PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

INTEGRAÇÃO E AVALIAÇÃO DE TÉCNICAS DE TESTE BASEADO EM SOFTWARE NO FLUXO DE PROJETO DE SOCS

por

ALEXANDRE DE MORAIS AMORY

Dissertação de mestrado submetida como requisito parcial
à obtenção do grau de Mestre em Ciência da Computação.

Prof. Dr. Fernando Gehm Moraes
Orientador

Porto Alegre, Janeiro de 2003

À minha mãe Sonia e à minha namorida Letícia.

"A mente que se abre a uma nova idéia jamais volta ao seu tamanho original"
Albert Einstein

"Não! Tentar não. Faça. Ou não faça. Não existe tentar"
Yoda

"O defeito só aparece depois que a unidade passou pela inspeção final"
Lei de Murphy

Resumo

Com o aumento da densidade e da freqüência de operação dos circuitos integrados, equipamentos de teste mais sofisticados e caros são necessários para garantir a qualidade dos dispositivos fabricados. Para reduzir as restrições tecnológicas destes equipamentos de teste e o custo total de teste, tem-se pesquisado formas de executar partes do teste dentro do próprio dispositivo. A abordagem mais comum e a adição de módulos de hardware projetados para realizar partes da etapa de teste. Esta abordagem é conhecida como BIST, *built-in selt-test*.

Entretanto, a adição destes módulos implica em custos principalmente em área de hardware e desempenho. Os atuais circuitos integrados complexos (SOC) são projetados, em geral, com um ou mais processadores embarcados. Baseado na capacidade de programação destes processadores e nas técnicas de teste de hardware, este trabalho busca estudar formas alternativas de inserir funcionalidades de teste no SOC de forma a minimizar os custos de teste.

A técnica estudada é chamada de teste baseado em software, onde um código escrito em uma linguagem de programação é responsável pelas principais tarefas do teste. Entretanto, sabe-se que o uso desta técnica também implica em certos custos, como maior requisito de memória de teste e tempo de teste. Os principais objetivos deste trabalho são avaliar os custos da técnica de teste baseada em software em relação ao teste baseado em hardware (i.e. BIST) e construir um ambiente para teste baseado em software. Os resultados preliminares desta avaliação mostram um pequeno aumento dos requisitos de memória e um aumento significativo no tempo de teste, quando comparado com a implementação puramente hardware.

Abstract

With the increasing transistor density and operating frequency of integrated circuits, more sophisticated and expensive test equipments are required to guarantee the quality of the manufactured devices. To reduce the technological requirements of those testers and the overall test cost, new embedded test methods have been investigated. The most common approach is to add embedded hardware modules to perform some part of the test procedure. This approach is popularly known as BIST, *built in self-test.*

However, the addition of these test modules results in area and performance overhead. State-of-the-art system-on-a-chip designs include, in general, at least one embedded processor. Based on the inherent programmability of processors and the well-established hardware test techniques, this work aims to study alternative methods to insert embedded test into the cores comprising the SOC under development.

This work studies software-based test for test of SOCs. Although, the use of this technique implies in some costs like additional memory requirements and test time. The main objectives of this work are to evaluate the costs of software-based test against the hardware-based test (i.e. BIST) and to develop an environment to integrate test tools and methods targeted to software-based techniques. Preliminary results point out to small amount of additional test memory and increased test time, when compared to purely hardware-based techniques.

Agradecimentos

Inicialmente gostaria de agradecer a Capes pelo apoio financeiro para a realização deste trabalho possibilitando minha dedicação íntegra ao mesmo, que sempre foi o meu desejo. Também gostaria de agradecer ao suporte técnico da Mentor Graphics e da Altera pela prontidão para responder algumas dúvidas técnicas.

Mudando agora de Entidades para Pessoas, muitas pessoas contribuíram para a realização deste trabalho. Alguns de forma indireta, através da sua amizade, carinho, bom humor, bom astral e conselhos. Outras que literalmente colocaram a "mão na massa" junto comigo.

Agradeço a Érika Cota, Fabian Vargas, Fabiano Hessel, Marcelo Lubaszewski, Marcon e Ney Calazans pelos comentários, críticas e sugestões que ajudaram a enriquecer este trabalho.

Em particular, além do orgulho desta conquista obtida, me sinto também orgulhoso dos diversos Amigos que conheci. Quero agradecer a todos colegas do mestrado e do grupo GAPH pelo excelente e agradável, porque não dizer prazeroso ???, ambiente de trabalho. Alguém mais céptico poderia perguntar "Como o trabalho pode ser prazeroso? Sempre é tão cansativo e monótono. As pessoas sempre reclamam de seus trabalhos. Que história é esta de trabalho prazeroso?". Sinceramente não sei responder, mas foi. Talvez seja aquele bom dia entusiasmado, talvez aquele convite a uma partida de fla-flu ou de D, talvez aquele bate-papo depois que você cansou de algo. Muito possivelmente tenha influência das nossas cervejadas sempre alegres e das famosas e inesquecíveis "casas dos mestrandos". Sei lá. Acho que é tudo um pouco, ou não. Por exemplo. Muitas vezes, no dia-a-dia, tive uma certa resistência a acordar cedo. Mas quando me lembrava que estava fazendo o que eu queria e que eu iria encontrar meus Amigos, tudo ficava mais fácil e eu até conseguia fazer este esforço. Te pergunto, como alguém não vai gostar de um ambiente assim ?. Deixo aqui registrado meu agradecimento, um tanto incomum, a estes Amigos. Desejo a vocês todo o sucesso e felicidade.

Agradeço a meu orientador Fernando Moraes e a meu fiel escudeiro Leandro Oliveira. Quanto ao Leandro, agradeço pela sua dedicação neste trabalho. Sei que a vida de bolsista não é fácil e recebe pouco. Por este motivo, por não escolher o caminho fácil (ver próximo parágrafo), e pela qualidade e volume do seu trabalho, eu o agradeço. Quanto ao Moraes, tive a felicidade de encontrar a seguinte passagem de um livro de Gandhi "*... I have always felt that the true text-book for the pupil is his teacher. I remember very little that my teachers taught me from books, but I have even now a clear recollection of the things they taught me independently of books.*". Eu acho que isto resume tudo. Agradeço os conhecimentos técnicos passados, mas acho, sinceramente, que não valeu tanto quanto a oportunidade de trabalhar contigo. Com um pouco de auto-disciplica eu até conseguiria sozinho um pouco de conhecimento "técnico". Seria mais vazio, mas conseguiria. Porém, o ensinamento "perpétuo" é mais sutil. Nunca encontrei um livro que ensinasse postura, seriedade e ética profissional e compromisso com a Verdade. Tenho a impressão que este último tem caído fora de moda, dando lugar à filosofia "*publish or perish*". Muitas vezes nem é por maldade, mas retira-se um resultado ruim aqui, modifica-se um dado ali, esconde-se as desvantagens. Isto não deve atrapalhar ninguém e deve possibilitar uma publicação a mais. Porém, este tipo de coisa pode parecer atraente, mas não faz parte dos meus princípios, aprendidos com ele.

Como não poderia ser diferente. Quero agradecer de todo o meu coração por todo amor, carinho, apoio e incentivo dado por minha família. Isto foi primordial para eu alcançar este entre diversos outros objetivos da minha vida. Não quero que esta "homenagem" a vocês seja vazia, mas como vou agradecer tudo que fizeram por mim por escrito ? Eu não sei. Por exemplo, mesmo antes de entrar no mestrado nos vemos em um dilema importante. Apesar do auxílio financeiro oferecido por algumas instituições de fomento à pesquisa, o valor da bolsa é reduzido. No mercado existem ofertas que oferecem algumas vezes 3 ou 4 vezes mais que a bolsa. Antes de fazer a inscrição no mestrado você para e analisa a situação. "Pô, tenho isso e aquilo para pagar, aluguel custa x, comida y, demais gastos z. x+y+z+isso+aquilo dá $$$$. A bolsa é $$. Acho que não vai dar". Você vai e

apresenta a situação para quem ? sua família. Muitas famílias poderiam simplesmente dizer: “É, realmente não dá”. Isto acontece e nem sempre é uma simples opção. Porém, minha família me apoiou pois sempre souberam da importância deste “próximo passo” para mim. Paramos, sentamos, calculamos, calculamos e calculamos ainda mais. Aperta daqui, tira dali, move para lá. Quase deu. Novamente aperta daqui, tira dali, move para lá. Surpresa !!! É, reconheço que não foi fácil. Poderíamos ter escolhido o caminho mais fácil, mas não. Pode ser teimosia, talvez, mas este valor, lutar pelo que se deseja, sempre foi uma característica da minha família. Agradeço a Ariela, Camila, Letícia e Sonia por tudo. Muito obrigado!!!

Sumário

1 INTRODUÇÃO 1

1.1 MOTIVAÇÃO 4
1.2 OBJETIVOS 5
1.3 ORGANIZAÇÃO DO DOCUMENTO 5

2 PROJETO VISANDO O TESTE 7

2.1 CONCEITOS BÁSICOS 7
2.1.1 Confiabilidade e Teste 7
2.1.2 Verificação e Teste 7
2.1.3 Defeito, Falha e Erro 8
2.1.4 Modelos de Falhas 9
2.1.5 Padrão de Teste 9
2.1.6 Métricas de Teste 10
2.1.7 Taxionomia de Teste 10
2.1.8 Rendimento e Nível de Defeito 11
2.2 PROJETO VISANDO A TESTABILIDADE 13
2.2.1 Técnicas *Ad Hoc* 13
2.2.2 *Scan Path* 13
2.3 AUTO-TESTE 14
2.3.1 Geração de Vetores de Teste 15
2.3.2 Compactação de Teste 16
2.3.3 Teste de Memória 18
2.3.4 Teste de Processador 20
2.4 CONCLUSÃO 21

3 TESTE DE DISPOSITIVOS BASEADOS EM NÚCLEOS DE HARDWARE 23

3.1 ARQUITETURA CONCEITUAL DE TESTE DE SOC 24
3.1.1 Fonte e Receptor de Informações de Teste 24
3.1.2 Mecanismo de Acesso de Teste 24
3.1.3 Lógica Envoltória 24
3.2 PADRÃO P1500 25
3.2.1 Linguagem de Teste de Núcleos 25
3.2.2 Arquitetura Escalonável para Teste de Núcleos 26
3.2.3 Exemplo de Uso da Lógica Envoltória 27
3.3 CONCLUSÃO 29

4 TÉCNICAS DE TESTE BASEADAS EM SOFTWARE PARA DISPOSITIVOS BASEADOS EM NÚCLEOS DE HARDWARE 31

4.1 VANTAGENS E DESVANTAGENS 31
4.2 TESTE DOS COMPONENTES DE SOCS 32
4.2.1 Teste de Processadores 32
4.2.2 Teste de Memória 34
4.2.3 Teste de Núcleos de Hardware não Programáveis 35
4.3 CONCLUSÃO 37

5 CO-SIMULAÇÃO 39

5.1 MODELAGEM DE SISTEMAS COMPUTACIONAIS 39

5.2 CARACTERÍSTICAS DE UM AMBIENTE DE CO-SIMULAÇÃO 40
5.3 AMBIENTE DE CO-SIMULAÇÃO DESENVOLVIDO 43
5.3.1 Estrutura do Roteador 43
5.3.2 Bibliotecas de Comunicação 45
5.4 ESTUDOS DE CASO 46
5.4.1 Validação do Ambiente 47
5.4.2 Avaliação de Desempenho do Ambiente de Co-simulação 49
5.4.3 Co-simulação para Validação de Lógica Envoltória 52
5.5 CONCLUSÃO 54

6 AMBIENTE DE PROTOTIPAÇÃO 57

6.1 ORGANIZAÇÃO DOS MÓDULOS DE HARDWARE 57
6.2 BARRAMENTO 58
6.3 INTERFACE DE HARDWARE E SOFTWARE 60
6.4 ARQUITETURA ALVO PARA AVALIAÇÃO 60

7 AMBIENTE PARA INTEGRAÇÃO DE TESTE NO FLUXO DE PROJETO DE SOCS 63

7.1 DESCRIÇÃO DE GERADORES DE PADRÕES E COMPACTADORES DE RESPOSTAS EM SOFTWARE 64
7.1.1 Validação da Descrição de LFSR e MISR em Software 67
7.2 FERRAMENTA DE GERAÇÃO DE CÓDIGO 67
7.3 FERRAMENTA DE INTEGRAÇÃO DE NÚCLEOS DE HARDWARE 71
7.3.1 Criação de Arquivos de Projetos de Núcleos 71
7.3.2 Criação de Projeto de SOCs 73
7.3.3 Lógica Envoltória 74
7.3.4 Interface do SOC 75
7.4 FERRAMENTA DE AUXÍLIO AO TESTE BASEADO EM SOFTWARE 76
7.5 LIMITAÇÕES 80
7.5.1 Quanto aos Núcleos de Hardware 80
7.5.2 Quanto aos Núcleos de Hardware do Tipo Memória 80
7.5.3 Quanto à Arquitetura de Interconexão dos Núcleos 80
7.5.4 Quanto ao Processador Embarcado 80
7.5.5 Quanto às Ferramentas de CAD 81
7.5.6 Quanto à Lógica Envoltória 81
7.5.7 Quanto às Cadeias de Varredura 81
7.5.8 Quanto ao Ambiente de Teste 81
7.6 CONCLUSÕES 81

8 AVALIAÇÃO DE TESTE BASEADO EM SOFTWARE 83

8.1 ESTUDOS DE CASO 83
8.2 ANÁLISE DE ÁREA DE MEMÓRIA 84
8.3 ANÁLISE DE TEMPO DE TESTE 86

9 CONCLUSÃO 89

10 ANEXOS 91

10.1 CLASSES PRINCIPAIS DA FERRAMENTA DE TESTE 91
10.2 EXEMPLOS DE CÓDIGOS GERADOS AUTOMATICAMENTE 95
10.2.1 Lógica Envoltória 95
10.2.2 *Testbench* da Lógica Envoltória 98
10.2.3 Interface do Soc 103

10.2.4 Código de Teste 105
***10.3 SCRIPTS* DE INTEGRAÇÃO COM FERRAMENTAS DE CAD 107**
10.3.1 Modelsim 107
10.3.2 Leonardo 109
10.3.3 Flextest e FastScan 110

11 REFERÊNCIAS BIBLIOGRÁFICAS 111

Lista de Tabelas

Tabela 1 – Estados pseudo aleatórios gerados pelo LFSR de quatro estágios. *16*
Tabela 2 – Seqüência de passos de um LFSR na geração de assinatura. *18*
Tabela 3 – Algoritmos de teste de memória e suas complexidades [54]. *19*
Tabela 4 – Estado dos multiplexadores durante os modos de operação [48]. O valor x significa que este multiplexador não importa neste modo de operação. *29*
Tabela 5 – Tempo de execução, em segundos, de diferentes mecanismos de validação *51*
Tabela 6 – Tempos de co-simulação, em segundos, das duas partições *51*
Tabela 7 – Tempo de co-simulação (s) em uma máquina, em uma LAN e em uma WAN. *51*
Tabela 8 – Exemplo de tabela do tipo Token. *68*
Tabela 9 – Tabela de símbolos do exemplo de testbench da lógica envoltória *71*
Tabela 10 – Número de linhas de código do ambiente de teste. *82*
Tabela 11 – Características gerais dos estudos de caso. *83*
Tabela 12 – Cobertura de falhas e número de padrões de teste. *84*
Tabela 13 – Tamanho total do código de teste em bytes *84*
Tabela 14 – Relação entre tamanho do código e tempo de teste. *85*
Tabela 15 – Tamanho da configuração de LFSR e dos padrões determinísticos em palavras de 32 bits. *86*
Tabela 16 – Tamanho das funções (em bytes) do código de teste quando o LFSR e o MISR estão implementados em software *86*
Tabela 17 – Tempo, em ciclos de relógio, entre a geração de um padrão de teste pseudo-aleatório ou determinístico *87*
Tabela 18 – Tempo de teste em ciclos de relógio utilizando diferentes abordagens. *87*

Lista de Figuras

Figura 1 – Modelo do time to market [54] 1
Figura 2 – Déficit de produtividade [68] 2
Figura 3 – Fluxo básico de codesign [33]. 3
Figura 4 – Custo de teste versus custo de desenvolvimento[68] 3
Figura 5 – Definição de defeito e falha. (a) Nível de leiaute. (b) Nível de transistor. (a) Nível de porta lógica [22] 8
Figura 6 – Classificação das falhas em CIs. 8
Figura 7 – Falhas stuck-at em uma porta lógica E [54] 9
Figura 8 – Teste pseudo-exustivo [54]. 10
Figura 9 – Nível de defeito como uma função do rendimento e da cobertura de falhas [1] 12
Figura 10 – Teste funcional+scan vs scan vs curva Williams. 13
Figura 11 – Arquitetura básica de scan path. 14
Figura 12 – Funcionamento do scan path 14
Figura 13 – Arquitetura de BIST genérica 15
Figura 14 – LFSR de n-1 estágios. 15
Figura 15 – Exemplo de LFSR de quatro estágios. 16
Figura 16 – Simulação da descrição VHDL de LFSR compactador de polinômio x^4+x^3+1. 16
Figura 17 – Circuitos de compactação de respostas de teste. 17
Figura 18 – LFSR compactador serial de polinômio $x^5 + x^3 + x + 1$. 17
Figura 19 – Simulação do LFSR compactador de polinômio $x^5 + x^3 + x + 1$. 18
Figura 20 – Cobertura de falhas de diversos algoritmos de teste de memória [75] 20
Figura 21 – Implementação de BIST para microprocessadores [65]. 21
Figura 22 – Arquitetura genérica de acesso de teste para núcleos[48]. 24
Figura 23 – Visão das interfaces internas e externas da lógica envoltória padrão no nível do núcleo [48]. 26
Figura 24 – Arquitetura da lógica envoltória padrão. Visão ao nível de sistema, composto por vários núcleos [48] 27
Figura 25 – Exemplo do núcleo A mais a lógica envoltória. 28
Figura 26 – Os vários modos de operação da lógica envoltória [48] 28
Figura 27 – Procedimento em linguagem assembly do algoritmo March de teste de memória [65] 34
Figura 28 – Motor de simulação. 40
Figura 29 – Modelo de sincronização. 41
Figura 30 – Arquitetura do ambiente de co-simulação 43
Figura 31 – Arquivo de coordenação do multiplicador 44
Figura 32 – Diagrama de classes simplificado do roteador. 45
Figura 33 – Estrutura geral do sistema de co-simulação em relação a bibliotecas utilizadas. 46
Figura 34 – Entidade do componente de comunicação de VHDL. 47
Figura 35 – Representação do testbench do multiplicador. 47
Figura 36 – Estrutura do testbench do multiplicador. 48
Figura 37 – Código fonte do software de multiplicador. 48
Figura 38 – Captura de tela da co-simulação hardware/software. 49
Figura 39 – Algoritmo de preenchimento de polígonos 50
Figura 40 – Particionamentos do estudo de caso de preenchimento de polígonos. 50
Figura 41 – Arquivo de coordenação do projeto da lógica envoltória 52
Figura 42 – Estrutura do código de teste do núcleo c880 e da lógica envoltória. 53
Figura 43 – Estrutura do testbench da lógica envoltória 53
Figura 44 – Co-simulação do projeto de lógica envoltória 54
Figura 45 – Modelo de interface hardware/software no ambiente Excalibur. 58
Figura 46 – Representação das operações de leitura e escrita realizadas pelo processador Nios. 59
Figura 47 – Exemplo de uma operação de escrita do processador. 59
Figura 48 – Exemplo de uma operação de leitura do processador. 59
Figura 49 – Exemplo de software utilizado para acessar o hardware. 60
Figura 50 – Estrutura genérica do SOC utilizada neste trabalho. 61
Figura 51 – Fluxo de desenvolvimento. 63
Figura 52 – Fluxo de teste. 63
Figura 53 – Exemplo de LFSR modular genérico. 64
Figura 54 – Equação de um LFSR modular genérico [14] 65
Figura 55 – Equação e circuito de um LFSR modular para $f(x) = 1 + x^2 + x^7 + x^8$. 65
Figura 56 – Rotina de LFSR modular desenvolvida. 66
Figura 57 – Exemplo de MISR modular genérico. 66
Figura 58 – Equação de um MISR modular genérico [14] 66
Figura 59 – Rotina desenvolvida de MISR modular. 66
Figura 60 – Instruções do Maple V para emular um LFSR modular. 67

Figura 61 – Instruções do Maple V para emular um MISR modular. *67*
Figura 62 – Representação dos módulos de geração de código. *68*
Figura 63 – Exemplo de estrutura de um gerador de código específico (xCodeGen). *69*
Figura 64 – Exemplo de estrutura de um gerador de símbolos (xTokenGen). *69*
Figura 65 – Gerador de código genérico (CodeGen). *70*
Figura 66 – Exemplo de trecho do template de testbench da lógica envoltória *70*
Figura 67 – Exemplo de trecho de código gerado do testbench da lógica envoltória. *71*
Figura 68 – Exemplo de arquivo de projeto de núcleos. *72*
Figura 69 – Telas do gerador de projeto de núcleos. *72*
Figura 70 – Exemplo de arquivo de projeto de SOCs. *73*
Figura 71 – Telas do gerador de projeto de SOCs. *74*
Figura 72 – Esquema de uma lógica envoltória. *75*
Figura 73 – Detalhe da criação da interface do SOC *75*
Figura 74 – Representação do processo de geração de código de teste. *76*
Figura 75 – Telas da ferramenta de geração de código de teste (referente a Figura 74(b)). *77*
Figura 76 – Exemplo de arquivo de configuração de LFSR. *78*
Figura 77 – Formato tabular gerado pelo Modelsim. *78*
Figura 78 – Exemplo de padrões de teste no formato ASCII suportado por Flextest e Fastscan *79*
Figura 79 – Formato de vetores utilizado para geração de assinatura do módulo. *79*
Figura 80 – Classe BaseCore. *91*
Figura 81 – Classe Core. *92*
Figura 82 – Classe Soc. *93*
Figura 83 – Registros de portas e generics de uma entidade. *94*
Figura 84 – Arquitetura da lógica envoltória gerada automaticamente. *97*
Figura 85 – Arquitetura do testbech gerado automaticamente. *102*
Figura 86 – Script do Modelsim para geração de padrões pseudo-aleatórios. *107*
Figura 87 – Script do Modelsim para geração da assinatura esperada. *107*
Figura 88 – Script do Modelsim para compilar um SOC completo *108*
Figura 89 – Script para síntese lógica de um SOC completo. *109*
Figura 90 – Script de avaliação de padrões pseudo-aleatórios. *110*
Figura 91 – Script de geração de padrões determinísticos *110*

Lista de Abreviaturas

ASIC	*Application Specific Integrated Circuit*
ATE	*Automatic Test Equipment*
ATPG	*Automatic Test Pattern Generator*
BIST	*Built-In Self-Test*
CAD	*Computer Aided Design*
CI	Circuito Integrado
CUT	*Core Under Test*
DFT	*Design for Testability*
DPM	*Defects per Million*
DSP	*Digital Signal Processor*
FF	*Flip-Flop D*
FIFO	*First in First Out*
FPGA	*Field Programmable Gate Array*
LDU	Lógica Definida pelo Usuário
LFSR	*Linear Feedback Shift Register*
LUT	*Look-Up Table*
MISR	*Multiple Input Signature Register*
MSN	Módulo do Sistema Nios
RAM	*Random Access Memory*
ROM	*Read Only Memory*
SDL	*Synchronous Design Language*
SEU	*Single Event Upset*
SIA	*Semiconductor Industry Association*
SOC	*System On a Chip*
SSA	*Single Stuck-at Fault*
TAM	*Test Access Mechanism*
VHDL	VHSIC *Hardware Description Language*

1 INTRODUÇÃO

O constante avanço tecnológico dos semicondutores, definido na lei de Moore [66], e o aumento do mercado de dispositivos eletrônicos tem impulsionado o desenvolvimento de sistemas computacionais em um único circuito integrado (CI). Tais sistemas são tipicamente compostos de milhões de transistores que englobam hardware digital e analógico. Como exemplos de módulos que compõem um *system-on-a-chip* (SOC) podemos citar microprocessadores, memória RAM e ROM, DSP, Conversores D/A e A/D, LDU (lógica definida pelo usuário) e até mesmo tecnologias mais recentes como MEMS (*MicroElectroMechanical Systems*) e eletro-ótica [33][46].

Com o avanço da tecnologia e da densidade dos CIs, do consumo de dispositivos eletrônicos e da heterogeneidade dos sistemas, é necessário um maior esforço no que diz respeito a concepção e a validação do projeto. Tal aumento de esforço implica em aumento de tempo e custo de projeto. Por outro lado, o rápido avanço tecnológico faz com que o ciclo de vida dos produtos diminua, fazendo com que o tempo para lançamento de um produto (em inglês, *time to market*) seja um dos parâmetros mais importantes com relação ao custo do produto final. A Figura 1 informa que uma variação no *time to market*, ΔT, acarreta em uma redução do total de lucros, representada pela área em branco. Porém, este modelo não é completo, pois não informa a perda de lucro relacionada à rejeição de produtos defeituosos. Portanto, é importante encontrar um compromisso entre atraso de lançamento do mercado e qualidade. Desta forma, existe uma grande pressão para reduzir o tempo de criação e lançamento de um CI no mercado antes que o mesmo já esteja ultrapassado tecnologicamente. Porém, sabe-se que existe um déficit crescente de produtividade, conforme a Figura 2. Este gráfico relaciona o número de transistores por CI (complexidade) e o número de transistores que podem ser desenvolvidos por homem-mês (produtividade). Enquanto que a complexidade dos CIs aumenta cerca de 58% ao ano, a produtividade dos projetistas aumenta somente 21% [68]. Técnicas de reuso de módulos e padronização de interface entre estes, amenizam o déficit de produtividade, mas não são suficientes [77]. Outra solução explorada é a crescente automatização do projeto através de ferramentas computacionais, chamadas ferramentas de CAD (*Computer Aided Design*).

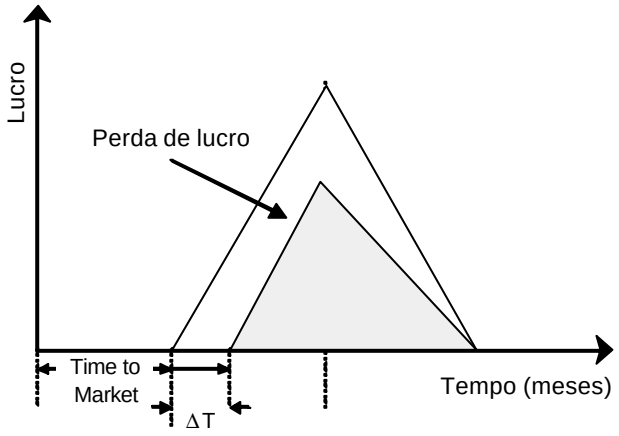


Figura 1 – Modelo do *time to market* [54].

A necessidade de reuso para aumentar a produtividade se reflete na previsão da *Semiconductor Industry Association* (SIA) que prediz que em 2005 90% da área dos CIs será composta por núcleos de hardware (em inglês, *cores*) [68]. Estes núcleos são descrições de hardware complexo pré-projetadas e pré-verificadas que podem ter sido implementadas pelo próprio usuário ou por terceiros [31].

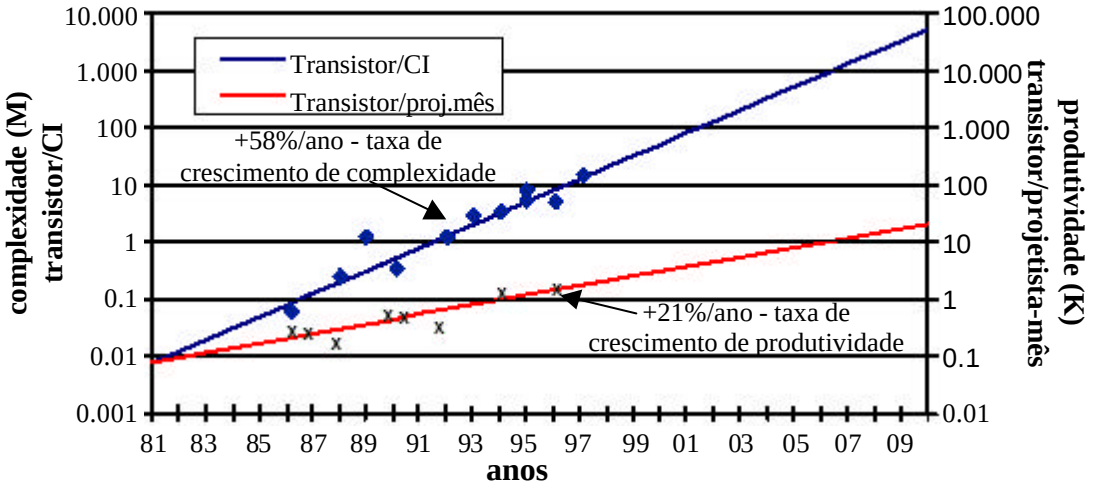


Figura 2 – Déficit de produtividade [68].

Para o projetista acompanhar a evolução da complexidade de projeto, técnicas mais abstratas devem ser desenvolvidas, baseadas no suporte de ferramentas de CAD. O projeto conjunto de hardware *e* software (em inglês, *hardware/software codesign*), visa o desenvolvimento de ferramentas de CAD para automatizar o processo de concepção paralela das partes de software e de hardware de um sistema computacional. O objetivo de *codesign* é encontrar a melhor partição entre hardware e software a partir de uma especificação do sistema que atenda às restrições desta. Como dito anteriormente, não são incomuns sistemas compostos por dezenas de módulos que compreendem hardware, software, memória, sub-sistemas analógico e eletro-mecânico [33][46]. Estes sistemas são chamados de sistemas heterogêneos. Para tornar gerenciável a complexidade imposta pela quantidade e heterogeneidade dos módulos são necessárias novas metodologias de concepção e de especificação de sistemas [33]. A metodologia de concepção de sistemas computacionais heterogêneos utilizando técnicas abstratas de desenvolvimento possui vantagens como:

- Uso de especificação de projeto realizada no nível sistêmico, que permite iniciar o desenvolvimento de uma especificação com pequeno nível de detalhes, sendo os mesmos incrementados no decorrer do projeto;
- Curto ciclo de concepção, atendendo restrições de tempo necessárias para a chegada do produto no mercado. Isto é possível devido à adoção do fluxo concorrente de concepção de hardware e software, ao invés do fluxo seqüencial tradicional.

A Figura 3 apresenta um fluxo básico de *codesign* [23][33]. O projeto inicia com uma especificação funcional do sistema em uma ou várias linguagens. O objetivo da co-simulação sistêmica é validar e explorar os algoritmos e a funcionalidade do sistema. A próxima etapa, executada por uma ferramenta de co-síntese, é responsável por produzir descrições de componentes de hardware e software e também por selecionar o protocolo de comunicação entre estes. A co-simulação arquitetural visa validar as descrições de hardware e software geradas, incluindo as primitivas abstratas de comunicação. O nível de ciclo, que utiliza ferramentas de síntese de hardware e compilação de software, mapeia as descrições de hardware e software para a arquitetura e processador alvo, respectivamente. As interfaces de comunicação são implementadas em hardware dedicado. A co-simulação no nível de ciclo utiliza um simulador HDL para a parte de hardware e de comunicação e um simulador do processador alvo para a parte de software para a validação do sistema completo. Uma validação mais precisa pode ser obtida através da prototipação do sistema em, por exemplo, dispositivos FPGA (*Field Programmable Gate Array*).

Como dito anteriormente, as etapas de concepção e verificação de um projeto compõem a maior parte do tempo de desenvolvimento de CIs. Porém, conforme a Figura 4, é previsto que próximo de 2015 o custo de teste de um transistor irá exceder o custo de desenvolvê-lo, a menos que outras técnicas de teste sejam empregadas [68].

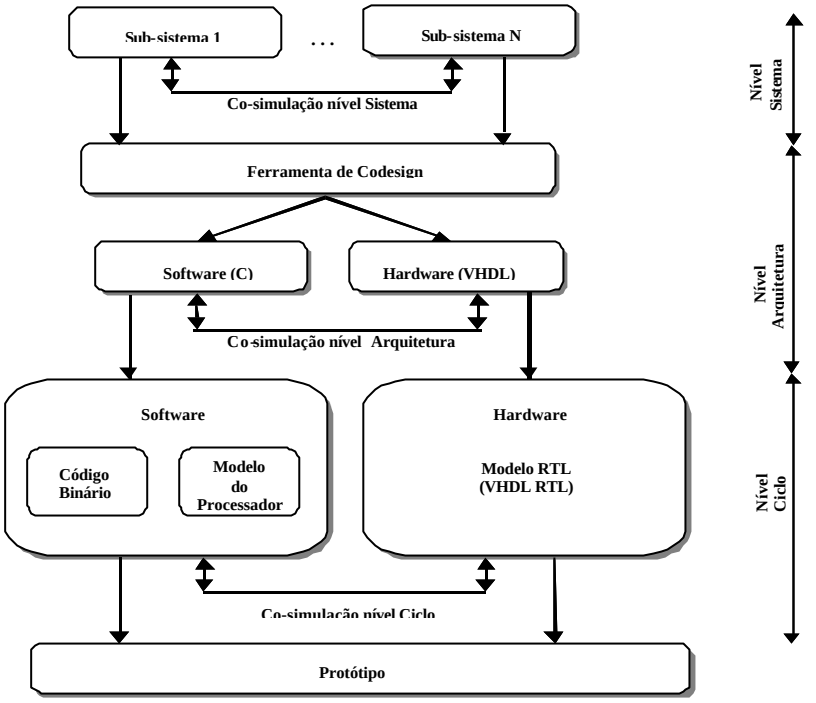


Figura 3 – Fluxo básico de *codesign* [33].

A Figura 4 apresenta duas curvas, uma que representa o custo de fabricação por transistor ao longo dos anos, e a outra representa o custo de teste de um transistor também ao longo do tempo. Conforme estas curvas, é previsto que próximo de 2012 o custo de fabricação esteja próximo do custo de teste de um sistema. Dentre os motivos que levam ao aumento de custo de teste podemos citar o aumento de número de transistores por pino, o que dificulta o acesso e o controle dos nós internos do CI por um equipamento de teste (em inglês, *Automatic Test Equipment* - ATE). Por outro lado, o desenvolvimento acelerado da tecnologia e o conseqüente aumento na freqüência de operação dos dispositivos, chegando a freqüências da ordem de GHz, tem tornado cada vez mais difícil realizar teste em tempo de execução, uma vez que os próprios equipamentos de teste devem acompanhar o estado-da-arte. Isto acarreta alto custo de tais equipamentos. A SIA prevê que um equipamento de teste estado-da-arte custará mais de 20 milhões de dólares em 2014 [68]. Além disto, o tempo de teste utilizando estes aparelhos levará horas por dispositivo devido ao aumento de nós internos que devem ser testados e a baixa largura de banda dos testadores. Para resolver este problema, tem-se pesquisado fluxos de projeto que visam facilitar a etapa de teste após a fabricação do CI. O nome deste conjunto de técnicas é *projeto visando a testabilidade* (em inglês, *Design for Testability* - DFT).

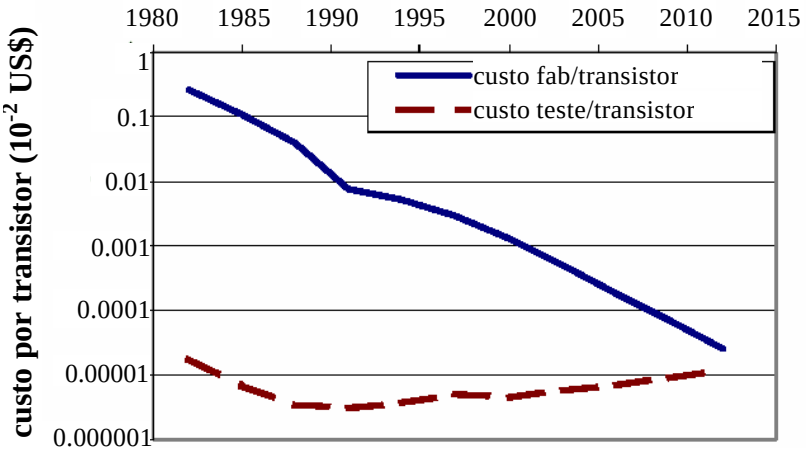


Figura 4 – Custo de teste versus custo de desenvolvimento[68].

1.1 Motivação

Diversos exemplos de aplicações desenvolvidas em SOCs podem ser encontrados não somente no ambiente acadêmico mas também na indústria. Na maioria destes casos existe algum módulo programável como um processador, processador de sinais digitais (*digital signal processor* – DSP) ou microcontrolador. A presença deste tipo de módulo em um SOC confere ao mesmo características de programabilidade, flexibilidade, reuso de software e hardware.

Por outro lado, existe a necessidade de garantir a qualidade, robustez, confiabilidade e disponibilidade dos dispositivos desenvolvidos. Técnicas de teste descritas em hardware são utilizadas para garantir estas qualidades aos dispositivos. Uma vez que os SOCs contêm normalmente ao menos um processador, estas mesmas técnicas de teste podem ser implementadas em software, atingindo os quesitos de flexibilidade e confiabilidade desejados. Explorar rotinas de software que testem hardware pode trazer benefícios como:

- A *ausência ou redução da área de silício adicional* para executar o teste, diminuindo o custo do produto final e não degradando o *desempenho* do produto;
- A possibilidade de executar múltiplos algoritmos de teste resulta em uma *maior qualidade de teste.* O número de algoritmos aplicados seria limitado somente pela restrição de tempo de teste da especificação. Benso *et. al.* [10] argumentam que o uso de múltiplos algoritmos de teste implementados em hardware aumenta a cobertura de falhas do sistema, portanto sua confiabilidade;
- A *flexibilidade* intrínseca do software que possibilita aplicar diferentes rotinas de teste em situações ou períodos de tempos variados. Por exemplo, na inicialização do dispositivo um teste mais robusto (com maior cobertura de falhas) poderia ser aplicado. Já durante o funcionamento, poderia ser escolhida uma rotina de teste com menor cobertura, mantendo um compromisso entre confiabilidade e tempo de teste. Outra vantagem da flexibilidade é a facilidade de trocar o algoritmo de teste. Hellebrand *et. al.* [32] afirmam que a flexibilidade do software favorece a escolha do melhor método de geração de estímulos de teste para cada módulo a ser testado;
- O *reuso* das rotinas de teste é facilitado por serem implementadas em software. Uma biblioteca de software de teste escrita em linguagem C é facilmente aplicável a um processador;
- A ausência ou redução de *lógica de controle de teste,* pois o próprio processador se encarrega do fluxo de teste.

As desvantagens são:

- SOC deve possuir um *componente programável* como processador, microcontrolador ou DSP;
- Tamanho do *programa de teste* e tamanho da *memória* para armazenar programa de teste e padrões de teste;
- Software de teste deve ser considerado livre de falhas;
- O possível aumento do *tempo de teste.*

Na revisão bibliográfica apresentada na Seção 4.2 foram encontradas algumas aplicações acadêmicas [18][72] e industriais [55] de teste baseado em software. Porém, a *automatização* e *avaliação* deste processo ainda são incompletas. Por exemplo, existem abordagens acadêmicas que automatizam a geração de código de teste para processadores e aplicações de software para testar memória. Porém, não foram encontrados trabalhos que automatizassem o processo de geração de código de teste para núcleos não programáveis, tampouco um *framework* que automatizasse o teste de um SOC completo considerando apenas módulos digitais. No que concerne a avaliação de técnicas de teste baseadas em software, não existem trabalhos que demonstrem de forma quantitativa suas vantagens e, principalmente, suas desvantagens. Esta lacuna impede a definição objetiva da aplicabilidade desta técnica de teste, deixando sem respostas perguntas como: *"Teste*

baseado em software pode ser utilizado para teste de fabricação?" ou *"Esta técnica pode ser aplicada em teste de campo? Quais são as penalidades e vantagens ?"*. Provavelmente não existe uma resposta única para estas perguntas devido à heterogeneidade e variações de restrições de cada projeto.

1.2 Objetivos

O objetivo estratégico deste trabalho é realizar a avaliação de teste de hardware baseado em software, e responder às perguntas acima utilizando estudos de caso. Desta forma, alguns *objetivos específicos* moldaram a busca pelo objetivo maior, como por exemplo:

- O desenvolvimento de *técnicas de teste flexíveis* para SOCs com arquitetura de interconexão por barramento;
- O desenvolvimento de técnicas de *teste de hardware digital* que possam ser aplicadas tanto em *teste de produção* quanto em *teste de campo*. Os SOCs podem possuir módulos analógicos, eletro-mecânicos e eletro-óticos. Porém, este trabalho limita-se ao uso de memórias RAM, ROM, núcleos de hardware não programáveis e microprocessadores simples, e.g. 8051;
- Explorar o teste de SOCs utilizando *somente técnicas de teste baseadas em software* ou *misto hardware/software*. Deve-se identificar, e não necessariamente implementar, formas de minimizar as principais desvantagens de teste baseado em software que são o tempo de teste e tamanho do programa de teste;
- Avaliar as técnicas de teste implementadas. Uma proposta é: (*i*) avaliar tempo de teste do SOC usando a mesma estratégia de teste implementada em software e a outra em hardware; (*ii*) explorar a flexibilidade do software em relação ao hardware para conseguir uma maior cobertura de falhas para o SOC.

Para dar suporte aos objetivos acima, este trabalho resultou no desenvolvimento das seguintes ferramentas de CAD:

- Ambiente de co-simulação funcional e geograficamente distribuída de descrições de hardware em VHDL e módulos de software em linguagem C (Capítulo 5);
- Ambiente de desenvolvimento e teste de SOC. O ambiente de desenvolvimento integra diversos núcleos descritos em VHDL e o ambiente de teste suporta geração de código de teste para núcleos não programáveis (Capítulo 7).

Não é objetivo deste trabalho:

- Explorar técnicas de teste mais abstratas que consideram teste no início do projeto como síntese para testabilidade e testabilidade no nível de sistema;
- Explorar técnicas de teste de software. Uma vez que o objetivo é teste de hardware, somente teste de hardware digital é considerado. Pressupõe-se que a parte de software do projeto foi devidamente verificada;
- Explorar problemas de teste de SOC tais como escalonamento de núcleos, consumo de potência durante teste, escolha da largura de barramento de teste, entre outros.

1.3 Organização do Documento

Este documento está organizado como segue:

- O Capítulo 2 apresenta conceitos e técnicas de teste de hardware digital tradicionais.
- O Capítulo 3 apresenta uma revisão de teste de dispositivos baseados em núcleos.

- O Capítulo 4 apresenta a motivação e uma revisão bibliográfica de técnicas de teste de hardware baseadas em software.
- É necessário o uso de um ambiente de co-simulação para validação conjunta do software de teste e dos núcleos. Desta forma, o Capítulo 5 visa apresentar a *primeira contribuição* deste trabalho que é o desenvolvimento e a avaliação de um ambiente de co-simulação funcional e geograficamente distribuído.
- O Capítulo 6 apresenta o ambiente de prototipação utilizado bem como a motivação do uso do mesmo. Este capítulo também apresenta a arquitetura alvo de SOC para este trabalho.
- O Capítulo 7 apresenta a *segunda contribuição* deste trabalho que é o arcabouço de projeto (em inglês, *framework*) de desenvolvimento e teste de SOC. Basicamente, este ambiente integra núcleos descritos em VHDL e gera código de teste em linguagem C dos núcleos não programáveis do SOC. Neste capítulo apresentamos a implementação e validação deste ambiente.
- O Capítulo 8 apresenta a *terceira contribuição* deste trabalho que é uma avaliação de técnicas de teste baseadas em software utilizando o ambiente de teste apresentado no capítulo anterior. O código gerado pelo ambiente de teste é avaliado em termos de tempo de teste, requisitos de memória e cobertura de falhas.
- O Capítulo 9 apresenta considerações finais sobre este trabalho e direções para trabalhos futuros. Este capítulo também retoma as vantagens, pontos de originalidade, desvantagens e limitações do trabalho desenvolvido.

2 PROJETO VISANDO O TESTE

A preocupação com teste, tipicamente até meados da década de 80, surgia somente na etapa final de concepção do CI. Entretanto, com o aumento da complexidade e o advento dos SOCs tornou-se obrigatório a presença de teste desde o início da fase de concepção.

Este capítulo tem como objetivo apresentar uma revisão geral de técnicas de teste. Assim, o mesmo é organizado da seguinte forma: na Seção 2.1 é apresentada uma introdução e conceitos relacionados ao teste. A Seção 2.2 apresenta uma breve revisão de projeto visando a testabilidade. A Seção 2.3 apresenta uma revisão de conceitos relacionados a auto-teste. Por fim, a Seção 2.4 apresenta as conclusões deste capítulo.

2.1 Conceitos Básicos

2.1.1 Confiabilidade e Teste

O requisito principal para obter um sistema confiável é determinar se um sistema eletrônico não possui erro. A confiabilidade em sistemas era principalmente desejável em aplicações militares e aeroespaciais. Porém, ultimamente tem crescido o interesse em confiabilidade na maioria das áreas que envolvem computação.

Para atingir o grau de confiabilidade desejado pode-se realizar *teste* no circuito desenvolvido procurando defeitos de fabricação. Dependendo do tipo de teste, falhas induzidas por efeitos do ambiente e flutuação na tensão de alimentação também podem ser detectadas. Devido à complexidade do processo de teste, pode-se optar por um fluxo de concepção de CIs que vise facilitar a etapa de teste após a fabricação. Esta abordagem é chamada de *projeto visando o teste.*

Outra abordagem para atingir o grau de confiabilidade é através de técnicas de *tolerância a falhas*. O objetivo da tolerância a falhas é permitir que o sistema continue funcionando na presença de falhas. Estas técnicas residem, principalmente, na utilização do conceito de redundância ao nível de módulos de hardware, software, informação ou tempo [61].

2.1.2 Verificação e Teste

Teste de um circuito antes de sua implementação é conhecido como *verificação*. Seu objetivo é verificar se o projeto está de acordo com a especificação. Atualmente, *simulação* é a ferramenta de verificação mais utilizada, uma vez que as técnicas de *verificação formal* ainda são imaturas e complexas.

