = PAD_Y_29;
    results[28] = operand1 * operand2;
  }
}
```

Figura 6.4: Rotina *ATPG Imediato* para o multiplicador

Na Figura 6.5 é ilustrada a classe *Multiplier_ATPGmem*. Ela contém a rotina de teste na qual os mesmos padrões gerados por ATPG são armazenados e buscados na memória de dados.

```
class Multiplier_ATPGmem {
  public static int[] results   = {0, 0, ..., 0};
  public static int[] operands1 = {PAD_X_1, PAD_X_2, ..., PAD_X_29};
  public static int[] operands2 = {PAD_Y_1, PAD_Y_2, ..., PAD_Y_29};

  public static void Testing() {

    for (int idx = 0; idx < 29; idx++) {
      results[idx] = operands1[idx] * operands2[idx];
    }
  }
}
```

Figura 6.5: Rotina *ATPG Memória* para o multiplicador

A rotina de teste com padrões pseudo-aleatórios é mostrada na Figura 6.6, como um método Java da classe *Multiplier_LFSR*.

```
class Multiplier_LFSR {
  public static int[] results = {0, 0, ..., 0};

  public static void Testing() {
    int operand2 = 0;       // bits menos significativos da semente (0x0000)
    int operand1 = -32768;  // bits mais significativos da semente (0x8000)
    int aux1, aux2;

    for (int idx1 = 0; idx1 < 56; idx1++) {
      results[idx1] = operand1 * operand2;
      aux1 = operand2 & 3;    // bits menos significativos do polinômio característico (0x0003)
      aux2 = aux1;
      for (int idx2 = 0; idx2 < 15; idx2++) {
        aux2 = aux1 ^ (aux2 << 1);
      }
      aux2 >>>= 15;
      aux1 = operand1 & 6144; // bits mais significativos do polinômio carcterístico (0x1800)
      aux2 ^= aux1;
      for (int idx2 = 0; idx2 < 15; idx2++) {
        aux2 = aux1 ^ (aux2 << 1);
      }
      operand2 = (operand2 >>> 1) | (operand1 << 15);
      operand1 = (operand1 >>> 1) | (aux2 & -32768); // máscara (0x8000)
    }
  }
}
```

Figura 6.6: Rotina *LFSR* para o multiplicador

A última rotina desenvolvida para o multiplicador utiliza padrões de teste determinísticos regulares (PASCHALIS et al., 2001). Nessa rotina, os 16 bits de cada operando (*operand1* e *operand2*) são constituídos por segmentos de 3 bits (denominados $b_2b_1b_0$ para *operand1*, e $b_5b_4b_3$ para *operand2*) repetidos da seguinte forma:

operand1 = b_0 $b_2b_1b_0$ $b_2b_1b_0$ $b_2b_1b_0$ $b_2b_1b_0$ $b_2b_1b_0$

operand2 = b_3 $b_5b_4b_3$ $b_5b_4b_3$ $b_5b_4b_3$ $b_5b_4b_3$ $b_5b_4b_3$

Assim, o número total de padrões de teste aplicados ao multiplicador, nessa rotina, é igual a 64 (todas as possíveis combinações dos 6 bits $b_5b_4b_3b_2b_1b_0$). A rotina de teste implementa dois laços aninhados que incrementam, cada um, o valor de um dos operandos. O valor do incremento é *1001001001001001*, de modo que cada grupo de 3 bits ($b_2b_1b_0$ para *operand1*, e $b_5b_4b_3$ para *operand2*) tenha incremento igual a *001*.

A classe *Multiplier_RegDet*, mostrada na Figura 6.7, contém o método Java correspondente à rotina de teste que gera os padrões determinísticos descritos acima e aplica-os ao multiplicador.

```
class Multiplier_RegDet {
  public static int[] results = {0, 0, ..., 0};

  public static void Testing() {
    int operand1 = 0;  // valor inicial do operando 1
    int operand2 = 0;  // valor inicial do operando 2
    int idx = 0;

    while (operand2 != -28088) {     // valor final do operando 1 (0x9248)
      while (operand1 != -28088) {  // valor final do operando 2 (0x9248)
        results[idx] = operand1 * operand2;
        operand1 += -28087;  // incremento do operando 1 (0x9249)
        idx++;
      }
      operand1 = 0;
      operand2 += -28087;     // incremento do operando 2 (0x9249)
    }
  }
}
```

Figura 6.7: Rotina *Determinística Regular* para o multiplicador

6.2.2 Deslocador

Para o deslocador, foram desenvolvidas três das quatro abordagens utilizadas no multiplicador. A rotina de teste baseada em padrões pseudo-aleatórios não foi desenvolvida para o deslocador, nem para a ULA. Isso porque se verificou que ela tem um tempo de execução duas ordens de grandeza maior do que o tempo de execução das demais abordagens, devido às operações necessárias para a implementação do LFSR em software. Além disso, essa abordagem não apresenta qualquer vantagem sobre as demais estratégias, uma vez que seu tempo de execução é maior, bem como sua ocupação das memórias de dados e de instruções, e seu consumo de energia. Essas constatações poderão ser comprovadas em breve, quando os custos de cada uma das rotinas da biblioteca forem apresentados.

A classe Java *Shifter_ATPGimm*, resumida na Figura 6.8, contém a rotina de teste na qual padrões gerados por ATPG são aplicados ao deslocador como operandos imediatos.

```
class Shifter_ATPGimm {
  public static int[] results = {0, 0, ..., 0};

  public static void Testing() {
    int operand1;
    int operand2;

    operand1  = PAD_X_1;
    operand2  = PAD_Y_1;
    results[0]  = operand1 << operand2;
    ...
    operand1  = PAD_X_6;
    operand2  = PAD_Y_6;
    results[5]  = operand1 << operand2;
    operand1  = PAD_X_7;
    operand2  = PAD_Y_7;
    results[6]  = operand1 >> operand2;
    ...
    operand1  = PAD_X_9;
    operand2  = PAD_Y_9;
    results[8]  = operand1 >> operand2;
    operand1  = PAD_X_10;
    operand2  = PAD_Y_10;
    results[9]  = operand1 >>> operand2;
    ...
    operand1  = PAD_X_14;
    operand2  = PAD_Y_14;
    results[13] = operand1 >>> operand2;
  }
}
```

Figura 6.8: Rotina *ATPG Imediato* para o deslocador

Na Figura 6.9 é ilustrada a classe *Shifter_ATPGmem*. Ela contém a rotina de teste na qual os mesmos padrões gerados por ATPG são armazenados e buscados na memória de dados.

```
class Shifter_ATPGmem {
  public static int[] results   = {0, 0, ..., 0};
  public static int[] operands1 = {PAD_X_1, PAD_X_2, ..., PAD_X_14};
  public static int[] operands2 = {PAD_Y_1, PAD_Y_2, ..., PAD_Y_14};

  public static void Testing() {

    for (int idx = 0; idx < 14; idx++) {
      if (idx < 6) {
        results[idx] = operands1[idx] << operands2[idx]; }
      else if (idx < 9) {
        results[idx] = operands1[idx] >> operands2[idx]; }
      else {
        results[idx] = operands1[idx] >>> operands2[idx];
      }
    }
  }
}
```

Figura 6.9: Rotina *ATPG Memória* para o deslocador

A última rotina desenvolvida para o deslocador utiliza padrões de teste determinísticos regulares (PASCHALIS et al., 2001). Uma adaptação da proposta original foi necessária porque o deslocador dos processadores Femtojava não realiza

operações de rotação, ao contrário do componente usado por Paschalis et al. A rotina aqui desenvolvida para geração e aplicação de padrões determinísticos regulares no deslocador implementa dois laços consecutivos. Cada laço realiza deslocamentos para a esquerda e para a direita, alternadamente. Para que ambos os sinais de controle sejam ativados, no primeiro laço os deslocamentos para a direita são lógicos e, no segundo, são aritméticos. No primeiro laço, o valor inicial do deslocamento (15 bits) é decrementado de um a cada operação, até atingir o valor zero. No segundo, o valor inicial do deslocamento (zero) é incrementado em uma cada operação, até atingir o valor 15. No total, essa rotina aplica 32 padrões de teste ao deslocador.

A classe *Shifter_RegDet*, mostrada na Figura 6.10, contém o método Java correspondente à rotina descrita acima.

```
class Shifter_RegDet {
  public static int[] results = {0, 0, ..., 0};

  public static void Testing() {
    int operand1 = 1;   // valor inicial do operando (0x0001)
    int operand2 = 16;  // valor inicial do deslocamento (0x0010)
    int idx = 0;

    while (operand2 > 0) {  // valor final do deslocamento (0x0000)
      operand2--;           // decremento do valor do deslocamento
      operand1 <<= operand2;
      results[idx] = operand1;
      idx++;
      operand2--;           // decremento do valor do deslocamento
      operand1 >>>= operand2;
      results[idx] = operand1;
      idx++;
    }
    operand1 = 21845;  // valor inicial do operando (0x5555)
    while (operand2 < 16) {  // valor final do deslocamento (0x0010)
      operand1 <<= operand2;
      results[idx] = operand1;
      operand2++;            // incremento do valor do deslocamento
      idx++;
      operand1 >>= operand2;
      results[idx] = operand1;
      operand2++;            // incremento do valor do deslocamento
      idx++;
    }
  }
}
```

Figura 6.10: Rotina *Determinística Regular* para o deslocador

6.2.3 ULA

Para a Unidade Lógica e Aritmética foram desenvolvidas as mesmas três abordagens utilizadas no deslocador. A classe Java *Alu_ATPGimm*, resumida na Figura 6.11, contém a rotina de teste na qual padrões gerados por ATPG são aplicados à ULA como operandos imediatos.

```
class Alu_ATPGimm {
  public static int[] results = {0, 0, ..., 0};

  public static void Testing() {
    int operand1;
    int operand2;

    operand1  = PAD_X_1;
    operand2  = PAD_Y_1;
    results[0]  = operand1 + operand2;
    ...
    operand1  = PAD_X_4;
    operand2  = PAD_Y_4;
    results[3]  = operand1 + operand2;
    operand1  = PAD_X_5;
    operand2  = PAD_Y_5;
    results[4]  = operand1 - operand2;
    ...
    operand1  = PAD_X_9;
    operand2  = PAD_Y_9;
    results[8]  = operand1 - operand2;
    operand1  = PAD_X_10;
    operand2  = PAD_Y_10;
    results[9]  = operand1 & operand2;
    ...
    operand1 = PAD_X_15;
    operand2 = PAD_Y_15;
    results[14] = operand1 & operand2;
    operand1 = PAD_X_16;
    operand2 = PAD_Y_16;
    results[15] = operand1 | operand2;
    ...
    operand1 = PAD_X_19;
    operand2 = PAD_Y_19;
    results[18] = operand1 | operand2;
    operand1 = PAD_X_20;
    operand2 = PAD_Y_20;
    results[19] = operand1 ^ operand2;
    operand1 = PAD_X_21;
    operand2 = PAD_Y_21;
    results[20] = operand1 ^ operand2;
    operand2 = PAD_Y_22;
    results[21] = - operand2;
    ...
    operand2 = PAD_Y_27;
    results[26] = - operand2;
  }
}
```

Figura 6.11: Rotina *ATPG Imediato* para a ULA

Na Figura 6.12 é ilustrada a classe *Alu_ATPGmem*. Ela contém a rotina de teste na qual os mesmos padrões gerados por ATPG são armazenados e buscados na memória de dados.

A última rotina desenvolvida para a ULA utiliza padrões de teste determinísticos regulares. Nessa rotina, são executas as instruções *iadd*, *isub*, *iand*, *ior* e *ixor* com as quatro possíveis combinações em que todos os bits de cada operando têm o mesmo valor (*0x0000* e *0xFFFF*). Em seguida, a instrução *ineg* é executada sobre os operandos *0x0000*, *0x5555*, *0xAAAA* e *0xFFFF*. Devido à regularidade na estrutura interna da ULA, a aplicação desses operandos é suficiente para a obtenção de uma alta cobertura de falhas.

```
class Alu_ATPGmem {
  public static int[] results   = {0, 0, ..., 0};
  public static int[] operands1 = {PAD_X_1, PAD_X_2, ..., PAD_X_21};
  public static int[] operands2 = {PAD_Y_1, PAD_Y_2, ..., PAD_Y_27};

  public static void Testing() {

    for (int idx = 0; idx < 27; idx++) {
      if (idx < 4) {
        results[idx] = operands1[idx] + operands2[idx]; }
      else if (idx < 9) {
        results[idx] = operands1[idx] - operands2[idx]; }
      else if (idx < 15) {
        results[idx] = operands1[idx] & operands2[idx];}
      else if (idx < 19) {
        results[idx] = operands1[idx] | operands2[idx];}
      else if (idx < 21) {
        results[idx] = operands1[idx] ^ operands2[idx];}
      else {
        results[idx] = - operands2[idx];
      }
} } }
```

Figura 6.12: Rotina *ATPG Memória* para a ULA

A classe *Alu_RegDet*, mostrada na Figura 6.13, contém o método Java correspondente à rotina de teste que gera os padrões determinísticos descritos acima e aplica-os à ULA.

```
class Alu_RegDet {
  public static int[] results = {0, 0, ..., 0};

  public static void Testing() {
    int operand1 = 0;  // valor inicial do operando 1 (0x0000)
    int operand2 = 0;  // valor inicial do operando 2 (0x0000)
    int idx = 0;

    while (operand2 != -2) {   // valor final do operando 2 (0xFFFE)
      while (operand1 != -2) { // valor final do operando 1 (0xFFFE)
        results[idx] = operand1 + operand2;
        idx++;
        results[idx] = operand1 - operand2;
        idx++;
        results[idx] = operand1 & operand2;
        idx++;
        results[idx] = operand1 | operand2;
        idx++;
        results[idx] = operand1 ^ operand2;
        idx++;
        operand1 += -1;  // incremento do operando 1 (0xFFFF)
      }
      operand1 = 0;      // valor inicial do operando 1 (0x0000)
      operand2 += -1;    // incremento do operando 2 (0xFFFF)
    }
    operand2 = 0;        // valor inicial do operando 2 (0x0000)
    while (operand2 != 21844) { // valor final do operando 2 (0x5554)
      results[idx] = - operand2;
      idx++;
      operand2 += 21845; // incremento do operando 2 (0x5555)
    }
} }
```

Figura 6.13: Rotina *Determinística Regular* para a ULA

6.2.4 Banco de Registradores

O teste do banco de registradores do Femtojava Pipeline é um pouco mais complicado do que o teste dos demais componentes. Não somente porque ele requer mais padrões de teste, mas também porque ele contém o repositório de variáveis locais do método e a pilha de operandos. Por isso, durante o desenvolvimento da rotina para esse componente, deve-se ter em mente que a própria rotina é um método que fará uso do repositório de variáveis e da pilha de operandos, independentemente do teste realizado.

Os padrões de teste podem ser aplicados ao banco de registradores tanto como variáveis locais do método, quanto como operandos da pilha. A vantagem da utilização de variáveis locais é a possibilidade de acesso aleatório a qualquer registrador, ao contrário da pilha de operandos, que permite leituras e escritas apenas no topo da pilha. Porém, as instruções de leitura do repositório de variáveis locais utilizam apenas uma das portas de leitura do banco de registradores, enquanto as operações lógicas e aritméticas utilizam as duas portas, para leitura dos dois operandos do topo da pilha. Assim, a aplicação dos padrões de teste por meio da pilha de operandos possibilita a obtenção de uma cobertura de falhas maior, uma vez que utiliza as duas portas de leitura do banco de registradores.

Tendo sido feita a escolha da pilha de operandos para a aplicação dos padrões de teste ao banco de registradores, a utilização de padrões gerados por ATPG foi logo descartada. Isso porque essa abordagem requer o acesso aleatório a registradores do banco, tanto para leitura quanto para escrita. A utilização de padrões de teste pseudo-aleatórios é também inviável, devido ao tempo necessário não apenas para geração dos padrões, mas também para sua aplicação por meio de uma estrutura de pilha. Para cada registrador alvo, seria preciso fazer uma escrita em todos os registradores abaixo dele, na pilha.

Desse modo, restaram apenas os padrões determinísticos regulares. Um algoritmo bastante utilizado para o teste de memórias e bancos de registradores é o algoritmo March (LALA, 1997). Entretanto, a estrutura de pilha impede sua aplicação, pois ele requer leituras e escritas em registradores específicos sem a alteração do conteúdo dos demais registradores (mesmo os que estão em posições abaixo na pilha). Por isso foi desenvolvido um algoritmo de teste para o banco de registradores que faz uso adequado da pilha de operandos. Tal algoritmo escreve e lê valores em registradores consecutivos, independentemente de valores previamente armazenados em outros registradores.

A rotina desenvolvida para o teste do banco de registradores realiza um XOR entre os dois valores do topo da pilha, de modo que as duas portas de leitura sejam utilizadas. A cada operação, o topo da pilha é deslocado em uma posição, fazendo com que todos os registradores do banco sejam escritos e lidos. Essa caminhada pelo banco de registradores é realizada quatro vezes, cada uma com valores diferentes no topo da pilha (*0x0000*, *0x5555*, *0xAAAA* e *0xFFFF*). Os valores abaixo das duas posições do topo da pilha são sempre preenchidos com zero. A Figura 6.14 ilustra a classe Java *RegBank16_RegDet* contendo o método que implementa a rotina de teste para o banco de registradores descrita acima.

