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

PAULO ROBERTO MIRANDA MEIRELLES

Teste Integrado de Software e Hardware: Reusando Casos de Teste de Software em Teste de Microprocessadores

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

Prof[a]. Dra. Érika Fernandes Cota
Orientador

Prof. Dr. Marcelo Soares Lubaszewski
Co-orientador

Porto Alegre, Junho de 2008

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Meirelles, Paulo Roberto Miranda

Teste Integrado de Software e Hardware: Reusando Casos de Teste de Software em Teste de Microprocessadores / Paulo Roberto Miranda Meirelles. – Porto Alegre: PPGC da UFRGS, 2008.

77 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2008. Orientador: Érika Fernandes Cota; Co-orientador: Marcelo Soares Lubaszewski.

1. Teste de microprocessadores. 2. Teste de software. 3. Teste de hardware. 4. Teste integrado. 5. Sistemas embarcados. 6. Injeção de falhas. I. Cota, Érika Fernandes. II. Lubaszewski, Marcelo Soares. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Prof[a]. Valquíria Linck Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenadora do PPGC: Prof[a]. Luciana Porcher Nedel
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Dedico este trabalho a minha querida mãe, Nizete Miranda.

“Você que está ai sentado,
levante-se.
Há um líder dentro de você.
Governe-o.
Faça-o falar.”
— CHICO SCIENCE

AGRADECIMENTOS

Agradeço aos meus orientadores Érika e Luba pela oportunidade e aprendizado. Também agradeço aos que fazem e se preocupam com a qualidade do ensino e pesquisa dentro do Instituto de Informática da UFRGS, em especial ao Professor Flávio Wagner. Por fim, um agradecimento especial aos meus colegas de laboratório, LSE e de sala de aula, pois a interação com eles proporcionou um aprendizado ainda mais rico.

Obrigado!

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 7

LISTA DE FIGURAS 8

LISTA DE TABELAS 9

RESUMO 10

ABSTRACT 11

1 INTRODUÇÃO 12

2 SELEÇÃO DE TERMINOLOGIAS 15
2.1 Defeito, Falha, Erro e mau funcionamento 15
2.1.1 Engenharia de software 15
2.1.2 Tolerância a Falhas 15
2.1.3 Sistemas Eletrônicos 16
2.2 Verificação, Validação e Teste 17
2.2.1 Engenharia de Software e Tolerância a Falhas 17
2.2.2 Sistemas Digitais 17
2.3 Terminologia selecionada para Teste de Sistemas Embarcados 18
2.3.1 Defeito, Falha, Erro e mau funcionamento 18
2.3.2 Verificação, Validação e Teste 19

3 TESTE DE SOFTWARE 21
3.1 Princípios do Teste de Software 21
3.2 Testabilidade 23
3.3 Casos de Teste e Cobertura de Teste 23
3.4 Critérios do Teste de Software 24
3.5 Técnicas de Teste de Software 25
3.5.1 Teste Funcional 25
3.5.2 Teste Estrutural 27

4 TESTE DE PROCESSADORES EMBARCADOS 32
4.1 Métodos e Tipos de Teste de Processadores Embarcados 32
4.1.1 Modelo de Falhas 34
4.2 Auto-Teste em Processadores Emabarcados Baseado em Software 35

5 REUSANDO CASOS DE TESTE DE SOFTWARE PARA TESTE DE MICROPROCESSADOR 39
5.1 Trabalhos Relacionados 39
5.2 Abordagem de Teste Integrado de Software e Hardware 40
5.3 O Método de Teste Integrado de Software e Hardware 41
5.3.1 Modelo Conceitual 42
5.3.2 Modelo Aplicado 43
5.3.3 Ferramentas por Etapas do Método de Teste Integrado 49
5.4 Estudos de Caso e Resultados 50
5.4.1 Experimentos com Biblioteca de Auto-Teste 54

6 CONSIDERAÇÕES FINAIS 61
6.1 Trabalhos Futuros 61
6.2 Conclusão 62

REFERÊNCIAS 64

APÊNDICE A CÓDIGO-FONTE DOS ESTUDOS DE CASO 69

LISTA DE ABREVIATURAS E SIGLAS

IEEE	Institute of Electrical and Electronics Engineers
HDL	Hardware Description Language
VV&T	Verification, Validation and Test
UML	Unified Modeling Language
OMG	Object Management Group
GFC	Grafo de Fluxo de Controle
DUG	Def-Use Graph
SoC	System-on-a-chip
CMOS	Complementary Metal Oxide Semiconductor
RTL	Register-Transfer Level
SBST	Software-Based Self-Test
HBST	Hardware-Based Self-Test
BIST	Built-In Self-Test
IRST	Instruction Randomization Self-Test
ASIP	Application Specific Instruction set Processor
HDL	Hardware Description Language
ATPG	Automatic Test Pattern Generator
DFT	Design For Testability
CAD	Computer Aided Design
SASHIMI	System As Software and Hardware In MIcrocontrollers
PC	Program Counter

LISTA DE FIGURAS

Figura 2.1: Defeito, Falha, Erro e mau funcionamento em teste de sistemas embarcados 18
Figura 2.2: Verificação, validação e teste em teste de sistemas embarcados 19

Figura 3.1: Representação da relação de dependência dos casos de testes com a especificação. 25
Figura 3.2: Representação da execução de casos de teste com JUnit. 28
Figura 3.3: Representação da relação de dependência dos casos de testes com o código da aplicação. 28
Figura 3.4: Representação da diferentes visões da análise de cobertura de teste do JaBUTi. 29

Figura 4.1: Conceito do auto-teste baseado em software. 37

Figura 5.1: Equação simplificada dos custos do teste do sistema embarcado. . . . 41
Figura 5.2: Equação dos custos do teste do sistema embarcado com teste integrado. 41
Figura 5.3: Fluxo das etapas do método de teste integrado de software e hardware. 42
Figura 5.4: Microarquitetura do FemtoJava Multiciclo. 44
Figura 5.5: Estágios FemtoJava *pipeline*. 44
Figura 5.6: Exemplo de Ciclo e Estágios das Instruções do FemtoJava. 45
Figura 5.7: Máquina de Estado das Instrução Invokestatic do FemtoJava. 46
Figura 5.8: Fluxo de Projeto no SASHIMI. 47
Figura 5.9: Ferramentas utilizadas no fluxo do teste integrado de software e hardware. 49
Figura 5.10: Casos de teste de software em código Java e RAM embarcada. . . . 52
Figura 5.11: Número de bits da RAM comparados para detecção de falhas. . . . 54
Figura 5.12: Detalhes das saídas das simulações com injeções de falhas. 56
Figura 5.13: Rotina determinística regular para o banco de registradores. 57
Figura 5.14: Rotina determinística regular para a ULA. 58

LISTA DE TABELAS

Tabela 3.1: Características das Ferramentas de Teste Java. 30

Tabela 5.1: Cobertura de falhas de hardware com casos de testes de software no FemtoJava multiciclo com aplicação de fluxo de dados. 51

Tabela 5.2: Cobertura de falhas de hardware com casos de testes de software no FemtoJava pipeline com aplicação de fluxo de dados 51

Tabela 5.3: Cobertura de falhas de hardware usando o algoritmo de ordenação como programa de testes no FemtoJava multiciclo. 52

Tabela 5.4: Cobertura de falhas de hardware usando o algoritmo de ordenação como programa de testes no FemtoJava pipeline. 53

Tabela 5.5: Cobertura de falhas dos casos de teste no FemtoJava pipeline em uma aplicação de fluxo de controle. 53

Tabela 5.6: Custos das abordagens de SBST funcional e teste integrado para um algoritmo de ordenação. 53

Tabela 5.7: Cobertura de falhas de hardware usando o algoritmos de Banco de Registradores. 57

Tabela 5.8: Cobertura de falhas de hardware usando o algoritmos de teste da Unidade Lógica-Aritmética. 59

Tabela 5.9: Cobertura de falhas de hardware usando o algoritmos de Banco de Registradores e Unidade Lógica-Aritmética. 59

RESUMO

Sistemas embarcados estão mais complexos e são cada vez mais utilizados em contextos que exigem muitos recursos computacionais. Isso significa que o hardware embarcado pode ser composto por vários processadores, memórias, partes reconfiguráveis e ASIPs integrados em um único silício. Adicionalmente, o software embarcados pode conter muitas rotinas de programação executadas sob restrição de processamento e memória. Esse cenário estabelece uma forte dependência entre o hardware e o software embarcado. Portanto, o teste de um sistema embarcado compreende o teste do hardware e do software. Neste contexto, a reutilização de procedimentos e estruturas de teste é um caminho para se reduzir o tempo de desenvolvimento e execução dos testes. Neste trabalho é apresentado um método de teste integrado de hardware e software. Nesse método, casos de teste desenvolvidos para testar o software embarcado também são usados para testar o seu processador. Comparou-se os custos e cobertura de falhas do método proposto com técnicas de auto-teste funcional. Os resultados experimentais demonstraram que foi possível reduzir os custos de aplicação e geração do teste do sistema usando um método de teste integrado de software e hardware.

Palavras-chave: Teste de microprocessadores. 2. Teste de software. 3. Teste de hardware. 4. Teste integrado. 5. Sistemas embarcados. 6. Injeção de falhas.

Integrated Test of Software and Hardware: Reusing Software Test Cases to Test of Microprocessor

ABSTRACT

Embedded Systems are more complexity. Nowadays, they are used in context that requires computational resources. This means an embedded hardware may be compound of several processors, memories, reconfigurable parts, and ASICs integrated in a single die. Additionally, an embedded software has a lot of programming procedures, which is under processing and memory constraints. This scenario provides a stronger connection between hardware and software. Therefore, the test of an embedded system is the test of both, hardware and software. In this context, reuse of testing structures and procedures is one way to reduce the test development time and execution. This work presents an integrated test of software and software method. In this method, test cases developed to test the embedded software are also used to test its processor. We compared the costs and fault coverage of our proposed method with techniques of functional self-test. The experimental results show that it is possible to reduce the implementation and test generation costs using an integrated test of software and hardware.

Keywords: Embedded Systems, 2. Microprocessor Testing, 3. Software Testing, 4. Hardware Testing, 5. Process Testing, 6. Fault Injection.

1 INTRODUÇÃO

O aumento do desempenho dos processadores e da redução dos custos das memórias possibilitou o crescimento dos sistemas embarcados, que passaram a executar as mais diversas funcionalidades. A complexidade das tarefas executadas por esses sistemas aumentou também significativamente (WONG; RAO; LINN, 2006). Um dos exemplos cotidianos são os aparelhos de telefones celulares que podem ser equipados com câmera fotográfica e filmadora, tocador de música, acesso à Internet, entre outras, além das opções normais de um telefone. Esses dispositivos cada vez mais complexos também são controlados por softwares complexos (WONG; RAO; LINN, 2006). Isso significa o hardware embarcado está mais complexo - vários processadores, memórias, partes reconfiguráveis e ASIPs integrados em uma única pastilha de silício, bem como o software embarcado - com milhares de linhas sob grande restrição de memória.

Testar é um processo que tem como objetivo encontrar as falhas em um sistema. Pode ser para eliminar erros ou por motivos de aceitação das funcionalidades (BROEKMAN; NOTENBOOM, 2003). Embora os participantes de um processo de desenvolvimento de um sistema concordem que é muito melhor impedir falhas a procurá-las e corrigi-las, a realidade é que ainda não se é capaz de produzir sistemas livres de falhas (BROEKMAN; NOTENBOOM, 2003). Assim, o teste é um elemento essencial no desenvolvimento do sistema, ajudando a produzir o que se foi proposto no início do desenvolvimento.

Em software, o objetivo final do teste é fornecer informações de como prosseguir (ou evoluir) no desenvolvimento do sistema em questão. De acordo com as falhas observadas, relacionadas às exigências do sistema, podendo-se tomar as melhores decisões e melhor alocar recursos disponíveis para se obter o sistema de acordo com o proposto (BROEKMAN; NOTENBOOM, 2003). As falhas em um software podem ser observadas quando se determinam conjuntos de dados específicos, denominados de casos de testes (MYERS, 2004), para testar o software em questão. O teste bem sucedido é aquele composto por casos de teste para os quais o programa em teste falhe (MYERS, 2004).

O teste de software é uma tarefa técnica, mas também envolve algumas questões econômicas e da psicologia (subjetividade) humana (MYERS, 2004). Teoricamente, o ideal é a execução de cada caso de teste possível para um programa. Entretanto, na maioria das vezes, isso não é viabilizado facilmente. Mesmo em um simples software pode-se ter centenas ou milhares de combinações possíveis de entradas e saídas (MYERS, 2004), ou seja, um grande conjunto de caso de teste para exercitar todas as alternativas de entradas e saídas e caminhos de execução do software. Criar casos do teste para todas essas possibilidades é pouco prático (MYERS, 2004).

Um teste completo de uma aplicação complexa faria exames demasiadamente longos, sendo o tempo um dos limitadores do teste de software, pois exige recursos humanos economicamente impraticáveis (MYERS, 2004). O testador (projetista de teste) do software

necessita de uma visão apropriada para testar com sucesso uma aplicação do software (MYERS, 2004). Isto é, os casos teste são determinados dependendo da análise realizada pelo testador, implicando na subjetividade do caso de teste como outro limitador do teste do software.

Ao se tratar de software embarcado outros aspectos são agregados ao funcionamento do mesmo. O software embarcado é armazenado em qualquer tipo de memória permanente. Freqüentemente, o código é armazenado em uma ROM, mas também pode ser armazenado em cartões, ou em disco rígido, ou em CD-ROM, e ainda seu *download* pode ocorrer através de uma rede ou de um satélite. O software embarcado é compilado para um processador (ou plataforma) alvo em particular, ou seja, uma unidade de processamento que requer geralmente uma determinada quantidade de RAM para operar, e também, assegura todas as entradas e saídas de sinais com uma camada dedicada de entrada e saída (BROEKMAN; NOTENBOOM, 2003). Além disso, um sistema embarcado pode interagir com outros sistemas (embarcados ou não) através de relações específicas, e geralmente obtém seus recursos de energia (potência) de uma fonte de alimentação própria e dedicada, como baterias.

De acordo com as características do software embarcado, é factível que o teste de software embarcado pode ser dividido em duas etapas, uma em ambiente de simulação e outra em ambiente de execução. A primeira consiste em cobrir falhas inerentes as funcionalidades especificadas no projeto do software e verificação de como o código do software está estruturado. A segunda etapa valida o software, executando-o na plataforma alvo, cobrindo falhas originadas das restrições específicas do ambiente embarcado.

Como o teste de um sistema embarcado compreende o teste do hardware e do software, na referida segunda etapa do teste do software pode-se pensar em verificar o hardware, uma vez que em cada, o esforço para gerar e aplicar os teste podem ser enormes. Assim, a reutilização de procedimentos e estruturas de teste, em todos os níveis, pode ser uma abordagem interessante para reduzir a complexidade dos testes. Para software embarcado, metodologias de teste são baseadas em conceitos de engenharia de software. Por exemplo, a cobertura de teste baseado no código-fonte é um tradicional critério para medir a qualidade do teste do software (HORGAN; LONDON; LYU, 1994) e válido para o teste de software embarcado. Quando se trata de hardware, o processador embarcado é provavelmente o mais complexo componente a ser testado na plataforma. Atualmente, técnicas auto-teste baseado em software (SBST) para microprocessadores são utilizados como uma alternativa de baixo custo em relação às abordagens baseadas em hardware, tais como teste de varredura (*scan*) e auto-teste (BIST - *Built-in Self-test*).

Em um processador projetado para executar instruções de um determinado software, isto é, um sistema embarcado composto de um programa que será executado em um hardware dedicado, é factível um estudo que vise não apenas o uso de técnicas da engenharia de software para aumentar a testabilidade do hardware, quando aplicado em uma HDL (Linguagem de de Descrição de Hardware), mas sim explorar melhor essas técnicas de maneira que aumente a testabilidade do software embarcado com o intuito de também detectar falhas no hardware (por exemplo, stuck-at), além dos erros e falhas de software.

Baseado nas idéias acima, neste trabalho é apresentado um método de teste que tem como base a abordagem de teste integrado de software e hardware para sistemas embarcados. Um processo que visa o teste para os sistemas em que o hardware é projetado para executar instruções de um determinado software, isto é, uma versão ASIP (*Application Specific Instruction set Processor*) de um processador, onde sua unidade de controle possui apenas as instruções usadas por aquele programa em específico.

O principal objetivo do método proposto é atingir um percentual de reuso dos casos de testes projetados para o teste do software no teste de hardware, o que constitui o teste integrado de software e hardware (MEIRELLES; COTA; LUBASZEWSKI, 2008). Uma vez que o processador (hardware) é gerado a partir do código do software, foi avaliado um mecanismo de verificação desse hardware através dos testes para verificar e validar o software. Para verificar o software são usadas as ferramentas de testes que permitam aplicar as técnicas e observar os critérios de teste de software, porém para validar esse software é necessário executá-lo no ambiente em que irá funcionar na prática. Neste ponto do fluxo de projeto, o hardware ainda não está pronto, porém se quer a integração dos testes. Então, a integração deve ser viabilizada por um ambiente de simulação "híbrido" da plataforma alvo, permitindo a validação do software e verificação do hardware quando os mesmos forem submitidos às condições de falhas.

O mecanismo de detecção é simples uma vez que para o método proposto não é necessário armazenar e comparar a RAM sem falhas com a RAM após a execução do programa com falhas, diferentemente de outras abordagens. Portando, há um equilíbrio do tempo de simulação e custos do método proposto. A detecção das falhas com o método de teste integrado é realizada apenas na verificação de específicos endereços de memória, observados de acordo com o número de casos de teste embarcados no código do software. Além disso, a cobertura de falhas, obtidas somente com o reuso, pode ser aumentada com a introdução de rotinas (algoritmos) de teste de hardware implementados no nível do software e sendo embarcado juntamente com os casos de testes construídos para o software, o que também constitui a integração de ténicas de teste de software e de hardware.

Neste trabalho é apresentado um estudo de caso para o método de teste integrado proposto. Casos de teste desenvolvidos para testar um software são usados para testar um microprocessador embarcado, denominado FemtoJava (ITO; CARRO; JACOBI, 2001). Assumiu-se como caso base um tipo auto-teste funcional para microprocessadores e comparou-se os custos e cobertura de falhas com o processo de teste proposto. Os resultados experimentais demonstram que é possível reduzir os custos de aplicação e geração do teste do sistema usando esse método de integração dos testes.

O presente trabalho é constituído de três partes. A primeira é estudo dos conceitos, critérios e diferentes técnicas de teste de software, sinalizando como aplicá-las no teste de software embarcado. Na segunda parte, levantam-se problemas para o teste de processadores embarcados, descrevendo tipos e técnicas de auto-teste para microprocessadores. Por fim, a descrição detalhada do método de teste integrado proposto e os resultados obtidos na aplicação do mesmo nos microprocessadores da família Femtojava. A organização do trabalho é dada como a seguir. No Capítulo 2 é apresentado um levantamento das terminologias em diferentes áreas (engenharia de software, tolerância a falhas e sistemas digitais) que norteiam os assuntos tratados nesta dissertação, selecionando uma terminologia comum para uma melhor compreensão do mesmo. No Capítulo 3 são apresentados os conceitos, critérios e técnicas de teste de software, relacionando com possibilidades de aplicação para o teste do software embarcado, bem como ferramentas de automatização do processo de teste. O Capítulo 3 aborda o teste de processadores embarcados, enfatizando o modelo de falhas (*stuck-at*) e as técnicas de auto-teste de processadores baseado em software. O Capítulo 4 atrás o objeto desta dissertação, um método de teste integrado de software e hardware, apresentando um modelo conceitual, um modelo prático e os resultados experimentais dos estudos de caso - uma aplicação de fluxo de dados (um algoritmo de ordenação) e uma aplicação de fluxo de dados (controle de guindaste). Por fim, no Capítulo 6, as conclusões e as possilidades de trabalhos futuros.

2 SELEÇÃO DE TERMINOLOGIAS

Ao longo do presente trabalho, serão utilizados termos, tais como: defeito, falha, erro, mau funcionamento, verificação, validação e teste. Esses, principalmente os quatro primeiros, possuem diferentes definições nas áreas de engenharia de software, tolerância a falhas (em hardware e em software) e teste de sistemas digitais. A terminologia de cada área é apresentada nas seções a seguir. Por fim, será selecionada uma combinação entre essas terminologias para serem empregadas no contexto de teste integrado de software e hardware.

2.1 Defeito, Falha, Erro e mau funcionamento

2.1.1 Engenharia de software

Há alguns anos o IEEE (*Institute of Electrical and Electronics Engineers*) tem realizados esforços a fim de promover a padronização em diferentes âmbitos da computação, entre esses está a terminologia utilizada no contexto da engenharia de software. Dessa forma, o padrão IEEE de número 610.121990 (IEEE, 1990) apresenta a diferenciação entre os seguintes termos:

- **Defeito** (*Fault*): É uma definição incorreta dos dados (especificação) de um software, por exemplo, levando a uma instrução ou comando incorreto.
- **Erro** (*Error*): É a diferença entre o valor computado e o valor esperado, ou teoricamente correto. Assim, resultado intermediário incorreto ou resultado inesperado na execução do software caracteriza um erro. Além disso, também corresponde a uma ação humana que produza um resultado incorreto. Por exemplo, uma ação incorreta de um programador ou operador.
- **Falha** (*Failure*): É a inabilidade de um sistema ou de um componente de software em executar suas funções requeridas dentro das exigências especificadas. Ocorre quando defeitos ou erros causam um resultado inesperado quando o programa está executando determinadas entradas, caracterizando também o que é demonimado com *bug*.

2.1.2 Tolerância a Falhas

A comunidade acadêmica brasileira da área de tolerância a falhas definiu a terminologia para os referidos conceitos em (WEBER; WEBER; PORTO, 1990), derivada dos trabalhos de Laprie (LAPRIE, 1985)e (LAPRIE, 1998). Assim, interessados no sucesso de um determinado sistema em atender sua especificação, os termos foram traduzidos e conceituados da seguinte forma:

- **Defeito** (*Failure*): Um defeito ocorre quando a entrega de um serviço por um sistema desvia da especificação, determinando um resultado incorreto.
- **Erro** (*Error*): Um sistema está em estado errôneo (em erro) se o processamento posterior, a partir desse estado, levar ao defeito (*failure*). Podendo não ser reconhecido como um erro ou não ser detectado. Além disso, um erro pode se propagar, produzindo outros erros. Já os defeitos são conhecidos quando os erros são detectados, mas não necessariamente um erro leva a um defeito.
- **Falha** (*Fault*): É a identificação ou hipótese da causa de um erro. Por exemplo, em hardware é a causa física do erro. Mas uma falha não obrigatoriamente leva a um erro. Falhas são inevitáveis. Os componentes físicos envelhecem e sofrem as interferências externas. Já os softwares podem estar sujeitos a uma alta complexidade, a uma deficiências nas especificações e/ou problemas humanos de trabalhar com grande volume de detalhe.