Existem dois tipos de simulação. A *simulação funcional* é utilizada para verificar se o projeto executa a função desejada sem incluir modelos temporais. Na *simulação temporal* atrasos são adicionados às portas lógicas, ocorrendo uma simulação mais precisa que a simulação funcional. Ainda é possível realizar uma simulação com elementos parasitas, obtidos da síntese física, técnica esta chamada de *back annotation*.

Uma vez que o projeto está implementado em silício, ele pode ser verificado. Esta verificação, responsável por *verificar erros de fabricação* do CI, é chamada *teste.* Existem dois tipos principais de teste: *teste paramétrico* e *teste funcional*. O primeiro está relacionado ao teste de parâmetros do circuito como tensão e corrente. O segundo tipo, responsável por testar a funcionalidade do circuito, é o objeto de estudo deste trabalho.

2.1.3 Defeito, Falha e Erro

Um *defeito* é uma imperfeição física ocorrida no circuito. A Figura 5(a) ilustra um defeito de fabricação. Uma área de metal não desejada provoca um curto entre duas áreas de metal, modificando o comportamento do circuito [14].

Uma *falha* é uma abstração de um defeito [14][54][61]. O defeito ilustrado em Figura 5(a) pode ser modelado como um curto em nível de transistor, Figura 5(b), ou SA (*stuck-at*) 1 no nível de portas lógicas, Figura 5(c).

Um *erro* é uma manifestação de um defeito [14], isto é, a sua propagação para as saídas primárias do sistema através da geração de uma resposta incorreta. Por exemplo, na Figura 5(c) existe um erro quando as portas de entrada A e B forem iguais ao nível lógico 1, pois o valor da saída é 1 quando deveria ser 0. Por outro lado, quando A e B forem igual a 0, não há erro uma vez que o valor esperado é igual ao valor gerado.

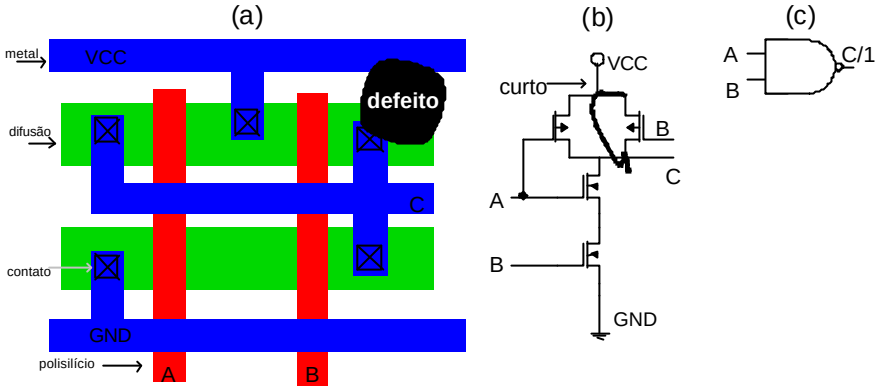


Figura 5 – Definição de defeito e falha. (a) Nível de leiaute. (b) Nível de transistor. (a) Nível de porta lógica [22].

Falhas podem ser caracterizadas de acordo com sua *duração* e *modo*, conforme ilustrado na Figura 6. De acordo com a duração as falhas podem ser *permanentes* ou *temporárias* [22][54][61]. Falhas permanentes existem indefinidamente e se referem, geralmente, a defeitos físicos no circuito. Falhas temporárias *transientes* são falhas que aparecem e desaparecem em um curto espaço de tempo, geralmente causadas por alguma perturbação externa como flutuações de energia, radiação do tipo SEU (*single-event upset*) e interferências eletromagnéticas. Falhas temporárias *intermitentes* existem somente em alguns intervalos. Este tipo de falha ocorre devido à degradação do componente ou também por efeito externo como temperatura. Por outro lado, de acordo com o modo, as falhas podem ser introduzidas [61]: (*i*) Na etapa de *especificação*, no nível mais alto de abstração, através de algoritmos e arquiteturas errôneas; (*ii*) Na etapa de *implementação* através de uma interpretação errônea da especificação; (*iii*) Na *fabricação* do circuito; (*iv*) Por *perturbações externas* como radiação, interferências eletromagnéticas, umidade, erro de operação e ambientes extremos.

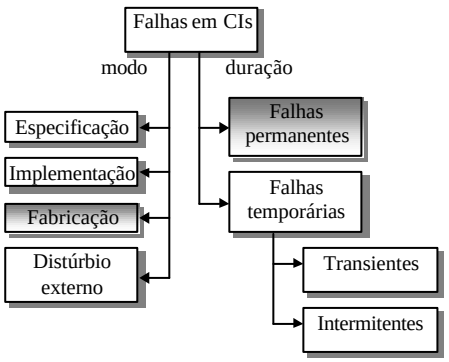


Figura 6 – Classificação das falhas em CIs.

Tendo em vista estas definições, *este trabalho se concentra na identificação de componentes com falhas permanentes induzidas por defeitos introduzidos na etapa de fabricação de ASICs e falhas temporárias que afetam a propagação de FPGAs.*

2.1.4 Modelos de Falhas

Como apresentado anteriormente, modelos de falhas são utilizados para representar defeitos físicos em um nível de abstração maior, geralmente no nível de porta lógica. A principal função desta abstração é reduzir a complexidade da análise do circuito [1][22][54].

Os modelos de falhas tradicionais correspondem a falhas que afetam as conexões no nível de portas lógicas. As falhas típicas são curtos e conexões abertas. Quando o curto ocorre entre uma linha *l* e a alimentação positiva, o modelo de falhas chama-se *stuck-at 1* (SA1). Este modelo costuma ser representado por *l*/1. Se a linha *l* está conectada ao terra, o modelo chama-se *stuck-at 0* (SA0 - *l*/0). O modelo de falhas *stuck-at* é o modelo de falhas mais básico e mais utilizado tanto na academia quanto na indústria de semicondutores. No caso de conexão aberta, a tensão na conexão permanece constante e, desta forma, também pode ser modelada como SA0 ou SA1. Quando um curto ocorre entre duas linhas, *l*1 e *l*2, ambas as linhas possuem a mesma tensão. Este tipo de modelo de falhas é chamado *bridging* [22].

Modelos de falhas variam de acordo com a estrutura a ser modelada. Por exemplo, memórias RAM possuem modelos de falhas específicos para esta estrutura de circuito. Estes modelos são brevemente apresentados na Seção 2.3.3.1.

2.1.5 Padrão de Teste

Para ilustrar como é feita a detecção de uma falha utilizamos uma porta lógica *e* com injeção de falhas *stuck-at*. Falhas são injetadas em todas as entradas e na saída da porta *e*, como mostra a Figura 7. As entradas são apresentadas na primeira coluna. Na segunda coluna constam os valores corretos e no restante da tabela os valores com falhas, onde as respostas erradas estão sombreadas. As falhas *stuck-at* na entrada A são indicadas por A/0 para *stuck-at* em nível 0 e A/1 para *stuck-at* 1. Uma notação similar é utilizada para B e Z. Em A/0 a saída com falhas difere da saída correta somente quando a entrada AB = 11 é aplicada na porta lógica. Esta combinação de valores de entrada é chamada de *padrão de teste*, pois detecta a falha A/0. Este mesmo padrão de teste também detecta B/0 e Z/0. Também há casos onde uma falha pode ser detectada por mais de um padrão, como é o caso de Z/1. Quaisquer dos padrões 10, 01 e 00 podem detectar esta falha. Por fim, para detectar todas as falhas possíveis em uma porta lógica e, utilizando o modelo *stuck-at*, três padrões de teste (01, 10 e 11) são necessários.

Entradas	Resposta	Resposta com Falha					
AB	Esperada	A/0	B/0	Z/0	A/1	B/1	Z/1
00	0	0	0	0	0	0	1
01	0	0	0	0	1	0	1
10	0	0	0	0	0	1	1
11	1	0	0	0	1	1	1

Figura 7 – Falhas *stuck-at* em uma porta lógica E [54].

2.1.6 Métricas de Teste

Três das principais métricas qualitativas de teste são: cobertura de falhas, observabilidade e controlabilidade. A *cobertura de falhas* (em inglês, *fault coverage*) é definida como a porcentagem de falhas detectáveis pelos padrões de teste pelo total de falhas do modelo. Uma de suas vantagens é que pode ser obtida por simulação, o que se torna economicamente mais atraente que a métrica de rendimento, apresentada na Seção 2.1.8, que é obtida por amostragem. É importante salientar que uma cobertura de falhas de 100% não garante que o circuito não possui falhas. O processo de teste verifica somente as falhas representadas pelo modelo de falhas utilizado, como o modelo *stuck-at* (SA). A *observabilidade* é representada pelo esforço necessário para observar o valor lógico de um nó interno através de um pino de saída. A *controlabilidade* é representada pelo esforço necessário para atribuir um determinado valor lógico a um nodo. O aumento da *integração dos CIs*, ou seja, o aumento do número de nodos internos por pino, afeta direta e negativamente estas duas últimas métricas de teste.

2.1.7 Taxionomia de Teste

Esta seção classifica técnicas de teste de acordo com três parâmetros: quanto ao método de geração dos padrões de teste, quanto ao momento em que o teste é realizado e quanto à origem dos padrões de teste.

2.1.7.1 Quanto ao Método de Geração dos Padrões de Teste

De acordo com o método de geração dos padrões de teste, existem quatro classificações possíveis para o teste: exaustivo, pseudo-exaustivo, pseudo-aleatório e determinístico.

O teste *exaustivo* utiliza todas as combinações de entradas como padrão de teste. Tem a vantagem de ser possível atingir 100% de cobertura de falhas, em geral sem grande dificuldade, para circuitos combinacionais. Para um circuito combinacional com 8 entradas, seriam necessários 256 padrões de teste para testar o circuito. Porém, para circuitos seqüenciais complexos não é viável a aplicação deste tipo de teste dado o grande número de estados do circuito.

O teste *pseudo-exaustivo* divide um circuito em sub-circuitos testado-os exaustivamente sem testar exaustivamente o circuito completo. A Figura 8 ilustra este conceito. O circuito em questão possui 8 entradas foi dividido em três sub-circuitos: alfa, beta e gama. Estas partições possuem, respectivamente, 2, 3 e 3 entradas, totalizando 4, 8 e 8 padrões de teste para cada sub-circuito. Desta forma são necessários 20 padrões de teste ao invés de 256 do teste exaustivo.

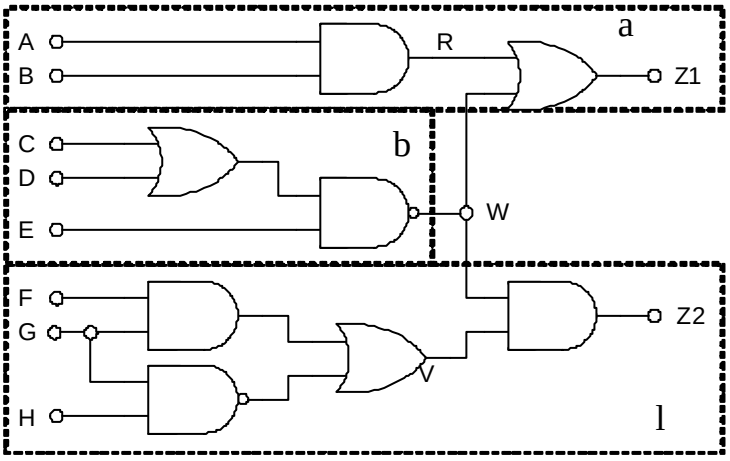


Figura 8 – Teste pseudo-exustivo [54].

O teste *determinístico* ou *orientado a falhas* gera padrões específicos a uma falha. Geralmente

estes padrões são gerados por ferramentas de geração automática de padrões de teste (*Automatic Test Pattern Generation* - ATPG). Uma vez que o processo de geração de padrões é NP-completo, este consome um longo tempo. Além disto, é necessário uma grande quantidade de memória para armazenar os padrões, visto que estes padrões costumam ser aplicados por um ATE. O tempo para gerar os padrões de teste mais a quantidade de memória necessária ao ATE são fatores que fazem aumentar o custo de teste de circuitos.

2.1.7.2 Quanto ao Momento em que o Teste é Realizado

Existem duas classificações possíveis para o teste de acordo com o momento em que o teste é realizado: *off-line* e *on-line*. O teste *off-line* é executado quando o circuito não está em uso. É a principal aplicação do BIST. Por outro lado, o teste *on-line* pode executar testes durante o funcionamento normal do circuito. Este tipo de teste geralmente faz uso de alguma técnica de codificação ou duplicação de dados (e.g. paridade simples, código de Hamming, votador, entre outros). A maior vantagem do teste *on-line* é a capacidade de detectar falhas transientes ou intermitentes.

2.1.7.3 Quanto à Origem dos Padrões de Teste

Dois principais fatos fazem com que o tempo de teste aumente: o primeiro refere-se ao aumento da densidade dos CIs ter tornado o volume dos padrões de teste extremamente grande. Outro fato é a disparidade da velocidade de operação interna e externa devido à limitação na quantidade de pinos e entrada e saída. Estes fatos fazem com que a largura de banda externa seja muito menor que a largura de banda interna. Portanto, para minimizar a interação com o mundo externo tem-se adicionado módulos de teste embutidos, procurando um compromisso entre tempo de teste e acréscimo de área. Assim, de acordo com o origem dos padrões de teste, existem duas classificações possíveis para teste: embutido (em inglês, *on-chip*) e externo (em inglês, *off-chip*).

O teste *embutido* faz uso de módulos de hardware para gerar padrões de teste, compactar respostas ou até avaliá-las. A sua grande vantagem é a velocidade de operação, pois realiza o teste na freqüência de operação nominal (em inglês, *at-speed*). A desvantagem é que pode induzir acréscimo de área de silício e reduzir a freqüência de operação devido à adição destes novos módulos de teste. Porém, existem diversos trabalhos que visam reduzir ou até eliminar este acréscimo de área [8]. Outra vantagem atribuída a este tipo de teste é a redução dos requisitos dos ATEs uma vez que o tráfego de dados entre o ATE e o CI sendo testado é menor. Isto reduz a quantidade de memória necessária para armazenar os padrões de teste e as repostas, pois o mesmo pode ser feito pelo módulo de hardware embutido. Também reduz a necessidade de uma alta freqüência de operação do ATE, uma vez que o processo de teste é menos dependente deste equipamento e o tráfego de dados é menor.

O teste *externo* é altamente dependente do ATE. Sabe-se que estes equipamentos tendem a aumentar de preço com o aumento de números de pinos e freqüência de operação dos CIs a serem testados e quantidade de memória necessária. Como dito anteriormente, a SIA prevê que o custo de ATEs será de aproximadamente 20 milhões de dólares em 2014. Este custo, mais o tempo adicional que esta técnica de teste demora em relação ao teste embutido, faz com que o custo do projeto cresça, além de aumentar o *time-to-market*.

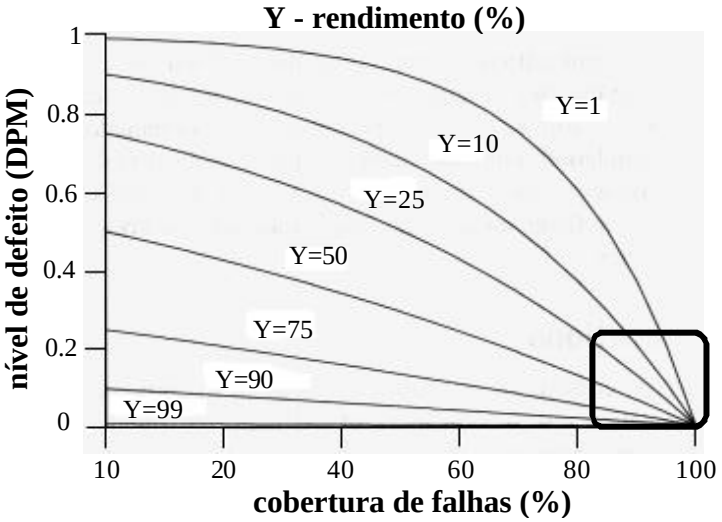
2.1.8 Rendimento e Nível de Defeito

O *rendimento* (em inglês, *yield*) de produção de CIs é definido pela fração de dispositivos que não acusam defeitos de fabricação em relação ao número total de dispositivos produzidos. É determinado por Y = G/(G+B), onde G e B representam respectivamente os número de circuitos bons e com defeitos. O rendimento pode ser interpretado como a probabilidade de se produzir um circuito sem defeito. Embora outros fatores também influenciem o rendimento como área do *wafer*,

maturidade do processo e número de etapas do processo, rendimento também é uma estimativa de métrica de qualidade de dispositivos, expressa em percentual.

Nível de defeito (em inglês, *defect level*) é a fração de CIs defeituosos que passam nos testes. Geralmente medido em defeitos por milhão (em inglês, *defects per million* - DPM). Pode ser interpretado como a probabilidade de vender um circuito com algum defeito. É uma das métricas de avaliação de qualidade de um processo de fabricação.

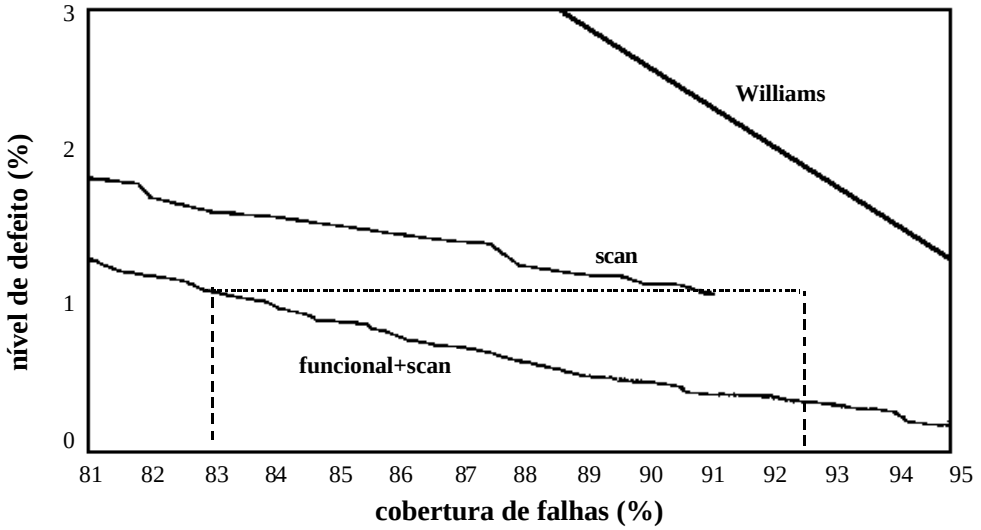
Existem várias fórmulas que visam relacionar o rendimento com o nível de defeito. Uma delas é $DL = 1-Y^{(1-T)}$, onde Y e T são respectivamente o rendimento e a cobertura de falhas [78]. A Figura 9 apresenta esta relação. Esta figura informa que, por exemplo, para um rendimento de 50%, é necessária uma cobertura de falhas de 30% para se alcançar um nível de defeito de 0.4 DPM. Mas se o rendimento fosse 25%, então seria necessária uma cobertura de falhas de 65% para o mesmo nível de defeito. É importante destacar que com uma cobertura de falhas de 100 % ainda podem ocorrer erros, portanto o nível de defeito diferente de zero, uma vez que cobertura de falhas detecta erros relacionados a um modelo de falhas. Outros tipos de falhas não cobertas pelo modelo utilizado podem ocorrer e não serem detectados.



A área destacada é representada pela Figura 10.

Figura 9 – Nível de defeito como uma função do rendimento e da cobertura de falhas [1].

Maxwell *et. al.* [49] realizaram um estudo para avaliar a qualidade de teste baseado na cobertura de falhas do modelo *stuck-at*. Para isto três tipos de teste foram realizados: funcional, *scan* (visto na seção 2.2.2) e misto funcional+*scan*. Conforme a Figura 10, o resultado é que o teste funcional+*scan* precisou de uma menor cobertura de falhas (83%) que o teste baseado em *scan* (92.5%) para obter o mesmo nível de defeito. O motivo disto é que teste funcional detecta falhas de temporização, o que o teste baseado em *scan* não consegue. A conclusão é que não basta ter como objetivo alcançar uma dada cobertura de falhas, mas também se deve levar em conta como a cobertura de falhas vai ser alcançada.



O teste funcional+*scan* com 83% de cobertura possui o mesmo nível de defeito que o teste baseado em *scan* com 93% de cobertura. A curva denominada Williams representa a fórmula $DL = 1-Y^{(1-T)}$ [49].

Figura 10 – Teste funcional+*scan* *vs* *scan* *vs* curva Williams.

2.2 Projeto Visando a Testabilidade

Dados os problemas relacionados ao teste citados anteriormente, é importante manter os custos relacionados ao teste em valores aceitáveis. Para que isso seja possível, uma maior quantidade de área de silício e degradação de desempenho pode ser tolerada, desde que sejam mantidos os compromissos em relação ao custo, área e desempenho. Esta abordagem é conhecida como projeto visando testabilidade (DFT) [1][14][47][54]. O objetivo de DFT é aumentar a controlabilidade e observabilidade, reduzindo os custos relacionados ao teste e aumentando a cobertura de falhas [54].

Embora vários problemas de testabilidade sejam amenizados com o uso de DFT, utilizar somente esta técnica já não é mais viável para circuitos grandes. Técnicas tradicionais de DFT estão ligadas ao uso de aparelhos testadores externos que aplicam estímulos e comparam respostas. Devido a este fato, o tempo para aplicar os vetores de teste se torna inaceitável, sem citar o volume de vetores de teste que testadores externos têm que armazenar [64]. Outra desvantagem desta abordagem é o custo destes testadores e o fato de não se poder testar o dispositivo já inserido no sistema e executar teste em tempo de execução. Por causa destas desvantagens faremos apenas um pequeno apanhado de tais técnicas.

2.2.1 Técnicas *Ad Hoc*

Técnicas *ad hoc* ou caso a caso são medidas tomadas para melhorar a testabilidade durante a fase de projeto de forma não estruturada e não sistematizada [1][14][54]. Como exemplo podemos citar a inicialização de circuitos seqüenciais, introdução de sinais de *reset*, particionamento de circuitos e a introdução de pontos de observação [54]. Algumas técnicas são utilizadas largamente em placas de circuito impresso. Porém, no universo de circuitos integrados as técnicas estruturadas, apresentadas a seguir, são mais aconselhadas.

2.2.2 Scan Path

As técnicas estruturais de DFT representam um enfoque sistemático para a questão de testabilidade, superando as limitações técnicas da abordagem *ad hoc* e viabilizando a automatização

desta fase através de ferramentas de CAD. Uma técnica de DFT largamente difundida é o *boundary scan* [34][59]. A técnica mais tradicional é o *scan path*. Nesta técnica, os registradores do sistema são modificados de forma a terem dois modos de operação: modo normal e de teste. No modo normal os registradores funcionam executando sua função de armazenar dados. Já no modo de teste, estes se tornam registradores de deslocamento, aumentando o número de nós internos que são controláveis e observáveis [54]. A Figura 11 ilustra esta estrutura.

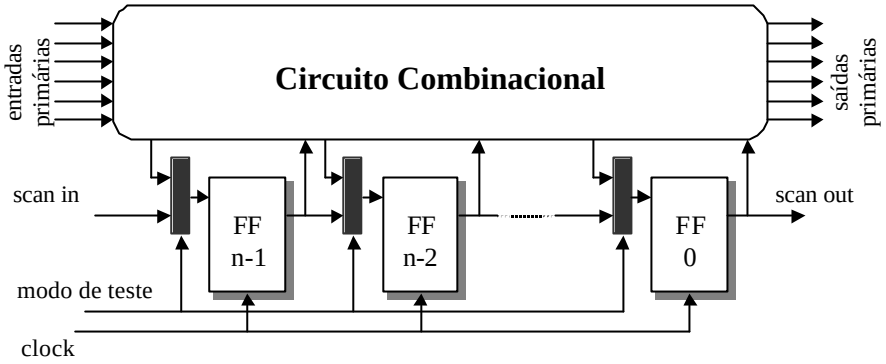


Figura 11 – Arquitetura básica de *scan path*.

O funcionamento desta técnica está ilustrado na Figura 12. É apresentada uma série de ciclos de relógio necessários para a carga dos FFs e a leitura dos seus conteúdos. Primeiramente, o sinal T é ativado configurando os FFs em modo de teste (registrador de deslocamento) e o vetor é inserido serialmente nas entradas dos FFs. Ao final de *n* períodos de relógio, onde *n* é igual ao tamanho do *Scan Path*, o vetor está carregado nos FF e as suas saídas estão acionando as entradas do circuito combinacional em teste. Por um período de relógio, o sinal de controle T sinaliza o modo normal de operação e na próxima borda de relógio as saídas do circuito combinacional, que foram produzidas a partir das entradas definidas pelo vetor de teste carregado previamente nos FFs, são carregados nos FFs. Estes valores são deslocados para a saída para serem comparados com o valor esperado. Desta forma, se consegue controlabilidade de todas as entradas dos circuitos combinacionais e observabilidade de todas suas saídas.

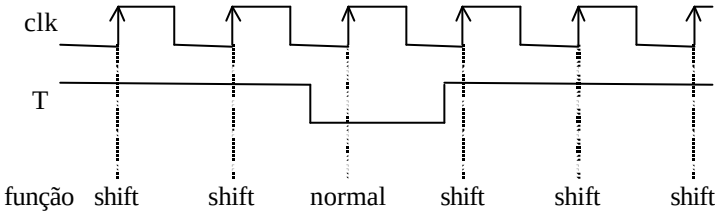


Figura 12 – Funcionamento do *scan path*.

2.3 Auto-Teste

Uma alternativa às técnicas de DFT, onde vetores de teste são gerados por um testador externo, é o auto-teste (em inglês, *Built-in Self-Test* - BIST). Em BIST são incorporados módulos de hardware responsáveis pela geração de vetores de teste, comparação da resposta (assinatura) e controle do fluxo de teste [1][2][3][8][47][71].

BIST reduz o custo e o tempo da etapa de teste, diminuindo o ciclo de concepção do dispositivo, minimizando o volume de dados com algoritmos de compactação embutidos no CI e diminui a necessidade e requisitos de testadores externos. BIST também tem a vantagem de possibilitar teste em diferentes níveis da hierarquia do sistema, simplificando o projeto de teste do sistema completo. Além disto, permite o teste em tempo de execução (em inglês, *at-speed*), podendo cobrir falhas de características temporais.

Porém, o uso de BIST está associado a certos custos adicionais. É necessária mais área de

silício para os módulos de geração, avaliação e controle de teste, e acarreta uma degradação de desempenho com a presença de multiplexadores adicionais. O último custo citado, principalmente, pode inviabilizar o uso de BIST em arquiteturas que precisem de alto desempenho [64]. A Figura 13 apresenta uma arquitetura de BIST genérica.

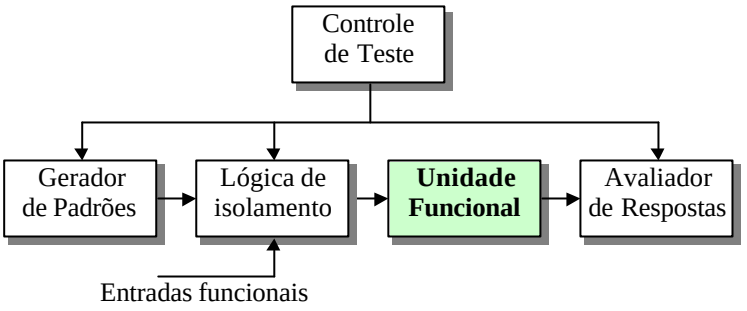


Figura 13 – Arquitetura de BIST genérica.

Nas Seções seguintes apresentaremos exemplos de módulos de geração de vetores de teste e de avaliação e compactação de respostas.

2.3.1 Geração de Vetores de Teste

A técnica mais utilizada para a geração de estímulos (vetores de teste) para o auto-teste é a geração de estímulos pseudo-aleatórios. A estrutura utilizada chama-se *Linear Feedback Shift Register* (LFSR) [8][71], cujo formato genérico é apresentado na Figura 14(b). Um LFSR está associado a uma função matemática, apresentada na Figura 14(a). Outras técnicas utilizadas para geração de padrões são *cellular automata* [8], *weighted pattern generator* [1] e contadores [64].

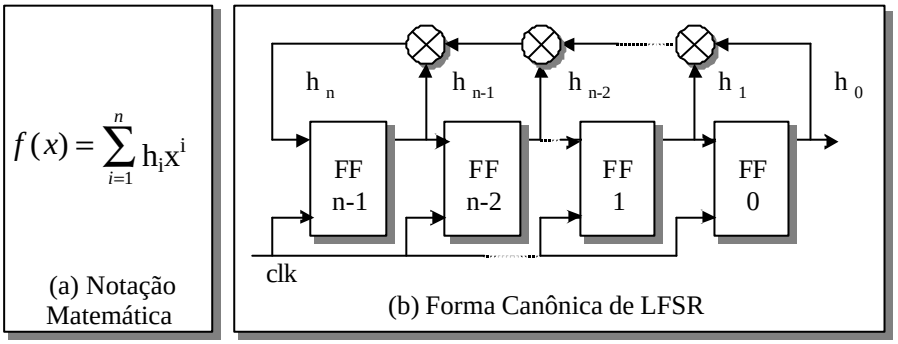
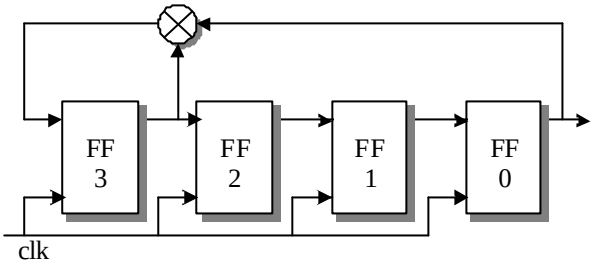


Figura 14 – LFSR de n-1 estágios.

Os coeficientes h_i, apresentados na Figura 14(a), são iguais a 1 quando existe ligação de realimentação e iguais a 0 quando não existe. Este polinômio associado ao LFSR é chamado de função geratriz e quando primitivo faz com que todas as combinações de valores (2^n –1 valores exceto o valor zero) sejam geradas pelo LFSR de forma aleatória. Em [8], existe uma tabela de polinômios primitivos com o menor número de termos de até o grau 300.

A Figura 15 apresenta um LFSR baseado em um polinômio primitivo ($f(x) = x^4+x^3+1$) que produz a seqüência de valores mostrada na Tabela 1. Observa-se que os valores são gerados de forma pseudo-aleatória e que o LFSR deve ser inicializado com um valor diferente de zero. Pode-se usar como estímulos a saída do FF 0 do LFSR (modo serial) ou todas as saídas dos FFs (modo paralelo).



(h4, h3, h2, h1, h0) = (1,1,0,0,1)

Figura 15 – Exemplo de LFSR de quatro estágios.

Tabela 1 – Estados pseudo aleatórios gerados pelo LFSR de quatro estágios.

clk	Q3	Q2	Q1	Q0
0	1	0	0	0
1	1	1	0	0
2	1	1	1	0
3	1	1	1	1
4	0	1	1	1
5	1	0	1	1
6	0	1	0	1
7	1	0	1	0

clk	Q3	Q2	Q1	Q0
8	1	1	0	1
9	0	1	1	0
10	0	0	1	1
11	1	0	0	1
12	0	1	0	0
13	0	0	1	0
14	0	0	0	1
15=0	1	0	0	0

A Figura 16 apresenta uma simulação utilizando o mesmo LFSR da figura anterior. A porta *lfsr_in* representa uma entrada onde o usuário atribuiu um valor inicial, semente, para o gerador de números pseudo-aleatórios. A partir deste ponto o LFSR segue gerando na porta *lfsr_out* a mesma seqüência de dados apresentada na tabela anterior. Por exemplo, na Tabela 1, após o dado A, em representação hexadecimal, é gerado o valor D e assim sucessivamente. A simulação acaba quando o LFSR gerou todos os valores possíveis exceto zero.

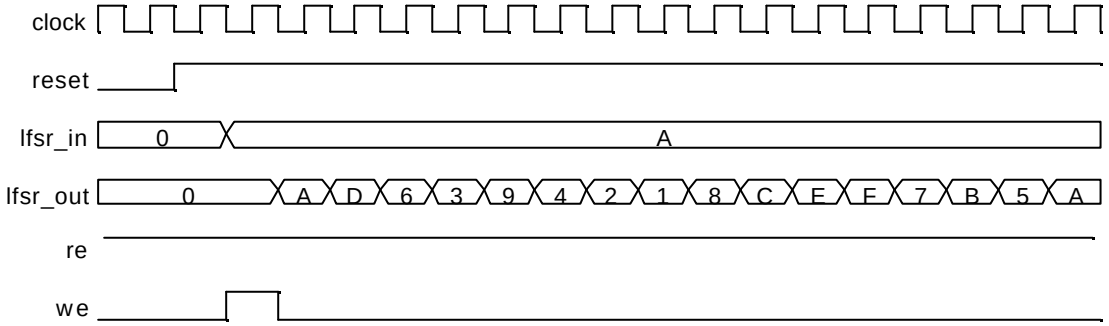


Figura 16 – Simulação da descrição VHDL de LFSR compactador de polinômio x^4+x^3+1.

2.3.2 Compactação de Teste

Nas técnicas de teste tradicionais, os valores de teste são comparados com um valor esperado pré-computado. Em BIST isto é inviável, devido à grande quantidade de memória necessária para armazenar este volume de dados. Esta Seção tem como objetivo apresentar alguns métodos de compactação de respostas apresentados na literatura. O objetivo de compactação de respostas de teste é reduzir este volume de dados a um tamanho gerenciável (geração de assinatura) que ainda assim seja capaz de detectar falhas ao custo da adição de um pequeno circuito.

Um fenômeno chamado de mascaramento de erro (em inglês, *aliasing*) pode ocorrer durante a compactação das respostas, fazendo com que circuitos com falhas gerem a mesma assinatura que um circuito sem falha. A probabilidade de ocorrência de mascaramento de erro é definida como a probabilidade média de nenhuma falha ser detectada devido à compactação depois de uma execução de teste suficientemente longa [54]. Entretanto, determinar a probabilidade de mascaramento de

erro não é uma tarefa simples. Uma primeira solução seria fazer simulações do algoritmo de compactação injetando falhas e verificar o número de falhas não detectadas, porém, esta abordagem pode ser computacionalmente intensiva para seqüências de testes longas. Outras abordagens são apresentadas em [64][71].

Existem diferentes formas de compactação de respostas de teste. Algumas são simplistas, como contadores de 1s e contadores de transições. Porém, as mais utilizadas são baseadas em LFSR (Figura 17(b)) e MISR (*"Multiple Input Signature Register"* - Figura 17(a)).

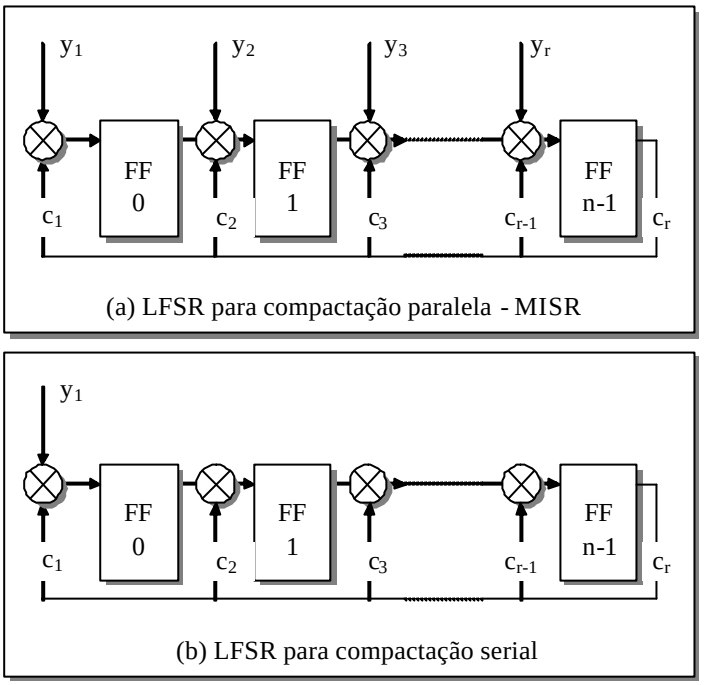


Figura 17 – Circuitos de compactação de respostas de teste.

O LFSR da Figura 17(b) possui uma modificação em relação à implementação original por receber uma entrada externa, agindo como um divisor de polinômios [1][2]. Esta modificação é necessária porque a versão de LFSR mostrada anteriormente não garante o valor correto do quociente da divisão. À medida que a divisão prossegue, o quociente aparece na saída do LFSR e o resto permanece no LFSR. Ao fim da etapa de teste, o conteúdo do LFSR é a assinatura do teste.

Apresentamos um exemplo de compactação com LFSR [8]. Considere um LFSR de 5 bits com polinômio gerador $h(x) = x^5 + x^3 + x + 1$, representado na Figura 18. A resposta sem falhas gerada pelo circuito sendo testado é "10001010" e sua assinatura é "10110". A seqüência de passos para gerar a assinatura é apresentada na Tabela 2.

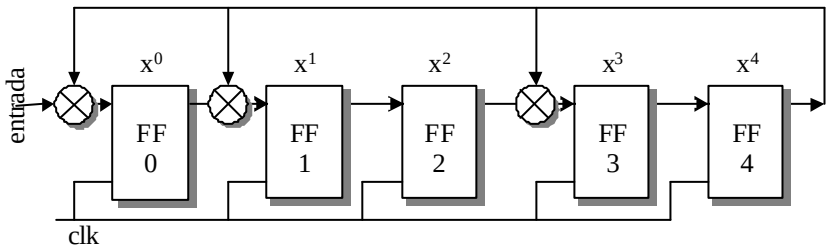


Figura 18 – LFSR compactador serial de polinômio $x^5 + x^3 + x + 1$.

Tabela 2 – Seqüência de passos de um LFSR na geração de assinatura.

Entrada	Conteúdo do LFSR [1]
-	00000
1	10000
0	01000
0	00100
0	00010
1	10001
0	10010
1	11001
0	**10110**

Supondo a resposta "11001010" com falhas, sua assinatura seria "11011". Uma comparação entre a assinatura gerada e a assinatura correta gerada anteriormente acusaria a falha no teste.

A Figura 19 apresenta uma simulação do LFSR de polinômio $x^5 + x^3 + x + 1$ com a mesma resposta que gerou a Tabela 2. A porta *lfsr_in* é carregada com a resposta do circuito que é inserida serialmente no LFSR. Ao fim da simulação o resultado do LFSR é 10110, que confere com a Tabela 2.

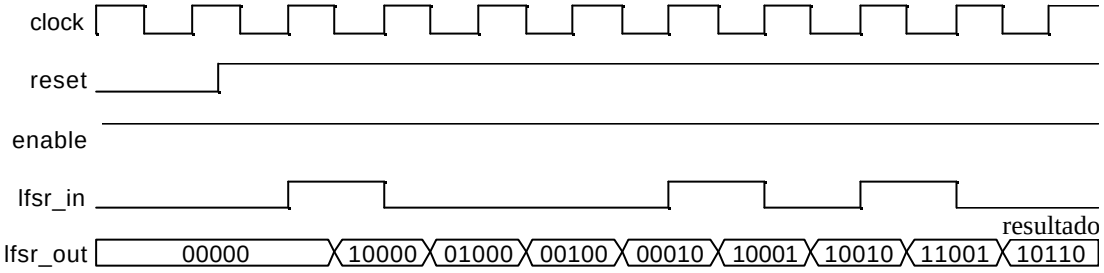


Figura 19 – Simulação do LFSR compactador de polinômio $x^5 + x^3 + x + 1$.

Por outro lado, MISR é uma variação do LFSR que aceita múltiplas entradas simultâneas [1][2][3][8][14][54][71]. Nota-se pela Figura 17(a) que, se todas as entradas do MISR menos a entrada y1 estiverem em zero, o MISR se transforma em um LFSR. MISR é recomendado para circuitos com múltiplas saídas, podendo executar teste de forma paralela. Assim como o LFSR, ao fim da etapa de teste a assinatura estará armazenada nos FFs do MISR.

Idealmente, uma dada técnica de BIST deveria poder ser aplicada a qualquer tipo de circuito. Porém, devido à diversidade de arquiteturas e requisitos de projeto, existem variações de BIST que são mais bem aplicadas a uma ou a outra situação. Dentro deste universo de variações nos limitaremos a apresentar as técnicas de BIST *off-line* aplicadas a memórias [10][27][56][75], a processadores [65] e teste baseado em software (Capítulo 4). Outras classes como teste de corrente [54], teste térmico e síntese de teste [54] (entre outras) não são exploradas neste texto.

2.3.3 Teste de Memória

Módulos de memória são as estruturas de hardware onde mais se utilizam técnicas de teste. O principal motivo está ligado à estrutura regular que facilita o teste [54]. Porém, um dos principais desafios no teste de memórias é o tempo de teste. Este tempo pode se tornar excessivo para memórias grandes sendo testadas por um testador externo. Uma solução atrativa para este problema é a aplicação de BIST.

[1] bit menos significativo à esquerda

2.3.3.1 Modelo de Falhas de Memórias

Modelo de falhas de memória têm por objetivo representar falhas que podem acontecer em uma memória. Abaixo segue uma lista dos principais modelos de falhas de memória [54][56][75]:

- *Stuck-at-Fault* – Refere-se a uma célula com valor fixo em nível lógico '0' ou '1';
- Falhas de Acoplamento – Uma dada posição de memória é afetada por outra posição de memória. Por exemplo, uma escrita no bit 5 da palavra 3 muda o bit 6 da palavra 2;
- Falhas de Endereçamento – São falhas localizadas nos decodificadores de endereço;
- Retenção de Dados – São falhas onde uma célula de memória não pode guardar dados por um longo período de tempo;
- Falhas de Transição – Este tipo de falha ocorre quando células não podem mudar de '0' para '1' ou de '1' para '0'.

2.3.3.2 Algoritmos de Teste de Memórias

A Tabela 3 apresenta uma comparação entre a complexidade de diversos algoritmos de teste de memória. Esta tabela também informa os modelos de falhas que cada algoritmo suporta. Algoritmos com complexidade maior tendem a se tornar impraticáveis com o aumento da capacidade de armazenamento das memórias. Nas leituras realizadas [10][65][75], observou-se a maior utilização dos algoritmos *march*, em especial *march* C e C+, por terem uma complexidade linear e possuírem boa cobertura de falhas (Figura 20). Maiores informações sobre os algoritmos de memória podem ser encontradas em [27][54][75].

Tabela 3 – Algoritmos de teste de memória e suas complexidades [54].

		Falhas Detectáveis[1]							
Algoritmo	**Complexidade**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
MSCAN	$4N$		x						
GALPAT	$2(2N^2 + N)$	x	x	x	x		x		
Marching 0/1	$2(N^2 + N)$		x	x	x				
GALDIA	$(2N^{1/2} + 4)N + 5N^{1/2}$		x	x	x			x	
GALCOL	$3N^{3/2} + 6$		x	x					x
March A	$30N$	x	x	x	x				
March B	$16N$	x	x	x	x				
March C	$11N$	x	x	x	x				
March C-	$10N$	x	x	x	x				
March C+	$14N$	x	x	x	x	x			
Checker	$N + 32N \log_2 N$		x	x	x				

Usando as técnicas de teste apresentadas anteriormente, sempre que a memória for testada, o seu conteúdo deve ser armazenado em uma memória auxiliar durante o teste para não perder o conteúdo. Isto é claramente uma grande desvantagem quando se precisa executar teste de memória periodicamente. Para contornar este problema foi criada uma nova abordagem chamada de BIST transparente [56]. BIST transparente pode transformar qualquer algoritmo de teste de memória de forma a não destruir o conteúdo da memória. Os padrões de teste utilizados são o próprio conteúdo da memória que, durante o teste, é complementado um número par de vezes. Antes de se iniciar o teste é feita uma varredura de toda a memória, de forma a criar uma assinatura de teste para o conteúdo da memória. Depois de gerada a assinatura, é executado o procedimento de teste complementando-se um número par de vezes cada posição de memória. Em [56], o autor afirma que a versão modificada do algoritmo *march* C com suporte a BIST transparente possui o acréscimo de área de 2% em relação à implementação do algoritmo *march* C original.

[1] 1 – falhas de endereço; 2 – *Stuck-at-Faults*; 3 – Falhas de transição; 4 – Falhas de acoplamento; 5 – Falhas de temporização; 6 – Falhas de transição de endereço entre cada célula; 7 - Falhas de transição de endereço entre cada célula e uma diagonal; 8 – Falhas de transição de endereço entre cada célula e uma coluna.

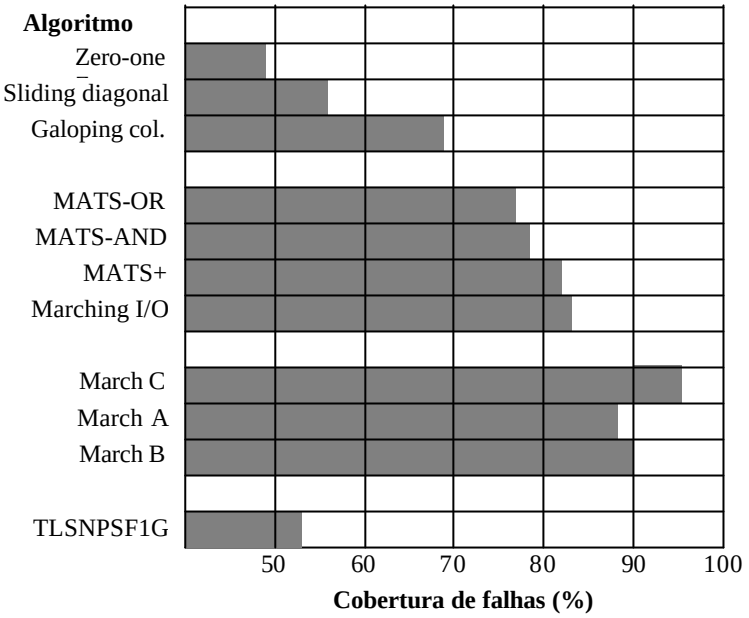


Figura 20 – Cobertura de falhas de diversos algoritmos de teste de memória [75].

