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

Projeto de uma Nova Arquitetura de FPGA para Aplicações BIST e DSP

por

ALEX DIAS GONSALES

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre
em Ciência da Computação

Prof. Dr. Marcelo Lubaszewski
Orientador

Prof. Dr. Luigi Carro
Co-orientador

Porto Alegre, agosto de 2002

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Gonsales, Alex Dias

Projeto de uma Nova Arquitetura de FPGA para Aplicações BIST e DSP / por Alex Dias Gonsales. — Porto Alegre: PPGC da UFRGS, 2002.
108 f.: il.

Dissertação (mestrado) — Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2002. Orientador: Lubaszewski, Marcelo; Co-orientador: Carro, Luigi.

1. Arquiteturas Reconfiguráveis. 2. Teste de Hardware. 3. BIST. 4. Processamento de Sinais Digitais. 5. DSP. I. Lubaszewski, Marcelo. II. Carro, Luigi. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Panizzi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitor Adjunto de Pós-graduação: Prof. Jaime Evaldo Fernsterseifer
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Ao Brasil!

Agradecimentos

Ao povo brasileiro, que financiou meus 20 anos de estudos, desde a primeira série do primeiro grau até este mestrado, e à Universidade Federal do Rio Grande do Sul (UFRGS) e CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), pelo suporte e apoio financeiro concedidos durante o período de mestrado. Esse investimento em mim realizado, e o conhecimento que adquiri, pretendo retribuir, com este trabalho, e dedicando minha vida para que o Brasil seja um país mais justo e melhor para se viver.

Aos meus orientadores, professor Marcelo Lubaszewski e professor Luigi Carro, por me conduzirem e me iniciarem nessa área (microeletrônica) até então desconhecida para mim. Com seus conhecimentos, paciência, disponibilidade e dedicação, sempre me incentivaram e me conduziram para o bom desenvolvimento deste trabalho.

À toda minha família, em especial aos meus pais, Vulmar e Eloni, pela total confiança creditada em todos os momentos, e a quem devo minha formação e grande parte de meus princípios.

Aos meus irmãos, Andersom, Albertine e Andrisom, que nunca exitaram em ajudar-me quando precisei, e pela verdadeira amizade e franqueza que sempre tivemos.

À minha namorada, Ednisse, por todo seu carinho e compreensão, e por suportar minha ausência e cansaço neste período de mestrado.

Aos meus colegas de apartamento, Michel e Mauro, pela amizade que fizemos nesse período de convívio.

Aos professores do PPGC, por todos os ensinamentos e contribuições durante este proveitoso período de estudos.

Aos colegas e amigos do Grupo de Microeletrônica da UFRGS, que muito contribuíram para o desenvolvimento deste trabalho e sempre me auxiliaram, de alguma forma, quando precisei. Meu muito obrigado em especial a, Júlio Mattos, Lisane Brisolara, Érika Cota, Luciano Agostini, Rafael Krapf e Lucas Brusamarello.

À todos os colegas e amigos do Instituto de Informática. Eles são muitos e certamente se fosse citar todos, cometeria a injustiça de esquecer alguém. Aos funcionários do Instituto de Informática, em especial ao Luís Otávio da administração dos laboratórios, à Eliane, Astrogildo e Luciano da portaria, e a todos os funcionários da biblioteca, pela eterna simpatia, disponibilidade e eficiência.

E a todas as pessoas que de alguma forma contribuíram para a minha formação, e portanto, viabilizaram a execução deste trabalho, meus sinceros agradecimentos!

Valeu Galera!!!

Sumário

Lista de Abreviaturas 9
Lista de Figuras 12
Lista de Tabelas 13
Resumo 15
Abstract 17
1 Introdução 19
2 Processamento Digital de Sinais 23
2.1 Filtros FIR 23
2.2 Filtros IIR 26
2.3 Transformada Discreta de Fourier 27
2.4 Considerações Sobre Processamento Digital de Sinais 27
3 Arquiteturas Reconfiguráveis 29
3.1 Conceitos 29
3.1.1 Programabilidade e Configurabilidade 29
3.1.2 Estilos de Reconfiguração 29
3.2 Dispositivos Reconfiguráveis 30
3.3 Tecnologias de Configuração 30
3.4 Arquitetura do Bloco Lógico 32
3.5 Arquitetura de Roteamento 35
3.6 Projeto de FPGAs 36
3.6.1 Complexidade de um Bloco Lógico 36
3.6.2 Flexibilidade de Interconexão 37
3.7 Arquiteturas Reconfiguráveis de Propósito Geral 37
3.7.1 FPGAs Altera 37
3.7.2 FPGAs Xilinx 39
3.8 Arquiteturas Reconfiguráveis para Aplicações Específicas 40
3.8.1 DP-FPGA 40
3.8.2 FPPA 43
3.8.3 PADDI 43
3.9 Considerações Sobre FPGAs 44
4 Teste de Hardware 45
4.1 Métodos de Teste 45
4.2 Modelos Lógicos de Falhas 45
4.3 Simulação de Falhas, Geração e Aplicação do Teste 46
4.4 Projeto Visando o Teste 47
4.4.1 Projeto Visando a Testabilidade 48
4.4.2 Auto-Teste Fora de Funcionamento 49
4.4.3 Auto-Teste em Funcionamento 50
4.5 Teste de Blocos Específicos 50

4.5.1 Teste de Memórias ROM 50
4.5.2 Teste de Memórias RAM 51
4.5.3 Teste de Filtros 54
4.5.4 Teste de FPGAs 55
4.6 Utilização de FPGAs Para Implementação de Teste 56
4.7 Considerações Sobre Algoritmos Para Teste 57

5 Metodologia e Descrição da Arquitetura 59
5.1 Metodologia 59
5.2 Requisitos da Arquitetura 60
5.3 Primeira versão 61
5.4 Segunda Versão 63
5.5 Arquitetura do Bloco Lógico 65
5.6 Configuração do Bloco Lógico 70
5.7 Arquitetura de Roteamento 70
5.8 Descrição VHDL 70
5.9 Implementação de Blocos Básicos 71
5.9.1 Contador 71
5.9.2 Acumulador 72
5.9.3 Comparador 72
5.9.4 Subtrator 74
5.9.5 Registrador-Deslocador 74
5.10 Considerações Sobre a Célula Base do BiFi-FPGA 76

6 Validação da Arquitetura 77
6.1 BIST de ROM 78
6.2 BIST de RAM 80
6.3 Geração de Vetores Pseudo-Aleatórios 81
6.4 Multiplicador 82
6.5 Filtros FIR 87
6.6 Considerações Sobre a Validação da Arquitetura 88

7 Estimativas de Área e Freqüência 89
7.1 Implementação Dedicada 90
7.2 Implementação no BiFi-FPGA 92
7.3 Implementação em FPGA Comercial 93
7.4 Análise Comparativa de Custos 94
7.5 Estimativas de Freqüência 96

8 Conclusões, Contribuições e Trabalhos Futuros 99

Bibliografia 103

Lista de Abreviaturas

ASIC	*Application Specific Integrated Circuit*
BiFi-FPGA	*Bist and Filter FPGA*
BIST	*Built-in Self Test*
CAD	*Computer Aided Design*
CI	Circuito Integrado
CUT	*Circuit Under Test*
DFT	*Discret Fourier Transform*
DP-FPGA	*Datapath FPGA*
DSP	*Digital Signal Processing*
EPROM	*Electrically PROM*
EEPROM	*Electrically Erasable PROM*
FPGA	*Field Programmable Gate Array*
FPPA	*Field Programmable Processor Array*
FIR	*Finite Impulse Response*
FSM	*Finite State Machine*
IIR	*Infinite Impulse Response*
LFSR	*Linear Feed-back Shift Register*
LUT	*Look-up Table*
MISR	*Multiple Input Signature Register*
PAL	*Programmable Array Logic*
PLA	*Programmable Logic Array*
PROM	*Programmable ROM*
RAM	*Random Access Memory*
ROM	*Ready Only Memory*
SOC	*Systems on Chip*
SRAM	*Static RAM*
SSI	*Small Scale Integration*
UE	Unidade de Execução
ULA	Unidade Lógica e Aritmética
VLSI	*Very Large Scale Integration*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very-High Speed Integrated Circuits*

Lista de Figuras

FIGURA 1.1 - BIST dedicado e BIST reconfigurável. 21

FIGURA 2.1 - Grafo do fluxo de sinais de um filtro FIR na forma direta I. 24
FIGURA 2.2 - Grafo do fluxo de sinais de um filtro FIR na forma direta II. 25
FIGURA 2.3 - Filtro FIR com uma unidade multiplica-acumula. 25
FIGURA 2.4 - Filtro FIR utilizando unidades de memória. 25
FIGURA 2.5 - Filtro autorregressivo na forma direta. 26
FIGURA 2.6 - Butterfly utilizada para implementar DFT. Fonte: [PIR 98]. 27

FIGURA 3.1 - Arquitetura típica de um FPGA. 31
FIGURA 3.2 - Célula SRAM controlando um transistor de passagem (a) e um multiplexador (b). 32
FIGURA 3.3 - Bloco lógico do Act-1 da Actel. 33
FIGURA 3.4 - Look-up Table de 3 entradas implementando a função lógica AND. 34
FIGURA 3.5 - Arquitetura de roteamento típica de um FPGA. 36
FIGURA 3.6 - Diagrama de blocos do Flex10k. Fonte:[ALT 2001] 38
FIGURA 3.7 - Elemento Lógico (LE) do Flex10k. Fonte:[ALT 2001] . . . 39
FIGURA 3.8 - Bloco lógico (CLB) do XC4000. Fonte:[XIL 99] 41
FIGURA 3.9 - Bloco lógico (CLB) do XC5200. Fonte:[XIL 98] 41
FIGURA 3.10 - Visão geral do DP-FPGA. 42
FIGURA 3.11 - Multiplexadores sem compartilhamento de bits de configuração (a) e com compartilhamento (b). 43

FIGURA 4.1 - (a) Scan Register, (b) Scan Storage Cell e (c) simbologia do Scan Register. 49
FIGURA 4.2 - Representação genérica de um LFSR. 50
FIGURA 4.3 - Circuito self-checking. 51
FIGURA 4.4 - Teste de ROM. 52
FIGURA 4.5 - Algoritmo March-B (versão 1 bit). 52
FIGURA 4.6 - Versão Marching and Walking (4 bits) do algoritmo March-B. 54
FIGURA 4.7 - Teste de RAM 55
FIGURA 4.8 - Estrutura para auto-teste de FPGA 56

FIGURA 5.1 - Célula básica ou unidade de execução (UE). 61
FIGURA 5.2 - Parte operativa da Unidade de Execução. 62
FIGURA 5.3 - Arquitetura de conexão. 63
FIGURA 5.4 - Visão geral do BiFi-FPGA. 64
FIGURA 5.5 - Interface do bloco lógico. 65
FIGURA 5.6 - Arquitetura do bloco lógico. 67
FIGURA 5.7 - Bit-slice da unidade lógica e aritmética. 68
FIGURA 5.8 - ULA completa (4 bits). 68
FIGURA 5.9 - Bloco ShiftRegLFSR. 69
FIGURA 5.10 - Hierarquia de arquivos. 71
FIGURA 5.11 - Configuração contador: bits 3-0 (a) e demais bits (b). . . 72
FIGURA 5.12 - Configuração acumulador: bits 3-0 (a) e demais bits (b). 73

FIGURA 5.13 - Configuração comparador. 73
FIGURA 5.14 - Configuração subtrator: bits 3-0 (a) e demais bits (b). . . 74
FIGURA 5.15 - Registrador-Deslocador: carga do registrador (a) e bloco funcionando como deslocador (b). 75

FIGURA 6.1 - CUT (*Circuit Under Test*) e BIST associado. 78
FIGURA 6.2 - BIST de ROM. 79
FIGURA 6.3 - Simulação do algoritmo Checksum em ROM sem defeito. . 79
FIGURA 6.4 - Simulação do algoritmo Checksum em ROM com defeito. . 80
FIGURA 6.5 - BIST de RAM. 80
FIGURA 6.6 - Formas de onda da simulação do teste de RAM. 81
FIGURA 6.7 - Três células implementando um LFSR de 12 bits. 82
FIGURA 6.8 - Simulação de um LFSR de 4 bits. 82
FIGURA 6.9 - Simulação de um LFSR de 8 bits. 82
FIGURA 6.10 - Multiplicador paralelo 4x4 (array multiplier). 84
FIGURA 6.11 - Multiplicador paralelo 8x8 (array multiplier). 85
FIGURA 6.12 - Simulação exaustiva do multiplicador paralelo 4x4. 85
FIGURA 6.13 - Detalhe da simulação do multiplicador paralelo 4x4. . . . 85
FIGURA 6.14 - Detalhe da Simulação do multiplicador paralelo 8x8. . . . 86
FIGURA 6.15 - Multiplicador serial 8x8. 86
FIGURA 6.16 - Detalhe da simulação do multiplicador serial 8x8. 86
FIGURA 6.17 - Implementação canônica de um filtro FIR na forma direta I. 87
FIGURA 6.18 - Implementação canônica de um filtro FIR na forma direta II. 87
FIGURA 6.19 - Simulação de um filtro FIR passa alta. 88

FIGURA 7.1 - Bloco Cascade Chain do Flex10k. Fonte:[ALT 2001] . . . 93
FIGURA 7.2 - Bloco Clear Preset Logic do Flex10k. Fonte:[ALT 2001] . 94
FIGURA 7.3 - Estimativa do caminho de maior atraso. 97

Lista de Tabelas

TABELA 4.1 - Primary data backgrounds (4 bits) 53
TABELA 4.2 - Primary data backgrounds (8 bits) 53
TABELA 4.3 - Marching and Walking Data backgrounds (4 bits) 53

TABELA 5.1 - Sinais de dados do bloco lógico 66
TABELA 5.2 - Sinais de controle do bloco lógico. 66
TABELA 5.3 - Controle do cin da ULA. 69
TABELA 5.4 - Possíveis configurações para o bloco ShiftRegLFSR. . . . 69

TABELA 7.1 - Estimativa de área para blocos básicos. 90
TABELA 7.2 - Estimativa de área para implementação dedicada. 92
TABELA 7.3 - Área da célula do BiFi-FPGA. 92
TABELA 7.4 - Estimativa de área para implementação no BiFi-FPGA. . 92
TABELA 7.5 - Estimativa de área para a célula do Flex10k30. 94
TABELA 7.6 - Área gasta para implementação no Flex10k30. 95
TABELA 7.7 - Comparação entre as três abordagens. 96
TABELA 7.8 - Atraso de cout em relação a selB (em ps). 97
TABELA 7.9 - Atraso de d_OutComb em relação a selB (em ps). 97

Resumo

Os sistemas eletrônicos digitais estão sendo cada vez mais utilizados em aplicações de telecomunicações, processamento de voz, instrumentação, biomedicina e multimídia. A maioria dessas aplicações requer algum tipo de processamento de sinal, sendo que essa função normalmente é executada em grande parte por um bloco digital. Além disso, considerando-se os diversos tipos de circuitos existentes num sistema, tais como memórias RAM (*Random Access Memory*) e ROM (*Read Only Memory*), partes operativas e partes de controle complexas, é cada vez mais importante a preocupação com o teste desses sistemas complexos.

O aumento da complexidade dos circuitos a serem testados exige também um aumento na complexidade dos circuitos testadores (teste externo), tornando estes últimos muito caros. Uma alternativa viável é integrar algumas ou todas as funções de teste no próprio chip a ser testado. Por outro lado, essa estratégia pode resultar em um custo proibitivo em termos de área em silício.

É interessante observar, no entanto, que se os testes e a função de processamento de sinal não necessitarem ser executados em paralelo, então é possível utilizar uma única área reconfigurável para realizar essas funções de uma maneira seqüencial.

Logo, este trabalho propõe uma arquitetura reconfigurável otimizada para a implementação desses dois tipos de circuitos (processamento digital de sinais e teste). Com esta abordagem pretende-se ter ganhos de área em relação tanto a uma implementação dedicada (*full-custom*) quanto a uma implementação em dispositivos reconfiguráveis comerciais.

Para validar essas idéias, a arquitetura proposta é descrita em uma linguagem de descrição de hardware, e são mapeados e simulados algoritmos de teste e de processamento de sinais nessa arquitetura. São feitas estimativas da área ocupada pelas três abordagens (dedicada, dispositivo reconfigurável comercial e nova arquitetura proposta), bem como uma análise comparativa entre as mesmas. Também são feitas estimativas de atraso e freqüência máxima de operação.

Palavras-chave: Arquiteturas Reconfiguráveis, FPGA, Teste de Hardware, BIST, Processamento Digital de Sinais, DSP, VHDL.

TITLE: "A NEW FPGA ARCHITECTURE FOR DSP AND BIST APPLICATIONS"

Abstract

Digital electronic systems have been increasingly used in a large spectrum of applications, such as communication, voice processing, instrumentation, biomedicine, and multimedia. Most of these applications require some kind of signal processing. Most of this task is usually performed by a digital block. Moreover, these complex systems are composed of different kinds of circuits, such as RAM (*Random Access Memory*) and ROM (*Read Only Memory*) memories, complex datapaths and control parts. This way, the test of such systems is ever more important.

Likewise, the increasingly complexity of the circuits to be tested requires more complex testers (external test), making the latter more expensive. An approach to address this problem is to embbed the test functions onto the chip to be tested itself. Nevertheless, this approach may bring a prohibitive cost in terms of area on silicon.

However, if the test and the signal processing functions are not required to run in parallel, then it is possible to use the same reconfigurable area to implement these functions one after another.

Thus, this work proposes an optimized reconfigurable architecture to implement this kind of circuits (digital signal processing and test). This approach intends to decrease the occupied area in comparison to a dedicated and also to a comercial reconfigurable device implementation.

To validate these ideas, the proposed architecture is described using a hardware description language and some test and digital signal processing applications are mapped and simulated on this architecture. In this work an estimative of the occupied area by the three approaches (dedicated, comercial reconfigurable device, and the new proposed architecture) as well as a comparison analysis between them are performed. Likewise, a delay estimate is performed and the maximum operation frequency is evaluated.

Keywords: Reconfigurable Architectures, FPGA, Hardware Test, BIST, Digital Signal Processing, DSP, VHDL.

1 Introdução

É cada vez maior a utilização de sistemas eletrônicos digitais nas áreas de telecomunicações, processamento de voz, instrumentação, biomedicina e multimídia. A maioria dessas aplicações requer algum tipo de processamento de sinal. Essa função normalmente é executada por um bloco digital devido às suas vantagens em relação a um bloco analógico, tais como maior resolução, maior imunidade a ruídos, uso de tecnologia mais avançada e maior grau de automação (facilidade de projeto/síntese).

A tarefa de processamento de sinais é muito cara do ponto de vista computacional. Filtros FIR (*Finite Impulse Response*), por exemplo, chegam a representar 80% do poder de computação requerido pela maioria das aplicações de telecomunicações [BEL 98]. Conseqüentemente, os circuitos digitais que atendem esse tipo de aplicação devem cumprir requisitos de alto desempenho [KAV 98, CHE 94]. Logo, a presença de circuitos digitais otimizados para esse tipo de tarefa nos sistemas atuais é muito importante. Da mesma forma, esses sistemas necessitam cada vez mais de blocos de memórias RAM e ROM, blocos operacionais e de controle cada vez maiores e mais complexos.

Verifica-se, portanto, uma crescente demanda por circuitos cada vez mais densos e mais complexos, com altíssimas velocidades e implementados em áreas ativas cada vez mais reduzidas. O cumprimento desses requisitos tem colocado aos projetistas de circuitos integrados constantes desafios. Um dos principais passos dados em direção à solução desses problemas é a integração do maior número de componentes de um sistema, incluindo processador, memória e unidades auxiliares, em um único chip, os chamados Sistemas em Silício (*Systems on Chip - SOC*). Com este paradigma se alcança um alto grau de integração, possibilitando a realização de uma vasta gama de aplicações em um mesmo dispositivo, o que antes necessitaria de uma grande quantidade de CIs (Circuitos Integrados) periféricos interconectados.

Essa alta densidade de integração proporciona diversas melhorias no desempenho geral dos dispositivos, em contrapartida ela traz a desvantagem de uma maior complexidade de projeto e de testabilidade dos sistemas desenvolvidos dessa forma. Logo, o aumento da complexidade dos circuitos a serem testados exige também um aumento na complexidade dos circuitos testadores (teste externo), tornando estes últimos muito caros.

Uma alternativa viável é o auto-teste integrado (*Built-in Self Test - BIST*), o qual consiste em integrar algumas ou todas as funções de teste no próprio chip ou na placa a ser testada [ABR 90, AGR 93, AGR 93a]. Essa técnica tem sido utilizada também para teste interno de blocos embutidos ou núcleos [RAV 99, KIE 98].

Além disso, a utilização de BIST mantém a proteção de propriedade intelectual de um núcleo sem perda da qualidade do teste [MAR 99]. Todavia, esta característica pode vir com um custo proibitivo em termos de área, como mostrado em [COT 99].

Deve-se considerar, também, que devido à diversidade de blocos incorporados em um mesmo CI, como por exemplo memórias, circuitos combinacionais grandes e máquinas de estados complexas, há a necessidade do uso de diferentes técnicas de auto-teste a fim de se verificar o funcionamento correto de todo o sistema. O que se faz normalmente é adicionar ao chip diversos blocos de hardware, sendo cada bloco dedicado exclusivamente ao teste de uma parte do circuito. Conseqüentemente, aumenta-se a área em silício necessária para acomodar todo o sistema juntamente com os blocos de teste correspondentes.

Logo, ao se integrar diversos blocos de teste a um dispositivo pré-existente, deve-se considerar os custos e limite dos recursos de hardware disponíveis. Essa limitação de recursos traz a necessidade da reutilização de módulos. Uma estratégia para a reutilização dos recursos de hardware é a reconfiguração, que consiste em alterar a funcionalidade de um circuito durante o seu intervalo operacional.

É interessante observar que se os testes não necessitarem ser executados em paralelo, então é possível utilizar uma única área reconfigurável para realizar os testes de todos os blocos de uma forma seqüencial [CAR 2000].

Em [CAR 2000] é apresentado um estudo de caso onde essa abordagem é utilizada, ou seja, a utilização de uma área reconfigurável do sistema para a implementação das estruturas de teste. Essa área reconfigurável é utilizada para implementar os controladores BIST de uma forma seqüencial. Tal estratégia tem por objetivo a redução da área no chip dedicada ao teste, uma vez que o mesmo hardware pode ser reconfigurado seqüencialmente para implementar cada um dos procedimentos de teste necessários. Além disso, essa área dedicada pode ser reutilizada para implementar outra função do sistema durante o modo de operação normal do mesmo. No mesmo trabalho, é sugerido que um circuito parcialmente reconfigurável poderia ser muito útil para atacar problemas desse tipo.

Analisando-se as aplicações de processamento digital de sinais (*Digital Signal Processing - DSP*) e BIST, pode-se verificar que as mesmas possuem algumas estruturas de hardware em comum (como será visto adiante), o que sugere a utilização do mesmo hardware reconfigurável para a implementação das duas. Além disso, a utilização de hardware reconfigurável traz outras vantagens, tais como: a possibilidade de se fazer atualizações no sistema sem haver a necessidade de trocar componentes e a prototipação rápida de sistemas.

Os dispositivos reconfiguráveis mais difundidos atualmente são os FPGAs (*Field Programmable Gate Array*) de uso geral [ALT 2002, XIL 2001, ALT 2001, XIL 99]. A utilização desses dispositivos como recurso extra para processamento de sinal tem sido estudada há algum tempo. No entanto, esses FPGAs são destinados para aplicações genéricas, não sendo otimizados para aplicações DSP [CHE 94, KAV 98] ou BIST [REN 2001]. Com isso, têm surgido diversas propostas de FPGAs para aplicações específicas, como por exemplo, os FPGAs otimizados para aplicações intensivas em partes operativas ou funções de processamento digital de sinais [CHE 92, WAN 93, CHE 94, BEL 98, KAV 98].

Com relação a FPGAs otimizados para implementação de BIST, poucos trabalhos foram encontrados. Renovell propõe em [REN 2001] a criação de uma cadeia de varredura (*Implicit Scan Chain*) nos FPGAs clássicos. Usando essa nova arquitetura de FPGA, qualquer circuito seqüencial implementado é implicitamente varrido e testado.

Verifica-se, portanto, que entre as soluções estudadas:

- ou uma nova arquitetura de FPGA é criada para atender somente às otimizações funcionais (DSP) [CHE 94, BEL 98, KAV 98];
- ou um FPGA clássico é modificado para melhorar testabilidade [REN 2001];
- ou um FPGA clássico é utilizado para implementar diferentes soluções de teste [CAR 2000].

Logo, a integração desses importantes pontos de vista desponta como alvo desta dissertação. A partir disso, surge a motivação para o desenvolvimento deste trabalho, que propõe o projeto de uma nova arquitetura reconfigurável (*BiFi-FPGA - BIST and Filter FPGA*) otimizada para a implementação de aplicações DSP e BIST. Este FPGA possui células específicas, otimizadas para implementação de filtros FIR e IIR (*Infinite Impulse Response*), BIST de memórias RAM e ROM, teste pseudo-aleatório, verificador de paridade e ainda a possibilidade de implementar testes baseados em cadeias de varredura (*scan chain*).

Com este novo FPGA pretende-se ter ganhos de área tanto em relação a implementações dedicadas (*full-custom*) quanto implementações em FPGAs comerciais de uso geral. Para ilustrar a idéia da proposta, na fig. 1.1 pode-se verificar a possível redução de área do sistema devido ao agrupamento das funções de teste e do filtro FIR na mesma área reconfigurável.

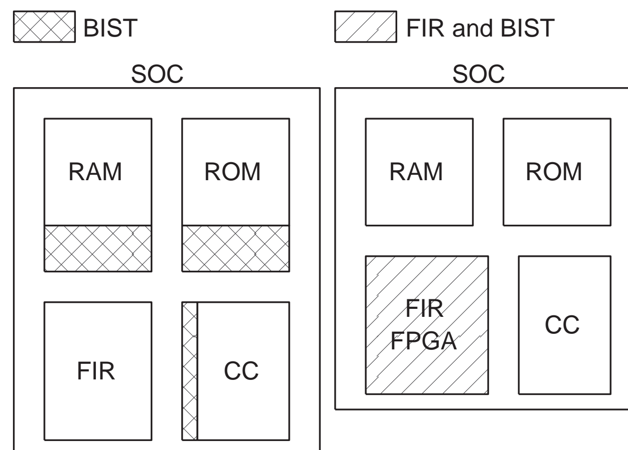

FIGURA 1.1 - BIST dedicado e BIST reconfigurável.

Para validar as idéias propostas neste trabalho, são feitas simulações de aplicações DSP e BIST na arquitetura proposta e compara-se a área gasta em

relação a outras duas abordagens: implementação dedicada e implementação num FPGA comercial de uso geral. São realizadas também, simulações de atraso e freqüência máxima de operação.

Este trabalho inicia abordando, no Capítulo 2, alguns conceitos de processamento digital de sinais, mostrando algumas das técnicas e algoritmos mais utilizados, bem como as arquiteturas de hardware para implementação desses algoritmos.

O Capítulo 3 trata de arquiteturas reconfiguráveis, mostrando exemplos de arquiteturas existentes. Aspectos de projeto dessas arquiteturas, tais como granularidade, arquitetura do bloco lógico, tecnologias de programação e modelos de conexão são tratados nesse capítulo também.

O Capítulo 4 diz respeito a técnicas de projeto visando a testabilidade e o teste de sistemas de hardware. São mostradas diversas arquiteturas BIST, tais como BIST de memórias RAM e ROM, LFSRs, teste com *Scan*, entre outras.

O Capítulo 5 apresenta o FPGA proposto (BiFi-FPGA), mostrando em detalhes a arquitetura do bloco lógico. A primeira fase de validação da proposta é feita, através da implementação de blocos básicos (como somadores e contadores) utilizando instâncias do bloco lógico.

No Capítulo 6 continua-se a validação funcional, neste caso, através do mapeamento de circuitos mais complexos (multiplicadores, filtros, e controladores BIST) no BiFi-FPGA.

No Capítulo 7 são feitas estimativas de área ocupada por esses circuitos bem como é realizada uma análise comparativa com outras duas abordagens (implementação dedicada e num FPGA comercial) para se verificar as vantagens da utilização do BiFi-FPGA. Faz-se uma análise do atraso provocado pelo bloco lógico, estimando-se a sua freqüência máxima de operação.

Finalmente, no Capítulo 8 são delineadas algumas conclusões sobre este trabalho, as principais contribuições que se espera ter atingido com o mesmo e indicações de trabalhos futuros para a continuação deste.

2 Processamento Digital de Sinais

Vários algoritmos de processamento digital de sinais (*Digital Signal Processing - DSP*) estão disponíveis, podendo-se destacar, entre os mais usados, os filtros FIR (*Finite Impulse Response*), os filtros IIR (*Infinite Impulse Response*) e a Transformada Discreta de Fourier (*Discret Fourier Transform - DFT*) [PIR 98, OPP 99, RAB 75].

Neste capítulo são mostradas algumas arquiteturas de hardware utilizadas para implementação desses algoritmos. São consideradas também as estruturas de conexão de portas lógicas necessárias para a implementação dessas arquiteturas.

2.1 Filtros FIR

Filtros podem ser classificados como lineares, não lineares ou adaptativos [PIR 98]. Os filtros lineares são preferíveis sempre que possível, uma vez que seu tratamento pode ser analisado de uma maneira mais simples [PIR 98].

Filtros digitais lineares pertencem a uma classe especial de sistemas, conhecidos como lineares e invariantes no tempo. Uma característica especial desses circuitos é que eles podem ser completamente descritos por sua resposta (*h(n)*)a um impulso. A resposta a um impulso é a reação de um sistema à aplicação de um sinal δ(n) chamado **delta de Dirac**:

$$\begin{aligned} \delta(n) &= 1 && \text{para } n = 0 \\ &= 0 && \text{para } n \neq 0 \end{aligned} \tag{2.1}$$

A reação *y(i)* de um sistema linear discreto invariante no tempo quando aplicada uma seqüência de entrada *x(i)* é determinada pela convolução do sinal de entrada com a resposta do sistema a um impulso:

$$y(i) = h(i) * x(i) \tag{2.2}$$

onde * é o operador de convolução, o qual representa

$$y(i) = \sum_{k=0}^{N-1} h(k)\, x(i-k) \tag{2.3}$$

onde N é o número de coeficientes do filtro.

Uma implementação de filtros FIR, chamada de Forma Direta I (Fig. 2.1), pode ser diretamente derivada a partir da equação 2.3. Ela consiste de elementos somadores, multiplicadores (triângulos) e registradores (representados no diagrama por Z^{-1}).

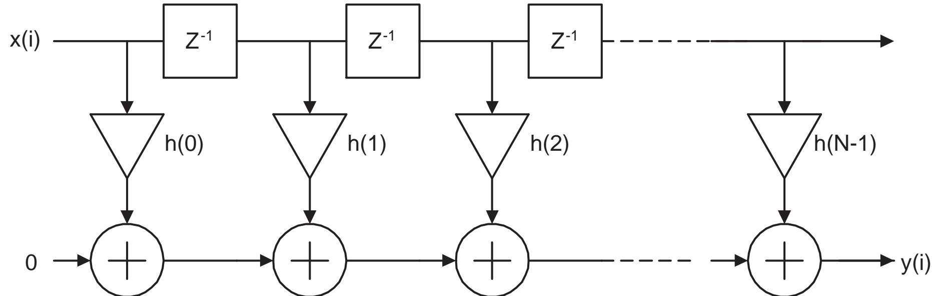

FIGURA 2.1 - Grafo do fluxo de sinais de um filtro FIR na forma direta I.

Outra forma bastante difundida na literatura é a Forma Direta II, a qual é obtida modificando-se a Forma Direta I através da propriedade de associatividade da adição (Fig. 2.2) [PIR 98].