```
class RegBank16_RegDet {
 public static int[] results = {0, 0, ..., 0 };

 public static void Testing() {
   int op = 0;  // valor inicial do operando

   for (int idx = 0; idx < 4; idx++) {
    results[idx*12+0] = op|(op&0);
    results[idx*12+1] = op^(op|(op&0));
    results[idx*12+2] = 0^(op^(op|(op&0)));
    results[idx*12+3] = 0^(0^(op^(op|(op&0))));
    results[idx*12+4] = 0^(0^(0^(op^(op|(op&0)))));
    results[idx*12+5] = 0^(0^(0^(0^(op^(op|(op&0))))));
    results[idx*12+6] = 0^(0^(0^(0^(0^(op^(op|(op&0)))))));
    results[idx*12+7] = 0^(0^(0^(0^(0^(0^(op^(op|(op&0))))))));
    results[idx*12+8] = 0^(0^(0^(0^(0^(0^(0^(op^(op|(op&0)))))))));
    results[idx*12+9] = 0^(0^(0^(0^(0^(0^(0^(0^(op^(op|(op&0))))))))));
    results[idx*12+10]= 0^(0^(0^(0^(0^(0^(0^(0^(0^(op^(op|(op&0)))))))))));
    results[idx*12+11]= 0^(0^(0^(0^(0^(0^(0^(0^(0^(0^(op|(op&0))))))))));
    op += 21845; // incremento do operando (0x5555)
   }
 }
}
```

Figura 6.14: Rotina *Determinística Regular* para o banco de registradores

Com relação a esta rotina de teste, algumas observações se fazem necessárias. A primeira observação diz respeito à utilização de *forwarding* pelo processador Femtojava Pipeline. Como foi dito anteriormente, os valores repassados por *forwarding* não são escritos no banco de registradores. Assim, os dois últimos valores (*op* e zero) de cada linha interna ao laço, na Figura 6.14, nunca são escritos no banco de registradores, pois sempre são repassados por *forwarding*. Esses valores foram incluídos na rotina justamente para garantir a escrita dos operandos nos registradores desejados.

Também se observa que, na rotina da Figura 6.14, apenas onze dos dezesseis registradores do banco são escritos com operandos. Isso pode ser verificado na décima primeira linha interna ao laço onde, retirando-se os dois últimos valores, os quais são repassados por *forwarding*, apenas onze valores são empilhados (nove zeros e dois *op*). Isso se deve ao fato de que alguns registradores do banco são utilizados para outros propósitos. Um deles armazena informações sobre o método corrente. Outros dois contêm as duas variáveis locais do método (*op* e *idx*), motivo pelo qual apenas um operando foi utilizado visando um maior número de registradores para a pilha. Por fim, dois registradores são utilizados, pela pilha, para armazenamento da referência ao *array* e do índice onde o resultado da operação corrente deve ser escrito. No total, são aplicados 48 padrões de teste determinísticos regulares ao banco de registradores.

6.2.5 Unidades de Controle

Até a data dessa dissertação, o número de instruções implementadas na versão Multiciclo do processador Femtojava era maior que o número de instruções implementadas na versão Pipeline. Por esse motivo, foram desenvolvidas rotinas independentes para o teste da unidade de controle de cada processador.

O objetivo da rotina de teste para a unidade de controle é a aplicação de todos os *opcodes* suportados pelo processador. Uma vez que o ambiente SASHIMI não fornece suporte a todas as instruções implementadas nos processadores Femtojava, ambas as rotinas foram descritas em nível de *bytecodes*. Assim, devido à difícil compreensão

dessas rotinas em tal nível de descrição, elas não foram incluídas nesta dissertação. Entretanto, vale informar que ambas utilizam padrões determinísticos regulares e incluem todos os *opcodes* suportados por cada versão do processador Femtojava.

É importante ressaltar que nenhuma das rotinas desenvolvidas para a biblioteca de auto-teste dos processadores Femtojava contém códigos de compactação. Porém, isso não constitui uma limitação da metodologia STEP. Nesse estudo de caso não foram incluídas rotinas com compactação porque se verificou que todas as rotinas desenvolvidas resultavam em um número pequeno de respostas de teste, dispensando a necessidade de compactação. Desse modo, a inclusão de códigos de compactação resultaria em um aumento considerável de ocupação de memória de programa e de tempo de execução de teste, possivelmente não compensado pela redução do tamanho da memória de dados e do tempo de execução do analisador de respostas. Entretanto, vale lembrar que a inclusão, na biblioteca de auto-teste, de rotinas com compactação das respostas de teste possibilita uma maior exploração do espaço de teste.

No que diz respeito às outras versões do processador Femtojava, algumas considerações se fazem necessárias. Para versões com largura de dados diferentes (8 ou 32 bits), as abordagens para o desenvolvimento de rotinas de teste podem ser facilmente estendidas. Na abordagem que utiliza padrões de teste gerados por ATPG, a própria ferramenta de ATPG se encarrega de fornecer os padrões de teste adequados, desde que ela conheça a estrutura do componente alvo com a largura de dados já especificada. Para a abordagem com padrões de teste pseudo-aleatórios, basta adaptar a rotina para a largura desejada e encontrar valores adequados para a semente e para o número de padrões de teste. Já na adaptação das rotinas que geram e aplicam padrões de teste determinísticos regulares, pode ser necessária a geração de um número maior de padrões de teste. Na maioria dos casos, isso pode ser obtido pela alteração da função de transição (redução no valor do incremento dos operandos).

O desenvolvimento de rotinas de teste para a versão VLIW do processador Femtojava requer alguns cuidados. Deve-se ter em mente que todos os D-VCs replicados devem ser testados. Uma vez que esses D-VCs são idênticos aos da versão Pipeline (unidades funcionais e banco de registradores), as mesmas rotinas de teste podem ser utilizadas. Porém, é preciso lembrar que a busca por paralelismo é feita, automaticamente, por um algoritmo de análise dos *bytecodes*. Logo, é difícil garantir que as operações de teste sejam direcionadas para os componentes desejados, evitando que alguns sejam testados mais de uma vez, enquanto outros não sejam testados. Uma solução possível é a geração manual dos *bytecodes* das rotinas de teste, sem o uso do analisador de código. Desse modo pode-se deixar explícito o paralelismo desejado, assegurando que todos os componentes sejam corretamente testados. O preço pago por isso é o aumento considerável no tempo de projeto do teste.

Finalmente, para o processador Femtojava DSP são necessárias poucas alterações na biblioteca de auto-teste. A instrução *imac* deve ser incluída na rotina de teste da unidade de controle, pois tanto o somador (ULA) quanto o multiplicador já são testados por suas respectivas rotinas. É importante que alguma das rotinas utilizadas seja baseada em laço, de modo a testar o controle eficiente de laços. Sendo o *buffer* circular implementado na memória de dados, não há necessidade de inclusão de uma rotina de teste para essa estrutura. Porém, uma rotina deve ser adicionada na biblioteca de auto-teste para o teste da estrutura de endereçamento com bit reverso. Essas poucas

alterações devem resultar em uma boa cobertura de falhas para o processador Femtojava DSP, uma vez que a diferença de área dessa versão para a versão Multiciclo é de apenas 15,5%.

A Tabela 6.1 contém os custos de cada uma das rotinas de teste desenvolvidas para o processador Femtojava Multiciclo. A coluna da esquerda mostra os componentes visados e a cobertura de falhas alcançada, para o respectivo componente, com qualquer das abordagens implementadas. Os valores de cobertura de falhas, para o modelo *stuck-at*, foram obtidos com a ferramenta Flextest® (Mentor Graphics). A segunda coluna apresenta os fatores de custo considerados, que são, em ordem: número de padrões/respostas de teste, tamanho da memória de dados ocupada, tamanho da memória de programa ocupada, tempo de execução e energia consumida.

Tabela 6.1: Custos das rotinas de teste para o processador Femtojava Multiciclo

		ATPG Imediato	***ATPG Memória***	***LFSR***	***Determinística Regular***
Multiplic.	***# Padrões/Respostas***	29	29	56	64
	***Tamanho RAM** (bytes)*	64	192	118	134
	***Tamanho ROM** (bytes)*	490	33	118	54
99.60%	***Tempo Exec.** (ciclos mem.)*	3.826	5.792	225.784	13.272
	***Energia** (num. chav. cap.)*	2.238.657	3.338.339	132.824.561	7.675.773
Deslocador	***# Padrões/Respostas***	14	14	---	32
	***Tamanho RAM** (bytes)*	34	102	---	70
	***Tamanho ROM** (bytes)*	214	83	---	94
91.50%	***Tempo Exec.** (ciclos mem.)*	1.822	3.614	---	5.550
	***Energia** (num. chav. cap.)*	1.094.418	2.089.685	---	3.237.166
ULA	***# Padrões/Respostas***	27	27	---	24
	***Tamanho RAM** (bytes)*	60	168	---	54
	***Tamanho ROM** (bytes)*	426	152	---	118
94.20%	***Tempo Exec.** (ciclos mem.)*	3.402	8.044	---	3.512
	***Energia** (num. chav. cap.)*	1.977.477	4.677.435	---	2.086.465
Controle	***# Padrões/Respostas***	---	---	---	10
	***Tamanho RAM** (bytes)*	---	---	---	48
	***Tamanho ROM** (bytes)*	---	---	---	198
93,80%	***Tempo Exec.** (ciclos mem.)*	---	---	---	1.640
	***Energia** (num. chav. cap.)*	---	---	---	964.242

Uma vez que não há compactação, o número de padrões de teste é igual ao número de respostas de teste, à exceção da rotina para a unidade de controle, para qual é mostrada apenas o número de respostas de teste. Os tamanhos de memória, expressos em bytes, foram obtidos a partir dos *bytecodes* gerados pela ferramenta SASHIMI. O tempo de execução das rotinas é mostrado em número de ciclos de memória (que equivalem a duas vezes o número de ciclos do processador) e foram obtidos por simulações na ferramenta Quartus II® (Altera). Finalmente, os valores de consumo de energia são expressos em número de chaveamentos de capacitâncias de portas, e foram obtidos através do simulador CACO-PS (BECK et al., 2003).

Tomando-se como exemplo os custos das rotinas para o multiplicador na Tabela 6.1, verifica-se que a abordagem com padrões gerados por ATPG apresenta o menor número de respostas de teste, resultando em um menor tempo de execução do analisador de respostas. A rotina *ATPG Imediato* tem a menor ocupação de memória RAM entre

todas, mas tem a maior ocupação de memória ROM. Por outro lado, a rotina *ATPG Memória* apresenta o menor tamanho de memória ROM, mas o maior tamanho de memória RAM. A abordagem pseudo-aleatória (*LFSR*) tem alto custo em termos de ocupação de memória, e custo excessivamente alto em termos de tempo de execução e consumo de energia quando comparada às outras rotinas. Por esse motivo, e também por seu alto custo de desenvolvimento, essa rotina não foi desenvolvida para os demais componentes. Por fim, a abordagem com padrões determinísticos regulares apresenta um bom compromisso entre tamanho de memória de dados e tamanho de memória de programa. Entretanto, essa rotina tem tempo de execução e consumo de energia bastante elevados em relação à abordagem baseada em ATPG.

O mesmo comportamento pode ser observado nos custos das rotinas para o deslocador. Já para a ULA, a rotina *Determinística Regular* vence em quase todos os critérios. Essa abordagem só perde para a rotina *ATPG Imediato* no tempo de execução e no consumo de energia e, mesmo assim, por diferenças muito pequenas. Tal vantagem se deve ao fato de que a ULA tem uma estrutura bem simples e extremamente regular. Assim, é possível atingir uma boa cobertura de falhas para esse componente com bem poucos padrões de teste, menos até do que o número de padrões gerados por ATPG. Os custos da rotina *Regular Determinística* desenvolvida para a unidade de controle também são mostrados na Tabela 6.1, embora não haja comparações possíveis.

Na Tabela 6.2, os custos das mesmas rotinas desenvolvidas para as unidades funcionais do Femtojava Multiciclo são mostrados, agora, para o processador Femtojava Pipeline. Os valores correspondentes ao número de padrões/respostas de teste e tamanhos das memórias ocupadas são idênticos aos valores mostrados na tabela anterior, uma vez que as rotinas são as mesmas. Porém, os tempos de execução e consumos de energia diferem daqueles obtidos para a versão Multiciclo do processador. Ainda assim, pode-se verificar, na Tabela 6.2, que as observações feitas em relação às variações de custos entre as diferentes abordagens desenvolvidas permanecem válidas para o processador Femtojava Pipeline. Os custos das rotinas desenvolvidas para o banco de registradores e para a unidade de controle desse processador também são apresentados na Tabela 6.2.

Com base nos custos das rotinas de teste mostrados na Tabela 6.1 e na Tabela 6.2, é possível constatar que não há uma abordagem ótima para o teste de todos os componentes do processador, mesmo que apenas os D-VCs sejam visados. O melhor programa de teste deve ser uma combinação de diferentes abordagens. Para sistemas de propósitos gerais, sem as fortes restrições de um sistema embarcado, poder-se-ia aplicar as regras propostas por Paschalis e Gizopoulos (PASCHALIS; GIZOPOULOS, 2005), apresentadas na Seção 4.3.3.4. Entretanto, quando se deseja aplicar auto-teste *on-line* a processadores embarcados, é indispensável a utilização de um método eficiente para a seleção das rotinas que formarão o programa de auto-teste. Os métodos de seleção propostos neste trabalho garantem a composição de um programa de auto-teste de baixo custo, que atenda todas as restrições do sistema embarcado e ainda os requisitos de tempo-real da aplicação.

Tabela 6.2: Custos das rotinas de teste para o processador Femtojava Pipeline

		ATPG Imediato	*ATPG Memória*	*LFSR*	*Determinística Regular*
Multiplic.	***# Padrões/Respostas***	29	29	56	64
	***Tamanho RAM** (bytes)*	64	192	118	134
	***Tamanho ROM** (bytes)*	490	33	118	54
99.60%	***Tempo Exec.** (ciclos mem.)*	1.724	2.322	60.498	5.074
	***Energia** (num. chav. cap.)*	2.501.716	2.914.116	82.781.529	7.144.308
Deslocador	***# Padrões/Respostas***	14	14	---	32
	***Tamanho RAM** (bytes)*	34	102	---	70
	***Tamanho ROM** (bytes)*	214	83	---	94
91.50%	***Tempo Exec.** (ciclos mem.)*	884	1.508	---	2.212
	***Energia** (num. chav. cap.)*	1.242.763	1.869.783	---	2.739.009
ULA	***# Padrões/Respostas***	27	27	---	24
	***Tamanho RAM** (bytes)*	60	168	---	54
	***Tamanho ROM** (bytes)*	426	152	---	118
94.20%	***Tempo Exec.** (ciclos mem.)*	1.564	3.384	---	1.596
	***Energia** (num. chav. cap.)*	2.281.602	4.256.049	---	2.075.039
Banco de Regs.	***# Padrões/Respostas***	---	---	---	48
	***Tamanho RAM** (bytes)*	---	---	---	102
	***Tamanho ROM** (bytes)*	---	---	---	340
80,95%	***Tempo Exec.** (ciclos mem.)*	---	---	---	4.196
	***Energia** (num. chav. cap.)*	---	---	---	5.070.477
Controle	***# Padrões/Respostas***	---	---	---	11
	***Tamanho RAM** (bytes)*	---	---	---	50
	***Tamanho ROM** (bytes)*	---	---	---	203
88,45%	***Tempo Exec.** (ciclos mem.)*	---	---	---	924
	***Energia** (num. chav. cap.)*	---	---	---	1.176.814

6.3 O Analisador de Respostas de Teste

O desenvolvimento do analisador de respostas constitui a segunda etapa do método STEP para o projeto e aplicação de auto-teste *on-line* periódico em processadores embarcados. Entretanto, essa etapa não foi realizada neste estudo de caso. O primeiro motivo para isso foi o tempo limitado para o desenvolvimento do presente trabalho. A proposta original desta dissertação não previa o projeto de nenhum módulo de hardware. Quando, durante a realização do trabalho, foi verificada a necessidade de um módulo responsável pela análise *on-line* das respostas de teste, o projeto de tal módulo foi desenvolvido e incluído na metodologia. Porém, uma vez que o método STEP pode ser parcialmente validado sem a implementação do analisador de respostas, essa etapa foi suprimida.

A continuidade do fluxo de projeto depende unicamente do conhecimento sobre o tempo de execução do analisador de respostas em função do número de respostas de teste. Assim como no projeto do analisador apresentado na Seção 5.3, considerou-se que esse tempo, em ciclos de memória, é igual a duas vezes o número de respostas de teste do programa de auto-teste. Desse modo, pôde-se prosseguir com este estudo de caso sem qualquer impacto nos resultados obtidos. Caso o valor considerado seja incorreto, a única diferença será observada no período de teste calculado e fornecido pelos métodos de seleção, o que, de modo algum, invalida a metodologia proposta.

6.4 A Seleção do Programa de Auto-Teste

A terceira e última etapa da metodologia STEP consiste na seleção do programa de auto-teste. Neste estudo de caso foram desenvolvidos e aplicados os dois métodos de seleção propostos nesta dissertação: o método pseudo-exaustivo e o método heurístico. Embora a biblioteca de auto-teste para os processadores Femtojava seja pequena o suficiente para a utilização apenas do método de seleção pseudo-exaustivo, ambos os métodos foram validados com resultados práticos. Os próximos parágrafos descrevem a implementação de cada um dos métodos com algumas características específicas para a plataforma Femtojava.

6.4.1 Método Pseudo-Exaustivo

O primeiro passo do método pseudo-exaustivo, o *Cálculo de Custos*, é responsável pela construção do espaço de busca. Nesse passo é atribuído um número identificador *i* para cada programa de teste candidato, ou seja, para cada possível combinação de rotinas de teste. Além disso, são calculados e armazenados os custos de todos os candidatos, relativos a quatro fatores: ocupação da memória de dados (indicado por *RAM*), ocupação da memória de programa (indicado por *ROM*), tempo de execução (diretamente relacionado ao período de teste e indicado por *TE*) e consumo de energia (indicado por *CE*). O cálculo dos custos segue a equação abaixo.

$$V_{i,j} = \sum_{k=1}^{C} v_{i,j,k} + u_j, \qquad 1 \le i \le Tam_{EB}, \qquad j \in F = \{RAM, ROM, TE, CE\} \qquad (6.1)$$