Nota-se que a tradução dos termos *fault* e *failure*, na área de tolerância a falhas, é oposta à tradução apresentada no contexto da engenharia de software.

2.1.3 Sistemas Eletrônicos

Em se trantando de sistemas de hardware, de acordo com (PRADHAN, 1996; BUSHNELL; AGRAWAL, 2000a), a terminologia a seguir é utilizada:

- **Defeito** (*Defect*): Diferentemente das demais, a área de sistemas eletrônicos trata de forma isolada os problemas que podem surgir no nível físico do sistema. Assim, define-se como defeito (defect) em um sistema eletrônico 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.
- **Falha** (*Fault*): É a abstração de um defeito no nível de funções (modelos físicos ou lógicos do dispositivo). A diferença entre defeito (*defect*) e falha (*fault*) é bastante sutil. De uma maneira simples, defeito e falha podem ser definidas como imperfeições no hardware e na função desejada, respectivamente.
- **Erro** (*Error*): Um valor de saída errôneo produzido por um circuito defeituoso (com falha). 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.
- **Mau funcionamento** (*Failure*): Causado por erros não tratados ou não contidos. Isto é, o mau funcionamento (*failure*) consiste no efeito, perceptível ao usuário final, de um erro não contido, que por sua vez foi originado por uma falha ativa.

O termo mau funcionamento é uma tradução livre apresentada em (MORAES, 2006), que pretende evitar conflito com as traduções de *fault* e *defect*, mais fortemente estabelecidas na literatura de teste de sistemas digitais.

2.2 Verificação, Validação e Teste

As diferenças de significados entre os termos verificação (*verification*), validação (*validation*) e teste (*test*) nas áreas de engenharia de software, tolerância a falhas e sistemas digitais não ocasionam grande discrepância de interpretação quanto aos demais tratados neste capítulo, assim como, não há traduções diversas para os mesmos.

A engenharia de software, em (PRESSMAN, 2006), e a tolerância a falhas, por (AVIZIENIS et al., 2004), referenciam a mesma definição, apresentado em (**?**), ao abordar os termos verificação e validação. Além disso, a taxonomia de teste é semelhante em ambas as áreas, de acordo com o apresentado em (PRESSMAN, 2006) e (IEEE, 1990) para engenharia de software e por (AVIZIENIS et al., 2004) para tolerância a falhas. Dessa forma, é necessária uma diferenciação desses termos apenas para um contexto de software, englobando a engenharia de software e a tolerância a falhas, e um contexto de hardware, constituído da área de sistemas digitais.

2.2.1 Engenharia de Software e Tolerância a Falhas

De acordo com os conceitos apresentados em (AVIZIENIS et al., 2004),(**?**), (IEEE, 1990), (PRESSMAN, 2006), e (KRUG, 2007), nesta subseção define-se a terminologia para verificação, validação e teste nas áreas de engenharia de software e tolerância a falhas como sendo a mesma, conforme citado anteriormente.

Uma observação, no sentido de melhorar o entendimento das definições, é que o termo software, utilizado nas descrições das taxonomias, pode ser substituído por produto ou sistema, não restringindo apenas à umas das áreas citadas, mas englobando as ambas tratadas nesta subseção.

- **Verificação:** Refere-se ao conjunto de atividades que garante que o software implementa corretamente uma função específica, avaliando se o sistema atende as propriedades dadas. O objetivo do processo de verificação é avaliar a consistência de uma implementação com uma especificação, assegurando que o produto será completo e correto. Pode ser definido, em suma, com a seguinte pergunta: estamos construindo o software corretamente?

- **Validação:** É o processo de avaliar a qualidade de um software, observando se o mesmo está desempenhando o que lhe foi requerido, no sentido de atender às reais necessidades do usuário. Tem como objetivo revelar as falhas de especificação e erros de codificação, assegurando que o produto final corresponda ao que foi solicitado. Também pode ser resumido na seguinte pergunta:estamos construindo o produto certo?

- **Teste:** É uma verificação dinâmica através do exercício (execução) do sistema, examinando o comportamento do produto. Assim, a partir de um conjunto de valores de entrada estimulam-se as variáveis do programa que, por sua vez, executa suas funcionalidades e por fim gera resultados, os quais são comparados com os resultados esperados para aquela aplicação. O foco do teste está em primeiro em verificar e depois validar, de forma a oferer elementos de auxílio aos processos de verificação e validação.

2.2.2 Sistemas Digitais

Os autores da área de sistemas digitais dividem as etapas de teste do fluxo de projeto de um circuito em verificação e teste (PRADHAN, 1996; SANGIOVANNI-VINCENTELLI;

MCGEER; SALDANHA, 1996; LUBASZEWSKI; COTA; KRUG, 2002). O termo validação não é enfatizado por esses, pois faz parte do processo de verificação, porém (KRUG, 2007) cita que novos trabalhos estão dando um sentido mais amplo ao termo validação, como descrito a seguir, na diferenciação dessas taxonomias:

- **Verificação:** É um processo realizado antes da implementação do circuito, consistindo em realizar uma reprodução do comportamento do sistema com uma execução próxima do projeto, através de um protótipo, ou ainda de técnicas matemáticas que envolvem a construção de um modelo e de um simulador em um computador, com intuito de assegurar o atendimento às especificações funcionais. Assim, as medidas virtuais ou reais são examinadas para avaliar a qualidade do projeto, tendo como objetivo verificar se o projeto está de acordo com a especificação, ou seja, correto.

- **Validação:** A validação de hardware é uma técnica de verificação de projeto. É um termo recentemente efatizado nessa área, pois, visualizando a similaridade entre as descrições HDL (*Hardware Description Language*) com as linguagens de programação tradicionais, vários trabalhos surgiram com o intuito de utilizar técnicas utilizadas para o teste de software em teste de hardware, principalmente no que é chamado de ”validação de projetos de hardware”. Em suma, pode ser destacado como o uso de técnicas de teste da engenharia de software no processo de verificação do hardware.

- **Teste:** O teste deve avaliar (verificar) se o dispositivo físico está de acordo com as especificações, se o circuito tem um comportamento funcional correto, denominado também de teste funcional, ou se a implementação física do circuito espelha seu esquemático, também denominado de teste estrutural. Quando o projeto está implementado em silício pode ser verificado, então, sendo denominado de teste, correspondendo à fase onde se deve assegurar que somente circuitos livres de defeitos sejam comercializados, detectando falhas de fabricação do circuito impresso.

2.3 Terminologia selecionada para Teste de Sistemas Embarcados

Esta seção define a taxonomias especificamente para teste de sistemas embarcados, isto é, software e hardware embarcado.

2.3.1 Defeito, Falha, Erro e mau funcionamento

A Figura 2.1 ilustra os termos selecionados (defeito, falha, erro e mau funcionamento), da mesma forma como se relacionam suas definições para este trabalho, apresentadas a seguir:

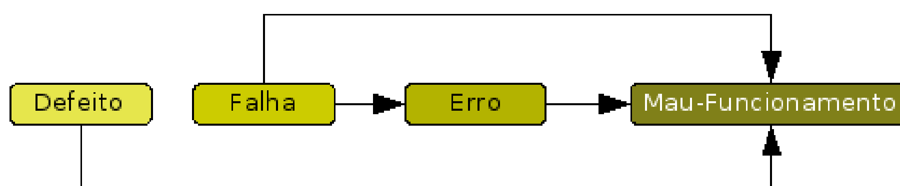


Figura 2.1: Defeito, Falha, Erro e mau funcionamento em teste de sistemas embarcados

- **Defeito** (Defect): Refere-se aos problemas físicos do sistema, isto é, defeito, falha, erro e mau funcionamento no hardware, conforme visto na seção 2.1.3, que possam interferir no funcionamento do sistema embarcado, podendo causar um mau funcionamento do mesmo.

- **Falha** (Fault): É o problema vinculado a complexidade do código do software e do hardware embarcado, deficiências nas especificações, com problemas humanos de trabalhar com grande volume de detalhe e/ou imperfeições no hardware e na função desejada. Por exemplo, levando a uma instrução ou comando incorreto.

- **Erro** (Error): É a diferença entre o valor computado e o valor esperado, ou teoricamente correto. Assim, resultado intermediário incorreto ou resultado inesperado na execução do software caracteriza um erro. Uma falha humana, por exemplo, pode causar um erro no software embarcado.

- **Mau funcionamento** (Failure): Defeitos de hardware, falhas e/ou erros no software e o hardware embarcado, não tratados, cujos efeitos sejam perceptíveis ao usuário final do sistema, constitui o mau funcionamento do sistema embarcado.

2.3.2 Verificação, Validação e Teste

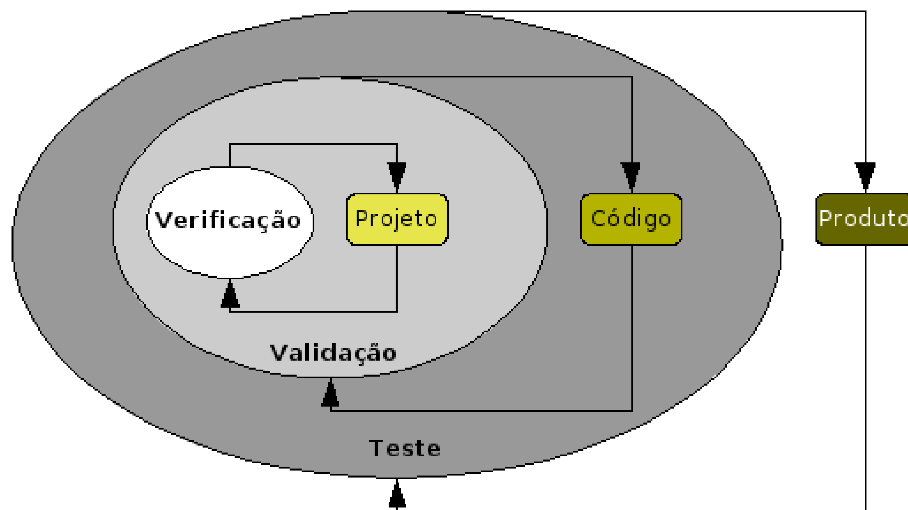


Figura 2.2: Verificação, validação e teste em teste de sistemas embarcados

A taxonomia escolhida para compor a definição particular neste trabalho de cada verificação, validação e teste é a mesma apresentada para as áreas de engenharia de software e tolerância a falhas, descrita na Seção 2.2.1. A Figura 2.2 demonstra que a atividade de teste é composta pelo processo de verificação, primeiramente, e posteriormente pela validação:

- As atividades de **verificação** estão vinculadas ao projeto do sistema embarcado, podendo fazer uso de simulações, de acordo com a definição do termo.

- A **validação** é um outro conjunto de passos que tem como foco e parâmetro o código do software e hardware embarcado, revelando falhas de especificação não sanadas pela verificação e erros de codificação.

- O termo **teste** se refere às atividades de examinar o comportamento do sistema embarcado (produto) pronto, a partir de um conjunto de valores de entrada estimula as variáveis do programa que, por sua vez, executa suas funcionalidades e por fim gera resultados, os quais são comparados com os resultados esperados para essa aplicação.

3 TESTE DE SOFTWARE

Uma vez especificado ou construído o código fonte, o software deve ser testado. Para isso, uma série de casos de teste que têm uma grande probabilidade de encontrar falhas e erros devem ser projetados. Isso é realizado através de técnicas de teste de software que fornecem diretrizes sistemáticas para projetar os testes (PRESSMAN, 2006).

O principal objetivo das atividades de teste é aumentar a confiabilidade e garantir a qualidade dos produtos de software produzidos, tendo como principal aspecto do teste de software encontrar falhas e erros, também aumentando a aceitabilidade do mesmo. Erroneamente, pensa-se em teste como a forma de provar que o software está correto. Ao pensar dessa forma o testador tende a produzir casos de teste pobres, com poucas chances de encontrar erros. Caso contrário, tem-se um bom potencial de produzir casos de teste para descobrir falhas e erros não detectados (MYERS, 2004).

No âmbito do software de propósito geral, (KRUG, 2007) comenta que entre os desenvolvedores de aplicativos é comum dizer que o mau funcionamento do software só aparece quando o usuário (cliente) vai utilizar o sistema, isto é, quando o mesmo já está em uso. Isso acontece porque o desenvolvedor ao realizar os testes o faz de forma viciosa, fazendo-os em cima das consistências existentes em seu programa, enquanto que o usuário, sob as mais variadas condições, irá utilizar o sistema para diversas situações e combinações, que em muitos casos não foram simuladas na fase de desenvolvimento do software. Portanto, deve-se destacar que o software deve estar preparado para qualquer situação, que são formuladas em forma de casos de teste.

3.1 Princípios do Teste de Software

Além dos objetivos e outros ponto enfatizados anteriormente, em (MYERS, 2004) são destacados os princípios do teste de software:

- Uma parte necessária de um caso de teste é uma definição da saída ou resultado esperado: há um desejo “subconsciente” em se obter o resultado correto. Uma maneira de combater isso é incentivar um exame detalhado de todas as saídas, adiantando as saídas previstas do programa. Conseqüentemente, um caso do teste deve consistir (i) em uma descrição das entradas do programa e (ii) uma descrição precisa das saídas corretas.

- Um programador deve evitar testar seu próprio programa: depois que um programador projetou e codificou, de forma “construtiva”, um programa, é extremamente difícil mudar a perspectiva de maneira que tenha visão “destrutiva” do seu trabalho. Além disso, o software pode conter erros devido às falhas do programador na

indicação ou na especificação do problema, provavelmente carregando as mesmas falhas ao testes. Nota-se que esses argumentos não se aplicam ao *debugging* (que corrige erros conhecidos, ao contrário do teste), sendo executado mais eficientemente pelo programador original.

- Uma equipe de programadores não deve testar seus próprios programas: os argumentos anteriores se aplicam a este caso, mas somado ao fato de uma equipe ser avaliada em produzir um software em um dado tempo com um certo custo, dificultando o discernimento em quantificar a confiabilidade do produto. Dessa forma, implicando, ao final, ser mais econômico que o teste seja executado por um grupo com o objetivo de somente testar.

- Inspecionar completamente os resultados de cada teste: é um princípio óbvio, mas é freqüentemente negligenciado. Varias experiências mostram casos que não se detectam determinados erros, mesmo quando os sintomas daqueles erros eram claramente verificáveis nas listas de saída.

- Os casos do teste devem ser escritos para as condições da entrada que são inválidas e inesperadas, bem como para aquelas que são válidas e esperadas: A tendência natural é produzir casos de teste com entradas válidas e esperadas. Porém, muitos erros são descobertos quando o programa é usado em alguma maneira nova ou inesperada. Conseqüentemente, os casos do teste que representam condições inesperadas e inválidas da entrada parecem ter um rendimento mais elevado na detecção de erros do que testar argumentos para condições válidas da entrada.

- Examinar um programa para verificar se não faz o que se propôs a fazer é somente metade do caminho; a outra metade está em ver se o programa faz o que não se propõe para fazer: justificativa semelhante ao princípio anterior. Os programas devem ser examinados para efeitos não desejados.

- Evitar casos de teste *throwaway* (de exame rápido) a menos que o programa seja verdadeiramente um de *throwaway*: Uma prática comum é sentar-se em um terminal e inventar os casos de teste *on the fly*. O problema principal é que os casos do teste representam um investimento valioso que, nesse ambiente, desaparece depois que o teste foi terminado.

- Não planejar testes sob a suposição implícita que nenhum erro será encontrado: não supor que o teste é um processo para mostrar que o programa funciona corretamente, uma vez que, a definição de teste determina em ser é um processo que executa um programa com o objetivo de encontrar falhas e erros.

- A probabilidade da existência de mais erros em uma seção de um programa é proporcional ao número dos erros já encontrados nessa seção: Considere um programa que consiste em dois módulos, em classes, ou nas rotinas secundárias A e B, e cinco erros foram encontrados no módulo A e somente um erro foi encontrado no módulo B. Caso o módulo A não seja exposto, propositadamente, a um teste mais rigoroso, provavelmente há mais erros no módulo A do que no módulo B. Isto é, os erros tendem a existir em conjuntos. Assim algumas seções parecem ser muito mais propensas aos erros do que outras. Esta informação é útil para fornecer uma introspecção do processo de teste. Se algumas seções em particular de um programa

parecem ser muito mais propensa aos erros do que as demais, os esforços em testes adicionais devem ser focalizados nessas seções.

- Testar é um desafio (tarefa) extremamente criativa e intelectual: é provavelmente verdadeiro que a criatividade requerida em testar um software excede a criatividade requerida em projetar esse programa, visto que é impossível testar suficientemente um programa para garantir a ausência de todas as falhas e erros.

3.2 Testabilidade

A testabilidade é definida como a capacidade de testar certos atributos internos ao sistema ou facilidade de realizar certos testes. É uma das medidas de dependabilidade (*dependability*), que é o objetivo de tolerância a falhas. Assim, quanto maior a testabilidade, melhor a mantenabilidade, por conseqüência menor o tempo de indisponibilidade do sistema devido à reparos. A mantenabilidade é a facilidade de realizar a manutenção do sistema, isto é, a probabilidade que um sistema com mau funcionamento seja restaurado a um estado operacional dentro de um período determinado (WEBER; WEBER; PORTO, 1990).

O software ou uma unidade do software é testável por um domínio se esse é controlável e observável, podendo ser modificado para ser testável por um domínio pela adição de todas as variáveis usadas no software em parâmetros de entrada e reduzindo as saídas declaradas para uso de domínios reais. Entende-se por domínio os valores possíveis que uma variável pode assumir de acordo com sua declaração (tipo de dado e tamanho). Em outras palavras, um produto é testável se oferece suporte a geração de testes, implementação e verificação de seus resultados de forma precisa.

No contexto de hardware, a controlabilidade refere-se à facilidade de controlar certo valor lógico em um ponto determinado do circuito, através da modificação dos valores das entradas do mesmo. Já a observabilidade avalia a facilidade de observar o valor lógico de um dado ponto do circuito nas saídas primárias desse (ABRAMOVICI; FRIEDMAN; BREUER, 1990).

3.3 Casos de Teste e Cobertura de Teste

Um conjunto de entradas de teste, em condições de execução, e de resultados previstos, projetado para um objetivo particular, como exercitar um trajeto particular do programa ou para verificar a conformidade com uma exigência específica (IEEE, 1990), é denominado de caso de teste. Diferentemente de “procedimento de teste” que é uma descrição dos passos necessários para executar um caso (ou um grupo de casos) de teste (CRAIG; JASKIEL, 2002).

O objetivo dos casos de teste é descobrir falhas e erros, utilizando-se de um critério de teste com um mínimo esforço e tempo (PRESSMAN, 2006). Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não detectado (MYERS, 2004).

O teste exaustivo é impraticável, principalmente devido às restrições de tempo e custo (WHITTAKER, 1997). Isso implica em determinar quais casos de teste devem ser utilizados para que a maioria dos erros possam ser encontrados e que o número de casos de teste utilizados não seja tão grande a ponto de ser impraticável (SOUZA DE, 1996). Para sanar tal problema, critérios e técnicas de teste foram elaboradas fornecendo sistematica-

mente e de forma rigorosa meios de selecionar um subconjunto de casos de teste, eficaz para revelar a presença dos erros existentes, respeitando as restrições de tempo e custo associados a um projeto de software (VINCENZI, 1998).

Uma maneira de mensurar um caso de teste é diagnosticar a sua cobertura. A cobertura de teste é o grau a que um teste ou um conjunto de casos dos testes corresponde a todas as exigências especificadas para um sistema ou um determinado componente de software (IEEE, 1990).

Existem duas classes de cobertura (MALDONADO et al., 1998): (i) coberturas baseadas em caminho, essas requerem a execução de componentes em particular de um programa, assim como sentenças, desvios, ou caminhos completos; (ii) e a cobertura baseada em erros que requer que o conjunto de teste exercite o programa em um caminho que revele prováveis falhas de especificação e erros no código.

Em torno de 30% das tarefas de teste podem ser gastas com a composição dos casos de teste (CHILLAREGE, 1999), se a mesma for realizada de maneira manual. Desse modo, observa-se a importância da automatização desse processo, através de ferramentas que utilizem técnicas para geração automática dos casos de teste ou que auxiliem a elaboração dos mesmos. Um levantamento das ferramentas para automatização do teste de software é tratado ao longo deste capítulo.

3.4 Critérios do Teste de Software

Um critério de teste é o que define quais propriedades precisam ser testadas para garantir inexistência do mau funcionamento do software(ROCHA; MALDONADO; WEBER, 2001). Como é impossível garantir inexistência, o conceito é utilizado, na prática, para definir uma qualidade mínima que será avaliada pelo teste.

Um critério de teste serve para selecionar e avaliar casos de teste de forma a aumentar as possibilidades de provocar erros ou, quando isso não ocorre, estabelecer um nível elevado de confiança na correção do produto (ROCHA; MALDONADO; WEBER, 2001).

Os critérios de teste tratam duas questões chaves na atividade de teste (MALDONADO et al., 2004): “Como os dados de teste devem ser selecionados?” e “Como decidir se um programa P foi suficientemente testado?”. Tais critérios são fundamentais para o sucesso dos casos de teste e o sucesso da atividade de teste. Os mesmos devem fornecer indicações de quais casos de teste devem ser utilizados com o intuito de aumentar as chances de revelar erros, assim como, estabelecer um nível elevado de confiança na correção do software, quando alguns erros não forem detectados.

Em geral, pode-se dizer que as propriedades mínimas que devem ser preenchidas por um critério de teste são (MALDONADO et al., 2004):

- Garantir, do ponto de vista de fluxo de controle, a cobertura de todos os desvios condicionais;

- Requerer, do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional; e

- Requerer um conjunto de casos de teste finito.

Por fim, como a cobertura de teste consiste basicamente em determinar o percentual de elementos requeridos por um dado critério de teste que foram exercitados pelo conjunto de casos de teste utilizado (MALDONADO et al., 2004). Então, a partir dessa informação

o conjunto de casos de teste pode ser aprimorado, acrescentando-se novos casos de teste para exercitar os elementos ainda não cobertos.

