Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação

Reconfigurabilidade dinâmica e remota de FPGAs

Alexandre Alves de Lima Ribeiro

São Carlos - SP

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 26.07.2002

Assinatura: _____

Reconfigurabilidade dinâmica e remota de FPGAs

Alexandre Alves de Lima Ribeiro

Orientador: *Prof. Dr. Eduardo Marques*

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP - São Carlos
Julho de 2002

“Nossas aspirações são nossas possibilidades.”

Robert Browning, século XIX

Agradecimentos

Ao amigo, incentivador e orientador, Prof. Dr. Eduardo Marques, que pacientemente soube transmitir seus conhecimentos e respeitar individualmente o ritmo de trabalho de cada um do grupo.

Aos amigos do LCR e do mestrado, que sempre proporcionaram um ambiente de companheirismo e cooperação, mesmo nas piores batalhas da vida acadêmica.

A todos que de alguma forma colaboraram com este trabalho, seja com sugestões, esclarecimentos, incentivo ou ajuda direta como ocorreu em algumas implementações. Porém, prefiro não citar nomes para não correr o risco de esquecer alguém.

Aos funcionários do ICMC, sempre prestativos e atenciosos.

À minha família, fonte de incentivo e carinho.

A Deus pelas oportunidades e dádivas em minha vida.

Sumário

SUMÁRIO	I
RESUMO	IV
ABSTRACT	V
LISTA DE FIGURAS	VI
LISTA DE TABELAS	IX
LISTA DE ABREVIACÕES	X
CAPÍTULO I	1
INTRODUÇÃO	1
1.1. Contexto	2
1.1.1. Computação Tradicional	2
1.1.2. Computação Reconfigurável	3
1.1.3. Hardware Tradicional	4
1.1.4. Hardware Reconfigurável	4
1.1.5. Hardware Virtual	5
1.1.6. Robôs Móveis	5
1.1.7. O Projeto ARMOSH	6
1.2. Motivação e Relevância	7
1.3. Objetivo	7
1.4. Organização do Trabalho	8
CAPÍTULO II	9
A TECNOLOGIA FPGA	9
2.1 Circuitos Digitais	9
2.2. Evolução dos PLDs	11
2.3. Arquitetura de FPGAs	14
2.3.1. Tecnologia de Programação	15
2.3.1.1. Tecnologia Antifuse	16
2.3.1.2. Tecnologia Gate Flutuante	17
2.3.1.3. Tecnologia SRAM	18
2.3.2. Arquitetura de Blocos Lógicos	19
2.3.2.1. Blocos de Granularidade Fina	20
2.3.2.2. Blocos de Granularidade Média	21
2.3.2.3. Blocos de Granularidade Grossa	22
2.3.2.4. Lógica Sequencial	22
2.3.3. Arquitetura de Roteamento	22
2.4. A Tecnologia SoC	24
2.4.1. FIPSoC	24
2.4.2. FPSLIC	26
CAPÍTULO III	28
ARQUITETURAS COMERCIAIS DE FPGAS	28
3.1. Os FPGAs da Xilinx	29
3.1.1. A Família XC4000	29
3.1.2. A Família XC6200	32
3.1.3. A Família Virtex	35
3.1.4. A Família Virtex II	37
3.1.5. A Família Virtex II Pro	40
3.2. Os FPGAs da Altera	42
3.2.1. A Família FLEX 8000	42
3.2.2. A Família FLEX 10K	44
3.2.3. A Família APEX 20K	45
3.2.4. A Família APEX II	48

3.2.5. A Família Stratix	48
3.3. <i>Os FPGAs da Atmel</i>	51
3.3.1. A Família AT6000	51
3.3.2. A Família AT40K	54
3.4. <i>Os FPGAs da Actel</i>	56
3.4.1. A Família ACT 3	56
3.5. <i>Os FPGAs da Lattice</i>	58
3.5.1. A Família ORCA Series 4	58
3.6. <i>Comparações</i>	61
CAPÍTULO IV	63
RECONFIGURABILIDADE DE FPGAS	63
4.1. <i>Reconfiguração Parcial</i>	63
4.2. <i>Reconfiguração Dinâmica</i>	64
4.3. <i>Dispositivos que Permitem Reconfiguração Dinâmica</i>	66
4.3.1. A Família APEX da Altera	68
4.3.2. A Família Stratix da Altera	69
4.3.3. A Família ORCA da Lattice	69
4.3.4. As Famílias AT6000 e AT40K da Atmel	70
4.3.5. A Família Virtex da Xilinx	72
4.4. <i>Arquiteturas de Computação Reconfigurável</i>	73
4.4.1. Acoplamento de RPU's com o Host	75
4.4.2. Acoplamento de FPGAs	76
4.4.3. Topologias de Acoplamento de FPGAs	77
4.4.4. Granularidade das RPU's	80
4.5. <i>Considerações Finais</i>	80
CAPÍTULO V	84
FERRAMENTAS DE DESENVOLVIMENTO PARA APLICAÇÕES DINÂMICAS E REMOTAS	84
5.1. <i>Considerações Iniciais</i>	84
5.2. <i>Introdução</i>	85
5.3. <i>Fluxo de Desenvolvimento e Execução</i>	88
5.3.1. Fluxo de Desenvolvimento e Execução Utilizando Ferramentas Estáticas	88
5.3.2. Fluxo de Desenvolvimento Estático e Execução Utilizando JBits	88
5.3.3. Fluxo de Desenvolvimento Dinâmico e Execução Utilizando JBits	90
5.4. <i>O Depurador de Hardware Boardscope</i>	91
5.5. <i>A API XHWIF</i>	92
5.5.1. XHWIF Interface	93
5.5.2. XHWIF Server	93
5.5.3. Sistemas Suportados pela API XHWIF	95
5.6. <i>O Simulador VirtexDS</i>	96
5.7. <i>O Waveform Viewer</i>	96
5.8. <i>Dispositivos Suportados por JBits</i>	97
5.9. <i>IRL</i>	97
5.10. <i>Considerações Finais</i>	98
CAPÍTULO VI	100
IMPLEMENTAÇÕES	100
6.1. <i>Considerações Iniciais</i>	100
6.2. <i>Sistema de Reconfiguração</i>	101
6.3. <i>Resultados e Considerações Finais</i>	104
CAPÍTULO VII	106
CONCLUSÕES	106
7.1. <i>Considerações deste Trabalho</i>	106
7.2. <i>Contribuições deste Trabalho</i>	106
7.3. <i>Sugestões para Desenvolvimento Futuro</i>	107
REFERÊNCIAS BIBLIOGRÁFICAS	110

APÊNDICE A.....	116
MODELO DE PROGRAMAÇÃO JBITS	116
A.1. O Construtor.....	116
A.2. O Método Read Bitstream	117
A.3. O Método Set Bits.....	117
A.4. O Método Write Bitstream.....	118
A.5. O Método Get Bits	119
A.6. Fluxo de Desenvolvimento	119
APÊNDICE B	120
DIAGRAMAS ESQUEMÁTICOS.....	120
B.1. Interface de Reconfiguração Multi-FPGAs	120
B.2. Interconexões dos FPGAs XC4025E.....	121
APÊNDICE C	122
PINAGEM E ENDEREÇAMENTO	122
C.1. Pinagem Externa da Interface de Reconfiguração	122
C.2. Pinagem e Endereçamento da Porta Paralela de um PC.....	123
APÊNDICE D	124
CÓDIGOS FONTES.....	124
D.1. Código do Driver	124
D.2. Código do Controlador.....	131
D.3. Código do Site.....	136

Resumo

Neste trabalho estudou-se diversas arquiteturas de dispositivos FPGAs presentes no mercado, visando a utilização desta tecnologia em arquiteturas de computação reconfigurável. Especificamente foram investigados recursos e técnicas de reconfigurabilidade dinâmica destes dispositivos.

A possibilidade de reconfigurar dinamicamente o *hardware* cria diversas expectativas de superação das arquiteturas de computação tradicional. Surge o conceito de *hardware* virtual, assim como inúmeras dificuldades em utilizar efetivamente esta tecnologia. Outra característica promissora que a tecnologia FPGA oferece é a possibilidade de realizar *upgrades* remotos do *hardware*, sem a necessidade de substituição física de equipamentos ou parte deles.

Em adicional, foi implementado um sistema multi-FPGAs com dispositivos reconfiguráveis individualmente de forma local ou remota. Este sistema poderá ser a base para uma arquitetura de computação reconfigurável, sendo ela dinâmica ou não.

Abstract

In this work it was studied several architectures of FPGAs devices available in the market, seeking the use of this technology in architectures of reconfigurable computing. It was specifically researched resources and techniques of dynamic reconfiguration of these devices.

The possibility of reconfigure the hardware dynamically creates countless expectations to overcome the architectures of traditional computing. The concept of virtual hardware emerges, as well as a lot of difficulties in using actually this technology. Another promising characteristic that the technology FPGA offers it is the possibility to accomplishing remote upgrades of the hardware, without the need of physical substitution of equipments or of parts of them.

In addition, a system multi-FPGAs was implemented with individually reconfigurable devices in local or remote way. This system can be the base for an architecture of reconfigurable computing, being it dynamics or not.

Lista de Figuras

FIGURA 2.1: TECNOLOGIAS PARA PROJETOS DE SISTEMAS DIGITAIS.	10
FIGURA 2.2: ESTRUTURA DE UM PLA.	12
FIGURA 2.3: ESTRUTURA DE UM PAL.	12
FIGURA 2.4: ESTRUTURA DE UM FPGA.	15
FIGURA 2.5: <i>ANTIFUSE</i> PLICE DA ACTEL: (A) SEÇÃO TRANSVERSAL; (B) CANAL <i>ANTIFUSE</i> FORMADO; E (C) CONTATOS DO <i>ANTIFUSE</i>	16
FIGURA 2.6: COMUTADOR PROGRAMÁVEL BASEADO EM EPROM.	17
FIGURA 2.7: TECNOLOGIA DE PROGRAMAÇÃO SRAM.	19
FIGURA 2.8: CIRCUITO LÓGICO DE 4 ENTRADAS.	22
FIGURA 2.9: ARQUITETURA GERAL DE ROTEAMENTO DE UM FPGA.	23
FIGURA 2.10: ARQUITETURA INTERNA DE UM FIPSoC.	25
FIGURA 2.11: ARQUITETURA INTERNA DE UM FPSLIC.	27
FIGURA 3.1: TIPOS DE ARQUITETURAS.	28
FIGURA 3.2: BLÓCO LÓGICO DA FAMÍLIA XC4000.	30
FIGURA 3.3: SEGMENTOS DE TRILHA SIMPLES E DUPLO DA SÉRIE XC4000.	31
FIGURA 3.4: DETALHES INTERNOS DE UM BLOCO DE COMUTACÃO (PSM).	32
FIGURA 3.5: UMA CÉLULA BÁSICA DA FAMÍLIA XC6200.	33
FIGURA 3.6: UNIDADE LÓGICA DA FAMÍLIA XC6200.	33
FIGURA 3.7: EXEMPLOS DE FUNÇÕES LÓGICAS IMPLEMENTÁVEIS PELAS UNIDADES LÓGICAS.	34
FIGURA 3.8: BLOCO DE CÉLULAS 4x4.	34
FIGURA 3.9: DISPOSITIVO XC6216.	35
FIGURA 3.10: ARQUITETURA DO FPGA VIRTEX.	36
FIGURA 3.11: CLB DA FAMÍLIA VIRTEX COMPOSTO POR DOIS <i>SLICES</i>	36
FIGURA 3.12: DETALHES DE UM <i>SLICE</i> DA FAMÍLIA VIRTEX.	37
FIGURA 3.13: ARQUITETURA DO FPGA VIRTEX II.	38
FIGURA 3.14: ARQUITETURA DO FPGA VIRTEX II Pro.	40
FIGURA 3.15: ARQUITETURA FLEX 8000 DA ALTERA.	43
FIGURA 3.16: DETALHES DE UM LE DA FAMÍLIA FLEX 8000.	43
FIGURA 3.17: ARQUITETURA DA FAMÍLIA FLEX 10000 DA ALTERA.	44
FIGURA 3.18: ARQUITETURA MULTICORE DA FAMÍLIA APEX 20K.	45
FIGURA 3.19: ESTRUTURA LAB.	46
FIGURA 3.20: ESTRUTURA MEGALAB.	46
FIGURA 3.21: DIAGRAMA DE BLOCOS DO PBM, NIOS E PERIFÉRICOS.	47
FIGURA 3.22: ARQUITETURA DA FAMÍLIA STRATIX.	49
FIGURA 3.23: ESTRUTURA LBA DA FAMÍLIA STRATIX.	50

FIGURA 3.24: ARRAY SIMÉTRICO CERCADO POR BLOCOS DE I/O DO AT6000.....	52
FIGURA 3.25: CANAIS DE ROTEAMENTO DE UM SETOR DO AT6000.	52
FIGURA 3.26: TRILHAS LOCAIS E EXPRESS DO AT6000.....	53
FIGURA 3.27: ESTRUTURA DA CÉLULA DO AT6000.....	53
FIGURA 3.28: ESTRUTURA DE UM DOS CINCO PLANOS DO BARRAMENTO DO AT40K.....	54
FIGURA 3.29: CONEXÕES DE UMA CÉLULA COM SUAS CÉLULAS VIZINHAS.....	55
FIGURA 3.30: CONEXÕES DE UMA CÉLULA COM OS SEGMENTOS DE TRILHAS LOCAIS.	55
FIGURA 3.31: ESTRUTURA DA CÉLULA DO AT40K.....	56
FIGURA 3.32: ARQUITETURA DOS FPGAS DA ACTEL.....	57
FIGURA 3.33: ARQUITETURA DE ROTEAMENTO DA FAMÍLIA ACT 3.....	58
FIGURA 3.34: ARQUITETURA ORCA SERIES 4.	59
FIGURA 3.35: DETALHES DE UMA PFU DA FAMÍLIA ORCA SERIES 4.	60
FIGURA 4.1: CLASSIFICAÇÃO DE FPGAS DE ACORDO COM SUA CONFIGURABILIDADE.	63
FIGURA 4.2: EXEMPLO DE RECONFIGURAÇÃO DINÂMICA.	65
FIGURA 4.3: EXECUÇÃO DAS TAREFAS COM RECONFIGURAÇÃO TOTAL.	65
FIGURA 4.4: EXECUÇÃO DAS TAREFAS COM RECONFIGURAÇÃO DINÂMICA.....	65
FIGURA 4.5: RECONFIGURAÇÃO UTILIZANDO <i>CACHE LOGIC</i>	71
FIGURA 4.6: <i>HARDWARE</i> DEDICADO COM COMPUTADOR <i>HOST</i> , PARA ACELERAR TAREFAS COMPUTACIONAIS.	73
FIGURA 4.7: TIPOS DE ACOPLAMENTO DE RPUS COM O SISTEMA <i>HOST</i>	75
FIGURA 4.8: ACOPLAMENTO DE FPGAS DA APEX ATRAVÉS DO BARRAMENTO PBM.	76
FIGURA 4.9: ACOPLAMENTO DE DISPOSITIVOS, ATRAVÉS DO BARRAMENTO AMBA.....	77
FIGURA 4.10: TOPOLOGIA <i>MESH</i>	77
FIGURA 4.11: TOPOLOGIA <i>CROSSBAR</i>	78
FIGURA 4.12: TOPOLOGIA DO APTIX AXB-AP4.....	79
FIGURA 5.1: FLUXO DE DESENVOLVIMENTO E EXECUÇÃO UTILIZANDO FERRAMENTAS ESTÁTICAS.	88
FIGURA 5.2: FLUXO DE DESENVOLVIMENTO ESTATICO E EXECUÇÃO UTILIZANDO JBITS.....	89
FIGURA 5.3: FLUXO DE DESENVOLVIMENTO DINÂMICO E EXECUÇÃO UTILIZANDO JBITS.....	90
FIGURA 5.4: INTERFACE GRÁFICA DO BOARDSCOPE.....	92
FIGURA 5.5: ACESSANDO REMOTAMENTE O <i>HARDWARE</i> UTILIZANDO A INTERFACE XHWIF.	94
FIGURA 5.6: ACESSO REMOTO À INTERFACE XHWIF SERVER.	95
FIGURA 5.7: INTERFACE GRÁFICA DO WAVEFORM VIEWER.	96
FIGURA 5.8: ELEMENTOS DE UMA IRL.	98
FIGURA 6.1: DIAGRAMA DO SISTEMA IMPLEMENTADO.....	102
FIGURA 6.2: FOTOS DA PLACA CONFECCIONADA EM <i>WIRE-WRAP</i>	102
FIGURA A.1: SINTAXE PARA A CHAMADA AO MÉTODO CONSTRUTOR E EXEMPLO.	116
FIGURA A.2: SINTAXE PARA A CHAMADA AO MÉTODO READ() E EXEMPLO.	117
FIGURA A.3: SINTAXE PARA A CHAMADA AO MÉTODO SET() E EXEMPLO.....	118
FIGURA A.4: SINTAXE PARA A CHAMADA AO MÉTODO WRITE() E EXEMPLO.....	118
FIGURA A.5: SINTAXE PARA A CHAMADA AO MÉTODO GET() E EXEMPLO.	119

FIGURA A.6: EXEMPLO DE UMA SIMPLES APLICAÇÃO JBITS.....	119
FIGURA B.1: DIAGRAMA ESQUEMÁTICO DA INTERFACE DE RECONFIGURAÇÃO MULTI-FPGAS.	120
FIGURA B.2: DIAGRAMA ESQUEMÁTICO DE INTERCONEXÕES INDIVIDUAIS DOS FPGAS XC4025E.....	121

Lista de Tabelas

TABELA 2.1: SUMÁRIO DAS TECNOLOGIAS DE PROGRAMAÇÃO.....	19
TABELA 2.2: LUT DE 4 LINHAS DE ENDEREÇAMENTO IMPLEMENTANDO O CIRCUITO DA FIGURA 2.5.....	21
TABELA 3.1: FABRICANTES DE FPGAS X TECNOLOGIA DE PROGRAMAÇÃO.....	29
TABELA 3.2: COMPARAÇÃO DE DISPOSITIVOS XILINX.....	41
TABELA 3.3: COMPARAÇÃO DE DISPOSITIVOS ALTERA.....	51
TABELA 3.4: COMPARAÇÃO ENTRE DISPOSITIVOS.....	61
TABELA 4.1: CAPACIDADE DE RECONFIGURAÇÃO DE FPGAS SRAM.....	73
TABELA 6.1: TEMPOS MÉDIOS DE RECONFIGURAÇÃO DOS DISPOSITIVOS XC4025E.....	105
TABELA C.1: PINAGEM EXTERNA DA INTERFACE DE RECONFIGURAÇÃO.....	122
TABELA C.2: PINAGEM E ENDEREÇAMENTO DA PORTA PARALELA DE UM PC.....	123