onde $V_{i,j}$ é o custo do fator *j* para o programa de teste candidato *i*, e $v_{i,j,k}$ é o custo do fator *j* para a rotina do componente *k* no programa de teste candidato *i*. *C* representa o número de componentes visados pelo programa de auto-teste, enquanto u_j é o custo do fator *j* para a união de todas as rotinas de teste. O tamanho do espaço de busca Tam_{EB} é descrito na Equação 5.4 da Seção 5.4.2.

O segundo passo do método, *Verificação da RAM*, é executado se o valor da restrição R_{RAM} para o tamanho da memória de dados total do sistema é maior do que zero. Nesse caso, são aprovados os programas de teste candidatos *i* que satisfazem a inequação

$$V_{i,RAM} + V_{AP,RAM} \le R_{RAM} \qquad (6.2)$$

onde $V_{i,RAM}$ é o custo em termos de tamanho de memória de dados para o candidato *i*, e $V_{AP,RAM}$ é o tamanho da memória RAM ocupada pela aplicação (tanto pelas tarefas da aplicação quanto pelo escalonador do sistema).

O terceiro passo do método, *Verificação da ROM*, é executado se o valor da restrição R_{ROM} para o tamanho da memória de programa total do sistema é maior do que zero. Nesse caso, são aprovados os programas de teste candidatos *i* que satisfazem a inequação

$$V_{i,ROM} + V_{AP,ROM} \le R_{ROM} \qquad (6.3)$$

onde $V_{i,ROM}$ é o custo em termos de tamanho de memória de programa para o candidato *i*, e $V_{AP,ROM}$ é o tamanho da memória ROM ocupada pela aplicação (tanto pelas tarefas da aplicação quanto pelo escalonador do sistema).

O quarto passo do método, *Verificação do Período de Teste*, é executado se o valor da restrição R_{PT} para o máximo período de teste requerido é maior do que zero. Nesse caso, são aprovados os programa de teste candidatos *i* que satisfazem a inequação

$$NUP_i + NUP_{AP} \leq NUP_{ESC} \qquad (6.4)$$

onde NUP_i denota o nível de utilização do processador pelo programa de teste candidato *i*, NUP_{AP} representa o nível de utilização do processador pela aplicação (tanto pelas tarefas da aplicação quanto pelo escalonador do sistema), e NUP_{ESC} indica o máximo nível de utilização do processador permitido pelo algoritmo de escalonamento utilizado pelo sistema.

Sabendo-se que nos processadores Femtojava as tarefas da aplicação são estáticas, periódicas e independentes, e considerando-se que seus períodos são iguais aos seus *deadlines*, o valor de NUP_{AP} pode ser facilmente calculado pela equação

$$NUP_{AP} = \sum_{t=1}^{T} \frac{TE_t}{P_t} + TUP_{ESC} \qquad (6.5)$$

onde *T* é o número de tarefas da aplicação, TE_t é o tempo de execução no pior caso da tarefa *t*, e P_t é período da tarefa *t*. TUP_{ESC} representa a taxa de utilização do processador pelo escalonador, valor esse previamente obtido por simulação.

Sabendo-se também que a memória de dados dos processadores Femtojava é *single port* e, por isso, o tempo de execução do analisador de respostas deve ser incluído no tempo de execução do programa de auto-teste (ver Seção 5.4.1), o valor de NUP_i, na Equação 6.4, é dado por

$$NUP_i = \frac{V_{i,TE} + TE_{AR,i}}{R_{PT}} \qquad (6.6)$$

onde $V_{i,TE}$ é o custo em termos de tempo de execução para o programa de teste candidato *i*, TE_{AR} é tempo de execução do analisador de respostas (dado pela Equação 5.2 da Seção 5.3) para o candidato *i*, e R_{PT} denota a restrição para o máximo período de teste requerido pelo sistema.

Ainda em relação à Equação 6.4, o valor de NUP_{ESC} depende da política de escalonamento utilizada pelo sistema. Sabendo-se que os processadores Femtojava têm duas políticas de escalonamento implementadas, RM e EDF, o nível permitido de utilização do processador pode ser (LEUNG; WHITEHEAD, 1992)

$$NUP_{RM} = T.\left(2^{\frac{1}{T}} - 1\right) \qquad (6.7)$$

onde *T* indica o número de tarefas da aplicação, ou

$$NUP_{EDF} = 1 \qquad (6.8)$$

O quinto passo do método, *Verificação do Consumo de Energia*, é executado se o valor da restrição R_{CE} para o consumo de energia total do sistema é maior do que zero. Nesse caso, são aprovados os programas de teste candidatos *i* que satisfazem a inequação

$$V_{i,CE} + V_{AP,CE} \leq R_{CE} \qquad (6.9)$$

onde $V_{i,CE}$ é o custo em termos de consumo de energia para o candidato *i*, e $V_{AP,CE}$ é a energia consumida pela aplicação (tanto pelas tarefas da aplicação quanto pelo escalonador do sistema).

O sexto passo do método pseudo-exaustivo, *Seleção do Programa de Auto-Teste*, seleciona o programa de auto-teste *i* que resulta no menor valor para a função de custo. Nessa função, a soma dos pesos de todos os fatores de custo deve ser igual a um. Visando simplificar o problema de otimização multivariável, uma única função de custo normalizada foi definida como

$$FC_i = \sum_{j}^{j \in F} \left(w_j . \frac{\sum_{k=1}^{C} v_{i,j,k}}{\sum_{k=1}^{C} \min\left(v_{j,k}\right)} \right), \qquad 1 \le i \le Tam_{EB}, \quad F = \{RAM, ROM, TE, CE\} \qquad (6.10)$$

onde *F* é o conjunto dos fatores de custo, w_j representa o peso atribuído pelo projetista do sistema ao fator *j*, e *C* é o número de componentes alvo do programa de auto-teste. $v_{i,j,k}$ denota o custo do fator *j* para a rotina do componente *k* no programa de teste candidato *i*, e $min(v_{j,k})$ é o menor custo para o fator *j* dentre todas as rotinas de teste implementadas para o componente *k*. Assim, os custos do fator *j* para um componente *k* estão normalizados, de modo que todos os fatores possam ser analisados e otimizados em uma mesma equação.

O sétimo e último passo, *Cálculo do Período de Teste Mínimo*, calcula o menor período de teste, para o programa de auto-teste selecionado, permitido pela aplicação de tempo-real. O período de teste *PT* (igual ao *deadline*) para o candidato *i* selecionado é dado pela equação

$$PT_i = \frac{V_{i,TE} + TE_{AR,i}}{NUP_{ESC} - NUP_{AP}} \qquad (6.11)$$

onde $V_{i,TE}$ é o tempo de execução do programa de auto-teste *i*, e $TE_{AR,i}$ é o tempo de execução do analisador de respostas para o programa de auto-teste *i*. NUP_{ESC} denota o nível máximo de utilização do processador permitido pelo escalonador utilizado, e é dado pela Equação 6.7, para o algoritmo RM, ou pela Equação 6.8, para o algoritmo EDF. NUP_{AP} é o nível de utilização do processador pela aplicação (tarefas da aplicação e escalonador), dado pela Equação 6.5.

Após o sétimo passo, o método fornece as rotinas de teste que compõem o programa de auto-teste selecionado no passo seis. Além disso, são fornecidos o período de teste calculado no passo sete, pela Equação 6.11, e os custos do sistema calculados nos passos dois, três e cinco, respectivamente nas Equações 6.2, 6.3 e 6.9.

6.4.2 Método Heurístico

A primeira etapa do método heurístico, a *Pré-Seleção do Programa de Auto-Teste*, é responsável pela pré-seleção de uma rotina para cada componente alvo do programa de auto-teste. A rotina *l* selecionada para o componente *k* é aquela que resulta no menor valor para a função de custo $FC_{k,l}$. Nessa função, a soma dos pesos de todos os fatores de custo deve ser igual a um. Visando simplificar o problema de otimização multivariável, uma única função de custo normalizada foi definida como

$$FC_{k,l} = \sum_{j}^{j \in F} \left(w_j . \frac{v_{j,k,l}}{\min(v_{j,k})} \right), \qquad 1 \le k \le C, \ 1 \le l \le S, \ F = \{RAM, ROM, TE, CE\} \quad (6.12)$$

onde F é o conjunto de fatores de custo, e w_j representa o peso atribuído pelo projetista do sistema ao fator de custo j. $v_{j,k,l}$ indica o custo para o fator j da rotina de teste l para o componente k, e $min(v_{j,k})$ é o menor custo para o fator j dentre todas as rotinas de teste implementadas para o componente k. Assim, os custos do fator j para um componente k estão normalizados, de modo que todos os fatores possam ser analisados e otimizados em uma mesma equação.

Na etapa de *Verificação de Requisito*, o programa de auto-teste pré-selecionado é verificado quanto ao atendimento das restrições do sistema embarcado alvo. Em cada execução dessa etapa, é verificado um dos fatores de custo cujos valores das restrições R_{RAM} para o tamanho da memória de dados total do sistema, R_{ROM} para o tamanho da memória de programa total do sistema, R_{PT} para o máximo período de teste requerido e R_{CE} para o consumo de energia total do sistema são maiores do que zero. As restrições são verificadas na ordem decrescente dos pesos atribuídos pelo projetista aos respectivos fatores de custo. Antes da verificação de cada requisito, o custo do programa de auto-teste para o respectivo fator deve ser calculado. O custo $V_{i,j}$, para o fator j, do programa de auto-teste pré-selecionado i, é calculado da mesma forma que no método pseudo-exaustivo, pela Equação 6.1.

A verificação dos requisitos R_{RAM}, R_{ROM} e R_{CE} para o programa de teste pré-selecionado i seguem, respectivamente, as Equações 6.2, 6.3 e 6.9. A verificação do requisito R_{PT} difere, somente na forma, daquela implementada no método pseudo-exaustivo, sendo realizada através da inequação

$$PT_i \le R_{PT} \qquad (6.13)$$

onde PT_i é o menor período de teste, para o programa de auto-teste pré-selecionado i, permitido pela aplicação de tempo real, considerando-se tarefas estáticas, independentes e periódicas (com período igual ao *deadline*). O cálculo do período de teste mínimo é realizado pela equação 6.11, onde NUP_{AP} é dado pela Equação 6.5 e NUP_{ESC} é dado pela Equação 6.7, para o algoritmo de escalonamento RM, ou pela Equação 6.8, para o algoritmo EDF.

Caso o programa de auto-teste pré-selecionado não atenda a um dos requisitos do sistema, o método segue para a etapa de *Eliminação de Rotina*. A rotina, para o componente k, eliminada do programa de auto-teste pré-selecionado i é aquela que apresenta o maior custo relativo $CR_{i,j,k}$ para o fator de custo j cuja restrição não foi atendida. O valor do custo relativo é dado pela equação

$$CR_{i,j,k} = \frac{v_{i,j,k}}{\min(v_{j,k})} \qquad (6.14)$$

onde $v_{i,j,k}$ representa o custo do fator j para a rotina do componente k no programa de auto-teste pré-selecionado i, e $min(v_{j,k})$ é o menor custo para o fator j dentre todas as rotinas de teste implementadas para o componente k.

Após essa etapa, é feito o *Ajuste de Pesos* na função de custo. Nessa implementação do método, o valor do ajuste é igual a *0,09*. Logo, o peso do fator de custo cuja restrição não foi atendida é acrescido de *0,09*, enquanto os pesos dos outros três fatores são

diminuídos em *0,03*, de modo a manter o equilíbrio da função de custo. Feito o ajuste de pesos, o laço é reiniciado.

No momento em que o programa de auto-teste pré-selecionado atender todas as restrições do sistema, o método fornece como resposta as rotinas de teste que o compõem. Além disso, são fornecidos o tamanho total ocupado da RAM, o tamanho total ocupado da ROM, o período de teste mínimo e o consumo de energia total do sistema, todos calculados na etapa de *Verificação de Requisito*, respectivamente, pelas Equações 6.2, 6.3, 6.11 e 6.9.

A validação de ambos os métodos de seleção, pseudo-exaustivo e heurístico, foi realizada tendo como plataforma alvo o processador Femtojava Multiciclo, com suas duas políticas de escalonamento. A biblioteca de auto-teste para o Femtojava Multiciclo tem onze rotinas de teste, e um espaço de busca com trinta e seis soluções possíveis (quatro rotinas para o multiplicador vezes três rotinas para o deslocador vezes três rotinas para a ULA vezes uma rotina para a unidade de controle). Embora seja um espaço de busca bastante pequeno, ele é suficiente para a verificação da funcionalidade e da importância de ambos os métodos. Os custos das rotinas pertencentes à biblioteca de auto-teste são aqueles apresentados na Tabela 6.1.

O conjunto de tarefas que compõe a aplicação de tempo-real usada nos experimentos é uma adaptação das séries PN definidas no *benchmark* Hartstone (WEIDERMAN; KAMENOFF, 1992). Séries PN são formadas por tarefas *Periódicas* e *Não harmônicas* e representam uma quantidade considerável das aplicações de tempo-real. A Tabela 6.3 contém as características relevantes das seis tarefas utilizadas nos experimentos. Os custos dos algoritmos de escalonamento para o processador Femtojava Multiciclo (BECKER et al., 2004) também são mostrados na Tabela 6.3. Os tempos de execução e os períodos (iguais aos *deadlines*) são expressos em número de ciclos de memória. Os tamanhos de memória são representados em bytes e os valores para consumo de energia são expressos em número de chaveamentos de capacitâncias de portas. A taxa de uso do processador é dada pela razão entre o tempo de execução e o período da tarefa. Note-se que as taxas de uso do processador pelas tarefas da aplicação são baixas porque os escalonadores necessitam de muito tempo do processador.

Tabela 6.3: Custos das tarefas de tempo-real e do escalonamento

	Tempo de Execução *(ciclos mem.)*	***Período*** *(ciclos mem.)*	***Taxa de Uso Proc.***	***Tamanho RAM*** *(bytes)*	***Tamanho ROM*** *(bytes)*	***Energia*** *(num. chav. cap)*
Tarefa 1	171.788	5.000.000	0,0344	4.324	2.141	37.562.141
Tarefa 2	97.114	1.850.000	0,0525	3.602	1.263	42.982.637
Tarefa 3	5.407	900.000	0,0060	2.322	1.874	9.884.025
Tarefa 4	109.251	3.400.000	0,0321	1.558	498	84.656.389
Tarefa 5	43.698	870.000	0,0502	2.238	1.476	52.578.521
Tarefa 6	116.070	1.800.000	0,0645	5.720	1.432	75.847.452
Escalon. RM	---	---	0,3165	206	295	535.000.000
Escalon. EDF	---	---	0,3813	208	358	684.000.000

Foram realizados dez experimentos com diferentes características de projeto. Cada experimento difere dos demais nos pesos fornecidos para a função de custo e/ou na política de escalonamento do sistema alvo e/ou, ainda, nas restrições de projeto quanto

ao máximo período de teste requerido e quanto ao máximo consumo de energia permitido para o sistema alvo. Em todos os casos, a memória de dados (RAM) é restrita a *20KB* (20.480 bytes), enquanto a memória de programa (ROM) é restrita a *10KB* (10.240 bytes). Todos os experimentos foram realizados tanto com a utilização do método de seleção pseudo-exaustivo, quanto com o método heurístico.

A Tabela 6.4 apresenta as características dos dez experimentos realizados e os resultados obtidos com o método de seleção pseudo-exaustivo. A coluna mais à esquerda contém um número identificador para cada experimento. As próximas duas colunas mostram as características do projeto analisado. A primeira dessas colunas contém os pesos fornecidos para a função de custo (para os fatores memória RAM, memória ROM, tempo de execução e consumo de energia, nessa ordem) e a segunda mostra a política de escalonamento utilizada pelo sistema alvo. As próximas três colunas apresentam o programa de auto-teste selecionado pelo método pseudo-exaustivo, com as rotinas escolhidas para o multiplicador, o deslocador e a ULA do processador Femtojava Multiciclo, nessa ordem. Note-se que a rotina para a unidade de controle não é mostrada, uma vez que não existem opções. Porém, esse componente também é considerado durante o processo de seleção. Por fim, as quatro colunas mais à direita contêm os custos totais e o período de teste calculados pelo método de seleção, para o programa de auto-teste selecionado. Os tamanhos de memória são expressos em bytes, o consumo de energia é representado em número de chaveamentos de capacitâncias de portas e o período de teste é mostrado em ciclos de memória.

Tabela 6.4: Resultados da seleção com o método pseudo-exaustivo

	Projeto		**Programa de Auto-Teste Selecionado**			**Resultados**			
	Pesos RA|RO|TE|CE	***Esc.***	***Multiplic.***	***Deslocador***	***ULA***	***RAM*** *(bytes)*	***ROM*** *(bytes)*	***Energia*** *(10^3 chav)*	***Per. de Teste*** *(ciclos)*
1	0,1|0,1|0,6|0,2	RM	Atpg Imed.	Det. Reg.	Det. Reg.	20.238	9.937	846.937	84.431
2	0,1|0,1|0,6|0,2	EDF	Det. Reg.	Atpg Mem.	Atpg Imed.	20.348	9.861	1.000.118	58.042
	Restrições: Período de Teste = 100.000 ciclos / Consumo de Energia = 1.000.000 x10^3 chav cap.								
3	0,1|0,5|0,2|0,2	RM	Atpg Mem.	Atpg Imed.	Atpg Imed.	20.336	9.908	845.785	73.400
4	0,1|0,5|0,2|0,2	EDF	Atpg Mem.	Atpg Imed.	Atpg Mem.	20.446	9.697	997.485	45.641
5	0,1|0,1|0,6|0,2	RM	Atpg Mem.	Det. Reg.	Det. Reg.	20.366	9.480	848.037	95.833
6	0,1|0,1|0,6|0,2	EDF	Det. Reg.	Atpg Imed.	Det. Reg.	20.274	9.684	999.232	53.588
7	---	RM	Det. Reg.	Det. Reg.	Det. Reg.	20.308	9.501	852.374	139.621
	Restrições: Período de Teste = 30.000 ciclos / Consumo de Energia = 1.000.000 x10^3 chav cap.								
8	0,1|0,5|0,2|0,2	RM	Não existe solução possível			---	---	---	---
9	0,1|0,5|0,2|0,2	EDF	Atpg Imed.	Atpg Imed.	Det. Reg.	20.204	10.120	993.794	28.480
10	---	EDF	Det. Reg.	Det. Reg.	Det. Reg.	20.310	9.564	1.001.374	63.520

Ainda em relação à Tabela 6.4, nota-se que os experimentos estão agrupados de acordo com as restrições de período de teste e consumo de energia. Nos experimentos *1* e *2*, não existem tais restrições. Para os projetos de *3* a *7*, o período de teste é restrito a *100.000* ciclos de memória e o consumo de energia é restrito a *1.000.000.000* chaveamentos de capacitâncias de portas. Já nos experimentos de *8* a *10*, a restrição de período de teste foi reduzida para *30.000* ciclos de memória, enquanto a restrição para o consumo de energia não foi alterada. É importante ressaltar, também, que nos experimentos *7* e *10*, programas de auto-teste compostos unicamente por rotinas determinísticas regulares foram propositadamente e manualmente selecionados, sem o

uso do método de seleção. O objetivo disso é possibilitar a comparação dos métodos de seleção aqui propostos, com as regras de aplicabilidade das rotinas de teste sugeridas por Paschalis e Gizopoulos (PASCHALIS; GIZOPOULOS, 2005), e apresentadas na Seção 4.3.3.4. De acordo com tais regras, a rotina determinística regular é a mais indicada para aplicação em componentes visíveis de dados.