2.3.4 Teste de Processador

O teste de processadores modernos tem se tornado um grande desafio devido às seguintes razões [54]: (*i*) por possuir uma arquitetura diversificada e complexa (e.g. MIPS e Pentium IV); (*ii*) por possuir memória embarcada (e.g. cache de vários níveis, banco de registradores, entre outros); (*iii*) por utilizar tecnologia submicrônica; (*iv*) por possuir alta velocidade de operação, que aumenta as restrições de tempo, impossibilitando adição de circuitos de BIST; (v) por possuir partes com difícil observabilidade e controlabilidade como circuitos de exceção, alguns registradores de controle, entre outros.

Existem diversas abordagens para o problema de teste de processadores. Duas delas são apresentadas. A primeira, apresentada a seguir, baseia-se na adição de módulos de hardware para geração de estímulos e compactação de respostas e no uso de ATE para inserir as instruções, ou seja, no uso combinado de BIST e ATE. A segunda abordagem, apresentada posteriormente na Seção 4.2.1, baseia-se na execução de um código cuidadosamente desenvolvido para exercitar a maior parte das funcionalidades do processador, ou seja, um teste funcional.

A estrutura geral de uma implementação de BIST para processadores é apresentada na Figura 21. A Figura 21(a) apresenta a estrutura geral de um processador. A unidade de busca de instruções (BI) obtém o *opcode* da próxima instrução, baseado no endereço do contador de programa (PC). Este *opcode* é decodificado pelo bloco de controle, que gera sinais para o bloco de dados. Baseado nestes sinais de controle, os operandos obtidos da memória do sistema são computados na unidade de execução. A estrutura da Figura 21(a) é então modificada para suportar auto-teste [65]. São adicionados três registradores conforme Figura 21(b). TCR (*Test Control Register*) que provê o *opcode* de uma instrução do processador. O LFSR para gerar os operandos das instruções providas por TCR. O MISR que compacta os resultados obtidos.

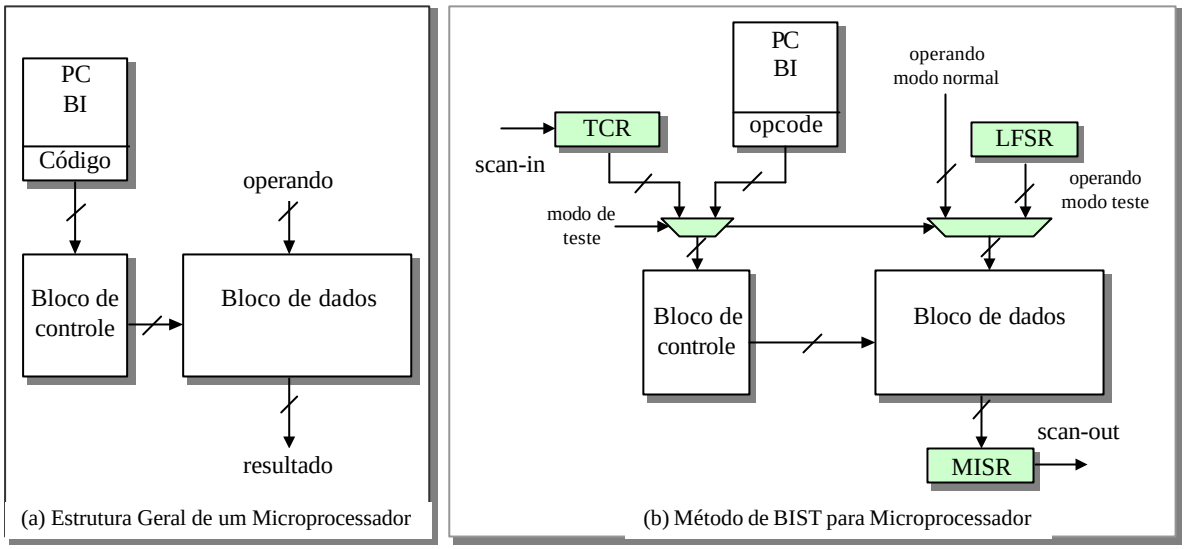


Figura 21 – Implementação de BIST para microprocessadores [65].

A seqüência de passos para executar o teste é:

1. Ativar o modo de teste;
2. Inicializar TCR, LFSR e MISR via sinais de controle de teste (omitidos da figura por questão de clareza);
3. Carregar o TCR serialmente pela entrada *scan-in* (por um testador externo) com *opcode* de uma instrução;
4. Executar um número fixo de ciclos de relógio, executando a instrução indicada por TCR várias vezes com diferentes operandos gerados pseudo-aleatoriamente a cada ciclo pelo LFSR;
5. Serialmente, enviar pelo pino *scan-out* o conteúdo do MISR para executar a comparação;
6. Comparar o conteúdo do MISR com um valor pré-computado em uma simulação para verificar se a dada instrução foi executada com sucesso. Um equipamento externo pode realizar esta comparação;
7. Repetir os passos 2 a 6 com diferentes instruções até que todas as instruções tenham sido testadas.

Não há um maior aprofundamento em teste de processadores por não estar diretamente ligado ao escopo deste trabalho.

2.4 Conclusão

As técnicas de teste apresentadas neste capítulo são utilizadas na indústria com sucesso há cerca de 20 anos. Porém, como já salientado anteriormente, o crescente grau de integração e o conseqüente aumento de desempenho dos SOCs têm inviabilizado o uso de tais técnicas. A criação de novos métodos de teste e modelos de falhas têm se tornado uma necessidade. Os dois próximos capítulos apresentam propostas de soluções para o teste de SOCs.

3 TESTE DE DISPOSITIVOS BASEADOS EM NÚCLEOS DE HARDWARE

Os recentes avanços na tecnologia de CIs permitem o desenvolvimento de um sistema completo em um único componente, chamado de SOC, fornecendo como vantagem maior desempenho, menor consumo de potência, menor volume e peso comparado com o projeto baseado em múltiplos CIs. O projeto de SOCs pode empregar módulos de hardware complexos reutilizáveis, chamados de núcleos de hardware. O principal objetivo do uso de núcleos é a redução do *time-to-market*, uma vez que estes módulos são pré-desenvolvidos e pré-verificados. O uso de núcleos envolve dois grupos, os provedores e os usuários de núcleos. No projeto baseado em múltiplos componentes em uma placa de circuito impresso, CIs já projetados, fabricados e testados são enviados para o usuário pelo provedor. O usuário precisa, neste caso, projetar, fabricar e testar a placa, utilizando os componentes como módulos livres de falhas. O teste se resume a teste de interconexão dos CIs. Em projeto de SOCs, os componentes são núcleos que não estão testados pois não estão fabricados. O teste e a fabricação de todo o sistema é responsabilidade do usuário, incluindo o teste dos próprios núcleos.

O teste de sistemas baseados em núcleos apresenta vários desafios adicionais em relação ao teste de sistemas em placa de circuito impresso, entre eles [48]:

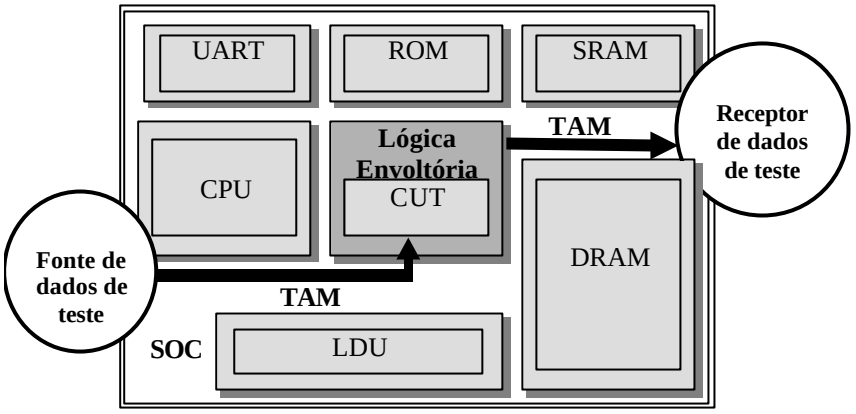
- *Teste Interno de Núcleos*: Com a redução do tamanho do transistor e o aumento da freqüência de operação, o desenvolvimento de teste de alta qualidade, porém, de baixo custo, está se tornando um desafio maior. Os modelos de falhas são inadequados, uma vez que defeitos de atraso e *crosstalk* induzidos pelo roteamento estão se tornado mais freqüentes.
- *Transferência de Conhecimento de Teste do Núcleo*: Uma vez que o teste envolve dois atores, o provedor e o usuário do núcleo, há uma necessidade de padronização na transferência de informação a respeito de teste. Por exemplo, cobertura de falhas, padrões de teste, protocolos de teste, arquitetura de DFT, modos de teste, entre outros.
- *Acesso de Teste aos Núcleos Embutidos*: Núcleos são módulos internos aos SOCs, podendo conter diversos níveis de hierarquia. Assim, é necessário definir um método para levar os padrões de teste às entradas e capturar as suas respostas. A origem dos padrões e o destino das respostas podem ser um ATE ou módulos de BIST embutidos.
- *Integração e Otimização do Teste do Sistema*: SOCs são formados por núcleos, lógica definida pelo usuário (LDU) e lógica de interconexão. Desta forma, o integrador do sistema deve escolher um compromisso entre diversos parâmetros como tempo de teste, acréscimo de área de silício, degradação de desempenho, qualidade de teste, consumo de potência e custo total de teste.

Padronização da interface dos núcleos, geração de estímulos e captura de respostas são somente alguns dos desafios que envolvem o teste de SOCs. Existe a necessidade de um conjunto de algoritmos que contemple escalonamento de teste, criação de mecanismo de acesso de teste (em inglês, *Test Access Mechanism* – TAM), seleção de conjunto de teste, permitir teste paralelo e encontrar o caminho de teste. Estes algoritmos devem utilizar como restrições o tempo de teste, consumo de potência, área de hardware, número de pinos e recursos de teste [16][17][21][37][45]. Porém, este trabalho não contempla esta parte de teste de SOC.

O objetivo e organização deste capítulo são: apresentar uma revisão do estado da arte em teste de SOC na Seção 3.1. A Seção 3.2 é voltada ao padrão P1500, que visa padronizar a interface entre distribuidor e usuário de núcleos. Finalmente, a Seção 3.3 conclui este capítulo.

3.1 Arquitetura Conceitual de Teste de SOC

A arquitetura conceitual de teste de SOCs consiste em três elementos, conforme a Figura 22: fonte e receptor de dados de teste, mecanismo de acesso de teste (TAM) e lógica envoltória.



CUT – *Core Under Test*. LDU – Lógica Definida pelo Usuário

Figura 22 – Arquitetura genérica de acesso de teste para núcleos [48].

3.1.1 Fonte e Receptor de Informações de Teste

A fonte gera estímulos para o núcleo. O receptor compara as respostas geradas com as respostas esperadas. Tanto a fonte quanto o receptor podem ser implementados fora do circuito, através de um ATE, ou dentro do circuito, através de BIST. A escolha do tipo de fonte e receptor é determinado por [80]: (*i*) tipo de circuito do núcleo; (*ii*) tipo de teste desenvolvido no núcleo; (*iii*) considerações sobre custo e qualidade de teste. Zorian [80] classifica os tipos de circuitos em três categorias principais: lógica, memória e analógico. Estes três tipos de circuito possuem defeitos diferenciados, portanto são testados diferentemente, requerendo fontes e receptores distintos tanto para ATE quanto para BIST. As vantagens e desvantagens de uso de BIST em relação ao uso de ATE foram apresentadas anteriormente.

3.1.2 Mecanismo de Acesso de Teste

O mecanismo de acesso de teste (TAM) se encarrega de transportar as informações de teste tanto da fonte até o núcleo quanto do núcleo até o receptor. O projeto de um TAM envolve um compromisso entre largura de banda e custo de teste. Uma maior largura de banda implica em maior custo de área de silício e menor tempo de teste. O tempo de teste também é resultante do volume de dados de teste.

Existem as seguintes opções no desenvolvimento de TAM: (*i*) TAM pode reutilizar caminhos já existentes como barramentos ou pode ser formado por um caminho exclusivo para teste; (*ii*) pode passar através dos núcleos ou passar em torno dos núcleos através da lógica envoltória; (*iii*) pode existir um TAM independente para cada núcleo ou o mesmo pode ser compartilhado; (*iv*) TAM pode simplesmente transportar dados de teste ou também ter sinais de controle.

3.1.3 Lógica Envoltória

A lógica envoltória é a interface entre o núcleo e o restante do SOC e o TAM. Ela permite o chaveamento entre o núcleo e os vários mecanismos de acesso. A lógica envoltória obrigatoriamente deve possuir os seguintes modos de operação [80][54]:

- *Modo Normal*, onde a lógica envoltória é transparente para o restante do circuito;

- *Modo de Teste Interno do Núcleo*, onde o TAM é conectado ao núcleo para transporte dos dados de teste para uma cadeia de varredura interna ao núcleo, por exemplo. Neste caso o TAM pode ser serial ou paralelo, porém, devido à pequena largura de banda do TAM serial, costuma-se utilizar TAM paralelo;
- *Modo de Teste Externo*, onde o TAM é conectado à interconexão que provê os dados de teste para os núcleos, sem enviar padrões internamente ao núcleo. Realiza teste das interconexões externas ao núcleo;
- *Modo de Passagem* (*bypass*), permite que vários núcleos sejam conectados a um mesmo TAM.

Uma vez que a largura do TAM é determinada pela largura de banda da fonte e do receptor, e o número de bits de entrada e saída do núcleo é determinado pela sua funcionalidade, uma função normalmente aplicada à lógica envoltória é o uso de conversores serial-paralelo e paralelo-serial para a adaptação das diferentes larguras de interface.

A Seção seguinte apresenta algumas definições de uma proposta de padrão para lógica envoltória chamada IEEE P1500.

3.2 Padrão P1500

Uma vez que o projeto baseado em núcleos envolve dois atores, o provedor e o usuário de núcleos, existe a necessidade de padronizar a interação entre eles. Isto facilita a integração de núcleos de origem de diferentes provedores, aumentando a eficiência do usuário de núcleos. Sendo assim, o padrão P1500[1] [36] está focado na padronização da interface entre usuário e provedores de núcleos. Esta padronização atende [80]: (*i*) uma *linguagem padrão* capaz de expressar todas as informações relacionadas a teste que devem ser transferidas para o usuário do núcleo; (*ii*) uma *lógica envoltória de teste padrão* configurável, que facilite a integração de núcleos no sistema.

P1500 não cobre [48]: (*i*) método de teste interno dos núcleos; (*ii*) integração do teste do sistema; (*iii*) otimizações do teste do sistema; (*iv*) mecanismos de acesso ao teste; (*v*) fonte e receptor de teste. Marinissen [48] alega que estes pontos devem estar completamente nas mãos dos provedores e do usuário de núcleos, e sua padronização não é viável devido às diferentes restrições dos núcleos e sistemas.

Os dois principais elementos do padrão P1500 são a *linguagem de teste de núcleos* (em inglês, *Core Test Language* - CTL) e a *arquitetura escalonável para teste de núcleos*.

3.2.1 Linguagem de Teste de Núcleos

O objetivo da linguagem de teste de núcleos é expressar todas informações relacionadas a teste de forma explícita e concisa e transferi-las do provedor para o usuário de núcleos. O escopo desta linguagem compreende [80]: (*i*) métodos de teste; (*ii*) modos de teste e seus correspondentes protocolos; (*iii*) padrões de teste, e.g. lista de vetores de teste, algoritmos de teste para memórias, polinômios primitivos para BIST; (*iv*) modelos e cobertura de falhas; (*v*) informações da estrutura de DFT interna dos núcleos, e.g. tamanho da cadeia *scan*, BIST; (*vi*) informações de diagnóstico, e.g. localização física de pontos de teste.

Uma vez que as informações contidas nesta linguagem são mais importantes que a sintaxe da mesma, esforços estão sendo concentrados na expansão de uma linguagem já existente, o padrão P1450 (*Standard Test Interface Language* - STIL) [35]. Atualmente esta linguagem esta focada na descrição de padrões de teste no formato de forma de ondas, e necessita de várias extensões para expressar as informações de teste de núcleos.

[1] Este padrão ainda está em fase de desenvolvimento

3.2.2 Arquitetura Escalonável para Teste de Núcleos

Seu objetivo é definir uma interface de hardware uniforme mas flexível, capaz de transportar padrões de teste para o núcleo. A padronização é necessária para garantir a fácil integração dos núcleos e, provavelmente, reduzir o *time to market*. Por outro lado, deve ser flexível para permitir que o provedor e o usuário do núcleo possam explorar compromissos entre qualidade de teste, tempo de teste, área de silício e impacto no desempenho.

O trabalho de padronização está focado no desenvolvimento de uma lógica envoltória com as seguintes características: (*i*) possuir múltiplos modos de operação, e.g. normal, teste de interconexão, teste interno e passagem; (*ii*) conectar qualquer número de portas do núcleo com qualquer tamanho de TAM, exigindo assim mecanismos de adaptação de largura; (*iii*) prevenir o escorregamento de relógio; (*iv*) caminho de controle de teste serial.

A Figura 23 apresenta um esquema da estrutura desta lógica envoltória. Os terminais da lógica envoltória são:

- As entradas e saídas funcionais correspondem às entradas e saídas do núcleo. Sua largura é definida pelo provedor;
- *Wc* é uma porta de 6 bits que determina o modo de operação da lógica envoltória;
- As interfaces de teste serial, *si* e *so*, são utilizadas para carregar o WIR e o TAM serial;
- Zero ou mais TAMs paralelos, chamados de *pi* e *po*, que possuem largura parametrizável, podendo variar para cada lógica envoltória dentro de um sistema. Além disto *pi* e *po* não necessariamente precisam ter a mesma largura;
- Controle de teste dinâmico, e.g. sinal de habilitação de varredura;
- Funções especiais correspondem a sinais que não possuem células associadas como, por exemplo, sinais de relógio ou analógicos.

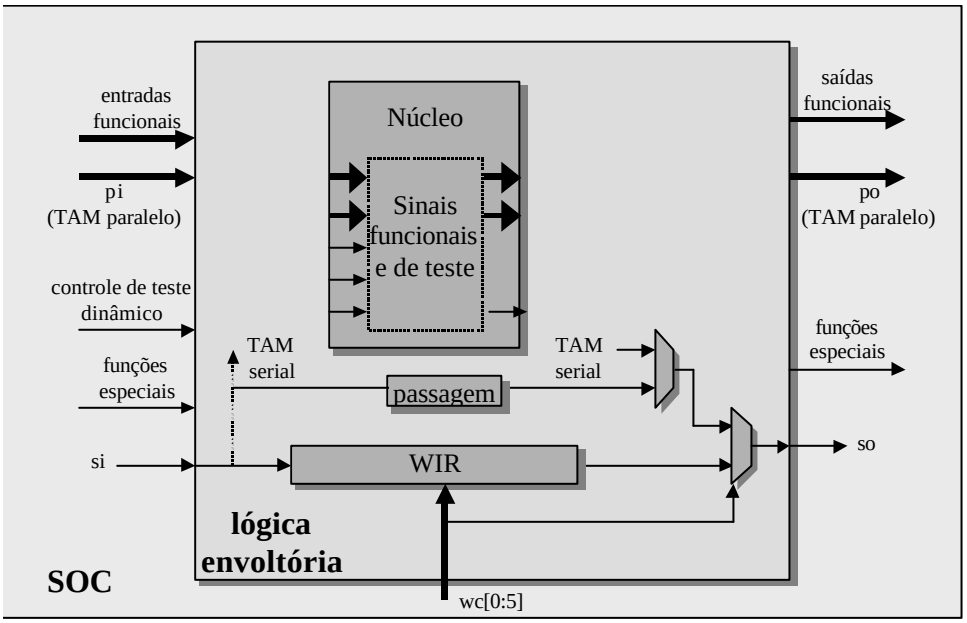


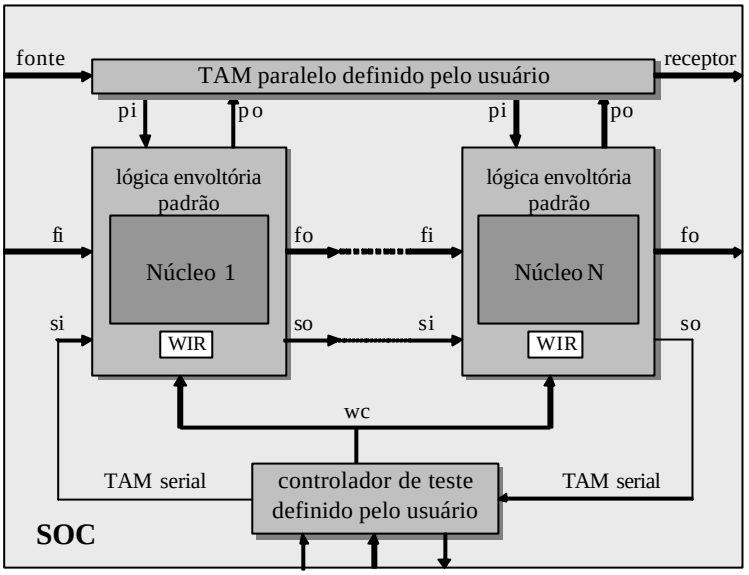
Figura 23 – Visão das interfaces internas e externas da lógica envoltória padrão no nível do núcleo [48].

Os componentes mínimos de uma lógica envoltória são:

- Registrador de instruções da lógica envoltória (WIR), que controla suas operações;
- Múltiplas células de entrada e saída, que fornecem controlabilidade e observabilidade aos terminais do núcleo. A cada terminal do núcleo é atribuída uma célula, sendo que alguns terminais especiais, e.g. relógio, não utilizam tais células;

- Um registrador de um bit, denominado de passagem, que serve para transportar os padrões de teste pela TAM serial;
- Fios de conexão e multiplexadores para selecionar os modos de operação.

A Figura 24 apresenta as conexões dos núcleos no nível do sistema. Os núcleos, que utilizam uma lógica envoltória para teste, estão conectados por uma TAM serial, obrigatória, e uma TAM paralela, opcional, definida pelo usuário. O bloco de controle de teste também deve ser definido pelo usuário.



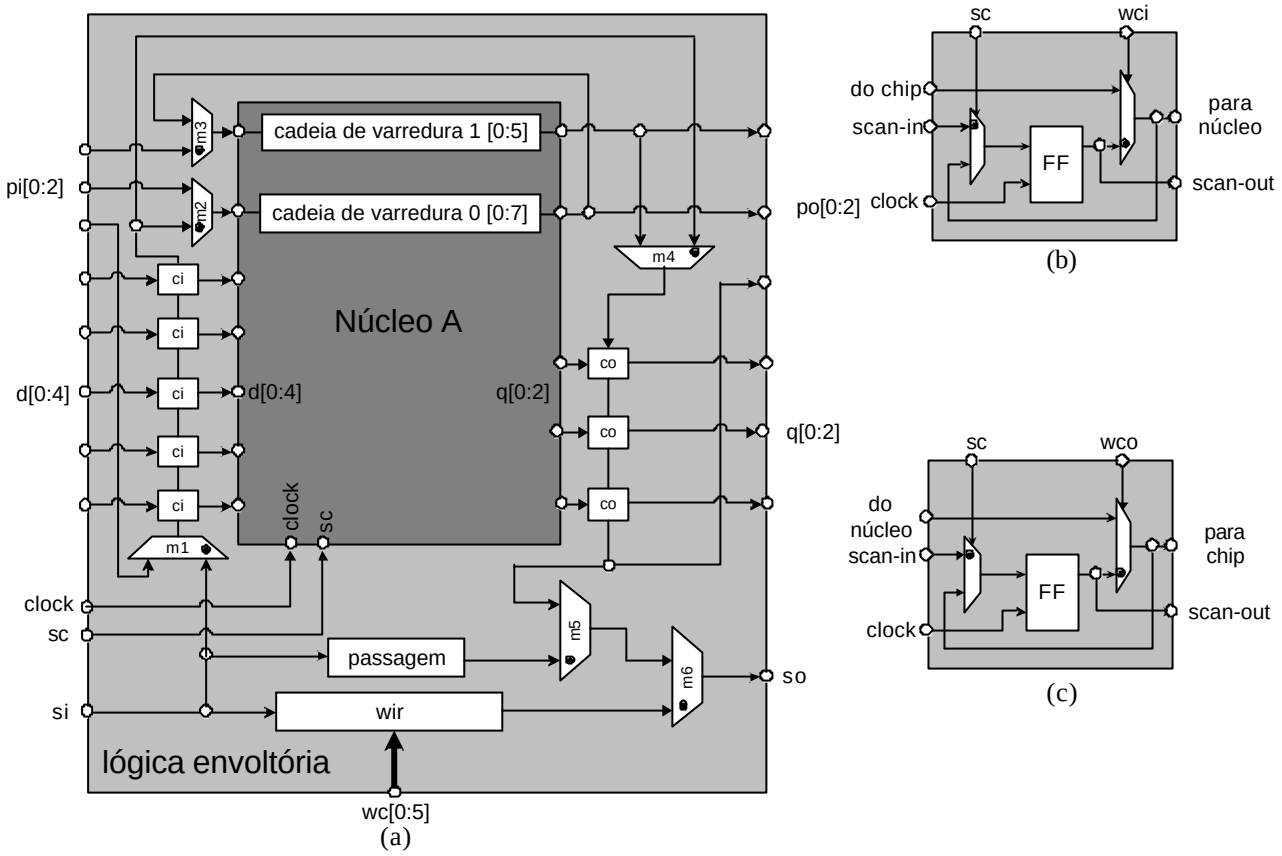
pi e po – entrada e saída paralela
si e so – entrada e saída serial
fi e fo – entrada e saída funcional

Figura 24 – Arquitetura da lógica envoltória padrão. Visão ao nível de sistema, composto por vários núcleos [48].

3.2.3 Exemplo de Uso da Lógica Envoltória

Esta seção apresenta um exemplo de uso da lógica envoltória padrão ilustrada na Figura 25. A lógica envoltória é adicionada a um núcleo simples. Este núcleo, chamado de *núcleo A*, possui duas cadeias de varredura interna, duas portas de dois bits para acesso de entrada e saída destas cadeias, uma porta de entrada de cinco bits, uma porta de saída de três bits, uma porta para habilitar a varredura interna e a porta de relógio. Neste caso específico o núcleo em questão possui um conjunto de padrões de teste pré-definidos, também chamados de padrões determinísticos. Porém, qualquer outro método de geração de padrões, inclusive pseudo-aleatório, poderia ser utilizado.

Para tornar o núcleo A compatível com P1500, uma lógica envoltória e um programa CTL devem ser desenvolvidos. O *programa CTL foi omitido* pois o objetivo desta linguagem é transportar conhecimento de teste do provedor ao usuário do núcleo. Como neste caso o núcleo está sendo desenvolvido por quem irá utilizá-lo, não há necessidade deste programa. A Figura 25(a) apresenta a lógica envoltória conectada ao núcleo. É importante destacar que esta descrição de lógica envoltória possui alguns detalhes de implementação. O padrão P1500 define somente o comportamento da lógica envoltória, deixando a implementação a cargo do projetista. Neste exemplo em particular, a lógica envoltória possui um TAM serial, *si* e *so*, e um TAM paralelo de três bits, *pi* e *po*. A implementação das células de entrada e saída são apresentadas, respectivamente, na Figura 25(b) e Figura 25(c). Nesta implementação da lógica envoltória foram desenvolvidos seis modos de operação que compreendem o modo normal, de teste de conexão dos TAMs serial e paralelo e o modo de passagem serial. Os caminhos ativados em cada um dos modos de operação são apresentados na Figura 26. A Tabela 4 apresenta o estado dos multiplexadores em cada um dos modos de operação.



(a) e a implementação da célula de entrada – ci (b) e da célula de saída – co (c) [48]. Os pontos nos multiplexadores indicam a entrada ativada quando a porta de controle possui o valor 1.

Figura 25 – Exemplo do *núcleo A* mais a lógica envoltória.

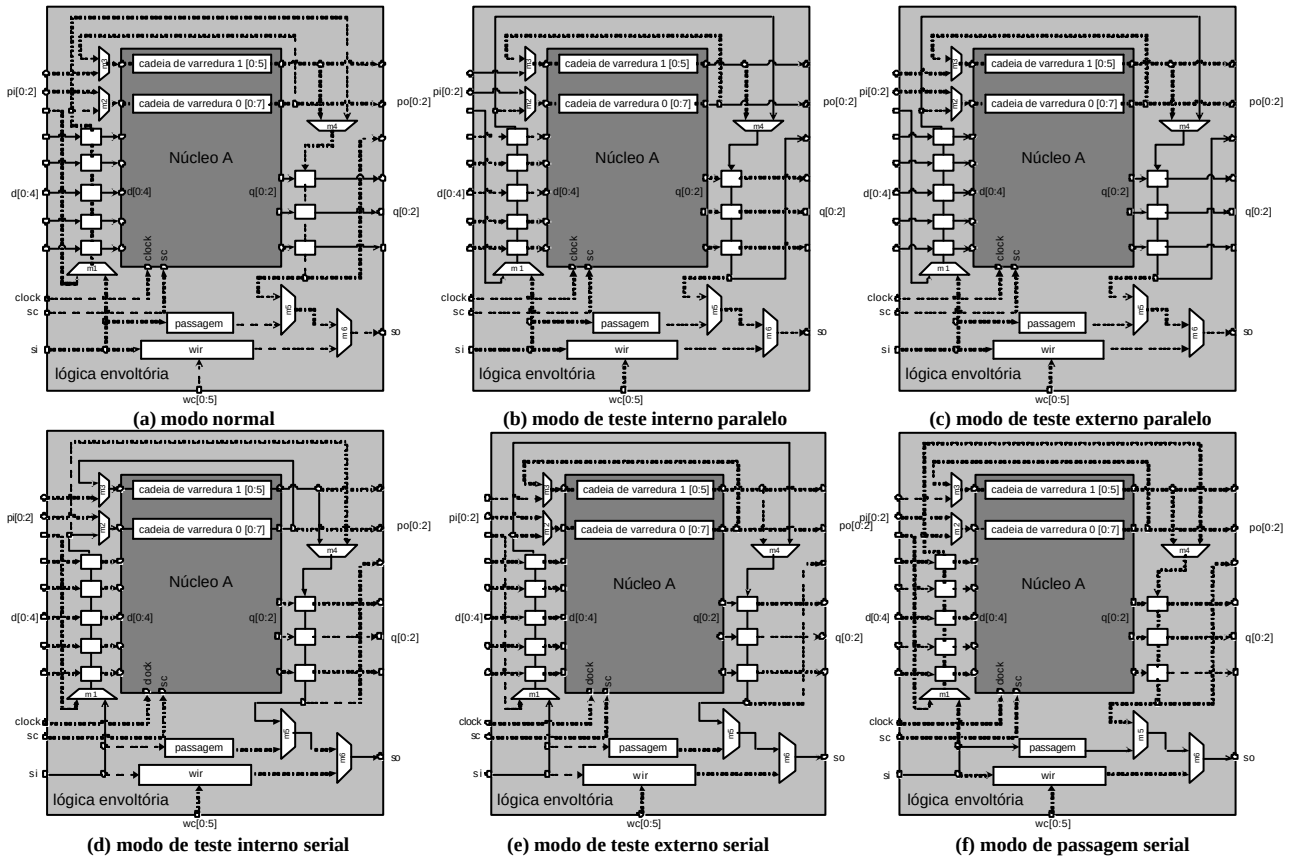


Figura 26 – Os vários modos de operação da lógica envoltória [48].

Tabela 4 – Estado dos multiplexadores durante os modos de operação [48]. O valor x significa que este multiplexador não importa neste modo de operação.

Modos de operação	Estado dos multiplexadores								
	sc	wci	wco	m1	m2	m3	m4	m5	m6
Normal	0	0	0	x	x	x	x	x	x
Teste interno paralelo	1	1	0	0	0	0	1	x	x
Teste externo paralelo	1	0	1	0	x	x	1	x	x
Teste interno serial	1	1	0	1	1	1	0	0	0
Teste externo serial	1	0	1	1	x	x	1	0	0
Passagem serial	x	x	x	x	x	x	x	1	0

3.3 Conclusão

Os conceitos apresentados nesta seção visam facilitar o teste de SOC, criando caminhos de teste para os estímulos e respostas. Porém, métodos de geração de estímulos e captura de respostas, que podem tanto ser origem externa quanto interna ao CI, devem ser definidos pelo usuário. Ao longo do próximo capítulo, propomos o uso de um processador como elemento de controle de teste de SOC, geração e captura dos estímulos gerados a partir de um software de teste.

4 TÉCNICAS DE TESTE BASEADAS EM SOFTWARE PARA DISPOSITIVOS BASEADOS EM NÚCLEOS DE HARDWARE

A adição de módulos de teste, como BIST, pode vir a impedir que os sistemas computacionais operem na freqüência determinada pelo projetista, dado o crescente aumento da freqüência de operação dos SOCs. Assim, são necessárias novas metodologias de teste menos intrusivas, porém, que permitam teste em tempo de execução, pois a redução do tempo de teste também é importante. Em uma revisão bibliográfica, apresentada na Seção 4.2, identificamos que teste de hardware baseado em software ameniza algumas desvantagens das técnicas de BIST. Por este motivo, utilizamos esta técnica neste trabalho.

Durante o desenvolvimento deste trabalho não foi encontrada referência bibliográfica contemplando o teste de um SOC genérico por completo utilizando software para teste de hardware. Rajski *et. al.* [64] utilizaram ABIST (*arithmetic BIST*) para testar a parte de dados de um processador. Chen *et. al.* [18] exploraram a implementação de LFSR e MISR em software como método de geração de estímulos e compactação de respostas para teste de processadores, porém, assumem que a memória foi previamente testada. Rajsuman [65] explorou a programabilidade de processadores embutidos em um SOC para testar memória e um DAC, porém, o processador era testado com técnicas estruturais de BIST lógico. Hellebrand *et. al.* [32] exploraram a flexibilidade de software em relação à hardware para implementar LFSRs com múltiplos polinômios e sementes, que foi demonstrado que pode aumentar a cobertura de falhas. Papachristou *et. al.* [58] utilizaram processadores embutidos em um SOC para testar núcleos, porém, o teste do processador e da memória não foram contemplados.

Este capítulo é organizado da seguinte forma: A Seção 4.1 apresenta as vantagens e desvantagens do uso de software para teste de hardware. A Seção 4.2 apresenta a arquitetura básica necessária para desenvolver teste de um SOC utilizando algoritmos implementados em software. Ao longo desta seção também é apresentada uma revisão bibliográfica do uso de software para testar os diferentes módulos de um SOC como processador, memória e núcleos não programáveis. Finalmente, a Seção 4.3 conclui este capítulo.

4.1 Vantagens e Desvantagens

Além das vantagens já conhecidas de BIST tais como redução do volume de padrões de teste aplicado pelo ATE, redução dos requisitos de freqüência e memória do ATE, realização de teste em tempo de execução, redução do tempo de teste, entre outras vantagens, o teste embutido baseado em software possui as seguintes *vantagens* adicionais:

- Redução ou exclusão total de área de *hardware adicional*;
- Redução ou exclusão total de perda de *desempenho* devido à lógica de isolamento de teste (i.e. multiplexador);
- Redução ou exclusão total da *lógica de controle de teste.* O próprio processador se encarrega do fluxo de teste;
- Fácil *reutilização* ou *modificação* das técnicas de teste;
- Maior *flexibilidade* e *programabilidade* para realização do teste. Basta modificar um código em software para modificar a rotina de teste. Por exemplo, para diminuir a probabilidade de mascaramento das rotinas de compressão pode-se optar por executar o teste 2 vezes com polinômios diferentes [32][67].

Porém, algumas das *desvantagens* inerentes ao uso de software para teste de hardware são:

- SOC deve possuir um *componente programável* como processador, microcontrolador ou DSP. Neste trabalho designamos de processador qualquer um destes três tipos de componente programável. Para resolver esta desvantagem pode-se optar pelo uso de um processador específico para teste. Estes têm a vantagem de serem dedicados para teste, sendo assim menos custosos em área e degradação de desempenho em relação a processadores genéricos [15];
- *Requisitos de memória* para armazenar programa de teste e padrões de teste. A compactação dos vetores de teste determinísticos pode ser utilizada para reduzir esta desvantagem [42];
- *Tempo de teste*. Certos trabalhos demonstram que com a compactação de padrões de teste e a sua descompactação realizada pelo processador pode diminuir o tempo de teste, uma vez que diminui a quantidade de tráfego de dados entre o ATE e o CI [42];
- *Menor dissipação de potência* durante o teste pois o teste ocorre em modo normal de operação [29][43]. Esta característica pode viabilizar o aumento do paralelismo, ou seja, múltiplos processadores testando o SOC, reduzindo o tempo de teste.

4.2 Teste dos Componentes de SOCs

O teste dos principais componentes de um SOC, utilizando técnicas de software é aplicado para: (*i*) teste do processador; (*ii*) teste de memória; (*iii*) teste dos núcleos não programáveis.

4.2.1 Teste de Processadores

Não faz parte do objetivo deste trabalho testar processador com teste baseado em software. Porém, apresentamos esta revisão no intuito de mostrar as dificuldades de realizar tal teste e também como forma de guiar trabalhos futuros. Teste do processador é uma tarefa difícil devido aos seguintes fatores: (*i*) número de dispositivos lógicos existentes, e.g. ULA, controle, registradores, entre outros; (*ii*) número limitado de pinos para acessar o processador; (*iii*) existência de dispositivos que não podem ser escritos ou lidos diretamente; (*iv*) grande variação de implementação dos processadores. Além destas características inerentes da arquitetura de processadores, outros obstáculos dificultam a realização do teste, entre eles: (*i*) uma vez que processadores utilizam o estado da arte da tecnologia para a sua concepção, a adição de cadeias de varredura compromete as restrições de área e desempenho do processador; (*ii*) métodos de detecção de falhas baseada em *stuck-at* são inviáveis devido à grande área de hardware necessária para desenvolver um processador e ao fato de nem sempre ter disponível a descrição deste.

Dados os problemas relacionados, Thatte e Abraham [73] propuseram um modelo de falhas funcional para processadores. Uma vez que a metodologia tinha que ser geral, a arquitetura do processador foi abstraída. Desta forma, este modelo pode ser aplicado a um grande número de processadores. Verhallen e Van de Goor [76] propuseram uma atualização ao modelo de falhas desenvolvido por Thatte, para suportar teste de cache de dados e instruções embutidos no processador. Estes dois trabalhos, principalmente o escrito por Thatte, incentivaram a pesquisa de técnicas de teste funcional para validar processadores [18][20][44][64][67][72]. Esta técnica baseia-se na escolha de um conjunto de instruções aleatórias ou de acordo com algum critério que tem como objetivo avaliar a funcionalidade do processador. O método de escolha de instruções aleatórias mostrou que possui uma baixa cobertura de falhas, tornando-se inadequado [9][41]. Isto é devido à complexidade da parte de controle dos processadores, que é altamente resistente a padrões pseudo-aleatórios [18][32][72]. Além disto, o tempo de teste tende a ficar grande. A alternativa defendida por diversos autores como Chen [18], Shen [67], Corno [20] e Tehranipour [72] é na escolha detalhada das instruções necessárias para testar o processador. Lai e Cheng [44] propõem a adição de instruções específicas para realizar teste. Paschalis *et. al.* [60] defendem o uso de padrões determinísticos cuidadosamente escolhidos para testar o bloco de dados de processadores. O critério utilizado varia de abordagem para abordagem, mas em geral, é proposto um método para alcançar

um conjunto mínimo de instruções com uma maior cobertura de falhas.

Chen *et. al.* [18] desenvolveram um método efetivo de teste para processador através de uma análise de restrições espaciais e temporais do conjunto de instruções. Porém, este cobre somente o bloco de dados do processador, não contemplando a parte de controle.

Shen *et. al.* [67] apresentaram uma ferramenta de geração automática de teste funcional que provê um ambiente único tanto para teste de fabricação quanto para validação do projeto. Este ambiente suporta geração de código para validação, para teste com ATE e para auto-teste. A diferença entre teste com ATE e auto-teste é que no modo com ATE os dados gerados são enviados para os pinos de entrada e, posteriormente, as respostas são lidas pelos pinos de saída. Já no auto-teste os dados gerados são comprimidos por uma rotina similar a um MISR. Utilizando tal ferramenta para teste do processador 8051, foi obtido 90.2% de cobertura de falhas, sendo que o programa de teste foi gerado em somente 3.1 minutos e o tempo de simulação foi 4.1 horas.

Corno *et. al.* [20] utilizaram o conceito de macro. Uma macro é gerada para cada instrução que é testada. Uma macro é um conjunto de instruções agrupadas de forma a carregar os operandos para a instrução alvo, executar a instrução alvo e propagar os resultados desta instrução para a memória. Desta forma, existe um trabalho manual de criação das macros. Corno alega que, para o processador 8051, leva-se 2 dias de trabalho de um programador *assembly* experiente para construir a biblioteca com 213 macros. Por fim, a versão final do programa de teste é formada por uma seqüência de macros cujos operandos são selecionados por um algoritmo genético. Esta seleção de operandos levou 24 horas em uma estação de trabalho de 400 MHz com 2Gbytes de RAM. A metodologia proposta obteve uma cobertura de falhas de 85,19% contra 80,19% da metodologia baseada na escolha aleatória de instruções.

Tehranipour *et. al.* [72] atribuíram uma rotina de teste específica para cada bloco do processador, por exemplo ULA. Os operandos destas rotinas são formados por padrões determinísticos e/ou pseudo-aleatórios. A seleção destes operandos é feita por uma investigação prévia não detalhada no artigo. Também é realizado o teste de memória por software. Porém, não foi detalhado o método de teste da parte de controle que é uma das partes mais difíceis.

Cheng Lai *et. al.* [44] alegam que a baixa cobertura de falhas atingida no teste de processadores é devida a circuitos de baixa controlabilidade e observabilidade como circuitos de controle de exceção, de interrupção e alguns registradores. Desta forma eles propõem a adição de um pequeno número de instruções no processador para aumentar a controlabilidade e a observabilidade destes circuitos. Algumas instruções também foram adicionadas para diminuir o tempo de teste e o tamanho do programa de teste. Para isto, seqüências de instruções freqüentes foram condensadas em uma única instrução. Foram utilizados dois processadores para avaliar a técnica. Em ambos os casos, 100% de cobertura foram alcançados com uma redução média de 20% no tempo de teste e tamanho do programa com o acréscimo de 1.6% de área. Porém, esta metodologia não pode ser aplicada para processadores *firm* ou *hard cores*, pois implica na modificação do conjunto de instruções do processador.

Paschalis *et. al.* [60] demonstraram vantagens de aplicação de teste determinístico ao invés de teste pseudo-aleatório. Teste pseudo-aleatório depende da arquitetura e da largura de barramento do processador. Por exemplo, uma ULA de 16 e uma de 32 bits utilizam polinômios diferentes para implementar um LFSR. Com isto, é necessário reescrever código e refazer a simulação. Além disto a execução de LFSR em software requer vários ciclos de relógio para a execução das rotinas. Paschalis alega que com teste determinístico é possível escrever rotinas de teste para MAC, ULA e registradores de deslocamento independente da arquitetura interna e da largura do barramento e com número reduzido de padrões de teste, reduzindo tempo de teste.

Nedwal [55] apresenta uma aplicação industrial de teste baseado em software. Nedwal desenvolveu manualmente um programa de teste para o microcontrolador CR16B. O tamanho deste código de teste é de cerca de 1Kbyte. Este reduzido tamanho foi alcançado pois não foram considerados todas as instruções e seus diferentes modos de operação. Somente as instruções mais

utilizadas (e.g. de carga, escrita, soma, comparação, salto, operações *e* e *ou* lógico) foram testadas. Esta lista de instruções foi selecionada varrendo diversos códigos em *assembly* gerados a partir de um compilador C. O tempo necessário para execução deste programa de teste não foi apresentado.

4.2.2 Teste de Memória

O teste da memória é o mais simples de realizar em um SOC por um processador. O motivo é que geralmente o teste de memória RAM utiliza padrões determinísticos, e.g. 55h e AAh. Desta forma, somente um trecho de código é necessário. Porém, o tempo de teste varia de acordo com os tipos de falhas que o algoritmo utilizado se propõem a encontrar. A Tabela 3 na página 19 apresentou uma lista de algoritmos e suas complexidades. Devido à simplicidade de implementar teste de memória por processador, existem diversas referências que o implementam [65][72].

Rajsuman [65] utiliza um código *assembly* do algoritmo *March* [75], apresentado na Figura 27, para testar uma memória de 2Mbits.

```
1.
2.  /* Procedimento para testar RAM embarcada */
3.  /* A0 é o contador de endereços, D0 contém dados de teste
4.  (escrita), D1 é usado para ler dado (resposta);
5.  A0, D0-D2 são registradores de propósito geral */
6.
7.  /* INICIALIZAÇÃO */
8.           MOVE       #0000H, A0    /* inicializa contador de endereço*/
9.           MOVE       #0000H, D0    /* inicializa dado*/
10.          MOVE       #0000H, D2    /* zera memória após leitura */
11.
12. /* PROCEDIMENTO DE TESTE (zera memória)*/
13. initial: MOVE       D0, [A0]      /* D0 é escrito na posição A0*/
14.          INCR       A0
15.          COMP       A0, FFFF#     /* FFFFH é o último endereço */
16.          BEQ        test_incr
17.          BRA        initial       /* inicia memória e o teste*/
18.
19. /* ESCREVE E LÊ EM ORDEM CRESCENTE */
20. test_incr:MOVE      #0, A0
21.          MOVE       #5555H,D0     /* este dado é (0101...)*/
22. cont_incr:MOVE      D0, [A0]      /* operação de escrita*/
23.          MOVE       [A0], D1      /* operação de leitura */
24.          MOVE       D2,[A0]       /* zera a palavra de memória */
25.          COMP       D0, D1
26.          BEQ        next_incr
27.          BRA        fail          /* valor lido não é 5555H*/
28. next_incr: INCR     A0
29.          COMP       A0, FFFFH     /* último endereço */
30.          BEQ        test_decr
31.          BRA        cont_incr
32.
33. /* ESCREVE E LÊ EM ORDEM DECRESCENTE */
34. test_decr: MOVE     #AAAAH, D0
35. cont_decr: MOVE     D0, [A0]      /* operação de escrita */
36.          MOVE       [A0], D1      /* operação de leitura */
37.          MOVE       D2,[A0]       /* zera a palavra de memória */
38.          COMP       D0, D1
39.          BEQ        next_decr
40.          BRA        fail          /* valor lido não é AAAAH*/
41. next_decr: DECR     A0
42.          COMP       A0, 0000H     /* 0000H é o último endereço*/
43.          BEQ        done
44.          BRA        cont_decr
45.
46. done: /* teste aprovado */
47. fail: /* teste reprovado */
```

Figura 27 – Procedimento em linguagem *assembly* do algoritmo *March* de teste de memória [65].