Lista de Abreviações

A/D	<i>Analogical / Digital</i>
ALU	<i>Arithmetic Logic Unit</i>
AMBA	<i>Advanced Microcontroller Bus Architecture</i>
API	<i>Application Program Interface</i>
ARMOSH	<i>Aprendizado em Robôs Móveis via Software e Hardware</i>
ASIC	<i>Application Specific IC</i>
BRAM	<i>Block RAM</i>
CAB	<i>Configurable Analog Block</i>
CAD	<i>Computer Aided Design</i>
CAM	<i>Content-Addressable Memory</i>
CI	<i>Circuito Integrado</i>
CLB	<i>Configurable Logic Block</i>
CPLD	<i>Complex PLD</i>
D/A	<i>Digital / Analogical</i>
DCM	<i>Digital Clock Manager</i>
DISC	<i>Dynamic Instruction Set Computer</i>
DLL	<i>Delay-Locked Loop</i>
DMC	<i>Digital Macro Cell</i>
DPGA	<i>Dynamic Probramable Gate Array</i>
DSP	<i>Digital Signal Processing</i>
EAB	<i>Embedded Array Block</i>
EBR	<i>Embedded Block RAM</i>
EDA	<i>Eletronic Design Automation</i>
EEPROM	<i>Electrical Erasable PROM</i>
EPLD	<i>Erasable PLD</i>
EPROM	<i>Erasable PROM</i>
ESB	<i>Embedded System Block</i>
FET	<i>Field-Effect Transistors</i>
FIFO	<i>First-In First-Out</i>
FIPSoC	<i>Field Programmable SoC</i>
FPGA	<i>Field Programmable Gate Array</i>
FPIC	<i>Field Programmable Interconnect Component</i>
FPSLIC	<i>Field Programmable System Level IC</i>
GRM	<i>General Routing Matrix</i>
HDL	<i>Hardware Description Language</i>
HTML	<i>Hipertext Markup Language</i>
HTTP	<i>Hipertext Transfer Protocol</i>
I/O	<i>Input / Output</i>
IC	<i>Integrated Circuit</i>
IOB	<i>I/O Block</i>
IOE	<i>I/O Element</i>
IP	<i>Internet Protocol</i>
IRL	<i>Internet Reconfigurable Logic</i>
ISP	<i>In System Programability</i>
LAB	<i>Logic Array Block</i>

LABIC	Laboratório de Inteligência Computacional
LCR	Laboratório de Computação Reconfigurável
LE	<i>Logic Element</i>
LUT	<i>Look Up Table</i>
MPGA	<i>Mask Programmable Gate Array</i>
ONO	<i>Oxido-Nitreto-Oxido</i>
PAL	<i>Programmable Array Logic</i>
PBM	<i>Peripheral Bus Module</i>
PC	<i>Personal Computer</i>
PCI	<i>Peripheral Component Interconnect</i>
PFU	<i>Programmable Function Unit</i>
PIO	<i>Parallel I/O</i>
PIO	<i>Programmable I/O cell</i>
PLA	<i>Programmable Logic Array</i>
PLC	<i>Programmable Logic Cell</i>
PLD	<i>Programmable Logic Device</i>
PLICE	<i>Programmable Low Impedance Circuit Element</i>
PLL	<i>Phase-Locked Loop</i>
PRISC	<i>Programmable Reduced Instruction Set Computers</i>
PRISM	<i>Processor Reconfiguration through Instruction-Set Metamorphosis</i>
PROM	<i>Programmable ROM</i>
PSM	<i>Programmable Switch Matrix</i>
RAM	<i>Random Access Memory</i>
RAW	<i>Reconfigurable Architecture Workstation</i>
ROM	<i>Read Only Memory</i>
RPM-2	<i>Rapid Prototyping engine for Multiprocessor systems 2</i>
RPU	<i>Reconfigurable Processing Unit</i>
RTL	<i>Register Transfer Logic</i>
RTR	<i>Run-Time Reconfiguration</i>
SLIC	<i>Supplemental Logic and Interconnect Cell</i>
SoC	<i>System-on-Chip</i>
SOPC	<i>System-on-a-Programmable-Chip</i>
SPLD	<i>Simple PLD</i>
SRAM	<i>Static RAM</i>
TCP	<i>Transmission Control Protocol</i>
UART	<i>Universal Asynchronous Receiver/Transmitter</i>
ULA	<i>Unidade Lógica Aritmética</i>
VHDL	<i>VHSIC HDL</i>
VHSIC	<i>Very High Speed IC</i>
VirtexDS	<i>Virtex Device Simulator</i>
VLSI	<i>Very Large Scale Integration</i>
XHWIF	<i>Xilinx Hardware Interface</i>

Introdução

Nos últimos anos, a tecnologia de dispositivos FPGAs (*Field Programmable Gate Arrays*) tem evoluído significativamente, alcançando elevados níveis de densidade, altos índices de desempenho e menores custos de fabricação. Esta evolução tem tornado cada vez menor a distância entre FPGAs e CIs (Circuitos Integrados) customizados. Além dos avanços em capacidade, desempenho e custos, os fabricantes de FPGAs têm introduzido, no decorrer dos anos, cada vez mais recursos de reconfigurabilidade.

Os recursos de reconfigurabilidade recentemente implantados por fabricantes de FPGAs têm possibilitado o projeto de sistemas dinamicamente reconfiguráveis. O termo “dinamicamente reconfigurável”, representa a possibilidade de se alterar parcialmente a funcionalidade de um dispositivo enquanto ativo, sem prejudicar o funcionamento da lógica restante que pode estar em operação [1].

A disponibilidade de tais recursos, somado aos elevados níveis de densidade e desempenho tem alterado o processo de desenvolvimento de sistemas digitais. Agora, torna-se possível desenvolver circuitos de *hardware*, nunca antes vistos [2, 3, 4]. Por exemplo, em um circuito integrado digital torna-se possível alterar funcionalmente uma área do *chip*, enquanto outros circuitos no mesmo componente continuam a funcionar, sem qualquer degradação no desempenho dos circuitos estáticos.

Estes novos sistemas digitais proporcionam muitas oportunidades e também apresentam novos desafios [5, 6].

1.1. Contexto

1.1.1. Computação Tradicional

Ao implementar um sistema computacional, tradicionalmente decide-se entre implementações em *hardware* customizado (fixo) ou *software*. Em alguns sistemas, toma-se esta decisão para cada sub-tarefa, alocando algumas em *hardware* e outras em *software* em processadores de uso geral [7]. Os projetos em *hardware* oferecem alto desempenho pois:

- são customizados para o problema em particular, não há um *overhead* extra para interpretação de instruções ou circuitos extras para solucionar um problema mais genérico; e
- comparados com implementações em *software*, são muito rápidos devido à execução espacial, com um alto grau de paralelismo.

Já as implementações em *software* exploram um processador de uso geral que interpreta um determinado fluxo de dados como instruções que indicam quais operações deve realizar. Como resultado o *software* é:

- flexível, uma tarefa pode ser mudada simplesmente trocando-se o código de instruções em uma memória regravável;
- comparado com implementações em *hardware*, muito lento devido principalmente à execução temporal; e
- ineficiente, uma vez que os operadores podem ser inadequados à tarefa a ser realizada.

Em implementações espaciais, cada operador existe em um diferente ponto no espaço (área do *chip*), permitindo que a computação explore o paralelismo e alcance uma alta taxa de transferência de dados e baixa latência. Em implementações temporais, um pequeno número de recursos computacionais é reutilizado ao longo do tempo, permitindo que a computação seja implementada de forma compacta.

1.1.2. Computação Reconfigurável

Em termos básicos, a computação reconfigurável combina a velocidade do *hardware* com a flexibilidade do *software*. Os sistemas de computação reconfigurável são plataformas onde a arquitetura pode ser modificada em tempo real, para melhor se adequar à aplicação que será executada. Deste modo, o processador reconfigurável passa a trabalhar com uma arquitetura desenvolvida exclusivamente para aquele determinado tipo de aplicação, permitindo uma eficiência muito maior do que a normalmente encontrada em processadores de uso geral. Isto ocorre porque o *hardware* é otimizado para executar os algoritmos necessários para aquela aplicação específica. Ou seja, para se obter um melhor desempenho de um algoritmo, este deve ser executado num *hardware* específico para ele.

Por exemplo, estudantes da University of California mostraram que um FPGA Altera pode checar 800.000 chaves por segundo enquanto um computador Pentium checa apenas 41.000 chaves por segundo [8]. Na Universidade de São Paulo foi implementada uma rede neural, em um FPGA Altera, capaz de responder aos dados de entrada em 10ns, enquanto a mesma rede executada em um computador Pentium necessitava de pelo menos 23us para responder aos mesmos dados de entrada [9].

Outro exemplo é a utilização de FPGAs, em plataformas computacionais de uso geral, como coprocessadores genéricos, podendo ser configurado de acordo com a necessidade do *software* que estiver rodando. A Xilinx Inc., em parceria com a Compac Systems e a University of Glasgow, implementaram em FPGAs filtros para o Adobe PhotoShop [10] e renderizadores de fontes PostScript [11].

Acentuados ganhos de desempenho podem ser obtidos com a utilização destas arquiteturas reconfiguráveis, onde os *chips* precisam ser reconfigurados no sistema e em tempo real [12, 13]. Embora os recursos para esta reconfiguração ainda sejam muito pobres, segundo DeHon [7], arquiteturas de computação reconfigurável complementam as arquiteturas convencionais, compostas apenas de *hardwares* específicos e processadores de uso geral.

1.1.3. *Hardware Tradicional*

Os computadores atuais são sistemas de *hardware* fixo, baseados em microprocessadores. Normalmente esses processadores são de uso geral, o que os tornam lentos para o processamento de alguns problemas específicos. Cada nova geração desses microprocessadores possibilita um aumento de desempenho na execução das aplicações. Entretanto, na maioria dos casos, a aplicação precisa ser recompilada para melhor usufruir as novas características desses processadores.

1.1.4. *Hardware Reconfigurável*

Um recente avanço nos esforços de obter maior poder de processamento é a utilização de PLDs (*Programmable Logic Devices*) para a construção de *hardwares* reconfiguráveis. Estes dispositivos contêm *hardware* configurável que pode ser reprogramado para a aplicação sendo executada, tornando-se possível obter um desempenho comparável aos ASICs (*Application Specific Integrated Circuits*) sem o custo e risco inerentes a esta tecnologia. Os PLDs mais empregados na construção de *hardwares* reconfiguráveis são os FPGAs. Os FPGAs contêm arranjos de blocos lógicos reconfiguráveis interligados por recursos de roteamento [14, 15].

O principal benefício dos FPGAs, e mais amplamente dos dispositivos lógicos programáveis, é que eles introduzem uma classe de dispositivos configuráveis pós-fabricação que suportam a realização de computação espacial.

Além do benefício óbvio do paralelismo espacial, permitindo a realização de mais operações por ciclo, destaca-se também a ótima flexibilidade dos dispositivos FPGAs, a alta capacidade lógica, a facilidade de programação pelo usuário final, que reduz o tempo de desenvolvimento para minutos, o reduzido tempo de projeto, que possibilita um rápido *time-to-market*, e o baixo custo dos dispositivos e custos fixos do projeto.

Muitos trabalhos têm sido realizados na construção de arquiteturas reconfiguráveis PRISC [16], DISC [17], Garp [18, 19], RAW [20, 21], entre vários outros. Estes trabalhos demonstram que esta área tende a se desenvolver muito, a médio e longo prazo.

1.1.5. *Hardware Virtual*

Semelhante ao conceito de Memória Virtual surge o termo “*Hardware Virtual*”, que se refere a *hardwares* com capacidade de reconfiguração dinâmica que podem ser reconfigurados e escalonados por um *host*, compartilhando *hardware* para as aplicações. Entretanto, as técnicas de particionamento temporal ainda não estão maduras. As implicações práticas para a aceleração de *hardware* não foram encontradas e os algoritmos propostos para a aceleração de programas em ambientes de computação reconfigurável, segundo Cardoso [22], são pouco satisfatórios. As pesquisas para solucionar as dificuldades relatadas, assim como as analogias, não param por aqui. Com a tecnologia do *hardware* reconfigurável dinamicamente, também surgem, entre outros, os termos e conceitos de paginação, multitarefa [3], preempção [23] e sistemas operacionais para *hardware* [24];

1.1.6. Robôs Móveis

Nos últimos anos tem-se observado um vertiginoso crescimento do potencial de sistemas robóticos. Numa primeira etapa houve um grande desenvolvimento na área de robótica industrial. A implantação no chão de fábrica de robôs industriais trouxe benefícios como a melhoria da eficácia, da qualidade, redução da mão-de-obra, além de maior eficiência, confiabilidade e redução de custos. Vantagens adicionais incluem a capacidade de realizar tarefas para as quais os humanos teriam grande dificuldade, e a remoção de humanos de tarefas em ambientes perigosos.

Numa nova etapa de evolução, pesquisadores em robótica têm concentrado esforços na construção de robôs móveis, introduzindo as capacidades de mobilidade e autonomia para reagir adequadamente ao ambiente, o que abre um vasto campo de novas aplicações e conseqüentemente muitos desafios.

Os problemas enfrentados no desenvolvimento de robôs móveis geralmente estão relacionados com a necessidade de interagir com os objetos físicos e entidades do ambiente. A plataforma deve ser capaz de navegar a partir de uma posição conhecida para uma nova localização desejada, evitando colisões com objetos fixos e móveis durante sua rota [25].

Para funcionar em um ambiente desestruturado, desconhecido ou dinâmico, um robô móvel deve ser capaz de perceber as circunstâncias e gerar ações que são apropriadas para o ambiente e para os objetivos do sistema robótico. Robôs móveis precisam combinar informações de várias fontes diferentes. Um bom sistema de sensoriamento é fundamental para adquirir dados que descrevem o ambiente, em tempo real e com alta resolução, e ainda com as limitações dos recursos computacionais e de consumo de energia do veículo móvel.

Mesmo atualmente, após décadas de pesquisas e desenvolvimento, a concretização dos “sonhos” dos pesquisadores, incluindo todas as capacidades e funcionalidades necessárias para as aplicações vislumbradas, esbarra em uma série de dificuldades tecnológicas. As principais limitações no desenvolvimento das aplicações, são os recursos computacionais (algoritmos e arquiteturas), recursos de comunicação e sensores. O emprego de *hardware* e computação reconfigurável, em arquiteturas de robôs móveis, traz aos engenheiros e cientistas da computação grandes expectativas de superar as limitações das arquiteturas atualmente utilizadas.

1.1.7. O Projeto ARMOSH

O projeto ARMOSH [28] (Aprendizado em Robôs Móveis via *Software* e *Hardware*) vem sendo desenvolvido pelo Laboratório de Computação Reconfigurável – LCR e pelo Laboratório de Inteligência Computacional - LABIC, do Instituto de Ciências Matemáticas e de Computação – ICMC, da Universidade de São Paulo – USP. Este projeto propõe o estudo e implementação de algoritmos de aprendizado de robôs móveis tanto no nível de *software* quanto no nível de *hardware*, visando uma comparação de desempenho dos principais algoritmos de aprendizado em ambas as implementações.

O projeto tem a finalidade de buscar novas alternativas na área, inicialmente através do estudo, trabalhando com implementações tanto no nível de *software* quanto no nível de *hardware* dos principais algoritmos de aprendizado de máquina, onde serão abordados:

- algoritmos para navegação, levando-se em conta dependência do número de sensores do robô (sonar, infravermelho ou câmera de vídeo), definição do

ambiente (fechado, aberto, desconhecido);

- algoritmos que envolvem aprendizado via programação dinâmica, via redes neurais e aprendizado com reforço; e
- algoritmos que envolvem reconhecimento de objetos via redes neurais, via técnicas clássicas de processamento de imagens tais como, forma, textura e cor, e em ambientes estáticos e dinâmicos.

O estudo levantado neste projeto será direcionado com vistas a sua evolução futura, através da construção de uma biblioteca dos algoritmos estudados em sua versão *hardware/software* e na montagem de um kernel, em longo prazo, que controlará de forma inteligente a execução destes algoritmos, proporcionando grande versatilidade ao sistema, pois em tempo real, o sistema poderá decidir qual algoritmo é mais adequado para realização de uma determinada tarefa, utilizando ainda, a melhor implementação (via *hardware/software*) do algoritmo escolhido.

1.2. Motivação e Relevância

A utilização da reconfigurabilidade dos FPGAs nos sistemas digitais é motivada por muitos fatores. Talvez, o mais importante seja o fato de que a reconfiguração possibilita a implementação de vários circuitos funcionais com o emprego de poucos FPGAs, um resultado surpreendente se comparado com outras tecnologias. O uso de FPGAs para o desenvolvimento de novas formas de máquinas computacionais é uma outra razão. Engenheiros e cientistas da computação, cansados de deparar com as limitações das máquinas de von Neumann, vêm no uso da reconfigurabilidade dos FPGAs uma alternativa excitante.

1.3. Objetivo

Neste trabalho, buscou-se estudar diversos dispositivos FPGAs do mercado, bem como suas técnicas de reconfigurabilidade dinâmica, atingindo o estado da arte em reconfigurabilidade de FPGAs. Também foram apresentadas vantagens e desvantagens entre diferentes famílias e fabricantes, assim como soluções e dificuldades, propondo

alternativas na utilização desta tecnologia.

Os resultados obtidos com este trabalho contribuirão com o projeto ARMOSH, na busca de uma arquitetura ideal para implementar os algoritmos de robótica em *hardware*.

1.4. Organização do Trabalho

O capítulo II apresenta um breve histórico evolutivo dos PLDs até os FPGAs, apresentando os aspectos mais relevantes à arquitetura FPGA quanto aos tipos de tecnologia de programação, arquitetura das células (blocos lógicos) e as estruturas de roteamento.

O capítulo III apresenta e compara algumas das principais arquiteturas comerciais, exemplificando os conceitos apresentados no capítulo II.

O capítulo IV apresenta o conceito de reconfigurabilidade de FPGAs, sendo classificados em: programáveis; reconfiguráveis; parcialmente reconfiguráveis; e dinamicamente reconfiguráveis. Também são apresentados dispositivos do mercado que suportam reconfiguração dinâmica e conceitos de arquiteturas de computação reconfigurável, discutindo as principais dificuldades.

O capítulo V apresenta o conceito e as dificuldades do desenvolvimento de *hardware* em tempo de execução para arquiteturas reconfiguráveis dinamicamente.

O capítulo VI apresenta o sistema de reconfiguração de FPGAs desenvolvido, composto por: uma interface em *hardware* para configuração e reconfiguração de sistemas multi-FPGAs; um driver para PCs; um *software* de controle local e remoto do sistema de reconfiguração; e uma placa de prototipação multi-FPGAs.