É possível observar, nos resultados fornecidos pela Tabela 6.4, que uma pequena modificação em uma ou mais características de projeto resulta na seleção de um programa de auto-teste completamente diferente. Além disso, nota-se que em todos os casos, os custos totais do sistema alvo, calculados e fornecidos pelo método de seleção, atendem todas as restrições do projeto. Em tais custos inclui-se também o máximo período de teste requerido. A exceção a isso ocorre apenas nos experimentos *7* e *10*, nos quais os programas de auto-teste foram selecionados com base nas regras definidas por Paschalis e Gizopoulos, e não nos métodos de seleção aqui propostos. Em ambos os experimentos, os resultados obtidos não atendem as respectivas restrições de período de teste, conforme pode ser verificado pelos valores hachurados. Logo, os resultados da Tabela 6.4 atestam a necessidade de utilização de um método eficiente para a seleção do programa de auto-teste mais adequado para dado projeto, de forma a atender todos os requisitos e restrições do sistema alvo. Tais resultados também comprovam a eficácia do metodo pseudo-exaustivo proposto.

A Tabela 6.5 mostra os resultados, para os mesmos experimentos, agora obtidos com o método heurístico de seleção do programa de auto-teste. A organização dos dados dessa tabela, bem como as unidades em que os valores são expressos são as mesmas da Tabela 6.4. Porém, aqui foram omitidos os experimentos *7* e *10*, pois nesses casos os programas de auto-teste foram selecionados manualmente, sem a utilização do método de seleção, e os resultados são idênticos aos apresentados na tabela anterior. Novamente, a rotina de teste para a unidade de controle bem como a rotina para o banco de registradores não são mostradas na tabela, visto que em ambos os casos não existem opções para serem selecionadas. Entretanto, ambos os componentes são considerados durante o processo de seleção do programa de auto-teste.

Tabela 6.5: Resultados da seleção com o método heurístico

	Projeto		**Programa de Auto-Teste Selecionado**			**Resultados**			
	Pesos RA|RO|TE|CE	***Esc.***	***Multiplic.***	***Deslocador***	***ULA***	***RAM*** *(bytes)*	***ROM*** *(bytes)*	***Energia*** *(10^3 chav)*	***Per. de Teste*** *(ciclos)*
1	0,1|0,1|0,6|0,2	RM	Atpg Mem.	Atpg Imed.	Det. Reg.	20.330	9.600	845.894	74.003
2	0,1|0,1|0,6|0,2	EDF	Atpg Mem.	Atpg Imed.	Det. Reg.	20.332	9.663	994.894	33.667
	Restrições: Período de Teste = 100.000 ciclos / Consumo de Energia = 1.000.000 x10^3 chav cap.								
3	0,1|0,5|0,2|0,2	RM	Atpg Mem.	Atpg Mem.	Det. Reg.	20.398	9.469	846.889	84.396
4	0,1|0,5|0,2|0,2	EDF	Atpg Mem.	Atpg Mem.	Det. Reg.	20.400	9.532	995.889	38.396
5	0,1|0,1|0,6|0,2	RM	Atpg Mem.	Atpg Imed.	Det. Reg.	20.330	9.600	845.894	74.003
6	0,1|0,1|0,6|0,2	EDF	Atpg Mem.	Atpg Imed.	Det. Reg.	20.332	9.663	994.894	33.667
	Restrições: Período de Teste = 30.000 ciclos / Consumo de Energia = 1.000.000 x10^3 chav cap.								
8	0,1|0,5|0,2|0,2	RM	Não existe solução possível			---	---	---	---
9	0,1|0,5|0,2|0,2	EDF	Atpg Imed.	Atpg Imed.	Det. Reg.	20.204	10.120	993.794	28.480

Verifica-se na Tabela 6.5 que, para todos os experimentos, os resultados fornecidos pelo método heurístico também garantem o atendimento de todas as restrições do sistema, incluido a restrição de período de teste. Porém, comparando-se os resultados da Tabela 6.5 com os resultados da Tabela 6.4, é possível observar que, na maior parte dos experimentos, os programas de auto-teste selecionados por ambos os métodos diferem. Conforme dito anteriormente, isso ocorre devido à característica heurística do segundo método, que implica na eliminação de soluções possíveis. Entretanto, os custos totais fornecidos por ambos os métodos mostram que as diferenças de custo para os programas de auto-teste selecionados é bem pequena, indicando que ambos os resultados são adequados para o projeto em questão, embora um deles não seja a solução ótima. Com relação ao experimento *8*, o método de seleção não foi capaz de encontrar uma solução porque não existe uma solução possível, dado que a mesma resposta foi fornecida pelo método pseudo-exaustivo. Logo, os resultados apresentados na Tabela 6.5 comprovam a eficácia do metodo de seleção heurístico, embora ele não seja a opção indicada para a aplicação alvo, devido ao tamanho reduzido da biblioteca de auto-teste.

6.5 A Ferramenta STEP-FJ

Com o objetivo de automatizar o projeto e a aplicação de auto-teste *on-line* periódico para os processadores da família Femtojava, foi desenvolvida, em linguagem Java, uma ferramenta denominada STEP-FJ (*Self-Test for Femtojava Embedded Processors*). Tal ferramenta implementa ambos os métodos de seleção da metodologia STEP e fornece, além dos dados resultantes dos processos de seleção, arquivos para simulação do programa de auto-teste selecionado nas ferramentas Quartus II® (Altera) e Flextest® (Mentor Graphics).

Embora seja especificamente voltado para os processadores Femtojava, o STEP-FJ permite a execução de ambos os métodos de seleção do programa de auto-teste para outras plataformas. Nesse caso, a ferramenta não fornece os arquivos para simulação, apenas indica as rotinas de teste selecionadas e informa os custos totais do sistema, bem como o período de teste mínimo calculado. Uma vez que a implementação da ferramenta seguiu as equações descritas na Seção 6.4, a aplicação alvo deve ter as mesmas características das aplicações suportadas pelos processadores Femtojava. Ou seja, as tarefas devem ser estáticas, independentes e periódicas, com períodos iguais aos *deadlines*. Além disso, a memória RAM do sistema deve ser *single port*, com analisador de respostas e processador compartilhando a mesma porta de acesso.

Todas as informações acerca do projeto e do sistema alvo (restrições, pesos para a função de custo e custos de escalonamento), da aplicação (custos e períodos das tarefas) e da biblioteca de auto-teste (custos das rotinas de teste e para a união das mesmas) são lidas pela ferramenta a partir de arquivos texto. Para a plataforma Femtojava, o arquivo contendo as informações da biblioteca de auto-teste já foi previamente escrito. Entretanto, à medida que novas rotinas de teste forem incluídas na biblioteca, esse arquivo pode ser facilmente incrementado com os novos dados. Para outras plataformas, esse arquivo deve ser desenvolvido juntamente com a biblioteca de auto-teste. Os demais arquivos devem ser escritos especificamente para cada projeto de sistema. Todos os arquivos de entrada para a ferramenta STEP-FJ são exemplificados no Apêndice B.

A Figura 6.15 ilustra a tela inicial do STEP-FJ, com as opções de execução da ferramenta. Note-se que apenas as opções *–e* e *–h* que executam, respectivamente, o método pseudo-exaustivo e o método heurístico, são válidas para outras plataformas que não a Femtojava. As opções *–m* e *–p* geram os arquivos (de extensão *.mif*) com os conteúdos das memórias de dados e de programa para simulação, na ferramenta Quartus II® (Altera), do programa de auto-teste para o Femtojava Multiciclo e para o Femtojava Pipeline, respectivamente. Ambas as opções também geram um arquivo do mesmo tipo com o conteúdo da memória ROM do analisador de respostas. Finalmente, a opção *–f* gera todos os arquivos necessários à ferramenta Flextest® (Mentor Graphics) para a verificação da cobertura de falhas total, para o processador Femtojava indicado, obtida com o programa de auto-teste selecionado. Entre esses arquivos há um *script* (de extensão *.dofile*) para a execução do Flextest®.

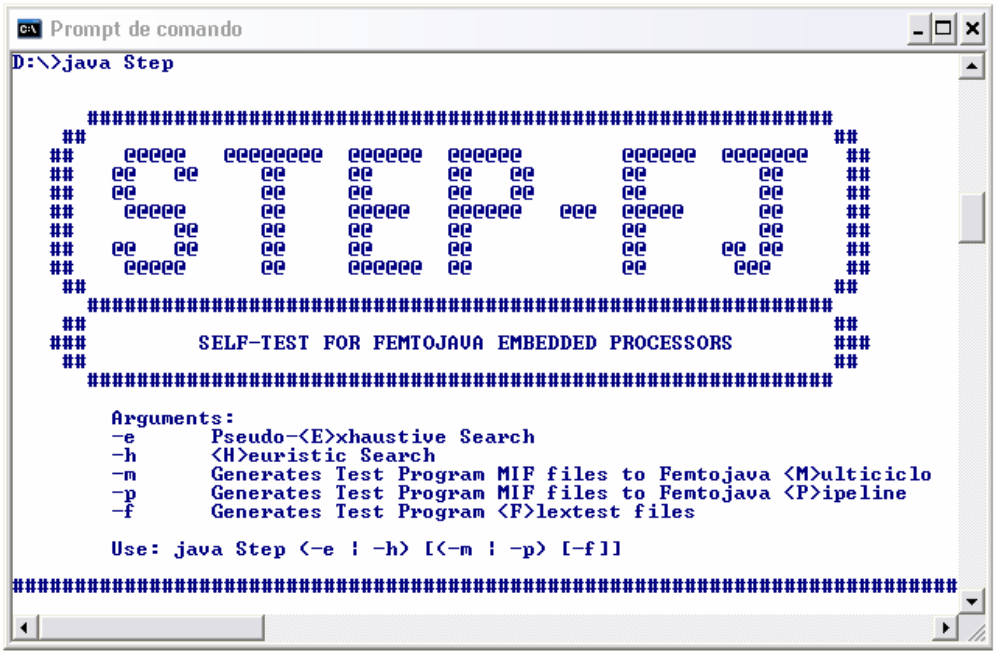

Figura 6.15: Menu principal da ferramenta STEP-FJ

A tela de execução do método pseudo-exaustivo no STEP-FJ, para o processador Femtojava Multiciclo, é ilustrada na Figura 6.16. Nesse exemplo é realizado o experimento *1*, cujos resultados foram apresentados na Tabela 6.4. Na figura, o *STEP 1* mostra o número inicial de soluções do espaço de busca. Os *STEPs* de *2* a *5* mostram o número de programas de auto-teste candidatos aprovados em cada passo e o percentual de redução do espaço de busca em relação ao passo exatamente anterior. Nesse exemplo, os *STEPs 5* e *6* são saltados porque não há restrições com relação ao período de teste e ao consumo de energia do sistema. O percentual total de redução do espaço de busca é mostrado logo abaixo. No *STEP 6* são apresentadas as rotinas selecionadas para cada componente alvo, enquanto o *STEP 7* informa os custos totais do sistema para o programa de auto-teste selecionado.

Figura 6.16: Execução do método pseudo-exaustivo no STEP-FJ

Na Figura 6.17, é ilustrada a tela de execução do método heurístico no STEP-FJ, também para o processador Femtojava Multiciclo. Nesse exemplo é realizado o experimento *9*, cujos resultados foram apresentados na Tabela 6.5. No método heurístico, cada *STEP* do processo mostra o programa de auto-teste pré-selecionado (corrente). O número de *STEPs* depende do número de iterações necessárias para a escolha de uma solução que atenda todos os requisitos do sistema. O penúltimo *STEP* apresenta a solução final, enquanto o último informa os custos totais do sistema para o

programa de auto-teste selecionado. Por fim, é mostrado o número de ajustes de pesos realizados durante o processo de seleção.

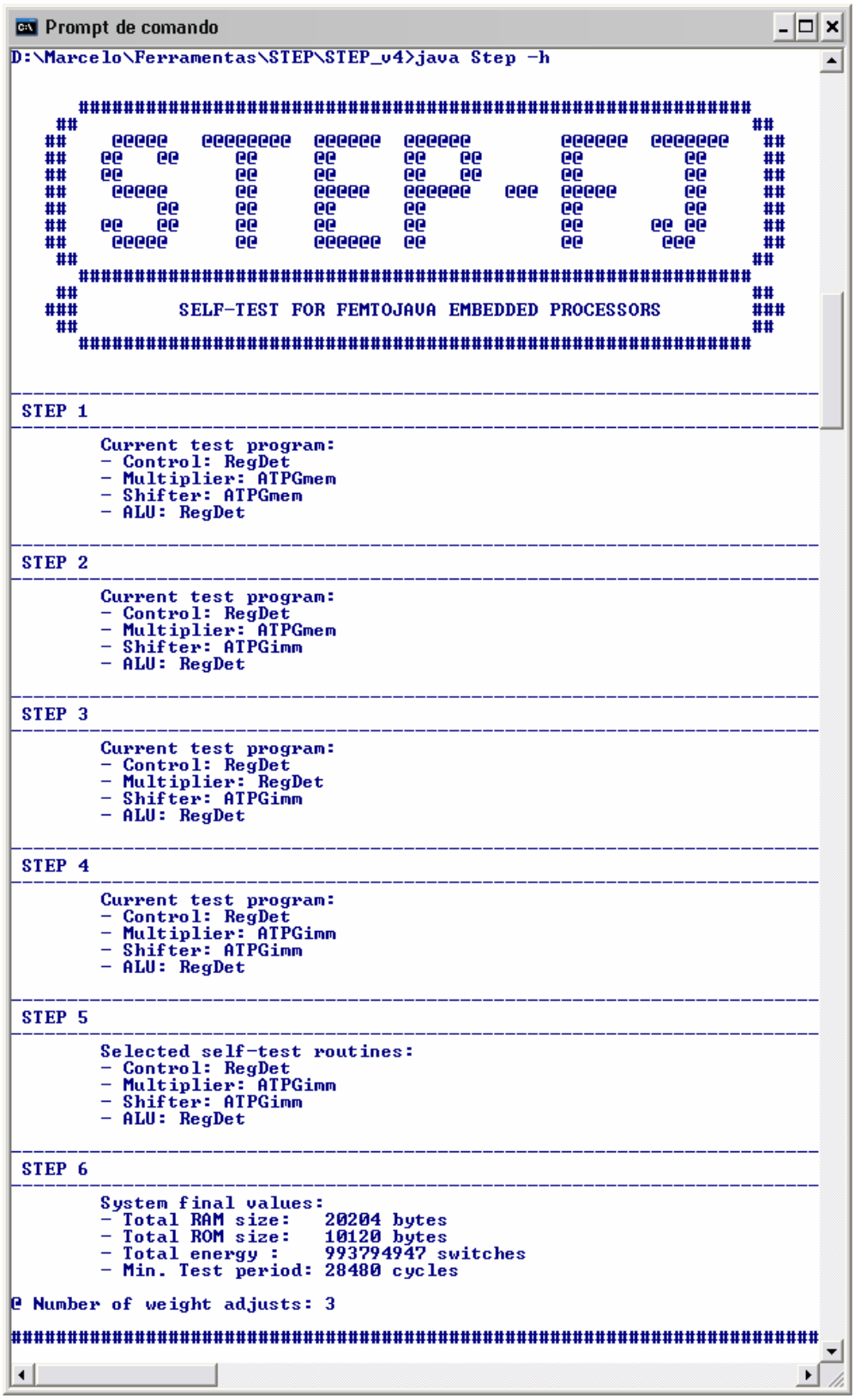

Figura 6.17: Execução do método heurístico no STEP-FJ

6.6 Resultados

A partir dos arquivos gerados pela ferramenta STEP-FJ, foram verificadas as coberturas de falhas alcançadas pelos programas de auto-teste selecionados em alguns dos experimentos descritos na Seção 6.4. Os valores de cobertura de falhas, para o modelo de falhas *stuck-at*, foram obtidos com a ferramenta Flextest® (Mentor Graphics).

A Tabela 6.6 apresenta os resultados, em termos de cobertura de falhas, para o processador Femtojava Multiciclo. Seis programas de auto-teste, escolhidos ao acaso dentre aqueles selecionados pela ferramenta STEP-FJ, foram verificados. A coluna mais à esquerda mostra o número do experimento para o qual foi selecionado o programa de auto-teste indicado nas três colunas seguintes. A sigla *P.E.* refere-se ao resultado da seleção com o método pseudo-exaustivo (Tabela 6.4), enquanto a indicação *H.* refere-se ao programa de auto-teste selecionado, para o experimento em questão, com o método heurístico (Tabela 6.5). Um sétimo programa de auto-teste foi adicionado na penúltima linha da tabela, o qual não é resultado de nenhum experimento. Ele foi incluído por ser composto por três abordagens diferentes de rotinas de teste.

As próximas três colunas da Tabela 6.6 contêm os valores de cobertura de falhas obtidos, com cada programa de auto-teste, para o bloco de controle, o bloco de dados e os árbitros de barramentos do Femtojava Multiciclo, respectivamente. A coluna mais à direita da tabela apresenta a cobertura de falhas total do processador alcançada em cada experimento. A última linha da tabela mostra o percentual das falhas modeladas que cada bloco do processador contém.

Tabela 6.6: Coberturas de falhas para o processador Femtojava Multiciclo

Programa de Auto-Teste				**Cobertura de Falhas (%)**			
Experim.	***Multiplic.***	***Deslocador***	***ULA***	***Bloco Controle***	***Bloco Dados***	***Árbitros Barram.***	***Total***
P.E. 4	Atpg Mem.	Atpg Imed.	Atpg Mem.	94,30	93,76	67,10	92,24
P.E. 5	Atpg Mem.	Det. Reg.	Det. Reg.	94,45	91,52	77,00	91,43
P.E. 6	Det. Reg.	Atpg Imed.	Det. Reg.	94,32	94,32	64,24	92,43
P.E. 7	Det. Reg.	Det. Reg.	Det. Reg.	94,37	94,42	64,40	92,52
P.E. 9	Atpg Imed.	Atpg Imed.	Det. Reg.	94,32	94,57	64,16	92,59
H. 3	Atpg Mem.	Atpg Mem.	Det. Reg.	94,30	93,90	63,91	92,13
---	Atpg Imed.	Atpg Mem.	Det. Reg.	94,35	94,44	64,39	92,52
Percentual de Falhas do Processador				28,02	65,70	6,28	100,00

Verifica-se, na Tabela 6.6, que todos os programas de auto-teste simulados fornecem valores de cobertura de falhas muito próximos. Isso ocorre porque, conforme dito na Seção 6.2, as rotinas de teste foram desenvolvidas de modo que a cobertura de falhas alcançada para cada componente seja a mesma com qualquer uma das abordagens implementadas. Com isso, é possível concluir que qualquer outro programa de auto-teste irá fornecer uma cobertura de falhas similar para o processador Femtojava Multiciclo.

Também de acordo com essa tabela, a cobertura de falhas do processador fornecida pelos programas de auto-teste gira em torno de 92%. Embora seja um valor bastante alto, ele pode ser insuficiente para alguns sistemas, visto que aplicações comerciais requerem, em geral, uma cobertura mínima de 95% das falhas modeladas. Sabendo-se

que o maior número de falhas modeladas encontra-se no bloco de dados (65,7%), o aprimoramento das rotinas para esse bloco parece ser a estratégia mais adequada. Dentro do bloco de dados, o *pool* de registradores (PC, MAR, IMM, SP, FRM, VAR e IR, na Figura 6.1) é o componente que apresenta o maior número de falhas não cobertas. Porém, se 100% das falhas desse componente fosse coberta, o aumento na cobertura de falhas total do processador seria, no máximo, de 1,15 ponto percentual. Além disso, o desenvolvimento de uma rotina para o *pool* de registradores não é uma tarefa trivial, pois a maioria desses registradores armazena endereços de memória. Assim, o aumento nos custos do programa de auto-teste não seria compensado pelo pequeno acréscimo na cobertura de falhas final do processador.

Dessa forma, restam otimizações no teste do bloco de controle e dos árbitros de barramentos para incrementar a cobertura de falhas total do Femtojava Multiciclo. A melhoria da rotina de teste para a unidade do controle desse processador pode fornecer um incremento de até 1,6 ponto percentual na cobertura de falhas total. Por outro lado, o teste dos árbitros de barramentos pode resultar em um aumento de até 2,25 pontos percentuais nesse valor. Assim, com otimizações nos testes desses dois blocos seria possível alcançar uma cobertura de falhas final maior do que 95%. Entretanto, deve-se ter em mente que, tanto a melhoria da rotina de teste para a unidade de controle quanto o desenvolvimento de uma rotina específica para os árbitros de barramento são tarefas bastante difíceis. O aumento no custo de desenvolvimento de teste e nos custos do programa de auto-teste final são justificáveis apenas se tal valor para cobertura de falhas é realmente indispensável.

Para completar o estudo de caso aqui apresentado, é preciso analisar a cobertura de falhas, fornecida pelos programas de auto-teste, para o processador Femtojava Pipeline. Sabendo-se que todos os programas de auto-teste fornecem cobertura de falhas muito parecidas para o processador Femtojava Multiciclo, considerou-se que o mesmo ocorre para a versão Pipeline desse processador. A necessidade de tal consideração pode ser justificada pelo longo tempo de simulação dos programas de auto-teste para o processador Femtojava Pipeline, na ferramenta Flextest®. Assim, para essa versão do processador, apenas um programa de auto-teste foi simulado, cujos resultados representam todos os demais programas.

O programa de auto-teste escolhido para a análise de cobertura de falhas no processador Femtojava Pipeline é aquele composto por três abordagens diferentes de rotinas de teste. Tal programa de auto-teste utiliza a rotina *ATPG Imediato* para o multiplicador, a rotina *ATPG Memória* para o deslocador e a rotina *Determinística Regular* para a ULA. Além dessas, ele inclui as rotinas únicas para o banco de registradores e para a unidade de controle. Os valores de cobertura de falhas obtidos pela execução desse programa de auto-teste no Femtojava Pipeline são apresentados no gráfico da Figura 6.18.

A faixa externa do gráfico mostra o percentual das falhas modeladas (*stuck-at*) contido em cada um dos cinco blocos do processador, enquanto a faixa interna representa a cobertura de falhas alcançada pelo programa de auto-teste em cada bloco. Ao contrário do processador Multiciclo (que possui três blocos principais), a versão Pipeline é dividida em cinco blocos, os quais representam o estágio de busca de instruções, o estágio de decodificação, o estágio de busca de operandos, o estágio de execução e o *pool* de registradores. O *pool* contém os registradores SP, VARS, FRM (da Figura 6.2), PC (da Figura 6.3) e mais dois registradores auxiliares de endereço. O

estágio de escrita de resultados não está representado porque a escrita é feita no banco de registradores, o qual pertence ao bloco de busca de operandos.

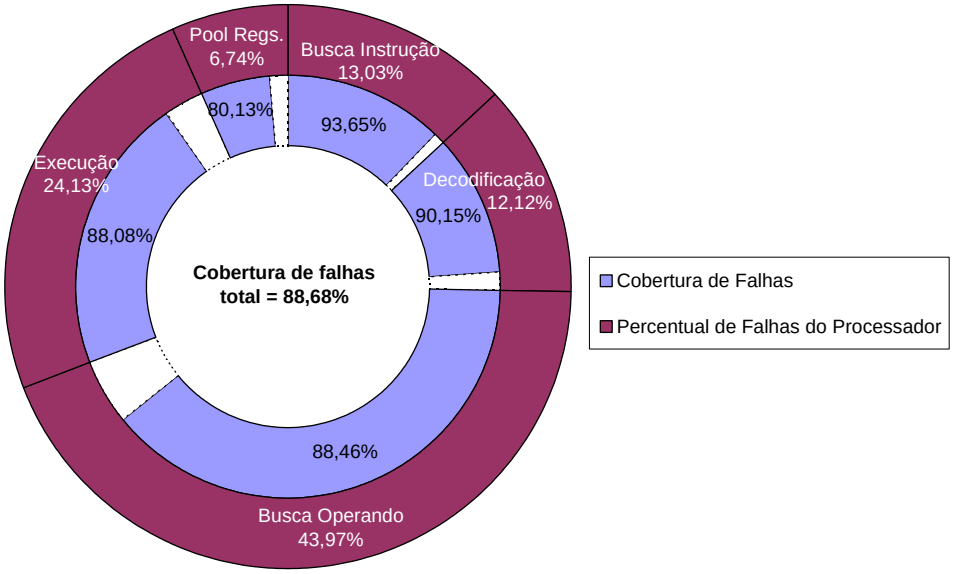

Figura 6.18: Cobertura de falhas para o processador Femtojava Pipeline

Como mostra o valor no centro do gráfico, a cobertura de falhas total obtida para o processador Femtojava Pipeline foi de 88,68%. Assim como na versão Multiciclo, a cobertura de falhas é alta, mas pode ser insuficiente para dada aplicação alvo. Sendo necessário aumentar esse valor, a melhoria do teste para o *pool* de registradores e para o estágio de busca de instruções seria ineficaz. Isso porque o percentual de falhas modeladas no *pool* de registradores é muito pequeno, resultando em um aumento de, no máximo, 1,34 ponto percentual na cobertura de falhas total do processador. Além disso, uma rotina para o teste desses registradores requer operações não triviais com endereços de memória e ponteiros para registradores, tornando o custo de tal rotina inadequado para o programa de auto-teste. Quanto ao bloco de busca de instruções, o aumento na cobertura de falhas total também é irrelevante, chegando ao máximo de 0,83 ponto percentual.

Assim sendo, os blocos que causam maior e/ou melhor impacto na cobertura de falhas total do processador são os blocos de decodificação, busca de operando e execução. O estágio de execução contém a unidade de controle. Portanto, a otimização da rotina para o teste desse componente poderia resultar em um acréscimo de até 1,2 ponto percentual na cobertura final, sem grande aumento de custo. Quanto ao estágio de busca de operandos, o maior percentual de falhas modeladas se encontra no banco de registradores. A melhoria da rotina de teste para esse componentes poderia fornecer uma aumento de até 5 pontos percentuais na cobertura de falhas do processador. Porém, é importante ressaltar que uma melhoria dessa magnitude é uma tarefa bastante complexa, devido à estrutura de pilha implementada no banco de registradores.