3.5 Técnicas de Teste de Software

As técnicas de teste são classificadas de acordo com a origem das informações utilizadas para estabelecer os requisitos de teste. Elas Abrangem diferentes perspectivas do software e impõem a necessidade de se estabelecer uma estratégia de teste que contemple as vantagens e os aspectos complementares dessas técnicas (MALDONADO, 1991).

As técnicas existentes podem ser divididas em três: técnica (teste) funcional, técnica (teste) estrutural, técnica (teste) com base em erros (VINCENZI, 1998; MALDONADO et al., 2004). Cada uma das técnicas de teste mencionadas possui diversos critérios de teste. Neste trabalho são apresentadas mais detalhadamente as duas primeiras técnicas citadas, com os seus respectivos critérios aplicados. Essas duas técnicas também são mais comumente aplicadas ao escopo do teste de software embarcado (BROEKMAN; NOTENBOOM, 2003).

3.5.1 Teste Funcional

O teste funcional, também denominando de caixa-preta, baseado em especificação ou comportamental, tem esse nome pelo fato de tratar o software como uma caixa cujo conteúdo é desconhecido e só é possível visualizar o lado externo. Desse modo, o testador utiliza essencialmente a especificação de requisitos do programa para derivar os casos de teste que serão empregados sem se importar com os detalhes de implementação (BEIZER, 1990). Assim, uma especificação correta e de acordo com os requisitos do usuário é de fundamental importância para apoiar a aplicação dos critérios relacionados a essa técnica. A Figura 3.1 (adaptado de (JORGENSEN, 2002)) representa que os casos de testes são um subconjunto da especificação, uma vez que os mesmos no teste funcional são exclusivamente implementados baseados na especificação.

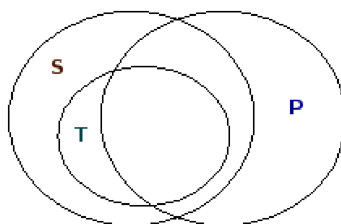


Figura 3.1: Representação da relação de dependência dos casos de testes com a especificação.

As principais vantagens do teste funcional, em resumo, são (JORGENSEN, 2002; PRESSMAN, 2002):

- Casos de Teste são independentes de como o software é implementado, tornando-os mais fáceis de serem re-usados;
- Desenvolvimento de casos de teste pode ocorrer mais cedo dentro do processo de desenvolvimento;
- Detectar erros de funções incorretas ou omitidas, erros de interfaces, erros de estruturas de dados ou de acesso a base de dados externas, erros de comportamento, desempenho, inicialização ou término;

- Aplicados aos programas procedurais e aos programas orientados a objetos.

Como desvantagens citadas pelos mesmos autores, de forma sumária, se tem:

- Casos de teste podem conter redundâncias;
- Cobertura de teste (em relação ao código) pode não ser efetiva;
- Dificuldade de quantificar a atividade de teste, visto que não se pode garantir que partes essenciais ou críticas do programa sejam executadas;
- Sujeito às inconsistências decorrentes da especificação;
- Dificuldade de automatização dos critérios funcionais. Visto que a especificação é feita de forma descritiva e informal, os requisitos derivados da especificação também são, de certa forma, descritivos e informais.

Exemplos de critérios de teste funcional são (PRESSMAN, 2002):

- Particionamento de Equivalência: o domínio de entrada e/ou saída do software é dividido em classes de equivalência válidas e inválidas, de acordo com as condições de entradas especificadas. Posteriormente um menor número de casos de teste é definido, sendo um elemento de uma dada classe representante da classe toda, gerando para cada uma das classes inválidas um caso de teste distinto. Com isso, os requisitos são verificados de forma mais sistematizada e com um menor número de casos de teste.
- Análise do Valor Limite: complementa o critério Particionamento de Equivalência, sendo que os casos de teste são escolhidos nos limites das classes, pois nesses pontos se concentra um grande número de falhas. O espaço de saída do programa também é particionado e são exigidos casos de teste que produzam resultados nos limites dessas classes de saída. Baseado em Grafos (Grafo de Causa-efeito): estabelece requisitos de teste baseados nas possíveis combinações das condições de entrada. Dessa forma, são levantadas as possíveis condições de entrada (causas) e as possíveis ações (efeitos) do programa. Posteriormente é construído um grafo relacionando as causas e efeitos levantados. Esse grafo é convertido em uma tabela de decisão a partir da qual são derivados os casos de teste.
- Teste de comparação: hardware e software redundantes para minimizar a possibilidade de erros, tendo a confiabilidade do sistema como crítica. No caso de software redundante equipes distintas produzem versões independentes de uma aplicação usando a mesma especificação. Assim, cada versão é testada com os mesmos dados, comparando as saída de cada versão. Cada uma das aplicações é investigada se os resultados são diferentes. Uma outra situação possível é o fato da especificação estar errada. Com isso, provavelmente todas as versões refletirão o erro.
- Teste de matriz ortogonal: Aplicado a problemas com domínio de entrada relativamente pequeno, mas grande demais para testes exaustivos. Se o domínio de entrada é limitado, é possível realizar testes exaustivos de todas as combinações de valores de entrada. Por exemplo, se três parâmetros que assumem três valores discretos cada um ($3 \times 3 \times 3 = 27$ casos de teste). Para domínios de entrada muito grandes, uma abordagem comum é variar um parâmetro de cada vez, mas não detectando interações entre parâmetros.

O teste estatístico de software (*statistical software testing*) também pode ser visto como teste funcional uma vez que o mesmo também leva em conta a especificação do software para gerar o conjunto de teste. Exemplos de tais critérios podem ser encontrados em (THéVENOD-FOSSE; WAESELYNCK, 1993; WHITTAKER; M.THOMASON, 1994; WHITTAKER, 1997). A idéia desses critérios é a de exercitar um programa com valores aleatórios selecionados do domínio de entrada utilizando-se uma função de distribuição. Assim, a eficácia desses critérios está condicionada à capacidade de se derivar uma função de distribuição que maximize a probabilidade de uma entrada que revele falhas. Basicamente, como definido por (THéVENOD-FOSSE; WAESELYNCK, 1993), os conjuntos de testes estatísticos são definidos por dois parametros: (i) o perfil de teste, ou a distribuição de entrada a partir da qual os dados de teste são selecionados aleatoriamente; e (ii) o número de dados de teste a serem gerados.

Em (OFFUTT; IRVINE, 1995) foi conduzido um estudo piloto para avaliar a aplicação dos critérios de teste funcional no contexto de programas orientado a objetos e observam que não existem evidências de que as técnicas e critérios de teste tradicionais, destinadas ao teste de programas procedimentais, tenham ineficácia para o teste de programas orientado a objetos. Os resultados obtidos por (OFFUTT; IRVINE, 1995) indicam que a combinação do método de partição de equivalência com uma ferramenta para detectar erros de alocação e desalocação de memória pode ser uma estratégia de teste eficaz para o teste de programas C++.

3.5.1.1 Ferramenta para Teste Funcional

JUnit [1] é um *framework* (arcabouço) de teste que é utilizado e viabiliza a documentação e execução automática de casos de teste em Java. O *framework* JUnit é de código aberto e pode ser utilizado para escrever e executar de forma automática um conjunto de teste fornecendo relatórios sobre quais casos de testes não se comportaram de acordo com o que foi especificado, conforme representado na Figura 3.2. A idéia básica é implementar algumas classes específicas que armazenam informações sobre os dados de entrada e a respectiva saída esperada para cada caso de teste. Após a execução de um caso de teste, a saída obtida é comparada com a saída esperada e qualquer discrepância é reportada. O principal problema do JUnit é que ele não fornece informação a respeito da cobertura obtida pelos casos de teste. JUnit pode ser utilizado mesmo que somente o bytecode e a especificação do programa estejam disponíveis.

3.5.2 Teste Estrutural

Na técnica de teste estrutural, também conhecida como teste caixa branca, os aspectos de implementação são fundamentais na escolha dos casos de teste. O teste estrutural baseia-se no conhecimento da estrutura interna da implementação. Em geral, a maioria dos critérios dessa técnica utiliza uma representação de programa conhecida como grafo de fluxo de controle ou grafo de dados. Um programa pode ser decomposto em um conjunto de blocos disjuntos de comandos (MALDONADO et al., 1998). A Figura 3.3 (adaptado de (JORGENSEN, 2002)) representa que os casos de testes podem ser um subconjunto do código, uma vez que os mesmos no teste estrutural podem ser baseado em detalhes da implementação ao ser acesso ao código do software.

Os critérios de teste estrutural mais utilizados são:

a) Critérios Baseados em Fluxo de Controle: utilizam apenas características de con-

[1] `http://www.junit.org/`

```
Package Explorer | Hierarchy | JUnit
Finished after 0,081 seconds
Runs: 9/9    Errors: 0    Failures: 0
agenda.AgendaTest [Runner: JUnit 3]

Control16.java | Sensor16.java | Crane.java | AgendaTest.java
91          System.out.print("Buscar por Pessoa: ");
92          assertEquals(agenda.getPessoa("Paulo"), (Pessoa)agenda.l
93      }
94
95      public void testListarPessoas(){
96          testAddPessoa();
97          Pessoa pessoa;
98          for(int i=0;i<agenda.qtdPessoas();i++){
99              pessoa=agenda.getPessoa(i);
100             System.out.println("Pessoa no Vetor numero: " + i );
101             if (pessoa.getClass().getName()=="Pf"){
102                 System.out.println(((Pf)pessoa).toString()); //
103             }
104             else if (pessoa.getClass().getName()=="Pf"){
105                 System.out.println(((Pj)pessoa).toString()); //
106             }
107             assertEquals(pessoa, (Pessoa)agenda.lista.elementAt(
108         }
109     }
110
111     public void testQtdPessoas() { // acrescentado na segunda rc
112         testAddPessoa();
113         assertEquals(agenda.qtdPessoas(), agenda.lista.size());
114     }
```

Figura 3.2: Representação da execução de casos de teste com JUnit.

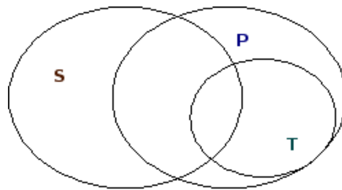


Figura 3.3: Representação da relação de dependência dos casos de testes com o código da aplicação.

trole da execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias. Os critérios mais conhecidos dessa classe são: (i)Todos-os-Nós - exige que a execução do programa passe ao menos uma vez em cada vértice do grafo de fluxo, ou seja, que cada comando do programa seja executado pelo menos uma vez; (ii)Todos-os-Arcos - requer que cada aresta do grafo, ou seja, cada desvio de fluxo de controle do programa, eja exercitada pelo menos uma vez; e (iii)Todos-os-Caminhos - requer que todos os caminhos possíveis do programa sejam executados (MALDONADO et al., 1998).

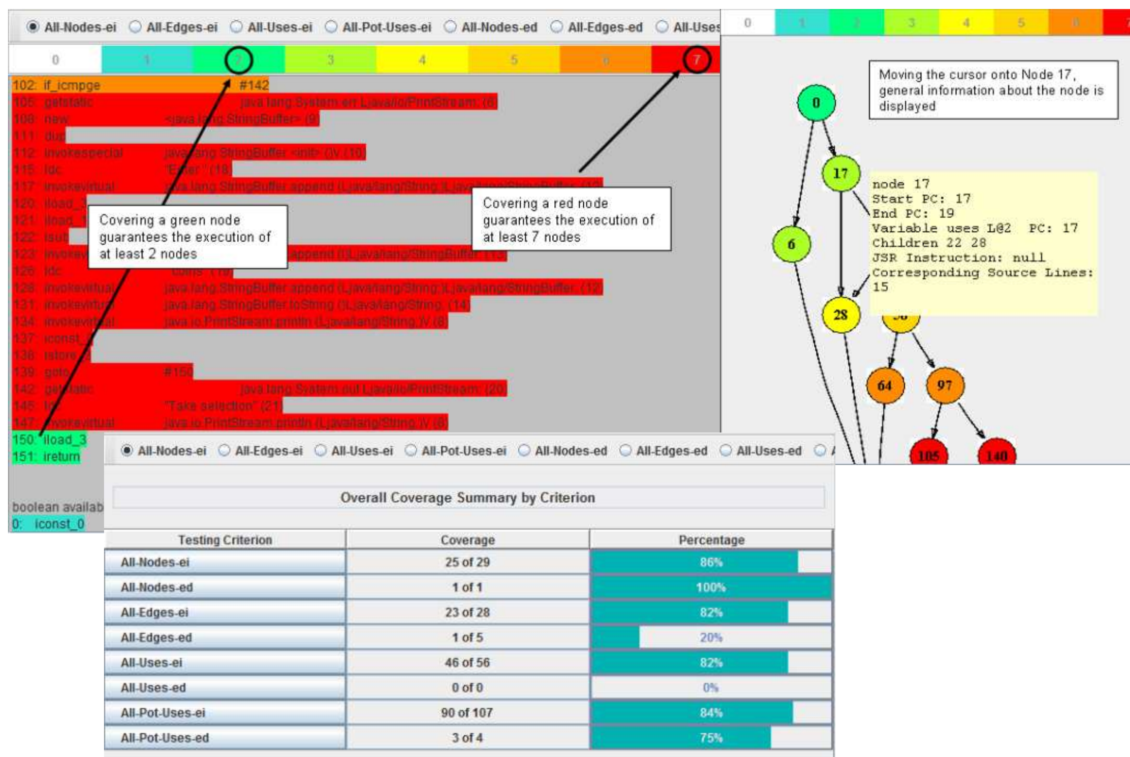
b) Critérios Baseados em Fluxo de Dados: utilizam informações do fluxo de dados do programa para determinar os requisitos de teste. Esses critérios exploram as interações que envolvem definições de variáveis e referências a tais definições para estabelecerem os requisitos de teste. Exemplos são: (i)Todos-os-Usos - requer que todas as associações entre uma definição de variável e seus sub-seqüentes usos sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição, ou seja, um caminho onde a variável não é redefinida; (ii)Todos-os-Potenciais-Usos - procura explorar todos os possíveis efeitos a partir de uma mudança de estado do programa em teste, decorrente de definição de variáveis em um determinado nó (MALDONADO et al., 2004).

3.5.2.1 Ferramenta para Teste Estrutural

Na prática, a aplicação de um critério de teste está fortemente condicionada a sua automatização. O desenvolvimento de ferramentas de teste é de fundamental importância uma vez que a atividade de teste é muito propensa a erros, além de improdutiva, se

aplicada manualmente. Além disso, ferramentas de teste facilitam a condução de estudos empíricos que visam a avaliar e a comparar os diversos critérios de teste.

A ferramenta JaBUTi (*Java Bytecode Understanding and Testing*) foi definida dentro de uma tese de doutorado apresentada em (VINCENZI, 2004). É um ambiente completo para o entendimento e teste de programas e componentes Java, conforme ilustrado na Figura 3.4. A JaBUTi fornece ao testador diferentes critérios de teste estruturais para a análise de cobertura, um conjunto de métricas estáticas para se avaliar a complexidade das classes que compõem o programa, e implementa ainda algumas heurísticas de particionamento de programas que visam a auxiliar a localização dos erros.



Testing Criterion	Coverage	Percentage
All-Nodes-ei	25 of 29	86%
All-Nodes-ed	1 of 1	100%
All-Edges-ei	23 of 28	82%
All-Edges-ed	1 of 5	20%
All-Uses-ei	46 of 56	82%
All-Uses-ed	0 of 0	0%
All-Pot-Uses-ei	90 of 107	84%
All-Pot-Uses-ed	3 of 4	75%

Figura 3.4: Representação da diferentes visões da análise de cobertura de teste do JaBUTi.

A Figura 3.4 apresenta três tipos de visões para análise da cobertura de teste pela JaBUTi: (i) visão de código, apresentando os niveis profundidade das linhas de código, isto é, se o teste atingir determinada linha de código fez com que o teste também exercitasse as linhas de nível de profundidade inferior; (ii) visão de grafos, baseados nos conceitos dos grafos IG e DUG; (iii) cobertura por caso de teste, apresentando o percentual de cobertura de teste, isto, percentual do código que é exercitado pelo caso de teste. Considerando o suporte à análise de cobertura de programas Java, foram analisados quatro critérios de teste intra-métodos implementados pelo JaBUTi, sendo dois de fluxo de controle (Todos-Nós,Todas-Arcos) e dois critérios de fluxo de dados (Todos-Usos e Todos-Pot-Usos).

3.5.2.2 *Características de Ferramentas de Teste Java*

Para melhor justificar as escolhas das ferramentas JUnit e JaBUTi para apoiarem, automatizando, os testes funcional e estrutural foi realizada uma análise de algumas ferramentas de teste, resumida na Tabela 3.1 (adaptada de (VINCENZI, 2004)), especificas para código Java, uma vez que nos estudos de caso deste trabalho são focadas em aplicações para um microprocessador Java.

Analisando a Tabela 3.1, podem ser observados alguns pontos com relação as ferramentas de testes. Primeiramente, considerando as ferramentas que permitem a avaliação de cobertura de código por meio de critérios estruturais, observa-se que das ferramentas analisadas todas apóiam somente o teste de fluxo de controle (cobertura de comandos e decisões) em programas. Nenhuma delas apóia a aplicação de algum critério de fluxo de dados, seja para o teste de unidade, integração ou sistema. Além disso, exceto pela ferramenta GlassJAR e as que apóiam o teste funcional, todas as demais necessitam do código fonte para a aplicação dos critérios, dificultando a utilização das mesmas no teste estrutural de componentes de software por parte dos clientes, os quais, em geral, não têm acesso ao código fonte.

Outro ponto a ser destacado é o fato do *framework* (arcabouço) JUnit ser uma ferramenta que pode ser utilizada tanto para o teste de programas quanto para o teste de componentes de software desenvolvidos em Java, porém tal ferramenta suporta apenas a realização de testes funcionais, não fornecendo informação sobre a cobertura de código obtida por determinado conjunto de testes. Outra ferramenta que também apóia somente o teste funcional é a CTB (BUNDELL et al., 2000).

Tabela 3.1: Características das Ferramentas de Teste Java.

Ferramentas	*TU*	*TI*	*TS*	*CF*	*CFC*	*CFD*	*CM*	*EC*	*AD*	*TR*
PiSCES	X				X			X		
SunTest	X		X		X			X		
JProbe Suite	X		X		X			X	X	
Panorama	X				X			X		
TCAT/Java JCover	X				X			X		
CTB	X		X	X						
JTest	X		X	X	X			X		X
Glass JAR Toolkit	X		X	X	X			X		
Object Mutation Engine		X					X	X		
Mutation Testing System		X					X	X		
JUnit		X			X					
JaBUTi	X		X		X	X			X	X

onde:

- TU = Teste de Unidade
- TI = Teste de Integração
- TS = Teste de Sistema
- CF = Critérios Funcionais
- CFC = Critérios de Fluxo de Controle
- CFD = Critérios de Fluxo de Dados
- CM = Critérios de Mutação
- EC = Exigências de Código
- AD = Atividade de Depuração

- TR = Teste de Regressão

Além disso, considerando as ferramentas que apóiam o teste baseado em erros, observa-se que todas dão suporte ao teste de integração inter-classe. Embora o teste de integração seja de grande importância no contexto de programas orientados a objetos, considera-se que ainda assim, os testes de unidade devam ser realizados visando, principalmente, eliminar as falhas de lógica e de programação antes das unidades individuais serem integradas.

Com base nas considerações relacionadas ao teste estrutural foi apresentado um conjunto de critérios de teste que viabiliza a realização do teste de fluxo de controle e de dados intra-método em programas Java. Tais critérios derivam os requisitos de testes diretamente a partir de bytecodes Java, podendo ser utilizados tanto no teste de programas quanto de componentes desenvolvidos em Java. Para apoiar a aplicação desses critérios e suprir parte das deficiências apresentada por outras ferramentas destinadas ao teste estrutural de programas, a ferramenta JaBUTi foi selecionada na aplicação prática do processo proposto neste trabalho, conforme descrito no Capítulo 5.

4 TESTE DE PROCESSADORES EMBARCADOS

Processadores têm constituído um papel importante no desenvolvimento de circuitos digitais sendo, constantemente, os elementos centrais em muitos tipos de aplicações. Atualmente, os processadores são ainda mais importantes devido ao seu crescente uso em sistemas embarcados. Em arquiteturas SoC (*System-on-a-chip*), 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 importância do teste do processador é equivalente à 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 o mesmo irão apresentar mau funcionamento. Embora a falha exista apenas dentro do processador, é muito provável que o sistema inteiro seja comprometido 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 mau funcionamento. O mesmo 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.

Como a fabricação de circuitos integrados atualmente permite a implantação de sistemas complexos e que possuem cada vez mais um número maior de transistores, conseqüentemente, aumentando a dificuldade do teste dos sistemas eletrônicos. 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 meio de testar os demais módulos do sistema.

4.1 Métodos e Tipos de Teste de Processadores Embarcados

Desde a elaboração do protótipo de um circuito, o projeto é submetido a uma série de etapas de verificação, seguindo as diversas fases de seu desenvolvimento. A escolha do método de teste depende de vários fatores, tais como: custo, tempo para aplicação, testabilidade desejada e possibilidade de interrupção das funções do sistema (ABRAMOVICI; FRIEDMAN; BREUER, 1990).

Os métodos de teste são classificados conforme alguns critérios (ABRAMOVICI; FRIEDMAN; BREUER, 1990):

- Momento da aplicação do teste: pode ocorrer de forma concorrente à aplicação (teste *on-line* ou teste concorrente), ou como uma atividade independente (teste *off-line*);

- Estímulos de teste: podendo ser gerado no próprio sistema (auto-teste) ou através de um dispositivo externo (testador). Esses estímulos podem ser recuperados de uma memória (teste com padrões pré-calculados) ou gerados durante o processo de teste (teste algorítmico ou teste por comparação). A aplicação destes estímulos pode ocorrer em uma ordem fixa, pré-determinada ou dependente de resultados obtidos até o momento (teste adaptativo). A aplicação dos estímulos pode ocorrer em uma velocidade menor que a velocidade de operação normal do sistema (estático) ou na velocidade normal de operação do sistema (*at-speed*);

- Alvo do teste: depende da etapa de projeto que é aplicado, o qual tenta identificar erros de projeto (verificação de projeto), erros e defeitos de fabricação (teste de aceitação e burn-in), defeitos físicos imediatas (teste de qualidade) e defeitos físicos que ocorrem durante o uso do sistema (teste de campo ou teste de manutenção);

- Análise dos resultados obtidos: pode-se observar todas as saídas ou apenas algumas funções (teste compacto);

- Verificador de resultados: refere-se a quem verifica os resultados, podendo ser o próprio sistema, através do auto-teste ou auto-verificação, ou através de um dispositivo testador (teste externo), e

- Objeto do teste: o objeto de teste pode ser um circuito integrado (teste de componente), uma placa (teste de placa) ou um sistema com vários níveis de complexidade (teste de sistema).