No capítulo VII encontram-se as conclusões deste trabalho onde são apresentadas algumas sugestões para trabalhos futuros.

A Tecnologia FPGA

“Um ASIC é como ter que utilizar um velho suéter azul da década de 70 para sempre. Com um FPGA, você tem um hardware, mas pode modificá-lo sempre que os estilos mudarem.”

Shackelford (HP Labs), 2000

2.1 Circuitos Digitais

Os circuitos digitais têm sofrido grande evolução nas últimas décadas. As constantes mudanças na tecnologia têm transformado de forma radical todo o processo de projeto de *hardware*.

Os componentes dos circuitos digitais evoluíram de transistores individuais para circuitos integrados VLSI (*Very Large Scale Integration*). A utilização de ferramentas EDA (*Electronic Design Automation*) tem simplificado e acelerado todo o ciclo de projeto. Atualmente, não é mais necessário desenhar portas lógicas individuais e planejar todas suas interconexões. As linguagens de descrição de *hardware* (HDLs) estão hoje consolidadas no meio acadêmico e industrial como forma padrão na elaboração de projetos. Existem também, ferramentas de síntese lógica automática, disponíveis para mapear circuitos em diversas tecnologias [15].

Todas essas mudanças na tecnologia exigem uma prototipação cada vez mais rápida, pois o ciclo de vida dos produtos modernos está tornando-se cada vez mais curto em relação ao tempo necessário para o projeto e desenvolvimento dos mesmos.

Os CIs digitais podem ser construídos utilizando-se de diversas tecnologias diferentes, figura 2.1, a escolha da tecnologia adequada deve ser realizada com base no tipo de projeto que se pretende executar.

FIGURA 2.1: TECNOLOGIAS PARA PROJETOS DE SISTEMAS DIGITAIS.

As implementações de circuitos podem ser agrupadas em diversas categorias de projeto[9]:

- CIs customizados ou ASICs: São aqueles que necessitam de um processo de fabricação especial, que requer máscaras específicas para cada projeto. Características desse tipo de implementação são os custos de projeto extremamente altos e o tempo de desenvolvimento longo. Em aplicações que requerem um grande volume de produção, o alto custo do projeto e dos testes é amortizado;
- MPGAs (*Mask Programmable Gate Arrays*): Nesse tipo de implementação, o processo de fabricação é agilizado pelo uso de máscaras genéricas de módulos pré-projetados, mas ainda necessita de máscaras específicas para a interconexão dos módulos. O projeto é normalmente facilitado por uma biblioteca de células, proporcionando um tempo de desenvolvimento mais curto e custos mais baixos em relação aos CIs customizados;
- *Standard Cells*: Essa tecnologia se assemelha muito a das MPGAs, o projeto também é facilitado pelo uso de módulos pré-projetados. Os módulos (*standard cells*) são geralmente salvos em bancos de dados. Os projetistas selecionam as células desejadas (nesses bancos de dados) para realizar seus projetos. Em comparação aos CIs customizados, os circuitos implementados

em *standard cells* são menos eficientes em tamanho e desempenho, entretanto, seu custo de desenvolvimento é mais baixo; e

- PLDs: Essa tecnologia possui como principal característica a capacidade de programação (configuração) pós-fabricação pelo usuário, facilitando assim as mudanças de projetos. Em comparação com outras tecnologias, os PLDs apresentam ciclo de projeto muito curto com baixos custos de desenvolvimento.

O mercado de PLDs encontra-se em plena expansão, de forma que atualmente existem diversos fabricantes e modelos de dispositivos desse tipo. Uma das principais tarefas do projetista hoje é pesquisar e selecionar, dentre as opções disponíveis no mercado, qual a que melhor atende suas necessidades.

2.2. Evolução dos PLDs

O primeiro tipo de *chip* programável pelo usuário, que podia implementar circuitos lógicos, foi a memória PROM (*Programmable Read Only Memory*), em que cada linha de endereço poderia ser usada como entrada do circuito e as linhas de dados como saídas. As funções lógicas, entretanto, raramente requerem mais que alguns termos de produto, e uma PROM contém um decodificador completo para seus endereços de entradas. A PROM, portanto, é uma arquitetura ineficiente para a realização de circuitos lógicos, e são raramente utilizadas na prática para esse fim [26].

O primeiro dispositivo desenvolvido especificamente para a implementação de circuitos lógicos foram os PLAs (*Programmable Logic Arrays*). Um PLA consiste de dois níveis de portas lógicas, figura 2.2: um plano de portas *wired-AND* seguido por um plano de portas *wired-OR*, ambos programáveis. Um PLA é estruturado de tal forma que cada saída do plano AND pode corresponder a qualquer termo de produto das entradas. Similarmente, cada saída do plano OR pode ser configurada para produzir a soma lógica de qualquer saída do plano AND. Com essa estrutura, os PLAs são bem adequados para implementação de funções lógicas na forma de soma de produtos. Eles são muito versáteis, pois tanto os termos AND como os termos OR podem ter muitas entradas.

FIGURA 2.2: ESTRUTURA DE UM PLA.

Quando os PLAs foram introduzidos no início dos anos 70, pela Philips, suas principais deficiências eram o alto custo de fabricação e o pobre desempenho em termos de velocidade. Ambas as desvantagens eram devido aos níveis de lógica configurável, porque os planos lógicos programáveis eram difíceis de serem fabricados e introduziam atrasos de propagação significantes. Para superar essas deficiências, dispositivos PAL (*Programmable Array Logic*) foram desenvolvidos. Os PALs possuem um único nível de programação, consistindo de um plano de portas *wired-AND* programáveis que alimenta portas OR fixas, figura 2.3. Para compensar a falta de generalidade devido ao plano OR fixo, diversos modelos de PALs foram produzidos, com diferentes números de entradas e saídas, e vários tamanhos de portas OR.

FIGURA 2.3: ESTRUTURA DE UM PAL.

Os PALs geralmente contêm *flip-flops* conectados as saídas das portas OR para que circuitos seqüenciais possam ser implementados. Dispositivos PAL foram

importantes porque, quando introduzidos, tiveram um profundo efeito no projeto de hardware digital, e também foram a base para algumas das novas e mais sofisticadas arquiteturas. Variantes da arquitetura básica do PAL são encontrados em outros produtos conhecidos por diferentes siglas. Todos os pequenos PLDs, como PLAs, PALs, e outros dispositivos similares são agrupados em uma única categoria chamada SPLDs (*Simple PLDs*), cujas características mais importantes são o baixo custo e alto desempenho.

Com o avanço da tecnologia, tornou-se possível a produção de dispositivos com maior capacidade que os SPLDs. A dificuldade de aumentar a capacidade da arquitetura de SPLDs, é que a estrutura dos planos lógicos programáveis aumenta muito rapidamente com o aumento do número de entradas. O único modo viável de produzir dispositivos com maior capacidade baseados nas arquiteturas de SPLDs foi integrar múltiplos SPLDs em um único *chip* e prover interconexões programáveis para conectar os blocos SPLDs. Muitos produtos PLDs são encontrados no mercado atualmente com essa estrutura básica, e são coletivamente chamados de CPLDs (*Complex PLDs*).

Os CPLDs foram introduzidos pela Altera Corporation, inicialmente com sua família de *chips* chamada Classic EPLDs (*Erasable PLDs*), e em seguida com a séries MAX. Devido ao rápido crescimento do mercado para grandes PLDs, outros fabricantes desenvolveram dispositivos na categoria CPLD e há atualmente diversas opções disponíveis. Os CPLDs provêm capacidade lógica de até 50 dispositivos SPLDs típicos, mas é difícil estender essa arquitetura para densidades maiores. Para construir PLDs com capacidade lógica muito alta, uma abordagem diferente é necessária.

A alta capacidade lógica em dispositivos de propósito geral começou a ser alcançada com os MPGAs. MPGAs consistem em um *arrays* de transistores pré-fabricados que podem ser customizados pelo usuário, através da definição das interligações e conexões a serem realizadas ainda na fase de manufatura do dispositivo, o que implica em alto custo de configuração e longo ciclo de manufatura. Assim como os MPGAs, os FPGAs incluem um arranjo de elementos de circuitos não conectados, chamados blocos lógicos, e recursos de interconexão, mas a configuração de FPGA é realizada através de programação pelo usuário final. Sendo o único tipo de PLD que suporta capacidade lógica elevada, os FPGAs têm sido responsáveis pela principal mudança no modo em que os circuitos digitais são projetados.

O primeiro FPGA disponível comercialmente foi desenvolvido pela Xilinx Inc., surgindo no mercado em 1984. Desde então, vários dispositivos têm sido desenvolvidos por diversos fabricantes e rapidamente têm se disseminado, o que pode-se atribuir ao reduzido tempo de projeto e ao relativo baixo custo destes dispositivos programáveis de alta capacidade.

Cada tipo de PLD apresenta vantagens que os tornam mais adequados para algumas aplicações do que outros. Um projetista hoje depara com a difícil tarefa de pesquisar os diferentes tipos de *chips*, entender qual sua melhor utilização, escolher um fabricante específico, aprender a utilizar as ferramentas EDA, para só então começar a projetar o *hardware*.

Neste capítulo estão apresentados os aspectos mais relevantes relativos a arquiteturas de FPGAs, detalhando suas principais características.

2.3. Arquitetura de FPGAs

Um FPGA consiste de um grande arranjo de células configuráveis (ou blocos lógicos) contidos em um único *chip*. Cada uma dessas células contém certa capacidade computacional para implementar funções lógicas, e/ou realizar roteamento para permitir a comunicação entre células. Essas operações podem acontecer simultaneamente no arranjo das células [26].

A arquitetura de um FPGA é bastante semelhante à arquitetura convencional de um MPGA. A principal diferença é que no MPGA as interconexões são feitas durante o processo de fabricação, como em circuitos integrados, enquanto os FPGAs são programados via comutadores programáveis, assim como os PLDs.

A arquitetura básica de um FPGA, ilustrada na figura 2.4, consiste de um arranjo 2-D de blocos lógicos. A comunicação entre blocos é feita através dos recursos de interconexão. A borda externa do arranjo consiste de blocos especiais capazes de realizar operações de entrada e saída (I/O).

Existem várias arquiteturas de FPGAs comercialmente disponíveis, por aproximadamente 10 fabricantes, como será apresentado no capítulo III. Três aspectos principais definem a arquitetura de um FPGA:

- tipo de tecnologia de programação;
- arquitetura das células; e
- estrutura de roteamento.

FIGURA 2.4: ESTRUTURA DE UM FPGA.

Estes aspectos influenciam diretamente o desempenho e a densidade das diferentes arquiteturas de FPGAs, entretanto não se pode afirmar que há uma melhor arquitetura, e sim a mais adequada para uma determinada aplicação.

As próximas seções descrevem com maiores detalhes as diferentes tecnologias de programação, complexidades dos blocos lógicos e arquiteturas de roteamento.

2.3.1. Tecnologia de Programação

Um FPGA é programado usando comutadores programáveis eletricamente. As propriedades desses comutadores, tais como tamanho, resistência, capacitância, tecnologia, afetam principalmente o desempenho e definem características como volatilidade e capacidade de reprogramação, que devem ser avaliadas na fase inicial do projeto para a escolha do dispositivo.

Em todos os tipos de FPGAs os comutadores programáveis ocupam uma grande área e apresentam valores de resistência e capacitância muito maior que as de um contato físico típico. Como consequência, o desempenho de um FPGA, se comparado a um MPGA de mesma tecnologia de fabricação, é menor.

Basicamente existem três tipos de tecnologias de programação:

- *Antifuse*: é originalmente um circuito aberto, que quando programado forma um caminho de baixa resistência;
- *Gate Flutuante*: o comutador é um transistor com *gate* flutuante; e
- *SRAM (Static Random Access Memory)*: o comutador é um transistor de passagem controlado pelo estado de um *bit* de SRAM.

2.3.1.1. Tecnologia Antifuse

O *antifuse* é um dispositivo de dois terminais, que no estado não programado apresenta uma alta impedância entre seus terminais. Se aplicarmos uma tensão entre 11 e 20 Volts o *antifuse* “queima”, criando uma conexão de baixa impedância, da ordem de 50 a 500 ohms. Entre as vantagens do *antifuse* está o seu tamanho reduzido, por consequência, sua baixa capacitância quando não programado e a baixa resistência quando programado, refletindo no desempenho, com maior velocidade quando comparado as outras tecnologias de programação, que ainda serão vistas neste capítulo. Porém, o *antifuse* não permite a reprogramação. Além disso, para programa-lo é necessário a presença de transistores com larga área a fim de suportar as altas correntes de programação. Finalmente, é necessário gastar um espaço extra para conseguir a isolamento dos circuitos de programação, pois os mesmos trabalham com tensão de até 20V. A figura 2.5 a seguir mostra a estrutura do *antifuse* PLICE (*Programmable Low Impedance Circuit Element*) da Actel [27, 50].

FIGURA 2.5: ANTIFUSE PLICE DA ACTEL: (A) SEÇÃO TRANSVERSAL; (B) CANAL ANTIFUSE FORMADO; E (C) CONTATOS DO ANTIFUSE.

O *antifuse* PLICE consiste de três camadas: embaixo uma camada n+ difusa;

em cima uma camada de polisilício; e entre estes eletrodos uma camada dielétrica, figura 2.5 (a). Com a ruptura do dielétrico forma-se um pequeno canal entre os eletrodos permitindo a condução de corrente elétrica, figura 2.5 (b). A camada dielétrica é composta por outras três camadas: oxido de silício (SiO_2); nitreto de silício (Si_3N_4); e novamente oxido de silício (SiO_2). Composição esta conhecida pela sigla ONO (Oxido-Nitreto-Oxido). O tamanho total do *antifuse* é praticamente o mesmo que o dos contatos que recebem as conexões, figura 2.5 (c).

A programação é realizada através de um pulso de até 20V através do *antifuse*, rompendo o dielétrico ONO e criando um caminho condutivo entre os eletrodos. Este caminho cresce à medida que aumenta a corrente de programação, permitindo obter baixas resistências.

Outros tipos de dispositivos *antifuse*, como alguns da QuickLogic, empregam silício amorfo entre os eletrodos condutores.

2.3.1.2. Tecnologia Gate Flutuante

Nesta tecnologia, os comutadores programáveis são baseados em transistores com *gate* flutuante iguais aos usados nas memórias EPROM (*Erasable PROM*) e EEPROM (*Electrical Erasable PROM*) [29]. A figura 2.6 ilustra um comutador programável do tipo EPROM.

FIGURA 2.6: COMUTADOR PROGRAMÁVEL BASEADO EM EPROM.

Um comutador EPROM consiste de um transistor que pode ser desabilitado,

isso é realizado através da injeção de cargas através de uma alta tensão entre o *gate* e *drain* do transistor. Esta carga aumenta a tensão de limiar de funcionamento do transistor, de modo que este não conduza mais. A carga é removida pela exposição do transistor ao ultravioleta, isto diminui a tensão de limiar e permite que o transistor volte a conduzir quando aplicada tensão ao seu *gate*.

A técnica para circuitos baseados em EEPROM é muito similar a EPROM, exceto por utilizar corrente elétrica, ao invés de ultravioleta, para remover as cargas que aumentam a tensão de limiar.

A maior vantagem da tecnologia EPROM é sua capacidade de reprogramação e a retenção dos dados. Além disso, com a EEPROM é possível programar e reter as informações com o *chip* já instalado na placa, característica denominada ISP (*In System Programability*).

Como desvantagens, a tecnologia EPROM exige três processos adicionais, além do processo normal de fabricação. Além disso, a resistência dos comutadores ligados não atinge valores baixos e o consumo total é maior devido aos resistores de *pull-down* ou *pull-up*.

Em relação a EEPROM, apesar de oferecer a reprogramação no sistema (ISP), cada célula ocupa o dobro de espaço de uma célula EPROM.

2.3.1.3. Tecnologia SRAM

A tecnologia de programação SRAM, ilustrada na figura 2.7, utiliza células de memória SRAM para controlar transistores de passagem ou multiplexadores.

A SRAM é uma memória volátil, portanto necessita ser configurada quando o *chip* é ligado. Isto implica que a FPGA baseada em SRAM necessita de uma memória externa do tipo PROM, EPROM, EEPROM ou disco magnético. Essa tecnologia tem a desvantagem de ocupar muito espaço no *chip*, pois para cada comutador estão associados pelo menos 6 transistores. No entanto, a SRAM apresenta duas grandes vantagens: é rapidamente reprogramável, comparada com a tecnologia EPROM ou EEPROM, requer apenas a tecnologia padrão de circuitos integrados para a sua fabricação [29].

FIGURA 2.7: TECNOLOGIA DE PROGRAMAÇÃO SRAM.

Na tabela 2.1 encontram-se resumidas as características: reconfigurabilidade; volatilidade; impedância das conexões; e processos extras de fabricação.

TABELA 2.1: SUMÁRIO DAS TECNOLOGIAS DE PROGRAMAÇÃO.

Tecnologia	Reconfigurabilidade	Volatilidade	Impedância (Ω)	Processos Extras
Antifuse	Não	Não	50-600	3
EPROM	Sim, fora da placa	Não	2-4k	3
EEPROM	Sim, na placa	Não	2-4k	>5
SRAM	Sim, na placa	Sim	0.5-2k	0

2.3.2. Arquitetura de Blocos Lógicos

Os blocos lógicos dos FPGAs variam muito de tamanho e capacidade de implementação lógica. A construção dos blocos lógicos pode ser tão simples como um transistor ou tão complexo como um microprocessador. Geralmente são capazes de implantar uma ampla variedade de funções lógicas combinacionais e sequenciais.

Os blocos lógicos dos FPGAs comerciais são baseados em um ou mais dos seguintes componentes:

- pares de transistores;
- portas básicas do tipo NAND ou XOR de duas entradas;
- multiplexadores;
- Look Up Tables (LUTs);
- estruturas AND e OR de múltiplas entradas; e
- *flip-flops* associados com diversos tipos de portas lógicas.

A fim de classificar os FPGAs quanto à capacidade lógica dos blocos lógicos, pode-se dividi-los em três categorias de granularidade: fina, média e grossa. A primeira categoria designa os blocos simples e pequenos, a segunda os blocos mais complexos e a terceira, sistemas completos em um único bloco lógico.

2.3.2.1. Blocos de Granularidade Fina

Nos dispositivos com granularidade fina há um grande número de blocos lógicos simples. Os blocos lógicos normalmente contêm uma função lógica de duas entradas ou um multiplexador 4 para 1 e um *flip-flop*. O melhor exemplo para um bloco de granularidade fina seria um bloco contendo alguns transistores interconectáveis ou portas lógicas básicas.