Por fim, o estágio de execução também é uma boa fonte para o aumento da cobertura de falhas total. Embora as unidades funcionais tenham sido suficientemente testadas, nenhuma rotina foi desenvolvida para o teste das unidades Branch e LD/ST (Figura 6.3). É possível que uma rotina de teste simples para esses componentes incremente em até 2 pontos percentuais a cobertura de falhas do processador. Tal rotina

deve ter diversas instruções de desvio (condicionais e incondicionais) e também instruções de transferência de dados entre pilha, memória e repositório de variáveis locais. Aqui vale a mesma observação feita para o Femtojava Multiciclo: as modificações sugeridas para a biblioteca de auto-teste implicam o aumento dos custos do programa de auto-teste, sendo justificáveis apenas quando a cobertura de falhas obtida originalmente é insuficiente para a aplicação alvo.

Para efeito de verificação da cobertura de falhas alcançada, os resultados obtidos com a metodologia proposta foram comparados com resultados da aplicação de SBST estrutural para outros processadores. Embora a comparação entre arquiteturas diferentes não seja completamente justa, ela é válida no sentido em que fornece uma idéia da máxima cobertura de falhas alcançada com essa técnica. A abordagem usada para comparação utiliza as regras de aplicabilidade das rotinas de teste propostas por Paschalis e Gizopoulos (PASCHALIS; GIZOPOULOS, 2005), sendo o programa de auto-teste composto unicamente por rotinas que geram e aplicam padrões de teste determinísticos regulares.

A cobertura de falhas obtida para o processador Femtojava Multiciclo foi comparada com a cobertura de falhas alcançada para um processador RISC de 32 bits, o Plasma/MIPS (OPEN CORES, 2006a). Esse processador possui um pipeline reduzido de três estágios e um mecanismo simples para suporte a interrupções. Porém, ele não implementa qualquer mecanismo para detecção de dependências de dados nem mecanismo de *forwarding*. No trabalho que gerou os resultados utilizados nessa comparação (KRANITIS et al., 2005), um multiplicador paralelo foi adicionado ao Plasma/MIPS visando aumentar a correspondência desse processador com processadores RISC de alto desempenho. Nesse trabalho, o programa de auto-teste desenvolvido contém rotinas para o teste dos D-VCs (banco de registradores, multiplicador paralelo, divisor serial, ULA e deslocador), do controlador de memória e da unidade de controle.

Nos experimentos de Kranitis et al., três sínteses foram realizadas para o processador Plasma/MIPS, para duas tecnologias e com diferentes parâmetros. Os resultados apresentados na Tabela 6.7 são da síntese para tecnologia 0,35µm, cujos parâmetros mais se aproximam daqueles usados na síntese do processador Femtojava Multiciclo (também sintetizado para tecnologia 0,35µm). Assim como no presente trabalho, foram utilizadas as ferramentas Leonardo Spectrum®, para síntese lógica, e Flextest®, para simulação de falhas, ambas da Mentor Graphics. A cobertura de falhas para o Femtojava Multiciclo é o maior valor obtido dentre os experimentos realizados neste estudo de caso (experimento P.E. 9).

Tabela 6.7: Resultados comparativos para o Femtojava Multiciclo

	Femtojava Multiciclo ***(metodologia STEP)***	**Plasma/MIPS** ***(SBST Det. Reg.)***
Freqüência de Operação	74,3 MHz	74 MHz
Número de Portas Lógicas	5.667	30.896
Cobertura de falhas	92,59%	94,50%

Pode-se observar, pelo número de portas lógicas de ambos os processadores da Tabela 6.7, que o Femtojava Multiciclo tem uma área bastante reduzida em relação ao processador Plasma/MIPS utilizado nos experimentos. Isso de deve, em primeiro lugar, à organização simples do primeiro, visto que ele foi desenvolvido visando aplicações em FPGAs e, portanto, otimizado para área. Além disso, essa versão do processador Femtojava tem largura de dados de 16 bits, enquanto o Plasma/MIPS é um processador de 32 bits. Por fim, deve-se lembrar que este último, ao contrário do Femtojava Multiciclo, possui um banco de registradores, o que contribui bastante para o aumento de sua área.

Apesar da grande diferença no número de portas lógicas dos dois processadores, a comparação é válida porque permite verificar a cobertura de falhas obtida pelo teste dos D-VCs e da unidade de controle. O que se observa nos resultados comparados é que a cobertura de falhas obtida em ambos os casos é muito parecida. Entretanto, verifica-se que o resultado para o processador Plasma/MIPS é cerca de 2% melhor. Isso se deve, possivelmente, ao maior número de D-VCs presentes nesse processador e à estrutura mais complexa desses componentes. Esses fatores implicam um programa de auto-teste com maior número de rotinas de teste, e rotinas mais complexas. Desse modo, o programa de teste opera em um espaço de endereçamento maior, possibilitando a obtenção de uma cobertura de falhas mais alta para os componentes de endereço. Além disso, a qualidade do teste da unidade de controle também é melhor.

É possível que as modificações sugeridas nesta seção para a melhoria da qualidade de teste do Femtojava Multiciclo igualem ou ultrapassem a cobertura de falhas alcançada para o processador Plasma/MIPS. Entretanto, é preciso lembrar que, assim como a maior complexidade do processador Plasma/MIPS implica um programa de auto-teste mais elaborado e, portanto, com custos mais altos (em termos de ocupação de memória, tempo de execução e consumo de energia), as modificações sugeridas para o teste do Femtojava Multiciclo também implicam o aumento de custos. Logo, essa é uma decisão de projeto que depende das necessidades da aplicação alvo.

Para as comparações com o processador Femtojava Pipeline, foram utilizados os resultados de um trabalho (HATZIMIHAIL et al., 2005) no qual a abordagem SBST estrutural é aplicada ao processador miniMIPS (OPEN CORES, 2006). O miniMIPS é um processador RISC de 32 bits, com pipeline de cinco estágios, mecanismo para detecção de dependências de dados e *forwarding*. Além disso, ele possui um co-processador para tratamento de interrupções. No trabalho de Hatzimihail et al., duas situações foram analisadas. Na primeira, o programa de auto-teste segue a mesma abordagem dos experimentos de Kranitis et al., com rotinas para o teste apenas dos D-VCs (banco de registradores e ULA) e da unidade de controle. Na segunda, o mesmo programa de auto-teste é incrementado com a adição de rotinas de teste, ou a melhoria das rotinas já existentes, para o aumento da cobertura de falhas dos componentes do pipeline (registradores de dados e endereços), dos mecanismos de detecção de dependência de dados e de *forwarding*, e do co-processador para tratamento de interrupções.

A Tabela 6.8 mostra os resultados comparativos da aplicação da metodologia STEP no processador Femtojava Pipeline com a aplicação de SBST determinístico regular no processador miniMIPS. Novamente a ferramenta Leonardo Spectrum® foi utilizada para síntese lógica, para a tecnologia 0,35µm, e a ferramenta Flextest® foi usada para simulação de falhas.

Tabela 6.8: Resultados comparativos para o Femtojava Pipeline

	Femtojava Pipeline ***(metodologia STEP)***	**miniMIPS** ***(SBST Det. Reg. Original)***	***miniMIPS*** ***(SBST Det. Reg. Incrementado)***
Freqüência de Operação	122,1 MHz	33 MHz	33 MHz
Num. de Portas Lógicas	9.640	32.348	32.348
Num. Falhas Modeladas	45.822	76.405	76.405
Cobertura de Falhas	88,68%	84,72%	95,04%

Novamente observa-se que o processador Femtojava é menor do que o processador miniMIPS utilizado na comparação. Os motivos para isso são os mesmos apresentados na comparação anterior, embora o Femtojava Pipeline também tenha um banco de registradores, mas com apenas 16 registradores. Mesmo com uma grande diferença no número de portas lógicas, a diferença no número de falhas modeladas não é tão grande, tornando a comparação mais realística.

Os resultados apresentados na Tabela 6.8 mostram que a cobertura de falhas obtida com a aplicação da metodologia STEP no processador Femtojava Pipeline é cerca de 4% maior do que o valor obtido com a aplicação de SBST determinístico regular para o processador miniMIPS. Entretanto, com as modificações realizadas para o teste deste último, a cobertura de falhas final ultrapassou em quase 7% o valor obtido neste estudo de caso. Considerando-se que nenhuma alteração foi realizada para o teste da versão Pipeline do Femtojava, a cobertura de falhas obtida para esse processador, 88,68%, é bastante razoável. Com as otimizações sugeridas para a biblioteca de auto-teste do Femtojava é possível atingir um valor bem mais próximo do valor alcançado no experimento de Hatzimihail et al. Porém, assim como no caso do miniMIPS, a melhoria da cobertura de falhas implica o aumento dos custos do programa de auto-teste, o que leva a uma decisão de projeto baseada nas necessidades da aplicação alvo.

Vale ressaltar que tanto nos experimentos de Kranitis et al., com o Plasma/MIPS, quanto nos experimentos e Hatzimihail et al., com o miniMIPS, as restrições e os requisitos do sistema embarcado alvo não são considerados. Assim, dado um conjunto de restrições e requisitos de sistema, o auto-teste desses processadores, quando integrados a um sistema embarcado, e com a cobertura de falhas apresentada, pode não ser factível. Por outro lado, a metodologia STEP avaliada neste estudo caso possibilita a obtenção de uma cobertura de falhas alta, bem próxima dos valores obtidos naqueles experimentos, e ainda garante o atendimento das restrições do sistema alvo e dos requisitos da aplicação de tempo-real, se for o caso.

6.7 Resumo e Conclusões

Este capítulo apresentou um estudo de caso no qual a metodologia STEP, proposta na presente dissertação, foi aplicada aos processadores Femtojava. O objetivo de tal estudo era avaliar a metodologia proposta visando sua validação como um método de projeto e aplicação de auto-teste *on-line* periódico para processadores embarcados, através de resultados obtidos por meio de ferramentas de simulação.

Para tanto, este capítulo descreveu, em primeiro lugar, os processadores que compõem a família Femtojava. Foi visto que os processadores Femtojava são baseados em pilha, executam nativamente *bytecodes* Java e foram desenvolvidos especificamente para aplicações embarcadas. Também foi descrito o *framework* que permite aos processadores Femtojava executar aplicações de tempo-real. A seguir, foram apresentadas as diferentes versões dos processadores Femtojava, incluindo a versão Multiciclo, Low-Power (Pipeline), VLIW e DSP. O estudo de caso teve como foco as versões Multiciclo e Pipeline, ambas de 16 bits.

Na seqüência, foi apresentada a biblioteca de auto-teste desenvolvida para os processadores Femtojava. Tal biblioteca contém rotinas para o teste dos D-VCs e das unidades funcionais dos Femtojava Multiciclo e Pipeline. A versão Multiciclo do processador tem três D-VCs: um multiplicador, um deslocador e uma ULA. Já a versão Pipeline, além dos mesmos três componentes visíveis de dados, tem também um banco com 16 registradores. Para o multiplicador de ambas as versões foram desenvolvidas quatro rotinas de teste, enquanto para o deslocador e para a ULA foram desenvolvidas três rotinas. Para as unidades funcionais dos processadores e para o banco de registradores da versão Pipeline, apenas uma rotina para cada componente foi desenvolvida.

Além da descrição das rotinas implementadas para a biblioteca de auto-teste dos processadores Femtojava Multiciclo e Pipeline, também foram fornecidas indicações e cuidados necessários para a inclusão de outras rotinas ou modificação de rotinas já existentes de modo a permitir o teste das versões VLIW e DSP do processador Femtojava. Por fim, nessa seção também foram apresentados os custos de cada uma das rotinas de teste, tanto para o Femtojava Multiciclo, quanto para o Femtojava Pipeline. Tais custos são relativos ao número de padrões/respostas de teste, ao tamanho das memórias de dados e programa, ao tempo de execução e ao consumo de energia das rotinas.

O analisador de respostas, cujo desenvolvimento compreende a segunda etapa da metodologia STEP, não foi implementado no presente estudo de caso. Os motivos que levaram a tal decisão foram explicados neste capítulo. Para dar continuidade à aplicação da metodologia, considerou-se que o tempo de execução do analisador de respostas equivale, em número de ciclos de memória, a duas vezes o número de respostas de teste do programa de auto-teste. De posse dessa informação, o estudo de caso teve seguimento com a descrição da terceira e última etapa do método STEP.

No estudo de caso apresentado neste capítulo, foram desenvolvidos os dois métodos propostos para a seleção do programa de auto-teste. Os detalhes de implementação de ambos os métodos, pseudo-exaustivo e heurístico, foram descritos incluindo as equações envolvidas em cada passo. Em seguida, diversos experimentos foram realizados com ambos os métodos utilizando um *benchmark* com tarefas de tempo-real, e tendo como processador alvo o Femtojava Multiciclo. Os resultados obtidos com tais experimentos permitiram a validação de ambos os métodos e mostraram a sua importância no desenvolvimento de auto-teste *on-line* para processadores embarcados, possivelmente em sistemas de tempo-real.

A seguir, foi apresentada a ferramenta STEP-FJ, a qual permite a execução de ambos os métodos de seleção do programa de auto-teste para qualquer plataforma embarcada. Tal ferramenta, implementada em linguagem Java, quando aplicada aos processadores Femtojava, possibilita não apenas a seleção do programa de auto-teste,

mas também a geração dos arquivos necessários para a simulação RTL do programa selecionado, e dos arquivos para a verificação da cobertura de falhas obtida, através da simulação de falhas na ferramenta Flextest® (Mentor Graphics). Figuras com exemplos da execução do STEP-FJ foram apresentadas, ilustrando ambos os métodos de seleção.

Finalmente, os resultados da aplicação da metodologia STEP, obtidos pelo uso da ferramenta STEP-FJ, foram apresentados para ambos os processadores estudados: Femtojava Multiciclo e Femtojava Pipeline. Os valores de cobertura de falhas, para o modelo *stuck-at*, fornecidos por alguns programas de auto-teste foram mostrados e analisados. Para a versão Multiciclo, a cobertura de falhas alcançada gira em torno de 92%, enquanto para a versão Pipeline, esse valor se aproxima de 89%. Além disso, foram propostas algumas sugestões para o aumento da cobertura de falhas total de ambos os processadores. A comparação desses resultados com resultados da aplicação de SBST estrutural para outros processadores mostrou que a metodologia proposta neste trabalho se aplica muito bem aos processadores Femtojava, e que algumas otimizações na biblioteca de auto-teste podem melhorar ainda mais a qualidade do teste.

Com base nos resultados fornecidos ao longo de todo este capítulo, algumas conclusões podem ser extraídas. No que diz respeito ao desenvolvimento da biblioteca de auto-teste, nota-se que o uso de uma linguagem de alto nível requer uma série de cuidados devidos ao processo de compilação, os quais são desnecessários quando utilizada a linguagem de máquina. Além disso, a baixa cobertura de falhas alcançada para o banco de registradores da versão Pipeline do processador Femtojava mostra o quão complexo é o desenvolvimento de uma rotina de teste para esse componente, uma vez que seu acesso é feito através de uma estrutura de pilha. Também é preciso ressaltar que, para todas as rotinas, a cobertura de falhas não é o único fator relevante, pois além de uma alta cobertura de falhas para o componente alvo, deve-se almejar os menores custos possíveis.

Com relação às diferentes abordagens para a implementação de rotinas de teste, pôde-se verificar que não existe uma abordagem ótima, que apresente menor custo para todos os fatores considerados. Logo, o programa de auto-teste mais adequado para dado processador embarcado deve ser uma combinação de diferentes abordagens de modo a compor uma solução de baixo custo, e que atenda a todas as restrições do sistema e a todos os requisitos da aplicação de tempo-real, se for o caso. Daí percebe-se a importância de um método eficiente para seleção do programa de auto-teste, seja para um espaço de busca pequeno, ou para um espaço de busca muito grande. Além disso, os custos da rotina *LFSR* para o multiplicador mostraram que a abordagem com padrões pseudo-aleatórios não é adequada para o teste *on-line* periódico de processadores embarcados, visto que seus custos são ordens de grandeza maiores do que os custos das demais abordagens implementadas.

Quanto ao analisador de respostas, conclui-se que ele deve ser otimizado em termos de área e de dissipação de potência, uma vez que ele constitui um módulo extra no sistema. Porém, também é importante que ele seja otimizado em termos de tempo de execução, visto que o atendimento dos requisitos de tempo-real da aplicação dependem dos tempos de execução do programa de auto-teste e do analisador de respostas.

No que tange os métodos de seleção do programa de auto-teste, pôde-se verificar que alguns detalhes de sua implementação variam de acordo com o sistema alvo. Por exemplo, o nível máximo permitido de utilização do processador depende da política de escalonamento utilizada pelo sistema. E a verificação do atendimento das restrições do