Rajsuman argumenta que com esta técnica diminuíram-se significantemente os requisitos de um ATE, provendo teste em tempo de execução, sem degradação de desempenho, sem área de silício adicional e com a facilidade de poder trocar o algoritmo de teste sem alterar o hardware, provendo maior portabilidade. O algoritmo apresentado na Figura 27 é simples e está bem comentado. São executadas operações de leitura e escrita em ordem crescente (0101...) e decrescente (1010...) em toda a memória. Assim que o valor lido divergir do valor escrito, é acusada uma falha e o procedimento é finalizado. De acordo com [75](ver Figura 20, página 20), o algoritmo *march* é um algoritmo que, apesar da simplicidade, apresenta as melhores coberturas de falha (aproximadamente 95%).

Zorian *et. al.* [79] apresentam um esquema para teste de ROM. O autor considera uma memória ROM como um circuito combinacional, onde o endereço é a entrada e o conteúdo é a saída. Desta forma é utilizado teste exaustivo, leitura de toda ROM, compactando a saída. É utilizado um método de compactação baseado em MISR bidirecional e contador de 1s para diminuir a probabilidade de mascaramento. Esta abordagem pode ser tanto utilizada em teste de fabricação quanto em teste de campo.

Para teste de campo, um algoritmo adequado para teste de memória RAM é o BIST transparente [56]. Este algoritmo de teste utiliza o próprio conteúdo da memória e seu complemento como padrão de teste. Além disto, este algoritmo pode ser facilmente implementado em software.

No caso do teste de fabricação, deve haver um ATE simples que apenas carrega o programa de teste no momento em que o mesmo seja necessário. Já em teste de campo, o programa de teste deve estar armazenado em uma ROM, bastando, por exemplo, aplicar teste de ROM apresentado anteriormente para testá-la.

4.2.3 Teste de Núcleos de Hardware não Programáveis

Para o teste dos núcleos não programáveis, o processador deve executar um programa que é responsável por *acessar os núcleos*, *gerar os estímulos* e *capturar ou compactar as respostas* dos núcleos testados. Devido à programabilidade dos processadores, é possível utilizar diferentes programas para geração de estímulos e compactação de respostas. Esta flexibilidade permite utilizar um algoritmo diferenciado de geração e de compactação para cada núcleo a ser testado, de acordo com cada caso. O acesso ao núcleo depende de alguma estratégia para levar os estímulos e capturar as respostas do núcleo. A acessabiblidade dos núcleos pelo processador depende da arquitetura do SOC. Por exemplo, em uma arquitetura de barramento provavelmente o acesso do processador é feito diretamente pelo barramento. Em arquiteturas onde o processador não tem acesso direto ao núcleo, estratégias de construção de TAMs são necessárias. Alguns exemplos das técnicas de acesso, geração e compactação de dados são apresentados a seguir.

Gupta *et. al.* [30] propuseram um método de geração de padrões pseudo-exaustivo de teste baseado em somadores. Uma vez que o somador existe na maioria das arquiteturas, não há degradação de desempenho e adição de hardware. Os padrões são gerados acumulando continuamente um valor constante escolhido de acordo com um critério. Estes padrões são gerados pela tripla (a, X_0, n) onde a representa a constante a ser acumulada, X_0 representa o valor inicial do acumulador e n a largura do acumulador.

Hellebrand *et. al.* [32] destacam o ganho de flexibilidade com uso de teste baseado em software. Com esta flexibilidade adicional, é possível explorar configurações diferenciadas de LFSRs de forma a alcançar a maior cobertura de falhas possível, pois se sabe que existem circuitos resistentes a padrões pseudo-aleatórios. Hellebrand alega que com esta flexibilidade é possível explorar a melhor configuração de gerador de padrões para cada módulo. Dentre as configurações estudadas estão o LFSR simples, LFSR com múltiplos polinômios e LFSR com múltiplos polinômios e múltiplas sementes. Foi demonstrado, utilizando *benchmarks*, que cada circuito responde melhor a uma dada configuração de LFSR. Esta flexibilidade adicional e a escolha adequada de gerador de padrões para cada módulo acarretam uma maior cobertura de falhas. Esta

maior cobertura de falhas pode tornar viável armazenar em memória padrões determinísticos de forma a alcançar 100% de cobertura de falhas. Por exemplo, o circuito c7552, pertencente ao ISCAS85 [13], é conhecido por ser resistente a padrões pseudo-aleatórios. Com um gerador de padrões baseado em um LFSR simples, de um único polinômio, se atinge uma cobertura de falhas de 95.79%, restando 4.21% para padrões determinísticos. Por outro lado, com o uso de um LFSR com múltiplos polinômios e múltiplas sementes é possível alcançar uma cobertura de falhas de 98.87%, restando somente 1.13 % ou, em valores absolutos, somente 84 falhas. Esta maior cobertura de falhas acarreta uma economia de memória para armazenar padrões determinísticos.

Em conjunto com a abordagem de Hellebrand, pode ser aplicada a abordagem de Jas [39] que utiliza um algoritmo embarcado para descomprimir as informações de teste que estão na memória. Esta abordagem diminui ainda mais a necessidade de memória e a restrição de largura de banda de comunicação entre o ATE e o circuito.

Radecka *et. al.* [62] demonstraram, utilizando *benchmarks*, que um bloco de ULA ou *multiply and accumulate* (MAC) pode gerar estímulos com qualidade comparável a um LFSR. Em certos casos, MAC precisa de menos vetores de teste para alcançar a mesma cobertura de falhas que um LFSR.

Stroele [69] demonstra que somadores, subtratores e multiplicadores podem ser utilizados para a geração de padrões pseudo-aleatórios com uma cobertura de falhas e comprimento de teste similar a um LFSR. Isto traz como vantagem o tempo de teste, pois a emulação de LFSR necessita de vários ciclos de relógio para a sua execução. Para esta demonstração foram utilizados *benchmarks*, sendo medido o número de padrões necessários para alcançar 100% de cobertura.

Dorsch *et. al.* [24] aplicam a técnica baseada no uso de múltiplas sementes a somadores. Esta técnica, normalmente atribuída a LFSR, é utilizada para codificar uma série de padrões determinísticos. A vantagem é que basta armazenar somente um dado, que é a semente, para poder gerar pseudo-aleatoriamente uma série de padrões determinísticos. Assim, algumas sementes são necessárias para codificar um conjunto inteiro de padrões determinísticos. Os resultados obtidos são semelhantes aos obtidos com uso de LFSR, porém, sem o impacto em área de hardware e desempenho. O problema deste esquema de múltiplas sementes é que o cálculo do número mínimo de sementes necessárias para representar um conjunto de padrões determinísticos é muito complexo, aplicado somente a conjuntos de padrões pequenos.

Rajski *et. al.* [63] avaliam, em termos de probabilidade de mascaramento, uma técnica de compactação de respostas baseada no uso de acumuladores. Os autores provam que acumuladores com sinal de *carry* rotativo[1] apresentam uma probabilidade de mascaramento semelhante a um MISR. A vantagem desta abordagem em relação ao MISR é que não há necessidade de hardware adicional para a compactação de respostas, portanto não há degradação de desempenho. Outra vantagem é o menor tempo de execução da compactação.

Stroele [70] critica o método de avaliação do trabalho de Rajski *et. al.* [63] e propõe outra avaliação. Porém, o resultado final desta nova avaliação é compatível com os resultados de Rajski, ou seja, acumuladores com sinal de *carry* rotativo são semelhantes a MISRs em termos de probabilidade de mascaramento.

Papachristou *et. al.* [58] apresentam uma abordagem efetiva de teste de núcleos não programáveis utilizando um processador, porém, não cobrem teste do processador e de memória. Na arquitetura proposta existem três módulos principais: o processador, a memória e um testador externo de baixo custo. O processo de teste foi dividido em duas etapas: de carga de memória e de teste. A carga da memória é realizada pelo testador através de acesso direto à memória (DMA). O teste é realizado pelo processador através da execução do programa de teste. Desta forma, foi feita uma sobreposição no tempo destas duas etapas, carga e teste, diminuindo o tempo total de teste. Antes da etapa de carga deve ocorrer uma análise do tempo estimado para carga e teste para cada núcleo do SOC, pois cada núcleo possui um tempo de carga e de teste diferentes e a ordem em que

[1] Somadores onde o sinal de *carry-out* é rotacionado para o bit menos significativo.

os dados são enviados para a memória também influencia no tempo total de teste. Outra proposta importante é um método baseado em grafos para escalonamento de teste. As informações obtidas através da análise deste grafo são utilizadas para que o processador monte adequadamente o caminho que os estímulos e respostas devem traçar do processador ao núcleo e do núcleo ao processador, garantindo sempre o caminho com menor custo de tempo. Uma das conclusões mais interessantes, obtidas através do exemplo demonstrado, é que o tempo de teste do sistema fica praticamente independente do testador externo. Foi utilizado um testador simples de 50MHz e constatou-se que o tempo de teste depende somente da freqüência de operação do processador. Quando a freqüência do processador é duplicada, o tempo de teste praticamente é reduzido pela metade.

4.3 Conclusão

Teste baseado em software minimiza os dois principais problemas relacionados ao uso de BIST: perda de desempenho devido à inserção de circuitos no caminho crítico e o conseqüente aumento da área de hardware. A minimização destes problemas possibilita o auto-teste de circuitos de alto desempenho e de difíceis restrições de concepção, tipicamente restrições de freqüência de operação e área. Porém, novos desafios surgem como o aumento do tempo de teste e o aumento da área de memória para teste. A redução do tempo de teste pode ser alcançada acrescentando instruções no processador específicas para teste e projetando um misto entre teste baseado em software e em hardware. A área de memória para teste pode ser reduzida utilizando múltiplos polinômios e múltiplas sementes, e algoritmos de compactação.

Na revisão bibliográfica apresentada, foram encontradas algumas aplicações acadêmicas e industriais de teste baseado em software. Porém, se identificou que a *automatização* e *avaliação* deste processo ainda são incompletas. No que diz respeito à *automatização* de técnicas de teste baseadas em software, nota-se que há várias abordagens de geração automática de código de teste para processadores. Também se nota que existem várias aplicações de software para testar memórias. Porém, não foram encontrados trabalhos que automatizassem o processo de geração de código de teste para núcleos não programáveis. No que concerne à *avaliação* de técnicas de teste baseadas em software, não existem trabalhos que demonstrem de forma quantitativa suas vantagens e, principalmente, suas desvantagens [43].

Dadas estas lacunas, apresentamos no Capítulo 7 um ambiente de desenvolvimento que automatiza o processo de geração de código de teste de núcleos não programáveis. Em seguida, no Capítulo 8, avaliamos as técnicas de teste empregadas. Porém, antes de apresentar o ambiente de teste desenvolvido, apresentamos o ambiente de co-simulação implementado para validação das técnicas de teste baseadas em software (Capítulo 5) e o ambiente de prototipação (Capítulo 6) utilizado para validar as descrições de hardware geradas.

5 CO-SIMULAÇÃO

A atual complexidade dos sistemas computacionais é tal que o projeto não pode mais ser realizado com uma única linguagem nem com um único nível de abstração. Desta forma, diferentes linguagens e modelos de computação são utilizados para cada um destes domínios. Na concepção de sistemas heterogêneos complexos, seus módulos são desenvolvidos e validados em separado. Porém, em certas etapas do projeto é necessária uma validação total do sistema. A validação deste tipo de sistema é uma tarefa complexa devido a sua heterogeneidade. Entretanto, atualmente a técnica de validação mais utilizada é a simulação, uma vez que as técnicas de verificação formal ainda são imaturas [25]. O princípio da co-simulação heterogênea é a execução paralela dos simuladores necessários para a validação de um sistema. Cada simulador é responsável por executar um módulo do sistema em uma linguagem, modelo de computação e nível de abstração específico ao domínio tratado [33].

Neste capítulo apresentamos nas Seções 5.1 e 5.2 uma breve revisão de modelagem de sistemas computacionais e conceitos básicos de co-simulação. A Seção 5.3 apresenta a implementação de um ambiente de co-simulação, sendo esta a primeira *contribuição* deste trabalho. O desenvolvimento de um ambiente de co-simulação é necessário para validar a interação dos algoritmos de teste implementados em software com os núcleos implementados em VHDL. A Seção 5.4 apresenta estudos de caso de aplicações e avaliações de desempenho da ferramenta desenvolvida. Por fim, a Seção 5.5 conclui este capítulo e apresenta direções para trabalhos futuros.

5.1 Modelagem de Sistemas Computacionais

A grande variedade de modelos de computação empregados é devido às diferentes funcionalidades envolvidas em um sistema computacional. Por exemplo, para descrever um sistema composto de hardware e software poderíamos escolher as linguagens VHDL e C, respectivamente, para sua especificação. Porém, se a decisão de projeto for adotar uma *única linguagem* (modelagem homogênea) de especificação, teríamos C descrevendo hardware e, naturalmente, software ou VHDL descrevendo software e hardware. Esta abordagem teria a grande vantagem de necessitar de um único conjunto de ferramentas para validar todo o sistema [57]. É evidente que a escolha de uma única linguagem apresenta limitações devido ao fato do modelo de computação empregado não ser aplicado aos dois domínios (hardware e software). Apesar disto, algumas abordagens tentam diminuir estas limitações adicionando bibliotecas que incluem funcionalidades que implementam diferentes modelos de computação (e.g. SystemC [11]). Outras abordagens tentam definir uma única linguagem que possua todos os conceitos necessários para a especificação de um sistema, porém, é difícil definir tal linguagem (e.g. SpecC [28]).

A alternativa defendida neste trabalho é manter as diferenças conceituais dos diferentes domínios. Isto implica em modelar cada módulo do sistema em uma linguagem específica e apropriada (*modelo heterogêneo multi-linguagem*). Porém, o problema principal no uso desta abordagem é definir a semântica de interação de modelos computacionais diferentes e uma técnica de sincronização entre os processos (simuladores) [25]. Desta forma linguagens como SDL e C++ podem ser utilizadas em descrições mais abstratas, matlab pode ser usado para modelar partes mecânicas e processos físicos, VHDL-AMS pode ser utilizado para descrição de hardware analógico, C e assembly para descrever software e VHDL para descrever hardware digital.

5.2 Características de um Ambiente de Co-simulação

- **Motor de Simulação**:

A *Co-simulação mono-motor* consiste em transformar os módulos descritos em diversas linguagens em uma mesma representação que descreve todo o sistema. A simulação ocorre nesta representação única do sistema, chamada de formato intermediário. A vantagem desta abordagem é a necessidade de uma única ferramenta de simulação para validar todo o sistema. Porém, tem a desvantagem de ser difícil converter qualquer linguagem com qualquer modelo de computação em um formato intermediário, pois seria difícil desenvolver um formato intermediário que suporte tais variações.

Na *co-simulação multi-motor* cada módulo é simulado por um simulador apropriado para cada linguagem. A co-simulação se resume a uma troca de mensagens entre os simuladores. A vantagem deste método é que permite a utilização das ferramentas existentes para realizar a simulação. Porém, é necessário que os simuladores envolvidos permitam o envio de seus dados de simulação para o exterior, via uma interface de comunicação. Além disto surge a necessidade da utilização de um método de sincronismo para coordenar a troca de dados entre esses simuladores que estão executando paralelamente ou concorrentemente. A Figura 28 ilustra estes conceitos.

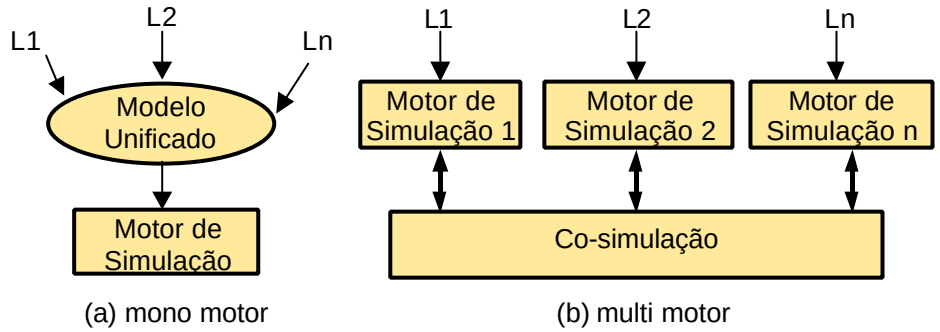


Figura 28 – Motor de simulação.

- **Mecanismos de Comunicação:**

Se for adotado o conceito de múltiplos motores de simulação, tem-se que escolher algum mecanismo de comunicação entre os diversos simuladores. A escolha do mecanismo de comunicação é feita através de um compromisso entre desempenho, flexibilidade e suporte à co-simulação distribuída. Mecanismos de comunicação entre processos como *pipes*, por exemplo, não suportam comunicação distribuída. Por outro lado *sockets* além de suportar comunicação distribuída, é suportada por diversos sistemas operacionais e linguagens de programação. Já mecanismos como *corba* e *RMI* conferem ao sistema uma maior flexibilidade e suporte à simulação distribuída.

- **Modelo Temporal**:

A *validação funcional* consiste em validar a funcionalidade do sistema sem sincronização temporal dos simuladores durante a execução da co-simulação. A troca de dados entre simuladores é controlada por eventos. A validação funcional é utilizada em níveis de abstração elevados onde a noção de tempo não é necessária à simulação.

A *validação temporal* permite realizar uma validação mais realista do sistema, considerando o tempo. O envio e a recepção de dados acontece em instantes precisos de tempo, chamados de janelas temporais. Assim, os diferentes simuladores são sincronizados e todos possuem o mesmo relógio global de simulação. Quando um dado não é consumido em um intervalo específico de tempo, este dado pode ser perdido. Este tipo de validação é muito utilizado para a validação de um sistema especificado em um baixo nível de abstração, onde se torna necessário realizar uma simulação precisa.

- **Modelo de Sincronização**:

Ao se optar pelo desenvolvimento de um ambiente que suporte co-simulação temporal deve-se definir qual é o modelo de sincronismo utilizado, já que os diversos simuladores que compõem o sistema podem ter velocidades de execução e modelos temporais diferentes. A função básica do modelo de sincronização é coordenar o envio e recepção de dados através do ambiente de co-simulação. A Figura 29 ilustra a diferença destas duas opções de sincronização.

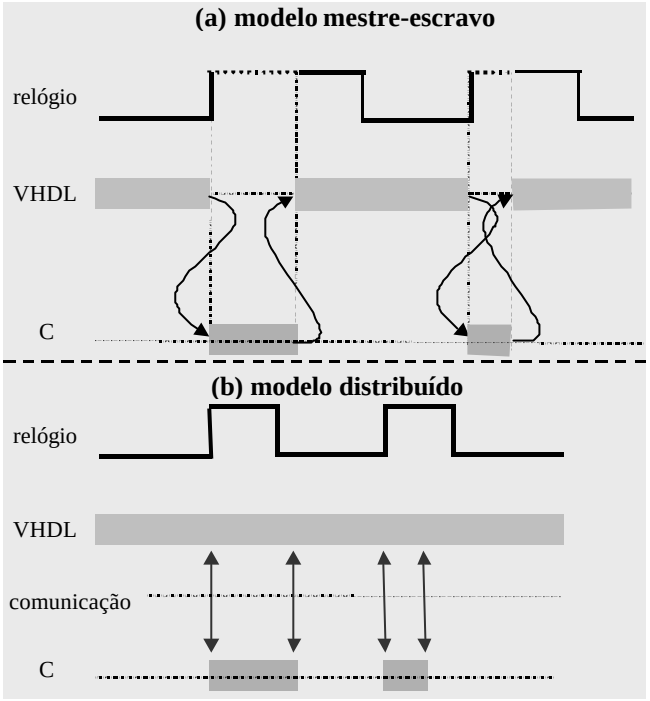


Figura 29 – Modelo de sincronização.

Existem diversas abordagens para sincronização. A mais simples é o *modelo mestre/escravo* que consiste em designar um simulador como mestre da simulação, o qual fica responsável por invocar os outros simuladores escravos [33]. A maior vantagem deste modelo é a simplicidade. Porém, este modelo restringe a arquitetura do sistema aos sistemas mestres/escravos, formados por um processador mestre ao qual são associados coprocessadores escravos. Este modelo de sincronização elimina toda forma de paralelismo entre os módulos.

O *modelo distribuído* permite a execução concorrente dos simuladores. Cada simulador pode enviar e receber dados sem restrições. Este modelo apresenta a vantagem de não restringir a arquitetura do sistema e assim permitir a sua utilização em um grande número de aplicações [33]. No entanto, a coerência entre os dados compartilhados pelos simuladores torna-se mais difícil de assegurar. A principal metodologia empregada para este modelo consiste em conectar cada simulador em um barramento de co-simulação, o qual é encarregado do envio e recepção de dados, e da sincronização entre os simuladores. Este barramento age como um servidor de comunicação e sua implementação pode ser baseada em mecanismos de comunicação entre processos, como por exemplo *ipc* ou *sockets* no caso do sistema UNIX.

Alguns dos principais *desafios encontrados no desenvolvimento de um ambiente de co-simulação* são sincronização, conversão de dados, desempenho de comunicação (tempo de simulação), facilidade de integração de novos simuladores e compromisso entre desempenho e precisão. Algumas soluções da literatura para estas decisões de projeto do ambiente de co-simulação são apresentadas:

- **Sincronização**:

Quando se opta por desenvolver um ambiente de co-simulação distribuído com co-simulação temporal deve-se escolher o método de sincronização entre os simuladores. Diversos algoritmos são encontrados na literatura como, por exemplo, o algoritmo "*synchronized handshake*".

- **Conversão de dados**:

Uma vez que o sistema é composto por diversos modelos de computação que podem conter tipos de dados diversos, é necessário definir uma estratégia responsável pela conversão de dados. Na Seção 5.3.2 apresentamos a solução de conversão de dados implementada.

- **Desempenho de comunicação**:

A co-simulação distribuída é a capacidade de simular um projeto em máquinas geograficamente distribuídas conectadas por uma LAN (*local area network*) ou WAN (*wide area network*). Algumas das motivações para a pesquisa neste campo são: descentralização do projeto, facilidade de cooperação entre diferentes grupos do projeto, gerenciamento de propriedade intelectual, gerenciamento de licenças de simuladores e compartilhamento de recursos. Porém, a desvantagem desta abordagem é que o desempenho da simulação se torna dependente do desempenho da rede. Pelo fato de permitir a execução paralela e concorrente de diversos simuladores, outro problema que surge é a necessidade de sincronização destes simuladores.

- **Facilidade de integração de novos simuladores**:

A abordagem tradicional para integração de novos simuladores é baseada na construção caso a caso das interfaces entre o ambiente de co-simulação e o simulador [50]. Quando um novo simulador for inserido no sistema, uma nova interface deve ser desenvolvida com pouco ou nenhum reuso de código, uma vez que na maioria dos casos as funções do simulador que provêm comunicação com o mundo externo são proprietárias. Deste ponto de vista, a maioria das abordagens são adequadas para atender um conjunto de problemas pré-definidos. Porém, problemas inesperados podem exigir a integração de novos simuladores. O desenvolvimento de uma nova interface pode ser inaceitável por questão de tempo ou por impor restrições ao simulador para se adaptar ao ambiente de co-simulação. Estas restrições poderiam comprometer, por exemplo, a precisão da simulação.

Porém, em [50] é apresentada uma abordagem mais flexível que as abordagens atuais. Este trabalho utiliza um padrão IEEE, chamado HLA (*high level architecture*), que propõe regras e mecanismos para facilitar interoperabilidade de simuladores heterogêneos distribuídos. Desta forma, todo o simulador que reconheça este padrão pode ser adicionado ao sistema de forma transparente.

- **Compromisso entre desempenho e precisão**:

A maioria das abordagens que trata este item permite que seja possível escolher o nível de abstração em que cada módulo é simulado. Desta forma, os módulos que podem ser simulados com menor número de detalhes contribuem com o desempenho do sistema uma vez que diminui a quantidade de comunicação [46]. Porém, em [40] é apresentada uma abordagem diferenciada, chamada técnica de prototipação gradual, que tem como objetivo diminuir o tempo total de simulação (desempenho) prototipando em FPGA componentes de hardware já validados e sintetizados. A cada módulo de hardware prototipado, menor fica o sistema que está sendo simulado, diminuindo a complexidade e o tempo de simulação. Este processo de validação por prototipação gradual continua até que todos módulos de hardware tenham sido prototipados. Desta forma, ao fim da validação do sistema, os componentes de hardware já estão prototipados. A co-simulação com FPGA obteve um ganho de desempenho de 1.7 sobre a co-simulação baseada

puramente em software [40].

5.3 Ambiente de Co-simulação Desenvolvido

Esta seção detalha a implementação e funcionamento do ambiente de co-simulação desenvolvido. Esta, portanto, é a primeira contribuição deste trabalho [6][7].

Um ambiente de co-simulação distribuído é apresentado, de forma geral, como a estrutura mostrada na Figura 30(a). Nesta figura observa-se um barramento no qual ocorre a comunicação entre os módulos. Para simular a existência de um barramento utilizamos primitivas de *sockets* para comunicação entre os módulos e um roteador de mensagens. Este roteador recebe as mensagens enviadas pelos módulos e as envia para todos os módulos que devam tomar conhecimento de tal mensagem. Esta organização está descrita na Figura 30(b).

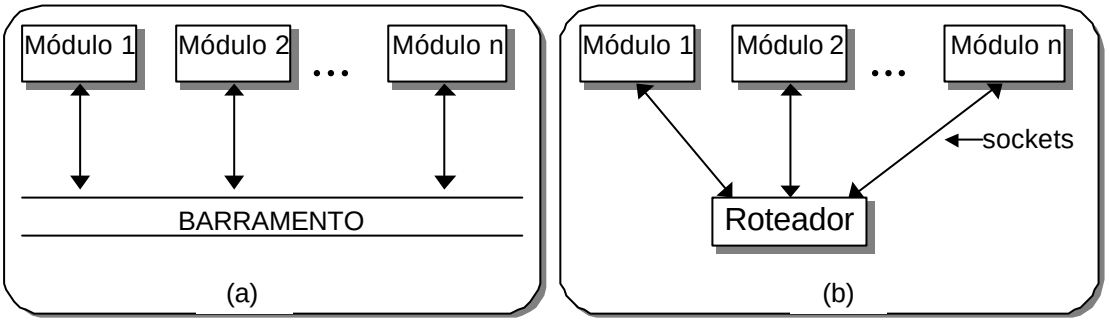


Figura 30 – Arquitetura do ambiente de co-simulação

Para comunicar-se com o ambiente de co-simulação, cada módulo deve utilizar uma biblioteca especificamente desenvolvida para seu simulador. Estas bibliotecas são partes integrantes do ambiente de co-simulação. Módulos escritos em VHDL utilizam entidades para envio e recebimento de dados. Descrições das bibliotecas de comunicação são encontradas na Seção 5.3.2.

A linguagem VHDL não possui interface com *sockets.* No entanto, alguns simuladores permitem que se defina o comportamento de um módulo VHDL em código C, utilizando uma biblioteca. O simulador ModelSim possui uma biblioteca chamada FLI (*Foreign Language Interface*)[53]. Desta forma é possível utilizar a biblioteca de *sockets* do sistema operacional para realizar a comunicação com o ambiente de co-simulação.

5.3.1 Estrutura do Roteador

A função principal do roteador é coordenar a comunicação entre os diversos módulos. Para que isso seja possível, o roteador necessita conhecer as definições dos módulos e suas interconexões. Para este fim, é fornecido um arquivo de coordenação, que é uma *netlist* contendo a conexão entre os módulos que se deseja co-simular. Um exemplo deste arquivo é apresentado na Figura 31.

Na primeira parte deste arquivo é apresentada a definição dos módulos que compõem a co-simulação. Neste exemplo, a co-simulação é realizada entre dois módulos, chamados *VHDL* e *M2*. O módulo *VHDL* é especificado na linguagem VHDL e simulado na ferramenta ModelSim. Este módulo é executado na máquina especificada pela diretiva *machine*, e possui as portas *a*, *b*, e *resultado* do tipo inteiro e *comeca* e *pronto* do tipo de controle. O módulo *M2* é descrito em linguagem C, tendo as portas necessárias para a comunicação com o módulo *VHDL*. Este arquivo de coordenação é utilizado no estudo de caso da Seção 5.4.

A seguir temos as definições de interconexões entre os módulos, representados pela diretiva *net*. Cada interconexão tem um nome (no exemplo acima, N1 a N5). Entre chaves são especificadas

as portas que fazem parte da interconexão, no formato *porta*(*módulo*).

```
1.  module VHDL {
2.      language("VHDL");
3.      command("vsim cossim_tb tb_architecture -do script.do");
4.      machine(name="hios.inf.pucrs.br",alias="hios");
5.      port {
6.          int A : in;
7.          int B : in;
8.          control comeca : in;
9.          control pronto: out;
10.         int resultado : out;
11.     }
12. }
13.
14. module M2 {
15.     language("C");
16.     command("M2");
17.     machine(name="hios.inf.pucrs.br",alias="hios");
18.     port {
19.         int A : out;
20.         int B : out;
21.         control comeca : out;
22.         control pronto: in;
23.         int resultado : in;
24.     }
25. }
26.
27. net N1 { A(VHDL),A(M2) }
28. net N2 { B(VHDL),B(M2) }
29. net N3 { comeca(VHDL),comeca(M2) }
30. net N4 { pronto (M2),pronto(VHDL) }
31. net N5 { resultado(VHDL),resultado(M2) }
```

Figura 31 – Arquivo de coordenação do multiplicador.

A partir do arquivo de coordenação o roteador monta suas estruturas internas de controle da co-simulação, tornando possível o roteamento de mensagens entre os diversos módulos participantes da co-simulação.

A co-simulação é interrompida no momento em que algum dos módulos se desconecta do sistema. Entretanto, o sistema garante que todas as mensagens já enviadas serão recebidas pelos módulos que ainda estiverem conectados.

O roteador foi desenvolvido em C++. Das classes que compõem o sistema, as seguintes são as mais importantes:

- CCoSim: É a interface para todo o sistema. Suas principais funções são: carregar o arquivo de coordenação, montando as estruturas internas do roteador; receber as conexões de todos os clientes, fazendo a autenticação dos mesmos; e rotear as mensagens que chegam para os módulos que tem portas conectadas à mesma rede da porta do módulo que originou a mensagem.
- CPort: Representa uma porta de um módulo. Tem como atributos o tipo da porta e sua direção (que pode ser de entrada, saída ou entrada e saída).
- CNet: Representa uma conexão entre as portas dos módulos.
- CModule: Representa um módulo, contendo informações sobre suas portas e o *socket* que está sendo usado para se comunicar com o simulador do módulo.

Para a execução do ambiente é necessário que cada módulo saiba em qual endereço (IP + porta) se localiza o roteador. No momento da conexão o roteador verifica se o módulo faz parte da co-simulação através das definições fornecidas pelo arquivo de coordenação. Isto torna necessário que o módulo tenha conhecimento do seu nome para conseguir se autenticar junto ao roteador.

A Figura 32 mostra o diagrama de classes do ambiente. A classe principal do roteador é

CCoSim, através dela é que são realizadas as operações de leitura do arquivo de coordenação, geração da estrutura de conexão dos módulos e o roteamento das mensagens. Como pode ser visto no diagrama, a classe tem conexão com módulos (CModule) e redes (CNet). Desta forma pode-se saber quais módulos fazem parte da co-simulação, e quais são as interconexões entre estes módulos. As redes sabem quais portas (Cport) estão ligadas a ela, podendo saber qual módulo está ligado àquela porta.

No momento em que o roteador recebe uma mensagem, ele verifica em qual *net* a porta originadora está conectada, e procede enviando mensagens para todos os módulos que têm portas conectadas a esta *net*, através do método SendMsg da classe CModule. A classe CModule guarda informação sobre qual *socket* mantem a conexão com o simulador do módulo correspondente, podendo assim enviar a mensagem ao mesmo.

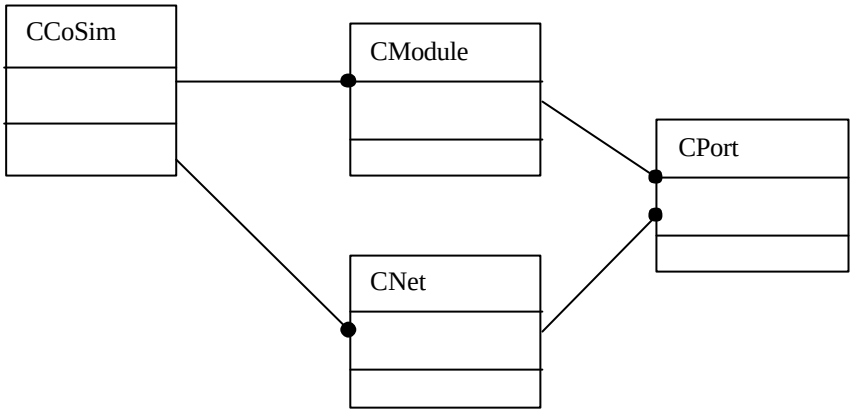


Figura 32 – Diagrama de classes simplificado do roteador.

5.3.2 Bibliotecas de Comunicação

A biblioteca de comunicação tem a função de integrar um simulador ao ambiente de co-simulação. Esta biblioteca consiste basicamente de cinco funções: inicialização (csiInitialize( )), envio (csiSend( )) e recebimento de dados (csiReceive( )) e envio (csiSendControl())e recebimento (csiWaitControl()) de sinais de controle. Atualmente as mensagens trafegam apenas com valores inteiros. Futuramente, utilizaremos um tipo unificado que possa representar qualquer tipo de dado que seja necessário.

A Figura 33 apresenta os módulos principais que compõem o ambiente de co-simulação, incluindo o projeto do usuário. Os módulos em cor clara representam partes do ambiente de co-simulação, os dois módulos de cor escura compõem o projeto do usuário. Para utilizar o ambiente de co-simulação o usuário deve incluir em cada módulo a ser simulado a biblioteca de comunicação com o ambiente de co-simulação. Na Figura 33 estas bibliotecas são designadas *ComLib*. Cada simulador deve possuir a sua biblioteca de comunicação, pois geralmente a interface do simulador com o mundo externo é proprietária. Atualmente o sistema possui bibliotecas que suportam C e VHDL. Bibliotecas para a linguagem Java e SDL estão sendo desenvolvidas.

Na parte de software do projeto usuário, o usuário deve inserir manualmente o código incluindo a primitiva de inicialização e as primitivas de envio e recebimento de dados. Já na parte de hardware, o usuário deve mudar, também manualmente, a descrição do seu projeto de forma a adicionar um componente VHDL que instancia o módulo do usuário e os componentes (ComLib VHDL) que possibilitam a troca de mensagens entre as portas do hardware e do software.

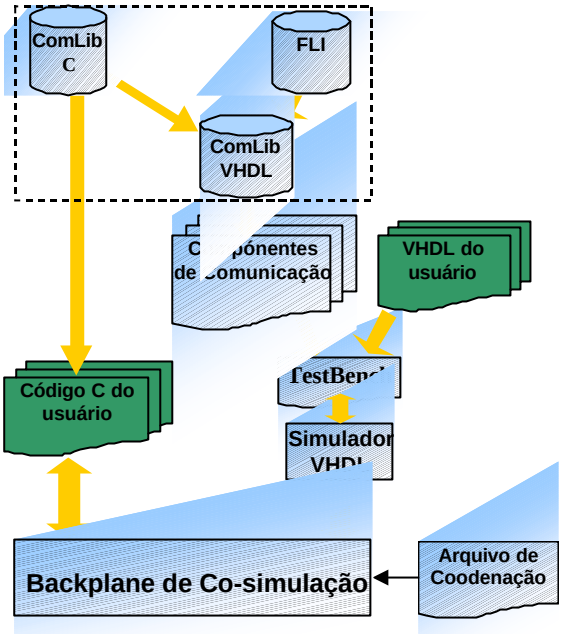


Figura 33 – Estrutura geral do sistema de co-simulação em relação a bibliotecas utilizadas.

Detalhando mais o funcionamento da primitiva de inicialização, o usuário deve incluir, no início do código, a função csiInitialize ( ). Esta função é responsável pela conexão do simulador no ambiente. Os parâmetros passados na sua execução são o nome do módulo do usuário e o endereço IP do roteador de mensagens do ambiente de co-simulação. Cada simulador deve informar o seu nome (nome do módulo do usuário) para o roteador, pois este deve relacionar o nome do módulo à conexão do *socket* e verificar se o nome do simulador consta no arquivo de coordenação.

Uma funcionalidade atribuída às bibliotecas de comunicação é a conversão de dados. Quatro alternativas foram analisadas em relação ao local onde esta conversão deve ser executada: Conversão no módulo de origem da mensagem, no roteador, nos módulos de destino e conversão tanto nos módulos de origem quanto nos de destino. (*i*) Conversão de dados na origem não é possível pois uma mesma mensagem pode ser enviada para vários módulos que podem representar diferentes simuladores e linguagens. Isto impossibilita que a conversão de dados ocorra no envio da mensagem, uma vez que os destinatários da mesma podem ser representados por tipos de dados diferentes; (*ii*) Conversão de dados no roteador também não é uma abordagem adequada pois o roteador deve conhecer todos os tipos de dados de todos os simuladores do sistema. Assim, sempre que um novo simulador for adicionado, o roteador deve ser recompilado de forma a incluir a biblioteca de conversões de dados deste novo simulador; (*iii*) Conversão de dados no destino da mensagem recai no mesmo problema do item anterior, pois se um módulo receber dados em um formato não suportado, a biblioteca de comunicação tem que ser alterada e recompilada para suportar este tipo de dado; (*iv*) Conversão de dados na origem e no destino implica na adoção de um tipo de dado único e compatível a todos os módulos envolvidos na co-simulação. Para este fim o tipo inteiro foi adotado. Isto implica que quando o módulo de origem envia um dado, o mesmo deve ser convertido para inteiro. Já o destinatário vai converter o dado de inteiro para o tipo de dado adequado. Esta última opção de conversão, a qual foi implementada, torna o ambiente escalável e sem necessidade de recompilação.

5.4 Estudos de Caso

Esta Seção apresenta três estudos de caso. O primeiro visa validar o ambiente de co-simulação, o segundo avaliar o seu desempenho e o terceiro demonstrar sua aplicação a este trabalho.

5.4.1 Validação do Ambiente

Utilizamos como estudo de caso uma aplicação simples de multiplicação por somas sucessivas particionada manualmente em software (C) e em hardware (VHDL) [7]. O módulo C gera operandos e recebe os resultados das multiplicações. O módulo hardware executa a multiplicação propriamente dita. É importante destacar que o objetivo desta aplicação é validar a metodologia de co-simulação, e não de validar um sistema complexo. Uma avaliação de desempenho do ambiente de co-simulação é apresentada na Seção 5.4.2 utilizando um segundo estudo de caso.

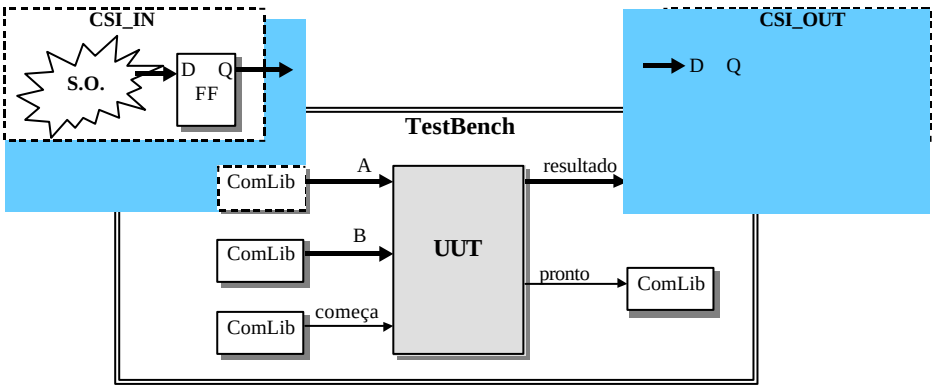
Do ponto de vista do usuário, é necessário acrescentar as bibliotecas de comunicação nos módulos que participam da co-simulação. Em relação ao módulo de hardware, deve ser construído um *testbench* que instancia o módulo VHDL (UUT) e um componente de comunicação para cada um dos sinais (operado A, operando B, começa, pronto e resultado) que se comunicam com o software. A Figura 35 ilustra esta estrutura. Existem os componentes de comunicação de entrada (CSI_IN) e os de saída (CSI_OUT). Apesar de quatro componentes de comunicação serem instanciados, o sistema está escrito de maneira que apenas um canal de comunicação (*socket*) é aberto para se comunicar com o ambiente de co-simulação. As portas do componente de comunicação de VHDL são apresentadas na Figura 34. A porta *porta* é uma constante do tipo *string* que define qual porta do VHDL (a, b, resultado, pronto ou começa) que esta instância da biblioteca monitora. Os componentes de entrada possuem uma porta de saída chamada *dout*. Já os componentes de saída possuem uma porta de entrada chamada *din*.

```
1.  component CSI_IN
2.  port(
3.    porta : in  string;
4.    dout  : out std_logic_vector(31 downto 0);
5.    clk   : in  std_logic);
6.  end component;
```

```
1.  component CSI_OUT
2.  port(
3.    porta : in  string;
4.    din   : in  std_logic_vector(31 downto 0);
5.    clk   : in  std_logic);
6.  end component;
```

Figura 34 – Entidade do componente de comunicação de VHDL.

O *testbench* apresentado na Figura 35 e Figura 36 possui uma estrutura regular, ou seja, um *testbench* formado por um simples conjunto de instâncias. Isto facilita a sua criação que ainda é feita manualmente, mas que no futuro deve ser criado automaticamente. Basicamente, deve ser instanciada a UUT e um componente de comunicação para cada porta da UUT.



S.O. – sistema operacional.

Figura 35 – Representação do *testbench* do multiplicador.

```
1.  entity cossim_tb is
2.  end cossim_tb;
3.
4.  architecture tb_architecture of cossim_tb is
5.    component CSI_IN
6.    end component;
7.
8.    component CSI_OUT
9.    end component;
10.
11.   component multi
12.   end component;
13. begin
14.   UUT : multi
15.         port map(...);
16.
17.   COMLIB1 : CSI_IN
18.         port map (dout => a);
19.   COMLIB2 : CSI_IN
20.         port map (dout => b);
21.   COMLIB3 : CSI_IN
22.         port map (dout => comeca);
23.   COMLIB4 : CSI_OUT
24.         port map (din => resultado);
25.   COMLIB5 : CSI_OUT
26.         port map (din => pronto);
27.
28.   clock and reset generation
29. end tb_architecture;
```

As linhas 5-9 declaram os componentes de comunicação, respectivamente o de entrada e o de saída de dados. O componente que é co-simulado é declarado e instanciado nas linhas 11 e 14. Nas linhas 17-26 são instanciados um componente de comunicação para cada porta do UUT. Finalmente, a linha 28 representa a geração dos sinais *clock* e *reset*.

Figura 36 – Estrutura do *testbench* do multiplicador.

Na Figura 37 apresentamos o código fonte da parte de software deste estudo de caso. Dentre as modificações necessárias estão a inclusão da biblioteca de comunicação da linguagem C (linha 1), a inicialização da comunicação (linha 9) e as chamadas de envio e recepção de dados (linhas 12-16).

```
1.  #include "csi.h"
2.  #include <stdio.h>
3.
4.  void main(int argc, char *argv[])
5.  {
6.    int x=2,result;
7.    int i;
8.
9.    csiInitialize();
10.
11.   for(i=0;i<10;i++,x++) {
12.         csiSend("B",x);
13.         csiSend("A",x);
14.         csiSendControl("comeca");
15.         csiWaitControl("pronto");
16.         csiReceive("resultado",&result);
17.         printf("%d x %d = %d\n",x,x,result);
18.   }
19.   getchar();
20. }
```

Figura 37 – Código fonte do software de multiplicador.

Alterado o projeto do usuário, deve ser criado o arquivo de coordenação que conecta os módulos do usuário ao ambiente. Este arquivo pode ser encontrado na Figura 31. Feito isto, o ambiente pode ser inicializado. Primeiro se inicializa o roteador, que começa aguardando a conexão

de todos os módulos indicados no arquivo de coordenação para iniciar a etapa de roteamento. Após, todos os simuladores que compõem o ambiente devem ser executados. Ao se conectarem, o roteador está pronto para rotear as mensagens para começar a simulação.

A Figura 38 apresenta a captura de tela da co-simulação do multiplicador. Existem quatro janelas nesta figura. A janela 1 é o console do simulador VHDL. Na área destacada desta janela apresentamos o momento em que este simulador recebe os três dados (a, b e começa nesta ordem) e envia o resultado da multiplicação (196). A janela 2 representa a forma de onda da descrição de hardware. No momento destacado, o módulo de hardware sinaliza na porta *pronto* o momento em que a multiplicação está realizada. O resultado da multiplicação está na porta *resultado*. Outros sinais desta descrição de hardware também são apresentados. A janela 3 representa o módulo de software. Cada linha impressa é o resultado de uma multiplicação. Está destacado nesta janela o momento em que ocorre a multiplicação 14 x 14. Por fim, a janela 4 representa o roteador. Destacamos nesta janela o momento em que ocorre a mesma multiplicação (14 x 14). Note que é impresso o momento em que o roteador recebe o dado 14 da porta *a* do módulo M2 (módulo de software). Também é apresentado o momento em que o módulo chamado VHDL envia ao roteador, pela porta *resultado*, o valor 196.