As implementações utilizando-se as formas diretas I e II são adequadas quando se deseja um alto número de operações por ciclo (*throughput*) [PIR 98]. Entretanto, quando a aplicação não exige um alto *throughput*, pode-se utilizar uma implementação serial (Fig. 2.3). Neste caso, utiliza-se apenas uma unidade multiplica-acumula (uma unidade multiplicadora e uma acumuladora). Os coeficientes *h(n)* e as amostras *x(i)* são fornecidas através de duas memórias de acesso cíclico ou registradores de deslocamento. No entanto, segundo [PIR 98], quando o número de coeficientes for muito alto (acima de 256), é aconselhável utilizar-se memórias RAM, com unidades de endereçamento especiais para realizar as leituras cíclicas (Fig. 2.4). Isto se deve ao fato que uma memória cíclica (implementada com *flip-flops*) torna-se muito cara (em termos de área) se comparada com uma memória RAM estática (implementada com *latches*). Por exemplo, considerando-se que um *flip-flop* custa 20 transistores e um *latch* apenas 8 transistores, uma memória cíclica de 256 palavras de 8 bits custaria

$$\begin{aligned} área &= 256 \times 8 \times área_flip_flop \\ &= 256 \times 8 \times 20 \\ &= 40960\, transistores \end{aligned}$$

Já a mesma memória implementada com *latches* custaria

$$\begin{aligned} área &= 256 \times 8 \times área_latch + área_decodificador \\ &= (256 \times 8 \times 8) + (256 \times NAND_8) \\ &= (16384) + (256 \times 16) \\ &= (16384) + (4096) \\ &= 20480\, transistores \end{aligned}$$

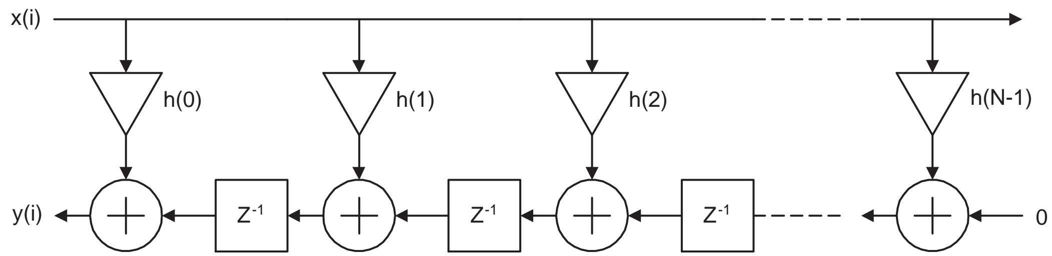

FIGURA 2.2 - Grafo do fluxo de sinais de um filtro FIR na forma direta II.

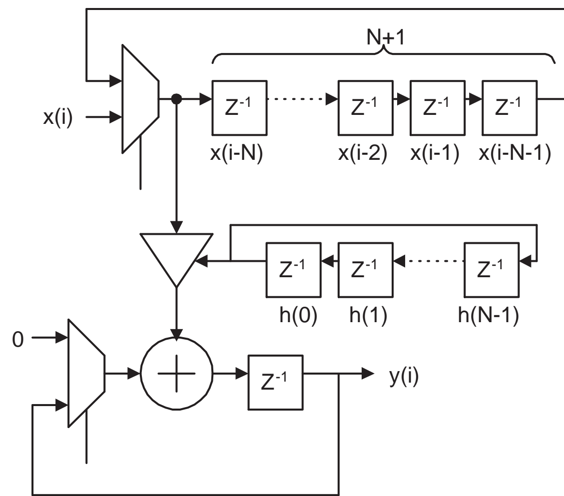

FIGURA 2.3 - Filtro FIR com uma unidade multiplica-acumula.

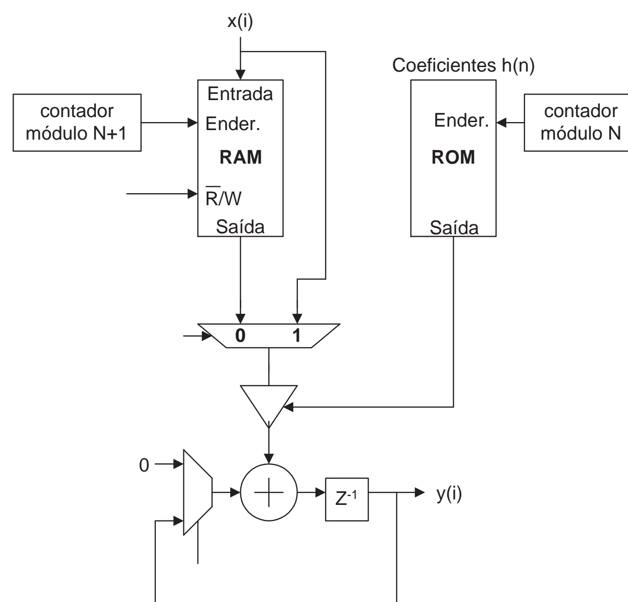

FIGURA 2.4 - Filtro FIR utilizando unidades de memória.

2.2 Filtros IIR

Em geral, filtros recursivos podem cumprir determinadas tarefas com um número de operações mais baixo que seus equivalentes FIR. Quanto menos coeficientes são necessários, menos componentes para sua implementação na forma direta. Em contrapartida, filtros recursivos podem apresentar problemas de estabilidade, devido à sua linearidade imperfeita [PIR 98].

Filtros recursivos podem ser realizados a partir da soma de uma parte puramente recursiva e uma parte não-recursiva. Isto significa que um filtro recursivo geral pode ser implementado por um filtro FIR não recursivo e um filtro IIR (*Infinite Impulse Response*) puramente recursivo (filtro autorregressivo) conectados em série [PIR 98]. Filtros IIR normalmente são representados no domínio tempo por equações de diferenças, como

$$y(i) = \sum_{j=1}^{M} (a(j)y(i-j)) + u(i) \tag{2.4}$$

onde M é o número de unidades somadoras-multiplicadoras (vide Fig. 2.5) e *u(i)* representa o filtro FIR, expresso por

$$u(i) = \sum_{k=0}^{N-1} b(k)x(i-k) \tag{2.5}$$

A Fig. 2.5 mostra a implementação de um filtro autorregressivo (eq. 2.4). É possível verificar que os filtros IIR utilizam as mesmas estruturas dos filtros FIR (somadores, multiplicadores e registradores), tendo como diferença apenas a maneira como esses blocos são conectados.

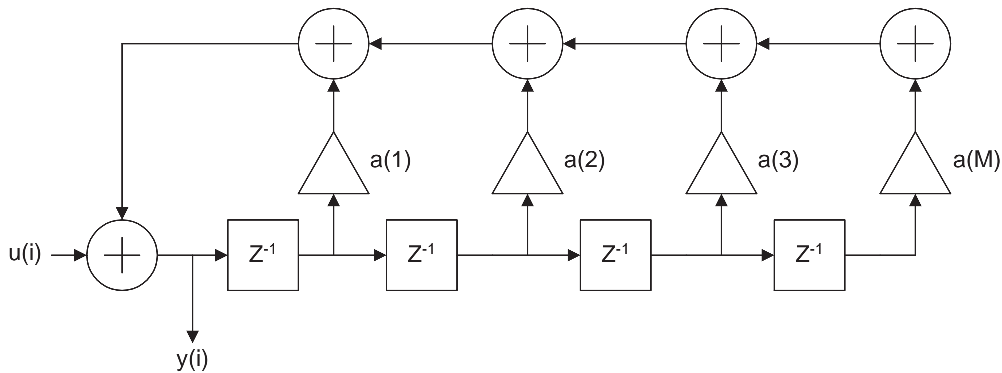

FIGURA 2.5 - Filtro autorregressivo na forma direta.

2.3 Transformada Discreta de Fourier

A Transformada Discreta de Fourier (*Discret Fourier Transform - DFT*) forma a base para muitos métodos de análise espectral. Sua principal utilização é a transformação de um sinal no domínio tempo para o domínio freqüência. A DFT de uma seqüência de comprimento finito é calculada por:

$$X[k] = \sum_{n=0}^{N-1} (x[n] W_N^{KN}) \qquad \text{k = 0, 1, 2, ..., N-1} \tag{2.6}$$

onde $W_N = e^{-j(2\pi/N)}$

Uma técnica muito empregada para a implementação da DFT consiste na utilização de um bloco operacional chamado *butterfly* (Fig. 2.6). Esse bloco tem como entrada dois números complexos e um termo exponencial consistindo de um termo seno e um coseno. Como saída são fornecidos dois números complexos. Para a implementação desse bloco são necessários um somador, um subtrator, um multiplicador e uma memória para armazenamento dos termos seno e coseno [PIR 98].

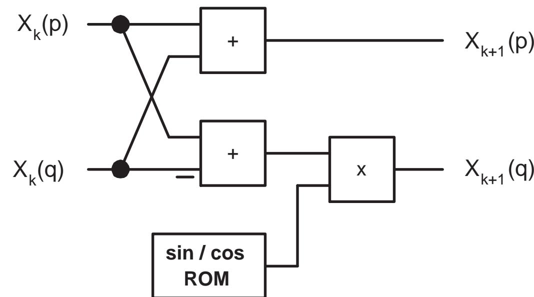

FIGURA 2.6 - Butterfly utilizada para implementar DFT. Fonte: [PIR 98].

2.4 Considerações Sobre Processamento Digital de Sinais

Este capítulo apresentou alguns algoritmos relativos ao processamento digital de sinais, tais como os filtros FIR e IIR bem como a Transfromada Discreta de Fourier. Também foram mostradas algumas estruturas de hardware necessárias para a implementação desses algoritmos. No próximo capítulo são abordados alguns conceitos relativos a arquiteturas reconfiguráveis, sendo apresentadas algumas arquiteturas existentes atualmente.

3 Arquiteturas Reconfiguráveis

As idéias de base da computação reconfigurável surgiram no início da década de 60 com Gerald Estrin [EST 63] *apud* [ADA 98]. Porém, as tecnologias de fabricação de circuitos integrados da época só permitiam a integração de um pequeno número de transistores em um único chip (*Small Scale Integration - SSI*), fazendo com que Estrin concretizasse somente um protótipo aproximado de sua visão [ADA 98].

Hoje em dia, as tecnologias de fabricação permitem a integração de milhões de transistores em um mesmo circuito (*Very Large Scale Integration - VLSI*), possibilitando o desenvolvimento de aplicações de grande porte em um único *chip*, incluindo os dispositivos de lógica reconfigurável.

Neste capítulo são abordados alguns conceitos relativos a arquiteturas reconfiguráveis, bem como são apresentadas algumas arquiteturas existentes e que serviram de subsídio para o desenvolvimento deste trabalho.

3.1 Conceitos

Diversos trabalhos na área de arquiteturas reconfiguráveis têm sido produzidos. No entanto, ainda não há um consenso sobre alguns termos fundamentais, como por exemplo, programabilidade e configurabilidade. Pretende-se, neste capítulo, abordar alguns desses conceitos, segundo a visão de alguns autores.

3.1.1 Programabilidade e Configurabilidade

A distinção entre circuitos programáveis e circuitos configuráveis ainda não é bastante clara, e os dois termos são utilizados como sinônimos muitas vezes. DeHon [DEH 96] define estes conceitos da seguinte forma:

Programável - Refere-se a todas as arquiteturas que reutilizam em larga escala e com grande freqüência uma única parte da lógica ativa do circuito para muitas funções diferentes. O exemplo mais comum de um dispositivo programável é um processador, que pode realizar em sua ULA (Unidade Lógica e Aritmética) uma instrução diferente a cada ciclo de relógio.

Configurável - São aqueles circuitos nos quais a lógica ativa pode realizar diferentes operações, mas cuja função não é alterável de ciclo para ciclo. Uma vez que a "instrução" é configurada, o circuito não tem a capacidade de mudar sua função durante o intervalo denominado operacional. Um FPGA [ALT 2001, XIL 99] é um exemplo de dispositivo configurável.

3.1.2 Estilos de Reconfiguração

Segundo [SHI 96], o estilo de reconfiguração geralmente recai em dois grupos: reconfiguração estática e reconfiguração dinâmica.

Na **reconfiguração estática**, também chamada reconfiguração em tempo de compilação, a operação normal do circuito é interrompida e a configuração do mesmo é efetuada. Após isso, o circuito entra em modo operacional.

Na **reconfiguração dinâmica**, o sistema pode ser reconfigurado sem que se necessite interromper o funcionamento normal do mesmo. Duas abordagens existem para esse tipo de reconfiguração: reconfiguração completa e reconfiguração parcial.

Na **reconfiguração completa** todo o circuito é reprogramado, enquanto que na **reconfiguração parcial** é possível realizar a programação de apenas uma parte do circuito.

É interessante observar que um dispositivo que possibilite reconfiguração dinâmica e parcial permite um alto grau de flexibilidade e exploração (utilização) das capacidades do hardware, já que, com esta abordagem, é possível a reconfiguração de algumas partes do hardware enquanto outras continuam operando normalmente.

3.2 Dispositivos Reconfiguráveis

Existem diversos tipos de dispositivos reconfiguráveis, como por exemplo: PROM (*Programmable ROM*), EPROM (*Erasable PROM*), EEPROM (*Electrically Erasable PROM*), PLA (*Programmable Logic Array*), PAL (*Programmable Array Logic*), CPLD (*Complex Programmable Logic Device*), MPGA (*Masked Programmable Gate Array*) e LPGA (*Laser Programmable Gate Array*). Atualmente, porém, os dispositivos reconfiguráveis mais difundidos no mercado são os FPGAs (*Field Programmable Gate Array*) [LIM 2000, TRI 94, SMI 97, DEH 96].

A arquitetura típica de um FPGA (Fig. 3.1) constitui-se de 3 recursos básicos: blocos lógicos, roteamento e entrada/saída. Os blocos lógicos são configuráveis e estão organizados sob a forma de uma matriz, separados por canais de roteamento. As interconexões entre os blocos lógicos também são configuráveis. Os blocos de entrada/saída estabelecem a comunicação entre o dispositivo e o meio externo.

A seguir serão tratados os principais tópicos relativos a FPGAs, tais como as tecnologias de programação mais utilizadas, tipos de blocos lógicos e modelos de conexões.

3.3 Tecnologias de Configuração

A configuração, também chamada de programação [ROS 93, TRI 94, SMI 97], refere-se à tecnologia utilizada para configurar (programar) um dispositivo reconfigurável. As tecnologias mais utilizadas em FPGAs são SRAMs, anti-fusíveis, EPROMs e EEPROMs.

Um anti-fusível é o contrário de um fusível normal, ou seja, ele é um circuito normalmente aberto até que se force uma corrente (corrente de programação) passar por ele (cerca de 5mA) [SMI 97]. Uma das vantagens desse tipo de

tecnologia é a baixa resistência e capacitância comparados com os transistores de passagem utilizados na tecnologia SRAM. Outra vantagem é o tamanho reduzido dos anti-fusíveis, permitindo um maior número de elementos de programação (anti-fusíveis) no *chip* [TRI 94].

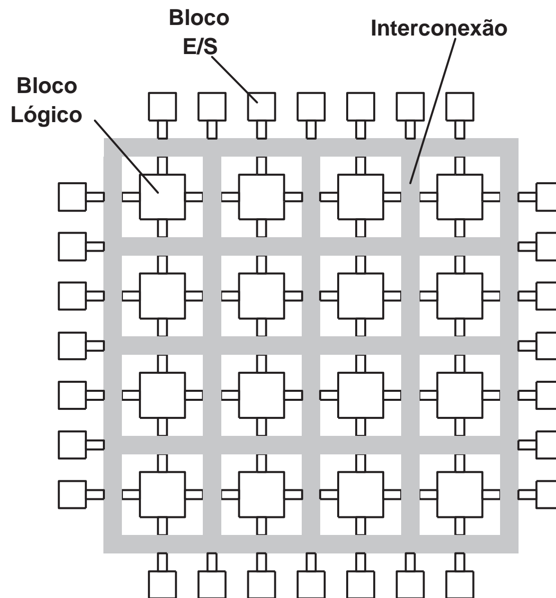

FIGURA 3.1 - Arquitetura típica de um FPGA.

A grande desvantagem é que uma vez programado, um anti-fusível não pode ser reprogramado, logo, o dispositivo é programável somente uma vez (*OTP - one-time programmable*) [SMI 97].

As tecnologias EPROM (*Electrically Programmable Read-Only Memory*) e EEPROM (*Electrically Erasable Programmable Read-Only Memory*) permitem o apagamento e, na seqüência, a reconfiguração do dispositivo via aplicação de luz ultra violeta (EPROM) ou com um sinal elétrico (EEPROM). As células EPROM são quase tão pequenas quanto os anti-fusíveis [SMI 97] e têm a vantagem de poder ser reprogramadas. Como desvantagens, têm-se a alta resistência de um transistor EPROM/EEPROM e um alto consumo de energia, devido à potência estática proveniente da utilização de transistores *pull-up* [ROS 93].

Na tecnologia SRAM (*Static RAM*), utilizam-se células de memórias RAM estáticas para armazenamento da configuração do dispositivo. Essas células normalmente controlam transistores de passagem ou multiplexadores (Fig. 3.2). Quando o valor lógico 1 (um) está armazenado na célula SRAM, o transistor de passagem funciona como uma chave fechada e pode ser usado para conectar dois segmentos de fios. Quando um 0 (zero) estiver armazenado, a chave está aberta e o transistor apresenta uma alta resistência entre os dois segmentos de fio. Para o caso do multiplexador, a SRAM conectada às linhas de seleção do MUX controla qual entrada será conectada à saída.

A maior desvantagem desta tecnologia é a área ocupada. Uma célula SRAM gasta pelo menos 5 transistores para ser implementada, e mais um transistor no mínimo para servir como chave programável [ROS 93]. Além disso, deve-se manter o chip alimentado, devido à volatilidade das memórias RAMs e, deve-se manter também uma memória não-volátil (ROM por exemplo) no circuito, para prover a configuração sempre que a alimentação for reativada.

No entanto, a grande vantagem desta tecnologia é a rápida reconfiguração do dispositivo, e a necessidade apenas de tecnologia padrão de processo de circuitos integrados para sua concepção [ROS 93]. Dispositivos FLEX10k da Altera [ALT 2001] e XC4000 da Xilinx [XIL 99] utilizam este tipo de tecnologia de programação.

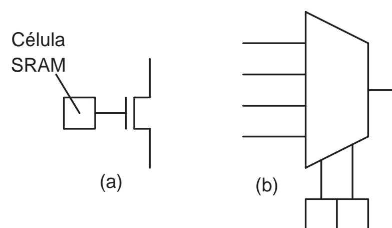

FIGURA 3.2 - Célula SRAM controlando um transistor de passagem (a) e um multiplexador (b).

3.4 Arquitetura do Bloco Lógico

Os blocos lógicos dos FPGAs atuais diferem grandemente em sua granularidade e arquitetura. O conceito de granularidade pode ser definido de vários modos, tais como: o número de funções booleanas que o bloco lógico pode implementar, o número de portas NANDs de duas entradas equivalentes, o número total de transistores ou o número de entradas e saídas do bloco [ROS 93]. Nesse sentido, um bloco lógico pode ser tão simples quanto um par de transistores, no caso dos FPGAs Crosspoint [ROS 93], ou tão complexo quanto um processador, como em [NUS 98].

Tipicamente, um bloco é capaz de implementar diferentes tipos de funções lógicas combinacionais ou seqüenciais [SMI 97, TRI 94]. Os tipos de blocos lógicos mais empregados são [ROS 93]:

- Par de transistores;
- Portas lógicas básicas, tais como NANDs de duas entradas;
- Multiplexadores;
- Look-up Tables (LUT);
- Estruturas AND-OR com grande número de entradas.

Exemplos de blocos lógicos de granularidade fina incluem os FPGAs Crosspoint e Plessey [ROS 93]. Os FPGAs Crosspoint utilizam pares de transistores como constituintes de seus blocos lógicos. Já nos FPGAs Plessey, o bloco lógico é composto por uma porta lógica NAND de duas entradas, um multiplexador e um *latch*.

A vantagem de blocos de granularidade fina é que eles podem ser utilizados mais eficientemente, e são facilmente mapeados pelas ferramentas de CAD. Em contrapartida, é necessário um grande número de segmentos de linha e chaves programáveis para o roteamento, aumentando a área e o atraso. Com isso, FPGAs de granularidade fina, em geral, são mais lentos e possuem menor **densidade** (quantidade de lógica utilizável por unidade de área do chip) que os FPGAs de granularidade grossa [ROS 93].

Os blocos lógicos dos FPGAs Actel, Quicklogic, Altera e Xilinx podem ser classificados como blocos de granularidade grossa [ROS 93]. Os FPGAs Act-1 [ACT 96] da Actel utilizam 3 multiplexadores 2x1 e uma porta OR para implementação dos blocos lógicos (Fig. 3.3). Possuem 8 entradas e uma saída e podem implementar até 702 funções lógicas . Os FPGAs Quicklogic utilizam 3 multiplexadores, um *latch* e algumas portas AND em seu bloco lógico [ROS 93].

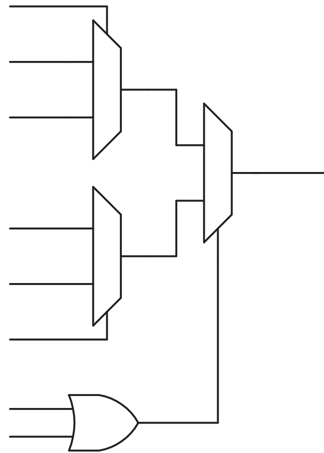

FIGURA 3.3 - Bloco lógico do Act-1 da Actel.

Os blocos lógicos baseados em multiplexadores têm a vantagem de proporcionar um alto grau de funcionalidade para um número de transistores relativamente pequeno. Em contrapartida, é necessário um grande número de pinos de entradas (8 no caso do Act-1 e 14 no caso do Quicklogic). Isto demanda mais recursos de roteamento, logo, estes blocos são mais adequados para FPGAs que utilizam chaves de programação pequenas, tais como os anti-fusíveis [ROS 93].

Outra estrutura muito utilizada para implementação dos blocos lógicos são as LUTs (*Look-up Table*). Uma k-LUT (LUT com k entradas) é implementada por uma memória 2^k x 1, onde k é o número de linhas de endereço da memória que funcionam como entradas da função lógica. A saída da memória fornece o valor da

função lógica. A Fig. 3.4 mostra uma LUT de 3 entradas implementando a função lógica Out = I2 AND I1 AND I0.

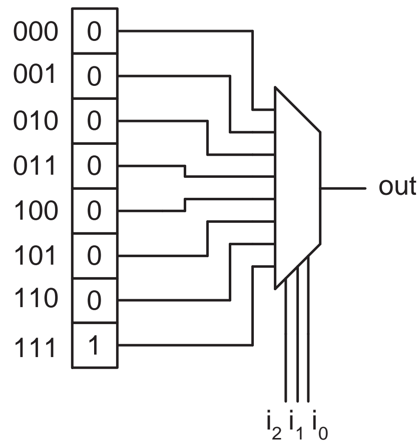

FIGURA 3.4 - Look-up Table de 3 entradas implementando a função lógica AND.

A vantagem das LUTs é que elas proporcionam uma alta funcionalidade para o bloco lógico, visto que uma k-LUT pode implementar qualquer função de k entradas, e existem um total de 2^{2^k} funções possíveis.

Os FPGAs XC4000 da Xilinx [XIL 99] e FLEX10k da Altera [ALT 2001] utilizam LUTs como componentes básicos para implementação das funções lógicas. O bloco lógico do XC4000 contém 2 LUTs de 4 entradas, 1 LUT de 3 entradas, alguns multiplexadores e dois *flip-flops*. Já o do FLEX10k contém uma LUT de 4 entradas, multiplexadores e um *flip-flop*.

Outro tipo de bloco lógico utilizado são as matrizes AND-OR programáveis. Nesse caso, uma matriz de transistores é utilizada para formar uma lógica *wired-AND*. Os transistores podem ser programados para conectar as entradas do bloco às entradas de uma porta AND com grande número de entradas (alto *fan-in*, 20 a 100) [ROS 93]. A saída da porta AND é conectada a uma porta OR de 3 a 8 entradas (*Programmable Array Logic - PAL*) [ROS 93]. Quando as entradas da porta OR também podem ser programadas, a estrutura é denominada *Programmable Logic Array* (PLA) [TRI 94]. Essas estruturas permitem a fácil implementação de lógica baseada em soma de produtos.

A vantagem desse tipo de bloco é que a porta lógica AND com grande número de entradas pode ser usada para implementar funções lógicas com poucos níveis de blocos lógicos, reduzindo a necessidade de conexões programáveis. Porém, pode ser difícil a utilização eficiente de todas as entradas de todas as portas, resultando em perda de densidade. No entanto, essa perda não é tão alta como parece, devido à alta densidade proporcionada pela lógica *wired-AND* da arquitetura [ROS 93]. Outra desvantagem é o alto consumo de energia devido à utilização de dispositivos *pull-up* [ROS 93]. Os dispositivos MAX 5000 e MAX 7000 da Altera [ALT 2000] utilizam essa estratégia de implementação.

A maioria dos blocos lógicos descritos anteriormente inclui alguma estrutura de lógica seqüencial. Os dispositivos Xilinx possuem 2 *flip-flops* em seu bloco lógico, o FLEX10k da Altera possui um *flip-flop* e os FPGAs Plessey possuem um *latch*.

Quando o bloco lógico não possuir um *flip-flop* ou *latch*, ainda assim poderá ser implementada uma lógica seqüencial utilizando-se os blocos combinacionais, desde que se possa fazer uma realimentação na arquitetura utilizando-se os recursos de roteamento [ROS 93]. Esse é o caso dos dispositivos Act-1 da Actel [ACT 96], por exemplo, que só possuem estruturas combinacionais em seu bloco lógico (Fig. 3.3).

A integração de *flip-flops* e *latches* no interior do bloco lógico permite uma implementação mais eficiente de lógica seqüencial. No entanto, quando o bloco for utilizado para implementar somente lógica combinacional, esses recursos adicionais (*flip-flops* e *latches*) podem representar um desperdício de área.

Além desses blocos, outras estratégias podem ser adotadas para a implementação do bloco lógico, conforme será visto na seção 3.8 que mostra algumas arquiteturas reconfiguráveis para aplicações específicas.

3.5 Arquitetura de Roteamento

A arquitetura de roteamento de um FPGA corresponde à maneira com que as chaves programáveis e os segmentos de fio são posicionados para permitir a interconexão programável dos blocos lógicos [ROS 93]. Um modelo típico de FPGAs constitui-se de uma matriz de blocos lógicos separados por canais de roteamento.

Como mostrado na fig. 3.5, esse modelo contém duas estruturas básicas, o bloco de conexão e o bloco de chaveamento (*switch block*). O bloco de conexão é responsável pela conectividade das entradas e saídas do bloco lógico aos segmentos de fios do canal de roteamento. Já o bloco de chaveamento possibilita a conexão entre os segmentos de fio do canal de roteamento.

Um segmento de fio é um fio sem interrupções. Tipicamente, nas extremidades de um segmento são conectadas as chaves programáveis. Uma trilha (*track*) é uma seqüência de um ou mais segmentos em uma linha e, um canal de roteamento é um grupo de trilhas paralelas [ROS 93]. A Fig. 3.5 ilustra, também, um exemplo de como se poderia conectar os blocos lógicos *BL0* e *BL4* utilizando as estruturas dessa arquitetura de roteamento.

Outras arquiteturas podem ser encontradas na literatura [SMI 97, TRI 94]. As seções 3.7.1 e 3.8.1 mostram mais duas arquiteturas de roteamento, uma utilizada num FPGA de uso geral e outra utilizada num FPGA de aplicação específica.

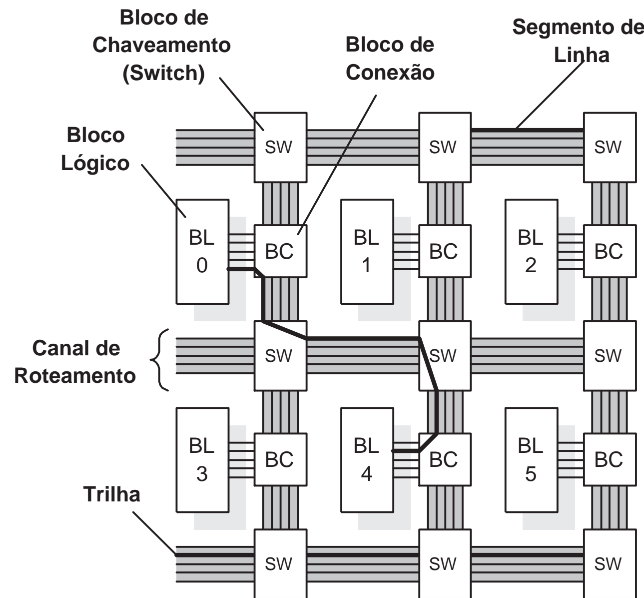

FIGURA 3.5 - Arquitetura de roteamento típica de um FPGA.

3.6 Projeto de FPGAs

Durante o projeto de um FPGA, diversos parâmetros devem ser considerados, tais como a freqüência máxima de operação, área gasta para sua implementação, número de pinos de entrada e saída, potência dissipada, entre outros. Neste trabalho, serão considerados os parâmetros funcionalidade ou complexidade de um bloco lógico e a flexibilidade de interconexão entre blocos, visto que esses influem diretamente na área e no desempenho geral do FPGA.

3.6.1 Complexidade de um Bloco Lógico

Segundo [BRO 96], o primeiro estudo publicado sobre arquiteturas de FPGAs foi a respeito da complexidade dos blocos lógicos. A idéia básica era estudar qual a quantidade de lógica que um único bloco deveria ser capaz de implementar. Por razões práticas, esses estudos assumiram que um bloco lógico era implementado por uma LUT e a complexidade do bloco foi definida como sendo o número de entradas K dessa LUT.

Quanto mais entradas uma LUT possuir, mais complexa será a lógica possível de se implementar com ela. Isto significa que será necessário um menor número de blocos lógicos para se implementar um determinado circuito. Menos blocos lógicos também significa menos conexões entre blocos e conseqüentemente menor área gasta para roteamento. No entanto, a complexidade de uma LUT cresce exponencialmente com o número de entradas K, aumentando significativamente o tamanho do bloco lógico e tornando-se impraticável a sua implementação [BET 98]. Uma solução intermediária é o agrupamento de várias LUTs, conectando-as através de roteamento local. Esses agrupamentos são chamados de *clusters* [BET 97, BET 98].

Logo, uma LUT muito pequena significa perda de área com roteamento, e uma LUT muito grande significa bloco lógico muito grande. Portanto, ao se projetar um FPGA deve-se tentar encontrar um tamanho ótimo para o bloco lógico.

Tradicionalmente, os projetistas têm projetado FPGAs para uso geral, com blocos lógicos de granularidade fina (1 bit). Uma abordagem diferente consiste em otimizar o circuito integrado para uma classe específica de circuitos e aplicações [BRO 96], como no DP-FPGA (*Data Path FPGA*, [CHE 94]). Nesse caso, cada bloco do DP-FPGA manipula um conjunto de bits, implementando operações aritméticas, multiplexadores, e outras funções, sobre esse conjunto de bits. O DP-FPGA é otimizado tanto em nível de blocos lógicos como em estruturas de roteamento. Segundo [BRO 96], utilizando-se o DP-FPGA para a implementação de partes operativas, conseguiu-se uma redução de área em torno de 50% comparando-se com FPGAs tradicionais. Na seção 3.8 mostra-se em mais detalhes essa arquitetura.

3.6.2 Flexibilidade de Interconexão

Além da complexidade do bloco lógico, outro parâmetro que determina a arquitetura de um FPGA é a sua estrutura de interconexão. Poucos recursos de interconexão podem comprometer o roteamento do circuito e recursos excessivos significam desperdício de área. Além disso, a estrutura de interconexão é responsável por cerca de 40 a 60% do tempo total de propagação do sinal [FRA 92] *apud* [BRO 96].

A granularidade do bloco lógico também influencia na escolha da estrutura de interconexão. De um modo geral, quanto mais fina a granularidade do bloco lógico, mais blocos são necessários para a implementação de uma determinada função e, conseqüentemente, mais recursos de conexão são necessários. Da mesma forma, em uma arquitetura de granularidade grossa, menos recursos de conexão são necessários.

3.7 Arquiteturas Reconfiguráveis de Propósito Geral

A seguir são feitas considerações sobre alguns FPGAs comerciais de propósito geral muito difundidos atualmente, destacando-se as famílias Flex10k [ALT 2001] da Altera, XC4000 e XC5200 [XIL 99, XIL 98] da Xilinx.

3.7.1 FPGAs Altera

Um dos dispositivos programáveis mais difundidos da Altera são os FPGAs da família Flex10k [ALT 2001]. Esses FPGAs utilizam tecnologia SRAM para configuração e sua arquitetura constitui-se de uma matriz composta por dois tipos de blocos (Fig. 3.6), o EAB e o LAB.

O **EAB** (*Embedded Array Block*) é utilizado para implementar memória e funções lógicas especializadas. Quando utilizado para implementar memória, cada EAB provê 2048 bits, que podem ser utilizados para implementar memórias ROM, RAM, Filas, etc.