Para verificar o comportamento de um circuito pode-se utilizar os tipos de teste tanto o funcional quanto o teste estrutural (BUSHNELL; AGRAWAL, 2000b), aplicando-se valores na entrada do mesmo e comparando a saída com o valor esperado, conceitualmente similar às técnicas de teste de software. O valor aplicado chama-se vetor de teste e define-se como cobertura de falhas o percentual de falhas detectadas pelos vetores de teste aplicados durante o teste, em relação ao total de falhas possíveis do circuito.

O teste funcional ignora a estrutura interna e identifica como um circuito sem falhas aquele que executa as funções a ele especificadas. Apresenta a possibilidade de ser executado de forma implícita, concorrente ou *on-line*, também chamado de checking, onde realiza testes durante a operação do sistema para a detecção de erros.

O objetivo do teste estrutural é verificar se a estrutura implementada fisicamente confere com a estrutura especificada no projeto. Chama-se estrutural, pois depende da estrutura do circuito (portas lógicas, transistores, por exemplo) e verifica esses elementos dentro do circuito, por isso pode apresentar coberturas de falhas elevadas, pois os vetores de teste são gerados para detectar falhas específicas dentro do circuito (ABRAMOVICI; FRIEDMAN; BREUER, 1990).

4.1.1 Modelo de Falhas

Os modelos de falhas são utilizados para representar defeitos físicos em um nível de abstração maior, geralmente em nível de porta lógica (BUSHNELL; AGRAWAL, 2000b). Portanto, um modelo de falhas é definido como a descrição abstrata dos efeitos de alguns defeitos ou de combinações de defeitos em um circuito. A principal função dessa abstração é reduzir a complexidade da análise de comportamento do circuito na presença de falhas (KRUG, 2007). As vantagens no uso de modelos de falhas no processo de teste referem-se à independência da tecnologia em uso e à cobertura do teste (LALA, 1997).

Na tentativa de representar defeitos físicos (WADSACK, 1978; GALAY; CROUZET; VERNIAULT, 1980; COURTOIS, 1981; RAJSUMAN, 1992) em dispositivos de hardware, surgiram vários modelos de falhas, entre eles: *stuck-at*, referindo-se às entradas e saídas de portas lógicas coladas em 1 ou 0, *stuck-on/open*, representando transistores que sempre ou nunca conduzem; *slow-to-rise* e *slow-to-fall*, referindo-se a portas com atraso de subida e descida, entre outros. Curtos entre fios (*bridging*), fios desconectados (*opens*) e atrasos cumulativos de portas e interconexões no caminho crítico (*path delay*) são exemplos de modelos de falhas de interconexão.

O modelo de falhas mais utilizado é o *stuck-at*. Neste modelo de falhas pressupõe-se que em um circuito os possíveis defeitos têm como efeito lógico colocar permanentemente um dado nó em valor 1 (*stuck-at* 1) ou 0 (*stuck-at* 0). O modelo de falhas *stuck-at* é adequado para modelar a maioria das falhas reais (ou defeitos) observadas nos circuitos.

Um processo comumente utilizado na preparação do teste de dispositivos de hardware é a simulação de falhas. Em um processo de simulação de falhas o circuito é acrescido de falhas, de acordo com um modelo, para que possa ser realizada a comparação dos resultados obtidos com os valores de uma simulação do circuito sem falhas. O processo de simulação de falhas tem como objetivo verificar se as falhas injetadas no circuito são detectadas através da aplicação de determinados estímulos de entrada.

4.1.1.1 Falhas de Atraso

Defeitos, como curto-circuito parcial ou circuito parcialmente aberto, resultam em falhas nos requisitos temporais do circuito sem quaisquer alterações em sua função lógica(LALA, 1997). 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 afetam o desempenho do circuito fazendo com que o seu atraso combinacional exceda o período de relógio. As 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, 2000b).

4.1.1.2 Falhas de Bridging

Podem ser modeladas tanto no nível lógico quanto no nível de transistores, as falhas de *bridging* representam um curto-circuito entre um grupo de sinais. Falhas de bridging podem ser uma falha de *input bridging* corresponde a um curto-circuito entre certo número de sinais de entrada primária; ou uma falha de *feedback bridging* ocorre se há um curto entre um sinal de saída e um sinal de entrada; ou ainda falha de *nonfeedback bridging* identifica uma falha de *bridging* que não pertence a nenhuma das categorias anteriores (LALA, 1997). Falhas de *input bridging* e *nonfeedback bridging* são combinacionais, e sua cobertura pelo teste de falhas *stuck-at* é normalmente alta (BUSHNELL; AGRAWAL,

2000b).

4.1.1.3 *Teste At-Speed*

Os vetores de teste são aplicados e as respostas observadas na mesma freqüência de operação normal do circuito (BUSHNELL; AGRAWAL, 2000b). A utilização de vetores de teste para falhas *sutck-at* em testes *at-speed* é uma estratégia freqüentemente aplicada, possibilita a detecção de algumas falhas de atraso, porém, geralmente, é incapaz de atingir uma cobertura de falhas muito alta para modelo de falhas de atraso. Nos casos em que uma alta cobertura de falhas de atraso se faz necessária, deve-se incluir ao *at-speed* padrões de teste para falhas de atraso de caminho (BUSHNELL; AGRAWAL, 2000b).

4.1.1.4 *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* (s-a-1) e *stuck-at-0* (s-a-0). Assim, uma linha com uma falha stuck-at-1 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, conhecido como o 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).

Os modelos de falhas mais adequados para o teste on-line de processadores são aqueles pertencentes ao nível lógico, ou RTL (*Register-Transfer Level*). No desenvolvimento deste trabalho, considerou-se apenas o modelo de falhas stuck-at simples. Entretanto o teste para falhas stuck-at detecta grande parte das falhas de *input bridging* e *nonfeedback bridging*. Também, algumas falhas de atraso também são cobertas, se a frequência da execução dos teste for a mesma do processador.

A metodologia de teste desenvolvida no presente trabalho não realiza teste *at-speed*, uma vez que um simulador do próprio processador é responsável pela aplicação dos padrões de teste e captura das respostas. Porém, o teste é executado na freqüência de operação do processador, já que é realizado um tipo de auto-teste no processador embarcado, conhecido como auto-teste baseado em software, uma vez que é reusado os casos de teste do software embarcado para testar o processador da aplicação. 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.

4.2 Auto-Teste em Processadores Emabarcados Baseado em Software

Processadores apresentam algumas características particulares que favorecem a aplicação de determinado método de teste *on-line* (de forma concorrente à aplicação). Por outro lado, quando são partes integrantes de sistemas embarcados, a aplicação de determinadas abordagens de teste torna-se impraticável devido às sérias restrições desses sistemas, por exemplos requisitos de tempo-real (MORAES, 2006).

Técnicas de projeto são aplicadas aos processadores 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, o desempenho máximo do processador é atingido sob a restrição de que o tamanho do circuito não exceda um limite especificado. Em outros exemplo, um fator que pode limitar 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 (MORAES, 2006).

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, num melhor caso, alterações periféricas no circuito e impacto marginal no desempenho e no consumo de energia (GIZOPOULOS; PASCHALIS; ZORIAN, 2004). A técnica de auto-teste baseado em software (SBST - *Software-Based Self-Test*) é bastante adequada para aplicação em processadores embarcados (MORAES, 2006). 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.

Auto-teste baseado em software (SBST) é uma metodologia alternativa ao auto-teste baseado em hardware (HBST -*Hardware-Based Self-Test*) que realiza o teste do processador usando suas próprias instruções (CHEN; DEY, 2001). Os tTipos de HBST mais conhecidos são o teste de varredura (*scan test*) e o BIST (*Built-In Self-Test*) (BUSHNELL; AGRAWAL, 2000b). O teste de varredura modifica os registradores (*flip-flops*) do sistema para que passem a possuir dois modos de operação: modo normal e de teste. O BIST é uma técnica que utiliza partes do circuito para testar internamente o restante do circuito. 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 pode ser observado na figura 4.1 (XENOULIS et al., 2003). 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).

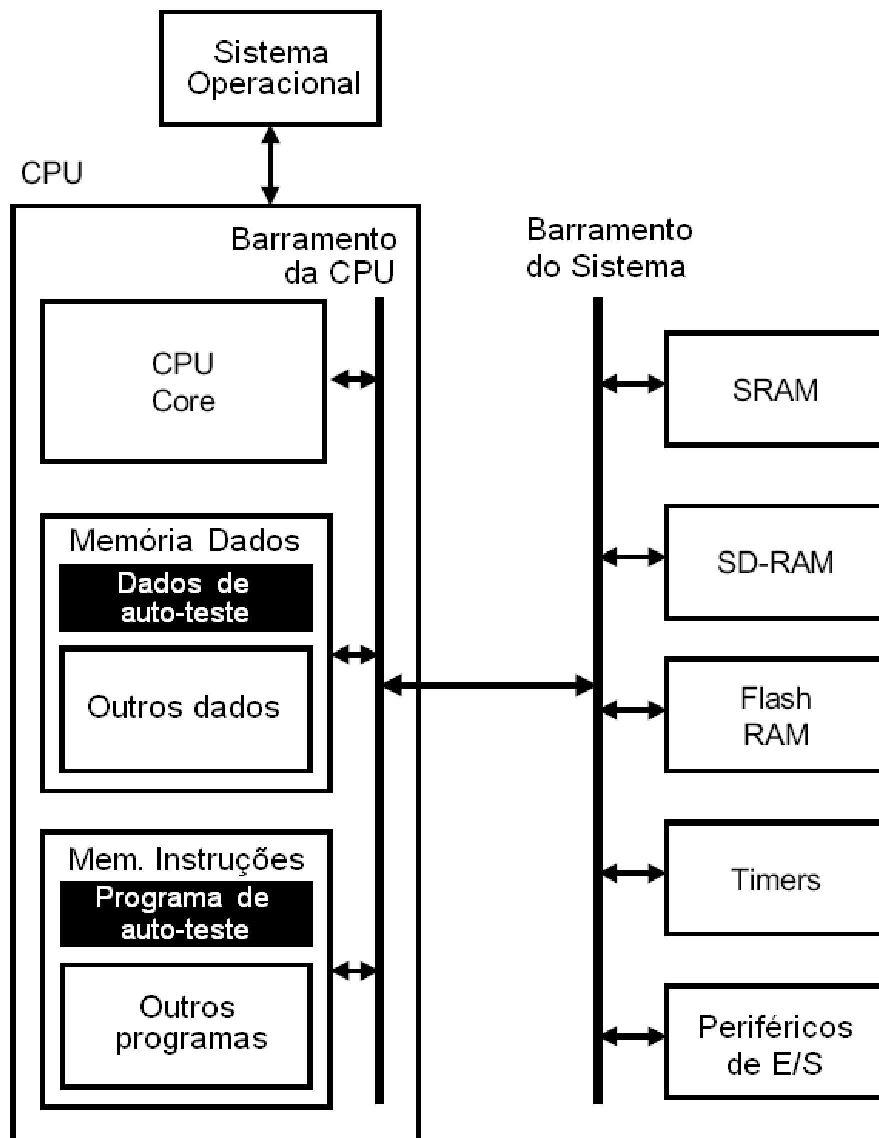


Figura 4.1: Conceito do auto-teste baseado em software.

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. 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.

A cobertura de falhas possível na técnica SBST são determinados 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). A qualidade do teste e a cobertura de falhas têm impacto direto no tempo de teste e no tamanho das memórias. Assim, todos esses fatores devem ser considerados no desenvolvimento do programa de teste mais adequado para dada aplicação alvo.

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. A principal dificuldade da estratégias SBST é encontrar um bom compromisso entre o esforço da geração do teste e a cobertura de falhas resultante. Porém, na abordagem SBST funcional, como o IRST (*Instruction Randomization Self-Test*) (BATCHER; PAPACHRISTOU, 1999), as instruções são escolhidos aleatoriamente para compor o programa teste. Isso garante alto nível abstração e baixo custo geração, mas exige um grande número de instruções para atingir uma elevada cobertura de falhas, pelo tempo de teste necessário. Na abordagem SBST estrutural as instruções de teste são deterministicamente selecionados com base na cobertura de falhas de cada bloco (PASCHALIS et al., 2001), assim com alto custo de geração e cobertura falhas, assim um programa de teste menor pode ser definido reduzido tempo de teste.

Em um sistema embarcado, apesar de hardware e software dependem fortemente uns dos outros, o teste do software e do hardware são geralmente tarefas separadas. O teste de cada parte compreende a seu próprio custo de geração e aplicação. No entanto, processadores embarcados implementam um conjunto de instruções necessárias reduzido, para uma única ou um pequeno grupo de aplicações. Assim, neste trabalho foi investigado o quanto interessante é considerar um teste SBST funcional do processador, reutilizando a aplicação executado no processador e os seus casos de teste. Tal integração apresenta potencial de reduzir teste geração e aplicação dos custos.

5 REUSANDO CASOS DE TESTE DE SOFTWARE PARA TESTE DE MICROPROCESSADOR

O método proposto neste trabalho tem como base uma abordagem de teste integrado de software e hardware para processadores embarcados. Um método que visa o teste para os sistemas em que o hardware é projetado para executar instruções de um determinado software, isto é, uma versão ASIP (*Application Specific Instruction set Processor*) de um processador, onde sua unidade de controle possui apenas as instruções usadas por aquele programa específico. O principal objetivo é atingir um percentual de reuso dos casos de testes projetados para o teste do software no teste de hardware, assim tendo o teste integrado de software e hardware (MEIRELLES; COTA; LUBASZEWSKI, 2008). Tal abordagem é definida e detalhada na próxima Seção 5.2.

Uma vez que o processador (hardware) será gerado a partir do código do software, foi avaliado um mecanismo de verificação desse hardware através dos testes desenvolvidos para verificar e validar o software. Para verificar o software são usadas as ferramentas de testes que permitem aplicar as técnicas e observar os critérios de teste de software. Porém, para validar esse software é necessário executá-lo no ambiente em que será operado. Neste ponto do fluxo de projeto o hardware ainda não está pronto, além disso, se quer integrar os testes. Então, a integração deve ser viabilizada por um ambiente de simulação "híbrido" da plataforma alvo, o que permite a validação do software e verificação do hardware quando os mesmos forem submetidos às condições de falhas.

5.1 Trabalhos Relacionados

Teste funcional tem sido estudado como solução há algum tempo para testes de processadores (BATCHER; PAPACHRISTOU, 1999; CORNO et al., 2001; **?**; HENTSCHKE et al., 2006; BECK FILHO et al., 2005). Nessas técnicas, geralmente, um conjunto de instruções ou um grupo de instruções é selecionado para definir um programa de teste. O teste é aplicado pela execução desse programa e posteriormente avalia resposta armazenada na memória do processador (HENTSCHKE et al., 2006; BECK FILHO et al., 2005). Na técnica de auto-teste baseado em software (SBST) funcional, o conjunto de instruções do processador é usado para gerar um programa de auto-teste composto por uma seqüência de instruções randômicas.

Cada técnica tem um baixo custo de desenvolvimento devido a seu alto nível de abstração. Entretanto, é necessário um grande número de instruções para garantir uma alta cobertura de falhas. Técnicas de SBST estrutural, aplicadas em componentes de processadores, são proposta em (MORAES et al., 2005). Nessa abordagem, uma rotina de auto-teste é desenvolvida para cada componente identificado no processador embarcado.

As rotinas de teste geram ou salvam os padrões de teste para todos os componentes sob teste, focando as falhas estruturais. As rotinas de auto-teste constituem justamente um programa de teste que é executado periodicamente. Embora essa abordagem tenha custos mais elevados do que o SBST funcional, necessita de menos instruções para atingir uma alta cobertura de falhas.

Em (BECK FILHO et al., 2005) demonstrou que um processador Java (arquitetura baseada em pilha) requer muito menos instruções aleatórias que um processador com arquitetura baseada em *load-store* para ser testado. Isso porque, uma vez que todas as operações de um processador baseado em pilha usa a mesma estrutura – a pilha – assim, todas as instruções no programa de teste contribuem para o teste. Dessa forma, o esforço de teste para essas arquiteturas podem ser reduzidas. No entanto, um programa de teste ainda deve ser gerado apenas para se obter uma alta cobertura de teste.

Em suma, neste trabalho assumiu-se uma estratégia de SBST funcional como caso base. De tal modo que é proposto um método para reduzir ainda mais o esforço de geração dos testes ao usar o próprio software e seus casos de teste para verificar o processador.

5.2 Abordagem de Teste Integrado de Software e Hardware

O uso de técnicas da engenharia de software para aumentar a testabilidade do hardware foi amplamente discutido em (KRUG; MORAES; LUBASZESKI, 2006; KRUG, 2007) no contexto em que essas técnicas são aplicadas em uma linguagem de descrição de hardware (HDL - Hardware Description Language), ou seja, no alto nível de abstração. No entanto, o referido trabalho, conclui que tão somente aplicar de maneira direta as técnicas de teste de software não garante boa cobertura de falhas, visto que, no nível onde são aplicadas estas técnicas não existem detalhes da implementação física do dispositivo de hardware. Então demonstrou que ao adaptar e combinar estas técnicas com ATPG (*Automatic Test Pattern Generator*) e DfT (*Design for Testability*) no nível de porta lógica, é possível aumentar a cobertura de falhas, reduzindo o tempo de preparação do teste. Ambas as técnicas mencionadas não são tratadas e nem abordadas neste trabalho. Entretanto, no método proposto pretende-se atingir uma cobertura de falhas satisfatória com aplicação direta e reuso dos testes de software.

Em um hardware projetado para executar instruções de um determinado software, isto é, um sistema embarcado composto de um programa que será executado em um hardware dedicado, é factível um estudo que vise explorar melhores técnicas de teste apresentadas na engenharia de software de maneira que integre o teste do software embarcado com o intuito de também detectar falhas no hardware, além da detecção dos erros e falhas de software. Essa abordagem é definida neste trabalho como "Teste Integrado de Software e Hardware"(MEIRELLES; COTA; LUBASZEWSKI, 2008).

Para simplificar o entendimento da abordagem, a Figura 5.1 (MEIRELLES; COTA; LUBASZEWSKI, 2008) ilustra uma visão e propõe uma equação para o teste de um sistema embarcado (Temb) como sendo a soma do custo do teste do software (Tsw) e do teste do hardware (Thw).

O teste integrado visa diminuir o custo total do teste do sistema embarcado, o que diminui os custos do teste do hardware que compõe esse sistema, através da aplicação dos mesmos testes definidos para o software para cobrir um número "X" (custo de cobertura de teste) de falhas no hardware. Dessa maneira, o teste de um sistema embarcado (Temb) fica igual à soma do custo do teste de software (Tsw) mais o custo do teste de hardware (Thw), menos o número "X" de falhas de hardware coberta pelo teste integrado (Ti), ou

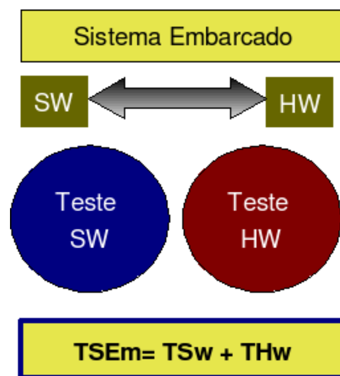


Figura 5.1: Equação simplificada dos custos do teste do sistema embarcado.

seja, teste de software que cobriu as falhas de hardware. Assim a nova visão e equação para o teste de um sistema embarcado é demonstrada pela Figura 5.2 (MEIRELLES; COTA; LUBASZEWSKI, 2008).

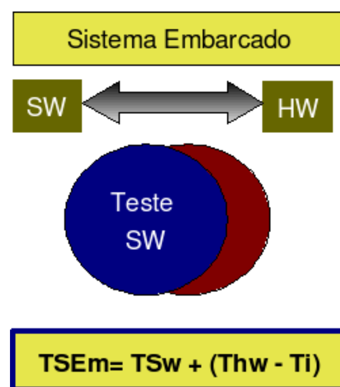


Figura 5.2: Equação dos custos do teste do sistema embarcado com teste integrado.

Com o reuso do teste de software para o teste do hardware, se obtém um ganho que compense os custos da aplicação do teste integrado, desde que se conheça quais partes do hardware não foram cobertas pelo teste integrado para que se aplique uma técnica de teste de hardware apenas nessas partes.

5.3 O Método de Teste Integrado de Software e Hardware

O método de teste integrado proposto neste trabalho testa o software sob o ponto de vista do funcionamento lógico (denominado aqui de teste lógico) num ambiente de teste de software convencional, apoiado pelas ferramentas adequadas de acordo com a tecnologia e linguagem utilizada. Em seguida, pretende-se reaplicar os mesmos testes na aplicação "sintetizada" para a plataforma alvo. Em outras palavras, os objetivos são:

- verificar se os mesmos casos de teste aplicados no ambiente de execução (simulação) são capazes de detectar falhas adicionais e falhas de hardware, relacionadas à operação do software na plataforma alvo;
- em caso negativo, definir novos casos de teste específicos para erros visíveis apenas no ambiente de execução;

- caso positivo, analisar se todos os casos de teste projetados precisam ser embarcados junto ao software para detectar as falhas de hardware.

5.3.1 Modelo Conceitual

Um modelo conceitual do método de teste integrado foi definido, abstraindo os passos e etapas independente da plataforma-alvo selecionada, demonstrando como aplicar quando se trata de um projeto ASIP. O método de teste integrado de software e hardware tem duas etapas na fase denominada de "pré-síntese" e até três etapas na segunda fase chamada de "ASIP" (depois da geração do ASIP), num total possível de cindo etapas. A Figura 5.3 representa o fluxo proposto.

Definidos os testes, de acordo com uma aplicação já conhecida, são aplicadas as técnicas de teste funcional e estrutural, observando os critério de cobertura definidos na engenharia de software. Essa é a etapa "b" que visa testar o software antes da síntese (pré-síntese) para ASIP. Os primeiros resultados indicarão o número de casos de teste e a cobertura dos testes, de acordo com os critérios dessas técnicas.