sistema embarcado depende de como tais restrições são especificadas. Além disso, os resultados dos experimentos realizados demonstraram a importância de métodos efetivos para a seleção de rotinas de teste. Comparações dos métodos propostos com as regras definidas por Paschalis e Gizopoulos mostraram que tais regras nem sempre se aplicam a determinadas aplicações alvo. Já os métodos pseudo-exaustivo e heurístico da metodologia STEP garantem a seleção de um programa de auto-teste de baixo custo (de acordo com a função de custo fornecida) e que atenda todas as restrições do sistema e todos os requisitos da aplicação de tempo-real. Outro fator que atesta a importância dos métodos de seleção é a constatação de que uma pequena alteração na especificação do sistema alvo (pesos da função de custo, algoritmo de escalonamento ou restrições do sistema) pode causar a modificação do programa de auto-teste mais adequado, alterando significativamente os custos finais do sistema.

A ferramenta STEP-FJ, desenvolvida para este estudo de caso, possibilitou a automatização do projeto e aplicação de auto-teste *on-line* para os processadores da família Femtojava. Os resultados obtidos através do uso de tal ferramenta mostraram que os programas de auto-teste selecionados fornecem uma alta cobertura de falhas, tanto para a versão Multiciclo quanto para a versão Pipeline desses processadores. Entretanto, a cobertura de falhas obtida para o modelo *stuck-at* pode ser insuficiente para dada aplicação alvo, visto que é inferior a 95%. Somando-se a esses dados as comparações realizadas com a aplicação de SBST estrutural para outras arquiteturas, conclui-se que algumas alterações na biblioteca de auto-teste dos processadores Femtojava podem ser necessárias para que a qualidade do auto-teste projetado seja comercialmente satisfatória. Porém, essa pequena deficiência na cobertura de falhas não invalida a metodologia STEP, a qual se mostrou suficientemente completa para os objetivos aos quais se propõe.

7 CONCLUSÕES GERAIS E TRABALHOS FUTUROS

Nesta dissertação foi proposta uma metodologia, denominada STEP, para o planejamento, a geração e a seleção de auto-teste *on-line* para processadores embarcados. Antes da apresentação da metodologia proposta, foi realizado um estudo sobre auto-teste de circuitos digitais e sobre teste *on-line* de processadores. Por fim, um estudo de caso onde a metodologia STEP foi aplicada aos processadores da família Femtojava comprovou a viabilidade da mesma.

O estudo apresentado no início do presente trabalho incluiu a descrição de algumas técnicas de auto-teste e a sua classificação de acordo com seus objetivos, suas características e suas formas de aplicação. Em seguida, os métodos de teste *on-line* de processadores também foram classificados em categorias, de modo a facilitar a escolha da técnica mais apropriada para cada aplicação alvo. Tal estudo, juntamente com a caracterização dos requisitos para a aplicação de teste *on-line* em processadores integrados a sistemas embarcados e/ou de tempo-real, justificou a escolha do método de auto-teste baseado em software como base para a metodologia proposta. Assim, tendo-se definido o SBST como foco da metodologia STEP, um estudo mais detalhado dessa técnica foi realizado e também apresentado nesta dissertação.

Por ter como base a técnica SBST, o método de geração do programa de auto-teste proposto neste trabalho se utilizou da metodologia proposta por Paschalis e Gizopoulos (PASCHALIS; GIZOPOULOS, 2005) para o desenvolvimento de rotinas de teste. Entretanto, o método STEP é específico para o auto-teste de processadores embarcados e por isso prevê, além do desenvolvimento de uma biblioteca de auto-teste, a implementação de um módulo analisador de respostas de teste e a seleção do programa de auto-teste com base em uma função de custo e nos requisitos e restrições do sistema alvo. Assim, propôs-se uma metodologia mais completa para o auto-teste de processadores embarcados que garante a qualidade do teste e o atendimento das especificações do sistema alvo.

A partir dos resultados apresentados no estudo de caso com os processadores da família Femtojava, é possível concluir que a metodologia STEP atinge os objetivos aos quais se propõe. Ela garante o auto-teste *on-line* periódico do processador alvo, com o menor período de teste possível e com uma boa cobertura de falhas, respeitando todas as restrições do sistema embarcado e todos os requisitos da aplicação de tempo-real. Nesse estudo, demonstrou-se a importância de cada etapa da metodologia e a eficácia da mesma para essa plataforma alvo. Futuramente, pretende-se verificar se a mesma eficácia é obtida para outras plataformas. Embora a metodologia STEP tenha sido pensada de forma genérica o suficiente para ser aplicada a quaisquer arquiteturas, até esse ponto não há garantias de que tal objetivo tenha sido alcançado, visto que apenas um estudo de caso foi realizado.

Antes, porém, da validação do método STEP para outras plataformas, algumas tarefas complementares são pretendidas ainda para a família Femtojava. Em primeiro lugar, faz-se necessário implementar e validar o analisador de respostas proposto. Depois, pretende-se complementar a biblioteca de auto-teste já desenvolvida com as modificações propostas para o aumento da cobertura de falhas dos processadores Femtojava Multiciclo e Pipeline. Também se pretende incluir as rotinas e as modificações propostas para o teste das versões VLIW e DSP dessa família e, por fim, desenvolver, para cada rotina da biblioteca, uma versão com código de compactação. Com isso, espera-se concluir a automatização do projeto de auto-teste *on-line* para todos os processadores Femtojava e ainda permitir uma maior exploração do espaço de teste.

Após a validação da metodologia STEP para outras plataformas, é interessante que se proponha a sua extensão de modo a permitir o teste de outros módulos de um SoC, como memórias, módulos de entrada e saída, DSPs e outros componentes dedicados. Tal extensão pode, ainda, incluir o teste das estruturas de comunicação de sistemas do tipo *Network-on-Chip*. Esse tipo de sistema ocupa cada vez mais espaço nas aplicações embarcadas atuais sendo, portanto, um nicho que requer fortemente a utilização de auto-teste *on-line*.

REFERÊNCIAS

ABRAMOVICI, M.; BREUER, M. A.; FRIEDMAN, A. D. **Digital Systems Testing and Testable Design**. Piscataway, USA: IEEE Press, 1994.

ADHAM, S. M. I.; GUPTA, S. DP-BIST: A Built-In Self-Test for DSP Datapaths – a Low Overhead and High Fault Coverage Technique. In: ASIAN TEST SYMPOSIUM, ATS, 5., 1996. **Proceedings...** [S.l.:s.n.], 1996. p.205-212.

AGARWAL, A.; BLAAUW, D.; ZOLOTOV, V. Statistical Clock Skew Analysis Considering Intra-Die Process Variations. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 2003, San Jose, USA. **Proceedings…** [S.l.: s.n.], 2003. p.914-921.

BATCHER, K.; PAPACHRISTOU, C. Instruction Randomization Self Test for Processor Cores. In: IEEE VLSI TEST SYMPOSIUM, VTS, 1999. **Proceedings…** [S.l.: s.n.], 1999. p.34-40.

BECK, A. C. S. **Uso da Técnica VLIW para Aumento de Performance e Redução do Consumo de Potência em Sistemas Embarcados Baseados em Java**. 2004. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

BECK, A. C. S.; CARRO, L. Low Power Java Processor for Embedded Applications. In: IFIP INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI-SoC, 12., 2003, Darmstadt, Germany. **Proceedings…** [S.l.: s.n.], 2003.

BECK, A. C. S.; CARRO, L. A VLIW Low Power Java Processors for Embedded Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 17., 2004. **Proceedings…** [S.l.: s.n.], 2004. p.157-162.

BECK, A. C. S.; HENTSCHKE, R.; MATTOS, J. C. B.; REIS, R.; CARRO, L. Fast and Efficient Test Generation for Embedded Stack Processors. In: LATIN AMERICAN TEST WORKSHOP, LATW, 6., 2005. Salvador, Brazil. **Proceedings…** [S.l.:s.n.], 2005. p.331-336.

BECK, A. C. S.; MATTOS, J. C. B.; WAGNER, F. R.; CARRO, L. CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 16., 2003. **Proceedings…** [S.l.: s.n.], 2003. p.349-1354.

BECKER, L. B.; WEHRMEISTER, M.; CARRO, L.; WAGNER, F. R.; PEREIRA, C. E. Assessing the Impact of Traditional Real-Time Scheduling Algorithms on Top of Embedded Applications. In: IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, WRTP, 28., 2004. **Proceedings…** [S.l.:s.n.], 2004.

BUSHNELL, M. L.; AGRAWAL, V. D. **Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits**. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2000.

BUTTAZZO, G. C. **Hard Real-Time Computing Systems**: Predictable Scheduling Algorithms and Applications. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2002.

CARRO, L.; WAGNER, F. Sistemas Computacionais Embarcados. In: JORNADAS DE ATUALIZAÇÃO EM INFORMÁTICA, 22., 2003, Campinas. **Livro Texto**. Campinas: SBC, 2003. p.45-94.

CHEN, L.; DEY, S. Software-Based Self-Testing Methodology for Processor Cores. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.20, n.3, p.369-380, Mar. 2001.

CLARK, J. A.; PRADHAN, D. K. Fault Injection: a Method for Validating Computer-System Dependability. **Computer**, [S.l.], v.28, n.6, p.47–56, June 1995.

CORNO, F. et al. On the Test of Microprocessor IP Cores. In: IEEE DESIGN, AUTOMATION AND TESTE IN EUROPE, DATE, 2001. **Proceedings…** [S.l.:s.n.], 2001. p.209-213.

COURTOIS, B. A Methodology for On-Line Testing of Microprocessors. In: FAULT-TOLERANT COMPUTING SYMPOSIUM, FTCS, 1981. **Proceedings…** [S.l.:s.n.], 1981. p.272-274.

COURTOIS, B. Failure Mechanisms, Fault Hypotheses and Analytical Testing of LSI-NMOS (HMOS) Circuits. In: VLSI CONFERENCE, 1981. **Proceedings…** [S.l.: s.n.], 1981. p.341-350.

FISHER, J. A. The VLIW Machine: a multiprocessor for compiling scientific code. **IEEE Computer**, [S.l.], v.17, n.7, p.45-53, July 1984.

GALAY, J. A.; CROUZET, Y.; VERNIAULT, M. Physical Versus Logical Fault Models MOS-LSI Circuits: Impact of their Testability. **IEEE Transactions on Computers**, [S.l.], v.29, n.6, p.527-531, 1980.

GIZOPOULOS, D.; PASCHALIS, A.; ZORIAN, Y. **Embedded Processor-Based Self-Test**. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2004.

HATZIMIHAIL, M. et al. Software-Based Self-Test for Pipelined Processors: A Case Study. In: IEEE INTERNATIONAL SYMPOSIUM ON DEFECT AND FAULT TOLERANCE IN VLSI SYSTEMS, DFT, 20., 2005. **Proceedings…** [S.l.:s.n.], 2005. p.535-543.

HENSHAW, B. An MC68020 users test program. In: INTERNATIONAL TEST CONFERENCE, ITC, 1986. **Proceedings…** [S.l.:s.n.], 1986. p.386-393.

HUGHES, H. L.; BENEDETTO, J. M. Radiation Effects and Hardening of MOS Technology: Devices and Circuits. **IEEE Transactions on Nuclear Science**, [S.l.], v.50, n.3, p.500-521, June 2003.

ITO, S. A.; CARRO, L.; JACOBI, R. P. Making Java Work for Microcontroller Applications. **IEEE Design & Test of Computers**, [S.l.], v.18, n.5, p.100-110, Sep./Oct. 2001.

KARPOVSKY, M. G.; VAN METER, R. G. An Approach to the Testing of Microprocessors. In: AMC/IEEE DESIGN AUTOMATION CONFERENCE, DAC, 21., 1984. **Proceedings…** [S.l.:s.n.], 1984. p.196-202.

KEATING, M.; BRICAUD, P. **Reuse Methodology Manual for System-on-Chip Designs**. 3rd ed. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2002.

KIM, H. B.; TAKAHASHI, T.; HA, D. S. Test Session Oriented Built-In Self-Testable Data Path Synthesis. In: INTERNATIONAL TEST CONFERENCE, ITC, 1998, Washington DC, USA. **Proceedings…** [S.l.: s.n.], 1998. p.154-163.

KRANITIS, N. et al. Instruction-Based Self-Testing of Processor Cores. **Journal of Electronic Testing: Theory and Applications**, The Netherlands, v.19, n.2, p.103-112, Apr. 2003.

KRANITIS, N. et al. Application and Analysis of RTL-Level Software-Based Self-Testing for Embedded Processor Cores. In: INTERNATIONAL TEST CONFERENCE, ITC, 2003. **Proceedings…** Piscataway: IEEE, 2003a. p.431-440.

KRANITIS, N. et al. Software-Based Self-Testing of Embedded Processors. **IEEE Transactions on Computers**, [S.l.], v.54, n.4, p.461-475, Apr. 2005.

KRAPF, R.; CARRO, L. Efficient Signal Processing in Embedded Java Systems. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 2003, Bangkok, Thailand. **Proceedings…** [S.l.: s.n.], 2003. v.4, p.IV-61-64.

LAGUNA, G.; TREECE, R. VLSI Modeling and Design for Radiation Environments. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN, ICCD, 1986. **Proceedings…** [S.l.:s.n.], 1986. p.380-384.

LALA, P. K. **Digital Circuit Testing and Testability**. San Diego, USA: Academic Press, 1997.

LEUNG, J. Y. T.; WHITEHEAD, J. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. **Performance Evaluation**, [S.l.], v.2, n.4, p.237-250, 1992.

LIU, C. L.; LAYLAND, J. W. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. **Journal of the ACM**, [S.l.], v.20, n.1, p.46-61, Jan. 1973.

LSE: Embedded Systems Lab. **SEEP**: Plataform-based Electronic Embedded Systems. 2006. Disponível em: <http://www.inf.ufrgs.br/~lse/pag_projeto.php?cod_projeto=1>. Acesso em: jan. 2006.

LUBASZEWSKI, M.; COTA, E. F.; KRUG, M. R. Teste e Projeto Visando o Teste de Circuitos e Sistemas Integrados. In: REIS, R. A. da L. (Ed.). **Concepção de Circuitos**

Integrados. 2.ed. Porto Alegre: Instituto de Informática da UFRGS: Sagra Luzzatto, 2002. p.167-189.

MORAES, M.; COTA, E.; CARRO, L.; WAGNER, F.; LUBASZEWSKI, M. A Constraint-Based Solution for On-Line Testing of Processors Embedded in Real-Time Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 18., 2005, Florianópolis, Brazil. **Proceedings…** Los Alamitos: IEEE, 2005. p.68-73.

MORAES, M.; COTA, E.; CARRO, L.; WAGNER, F.; LUBASZEWSKI, M. An Application-Oriented Method for the Selection of Self-Test Routines in Real-Time Embedded Systems. In: LATIN AMERICAN TEST WORKSHOP, LATW, 6., 2005, Salvador, Brazil. **Proceedings…** [S.l.: s.n.], 2005. p.343-348.

MORAES, M.; COTA, E.; CARRO, L.; LUBASZEWSKI, M. An Efficient Constraint-Based Test Selection Method for Embedded Processor Cores. In: LATIN-AMERICAN WORKSHOP ON DEPENDABLE AUTOMATION SYSTEM, WDAS, 1., 2005, Salvador, Brazil. **Proceedings…** [S.l.: s.n.], 2005.

NICOLAIDIS, M.; ZORIAN, Y. On-Line Testing for VLSI – a Compendium of Approaches. **Journal of Electronic Testing: Theory and Applications**, [S.l.], v.12, n.1-2, p.7-20, 1998.

OH, N.; MCCLUSKEY, E. J. Error Detection by Selective Procedure Call Duplication for Low Energy Consumption. **IEEE Transactions on Reliability, [**S.l.**]**, v.51, n.4, p.392-402, Dec. 2002.

OPEN CORES. **MiniMIPS CPU Core**. 2006. Disponível em: <http://www.opencores.org/projects/minimips>. Acesso em: jan. 2006.

OPEN CORES. **Plasma CPU Core**. 2006a. Disponível em: <http://www.opencores.org/projects/mips>. Acesso em: jan. 2006.

PARVATHALA, P.; MANEPARAMBIL, K.; LINDSAY, W. FRITS – A Microprocessor Functional BIST Method. In: IEEE INTERNATIONAL TEST CONFERENCE, ITC, 2002. **Proceedings…** [S.l.: s.n.], 2002. p.590-598.

PASCHALIS, A.; GIZOPOULOS, D. Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.24, n.1, p.88-99, Jan. 2005.

PASCHALIS, A. et al. Deterministic Software-Based Self-Testing of Embedded Processor Cores. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2001, Munich, Germany. **Proceedings…** Los Alamitos: IEEE Computer Society, 2001. p.92-96.

PFLANZ, M.; VIERHAUS, H. T. Online Check and Recovery Techniques for Dependable Embedded Processors. **IEEE Micro**, [S.l.], v.21, n.5, p.24-40, Sept./Oct. 2001.

PRASAD, V. B. Fault Tolerant Digital Systems. **IEEE Potentials**, [S.l.], v.8, n.1, p.17-21, Feb. 1989.

RADECKA, K.; RAJSKI, J.; TYSZER, J. Arithmetic Built-In Self-Test for DSP Cores. **IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems**, [S.l.], v.16, n.11, p.1358-1369, Nov. 1997.

RAJSUMAN, R. **Digital Hardware Testing:** Transistor-Level Fault Modelling and Testing. [S.l.]: Artech House, 1992.

SHEN, J.; ABRAHAM, J. Native Mode Functional Test Generation for Processors with Applications to Self-Test and Design Validation. In: IEEE INTERNATIONAL TEST CONFERENCE, ITC, 1998, Washington DC, USA. **Proceedings…** [S.l.: s.n.], 1998. p.990-999.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. 6th ed. New York, USA: John Wiley & Sons, 2002.

STANKOVIC, J. A. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. **Computer**, [S.l.], v.21, n.10, p.10-19, Oct. 1988.

TEXAS INSTRUMENTS PRODUCTS. **C62x DSPs**: Digital Signal Processing. 2006. Disponível em:<http://focus.ti.com/paramsearch/docs/parametricsearch.tsp?family=dsp&sectionId=2&tabId=134&familyId=326&paramCriteria=no>. Acesso em: jan. 2006.

TIMOC, C. et al. Adaptative Self-Test for a Microprocessor. In: IEEE INTERNATIONAL TEST CONFERENCE, ITC, 1983. **Proceedings…** [S.l.:s.n.], 1983. p.701-703.

VENNERS, B. **Inside the Java Virtual Machine**. New York, USA: McGraw-Hill, 1998.

WADSACK, R. L. Fault Modelling and Logic Simulation of CMOS and MOS Integrated Circuits. **The Bell System Technical Journal**, New York, 1978.

WEHRMEISTER, M. A.; BECKER, L. B.; PEREIRA, C. E. Optimizing Real-Time Embedded Systems Development Using a RTSJ-based API. In: WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, JTRES, 2., 2004, Larnaca, Cyprus. **Proceedings…** [S.l.]: Springer, 2004. (Lecture Notes in Computer Science, v.3292).

WEIDERMAN, N.; KAMENOFF, N. Hartstone Uniprocessor Benchmark: Definitions and Experiments for Real-Time Systems. **Journal of Real-Time Systems**, [S.l.], v.4, p.353-382, 1992.

WOLF, W. **Computers as Components**. [S.l.]: McGraw-Hill, 2001.

XENOULIS, G. et al. Low-Cost, On-Line Software-Based Self-Testing of Embedded Processor Cores. In: IEEE INTERNATIONAL ON-LINE TESTING SYMPOSIUM, IOLTS, 9., 2003. **Proceedings…** [S.l.: s.n.], 2003. p.149-154.

APÊNDICE A EXECUÇÃO DO ANALISADOR DE RESPOSTAS

O algoritmo abaixo, escrito em uma pseudo-linguagem, descreve as principais operações realizadas pelo analisador de respostas proposto no Capítulo 5.