A principal vantagem no uso de blocos com granularidade fina é que estes são quase sempre totalmente utilizados, fornecendo um alto grau de funcionalidade com um número relativamente pequeno de transistores. A desvantagem reside no fato de serem em um número muito grande devido à baixa capacidade lógica, requerendo, portanto, uma grande quantidade de trilhas de conexões e comutadores programáveis o que sobrecarrega o roteador. Um roteador desse tipo de FPGA se torna lento e ocupa grande área no *chip*. Logo, a tecnologia *antifuse* é a mais adequada para a fabricação desse tipo de FGPA, devido ao tamanho reduzido dos comutadores *antifuse*. As famílias AT6000 da Atmel e XC6200 da Xilinx são exemplos de dispositivos de granularidade fina.

2.3.2.2. Blocos de Granularidade Média

Os blocos lógicos dos FPGAs que têm granularidade média frequentemente contêm duas ou mais LUTs e dois ou mais *flip-flops*. A maioria das arquiteturas de FPGAs implementa a lógica em LUTs de quatro entradas. As famílias APEX da Altera e Virtex da Xilinx são exemplos de dispositivos de granularidade média.

Uma LUT é uma pequena memória de um *bit* de largura, onde suas linhas de endereçamento funcionam como entradas do bloco lógico e suas saídas fornecem o valor da função lógica. A tabela verdade para uma função de k entradas é armazenada em uma SRAM de $2^k \times 1$. A tabela 2.2 é um exemplo de uma LUT de 4 linhas de endereçamento implementando o circuito de 4 entradas da figura 2.8. A vantagem das LUTs é que apresentam um alto grau de funcionalidade. Uma LUT de k entradas pode implementar qualquer função de k entradas e existem 2^{2^k} funções, assim no exemplo da tabela 2.2 temos $2^{2^4} = 65536$ (64k) funções. A desvantagem é que são inaceitavelmente grandes para mais de cinco entradas. Apesar do número de funções que podem ser implementadas aumentar muito rapidamente com o aumento do número de entrada, essas funções adicionais não são geralmente utilizadas em projetos lógicos e são difíceis de serem manipuladas por ferramentas de síntese lógica.

TABELA 2.2: LUT DE 4 LINHAS DE ENDEREÇAMENTO IMPLEMENTANDO O CIRCUITO DA FIGURA 2.8.

A (A3)	B (A2)	C (A1)	D (A0)	Z (D0)
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

FIGURA 2.8: CIRCUITO LÓGICO DE 4 ENTRADAS.

2.3.2.3. Blocos de Granularidade Grossa

Os dispositivos de granularidade grossa podem possuir blocos lógicos que sejam unidades lógicas aritméticas (ULAs) e/ou pequenos microprocessadores e/ou memórias. Como exemplos desse tipo de arquitetura podem ser citadas as máquinas RAW [30] e a arquitetura Garp [31].

2.3.2.4. Lógica Seqüencial

A maioria dos blocos lógicos apresentam alguma forma de lógica seqüencial. Geralmente utilizam *flip-flops* tipo D que podem ser conectados, via programação, às saídas dos blocos combinacionais. Em alguns dispositivos, a lógica seqüencial não está explicitamente presente, e deve ser formada utilizando-se o roteamento programável e os blocos puramente combinacionais.

2.3.3. Arquitetura de Roteamento

A arquitetura de roteamento de um FPGA é a maneira pela qual os comutadores programáveis e segmentos de trilhas são posicionados para permitir a interconexão dos blocos lógicos [29]. As arquiteturas de roteamento podem ser descritas a partir de um modelo geral, conforme a figura 2.9. São necessários alguns conceitos para um melhor entendimento desse modelo:

- pinos: são as entradas e saídas dos blocos lógicos, é válido ressaltar que estes pinos são internos aos FPGAs (conforme indicado na figura 2.9) e não

devem ser confundidos com os pinos externos do encapsulamento que são ligados aos blocos de I/O;

- conexão: é a ligação elétrica entre um pino e um segmento de trilha, as conexões são realizadas pelo bloco de conexão;
- bloco de conexão: é o dispositivo utilizado para conectar eletricamente um pino e um segmento de trilha, estes dispositivos utilizam alguma tecnologia de programação, como as apresentadas no item 2.3.1;

FIGURA 2.9: ARQUITETURA GERAL DE ROTEAMENTO DE UM FPGA.

- segmentos de trilha: são os fios entre dois blocos de comutação;
- trilhas: é uma seqüência de um ou mais segmentos de trilha em uma direção, estendendo-se por todo o comprimento de um canal de roteamento, pode ser composta de segmentos de tamanhos variados;
- bloco de comutação (*switch*): é o dispositivo utilizado para conectar eletricamente dois segmentos de trilha, estes dispositivos também utilizam alguma tecnologia de programação, como as apresentadas no item 2.3.1; e

- canal de roteamento: é a área entre duas linhas ou colunas de blocos lógicos, sendo que um canal é formado por várias trilhas paralelas.

O modelo da figura 2.9 contém duas estruturas básicas. A primeira é o bloco de conexões que aparece em todas as arquiteturas. O bloco de conexão permite a conectividade das entradas e saídas (pinos) de um bloco lógico com os segmentos de trilhas nos canais. A segunda estrutura é o bloco de comutação que permite a conexão entre os segmentos de trilhas horizontais e verticais. Em algumas arquiteturas, o bloco de comutação e o bloco de conexões são distintos, em outras estão combinados numa mesma estrutura. Nem todas as arquiteturas seguem esse modelo, pois há arquiteturas que apresentam uma estrutura hierárquica e outras que possuem somente canais horizontais.

Uma importante questão a ser considerada é se a arquitetura de roteamento permite que se alcance um roteamento completo e, ao mesmo tempo, uma alta densidade lógica. Sabe-se que usando um grande número de comutadores programáveis torna-se fácil alcançar um roteamento completo, mas esses comutadores consomem área, que é desejável minimizar. Algumas pesquisas estabelecem uma relação entre a flexibilidade da arquitetura de roteamento, a capacidade de roteamento e uso eficiente da área.

2.4. A Tecnologia SoC

A tecnologia SoC (*System on Chip*), consiste no desenvolvimento de sistemas completos em um único *chip*. Ou seja, em apenas um *chip* é embutido processador, memória, *timers*, interfaces etc. Dessa forma, todos os circuitos necessários para o funcionamento de um determinado sistema são implementados em um único dispositivo. Essa idéia tem sido amplamente adotada pela maioria dos fabricantes da área de *hardware*. Devido ao seu excelente desempenho e versatilidade, a tecnologia FPGA tem se mostrado perfeita para implementar os sistemas SoCs digitais.

2.4.1. FIPSoC

Um dispositivo SoC de destaque é o FIPSoC (*Field Programmable System-on-Chip*) da empresa SIDA [32]. Um FIPSoC possui uma arquitetura que vai além da

arquitetura de um FPGA, por possuir partes analógicas e digitais, integradas no mesmo *chip*, estes dispositivos têm por objetivo o desenvolvimento rápido de aplicações analógicas e digitais integradas. A figura 2.10 mostra que o FIPSoC possui um microcontrolador 8051 embutido, um FPGA, e um conjunto de células analógicas, otimizadas para aquisição de sinais e conversões A/D e D/A. A SIDA disponibiliza várias ferramentas de CAD integradas, que possibilitam ao usuário especificar, simular e mapear todo o projeto em apenas um circuito integrado.

FIGURA 2.10: ARQUITETURA INTERNA DE UM FIPSoC.

A lógica programável do FIPSoC é uma matriz de macro células digitais (DMCs - *Digital Macro Cells*) que podem ser configuradas para funções específicas. As DMCs são de granularidade média, baseadas em células de SRAM programáveis, que incluem LUTs de 4 entradas e 4 flip-flops para cada uma. O subsistema analógico consiste de blocos analógicos configuráveis (CABs - *Configurable Analog Blocks*) de granularidade grossa, onde normalmente a mínima unidade é um amplificador operacional. Os CABs permitem configurar diferentes funções analógicas, tais como amplificação diferencial e conversão de dados. Existe uma parte digital das células

analógicas que pode ser independentemente controlada pelo microprocessador ou pelo FPGA.

A configuração do circuito integrado é armazenada em *bits* de SRAM. Pode ser feito o *download* de uma nova configuração, para a memória de configuração, enquanto a célula está em operação. Não há a necessidade de parar o circuito integrado para reconfigurá-lo. Configurações externas podem ser carregadas em tempo de execução, de forma total ou parcial, mantendo parte do dispositivo ativo, caracterizando assim ser este um dispositivo capaz de suportar reconfiguração dinâmica. O conceito de reconfiguração dinâmica ainda será visto no capítulo IV.

Por exemplo, enquanto uma parte do FIPSoC realiza uma recuperação de contexto em uma determinada área, outra parte do dispositivo pode estar fazendo uma computação de propósito geral. Esta reconfiguração em tempo de execução, parcial ou total, pode também ser colocada em funcionamento pelo próprio *hardware* reconfigurável, sem a intervenção do microprocessador. A reconfiguração pode ser aplicada sobre uma única célula digital configurável, sobre um conjunto selecionado delas, ou sobre toda a lógica reconfigurável.

2.4.2. FPSLIC

O FPSLIC (*Field Programmable System Level Integrated Circuits*), produzido pela Atmel, é outro exemplo de dispositivo SoC. Este dispositivo integra em um único *chip*: o *core* de um processador AVR RISC de 8 *bits*, que atua como controlador, capaz de processar até 30 MIPS, figura 2.11 (a); um FPGA AT40K, figura 2.11 (b); um multiplicador em *hardware*, figura 2.11 (c); controladores de memória e de configurações; e 36kB de memória SRAM, figura 2.11 (d), tudo no mesmo *chip* [33]. A figura 2.11 ainda mostra detalhes do FPGA AT40K, como a presença de memória a cada setor de blocos lógicos, figura 2.11 (e) e os blocos lógicos com conexões em 8 lados, para implementar funções complexas, sem causar impacto nos recursos de barramento, figura 2.11 (f). No capítulo seguinte serão apresentados maiores detalhes da arquitetura interna do FPGA AT40K da Atmel.

FIGURA 2.11: ARQUITETURA INTERNA DE UM FPSLIC.

Esta arquitetura FPSLIC possibilita a prototipação rápida de sistemas complexos, com um baixo consumo de potência, e é passível de ser reconfigurada dinamicamente.

Devido a esta integração entre componentes de *hardware* e *software*, problemas de implementação podem ser detectados em estágios preliminares de projeto, diminuindo o impacto desses problemas na estimativa de tempo do projeto. Esta integração permite também a redução no consumo de potência, executando funções de algoritmos complexos em DSP (*Digital Signal Processing*) ou FPGAs, ao invés de usar componentes em *software*. Além disto, o microcontrolador pode ser usado para controlar o *Cache Logic*, auxiliando no processo de reconfiguração dinâmica do sistema. Reconfiguração dinâmica e *Cache Logic* ainda serão vistos no capítulo IV.

Arquiteturas Comerciais de FPGAs

O mercado oferece uma ampla gama de dispositivos FPGAs, de diversos fabricantes, em geral a estrutura de roteamento, os blocos lógicos e a forma de programação variam. Nota-se, no entanto, a existência de arquiteturas semelhantes que em geral podem ser classificadas em quatro classes, como ilustra a figura 3.1. Dependendo da aplicação, uma arquitetura pode ter características mais desejáveis para a aplicação em questão.

FIGURA 3.1: TIPOS DE ARQUITETURAS.

A tabela 3.1 apresenta os principais fabricantes de FPGAs e suas tecnologias de programação.

TABELA 3.1: FABRICANTES DE FPGAs X TECNOLOGIA DE PROGRAMAÇÃO.

Fabricantes	Tecnologia de Programação
Actel Corp.	<i>Antifuse</i> / EEPROM
Altera Corp.	SRAM / EEPROM
Atmel Corp.	SRAM
Cypress Semiconductor Corp.	EEPROM
Integrated Circuit Technology (ICT) Corp.	EEPROM
Lattice Semiconductor Corp.	SRAM / EEPROM
QuickLogic Corp.	<i>Antifuse</i>
Xilinx Corp.	SRAM / EEPROM

A seguir são apresentadas, de forma sucinta, algumas das mais famosas arquiteturas de FPGAs, assim como algumas das mais recentes famílias encontradas no mercado. Também são exemplificados alguns dos conceitos e tecnologias apresentados no capítulo II.

3.1. Os FPGAs da Xilinx

A Xilinx, fundada em 1984, se tornou a primeira fabricante de FPGAs do mercado, lançando em 1985 a família XC2000. A cada ano ela oferece novas gerações, introduzindo inovações tecnológicas que aumentam a capacidade lógica e a velocidade dos dispositivos, que já ultrapassam o patamar de 10 milhões de portas lógicas [34]. A seguir, são apresentadas algumas das famílias mais famosas e recentes da Xilinx: a XC4000; a família XC6200 que possui capacidade de reconfigurabilidade dinâmica; e as atuais famílias Virtex, Virtex II e Virtex II Pro.

3.1.1. A Família XC4000

A família XC4000, baseada na tecnologia de programação SRAM, apresenta uma arquitetura simétrica na distribuição de seus blocos lógicos entre os recursos de roteamento. O bloco lógico da Xilinx, chamado de CLB (*Configurable Logic Block*), é baseado em uma memória do tipo SRAM funcionando como LUT. O CLB da série XC4000 contém duas LUTs de quatro entradas que alimentam uma LUT de três entradas, como mostra a figura 3.2. Nesse bloco, todas as entradas são distintas e

disponíveis externamente ao bloco, possibilitando a implantação de muitas funções lógicas de até 9 entradas, duas funções separadas de 4 entradas ou outras possibilidades. Cada CLB também contém dois *flip-flops* nas saídas e recursos de roteamento interno a CLB baseados em multiplexadores [35].

FIGURA 3.2: BLÓCO LÓGICO DA FAMÍLIA XC4000.

Nesse bloco, são utilizadas LUTs de quatro e três entradas, dando ao bloco uma característica de heterogeneidade, que permite um melhor balanceamento entre desempenho e densidade lógica. Para a interligação das saídas das LUTs de quatro entradas às entradas da LUT de três entradas, há duas conexões não programáveis. Essas conexões são significativamente mais rápidas que qualquer conexão programável, podendo otimizar o desempenho, porém sua inflexibilidade pode reduzir a densidade lógica, quando a LUT de três entradas não for utilizada.

A família XC4000 da Xilinx incorpora diversas características, com o objetivo de prover dispositivos de alta capacidade que suportem a integração de um sistema inteiro. Os *chips* XC4000 têm características orientadas a sistemas, como por exemplo, circuitos que permitem a realização de operações aritméticas eficientemente, e também LUTs que podem ser reconfiguradas como células RAM, para a implantação de pequenas memórias. Também, cada *chip* XC4000 inclui um plano de portas AND de

várias entradas na periferia do arranjo de blocos lógicos, para facilitar a implementação de circuitos decodificadores. Os dispositivos dessa família variam em capacidade lógica podendo chegar a 500.000 portas lógicas com a família XC4000XV.

Além da lógica, outro aspecto importante que caracteriza um FPGA é a sua estrutura de roteamento. Os recursos de roteamento da série XC4000 são arranjos em canais verticais e horizontais. Cada canal contém uma quantidade de segmentos de trilha simples que se estendem por apenas um bloco de comutação (PSM - *Programmable Switch Matrix*), segmentos de trilha duplos que atravessam dois PSMs, como mostra a figura 3.3, e trilhas longas que atravessam a largura ou altura inteira do *chip*.

FIGURA 3.3: SEGMENTOS DE TRILHA SIMPLES E DUPLO DA SÉRIE XC4000.

A vantagem de segmentos maiores é que oferece um caminho de menor resistência em série ao sinal que passa. Além disso, utiliza menos comutadores e células programáveis, melhorando a densidade do FPGA. A figura 3.4 mostra detalhes internos de um PSM.

FIGURA 3.4: DETALHES INTERNOS DE UM BLOCO DE COMUTAÇÃO (PSM).

Um importante ponto a ser observado é que os sinais têm que passar por comutadores para alcançar um CLB, e o total de comutadores que são atravessados depende do conjunto particular de segmentos de trilha utilizados. Assim, o desempenho de um circuito implementado depende em parte de como os segmentos de trilha são alocados pelas ferramentas EDA.

3.1.2. A Família XC6200

A família XC6200 [36] possui dispositivos de granularidade fina, com tecnologia de programação SRAM e arquitetura baseada em mar de portas lógicas. Esta família foi projetada para operar em conjunto com microprocessadores ou microcontroladores, e para implementar funções normalmente efetuadas por coprocessadores ASIC. Uma de suas características mais interessante é a capacidade de reconfiguração dinâmica. Isso é possível devido a um controle de armazenamento SRAM altamente estável que permite a rápida reconfigurabilidade do dispositivo.

A arquitetura da família XC6200 é hierárquica e possui um grande arranjo de células simples, conhecido como mar de portas. Cada célula básica, figura 3.5, possui uma unidade lógica e alguns multiplexadores para a conexão com os segmentos de trilhas dos recursos de roteamento, ou ainda serem utilizados para a comutação entre segmentos de trilhas. Cada unidade lógica, figura 3.6, é composta por um registrador e alguns multiplexadores, podendo implementar algumas funções lógicas simples, como as possibilidades ilustradas na figura 3.7.

FIGURA 3.5: UMA CÉLULA BÁSICA DA FAMÍLIA XC6200.

FIGURA 3.6: UNIDADE LÓGICA DA FAMÍLIA XC6200.

As células básicas vizinhas são agrupadas em blocos com 4x4 células, figura 3.8, formando a primeira hierarquia. Esses blocos comunicam-se com seus vizinhos, formando blocos maiores de 16x16 células. No dispositivo XC6216, como ilustra a figura 3.9, um arranjo 4x4 desses blocos de 16x16 células formam um arranjo de 64x64 células, que possui um pino de I/O em cada célula da borda. Cada nível de hierarquia possui seus próprios recursos de roteamento. Cada célula pode se comunicar com suas vizinhas com segmentos de trilhas de comprimento 1, ou com células distintas

utilizando segmentos de trilha longos que atravessam 4, 16 ou 64 células. Esses segmentos longos são denominados FastLANEs.

FIGURA 3.7: EXEMPLOS DE FUNÇÕES LÓGICAS IMPLEMENTÁVEIS PELAS UNIDADES LÓGICAS.

FIGURA 3.8: BLOCO DE CÉLULAS 4x4.

FIGURA 3.9: DISPOSITIVO XC6216.