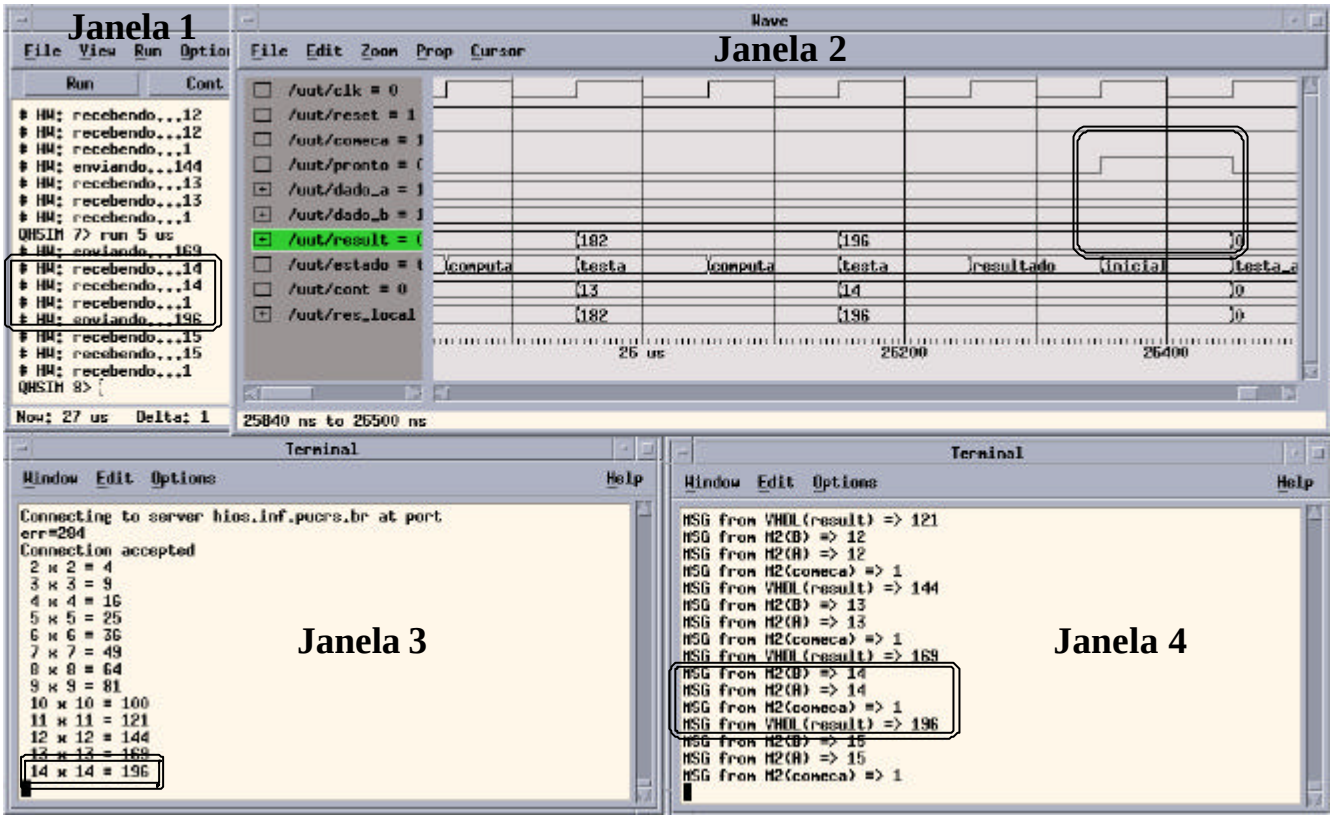


Figura 38 – Captura de tela da co-simulação hardware/software.

5.4.2 Avaliação de Desempenho do Ambiente de Co-simulação

Um segundo estudo de caso foi desenvolvido para avaliar o desempenho do ambiente de co-simulação [6]. O estudo de caso é um algoritmo de preenchimento de polígonos. A Figura 39 ilustra o seu funcionamento. Dado um polígono convexo, este algoritmo encontra todas as linhas horizontais, por exemplo *y4*, que preenchem este polígono. Existe uma tabela de preenchimento que armazena as coordenadas *x* que representam cada linha horizontal.

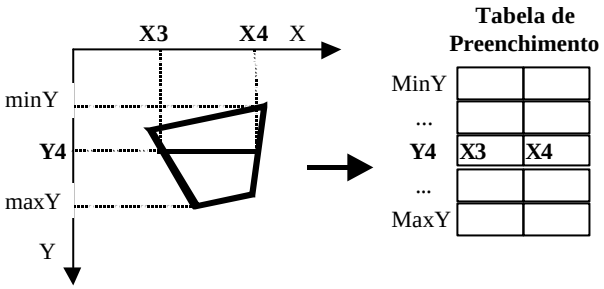
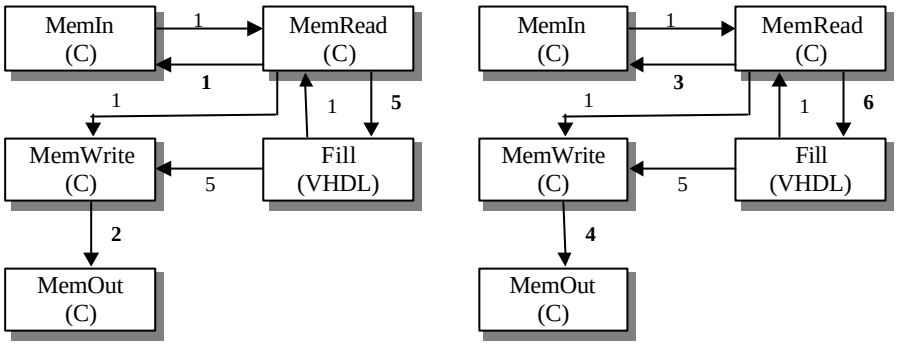


Figura 39 – Algoritmo de preenchimento de polígonos.

Existem dois particionamentos deste estudo de caso. Ambos possuem 5 módulos, conforme a Figura 40. Quatro módulos estão implementados em software e um em hardware. Estes particionamentos foram escolhidos para explorar um maior grau de comunicação entre os módulos. A Figura 40 também apresenta a direção e o número de canais dos dois particionamentos.



(a) particionamento arquitetural (b) particionamento funcional

Figura 40 – Particionamentos do estudo de caso de preenchimento de polígonos.

A principal diferença entre as partições é o número de canais. Esta diferença representa diferentes níveis de abstração empregados para descrever o sistema. A primeira partição, chamada "C + VHDL" possui 20 canais. Esta partição possui canais de dados e controle, representando uma descrição arquitetural do sistema. Por outro lado, a partição chamada "C + VHDL otim" possui 15 canais de dados, sem canais de controle. Esta partição representa uma descrição funcional do sistema.

Três experimentos foram realizados para avaliar o ambiente de co-simulação. Em todos os casos diferentes números de polígonos foram executados para averiguar o tempo de validação com o aumento da complexidade do projeto.

As seguintes estações de trabalho foram utilizadas nestes experimentos: (1) WS1: Sun Ultra10, 333 MHz, 256 MB RAM; (2) WS2: Sun Ultra1, 167 MHz, 192 MB RAM; (3) WS3: Sun SPARC 4, 110 MHz, 72 MB RAM; (4) WS4: Sun SPARC 4, 110 MHz, 72 MB RAM; (5) WS5: Sun SPARC 4, 110 MHz, 64 MB RAM; (6) WS6: Sun Ultra2, 296 MHz, 256 MB RAM. As estações WS1 até WS5 estão em uma LAN Ethernet 10Mbps. Já a estação WS6 está em uma WAN a uma distância de 5 *hops*.

O primeiro experimento, Tabela 5, compara os tempos de execução de diferentes abordagens de validação. A primeira abordagem analisada é a implementação em software do estudo de caso. Como o esperado, seu tempo de execução é muito menor que as outras abordagens, uma vez que se trata de código compilado e sem tráfego de mensagens. Na segunda abordagem o estudo de caso foi implementado em VHDL e simulado. Como é código interpretado o tempo de execução é maior que a primeira abordagem. Porém, este tempo de execução não é excessivamente maior porque não possui troca de mensagens. A terceira abordagem representa o tempo de validação obtido na ferramenta desenvolvida. Nota-se que o tempo de execução é muito maior que as outras abordagens devido principalmente às milhares de trocas de mensagens e chamadas do sistema operacional.

Além disto, esta avaliação foi feita com o ambiente utilizando socket TCP para comunicação entre processos. Este mecanismo foi utilizado para facilitar a construção do ambiente, pois o mesmo possui controle de erros. Porém, sabe-se que este mecanismo é custoso em termos de tempo de conexão e transmissão. As próximas versões da ferramenta visam suportar algum outro mecanismo de comunicação mais rápido.

Tabela 5 – Tempo de execução, em segundos, de diferentes mecanismos de validação.

# Polígnos	Compilado	Simulado	Co-simulado
50	0.002	4.4	34.8
100	0.003	4.9	69.4
500	0.017	9.4	352.8
1000	0.033	15.4	703.3
3000	0.100	40.0	2109.2

O segundo experimento, Tabela 6, avalia a variação de tempo de co-simulação das duas partições implementadas, “C+VHDL” e “C+VHDL (otim)”. Também é apresentado o número de trocas de mensagens executadas durante a co-simulação. Nota-se que com a remoção de 5 canais da versão otimizada obtivemos um ganho de desempenho de cerca de três vezes. Estes dados demonstram que o tempo de co-simulação varia com a quantidade de comunicação entre os módulos.

Tabela 6 – Tempos de co-simulação, em segundos, das duas partições.

# Polígonos	C + VHDL		C + VHDL (otim)	
	tempo	# msgs	Tempo	# msgs
50	113.7	90495	34.8	56342
100	224.2	183527	69.4	114407
500	1120.9	924745	352.8	576312
1000	2160.3	1840089	703.3	1250308
3000	9608.2	5522387	2109.2	3755731

O terceiro experimento, Tabela 7, avalia o impacto da comunicação entre redes no tempo de co-simulação. Três abordagens são comparadas: Na primeira abordagem todos os módulos do estudo de caso são executados em uma única estação de trabalho; Na segunda abordagem estes módulos estão distribuídos em estações de trabalho diferentes conectados por uma LAN; Na última abordagem quatro módulos estão em uma LAN e um está em uma WAN a 5 *hops* de distância. Observa-se que há um pequeno impacto no tempo de co-simulação, pois o tempo em uma única estação é sempre menor do que nas outras abordagens. Porém, o tempo adicional imposto pela distância da WAN é percebido somente quando se aumenta a complexidade do sistema (> 1000 polígonos). Estes dados demonstram que é possível ter módulos distribuídos em uma rede com pequeno impacto, corroborando as vantagens da co-simulação geograficamente distribuída.

Tabela 7 – Tempo de co-simulação (s) em uma máquina, em uma LAN e em uma WAN.

# Polígonos	Único computador	Distribuídos em LAN	Distribuídos em LAN + WAN
50	34.8	41.0	36.3
100	69.4	76.4	72.5
500	352.8	423.1	368.1
1000	703.3	725.5	751.9
3000	2109.2	2177.4	2238.8

5.4.3 Co-simulação para Validação de Lógica Envoltória

A motivação para o desenvolvimento desta ferramenta de co-simulação foi a ausência de um ambiente que suportasse validação integrada entre módulos de hardware e software. Desta forma, o objetivo deste estudo de caso é apresentar uma aplicação desta ferramenta no contexto de teste de núcleos utilizando teste baseado em software.

Este estudo de caso visa a validação funcional de uma lógica envoltória e do software de geração de estímulos e compactação de respostas. Neste exemplo existe um módulo de hardware, que é a lógica envoltória, e um módulo de software, que implementa o LFSR e o MISR. O arquivo de coordenação deste projeto é apresentado na Figura 41.

```
1.  module tb_wr_c880 {
2.    language("VHDL");
3.    command("vsim tb_wr_c880 TB_ARCHITECTURE");
4.    machine(name="hios.inf.pucrs.br",alias="hios");
5.    port {
6.         int q : out;
7.         int d : in;
8.         int addr : in;
9.         control ready: out;
10.        control cs: in;
11.        control wr: in;
12.        control rd: in;
13.   }
14. }
15. module lfsr32 {
16.   language("C");
17.   command("lfsr32");
18.   machine(name="hios.inf.pucrs.br",alias="hios");
19.   port {
20.        int q: in;
21.        int d: out;
22.        int addr: out;
23.        control ready: out;
24.        control cs: out;
25.        control wr: out;
26.        control rd: out;
27.   }
28. }
29.
30. net N1 { q(lfsr32), q(tb_wr_c880) }
31. net N2 { d(tb_wr_c880), d(lfsr32) }
32. net N3 { addr(lfsr32), addr(tb_wr_c880) }
33. net N4 { ready(tb_wr_c880), ready(lfsr32) }
34. net N5 { cs(lfsr32), cs(tb_wr_c880) }
35. net N6 { wr(lfsr32), wr(tb_wr_c880) }
36. net N7 { rd(lfsr32), rd(tb_wr_c880) }
```

Figura 41 – Arquivo de coordenação do projeto da lógica envoltória.

A lógica envoltória, chamada tb_wr_c880, possui a porta *q* de saída de dados, *d* de entrada de dados, *addr* de endereço, as portas *cs*, *wr* e *rd* para controle de escrita e leitura e a porta *ready* para sincronização da co-simulação. O software, chamado lfsr32, possui interface equivalente.

A Figura 42 apresenta de forma simplificada um código de teste do núcleo c880 pronto para co-simulação. Além das modificações básicas para co-simulação com a inserção da biblioteca e a função de inicialização, este programa possui uma constante (POLY) que define o polinômio utilizado para gerar padrões e compactar respostas (linha 4). Logo após segue a descrição do LFSR e do MISR (linhas 5 e 6). Este programa gera padrões através de um LFSR (linha 14) e os envia para o núcleo sendo testado. Este núcleo processa o padrão e gera uma saída que é lida e compactada pelo software (linha 28).

```
1.  #include "csi.h"
2.
3.  //x^32 + x^7 + x^5 + x^3 + x^2 + x + 1
4.  #define POLY 0x0AE
5.  void lfsr(){...}
6.  void misr(){...}
7.
8.  int main(void)
9.  { csiInitialize();
10.
11.   signature=-1;
12.   data=-1;
13.   // envia dados
14.   lfsr(POLY,&data);
15.   csiSend("d",data);
16.   // sinaliza a escrita
17.   csiSendControl("cs");
18.   csiSendControl("wr");
19.   csiWaitControl("ready");
20.
21.   // sinaliza leitura
22.   csiSendControl("cs");
23.   csiSendControl("rd");
24.   csiWaitControl("ready");
25.   // recebe dados
26.   csiReceive("q", &response);
27.   // compacta resposta
28.   misr(POLY,response,&signature);
29.   printf("%8x %8x\n",response, signature);
30. }
```

Detalhes da implementação das funções misr e lfsr serão apresentados na Seção 7.1

Figura 42 – Estrutura do código de teste do núcleo c880 e da lógica envoltória.

A Figura 43 apresenta a estrutura do *testbench* criado para co-simulação. A lógica envoltória (linha 13) e os componentes de comunicação (linha 16-29) são instanciados . Cada componente de comunicação conecta uma porta da lógica envoltória ao roteador. Estes componentes de comunicação podem ser de entrada (CSI_IN) ou de saída (CSI_OUT), dependendo da direção da porta em questão da lógica envoltória. Ao final do testbench são descritos os sinais de clock e reset.

```
1.  entity tb_wr_c880 is
2.  end tb_wr_c880;
3.
4.  architecture TB_ARCHITECTURE of tb_wr_c880 is
5.    component CSI_IN
6.    end component;
7.    component CSI_OUT
8.    end component;
9.
10.   component wr_c880
11.   end component;
12. begin
13.   UUT : wr_c880
14.         port map(...);
15.
16.   COMLIB1 : CSI_OUT
17.         port map (din => q);
18.   COMLIB2 : CSI_OUT
19.         port map (din => ready);
20.   COMLIB3 : CSI_IN
21.         port map (dout => d);
22.   COMLIB4 : CSI_IN
23.         port map (dout => addr);
24.   COMLIB5 : CSI_IN
25.         port map (dout => wr);
26.   COMLIB6 : CSI_IN
27.         port map (dout => rd);
28.   COMLIB7 : CSI_IN
29.         port map (dout => cs);
30.
31.   geração de clock e reset
32. end TB_ARCHITECTURE;
```

Figura 43 – Estrutura do *testbench* da lógica envoltória.

A Figura 44 representa uma captura de tela da co-simulação da do sistema de teste particionado em hardware e software. Existem três telas nesta figura. Uma representa o simulador de hardware, a segunda representa a parte de software destacando o fim da execução e a assinatura gerada, e a última representa uma forma de onda da lógica envoltória. Esta forma de onda destaca as operações de escrita e leitura na/da lógica envoltória. Existem duas escritas pois o núcleo em questão possui mais de 32 bits de entrada. Pode-se perceber que no momento em que ocorre uma escrita na lógica envoltória, os sinais *cs* e *wr* estão ativados (nível lógico 1). Na leitura os sinais *cs* e *rd* são ativados.

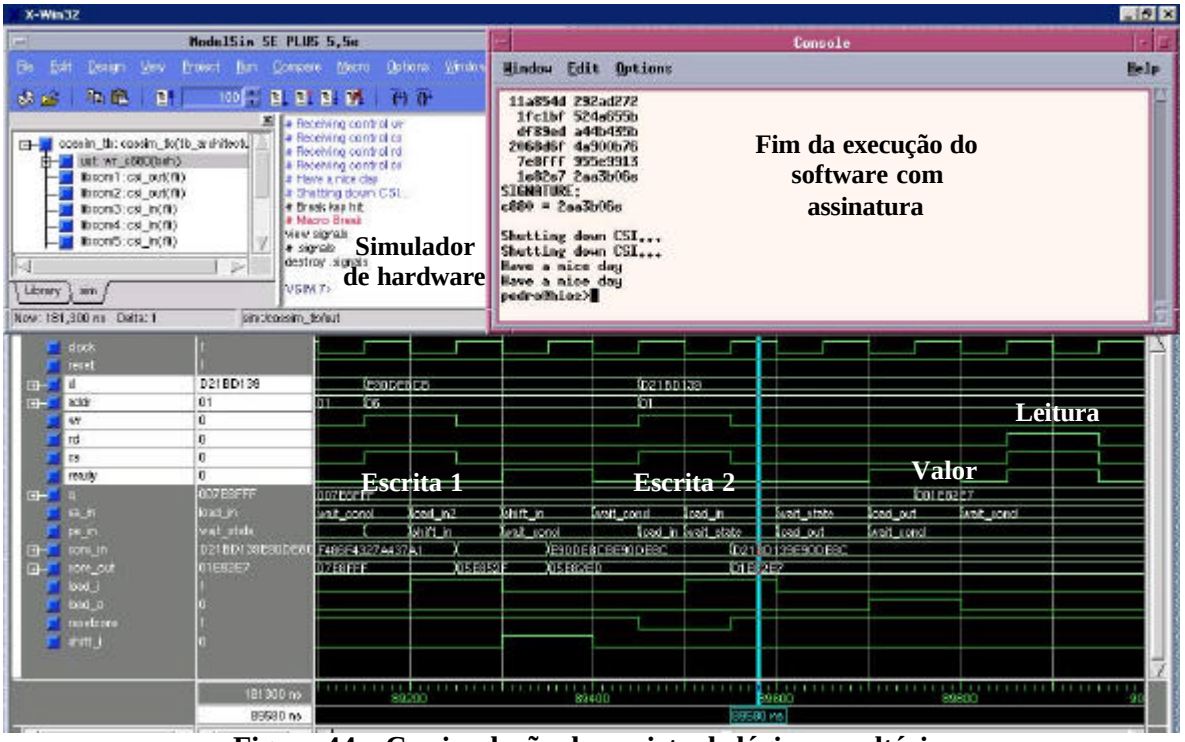


Figura 44 – Co-simulação do projeto de lógica envoltória.

5.5 Conclusão

Este Capítulo apresentou o desenvolvimento de um ambiente de co-simulação [6][7], tendo como objetivo a validação da metodologia de teste de hardware baseado em software. Esta validação é funcional, não informando o tempo necessário ao teste, apenas se a metodologia desenvolvida está funcionalmente correta.

A principal contribuição deste ambiente é a integração de diversos simuladores. As principais características deste ambiente são: (*i*) lançamento automático dos simuladores; (*ii*) suporta co-simulação geograficamente distribuída; (*iii*) suporta VHDL e C (inclusive suas variações como C++ e SystemC); (*iv*) suporta co-simulação de modelos arquiteturais e RTL; (*v*) flexibilidade do roteador de co-simulação. É importante ressaltar que a ferramenta de co-simulação, o roteador, é independente dos simuladores utilizados, pois o tipo de dados que trafega pela rede é único. Desta forma, novas linguagens e simuladores podem ser facilmente suportados.

Por outro lado, este ambiente ainda apresenta desvantagens, as quais devem ser investigadas em trabalho futuro. Dentre as desvantagens citamos: (*i*) baixa portabilidade, devido ao fato que cada nova linguagem ou simulador adicionado ao sistema deva possuir uma biblioteca de comunicação para troca de mensagens com o roteador; (*ii*) alto tempo de co-simulação, devido ao grande número de trocas de mensagens e chamadas ao sistema operacional.

Novas bibliotecas de comunicação estão sendo incorporadas ao sistema para dar suporte a SDL (*Synchronous Description Language*) e Java. Como trabalhos futuros visamos principalmente diminuir o tempo de co-simulação. Atualmente, o meio de comunicação utilizado é *socket* TCP, inclusive quando a co-simulação ocorre em um único computador. Nestes casos pode-se utilizar outro meio de comunicação como *pipes* ou memória compartilhada. Também é possível o desenvolvimento de um gerador automático de *testbench* para VHDL, dada a regularidade do mesmo, cujo objetivo é facilitar a utilização do ambiente.

6 AMBIENTE DE PROTOTIPAÇÃO

Neste Capítulo apresentamos o ambiente de prototipação que foi utilizado para desenvolver e validar a parte prática deste trabalho. O ambiente de prototipação Excalibur, comercializado pela empresa Altera, é destinado à prototipação em FPGA de sistemas digitais baseados em módulos de hardware e software. O ambiente Excalibur possui um conjunto de ferramentas para depuração e configuração de hardware e ferramentas para compilação e depuração de software, destinadas a um processador *firm core* proprietário, denominado Nios [5]. Este ambiente foi escolhido por possuir uma série de facilidades para prototipação e validação de um sistema composto por núcleos. Dentre as características que tornam o ambiente Excalibur uma plataforma adequada à avaliação das metodologias de teste de SOC enumeramos:

- Ferramenta *SignalTap*, a qual possibilita capturar valores de sinais com o sistema em execução no FPGA. Quando se desejam avaliar sinais internos de um dado sistema com um analisador lógico, é necessário levar estes sinais para o nível mais alto de hierarquia, modificando-se assim o circuito em desenvolvimento. Já com a ferramenta *SignalTap* basta o projetista especificar os sinais que devem ser capturados e realizar a síntese do sistema. Os blocos responsáveis pela captura dos dados são inseridos automaticamente. Estes valores são enviados pela porta serial para um programa que mostra a variação dos valores da mesma forma que um analisador lógico o faria.
- Descrições de modelos no nível lógico de abstração do processador NIOS (EDIF sintetizado a partir do VHDL proprietário da Altera) e do barramento Avalon. Estas descrições permitem simular os núcleos em desenvolvimento e o processador no nível de ciclo de relógio, utilizando o simulador *ModelSim*. O ambiente de co-simulação apresentado no Capítulo 5 permite uma validação funcional das técnicas de teste propostas, enquanto que a simulação no nível de ciclos de relógio permite um avaliação precisa do tempo necessário para a execução das rotinas de teste.

O objetivo deste capítulo é apresentar alguns recursos do ambiente Excalibur. Desta forma, somente os pontos relevantes a este trabalho estão sendo destacados. Entre estes pontos citamos: a organização dos módulos de hardware, a estrutura do barramento e interface de hardware e software do sistema.

6.1 Organização dos Módulos de Hardware

A Figura 45 apresenta a organização dos módulos do SOC implementados em FPGA. Normalmente são instanciados o processador Nios [5], o barramento Avalon [4] e alguns módulos de hardware. Estes módulos de hardware podem ser instanciados dentro do "Módulo de Sistema Nios" (MSN) ou externamente ao MSN. Os módulos internos ao MSN são: o processador, o barramento e, geralmente, os módulos de hardware (*firm core*) responsáveis pela interface com algum recurso físico da placa (e.g. UART, LCD, memória, entre outros). Por outro lado, os módulos externos ao MSN são necessariamente LDU (lógica definida pelo usuário). Na Figura 45 temos o processador, o barramento e quatro módulos de hardware:

- O módulo I/O 1 está definido dentro do MSN. Ele não possui interface externa ao MSN nem ao FPGA. Um exemplo de módulo com esta característica é um temporizador;
- O módulo I/O 2 também está definido internamente ao MSN. Ele possui interface externa ao FPGA. Um exemplo de módulo com esta característica é uma UART, com acesso aos pinos de entrada e saída serial;
- Os módulos *user IF 1* e *IF 2* representam LDUs. O primeiro é somente uma interface de comunicação com um hardware externo ao FPGA e o segundo faz interface com módulos

internos ao FPGA. Uma aplicação para estes exemplos é uma interface com memória externa e uma LDU que acelera uma operação específica do software (IF2). Percebe-se que associado a uma LDU deve haver uma interface e que a LDU pode ter ou não comunicação externa ao FPGA. A seta tracejada ligada à LDU indica que a mesma pode possuir portas de comunicação com o mundo externo. Uma vez que este trabalho se restringe a trabalhar com SOCs dotados de processadores, o tipo de interface que nos interessa é o representado pela *user IF 2*, onde se utiliza a LDU internamente ao FPGA.

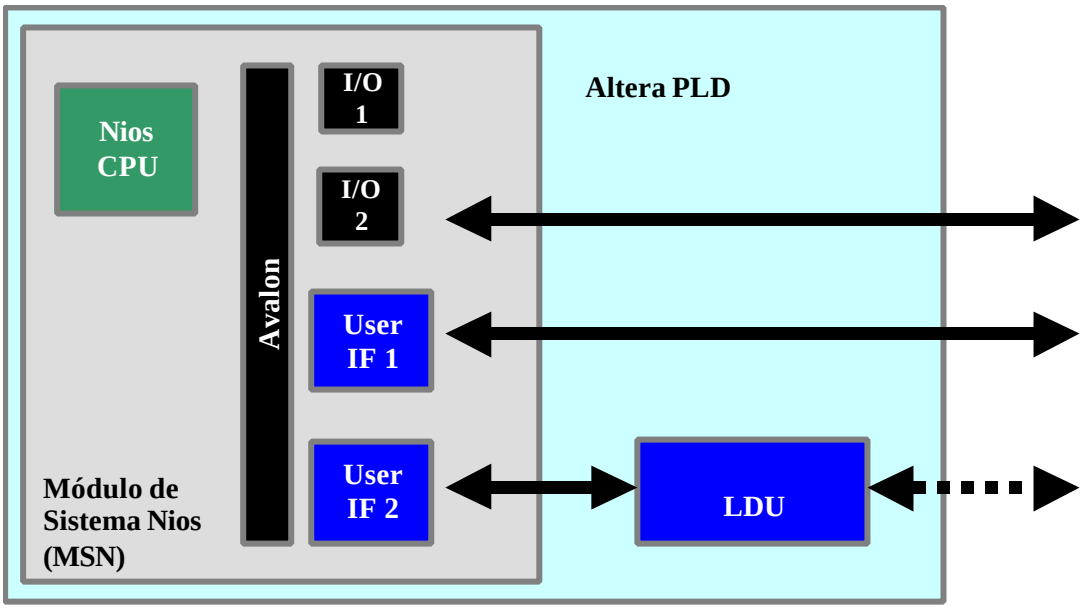
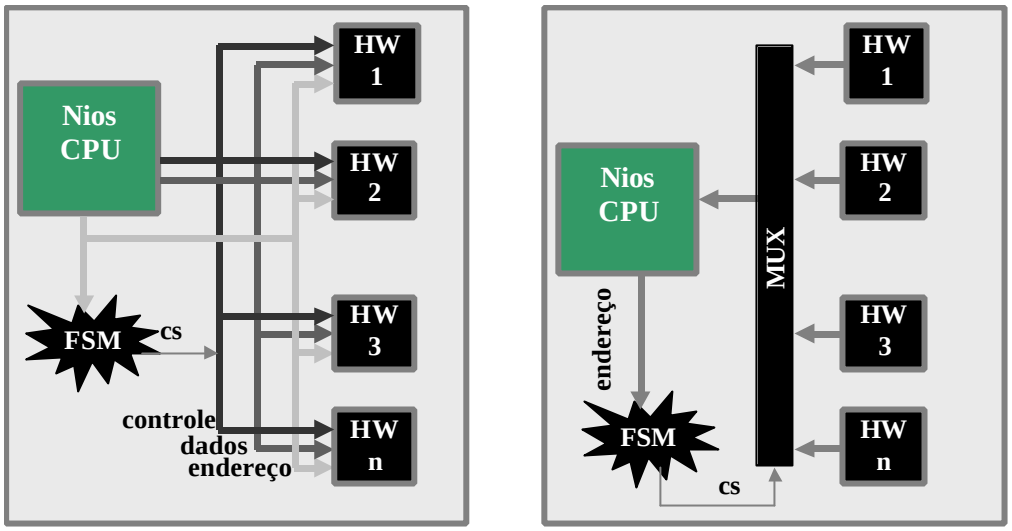


Figura 45 – Modelo de interface hardware/software no ambiente Excalibur.

6.2 Barramento

A comunicação no ambiente Excalibur entre o mestre (geralmente é o processador) e os escravos (módulos de hardware) é feita por entrada e saída mapeada em memória. Por exemplo, define-se para o módulo 1 a faixa de endereços 4 a 8. Assim, sempre que o processador executar uma leitura ou escrita nesta faixa de endereços, estará sendo feito um acesso de leitura ou escrita no módulo 1. Por este motivo que os sinais de interface entre processador e hardware são sinais padrão de comunicação processador/memória (e.g. habilitador de dispositivo, de escrita, de leitura e barramento de dados e de endereços) mais o sinal de interrupção.

Apresentamos na Figura 46 uma representação esquemática de como ocorre a leitura e escrita executada pelo processador no *modelo de comunicação* empregado entre o processador Nios e os módulos de hardware do sistema. Uma vez que o FPGA utilizado não suporta barramento *tri-state* interno, a forma encontrada para direcionar o dado lido de um hardware é por meio de multiplexador (Figura 46(b)). No processo de leitura de um módulo pelo processador, a lógica de decodificação de endereço ativa o sinal *cs* do módulo correspondente a este endereço, fazendo com que o dado do hardware seja transferido até o processador através do multiplexador. Já quando o processador escreve (Figura 46(a)) em um hardware, novamente existe uma lógica no barramento que verifica o endereço de escrita e aciona o sinal *cs* do hardware em questão. Desta forma, todos os módulos do sistema recebem os mesmos valores para suas portas de dados, endereço e sinais de controle como *read* e *write*. O que diferencia a que módulo o dado é endereçado é o sinal *cs* que é único a cada módulo.



(a) Escrita (b) Leitura

Figura 46 – Representação das operações de leitura e escrita realizadas pelo processador Nios.

A Figura 47 apresenta um exemplo de uma operação de escrita executada pelo processador Nios: (A) inicia a operação de escrita na borda de subida do *clock*; (B) sinais *writedata, address* e *writen* registrados; (C) barramento Avalon decodifica o endereço e seta *chipselect* para o módulo; (D) o módulo captura os sinais na borda de subida do *clock* e a operação de escrita termina.

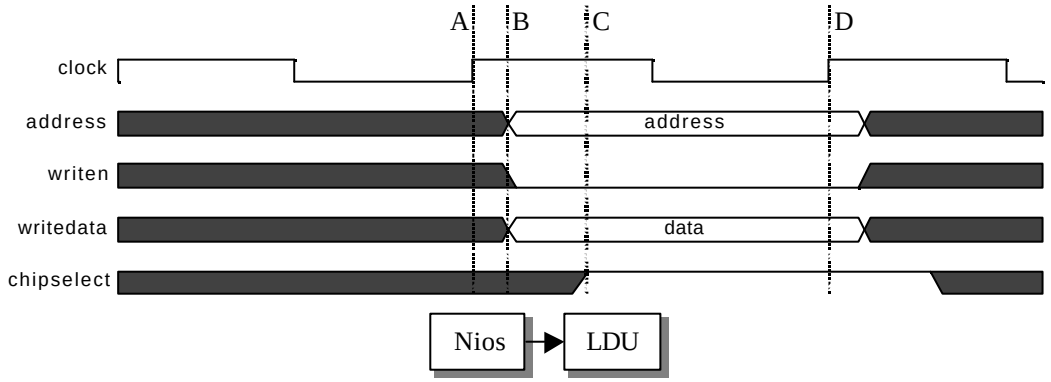


Figura 47 – Exemplo de uma operação de escrita do processador.

A Figura 48 apresenta um exemplo de uma operação de leitura executada pelo processador Nios: (A) inicia a operação de leitura na borda de subida do *clock*; (B) sinais *address* e *readn* registrados; (C) barramento Avalon decodifica o endereço e seta *chipselect* para o módulo; (D) o módulo responde com dado válido; (E) o barramento captura o sinal *readdata* na borda de subida do *clock* e a operação de leitura termina.

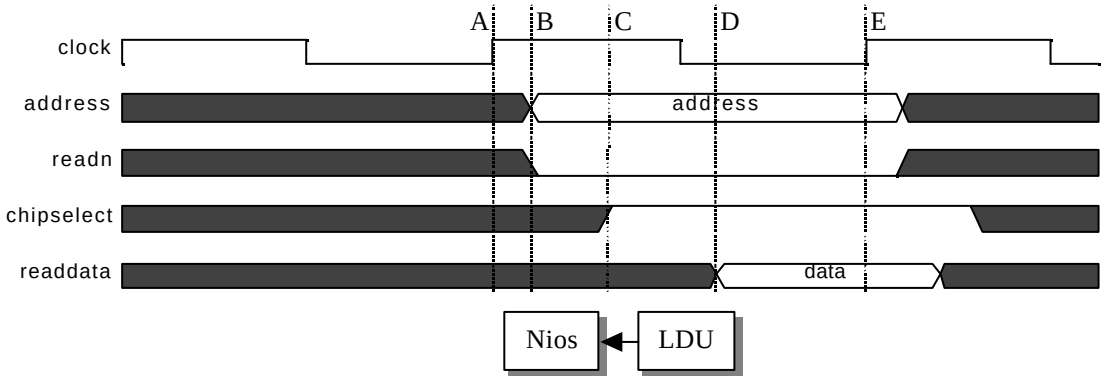


Figura 48 – Exemplo de uma operação de leitura do processador.

Pode-se notar em ambas figuras (Figura 47 e Figura 48) que há um certo atraso combinacional para a geração do sinal *chipselect*. Isto ocorre pois este atraso é o tempo que a lógica interna do barramento Avalon necessita para decodificar o endereço.

Novas versões do ambiente Excalibur (SOPC Builder 2.52) possuem recursos que podem ser explorados em trabalhos futuros, como: múltiplos mestres, suporte a DMA (*Direct Memory Access*) e leitura e escrita em rajadas. Com suporte a múltiplos mestres pode-se investigar teste de SOCs com múltiplos processadores Estes poderiam executar o programa de teste em paralelo, reduzindo o tempo de teste. Com o suporte a DMA é possível que seja definida uma memória de teste reduzida, pois o programa de teste pode ser carregado somente quando for utilizado. Com o suporte à escrita e leitura em rajada pode-se diminuir o tempo de transferência de dados entre o processador e o módulo. Estes recursos não foram utilizados neste projeto por delimitação de escopo do projeto.

6.3 Interface de Hardware e Software

Para integrar um módulo de hardware no MSN é necessário que este módulo respeite algumas restrições do barramento Avalon. Devido ao fato de desejarmos que o ambiente suporte módulos de propriedade intelectual, onde não é possível alterar a descrição de hardware de forma a se adequar ao barramento Avalon, há a necessidade do desenvolvimento de uma lógica envoltória para adequar a LDU ao barramento Avalon. Esta lógica envoltória é descrita na Seção 7.3.3.

Na Figura 49 é apresentado um trecho de código em linguagem C com as operações necessárias para acessar o hardware. Inicialmente o usuário deve declarar um ponteiro para a posição de memória referente ao hardware (linha 4). É importante destacar que a constante *na_user_hwif*, definida na biblioteca *nios.h*, contém este endereço de memória definido pelo usuário. Depois, basta ler e escrever nesta posição de memória para enviar e receber dados do hardware (linhas 7 e 9).

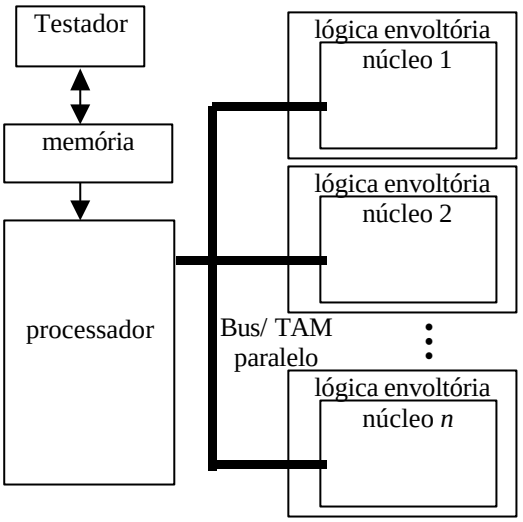
```
1.  #include "nios.h"
2.
3.  int main(void) {
4.    volatile np_usersocket *hwif = na_user_hwif;
5.    ...
6.    *hwif = data;
7.    ...
8.    data = *hwif;
9.
10.   return 0;
11. }
```

Figura 49 – Exemplo de software utilizado para acessar o hardware.

6.4 Arquitetura Alvo para Avaliação

A Figura 50 mostra como as descrições de núcleos geradas pelo ambiente de teste apresentado na Seção 7.3.3 são inseridas no ambiente Excalibur. Estes núcleos devem possuir uma lógica envoltória para facilitar a integração do núcleo no sistema e facilitar o processo de teste. Um sub-conjunto dos *benchmarks* ISCAS85 [13] e ISCAS89 [12] é utilizado como núcleos neste trabalho. Por motivo de limitação do ambiente Excalibur e de delimitação de escopo do trabalho, a forma de comunicação deve ser baseada em barramento[1].

[1] Porém a metodologia empregada está sendo desenvolvida sem se limitar a esta arquitetura.



Neste sistema o testador é um PC conectado à placa de prototipação por uma porta serial. A memória utilizada para armazenar o programa e dados de teste pode ser a memória disponível internamente ao FPGA ou uma memória externa. O próprio barramento é usado como mecanismo de acesso ao teste. Todos os núcleos do sistema possuem uma lógica envoltória responsável pela interface com o barramento. Esta lógica envoltória será detalhada na Seção 7.3.3.

Figura 50 – Estrutura genérica do SOC utilizada neste trabalho.

7 AMBIENTE PARA INTEGRAÇÃO DE TESTE NO FLUXO DE PROJETO DE SOCS

Este Capítulo apresenta o desenvolvimento de uma ferramenta de CAD que tem por objetivo inserir teste de hardware a partir de software no fluxo de concepção de um SOC.

O ambiente desenvolvido possui quatro funções principais: (*i*) inserir cadeias de varredura com DFTAdvisor nos núcleos, quando for necessário; (*ii*) parametrizar a lógica envoltória; (*iii*) gerar a interface do SOC e integrar os núcleos; (*iv*) gerar o código de teste.

A Figura 51 e a Figura 52 apresentam o fluxo de projeto dividido em duas partes: o fluxo de integração de núcleos e o fluxo de geração de código de teste.

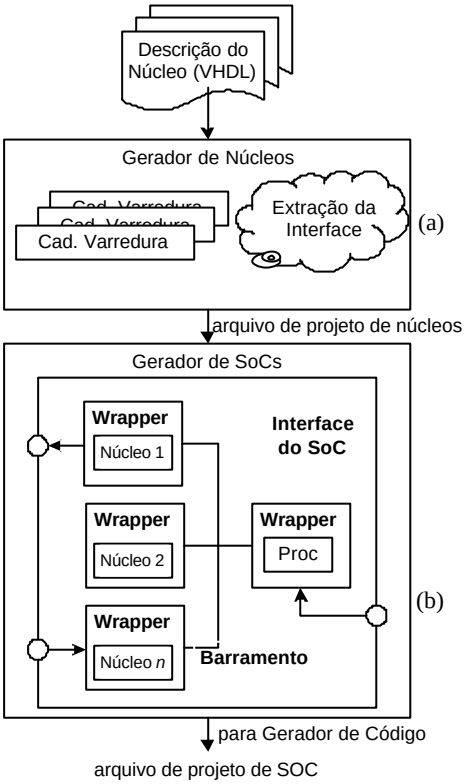


Figura 51 – Fluxo de desenvolvimento.

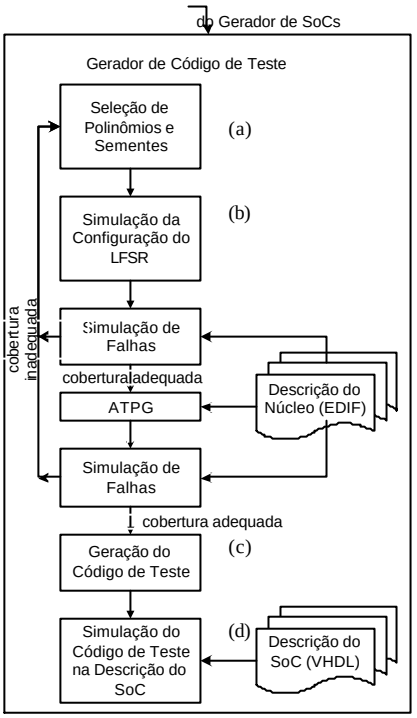


Figura 52 – Fluxo de teste.

O fluxo de integração de núcleos inicia com uma ferramenta chamada gerador de núcleos – Figura 51(a). Esta ferramenta não gera o núcleo propriamente dito mas extrai a interface do núcleo e pode inserir cadeias de varredura quando necessário. Estas cadeias de varredura são inseridas com o auxílio da ferramenta DFTAdvisor™ da Mentor. A fase de extração da interface gera um arquivo chamado de "arquivo de projeto de núcleos".

A próxima etapa é a geração do SOC - Figura 51(b). Nesta etapa são executadas as seguintes ações: (*i*) seleção dos núcleos e do processador; (*ii*) seleção do meio de comunicação; (*iii*) síntese da lógica envoltória para cada núcleo de hardware; (*iv*) geração da interface do SOC com meio externo.

Uma vez que o SOC é criado, a próxima etapa é a criação do código de teste que gera os padrões de teste, compacta as respostas e avalia cada núcleo de hardware. A abordagem baseia-se no trabalho de Hellebrand [32], o qual utiliza diferentes configurações de LFSR para aumentar a cobertura de falhas obtida de padrões pseudo-aleatórios. Desta forma, menos padrões determinísticos são armazenados na memória. A maior vantagem desta abordagem é a minimização dos requisitos do testador como largura de banda, memória e freqüência de operação. Entretanto, o

desafio é encontrar um compromisso entre tempo de teste para padrões pseudo-aleatórios e requisitos de memória para padrões determinísticos.

A primeira ação do gerador de código de teste (Figura 52(a)) é selecionar os polinômios e sementes para o gerador de padrões, sendo que múltiplos polinômios e sementes podem ser selecionados. O usuário também deve selecionar o polinômio e a semente do compactador. Neste caso somente um polinômio e uma semente podem ser selecionados. O gerador de padrões e o compactador de respostas suportados são respectivamente o LFSR e o MISR modular.

Cada configuração de LFSR é simulada de forma a criar os padrões de teste - Figura 52(b). Os padrões gerados são traduzidos para o formato do simulador de falhas (Flextest™/Fastscan™) [51] para avaliar a cobertura de falhas. Se a cobertura resultante não for suficiente, o usuário pode executar a ferramenta de ATPG para gerar o restante dos padrões de teste ou escolher uma nova configuração de LFSR. Este processo é repetido para cada núcleo de hardware do SOC de forma a gerar os padrões de teste para todo o sistema.

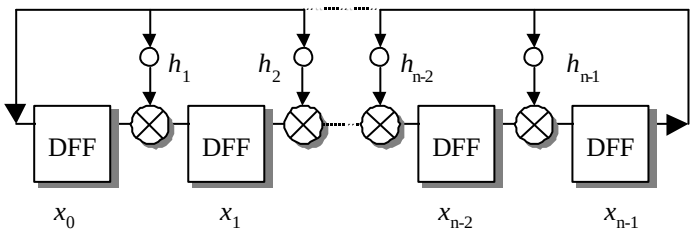
Finalmente, estes padrões de teste (pseudo-aleatório e determinísticos) são transformados para o código de teste em linguagem C - Figura 52(c). Depois da geração de código, o usuário pode utilizar uma ferramenta de co-simulação temporal para avaliar o tempo de teste - Figura 52(d). A ferramenta de co-simulação utilizada para este fim faz parte do ambiente Excalibur, a qual modela o processador no nível lógico de abstração.

A originalidade desta ferramenta encontra-se na integração das diversas ferramentas de CAD (Leonardo, ModelSim, FlexTest, FastScan e DFTAdvisor) e na criação de diversos módulos parametrizáveis, tanto de hardware como de software. A ferramenta foi desenvolvida utilizado QT™, o qual é um ambiente de programação que possibilita desenvolver interface gráfica multiplataforma em C++ [74].