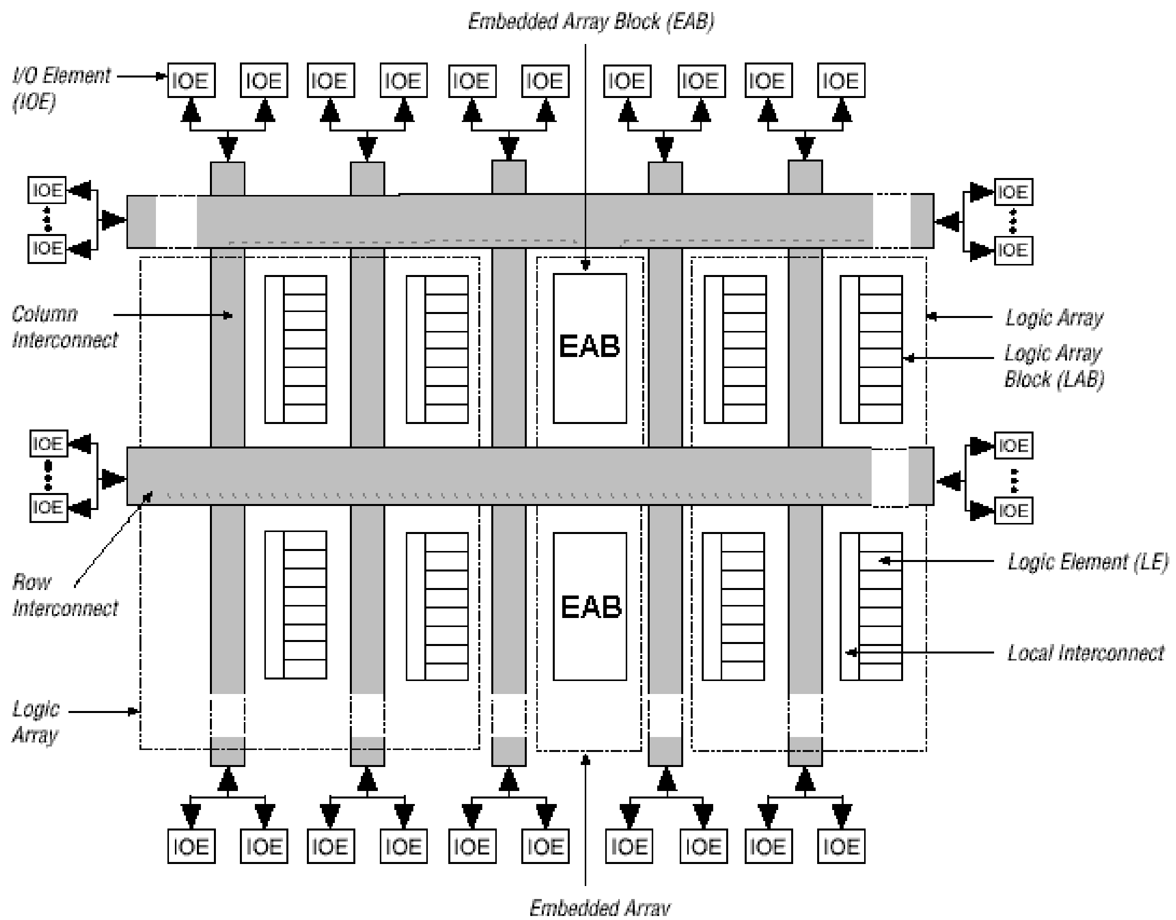

FIGURA 3.6 - Diagrama de blocos do Flex10k. Fonte:[ALT 2001]

Já o **LAB** (*Logic Array Block*) consiste de um agrupamento de 8 LEs (*Logic Element*) mais recursos de conexões locais. Um LE consiste de uma LUT de 4 entradas, um *flip-flop* programável e sinais dedicados para cascateamento de funções e de lógica de *carry* (Fig. 3.7). Cada LE pode ser independentemente configurado, constituindo uma arquitetura de granularidade de 1 bit.

Os recursos para conexões locais em cada LAB têm a função de facilitar o roteamento, otimizando a utilização do dispositivo e possibilitando um melhor desempenho [ALT 2001]. Num LAB é possível se implementar um somador completo de 8 bits.

Famílias mais novas de FPGAs, tais como Apex20k [ALT 2002], Apex II [ALT 2001a] e Mercury [ALT 2002a] também possuem LABs e LEs muito semelhantes aos do Flex10k. Uma das diferenças é que cada LAB é formado pela concatenação de 10 LEs e não 8 como é o caso do Flex10k.

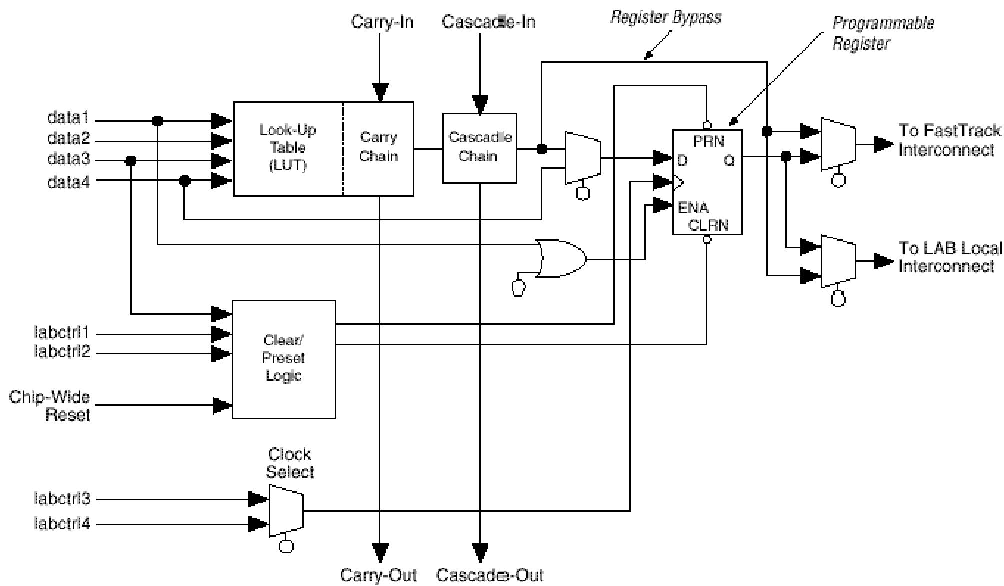

FIGURA 3.7 - Elemento Lógico (LE) do Flex10k. Fonte:[ALT 2001]

Além disso, essas famílias adicionam outras estruturas bem mais complexas, tais como: blocos de memórias com 589824 bits (incluindo bits de paridade) [ALT 2002b]; macrocélulas para implementação de lógica termo-produto [ALT 2002]; cadeias de carry com capacidades para implementação de somadores tipo *carry select look ahead* [ALT 2002a]; estruturas destinadas à implementação de blocos DSP, tais como multiplicadores e filtros FIR [ALT 2002b], entre outros.

Em todas famílias, os blocos lógicos podem ser configurados independentemente, constituindo uma arquitetura de 1 bit de granularidade.

3.7.2 FPGAs Xilinx

Os FPGAs das famílias *XC4000, XC5200, Virtex* e *Spartan II* da Xilinx [XIL 99, XIL 98, XIL 2001, XIL 2001a] também utilizam SRAM como tecnologia

de programação. Os blocos lógicos (*Configurable Logic Block - CLB*) dessas arquiteturas utilizam Look-up Tables como gerador de função lógica, e estão organizados na forma de uma matriz.

Na família XC4000 [XIL 99], o bloco lógico (Fig. 3.8) contém duas LUTs de 4 entradas e uma LUT de 3 entradas, permitindo implementar em um único CLB qualquer função de 5 entradas ou algumas funções de até nove entradas. Cada CLB possui lógica aritmética dedicada para geração de sinais *carry* e *borrow*, permitindo implementar um somador de 2 bits num único CLB. O bloco também possui dois *flip-flops*, e são gerados quatro sinais de saída, sendo dois com registrador e dois sem registrador. Ao contrário do Flex10k, o XC4000 não possui blocos específicos para implementação de memórias RAM. Para essa função são utilizadas as LUTs presentes no bloco lógico, permitindo que cada CLB implemente duas RAMs 16x1 ou uma RAM 32x1.

A família XC5200 [XIL 98] é formada por uma matriz de blocos chamados *VersaBlock*. Cada *VersaBlock* é composto por um agrupamento (*Configurable Logic Block - CLB*, Fig. 3.9) de 4 células lógicas (*Logic Cell - LC*) mais recursos de roteamento local. Cada LC possui apenas uma LUT de 4 entradas e um *flip-flop*. O CLB possui lógica dedicada para geração de *carry* e cadeias de cascateamento para implementação de funções com grande número de entradas. No entanto, para a implementação de um somador de N bits são necessárias *2N* LCs. Ao contrário do XC4000, as LUTs não podem ser utilizadas para implementação de memórias RAM.

As famílias Virtex [XIL 2001] e Spartan II [XIL 2001a] implementam seus blocos lógicos com duas LUTs de 4 entradas e dois *flip-flops*. Os blocos lógicos dessas 4 famílias (XC4000, XC5200, Virtex e SpartanII) podem ser configurados independentemente, constituindo arquiteturas de 1 bit de granularidade.

3.8 Arquiteturas Reconfiguráveis para Aplicações Específicas

Diversas propostas de arquiteturas reconfiguráveis para aplicações específicas podem ser encontradas na literatura. Muitas delas são direcionadas para aplicações DSP, ou seja, uma arquitetura reconfigurável otimizada para a implementação desta classe de aplicações. Essas propostas diferem basicamente pela granularidade de seus blocos programáveis, modelo de conexões e estratégia de reconfiguração. Uma breve descrição de algumas arquiteturas estudadas é feita a seguir.

3.8.1 DP-FPGA

Cherepacha propõe em [CHE 94] o DP-FPGA (*Datapath* FPGA), um FPGA otimizado para a implementação de aplicações intensivas em partes operativas. A proposta é desenvolver um FPGA composto de 3 partes: memória, controle e parte operativa (Fig. 3.10). Cada parte do DP-FPGA é otimizada para implementar uma classe de circuito (memória, controle ou parte operativa). O trabalho, no entanto, foca somente o desenvolvimento da parte operativa, detalhando a arquitetura do bloco lógico e abordando alguns aspectos sobre a arquitetura de roteamento.

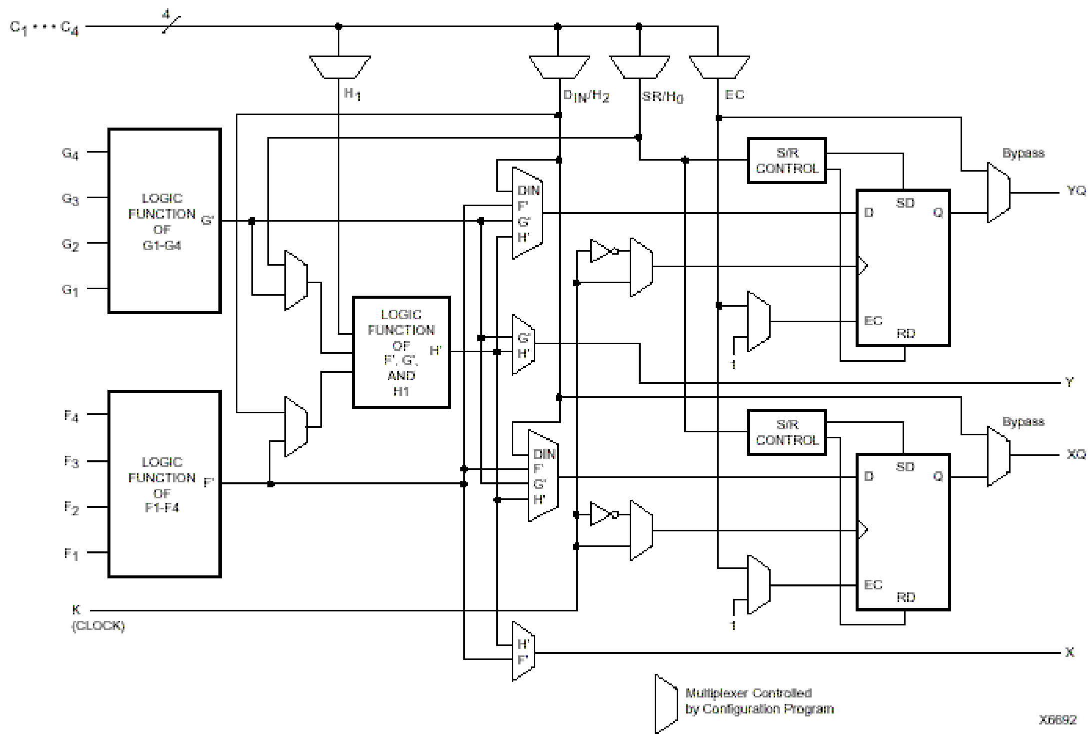

FIGURA 3.8 - Bloco lógico (CLB) do XC4000. Fonte:[XIL 99]

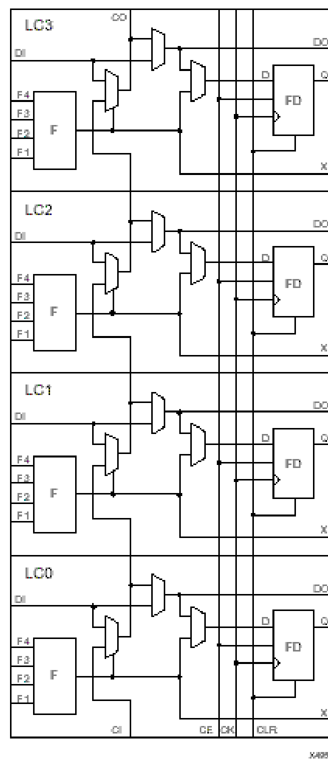

FIGURA 3.9 - Bloco lógico (CLB) do XC5200. Fonte:[XIL 98]

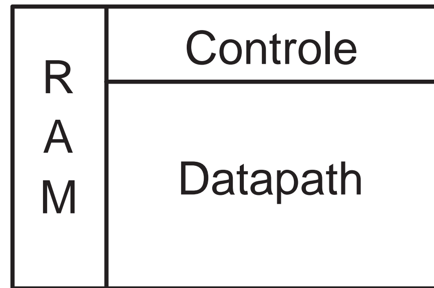

FIGURA 3.10 - Visão geral do DP-FPGA.

O DP-FPGA utiliza SRAM como tecnologia de programação. O bloco lógico possui granularidade de 4 bits e é composto por uma LUT de 4 entradas e 4 saídas. Ao contrário dos FPGAs Flex10k e XC4000, no DP-FPGA foi adotada a estratégia de compartilhamento de bits de configuração, ou seja, uma célula SRAM é utilizada para configurar 4 bits do bloco lógico. Para exemplificar, considere que para implementar um multiplexador 2x1 de 4 bits numa arquitetura com granularidade de 1 bit sejam necessários 4 multiplexadores 2x1. Obviamente, como a granularidade da aquitetura é de um bit, cada multiplexador possuirá uma célula SRAM na entrada de seleção do multiplexador, para configuração do mesmo (Fig. 3.2, 3.11a). Visto que todas as células sempre armazenarão o mesmo valor (0 ou 1), pode-se utilizar apenas uma célula SRAM e compartilhar a mesma entre todos os multiplexadores (Fig. 3.11b). Com isso, reduz-se o número de células SRAM necessárias no chip [CHE 94].

Segundo [CHE 94], o roteamento de sinais em partes operativas possui estruturas mais regulares do que as encontradas em lógica de controle. Dessa forma, no DP-FPGA foram implementados canais de roteamento separados para os sinais de dados e de controle, ao contrário dos FPGAs convencionais que utilizam o mesmo canal para o roteamento de ambos os sinais.

O roteamento dos sinais de dados é feito em blocos de 4 bits, ou seja, uma célula SRAM é utilizada para configurar 4 segmentos de fio do canal de roteamento, novamente com o intuito de diminuir a área no chip gasta com células SRAMs. Além disso, existem muito mais canais horizontais do que verticais, pois a maioria das conexões são feitas entre blocos na mesma linha [CHE 94].

Já os sinais de controle são roteados individualmente e, em vez de compartilharem bits de configuração, é cada sinal de controle que é compartilhado entre os 4 bits do bloco lógico. Isso leva a uma economia de área, uma vez que somente 1 segmento de fio é requerido para conectar um sinal de controle ao bloco lógico de 4 bits. Normalmente, em partes operativas, um sinal de controle é utilizado por cada fatia (*bit-slice*) do bloco. Isso faz com que os sinais de controle se estendam perpendicularmente em relação ao fluxo de dados. Logo, o DP-FPGA possui muito mais canais verticais do que horizontais para o roteamento dos sinais de controle [CHE 94].

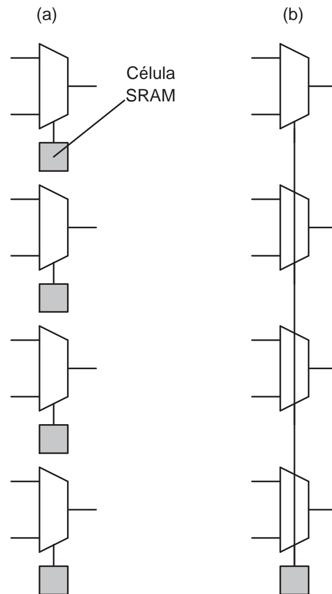

FIGURA 3.11 - Multiplexadores sem compartilhamento de bits de configuração (a) e com compartilhamento (b).

3.8.2 FPPA

Em [NUS 98] é apresentado um trabalho que tem dois objetivos principais: o primeiro é desenvolver uma matriz de unidades para processamento massivamente paralelo, e o segundo é obter um bom substrato para a implementação de algoritmos evolutivos. O dispositivo também tolera defeitos através de auto-teste e reconfiguração autônoma. Os autores consideram o dispositivo como um FPGA de alta granularidade com características de uma arquitetura MIMD (*Multiple Instruction Multiple Data*), que eles denominaram FPPA (*Field Programmable Processor Array*).

Esse FPPA é formado por uma matriz de células ligadas vizinho-a-vizinho. Cada célula é constituída por um processador de 8 bits de baixo consumo, mais memórias RAM e ROM e uma interface para rede de comunicação. O processador é um CoolRISC 816, com 16 registradores, um multiplicador 8x8, 4 ponteiros de dados com pós-incremeto e divisor programável de frequência (1 a 16). O modelo de memória é Harvard, com uma memória RAM de dados com 512 bytes, uma memória RAM de código com 512 instruções, mais uma memória ROM de código com 2048 instruções.

3.8.3 PADDI

Em [CHE 92] é proposta uma arquitetura reconfigurável de granularidade grossa destinada à prototipação de partes operativas típicas de sistemas de processamento de sinais de tempo real (*PADDI - Programmable Arithmetic Devices for High-Speed Digital Signal Processing*). Essa arquitetura consiste de *clusters* de 8 processadores conectados por uma rede de chaveamento configurável tipo *cross-bar*.

Cada processador possui uma ULA (Unidade Lógica e Aritmética) de 16 bits de largura, dois bancos de registradores com 6 registradores cada um, um registrador de deslocamento e um registrador para *pipeline*. Um dos registradores do banco de registradores pode ser configurado como um registrador de varredura, para ser utilizado em teste de varredura (*scan testing*).

3.9 Considerações Sobre FPGAs

Neste capítulo foram apresentados alguns conceitos relativos a arquiteturas reconfiguráveis, bem como foram mostradas algumas arquiteturas reconfiguráveis existentes. Foi dada maior ênfase aos FPGAs (*Field-Programmable Gate Array*), uma vez que são os dispositivos reconfiguráveis mais difundidos atualmente, e a arquitetura proposta neste trabalho se assemelha mais a esse tipo de dispositivo.

Os diversos tipos de blocos lógicos, arquiteturas de roteamento e tecnologia de configuração estudados serviram de subsídio para o projeto da nova arquitetura. Além disso, pode-se dizer que os FPGAs comerciais estudados (Flex10k, Apex20k, Apex II, Mercury, XC4000, XC5200, Virtex e Spartan II) possuem todos granularidade de 1 bit, não sendo otimizados para implementação de partes operativas e funções DSP, a menos da família *Stratix*, que possui alguns blocos específicos para implementação de multiplicadores e filtros FIR.

No próximo capítulo são mostradas algumas estruturas e arquiteturas destinadas à implementação de técnicas de teste de hardware. Com isso ter-se-á obtido os dados necessários para a implementação do bloco lógico do BiFi-FPGA.

4 Teste de Hardware

Este capítulo aborda algumas técnicas de projeto visando a testabilidade e o teste de sistemas de hardware. São mostradas algumas arquiteturas para teste de memórias RAM e ROM, geração de vetores pseudo-aleatórios e análise de assinaturas utilizando LFSRs, teste utilizando varredura (*Scan*), entre outras.

4.1 Métodos de Teste

Desde a fase de prototipação de um circuito, este é submetido a uma série de etapas para a verificação de seu funcionamento: depuração do protótipo, teste de produção (pós-fabricação) e teste de manutenção [FUJ 85, LUB 2000].

Durante essas etapas de verificação, pode-se identificar e isolar os dispositivos que apresentam falhas ou até mesmo substituir partes defeituosas. Esse tipo de teste é chamado teste fora de funcionamento (*off-line*), uma vez que se necessita que a operação do circuito seja interrompida para a aplicação dos procedimentos de teste.

A verificação fora de funcionamento normalmente utiliza a técnica de teste externo [ABR 90]. Esta técnica consiste na aplicação de estímulos (vetores de teste) nas entradas do circuito e observação dos resultados nas suas saídas. Outra técnica também utilizada é a de auto-teste integrado (*Built-In Self Test - BIST*), na qual as funções de geração dos estímulos e observação dos resultados estão integradas no próprio circuito a ser testado.

Já no teste em funcionamento (*on-line*), a detecção de falhas ocorre concorrentemente com a aplicação. Sua utilização tem lugar em sistemas que requerem alta segurança, como por exemplo, sistemas automotivos, de aviação, de controle (plantas nucleares, trens), entre outros. A detecção *on-line* pode ser assegurada através de mecanismos especiais, como temporizadores *watchdog* e dados codificados, ou por dispositivos *self-checking* [LUB 2000].

4.2 Modelos Lógicos de Falhas

A manifestação de uma operação incorreta de um circuito é chamada de erro (observado). As causas dos erros observados podem ser erros de projeto (p. ex. especificações inconsistentes ou incompletas), erros de fabricação (p. ex. soldagem incorreta de componentes), defeitos de fabricação (p. ex. desalinhamento de máscaras) e defeitos físicos (p. ex. desgaste de componentes devido a temperatura e umidade) [ABR 90].

Os erros de fabricação, defeitos de fabricação e defeitos físicos são chamados todos de falhas físicas. Em geral, as falhas físicas não permitem um tratamento matemático direto quando se está lidando com o teste de circuitos [ABR 90].

Dessa forma, criaram-se os modelos lógicos de falhas, os quais consistem em

representar em nível lógico os defeitos físicos que podem ocorrer num sistema. Com isso, os problemas de análise e simulação de falhas tornam-se problemas lógicos, ao invés de problemas físicos, simplificando a complexidade de tratamento do teste. Além disso, alguns modelos lógicos de falhas são independentes de tecnologia, de modo que o mesmo modelo pode ser aplicado a muitas tecnologias [ABR 90].

A modelagem de falhas está fortemente relacionada ao tipo de modelo utilizado para representar o sistema [ABR 90]. Falhas definidas em conjunção com um modelo estrutural do sistema são ditas falhas estruturais, e seu efeito é modificar as interconexões entre componentes do sistema, por exemplo.

Modelos estruturais de falhas típicos são os modelos de colagem simples (*Single Stuck-at Faults - SSF*), modelos de colagem múltipla (*Multiple Stuck-at Faults - MSF*) e modelos de curto-circuito (*Bridging Faults - BF*). No modelo de colagem, considera-se que um sinal está sempre colado no valor zero lógico (*s-a-0 - Stuck at Zero*) ou um lógico (*s-a-1 - Stuck at One*). No modelo de colagem simples considera-se que há no máximo uma falha no sistema. No modelo de falhas múltiplas considera-se que pode haver mais de uma falha simultânea no sistema. Falhas típicas em conexões são os curto-circuitos (*short*) e circuitos abertos (*open*) [ABR 90].

Falhas funcionais são definidas em conjunção com um modelo funcional do sistema. O efeito de uma falha funcional pode ser, por exemplo, a mudança na tabela verdade de um componente.

4.3 Simulação de Falhas, Geração e Aplicação do Teste

Vários aspectos estão relacionados com o processo de teste, podendo-se destacar: a qualidade do teste gerado, o custo de geração do teste e o custo de aplicação do teste.

A qualidade do teste depende muito da cobertura de falhas como também do número de vetores de teste aplicados [FUJ 85]. A cobertura de falhas é o percentual de falhas que podem ser detectadas ou localizadas dentro do circuito sob teste. Esta pode ser determinada através da **simulação de falhas**, que consiste basicamente na simulação do circuito na presença de falhas e na comparação dos resultados individuais com o resultado obtido da simulação do circuito sem falhas [ABR 90].

Através desse processo pode-se verificar se estas falhas são ou não detectadas através da aplicação de determinados estímulos de entrada. Com isso, pode-se fazer uma avaliação da eficácia do teste, pela verificação da cobertura de falhas [LUB 2000].

A **geração de teste** consiste em encontrar um conjunto de estímulos de entrada (vetores de teste) e medidas de saídas que garantam uma determinada cobertura de falhas. Já a **aplicação do teste** consiste em aplicar nas entradas do circuito os vetores de teste que foram obtidos na fase de geração de teste,

seguindo-se da observação dos valores nas saídas do mesmo [FUJ 85].

No domínio digital, a geração de vetores de teste pode ser feita através das abordagens exaustiva, pseudo-aleatória ou determinística [ABR 90, LUB 2000].

O **teste exaustivo** considera a geração e aplicação de todos os possíveis vetores de entrada do circuito, o que perfaz um total de 2^N vetores, onde N é o número de entradas do circuito. Embora esse método de geração seja simples e assegure a maior cobertura de falhas possível, essa abordagem só pode ser aplicada a circuitos com poucas entradas, pois o tempo de aplicação do teste pode se tornar longo para circuitos com muitas entradas.

Já no **teste pseudo-aleatório**, geradores especiais produzem vetores pseudo-aleatórios (geradores de números aleatórios, por exemplo), considerando diferentes valores de inicialização e tamanhos para as seqüências de teste. Esse método produz, em geral, boa cobertura de falhas dentro de um tempo razoável de aplicação do teste. Entretanto, para alguns tipos de circuitos, tais como os baseados em estruturas regulares (memórias, por exemplo), são necessárias seqüências de teste muito longas para se alcançar a qualidade de teste desejada. Nesse caso, o tempo de aplicação do teste pode se tornar muito longo [LUB 2000].

No **teste determinístico**, a geração dos vetores de teste é feita a partir de um modelo do circuito a ser testado. Dois tipos de métodos podem ser utilizados: métodos algébricos e topológicos [LUB 2000]. Nos métodos algébricos, os vetores de teste são computados a partir de expressões booleanas que descrevem o funcionamento do circuito sob teste. Nos métodos topológicos, os vetores de teste são derivados a partir da estrutura do circuito sob teste.

O método topológico mais conhecido é o algoritmo D, que consiste essencialmente em propagar uma falha de colagem para uma saída primária do circuito, justificando, em seguida, as entradas do circuito, de forma a produzir no nodo com falha o comportamento oposto ao previsto pela falha injetada [FUJ 85, ABR 90, LUB 2000].

O algoritmo D é ineficiente na geração de vetores de teste para circuitos com muitas portas XOR [FUJ 85]. Com isso, foram desenvolvidos os algoritmos PODEM (*Path-Oriented DEcision Making*) e FAN (*Fanout-Oriented TG*), que são extensões do algoritmo D [FUJ 85, ABR 90].

Apesar de ser mais caro (computacionalmente) que a geração aleatória e a exaustiva, o teste determinístico permite uma alta qualidade do teste gerado e com um tempo de aplicação mais baixo [ABR 90].

4.4 Projeto Visando o Teste

Mesmo que se tenha uma ferramenta de geração de teste, falhas difíceis de detectar impedem que se consiga um bom compromisso entre a cobertura de

falhas e o tempo de aplicação do teste. Nestes casos, pode-se re-projetar o circuito para melhorar a acessabilidade aos elementos difíceis de testar (projeto visando a testabiliade) [ABR 90].

Outra possibilidade é o auto-teste fora de funcionamento, que consiste em integrar no circuito a ser testado, estruturas para geração e avaliação do teste. Se a aplicação requerer que falhas sejam detectadas durante o funcionamento normal do circuito, pode-se projetar um circuito auto-testável em funcionamento. A seguir, são apresentados esses três métodos de projeto visando o teste.

4.4.1 Projeto Visando a Testabilidade

Técnicas de projeto visando a testabilidade são esforços de projeto empregados especificamente para assegurar que um dispositivo possa ser testado [ABR 90]. Dois importantes atributos estão relacionados com a testabilidade:

- **Observabilidade**, que é a capacidade de se observar nas saídas do circuito o comportamento de nodos internos do mesmo e,
- **Controlabilidade**, que é a capacidade de se levar os sinais de entrada do circuito a seus nodos internos.

Técnicas *ad hoc*, baseadas na inserção de pontos de teste (controle e observação), no particionamento, no uso de multiplexadores para dar acesso a nodos difíceis de testar, na inibição de osciladores e caminhos de realimentação, na inserção de facilidades de inicialização, etc. podem ser utilizadas para melhorar a testabilidade dos circuitos [ABR 90, LUB 2000].

Entretanto, técnicas estruturadas são mais satisfatórias para enfrentar os problemas de teste típicos de circuitos integrados e sistemas complexos [LUB 2000]. A técnica estruturada de projeto visando a testabilidade mais popular usada para teste externo é conhecida como projeto com varredura (*scan design*), uma vez que ela emprega registradores de varredura (*scan registers*) [ABR 90]. Um registrador de varredura é um registrador com capacidades de carga paralela e deslocamento (*shift*).

A Fig. 4.1 mostra a forma genérica de um registrador de varredura, bem como a célula de armazenamento e varredura (*Scan Storage Cell - SSC*). No modo normal ($\overline{N}/T = 0$), o dado carregado na SSC vem da entrada D. No modo teste ($\overline{N}/T = 1$), o dado vem da entrada *Sin*.

Com a estrutura da Fig. 4.1, consegue-se acesso a vários nodos internos do circuito utilizando-se apenas 3 sinais (*Sin, Sout e* $\overline{N}/T$). Como desvantagens têm-se maior custo em área, devido à inserção de registradores de varredura, e em tempo de aplicação do teste, uma vez que os sinais de teste, bem como os resultados, devem ser deslocados de forma serial.

Uma aplicação típica dessa técnica é o *Boundary Scan* [ABR 90], o qual consiste na inserção de registradores de deslocamento na periferia do circuito.

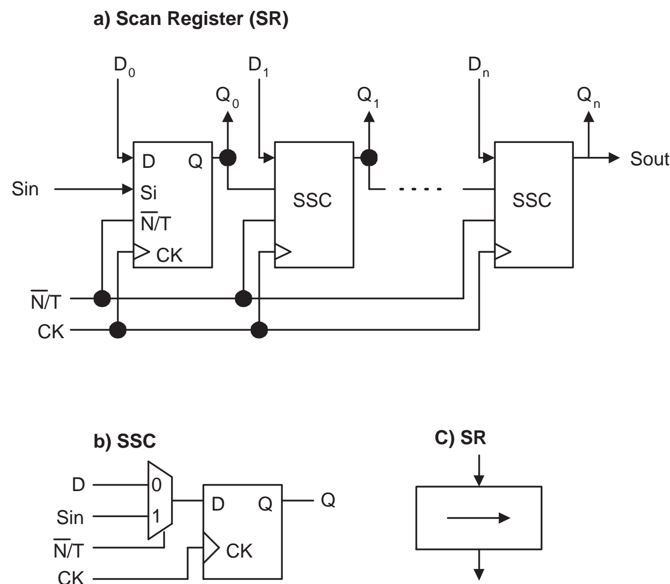

FIGURA 4.1 - (a) Scan Register, (b) Scan Storage Cell e (c) simbologia do Scan Register.

4.4.2 Auto-Teste Fora de Funcionamento

O aumento da complexidade dos circuitos integrados e sistemas tem tornado mais difícil a aplicação do teste externo utilizando-se equipamentos de teste. Uma alternativa viável é a integração de algumas ou todas as funções de teste no próprio circuito ou sistema a ser testado. Essa técnica é conhecida como auto teste integrado (*Built-In Self Test - BIST*) [ABR 90], e consiste em integrar no próprio sistema a ser testado mecanismos para a geração de estímulos de teste, avaliação das respostas de teste, controle do teste e isolamento das entradas e saídas do circuito durante a aplicação do teste [LUB 2000].