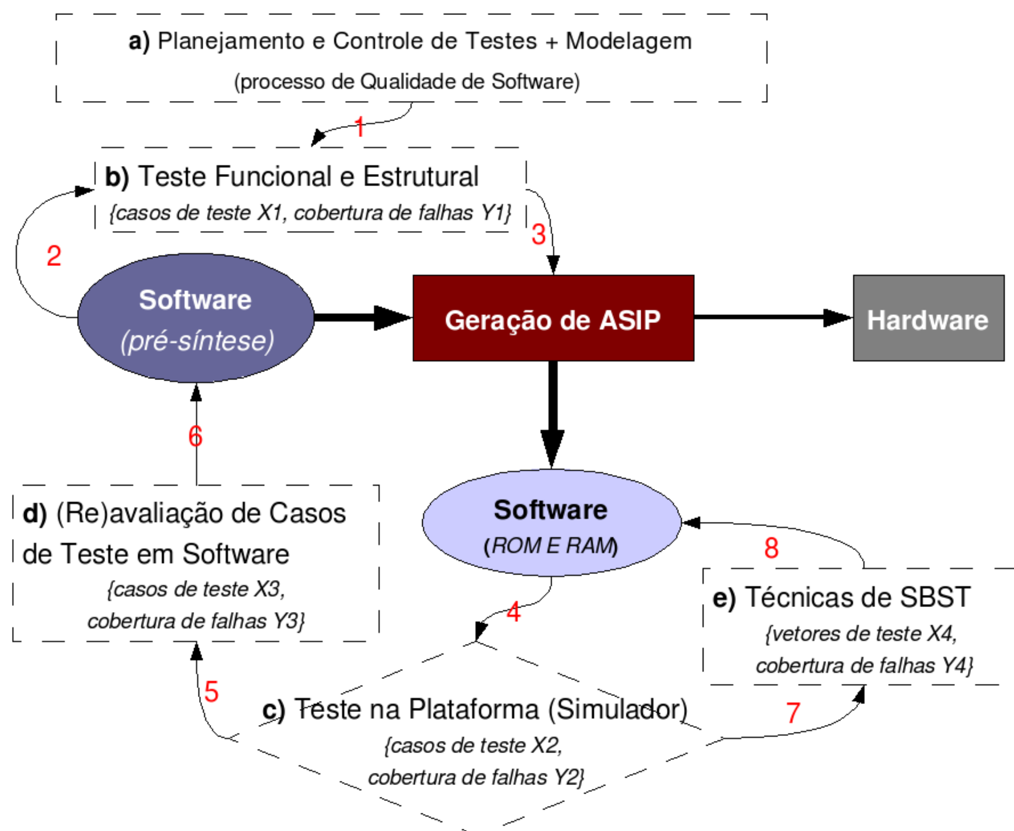


Figura 5.3: Fluxo das etapas do método de teste integrado de software e hardware.

O passo seguinte é a geração do ASIP, assim, além do hardware, se tem os dados na RAM e o programa na ROM. Parte-se para a etapa "c", o que consiste em aplicar os testes executados no software de pré-síntese quando o software estiver embarcado na plataforma. Neste ponto é usado um simulador que permita validar o software do ponto de vista de suas funcionalidades na plataforma e verificar o hardware, o que constitui um ambiente híbrido e integrado de testes. No hardware são injetadas falhas e observada a cobertura de falhas detectadas pelos casos definidos para o teste do software, caracterizando o reuso dos testes.

De acordo com a cobertura de falhas atingidas são consideradas duas possibilidades. A primeira (etapa “d”) é uma reavaliação dos casos de teste de software, mas levando em conta também quais partes do hardware não foram cobertas pela primeira rodada de teste, isso observado no ambiente de simulação integrado. Outra alternativa é complementar os testes acrescentando rotinas SBST, o auto-teste baseado em software, através de uma metodologia que aplique auto-teste no hardware.

5.3.2 Modelo Aplicado

A qualidade e produtividade da atividade de teste são dependentes dos critérios de teste utilizados e da existência de ferramentas que apoiem a aplicação dos testes. Sem automatizar, os testes tornam-se uma atividade propensa a erros e limitada a programas muito simples (PÁDUA, 2003). Assim, as ferramentas escolhidas para apoiar o método de teste integrado dependem da plataforma escolhida para o desenvolvimento do sistema embarcado. Este trabalho foi validado em estudo de casos com o processador FemtoJava (ITO; CARRO; JACOBI, 2001), apresentado na próxima seção.

5.3.2.1 Processador FemtoJava

O processador FemtoJava (ITO; CARRO; JACOBI, 2001) é o resultado de uma metodologia adotada para a geração semi-automática de um sistema embarcado a partir de uma descrição Java. Foi criado com restrições de área e potência visando especificamente sistemas embarcados.

O FemtoJava multiciclo implementa um subconjunto de 68 instruções Java. Neste subconjunto encontram-se instruções necessárias para operações básicas de pilha, manipulação de vetores, desvios condicionais e incondicionais, execução de métodos estáticos e acesso a campos de classes. A execução de instruções de entrada e saída (E/S), programação de interrupções e também para colocar o processador em modo suspenso ocorre por *bytecodes* estendidos.

O FemtoJava multiciclo não aloca objetos dinamicamente porque seu conjunto de instruções apenas suporta determinados *bytecodes* como: *invokestatic, return e ireturn* como instruções para manipulação de métodos, executando apenas o código de classes. Sua organização de memória é baseada na alocação de *frames* como manda a especificação da linguagem Java, não suportando *multithreading*. A implementação do sistema de E/S é mapeada em memória.

Outras características do FemtoJava são o conjunto reduzido de instruções, arquitetura *Harvard*, pequeno tamanho e facilidade de inserção e remoção de instruções. A microarquitetura do Femtojava multiciclo é ilustrada na Figura 5.4. Além dessas características, o tamanho da máquina de controle é diretamente proporcional ao número de instruções utilizadas, gerado a partir de um ambiente de CAD (*Computer Aided Design*), denominado SASHIMI (*System As Software and Hardware In MIcrocontrollers*) (ITO; CARRO; JACOBI, 2001).

Posteriormente ao multiciclo, foi desenvolvida uma versão *pipeline* desse microprocessador, também denominado FemtoJava Low-Power (BECK FILHO; CARRO, 2003; GOMES; BECK FILHO; CARRO, 2004), possuindo um *pipeline* de cinco estágios: busca de instruções, decodificação, busca de operandos, execução e escrita de resultados, conforme demonstrado na Figura 5.5 (BECK FILHO, 2004).

De acordo com (BECK FILHO, 2004), melhorou-se a velocidade de execução de programas Java e através de otimizações da microarquitetura do FemtoJava, como o uso da técnica de *forwarding* (HENNESSY; PATTERSON, 2003), os acessos para escrita no

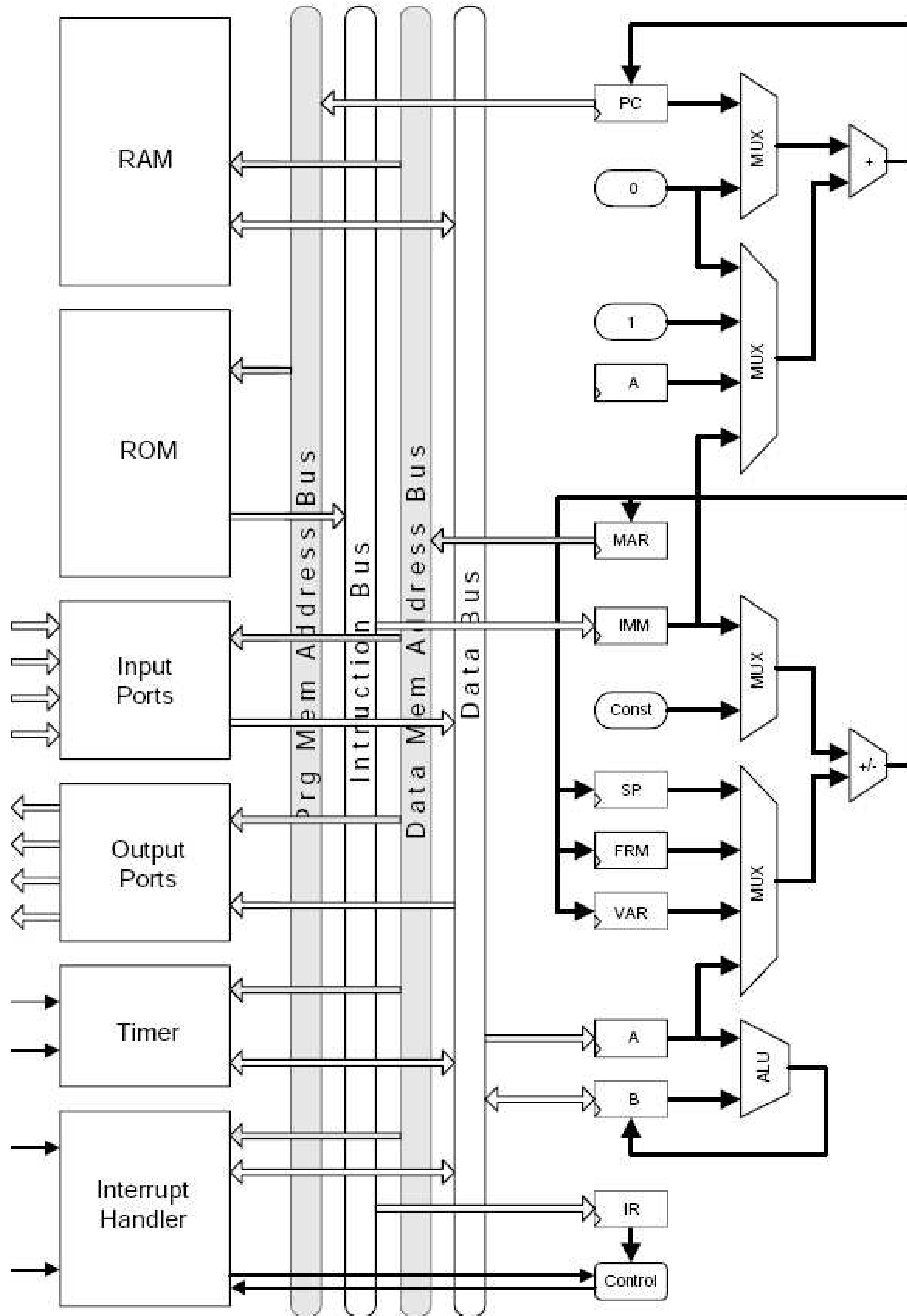


Figura 5.4: Microarquitetura do FemtoJava Multiciclo.

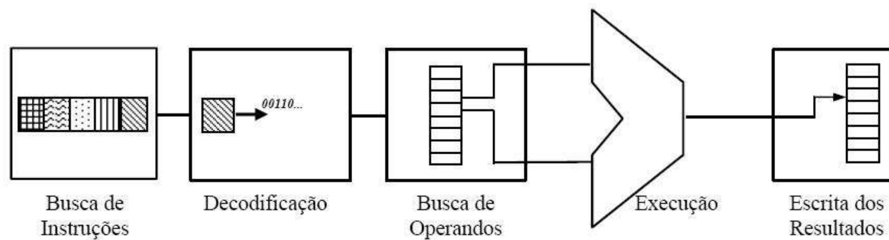


Figura 5.5: Estágios FemtoJava *pipeline*.

banco de registradores onde está localizada a pilha podem ser reduzidos em média de 70% em aplicações típicas de sistemas embarcados, pelo fato do processador ser baseado

em uma máquina de pilha. Assim, uma das principais características da versão *pipeline* 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.

Uma outra característica peculiar do FemtoJava pipeline, que foi observada durante a análise da arquitetura neste trabalho, é a existência de instruções com múltiplos ciclos de execução dentro do pipeline. Essa característica é normalmente esperada em arquiteturas do tipo multiciclo e sua adoção dentro de uma arquitetura de pipeline exige uma máquina de controle mais sofisticada, que permita "travar" o fluxo de execução das instruções adjacentes por um certo número de ciclos enquanto a instrução atual gera vários sinais diferentes para todos os estágios a partir de um determinado ciclo. Esse comportamento é decorrente da existência de instruções complexas, definidas na especificação da máquina virtual Java (SUN-MICROSYSTEMS, 1999), que necessitam de uma seqüência de sub operações para serem realizadas.

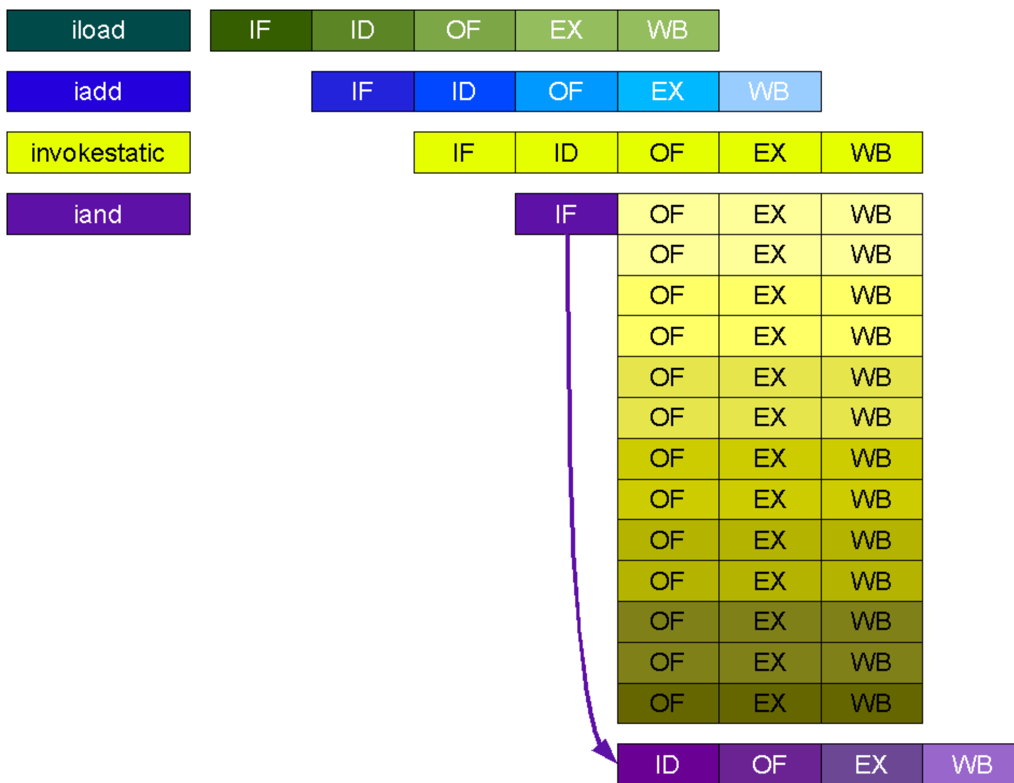


Figura 5.6: Exemplo de Ciclo e Estágios das Instruções do FemtoJava.

A Figura 5.6 ilustra um trecho de execução de instruções no processador FemtoJava pipeline onde se tem quatro instruções enviadas para execução. Pode-se observar que as instruções "iload" e "iadd" são enviadas e avançam normalmente por todos os estágios do pipeline (considerando que não exista uma dependência de dados entre elas). Na seqüência a instrução *invokestatic* é "buscada" e enviada para o estágio de decodificação. Neste momento a parte de controle detecta que se trata de uma instrução complexa e que necessita de várias "micro-instruções" para ser executada. Para tanto, é ativada uma máquina de estados que terá a função de controlar a geração de novos sinais a cada ciclo, fazendo com que as instruções subseqüentes fiquem "congeladas" enquanto a instrução complexa é decomposta em "micro-instruções" e flui ciclo após ciclo através do *pipeline*.

O diagrama apresentado na Figura 5.7 representa a máquina de estados da instrução "invokestatic" implementada na parte de controle para realizar a geração dos sinais que

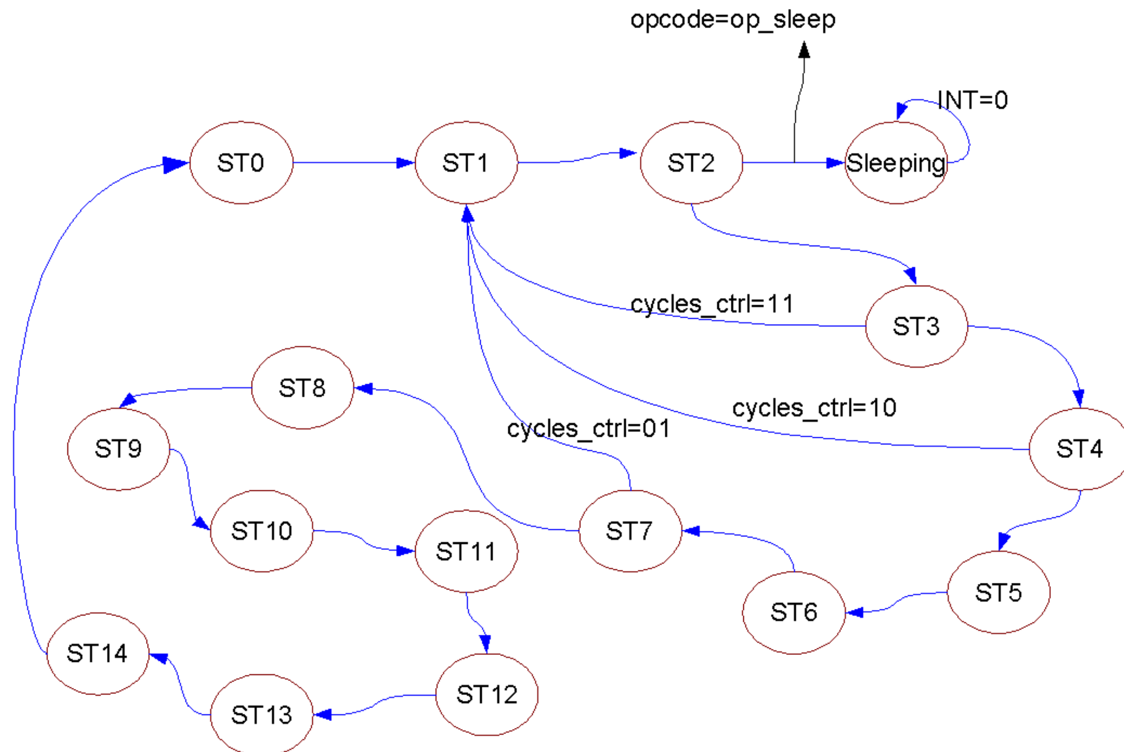


Figura 5.7: Máquina de Estado das Instrução Invokestatic do FemtoJava.

determinam o comportamento do *pipeline*. Esse comportamento pode ser observado na Figura 5.7, isto é, são necessários 14 ciclos para a execução do "invokestatic", isso vai justificar o aumento do número de ciclos da aplicação que contenha os testes, definidos pelos critérios de teste funcional e estrutural da engenharia de software, embutidos no software embarcado, de acordo com o que é proposto para o teste integrado de software e hardware. Isso porque, cada caso de teste corresponde a um método da implementação da aplicação em Java, de tal forma que para cada chamada de método é executada a instrução "invokestatic".

5.3.2.2 SASHIMI

O SASHIMI é um ambiente destinado à síntese de sistemas *microcontrolados* especificados em linguagem Java. O ambiente SASHIMI utiliza as vantagens da tecnologia Java e fornece ao projetista um método simples, rápido e eficiente para obter soluções baseadas em hardware e software para microcontroladores. O conjunto de ferramentas disponível no SASHIMI também foi desenvolvido inteiramente em Java, tornando-o portável para diversas plataformas (ITO; CARRO; JACOBI, 2001)

O projetista pode modelar, simular e construir uma implementação do sistema diretamente em Java, já que o SASHIMI disponibiliza bibliotecas e permite um mapeamento direto das classes usadas para simulação para o código da implementação final. A partir do código gerado pelo compilador Java, uma versão ASIP do FemtoJava é gerada, onde sua unidade de controle possui apenas as instruções usadas por aquele programa em específico. O fluxo completo do SASHIMI pode ser observado na Figura 5.8 (ITO; CARRO; JACOBI, 2001).

As demais versões do Femtojava foram baseadas na versão multiciclo, implementando o mesmo conjunto de instruções, bem como projetadas de forma que elas sejam fa-

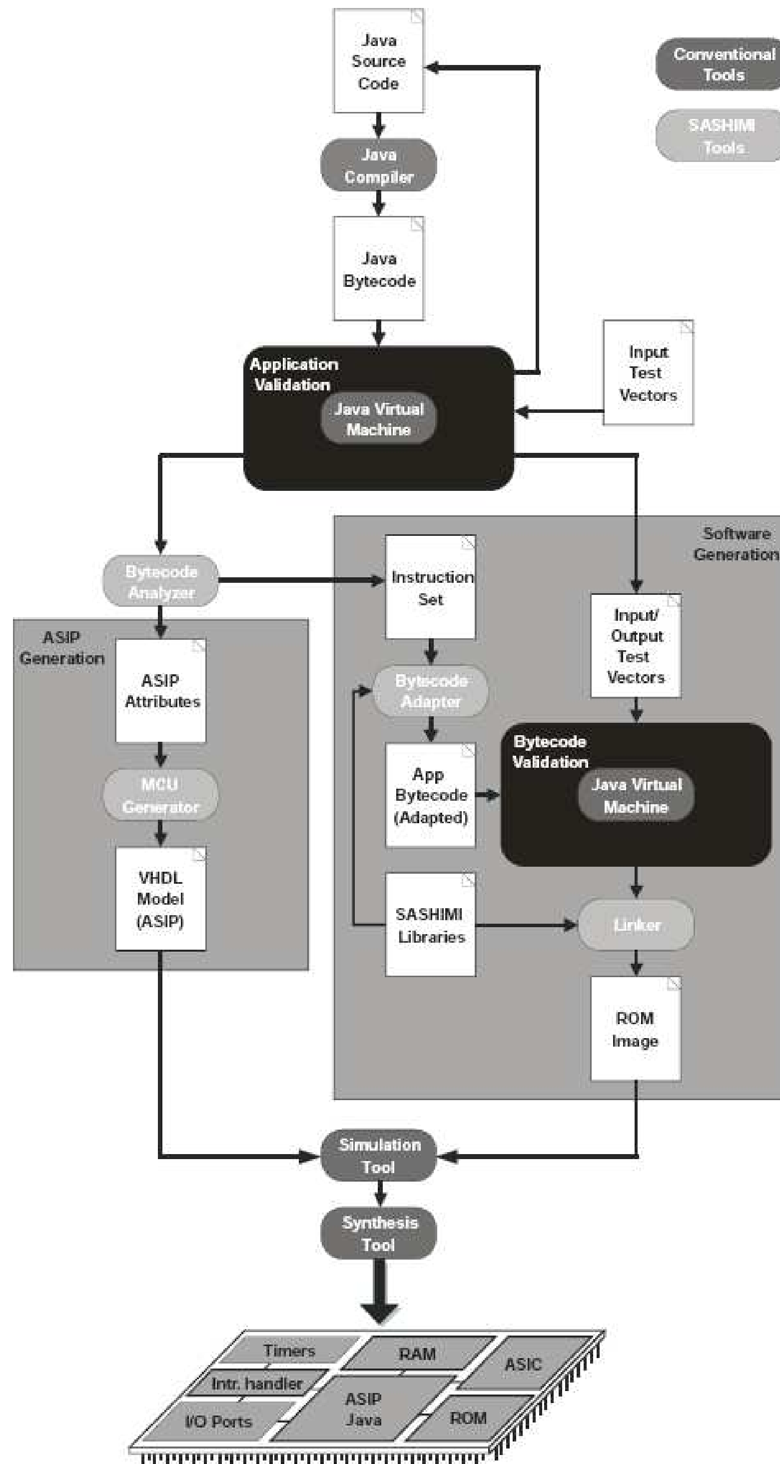


Figura 5.8: Fluxo de Projeto no SASHIMI.

cilmente parametrizáveis na ferramenta SASHIMI. Portanto, nos estudos de caso prático do método de teste integrado de software e hardware, o SASHIMI refere-se ao componente central entre a fase de pré-síntese e a fase ASIP, constituindo o “passo 3” do modelo conceitual, antes de iniciar o teste no simulador que permite o teste integrado.