Este capítulo está organizado como segue. A seção 7.1 apresenta como foram implementadas e validadas as rotinas de LFSR e MISR modular descritas em linguagem C. A seção 7.2 apresenta as ferramentas de geração automática de código C e VHDL. Detalhamento das classes utilizadas no decorrer do desenvolvimento são apresentadas no Anexo 10.1. A Seção 7.3 apresenta o ambiente de desenvolvimento de SOCs baseados em barramento. A seção 7.4 apresenta a ferramenta que automatiza o fluxo de teste de hardware baseado em software. A Seção 7.5 apresenta algumas limitações das ferramentas desenvolvidas. Por fim, a Seção 7.6 apresenta considerações finais.

7.1 Descrição de Geradores de Padrões e Compactadores de Respostas em Software

Conforme Bushnell e Agrawal [14], um LFSR modular genérico, apresentado na Figura 53, pode ser representado pela equação de matrizes representada na Figura 54.



n representa o número de estágios do LFSR. h_i representa o coeficiente do polinômio característico, onde $0 \leq i \leq n\text{-}1$. Quando h_i possui valor 1 significa que existe uma realimentação no coeficiente *i*.

Figura 53 – Exemplo de LFSR modular genérico.

$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ X_2(t+1) \\ \vdots \\ X_{n-3}(t+1) \\ X_{n-2}(t+1) \\ X_{n-1}(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 0 & h_1 \\ 0 & 1 & 0 & \cdots & 0 & 0 & h_2 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 & h_{n-3} \\ 0 & 0 & 0 & \cdots & 1 & 0 & h_{n-2} \\ 0 & 0 & 0 & \cdots & 0 & 1 & h_{n-1} \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ X_2(t) \\ \vdots \\ X_{n-3}(t) \\ X_{n-2}(t) \\ X_{n-1}(t) \end{bmatrix}$$

Esta equação pode ser representada por $X(t + 1) = T_s X(t)$. Se X é o estado inicial do LFSR, então os próximos estados do LFSR podem ser representados por $X, T_S X, T_S^2 X, T_S^3 X$

Figura 54 – Equação de um LFSR modular genérico [14].

A Figura 55 apresenta um exemplo de equação e do circuito para o polinômio $1 + x^2 + x^7 + x^8$.

$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ X_2(t+1) \\ X_3(t+1) \\ X_4(t+1) \\ X_5(t+1) \\ X_6(t+1) \\ X_7(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ X_2(t) \\ X_3(t) \\ X_4(t) \\ X_5(t) \\ X_6(t) \\ X_7(t) \end{bmatrix}$$

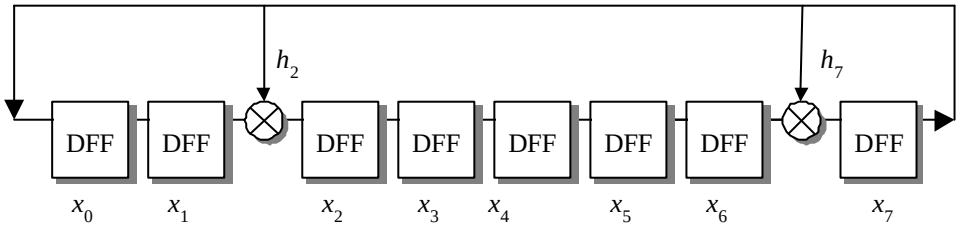


Figura 55 – Equação e circuito de um LFSR modular para f(x) = 1 + x^2 + x^7 + x^8.

Observando o comportamento do circuito de um LFSR modular desenvolveu-se o código em linguagem C apresentado na Figura 56. Observa-se que existe um registrador de deslocamento no circuito LFSR, que é equivalente a uma instrução de rotação. Porém, a linguagem C não possui esta instrução, somente a de deslocamento. Portanto, uma instrução de rotação é emulada utilizando uma instrução de deslocamento e incremento. Caso o bit mais significativo seja '0' acontece apenas a propagação dos conteúdos dos flip-flops (*i<<=1*). Caso o bit mais significativo seja '1' ocorre a propagação dos conteúdos dos flip-flops, o incremento do bit menos significativo e a inversão dos valores onde o índice do polinômio for igual a 1 (operação *xor)*.

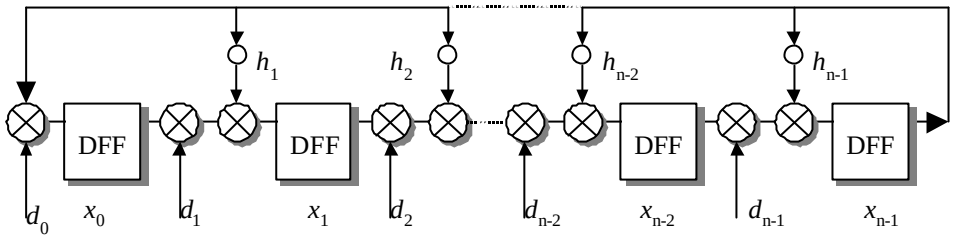
É necessário descobrir uma forma genérica de representar o polinômio característico do LFSR. Observando a Figura 54 pode-se chegar a conclusão que a representação desejada para o polinômio na rotina é semelhante à representação também utilizada para representar o polinômio na matriz. Ou seja, h_0 ... h_{n-1}. A diferença é que o valor lógico de h_0 na matriz é fixo em '1' e na rotina é fixo em '0'. Por exemplo, como pode ser verificado na Figura 55, o valor do vetor *h* que representa na matriz o polinômio $1 + x^2 + x^7 + x^8$ é 10100001. Já na rotina este valor seria 00100001, onde o valor sublinhado representa o bit menos significativo. O motivo desta diferença na implementação da representação do polinômio na matriz e na rotina é que na rotina o bit menos significativo é alterado em função da instrução de incremento que é realizada antes da instrução ou-

exclusivo. Com isto, o funcionamento de um LFSR modular foi simplificado a um conjunto de instruções simples, adequado para a execução em um software embarcado o que, ao mesmo tempo, auxilia na redução do tempo total de teste do sistema e na redução do tamanho do código de teste.

```
1. void lfsr( unsigned polynomial, int *state)
2. {  int i = *state;
3.    i <<=1;
4.    if (*state < 0){ // bit mais significativo igual a '1'
5.          i++;
6.          i ^= polynomial;
7.    }
8.    *state = i;
9. }
```

Figura 56 – Rotina de LFSR modular desenvolvida.

De forma semelhante ao LFSR modular, um MISR modular, representado na Figura 57, pode ser emulado utilizando a equação de matrizes representada na Figura 58.



n representa o número de estágios do MISR. h_i representa o coeficiente do polinômio característico, onde $0 \leq i \leq n\text{-}1$. Quando h_i possui valor 1 significa que existe uma realimentação no coeficiente i. d_i representa a resposta do UUT que é compactada.

Figura 57 – Exemplo de MISR modular genérico.

$$\begin{bmatrix} X_0(t+1) \\ X_1(t+1) \\ X_2(t+1) \\ \vdots \\ X_{n-3}(t+1) \\ X_{n-2}(t+1) \\ X_{n-1}(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 0 & h_1 \\ 0 & 1 & 0 & \cdots & 0 & 0 & h_2 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 & h_{n-3} \\ 0 & 0 & 0 & \cdots & 1 & 0 & h_{n-2} \\ 0 & 0 & 0 & \cdots & 0 & 1 & h_{n-1} \end{bmatrix} \begin{bmatrix} X_0(t) \\ X_1(t) \\ X_2(t) \\ \vdots \\ X_{n-3}(t) \\ X_{n-2}(t) \\ X_{n-1}(t) \end{bmatrix} + \begin{bmatrix} d_0(t) \\ d_1(t) \\ d_2(t) \\ \vdots \\ d_{n-3}(t) \\ d_{n-2}(t) \\ d_{n-1}(t) \end{bmatrix}$$

Esta equação pode ser representada por $X(t + 1) = T_s X(t) + d(t)$.

Figura 58 – Equação de um MISR modular genérico [14].

A Figura 59 apresenta a rotina de MISR desenvolvida. A diferença em relação à implementação do LFSR é que o MISR deve considerar a entrada de dados que representa as respostas do UUT. Como pode ser observado, a única instrução adicionada é o ou-exclusivo com a resposta (*input*).

```
1. void misr(const unsigned int polynomial,
2.             int input, int *state)
3. { int i = *state;
4.   i <<=1;
5.   if (*state < 0){
6.         i++;
7.         i ^= polynomial;
8.   }
9.   i^=input;
10.  *state = i;
11.}
```

Figura 59 – Rotina desenvolvida de MISR modular.

É importante destacar o motivo da escolha de implementação do LFSR e do MISR modular ao invés do LFSR e do MISR padrão. Como já apresentado na Figura 14 na página 15, tanto o LFSR quanto o MISR padrão possuem realimentações que dependem do estágio anterior. Isto induz à implementação de um laço na rotina de software, o que aumentaria o tempo de teste do sistema.

Uma alternativa a estes métodos, como apresentado por Rajski e Tyszer [64], é através do uso de somadores como geradores de padrões pseudo-aleatórios e compactadores de respostas. Esta abordagem diminui ainda mais o tempo de teste baseado em software pois, ao invés do conjunto reduzido de instruções apresentadas na Figura 56 e Figura 59, somente uma instrução as substituiria. Porém, esta abordagem ainda não é comum e também não é suficientemente documentada, pois, por exemplo, o próximo valor de um LFSR é definido pelo seu polinômio característico. Em somadores, como se define este próximo valor ? somente somando, por exemplo, o valor 5 ao valor atual ? existe alguma implicação na escolha deste valor somado ? Estas são algumas questões que impediam o uso de somadores.

7.1.1 Validação da Descrição de LFSR e MISR em Software

Para validar as rotinas de LFSR e MISR modular desenvolvidas, foi utilizado o software matemático Maple V. O conjunto de instruções do Maple V, apresentado na Figura 60 e Figura 61, foi utilizado para emular um LFSR e um MISR modular, respectivamente.

```
1. with(linalg):
2. Ts := array( [[0,0,1],[1,0,1] ,[0,1,0]] ):
3. Xt := vector( [1,1,0] ):
4. multiply(Ts, Xt);
```

Estas instruções criam uma matriz chamada Ts que representa o polinômio característico $x^3 + x + 1$ e um vetor chamado Xt que representa o estado atual do LFSR. A multiplicação de Ts e Xt gera X(t+1), ou seja, o próximo estado do LFSR. Neste caso, como o valor do estado atual é 110b, o próximo estado deve ser 011b, sendo o bit sublinhado menos significativo.

Figura 60 – Instruções do Maple V para emular um LFSR modular.

```
1. with(linalg):
2. Ts := array( [[0,0,1],[1,0,1] ,[0,1,0]] ):
3. Xt := vector( [1,1,0] ):
4. dt := vector( [1,1,1] ):
5. matadd(multiply(Ts, Xt),dt);
```

Estas instruções criam uma matriz chamada Ts que representa o polinômio característico $x^3 + x + 1$, um vetor chamado Xt que representa o estado atual do MISR e um vetor dt que representa a resposta do UUT. A multiplicação de Ts e Xt somado com dt gera X(t+1), ou seja, o próximo estado do MISR. Neste caso, como o valor do estado atual é 110b e da resposta é 111b, então o próximo estado deve ser 100b, sendo o bit sublinhado menos significativo.

Figura 61 – Instruções do Maple V para emular um MISR modular.

Os exemplos validados com Maple V utilizaram um LFSR e um MISR de 32 estágios, o que representa uma multiplicação de uma matriz 32x32 e um vetor de 32 posições.

7.2 Ferramenta de Geração de Código

Uma das funções das ferramentas desenvolvidas é a geração automática de códigos. Dentre os códigos gerados citamos a lógica envoltória, o *testbench* de lógica envoltória, a interface do SOC e o código de teste. Os três primeiros códigos são descritos em VHDL enquanto o último é descrito

em linguagem C. Desta forma, foi desenvolvido em linguagem C++ um gerador automático de código reutilizável.

A Figura 62 resume a estrutura de um gerador de código específico. Um gerador de código específico, chamado de *xCodeGen*, instancia dois módulos principais, o gerador de tabela de símbolos e o gerador de código genérico, respectivamente *xTokenGen* e *CodeGen*. A interface entre estes módulos ocorre através dos métodos apresentados nesta figura, sendo que o tipo de dado principal é chamado de *Token*. O módulo *xTokenGen* recebe todos os parâmetros necessários (basicamente o arquivo de projeto de SOC de extensão .soc) para atribuir valores aos símbolos existentes no arquivo de *template* (extensão .tpl). A linha tracejada entre *xTokenGen* e o arquivo de *template* significa que *xTokenGen* não manipula diretamente este arquivo, mas deve conhecer os símbolos existentes no mesmo. Por outro lado, o módulo *CodeGen* recebe três parâmetros que são o arquivo de *template*, a tabela de símbolos preenchida pelo *xTokenGen* e o nome do arquivo que deve ser gerado, representado pela extensão .out, mas que poderia ser um programa nas linguagens C ou VHDL.

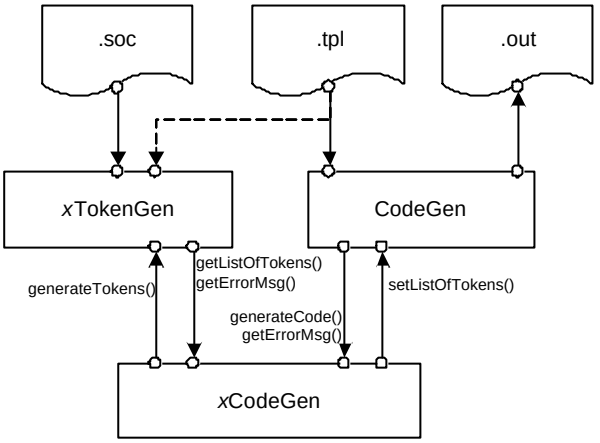


Figura 62 – Representação dos módulos de geração de código.

O gerador de código possui três módulos de software descritos em C++ e um modelo (*template*) descrito na linguagem do código gerado (e.g. VHDL ou C). A comunicação entre estes módulos ocorre através da passagem de um tipo de dados chamado de *Token*. Este tipo de dados, exemplificado na Tabela 8, representa uma tabela do tipo *string* de dimensão 2 x *n*, onde *n* é o número de símbolos da tabela. A primeira coluna desta tabela contém o nome dos símbolos e a segunda coluna contém o seu valor. O valor de um símbolo pode ser uma palavra ou várias linhas de código.

Tabela 8 – Exemplo de tabela do tipo *Token.*

WRAPPER_NAME	WR_C880
DATAIN_WIDTH	60
DATAOUT_WIDTH	26

O primeiro módulo de software, chamado de *xCodeGen* ou gerador de código específico, é apresentado na Figura 63. Este módulo instancia os outros dois módulos de software, chamados de *xTokenGen* ou gerador de tabela de símbolo específico e *CodeGen* ou gerador de código genérico. Este código é diferente para cada gerador de código desenvolvido, porém, a estrutura básica é semelhante. Basicamente as alterações que ocorrem são nas mensagens de saída e nos parâmetros que são passados para o gerador de tabela de símbolos (linha 8).

```
1.  #include "XTokenGen.h"
2.  #include "CodeGen.h"
3.
4.  int main(int argc, char **argv)
5.  {
6.    cout << "Generating X " << endl;
7.    // inicializa gerador de símbolos
8.    XTokenGen xTokenGen(argv[1],...,argv[n]);
9.    CodeGen codeGen;
10.   // gera símbolos
11.   if (!xTokenGen.generateTokens()){
12.         cout << xTokenGen.getErrorMsg();
13.         cout << "X not generated due to errors !!!\n";
14.         return 0;
15.   }
16.   // envia lista de símbolos para Code Generator
17.   codeGen.setListOfTokens(xTokenGen.getListOfTokens());
18.
19.   // gera código x
20.   if (!codeGen.generateCode(templateFile,generatedFile)){
21.         cout << codeGen.getErrorMsg();
22.         cout << "X not generated due to errors !!!\n";
23.         return 0;
24.   }else
25.         cout << "X generated succesfully !!!\n";
26.   return 1;
27. }
```

Um gerador de código específico é composto pela combinação de um gerador de símbolos com o gerador de código genérico. Inicialmente este código deve acrescentar a definição da classe do gerador de símbolos (*xTokenGen*) e da classe do gerador de código genérico (*CodeGen*). Após, as seguintes operações devem ser executadas: inicializar o gerador de símbolos com os valores pertinentes (linha 8); executar geração de símbolos (linha 11); enviar os símbolos criados para o gerador de código genérico (linha 17); gerar o código (linha 20). O método *getErrorMsg*() é executado em caso de erro nestas etapas.

Figura 63 – Exemplo de estrutura de um gerador de código específico (*xCodeGen*).

O segundo módulo, chamado de *xTokenGen* ou gerador de tabela de símbolo específico, é apresentado na Figura 64.

```
1.  class xTokenGen {
2.    string m_socFile,m_errorMsg;
3.    vector<Token> m_tokens;
4.
5.    // métodos e atributos específicos
6.  public:
7.    ~xTokenGen();
8.    xTokenGen(string socFile){m_socFile = socFile;}
9.    bool generateTokens();
10.   vector<Token>& getListOfTokens() {return m_tokens;}
11.   string getErrorMsg() {return m_errorMsg;}
12. };
```

Os três atributos básicos desta classe são *m_socFile,m_errorMsg* e *m_tokens*, cujos significados respectivos são o nome do arquivo de projeto de SOC, a mensagem de erro e a lista de símbolos. Os cinco métodos públicos são: o destrutor *~xTokenGen*() utilizado para desalocar memória; o construtor *xTokenGen*() para atribuir valores aos atributos; o método principal chamado *generateTokens*() utilizado gerar os símbolos; *getListOfTokens*() utilizado para retornar a tabela de símbolos gerada; *getErrorMsg*() utilizado para capturar as mensagens de erros. Outros métodos e atributos podem ser declarados como privados.

Figura 64 – Exemplo de estrutura de um gerador de símbolos (*xTokenGen*).

O código da Figura 64 reconhece os símbolos existentes no *template* do código. Portanto, sua função é atribuir valores a estes símbolos de acordo com os parâmetros que são passados para esta classe. Desta forma, este programa personaliza o *template* genérico gerando diferentes códigos. A

implementação deste módulo é caso a caso, dependendo do número de símbolos existente no *template* e da complexidade de montar os valores dos mesmos. Somente a interface desta classe deve seguir um padrão para que a integração do módulo *xCodeGen* seja genérica. Dentre os métodos obrigatórios a este tipo de classe citamos *getTokens*(), *getListOfTokens*() e *getErrorMsg*().

O terceiro módulo de software, chamado de CodeGen ou gerador de código genérico, é apresentado na Figura 65. Este módulo é responsável por substituir os símbolos enviados por xTokenGen e alterá-los no *template* de forma a gerar um código específico.

```
1.  Class CodeGen {
2.    Vector<Token> m_tokens;
3.    string m_errorMsg;
4.    char * findToken(char *token_word);
5.  public:
6.    ~CodeGen(){m_tokens.clear();}
7.    CodeGen(){}
8.    void setListOfTokens(vector<Token> tokens) { m_tokens = tokens;}
9.    string getErrorMsg() {return m_errorMsg;}
10.   bool generateCode(string in_filename,string out_filename);
11. };
```

Esta classe é instanciada em cada programa de geração de código. Sua função básica é substituir os símbolos existentes no template pelo seu valor real, gerado pelo *xTokenGen*. A classe *CodeGen* possui os atributos *m_tokens* e *m_errorMsg* que armazenam, respectivamente, a lista de símbolos gerados pelo *xTokenGen* e as mensagens de erro. Os métodos públicos são: destrutor e construtor da classe; *setListOfTokens*() utilizado para receber os símbolos de *xTokenGen*; *getErrorMsg*() que retorna as mensagens de erro; *generateCode*() executado para realizar a substituição dos símbolos gerando o código específico.

Figura 65 – Gerador de código genérico (*CodeGen*).

A Figura 66 apresenta um exemplo de *template* utilizado para demonstrar o funcionamento do gerador de código.

```
1.  entity #define TB_NAME is
2.      generic(...);
3.  end entity #define TB_NAME ;
4.
5.  architecture beh of #define TB_NAME is
6.
7.      file PatternFile : TEXT open READ_MODE is PATTERN_FILE;
8.
9.      component #define WRAPPER_NAME
10.         port(
11.         d           : in  std_logic_vector(31 downto 0);
12.         addr        : in  std_logic_vector(5 downto 0);
13.         reset       : in  std_logic;
14.         clock       : in  std_logic;
15.         wr          : in  std_logic;
16.         rd          : in  std_logic;
17.         cs          : in  std_logic;
18.         q           : out std_logic_vector(31 downto 0)
19.         );
20.     end component;
21.
22.     component #define CORE_NAME is
23.       port (...);
24.     end component;
25.
26. begin
27. ...
```

Os símbolos são os trechos apresentados em itálico. Os valores constantes do componente referem-se à largura do barramento utilizado.

Figura 66 – Exemplo de trecho do *template* de *testbench* da lógica envoltória.

Associado a este *template* deve existir uma classe do tipo *xTokenGen* responsável por atribuir valores aos símbolos apresentados na Tabela 9. O valor associado a estes símbolos são preenchidos somente depois da execução do método *generateTokens*() da classe *xTokenGen*. Este símbolos são passados para a classe *CodeGen* que é responsável por substituir os símbolos pelo valor associado ao mesmo através do método *generateCode*(). A Figura 67 apresenta um exemplo de código gerado.

Tabela 9 – Tabela de símbolos do exemplo de *testbench* da lógica envoltória.

Antes da geração de símbolos	
TB_NAME	
WRAPPER_NAME	
CORE_NAME	

Depois da geração de símbolos	
TB_NAME	TB_WR_C880
WRAPPER_NAME	WR_C880
CORE_NAME	C880

```
1.  entity TB_WR_C880 is
2.      generic(...);
3.  end entity TB_WR_C880 ;
4.
5.  architecture beh of TB_WR_C880 is
6.
7.      file PatternFile : TEXT open READ_MODE is PATTERN_FILE;
8.
9.      component WR_C880
10.         port(
11.         d           : in  std_logic_vector(31 downto 0);
12.         addr        : in  std_logic_vector(5 downto 0);
13.         reset       : in  std_logic;
14.         clock       : in  std_logic;
15.         wr          : in  std_logic;
16.         rd          : in  std_logic;
17.         cs          : in  std_logic;
18.         q           : out std_logic_vector(31 downto 0)
19.         );
20.     end component;
21.
22.     component C880 is
23.       port (...);
24.     end component;
25.
26. begin
27. ...
```

Os símbolos foram removidos de acordo com a tabela de símbolos de forma a criar uma descrição VHDL personalizada.

Figura 67 – Exemplo de trecho de código gerado do *testbench* da lógica envoltória.

7.3 Ferramenta de Integração de Núcleos de Hardware

O principal objetivo desta ferramenta é integrar os núcleos e gerar a interface do SOC. Para tanto, este fluxo de desenvolvimento é dividido em duas partes: a criação de projetos de núcleos e a criação de projetos de SOCs (Figura 51, página 63).

7.3.1 Criação de Arquivos de Projetos de Núcleos

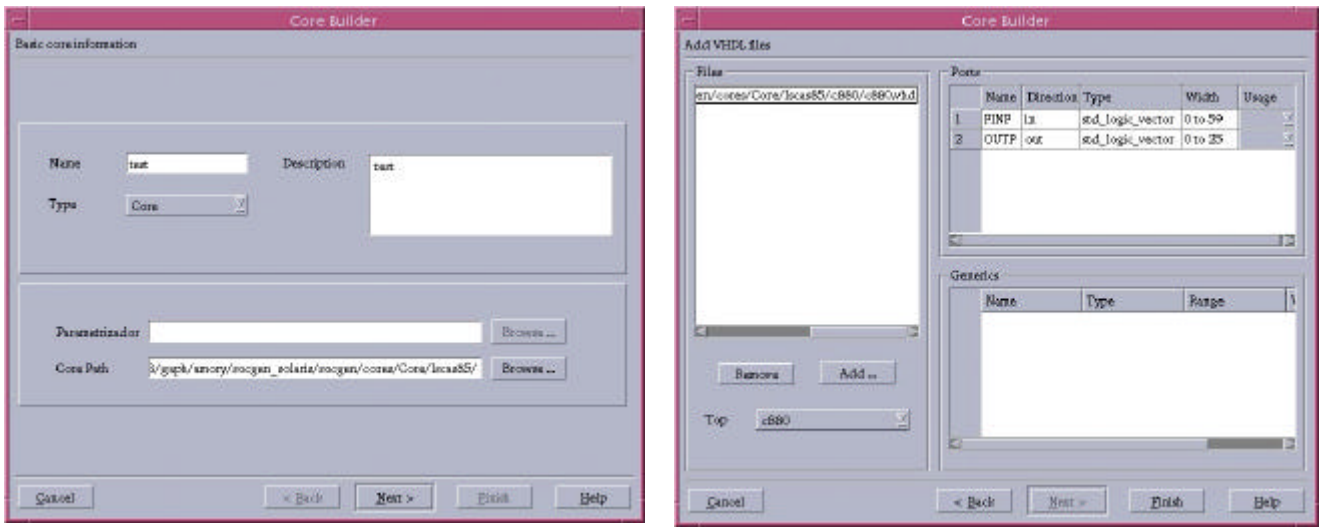
Um projeto de núcleo de hardware é um arquivo como o apresentado na Figura 68. Este arquivo concentra todas as informações necessárias para a posterior integração deste núcleo em um SOC. Dentre as informações podemos citar: o nome do núcleo; seu tipo (e.g. núcleo, processador ou memória); os arquivos VHDL que compõem o projeto; o nome da entidade da interface; e a descrição das portas da interface.

```
1.  NAME s1196
2.  TYPE Core
3.  FILES 1
4.  S1196.vhd
5.  TOP s1196
6.  PORTS 4
7.  PINP in std_logic_vector 0 to 13 data
8.  OUTP out std_logic_vector 0 to 13 data
9.  CLK in std_logic  clock
10. reset in std_logic  reset
```

Este exemplo demonstra um arquivo de projeto de um núcleo pertencente ao *bechmark* ISCAS89. Este núcleo se chama s1196, é composto por somente um arquivo VHDL e possui quatro portas: pinp, outp, clk e reset. A definição de tipo de cada porta é apresentada no decorrer da linha sendo que a última informação apresentada (sublinhado) é o uso da porta.

Figura 68 – Exemplo de arquivo de projeto de núcleos.

A Figura 69 apresenta as principais telas desta etapa de criação de projeto de núcleos. As informações que devem ser preenchidas na primeira tela são o nome do núcleo, sua descrição, seu tipo e o local onde o projeto deve ser gravado. Na segunda tela o usuário deve informar os arquivos VHDL que compõem este núcleo. As entidades existentes em cada arquivo adicionado são acrescentadas automaticamente na lista de seleção do campo *top*. Neste campo o usuário deve selecionar qual das entidades do projeto é a interface do núcleo. Esta ação executa automaticamente um analisador sintático desenvolvido para realizar a extração da interface. Assim, todas as portas e *generics* desta entidade são apresentadas nas tabelas *ports* e *generics*, respectivamente.



(a) (b)

Figura 69 – Telas do gerador de projeto de núcleos.

O usuário deve obrigatoriamente definir o campo *uso da porta* (última coluna da tabela *ports*), pois é com base nesta informação que a ferramenta sabe distinguir a função de cada porta do núcleo. Esta informação é utilizada para a integração dos núcleos e geração da interface do futuro SOC que utilizará este núcleo.

Todos os projetos de núcleos são acrescentados em uma biblioteca que os disponibiliza para a futura integração em um SOC.

7.3.2 Criação de Projeto de SOCs

Dada uma biblioteca de núcleos, a próxima tarefa é selecionar os núcleos que compõem o SOC, integrá-los de acordo com algum meio de comunicação e gerar a interface do SOC. Estas tarefas fazem parte da geração de projeto de SOC que gera o arquivo apresentado na Figura 70. Dentre as informações mais importantes deste arquivo destacamos: o nome, o tipo (e.g. ASIC ou FPGA), os núcleos que compõem o SOC e a definição da integração dos núcleos.

```
1.  NAME d281
2.  TYPE ASIC
3.  DESCRIPTION "benchmark from ITC02"
4.  CORES 9
5.  U1 c880.core 0
6.  U2 c2670.core 0
7.  U3 c7552.core 0
8.  U4 s953.core 0
9.  U5 s5378.core 0
10. U6 s1196.core 0
11. U7 s13207.core 0
12. U8 s1238.core 0
13. U9 8051.core 0
14. DFT_LIB default
15. TECH_LIB default
16. COMMUNICATION Bus Tri-state
17. MASTER U9
```

Este exemplo de projeto de um SOC, chamado de d281, foi retirado dos *benchmarks* ITC02. Este exemplo é implementado em ASIC, possui uma descrição textual e é formado por nove núcleos. Cada linha de definição de um núcleo é formada pelo nome de instância do núcleo, o arquivo de projeto do núcleo e o número de *generics* que este núcleo possui. Logo após segue o nome da biblioteca de DFT e de tecnologia usadas. Por fim, seguem informações do tipo de comunicação selecionada para integrar estes núcleos.

Figura 70 – Exemplo de arquivo de projeto de SOCs.

A Figura 71 apresenta as principais telas que integram a criação de projeto de SOC. Na primeira tela (Figura 71(a)) o usuário deve selecionar da biblioteca de núcleos quais são os núcleos que devem compor o SOC. Esta biblioteca deve ter sido previamente gerada como especificado na seção anterior. Ao adicionar um núcleo ao projeto, o usuário deve informar o nome da instância do módulo e os valores dos *generics*, se os mesmos existirem. Caso o núcleo que deve ser adicionado ao SOC não exista na biblioteca, o usuário pode adicioná-lo selecionando o módulo *user defined*. Este módulo lança as telas de definição de um núcleo de hardware apresentadas na Figura 69. Ainda na tela de seleção de núcleos, o usuário deve selecionar a forma de integração dos núcleos. Atualmente somente integração por barramento é suportada.

No próximo passo, apresentado na Figura 71(b), o usuário deve parametrizar o meio de comunicação especificado na etapa anterior. No caso de barramento, o usuário deve informar o mestre do barramento e como este barramento deve ser implementado fisicamente (e.g. *tri-state* ou multiplexador). Esta tela seria diferente se outro meio de comunicação fosse suportado, refletindo os parâmetros relacionados ao mesmo. Acreditamos que esta forma de implementação confere flexibilidade à ferramenta uma vez que a mesma está preparada para suportar outros meios de comunicação.

A terceira e a quarta telas, apresentadas na Figura 71(c,d), executam a ferramenta de geração de lógica envoltória, de *testbench* da lógica envoltória e da interface do SOC apresentadas na Seção 7.2. Ao fim desta etapa são criados *scripts* de síntese lógica com Leonardo Spectrum™ e de compilação com Modelsim™.

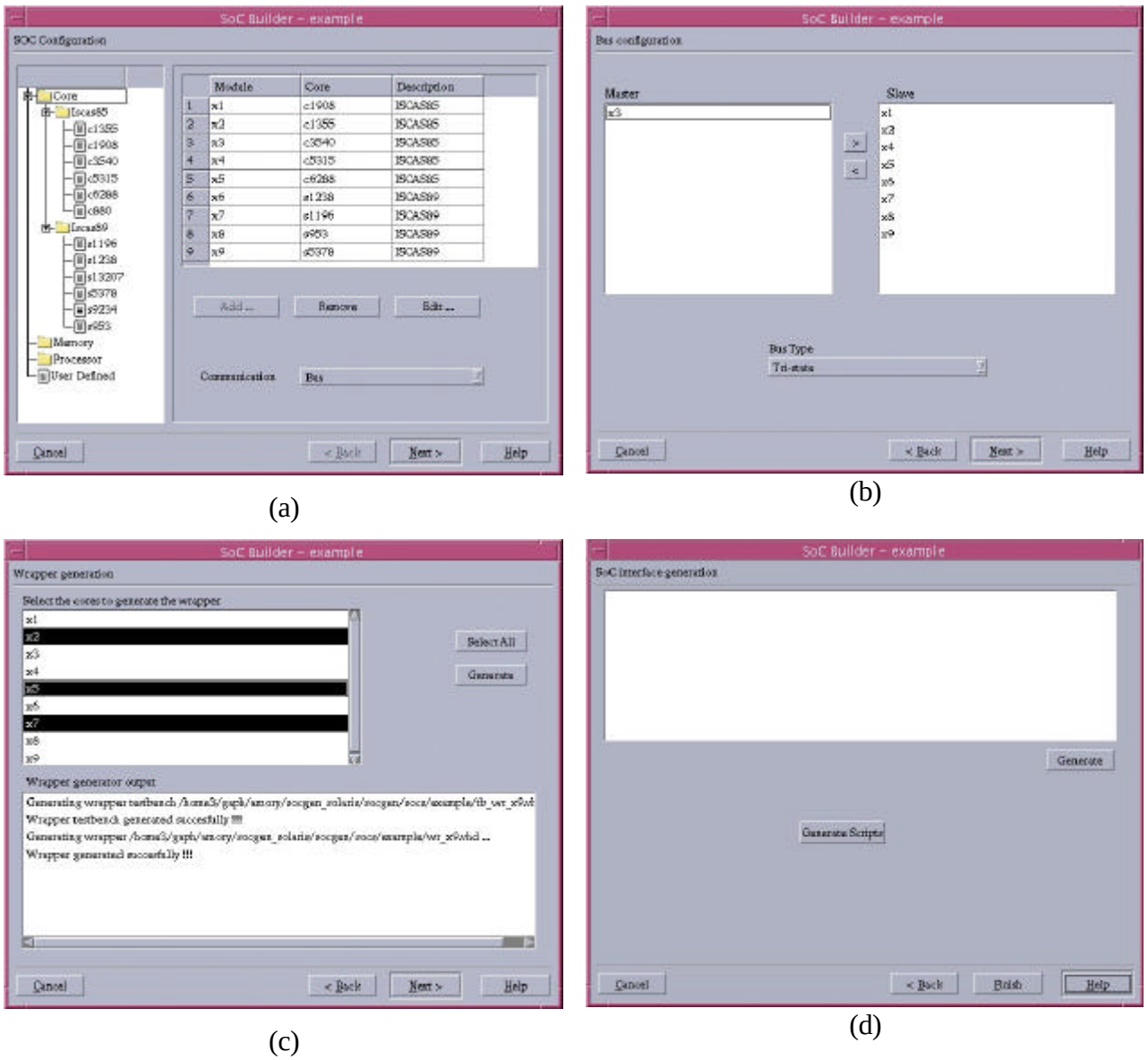


Figura 71 – Telas do gerador de projeto de SOCs.

7.3.3 Lógica Envoltória

Uma vez que o protocolo de comunicação foi selecionado, a lógica envoltória é sintetizada para cada núcleo. A lógica envoltória é gerada depois da definição do meio de comunicação, pois o protocolo de comunicação e a interface da lógica envoltória estão associados ao meio de comunicação.

A Figura 72 apresenta um exemplo de lógica envoltória. Ela possui cadeias de varredura, chamadas de células de entrada (*ci*) e saída (*co*), utilizadas para adaptar a largura do barramento (32 bits) à largura das entradas e saídas do núcleo. Neste exemplo, a largura das portas de entrada é de 68 bits, e a saída é de 42 bits. O teste é executado em duas etapas: carga dos padrões de teste e leitura das respostas. Os seis bits menos significativos da porta *addr* são utilizados para ativar o modo de teste, isto é, a carga de padrões ou leitura de respostas.

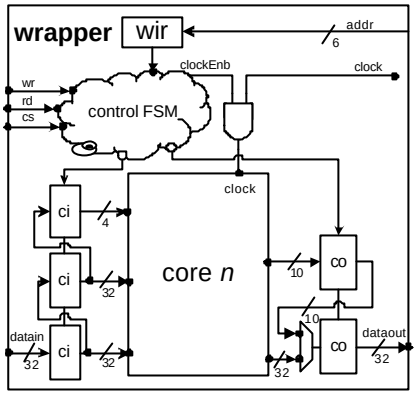


Figura 72 – Esquema de uma lógica envoltória.

Quando a lógica envoltória está no modo de teste e ocorre uma operação de escrita (porta *wr*), o sinal *clockEnb* é ativado durante um ciclo de relógio para processar os padrões atribuídos. Observar que o sinal *clockEnb* é ativado somente quando a última parte do padrão é escrita na célula de entrada. As respostas são armazenadas nas células de saída um ciclo após *clockEnb* ter sido ativado. O processador lê as respostas quando a porta *rd* é ativada.

Nesta etapa do fluxo de projeto, dois arquivos são criados para cada núcleo: a lógica envoltória e seu *testbench*. Este *testbench* possui dois modos de funcionamento: o modo com padrões gerados internamente e com padrões lidos de arquivo. O modo de padrões internos é preferencialmente utilizado para validar a lógica envoltória enquanto o outro modo é utilizado para executar a simulação que gera a assinatura esperada.

É interessante destacar que o *testbench* gerado automaticamente foi o mesmo *testbench* utilizado durante a validação da lógica envoltória. Desta forma, foi necessário apenas o esforço de adicioná-lo ao ambiente. Ainda sobre a validação da lógica envoltória, a mesma foi validada por simulação funcional, temporal e por prototipação em FPGA.

Nos Anexos 10.2.1 e 10.2.2 é apresentado um exemplo de lógica envoltória e de seu *testbench*.

7.3.4 Interface do SOC

A última parte do processo de integração de núcleos é a geração da interface do SOC, a qual é obtida através da informação do uso das portas contida no arquivo de projeto de núcleos. Quando uma porta é do tipo *external*, ela é conectada à interface do SOC conforme a Figura 73. A lógica envoltória em destaque possui portas conectadas à interface de comunicação e portas do tipo *external* conectadas à interface do SOC. Este tipo de porta impõe modificações (adição de portas) na lógica envoltória e na própria interface do SOC.

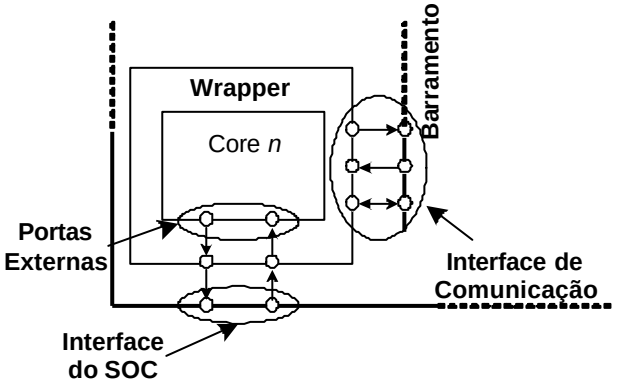
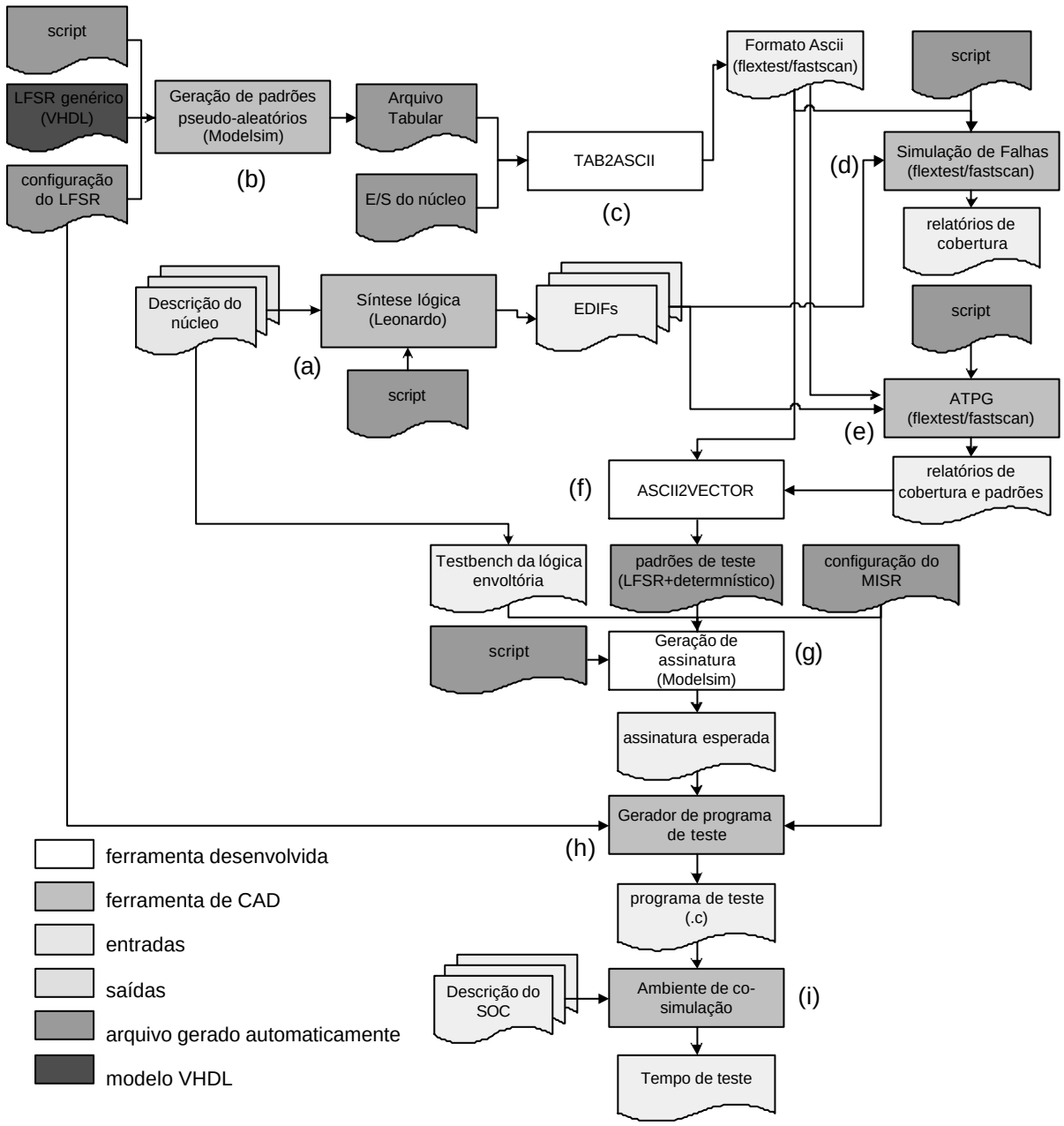


Figura 73 – Detalhe da criação da interface do SOC.

Um exemplo de interface de SOC é apresentado no Anexo 10.2.3.

7.4 Ferramenta de Auxílio ao Teste Baseado em Software

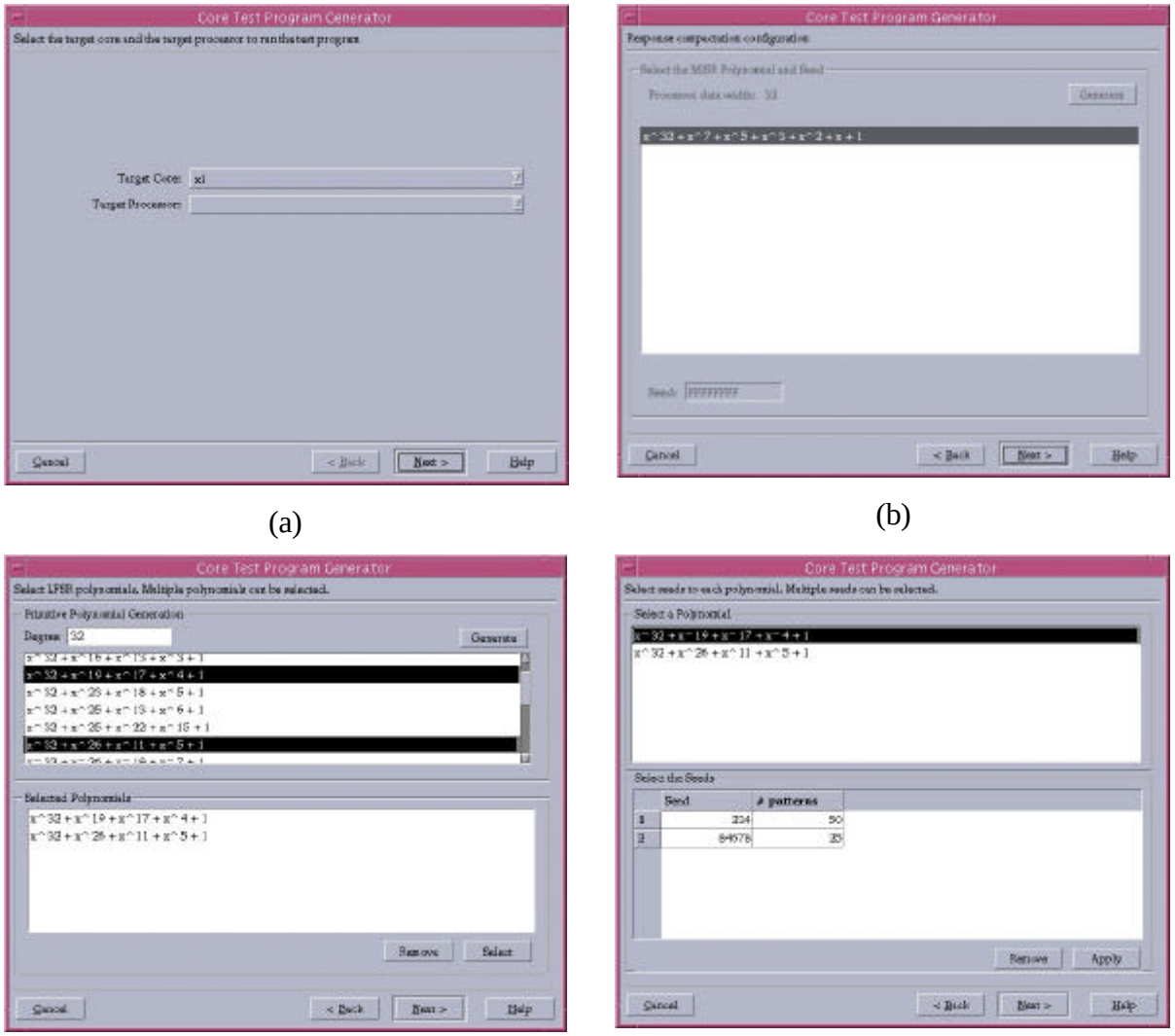
O objetivo principal da ferramenta de teste é a geração do código de teste. Para alcançar este objetivo diversas tarefas intermediárias são realizadas, conforme a Figura 74: seleção de configuração do LFSR e do MISR, execução de simulações de falhas e de ATPG, diversas conversões de formatos de arquivo, geração de assinatura e co-simulação do código de teste.