A flexibilidade dessa família dada a fina granularidade de sua arquitetura, associada à capacidade pioneira de reconfiguração dinâmica, fez dessa família um marco na área de computação reconfigurável. Porém, as aplicações idealizadas na concepção deste *hardware* se depararam com as limitações das ferramentas de desenvolvimento da época, fazendo com que tal família fosse descontinuada pela Xilinx que retirou toda a documentação sobre este dispositivo de seu *site*. Mesmo hoje não existem ferramentas de alto nível que possibilite explorar plenamente as características de reconfigurabilidade dinâmica destes e dos atuais dispositivos reconfiguráveis dinamicamente. Não é exagero afirmar que a tecnologia da família XC6200, quanto aos recursos de reconfigurabilidade dinâmica, estava à frente do seu tempo e que mesmo hoje este fato ainda é verdade.

3.1.3. A Família Virtex

Os elementos da arquitetura Virtex, baseada na tecnologia de programação SRAM, de granularidade média, são distribuídos de forma hierárquica conforme ilustrado na figura 3.10. Toda a periferia do dispositivo, assim como em qualquer outra família de qualquer fabricante, é composta por blocos de I/O, chamados pela Xilinx de

IOBs (*Input/Output Blocks*). Uma exceção quanto a presença de IOBs na periferia está nos vértices dos dispositivos Virtex, onde são encontrados blocos de *clocks* que permitem melhor compensação do atraso de *clock* distribuído, estes blocos são chamados de DLL (*Delay-Locked Loop*) [37].

FIGURA 3.10: ARQUITETURA DO FPGA VIRTEX.

No centro do dispositivo estão distribuídos os CLBs, figura 3.11, que por sua vez são compostos por dois *slices*, figura 3.12. Cada *slice* possui duas LUTs e dois *flip-flops* tipo D. Também existem alguns multiplexadores e portas lógicas básicas, que além de permitir o roteamento interno do *slice* também podem ser utilizados diretamente na implementação da funcionalidade desejada.

FIGURA 3.11: CLB DA FAMÍLIA VIRTEX COMPOSTO POR DOIS SLICES.

FIGURA 3.12: DETALHES DE UM SLICE DA FAMÍLIA VIRTEX.

FPGAs com CLBs baseados em LUTs podem utilizá-las como blocos simples de memória SRAM em vez de implementar funções lógicas. Como complemento, a arquitetura Virtex incorporou nas extremidades horizontais do conjunto de CLBs duas colunas de blocos de memória SRAM (BRAMs), conforme ilustra a figura 3.10. Outro recurso incorporado na arquitetura Virtex foi uma interface adicional de roteamento dos blocos de I/O chamada VersaRing, figura 3.10. Esta interface flexibiliza a alocação de pinos sem comprometer recursos de lógica do dispositivo.

3.1.4. A Família Virtex II

Os dispositivos da família Virtex II tiveram sua arquitetura otimizada para alta densidade e projetos lógicos de alto desempenho [39]. A figura 3.13 mostra a arquitetura deste dispositivo, que entre outros elementos, é composto de: IOBs, CLBs, DCMs, blocos de SelectRAM, *clock* global e multiplicadores.

FIGURA 3.13: ARQUITETURA DO FPGA VIRTEX II.

Os IOBs programáveis do dispositivo proporcionam interface entre os pinos físicos do encapsulamento e a lógica configurável interna, como em qualquer outro FPGA.

A lógica configurável interna possui quatro elementos principais organizados em um *array* regular:

- CLBs: proporcionam elementos funcionais, de granularidade média, para lógica sequencial e concorrente, assim como nas famílias XC4000 e Virtex, incluindo elementos básicos de armazenamento. Cada CLB deste é composto por 4 *slices* como os da família Virtex, figura 3.12;
- Blocos SelectRAM: são módulos de memória RAM dual que fornecem elementos de armazenamento de 18 *kbits*, distribuído em colunas de espaçamentos regulares pelo dispositivo e não apenas nas extremidades como na família Virtex;
- Blocos multiplicadores dedicados: fornecem o produto de 18-*bits* x 18-*bits*;
- e
- Blocos DCM (*Digital Clock Manager*): junto com o *Clock Global*, fornecem uma completa solução para sistemas de alta frequência e atrasos de *clocks* distribuídos.

Uma nova geração de recursos de roteamento chamado *Active Interconnect Technology* interconecta todos estes elementos. O GRM (*General Routing Matrix*) é um *array* de comutadores de roteamento. Cada elemento programável é ligado a uma *matrix* de comutadores, permitindo múltiplas conexões ao GRM. As interconexões programáveis globais são hierárquicas e projetadas para suportar altas frequências.

Todos elementos programáveis, incluindo os recursos de roteamento, são controlados por valores armazenados em células de memória estática (SRAM). Estes valores são carregados nas células de memória durante a configuração e podem ser recarregados para mudar as funções dos elementos programáveis. Permitindo assim, a reconfiguração parcial e/ou dinâmica, necessária para suportar a implementação de *hardware* virtual. Esta capacidade de reconfiguração dinâmica também já era suportada pela família Virtex.

Em 9 de abril de 2001 a Xilinx anunciou seu kit de desenvolvimento Microblaze Soft Processor, para a plataforma Virtex-II, como tentativa de competir com o sistema Excalibur da Altera, lançado mundialmente 12 de junho de 2000. O Microblaze tem como processador alvo o *hardware core* do PowerPC, desenvolvido pela IBM. O processador alvo da Altera até então era o *software core* Nios, sendo que ela também veio a disponibilizar o *hardware core* ARM, ambos serão comentado junto às referências de projeto SoC, dos dispositivo da Altera (item 3.2).

Em 4 de março de 2002 a Xilinx, junto com a IBM, anunciaram uma parceria para construção de *chips* híbridos utilizáveis em telecomunicações, armazenamento e aplicações de consumo generalizado. Esta parceria utiliza a tecnologia da Xilinx de construção de FPGAs e a tecnologia da IBM de construção de *chips* em cobre a 0.13 e 0.10 *mícron*, sendo a primeira vez que a IBM constrói componentes para um cliente utilizando tecnologia geralmente aplicada em microprocessadores, *chips* customizados e memórias [74]. Por norma, a IBM costuma desenvolver a tecnologia, que depois licenciava a outras empresas que depois se encarregam da construção dos produtos. O resultado desta parceria, por enquanto, foi o lançamento da família de dispositivos FPGAs Virtex II Pro da Xilinx, que entre outras elementos possui *hardware cores* do processador RISC PowerPC da IBM, conforme será apresentado a seguir.

3.1.5. A Família Virtex II Pro

Com esta família de FPGAs surge uma nova geração de dispositivos SoCs, baseados em FPGAs, já anunciada pela família Virtex II. Diferentemente dos dispositivos SoCs apresentados no capítulo II, como o FIPSoC da SIDA e o FPSLIC da Atmel, que possuíam FPGAs como blocos de seus sistemas, o família Virtex II Pro fornece vários elementos de granularidade grossa distribuídos pelo dispositivo, indo além dos elementos básicos comumente encontrados em arquiteturas FPGAs, figura 3.14. Porém, nem por isso estes dispositivos deixam de ser FPGAs, com os tradicionais blocos lógicos, chamadas de CLBs por este fabricante, caracterizando uma arquitetura híbrida de granularidade média com elementos de granularidade grossa. Entretanto, a Xilinx não se limita a chamar este dispositivo de FPGA, mas também de Plataforma FPGA.

FIGURA 3.14: ARQUITETURA DO FPGA VIRTEX II PRO.

A figura 3.14 ilustra alguns destes elementos da arquitetura Virtex II Pro [40]. Nesta arquitetura, além dos elementos comuns à família Virtex II, como os CLBs, IOBs, blocos SelectRAM, Multiplicadores e DCMs, existem outros elementos como:

- Rocket I/O: é um *hardware core* que implementa um conversor paralelo-serial e serial-paralelo de alto desempenho, posicionados estrategicamente na periferia do dispositivo, junto aos IOBs, facilitando assim a comunicação do dispositivo com o meio externo; e
- Processador PowerPC: é o *hardware core* do processador RISC PowerPC 405 da IBM. Este elemento é distribuído como um bloco entre a lógica configurável dos CLBs, multiplicadores e SeletRAMs. O dispositivo Virtex II Pro de maior capacidade atualmente fabricado pela Xilinx (XC2VP125) possui quatro destes blocos.

A tabela 3.2 apresenta uma comparação entre os elementos de alguns dispositivos das famílias XC4000, XC6200, Virtex, Virtex II e Virtex II Pro da Xilinx. Esta comparação torna-se complicada, pois não existe um padrão certo para a contagem de cada elemento. Por exemplo, embora os *slices* são elementos que vêm mantendo um padrão desde a família Virtex, os CLBs da Virtex II e Virtex II Pro tem o dobro de *slices*.

TABELA 3.2: COMPARAÇÃO DE DISPOSITIVOS XILINX.

Família	Dispositivo	CLBs	Slices	Cel. Lógicas	IOBs (Max.)	Multip.	DCMs	µP
XC4000	XC4085XL	3136	-	6272	448	0	0	0
XC6200	XC6464	-	-	16.384	512	0	0	0
Virtex	XCV1000	6.144	12.288	24.576	512	0	0	0
Virtex II	XC2V8000	11.648	46.592	93.184	1.108	168	12	0
Virtex II Pro	XC2VP125	13.904	55.616	111.232	1.200	556	12	4

A contagem de células lógicas foi definida como sendo a mínima unidade lógica que sempre é finalizada com a presença de um *flip-flop*. Assim, a contagem de células lógicas da tabela 3.2 coincide com a de *flip-flops*. Logo, com este critério um *slice*, figura 3.12, conta duas células lógicas. Porém, mesmo assim esta contagem não mantém um bom padrão da capacidade lógica dos elementos, pois basta observar que uma célula lógica da família XC6200, figura 3.6, está tendo o mesmo peso de meio *slice*. Porém, meio *slice* possui uma LUT, o que não é o caso da célula lógica da família XC6200.

3.2. Os FPGAs da Altera

Enquanto a Xilinx foi a primeira fabricante de FPGAs, com a família XC2000, lançada em 1985, a Altera inovou em 2000, fornecendo uma completa solução SoC para seus FPGAs da família APEX, apresentados a seguir. Antes disso, é apresentado a família FLEX 8000 e a FLEX 10K, que são uma das mais famosas famílias de dispositivos da Altera. Ainda neste tópico é apresentado a mais recente família de FPGAs da Altera, a Stratix.

A Altera atribui o nome SOPC (*System-on-a-Programmable-Chip*) à sua tecnologia SoC.

3.2.1. A Família FLEX 8000

A família FLEX 8000 da Altera apresenta uma arquitetura hierarquia de três níveis muito parecida às encontradas em CPLDs. Entretanto, o nível mais baixo da hierarquia consiste de um conjunto de LUTs, ao invés de blocos como os SPLDs, portanto a família FLEX 8000 pertence à categoria dos FPGAs. Deve ser notado, entretanto, que a FLEX 8000 é baseada na tecnologia de programação SRAM e possui uma LUT de quatro entradas como seu bloco lógico básico. Sua capacidade lógica está na faixa de aproximadamente 2.500 a 1.6000 portas lógicas [41]. A arquitetura do FLEX 8000 é ilustrada na figura 3.15. Os blocos de I/O são nomeados pela Altera como IOEs (*I/O Elements*). O bloco lógico básico, chamado LE (*Logic Element*) contém uma LUT de quatro entradas, um *flip-flop*, e elementos básicos como portas básicas e multiplexadores, que além de permitir o roteamento interno ao LE, auxiliam em algumas implementações, como circuitos especiais aritméticos. Detalhes do LE estão ilustrados na figura 3.16.

Os FastTracks são similares às FastLANEs da Xilinx. Cada trilha da FastTrack estende-se pela altura ou largura completa do dispositivo. Entretanto, a principal diferença entre a FLEX 8000 e os *chips* da Xilinx é que a FastTrack consiste somente de trilhas longas. Isso facilita a configuração automática feita pelas ferramentas EDA. Todas as FastTrack horizontais são idênticas. Portanto, os atrasos de conexões na família FLEX 8000 são mais previsíveis do que em outros FPGAs que empregam

muitos segmentos de comprimentos menores, pois há menos comutadores programáveis em caminhos longos. Porém, pode prejudicar o roteamento, diminuindo a densidade lógica do dispositivo.

FIGURA 3.15: ARQUITETURA FLEX 8000 DA ALTERA.

FIGURA 3.16: DETALHES DE UM LE DA FAMÍLIA FLEX 8000.

3.2.2. A Família FLEX 10K

A arquitetura da família FLEX 8000 foi estendida na família FLEX 10K. A FLEX 10K oferece todas as características da FLEX 8000, com a adição de blocos de SRAM de tamanhos variados, chamados EABs (*Embedded Array Blocks*) [42]. Essa característica é ilustrada na figura 3.17, que mostra os EABs entre as linhas e colunas de interconexão do *chip*.

Cada EAB é configurável para servir como um bloco SRAM com tamanhos variáveis entre: 256x8, 512x4, 1kx2 e 2kx1. Além disso, um EAB pode alternadamente ser configurado para implementar um circuito lógico complexo, como um multiplicador, empregando-o como uma grande LUT de múltiplas saídas. A Altera provê, como parte de suas ferramentas EDA, várias macros que implementam circuitos lógicos úteis em EABs.

FIGURA 3.17: ARQUITETURA DA FAMÍLIA FLEX 10000 DA ALTERA.

3.2.3. A Família APEX 20K

Muitos projetistas de sistemas usam atualmente múltiplos dispositivos FPGAs na mesma placa, como o FLEX 10K, FLEX 6000 e MAX 7000, porque cada um destes possui uma arquitetura ideal para cada tipo de aplicação. Combinando os benefícios destas três famílias em uma única arquitetura, baseada na tecnologia de programação SRAM, a família APEX permite que projetistas integrem um complexo sistema em um único dispositivo, eliminando a necessidade de múltiplos módulos e *chips*. Deste modo é reduzido o tamanho das placas de circuito impresso, simplificando a implementação de projetos complexos. Esta inovação arquitetural é chamada de arquitetura APEX MultiCore, figura 3.18.

FIGURA 3.18: ARQUITETURA MULTICORE DA FAMÍLIA APEX 20K.

A arquitetura MultiCore é uma combinação de três diferentes tipos de estruturas: LUTs como as encontradas nas famílias FLEX 10K, FLEX 8000 e XC4000; blocos lógicos baseados em termos de produtos como os encontrados nos dispositivos MAX 7000, porém com tecnologia de programação SRAM; e blocos internos de memória como encontrado nos dispositivos FLEX 10K e na família Virtex da Xilinx. Juntos, estas estruturas permitem o desenvolvimento de complexas funções, como as *megafunctions*, em um processo simples e eficiente [43].

Os ESBs (*Embedded System Blocks*) constituem-se no principal elemento da arquitetura MultiCore. Eles podem implementar vários tipos de blocos de memória,

incluindo *dual-port* RAM, ROM, FIFO e blocos CAM (*Content-Addressable Memory*). Os ESBs implementam lógica em termos de produtos usada em funções combinatórias.

A arquitetura MultiCore é composta de LABs (*Logic Array Blocks*), figura 3.19, cada um consistindo de 10 LEs, como os da família FLEX 8000, combinados em uma nova estrutura hierárquica chamada de estrutura MegaLAB, figura 3.20, que é uma “LAB de LABs”. Cada estrutura MegaLAB contém 16 LABs e uma estrutura interna avançada chamada de ESB (*Embedded System Block*).

FIGURA 3.19: ESTRUTURA LAB.

FIGURA 3.20: ESTRUTURA MEGALAB.

Com o ambiente de desenvolvimento Quartus e o kit de desenvolvimento Excalibur [44], a Altera inclui referência de projetos que são uma completa configuração SoC para essa arquitetura MultCore, incluindo o processador *software core* Nios, alguns periféricos e um barramento *on-chip* (PBM - *Peripheral Bus Module*), figura 3.21. Os periféricos incluídos no projeto de referência são:

- UART;
- timer;
- interface para memória externa (SRAM, FLASH);
- interface para memória interna (ESB); e
- PIO (*Parallel I/O*).

FIGURA 3.21: DIAGRAMA DE BLOCOS DO PBM, NIOS E PERIFÉRICOS.

O processador Nios se comunica com os dispositivos periféricos através do PBM. Este componente é gerado automaticamente pelo *software SOC builder*, baseado nos periféricos requisitados pelo usuário. Isto inclui:

- interfaces para os periféricos;

- codificação de endereço;
- multiplexação de dados;
- controle de interrupções; e etc.

Além do *software core* Nios a Altera, em parceria com a ARM, passou a fornecer o *hardware core* do processador RISC ARM922T. Este *hardware core* deu origem a uma nova família de dispositivos Altera, chamada Excalibur [75]. Os dispositivos Excalibur são baseados na arquitetura APEX 20K, apresentando como principal diferencial o *hardware core* ARM, embutido entre seus elementos. Assim como para o Nios, também foi fornecida uma série de elementos de suporte ao ARM922T, para o desenvolvimento de aplicações no ambiente Quartus. Entre estes elementos fornecidos está o barramento AMBA [64].

3.2.4. A Família APEX II

A evolução dos dispositivos da família APEX veio com a introdução da família APEX II da Altera. Com a mesma arquitetura da família APEX, os dispositivos APEX II [45] apresentam, basicamente, apenas um aumento na quantidade de elementos lógicos, o que provê maior capacidade lógica a esta família. Além disso, foram introduzidos alguns novos recursos, principalmente na estrutura de I/O e na estrutura de *clocks* distribuídos com a introdução de PLLs (*Phase-Locked Loops*).

3.2.5. A Família Stratix

Com a família Stratix [46] a Altera introduz no mercado de FPGAs dispositivos com evolução arquitetural mais significativa diante da família APEX 20K. Além da evolução na arquitetura, também foram adicionados nestes dispositivos elementos de granularidade grossa distribuídos pelo FPGA, semelhante ao que foi realizado pela Xilinx nos dispositivos Virtex II.

As famílias Stratix e APEX II foram projetadas para serem evoluções da

família APEX 20K. Assim, todo o sistema SOPC (SoC) suportado pelos dispositivos APEX 20K, também são suportados por estas famílias.

A arquitetura hierárquica da família Stratix, figura 3.22, apresenta blocos de memória de tamanhos variados distribuídos por todo o dispositivo. Nesta arquitetura existem colunas de blocos chamados M512 RAM, que são blocos RAM de $32 \times 18 \text{ bits}$, colunas de blocos chamados M4k RAM ($128 \times 36 \text{ bits}$), e blocos chamados MegaRAM ($4k \times 144 \text{ bits}$). Esta distribuição de blocos de memória de tamanhos variados flexibiliza muito este dispositivo, pois além do uso tradicional destes blocos como memória eles também podem ser utilizados como LUTs de tamanhos distintos, implementando funções de várias variáveis de entrada e saída. Como estes blocos são de diferentes tamanhos, fica mais fácil manter a densidade lógica. O dispositivo atualmente de maior capacidade da família Stratix (EP1S120) possui o total de 10.118.016 *bits* de RAM.