5.3.2.3 *CACO-PS: Cycle-Accurate COnfigurable Power Simulator*

Com objetivo de fazer a análise de potência do FemtoJava *pipeline* optou-se por uma ferramenta em um alto nível de abstração para ser utilizada no início do ciclo de projeto do sistema embarcado. O CACO-PS (*Cycle-Accurate COnfigurable Power Simulator*) (BECK FILHO et al., 2003) é um simulador de código compilado, que calcula a potência baseado em ciclos de relógio. Ele oferece a possibilidade da descrição estrutural de qualquer arquitetura, sendo de propósito geral. Além disso, é possível descrever a arquitetura em diferentes níveis de abstração, conforme o nível de detalhamento desejado pelo projetista, não exigindo a descrição do sistema em RTL (*Register Transfer Level*).

Outra característica do CACO-PS é a possibilidade de extensão de suas funcionalidades para suportar injeção de falhas do tipo stuck-at nas arquiteturas descritas usando sua sintaxe. O simulador, além de inserir as falhas, pode detectar, em locais especificados pelo usuário, se esta falha foi propagada ou não.

Extensões para testes e injeção de falhas foram implementadas no CACO-PS para aplicar métodos de teste funcional em microprocessador. O primeiro método usa algoritmos genéticos, bibliotecas de macros, como gerador dos programas de teste (HENTSCHKE et al., 2006). Outro método usa instruções aleatórias para gerar os programas de teste, detectando as falhas ao verificar toda a memória, comparando com os valores sem falhas, e o valor do PC (BECK FILHO et al., 2005). Assim, o CACO-PS refere-se ao ambiente de simulação híbrido, de acordo com o modelo conceitual do proposto neste trabalho, o que permite a integração entre a validação do software e a verificação do hardware (ambiente da etapa “c” do modelo conceitual). A análise dos custos desses métodos é apresentada na seção que são descritos os estudos de caso e os resultados.

Para o funcionamento geral do simulador, três arquivos devem ser passados ao CACO-PS:

- O que descreve, através de uma sintaxe própria e estruturada, a arquitetura desejada;
- O que descreve os componentes funcionais, que serão instanciados pela descrição da arquitetura;
- O arquivo que agrega uma função de cálculo de potência para cada componente.

Na simulação do FemtoJava, arquivos de imagem da memória de dados e memória de programa (ou instruções) são informados como parâmetro ao simulador. Essas memórias são do formato “.MIF”, que é um dos formatos de memória utilizado pelos programas de VHDL (*VLSI hardware description language*). Esses arquivos são um conjunto de linhas, onde em cada uma há um endereço de memória e depois seu valor. Também a opção de carregar um arquivo que contém todas as instruções do FemtoJava, cada qual com o seu *opcode* e seus *microopcodes* para cada estado e seu nome, foi incluída como uma característica específica do CACO-PS.

Quando a simulação está em modo teste, utilizando a extensão de injeção de falhas, primeiramente todo o sistema é simulado, e ao final de sua execução, os valores corretos de todos os sinais e da memória são guardados, sendo os valores de “ouro” para comparação futura. Também o número de ciclos é armazenado para ser um dos mecanismos de interrupção da simulação.

Deste ponto em diante, uma simulação inteira é feita novamente para cada falha injetada. A falha, do tipo *stuck-at*, é injetada em todos os *bits* de todos os sinais da arquitetura passada ao CACO-PS. Inicia-se do primeiro *bit* do primeiro sinal declarado sempre em

zero (*stuck-at-0*), posteriormente declarando sempre em zero o segundo *bit* dos sinais e mantendo o primeiro com seu valor original. O procedimento é repetido para todos os *bits* de todos os sinais até o último *bit* do último sinal ser atingido e simulado. O mesmo ocorre para *stuck-at-1*. Caso tenha dois mil *bits* nos sinais da arquitetura serão injetadas quatro mil falhas e quatro mil simulações serão realizadas. Ao final de cada simulação os valores da memória “ouro”, armazenados na primeira rodada, são comparados com a memória da simulação com falhas.

A comparação entre as memórias da simulação sem falhas e com das simulações com falhas também depende do método usado. Por exemplo, o método de teste funcional usando instruções randômicas (BECK FILHO et al., 2005) percorre toda a memória e verifica todos os endereços, comparando os valores entre a memória ouro e a memória com falha. A detecção das falhas no caso do método de teste integrado de software e hardware não faz comparações entre as memórias, apenas em pontos específicos da memória com falhas são observados de acordo com o número de casos de testes embutidos no código do software embarcado. Assim, caso sejam selecionados três casos de testes, apenas três pontos da memória serão comparados, pois correspondem a variáveis no programa que indicam se o caso de teste detectou ou não a falha.

Finalizada toda a simulação, o simulador mostra em uma tabela se a falha de cada *bit* propagou ou não para a memória, ou para algum sinal em específico, escolhido pelo usuário. Uma listagem dos *bits* dos sinais que não foram detectados também é informada, com isso, sendo possível ao testador de hardware focar a construção dos vetores de teste, em caso de querer complementar os testes por técnicas de teste de hardware.

5.3.3 Ferramentas por Etapas do Método de Teste Integrado

Na primeira fase do método de teste integrado de software e hardware – pré-síntese – duas ferramentas auxiliam as atividades de teste: Junit e JABUTi, descritas ao longo do Capítulo 3. A Figura 5.9 (MEIRELLES; COTA; LUBASZEWSKI, 2008) demonstra o fluxo da utilização dessas ferramentas conforme o modelo conceitual e a seleção das ferramentas de acordo com características da plataforma-alvo escolhida, FemtoJava. Nessa primeira fase são realizados os testes funcionais e estruturais do software, baseados nos critérios da engenharia de software. O *framework JUnit* possibilita a aplicação de testes unitários para código Java, inclusive código escrito de acordo com as restrições para o FemtoJava, assim as funcionalidades do software são verificadas de acordo com a especificação definida anteriormente. Como o JUnit não informa a cobertura dos testes, apenas indicando se o software passou ou não no conjunto de casos de teste, uma avaliação da cobertura dos testes de acordo com o critérios de teste estrutural é realizada através da JaBUTi, importando os mesmos casos de teste definidos no JUnit. No caso da cobertura não ficar próxima aos 100% novos casos de teste devem ser definidos.

Figura 5.9: Ferramentas utilizadas no fluxo do teste integrado de software e hardware.

Antes da geração do ASIP, baseado no código Java da aplicação (para o FemtoJava), uma análise feita pelo testador é realizada, de acordo com o percentual de cada caso de teste indicado pela JaBUTi, para selecionar quais métodos serão inseridos nas classes Java do software, de modo que os mesmos testes possam ser executados no simulador da plataforma. Essa análise é justificada pelo fato de cada chamada ao caso de teste implicar em uma chamada de método no FemtoJava. Isso, como explicado anteriormente em relação à instrução *invokestatic*, gera um aumento no número de ciclos na execução da aplicação além do esperado. Assim, existindo casos de teste que tenham coberturas parcialmente redundantes, um deles pode deixar de ser embarcados junto com a aplicação.

O ambiente SASHIMI gera um ASIP do FemtoJava, de acordo com a implementação do software e dos casos de teste, fornecendo as memórias de programa e de dados (RAM e ROM), em formato MIF, que são as entradas referentes ao software embarcado para o simulador CACO-PS, descrito na seção anterior. Nesse ponto chega-se a fase ASIP do método proposto de teste integrado de software e hardware, partindo para a etapa "c" do modelo conceitual (Figura 5.3). O SASHIMI, ao final da geração do ASIP, informa o tamanho do programa (ROM), tamanho da memória (RAM), número de instruções e número de *opcodes* diferentes. Esses dados permitem comparar esses custos da aplicação com casos de teste e sem casos de testes.

No CACO-PS é possível simular o software embarcado em um ambiente com as mesmas características do local de execução final, possibilitando validar em condições próximas às restrições do sistema embarcado. Uma primeira execução, sem ativar o módulo de injeção de falhas no hardware do CACO-PS, é realizada para verificar se o resultado das saídas dos casos de testes são os mesmos observados no JUnit e JaBUTi. Isso significa validar a execução dos casos de teste e obter o número de ciclos da execução do software com os testes, que será comparado com o número de ciclos da execução sem casos de teste.

Por fim, executa-se o CACO-PS com módulo de injeção de falhas e testes habilitados de modo que constitui-se o ambiente integrado, permitindo a validação do software embarcado e a verificação do hardware de acordo com o modelo de falhas *stuck-at*. Ao final da simulação, conforme foi explicado na seção anterior que descreveu o funcionamento do CACO-PS, é informado o número de falhas injetadas, a cobertura de falhas provida pelos casos de teste definidos para o software na fase de pré-síntese, detalhando a cobertura da parte de controle e de dados do processador, assim como o número de bits de controle, dados e portas. Também são informadas quantas falhas atingiram o contador de programa (PC - *Program Counter*) e quantas falhas foram detectadas antes ou depois do número de ciclos máximo da aplicação.

5.4 Estudos de Caso e Resultados

O método de teste integrado de software e hardware proposto neste trabalho foi validado em duas aplicações desenvolvidas para FemtoJava, uma de fluxo de dados (*data flow*) e outra de fluxo de controle (*control flow*). A primeira é um algoritmo de ordenação (*bubble sort*) com dez elementos de entrada. Para esta aplicação a cobertura de falhas é influenciada pelos dados que serão ordenados. Neste caso, empiricamente constatou-se que é possível ter uma melhor cobertura quando entre os valores existem números negativos, assim aumentando a quantidade de "1s" (uns) presentes nos sinais do processador. Um caso médio, por exemplo, foi considerado de 5 negativo a 5 positivo. Os resultados da cobertura de falhas para o algoritmo de ordenação é apresentado nas Tabelas 5.1 e

5.2, com os experimentos executados nas versões multiciclo e *pipeline* do processador, respectivamente. Em ambas as tabelas, a primeira coluna mostra o intervalo dos dados a serem ordenados. A segunda coluna mostra o número de ciclos para executar o teste. A terceira coluna apresenta a cobertura de falhas e a quarta coluna indica a cobertura de falhas que atingiram o contador de programa sozinho, ou seja, o número de falhas que afetam o PC (*Program Counter*).

Tabela 5.1: Cobertura de falhas de hardware com casos de testes de software no Femto-Java multiciclo com aplicação de fluxo de dados.

Bubble10	*Ciclos*	*Cobertura*	*PC*
tudo 0	12.406	84,90%	57,88%
tudo 1	12.406	85,02%	58,53%
tudo -1	12.406	**86,07%**	60,32%
-5 até 5	12.406	85,08%	58,47%

Os resultados apresentados nas Tabelas 5.1 (falhas injetadas = 1.676) e 5.2 (falhas injetadas = 2.178) demonstram boa cobertura de falhas com o método de teste integrado aplicado ao FemtoJava. As melhores coberturas estão relacionadas às falhas no PC (falhas que afetaram também o contador de programa), mas não ao número de ciclos e ao tamanho das entradas. A principal razão para a boa cobertura de falha é o fato de todos os casos de teste de software verificarem os resultados em tempo de execução, conforme pode ser observado na Figura 5.10 (MEIRELLES; COTA; LUBASZEWSKI, 2008). Assim, se uma falha interfere nas rotinas de teste, ou não o faz executar, a falha é indicada porque também atingiu o PC (*Program Counter*).

Tabela 5.2: Cobertura de falhas de hardware com casos de testes de software no Femto-Java pipeline com aplicação de fluxo de dados

Bubble10	*Ciclos*	*Cobertura*	*PC*
tudo 0	3.758	74,06%	44,35%
tudo 1	3.758	77,31%	45,82%
tudo -1	3.758	**79,72%**	46,88%
-5 até 5	3.758	79,59%	45,54%

Um dado não detalhado nas tabelas citadas, mas analisando separadamente *stuck-at-1*, a melhor cobertura observada foi de 94% para o algoritmo de ordenação (com entradas variando de 5 negativo à 5 positivo) no FemtoJava pipeline. Essa melhor cobertura é explicada pelo fato de se ter uma grande quantidade de “Os” (zeros) circulando nos sinais da arquitetura – circuito aberto, especialmente onde não há fluxo dos dados de entrada.

Para poder comparar os dados das Tabelas 5.1 (falhas injetadas = 1.676) e 5.2 (falhas injetadas = 2.178), dentro do escopo que considera-se que em um sistema embarcado o processador é específico para um pequeno grupo de aplicações, é possível considerar a utilização do aplicativo em si como o teste do programa. Assim, as Tabelas 5.3 e 5.4 mostram os resultados quando um algoritmo de ordenação é utilizado como um programa de teste para o FemtoJava multiciclo e *pipeline*. Essas últimas tabelas contêm o mesmo tipo de informação das primeiras.

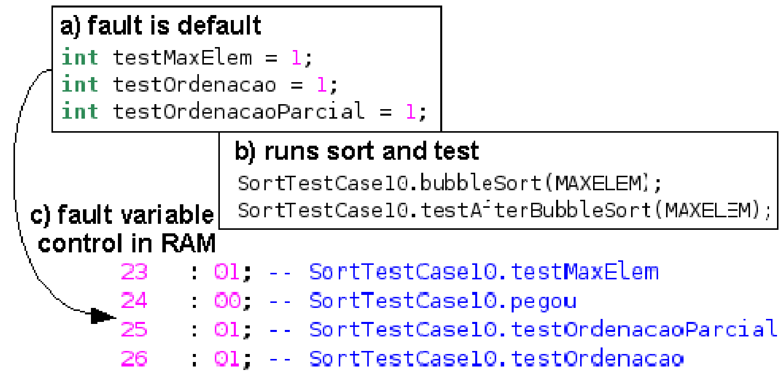


Figura 5.10: Casos de teste de software em código Java e RAM embarcada.

Tabela 5.3: Cobertura de falhas de hardware usando o algoritmo de ordenação como programa de testes no FemtoJava multiciclo.

Bubble10	*Ciclos*	*Cobertura*	*PC*
tudo 0	5.740	83,65%	31,21%
tudo 1	5.740	84,07%	32,58%
tudo -1	5.740	84,07%	39,26%
-5 até 5	8.200	**84,25%**	32,58%

Os resultados de ambas as tabelas mostram que se obteve uma boa cobertura de falhas da arquitetura quando um aplicativo (algoritmo de ordenação) é considerado um conjunto de rotinas de aleatória (*full randon*) (BECK FILHO et al., 2005). Porém, em todos os casos de intervalos de dados ordenados, o método de reuso dos casos de teste de software apresenta melhores resultados que a abordagem de rotinas aleatórias. Uma observação interessante é o número de instruções no algoritmo de ordenação ser de 57 instruções (mais detalhes na Tabela 5.6), podendo ser comparado com os dados apresentados em (BECK FILHO et al., 2005), onde a cobertura de falhas para 50 instruções aleatória no FemtoJava multiciclo é de 90,11% e para a arquitetura pipeline é de 41,4%. No entanto, para atingir essa cobertura de falhas é necessário armazenar e verificar todos dos dados da memória durante o teste, fazendo com que o custo dos testes seja praticamente inviável, quando se tratando de aplicações maiores.

A Tabela 5.5 apresenta os resultados dos testes em uma aplicação de fluxo de controle denominada Crane (controle de guindaste) (MOSER; NEBEL, 1999) desenvolvida para o FemtoJava. Na primeira coluna dessa tabela se tem o números de falhas injetadas na aplicação, o número de ciclos para executar a aplicação junto com os testes e as coberturas de falhas obtidas. No FemtoJava os resultados de cobertura de falhas para os testes em aplicações de fluxo de dados e de fluxo de controle são similares, isso ocorre porque o número de *bits* de controle e o de dados são equilibrados. Em sua arquitetura existem mais sinais de controle, mas com poucos *bits*, e poucos sinais de fluxo de dados, porém esses com muitos *bits*. Porém observa-se em aplicações um pouco mais complexas que o número de ciclos aumenta consideravelmente quando acrescidas as rotinas de testes. Especificamente nesse experimento o simulador “híbrido” (CACO-PS) não foi eficiente em relação ao tempo da execução da aplicação mais as rotinas de teste, consumindo dias para o término dos estudos de caso.

Na Tabela 5.5 é observada uma comparação entre os custos do uso de uma técnica de

Tabela 5.4: Cobertura de falhas de hardware usando o algoritmo de ordenação como programa de testes no FemtoJava pipeline.

Bubble10	*Ciclos*	*Cobertura*	*PC*
tudo 0	1.701	66,30%	34,71%
tudo 1	1.701	66,53%	34,25%
tudo -1	1.701	**74,79%**	36,87%
-5 até 5	3.330	73,19%	33,70%

Tabela 5.5: Cobertura de falhas dos casos de teste no FemtoJava pipeline em uma aplicação de fluxo de controle.

Crane	*Falhas*	*Ciclos*	*Cobertura*
stuck-at 0	1.780	37,564	60,60%
stuck-at 1	1.780	37,564	**91,29%**
total/média	3.560	37,564	75,94%

SBST funcional (*full random*) e o método de teste integrado (baseado nos casos de teste de software) proposto neste trabalho. Primeiramente, os custos do SBST funcional são correspondentes aos estudos de caso, tendo o algoritmo de ordenação sem os casos de teste como um conjunto de instruções consideradas randômicas, o que não apresenta nenhum custo em relação ao desenvolvimento de rotinas de teste. Já os custos referentes ao método de teste integrado refletem o acréscimo das rotinas de teste (como observado na Figura 5.10) no software embarcado para validar o software na plataforma alvo e verificar o hardware do sistema embarcado. Porém, na Tabela 5.6 outras métricas são abordadas para comparar outros custos dessas abordagens. Tais métricas são: tamanho do programa, tamanho da memória, número de opcodes diferentes usados e número de ciclos para executar os testes.

Tabela 5.6: Custos das abordagens de SBST funcional e teste integrado para um algoritmo de ordenação.

Custos	*Programa*	*Dados*	*Instruções*	*Opcodes*
Full Randon	160 bytes	35X16 bits	57	19
Teste Integrado	279 bytes	39X16 bits	111	22

Em (BECK FILHO et al., 2005) é apresentado 41,4%, 73,42% e 88,09% de cobertura de falhas para a arquitetura *pipeline* respectivamente com 50, 200 e 1000 instruções randômicas. Na abordagem apresentada no método de teste integrado acrescentaram-se casos de teste de software no código da aplicação (como visto na Figura 5.10), mas com um ganho na cobertura de falhas (86,07%) usando 3 casos de teste de software que não foram desenvolvidos inicialmente para o teste do hardware. O número de instruções do algoritmo de ordenação com os testes é de 111 instruções. Comparando com a cobertura observada com as 200 instruções randômicas, isto é, SBST funcional, o método de teste integrado demonstra um ganho de aproximadamente 13%, com um número menor de instruções e com um mecanismo de detecção mais rápido. O mecanismo de detecção

é simples porque para o método de teste integrado não é necessário armazenar e comprarar a RAM sem falhas com a RAM após a execução com falhas, diferentemente da abordagem de macros (HENTSCHKE et al., 2006) e totalmente aleatório (*full randon*) (BECK FILHO et al., 2005). Assim, há um equilíbrio do tempo de simulação e custos do método de teste integrado.

No gráfico ilustrado na Figura 5.11 (MEIRELLES; COTA; LUBASZEWSKI, 2008) é apresentada uma comparação entre os custos na memória de detecção dos métodos de SBST funcional (*full randon*) e de teste integrado. Observa-se uma redução do número de *bits* comparados,o que demonstra outro ganho além do reuso dos casos de testes de software por parte do método de teste integrado. A Figura 5.11 ilustra um gráfico com número de *bits* da RAM comparados para verificar a detecção de falhas inseridas nos sinais da arquitetura do FemtoJava (para um algoritmo de ordenação). Como a técnica SBST funcional totalmente aleatória compara toda a memória, o custo da detecção é obtido pelo tamanho da RAM (35X16 bits = 560). A detecção das falhas pelo teste integrado é realizada na verificação de específicos endereços de memória, observados de acordo com o número de casos de teste embarcado no código do software. Assim, para o algoritmo de ordenação (*bubble sort*) foram selecionados 3 casos de testes, então somente 3 pontos da memória (3x16 bits = 48) são checados (se uma dessas variáveis for igual a 1 há falha), porque variáveis são adicionadas ao programa e elas indicam se ocorreu ou não a falha.

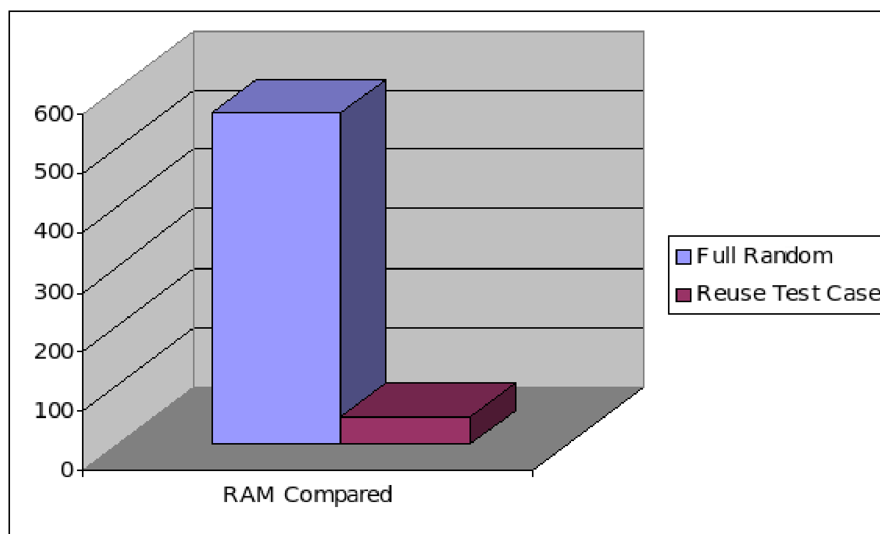


Figura 5.11: Número de bits da RAM comparados para detecção de falhas.