Exemplos de scripts para as ferramentas de CAD são apresentados na Seção 10.3.

Figura 74 – Representação do processo de geração de código de teste.

A Figura 75 apresenta as principais telas que compõem este fluxo: tela de seleção de núcleo de hardware, de configuração do MISR, de seleção de polinômio e de semente para o LFSR.



(a) (b)

(c) (d)

Figura 75 – Telas da ferramenta de geração de código de teste (referente a Figura 74(b)).

Na primeira etapa, apresentada na Figura 75(a), o usuário deve selecionar o núcleo de hardware que vai ser configurado para geração de software de teste. Existem dois campos de seleção, o primeiro é usado para selecionar o núcleo de hardware e o outro é usado para selecionar o processador que testará o núcleo. O campo de seleção de processador deve conter somente um processador, pois atualmente o ambiente suporta somente barramento com um mestre. É importante que exista este campo, pois indica que a interface gráfica está preparada para suportar múltiplos processadores no sistema.

Na segunda etapa, apresentada na Figura 75(b), o usuário deve selecionar a configuração do MISR. Esta configuração suporta somente um polinômio e uma semente. Os dados desta tela são gravados em um arquivo com o seguinte formato <NomeDaInstancia> <Polinômio> <Semente> <AssinaturaEsperada>.

Na terceira e na quarta etapa, apresentadas na Figura 75(c,d), o usuário deve selecionar a configuração do LFSR. Primeiramente o usuário seleciona um ou mais polinômios e depois seleciona as sementes para cada polinômio. A configuração de LFSR selecionada é salva em um arquivo no formato apresentado na Figura 76.

```
1. <NúmeroDePolinômios>
2. <Largura>
3. <Polinômio1>
4. <NúmeroDeSementes>
5. <Semente1>
6. <NúmeroDePadrões>
7. ...
8. <Sementen>
9. <NúmeroDePadrões>
10. ...
11. <Polinômion>
12. <NúmeroDeSementes>
13. <Semente1>
14. <NúmeroDePadrões>
15. ...
16. <Sementen>
17. <NúmeroDePadrões>
```

Figura 76 – Exemplo de arquivo de configuração de LFSR.

Este arquivo de configuração é utilizado por um modelo de LFSR genérico desenvolvido em VHDL conforme apresentado na Figura 74(b). Usando o arquivo de configuração do LFSR este modelo cria os padrões que esta configuração geraria no momento do teste através de simulação (ModelSim). Estes padrões são salvos no formato tabular que é suportado pelo Modelsim. Um exemplo do formato tabular é apresentado na Figura 77.

```
1.  ns       /lfsr_gen/outlfsr
2.            delta
3.             0  +0      UUUUUUUUUUUUUU
4.             1  +1      01010101000101
5.             2  +1      10101010001010
6.             3  +1      01010100101101
7.             4  +1      10101001011010
8.             5  +1      01010010001101
9.             6  +1      10100100011010
10.            7  +1      01001000001101
11.            8  +1      10010000011010
12.            9  +1      00100000001101
13.           10  +1      01000000011010
14.           11  +1      10000000110100
15.           12  +1      00000001010001
16.           13  +1      00000010100010
17.           14  +1      00000101000100
18.           15  +1      00001010001000
19.           16  +1      00010100010000
20.           17  +1      00101000100000
21.           18  +1      01010001000000
22.           19  +1      10100010000000
23.           20  +1      01000100111001
```

A primeira linha define o sinal que está sendo gravado. Cada linha define o valor assumido por este sinal. Este formato é utilizado principalmente devido a sua simplicidade para gerar os padrões de uma configuração de LFSR.

Figura 77 – Formato tabular gerado pelo Modelsim.

O formato tabular é convertido para o formato do simulador de falhas, apresentado na Figura 78, para que a configuração de LFSR seja avaliada. O programa *tab2ascii*, apresentado na Figura 74(c), foi desenvolvido para realizar esta conversão.

Um *script* contendo os comandos do simulador de falhas é personalizado pela ferramenta de teste para a execução da simulação - Figura 74(d). É importante destacar que a simulação de falhas deve ter como entrada o circuito já sintetizado para a tecnologia alvo. Isto é obtido através da síntese lógica da descrição VHDL do núcleo - Figura 74(a), gerando uma descrição no nível lógico de abstração deste (formato EDIF).

```
1.  SETUP =
2.      declare input bus "PI" =
3.            "/PINP<0>", "/PINP<1>", "/PINP<2>", "/PINP<3>", "/PINP<4>",
4.            "/PINP<5>", "/PINP<6>", "/PINP<7>", "/PINP<8>", "/PINP<9>";
5.      declare output bus "PO" =
6.           "/OUTP<0>", "/OUTP<1>", "/OUTP<2>", "/OUTP<3>", "/OUTP<4>",
7.           "/OUTP<5>", "/OUTP<6>", "/OUTP<7>", "/OUTP<8>", "/OUTP<9>";
8.  end;
9.  SCAN_TEST =
10.     pattern = 0;
11.     force   "PI" "0011000011" 0;
12.     measure "PO" "1001100001" 1;
13.
14.     pattern = 1;
15.     force   "PI" "1100110000" 0;
16.     measure "PO" "1100110000" 1;
17.
18.     pattern = 2;
19.     force   "PI" "0111111001" 0;
20.     measure "PO" "0111111001" 1;
21. end;
```

Este exemplo representa padrões de teste gerados para um circuito combinacional com uma entrada e uma saída, respectivamente PINP e OUTP, de dez bits cada uma. Este formato é dividido em duas seções: setup e scan_test. A seção *setup* define, entre outras funções, as portas de entrada e saída do circuito. A seção *scan_test* define os padrões de teste. O comando force atribui um valor ao circuito e o comando *measuse* lê os valores de saída.

Figura 78 – Exemplo de padrões de teste no formato ASCII suportado por Flextest e Fastscan.

A simulação da cobertura de falhas obtida com esta configuração de LFSR é detalhada em um relatório. Depois deste passo o usuário pode executar o ATPG para gerar padrões determinísticos aumentando a cobertura de falhas - Figura 74(e). Porém, antes da execução de ATPG outro *script* deve ser personalizado para gerar padrões determinísticos para este núcleo de hardware. É importante ressaltar que este *script* deve indicar ao simulador de falhas para gerar padrões apenas para as falhas não detectadas pelos padrões do LFSR. Ao fim destes processos existem dois arquivos de padrões de teste no formato texto, um para padrões pseudo-aleatórios e outro para padrões determinísticos.

Com os padrões de teste gerados, a próxima etapa é a geração da assinatura do núcleo. Para gerar a assinatura os padrões de teste são convertidos, conforme Figura 74(f), para um formato mais simples apresentado na Figura 79. Este formato concatena os padrões pseudo-aleatórios e determinísticos em um único arquivo. Este arquivo é utilizado pelo *testbench* da lógica envoltória para simular o núcleo de hardware e apresentar ao final a assinatura esperada - Figura 74(g).

```
1.   01010101000101
2.   10101010001010
3.   01010100101101
4.   10101001011010
5.   01010010001101
6.   10100100011010
7.   01001000001101
8.   10010000011010
9.   00100000001101
10.  01000000011010
```

Este formato é gerado pelo programa Ascii2Vector e é utilizado para geração de assinatura do módulo. Os padrões pseudo-aleatórios e determinísticos são concatenados neste formato de arquivo. Desta forma, o *testbench* do módulo pode ser utilizado para ler este arquivo e simular os valores de saídas esperados.

Figura 79 – Formato de vetores utilizado para geração de assinatura do módulo.

A etapa subseqüente é a geração de código de teste que utiliza a configuração de LFSR, do MISR e o arquivo de padrões de teste determinístico para criar o programa em linguagem C - Figura 74(h). Um exemplo de código de teste gerado é apresentado e comentado no Anexo 10.2.4. A última etapa é a co-simulação do código de teste, afim de obter o tempo de teste do sistema Figura 74(i). Para este fim utilizamos o ambiente de co-simulação Excalibur.

7.5 Limitações

Embora um fluxo de projeto completo tenha sido desenvolvido, o mesmo possui diversas limitações apresentadas nesta Seção.

7.5.1 Quanto aos Núcleos de Hardware

Não existem limitações quanto à estrutura dos núcleos, mas existem limitações quanto à interface do mesmo. Estas limitações na interface existem pois há limitações no analisador sintático que executa extração da interface do núcleo. Estas limitações são:

- A interface não pode possuir portas de tipos compostos ou tipos definidos em *packages*;
- Os tipos suportados na interface (i.e. nas portas e *generics*) são *std_logic*, *bit*, *std_logic_vector*, *bit_vector*, *integer*, *natural* e *positive*;
- Somente *generics* do tipo *integer*, *natural* ou *positive* podem ser inicializados com valor padrão;
- Não é possível declarar portas do tipo “data : in std_logic_vector(D-1 downto 0)”, pois a constante D não é suportada.

Uma limitação não relacionada à interface é que o núcleo deve ser descrito em VHDL (*soft core*).

7.5.2 Quanto aos Núcleos de Hardware do Tipo Memória

Núcleos do tipo memória não foram avaliados em termos de cobertura de falhas por não possuirmos uma ferramenta de CAD que executa tal função. Por este motivo memórias não são suportadas atualmente pelo sistema. Porém, este suporte pode ser facilmente implementado com a adição de uma biblioteca de algoritmos de teste de memória, conforme apresentado na Seção 4.2.2.

7.5.3 Quanto à Arquitetura de Interconexão dos Núcleos

O único meio de comunicação que foi implementado é um modelo parametrizável de barramento que permite apenas um mestre.

7.5.4 Quanto ao Processador Embarcado

Um processador não foi integrado à biblioteca de núcleos. O processador utilizado para a validação do ambiente é o Nios que está disponível somente na forma de *firm core*. Um dos trabalhos futuros é integrar o processador R8[1] ao sistema. Assim, será possível descrever um SOC por completo.

[1] Processador de 16 bits desenvolvido no GAPH (PUCRS).

7.5.5 Quanto às Ferramentas de CAD

As ferramentas de CAD suportadas são Modelsim 5.5e, Leonardo 2001_1d.45, Flextest 8.2002_1.10, Fastscan 8.2002_1.10 e DFTAdvisor 8.2002_1.10. Outras ferramentas de CAD podem ser facilmente integradas ao ambiente, pois este possui uma interface genérica de configuração e lançamento de ferramentas de terceiros.

7.5.6 Quanto à Lógica Envoltória

- Embora a ferramenta de extração de interface de núcleos suporte diversos tipos de portas, a lógica envoltória suporta somente tipos *std_logic* e *std_logic_vector*. Para o suporte a outros tipos, deve-se acrescentar à ferramenta de geração de lógica envoltória recursos de conversão de dados;
- Os núcleos testados possuem interface com uma porta de entrada e uma de saída e portas de *clock* e *reset*. Outras interfaces devem ser testadas;
- Não foi implementado o suporte às cadeias de varredura.

7.5.7 Quanto às Cadeias de Varredura

- Através da ferramenta DFTAdvisor, a ferramenta desenvolvida é capaz de inserir cadeias de varredura em um núcleo, porém, este recurso não foi testado adequadamente;
- O fluxo de teste não suporta núcleos com cadeias de varredura. Isto ocorre principalmente devido à complexidade do formato do arquivo de padrões de teste do Flextest e Fastscan quando o mesmo possui estas cadeias. Para suportá-las devem ser feitas modificações nos programas de conversão de formato de padrões como o tab2ascii e ascii2vec. O modelo VHDL usado para gerar a assinatura e o gerador de código de teste também deve ser alterado para suportar as cadeias.

7.5.8 Quanto ao Ambiente de Teste

- O ambiente de teste não possui uma ferramenta que auxilie na seleção de polinômios e sementes para um dado núcleo. Esta seleção deve ser realizada pelo usuário.

7.6 Conclusões

Este Capítulo apresentou um protótipo *funcional* de um ambiente integrado de projeto e teste de SOCs baseados em barramento. Esta ferramenta automatiza o processo de geração de código de teste para núcleos de hardware não programáveis. Este ambiente foi concebido como um arcabouço de projeto, capaz de incorporar vários outros módulos e técnicas de teste. Isto porque os principais preceitos empregados no desenvolvimento do mesmo foram facilidade de uso, escalabilidade, flexibilidade e modularidade. Acreditamos que devido a estas características esta ferramenta venha a contribuir no desenvolvimento de outros trabalhos como, por exemplo, o teste de núcleos em redes intra-chip (NOCs).

A **originalidade** desta ferramenta não está no desenvolvimento de novas técnicas ou algoritmos de teste, mas na **integração** de diversas técnicas conhecidas. A complexidade deste ambiente pode ser avaliada, de forma relativa, através do número de linhas de código apresentado na Tabela 10. A Figura 74 na página 76 demonstra que um fluxo de projeto relativamente complexo foi drasticamente simplificado com a criação desta ferramenta de teste.

Tabela 10 – Número de linhas de código do ambiente de teste.

módulos	linhas
GUI de Integração	1100
Outras GUI	600
GUI de Teste	1300
Classes básicas	800
Ferramentas	3200
Templates	1250
Total	**8250**

No próximo capítulo avalia-se o código de teste gerado em termos de tempo de teste e requisitos de memória.

8 AVALIAÇÃO DE TESTE BASEADO EM SOFTWARE

Como apresentado na introdução deste trabalho, na literatura são encontradas referências às vantagens do uso procedimentos de teste implementados em software e algumas avaliações preliminares destas. Entretanto, as conclusões destes trabalhos não permitem avaliar a aplicabilidade desta técnica, principalmente por não avaliarem suas desvantagens. Desta forma, a *terceira contribuição* deste trabalho é a avaliação das principais desvantagens do teste baseado em software (i.e. utilização de área de memória e tempo de teste) utilizando o código de teste gerado pela ferramenta descrita no capítulo 7.

Este capítulo está organizado como segue. Na Seção 8.1 apresentamos os estudos de caso utilizados na avaliação. Nas Seções 8.2 e 8.3 avaliamos a área de memória e o tempo de teste, respectivamente.

8.1 Estudos de Caso

Os estudos de caso utilizados fazem parte do conjunto de *benchmarks* ISCAS85 e ISCAS89. O critério para a escolha deste subconjunto foi o tamanho reduzido dos núcleos, devido ao fato que a ferramenta de teste desenvolvida não possui um recurso que ajude na seleção de polinômios e sementes. Para circuitos maiores é importante um recurso que ajude na seleção de polinômios e sementes, fora do escopo deste trabalho. A abordagem utilizada para selecionar os polinômios foi:

- Seleciona-se diferentes polinômios e sementes, tendo cuidado com a minimização de número de padrões pseudo-aleatórios, pois este valor está diretamente relacionado ao tempo teste;
- Seleciona-se polinômios e sementes somente quando sua contribuição à cobertura de falhas for significativa.

Esta abordagem necessita de diversas execuções de simulações de falhas e de ATPG até chegar a uma configuração de LFSR satisfatória. Estas tarefas consomem muito tempo de execução se for considerado um circuito maior. Isto justifica o uso destes estudos de caso menores.

Dentre os núcleos selecionados citamos: c1908, c1355, c3540 e s1196[1]. As características gerais dos mesmos são apresentadas na Tabela 11. Os atributos apresentados são o número de portas de entrada, de saída e o número de portas lógicas equivalentes para a implementação do núcleo e do núcleo com a lógica envoltória gerada automaticamente. Nota-se que a área da lógica envoltória é elevada. Isto ocorre principalmente pela adição de células de entrada e saída para adaptar a interface do núcleo à largura do barramento (32 bits).

Em relação à lógica envoltória podemos observar que seu uso corresponde à integração do núcleo ao barramento, não sendo específica para teste. Esta área adicional da lógica envoltória pode parecer significativa, porém, deve-se observar que os núcleos são de baixa complexidade, não correspondendo a estudos de caso reais para SOC.

Tabela 11 – Características gerais dos estudos de caso.

Dados gerais dos benchmarks utilizados (área em portas lógicas)				
	entradas	saídas	área núcleo	área núcleo + wrapper
c1908	33	25	286	843
c1355	41	32	271	948
c3540	50	22	756	1499
s1196	14	14	476	854

[1] 'c' denota *benchmark* combinacional e 's' seqüencial.

A Tabela 12 apresenta a configuração de teste utilizada no restante da avaliação dos resultados. Esta tabela apresenta:

- A cobertura de falhas obtida com a configuração de LFSR selecionada na primeira coluna;
- A cobertura de falhas obtida com a configuração de LFSR e os padrões determinísticos na segunda coluna;
- O número total de padrões de teste pseudo-aleatórios e determinísticos nas colunas 3 e 4, respectivamente.

Tabela 12 – Cobertura de falhas e número de padrões de teste.

Cobertura de falhas (CF em %) e número de padrões				
	rand CF	det + rand CF	pat rand	pat det
c1908	67.94	99.52	40	68
c1355	88.89	100,00	66	29
c3540	75.22	97.48	105	184
s1196	53.41	94.57	159	459

É importante salientar que a qualidade da configuração de LFSR utilizada reflete-se nos dados desta tabela de duas formas diferentes: (*i*) Uma boa configuração de LFSR poderia atingir a mesma cobertura de falhas utilizando menos padrões pseudo-aleatórios. Isto implica na redução dos dados da terceira coluna da tabela o qual implica no tempo de teste; (*ii*) Uma boa configuração de LFSR poderia atingir uma cobertura de falhas maior, possivelmente aumentando o número de padrões pseudo-aleatórios, o que diminuiria o número de padrões determinísticos. A redução de padrões determinísticos influenciaria pouco no tempo de teste, pois possivelmente o número de padrões pseudo-aleatórios aumentaria, mas influenciaria diretamente na área de memória utilizada para teste.

8.2 Análise de Área de Memória

A Tabela 13 apresenta diferentes configurações do código de teste. Três configurações principais foram analisadas: a puramente software, o LFSR em software e o MISR em hardware e um somador como gerador de padrões e o MISR também em hardware. Outras duas configurações otimizadas para o núcleo s1196 foram exploradas para destacar o compromisso entre flexibilidade do código de teste e tempo de teste. O otimizado 1 e 2 refletem otimizações, respectivamente, nas implementações representadas pela coluna 1 e 3.

Tabela 13 – Tamanho total do código de teste em bytes.

Tamanho do código em bytes					
	lfsr+misr software	lfsr sw misr hw	somador sw misr hw	otimizado 1	otimizado 2
c1908	564	428	394	-	-
c1355	564	428	394	-	-
c3540	564	428	394	-	-
s1196	564	428	394	246	188

O tamanho do código de teste é independente do *benchmark*. Como esperado, à medida que se remove funcionalidade do software para o hardware o tamanho do código de teste diminui, como apresentado na primeira e segunda coluna. O código de teste também diminui quando o código é simplificado substituindo o LFSR por um somador, conforme colunas 2 e 3.

O núcleo s1196 possui portas de entrada e de saída de 14 bits. Porém, um código de teste genérico deve considerar núcleos com largura maior, inclusive maior que a largura de barramento de 32 bits. Entretanto, a generalidade do código aumenta o tempo de teste de certos núcleos, e.g. s1196 que possui tanto portas de entrada e saída menor que 32 bits. A Tabela 14 apresenta um trecho das funções de teste pseudo-aleatório e determinístico implementadas de forma otimizada (coluna 1 da Tabela 13) e não otimizada (coluna 4 da Tabela 13) para o núcleo s1196 com LFSR e MISR em software.

Tabela 14 – Relação entre tamanho do código e tempo de teste.

	código não otimizado	código otimizado

rand	1. **for**(k=0;k<n_patterns;k++){
	2. // escritas
	3. **for**(l=0;l<core->nWrites;l++) {
	4. lfsr(poly,&value);
	5. **if** (l==core->nWrites-1)
	6. *(core->hwif+LOAD_ADDR) = value;
	7. **else**
	8. *(core->hwif+WRITE_ADDR) = value;
	9. }
	10. // leituras
	11. **for**(l=0;l<core->nReads;l++){
	12. response=*(core->hwif + LOAD_ADDR);
	13. **if** (l==core->nReads-1)
	14. response&=core->mask;
	15. misr(core->misrPoly,response,&sig);
	16. }
	17. }
		1. **for**(k=0;k<n_patterns;k++){
		2. // escrita
		3. lfsr(poly,&value);
		4. *(core->hwif+LOAD_ADDR) = value;
		5. // leitura
		6. response=*(core->hwif+LOAD_ADDR);
		7. response&=core->mask;
		8. misr(core->misrPoly,response,&sig);
		9. }

det	1. **for**(i=1;i<=core->vetDet[0];i++){
	2. // escritas
	3. **for**(l=0;l<core->nWrites;l++,ind++){
	4. value = core->vetDet[ind];
	5. **if** (l==core->nWrites-1)
	6. *(core->hwif+LOAD_ADDR) = value;
	7. **else**
	8. *(core->hwif+WRITE_ADDR) = value;
	9. }
	10. // leituras
	11. **for**(l=0;l<core->nReads;l++){
	12. response=*(core->hwif + LOAD_ADDR);
	13. **if** (l==core->nReads-1)
	14. response&=core->mask;
	15. misr(core->misrPoly,response,&sig);
	16. }
	17. }
		1. **for**(i=1;i<=core->vetDet[0];i++){
		2. // escrita
		3. value = core->vetDet[i];
		4. *(core->hwif+LOAD_ADDR) = value;
		5. // leitura
		6. response=*(core->hwif + LOAD_ADDR);
		7. response&=core->mask;
		8. misr(core->misrPoly,response,&sig);
		9. }

Como é observado na Tabela 14, o código de teste otimizado possui um conjunto de instruções significativamente menor que o código genérico. Deve-se salientar que foram removidos da versão otimizada somente os laços específicos para leitura e escrita em núcleos com entrada e saída maiores que 32 bits, o que não alterou a funcionalidade do núcleo em questão (s1196).

Isto demonstra que pode haver um compromisso entre flexibilidade do código de teste, o que implica em aumento na área de memória, e tempo de teste.

A Tabela 15 apresenta o número de palavras de memória (32 bits) utilizadas para armazenar a configuração do LFSR e os padrões determinísticos. Esta informação é complementar ao número de padrões apresentados na Tabela 12. Por exemplo, o núcleo c1908 utiliza 40 padrões pseudo-aleatórios e 68 padrões determinísticos. A configuração de LFSR destes 40 padrões ocupa 9 palavras, enquanto que os 68 padrões determinísticos ocupam 137 palavras pois o núcleo possui mais de 32 bits de porta de entrada.

Tabela 15 – Tamanho da configuração de LFSR e dos padrões determinísticos em palavras de 32 bits.

Tamanho da parte de dados do código, em bytes		
	pseudo-aleatório	determinístico
c1908	9	137
c1355	17	59
c3540	21	369
s1196	15	460

A Tabela 16 apresenta o número de bytes necessários para armazenar as funções do código de teste na configuração onde o LFSR e o MISR são implementados em software.

Tabela 16 – Tamanho das funções (em bytes) do código de teste quando o LFSR e o MISR estão implementados em software.

Nome da função	bytes
Lfsr	26
Misr	28
Pseudo	238
Det	182
main	90
total	**564**

De acordo com os dados apresentados podemos concluir que o tamanho do código de teste não impõem uma restrição forte ao sistema, pois o código pode ser utilizado para testar diversos núcleos. Porém, a generalidade do código pode aumentar o tempo de teste de certos núcleos. Para estes casos é interessante descrever funções de teste otimizadas. Deve-se notar que o código de teste genérico necessita saber somente o número de portas de entrada e saída de cada núcleo para executar o teste caso-a-caso. Desta forma, haverão apenas quatro possíveis configurações de núcleos: os de entrada e saída maior que 32 bits, os de entrada maior e saída menor (e vice-versa) e os de entrada e saída menor que 32 bits. Em detrimento do tempo de teste, pode ser interessante replicar as funções de teste para estas quatro configurações, uma vez que as mesmas ocupam pouca área de memória conforme demonstrado.

Finalmente, conclui-se que a *área de memória para armazenar o código de teste* não é uma restrição como alegado na literatura [43], pois com poucos bytes de memória armazena-se o programa de teste. Isto porque, como o mesmo código é utilizado para testar diversos núcleos, o tamanho do código de teste permanece inalterado à medida que aumenta o número de núcleos no sistema. Obviamente esta conclusão é aplicada no caso de um código de teste elaborado de forma a aumentar o reuso do mesmo, que é o caso do código gerado pela ferramenta desenvolvida. Desta forma, no que diz respeito ao tamanho do código de teste, teste baseado em software pode ser aplicado em sistemas embarcados com baixo custo de memória.

Já no que diz respeito à *área de memória necessária para armazenar a configuração de LFSR e determinísticos*, é importante selecionar uma boa configuração de LFSR que possa minimizar os padrões determinísticos ou utilizar um algoritmo de compressão de dados. Diversas implementações destas abordagens são encontradas na literatura [26][39][42].

8.3 Análise de Tempo de Teste

Esta Seção avalia em termos de tempo de teste as três configurações de código de teste utilizadas anteriormente e a configuração puramente hardware.

Para avaliar o tempo de teste, utilizamos o ambiente de co-simulação específico para o

processador Nios. Este ambiente suporta co-simulação temporal considerando inclusive elementos como bolhas no *pipeline* do processador, tornando a medida de tempo precisa.

Por outro lado, a configuração puramente hardware foi modelada de forma que o tempo de teste total fosse representado pela equação $Tt = Pp + 10 * Pd$, onde Tt é o tempo de teste, Pp é o número de padrões pseudo-aleatórios e Pd é o número de palavras de memória utilizadas para armazenar os padrões determinísticos.

A Tabela 17 apresenta o número de ciclos de relógio transcorridos entre a geração de um padrão determinístico e um padrão pseudo-aleatório, obtidos por co-simulação. A primeira coluna desta Tabela mostra que um LFSR pode gerar um novo estímulo a cada ciclo de relógio, e devido ao atraso imposto pela leitura da memória de teste os padrões determinísticos são gerados em média a cada 10 ciclos de relógio (arbitragem do meio físico e geração de sinais de controle), resultando na equação $Tt = Pp + 10 * Pd$. Já a geração de padrões a partir do processador embarcado, para os experimentos realizados, consumiu em média 180 ciclos de relógio, tanto para padrões randômicos quanto determinísticos. Estes dados indicam que o tempo de teste para técnicas baseadas em software comparadas com técnicas baseadas em hardware possui um custo adicional de 1 a 2 ordens de grandeza. As colunas 3 a 6 mostram a redução do tempo de teste obtida com otimizações no software e compromisso entre partições hardware e software dos módulos de teste.

Tabela 17 – Tempo, em ciclos de relógio, entre a geração de um padrão de teste pseudo-aleatório ou determinístico.

tempo de teste em ciclos de relógio para gerar um padrão						
	hardware	lfsr+misr software	lfsr sw misr hw	somador sw misr hw	otimizado 1	otimizado 2
rand	1	185	102	73	89	12
det	10	175	43	43	67	22

A Tabela 18 apresenta o tempo total de teste para os três códigos de teste, para a abordagem puramente hardware e para as otimizações do núcleo s1196. As colunas destacadas, segunda e quinta, apresentam os tempos de teste para as implementações puramente software e puramente hardware, respectivamente. A relação entre os tempos destas colunas nos permite responder à questão "quanto custa a implementação de técnicas de teste baseadas em software em termos de tempo de teste?". Este tempo é inferior a duas ordens de grandeza, e está fortemente realcionado ao número de padrões determinísticos. Se este for pequeno, como é o caso do *benchmark* c1908 o custo é mais elevado (54 vezes), porém, para os demais casos temos uma relação média de 25 vezes no tempo de teste.

Tabela 18 – Tempo de teste em ciclos de relógio utilizando diferentes abordagens.

Tempo de teste em ciclos de relógio						
	lfsr+misr sw	lfsr sw misr hw	somador sw misr hw	hardware	otimizado 1	otimizado 2
c1355	**40168**	28034	18817	**1400**	-	-
c1908	**35313**	23246	17552	**646**	-	-
c3540	**94232**	61906	46790	**3785**	-	-
s1196	**110046**	53452	48723	**4749**	45138	12676

Nota-se ainda que o tempo de teste diminui à medida que funcionalidades são passadas para o hardware ou o software é simplificado com a substituição de um LFSR por um somador. Este fato justifica maiores estudos no particionamento hardware/software da estratégia de teste e também justifica a inclusão de rotinas específicas de teste no processador como geradores de padrões ou compactadores de respostas. O compromisso entre generalidade de código de teste e tempo de teste também é novamente evidenciado conforme dados apresentados nas versões de código otimizado

do núcleo s1196.

Os resultados indicam que o tempo de teste é uma restrição importante. Comparando a abordagem puramente hardware com a baseada em somador e MISR em hardware, pode-se notar que a diferença é um pouco mais que uma ordem de grandeza. A abordagem de teste baseada em somador e MISR em hardware pode ser interessante se comparada com um teste baseado em um ATE lento (i.e. baixa largura de banda e baixa freqüência de operação). Exceto este caso, substituição de ATE lento, o tempo de teste da abordagem baseada em software comprometeria o teste de fabricação. Entretanto, esta abordagem é adequada para teste de campo, pois os requisitos de memória são baixos (inferior a 1024 bytes por núcleo) e o tempo de teste está na ordem de mili-segundo (100.000 ciclos para freqüências na ordem de dezenas de MHz).

9 Conclusão

Nos objetivos apresentados na Seção 1.2, exploram-se duas perguntas que este trabalho se propunha a responder:

- Teste baseado em software pode ser utilizado para teste de fabricação?
  Resposta: Teste baseado em software pode ser aplicado em teste de fabricação com ganhos se o mesmo substituir um processo de teste baseado em um ATE com baixa largura de banda e freqüência de operação. Caso contrário, se comparado com BIST implementado em hardware, teste baseado em software aumenta o tempo de teste 10 a 40 vezes, dependendo da implementação do código de teste.
- Esta técnica pode ser aplicada em teste de campo? Quais são as penalidades e vantagens?
  Resposta: Teste baseado em software pode ser aplicado em teste de campo, pois: (*i*) os requisitos de memórias são baixos, em contrariando afirmações constantes na literatura recente [43]. Estes requisitos de memórias podem ser diminuídos ainda mais se algoritmos de compressão de dados e seleção de sementes forem explorados; (*ii*) o tempo de teste é na ordem de mili-segundos, o que possibilita que o sistema seja interrompido para teste a intervalos regulares de acordo com a aplicação do sistema ou no momento da inicialização do mesmo.

O teste baseado em software minimiza as duas desvantagens principais do BIST: degradação do desempenho e aumento da área de hardware. Porém, de acordo com a literatura, o teste baseado em software apresenta a desvantagem de aumentar os requisitos de memória e o tempo de teste. O presente trabalho contribuiu no sentido de *avaliar quantitativamente as desvantagens do teste baseado em software*. Como apresentado nos resultados, o tempo de teste realmente é uma desvantagem significativa quando se utiliza somente teste baseado em software. Porém, foi demonstrado que este tempo de teste pode ser minimizado particionando os módulos de teste em hardware e software. Por outro lado, o aumento dos requisitos de memória de teste não é desvantagem. Um projeto do código de teste visando o reuso pode anular esta desvantagem visto que diferentes módulos do sistema são testados com o mesmo código. É ainda possível reduzir mais as restrições da memória de teste utilizando algoritmos de compressão e seleção de sementes. Porém, estes recursos não foram explorados neste trabalho.

Outra contribuição deste trabalho foi o desenvolvimento de uma *ferramenta de auxílio ao teste baseado em software*. Esta ferramenta integra o fluxo de desenvolvimento de SOC ao fluxo de teste. Como mostrado na Figura 74, página 76, um fluxo relativamente complexo foi drasticamente simplificado com a implementação destas ferramentas. A ferramenta desenvolvida permite incorporar vários outros módulos e técnicas de teste. Isto porque os principais preceitos empregados no desenvolvimento do mesmo foram facilidade de uso, escalabilidade, flexibilidade e modularidade

Uma contribuição adicional deste trabalho foi o desenvolvimento de uma ferramenta de *co-simulação* funcional e geograficamente distribuída. As principais características deste ambiente de co-simulação são: (*i*) lançamento automático dos simuladores; (*ii*) suporte à co-simulação geograficamente distribuída; (*iii*) suporte à VHDL e C (inclusive suas variações como C++ e SystemC); (*iv*) suporte à co-simulação de modelos arquiteturais e RTL; (*v*) flexibilidade do roteador de co-simulação.

Ao fim deste trabalho aproximadamente 10000 linhas de código foram desenvolvidas, sendo 8250 linhas referentes ao ambiente de integração de núcleos e de teste e 1700 linhas do ambiente de co-simulação. As *limitações* das ferramentas desenvolvidas foram destacadas nos Capítulos 5 e 7. Estas limitações correspondem a motivações para trabalhos futuros. Dentre estes trabalhos podemos citar:

1. Avaliação das ferramentas desenvolvidas em um conjunto maior dos *benchmarks* ISCAS 85/89 e ITC99/02 [19][38].
2. Acrescentar na avaliação os critérios acréscimo/redução da área de hardware, impacto na freqüência de operação e dissipação de potência. Porém, é interessante ter uma ferramenta de inserção automática de BIST para simplificar a implementação de teste baseado em hardware.
3. Acrescentar novos recursos nas ferramentas desenvolvidas, como: (*i*) diferentes meios de comunicação para diminuir o tempo de co-simulação; (*ii*) implementar algoritmo de compressão de dados e de seleção de polinômios e sementes para reduzir ainda mais os requisitos de memória do sistema; (*iii*) integrar um processador descrito em VHDL ao ambiente de teste desenvolvido; (*iv*) suportar cadeias de varredura; (*v*) suportar outros meios de comunicação além do barramento; (*vi*) suportar interfaces padrão como OCP; (*vii*) suportar múltiplos processadores.

10 ANEXOS

10.1 Classes Principais da Ferramenta de Teste

As classes apresentadas nesta seção foram utilizadas nas ferramentas apresentadas nas anteriores seções. Estas classes são utilizadas para representar núcleos e SOCs. Como núcleos e SOCs possuem informações semelhantes, foi construída uma classe chamada de *BaseCore*, apresentada na Figura 80, responsável por estas definições comuns. Dentre as informações comuns citamos : nome, descrição, arquivo de projeto e métodos de leitura e escrita do arquivo de projeto.

```
1.  class BaseCore {
2.  protected:
3.      BaseCore() {}
4.      string m_name;
5.      string m_type;
6.      string m_description;
7.      string m_path;
8.  public:
9.      vector<Port*> ports;
10.     virtual ~BaseCore();
11.     virtual bool load(const char *filename)=0;
12.     void setName(const string &name);
13.     const char *name();
14.     void setPath(const char *path);
15.     const char *path();
16.     void addPort(Port *port);
17.     void setDescription(const char *desc);
18.     const char *description();
19.     void setType(const char *type);
20.     const char *type();
21. };
```

A classe *BaseCore* contém definições comuns a classe *Core* e *Soc*. Entre elas estão nome do projeto, descrição, caminho do arquivo, e portas da entidade.

Figura 80 – Classe *BaseCore.*

A partir da classe *BaseCore* deriva-se duas outras classes chamadas de *Core* e *Soc*, respectivamente a Figura 81 e a Figura 82.

```
1.  class Core : public BaseCore {
2.      list<string> m_files;
3.      string m_top;
4.      string m_parametrizador;
5.      string m_instanceName;
6.
7.      // atributos do processador
8.      string m_assembler;
9.      string m_compiler;
10.     string m_debugger;
11. public:
12.     vector<Generic*> generics;
13.     Core() {}
14.     virtual ~Core();
15.     bool load(const char *filename);
16.     void addFile(const char *filename);
17.     void setInstanceName(const char *name);
18.     const char *instanceName();
19.     list<string>& fileList();
20.     void setTop(const char *top);
21.     const char *top();
22.     void addGeneric(Generic *generic);
23.     void setGeneric(const char *gen_name, const char *gen_value);
24.     void setParametrizador(const char *param);
25.     const char *parametrizador();
26.
27.     // métodos do processador
28.     void setAssembler(const char *assembler);
29.     const char *assembler();
30.     void setCompiler(const char *compiler);
31.     const char *compiler();
32.     void setDebugger(const char *debugger);
33.     const char *debugger();
34. };
```

A classe *Core* herda definições básicas da classe *BaseCore* e adiciona definições específicas de núcleos. Desta forma, um núcleo pode ter *generics* (*generics*), pode ser composto por múltiplos arquivos VHDL (*m_files*), possui um nome de instancia (*m_instanceName*), possui uma entidade que é o *top* do núcleo (*m_top*). Se o núcleo for do tipo processador, outros atributos são utilizados como o caminho do arquivo montador de código, do compilador e do depurador (*m_assembler*, *m_compiler* e *m_debugger*).

Figura 81 – Classe *Core*.

```
1.  class Soc : public BaseCore {
2.      string m_commType;
3.      string m_master;
4.      string m_busType;
5.      // ASIC
6.      string m_dftLib;
7.      string m_techLib;
8.      // FPGA
9.      string m_vendor;
10.     string m_family;
11.     string m_device;
12. public:
13.     vector<GenericValue*> genericValue;
14.     virtual ~Soc();
15.     vector<Core*> components;
16.     virtual bool load(const char *filename);
17.     void addGeneric(GenericValue* gen);
18.     GenericValue* findGeneric(string instanceName,string genericName);
19.     Core* findComponent(const char *name);
20.     void setCommType(const char *type);
21.     const char *commType();
22.     void setMaster(const char *master);
23.     const char *master();
24.     void setBusType(const char *type);
25.     const char *busType();
26.     // ASIC
27.     void setDftLib(const char *dftLib);
28.     const char *dftLib();
29.     void setTechLib(const char *techLib);
30.     const char *techLib();
31.     // FPGA
32.     const char *vendor();
33.     void setVendor(const char *vendor);
34.     const char *family();
35.     void setFamily(const char *family);
36.     const char *device();
37.     void setDevice(const char *device);
38. };
```

A classe *Soc* herda definições básicas da classe *BaseCore* e adiciona definições específicas de um SOC. Desta forma, um SOC possui valores de *generics* utilizados nas instancias, uma lista de núcleos, tipos de meios de comunicação utilizados para integrar os núcleos. Um SOC pode ser implementado em FPGA ou ASIC tendo cada um seus atributos e métodos específicos.

Figura 82 – Classe *Soc*.

```
1.  struct Port {
2.      string name;
3.      string direction;
4.      unsigned range_start, range_end;
5.      string type;
6.      string usage;
7.  };
8.
9.  struct Generic {
10.     string name;
11.     string type;
12.     unsigned range_start, range_end;
13.     string value;
14. };
```

O registro *Port* possui os campos referentes ao nome da porta de uma entidade, a direção (*in* ou *out*), a largura da porta (e.g. std_logic_vector(0 to 15)), tipo da porta (e.g. std_logic, bit, integer, etc), e uso ou função da porta (e.g. data, clock, reset_n, address, chipselect, etc). De forma similar o registro *Generic* possui informações referentes aos *generics* da entidade. O único campo diferenciado é o campo de valor do *generic* (e.g. “string”, 57, ”100011”, etc).

Figura 83 – Registros de portas e *generics* de uma entidade.