Para a geração do teste no próprio *chip* pode-se utilizar vetores gerados de maneira determinística, armazenados numa memória ROM, por exemplo. Também pode-se gerar os vetores utilizando-se decodificadores, contadores ou qualquer máquina de estados. A vantagem desse tipo de teste é o tempo de aplicação reduzido. Como desvantagem tem-se o grande aumento de área no *chip* final [LUB 2000].

Uma alternativa é a utilização de geradores pseudo-aleatórios, implementados com LFSR (*Linear Feedback Shift Register*) [AGR 93, AGR 93a]. Um LFSR consiste de *flip-flops* tipo D conectados em configuração de registrador de deslocamento, com portas OU-Exclusivo (XOR) implementando uma rede de realimentação linear.

Para avaliação do teste *on-chip*, pode-se utilizar uma memória para armazenar os resultados esperados ou então utilizar algum tipo de compactação das respostas. A vantagem da compactação é a redução de área do *chip*. Porém, perde-se informação nesse processo, podendo levar a resultados idênticos em um circuito com e sem falhas (*aliasing*).

Os métodos de compactação mais utilizados baseiam-se na contagem de 1s ou 0s, contagem de transições 0 para 1 (ou vice-versa), integração digital [RAJ 93, CHA 97] e divisão polinomial utilizando LFSR. Outro método de compactação muito utilizado também é a verificação de paridade, que pode ser implementada com portas XOR.

A maior parte das aplicações utiliza o método da divisão polinomial, também chamado analisador de assinatura. Quando o analisador possui várias entradas ele é chamado de MISR (*Multiple Input Signature Register*) [ABR 90]. A Fig. 4.2 mostra a forma genérica de um LFSR [AGR 93], o qual é representado pelo polinômio

$$g(x) = x^n + g_{n-1}\,x^{n-1} + g_{n-2}\,x^{n-2} + ... + g_1 x + g_0$$

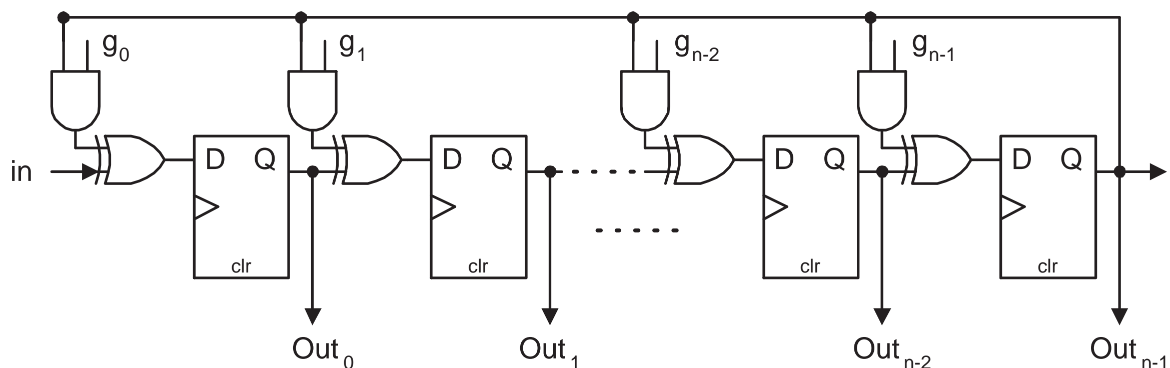

FIGURA 4.2 - Representação genérica de um LFSR.

4.4.3 Auto-Teste em Funcionamento

Os circuitos capazes de detectar um erro durante o funcionamento normal do sistema são chamados de *self-checking* (circuitos com auto-verificação) [ABR 90]. Esses sistemas são compostos de um bloco funcional e um bloco verificador (Fig. 4.3). O bloco funcional produz uma saída codificada e o bloco verificador verifica se a saída pertence a um código válido. Se a saída do bloco funcional não pertencer ao código de detecção de erros utilizado, o circuito verificador deve sinalizar um erro. Os códigos mais utilizados são o de paridade, o de Berger e o código dual (*double-rail*) [LUB 2000].

4.5 Teste de Blocos Específicos

A seguir são apresentados alguns métodos de teste para blocos específicos, tais como memórias ROM e RAM, filtros FIR e FPGAs.

4.5.1 Teste de Memórias ROM

O teste de memórias ROM pode ser executado através da compactação de todos os valores contidos na ROM, gerando um valor de assinatura que será comparado com o valor de uma memória sabidamente sem defeitos. Para tanto,

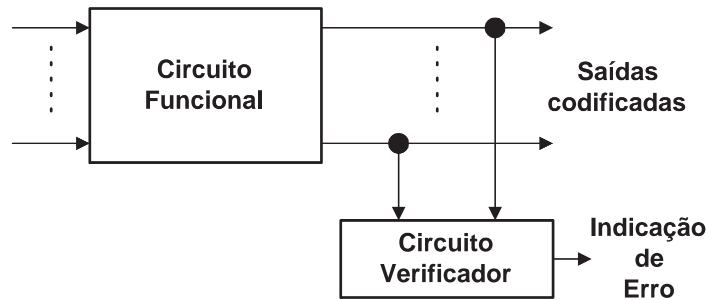

FIGURA 4.3 - Circuito self-checking.

pode-se utilizar um LFSR como analisador de assinatura, como mostrado em [BAR 91, IWA 96].

Outro método de compactação que poderia ser utilizado é a integração digital [RAJ 93, CHA 97], também chamada de *Check-Sum*, a qual consiste em somar todas as palavras da memória e comparar o resultado com um valor pré-armazenado de uma memória sabidamente sem defeitos. Se os dois valores forem diferentes, então um sinal de erro é ativado, indicando que a memória está com defeito. As seguintes estruturas são necessárias para a implementação em hardware desse teste: um contador, um somador, um registrador e um comparador.

A Fig. 4.4 mostra um exemplo de implementação dedicada para o teste de uma memória ROM 16x4 (16 palavras de 4 bits cada). O bloco PC (parte de controle) recebe o sinal para início do teste (*start*) e gera adequadamente os sinais *reset* e $\overline{N}/T$. O valor (*count*) do contador é utilizado pela parte de controle para saber quando encerrar o teste. Nesse exemplo, o valor do *check-sum* a ser comparado é 3 e como comparador utilizou-se uma porta NOR de 4 entradas e dois inversores nas entradas correspondentes aos bits 0 e 1 do resultado da acumulação, pois o valor de *check-sum* desse exemplo é 3 (0011_2). O sinal $\overline{Error}$ assumirá o valor lógico 0 (zero) sempre que o valor sendo comparado for diferente de 3. Caso contrário, se o valor sendo comparado for igual a 3, então o sinal $\overline{Error}$ irá para o valor lógico 1 (um). Dessa forma, após terem sido acumuladas as 16 palavras da memória, o sinal $\overline{Error}$ é observado para se verificar o resultado do teste de *check-sum* (1=*check-sum* correto, 0=*check-sum* incorreto).

4.5.2 Teste de Memórias RAM

O teste de memórias RAM está se tornando cada vez mais importante, principalmente devido ao fato dessas memórias estarem embutidas em sistemas cada vez maiores. Nesses casos, a utilização de técnicas de teste baseadas em pontos de observação e controle muitas vezes falham [CAM 95]. O objetivo do teste funcional de memórias é garantir que cada célula esteja apta a armazenar um dado conforme especificado e ser unicamente endereçada, escrita e lida [CAM 95].

Diversos algoritmos para teste de SRAMs podem ser encontrados na literatura [GOO 93, TRE 93, ALV 95, CAM 95]. Em [GOO 93] são apresentados diversos algoritmos baseados em elementos *March* para o teste de memórias de 1 bit de

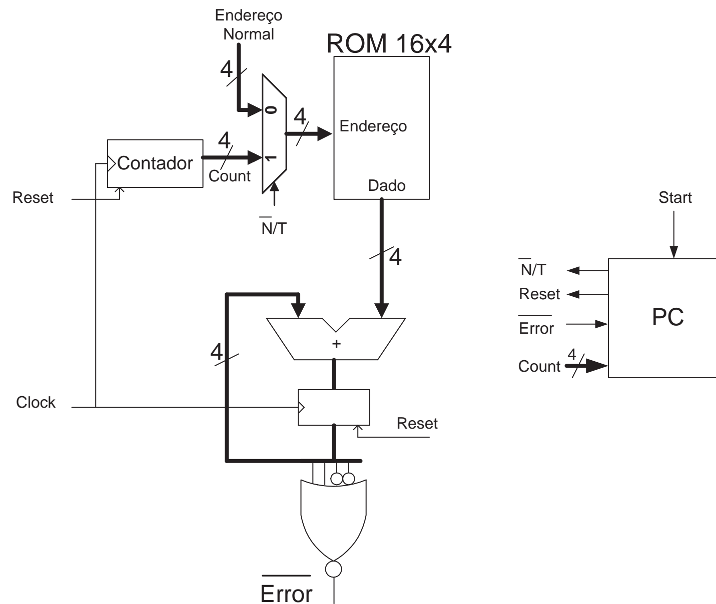

FIGURA 4.4 - Teste de ROM.

largura de dados (*bit-wide*). Os testes *March* consistem em uma seqüência de elementos *March*. Um elemento *March* consiste de uma seqüência de operações aplicadas a cada célula de memória, antes de se proceder à próxima célula. Uma operação pode consistir em escrever zero (w0) ou um (w1) numa célula ou, ler uma célula com valor esperado 1 (r1) ou 0 (r0). Após todas as operações de um elemento March terem sido aplicadas a uma determinada célula, elas serão aplicadas à célula seguinte. O endereço da célula seguinte é determinado pela ordem de endereçamento, podendo ser ascendente (⇑), descendente (⇓) ou irrelevante (⇕). Como exemplo, na Fig. 4.5 é apresentado o algoritmo March-B [GOO 93].

⇑ (w0)
⇑ (r0, w1, r1, w0, r0, w1)
⇑ (r1, w0, w1)
⇓ (r1, w0, w1, w0)
⇓ (r0, w1, w0)

FIGURA 4.5 - Algoritmo March-B (versão 1 bit).

O algoritmo March-B apresentado funciona somente com memórias *bit-wide*. Para memórias *word-wide*, podem ser utilizadas duas técnicas:

1. **Primary Data Backgrounds**: Para uma memória de 4 bits, replica-se a versão 1 bit do algoritmo March-B três vezes, substituindo-se cada ocorrência de w0 e w1 por w0101 e w1010 na primeira réplica, w0 e w1 por w0011 e w1100 na segunda réplica e w0 e w1 por w0000 e w1111 na terceira réplica (Tab. 4.1). As operações r0 e r1 são substituídas de maneira similar [TRE 93]. A Tab. 4.2 mostra os padrões para uma memória de 8 bits.

2. **Marching and Walking Data Backgrounds**: Nesse tipo de teste, as ocorrências de w0 e w1 são substituídas conforme mostra a Tab. 4.3. A Fig. 4.6 apresenta a versão (4 bits) *Marching and Walking* do algoritmo March-B [TRE 93].

TABELA 4.1 - Primary data backgrounds (4 bits)

0 trocado por	**1 trocado por**
0101	1010
0011	1100
0000	1111

TABELA 4.2 - Primary data backgrounds (8 bits)

0 trocado por	**1 trocado por**
01010101	10101010
00110011	11001100
00001111	00001111
00000000	11111111

TABELA 4.3 - Marching and Walking Data backgrounds (4 bits)

	⇑w0	**⇓w0**	**⇑w1**	**⇓w1**
odd-marching	0111	1110	1000	0001
	0011	1100	1100	0011
	0001	1000	1110	0111
	0000	0000	1111	1111
even-marching	0000	0000	1111	1111
	1000	0001	0111	1110
	1100	0011	0011	1100
	1110	0111	0001	1000
odd-walking	0111	1110	1000	0001
	1011	1101	0100	0010
	1101	1011	0010	0100
	1110	0111	0001	1000
even-walking	0000	0000	1111	1111
	0000	0000	1111	1111
	0000	0000	1111	1111
	0000	0000	1111	1111

Na Fig. 4.7 é mostrado um exemplo de implementação dedicada que poderia ser utilizada para o teste de uma memória RAM 16x4 utilizando a versão *Marching and Walking* do algoritmo March-B. O bloco PC (parte de controle) é responsável pelo controle do teste, ou seja, geração correta dos sinais de controle ($\overline{N}/T$, $\overline{R}/W$, *CControl, RControl e SRControl*), verificação dos resultados (monitoração do sinal $\overline{Error}$) e término do teste (ativação do sinal *EndTest*). O bloco *Counter* gera os valores para endereçamento da memória, e o bloco *ShiftRegister* gera os padrões

```
⇑ (w0000)

⇑ (r0000, w1000, r1000, w0000, r0000, w1000,
   r1000, w1100, r1100, w1000, r1000, w1100,
   r1100, w1110, r1110, w1100, r1100, w1110,
   r1110, w1111, r1111, w1110, r1110, w1111)

⇑ (r1111, w0111, w1111,
   r1111, w1011, w1111,
   r1111, w1101, w1111,
   r1111, w1110, w1111)

⇓ (r1111, w1110, w1111, w1110,
   r1110, w1100, w1110, w1100,
   r1100, w1000, w1100, w1000,
   r1000, w0000, w1000, w0000)

⇓ (r0000, w0001, w0000,
   r0000, w0010, w0000,
   r0000, w0100, w0000,
   r0000, w1000, w0000)
```

FIGURA 4.6 - Versão Marching and Walking (4 bits) do algoritmo March-B.

Marching and Walking. O bloco *Register* contém o último padrão gerado pelo bloco *ShiftRegister*. Esse valor é comparado com o valor lido da memória, sendo que a comparação é feita através de 4 portas XOR e uma porta NOR. Se os valores comparados forem diferentes, então o sinal $\overline{Error}$ é ativado (vai para o nível lógico zero), indicando que há um erro na memória. Após executado todo o algoritmo, a parte de controle ativa o sinal *EndTest*.

4.5.3 Teste de Filtros

Diversos métodos para teste funcional podem ser encontrados na literatura. Em [BAY 99] é apresentada uma técnica de teste em funcionamento que consiste em acumular as entradas e saídas do filtro para se calcular o ganho DC do mesmo. Esse valor é invariante, a menos de um valor chamado de tolerância. Para a realização desse teste são necessários 2 acumuladores e um subtrator.

Em [COT 2001] é apresentado um método de teste fora de funcionamento que consiste em definir a assinatura do filtro como sendo a convolução de seus coeficientes com ele mesmo. Dessa forma, os coeficientes do filtro são colocados na entrada do filtro e o mesmo é posto a funcionar normalmente. A saída do filtro é considerada sua assinatura, a qual será um vetor de comprimento $2N$ (onde N é o número de coeficientes do filtro). O resultado dessa execução é comparado com a assinatura que havia sido previamente armazenada. A comparação pode ser feita com portas XOR.

Em [COU 92], uma seqüência de vetores é aplicada à entrada do filtro e sua saída é conectada a um MISR que funciona como analisador de assinatura.

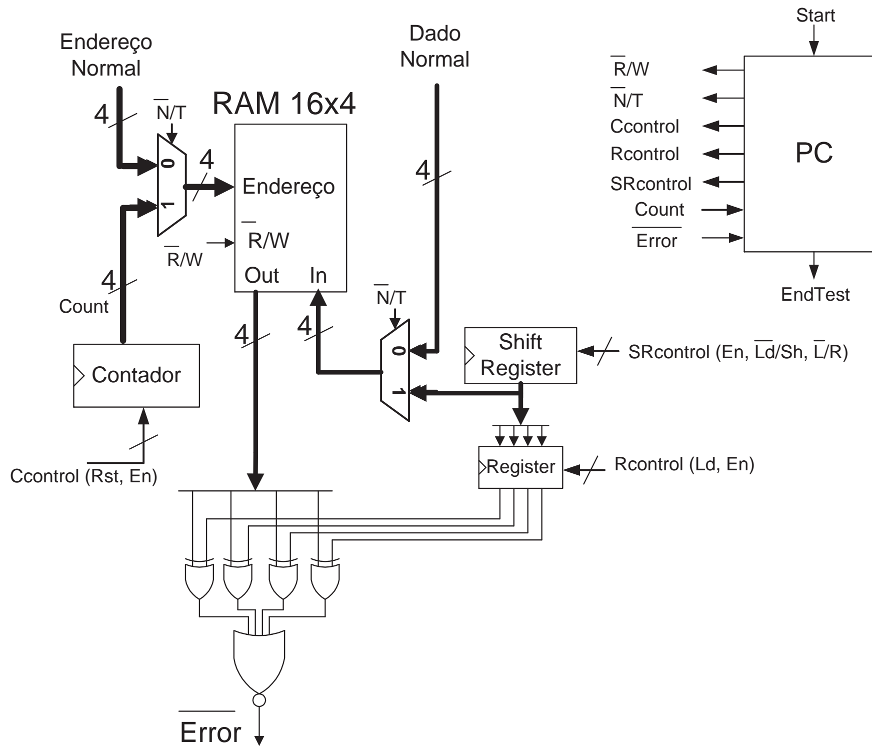

FIGURA 4.7 - Teste de RAM

4.5.4 Teste de FPGAs

Devido às diversas estruturas diferentes existentes nos FPGAs, tais como blocos lógicos, estruturas de roteamento, memórias, etc., diferentes estratégias de teste [REN 96, REN 97, STR 96, STR 98a] devem ser adotadas para cada uma dessas estruturas, a fim de se conseguir um teste efetivo.

Uma dessas estratégias consiste em mapear diversos circuitos no FPGA a ser testado e excitar o mesmo com vetores de teste desenvolvidos especificamente para tais circuitos [REN 97]. Considerando-se que esses circuitos compartilham o mesmo conjunto de falhas, é suficiente detectar as falhas em apenas um dos circuitos.

Entretanto, a simulação para determinar a cobertura de falhas pode se tornar muito cara computacionalmente, uma vez que todos os circuitos devem ser simulados. Além disso, é necessário armazenar todos os vetores de teste. Logo, a soma da área do controlador BIST e da memória para armazenamento dos vetores pode se tornar muito grande [STR 96].

Stroud et. al. [STR 96] propõem uma estratégia diferente para o teste de FPGAs, que não adiciona hardware para o BIST, uma vez que utiliza as próprias células do FPGA para realizar seu auto-teste. O auto-teste consiste de várias sessões de teste, sendo que, em cada sessão, determinados grupos de células funcionam como geradores exaustivos pseudo-aleatórios (*TPG - Test Pattern Generators*), outros grupos funcionam como analisadores de resposta (*ORA - Output Response Analyzer*), e outros grupos são as células que serão testadas (*BUT - Block Under*

Test). Na sessão de teste seguinte, as células que funcionaram como TPGs ou ORAs na sessão anterior agora serão as células a serem testadas (BUTs) e vice-versa.

Cada sessão de teste consiste de uma seqüência de fases de teste, onde são configurados diversos circuitos nas BUTs que serão testadas. Cada fase consiste dos seguintes passos: 1) configurar o circuito; 2) inicializar o teste; 3) gerar os padrões de teste; 4) analisar as respostas e 5) ler os resultados do teste.

Considerando-se que todas as células dos FPGAs são iguais, logo, diversas células são configuradas para implementar TPGs indênticos, bem como ORAs e BUTs. Dessa forma, os blocos ORAs são configurados como simples comparadores, visto que as saídas de todas as BUTs sempre serão iguais entre si (Fig. 4.8). Esse tipo de ORA tem a vantagem sobre os analisadores de assinatura no sentido de não introduzirem problemas de *aliasing* [STR 96].

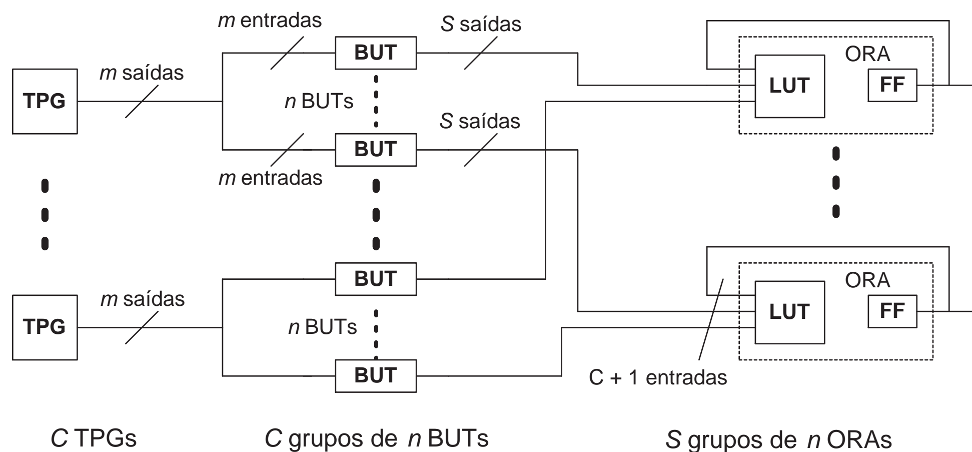

FIGURA 4.8 - Estrutura para auto-teste de FPGA

4.6 Utilização de FPGAs Para Implementação de Teste

Recentemente têm surgido propostas de utilização de hardware reconfigurável para a implementação de técnicas de teste.

Carro *et. al.* [CAR 2000] propõem uma implementação auto-testável do microcontrolador 8051 utilizando FPGAs FLEX10K da Altera. Esta estratégia tem por objetivo diminuir o sobrecusto de área causado pela inserção de estruturas BIST, uma vez que a mesma área poderá ser utilizada para implementar tanto estruturas de BIST, quanto as funcionalidades do 8051, mas em tempos diferentes.

Em [REN 2001] é proposta a alteração da arquitetura da célula básica de FPGAs tradicionais permitindo que os mesmos possam implementar técnicas de teste baseadas em varredura (*scan chain*) com um custo de área adicional mínimo.

Nenhum dos trabalhos, no entanto, trata da utilização de FPGAs otimizados para implementação de outras técnicas de teste, como por exemplo, geração de

padrões pseudo-aleatórios, analisadores de assinatura e testes de memórias RAM e ROM, por exemplo.

Além disso, pode-se verificar diversas semelhanças nas estruturas de hardware necessárias para implementação de funções DSP e BIST (Cap. 2 e 4). Isto sugere que a existência de um FPGA otimizado para a implementação dessas funções poderia ser muito interessante. Dessa forma, este trabalho tenta preencher essa lacuna, propondo uma nova arquitetura de FPGA que possui estruturas otimizadas para implementação dessas duas classes de aplicações (DSP e BIST).

4.7 Considerações Sobre Algoritmos Para Teste

Este capítulo apresentou alguns tópicos relacionados com teste de hardware, mostrando as principais técnicas utilizadas. Foram mostradas em mais detalhes as técnicas de geração de vetores pseudo-aleatórios e analisadores de assinatura, teste de memórias RAM e ROM, teste de filtros digitais e teste de FPGAs. Também foi mencionada a existência de trabalhos relacionados à utilização de FPGAs para implementação de BIST. As técnicas mostradas neste capítulo foram implementadas no BiFi-FPGA, como será visto nos capítulos seguintes.

5 Metodologia e Descrição da Arquitetura

Este Capítulo inicia apresentando a metodologia que foi adotada para o desenvolvimento deste trabalho. Em seguida, faz-se uma descrição detalhada dos componentes do bloco lógico proposto e a função de cada um dentro do contexto das aplicações DSP e BIST. No final, mostra-se a primeira fase de validação da arquitetura, através da implementação de blocos básicos (somadores, subtratores, acumuladores, comparadores, entre outros) na mesma.

5.1 Metodologia

Para o desenvolvimento deste trabalho, fez-se inicialmente uma revisão bibliográfica sobre as diversas técnicas de DSP e BIST e suas aplicações, como mostrado nos capítulos anteriores. O objetivo do estudo teórico foi verificar quais os requisitos, em termos de estruturas de hardware necessárias para a implementação de tais aplicações, bem como a possível semelhança entre tais estruturas. Esse estudo ajudaria na escolha da arquitetura do bloco lógico do FPGA.

Do mesmo modo, diversos trabalhos na área de arquiteturas reconfiguráveis foram analisados. Nesse sentido, verificou-se a vastidão do tema e, dessa forma, algumas decisões tiveram de ser realizadas a priori, a fim de reduzir o espaço de projeto e prover uma implementação em tempo hábil para a conclusão deste trabalho.

Outro ponto importante a se considerar é o desenvolvimento de ferramentas de CAD (*Computer Aided Design*) destinadas ao mapeamento de aplicações na arquitetura proposta. Na primeira versão da arquitetura foi desenvolvido um *micro-assembler*, com o objetivo de facilitar a programação das UEs (unidades de execução). Nas versões posteriores da arquitetura abandonou-se o desenvolvimento das ferramentas, pois o tempo necessário para desenvolvê-las seria alto. Dessa forma, o mapeamento dos circuitos alvo (*benchmarks*) foi realizado a mão.

Para a modelagem da arquitetura utilizou-se a linguagem VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) [BRO 2000]. O projeto foi organizado de forma modular, com descrição em alto nível dos blocos e, sempre que possível, parametrizável. Dessa forma, diversas versões do circuito puderam ser implementadas, permitindo uma gama maior de alternativas para a escolha da melhor arquitetura para a implementação final.

Após obtida uma versão da descrição VHDL da arquitetura, alguns algoritmos de teste e DSP eram mapeados na nova arquitetura e simulados (simulação funcional), a fim de validar a idéia proposta. Nessa etapa, caso alguns requisitos não fossem cumpridos, modificava-se a descrição VHDL e novas simulações eram feitas. Após a validação da arquitetura, otimizações foram realizadas visando a redução da área ocupada.

5.2 Requisitos da Arquitetura

A partir dos estudos realizados das técnicas de DSP e BIST (Caps. 2 e 4), pôde-se verificar algumas estruturas de hardware e conjunto de operadores necessários para a implementação de sistemas dessas classes de aplicações.

Para a implementação de filtros FIR, filtros IIR e DFT, necessita-se basicamente de somadores, subtratores, multiplicadores e registradores. Operações de deslocamento, comparação e multiplexagem 2x1 são também utilizadas em aplicações DSP [CHE 92].

Quanto às características de testabilidade, excluíram-se métodos de teste para máquinas de estados finitos (*Finite State Machines - FSMs*), uma vez que tal tipo de teste é possível mediante a recodificação dos próprios estados da máquina [LEE 96].

Nos algoritmos de teste de memórias RAM necessita-se basicamente de contadores, deslocadores e comparadores [GOO 93, TRE 93, ALV 95, CAM 95]. Além disso, tendo em vista a variabilidade nas larguras de endereçamento de memórias, foram definidos blocos que pudessem ser configurados com 4, 8, 16 e 32 bits, ou ainda em palavras maiores.

Para o teste de memórias ROM é necessário um contador, um acumulador e um comparador, se for utilizado o algoritmo checksum, ou então, um LFSR, se for executado um teste pseudo-aleatório. As mesmas considerações sobre a variabilidade nas larguras de endereço e dados em relação às memórias RAM também valem aqui.

Para o teste de partes operativas pode-se usar a técnica de BIST circular [STR 98], a qual utiliza operadores (somadores e multiplicadores, por exemplo) para a geração de vetores de teste e compactação da resposta. Essa técnica foi utilizada em [COT 99] para o teste da parte operativa do microcontrolador 8051. Além disso, como foi mostrado no Cap. 2, o suporte à operação de multiplicação seria muito adequado quando se estivesse utilizando a célula para processamento DSP ou testes de partes operativas.

Sabe-se, porém, que um operador de multiplicação é bastante custoso em termos de área, e a implementação desse operador em cada célula representaria um desperdício, já que para as aplicações de BIST dificilmente todos os multiplicadores estariam sendo utilizados ao mesmo tempo. Logo, com o objetivo de não aumentar em demasia o custo por célula, optou-se por não implementar o multiplicador diretamente em cada célula, mas sim, realizar agrupamentos de células, conectando-as através dos canais de roteamento, para a implementação de multiplicadores (conforme será visto na seção 6.4).

Finalmente, o tamanho da palavra de dados tem importância considerável pois, se for excessivamente grande, trará desperdício de área quando se necessitar trabalhar com palavras menores, como por exemplo em testes de memórias que

tenham largura de dados de 8 bits. Do mesmo modo, uma palavra de dados muito estreita implicará na necessidade de utilização de recursos extra de interconexão quando se for trabalhar com palavras maiores. A seguir, mostram-se duas abordagens adotadas para o desenvolvimento deste trabalho, discutindo-se sua aplicabilidade.

5.3 Primeira versão

Um dos principais gargalos na utilização de um FPGA é a área necessária para as interconexões dos seus diversos blocos componentes [DEH 96]. Logo, se o objetivo é reduzir a área gasta em conexões, então a escolha de um bloco lógico complexo e capaz de implementar funções de alto nível seria a mais adequada. Isto só é possível quando o conjunto de aplicações aceitar essa maior granularidade.

Dessa forma, em um estudo anterior a este trabalho [SOU 2000], foi projetada uma arquitetura de granularidade grossa estilo FPPA (*Field Programmable Processor Array*) [NUS 98]. Cada célula (bloco lógico) do FPPA era constituída por uma memória de programa (256 palavras de 32 bits), uma parte de controle e uma parte operativa (8 bits), formando um pequeno processador ou Unidade de Execução (UE) (Fig. 5.1).

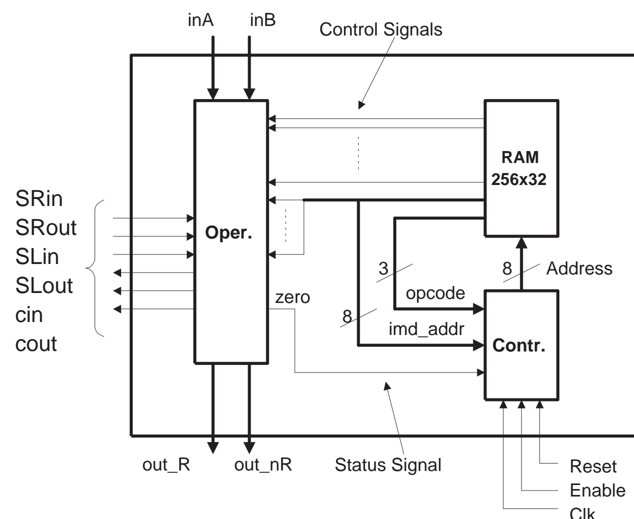

FIGURA 5.1 - Célula básica ou unidade de execução (UE).

A parte operativa da UE foi definida em função dos requisitos necessários para a implementação das funções BIST e DSP, conforme observado no estudo realizado a respeito dessas aplicações (Cap. 2 e 4). Os principais componentes da parte operativa (Fig. 5.2) eram: um banco de 8 registradores de 8 bits cada, uma unidade lógica e aritmética com as funções soma, AND, OR e XOR, três deslocadores e três inversores controlados.

A finalidade do banco de registradores locais era facilitar a divisão de tarefas e maximizar a localidade de processamento. O uso de registradores de entrada e

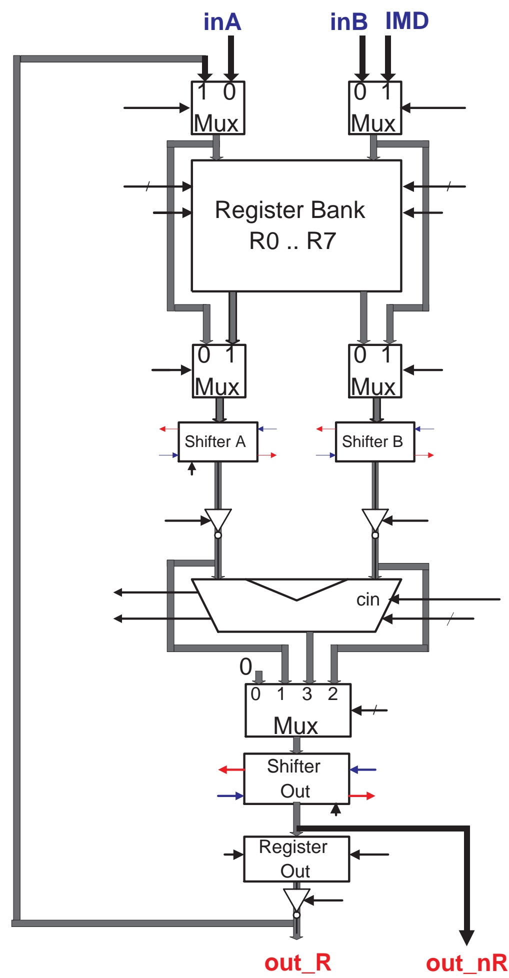

FIGURA 5.2 - Parte operativa da Unidade de Execução.

saída na ULA poderia ser definido a fim de permitir a implementação de pipeline com 2 estágios. No entanto, a versão inicial deste projeto possuía apenas um registrador na saída da célula [SOU 2000].