FIGURA 3.22: ARQUITETURA DA FAMÍLIA STRATIX.

Além dos blocos de memória esta arquitetura também apresenta blocos DSPs distribuídos pelo FPGA, tendo o dispositivo EP1S120 28 destes blocos DSPs. Estes blocos podem ser utilizados, entre outras coisas, para implementar multiplicadores e filtros digitais.

Outra mudança nesta arquitetura fica por conta das trilhas (linhas e colunas) de interconexão das LABs, figura 3.23, que também foram divididas em diferentes comprimentos e velocidades, com a finalidade de flexibilizar a arquitetura garantindo assim a densidade lógica.

Os LEs embora também tiveram algumas modificações, principalmente para permitir o roteamento com estas diferentes trilhas, eles mantiveram basicamente a mesma estrutura dos LEs da família FLEX 8000, figura 3.16. Estes LEs são compostos por uma LUT de quatro entradas, um *flip-flop* tipo D e algumas portas lógicas básicas e multiplexadores que, além de alguns controles e do roteamento, também auxiliam na implementação da lógica desejada.

A presença de blocos de DSPs e memória nos dispositivos Stratix caracteriza uma arquitetura híbrida, pois a capacidade lógica dos LEs, que estão distribuídos por todo o dispositivo agrupados em LABs, é típica de uma arquitetura de granularidade média, porém os DSPs são blocos de granularidade grossa.

FIGURA 3.23: ESTRUTURA LBA DA FAMÍLIA STRATIX.

A tabela 3.3 apresenta uma comparação entre os elementos de alguns dispositivos das famílias FLEX 8000, FLEX 10K, APEX 20K, APEX II e Stratix.

TABELA 3.3: COMPARAÇÃO DE DISPOSITIVOS ALTERA.

Família	Dispositivo	LABs	LEs	IOBs	RAM <i>bits</i>	PLLs	DSPs
FEX 8000	EPF81500A	162	1.296	208	0	0	0
FLEX 10K	EPF10K250A	1.520	12.160	470	40.960	0	0
APEX 20K	EP20K1500E	5.184	51.840	808	442.368	0	0
APEX II	EP2A70	6.720	67.200	1.060	1.146.880	4	0
Stratix	EP1S120	11.414	114.140	1.310	10.118.016	12	28

Entre estas famílias é possível realizar uma comparação de capacidade lógica, pois os LEs mantiveram basicamente a mesma estrutura desde a família FLEX 8000. Quanto aos LABs, cabe ressaltar que até a família FLEX 10K eles eram compostos por 8 LEs, sendo ampliados a quantidade para 10 LEs por LAB a partir da família APEX 20K. A análise do aumento de RAM *bits* é direto e os blocos de PLLs e DSPs refletem a tendência do mercado de fornecer blocos de granularidade grossa.

3.3. Os FPGAs da Atmel

A seguir são apresentadas as famílias AT6000 e AT40K que são caracterizadas pela estrutura simétrica.

3.3.1. A Família AT6000

A família AT6000 da Atmel é baseada na tecnologia de programação SRAM. A arquitetura desta família emprega *array* simétrico de pequenas células idênticas, figura 3.24, conectadas a uma estrutura de roteamento flexível de granularidade fina. Os *clocks* são independentemente controlados com *resets* por colunas de células. O *array* é cercado por blocos de I/O programáveis em sua periferia [47].

O *array* é contínuo de uma extremidade à outra, exceto pelo espaço dos repetidores de trilhas, como pode ser notado no setor ilustrado na figura 3.25, onde são

mostrados detalhes dos canais de roteamento deste setor. Além da funcionalidade lógica, células também podem ser usadas também como segmentos de trilhas para conexões através de curtas distâncias, podendo ser muito útil em pequenos espaços.

FIGURA 3.24: ARRAY SIMÉTRICO CERCADO POR BLOCOS DE I/O DO AT6000.

FIGURA 3.25: CANAIS DE ROTEAMENTO DE UM SETOR DO AT6000.

Existem dois tipos de trilhas, local e *express*, como mostra a figuras 3.26. As trilhas locais fazem ligações entre células, e as trilhas *express* são as não conectadas diretamente as células, proporcionando assim alta velocidade. A utilização das trilhas *express* é o modo mais rápido de cobrir longas distâncias dentro do *array*.

FIGURA 3.26: TRILHAS.LOCAIS E EXPRESS DO AT6000.

A estrutura da célula (bloco lógico) Atmel, figura 3.27, é simples e pequena, podendo ser utilizada para qualquer função lógica necessária para implementar um circuito digital. Seus quatro lados são funcionalmente idênticos, assim cada célula é completamente simétrica.

FIGURA 3.27: ESTRUTURA DA CÉLULA DO AT6000.

3.3.2. A Família AT40K

A família AT40K da Atmel, comparada a família AT6000, apresenta significativa evolução em sua arquitetura. Também baseada na tecnologia SRAM, manteve-se o arranjo simétrico da família anterior.

Os repetidores de linhas e colunas (trilhas horizontais e verticais) agora são blocos RAM de 32x4 acessíveis por barramentos adjacentes. A RAM pode ser configurada como uma memória de acesso simples ou dual, com operação síncrona ou assíncrona [48].

A figura 3.28 apresenta um dos cinco planos do barramento. Cada plano possui três trilhas, uma local (trilha central) e duas trilhas *express*. Recursos de comutação são conectados via repetidores, podendo cada repetidor conectar-se a dois segmentos de trilhas locais adjacentes e dois segmentos de trilhas *express*. Cada segmento de trilha local percorre quatro células e conecta repetidores consecutivos. Cada segmento de trilha *express* percorre oito células e atravessa um repetidor.

FIGURA 3.28: ESTRUTURA DE UM DOS CINCO PLANOS DO BARRAMENTO DO AT40K.

Outra novidade desta arquitetura é o fato de cada célula possuir recursos de roteamento com suas 8 células vizinha, quatro transversais e quatro diagonais, figura 3.29. A figura 3.30 mostra detalhes da conexão entre uma célula com cinco segmentos de trilhas horizontais locais e cinco segmentos de trilhas verticais locais (um por plano).

FIGURA 3.29: CONEXÕES DE UMA CÉLULA COM SUAS CÉLULAS VIZINHAS.

FIGURA 3.30: CONEXÕES DE UMA CÉLULA COM OS SEGMENTOS DE TRILHAS LOCAIS.

A figura 3.31 apresenta a estrutura de uma célula (bloco lógico) do AT40K. Esta estrutura possui duas LUTs de oito entradas e um *flip-flop* tipo D, além de alguns multiplexadores para o roteamento interno e com os cinco planos externos. Esta estrutura é bem simples embora a capacidade lógica fornecida pelas LUTs caracterize uma arquitetura de granularidade média.

FIGURA 3.31: ESTRUTURA DA CÉLULA DO AT40K.

3.4. Os FPGAs da Actel

Ao contrário dos FPGAs descritos anteriormente, as famílias de dispositivos FPGAs ACT 1, ACT 2 e ACT 3 da Actel são baseados na tecnologia de programação *antifuse*. Embora estas famílias não sejam muito recentes, será descrita a mais recente delas por ser um bom exemplo de arquitetura baseada em linha e com tecnologia *antifuse*, ambas características não encontradas nos dispositivos descritos anteriormente. Atualmente, a mais recente família de FPGAs da Actel é a Axcelerator que também é baseada na tecnologia de programação *antifuse*.

3.4.1. A Família ACT 3

Nos dispositivos da família ACT 3 [27] os blocos lógicos são distribuídos em linhas e os canais de roteamento são horizontais conforme ilustra a figura 3.32.

FIGURA 3.32: ARQUITETURA DOS FPGAS DA ACTEL.

Os blocos lógicos dos dispositivos ACT 3 são baseados em multiplexadores, sendo relativamente menores, comparados aos baseados em LUTs. Estes blocos lógicos são compostos de portas AND e OR conectadas a um bloco de circuito com multiplexadores. O circuito com multiplexadores é arranjado de tal forma que, em combinação com as duas portas lógicas, uma grande variedade de funções lógicas podem ser realizadas com um único bloco lógico. Aproximadamente metade dos blocos lógicos também possuem *flip-flops*.

Os recursos de roteamento estão organizados em canais de roteamento horizontais. Os canais consistem de segmentos de trilhas de vários comprimentos com comutadores *antifuses* que conectam os blocos lógicos aos segmentos de trilha ou um segmento ao outro. Estes dispositivos, como mostra a figura 3.33, têm trilhas verticais para a conexão dos blocos lógicos aos canais de roteamento. A arquitetura de roteamento da família ACT 3 é assimétrica porque há mais trilhas na horizontais do que verticais.

Nesta arquitetura não há um bloco de comutação claramente definida. A comutação é distribuída por todos os canais horizontais. Todas as trilhas verticais podem ser conectadas a todas as trilhas horizontais incidentes. Essa arquitetura fornece maior conectividade que a apresentada por blocos de comutação tradicionais. Essa conectividade permite que o roteamento dos canais horizontais seja feito independentemente, pois para uma dada rota que atravessa dois canais, a escolha de uma trilha em um canal não limita o número de opções de trilhas em um outro canal,

como acontece em outras arquiteturas. Esta conectividade simplifica o problema de roteamento, mas em compensação aumenta o número de comutadores necessários.

FIGURA 3.33: ARQUITETURA DE ROTEAMENTO DA FAMÍLIA ACT 3.

3.5. Os FPGAs da Lattice

A Lattice Semiconductor, que até o ano passado não produzia FPGAs com tecnologia de programação SRAM, agora produz. Não por ter desenvolvido uma nova linha de FPGAs Lattice com esta tecnologia, mas sim por ter adquirido em 18 de janeiro deste ano (2002) a divisão de FPGAs da Lucent Technologies, tornando-se assim a atual fabricante das já conhecidas famílias de FPGAs ORCA. A seguir é apresentado a família de FPGAs ORCA Series 4 [49].

3.5.1. A Família ORCA Series 4

A arquitetura hierárquica da família ORCA Serie 4 é a mais nova geração de FPGAs da Lattice baseados na tecnologia SRAM. Esta arquitetura de granularidade

média consiste de 4 elementos básicos: blocos lógicos baseados em LUTs, chamados pela Lattice de PLCs (*Programmable Logic Cells*); blocos de memória RAM (EBRs – *Embedded Block RAMs*); blocos de I/O, chamados de Lattice de PIOs (*Programmable Input/Output cells*); e características de sistema.

A figura 3.34 ilustra a arquitetura ORCA Series 4 com suas PLCs distribuídas em linhas e colunas. Na periferia encontram-se os PIOs, sendo que apenas nas extremidades superior e inferior existem EBRs interconectadas por um barramento específico. Este barramento também é conectado à módulos de interface para processadores e FPGAs externos, e nos vértices são encontrados PLLs semelhante aos das famílias APEX II e Stratix da Altera.

FIGURA 3.34: ARQUITETURA ORCA SERIES 4.

Os PLCs são compostos por dois elementos: as PFUs (*Programmable Function Units*), que são os elementos básicos onde é implementado a lógica desejada; e os SLICs (*Supplemental Logic and Interconnect Cells*), que auxiliam no roteamento dos PLCs, ao canais de roteamento do dispositivo. A figura 3.35 mostra detalhes de uma PFU da arquitetura ORCA Series 4.

FIGURA 3.35: DETALHES DE UMA PFU DA FAMÍLIA ORCA SERIES 4.

Como pode ser notado na figura 3.35, as PFUs desta família são baseadas em LUTs o que caracteriza uma arquitetura de granularidade média. Cabe observar que nenhuma arquitetura anteriormente apresenta tantas LUTs (total de 8) em seu elemento lógico básico como esta. Embora este fato mostre que estes dispositivos apresentam granularidade mais elevada que os FPGAs de granularidade média vistos anteriormente, esta família não pode ser enquadrada na classe de dispositivos de granularidade grossa.

3.6. Comparações

Na tabela 3.4 são realizadas algumas comparações entre diversos dispositivos, com tecnologia de programação SRAM, de diferentes fabricantes.

TABELA 3.4: COMPARAÇÃO ENTRE DISPOSITIVOS.

Fabricante	Família	Dispositivo	LUTs	Flip-flops	IOBs *	RAM bits *	U\$ **
Xilinx	XC4000	XC4002XL	128	256	64	2.048	12 / 21
Xilinx	XC4000	XC4085XL	9.408	7.168	448	100.352	661 / 2.459
Xilinx	XC6200	XC6464	0	16.384	512	0	-
Xilinx	Virtex	XCV50	1.536	1.536	180	32.768	33 / 106
Xilinx	Virtex	XCV1000	24.576	24.576	512	131.072	1.032 / 2.972
Xilinx	Virtex II	XC2V40	512	512	88	8.192	27 / 76
Xilinx	Virtex II	XC2V6000	67.584	67.584	1.104	1.081.344	3.470 / 8.363
Xilinx	Virtex II	XC2V8000	93.184	93.184	1.108	3.096.576	-
Xilinx	Virtex II Pro	XC2VP125	111.232	111.232	1.200	10.248.192	-
Altera	FEX 8000	EPF8282A	208	282	78	0	11 / 33
Altera	FEX 8000	EPF81500A	1.296	1.500	208	0	79 / 1.194
Altera	FLEX 10K	EPF10K30A	1.728	1.728	246	12.288	64 / 252
Altera	FLEX 10K	EPF10K250A	12.160	12.160	470	40.960	1.220 / 3.880
Altera	APEX 20K	EP20K30E	1.200	1.200	128	24.576	32 / 74
Altera	APEX 20K	EP20K1500E	51.840	51.840	808	442.368	1.325 / 4.030
Altera	APEX II	EP2A15	16.640	16.640	492	425.984	600 / 1.210
Altera	APEX II	EP2A70	67.200	67.200	1.060	1.146.880	4.600 / 9.410
Altera	Stratix	EP1S120	114.140	114.140	1.310	10.118.016	-
Atmel	AT6000	AT6002	0	1.204	96	0	21 / 32
Atmel	AT6000	AT6010	0	6.400	204	0	116 / 198
Atmel	AT40K	AT40K05	512	256	128	2.048	9 / 26
Atmel	AT40K	AT40K40	4.608	2.304	384	18.432	50 / 299
Lattice	ORCA 4	OR4E06	16.192	18.216	720	150.528	-

* Máxima quantidade possível; ** Fonte: Avnet Electronics Marketing e Arrow Electronics (09/06/2002), os valores indicam respectivamente o menor e o maior preço encontrado por dispositivo.

Como pode ser notado na tabela 3.4, as comparações se baseiam em quantidade de LUTs, *flip-flops*, IOBs e RAM *bits*. O fato de realizar a contagem de LUTs e *flip-flops* para tentar estabelecer uma relação veio da impossibilidade de comparar-se blocos lógicos, pois como foi visto neste capítulo existem diferentes arquiteturas o que tornaria inútil esta comparação. A comparação por portas lógicas típicas mostrou-se também inviável dado às diferenças arquiteturais e granulares dos elementos que constituem um FPGA. Nas recentes famílias Stratix da Altera e Virtex II Pro da Xilinx os fabricantes não fazem mais menção à quantidade de portas lógicas típicas em seus dispositivos. Agora os fabricantes dão destaque a quantidade presente de *bits* de RAM e elementos básicos constituídos por uma ou mais LUTs (LEs, *slices*, etc.).

A quantidade de LUTs e *flip-flops* não é explicitamente divulgada por todos os fabricantes. Porém, observando as arquiteturas de cada dispositivo é possível chegar a estes números em função de outros dados fornecidos pelos fabricantes, como a quantidade de blocos lógicos, CLB, ELs, células lógicas, *slices*, e etc. Eventualmente, esta contagem pode ter alguma variação em função de outros elementos do dispositivo. Outro agravante é o fato de que nem todas arquiteturas utilizam apenas LUTs de 4 entradas. Por exemplo, a família XC4000 possui uma LUT de 3 entradas e duas de 4 por CLB, e a família AT40K é baseada em LUTs de 8 entradas. Também não foram considerados elementos de granularidade grossa como: microprocessadores; DSPs; multiplicadores; e etc. Porém, é importante observar que estes elementos de granularidade grossa tem relevante papel na arquitetura, dada a grande capacidade de implementação lógica dos mesmos.

Os preços apresentados na tabela 3.4 são, respectivamente, a menor e a maior cotação obtida por dispositivo. Foram orçados os dispositivos com menor e maior capacidade lógica das famílias XC4000, Virtex, Virtex II, Virtex II Pro, FLEX 8000, FLEX 10K, APEX 20K APEX II, AT6000 e AT40K. A família XC6200 foi descontinuada pelo fabricante e por este motivo não se encontra à venda. As famílias Stratix e Virtex II Pro não possuem seus preços divulgados de forma *on-line* e a família ORCA Serie 4 não é distribuída pelos fornecedores consultados (Avnet e Arrow).

Reconfigurabilidade de FPGAs

Os dispositivos FPGAs podem ser classificados de acordo com suas configurabilidades como mostra a figura 4.1 [1]. Todos dispositivos FPGAs são por definição programáveis, ou seja, configurável pelo menos uma única vez, como qualquer PLD. Um pequeno subconjunto destes dispositivos pode ser reconfigurado várias vezes por alguma operação que recarrega completamente a configuração do dispositivo. Note que dispositivos que utilizam a tecnologia *antifuse* não são reconfiguráveis.

FIGURA 4.1: CLASSIFICAÇÃO DE FPGAs DE ACORDO COM SUA CONFIGURABILIDADE.

4.1. Reconfiguração Parcial

Um dispositivo é definido como reconfigurável parcialmente se é possível reconfigurá-lo seletivamente, enquanto o resto do dispositivo permanece inativo, mas retém sua configuração. Caso o dispositivo só possa ser reconfigurado por completo, mesmo que na própria placa do sistema, ou fora dela como com dispositivos EPROMs,

ele é considerado apenas reconfigurável, pois obviamente ele não permanece ativo em nenhum dos casos, por não permitir configuração seletiva.

4.2. Reconfiguração Dinâmica

FPGAs são classificados como dinamicamente reconfiguráveis se seus circuitos internos de armazenamento da configuração podem ser atualizados seletivamente, sem prejudicar o funcionamento da lógica restante que pode estar em operação. Estes dispositivos podem assim ser reconfigurados seletivamente enquanto estiverem ativos. Outros termos atualmente encontrados na literatura que se referem a reconfiguração dinâmica são, *run-time reconfiguration* (RTR), *real-time reconfiguration*, *on-the-fly reconfiguration* e *in-circuit reconfiguration*. Todas essas expressões podem ser traduzidas também como reconfiguração em tempo de execução.

Exemplos destes dispositivos dinamicamente reconfiguráveis são os da família XC6200 e Virtex da Xilinx e AT6000 e AT40K da Atmel.