10.2 Exemplos de Códigos Gerados Automaticamente

10.2.1 Lógica Envoltória

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.
4.  entity wr_x1 is
5.    port(
6.    d          :  in  std_logic_vector(31 downto 0);
7.    addr       :  in  std_logic_vector(5 downto 0);
8.    reset      :  in  std_logic;
9.    clock      :  in  std_logic;
10.   wr         :  in  std_logic;
11.   rd         :  in  std_logic;
12.   cs         :  in  std_logic;
13.   q          :  out std_logic_vector(31 downto 0)
14.   );
15. end wr_x1;
16.
17. architecture beh of wr_x1 is
18.
19.   constant DIN    : natural:=33;
20.   constant RDIN   : natural:=1;
21.   constant DOUT   : natural:=25;
22.   constant RDOUT  : natural:=25;
23.
24.   --***********************
25.   -- ADD THE CORE COMPONENT
26.   --***********************
27.   component c1908 is
28.        port (
29.        PINP : in std_logic_vector(0 to 32);
30.        OUTP : out std_logic_vector(0 to 24)
31.        );
32.   end component;
33.
34.   component IOCell2
35.        generic( N : natural);
36.        port(
37.        data_in      :  in  std_logic_vector(N-1 downto 0);
38.        clock        :  in  std_logic;
39.        reset        :  in  std_logic;
40.        enable       :  in  std_logic;
41.        data_out     :  out std_logic_vector(N-1 downto 0)
42.        );
43.   end component;
44.
45.   --  OPERATION MODES
46.   constant  NORMAL_MODE  :  std_logic_vector(5 downto 0):= "000001";
47.   constant  SERIAL_INTERNAL_MODE  :  std_logic_vector(5 downto 0):= "000011";
48.   constant  SERIAL_EXTERNAL_MODE  :  std_logic_vector(5 downto 0):= "000100";
49.   constant  BYPASS_MODE  :  std_logic_vector(5 downto 0):= "000101";
50.   constant  PARALLEL_INTERNAL_MODE  :  std_logic_vector(5 downto 0):= "000111";
51.   constant  PARALLEL_EXTERNAL_MODE  :  std_logic_vector(5 downto 0):= "000110";
52.
53.   --  functional I/O
54.   signal  core_in       :  std_logic_vector(DIN-1 downto 0);
55.   signal  data_in       :  std_logic_vector(31 downto 0);
56.   signal  core_out      :  std_logic_vector(DOUT-1 downto 0);
57.
58.   --  control signals
59.   signal  load_i        : std_logic;
60.   signal  shift_i       : std_logic;
61.   signal  load_o        : std_logic;
62.   signal  gatedClock    : std_logic;
63.   type  shiftreg_in_fsm  is (WAIT_COND_IN,LOAD_IN,LOAD_IN2,SHIFT_IN,WAIT_STATE,
64.                              LOAD_OUT);
```

```
  signal  EA_IN,PE_IN:  shiftreg_in_fsm;

begin

  data_in  <=  d;
  --***************************
  --  ADD  THE  CORE INSTANTIATION
  --***************************
       x1:   c1908
       port map(
            PINP => core_in,
            OUTP => core_out
       );

  --***************************
  --  I/O CELLS
  --***************************
  input_cell1:IOCell2
       generic map (N => 32)
       port map (
       data_in     =>data_in,
       clock       =>clock,
       reset       =>reset,
       enable      =>load_i,
       data_out    =>core_in(32 downto 1)
  );
  input_cell2:IOCell2
       generic map (N => RDIN)
       port map (
       data_in     =>core_in(32 downto 32),
       clock       =>clock,
       reset       =>reset,
       enable      =>shift_i,
       data_out    =>core_in(0 downto 0)
  );
  output_cell1:IOCell2
       generic map (N => DOUT)
       port map (
       data_in     =>core_out,
       clock       =>clock,
       reset       =>reset,
       enable      =>load_o,
       data_out    =>q(DOUT-1 downto 0)
  );

  --***************************
  -- CONTROL LOGIC
  --***************************
  process (reset, clock)
  begin
       if reset='0' then
       EA_IN <= WAIT_COND_IN;
       elsif clock'event and clock='0' then
            EA_IN <= PE_IN;
       end if;
  end process;

  -- Control of Input Shift Register
  process(cs,rd,wr,EA_IN,addr)
  begin
       case EA_IN is
       when WAIT_COND_IN =>
            if cs='1' and rd = '0' and wr = '1' then
                 case addr is
                 when NORMAL_MODE =>
                      PE_IN <= LOAD_IN;
                 when PARALLEL_EXTERNAL_MODE =>
                      PE_IN <= LOAD_IN2;
                 when others =>
                      PE_IN <= WAIT_COND_IN;
                 end case;
```

```
136.            else
137.                 PE_IN <= WAIT_COND_IN;
138.            end if;
139.       -- load input
140.       when LOAD_IN =>
141.            PE_IN <= WAIT_STATE;
142.       -- wait to process
143.       when WAIT_STATE =>
144.            PE_IN <= LOAD_OUT;
145.       -- load output
146.       when LOAD_OUT =>
147.            PE_IN <= WAIT_COND_IN;
148.       -- cycle1:load input (if input has more than 32 bits)
149.       when LOAD_IN2 =>
150.            PE_IN <= SHIFT_IN;
151.       -- cycle2: shift input (if input has more than 32 bits)
152.       when SHIFT_IN =>
153.            PE_IN <= WAIT_COND_IN;
154.       when others =>
155.            PE_IN <= WAIT_COND_IN;
156.       end case;
157.  end process;
158.
159.  load_i <= '1' when (EA_IN = LOAD_IN or EA_IN = LOAD_IN2)
160.            else '0';
161.  load_o <= '1' when (EA_IN = LOAD_OUT)
162.            else '0';
163.  shift_i <= '1' when (EA_IN = SHIFT_IN)
164.            else '0';
165.  gatedClock <= clock;
166.
167.  end beh;
```

Figura 84 – Arquitetura da lógica envoltória gerada automaticamente.

10.2.2 *Testbench* da Lógica Envoltória

```
1.  library ieee,std;
2.  use ieee.std_logic_1164.all;
3.  use ieee.std_logic_arith.all;
4.  use ieee.std_logic_textio.all;
5.  use std.textio.all;
6.
7.  entity tb_wr_x1 is
8.    generic(
9.    DIN        :    natural:= 33;
10.   RDIN       :    natural:= 1;
11.   DOUT       :    natural:= 25;
12.   RDOUT      :    natural:= 25;
13.   WIDTH      :    integer:= 32;
14.   PATTERNS   :    integer:= 100;
15.   PATTERN_FILE:   string:="pat.txt";
16.   RANDOM     :    boolean:=true
17.   );
18. end entity tb_wr_x1;
19.
20. architecture beh of tb_wr_x1 is
21.
22.   --**********************************
23.   -- GENERIC LFSR
24.   --**********************************
25.   procedure lfsr(constant polynomial : in std_logic_vector;
26.             variable sig: inout std_logic_vector) is
27.   variable i : std_logic_vector(sig'range) := sig;
28.   begin
29.        i := i(i'left-1 downto 0) & i(i'left);
30.        if sig(i'left) = '1' then
31.             i := i xor polynomial;
32.        end if;
33.        sig := i;
34.   end procedure;
35.
36.   --**********************************
37.   -- GENERIC MISR
38.   --**********************************
39.   procedure misr(constant polynomial : in std_logic_vector;
40.        variable input : in std_logic_vector;
41.        variable sig: inout std_logic_vector) is
42.        variable i : std_logic_vector(sig'range) := sig;
43.   begin
44.        i := i(i'left-1 downto 0) & i(i'left);
45.        i := i xor input;
46.        if sig(i'left) = '1' then
47.             i := i xor polynomial;
48.        end if;
49.        sig := i;
50.   end procedure;
51.
52.   constant POLY : std_logic_vector(31 downto 0):=
53.             "00000000000000000000000010101110";
54.   constant SEED : std_logic_vector(31 downto 0):=(others => '1');
55.   file PatternFile : TEXT open READ_MODE is PATTERN_FILE;
56.
57.   --**********************************
58.   -- COMPONENTS
59.   --**********************************
60.   component wr_x1
61.        port(
62.        d          : in  std_logic_vector(31 downto 0);
63.        addr       : in  std_logic_vector(5 downto 0);
64.        reset      : in  std_logic;
65.        clock      : in  std_logic;
66.        wr         : in  std_logic;
67.        rd         : in  std_logic;
68.        cs         : in  std_logic;
69.        q          : out std_logic_vector(31 downto 0)
```

```
70.       );
71.  end component;
72.
73.  component c1908 is
74.       port (
75.       PINP : in  std_logic_vector(0 to 32);
76.       OUTP : out std_logic_vector(0 to 24)
77.       );
78.  end component ;
79.
80.  -- OPERATION MODES
81.  constant  NORMAL_MODE  :  std_logic_vector(5 downto 0):= "000001";
82.  constant  SERIAL_INTERNAL_MODE  :  std_logic_vector(5 downto 0):= "000011";
83.  constant  SERIAL_EXTERNAL_MODE  :  std_logic_vector(5 downto 0):= "000100";
84.  constant  BYPASS_MODE  :  std_logic_vector(5 downto 0):= "000101";
85.  constant  PARALLEL_INTERNAL_MODE  :  std_logic_vector(5 downto 0):= "000111";
86.  constant  PARALLEL_EXTERNAL_MODE  :  std_logic_vector(5 downto 0):= "000110";
87.
88.  --  CORE WITH WRAPPER
89.  signal  d            :  std_logic_vector(31 downto 0):= (others => '0');
90.  signal  q            :  std_logic_vector(31 downto 0):= (others => '0');
91.  signal  rd           :  std_logic:= '0';
92.  signal  wr           :  std_logic:= '0';
93.  signal  core_select  :  std_logic:= '0';
94.  signal  wc           :  std_logic_vector(5 downto 0):= (others => '0');
95.
96.  -- REFERENCE CORE
97.  signal  gold_in      :  std_logic_vector(DIN-1 downto 0):= (others => '0');
98.  signal  gold_out     :  std_logic_vector(DOUT-1 downto 0):= (others => '0');
99.
100. signal  sig_gold_out :  std_logic_vector(DOUT-1 downto 0):= (others => '0');
101. signal  sig_wrapper_out   :  std_logic_vector(DOUT-1 downto 0):= (others => '0');
102. signal  wrapper_out       :  std_logic_vector(31 downto 0):= (others => '0');
103. signal  signature         :  std_logic_vector(31 downto 0):= (others => '1');
104. signal  clock        : std_logic;
105. signal  reset        : std_logic;
106. signal  gatedClock   : std_logic:= '0';
107.
108. begin
109. --**************************
110. -- CLOCK and RESET GENERATION
111. --**************************
112. process
113. begin
114.      clock <= '0', '1' after 15 ns;
115.      wait for 30 ns;
116. end process;
117. reset <= '0', '1' after 50 ns;
118.
119. -- ***************
120. -- UUT
121. -- ***************
122. x1:  wr_x1
123. port map(
124.      d    =>d,
125.      addr =>wc,
126.      reset=>reset,
127.      clock=>clock,
128.      wr   =>wr,
129.      rd   =>rd,
130.      cs   =>core_select,
131.      q    =>q
132. );
133.
134. -- ***************
135. – REFERENCE UNIT
136. -- ***************
137. x2:  c1908
138. port map(
139.      PINP => gold_in,
140.      OUTP => gold_out
```

```
141. );
142.
143. --*****************************************
144. – STIMULUS GENERATION and RESPONSE COMPARATION
145. --*****************************************
146. process
147.      variable  lfsr_data  :  std_logic_vector(31 downto 0):= SEED;
148.      variable  sig        :  std_logic_vector(31 downto 0):= SEED;
149.      variable  misr_in    :  std_logic_vector(31 downto 0);
150.      variable  pattern    :  std_logic_vector(DIN-1 downto 0);
151.      variable  full_data  :  std_logic_vector(DIN-1 downto 0):= (others => '0');
152.      variable  aux_addr   :  std_logic_vector(5 downto 0);
153.      variable  reg_wrapper_out  :  std_logic_vector(DOUT-1 downto 0);
154.      variable  writes,reads  :  integer:= 0;
155.
156.      variable  lineAux     : line;
157.      file    ResponseFile  : TEXT  open  WRITE_MODE is "response.txt";
158.      --*****************************************
159.      --  CONVERT STD_LOGIC_VECTOR TO STRING
160.      --*****************************************
161.      function vector2string(VALUE:in STD_LOGIC_VECTOR) return string is
162.           variable quad: std_logic_vector(0 to 3);
163.           constant ne:   integer := value'length/4;
164.           variable bv:   std_logic_vector(0 to value'length-1) := value;
165.           variable s:    string(1 to ne);
166.      begin
167.           if value'length mod 4 /= 0 then
168.                assert FALSE report
169.                     "Error: Trying to read vector " &
170.                     "with an odd (non multiple of 4) length";
171.                return "";
172.           end if;
173.           for i in 0 to ne-1 loop
174.                quad := bv(4*i to 4*i+3);
175.                case quad is
176.                     when x"0" => s(i+1) := '0';
177.                     when x"1" => s(i+1) := '1';
178.                     when x"2" => s(i+1) := '2';
179.                     when x"3" => s(i+1) := '3';
180.                     when x"4" => s(i+1) := '4';
181.                     when x"5" => s(i+1) := '5';
182.                     when x"6" => s(i+1) := '6';
183.                     when x"7" => s(i+1) := '7';
184.                     when x"8" => s(i+1) := '8';
185.                     when x"9" => s(i+1) := '9';
186.                     when x"A" => s(i+1) := 'A';
187.                     when x"B" => s(i+1) := 'B';
188.                     when x"C" => s(i+1) := 'C';
189.                     when x"D" => s(i+1) := 'D';
190.                     when x"E" => s(i+1) := 'E';
191.                     when x"F" => s(i+1) := 'F';
192.                     when others => null;
193.                end case;
194.           end loop;
195.           return s;
196.      end vector2string;
197.      --*****************************************
198.      -- BUS FUNCTIONAL MODEL
199.      --*****************************************
200.      procedure bfm_write(variable din : in std_logic_vector(31 downto 0);
201.           signal  d             :  out  std_logic_vector(31 downto 0);
202.           variable full_data    :  in   std_logic_vector;
203.           signal  gold_in       :  out  std_logic_vector;
204.           variable addr_in      :  in   std_logic_vector(5 downto 0);
205.           signal  wc            :  out  std_logic_vector(5 downto 0);
206.           signal  wr,rd         :  out std_logic;
207.           signal  core_select   :  out std_logic;
208.           signal  gold_out      :  in   std_logic_vector;
209.           signal  sig_gold_out  :  out  std_logic_vector) is
210.      begin
211.           wc <= addr_in;
```

```
212.            -- simulate combinational delay
213.            core_select <= '1' after 5 ns;
214.            d <= din;
215.            wr <= '1';
216.            rd <= '0';
217.            wait until clock = '1' and clock'event;
218.            -- insert pattern in the reference
219.            if addr_in = NORMAL_MODE then
220.                 gold_in <= full_data;
221.            end if;
222.            wr <= '0';
223.            rd <= '0';
224.            wc <= (others => '0');
225.            -- simulate combinational delay
226.            core_select <= '0' after 5 ns;
227.            d <= (others => '0');
228.            wait until clock = '1' and clock'event;
229.            if addr_in = NORMAL_MODE then
230.                 gatedClock <= '1';
231.                 wait until clock = '0' and clock'event;
232.                 gatedClock <= '0';
233.            end if;
234.            wait until clock = '1' and clock'event;
235.            -- register the reference output in the next cycle
236.            if addr_in = NORMAL_MODE then
237.                 sig_gold_out <= gold_out;
238.            end if;
239.       end procedure;
240.       procedure bfm_read(signal wr,rd : out std_logic;
241.            signal core_select : out std_logic;
242.            signal q           : in  std_logic_vector;
243.            signal wrapper_out : out std_logic_vector)is
244.       begin
245.            core_select <= '1' after 5 ns;
246.            wr <= '0';
247.            rd <= '1';
248.            wait until clock = '1' and clock'event;
249.            wrapper_out <= q;
250.            wr <= '0';
251.            rd <= '0';
252.            wc <= (others => '0');
253.            core_select <= '0' after 5 ns;
254.            wait until clock = '1' and clock'event;
255.       end procedure;
256.  begin
257.       wait until reset = '1';
258.       wait until clock = '1' and clock'event;
259.       for j in 1 to PATTERNS loop
260.            -- ########################################
261.            -- WRITE
262.            -- ########################################
263.            aux_addr := PARALLEL_EXTERNAL_MODE;
264.            writes := DIN/32;
265.            if DIN rem 32 /= 0 then
266.                 writes := writes+1;
267.            end if;
268.            if RANDOM = false then
269.                 readline(PatternFile,lineaux);
270.                 read(lineaux,pattern);
271.            end if;
272.            for i in 0 to writes-1 loop
273.                 -- choose type of pattern
274.                 if RANDOM = true then
275.                      lfsr(POLY,lfsr_data);
276.                 else
277.                      if i = 0 then
278.                           lfsr_data := pattern(RDIN-1 downto 0)
279.                                     &"00000000000000000000000000000000" ;
280.                      else
281.                           lfsr_data := pattern((i*32)+RDIN-1 downto
282.                                                ((i-1)*32)+RDIN);
```

```
283.                     end if;
284.                end if;
285.                -- write pattern used
286.                hwrite(lineaux,lfsr_data);
287.                writeline(ResponseFile,lineaux);
288.                -- the last addr must be
289.                if i = writes-1 then
290.                     aux_addr := NORMAL_MODE;
291.                end if;
292.                -- apply pattern
293.                full_data := lfsr_data & full_data(DIN-1 downto 32);
294.                bfm_write(lfsr_data,d,full_data,gold_in,aux_addr,wc,
295.                wr,rd,core_select,gold_out,sig_gold_out);
296.           end loop;
297.           -- ########################################
298.           -- WAIT TO PROCESS DATA
299.           -- ########################################
300.           for j in 1 to 5 loop
301.                wait until clock = '1' and clock'event;
302.           end loop;
303.           -- ########################################
304.           -- READ
305.           -- ########################################
306.           -- addr dont care to read data
307.           aux_addr := (others => '0');
308.           reads := DOUT/32;
309.           if DOUT rem 32 /= 0 then
310.                reads := reads+1;
311.           end if;
312.           for i in 1 to reads loop
313.                -- read core
314.                bfm_read(wr,rd,core_select,q,wrapper_out);
315.                -- write response to file
316.                hwrite(lineaux,wrapper_out);
317.                writeline(ResponseFile,lineaux);
318.                -- generate signature
319.                misr_in := wrapper_out;
320.                misr(POLY,misr_in,sig);
321.                hwrite(lineaux,sig);
322.                writeline(ResponseFile,lineaux);
323.                signature <= sig;
324.                -- shift response
325.                reg_wrapper_out := wrapper_out(DOUT-1 downto 0);
326.           end loop;
327.           -- ########################################
328.           -- RESULT COMPARISON
329.           -- ########################################
330.           sig_wrapper_out  <= reg_wrapper_out;
331.           wait until clock = '1' and clock'event;
332.           if (sig_wrapper_out(DOUT-1 downto 0)/=sig_gold_out)then
333.                assert(false)
334.                report "ERROR IN PATTERN "&integer'image(j)&" !!!"&lf
335.                severity failure;
336.           else
337.                write(output,"PATTERN "&integer'image(j)&" OK!!!"&lf);
338.           end if;
339.           -- ########################################
340.           -- WAIT TO SEND MORE DATA
341.           -- ########################################
342.           for j in 1 to 5 loop
343.                wait until clock = '1' and clock'event;
344.           end loop;
345.      end loop;
346.      write(output,"THE SIGNATURE GENERATED IS: " &
347.                vector2string(signature) & lf);
348.      -- stop the simulation
349.      assert(false)
350.      report  "END  OF SIMULATION" severity failure;
351. end process;
352. end architecture;
```

Figura 85 – Arquitetura do *testbech* gerado automaticamente.

10.2.3 Interface do Soc

```
1.  library ieee,work;
2.  use ieee.std_logic_1164.all;
3.  use work.soc_bus_pkg.all;
4.
5.  entity example is
6.    port(
7.    reset : in std_logic;
8.    clock : in std_logic
9.    );
10. end example ;
11.
12. architecture beh of example is
13.
14.   constant   N           :  natural:=8;
15.   constant   ADDR_WIDTH  :  natural:=10;
16.   --***********************
17.   -- THE CORES COMPONENT
18.   --***********************
19.   component wr_x1 is
20.        port (
21.        d           : in  std_logic_vector(31 downto 0);
22.        addr        : in  std_logic_vector(5 downto 0);
23.        reset       : in  std_logic;
24.        clock       : in  std_logic;
25.        wr          : in  std_logic;
26.        rd          : in  std_logic;
27.        cs          : in  std_logic;
28.        q           : out std_logic_vector(31 downto 0));
29.   end component;
30.   ...
31.   component wr_x9 is
32.        port (
33.        d           : in  std_logic_vector(31 downto 0);
34.        addr        : in  std_logic_vector(5 downto 0);
35.        reset       : in  std_logic;
36.        clock       : in  std_logic;
37.        wr          : in  std_logic;
38.        rd          : in  std_logic;
39.        cs          : in  std_logic;
40.        q           : out std_logic_vector(31 downto 0));
41.   end component;
42.
43.   --***********************
44.   --  BUS COMPONENT
45.   --***********************
46.   component  soc_bus is
47.        generic(
48.        MODE       : string:= "MUX";
49.        N          : positive:= 2;
50.        ADDR_WIDTH : positive
51.        );
52.        port(
53.        master_in   :  out std_logic_vector(31 downto 0);
54.        master_out  :  in  std_logic_vector(31 downto 0);
55.        master_addr :   in  std_logic_vector(ADDR_WIDTH-1 downto 0);
56.        master_rd   :  in  std_logic;
57.        master_wr   :  in  std_logic;
58.        slave_wr    :  out std_logic_vector(N-1 downto 0);
59.        slave_rd    :  out std_logic_vector(N-1 downto 0);
60.        slave_enb   :  out std_logic_vector(N-1 downto 0);
61.        slave_in    :  out slave_data(N-1 downto 0);
62.        slave_out   :  in  slave_data(N-1 downto 0);
63.        slave_addr  :  out slave_addr(N-1 downto 0)
64.        );
65.   end component;
66.
67.   signal     master_in   : std_logic_vector(31 downto 0);
68.   signal     master_out  : std_logic_vector(31 downto 0);
69.   signal     master_addr      : std_logic_vector(ADDR_WIDTH-1 downto 0);
```

```
70.   signal     master_rd   : std_logic;
71.   signal     master_wr   : std_logic;
72.   signal     slave_wr    : std_logic_vector(N-1 downto 0);
73.   signal     slave_rd    : std_logic_vector(N-1 downto 0);
74.   signal     slave_enb   : std_logic_vector(N-1 downto 0);
75.   signal     slave_in    : slave_data(N-1 downto 0);
76.   signal     slave_out   : slave_data(N-1 downto 0);
77.   signal     slave_addr  : slave_addr(N-1 downto 0);
78.   signal     dummy       : std_logic;
79.
80. begin
81.
82. --**************************
83. --  ADD  THE  CORE INSTANTIATION
84. --**************************
85. x1 : wr_x1
86. port map(
87.   d => slave_in(0),
88.   addr => slave_addr(0)(5 downto 0),
89.   reset => reset,
90.   clock => clock,
91.   wr => slave_wr(0),
92.   rd => slave_rd(0),
93.   cs => slave_enb(0),
94.   q => slave_out(0)
95. );
96. x9 : wr_x9
97. port map(
98.   d => slave_in(7),
99.   addr => slave_addr(7)(5 downto 0),
100.  reset => reset,
101.  clock => clock,
102.  wr => slave_wr(7),
103.  rd => slave_rd(7),
104.  cs => slave_enb(7),
105.  q => slave_out(7)
106.  );
107.
108.  --***************************
109.  -- BUS COMPONENT
110.  --***************************
111.  x1_bus:soc_bus
112.  generic map(
113.       MODE       => "TRI-STATE" ,
114.       N          => N,
115.       ADDR_WIDTH => ADDR_WIDTH)
116.  port map(
117.       master_in   => master_in,
118.       master_out  => master_out,
119.       master_addr => master_addr,
120.       master_rd   => master_rd,
121.       master_wr   => master_wr,
122.       slave_wr    => slave_wr,
123.       slave_rd    => slave_rd,
124.       slave_enb   => slave_enb,
125.       slave_in    => slave_in,
126.       slave_out   => slave_out,
127.       slave_addr  => slave_addr
128.  );
129.
130.  end beh;
```

10.2.4 Código de Teste

```
1. // definições da Nios
2. #include "nios.h"
3. #include<stdio.h>
4.
5. // tipo de definiçoes de teste de um núcleo de hardware
6. struct Core{
7.   char nWrites,nReads;
8.   int mask;
9.   unsigned misrPoly,misrSeed,expectedSig;
10.  volatile  np_usersocket *hwif;
11.  unsigned *lfsrCfg,*vetDet;
12.};
13.
14.// constantes globais
15.const unsigned WRITE_ADDR=6;
16.const unsigned LOAD_ADDR=1;
17.const unsigned nCores=x;
18.
19.// configuração deo LFSR
20.const unsigned lfsrCfg_x1[5]= {...};
21.const unsigned lfsrCfg_xn[5]= {...};
22.
23.// padrões determinísticos
24.const unsigned vetDet_x1[123]= {...};
25.const unsigned vetDet_xn[345]= {...};
26.
27.const struct Core core[x]={
28.{0x2,0x1,0x01FFFFFF,0xAC,0xFFFFFFFF,0x604A2D86,na_c1908,(unsigned*)lfsrCfg_x1,(unsigned*)vetDet_x1},
29.{0x2,0x2,0x0001FFFF,0xAC,0xFFFFFFFF,0xBCBEC2DD,na_s5378,(unsigned*)lfsrCfg_xn,(unsigned*)vetDet_xn}
30.};
31.
32.//  GENERIC LFSR
33.void lfsr(unsigned  polynomial, int *state)
34.{ int  i  = *state;
35.  i <<=1;
36.  if (*state  < 0){ // bit mais significativo em um
37.       i++;
38.       i ^= polynomial;
39.  }
40.  *state  = i;
41.}
42.
43.void  misr(unsigned polynomial, int input, int *state)
44.{ int  i  = *state;
45.  i <<=1;
46.  if (*state  < 0){ // bit mais significativo em um
47.       i++;
48.       i ^= polynomial;
49.  }
50.  i^=input;
51.  *state  = i;
52.}
53.
54.unsigned ApplyPseudoRandomPattern(unsigned lastSig, struct Core *core)
55.{
56.  unsigned i,j,k,poly,n_seed,value,n_patterns;
57.  unsigned ind=1,l,response,sig=lastSig;
58.
59.  for(i=0,ind=1;i<core->lfsrCfg[0];i++){
60.       poly = core->lfsrCfg[ind];
61.       n_seed = core->lfsrCfg[ind+1];
62.       ind+=2;
63.       for(j=0;j<n_seed;j++){
64.            value = core->lfsrCfg[ind];
65.            n_patterns = core->lfsrCfg[ind+1];
66.            ind+=2;
67.            for(k=0;k<n_patterns;k++){
68.                 // escreve
69.                 for(l=0;l<core->nWrites;l++)]{
```

```
70.                   lfsr(poly,&value);
71.                   if (l==core->nWrites-1)
72.                        *(core->hwif  +  LOAD_ADDR)  = value;
73.                   else
74.                        *(core->hwif  +  WRITE_ADDR)  = value;
75.                   printf("%8X ",value);
76.                   }
77.                   //lê
78.                   for(l=0;l<core->nReads;l++){
79.                        response=*(core->hwif   + LOAD_ADDR);
80.                        if (l==core->nReads-1)
81.                             response&=core->mask;
82.                        misr(core->misrPoly,response,&sig);
83.                        printf("%8X ",response);
84.                   }
85.                   printf("%8X\n ",sig);
86.              }
87.         }
88.   }
89.   return sig;
90. }
91.
92. unsigned ApplyDeterministicPattern(unsigned lastSig, struct Core *core)
93. {
94.   unsigned i,sig=lastSig,ind=1;
95.   for(i=1;i<=core->vetDet[0];i++){
96.         // escreve
97.         for(l=0;l<core->nWrites;l++,ind++){
98.              // apply
99.              value = core->vetDet[ind];
100.             if (l==core->nWrites-1)
101.                  *(core->hwif + LOAD_ADDR) = value;
102.             else
103.                  *(core->hwif + WRITE_ADDR) = value;
104.             printf("%8X ",value);
105.        }
106.        //reads
107.        for(l=0;l<core->nReads;l++){
108.             response=*(core->hwif + LOAD_ADDR);
109.             if (l==core->nReads-1)
110.                  response&=core->mask;
111.             misr(core->misrPoly,response,&sig);
112.             printf("%8X ",response);
113.        }
114.        printf("%8X\n ",sig);
115.  }
116.  return sig;
117.  }
118.  /******************
119.  FUNÇÂO PRINCIPAL
120.  ******************/
121.  int main(void)
122.  {     int i;
123.
124.        for(i=0;i<nCores;i++){
125.             if(ApplyDeterministicPattern(ApplyPseudoRandomPattern(
126.                  core[i].misrSeed,&core[i]),&core[i])!= core[i].expectedSig){
127.                  printf("Simulation Error!!!\n");
128.                  return 0;
129.             }
130.        }
131.        printf("Simulation OK!!!\n");
132.        return 1;
133.  }
```

10.3 *Scripts* de Integração com Ferramentas de CAD

Os *scripts* das ferramentas de CAD são uma parte importante do ambiente desenvolvido. Todos os *scripts* apresentados nesta seção são automaticamente personalizados pelo ambiente.

10.3.1 Modelsim

```
1.  onbreak {quit -sim; exit -f}
2.  onerror {quit -sim; exit -f}
3.  vcom -93 /soft/socgen/templates/genRandPatterns.vhd
4.  vsim work.lfsr_gen -gWIDTH=32 -gPROC_WIDTH=32 -gINPUT_FILE= "socs/example/x1.cfg"
5.  add list /lfsr_gen/outlfsr
6.  run -all
7.  write list -window .list /soft/socgen/socs/example/x1.tab
8.  quit -sim
9.  echo SIMULATION SUCESSFULL !!!
10. exit –f
```

Este *script* do Modelsim é responsável por criar os padrões pseudo-aleatórios de acordo com uma configuração de LFSR. Primeiramente deve-se compilar o modelo de LFSR genérico (linha 3). Depois executar o comando de simulação passando alguns valores de *generics* pertinentes ao projeto. O *generic* INPUT_FILE representa a configuração de LFSR que é simulada. A porta de saída do LFSR genérico chamada *outlfsr* é selecionada para sua captura de dados. Por fim, a simulação é executada e o arquivo no formato tabular é salvo.

Figura 86 – *Script* do Modelsim para geração de padrões pseudo-aleatórios.

```
1.  onbreak {quit -sim; exit -f}
2.  onerror {quit -sim; exit -f}
3.  vcom -93 tb_wr_x1.vhd
4.  vsim work.tb_wr_x1 -gPATTERNS=110 -gPATTERN_FILE=x1.vec -gRANDOM=false
5.  run –all
6.  quit –sim
7.  echo SIMULATION SUCESSFULL !!!
8.  exit –f
```

Este *script* é responsável por gerar a assinatura esperada de um dado núcleo. Inicialmente deve-se compilar o *testbench* da lógica envoltória. Depois executar o comando de simulação passando alguns valores de *generics*. O *generic* PATTERN_FILE indica o arquivo no formato vector que contém os padrões de teste. Em seguida a simulação é executada e ao final é impresso no console do simulador a assinatura esperada.

Figura 87 – *Script* do Modelsim para geração da assinatura esperada.

```
1.  onbreak {quit -sim; exit -f}
2.  onerror {quit -sim; exit -f}
3.  if { [file exists work] == 1 } {
4.    vdel -all work
5.  }
6.  vlib work
7.  vmap work work
8.  vcom -93 -explicit socgen/templates/iocell2.vhd
9.  vcom -93 -explicit socgen/cores/Core/Iscas85/c1908/c1908.vhd
10. vcom -93 -explicit socgen/socs/example/wr_x1.vhd
11. vcom -93 -explicit socgen/cores/Core/Iscas85/c1355/c1355.vhd
12. vcom -93 -explicit socgen/socs/example/wr_x2.vhd
13. vcom -93 -explicit socgen/cores/Core/Iscas85/c3540/c3540.vhd
14. vcom -93 -explicit socgen/socs/example/wr_x3.vhd
15. vcom -93 -explicit socgen/cores/Core/Iscas85/c5315/c5315.vhd
16. vcom -93 -explicit socgen/socs/example/wr_x4.vhd
17. vcom -93 -explicit socgen/cores/Core/Iscas85/c6288/c6288.vhd
18. vcom -93 -explicit socgen/socs/example/wr_x5.vhd
19. vcom -93 -explicit socgen/cores/Core/Iscas89/s1238/s1238.vhd
20. vcom -93 -explicit socgen/socs/example/wr_x6.vhd
21. vcom -93 -explicit socgen/cores/Core/Iscas89/s1196/s1196.vhd
22. vcom -93 -explicit socgen/socs/example/wr_x7.vhd
23. vcom -93 -explicit socgen/cores/Core/Iscas89/s953/s953.vhd
24. vcom -93 -explicit socgen/socs/example/wr_x8.vhd
25. vcom -93 -explicit socgen/cores/Core/Iscas89/s5378/s5378.vhd
26. vcom -93 -explicit socgen/socs/example/wr_x9.vhd
27. vcom -93 -explicit socgen/templates/soc_bus_pkg.vhd
28. vcom -93 -explicit socgen/templates/soc_bus.vhd
29. vcom -93 -explicit socgen/socs/example/example.vhd
30. echo COMPILATION SUCESSFULL !!!
31. quit -sim
32. exit -f
```

Este *script* é responsável por compilar um SOC completo. Inicialmente cria-se o projeto do Modelsim com os comandos vdel, vmap e vlib. Depois todos os VHDL são compilados em ordem de hierarquia de projeto. A primeira descrição compilada é a célula da lógica envoltória, depois os núcleos, as suas lógicas envoltórias, o meio de comunicação e, por fim, a interface do SOC.

Figura 88 – Script do Modelsim para compilar um SOC completo.

10.3.2 Leonardo

```
1.  load_library tsmc035_typ
2.  # núcleo 1
3.  read -work work -technology "tsmc035_typ" -format vhdl {
4.    socgen/cores/Core/Iscas85/c1908/c1908.vhd}
5.  pre_optimize .work.c1908.rtl -common_logic -unused_logic -boundary
6.    -xor_comparator_optimize -extract
7.  optimize .work.c1908.rtl -target tsmc035_typ -macro -area -effort quick
8.    -hierarchy auto
9.  write –format edif c1908.edif
10. report_area c1908.rpt -cell_usage
11. report_delay c1908_tim.rpt
12.
13. # sintese da logica envoltória
14. read –work work -technology "tsmc035_typ" -format edif
15.   /soft/socgen/socs/example/c1908.edif
16. read –work work -technology "tsmc035_typ" -format vhdl {
17.   /soft/socgen/templates/iocell2.vhd
18.   /soft/socgen/socs/example/wr_x1.vhd}
19. pre_optimize .work.wr_x1.beh -common_logic -unused_logic -boundary
20.   -xor_comparator_optimize -extract
21. optimize .work.wr_x1.beh -target tsmc035_typ -macro -area -effort quick
22.   -hierarchy auto
23. write –format edif wr_x1.edif
24. report_area wr_x1.rpt -cell_usage
25. report_delay wr_x1_tim.rpt
26.
27. ...
28. # núcleo n
29. ...
30. # síntese do SOC
31. read –work work -technology "tsmc035_typ" -format edif {
32.   /soft/socgen/socs/example/wr_x1.edif
33.   ...
34. }
35. read -work work -technology "tsmc035_typ" -format vhdl {
36.   /soft/socgen/templates/soc_bus_pkg.vhd
37.   /soft/socgen/templates/soc_bus.vhd
38.   /soft/socgen/socs/example/example.vhd
39. }
40. pre_optimize .work.example.beh -common_logic -unused_logic -boundary
41.   -xor_comparator_optimize -extract
42. optimize .work.example.beh -target tsmc035_typ -macro -area -effort quick
43.   -hierarchy auto
44. write –format edif example.edif
45. report_area example.rpt -cell_usage
46. report_delay example_tim.rpt
47. echo SYNTHESIS SUCESSFULL !!!
```

Este *script* executa a síntese lógica de um SOC completo. Inicialmente deve-se sintetizar todos os núcleos e suas lógicas envoltórias. O edif do núcleo é utilizado na simulação de falhas e ATPG. No final os edifs de todas as lógicas envoltórias, o meio de comunicação (barramento) e a interface do SOC são sintetizados para executar síntese lógica de todo o projeto.

Figura 89 – *Script* para síntese lógica de um SOC completo.

10.3.3 Flextest e FastScan

```
1. add black box -auto
2. set system mode fault
3. set fault type stuck
4. set pattern source external socgen/socs/example/x1.ascii
5. add faults -all
6. delete faults -untestable
7. run
8. report statistics
9. exit -D
```

Este *script* é responsável por avaliar a cobertura de falhas dos padrões de teste pseudo-aleatório. As principais funções deste *script* são ler o padrão de teste do LFSR, executar a simulação e apresentar os relatórios com a cobertura de falhas atingida.

Figura 90 – *Script* de avaliação de padrões pseudo-aleatórios.

```
1. add black box -auto
2. set system mode fault
3. set fault type stuck
4. set pattern source external /soft/socgen/socs/example/x1.ascii
5. add faults -all
6. set system mode atpg
7. run               # executa simulação de falhas
8. report statistics
9. write faults /soft/socgen/socs/example/x1.fault_list -rep
10. set pattern source internal
11. reset state
12. load faults /soft/socgen/socs/example/x1.fault_list -retain
13. run              # executa ATPG nas falhas restantes
14. report statistics
15. save patterns /soft/socgen/socs/example/x1_det.ascii -ascii -rep
16. exit –d
```

Este *script* gera padrões determinísticos para as falhas não detectadas pelos padrões pseudo-aleatórios.

Figura 91 – *Script* de geração de padrões determinísticos.

11 REFERÊNCIAS BIBLIOGRÁFICAS

[1] ABRAMOVICI, M.; BREUER, M.A.; FRIEDMAN, A.D. “Digital Systems Testing and Testable Design”. IEEE Press, 1990, 652 p.

[2] AGRAWAL, V.D.; KIME, C.R.; SALUJA, K.K. “A Tutorial on Built-In Self-Test I”. *IEEE Design and Test of Computers*, vol: 10-1, 1993, pp. 73 –82.

[3] AGRAWAL, V.D.; KIME, C.R.; SALUJA, K.K. “A Tutorial on Built-In Self-Test II”. *IEEE Design and Test of Computers*, vol: 10-2, 1993, pp. 69–77.

[4] ALTERA Inc. “Avalon Bus Specification: Reference Manual”. March 2002. ver 1.1, 86 p.

[5] ALTERA Inc. “Nios Embedded Processor Programmer’s Reference Manual”. March 2002. ver 2.1, 124 p.

[6] AMORY, A. M.; MORAES, F. G.; OLIVEIRA, L. A.; CALAZANS, N. V.; HESSEL, F. "A Heterogeneous and Distributed Co-Simulation Environment". In: *Symposium on Integrated Circuits and System Design*, 2002, pp. 115-120.

[7] AMORY, A. M.; MORAES, F. G.; OLIVEIRA, L. A.; HESSEL, F.; CALAZANS, N. V. "Desenvolvimento de um Ambiente de Co-Simulação Distribuído e Heterogêneo". In: *Iberchip*, 2002.

[8] BARDELL, P.H. “Built-In Test for VLSI: Pseudorandom Techniques”. John Wiley & Sons, 1987, 354 p.

[9] BATCHER, K.; PAPACHRISTOU, C. “Instruction Randomization Self Test for Processors Cores”. In: *IEEE VLSI Test Symposium*, 1999. pp 34-40.

[10] BENSO, A.; CHIUSANO, S.; DI NATALE, G.; PRINETTO, P.; BODONI, M.L. “Online and Offline BIST in IP-Core Design”. *IEEE Design and Test of Computers*, vol: 18-5, 2001, pp. 92–99.

[11] BHASKER, J. “A SystemC Primer”. Star Galaxy, 2002, 268 p.

[12] BRGLEZ, F.; BRYAN, D.; KOZMINSKI, K. "Combinational Profiles of Sequential Benchmark Circuits”. In: *IEEE International Symposium on Circuits and Systems*, 1989, pp. 1929-1934.

[13] BRGLEZ, F.; FUJIWARA, H. "A Neutral Netlist of 10 Combinatorial Benchmark Circuits”. In: *IEEE International Symposium on Circuits and Systems*, 1985, pp. 695-698.

[14] BUSHNELL, M. L.; AGRAWAL, V. D. “Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits”. Kluwer Academic, 2000, 690p.

[15] CASSOL, L. J. "Teste de Sistemas Integrados Utilizando Controladores Específicos". Dissertação de mestrado, Universidade Federal do Rio Grande do Sul - UFRGS/CPGEE, Porto Alegre, RS, 2002.

[16] CHAKRABARTY, K. "Optimal Test Access Architectures for System-On-A-Chip”. *ACM Transactions on Design Automation of Electronic Systems*, vol. 6-1, 2001, pp. 26-49.

[17] CHANDRA, A.; CHAKRABARTY, K. “Test Resource Partitioning for SoCs”. *IEEE Design and Test of Computers*, vol: 18-5, 2001, pp. 80–91.

[18] CHEN, L.; DEY, S. "Software-Based Self-Testing Methodology for Processor Cores". *IEEE Transactions on Computer-Aided Designs*, vol: 20-3, 2001, pp. 369-380.

[19] CORNO, F.; REORDA, M.S.; SQUILLERO, G. "RT-Level ITC'99 Benchmarks and First ATPG Results". *IEEE Design and Test of Computers*, vol. 17-3, 2000, pp. 44-53.

[20] CORNO, F.;REORDA, M.S.; SQUILLERO, G. AND VIOLANTE, M. "On the Test of Microprocessor IP Cores". In: *Design, Automation and Test in Europe Conference*, 2001, pp. 209-214.

[21] COTA, E.; CARRO, L.; ORAILOGLU, A.; LUBASZEWSKI, M. "Test Planning and Design Space Exploration in a Core-Based Environment". In: *Design, Automation and Test in Europe Conference*, 2002, pp. 478–485.

[22] DAVID, R. "Random Testing of Digital Circuits: Theory and Applications". Marcel Dekker Inc., 1998, 475 p.

[23] DE MICHELI, G.; *et. al.* "Hardware/Software Co-Design". Kluwer Academic, 1996, 467 p.

[24] DORSCH, R.; WUNDERLICH, H.-J. "Accumulator Based Deterministic BIST". In: *IEEE International Test Conference*, 1998, pp. 412-421.

[25] EDWARDS, S.; LAVAGNO, L.; LEE, E.; SANGIOVANNI-VINCENTELLI, A. "Design of Embedded Systems: Formal Models, Validation and Synthesis". *Proceedings of the IEEE*, vol: 85-3, 1997, pp. 366-390.

[26] FAGOT, C.; GASCUEL, O.; GIRARD, P.; LANDRAULT, C. "On Calculating Efficient LFSR Seeds for Built-In Self Test". In: *European Test Workshop*, 1999, pp. 7 –14.

[27] FRANKLIN, M.; SALUJA, K.K. "Built-In Self-Testing of Random-Access Memories". *Computer*, vol: 23-10, 1990, p. 45–56.

[28] GAJSKI, D.D.; *et. al.* "SpecC: Specification Language and Methodology". Kluwer Academic, 2000, 313 p.

[29] GIRARD, P.; "Survey of Low-power Testing of VLSI Circuits". *IEEE Design and Test of Computers*, vol: 19-3, 2002, pp. 80-90.

[30] GUPTA, S.; RAJSKI, J.; TYSZER, J. "Test Pattern Generation Based on Arithmetic Operations". In: *IEEE/ACM International Conference on Computer-Aided Design*, vol: 6-10, 1994, pp. 117–124.

[31] GUPTA, R.K.; ZORIAN, Y. "Introducing Core-Based System Design". *IEEE Design and Test of Computers*, vol: 13-4, 1997, pp. 15-25.

[32] HELLEBRAND, S.; WUNDERLICH, H. J.; HERTWIG, A. "Mixed-Mode BIST Using Embedded Processors". In: *IEEE International Test Conference*, 1996, pp 195 -204.

[33] HESSEL, F. "Concepção de Sistemas Heterogêneos Multi-Linguagens". Jornada de Atualização em Informática – JAI. XXI Congresso da Sociedade Brasileira de Computação. 2001.

[34] IEEE. "IEEE Standard Test Access Port and Boundary-Scan Architecture". IEEE Standard 1149-1b. IEEE Press, New York. 1994.

[35] IEEE P1450 Web Site. Capturado em: http://grouper.ieee.org/groups/1450/, 2002.

[36] IEEE P1500 Web Site. Capturado em: http://grouper.ieee.org/groups/1500/, 2002.

[37] IYENGAR, V.; CHAKRABARTY, K. "Precedence-Based, Preemptive, and Power-Constrained Test Scheduling for System-on-a-Chip". In: *VLSI Test Symposium*, 2001, pp. 368 –374.

[38] "ITC'02 SOC Test Benchmarks Home Page". Capturado em: http://www.extra.research.philips.com/itc02socbenchm/, 2002.

[39] JAS, A.; TOUBA, N.A. "Using an Embedded Processor for Efficient Deterministic Testing of Systems-on-a-Chip". In: *IEEE International Conference on Computer Design*, 1999, pp. 418-423.

[40] KIM, Y.; *et. al.* "An Integrated Hardware-Software Cosimulation Environment for Heterogeneous Systems Prototyping". In: *Asia and South Pacific Design Automation Conference*, 1995, pp. 101-106.

[41] KLUG, H. P. "Microprocessor Testing by Instruction Sequences Derived from Random Patterns". In: *IEEE International Test Conference*, 1988, pp 73-80.

[42] KRISHNA, C.V.; TOUBA, N.A. "Reducing Test Data Volume Using LFSR Reseeding with Seed Compression". In: *IEEE International Test Conference*, 2002, pp. 321-330.

[43] KRSTIC, A.; WEI-CHENG LAI; KWANG-TING CHENG; CHEN, L.; DEY, S.; "Embedded Software-Based Self-Test for Programmable Core-Based Designs". *IEEE Design and Test of Computers*, vol: 19-4, 2002, pp. 18–27.

[44] LAI, W. C.; CHENG, K. T. "Instruction-Level DFT for Testing Processor and IP Cores in System-on-a-Chip". In: *Design Automation Conference*, 2001. pp. 59 –64.

[45] LARSSON, E; PENG, Z. "An Integrated System-On-Chip Test Framework". In: *Design, Automation and Test in Europe Conference*, 2001, pp. 138-144.

[46] LEMARREC, P.; *et. al.* "Hardware, Software and Mechanical Cosimulation for Automotive Applications". In: *International Workshop on Rapid System Prototyping*, 1998, pp. 202-206.

[47] LUBASZEWSKI, M.; COTA, E. F.; KRUG, M. R. "Concepção de Circuitos Integrados". Capítulo 9: Teste e Projeto Visando o Teste de Circuitos de Sistemas Integrados. Instituto de Informática da UFRGS. Sagra Luzzatto. Porto Alegre, Brasil, 2000.

[48] MARINISSEN, E. J.; ZORIAN, Y. *et. al.* "Towards a Standard for Embedded Core Test: An Example". In: *IEEE International Test Conference*, 1999, pp. 616-626.

[49] MAXWELL, P. C.; AITKEN, R. C.; JOHANSEN, V.; CHIANG, I. "The Effect of Different Test Sets on Quality Level Prediction: When is 80% Better than 90% ?". In: *IEEE International Test Conference*, 1991. pp 358-364.

[50] MELLO, B.A.; WAGNER, F.R. "A Standard Co-Simulation Backbone". In: VLSI-SOC, 2001.

[51] MENTOR GRAPHICS Inc. "Design for Test". Capturado em: http://www.mentor.com/dft/, 2002.

[52] MENTOR GRAPHICS Inc. "Modelsim User Manual". Version 5.5e, 2001.

[53] MENTOR GRAPHICS Inc. "Foreing Language Interface". Version 5.5e, 2001.

[54] MOURAD, S.; ZORIAN, Y. “Principles of Testing Electronic Systems”. John Wiley & Sons, 2000, 348 p.

[55] NEDWAL T. “BIST Implementation Using the CR16B Family”. National Semiconductor. Application Note 001, 2000, 20 p.

[56] NICOLAIDIS, M. “Transparent BIST for RAMs”. In: *IEEE International Test Conference*, 1992, pp. 598-607.

[57] OYAMADA, M.; WAGNER, F.R. "Co-simulation of Embedded Electronic Systems". In: *European Simulation Symposium*, 2000.

[58] PAPACHRISTOU, C.A.; MARTIN, F.; NOURANI, M. “Microprocessor Based Testing for Core-Based System on Chip” In: *Design Automation Conference*, 1999, pp. 586-591.

[59] PARKER, K.P. “The Boundary-Scan Handbook: Analog and Digital”. Kluwer Academic, 2nd edition, 1998, 288 p.

[60] PASCHALIS, A.; GIZOPOULOS, D.; KARNITIS, N.; PSARAKIS, M.; ZORIAN, Y. “Deterministic Software-Based Self-Testing of Embedded Processor Cores”. In: *Design, Automation and Test in Europe*, 2001. pp. 92 –96.

[61] PRADHAN, D.K. “Fault-Tolerant Computer System Design”. Prentice Hall, 1996, 550 p.

[62] RADECKA, K.; RAJSKI, J.; TYSZSER, J. “Arithmetic Built-In Self-Test for DSP Cores”. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16-11, 1997, pp. 1358–1369.

[63] RAJSKI, J.; TYSZER, J. “Test Responses Compaction in Accumulators with Rotate Carry Adders”. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol: 12-4, 1993, pp. 531–539.

[64] RAJSKI, J; TYSZER, J. “Arithmetic Built-In Self-Test for Embedded Systems”. Prentice Hall, 1998, 268 p.

[65] RAJSUMAN, R. “Testing a System-On-A-Chip with Embedded Microprocessor”. In: *IEEE International Test Conference*, 1999, pp. 499–508.

[66] SCHALLER, R. R. "Moore´s Law: Past, Present and Future". *IEEE Spectrum*, vol: 34-6, 1997, pp. 52-59.

[67] SHEN, J.; ABRAHAM, J. A. “Native Mode Functional Test Generation for Processors with Applications to Self-Test and Design Validation”. In: *IEEE International Test Conference*, 1998, pp. 990-999.

[68] SIA - Semiconductor Industry Association, “The National Technology Roadmap for Semiconductors”. Capturado em: http://public.itrs.net/files/1999_SIA_Roadmap/, 1999

[69] STROELE A. P. “Arithmetic Pattern Generation for Built-In Self-Test”. In: *International Conference Computer Design*, 1996. pp. 131–134.

[70] STROELE, A.P "Test Response Compaction Using Arithmetic Functions". In: *VLSI Test Symposium*, 1996, pp. 380 –386.

[71] STROUD, C.E "A Designer’s Guide to Built-In Self-Test". Kluwer Academic, 2002, 319 p.

[72] TEHRANIPOUR, M. H.; NOURANI, M; FAKHRAIE S. M. “Testing Embedded Processor and Memory Cores in SOCs”. In: *VLSI Test Symposium*, 2002.

[73] THATTE, S. M.; ABRAHAM, J. A. “A Methodology for Functional Level Testing of Microprocessors”. In: *International Symposium on Fault-Tolerant Computing*, 1978, pp. 90-95.

[74] TROLLTECH Inc. "QT Home Page". Capturado em: http://www.trolltech.com, 2002.

[75] VAN DE GOOR, A.J. “Using March Tests to Test SRAMs”. *IEEE Design and Test of Computers*, vol: 10-1, 1993, pp. 8–14.

[76] VERHALLEN, T. J. W.; VAN DE GOOR, A. J. “Functional Testing of Modern Microprocessors”. In: *Design Automation Conference*, 1992, pp. 350–354.

[77] VSI Alliance Capturado em: http://www.vsi.org/, 2002.

[78] WILLIAMS, T. W.; BROWN, N. C.; “Defect Level as a Function of Fault Coverage”. *IEEE Transactions on Computers*, vol. C-30, Co. 12, 1981, pp. 987-988.

[79] ZORIAN, Y; IVANOV, A. “An Effective BIST Scheme for ROM’s”. *IEEE Transactions on Computers*, vol: 41-5, 1992, pp. 646-653.

[80] ZORIAN, Y.; MARINISSEN, E.J.; DEY, S. “Testing Embedded-Core-Based System Chips”. *IEEE Computer*, vol: 32-6, 1999, pp. 52–60.