```
IN wake;
wake_up();
IN clk; ------------------------------------------------------------------
#ROUTINES <= ROM[ROM_COUNTER];
ROM_COUNTER++;
IN clk; ------------------------------------------------------------------
INITIAL_VALUE <= ROM[ROM_COUNTER];
ROM_COUNTER++;
IN clk; ------------------------------------------------------------------
while (#ROUTINES != ROUT_COUNTER) {
  ROUT_COUNTER++;
  IN clk; ----------------------------------------------------------------
  BASE_ADDRESS <= ROM[ROM_COUNTER];
  ROM_COUNTER++;
  RESP_COUNTER <= 0;
  IN clk; ----------------------------------------------------------------
  #RESPONSES <= ROM[ROM_COUNTER];
  ROM_COUNTER++;
  RAM_ADDRESS <= BASE_ADDRESS + RESP_COUNTER;
  OUT ram_rd;
  IN clk; ----------------------------------------------------------------
  while (#RESPONSES != RESP_COUNTER) {
    RESP_COUNTER++;
    IN clk; --------------------------------------------------------------
    wait_read();
    if (ROM[ROM_COUNTER] != RAM[RAM_ADDRESS]) {OUT FAULT};
    ROM_COUNTER++;
    IN clk; --------------------------------------------------------------
    OUT ram_wr;
    RAM[RAM_ADDRESS] <= INITIAL_VALUE;
    wait_write();
    IN clk; --------------------------------------------------------------
    RAM_ADDRESS <= BASE_ADDRESS + RESP_COUNTER;
    OUT ram_rd;
    IN clk; --------------------------------------------------------------
  }
  IN clk; ----------------------------------------------------------------
}
IN clk; ------------------------------------------------------------------
OUT done;
sleep();
```

Nesse algoritmo, as expressões *IN <porta_de_entrada>* e *OUT <porta_de_saída>* indicam a ativação do referido sinal de entrada ou saída, respectivamente (por exemplo, *IN clk* representa a subida do relógio). Assim, todas as operações compreendidas entre duas sentenças *IN clk* consecutivas, ocorrem simultaneamente. O processo *wake_up()* resseta todos os registradores e inicia a máquina de estados da unidade de controle. O processo *sleep()* coloca o analisador em estado de espera até o recebimento de um novo sinal *wake*. Finalmente, os processos *wait_read()* e *wait_write()* são responsáveis pela sincronização entre o analisador de respostas e a memória RAM do processador.

APÊNDICE B ARQUIVOS DE ENTRADA PARA A FERRAMENTA STEP-FJ

Os exemplos de arquivos aqui apresentados são os mesmos utilizados no experimento 1, descrito na Tabela 6.4 e na Tabela 6.5. As unidades nas quais os valores são expressos são as mesmas utilizadas nos experimentos da Seção 6.4.

O arquivo exemplificado abaixo, cujo nome deve ser *routines.prj*, contém informações sobre a biblioteca de auto-teste. Nele são informados o número de componentes alvo do programa de auto-teste, o nome de cada componente, o número de rotinas desenvolvida para cada um deles e seus respectivos nomes. Também são fornecidos os custos de cada rotinas e os custos para a sua união, em termos de tamanho de memória RAM, tamanho de memória ROM, tempo de execução, consumo de energia e número de padrões/respostas de teste.