Estes FPGAs possuem características comuns: eles são *arrays* de blocos lógicos, de granularidade fina ou média, cuja configuração é controlada por escrita de dados em uma memória estática. Para um FPGA ser reconfigurável dinamicamente, a definição implica que ele deve ser capaz de ser reconfigurado parcialmente enquanto ativo (energizado e em operação). Em nível de sistema, um módulo que contenha múltiplos FPGAs reconfiguráveis, total ou parcialmente, pode ser classificado como reconfigurável dinamicamente se os componentes FPGAs são reconfigurados individualmente.

Como os tempos de configuração não são desprezíveis, a habilidade de intercalar execução e reconfiguração, sem prejuízo do desempenho, é uma questão que ainda merece atenção e esforços de pesquisa. Um sistema de reconfiguração dinâmica inclui pelo menos uma área de reconfiguração onde blocos lógicos podem ser carregados em tempo de execução. A figura 4.2 ilustra a reconfiguração dinâmica de um sistema composto de cinco circuitos ou tarefas. A tarefa A é uma função de entrada que alimenta as tarefas B, C ou D, que alimentam a tarefa E que é a função de saída do sistema. As tarefas de entrada e saída são permanentemente residentes no FPGA enquanto as três tarefas dinâmicas alternam-se sob o controle de um sinal de *Swap*.

FIGURA 4.2: EXEMPLO DE RECONFIGURAÇÃO DINÂMICA.

O gráfico da figura 4.3 ilustra a execução, em função do tempo, das tarefas A, B, C, D e E da figura 4.2, durante reconfiguração total do dispositivo. No gráfico da figura 4.4 é ilustrada a execução das mesmas tarefas, durante reconfiguração dinâmica.

FIGURA 4.3: EXECUÇÃO DAS TAREFAS COM RECONFIGURAÇÃO TOTAL.

FIGURA 4.4: EXECUÇÃO DAS TAREFAS COM RECONFIGURAÇÃO DINÂMICA.

A capacidade de reconfiguração dinâmica permite o compartilhamento ao longo do tempo de tarefas diferentes, o que pode reduzir significativamente a área de silício exigida. Esta tecnologia torna possível o conceito de *hardware* ilimitado ou “*Hardware Virtual*” [22]. Este conceito é similar ao de memória virtual e está sendo adotado para permitir que múltiplos conjuntos de configuração sejam armazenados para

serem usados pelas RPU's (*Reconfigurable Processing Units*), quando necessário. O *hardware* virtual é implementado por compartilhamento de tempo de uma dada RPU, necessitando que um escalonador seja responsável pelas configurações, execuções e comunicações entre as partições temporais.

O projeto de um sistema que explore essa possibilidade possui vários problemas [24]:

- o particionamento da descrição em alto nível de um sistema em blocos (tarefas) que possam ser trocadas (carregadas e descarregadas) na área reconfigurável;
- a busca de um escalonamento otimizado para esses blocos; e
- a determinação de um conjunto de serviços que auxiliam a executar apropriadamente a funcionalidade requerida.

A questão crucial nesta abordagem é a implementação de um controlador capaz de manipular a reconfiguração de todas essas tarefas. Este controlador deve oferecer (assim como Sistemas Operacionais) serviços como o carregamento e remoção de tarefas, escalonamento de tarefas e gerenciamento dos recursos.

A reconfiguração dinâmica tem sido proposta no projeto de alguns sistemas experimentais. A Brigham Young University, em Provo, tem usado computação reconfigurável para construir um computador com um conjunto de instruções dinâmicas (DISC) [17], que efetivamente casa um microprocessador com um FPGA e demonstra o potencial da reconfiguração automática usando configurações armazenadas. À medida que um programa é executado, o FPGA requisita a reconfiguração se o circuito designado não está residente. A arquitetura DISC permite que o projetista crie e armazene um grande número de configurações e as ative assim como um programador costuma chamar uma sub-rotina de *software* em um microprocessador.

4.3. Dispositivos que Permitem Reconfiguração Dinâmica

Nota-se, nas diversas pesquisas na área [2, 3, 51, 52, 53], que a tecnologia de programação de FPGAs utilizada em arquiteturas de reconfiguração parcial e reconfiguração dinâmica, é a baseada em SRAM. É óbvio o porquê de não se utilizar

FPGAs *antifuse* ou baseados em EPROM, devido à incapacidade de reconfiguração parcial destas tecnologias.

Quanto à utilização de FPGAs baseados em EEPROMs, embora seja possível, a tecnologia SRAM apresenta inúmeras vantagens para arquiteturas dinamicamente reconfiguráveis, tais como: maior velocidade de reconfiguração; maior densidade lógica; mecanismos de roteamento que permitem funções mais complexas; e etc. Estes mecanismos apresentam maior nível de granularidade que FPGAs baseados em EEPROMs, que em geral fazem o roteamento por termos de produtos de ANDs e ORs. Outro fato relevante a considerar é que a tecnologia SRAM, comparado com o processo de fabricação EEPROM, possui um processo de fabricação bem mais simples, tendo os dispositivos FPGAs baseados em SRAM evoluído tanto quanto as memórias de computadores que utilizam esta tecnologia. Uma desvantagem da tecnologia SRAM comparada com a EEPROM é a volatilidade.

Comparando a tecnologia SRAM com a, *antifuse*, além da desvantagem da volatilidade, a tecnologia SRAM consome mais energia, dada à maior impedância das conexões, são mais lentas e expõe a propriedade intelectual, ficando vulnerável à pirataria [54]. Porém, como já discutido é impossível se quer obter reconfiguração parcial com a tecnologia *antifuse*.

Assim, dos fabricantes de FPGAs apresentados na tabela 3.1, apenas os com tecnologia SRAM serão utilizados para esse trabalho. Vale observar que embora a tecnologia SRAM apresenta inúmeras vantagens para a reconfiguração dinâmica de dispositivos FPGAs, não significa que todos dispositivos FPGAs que utilizam esta tecnologia são dinamicamente reconfiguráveis. Isso dependerá de como são as arquiteturas e mecanismos de configuração e roteamento de cada família e fabricante destes dispositivos.

Existem outros dispositivos que podem fornecer reconfiguração parcial e reconfiguração dinâmica, como por exemplo, os FIPSoCs e FPSLIC, vistos no item 2.4 do capítulo II, e os DPGAs (*Dynamic Probramable Gate Arrays*), apresentados a seguir.

Semelhantemente aos FPGAs, DPGAs são compostos de uma cadeia de elementos computacionalmente simples. Muitos FPGAs são construídos como LUTs programáveis. A LUT também é um elemento configurável do DPGA. Cada elemento do DPGA usa uma segunda LUT para armazenar um diferente contexto. Uma

mensagem de configuração global informa a cada elemento do DPGA qual a função que irá desempenhar no próximo ciclo de *clock*. A interconexão configurável em um DPGA tem uma tabela de configurações carregadas, e seleciona entre as configurações baseada no identificador de contexto corrente.

A organização interna de um DPGA é particularmente bem adaptada para computação reconfigurável [55]. Diferentemente de FPGAs normais, onde a função de cada elemento da matriz é determinada por seqüências relativamente lentas de reconfiguração, os elementos do DPGA podem ser chaveados, em apenas um ciclo de *clock*, entre várias configurações pré-programadas.

Ao custo de um tamanho maior de elementos e interconexão, o DPGA serve como um FPGA de múltiplo contexto. Com o espaço dos contextos pré-carregados, o DPGA pode chavear entre funcionalidades completamente diferentes, de um ciclo de *clock* para outro. A tabela de configuração do DPGA atua como uma memória *cache* de configurações de elementos da matriz. Dando a cada elemento uma pequena memória *cache* de configuração, consegue-se efetivamente uma alta taxa de reconfiguração.

Os múltiplos contextos carregados permitem a utilização mais eficiente da matriz de elementos. Em aplicações mais pesadas, a rápida reconfiguração permite a uma única matriz DPGA ser carregada com múltiplas configurações simultaneamente. O DPGA pode chavear entre configurações para acelerar diferentes partes de uma aplicação.

Embora existam outras arquiteturas que forneçam reconfiguração dinâmica para arquiteturas de computação reconfigurável, o foco deste trabalho é a tecnologia FPGA. A seguir é discutido a reconfigurabilidade dos dispositivos FPGAs das famílias APEX e Stratix da Altera, ORCA da Lattice, AT40K da Atmel e Virtex da Xilinx.

4.3.1. A Família APEX da Altera

A Altera alega que a família APEX permite alterações *real-time* durante a operação do sistema [43, 45]. Contudo estas alterações *real-time* são muito limitadas diante do conceito de reconfiguração dinâmica. As alterações *real-time* só são possíveis através do projeto da lógica em ESBs, utiliza-se LUTs onde podem ser implementadas funções com até 4 entradas. Depois dessa lógica ser implementada em ESBs o sistema

pode reescrevê-la em qualquer momento, mudando a configuração de parte do sistema. A grande limitação desta abordagem é que em algum lugar do circuito, ESBs implementados como blocos de memória RAM devem armazenar todas as configurações possíveis que irão modificar a LUT, isto porque não há como fazer o carregamento externo de novas configurações, muito menos o *readback*. *Readback* é a capacidade que alguns dispositivos tem de permitir a leitura externa dos dados de configuração corrente, enquanto ativo. A Altera também alega que “*real-time reconfiguration* é realizada colocando o dispositivo em modo de configuração, carregando novos dados de configuração, reiniciando o dispositivo e voltando o dispositivo para o modo usuário” [43]. Porém isto que a Altera chama de *real-time reconfiguration*, difere do sinônimo de reconfiguração dinâmica apresentado no item 4.2 deste capítulo, pois o dispositivo é reinicializado, o que afeta todo o dispositivo.

4.3.2. A Família Stratix da Altera

Com a família de dispositivos Stratix, embora a Altera tenha apresentado uma evolução arquitetural diante das famílias APEX e APEX II, ela ainda não oferece dispositivos com recursos para a reconfiguração dinâmica ou mesmo parcial, nos termos discutidos no item 4.2 deste capítulo.

Nesta arquitetura a Altera se refere à possibilidade de reconfiguração parcial do dispositivo [46], sem afetar toda a funcionalidade do sistema, por meio do controle externo de PLLs, podendo reconfigurar multiplicadores, divisores e configurações de atraso de tempo. Ainda assim este é um conceito limitado a alguns elementos de granularidade grossa, que não deve ser confundido com a reconfigurabilidade dinâmica necessária ao *hardware* virtual.

4.3.3. A Família ORCA da Lattice

Todos dispositivos da família ORCA foram projetados para permitir reconfiguração parcial [49]. Isto é realizado configurando previamente uma opção na *bitstream* que inibe, durante a reconfiguração, o *reset* total dos dados de configuração correntes do dispositivo. Então, apenas as partes que necessitarão ser alteradas terão que

ser sobrescritas, reduzindo assim o tempo de reconfiguração. *Bitstream* é o conjunto de dados de configuração de um dispositivo FPGA.

Outra opção de configuração da *bitstream* também disponível permite que uma parte do dispositivo permaneça em operação enquanto uma reconfiguração parcial é realizada, caracterizando assim a capacidade de reconfiguração dinâmica destes dispositivos.

O menor conjunto de dados de configuração que pode ser reconfigurado atomicamente é um *frame*, que nesta família contém cerca de 1 kbit de dados de configuração, um dispositivo ORCA chega a necessitar de mais de 3000 *frames* para a sua reconfiguração total. Existem dois formatos de *frames* para a configuração destes dispositivos, o auto-incremental e o explícito, os dois modos são similares exceto que o modo auto-incremental assume que os endereços dos *frames* são incrementais, reduzindo assim o tamanho da *bitstream*, e o modo explícito envia junto com cada *frame* seu endereço de armazenamento na memória de configuração do dispositivo, permitindo assim que um único *frame* seja enviado para um endereço específico, reconfigurando uma região específica do dispositivo.

Os dispositivos ORCA permitem o *readback* dos dados de configuração correntes.

4.3.4. As Famílias AT6000 e AT40K da Atmel

Em dispositivos das famílias AT6000 e AT40K é possível implementar *Cache Logic* [56] para construir sistemas e lógicas adaptáveis, permitindo reconfigurar parte do dispositivo sem interromper a operação da lógica remanescente [47, 48]. *Cache Logic* é uma tecnologia desenvolvida e patenteada pela Atmel. A implementação desta tecnologia é conceitualmente similar ao conceito de memória *cache*. Num sistema de *Cache Logic* somente as porções da aplicação que estão ativas em um dado momento realmente estão implementadas no FPGA, enquanto configurações de funções inativas são armazenadas externamente em uma memória. Se novas funções são necessárias, as configurações são sobrescritas parcialmente, como mostrado a figura 4.5.

FIGURA 4.5: RECONFIGURAÇÃO UTILIZANDO *CACHE LOGIC*.

A habilidade para implementar *Cache Logic* requer FPGAs que sejam capazes de ser reconfigurados parcialmente no sistema, sem que as funções ativas sejam prejudicadas durante a reconfiguração da parte que estará sendo sobrescrita, caracterizando assim a reconfigurabilidade dinâmica destes dispositivos que suportam *Cache Logic*. Outro requisito para a implementação de *Cache Logic* é uma arquitetura simétrica, como é o caso da arquitetura das famílias AT6000 e AT40K vista no capítulo III. Esta simetria é necessária para tornar possível o posicionamento de blocos genéricos na localização que estiver disponível na ocasião.

Existem dois tipos de *Cache Logic*, pré-determinada e dinâmica. O primeiro tipo envolve o uso de funções predeterminadas, gravadas numa memória externa não volátil (EPROM, por exemplo). Essas funções são previamente roteadas e posicionadas, gerando a *bitstream* correspondente. A implementação dessas funções é previamente controlada por um gerenciador residente na *Cache Logic*. Novas funções devem ser carregadas na *Cache Logic* em *background*, sem promover uma parada na operação de *cache*.

O segundo tipo de *Cache Logic*, dinâmica, é a base para construção de um *hardware* evolutivo. *Cache* dinâmico envolve a determinação da lógica, posicionamento, roteamento, geração da *bitstream* e configuração da *Cache Logic* em tempo de execução. Os principais aspectos abrangidos no desenvolvimento dessa capacidade incluem o escalonamento, alocação de funções, coleta de lógica aleatória e detecção de colisão.

4.3.5. A Família Virtex da Xilinx

A Xilinx, com sua família de FPGAs XC6200 [36], apresentada no capítulo III, foi uma das empresas pioneiras na fabricação de dispositivos que suportassem reconfiguração parcial. Mesquita [53] aponta as famílias de FPGAs CLAY [45] da National e CAL1024 [46] da Algotronix, assim como a família XC6200 da Xilinx, como sendo os primeiros dispositivos a suportarem reconfiguração parcial. Porém, tais FPGAs não alcançaram grande sucesso comercial, principalmente, pelo fato de não terem sido produzidas ferramentas eficientes de projeto, roteamento e posicionamento, que explorassem a reconfigurabilidade parcial destes dispositivos.

Com a família Virtex, a Xilinx volta a fornecer dispositivos que suportam reconfiguração parcial e reconfiguração dinâmica, esta última é chamada pela Xilinx de reconfiguração *on-the-fly* [39]. Assim como nos dispositivos ORCA da Lattice, o menor conjunto de dados de configuração que pode ser reconfigurado por vez é um *frame*. O tamanho de um *frame* depende do tamanho do dispositivo, o dispositivo XC2V8000 da família Virtex II contém 9.152 *bits* de dados de configuração por *frame* e possui sua memória de configuração dividida em 2860 *frames*, ou seja, existem 2860 partes reconfiguráveis individualmente. Estes dispositivos também permitem o *readback* dos dados de configuração correntes.

Os dispositivos da família XC4000 da Xilinx já possuíam mecanismos de *readback* dos dados de configuração, assim como divisão destes dados em *frames* [35]. Por exemplo, o dispositivo XC4025E possuía 1220 *frames*, com 346 *bits* por *frame*. Estas características já permitiam reduzidos tempos de reconfiguração dos dispositivos, por permitirem que apenas os *frames*, diferentes dos *frames* correntes, necessitassem ser recarregados, porém nesta família a operação lógica do dispositivo era completamente suspensa durante a reconfiguração, mesmo que para um único *frame*.

A tabela 4.1 compara os dispositivos FPGAs com tecnologia de programação SRAM, quanto à capacidade de reconfiguração e granularidade.

Como mostra a tabela 4.1, nenhuma família de dispositivos da Altera permite reconfiguração dinâmica, sendo portanto inviável utilizá-las em arquiteturas dinâmicas de computação reconfigurável.

TABELA 4.1: CAPACIDADE DE RECONFIGURAÇÃO DE FPGAs SRAM.

Família	Fabricante	Parcial	Dinâmica	Granularidade
XC4000	Xilinx	Sim	Não	Média
XC6200	Xilinx	Sim	Sim	Fina
Virtex	Xilinx	Sim	Sim	Média
Virtex II	Xilinx	Sim	Sim	Média
Virtex II Pro	Xilinx	Sim	Sim	Média / Grossa
APEX 20K	Altera	Não	Não	Média
APEX II	Altera	Não	Não	Média
Stratix	Altera	Não	Não	Média / Grossa
AT6000	Atmel	Sim	Sim	Fina
AT40K	Atmel	Sim	Sim	Média
ORCA 4	Lattice	Sim	Sim	Média

4.4. Arquiteturas de Computação Reconfigurável

Normalmente um sistema de computação reconfigurável é implementado com múltiplos FPGAs (baseados em SRAM), os quais atuam como RPU (*Reconfigurable Processing Units*), acoplados a um computador hospedeiro (*host*) [22]. A junção de *hardware* dedicado com um computador *host*, para acelerar algumas tarefas computacionalmente intensivas, como mostra a figura 4.6, não é um conceito novo. Um exemplo comum na indústria de PCs é o uso de placas aceleradoras gráficas. Porém, tais circuitos aceleradores para aplicações específicas requerem grande esforço de projeto e tipicamente usam mais área de silício que o próprio microprocessador.

FIGURA 4.6: *HARDWARE* DEDICADO COM COMPUTADOR *HOST*, PARA ACELERAR TAREFAS COMPUTACIONAIS.

O uso de *hardware* reconfigurável torna possível juntar o computador *host* com

recursos de *hardware* mais flexíveis, melhor adaptados à aplicação sendo executada, e com a possibilidade de adaptar-se às versões novas de uma aplicação.

A partir do amadurecimento da tecnologia FPGAs, alguns centros de pesquisa criaram as primeiras arquiteturas reconfiguráveis, com o intuito principal de aumentar o desempenho de algoritmos, que até então eram executados em *software*. Dentro desta primeira geração estão projetos como o PRISM [59] e o Splash [60]. Alguns sistemas mais modernos ainda utilizam essa abordagem, como o Transmogrier-2 [61], o RPM-2 [62] e o Spyder [63].