A interconexão (Fig. 5.3) do FPPA foi organizada de forma hierárquica, com agrupamentos (*clusters*) de 4 células que podiam ser cascateadas para formar partes operativas de largura maior (16, 24 ou 32 bits) [SOU 2000, GON 2001]. O objetivo dessa arquitetura de roteamento era diminuir a área gasta com conexões.

No entanto, após o mapeamento de alguns algoritmos de teste nessa arquitetura, verificou-se não ser necessária a parte de controle em todas as células, sendo necessários apenas alguns sinais de controle. Desse modo, iniciou-se um estudo para verificar a viabilidade de se eliminar a parte de controle, optando-se por uma nova estratégia de implementação [GON 2001a, GON 2001b, GON 2002a, GON 2002], baseada em uma granularidade mais fina, eliminando-se então a parte de controle da célula básica.

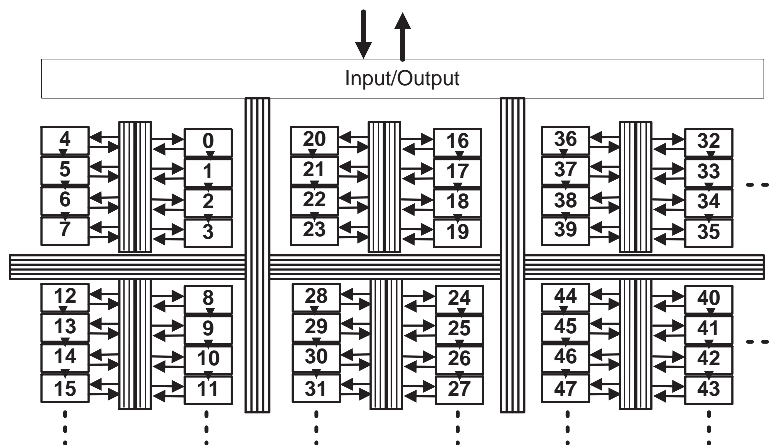

FIGURA 5.3 - Arquitetura de conexão.

5.4 Segunda Versão

Como foi dito anteriormente, a escolha de uma granularidade grossa para o bloco lógico (processador) não se mostrou muito adequada, em termos de área ocupada, para a implementação dos algoritmos estudados. Além disso, a granularidade grossa implicaria em elevado desperdício de área quando se desejasse implementar circuitos de baixa complexidade.

A escolha de uma granularidade fina (1 bit) também não seria adequada para a implementação de aplicações complexas, visto que seriam necessárias muitas células, aumentando o custo de roteamento e praticamente tendo-se os mesmos problemas de FPGAs comerciais. Logo, procurou-se encontrar um compromisso entre esses dois extremos (alta e baixa granularidade), que fosse capaz de resolver problemas no universo das aplicações de DSP e de teste de uma forma satisfatória.

Assim, adotou-se uma segunda abordagem para o desenvolvimento da arquitetura. Decidiu-se que o FPGA (*BiFi-FPGA - BIST and Filter FPGA*) seria composto de 3 partes, cada uma otimizada para a implementação de uma classe de circuitos (parte operativa, controle e memória, Fig. 5.4). Nesse sentido, ele segue a mesma idéia proposta por Cherepacha [CHE 94].

A diferença entre os dois trabalhos está justamente na arquitetura do bloco lógico (célula). Enquanto [CHE 94] propõe uma arquitetura para a implementação de partes operativas genéricas, este trabalho é mais específico, otimizando a célula para a implementação de partes operativas destinadas a aplicações DSP e BIST.

Entre as diferenças mais marcantes, pode-se destacar que o DP-FPGA (sec. 3.8.1) utiliza uma LUT para implementação de funções lógicas, enquanto o BiFi-FPGA utiliza uma ULA. Além disso, o BiFi-FPGA possui um bloco específico para geração de vetores pseudo aleatórios, o qual não está presente no DP-FPGA.

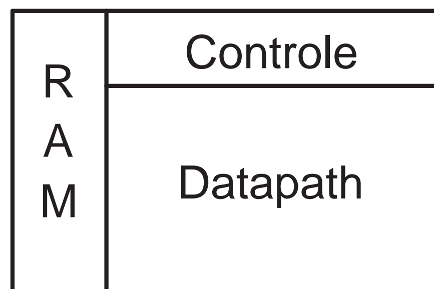

FIGURA 5.4 - Visão geral do BiFi-FPGA.

Este trabalho se concentra somente no bloco parte operativa (*datapath*), uma vez que a parte de controle é bem implementada pelos FPGAs de propósito geral, tais como Altera e Xilinx, segundo [CHE 94]. Com relação à memória, os FPGAs da Altera [ALT 2001, ALT 2001a, ALT 2002, ALT 2002b, ALT 2002a] já atacaram esse problema, embutindo blocos no interior dos FPGAS para a implementação de memórias, filas, etc., conforme foi mostrado na seção 3.7.

Como foi mostrado nos capítulos anteriores, as seguintes estruturas de hardware são necessárias para a implementação das funções de DSP e BIST: contadores, somadores/acumuladores, registradores, multiplicadores e portas XOR.

Além disso, analisando-se as aplicações alvo, pode-se verificar que a maior parte das operações se darão sobre dados múltiplos de 4, 8 ou 16 bits. Foi escolhida uma granularidade de 4 bits para o bloco lógico, visto que é o divisor comum entre 4, 8 e 16. Nas seções a seguir, apresenta-se em detalhes o bloco lógico proposto e a primeira fase de validação do mesmo.

5.5 Arquitetura do Bloco Lógico

Para a implementação da função lógica da célula base do BiFi-FPGA, diversas estratégias podem ser utilizadas: LUTs, multiplexadores, portas lógicas, ULAs, etc. Como foi visto no Cap. 3, os FPGAs tradicionais como Altera e Xilinx utilizam LUTs de 4 ou 5 entradas como componentes básicos do bloco lógico. Apesar de uma k-LUT (LUT de k entradas) poder implementar qualquer função de k entradas, esta não seria uma escolha adequada neste trabalho, pois a célula do BiFi-FPGA deverá implementar somente poucas funções básicas (como soma e XOR, por exemplo) e, a utilização de LUTs representaria um desperdício de área. Logo, projetou-se uma célula baseada em portas lógicas (na forma de uma ULA) e multiplexadores. A configuração do bloco lógico é feita através de 8 *latches* (células SRAM), como será visto na seção 5.6.

A Fig. 5.5 mostra a interface do bloco lógico bem como um esquemático simplificado, em blocos, contendo seus principais componentes (ULA, registrador de saída e Registrador-Deslocador-LFSR). Há dois tipos de sinais na interface da célula (Fig. 5.6): os sinais de dados (linhas cheias) e os sinais de controle (linhas pontilhadas). Os sinais de dados (Tab. 5.1) podem ser de 4 bits (linhas grossas) ou de 1 bit (linhas finas). Já os sinais de controle (Tab. 5.2) são todos de 1 bit.

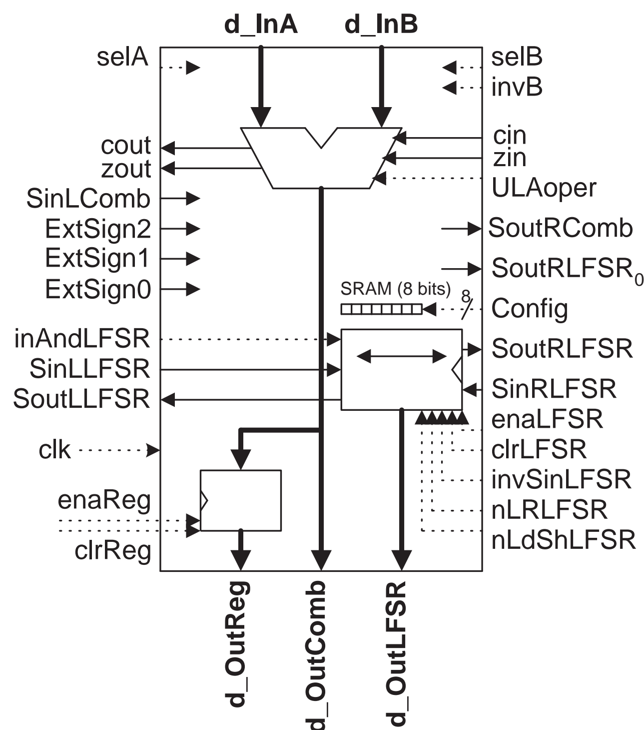

FIGURA 5.5 - Interface do bloco lógico.

A Fig. 5.6 mostra de forma detalhada a arquitetura do bloco lógico (célula), cujos principais componentes são: uma ULA, um registrador de saída (*RegOut*), um registrador-deslocador-LFSR (*ShiftRegLFSR*), um inversor e 4 multiplexadores

2x1. Todos esses blocos, bem como as entradas (*d_inA* e *d_inB*) e saídas (*d_OutReg*, *d_OutComb* e *d_OutLFSR*), são de 4 bits.

TABELA 5.1 - Sinais de dados do bloco lógico

Sinal	Função
d_InA	Entrada (operando A).
d_InB	Entrada (operando B).
d_OutComb	Saída combinacional (não-registrada).
d_OutReg	Saída registrada.
d_OutLFSR	Saída do bloco ShiftRegLFSR.
cin	Entrada de carry (carry-in).
cout	Saída de carry (carry-out).
zin	Entrada para cascateamento do sinal zout.
zout	Saída indicadora de zero.
SinLComb	Entrada combinacional esquerda, para implementação de multiplicador.
SoutRComb	Saída combinacional direita, para implementação de multiplicador.
SinLLFSR	Entrada serial esquerda do ShiftRegLFSR.
SinRLFSR	Entrada serial direita do ShiftRegLFSR.
SoutLLFSR	Saída serial esquerda do ShiftRegLFSR.
SoutRLFSR	Saída serial direita do ShiftRegLFSR.
SoutR0LFSR	Saída serial direita auxiliar do ShiftRegLFSR (flip-flop FF_0).

TABELA 5.2 - Sinais de controle do bloco lógico.

Sinal	Função
selA	Seleciona o operando A. 0 - Realimenta saída d_OutReg 1 - Entrada d_inA
selB	Seleciona o operando B. 0 - Constante Zero 1 - Entrada d_inB
invB	Controla inversão do operando B. 0 - Não inverte 1 - Inverte Obs.: Este sinal também controla o cin da ULA se $CONFIG_7$=$CONFIG_6$=1.
ULAoper	Seleciona a operação da ULA: 0 - Soma. 1 - XOR bit a bit.
clrReg	Zera registrador RegOut.
enaReg	Habilita carga do registrador RegOut.
clrLFSR	Zera ShiftRegLFSR e FF0.
enaLFSR	Habilita ShiftRegLFSR e FF0.
nLdShLFSR	0 - Habilita carga do ShiftRegLFSR. 1 - Executa um shift no ShiftRegLFSR. Obs.: Somente tem sentido se clrLFSR=0 e enaLFSR=1.
nLRLFSR	Sentido de deslocamento do ShiftRegLFSR: 0 - Desloca ShiftRegLFSR 1 bit para a esquerda. 1 - Desloca ShiftRegLFSR 1 bit para a direita. Obs.: Somente tem sentido se clrLFSR=0, enaLFSR=1 e nLdShLFSR=1.
invSinLFSR	Quando ativo (1), inverte os valores das entradas SinLLFSR e SinRLFSR. Útil na geração de padrões para teste de memórias RAM.
inANDLFSR	Entrada de realimentação do ShiftRegLFSR.
ExtSign0 ExtSign1 ExtSign2	Sinais de controle de extensão de sinal.

A Fig. 5.7 mostra o *bit-slice* da ULA, a qual foi implementada com base nos trabalhos desenvolvidos em [SUS 79, SUS 81]. A Fig. 5.8 mostra o diagrama completo da ULA, a qual consiste de 4 *bit-slices*, mais dois inversores para ajuste de *carry*, um inversor para ajuste de *zin* e uma porta NOR de 5 entradas para a

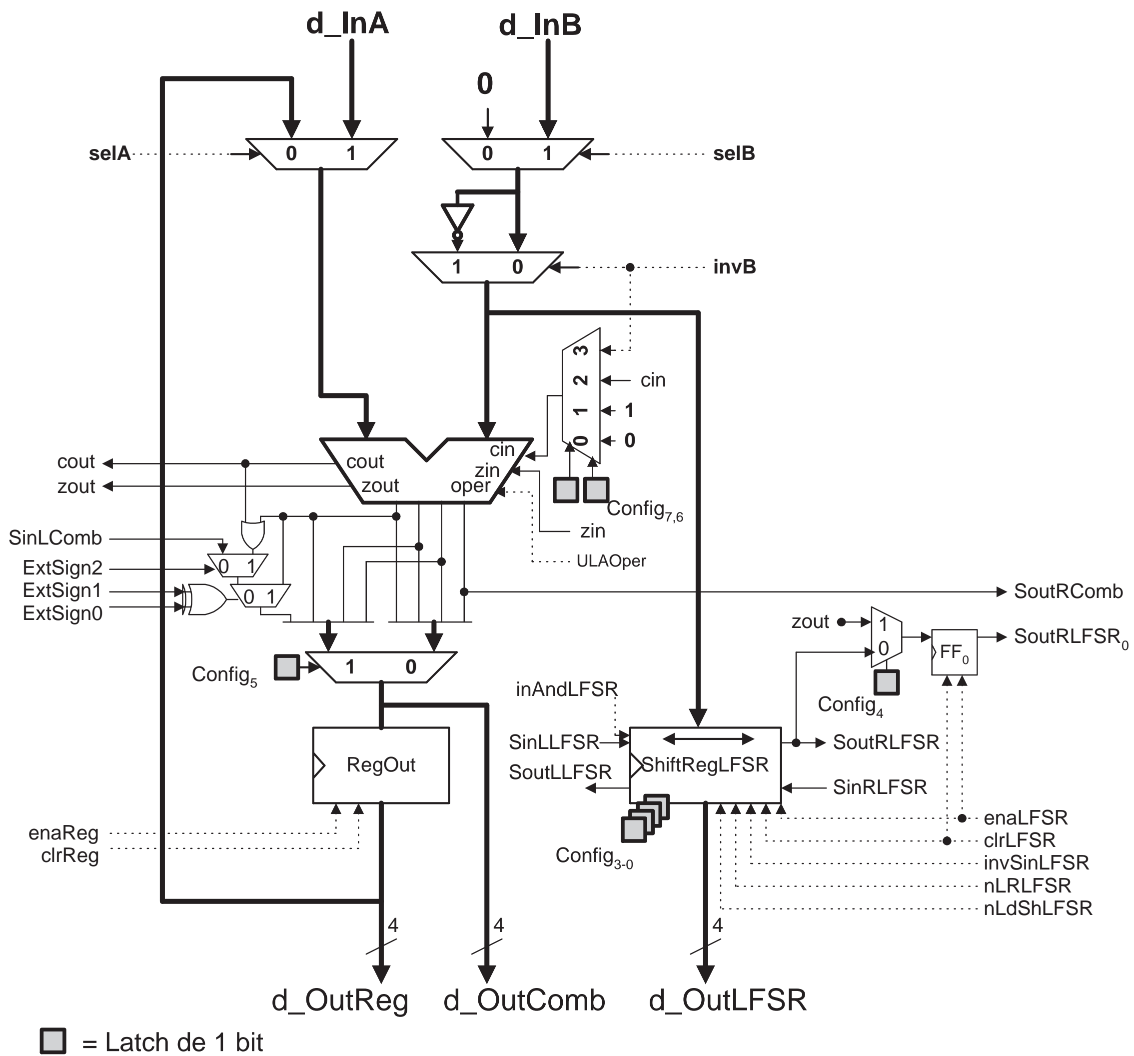

FIGURA 5.6 - Arquitetura do bloco lógico.

detecção de zero (*Zout*).

A ULA pode executar duas funções, soma e XOR bit a bit, selecionáveis através do sinal de controle *ULAoper* (*ULAoper=0* para soma ou *ULAoper=1* para XOR). A operação de subtração (*d_inA-d_inB*) é possível através da inversão do operando *d_inB* e configuração do *cin* da ULA para receber o valor lógico 1 (um) (Tab. 5.3).

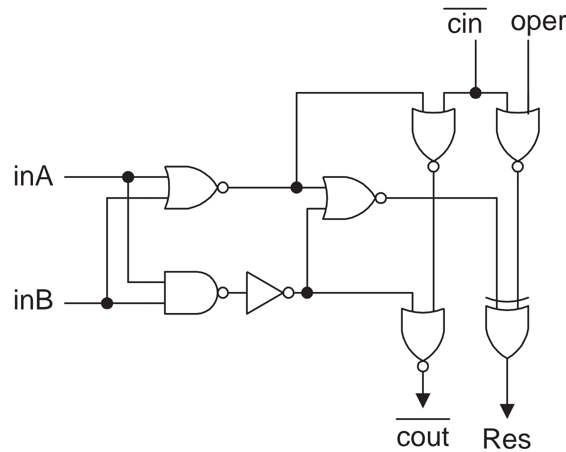

FIGURA 5.7 - Bit-slice da unidade lógica e aritmética.

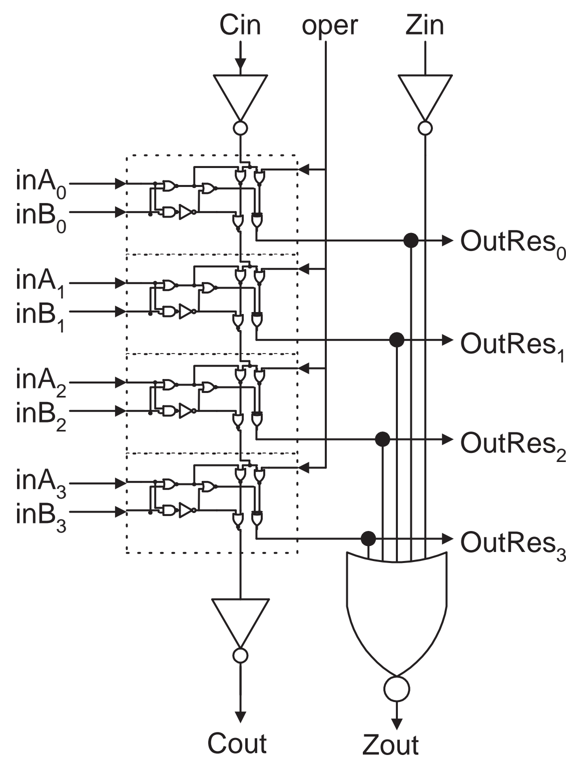

FIGURA 5.8 - ULA completa (4 bits).

O bloco *RegOut* na saída *d_OutReg* da Fig. 5.6 é simplesmente um registrador, podendo ser utilizado para implementação de contadores e somadores/acumuladores, por exemplo.

TABELA 5.3 - Controle do cin da ULA.

$\mathbf{CONFIG_7}$	$\mathbf{CONFIG_6}$	**Resultado cin_ULA**
0	0	0
0	1	1
1	0	cin
1	1	invB

O registrador-deslocador *ShiftRegLFSR*, mostrado em detalhe na Fig. 5.9, pode ser configurado como um registrador, como um registrador-deslocador (direita ou esquerda) ou ainda como um LFSR configurável. A Tab. 5.4 mostra as possíveis configurações para esse bloco. A configuração do LFSR dá-se através dos *latches* de configuração $CONFIG_0$ a $CONFIG_3$. Nesse caso, o LFSR implementa o seguinte polinômio

$$g(x) = x^4 + g_3\, x^3 + g_2\, x^2 + g_1\, x^1 + g_0 \tag{5.1}$$

onde
$g_i = CONFIG_i$

TABELA 5.4 - Possíveis configurações para o bloco ShiftRegLFSR.

Função	**enaLFSR**	**nLdShLFSR**	**nLRLFSR**	$\mathbf{CONFIG_{3_0}}$ **e inAND**
Desabilitar registrador	0	x	x	xxxx e inAND=x
Carregar registrador	1	0	x	0000 ou inAND=0
Shift para esquerda	1	1	0	0000 ou inAND=0
Shift para direita	1	1	1	0000 ou inAND=0
LFSR	1	1	1	abcd* e inAND=1

* abcd = bits de configuração do LFSR.

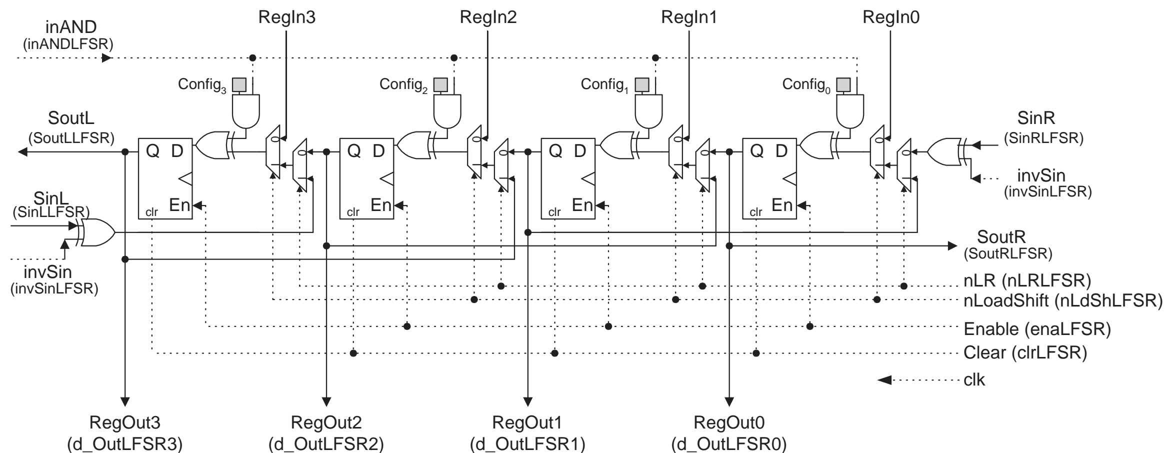

FIGURA 5.9 - Bloco ShiftRegLFSR.

O bloco *ShiftRegLFSR* é utilizado também para a geração dos padrões de teste durante o teste de memórias RAM. Durante a implementação de multiplicadores seriais, este registrador armazena o operando multiplicador, como será visto a seguir. Além disso, este registrador pode ser utilizado para a implementação de técnicas de teste baseadas em varredura (*scan chain*) [ABR 90, REN 2001].

5.6 Configuração do Bloco Lógico

Para armazenamento da configuração da célula, utilizaram-se 8 *latches* (sinal CONFIG_{7_0}), tendo estes as seguintes funções:

- CONFIG_{3_0}: Bits de configuração do LFSR
- CONFIG_4: Seleciona a entrada do *flip_flop* FF_0. CONFIG_4=1 permite que se tenha a saída *zout* registrada. CONFIG_4=0 seleciona *SoutRLFSR*, permitindo implementar, juntamente com o bloco *ShiftRegLFSR*, um registrador de deslocamento de 5 bits.
- CONFIG_5: Permite selecionar o valor a ser colocado na entrada do registrador de saída (*RegOut*) ou saída *d_OutReg*. Pode ser a saída da ULA ou então a saída da ULA deslocada 1 bit para a direita. Esta última opção é utilizada na implementação dos multiplicadores.
- $\mathrm{CONFIG}_{7,6}$ (Lógica de *carry*). Permite colocar em '1' ou '0' o *carry-in* da ULA, habilitar o cascateamento com outra célula ou conectar o *cin* com o sinal *invB* (Tab. 5.3).

5.7 Arquitetura de Roteamento

O presente trabalho tem por objetivo o projeto do bloco lógico do FPGA, ficando em aberto a definição da arquitetura de roteamento. No entanto, considerando-se as características do FPGA aqui proposto, pode-se dizer que uma arquitetura baseada em canais de roteamento horizontais e verticais (Sec. 3.5), tal como o utilizado no DP-FPGA (Sec. 3.8.1) é uma opção factível e muito interessante. Além disso, a arquitetura deve possuir segmentos dedicados para o cascateamento dos sinais de *carry* e *shift* (*cout, zout, SLout, SRout,* etc.).

5.8 Descrição VHDL

Para a modelagem da arquitetura utilizou-se a linguagem VHDL (*Very-High Speed Integrated Circuit Hardware Description Language*) [BRO 2000]. O projeto foi organizado de forma estrutural, com módulos parametrizáveis sempre que possível. A Fig. 5.10 mostra a hierarquia dos arquivos, os quais são identificados a seguir:

- po_cell - Bloco Lógico ou célula básica, é o arquivo de topo de projeto.
- ula_nbit - ULA de 4 bits

- ula_1bit - Bit-slice da ULA
- shift_reg_lfsr - Shift Register de 4 bits (*Left or Right*) e LFSR configurável
- reg_up_clr - Registrador de 4 bits, ativo na borda de subida, reset assíncrono.
- ff_up_clr - Flip-flop de 1 bit, ativo na borda de subida, reset assíncrono.

Após feita a descrição do bloco lógico (*po_cell*) do BiFi-FPGA, iniciou-se a primeira fase de validação do mesmo, a qual é apresentada a seguir.

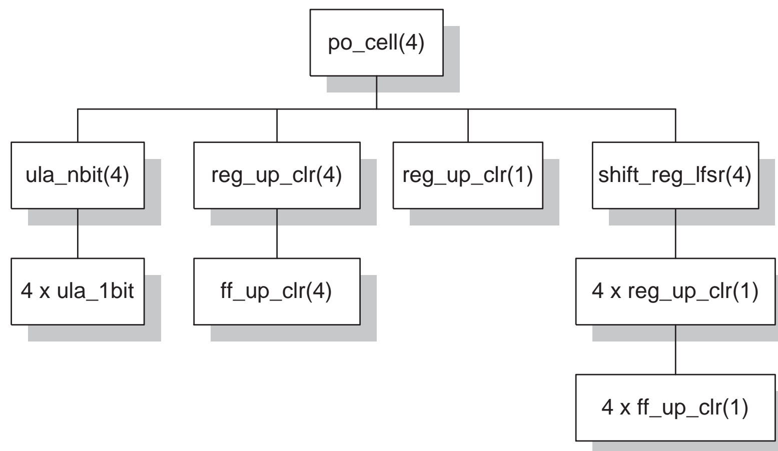

FIGURA 5.10 - Hierarquia de arquivos.

5.9 Implementação de Blocos Básicos

Como primeira etapa de validação, foram implementados os diversos blocos básicos (contadores, acumuladores, subtratores, comparadores e deslocadores) necessários nas aplicações DSP e BIST, utilizando-se o bloco lógico aqui proposto. Os multiplicadores serão apresentados no próximo capítulo.

Visto que cada célula manipula um conjunto de 4 bits, quando se desejar implementar uma parte operativa maior, será necessário fazer o cascateamento de várias células. Para as funções contador, acumulador e subtrator, é necessário fazer o cascateamento da lógica de *carry*. Logo, nos exemplos a seguir são mostradas duas configurações de célula: uma para os quatro bits menos significativos (3-0) e outra para os demais bits. A diferença entre as duas configurações está basicamente na lógica de *carry*.

5.9.1 Contador

Um contador pode ser implementado configurando-se a ULA para realizar a operação soma e colocando-se um 1 lógico no *cin* da primeira célula (a que irá implementar os bits 3-0) (Fig. 5.11a). Para as demais células (Fig. 5.11b) deve-se fazer o cascateamento com o *carry-out* da célula anterior. O valor da contagem é

fornecido pela saída *d_OutReg*.

É interessante observar que para a implementação de contadores módulo 4N, nesta configuração, não são utilizadas as entradas *d_inA* e *d_inB*, ou seja, não é necessário prover-se quaisquer sinais nessas entradas. Dessa forma, não é necessário se utilizar o canal de roteamento para fornecer dados para essas entradas, o que poderia economizar segmentos de fios, que poderiam ser utilizados para roteamento de outras células.

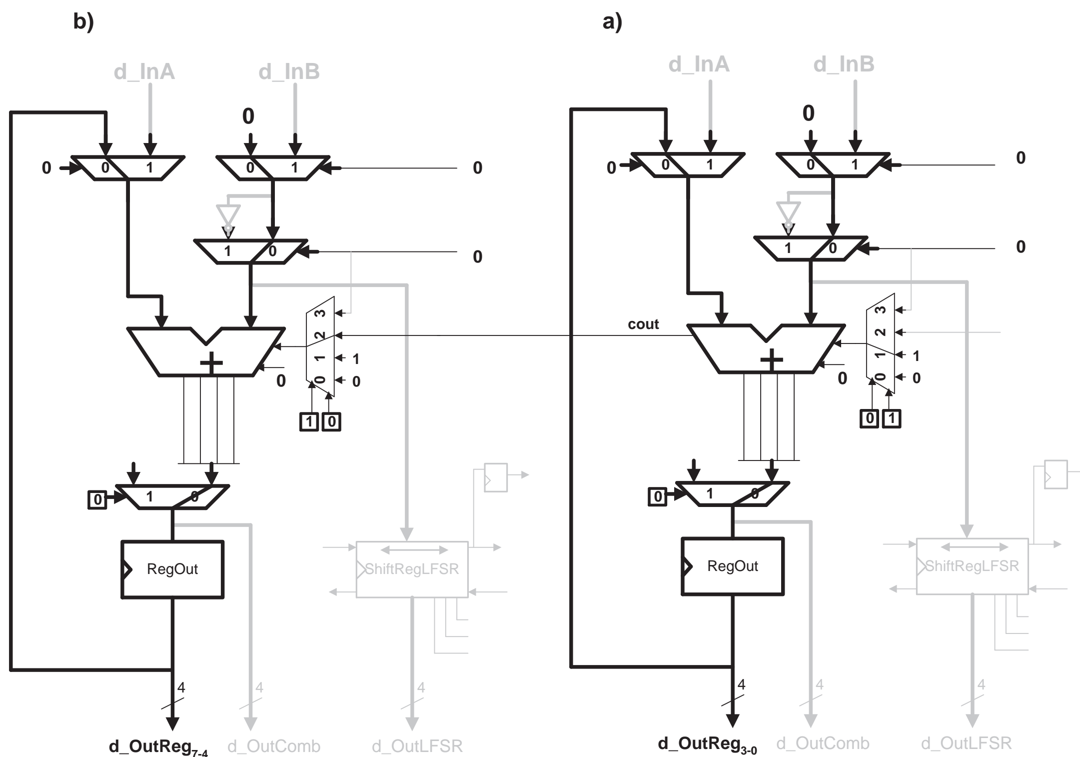

FIGURA 5.11 - Configuração contador: bits 3-0 (a) e demais bits (b).

5.9.2 Acumulador

Da mesma forma que no exemplo anterior, duas configurações são necessárias para a implementação de um acumulador (Fig. 5.12). Os valores a serem acumulados devem ser fornecidos pela entrada *d_inB* e o resultado é obtido na saída *d_OutReg*.

5.9.3 Comparador

Um comparador é implementado utilizando-se a função XOR bit a bit da ULA. Os operandos a serem comparados devem ser fornecidos pelas entradas *d_inA* e *d_inB*. A saída *Zout* fornecerá o valor 1 (um lógico) quando os operandos forem iguais ou 0 (zero lógico) se eles forem diferentes. Para comparadores maiores que 4 bits deve-se colocar um 1 lógico na entrada *zin* da primeira célula e fazer o cascateamento do sinal *zout* com as células seguintes. O sinal indicador de igualdade é obtido no *zout* da última célula, conforme pode ser visto na Fig. 5.13.