```
begin components 4
  component Control
    begin routines 1
    routine RegDet
      ram = 48
      rom = 198
      exectime = 1640
      energy = 964242
      patterns = 10
  end;
  component Multiplier
    begin routines 4
    routine ATPGmem
      ram = 192
      rom = 33
      exectime = 5792
      energy = 3338339
      patterns = 29
    routine ATPGimm
      ram = 64
      rom = 490
      exectime = 3826
      energy = 2238657
      patterns = 29
    routine RegDet
      ram = 134
      rom = 54
      exectime = 13272
      energy = 7675773
      patterns = 64
    routine LFSR
      ram = 118
      rom = 118
      exectime = 225784
```

```
      energy = 132824561
      patterns = 56
  end;
  component Shifter
    begin routines 3
    routine ATPGmem
      ram = 102
      rom = 83
      exectime = 3614
      energy = 2089685
      patterns = 14
    routine ATPGimm
      ram = 34
      rom = 214
      exectime = 1822
      energy = 1094418
      patterns = 14
    routine RegDet
      ram = 70
      rom = 94
      exectime = 5550
      energy = 3237166
      patterns = 32
  end;
  component ALU
    begin routines 3
    routine ATPGmem
      ram = 168
      rom = 152
      exectime = 8044
      energy = 4677435
      patterns = 27
    routine ATPGimm
      ram = 60
      rom = 426
      exectime = 3402
      energy = 1977477
      patterns = 27
    routine RegDet
      ram = 54
      rom = 118
      exectime = 3512
      energy = 2086465
      patterns = 24
  end;

  join
    ram = 0
    rom = 15
    exectime = -160
    energy = -100000

end.
```

O próximo arquivo, cujo nome deve ser *application.prj*, contém informações sobre as tarefas da aplicação, sobre custos inerentes da plataforma e sobre as políticas de escalonamento implementadas no sistema alvo. Sobre a aplicação são fornecidos o número de tarefas, seus períodos e seus custos em termos de tamanho de memória RAM, tamanho de memória ROM, tempo de execução e consumo de energia. Em relação à plataforma alvo, é informado o número de palavras reservadas na memória de dados e de programa. Quanto aos algoritmos de escalonamento, são informados os seus

custos em termos de tamanho de memória RAM, tamanho de memória ROM, consumo de energia e nível de utilização do processador, para quantidades específicas de tarefas da aplicação.

```
begin tasks 6
  task 1
    ram = 4324
    rom = 2141
    exectime = 171788
    energy = 37562141
    period = 5000000
  task 2
    ram = 3602
    rom = 1263
    exectime = 97114
    energy = 42982637
    period = 1850000
  task 3
    ram = 2322
    rom = 1874
    exectime = 5407
    energy = 9884025
    period = 900000
  task 4
    ram = 1558
    rom = 498
    exectime = 109251
    energy = 84656389
    period = 3400000
  task 5
    ram = 2238
    rom = 1476
    exectime = 43698
    energy = 52578521
    period = 870000
  task 6
    ram = 5720
    rom = 1432
    exectime = 116070
    energy = 75847452
    period = 1800000
end.

begin platfform
  ram = 32
  rom = 43
  exectime = 0
  energy = 0
end.

begin scheduler
  scheduler RM
    ram = 206
    rom = 295
    exectime = 0
    energy = 535000000
    cpu usage
      1 = 0.1622
      2 = 0.1956
      3 = 0.2306
      4 = 0.2561
      5 = 0.2863
      6 = 0.3165
      7 = 0.3452
    end;
```

```
  scheduler EDF
    ram = 208
    rom = 358
    exectime = 0
    energy = 684000000
    cpu usage
      1 = 0.1941
      2 = 0.2353
      3 = 0.2790
      4 = 0.3083
      5 = 0.3448
      6 = 0.3813
      7 = 0.4154
    end;

end.
```

No terceiro e último arquivo, cujo nome deve ser *design.prj*, sao fornecidos os dados do projeto. Ele contém as restrições do sistema para tamanho de memória RAM, tamanho de memória ROM, período de teste e consumo de energia. Além disso, neste arquivo são informados os pesos da função de custo para cada um dos fatores recém citados e o algoritmo de escalonamento utilizado pelo sistema no projeto em questão.

```
constraints
  ram = 20480
  rom = 10240
  testperiod = 0
  energy = 0
end;

weights
  ram = 0.1
  rom = 0.1
  testperiod = 0.6
  energy = 0.2
end;

scheduler RM

end.
```

Espaços e linhas em branco são ignorados em todos os arquivos.