Esses resultados significam uma redução de aproximadamente 91,5% no número de verificações na RAM. O método de teste integrado não armazena toda memória na primeira simulação para comparar com toda memória após cada simulação com falha. Portanto, o custo da detecção também é menor, representando mais um ganho da utilização desse método.

5.4.1 Experimentos com Biblioteca de Auto-Teste

A cobertura de falhas, obtida somente com o reuso, pode ser aumentada com a introdução de rotinas (algoritmos) de teste de hardware implementados no nível do software e sendo embarcadas juntamente com os casos de testes construídos para o software, constituindo a integração de técnicas de teste de software e de hardware.

Em (MORAES, 2006) é apresentada a metodologia STEP (*Self-Test for Embedded Processors*), que consiste no desenvolvimento da biblioteca de auto-teste para os componentes visíveis de dados e para a unidade de controle do processador alvo, também seu estudo de caso é o FemtoJava multiciclo e *pipeline*, para os quais foi desenvolvida uma única biblioteca de auto-teste. O objetivo é facilitar a automatização do projeto e aplicação de auto-teste para o FemtoJava, uma vez que os processadores dessa família possuem alguns componentes em comum.

Para cada componente em comum entre o FemtoJava multiciclo e *pipeline*, uma única rotina de teste foi desenvolvida com cada uma das abordagens utilizadas. Essas rotinas podem ser aplicadas para ambas as versões do processador. Porém, o tamanho das palavras da memória de programa desses processadores é diferente (ITO; CARRO; JACOBI, 2001). 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 (ITO; CARRO; JACOBI, 2001).

A biblioteca de auto-teste foi desenvolvida de tal modo que, para cada componente, qualquer uma das abordagens implementadas forneçam a mesma cobertura de falhas (ou muito próxima). Essa característica foi alcançada pelo ajuste no número de padrões de teste em cada rotina. 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 (MORAES, 2006).

De acordo com o exposto, as características da metodologia STEP permitem uma avaliação do quão é possível complementar o teste integrado para aumentar a cobertura de falhas, usando o teste integrado proposto e rotinas de teste de hardware implementadas em software. Empiricamente, observando os detalhes das saídas das simulações com injeção de falhas, como exemplificado na Figura 5.12, se detectou que os sinais referentes ao banco de registradores do FemtoJava correspondiam aos sinais com menor cobertura de falhas. Dessa forma, entre as rotinas da biblioteca de auto-teste do STEP, a rotina que testa do banco de registradores foi selecionada, juntamente com a rotina da unidade lógica-aritmética (ULA), uma vez que esta também é um bloco da arquitetura do FemtoJava sensível às falhas.

A Figura 5.12 representa parte do relatório gerado pelo módulo de teste do CACO-PS. É detalhado qual o tipo de stuck-at (s-a-0 ou s-a-1) foi injetado, o número correspondente ao sinal, o nome do sinal e os bits onde não foram detectadas falhas no sinal, possibilitando assim, um mapeamento de onde concentrar os esforços para aplicação de técnicas de teste de hardware."

```
s-a-0,sinal 151 (set) no bit 0
s-a-0,sinal 152 (write_vars_t) no bit 1, 3, 4
s-a-0,sinal 153 (write_vars) no bit 0, 1, 3, 4
s-a-0,sinal 154 (write_address) no bit 0, 4
s-a-0,sinal 155 (sig_reg_bank_t) no bit 0
s-a-0,sinal 158 (reg_bank_addr1) no bit 4
s-a-1,sinal 6 (value_4) no bit 3
s-a-1,sinal 15 (queue_fsm_out) no bit 3, 4
s-a-1,sinal 18 (queue_mux_out) no bit 4
s-a-1,sinal 19 (queue_count_out) no bit 0, 4
s-a-1,sinal 20 (adder_queue_out) no bit 0, 4
s-a-1,sinal 21 (op_length_stop) no bit 0
s-a-1,sinal 25 (decoder2_out_t) no bit 0
s-a-1,sinal 35 (dd_wsp_force2) no bit 0
s-a-1,sinal 40 (nop_signal2) no bit 0
s-a-1,sinal 41 (nop_signal3) no bit 0
s-a-1,sinal 42 (dd_rsp2) no bit 0
s-a-1,sinal 44 (dd_rsp2_3) no bit 0
s-a-1,sinal 45 (dd_rsp2_3_not) no bit 0
s-a-1,sinal 46 (dd_wsp2_f) no bit 0
s-a-1,sinal 58 (dd_rspvar) no bit 0
s-a-1,sinal 62 (sig_dep) no bit 0
s-a-1,sinal 66 (not_same_var) no bit 0
```

Figura 5.12: Detalhes das saídas das simulações com injeções de falhas.

5.4.1.1 Banco de Registradores

O teste do banco de registradores do Femtojava é um pouco diferente 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, a própria rotina é um método que fará uso do repositório de variáveis e da pilha de operandos (MORAES, 2006). A aplicação dos padrões de teste ao banco de registradores foi baseada na pilha de operandos. De acordo com (MORAES, 2006), padrões determinísticos regulares foram selecionados para 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. Isso é realizado 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 5.13 ilustra o método em Java que implementa a rotina de teste para o banco de registradores. No total, são aplicados 48 padrões de teste determinísticos regulares ao banco de registradores (MORAES, 2006).

No algoritmo 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 5.13, nunca são escritos no banco de registradores, pois sempre são repassados por forwarding. Esses valores foram incluídos na rotina para garantir a escrita dos operandos nos registradores desejados. Outro ponto é que apenas onze dos dezesseis registradores do banco são escritos com operandos. Retirando-se os dois últimos valores, repassados por *forwarding*, apenas onze valores são empilhados (nove zeros e dois operandos). Isso

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

public static void regBankTesting() {
          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 = op + 21845; // incremento do operando (0x5555)
          }
 }
```

Figura 5.13: Rotina determinística regular para o banco de registradores.

se deve ao fato de que alguns registradores do banco são utilizados para outros propósitos (MORAES, 2006).

A Tabela 5.7 apresenta os resultados de uma séries de simulações envolvendo a rotina de teste do banco de registradores para as versões pipeline e multiciclo do FemtoJava, combinadas ou não com os casos de teste de software para o algoritmo de ordenação, já citado anteriormente.

Tabela 5.7: Cobertura de falhas de hardware usando o algoritmos de Banco de Registradores.

FemtoJava/Teste	*Ciclos*	*Falhas*	*Cobertura*	*PC*
Multiciclo	4.853	1.676	78,76%	50,42%
Multiciclo/Teste SW	17.243	1.676	86,92%	61,34%
Pipeline	7.811	2.178	77,64%	50,32%
Pipeline/Teste SW	2.091	2.178	**83,74%**	57,92%

Na Tabela 5.7 são apresentados a versão do FemtoJava a que se referem os dados, se foi combinado ou não com os casos de teste de software do algoritmo de ordenação, também o número de ciclos para ser executado, número de falhas injetadas, a cobertura de teste obtida e o número de falhas que atingiu o contador de programa. Os resultados demonstram uma variação pequena de cobertura de falhas entre as versões do FemtoJava. Quando aplicado combinado o teste do banco de registradores com os casos de teste de software (para o algoritmo de ordenação), há um aumento da cobertura em relação aos experimentos com o reuso dos casos de teste, principalmente na versão pipeline (em torno de 4%). O custo relativo ao acréscimo da rotina de teste de hardware no teste integrado se resume, uma vez que é uma rotina em Java, aplicado da mesma forma que os casos de teste de software, ao aumento da detecção, pelo fato de mais pontos da memória serem

verificados para se detectar as falhas.

5.4.1.2 Unidade Lógica-Aritmética

Para a ULA (Unidade Lógica-Aritmética) foi selecionada uma rotina desenvolvida que utiliza padrões de teste determinísticos regulares, para ficar de acordo com a rotina do banco de registradores. O método *aluRegDetTesting*, apresentado na Figura 5.14, contém o código Java correspondente à rotina de teste que gera os padrões determinísticos aplicados à ULA.

```
public static int resultsULA[]={0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
public static void aluRegDetTesting() {
            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)
                                resultsULA[idx] = operand1 + operand2;
                                idx++;
                                resultsULA[idx] = operand1 - operand2;
                                idx++;
                                resultsULA[idx] = operand1 & operand2;
                                idx++;
                                resultsULA[idx] = operand1 | operand2;
                                idx++;
                                resultsULA[idx] = operand1 ^ operand2;
                                idx++;
                                operand1 = 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)
                      operand2 = operand2 - 1;
                      resultsULA[idx] = operand2;
                      idx++;
                      operand2 = operand2 + 21845; // incremento do operando 2 (0x555)
                //System.out.println(""+operand2);
            }
}
```

Figura 5.14: Rotina determinística regular para a ULA.

De acordo com (MORAES, 2006), nessa rotina são executadas 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 (MORAES, 2006).

A Tabela 5.8 apresenta os resultados das simulações com a rotina de teste da ULA para as versões do FemtoJava, da mesma forma como os experimentos com a rotina de teste do banco de registradores, combinadas ou não com os casos de teste de software para o algoritmo de ordenação.

Na Tabela 5.8 também são apresentados a versão do FemtoJava a que se referem os dados, se foi combinado com os casos de teste de software ou não, também o número de ciclos para ser executado, número de falhas injetadas, a cobertura de teste obtida e o número de falhas que atingiu o contador de programa. Os resultados são similares aos obtidos com o banco de registrado, devido às próprias características da biblioteca de auto-teste, descritas no início desta seção. Observa-se nos dados uma variação pequena

Tabela 5.8: Cobertura de falhas de hardware usando o algoritmos de teste da Unidade Lógica-Aritmética.

FemtoJava/Teste	*Ciclos*	*Falhas*	*Cobertura*	*PC*
Multiciclo	1.491	1.676	78,70%	68,08%
Multiciclo/Teste SW	13.881	1.676	86,74%	72,67%
Pipeline	701	2.178	77,77%	58,91%
Pipeline/Teste SW	4.441	2.178	**81,99%**	65,93%

de cobertura de falhas entre as versões do FemtoJava. Quando aplicada combinando o teste da ULA com os casos de teste de software, há um pequeno aumento da cobertura, comparando com os dados obtidos somente com o reuso dos casos de teste de software, mais perceptível na versão pipeline (cerca de 2%).

Objetivando saturar um pouco mais as possibilidades de aplicação das rotinas de teste, juntamente com os casos de teste de software para um algoritmo de ordenação, foi executado um experimento, apenas para o FemtoJava *pipeiline*, onde as rotinas de teste da ULA e do banco de registradores são utilizados na mesma simulação com injeção de falhas. A Tabela 5.9 apresenta se os dados desse estudo foi combinado com os casos de teste de software ou não, o número de ciclos, número de falhas injetadas, a cobertura de teste obtida e o número de falhas que atingiram o contador de programa.

Tabela 5.9: Cobertura de falhas de hardware usando o algoritmos de Banco de Registradores e Unidade Lógica-Aritmética.

FemtoJava/Teste	*Ciclos*	*Falhas*	*Cobertura*	*PC*
Pipeline	6.617	2.178	83,05%	78,70%
Pipeline/Teste SW	11.252	2.178	**88,07%**	79,20%

Os resultados apresentados na Tabela 5.9 demonstram um aumento da cobertura de falhas para o FemtoJava *pipeline* em ambos os casos, agregando ou não as rotinas da biblioteca de auto-teste aos casos de teste de software. Primeiro se observa que, sem os casos de testes, há uma melhor cobertura de falhas quando se junta as duas rotinas de teste de hardware, comparada quando aplicadas isoladamente (uma variação de 6%). Justamente porque cada rotina exercita blocos diferentes, porém há uma grande redundância. Outro dado observado é quando, além das duas rotinas de teste de hardware (ULA e banco de registradores), se tem os casos de teste de software. Comparando com os números de quando se tem apenas o reuso, o ganho de cobertura é mais significativo, próximo aos 10%. Quando comparado com o reuso combinado com apenas uma das técnicas, o ganho fica por volta dos 5%. Por outro lado, algumas vantagens apresentados pelo método de reuso proposto, como tempo de simulação, número de ciclos e custo de verificação, são comprometidas quando esse nível de saturação e combinação com rotinas de teste de hardware é grande.

Somando a aplicação das duas rotinas da biblioteca de auto-teste e os casos de teste de software poderia se esperar uma cobertura de falhas próxima aos 100%, pelas próprias características das rotinas apresentadas em (MORAES, 2006) e resumidas nesta seção. Entretanto, a limitação para se atingir uma cobertura maior está ligada ao simulador CACO-PS. Como já explicado anteriormente, ele oferece a possibilidade da descrição es-

trutural de qualquer arquitetura, sendo de propósito geral. Assim, o ambiente não simula exatamente o ASIP específico correspondente ao software desenvolvido. Mesmo para os casos das rotinas de hardware a cobertura não é máxima porque o CACO-PS testa todas entradas de um componente, com isso varia a entrada de dados chamando a função do comportamento para verificar a alteração na saída. Por exemplo, comparando com uma simulação em *SystemC* (GROTKER, 2002) (uma linguagem de modelagem de hardware de alto nível baseado em C++) se pode define as entradas às quais os componentes serão sensíveis (como um *flip-flop*). Além disso, em *SystemC* pode ser definido que só deve chamar a função quando o *clock* variar (BARCELOS, 2008). Dessa forma, somando todos os componentes é possível se obter uma variação no resultado dos testes. Como o desenvolvimento de um simulador não é o foco deste trabalho, o CACO-PS se apresenta com uma boa ferramenta para se constituir o ambiente híbrido de validação do método de teste integrado de software e hardware aplicado ao FemtoJava, apresentando bons resultados de cobertura de falhas.

6 CONSIDERAÇÕES FINAIS

Neste trabalho foi apresentado um método de teste que tem como base a abordagem de teste integrado de software e hardware para sistemas embarcados. Esse método visa o teste para os sistemas em que o hardware é projetado para executar instruções de um determinado software. Esse tipos hardware é uma versão ASIP (*Application Specific Instruction set Processor*) de um processador, onde sua unidade de controle possui apenas as instruções usadas por aquele programa em específico.

Foi realizado um levantamento das terminologias em diferentes áreas (engenharia de software, tolerância a falhas e sistemas digitais) que norteiam os assuntos tratados nesta dissertação, selecionando uma terminologia comum para uma melhor compreensão do mesmo. Como o método de teste proposto parte da engenharia de software, um estudo dos conceitos, critérios e diferentes técnicas de teste de software, sinalizando como aplicá-los no teste de software embarcado foi apresentado. Neste contexto, também foram apresentadas algumas ferramentas para automatizar os testes. Posteriormente, levantou-se problemas em relação ao teste de processadores, descrevendo tipo e técnicas de auto-teste para microprocessadores, enfatizando o modelo de falhas (*stuck-at*), utilizado para a injeção de falhas nos estudos de caso deste trabalho. Além disso, as técnicas de auto-teste de processadores baseado em software foram discutidas e comparadas com o método de teste integrado proposto. Com a visão geral da área de testes nos dois mundo (software e hardware) e as questões de testes em microprocessadores, foi possível apresentar um modelo prático do método de teste integrado. Por fim, foram apresentados uma instância do que foi colocado teoricamente num modelo conceitual independente de plataforma e os resultados dos estudos de caso - uma aplicação de fluxo de dados (um algoritmo de ordenação) e uma aplicação de fluxo de dados (controle de guindaste).

6.1 Trabalhos Futuros

Como trabalhos futuros, no campo teórico, uma outra conseqüência possível do método de teste integrado é a definição dos requisitos e restrições de teste de software embarcado para a plataforma alvo, realizando a validação do software sob o ponto de vista das restrições da plataforma. Nessa etapa assume que o software está logicamente correto (conforme já apresentado neste trabalho), mas exige, para sua execução (validação), os recursos disponíveis na plataforma de hardware. Com isso, o método permitiria não apenas a verificação da lógica, mas também a adequação e o impacto físico (tamanho da memória e desempenho, número de ciclos, por exemplo) da execução software com os testes na plataforma específica.

Do ponto de vista prático, o trabalho apresentou limitação devido à plataforma de simulação. Há necessidade de outros estudos de casos mais complexos para explorar o

potencial do método de teste integrado de software e hardware. Também foram encontradas limitações de tempo de teste já no estudo de caso do algoritmo do Crane, uma vez que o número de chamadas de métodos (e casos de teste) faz aumentar consideravelmente o número de ciclos no FemtoJava. Entretanto, isso não seria problema se o simulador, ambiente híbrido necessário para validação do método de teste integrado, for mais eficiente. Então, como trabalho futuro sugere-se a utilização do simulador YAFJS (yet Another femtoJava Simulator) (BARCELOS, 2008), não utilizado neste trabalho pelo seu desenvolvimento ter ocorrido em paralelo ao mesmo, para refazer os atuais experimentos e ampliar os estudos de casos com outras aplicações.

Ao final da realização do estudos de caso apresentados neste trabalho foram levantadas as alternativas para diminuir as limitações comentadas. No caso da utilização do simulador YAFJS também será necessário uma adaptação de um módulo de teste nessa ferramenta, similar ao realizado para o CACO-PS, conforme descrito neste trabalho. Entretanto, uma vez que simulador YAFJS foi desenvolvido em SystemC, existem mecanismos de teste nativos no System C que facilitam a extensão de módulos de teste. Além disso, no projeto do YAFJS procurou-se reutilizar códigos de modelos de componente já existentes e descritos para os outros simuladores como CACO-PS. O simulador foi desenvolvido com o objetivo de ser rápido e escalável. A simulação dos processadores atualmente implementados, derivados do CACO-PS, ocorre em nível de ciclo de relógio, ou seja, é preciso. Hoje, o simulador é usado para suportar os processadores FemtoJava multiciclo e *pipeline*, os mesmo dos estudos de caso para o método de teste integrado. Por fim, o YAFJS é sete vezes mais rápido que o CACO-PS (BARCELOS, 2008), porque o CACO-PS testa todas entradas de um componente, com isso varia a entrada de dados chamando a função do comportamento para ver se vai ter alteração na saída. Por exemplo, comparando com uma simulação no YAFJS que é possível definir as entradas às quais os componentes serão sensíveis. Além disso, em *SystemC* pode ser definido que só deve chamar a função quando o *clock* variar (BARCELOS, 2008). Assim, somando todos os componentes é possível se obter uma variação no resultado e tempo dos testes. Em suma, uma investigação do método de integrado de software e hardware usando o YAFJS é o próximo passo para uma melhor validação e ampliação dos estudos de caso apresentados nesta dissertação.

6.2 Conclusão

O principal objetivo do método de teste integrado é atingir um percentual de reuso dos casos de testes desenvolvidos para o teste do software no teste de hardware, o que constitui o teste integrado de software e hardware. Uma vez que o processador (hardware) é gerado a partir do código do software, foi avaliado um mecanismo de verificação desse hardware através dos testes para verificação e validação do software. Para verificar o software são usadas as ferramentas de testes (JUnit e JaBUTi) que permitiram aplicar as técnicas e observar os critérios de teste de software. Entretanto, para validar esse software é necessário executá-lo. Neste ponto do fluxo de projeto, o hardware ainda não está pronto, porém uma integração dos testes tornou-se uma abordagem interessante. Então, a integração deve ser viabilizada por um ambiente de simulação "híbrido" da plataforma alvo, permitindo a validação do software e verificação do hardware quando os mesmos forem submetidos às condições de falhas.

Apenas com o reuso dos testes de software para a verificação do hardware foi obtido uma cobertura de falhas (com o modelo de falhas *stuck-at*) de até 86% para a versão

multiciclo do microcontrolador FemtoJava e de até 79% para a versão *pipeline*. Além disso, os resultados apresentam uma redução de aproximadamente 91,5% no número de verificações na RAM quando se reusa os testes de software para o teste do processador FemtoJava, comparando com os trabalhos relacionados que tem a mesma plataforma com estudo de caso. O método de teste integrado não armazena toda a memória na primeira simulação para comparar com a memória após cada simulação com falha. Portanto, o custo da detecção também é menor. Assim, esse fator somado à reutilização dos casos teste do software para detectarem falhas em hardware, representam os principais ganhos da utilização desse método.

A cobertura de falhas pode ser aumentada com a introdução de rotinas (algoritmos) de teste de hardware implementados no nível do software e sendo embarcado juntamente com os casos de testes construídos para o software, o que constituiu a integração dos teste de software e de hardware. Os novos resultados apresentaram até 88% cobertura de falhas para as mesmas aplicações e microcontrolador citados. As rotinas utilizadas são parte de uma biblioteca de auto-teste para testar a família FemtoJava. Assim, foi realizada uma comparação entre a cobertura obtida apenas pelo reuso dos casos de teste com a cobertura de falhas usando as rotinas da biblioteca de auto-teste e também com o caso em foram unidas ambas as abordagens. O que diferenciou o custo delas foi o número de ciclos necessários para executar as aplicações com programas de teste e o número de verificações na memórias para se obter a cobertura de falhas. De tal forma que, apenas o reuso dos casos de teste ainda se mostraram com um método interessante pelo auto nível de abstração, maior que os demais, uma vez que não é desenvolvido para testar o hardware.

Por fim, se obteve como resultante deste trabalho um método de teste que se origina do teste de software, suportado por ferramentas de automação desses testes. Casos de teste são selecionados de acordo com a cobertura de teste observadas para o teste do software embarcado. Posteriormente, os testes são adaptados a plataforma em que será embarcado, o que os habilita em indicar as falhas inseridas no hardware. Assim, para o testador de software as questões do teste de hardware são abstraídas. Além disso, o testador do hardware pode focar os testes de acordo com o relatório de saída das simulações com o reuso dos casos de teste de software, os quais não tem custo algum de desenvolvimento do ponto de vista de teste de hardware. Com isso, se obtém um ganho no tempo de desenvolvimento e aplicação do teste de hardware, em caso de se objetivar uma cobertura maior à apresentada quando apenas se reusa os casos de teste de software.

REFERÊNCIAS

ABRAMOVICI, M.; FRIEDMAN, A. D.; BREUER, M. A. **Digital Systems Testing and Testable Design**. Washington: IEEE Press, 1990.

AVIZIENIS, A. et al. Basic Concepts and Taxonomy of Dependable and Secure Computing. **IEEE Trans. Dependable Secur. Comput.**, Los Alamitos, CA, USA, v.1, n.1, p.11–33, 2004.

BARCELOS, D. **Modelo de Migração de Tarefas para MPSoCs baseados em Redes-em-chip**. 2008. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

BATCHER, K.; PAPACHRISTOU, C. Instruction Randomization Self Test For Processor Cores. In: IEEE VLSI TEST SYMPOSIUM, 17., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1999. p.34–40.

BECK FILHO, 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 FILHO, A. C. S.; CARRO, L. 12th Low Power Java Processor for Embedded Applications. In: IFIP WG 10.5 INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION OF SYSTEM-ON-CHIP, Darmstadt, Germany. **Proceedings...** Technische Universität Darmstadt: Insitute of Microelectronic Systems, 2003. p.239–244.

BECK FILHO, A. C. S. et al. CACO-PS: a general purpose cycle-accurate configurable power simulator. In: INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 16., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2003. p.349–354.

BECK FILHO, A. C. S. et al. Fast and Efficient Test Generation for Embedded Stack Processors. In: LATIN-AMERICAN TEST WORKSHOP, 6., Salvador, Brazil. **Proceedings...** IEEE Latin-American Test Workshop, 2005. p.37–42.

BEIZER, B. **Software Testing Techniques**. 2.ed. [S.l.]: Van Nostrand Reinhold, 1990.

BROEKMAN, B.; NOTENBOOM, E. **Testing Embedded Software**. London, UK: Addison-Wesley, 2003.

BUNDELL, G. A. et al. A software component verification tool. In: INTERNATIONAL CONFERENCE ON SOFTWARE METHODS AND TOOLS, Wollongong. **Proceedings...** IEEE Computer Society Press, 2000. p.137–147.

BUSHNELL, M.; AGRAWAL, V. **Essentials of Eletronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits**. Boston: Kluwer Academic Publishers, 2000.

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

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, March 2001.

CHILLAREGE, R. **Software Testing Best Practices**. [S.l.]: IBM Technical Report, 1999.

CORNO, F. et al. On the test of microprocessor IP cores. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, Piscataway, NJ, USA. **Proceedings...** IEEE Press, 2001. p.209–213.

COURTOIS, B. Failure Mechanisms, Fault Hypotheses and Analytical Testing of LSI-NMOS (HMOS) Circuits. In: INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, Maryland. **Proceedings...** Computer Science Press, 1981.

CRAIG, R. D.; JASKIEL, S. P. **Systematic Software Testing**. [S.l.]: Artech House, 2002.

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: Kluwer Academic Publishers, 2004.

GOMES, V. F.; BECK FILHO, A. C.; CARRO, L. A VHDL Implementation of a Low Power Pipelined Java Processor for Embedded Applications. In: IBERCHIP, 10., Cortagena de Indias, Colombia. **Proceedings...** [S.l.: s.n.], 2004. p.102–108.

GROTKER, T. **System design with SystemC**. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture**: a quantitative approach. 3rd.ed. [S.l.]: Morgan Kaufmann Publishers, 2003.

HENTSCHKE, R. et al. Using Genetic Algorithms to Accelerate Automatic Software Generation for Microprocessor Functional Testing. **Journal of Integrated Circuits and Systems**, [S.l.], v.1, p.5–10, 2006.

HORGAN, J. R.; LONDON, S.; LYU, L. R. Achieving Software Quality with Testing Coverage Measures. **Computer**, [S.l.], v.27, n.9, p.60–69, September 1994.

IEEE. **Institute of Electrical and Electronics (IEEE) Standard Glossary of Software Engineering Terminology**. [S.l.: s.n.], 1990.

ITO, S. A.; CARRO, L.; JACOBI, R. Making Java Work for Microtrolles Applications. **IEEE Design & Test of Computer**, [S.l.], v.18, n.5, p.100–110, 2001.

JORGENSEN, P. C. **Software Testing - A Craftsmanś Approach**. [S.l.]: CRC Press, 2002.

KRUG, M. R. **Aumento da Testabilidade do Hardware com Auxílio da Engenharia de Software**. 2007. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

KRUG, M. R.; MORAES, M. S.; LUBASZESKI, M. S. Using a Software Testing Technique to Identify Registers for Partial Scan Implementation. In: SYMPOSIUM ON INTEGRATED CIRCUITS DESIGN, 19., Ouro Preto, Brazil. **Proceedings...** New York: ACM, 2006. p.208–213.

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

LAPRIE, J.-C. Dependable Computing and Fault-Tolerance: concepts and terminology. In: IEEE INTERNATIONAL FAULT-TOLERANT COMPUTING SYMPOSIUM, 15. **Proceedings...** [S.l.: s.n.], 1985. p.2–11.

LAPRIE, J.-C. Dependability of Computer Systems: from concepts to limits. In: IFIP INTERNATIONAL WORKSHOP ON DEPENDABLE COMPUTING AND ITS APPLICATIONS, Johannesburg, South Africa. **Proceedings...** [S.l.: s.n.], 1998.

LUBASZEWSKI, M.; COTA, E.; KRUG, M. R. **Teste e Projeto Visando o Teste de Circuitos e Sistemas Integrados**. 2nd.ed. Porto Alegre: Instituto de Informática da UFRGS: Concepção de Circuitos Integrados, 2002. p.167–189.

MALDONADO, J. C. **Critérios Potenciais Usos**: uma contribuição ao teste estrutural de software. 1991. Tese (Doutorado em Ciência da Computação) — DCA/FEE/UNICAMP, Campinas.

MALDONADO, J. C. et al. **Aspectos teóricos e empíricos de teste de cobertura de software**. [S.l.]: Instituto de Ciências Matemáticas e de Computação - ICMC/USP, 1998.

MALDONADO, J. C. et al. **Introdução ao Teste de Software**. [S.l.]: Instituto de Ciências Matemáticas e de Computação - ICMC/USP, 2004.

MEIRELLES, P.; COTA, E.; LUBASZEWSKI, M. Reusing Software Test Cases to Test an Embedded Microprocessor: a case study. In: IEEE LATIN AMERICAN TEST WORKSHOP, 9., Puebla, México. **Proceedings...** [S.l.: s.n.], 2008. p.171–176.

MORAES, M. et al. A constraint-based solution for on-line testing of processors embedded in real-time applications. In: INTEGRATED CIRCUITS AND SYSTEM DESIGN, 18., Florianolpolis, Brazil. **Proceedings...** ACM, 2005. p.68–73.

MORAES, M. S. **STEP**: planejamento, geração e seleção de auto-teste on-line para processadores embarcados. 2006. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática - Universidade Federal do Rio Grande do Sul, Porto Alegre.

MOSER, E.; NEBEL, W. Case study: system model of crane and embedded control. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, Munich, Germany. **Proceedings...** ACM, 1999. p.139.

MYERS, G. J. **The art of software testing**. New Jersey, USA: John Wiley & Sons, 2004.

OFFUTT, A. J.; IRVINE, A. Testing object-oriented software using the category-partition method. In: INTERNATIONAL CONFERENCE ON TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS, 17., Santa Barbara. **Proceedings...** Prentice-Hall, 1995. p.293–304.

PÁDUA, W. de. **Engenharia de Software**: fundamentos, métodos e padrões. Rio de Janeiro: LTC, 2003.

PASCHALIS, A. et al. Deterministic Software-Based Self-Testing of Embedded Processor Cores. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, Munich. **Proceedings...** [S.l.: s.n.], 2001. p.92–96.

PRADHAN, D. K. **Fault-Tolerant Computer System Design**. [S.l.]: Prentice Hall, 1996. 560p.

PRESSMAN, R. S. **Engenharia de software**. São Paulo: McGraw-Hill, 2002.

PRESSMAN, R. S. **Engenharia de software**. São Paulo: McGraw-Hill, 2006.

RAJSUMAN, R. **Digital Hardware Testing**: transistor-level fault modelling and testing. [S.l.]: Artech House, 1992.

ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. **Qualidade de software - Teoria e prática**. São Paulo: Prentice Hall, 2001.

SANGIOVANNI-VINCENTELLI, A. L.; MCGEER, P. C.; SALDANHA, A. Verification of Electronic Systems. In: DESIGN AUTOMATION CONFERENCE, Las Vegas. **Proceedings...** [S.l.: s.n.], 1996. p.106–111.

SOUZA DE, S. d. R. S. **Avaliação do Custo e Eficácia do Critério Análise de Mutantes na Atividade de Teste de Software**. 1996. Dissertação (Mestrado em Ciência da Computação) — ICMC/USP, São Carlos.

SUN-MICROSYSTEMS. **PicoJava-II Programmer's Reference Manual**. [S.l.: s.n.], 1999.

THéVENOD-FOSSE, P.; WAESELYNCK, H. STATEMATE applied to statistical software testing. In: ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, Cambridge, Massachusetts, United States. **Proceedings...** ACM, 1993. p.99–109.

VINCENZI, A. M. R. **Subídios para o estabelecimento de estratégias de teste baseadas na técnica de mutação**. 1998. Dissertação (Mestrado em Ciência da Computação) — ICMC-USP, São Carlos.

VINCENZI, A. M. R. **Orientação a Objeto**: definição, implementação e análise de recursos de teste e validação. 2004. Tese (Doutorado em Ciência da Computação) — ICMC-USP, São Carlos.

WADSACK, R. L. Fault Modelling and Logic Simulation of CMOS and MOS Integrated Circuits. **The Bell System Technical Journal**, [S.l.], v.57, n.5, p.1449–1474, May 1978.

WEBER, T.; WEBER, R.; PORTO, I. J. **Fundamentos de tolerância a falha**. Vitória: SBC, 1990. Apostila preparada para o IX JAI – Jornada de tualização em Informática.

WHITTAKER, J. A. Stochastic Software Testing. **Annals of Software Engineering**, [S.l.], v.4, p.115–131, 1997.

WHITTAKER, J. A.; M.THOMASON. A markov chain model for statistical software testing. **A markov chain model for statistical software testing**, [S.l.], v.20, n.10, p.812–824, 1994.

WONG, E. W.; RAO, S.; LINN, J. Coverage Testing Embedded Software on Symbian/OMAP. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, 18., San Francisco. **Proceedings...** [S.l.: s.n.], 2006.

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

APÊNDICE A CÓDIGO-FONTE DOS ESTUDOS DE CASO