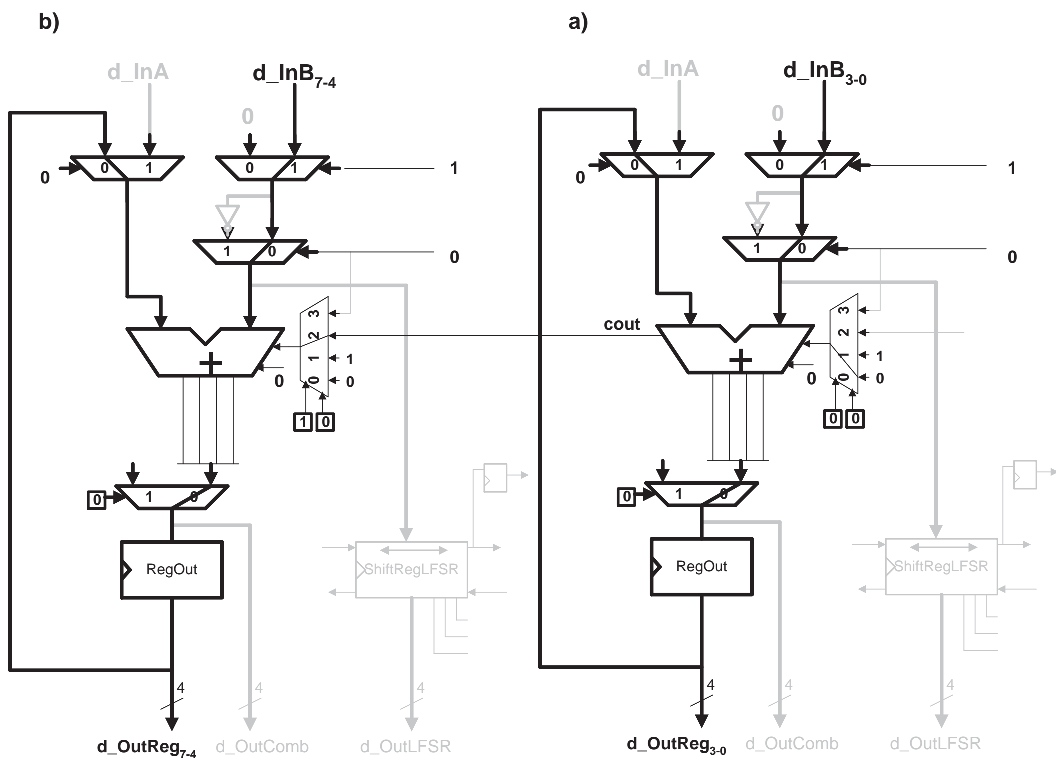

FIGURA 5.12 - Configuração acumulador: bits 3-0 (a) e demais bits (b).

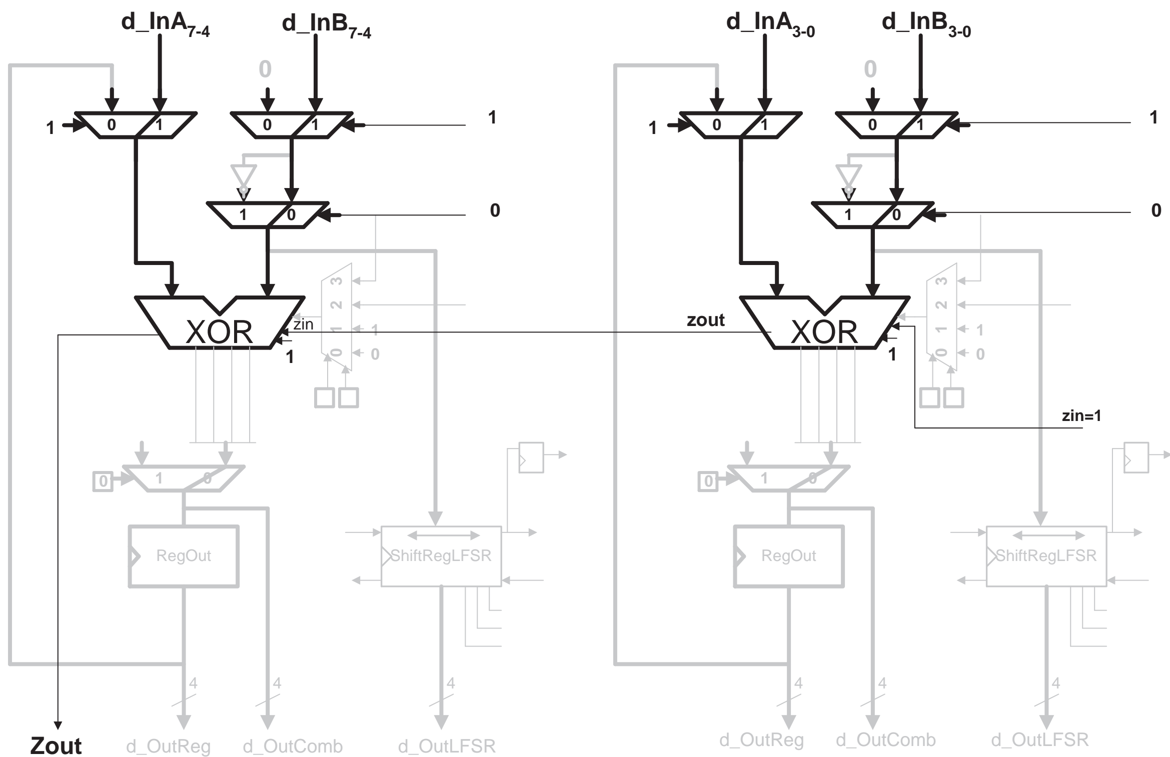

FIGURA 5.13 - Configuração comparador.

5.9.4 Subtrator

A operação d_inA - d_inB é possível através da soma do operando d_inA com o complemento de 2 do operando d_inB. O complemento de 2 do operando d_inB é conseguido através de sua inversão (sinal invB=1) e somando-se 1 (cin=1) a esse resultado. O resultado da subtração é fornecido na saída *d_OutComb*. A Fig. 5.14 mostra essa configuração.

5.9.5 Registrador-Deslocador

Para a implementação de um registrador-deslocador utiliza-se o bloco *ShiftRegLFSR*. Primeiramente, habilita-se o sinal de carga do registrador (*nLdShLFSR=0*) e, através da entrada *d_inB*, é carregado um valor de inicialização no registrador (Fig. 5.15a). Num segundo momento, o sinal de carga é desabilitado (*nLdShLFSR=1*), e o registrador começa a funcionar como um deslocador (Fig. 5.15b). É interessante observar que após a inicialização do registrador, este pode ser utilizado concorrentemente com os outros blocos da célula, sem interferir no funcionamento da mesma. Dessa forma, é possível implementar em uma única célula, por exemplo, um contador e um deslocador, e os dois podem trabalhar concorrentemente. Com isso, pode-se implementar uma cadeia de varredura (*scan chain*) para se observar os valores de algumas células. Para isso, basta conectar serialmente os blocos *ShiftRegLFSR* das células que se deseja observar.

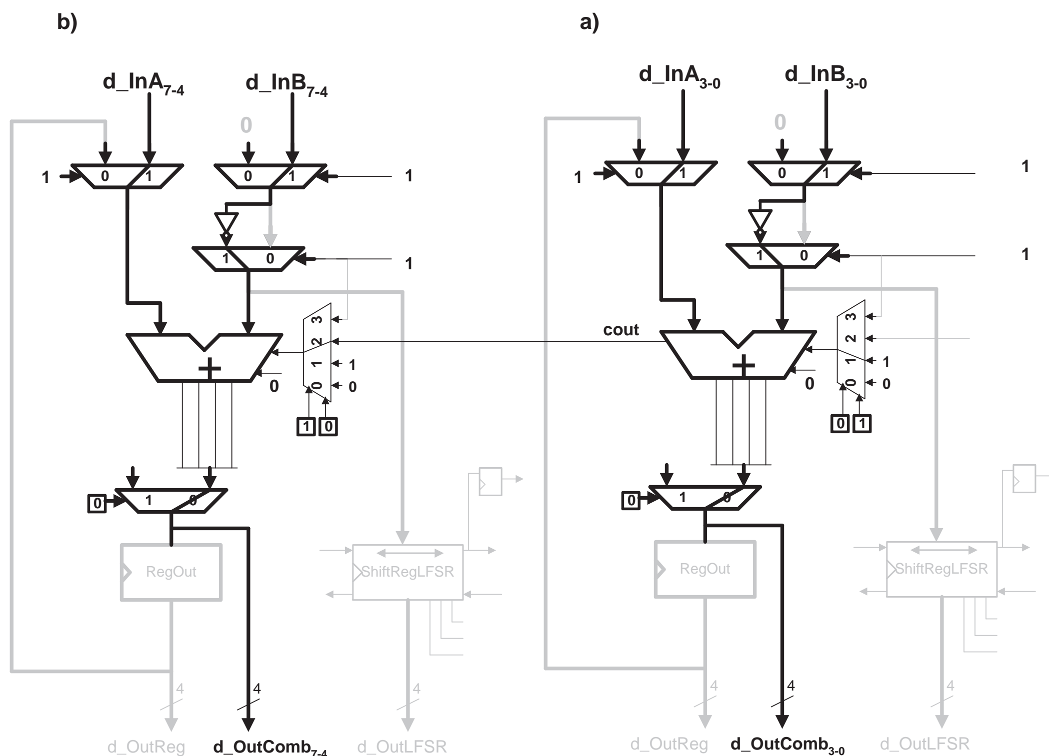

FIGURA 5.14 - Configuração subtrator: bits 3-0 (a) e demais bits (b).

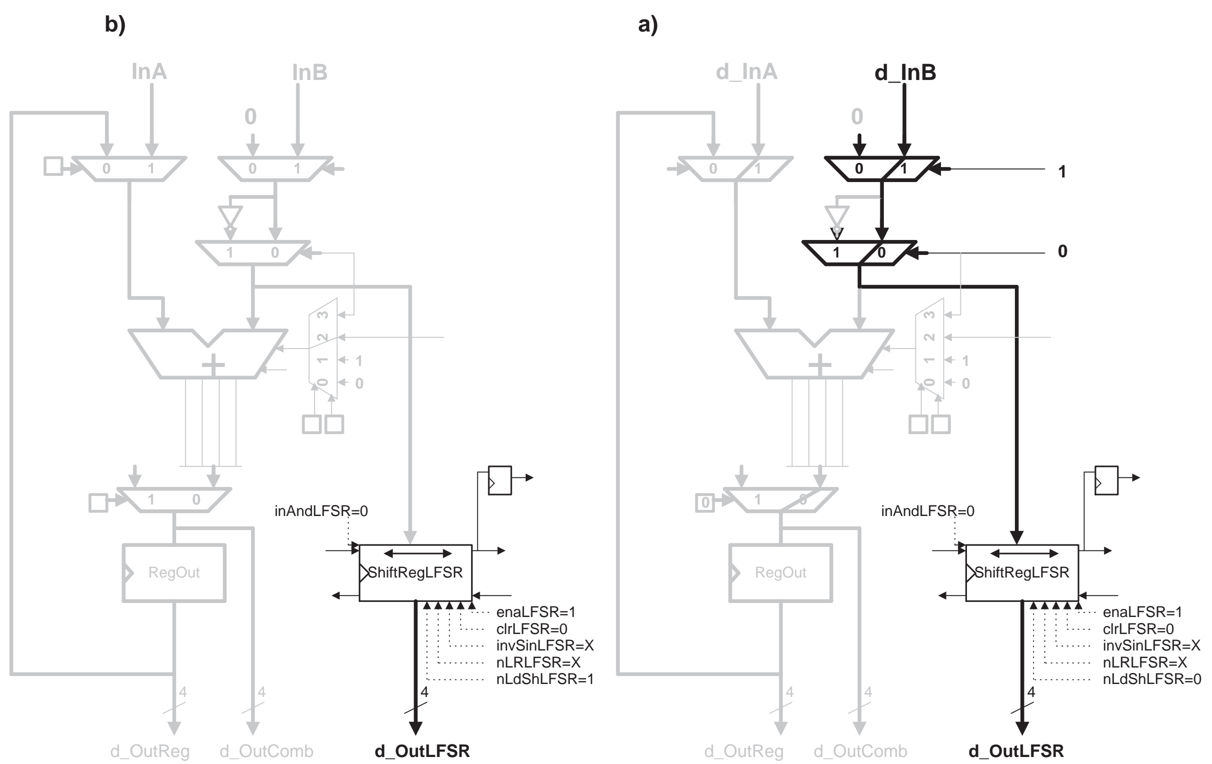

FIGURA 5.15 - Registrador-Deslocador: carga do registrador (a) e bloco funcionando como deslocador (b).

5.10 Considerações Sobre a Célula Base do BiFi-FPGA

Neste capítulo foi mostrada a proposta de um novo FPGA, constituído de três partes (operativa, controle e memória). Visto que o escopo deste trabalho é a definição da parte operativa, mostrou-se então, de forma detalhada, a arquitetura do bloco lógico (célula). Também foi mostrado que essa célula permite a fácil implementação dos diversos blocos básicos (somadores, contadores e comparadores) necessários na construção dos sistemas maiores de DSP e BIST. Dessa forma, basta conectar as células do BiFi-FPGA para se implementar os circuitos de DSP e BIST desejados.

No próximo capítulo continua-se a validação do BiFi-FPGA, com a implementação e simulação de multiplicadores bem como de circuitos DSP e BIST.

6 Validação da Arquitetura

Conforme foi mostrado no Cap. 5, a primeira etapa para a validação da idéia proposta foi realizar o mapeamento de blocos básicos utilizando as células do BiFi-FPGA. Após o mapeamento e simulação funcional desses blocos, iniciou-se a implementação de diversos circuitos de aplicação alvo (*benchmark*) utilizando as células do BiFi-FPGA. Portanto, o primeiro passo foi definir quais seriam os circuitos alvo a serem utilizados. Visto que o BiFi-FPGA é destinado à implementação de circuitos DSP e BIST, os seguintes ensaios foram definidos:

- BIST_ROM_AxD: BIST de memória ROM utilizando o algoritmo *checksum* mostrado na seção 4.5.1 (Fig. 4.4). *A* e *D* representam a largura de endereço e dados, respectivamente. Por exemplo, BIST_ROM_8x16 representa o BIST de uma memória ROM com 256 palavras de 16 bits;
- BIST_RAM_AxD: BIST de memória RAM utilizando o algoritmo March. *A* e *D* representam a largura de endereço e dados, respectivamente;
- LFSR_N: *Linear Feedback Shift Register* de *N* bits;
- Mult_Par_N: Multiplicador paralelo NxN bits;
- Mult_Ser_N: Multiplicador serial NxN bits;
- FIR_Can_Par_TxN: Filtro FIR canônico (T taps) utilizando multiplicador paralelo NxN bits (*Mult_Par_N*) (Fig. 2.1);
- FIR_Can_Ser_TxN: Filtro FIR canônico (T taps) utilizando multiplicador serial NxN bits (*Mult_Ser_N*) (Fig. 2.1);
- FIR_nCan_Par_TxN: Filtro FIR não-canônico (T taps) utilizando multiplicador paralelo NxN bits (*Mult_Par_N*) e unidades para endereçamento de memória (Fig. 2.4).
- FIR_nCan_Ser_TxN: Filtro FIR não-canônico (T taps) utilizando multiplicador serial NxN bits (*Mult_Ser_N*) e unidades para endereçamento de memória (Fig. 2.4).

Devido ao tempo disponível para término deste trabalho, não foi possível realizar ensaios com implementação de DFT. No entanto, como foi mostrado na sec. 2.3, as estruturas básicas necessárias para sua implementação são um somador, um subtrator, um multiplicador e uma memória para armazenamento dos termos seno e coseno. A simulação e validação de somadores e subtratores já foi mostrada na sec. 5.9. Como será visto nas seções a seguir, também foram simulados e validados diversos tipos de multiplicadores. A memória, deverá ser implementada na seção específica do BiFi-FPGA. Isso leva a crer que é perfeitamente possível a implementação da DFT utilizando-se as células do BiFi-FPGA.

A Fig. 6.1 mostra a metodologia adotada para a simulação dos circuitos exemplos de BIST: Um circuito sob teste (*Circuit Under Test - CUT*) e

seu respectivo BIST são descritos em VHDL e simulados. O controlador BIST é composto de uma parte de controle e uma parte operativa. Como já foi mencionado, o foco deste trabalho está na definição da parte operativa do BiFi-FPGA. Logo, somente as partes operativas dos circuitos foram implementadas no BiFi-FPGA. As partes de controle necessárias foram descritas em um bloco VHDL separado, que, na prática, seria implementado na região do BiFi-FPGA destinada à implementação de partes de controle (vide Fig. 5.4). Como outra alternativa, poderia-se implementar a parte de controle de forma dedicada e simplesmente conectá-la aos sinais de controle do bloco lógico do BiFi-FPGA.

A seguir, mostra-se cada um dos circuitos de exemplo mapeados no BiFi-FPGA e simulados em VHDL.

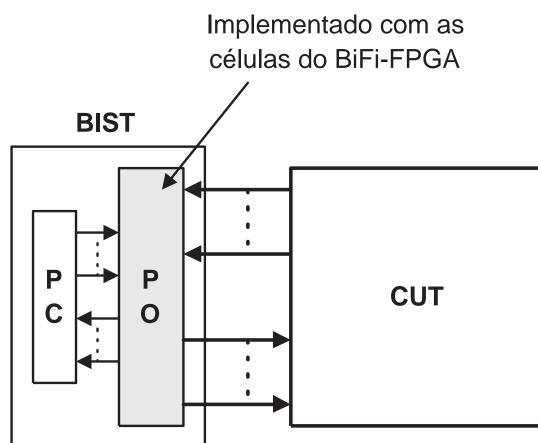

FIGURA 6.1 - CUT (*Circuit Under Test*) e BIST associado.

6.1 BIST de ROM

Conforme visto na seção 4.5.1, os blocos funcionais necessários para a implementação de BIST de ROM utilizando o algoritmo *checksum* são: um contador, um acumulador e um comparador. Na seção 5.9 já foi mostrada a implementação desses blocos em separado. Agora, abstraindo-se os detalhes da célula, na Fig. 6.2 é mostrada a implementação em blocos da parte operativa do algoritmo *checksum* para o teste de uma memória de 16 palavras de 4 bits. Neste exemplo foram utilizadas três células do BiFi-FPGA.

Antes de iniciar o algoritmo, o valor esperado para o *checksum* é armazenado no *ShiftRegLFSR* da célula *Contador*. No final do algoritmo, esse valor é comparado pela célula *Comparador* com o valor contido no *RegOut* da célula *Acumulador* (*checksum* calculado). A saída *Zout* da célula *Comparador* indica o resultado do *checksum* (1 para *checksum* correto e 0 para *checksum* incorreto).

A Fig. 6.3 mostra a forma de onda obtida na simulação do *checksum* em uma memória ROM de 16 palavras de 4 bits que não possui defeitos, contendo os seguintes

valores armazenados: 0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13 e 15. O valor correto do *checksum* é 8, pois é realizada uma soma truncada em 4 bits:

$$0+2+4+6+8+10+12+14+1+3+5+7+9+11+13+15 = 120$$
$$120 \mod 4 = 8$$

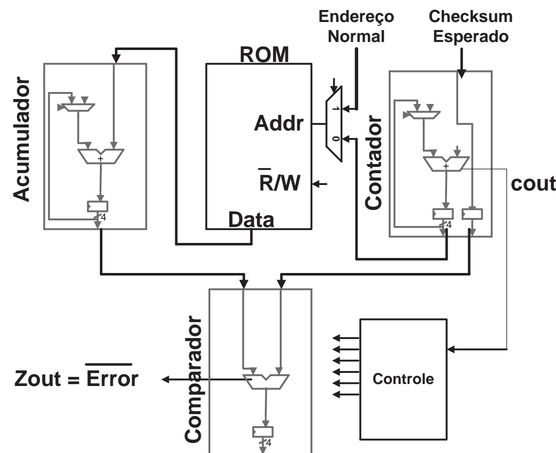

FIGURA 6.2 - BIST de ROM.

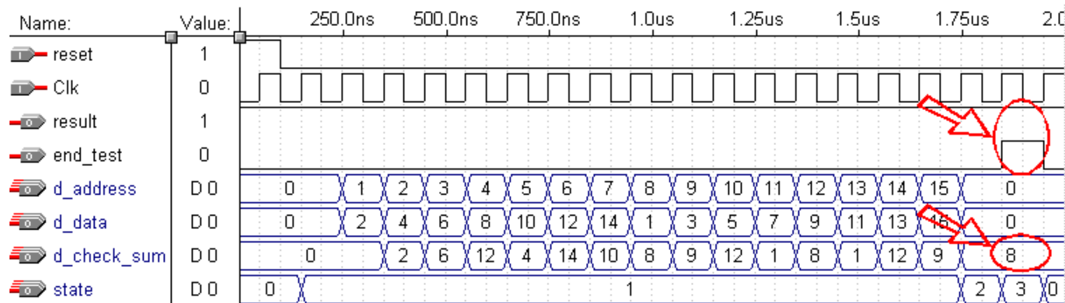

FIGURA 6.3 - Simulação do algoritmo Checksum em ROM sem defeito.

Na Fig. 6.4 é mostrada a simulação de uma falha na palavra localizada no endereço 2 da ROM (trocou-se o valor 4 por 3).

$$0+2+3+6+8+10+12+14+1+3+5+7+9+11+13+15 = 119$$
$$119 \mod 4 = 7$$

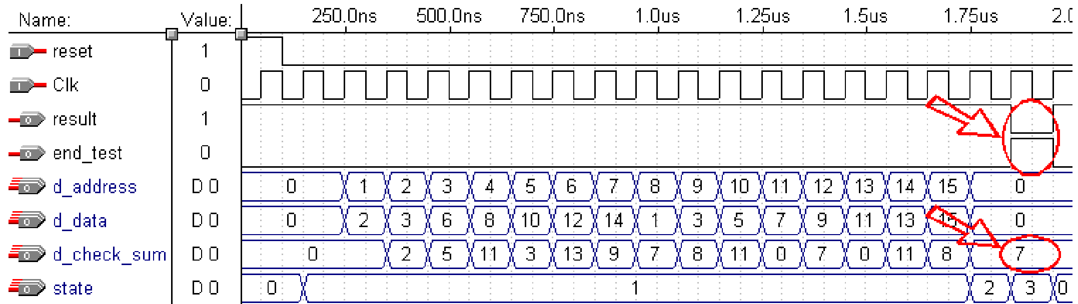

FIGURA 6.4 - Simulação do algoritmo Checksum em ROM com defeito.

6.2 BIST de RAM

Conforme o exemplo de BIST de RAM apresentado na seção 4.5.2, os seguintes blocos funcionais são necessários para sua implementação: um contador, um comparador e um gerador de padrões (*shift register*). Uma possível implementação desse algoritmo, para o teste de uma memória RAM 16x4, utilizando-se o BiFi-FPGA pode ser vista na Fig. 6.5. Como se pode constatar, 3 células do BiFi-FPGA são necessárias para esta implementação.

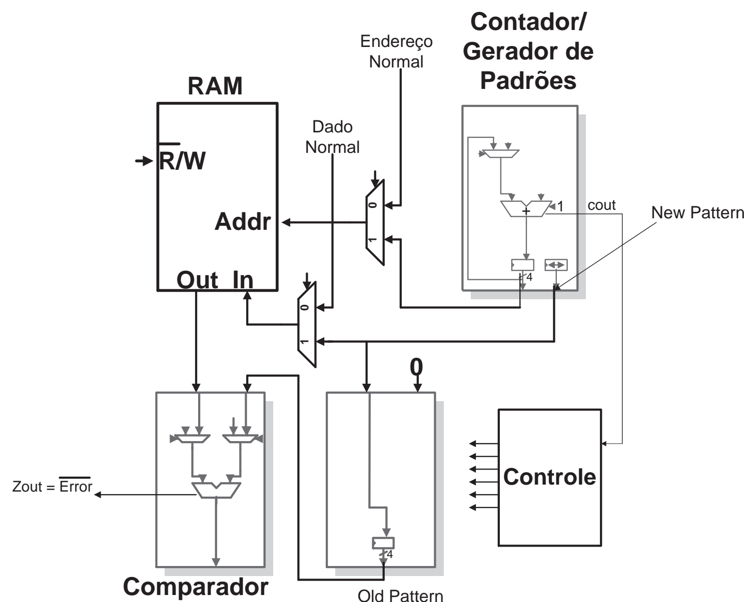

FIGURA 6.5 - BIST de RAM.

Observando-se os padrões *Marching* e *Walking* apresentados na seção 4.5.2 (Tab. 4.3), pode-se verificar que todos eles podem ser gerados com um registrador de deslocamento. Portanto, o *ShiftRegLFSR* do bloco lógico do BiFi-FPGA pode ser utilizado para gerar esses padrões.

Como não teria sentido a geração exaustiva de todos esses padrões, escolheu-se um algoritmo mais simples para o teste de RAM. Dessa forma, implementou-se o mesmo algoritmo utilizado em [COT 99] para o teste da RAM do microcontrolador 8051. Nesse algoritmo são gerados os seguintes vetores de teste: 0000, 0001, 0011, 0111, 1111, 1110, 1100, 1000 e 0000.

A Fig. 6.6 mostra as formas de onda da simulação desse algoritmo, onde foi injetada uma falha de colagem em zero no bit 2 da palavra 6 da memória. Os vetores a serem escritos na RAM estão indicados pelo vetor *NewPattern* (valores destacados na Fig. 6.6). Na primeira fase do algoritmo, a memória é inicializada com 0000. Depois, ocorre uma seqüencia de leitura do padrão anteriormente armazenado e escrita do novo padrão. A escrita dos vetores 0001, 0011 e 0111 ocorre normalmente. A escrita do vetor 1111 (*NewPattern*) ocorre normalmente até a posição 5 da memória. Na posição 6, é feita uma leitura do dado contido nesse endereço, obtendo-se o valor 0011 (*q_out*). O valor esperado nessa leitura era o vetor 0111 (*OldPattern*). A célula *Comparador* faz a comparação desses dois valores, e como os mesmos não são iguais, o sinal *zout* dessa célula estará com o valor lógico zero (*data_equal_out=0*). Esse sinal é passado à parte de controle que ativa os sinais *end_test* para finalizar o teste e *error_out* para indicar que um erro ocorreu.

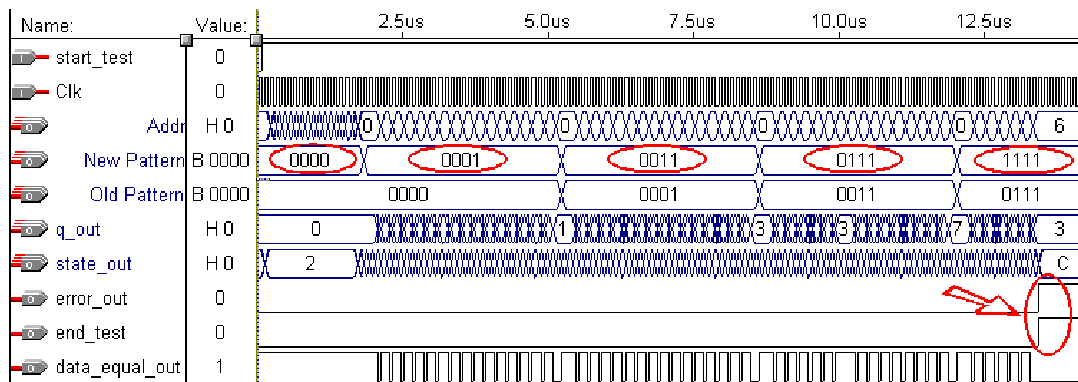

FIGURA 6.6 - Formas de onda da simulação do teste de RAM.

6.3 Geração de Vetores Pseudo-Aleatórios

Para a implementação de um LFSR ou de um analisador de assinaturas ultiliza-se o bloco *ShiftRegLFSR* da célula lógica do BiFi-FPGA. Cada bloco *ShiftRegLFSR* implementa 4 bits. Para LFSRs maiores, basta utilizar mais células e conectar apropriadamente os sinais *SoutLLFSR*, *SinRLFSR* e *inANDLFSR*, conforme mostra a Fig. 6.7, para o caso de um LFSR de 12 bits.

A Fig. 6.8 mostra os vetores gerados por um LFSR de 4 bits, inicializado com o valor 6 (0110_2) e implementando o polinômio $x^4 + x + 1$ ($CONFIG_{3_0} = 0011$). Como se pode oberservar na Fig. 6.8, com esse polinômio são gerados todos os valores possíveis entre 1 e 15 (6, C, B, 5, A, 7, E, F, D, 9, 1, 2, 4, 8, 3), e a seqüência repete-se ao longo do tempo (a partir de $1.65\mu s$).

A Fig. 6.9 mostra alguns dos vetores gerados por um LFSR de 8 bits, implementado com duas células. O LFSR implementa o polinômio $x^8 + x^6 + x^5 + x + 1$ ($configLFSR = 01100011$) e foi inicializado com o valor 33.

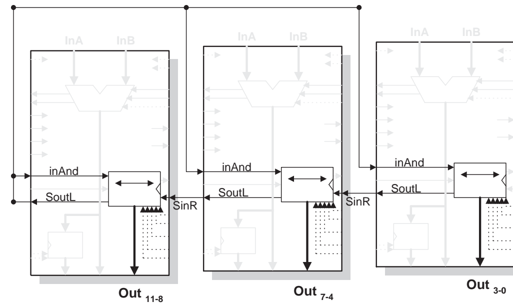

FIGURA 6.7 - Três células implementando um LFSR de 12 bits.

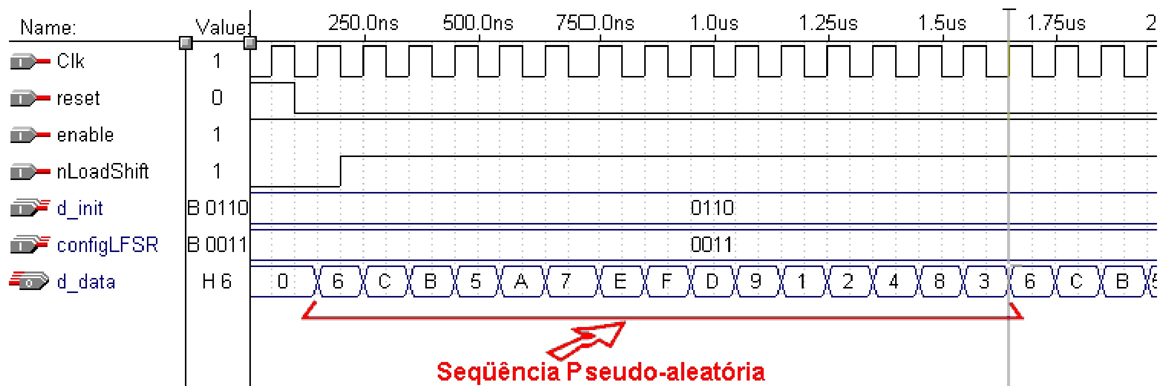

FIGURA 6.8 - Simulação de um LFSR de 4 bits.

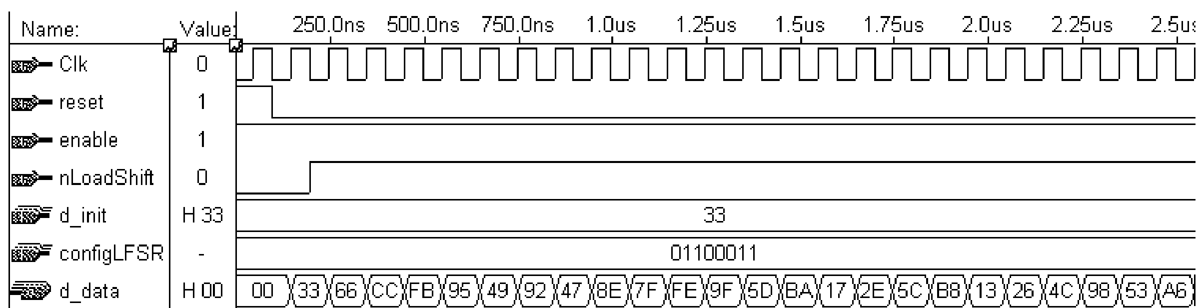

FIGURA 6.9 - Simulação de um LFSR de 8 bits.

6.4 Multiplicador

Como já foi dito anteriormente, embora seja necessário um operador de multiplicação, decidiu-se não implementar esse operador diretamente na célula, visto que esse tipo de estrutura consome muita área, e haveria um desperdício

quando não se estivesse utilizando a célula para a implementação de filtros. Além disso, com o objetivo de fornecer um espaço de projeto maior, decidiu-se implementar dois tipos de multiplicadores (paralelo e serial), um visando maior desempenho (multiplicador paralelo) e o outro visando menor área (multiplicador serial). Outra decisão tomada foi quanto ao formato dos operandos, decidindo-se que o multiplicador deveria ser capaz de trabalhar com números com sinal (complemento de dois).

Tendo sido escolhido que a arquitetura do BiFi-FPGA fosse regular, ou seja, todas as suas células seriam iguais, e que a granularidade da célula seria de 4 bits, tornou-se impossível a implementação eficiente dos multiplicadores tipo *Booth*, *Baugh Wookey* e *Wallace Tree* [BRO 2000, WES 93, BEL 95]. Isto deve-se ao fato desses multiplicadores possuírem algumas não-regularidades, tais como, tamanho de dados diferentes (no caso do *booth*, operandos de 4 bits e produtos parciais de 5 bits, por exemplo) e de terem a necessidade de mais de um tipo de somador (*Baugh Wookey* e *Wallace Tree*), o que implicaria em modificações drásticas no bloco lógico do BiFi-FPGA.

Logo, com o objetivo de não se fazer modificações drásticas na arquitetura do BiFi-FPGA, decidiu-se que a multiplicação seria feita através da soma de produtos parciais de um dos operandos e seria utilizada uma lógica adicional para se fazer o devido ajuste de sinal quando necessário. Logo, a estratégia seria implementar um multiplicador utilizando-se diversas células do BiFi-FPGA.

Os dois tipos de multiplicadores implementados foram:

1. Multiplicador paralelo: Quatro células implementam um multiplicador 4x4 puramente combinacional;

2. Multiplicador serial (soma-desloca): Uma célula implementa um multiplicador 4x4 que executa em 5 ciclos.

No multiplicador paralelo (Fig. 6.10, 6.11), o multiplicando deverá ser colocado na entrada *d_inB* da célula. Além disso, o bit mais significativo do multiplicando é conectado ao sinal *ExtSign0* das células Cel0, Cel1 e Cel2, e ao sinal *ExtSign1* da última célula (Cel3).

Já o operando multiplicador é conectado a alguns sinais de controle das células. Cada bit do operando multiplicador é conectado a linha de controle *selB* de uma das células. O bit mais significativo do multiplicador é conectado às linhas de controle *invB* e *ExtSign2* da última célula (Cel3).

Essa implementação é totalmente combinacional, não contendo nenhum registrador ao longo do caminho de dados, logo, não se necessita de relógio para seu funcionamento.

Para a validação dos multiplicadores 4x4 e 8x8 utilizou-se uma simulação exaustiva de todas as combinações de valores dos operandos multiplicando e multiplicador. Na Fig. 6.12 observa-se o operando multiplicador variando de 0 a

15. Para cada valor do operando multiplicador, o multiplicando variava de 0 a 15 também, cobrindo todas as combinações possíveis. Os resultados da multiplicação foram gerados em forma de arquivo texto e para verficar a correção dos valores utilizou-se o aplicativo Excel da Microsoft [MIC 2002].

Na Fig. 6.13 mostram-se alguns casos da multiplicação 4x4, destacando-se $-3 \times 7 = -21$ (lembrando que -3=13 e -21=235 em complemento de dois). Na Fig. 6.14 mostra-se alguns casos da multiplicação 8x8, destacando-se $127 \times -3 = -381$ (onde -3=253 e -381=65155 em complemento de dois).

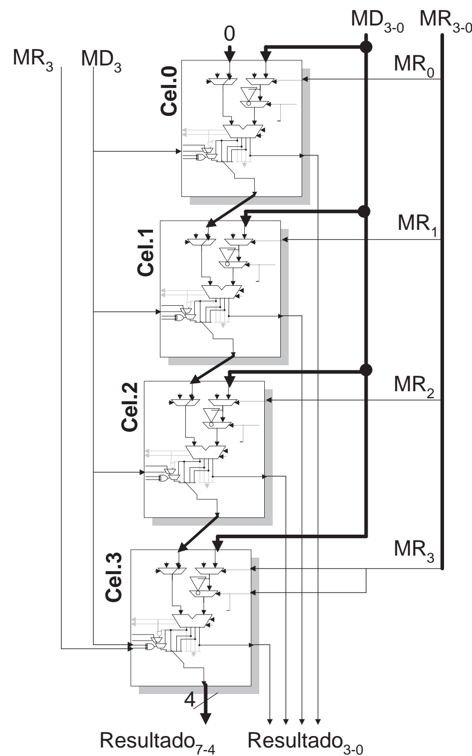

FIGURA 6.10 - Multiplicador paralelo 4x4 (array multiplier).

Para a implementação de um multiplicador serial 8x8, utilizam-se duas células mais uma máquina de estados para gerar os sinais de controle apropriados. A Fig. 6.15 mostra as principais conexões entre as duas células (linhas tracejadas). O resultado da multiplicação é obtido após *n+1* pulsos de relógio, onde n é o número de bits do operando multiplicador. No primeiro pulso, o operando multiplicador é colocado na entrada *d_inB* das células e o *ShiftRegLFSR* é carregado com esse operando. Nos pulsos seguintes, o multiplicando é colocado na entrada *d_inB* e, após n pulsos de relógio, obtém-se o resultado. A parte alta do resultado estará armazenada nos registradores (*RegOut*) e a parte baixa nos registradores *ShiftRegLFSR*. A Fig. 6.16 mostra a simulação da seguinte operação: $-127 \times 57 = -7239$ (onde -127=129 e -7239=58297 em complemento de dois).

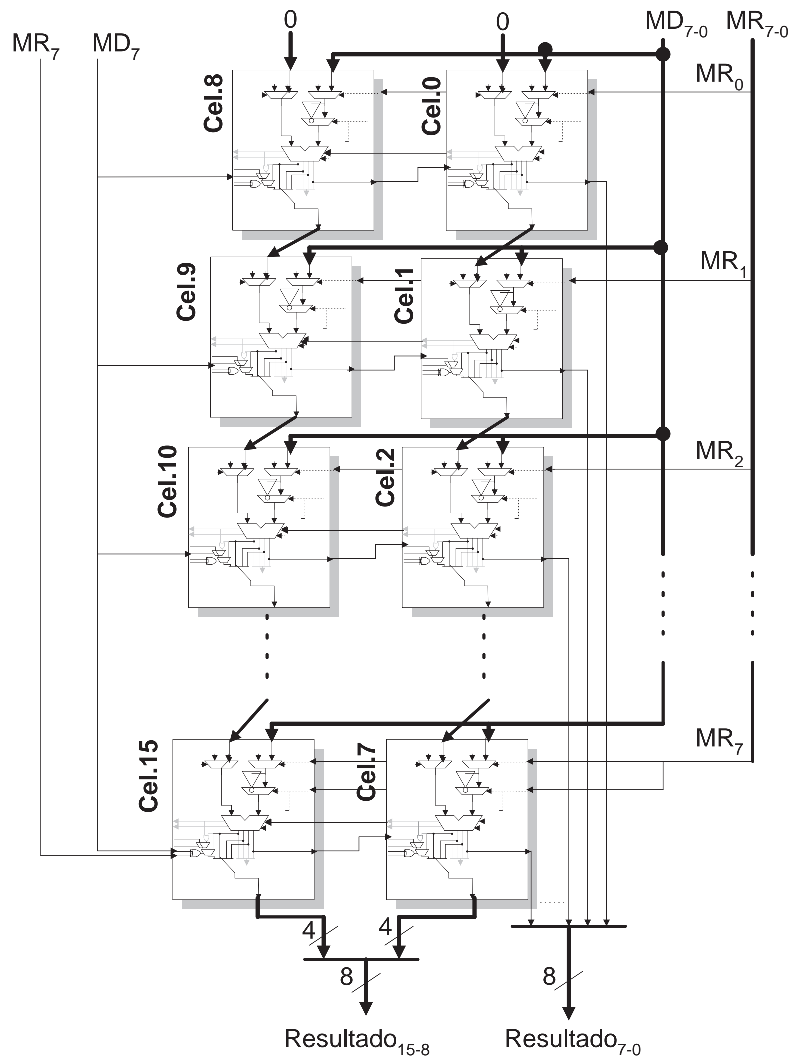

FIGURA 6.11 - Multiplicador paralelo 8x8 (array multiplier).

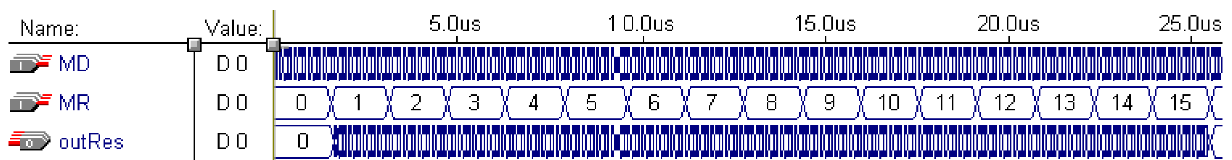

FIGURA 6.12 - Simulação exaustiva do multiplicador paralelo 4x4.

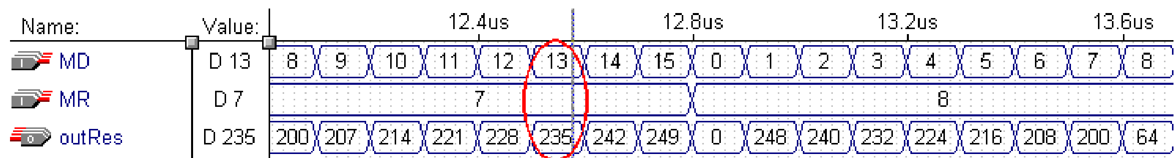

FIGURA 6.13 - Detalhe da simulação do multiplicador paralelo 4x4.

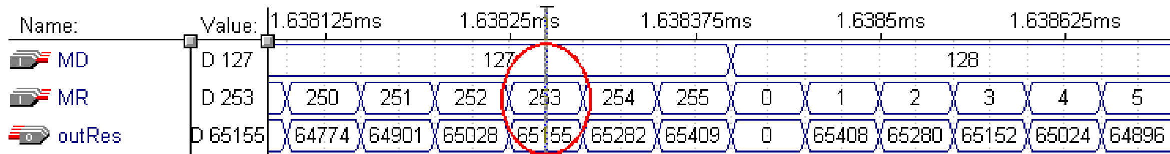

FIGURA 6.14 - Detalhe da Simulação do multiplicador paralelo 8x8.

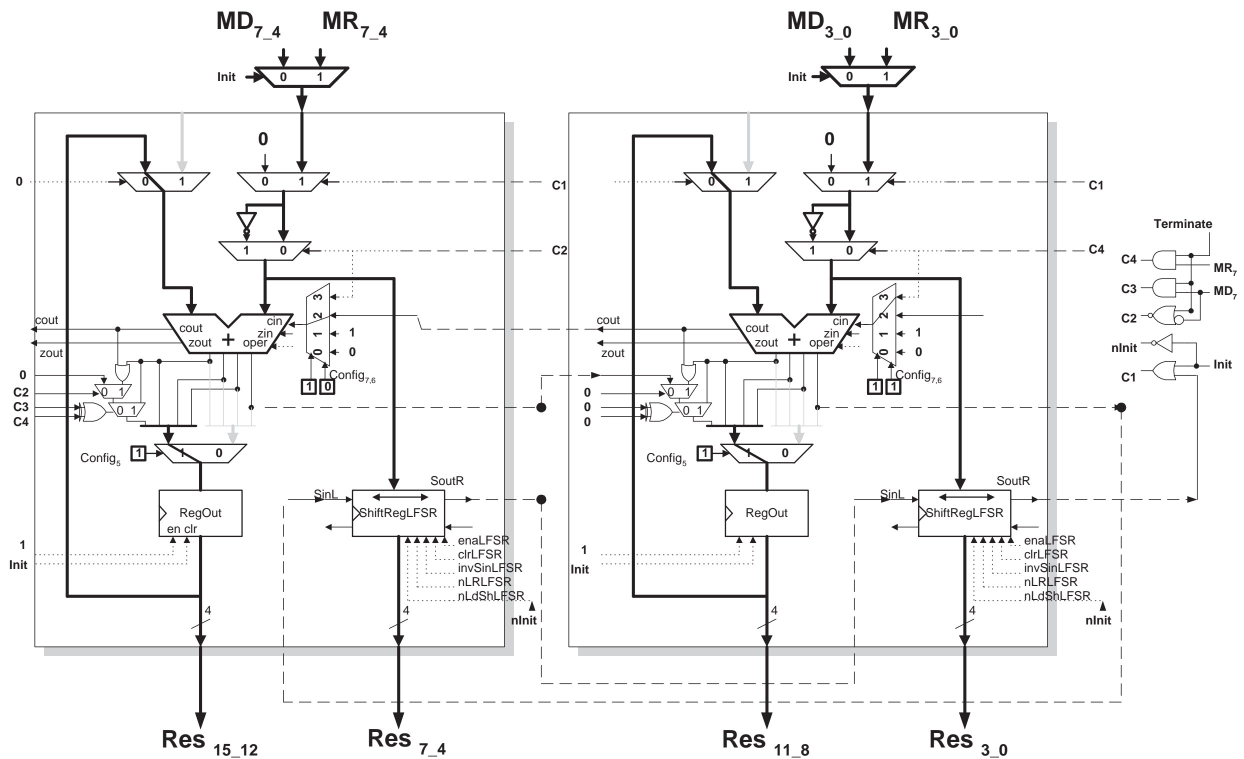

FIGURA 6.15 - Multiplicador serial 8x8.

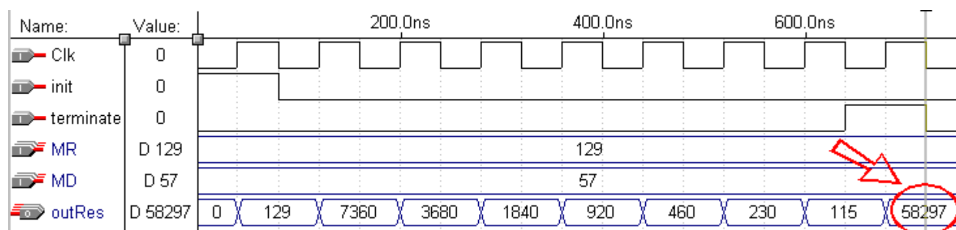

FIGURA 6.16 - Detalhe da simulação do multiplicador serial 8x8.

6.5 Filtros FIR

As figuras 6.17 e 6.18 apresentam as implementações de filtros FIR canônicos nas formas direta I e II, respectivamente. Os blocos multiplicadores podem ser implementados com qualquer uma das estratégias apresentadas na seção 6.4.

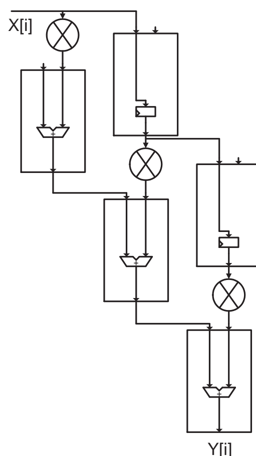

FIGURA 6.17 - Implementação canônica de um filtro FIR na forma direta I.

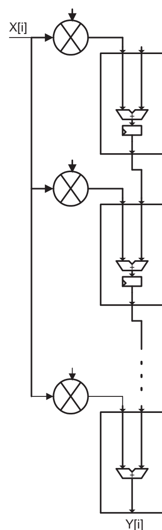

FIGURA 6.18 - Implementação canônica de um filtro FIR na forma direta II.

A Fig. 6.19 mostra a simulação de um filtro FIR passa alta canônico de 8 bits e 4 taps implementado na forma direta II utilizando multiplicador paralelo 8x8. Os coeficientes *h(k)* e os dados *x(i)* utilizados foram

h(k) = [-1 1 2 -1]

x(i) = [64 -64 0 64 -64 0 64 -64 0 64 -64 0]

observando-se que a representação em 8 bits complemento de dois de -1 é 255 e de -64 é 192.

Utilizando-se a equação da convolução (eq. 2.3)

$$y(i) = \sum_{k=0}^{N-1} h(k)\, x(i-k)$$

onde $N = 4$ e $i = 0, 1, 2, \ldots 11$

obtém-se os seguintes valores para *y(i)*

y(i) = [-64 128 64 -256 192 64 -256 192 64 -256 192 64]

os quais são confirmados pela Fig. 6.19, lembrando que a representação em 16 bits complemento de dois de -64 é 65472 e de -256 é 65280.

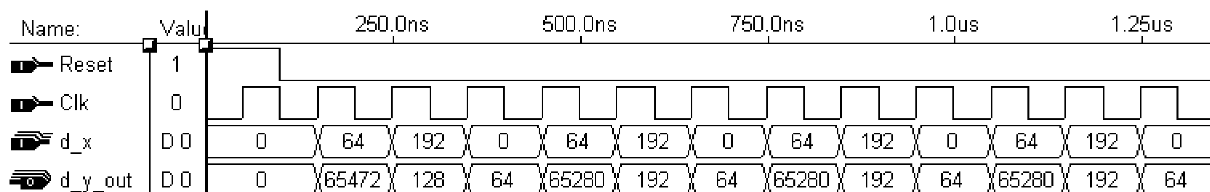

FIGURA 6.19 - Simulação de um filtro FIR passa alta.

6.6 Considerações Sobre a Validação da Arquitetura

As simulações dos circuitos DSP e BIST apresentadas neste capítulo possibilitaram validar a parte operativa da aquitetura proposta. No próximo capítulo são feitas estimativas da área gasta e degradação de desempenho para a implementação desses circuitos, utilizando-se o BiFi-FPGA. Uma análise comparativa com outras duas abordagens de projeto (dedicada e FPGA comercial) é também realizada.

7 Estimativas de Área e Freqüência

Após validada a arquitetura do BiFi-FPGA, iniciou-se um estudo para se fazer uma estimativa da área gasta para implementar os circuitos alvo bem como uma estimativa do atraso máximo provocado por um bloco lógico.

Com relação à estimativa de área, três abordagens foram analisadas: implementação dedicada, implementação no BiFi-FPGA e implementação num FPGA comercial de uso geral. A seguir, apresenta-se a metodologia adotada para se fazer a estimativa de área bem como os resultados obtidos e uma análise comparativa entre as três abordagens.

Foi decidido que o parâmetro utilizado para medir área seria o número de transistores. Todas as estimativas e comparações realizadas utilizariam esse parâmetro. Dessa forma, como primeiro passo, adotou-se a estratégia de se fazer a estimativa do número de transistores gastos na implementação de alguns blocos básicos de 1 bit. Os resultados são mostrados na Tab. 7.1.

O número total de transistores gastos, para um bloco de 1 bit, aparece na coluna 5(*#Transistores*). Essa coluna é a soma das colunas 3(*Base*) e 4(*Extra*). A coluna *Base* indica quantos transitores são necessários para a implementação de um bit do bloco. Logo, para um bloco de N bits são necessários $N \times Base$ transistores. Alguns blocos, no entanto, necessitam de alguns transistores extra, como no caso do multiplexador 2x1, que possui um inversor conectado à entrada de seleção para gerar os sinais *sel* e NOT(*sel*), o qual permite selecionar qual das duas entradas do multiplexador será colocada na saída. A coluna 4 indica esses transistores extras necessários. Na implementação de multiplexadores 2x1 mais largos, esse inversor pode ser compartilhado entre todas as fatias (*bit-slice*) do multiplexador. Logo, a coluna *Extra* da Tab. 7.1 indica quantos transitores extra são necessários para implementar o bloco, não importa o seu tamanho. Portanto, para se calcular o número de transistores gastos na implementação de um determinado bloco utiliza-se a fórmula

$$\#Transistores = N \times Base + Extra \tag{7.1}$$

onde N é a largura do bloco (em bits).

Um multiplexador 2x1 de 8 bits, por exemplo, necessita de

$$\begin{aligned} \#Transistores &= N \times Base + Extra \\ &= 8 \times 4 + 2 \\ &= 34 \text{ transistores} \end{aligned}$$

Após isso, todos os circuitos dos exemplos utilizados serão compostos desses blocos básicos. Dessa forma, torna-se fácil obter o número de transistores gastos por qualquer circuito, bastando fazer uma conversão utilizando os dados da Tab. 7.1.

TABELA 7.1 - Estimativa de área para blocos básicos.

		#Transistores		
Bloco	**Descrição**	**Base**	**Extra**	**Total**
INV	Inversor	2	0	2
NAND2	NAND de 2 entradas	4	0	4
NOR2	NOR de 2 entradas	4	0	4
AND2	AND de 2 entradas	6	0	6
XOR2	XOR de 2 entradas	6	4	10
NOR4	NOR de 4 entradas	8	0	8
MUX2x1	Multiplexador 2x1	4	2	6
MUX4x1	Multiplexador 4x1	12	4	16
MUX16x1	Multiplexador 16x1	60	8	68
Adder	Somador	28	0	28
Latch	Latch (com enable)	8	2	10
FF_D	Flip-flop tipo D com clr	18	2	20
Reg	FF_D com clear e enable (FF_D + Mux2x1)	22	4	26
Count	Contador (Adder+Reg)	50	4	54
SRegLR	Shift Register Left and Right (Reg+2.Mux2x1)	30	8	38

7.1 Implementação Dedicada

Na primeira análise, fez-se a estimativa do número de transistores gastos para a implementação dos circuitos alvo numa implementação dedicada (*full-custom*). Os resultados, em forma de equações, são apresentados na Tab. 7.2. Nessa tabela, a e d referem-se ao número de linhas de endereço (*address*) e dados (*data*) respectivamente. O motivo para a definição dessas equações é que, com elas, pode-se calcular a área ocupada por circuitos de qualquer tamanho, permitindo-se realizar diversas comparações de custo.

A dedução dessas equações foi feita com base nos circuitos mostrados nos Cap. 2 e 4, e nas estimativas da Tab. 7.1. Como exemplo do método utilizado, a primeira linha da tabela mostra o custo para a implementação do BIST de memórias ROM, utilizando o circuito mostrado na Fig. 4.4 (*checksum*).

Para a implementação desse circuito, necessita-se de um contador, um somador, um registrador, uma porta NOR e alguns inversores. Para simplificação de cálculo, desconsideraram-se os inversores, visto que seu número é variável (depende do valor do *checksum*) e, sua influência na área é mínima. Também para simplificação, considerou-se a utilização de portas NOR de 4 entradas. Abaixo, segue a dedução das equações (Tab. 7.2):

$$\begin{aligned} BIST_ROM_axd &= a(Count) + d\left(Adder + Reg + \frac{NOR4}{4}\right) \\ &= 50.a + d(28 + 22 + 2) \\ &= 50.a + 52.d \end{aligned} \tag{7.2}$$

$$BIST_RAM_axd = a(Count) + d\left(Reg + ShiftRegLR + XOR2 + \frac{2.NOR4}{4}\right) \tag{7.3}$$
$$= 50.a + d(22 + 30 + 6 + 4)$$
$$= 50.a + 62.d$$

$$\begin{aligned} LFSR_N &= N.(Reg + Mux2x1 + XOR2) \\ &= N.(22 + 4 + 6) \\ &= 32.N \end{aligned} \tag{7.4}$$

$$\begin{aligned} Mult_Par_N &= N^2.(Adder + AND2) \\ &= N^2.(28 + 6) \\ &= 34.N^2 \end{aligned} \tag{7.5}$$

$$\begin{aligned} Mult_Ser_N &= N.(Adder + AND2 + 2.Reg) \\ &= N.(28 + 6 + 44) \\ &= 78.N \end{aligned} \tag{7.6}$$

$$\begin{aligned} FIR_Can_Par_TxN &= T.(Mult_Par_N + Reg_N + Adder_N) \\ &= T.(34.N^2 + 22.N + 28.N) \\ &= T.(34.N^2 + 50.N) \end{aligned} \tag{7.7}$$

$$\begin{aligned} FIR_Can_Ser_TxN &= T.(Mult_Ser_N + Reg_N + Adder_N) \\ &= T.(78.N + 22.N + 28.N) \\ &= T.(128.N) \end{aligned} \tag{7.8}$$

$$FIR_nCan_Par_TxN = Mult_Par_N + Reg_N + Adder_N + 2.log_2T.Count \tag{7.9}$$
$$= 34.N^2 + 22.N + 28.N + (2.log_2T).50$$
$$= 34.N^2 + 50.N + 100.log_2T$$

$$FIR_nCan_Ser_TxN = Mult_Ser_N + Reg_N + Adder_N + 2.log_2T.Count \tag{7.10}$$
$$= 78.N + 22.N + 28.N + (2.log_2T).50$$
$$= 128.N + 100.log_2T$$

TABELA 7.2 - Estimativa de área para implementação dedicada.

Circuito	#Transistores
BIST_ROM_axd	$50a + 52d$
BIST_RAM_axd	$50a + 62d$
LFSR_N	$32.N$
Mult_Ser_N	$78.N$
Mult_Par_N	$34.N^2$
FIR_Can_Par_TxN	$T.(34.N^2 + 50.N)$
FIR_Can_Ser_TxN	$T.(128.N)$
FIR_nCan_Par_TxN	$34.N^2 + 50.N + 100.log_2T$
FIR_nCan_Ser_TxN	$128.N + 100.log_2T$

7.2 Implementação no BiFi-FPGA

Na segunda abordagem, foram analizados os mesmos circuitos sendo implementados no BiFi-FPGA. Para se fazer a estimativa de área, primeiramente calculou-se o número de transistores gastos para implementar 1 bloco lógico do BiFi-FPGA. Para isso, foram utilizados os mesmos blocos básicos apresentados na Tab. 7.1. O resultado da estimativa (672 transistores) é mostrado na Tab. 7.3.

TABELA 7.3 - Área da célula do BiFi-FPGA.

Bloco	Quant.	Cálculo	#Transistores
Mux2x1 (1 bit)	3	3x(4+2)	18
Mux2x1 (4 bits)	4	4x(4x4 + 2)	72
Mux4x1 (1 bit)	1	12 + 4	16
ULA	1	160	160
Reg (4 bits)	1	4x22 + 4	92
ShiftRegLFSR	1	184	184
Latch	8	8x(8+2)	80
Reg (1 bit)	1	22 + 4	26
Inversor	4	4 x 2	8
OR2	1	6	6
XOR2	1	6 + 4	10
Total			**672**

Novamente, foram derivadas equações para o número de células e, conseqüentemente, o número de transistores necessários para a implementação dos mesmos circuitos alvo, Tab. 7.4. Nessa tabela, considera-se que a, d, T e N são múltiplos de 4.

TABELA 7.4 - Estimativa de área para implementação no BiFi-FPGA.

Circuito	#Células	#Transistores
BIST_ROM_axd	$(a + 2d)/4$	$168a + 336d$
BIST_RAM_axd	$(a + 2d)/4$	$168a + 336d$
LFSR_N	$N/4$	$168N$
Mult_Ser_N	$N/4$	$168N$
Mult_Par_N	$(N^2)/4$	$168N^2$
Fir_Can_Par_TxN	$T.(N^2 + 2N)/4$	$T.(168N^2 + 336N)$
Fir_Can_Ser_TxN	$T.(3N)/4$	$T.504N$
Fir_nCan_Par_TxN	$(N^2 + 2N)/4 + 2.log_2T$	$168N^2 + 336N + 1344.log_2T$
Fir_nCan_Ser_TxN	$3N/4 + 2.log_2T$	$504N + 1344.log_2T$

7.3 Implementação em FPGA Comercial

Na terceira análise, utilizou-se um FPGA comercial, o Flex10k30 da Altera [ALT 2001]. Neste caso, os mesmos circuitos alvo foram descritos em VHDL, simulados e sintetizados em dispositivos Flex10k30 utilizando-se a ferramenta MaxPluxII [ALT 92]. Essa ferramenta fornece o número de células lógicas (*Logic Elements*) gastas para a implementação do circuito.

O que interessava neste trabalho era o número de transistores gastos e não o número de células. Entretanto, não encontrou-se na bibliografia nenhuma referência a respeito do número de transistores gastos ou área ocupada por uma célula no Flex10k30. Logo, adotou-se a mesma metodologia dos casos anteriores (implementação dedicada e BiFi-FPGA): a partir do manual de referência [ALT 2001], analizou-se os componentes contidos na célula e, a partir daí, utilizando os dados da Tab. 7.1, fez-se uma estimativa do número de transistores necessários para implementar uma célula desse dispositivo (Tab. 7.5).

Observando-se a arquitetura do bloco lógico do Flex10k (Fig. 3.7), pode-se verificar a presença de uma LUT de 4 entradas, um flip-flop (Reg), uma porta OR de duas entradas, 4 Mux2x1 e 5 latches de programação, cujas áreas podem ser estimadas diretamente a partir dos dados da Tab. 7.1. Além disso, verifica-se a existência dos blocos *Carry Chain*, *Cascade Chain* e *Clear/Preset Logic*.

A Fig. 7.1 mostra duas configurações possíveis para a rede *Cascade Chain* (AND ou OR). Como a rede pode funcionar em dois modos, supõe-se que a mesma é composta no mínimo por uma porta AND2 ou por uma porta OR2. Logo, seriam necessários no mínimo 6 transistores para sua implementação.

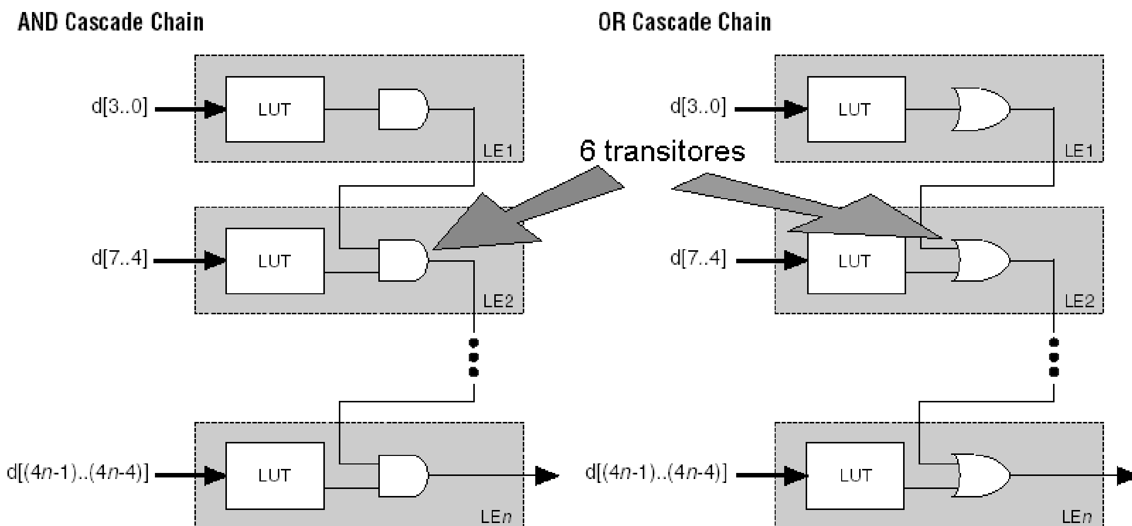

FIGURA 7.1 - Bloco Cascade Chain do Flex10k. Fonte:[ALT 2001]

O bloco *Clear/Preset Logic* pode assumir diversas configurações, sendo que a maior delas tem o tamanho de 26 transistores e é mostrada na Fig. 7.2. Em relação

ao bloco *Carry Chain*, não se conseguiu maiores informações a respeito, logo, o mesmo não foi incluído no cálculo da área da célula.

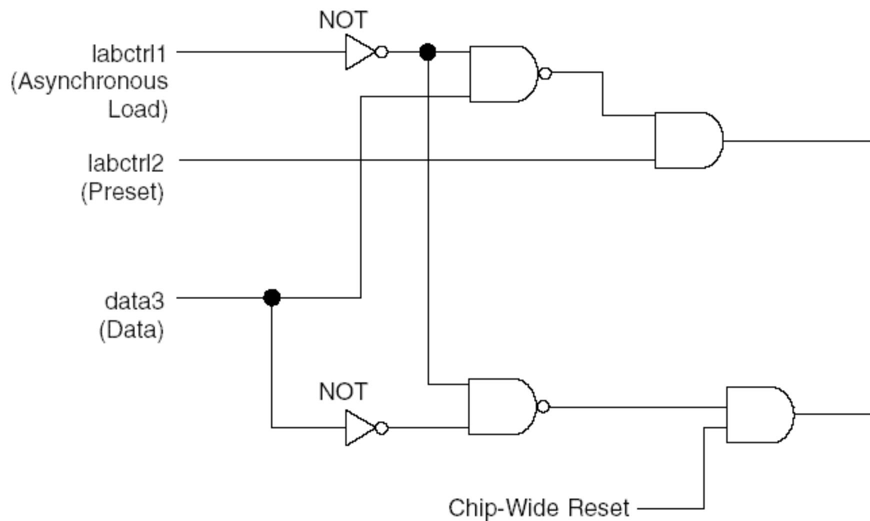

FIGURA 7.2 - Bloco Clear Preset Logic do Flex10k. Fonte:[ALT 2001]

A Tab. 7.5 mostra os resultados da estimativa para cada bloco e o total de 366 transistores para a célula lógica do Flex10k30. Para se obter o número de transistores gastos nos circuitos alvo implementados num Flex10k30, basta multiplicar o número de células lógicas por 366 (Tab. 7.6).

TABELA 7.5 - Estimativa de área para a célula do Flex10k30.

Bloco	**Quant.**	**#Transistores**
Latches (LUT)	16	160
Mux16x1 (LUT)	1	68
Reg	1	26
OR2	1	6
MUX2x1	4	24
Latches (programação)	5	50
Clear/Preset Logic	1	26
Cascade Chain	1	6
Total		**366**

7.4 Análise Comparativa de Custos

Na Tab. 7.7 é feita uma comparação entre as três abordagens analisadas. Nas comparações realizadas, não foi levada em conta a área gasta em conexões, visto que a arquitetura de conexão não foi implementada neste trabalho, pois não era esse o foco.