```
1  /*
    *  A classe nao pode pertencer a um Pacote especifico
    */
   import saito.sashimi.IOInterface;
   import saito.sashimi.FemtoJavaIO;
6  //import java.io.*;

   public class SortTestCase10 implements IOInterface {

     /*
11    *  Todas variaveis sao obrigatoriamente estaticas
      */
     public static int i,j;
     /*
      *  NUmero de Elementos do Vetor
16    */
     public static int MAXELEM = 10;
     /*
      * Vetor com as entradas
      */
21   public static int[] vetorDados = {5,4,3,2,1,-1,-2,-3,-4,-5};
     /*
      *  Variaveis para observar resposta dos test case
      */
     public static int pegou = 0;
26   public static int testMaxElem = 1;
     public static int testOrdenacao = 1;
     public static int testOrdenacaoParcial = 1;
     /*
          * Construtor Vazio
31        */
         public SortTestCase10() {}
         /*
          * Todos o metodos sao obrigatoriamente estaticos
          */
36   public static void main(String[] argv) {
          SortTestCase10 sort = new SortTestCase10();
            FemtoJavaIO.setIOClass(sort);
       SortTestCase10.initSystem();
     }
41
     public static void initSystem() {
       //Chamada dos Test case
       //System.out.println("TestMaxElem = " + testMaxElem);
```

```
    SortTestCase10.testInitSystem();
46  //System.out.println("TestIncializacao = " + testIncializacao);
    SortTestCase10.bubbleSort(MAXELEM);
    SortTestCase10.testAfterBubbleSort(MAXELEM);

   }
51
      public static int bubbleSort(int maxnos) {
    int temp;

    for (i=0; i<maxnos-1; i++) {
56   for (j=maxnos-1; j>i; j--) {
     if (vetorDados[j]<vetorDados[j-1]) {
      temp=vetorDados[j];
       vetorDados[j]=vetorDados[j-1];
       vetorDados[j-1]=temp;
61    }
      //Chamada do Test case
      testBubbleSort(j);
     }
    }
66  return 0;
   }
   /*
    * (non-Javadoc)
    * @see saito.sashimi.IOInterface
71  * Metodos correspondesnte as assinaturas em  saito.sashimi.
       IOInterface
    */
   public synchronized int read(int channel) {
    // TODO Auto-generated method stub
          return 0;
76 }
      public synchronized void write(int value, int channel) {
        // TODO Auto-generated method stub
       }
   public int getSerialData() {
81  // TODO Auto-generated method stub
    return 0;
   }
   public int getSerialStatus() {
    // TODO Auto-generated method stub
86  return 0;
   }
   public void sendSerialData(int value) {
    // TODO Auto-generated method stub

91 }

  // Metodos Test Cases

   public static void testInitSystem() {
96  if (MAXELEM ==10){
     testMaxElem = 0;
    }
    //System.out.println("TestMaxElem = " + testMaxElem);
   }
101
```

```
    public static void testBubbleSort(int index){

     if (vetorDados[index]<vetorDados[index -1]) {
      pegou = 1;
106  }

     if (pegou==0) {
      testOrdenacaoParcial = 0;
     }
111  //System.out.println("testOrdenacaoParcial= " + testOrdenacaoParcial
          + " -> " + vetorDados[index]+ ">" + vetorDados[index -1]);
    }

    public static void testAfterBubbleSort (int maxElem){
     for (i=0;i<maxElem-1;i++) {
116      if ( vetorDados[i]<= vetorDados[i+1] ){
          testOrdenacao = 0;
          //System.out.println("testOrdenacao = " + testOrdenacao );
         }else{
          testOrdenacao = 1;
121       //System.out.println("testOrdenacao Else = " + testOrdenacao );
         }
       }

    }
126 //fim test cases

   }
```

Listing 6.1: Algoritmo de ordenação com os casos de teste selecionados

```
/*
 *  A classe nao pode pertencer a um Pacote especifico
 */
import saito.sashimi.IOInterface;
import saito.sashimi.FemtoJavaIO;
//import java.io.*;

public class SortTestCaseAluRegBankTesting implements IOInterface {

  /*
   *  Todas variaveis sao obrigatoriamente estaticas
   */
  public static int i,j;
  /*
   *  NUmero de Elementos do Vetor
   */
  public static int MAXELEM = 10;
  /*
   * Vetor com as entradas
   */
  public static int[] vetorDados = {5,4,3,2,1,-1,-2,-3,-4,-5};
  /*
   *  Variaveis para observar resposta dos test case
   */
  public static int pegou = 0;
  public static int testMaxElem = 1;
  public static int testOrdenacao = 1;
  public static int testOrdenacaoParcial = 1;
  /*
       * Construtor Vazio
       */
      public SortTestCaseAluRegBankTesting() {}
      /*
       * Todos o metodos sao obrigatoriamente estaticos
       */
  public static void main(String[] argv) {
       SortTestCaseAluRegBankTesting sort = new
            SortTestCaseAluRegBankTesting();
          FemtoJavaIO.setIOClass(sort);
    SortTestCaseAluRegBankTesting.initSystem();
  }

  public static void initSystem() {
    //Chamada dos Test case
    //System.out.println("TestMaxElem = " + testMaxElem);
    SortTestCaseAluRegBankTesting.testInitSystem();
    //System.out.println("TestIncializacao = " + testIncializacao);
    SortTestCaseAluRegBankTesting.bubbleSort(MAXELEM);
    SortTestCaseAluRegBankTesting.testAfterBubbleSort(MAXELEM);
    SortTestCaseAluRegBankTesting.aluRegDetTesting();
    SortTestCaseAluRegBankTesting.regBankTesting();

  }

      public static int bubbleSort(int maxnos) {
    int temp;

    for (i=0; i<maxnos-1; i++) {
```

```
        for (j=maxnos-1; j>i; j--) {
        if (vetorDados[j]<vetorDados[j-1]) {
         temp=vetorDados[j];
          vetorDados[j]=vetorDados[j-1];
62        vetorDados[j-1]=temp;
         }
         //Chamada do Test case
         testBubbleSort(j);
        }
67    }
      return 0;
     }
     /*
      * (non-Javadoc)
72    * @see saito.sashimi.IOInterface
      * Metodos correspondesnte as assinaturas em  saito.sashimi.
         IOInterface
      */
     public synchronized int read(int channel) {
      // TODO Auto-generated method stub
77          return 0;
     }
        public synchronized void write(int value, int channel) {
          // TODO Auto-generated method stub
         }
82   public int getSerialData() {
      // TODO Auto-generated method stub
      return 0;
     }
     public int getSerialStatus() {
87    // TODO Auto-generated method stub
      return 0;
     }
     public void sendSerialData(int value) {
      // TODO Auto-generated method stub
92
     }

    // Metodos Test Cases

97   public static void testInitSystem() {
      if (MAXELEM ==10){
       testMaxElem = 0;
      }
      //System.out.println("TestMaxElem = " + testMaxElem);
102  }

     public static void testBubbleSort(int index){

      if (vetorDados[index]<vetorDados[index-1]) {
107    pegou = 1;
      }

      if (pegou==0) {
       testOrdenacaoParcial = 0;
112   }
      //System.out.println("testOrdenacaoParcial= " + testOrdenacaoParcial
          + " - > " + vetorDados[index]+ ">" + vetorDados[index-1]);
```

```
    }

    public static void testAfterBubbleSort (int maxElem){
117   for (i=0;i<maxElem-1;i++) {
        if ( vetorDados[i]<= vetorDados[i+1] ){
         testOrdenacao = 0;
         //System.out.println("testOrdenacao = " + testOrdenacao );
        }else{
122      testOrdenacao = 1;
         //System.out.println("testOrdenacao Else = " + testOrdenacao );
        }
      }

127 }
  //fim test cases


  // test RegBank
132 public static int[] results = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
137     };

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

142     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))));
147       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)))))))));
152       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 = op + 21845; // incremento do operando (0x5555)
        }
157  }

  //fim test RegBank

  // test ULA
162 public static int resultsULA[]={0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0};
    public static void aluRegDetTesting() {
         int operand1 = 0;  // valor inicial do operando 1 (0x0000)
         int operand2 = 0;  // valor inicial do operando 2 (0x0000)
         int idx = 0;
167
```

```
        while (operand2 != -2) {    // valor final do operando 2 (0xFFFE)
            while (operand1 != -2) { // valor final do operando 1 (0xFFFE)
                resultsULA[idx] = operand1 + operand2;
                idx++;
172             resultsULA[idx] = operand1 - operand2;
                idx++;
                resultsULA[idx] = operand1 & operand2;
                idx++;
                resultsULA[idx] = operand1 | operand2;
177             idx++;
                resultsULA[idx] = operand1 ^ operand2;
                idx++;
                operand1 = operand1 - 1;  // incremento do operando 1 (0
                    xFFFF)
            }
182         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)
187         operand2 = operand2 - 1;
            resultsULA[idx] = operand2;
            idx++;
            operand2 = operand2 + 21845; // incremento do operando 2 (0
                x555)
      //System.out.println(""+operand2);
192     }
    }

  //fim test

197 }
```

Listing 6.2: Rotinas da biblioteca de auto-teste inseridas ao código do algoritmo de ordenação com os casos de teste

```
public class Crane {
 //static int i;
3 static int testControl = 1;
 static int testReadSensors = 1, testDiagnosis=1;
 public static void initSystem(){
  //for (i=0; i < 10; i++) {
  Sensor16.readSensors();
8 Sensor16.testReadSensors();
  Sensor16.readSensors();
  Sensor16.testReadSensors();
          Sensor16.readSensors();
          Sensor16.testReadSensors();
13        Sensor16.readSensors();
          Sensor16.testReadSensors();
          Sensor16.readSensors();
          Sensor16.testReadSensors();
          Control16.control();
18 Control16.testControl();
       //}

 }

23 public static void main(String args[]) {
     initSystem();
 }

}
28

public class Control16 {
/* codigo do Control16 nao detalhado aqui ... */
.
33 .
.
/* Metodos de Testes */
 public static void testControl(){
  if ((poscar==0x3c00)&&(alfa==0x2e66)){
38  if ((y!=0) && (z!=0) &&(u!=0)){

    if (EmergencyStop){
     if (VC_temp==0)
      Crane.testControl = 0;
43  }else {
     int num1 = SoftFloat16.float16_sub(u,y);

     if (SoftFloat16.float16_lt(0x5100, num1) !=1){
      if (VC_temp== 0x5100)
48     Crane.testControl = 0;
     }else  if (SoftFloat16.float16_lt(num1, 0xd100) ==1){
      if (VC_temp== 0xd100)
       Crane.testControl = 0;
     }else{
53    if (VC_temp== num1)
       Crane.testControl = 0;
     }
    }

58  }
```

```
      for (i=0; i < 5; i++) {
       if ((q[i]==0)||(q1[i]==0))
        Crane.testControl = 1;
      }
63    }
      //System.out.println("testControl = " + Crane.testControl);
    }
  //}

68 public class Sensor16{
   // codigo do Sensor16 nao detalhado aqui ...
   .
   .
   .
73  public static void testReadSensors(){
     if ((alfa==0x2e66)&&(poscar==0x3c00)){
      Crane.testReadSensors = 0;
     }
     //System.out.println("testReadSensors = " + Crane.testReadSensors);
78
    }
    public static void testDiagnosis(){
     if ((counter == 0) && (sensor_warning==0)&&(alfa_warning==0)){
      Crane.testDiagnosis = 0;
83   }else if (counter >=2){
       if (a== (SoftFloat16.abs(0x2e66))){
        if (alfa_warning >= 2){
         if (SoftFloat16.float16_lt(a,AlfaMax)!=1)
          Crane.testDiagnosis = 0;
88      }
       }
       if (sensor_warning >= 2){
        if (swposcarmax || swposcarmin){
         Crane.testDiagnosis = 0;
93      }else{
         Crane.testDiagnosis = 1;
        }
       }

98   }
     //System.out.println("testDiagnosis = " + Crane.testDiagnosis);
    }
   }
```

Listing 6.3: Casos de teste selecionados para o Crane