TABELA 7.6 - Área gasta para implementação no Flex10k30.

Circuito	#Células	#Transistores
Mult_Ser_8x8	38	13908
Mult_Par_8x8	64	23424
BIST_ROM_8x8	20	7320
BIST_RAM_8x8	31	11346
BIST_RAM_12x8	35	12810
BIST_RAM_8x16	53	19398
BIST_RAM_12x16	57	20862
LFSR_8	25	9150
LFSR_32	97	35502
FIR_Can_Par_16x8	468*	171288
FIR_Can_Ser_16x8	272*	99552

*Filtro simétrico. Fonte:[ALT 96].

É possível verificar que, em média, as implementações no BiFi-FPGA ocupam 50% menos área que as implementações no Flex10k30. Além disso, pôde-se concluir que a área gasta com conexões deverá ser menor no BiFi-FPGA, pois o mesmo possui maior granularidade (4 bits contra 1 bit do Flex10k30).

Com relação às implementações dedicadas, o BiFi-FPGA ocupa em média 6 vezes mais área (novamente não foram consideradas as conexões). No entanto, quando se considerar um sistema como um todo, a utilização do BiFi-FPGA pode se tornar mais adequada, visto que a reconfiguração permite a utilização da mesma área para a implementação de diversos circuitos, reduzindo a área do sistema final.

Como exemplo, considere-se que num determinado sistema serão implementados controladores BIST para uma memória RAM 4kx16, 2 RAMs 256x8, 1 RAM 4kx8, 2 ROMs 256x8 e será implementado também um LFSR de 32 bits com respectivo analisador de assinatura. O custo em número de transistores, numa implementação dedicada seria

$$1592 + 2 \times 896 + 1096 + 816 + 2 \times 1024 = 7344 \text{ transistores}$$

A mesma implementação no BiFi-FPGA custaria 7392 transistores, que é o custo do maior dos blocos implementados (BIST da RAM 4Kx16). Salienta-se novamente que isso só é possível se os testes puderem ser serializados no tempo.

Observa-se então, que para sistemas maiores pode se tornar vantajosa a utilização do BiFi-FPGA. Além disso, a implementação do BIST em área reconfigurável permite a fácil atualização ou modificação dos algoritmos utilizados sem necessidade de alteração do hardware, o que não seria possível na implementação dedicada.

Deve-se considerar também o fator quadrático do multiplicador paralelo, e quando se for utilizar o BiFi-FPGA para implementar um filtro mais os controladores BIST, a área dominante será aquela necessária para implementar o filtro (se este for implementado com multiplicador paralelo).

Dessa forma, a idéia de substituir uma implementação dedicada de filtro e

BIST por uma implementação reconfigurável (no BiFi-FPGA) com o intuito de se ter ganho em área, só será válida para o caso de filtros canônicos pequenos (menos de 16 Taps) ou então filtros não-canônicos.

É claro que a implementação em área reconfigurável tem as vantagens já mencionadas (prototipação, facilidade para atualização, etc.), além disso, já mostrou-se a validade da idéia quando comparada com um FPGA comercial de uso geral.

TABELA 7.7 - Comparação entre as três abordagens.

Bloco	Dedicado	FLEX10k	BiFi
Mult_Ser_8	624	13908	1344
Mult_Par_8	2176	23424	10752
BIST_ROM_8x8	816	7320	4032
BIST_RAM_8x8	896	11346	4032
BIST_RAM_12x8	1096	12810	4704
BIST_RAM_8x16	1392	19398	6720
BIST_RAM_12x16	1592	20862	7392
LFSR_8	256	9150	1344
LFSR_32	1024	35502	5376
FIR_Can_Par_16x8	41216 ** 20608	* 171288	215040 ** 107520
FIR_Can_Ser_16x8	16384 ** 8192	* 99552	64512 ** 32256
FIR_nCan_Par_16x8	2876	—-	18816
FIR_nCan_Ser_16x8	1324	—-	9408

* Filtro simétrico (Fonte: [ALT 96]).
** Estimativa de um filtro simétrico (50% de um filtro não simétrico).

7.5 Estimativas de Freqüência

Com o objetivo de se ter uma primeira idéia do desempenho do BiFi-FPGA, fez-se uma estimativa do atraso máximo provocado pelos circuitos do mesmo. Primeiramente, foram descritos em nível de transistores (*spice*) os diversos blocos componentes do bloco lógico do BiFi-FPGA (inversores, portas OR, NOR, AND, XOR, multiplexadores e ULA). Utilizou-se tecnologia $AMS\,0.35\mu m$, com largura de canal para os transistores $NMOS = 1.0\mu m$ e $PMOS = 2.0\mu m$.

Esses blocos foram simulados separadamente para se verificar qual a situação de maior atraso de cada um. Após isso, verificou-se qual seria o caminho de maior atraso do bloco lógico, o qual é mostrado na Fig. 7.3. Conectou-se então, para simulação elétrica, um multiplexador 2x1 de 4 bits, um inversor, um multiplexador 2x1 de 4 bits, a ULA e mais um multiplexador 2x1 de 4 bits.

Além disso, visto que o maior atraso provocado no multiplexador ocorre quando a entrada de seleção varia, foram consideradas duas situações: *selB* variando de 0 para 1 e *selB* variando de 1 para 0 (01 e 10 indicados nas Tab. 7.8 e 7.9). Esses estímulos foram aplicados à linha *selB*, observando-se e medindo-se o atraso para o sinal se propagar até *cout* (Tab. 7.8) e *d_OutComb* (Tab. 7.9).

Para simular a carga existente na saída da célula, provocada pelas chaves de roteamento que deverão constituir a arquitetura de roteamento, foram conectados inversores às saídas *cout* e *d_OutComb*. As Tab. 7.8 e 7.9 mostram também os resultados dessas simulações para diversas cargas. Observando-se o pior caso (destacado na Tab. 7.8), verifica-se que o atraso máximo provocado pela célula seria de $\frac{2.366+1.567}{2} = 1.966ns$, o que daria uma freqüência máxima de $\frac{1}{1.966ns} = 508MHz$. Todas as simulações foram feitas utilizando-se o simulador **OrCAD PSpice** versão 9.0 [ORC 98].

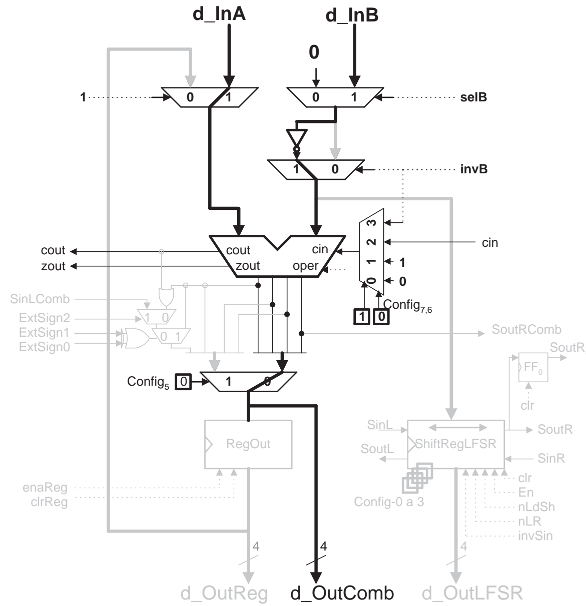

FIGURA 7.3 - Estimativa do caminho de maior atraso.

TABELA 7.8 - Atraso de cout em relação a selB (em ps).

		Número de inversores na saída					
SelB	**cout**	**0**	**1**	**2**	**5**	**10**	**20**
01	10	931	963	985	1040	1125	1283
10	01	888	919	945	1035	1191	1487

TABELA 7.9 - Atraso de d_OutComb em relação a selB (em ps).

		Número de inversores na saída					
SelB	**d_OutComb**	**0**	**1**	**2**	**5**	**10**	**20**
01	01	948	1007	1080	1292	1653	**2366**
10	10	983	1019	1048	1135	1283	**1567**

8 Conclusões, Contribuições e Trabalhos Futuros

Este trabalho apresentou a proposta de uma nova arquitetura de FPGA (*BIST and Filter Field Programmable Gate Array* - BiFi-FPGA) destinada à implementação de circuitos de processamento digital de sinais (*Digital Signal Processing* - DSP) e auto-teste embutido (*Built-in Self Test* - BIST).

A proposta geral do BiFi-FPGA é baseada na idéia do DP-FPGA (*Data Path FPGA*) [CHE 94], onde um FPGA deveria ser dividido em 3 partes (blocos) principais, sendo cada uma otimizada para a implementação de uma classe de circuitos (memória, parte de controle e parte operativa). Entretanto, devido à complexidade desse projeto, este trabalho preocupou-se em focar somente o bloco parte operativa, deixando a implementação dos blocos de controle e de memória para trabalhos futuros. Além disso, como já foi mencionado, a parte de controle é bem implementada pelos FPGAs atuais de propósito geral.

É importante salientar que este trabalho possibilitou a investigação de várias áreas: processamento digital de sinais, teste de hardware e projeto de arquiteturas reconfiguráveis.

Na área de processamento digital de sinais, os algoritmos mais utilizados são os filtros FIR, os filtros IIR e a transformada discreta de Fourier (DFT). Nesse sentido, foi necessário se fazer uma revisão bibliográfica e um estudo de arquiteturas para implementação desses algoritmos (Cap. 2). Esse estudo possibilitou a otimização do bloco lógico do BiFi-FPGA visando sua adequação à implementação desses algoritmos.

Com relação ao teste de hardware e projeto visando a testabilidade, confirmou-se a importância cada vez maior dessa área no projeto dos sistemas atuais. Não se pode mais projetar grandes sistemas sem a preocupação, desde a fase de projeto, com o teste.

Novamente, foram estudadas várias técnicas de teste, principalmente aquelas relativas ao teste fora de funcionamento (*off-line*): técnicas de BIST, teste de memórias RAM e ROM. Arquiteturas para implementação dessas técnicas foram também estudadas.

A partir da análise dessas duas classes de aplicações e suas implementações em hardware, verificou-se as semelhanças entre as estruturas básicas para implementação das partes operativas das duas. Dessa forma, a escolha de uma arquitetura reconfigurável para implementação de circuitos dessas duas classes de aplicações mostrou-se ser muito adequada pois, utilizando-se a estratégia de reconfiguração poder-se-ia reduzir a área gasta para implementação de sistemas desses tipos.

A partir daí foram estudadas algumas arquiteturas reconfiguráveis existentes e aspectos relativos ao projeto destas arquiteturas, focando-se principalmente nos

FPGAs. Verificou-se a vastidão do tema, a diversidade de arquiteturas e a expansão cada vez maior de pesquisas nessa área.

As principais decisões a serem tomadas com relação a esse tópico seriam a tecnologia de programação, granularidade e arquitetura do bloco lógico. A tecnologia de programação escolhida foi a SRAM principalmente por sua versatilidade e facilidade de programação. A granularidade do bloco lógico escolhida foi de 4 bits, visto que as aplicações alvo trabalham sobre múltiplos desse valor (8,16, 32 ou mais bits).

A partir disso definiu-se o bloco lógico do BiFi-FPGA e foi feita sua descrição em VHDL. Após validado funcionalmente o bloco, iniciou-se a fase de mapeamento de circuitos DSP e BIST utilizando-se instâncias do bloco lógico interconectadas. Após certificar-se, através de simulações funcionais, que o BiFi-FPGA podia implementar as aplicações visadas (DSP e BIST), iniciou-se a fase de estimativas de área.

Foram feitas estimativas da área em silício ocupada pelo BiFi-FPGA na implementação dos circuitos alvo. A estratégia adotada é independente de tecnologia de processo, pois considerou-se o número de transistores gastos e não a área de silício em si. Com isso, foi possível realizar-se comparações com outras duas abordagens (dedicada e FPGA comercial) de forma mais fácil e ainda assim realista.

Nas análises feitas pôde-se demonstrar que o BiFi-FPGA ocupa menor área, comparado com um FPGA comercial (Flex10k30), na implementação de partes operativas de diversos circuitos de DSP e controladores BIST.

Nas comparações realizadas não foi levada em conta a área gasta em conexões, visto que a arquitetura de conexão não foi definida neste trabalho. Os resultados mostraram que, em média, as implementações no BiFi-FPGA ocupam 50% menos área que as implementações no Flex10k30. Acredita-se, também, que a área gasta com conexões deverá ser menor no BiFi-FPGA, pois o mesmo possui maior granularidade (4 bits contra 1 bit do Flex10k30).

Com relação às implementações dedicadas, o BiFi-FPGA ocupa em média 5 a 6 vezes mais área (novamente não foram consideradas as conexões). No entanto, quando se considerar um sistema como um todo, a utilização do BiFi-FPGA pode se tornar menos custosa, visto que a reconfiguração permite a utilização da mesma área para a implementação de diversos circuitos, o que poderá reduzir a área do sistema final.

Fez-se também, uma estimativa do atraso máximo provocado pelo bloco lógico do BiFi-FPGA. Através de simulação pspice, estimou-se que a freqüência máxima de operação seria de 508MHz (para tecnologia $0.35\mu m$). Considerando-se que o BiFi-FPGA possuirá menos conexões que um Flex10k, conforme já foi explicado, acredita-se que o BiFi-FPGA poderá operar numa freqüência maior que o Flex10k (considerando que os dois FPGAs sejam implementados na mesma tecnologia).

É possível verificar, portanto, que o BiFi-FPGA fornece um espaço para soluções entre os FPGAs comerciais e os circuitos dedicados, quando se desejar implementar aplicações BIST e DSP. Ou seja, 50% menos área que os FPGAs comerciais (e provavelmente maior freqüência de operação) e maior flexibilidade que as implementações dedicadas.

Como contribuições adicionais, este trabalho poderá servir de referência para a pesquisa e o desenvolvimento de novas arquiteturas reconfiguráveis. Além disso, a estratégia de comparação de área que foi adotada (número de transistores), bem como as estimativas do número de transistores gastos na implementação de diversos blocos de hardware (uma célula lógica do FPGA Flex10k30, multiplicadores, filtros FIR, LFSRs, BIST de memórias RAM e ROM, etc.), poderão ser utilizadas em outros trabalhos que necessitem fazer comparações de área.

Como trabalhos futuros, podem-se destacar:

- Descrição da arquitetura de conexão;
- Implementação do bloco responsável por controlar a reconfiguração do BiFi-FPGA;
- Descrição das outras duas partes do FPGA destinadas à implementação de memória e partes de controle;
- Interligação de todos os blocos para obtenção do FPGA final;
- Implementação de auto-teste do próprio BiFi-FPGA;
- Desenvolvimento de ferramentas de CAD, tais como compiladores e simuladores, tendo o BiFi-FPGA como arquitetura alvo. Desse modo, teria-se um ambiente de programação de alto nível para síntese de aplicações no BiFi-FPGA.

Bibliografia

[ABR 90] ABRAMOVICI, M.; BREUER, M.; FRIEDMAN, A. **Digital Systems Testing and Testable Design**. New York: IEEE, 1990. 652p.

[ACT 96] ACTEL. **ACT 1 Series FPGAs**. [S.l.]: Actel Corporation, 1996. 24p. Disponível em: <http://www.actel.com/docs/datasheets/db97s01d07.pdf>. Acesso em: 2001.

[ADA 98] ADARIO, A. **Síntese de alto nível de arquiteturas reconfiguráveis**. Porto Alegre, RS: Universidade Federal do Rio Grande do Sul, 1998. (EQ-24).

[AGR 93] AGRAWAL, V. D.; KIME, C. R.; SALUJA, K. K. A Tutorial on Built-In Self-Test - Part 1 - Principles. **IEEE Design & Test of Computers**, Los Alamitos, v.10, n.1, p.73–82, Mar. 1993.

[AGR 93a] AGRAWAL, V. D.; KIME, C. R.; SALUJA, K. K. A Tutorial on Built-In Self-Test - Part 2 - Applications. **IEEE Design & Test of Computers**, Los Alamitos, v.10, n.2, p.69–77, June 1993.

[ALT 92] ALTERA. **MAX+plusII User Guide**. San Jose, 1992. 233p.

[ALT 96] ALTERA. **FIR Filters. Functional Specification 1**. San Jose, 1996. Disponível em: <http://www.altera.com/literature>. Acesso em: 2001.

[ALT 2000] ALTERA. **Altera Corporation**. Disponível em: <http://www.altera.com/>. Acesso em: 2000.

[ALT 2001] ALTERA. **FLEX 10K Embedded Programmable Logic Device Family Data Sheet**. San Jose, 2001. Disponível em: <http://www.altera.com/literature/ds/dsf10k.pdf>. Acesso em: 2001.

[ALT 2001a] ALTERA. **APEX II Programmable Logic Device Family Data Sheet**. San Jose, 2001. Disponível em: <http://www.altera.com/literature/ds/apex2.pdf>. Acesso em: 2002.

[ALT 2002] ALTERA. **APEX 20k Programmable Logic Device Family Data Sheet**. San Jose, 2002. Disponível em: <http://www.altera.com/literature/ds/apex.pdf>. Acesso em: 2002.

[ALT 2002a] ALTERA. **Mercury Programmable Logic Device Family Data Sheet**. San Jose, 2002. Disponível em: <http://www.altera.com/literature/ds/apex.pdf>. Acesso em: 2002.

[ALT 2002b] ALTERA. **Stratix Programmable Logic Device Family Data Sheet**. San Jose, 2002. Disponível em: <http://www.altera.com/literature/ds/ds_stx.pdf>. Acesso em: 2002.

[ALV 95] ALVES, V. Functional Test of Single and Multi-Port SRAMs. In: CONGRESS OF THE BRAZILIAN MICROELECTRONICS SOCIETY, 10., 1995, Canela, RS, Brasil. **Proceedings...** Porto Alegre: Instituto de Informática da UFRGS, 1995. p.169–189.

[BAR 91] BARBAGALLO, S. et al. An Experimental Comparison of Different Approaches to ROM BIST. In: ADVANCED COMPUTER

TECHNOLOGY, RELIABLE SYSTEMS AND APPLICATIONS. ANNUAL EUROPEAN COMPUTER CONFERENCE, 5., 1991. **Proceedings. . .** [S.l.: s.n.], 1991. p.567–571.

[BAY 99] BAYRAKTAROGLU, I.; ORAILOGLU, A. Low-Cost On-Line Test for Digital Filters. In: IEEE VLSI TEST SYMPOSIUM, 17., 1999. **Proceedings. . .** [S.l.]: IEEE, 1999. p.446–451.

[BEL 95] BELLAOUAR, A. **Low-power Digital VLSI Design - Circuits and Systems**. Boston: Kluwer Academic, 1995. 530p.

[BEL 98] BELLILE, O.; DUJARDIN, E. Architecture of a Programmable FIR Filter Co-Processor. In: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 1998. **Proceedings. . .** [S.l.]: IEEE, 1998. p.433–436.

[BET 97] BETZ, V.; ROSE, J. Cluster-Based Logic Blocks for FPGAs - Area-Efficiency vs. Input Sharing and Size. In: IEEE CUSTOM INTEGRATED CIRCUITS CONFERENCE, 1997, Los Alamitos, California. **Proceedings. . .** [S.l.]: IEEE Computer Society Press, 1997. p.551–554.

[BET 98] BETZ, V.; ROSE, J. How Much Logic Should Go in an FPGA Logic Block. **IEEE Design & Test of Computers**, Los Alamitos, v.15, n.1, p.10–15, Jan.-Mar. 1998.

[BRO 96] BROWN, S. FPGA Architectural Research - A Survey. **IEEE Design & Test of Computers**, Los Alamitos, v.13, n.4, p.9–15, Winter 1996.

[BRO 2000] BROWN, S. D.; VRANESIC, Z. G. **Fundamentals of digital logic with VHDL design**. Boston: McGraw-Hill, 2000. 840p.

[CAM 95] CAMURATI, P. et al. Industrial BIST of Embedded RAMs. **IEEE Design & Test of Computers**, Los Alamitos, v.12, n.3, p.86–95, Fall 1995.

[CAR 2000] CARRO, L.; AGOSTINI, L.; PACHECO, R.; LUBASZEWSKI, M. Using Reconfigurability Features to Break Down Test Costs - a Case Study. In: IEEE LATIN AMERICAN TEST WORKSHOP, LATW, 1., 2000, Rio de Janeiro. **Proceedings. . .** [S.l.]: IEEE, 2000. p.209–214.

[CHA 97] CHAKRABARTY, K.; HAYES, J.P. On The Quality of Accumulator-based Compaction of Test Responses. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l], v.16, n.8, p.916–922, Aug. 1997.

[CHE 92] CHEN, D. C. A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithmic-Specific High-Speed DSP Data Paths. **IEEE Journal of Solid-State Circuits**, [S.l.], v.27, n.12, p.1895–1904, Dec. 1992.

[CHE 94] CHEREPACHA, D. **A Field-Programmable Gate Array Architecture Optimized for Datapaths**. Toronto, Canada: Graduate Department of Electrical and Computer Engineering, 1994. Master Thesis of Applied Science.

[COT 99] COTA, E.; KRUG, M.; LUBASZEWSKI, M.; CARRO, L.; SUSIN, A. Implementing a Self-Testing 8051 Microprocessor. In: BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUIT DESIGN, 1999, Natal. **Proceedings. . .** [S.l.]: IEEE Computer Society, 1999. p.202–205.

[COT 2001] COTA, E.; CARRO, L.; LUBASZEWSKI, M. A Test Method for Digital Signal Processors. In: IEEE LATIN AMERICAN TEST WORKSHOP, 2., 2001. **Proceedings. . .** [S.l.]: IEEE, 2001. p.214–219.

[COU 92] COUNIL, C.; CAMBON, G. A Functional BIST Approach for FIR Digital Filters. In: IEEE VLSI TEST SYMPOSIUM, 10., 1992. **Proceedings. . .** [S.l.]: IEEE, 1992. p.90–95.

[DEH 96] DEHON, A. **Reconfigurable Architectures for General-Purpose Computing**. Massachussets: [s.n.], 1996. Tese de Doutorado.

[EST 63] ESTRIN, G.; BUSSELL, B.; TURN, R. Parallel Processing in a Restructurable Computer System. **IEEE Transactions on Electronic Computers**, [S.l.], v.EC-12, n.5, p.747–755, Dec. 1963.

[FRA 92] FRANKLE, J. Iterative and Adaptive Slack Allocation for Performance-Driven Layout and FPGA Routing. In: ACM /IEEE DESIGN AUTOMATION CONFERENCE, 1992, Anaheim, California. **Proceedings. . .** [S.l.]: IEEE Computer Society Press, 1992. p.536–542.

[FUJ 85] FUJIWARA, H. **Logic Testing and Design for Testability**. London: The MIT Press, 1985. 284p.

[GON 2001] GONSALES, A. D.; LUBASZEWSKI, M.; CARRO, L.; SUZIN, A. Projeto de um PLD com Características de Testabilidade. In: WORKSHOP IBERCHIP, 7., 2001, Montevideo, Uruguay. **Proceedings. . .** [S.l.: s.n.], 2001. p.71–96.

[GON 2001a] GONSALES, A. D.; LUBASZEWSKI, M.; CARRO, L. Design of a Field Programmable Gate Array with Testability Features. In: UFRGS MICROELETRONICS SEMINAR, 16., 2001, Santa Maria, RS. **Proceedings. . .** Porto Alegre: Instituto de Informática da UFRGS, 2001. p.105–108.

[GON 2001b] GONSALES, A. D.; LUBASZEWSKI, M.; CARRO, L. Design of a Field Programmable Gate Array for BIST and Datapath-Intensive Applications. In: USER FORUM ON MICROELECTRONICS, MEMS AND DIGITAL SYSTEMS, 1., 2001, Pirenopolis, GO, Brasil. **Proceedings. . .** Porto Alegre: SBC, 2001.

[GON 2002] GONSALES, A. D.; LUBASZEWSKI, M.; CARRO, L.; COTA, É. Uma Nova Arquitetura de FPGA para Aplicações DSP Integrando Capacidades de BIST. In: WORKSHOP IBERCHIP, 8., 2002, Guadalajara, Mexico. **Proceedings. . .** Guadalajara: Cinvestav, 2002.

[GON 2002a] GONSALES, A. D.; LUBASZEWSKI, M.; CARRO, L.; RENOVELL, M. A New FPGA for DSP Applications Integrating BIST Capabilities. In: IEEE LATIN AMERICAN TEST WORKSHOP, 3., 2002, Montevideo, Uruguay. **Proceedings. . .** [S.l.]: IEEE, 2002.

[GOO 93] GOOR, A. V. Using March Tests to Test SRAMs. **IEEE Design & Test of Computers**, Los Alamitos, v.10, n.1, p.8–14, Mar. 1993.

[IWA 96] IWASAKI, K.; NAKAMURA, S. Aliasing Error for a Mask ROM Built-In Self Test. **IEEE Transactions on Computers**, [S.l.], v.45, n.3, p.270–277, Mar. 1996.

[KAV 98] KAVIANI, A.; VRANESIC, D.; BROWN, S. Computational Field Programmable Architecture. In: CUSTOM INTEGRATED CIRCUITS CONFERENCE, 1998, Santa Clara, CA. **Proceedings...** [S.l.]: IEEE Computer Society Press, 1998.

[KIE 98] KIEFER, G.; WUNDERLICH, H-J. Deterministic BIST with Multiple Scan Chains. In: INTERNATIONAL TEST CONFERENCE, 1998. **Proceedings...** [S.l.: s.n.], 1998. p.910–917.

[LEE 96] LEE, D.; YANNAKAKIS, M. Principles and Methods of Testing Finite State Machines - A Survey. **Proceedings of the IEEE**, Los Alamitos, v.84, n.8, p.1090–1123, Aug. 1996.

[LIM 2000] LIMA, F.; GÜNTZEL, J. L. Componentes Programáveis. In: ESCOLA DE MICROELETRôNICA DA SBC-SUL, 2., 2000, Torres, RS. **Proceedings...** Porto Alegre: UFRGS, 2000. p.71–96.

[LUB 2000] LUBASZEWSKI, Marcelo. Teste e Projeto Visando o Teste de Circuitos e Sistemas Integrados. In: REIS, R (Ed.). **Concepção de Circuitos Integrados**. Porto Alegre, RS: II da UFRGS, 2000. p.163–185.

[MAR 99] MARINISSEN, E. J. et al. Towards a Standard for Embedded Core Test - An Example. In: INTERNATIONAL TEST CONFERENCE, 1999. **Proceedings...** [S.l.: s.n.], 1999. p.616–627.

[MIC 2002] MICROSOFT. **Microsoft Home Page**. Disponível em: <http://www.microsoft.com/>. Acesso em: 2001.

[NUS 98] NUSSAUM, P.; GIRAU, B.; TISSERAND, A. Field Programmable Processor Arrays. In: INTERNATIONAL CONFERENCE ON EVOLVABLE SYSTEMS, 2., 1998. **Proceedings...** Berlin: Springer-Verlag, 1998. p.311–322.

[OPP 99] OPPENHEIM, A. V.; SCHAFER, R. W. **Discret-Time Signal Processing**. New Jersey: Prentice-Hall, 1999. 870p.

[ORC 98] ORCAD. **OrCAD Home Page**. 1998. Disponível em: <http://www.orcad.com/>. Acesso em: 2002.

[PIR 98] PIRSCH, P. **Architectures for Digital Signal Processing**. Chichester, England: John Wiley, 1998. 417p.

[RAB 75] RABINER, L. R.; GOLD, B. **Theory and Application of Digital Signal Processing**. New Jersey: Prentice-Hall, 1975. 762p.

[RAJ 93] RAJSKI, J.; TYSZER, J. Accumulator-based Compaction of Test Responses. **IEEE Transactions on Computers**, [S.l.], v.42, n.6, p.643–650, June 1993.

[RAV 99] RAVI, S.; JHA, N. K.; LAKSHMINARAYANA, G. TAO-BIST - A Framework for Testability Analysis and Optimization of RTL Circuits for BIST. In: IEEE VLSI TEST SYMPOSIUM, 17., 1999. **Proceedings...** [S.l.: s.n.], 1999. p.398–406.

[REN 96] RENOVELL, M.; FIGUERAS, J.; ZORIAN, Y. Testing the Interconnect Structures of Unconfigurated FPGA. In: IEEE EUROPEAN TEST WORKSHOP, 1996, France. **Proceedings...** [S.l.]: IEEE, 1996. p.125–129.

[REN 97] RENOVELL, M. et al. Testing Unconfigurated FPGA Logic Modules. In: IEEE EUROPEAN TEST WORKSHOP, 2., 1997, Cagliari, Italy. **Proceedings...** [S.l.]: IEEE, 1997. p.125–129.

[REN 2001] RENOVELL, M.; FIGUERAS, J.; ZORIAN, Y. IS-FPGA - A New Symmetric FPGA Architecture with Implicit SCAN. In: INTERNATIONAL TEST CONFERENCE, 2001. **Proceedings...** [S.l.: s.n.], 2001.

[ROS 93] ROSE, J.; GAMAL, A. El; SANGIOVANNI-VINCENTELLI, A. Architecture of Field-Programmable Gate Arrays. **Proceedings of The IEEE**, Los Alamitos, v.81, n.7, p.1013–1029, July 1993.

[SHI 96] SHIRATSUCHI, S. FPGA as a Key Component for Reconfigurable System. In: EVOLVABLE SYSTEMS: FROM BIOLOGY TO HARDWARE, 1996, Tsukuba, Japan. **Proceedings...** [S.l.: s.n.], 1996. p.23–32.

[SMI 97] SMITH, M. J. S. **Application-specific integrated circuits**. Reading: Addison-Wesley, 1997. 1026p.

[SOU 2000] SOUZA JÚNIOR, A. A.; GONSALES, A. D.; FURLANETTO, E.; CARRO, L.; LUBASZEWSKI, M. Modeling of a Dinamically Reconfigurable Test Targeted EPLD - TTEPLD. In: UFRGS MICROELETRONICS SEMINAR, 15., 2000, Torres, RS. **Proceedings...** Porto Alegre: Instituto de Informática da UFRGS, 2000. p.13–16.

[STR 98] STROELE, A. P. Bit Serial Pattern Generation and Response Compaction Using Arithmetic Functions. In: IEEE VLSI TEST SYMPOSIUM, 16., 1998, Monterey, CA. **Proceedings...** [S.l.: s.n.], 1998. p.78–84.

[STR 96] STROUD, C. et al. Built-In Self-Test of Logic Blocks in FPGAs. In: IEEE VLSI TEST SYMPOSIUM, 14., 1996. **Proceedings...** [S.l.]: IEEE, 1996. p.387–392.

[STR 98a] STROUD, C. et al. Built-In Self-Test of FPGA Interconnect. In: IEEE VLSI TEST SYMPOSIUM, 16., 1998, Monterey, CA. **Proceedings...** [S.l.: s.n.], 1998. p.404–411.

[SUS 79] SUSIN, A. A. **Parties operatives a elements modulaires**. Grenoble: Ensimag, 1979. 54p.

[SUS 81] SUSIN, A. A. **Etude des parties operatives a elements modulaires pour processeurs monolithiques**. Grenoble: Institut National Polytechnique, 1981. 153p.

[TRE 93] TREUER, R.; AGARWAL, V. Built-In Self-Diagnosis for Repairable Embedded RAMs. **IEEE Design & Test of Computers**, Los Alamitos, v.10, n.2, p.24–33, June 1993.

[TRI 94] TRIMBERGER, S. M. **Field-programmable gate array technology**. Boston: Kluwer Academic Publishers, 1994. 258p.

[WAN 93] WANG, Q. **An Array Architecture for Reconfigurable Datapaths**. Toronto, Canada: Department of Electrical and Computer Engineering, University of Toronto, 1993. Master Thesis of Applied Science.

[WES 93] WESTE, N. H. E. **Principles of CMOS VLSI design - a systems perspective**. Reading: Addison-Wesley, 1993. 713p.

[XIL 98] XILINX. **XC5200 Series Field Programmable Gate Arrays. Product Specification**. San Jose, 1998. Disponível em: <http://www.xilinx.com/partinfo/5200.pdf>. Acesso em: 2001.

[XIL 99] XILINX. **XC4000E and XC4000X Series Field Programmable Gate Arrays. Product Specification**. San Jose, 1999. Disponível em: <http://www.xilinx.com/partinfo/4000.pdf>. Acesso em: 2001.

[XIL 2001] XILINX. **Virtex 2.5 V Field Programmable Gate Arrays**. San Jose, 2001. Disponível em: <http://www.xilinx.com/partinfo/ds003.pdf>. Acesso em: 2001.

[XIL 2001a] XILINX. **Spartan-IIE 1.8V FPGA Family - Functional Description**. San Jose, 2001. Disponível em: <http://www.xilinx.com/partinfo/ds077_2.pdf>. Acesso em: 2001.