Já neste primeiro momento verificou-se a eficiência da utilização de FPGAs no domínio de aplicação em questão, tanto em termos de desempenho, com relação a abordagens em *software*, quanto no que tange ao critério econômico, quando comparada a soluções ASIC. Contudo também alguns problemas foram levantados. Em geral esses sistemas possuíam um gargalo de comunicação entre *host* e FPGAs e apresentavam um tempo de reconfiguração muito alto, além de poderem ser reconfigurados apenas totalmente. Essa última desvantagem significa que o sistema precisava necessariamente ser parado para que pudesse ser reconfigurado.

Em função destas dificuldades novas propostas de arquiteturas surgiram. A possibilidade de reconfiguração dinâmica, apresentada nos itens 4.2 e 4.3 deste capítulo, permitiu que as arquiteturas em questão pudessem ser reconfiguradas sem que precisassem parar totalmente de desempenhar suas funções, além de reconfigurar-se mais rapidamente por não precisar carregar todo o contexto novamente. Outro avanço tecnológico ocorrido foi o aumento do número de transistores por *chip*, tornando possível o desenvolvimento de sistemas SoCs que viessem a combater o problema de comunicação entre *hosts* e FPGAs.

O tipo de interconexão entre a RPU e o sistema hospedeiro e o nível de granularidade da RPU constituem dois importantes aspectos a serem considerados na implementação de sistemas reconfiguráveis. Muitos pontos relativos a esses tópicos já foram explorados e muitos ainda remanescem para pesquisa adicional. Porém, ainda não é possível identificar uma solução dominante para todos os tipos de aplicações.

4.4.1. Acoplamento de RPU's com o Host

O tipo de acoplamento das RPU's com o sistema de computação existente, tem um impacto grande no custo de comunicação, entendendo-se como custo o tempo gasto na comunicação. O acoplamento pode ser classificado em um dos quatro grupos ilustrados na figura 4.7.

FIGURA 4.7: TIPOS DE ACOPLAMENTO DE RPU'S COM O SISTEMA HOST.

Os tipos de acoplamento ilustrados na figura 4.7 são apresentados em ordem decrescente de custo de comunicação [22]:

- RPU acoplada ao barramento de I/O do *host*. Este grupo inclui muitas placas de circuitos comerciais. Alguns deles são conectados ao barramento PCI de um PC ou *workstation*;

- RPU acoplada ao barramento local do *host*;
- RPU acoplada como coprocessadores; e
- RPU atuando como um *datapath* estendido do processador.

4.4.2. Acoplamento de FPGAs

Embora seja possível (dependendo da capacidade do dispositivo e complexidade do sistema) implementar em um único dispositivo FPGA um completo sistema SoC, caracterizando o tipo de acoplamento apresentado na figura 4.7(d), pode ser necessária a utilização de vários FPGAs para implementar pelo menos uma ou mais RPUs, ou para fornecer a capacidade de *hardware* desejado ao sistema. Assim surge a necessidade do acoplamento de FPGAs.

Neste caso não seria necessário o uso de barramentos complexos e genéricos como o PCI. Como se trata de acoplamento entre *chips*, poderia-se utilizar algum barramento específico como o PBM da Altera (item 3.2.3), ou algum barramento independente da tecnologia como o AMBA (*Advanced Microcontroller Bus Architecture*) da ARM [64].

A figura 4.8 ilustra o acoplamento de diversos FPGAs APEX por meio do barramento PBM da Altera. A figura 4.9 ilustra o acoplamento de diversos dispositivos do mercado, independentemente da tecnologia, por meio do barramento AMBA.

FIGURA 4.8: ACOPLAMENTO DE FPGAs DA APEX ATRAVÉS DO BARRAMENTO PBM.

FIGURA 4.9: ACOPLAMENTO DE DISPOSITIVOS, ATRAVÉS DO BARRAMENTO AMBA.

4.4.3. Topologias de Acoplamento de FPGAs

Outra opção no acoplamento de vários dispositivos FPGAs, além da utilização de barramentos paralelos, seria a construção ou utilização de alguma topologia de acoplamento dos pinos de I/O do dispositivo. Existem inúmeras topologias de acoplamento e também seria possível criar uma topologia, específica para uma dada necessidade, sempre visando atingir a capacidade de *hardware* desejado, quando o sistema não é comportado por um único dispositivo. A seguir são apresentadas algumas topologias conhecidas, sem entrar nos detalhes de funcionamento de cada topologia.

As mais comuns topologias de interconexão multi-FPGAs são a topologia *Mesh* e a topologia *Crosbar* [65]. Na topologia *Mesh* os *chips* são conectados entre seus vizinhos, como mostra a figura 4.10. Esta topologia tem a vantagem da simplicidade, devido ao padrão de interconexão puramente local, facilitando futuras expansões.

FIGURA 4.10: TOPOLOGIA *MESH*.

A topologia *Grossbar*, figura 4.11, separa os elementos do sistema entre dispositivos FPGAs (W-Z), que implementam a lógica do sistema e *chips* (A-D), que implementam apenas o roteamento dos dispositivos de lógica.

FIGURA 4.11: TOPOLOGIA *CROSSBAR*.

Os *chips* de roteamento são conectados apenas aos FPGAs, e usualmente tem exatamente o mesmo número de conexões para toda a lógica implementada nos FPGAs. Da mesma forma, os FPGAs que implementam a lógica do sistema só são conectados com os *chips* que implementam o roteamento. É claro que os *chips* de roteamento também podem ser implementados em FPGAs, desde que não sejam nos mesmos que implementam a lógica do sistema. A idéia atrás desta topologia é que para rotear pontos entre algum conjunto de FPGAs, que implementam a lógica do sistema, seja utilizado um *chip* extra. Desta forma esta topologia garante que qualquer ponto de lógica do sistema pode ser interconectado pelos *chips* de roteamento, sem que se gaste esforços ou recursos lógicos dos FPGAs, que implementam a lógica do sistema.

Comparando a topologia *Crossbar* com a topologia *Mesh*, nota-se que a topologia *Mesh* pode levar a situações, em que uma dada necessidade de roteamento, exija que dispositivos diferentes dos que necessitam ser roteados sejam necessários, como por exemplo, à interconexão de um pino do dispositivo A da figura 4.10, com um pino do dispositivo E, neste caso será necessário que parte da lógica do dispositivo D ou B seja utilizada para realizar a interconexão.

Embora a topologia *Crossbar* apresente a vantagem do roteamento total, sem prejudicar a implementação da lógica do sistema, esta topologia além de necessitar de *chips* exclusivos para a implementação do mecanismo de roteamento, ainda assim não é

uma topologia expansível como a topologia *Mesh*. Uma expansão na topologia *Crossbar* implica na troca ou reconfiguração de todos os dispositivos de roteamento do sistema, o que afetaria todo o sistema.

Um exemplo de topologia implementada pela indústria é o sistema de emulação da Aptix [65]. A figura 4.12 apresenta a topologia do sistema Aptix AXB-AP4.

FIGURA 4.12: TOPOLOGIA DO APTIX AXB-AP4.

Este sistema possui uma topologia semelhante à topologia *Crossbar*, neste caso existe um FPIC (*Field Programmable Interconnect Component*) para cada conjunto de 5 FPGAs, tendo ao total 4 FPICs e 20 FPGAs. Nesta topologia seria necessária a utilização de no máximo 2 FPIC para rotear os mais distantes FPGAs, como por exemplo os dispositivos A e K que poderiam ser interconectados pelos FPICs 1 e 3. Esta topologia, assim como a *Crossbar*, também não é expansível.

4.4.4. Granularidade das RPU's

A maioria dos esforços atuais de pesquisa utilizam FPGAs manipulando diretamente os elementos de baixo nível. Uma célula típica de FPGA consiste basicamente de um *flip-flop* e algumas LUTs que implementam funções booleanas de até 4 variáveis. Estes elementos podem ser usados para implementar quase que qualquer circuito digital. Entretanto, esta granularidade tem se mostrado ineficiente em tempo e área, para certas classes de problemas. Algumas pesquisas usam ALUs (*Arithmetic Logic Units*) como as primitivas básicas de *hardware*, aumentando a granularidade do sistema. Esta generalização pode fornecer soluções mais eficientes, porém diminuirá a flexibilidade do sistema.

Para se obter um melhor nível de abstração é necessário, definido o nível de granularidade, construir bibliotecas de unidades funcionais com correspondência direta aos operadores do *software* em linguagens de alto nível. Estas bibliotecas poderão ter mais de uma implementação do mesmo operador (relações diferentes de tempo, área, latência, configuração e etc.). Além disso, o uso de uma biblioteca diminuirá o tempo global de compilação do *hardware* reconfigurável.

Vários conceitos de arquiteturas foram considerados para uso em RPU's, com diferenças na flexibilidade e nível de granularidade usado. Com a existência de RPU's que permitem reconfiguração dinâmica, é possível reconfigurar regiões de RPU enquanto outras são executadas. Este tipo de RPU apresenta grande vantagem sobre FPGAs tradicionais [22].

4.5. Considerações Finais

Dos dispositivos reconfiguráveis dinamicamente apresentados no item 4.3 deste capítulo pode-se separá-los basicamente em quatro grupos distintos:

- primeiro grupo: são os dispositivos SoCs como o FIPSoC e o FPSLIC que suportam reconfiguração dinâmica de partes específicas do dispositivo, caracterizando uma reconfigurabilidade parcial de granularidade grossa, exceto em algumas partes do dispositivo que possuem células de menor granularidade;

- segundo grupo: ficam os dispositivos que provêem a reconfiguração dinâmica por meio de chaveamento de contexto, como o caso dos DPGAs e os FPGAs da Atmel que implementam *Cache Logic* no modo pré-determinado;
- terceiro grupo: temos os FPGAs ORCA, Virtex, e AT40K (implementando *Cache Logic* no modo dinâmico), que apresentam capacidade de reconfiguração dinâmica com granularidade média, *readback* dos dados de configuração correntes, flexibilidade e simetria na arquitetura, estas características são essenciais quando se almeja um sistema eficiente de *hardware* virtual (*hardware* evolutivo), mecanismos de escalonamento, paginação, desfragmentação de *hardware* e etc.; e
- quarto grupo: são os FPGAs da família XC6200 e da família AT6000 (implementando *Cache Logic* no modo dinâmico), que apresentam granularidade fina, o que teoricamente flexibiliza a implementação lógica, mas na prática complica muito a implementação de sistemas dinâmicos dado os pobres recursos apresentados pelas atuais e surpreendente ferramentas que se propõem à realizar compilação e controle do *hardware* para plataformas dinâmicas.

Vale observar que mesmo para o terceiro grupo não existe ferramentas e mecanismos maduros para tal fim e que para atingir este objetivo ainda é necessário que o nível de granularidade seja definido, sendo a unidade atômica uma LUT, uma ALU, um processador ou um *bit*. É claro que um processador ou um *bit* são os extremos da flexibilidade sobre a complexidade, não sendo nenhuma destas uma boa solução. É necessário obter-se um ponto ótimo nesta definição, possibilitando a abstração do *hardware* para o desenvolvimento destes mecanismos, aplicações e sistemas de controle, o que ajuda em parte dos problemas, porém não todos.

O posicionamento e roteamento para sistemas dinâmicos ainda apresentam grandes desafios que necessitarão de ferramentas conceitualmente diferente das atuais disponíveis pelos fabricantes, onde a abstração do *hardware* é apenas um pequeno passo deste desafio. Surge a necessidade de um sistema operacional para o *hardware* [24] que possa implementar conceitos semelhantes aos utilizados em sistemas que implementam memória virtual, é claro que as dificuldades são muitas, porém menores que as

perspectivas de superação dos limites impostos pelas tradicionais arquiteturas de computação.

Embora ainda não exista uma granularidade definida, o projeto JBits SDK da Xilinx [66] é uma tentativa de fornecer ferramentas dinâmicas de desenvolvimento, assim como suporte a criação de ferramentas dinâmicas de desenvolvimento. Uma das maiores limitações do JBits é justamente quanto aos recursos de posicionamento e roteamento. O projeto JBits é apresentado no capítulo seguinte.

Outra consideração relevante é quanto à mínima quantidade de *bits* de configuração reconfiguráveis atômicamente. Como visto, no caso dos FPGAs ORCA e Virtex, a mínima quantidade de *bits* possíveis de serem reconfigurados é um *frame*, que tem sua quantidade de *bits* proporcional ao tamanho do dispositivo. De fato, o que deve ser considerado é que os *bits* destes *frames* são unidades de memória SRAM utilizadas para configurar o circuito. Considerando em particular o caso de uma configuração de roteamento, que se dá por um *bit* de memória SRAM ligado ao *gate* de um transistor FET (*Field-Effect Transistors*), trabalhando como chave (corte/saturação), mesmo que seja reconfigurado o *frame* que contenha o *bit* do *gate*, a impedância entre o *drain* e o *source* só será alterada caso o *bit* referente ao *gate* deste transistor também seja alterado. Outro exemplo é o caso de uma LUT, onde o *bit* de configuração é o próprio *bit* de saída da LUT, que também só será alterado se os *bits* referentes às saídas da LUT forem alterados no novo *frame*.

Assim, a quantidade mínima de *bits* de configuração reconfiguráveis atômicamente, na realidade, é de um único *bit*, desde que o *frame* de reconfiguração seja enviado ao dispositivo com um único *bit* alterado. Isto é possível certificar-se com a leitura do *frame* corrente, pelo processo de *readback* destes dispositivos. Então, o fato de não ser possível enviar um único *bit* de reconfiguração, não implica na impossibilidade de reconfigurar apenas um único *bit* de configuração, mantendo em funcionamento o restante do circuito configurado pelos outros *bits* do *frame*, além de todas as regiões configuradas pelos outros *frames* do dispositivo. Porém essa atomicidade, como visto anteriormente, deve ser descartada dada a complexidade que apresentaria às ferramentas de desenvolvimento e controle. Entretanto, cabe ressaltar que é possível atingir esta meta.

Neste capítulo foi discutido a reconfigurabilidade dinâmica de um único

dispositivo. Porém, é interessante observar que para um sistema multi-FPGAs, que possa ser composto por um barramento de controle, como o AMBA, ou uma topologia de acoplamento, como a *Mesh* ou a *Crossbar*, o *hardware* pode ser dinamicamente reconfigurável, mesmo utilizando-se dispositivos que não provêem tal capacidade individualmente. Considerando-se que a reconfiguração total de um único dispositivo do sistema não afete todo o funcionamento do mesmo, este sistema será dinamicamente reconfigurável. Embora tal reconfigurabilidade seja de granularidade muito grossa, esse artifício pode significar substancial economia de *hardware* para o sistema, visto que inúmeros contextos podem ser carregados por um *host* ou armazenados em memórias externas, caracterizando um sistema muito mais flexível, com um custo muito menor do que o de implementar todo o *hardware*.

Ferramentas de Desenvolvimento para Aplicações Dinâmicas e Remotas

“Minha crença pessoal é que se a computação reconfigurável pretende ter sucesso ela deve criar uma metodologia para converter automaticamente um programa numa linguagem de programação padrão para o hardware do sistema.”

Scott Hauck (University of Washington), 1998

5.1. Considerações Iniciais

JBits SDK [66] é um projeto da Xilinx na tentativa de desenvolver uma ferramenta, ou suporte ao desenvolvimento de ferramentas, para o desenvolvimento de aplicações reconfiguráveis em tempo de execução, seja de forma dinâmica, parcial [78] ou total. Nenhuma outra ferramenta de desenvolvimento de projeto, dos atuais fabricantes de FPGAs, sai da linha de desenvolvimento convencional. JBits apresenta um novo conceito no desenvolvimento de projetos e aplicações, como será apresentado a seguir.

Embora JBits seja um projeto pioneiro na indústria, ferramentas para reconfiguração dinâmica já é objeto de pesquisa a algum tempo. Em 1996 Lysaght [67] reporta a necessidade de novas ferramentas que suportem reconfiguração dinâmica. Um projeto de destaque da Xilinx é o desenvolvimento, em parceria com a Chalmers University da Suécia, da linguagem de descrição de *hardware* Lava HDL que propõe uma nova abordagem de alto nível para a abstração do *hardware* [70].

Lava HDL tem alcançando sucesso e desempenho, cerca de 12 vezes mais rápida que as ferramentas tradicionais [71], porém conceitualmente não foi projetada para permitir reconfiguração parcial, o que limita muito a utilização desta ferramenta. Ainda assim, a abstração em alto nível obtida por esta linguagem, aliada ao desempenho

superior, mostra ser esta uma ferramenta muito interessante para sistemas reconfiguráveis dinamicamente como os discutidos no capítulo anterior, onde dispositivos são reconfigurados individualmente e por completo obtendo substancial economia de *hardware*. Outras ferramentas e pesquisas também buscam abstração em alto nível para a síntese de *hardware* [76, 77].

Outro projeto semelhante ao JBits é a pesquisa acadêmica realizada pela Brigham Young University em Provo, que desenvolve e administra o código aberto da ferramenta JHDL [68, 69]. Entretanto, neste capítulo será visto apenas a ferramenta JBits, com o objetivo de melhor exemplificar os novos conceitos e dificuldades do desenvolvimento de aplicações dinâmicas para arquiteturas de computação reconfiguráveis. Também serão vistos os recursos remotos desta ferramenta, assim como o conceito de IRL (*Internet Reconfigurable Logic*).

5.2. Introdução

JBits é um conjunto de classes Java, em desenvolvimento pela Xilinx, que provê uma API (*Application Program Interface*) para as *bitstreams* dos dispositivos Virtex da Xilinx. Esta interface é capaz de atuar sobre uma *bitstream* gerada por uma ferramenta de projeto Xilinx, ou sobre uma *bitstream* que seja lida do *hardware* atual pelo processo de *readback* dos dispositivos Virtex. Considerando-se que JBits tenha como entrada *bitstreams* nulas, previamente geradas por ferramentas convencionais de projeto Xilinx, pode-se dizer que JBits também poderá criar uma *bitstream* para um dispositivo Virtex. Assim, esta interface provê a capacidade de projetar, modificar totalmente, parcialmente e dinamicamente circuitos em um dispositivo Virtex. O modelo de programação adotado por JBits utiliza-se de uma matriz de CLBs, sendo uma CLB o elemento atômico deste modelo. Cada CLB é mapeado por sua linha e coluna, e todos recursos configuráveis em um CLB podem ser manipulados ou monitorados. Adicionalmente, os controles de todos os recursos de roteamento adjacentes ao CLB selecionado são disponíveis.

Devido ao controle ser ao nível de CLBs, com granularidade média, *bitstreams* podem ser modificadas ou geradas rapidamente. Esta API pode ser usada para construir circuitos completos ou para modificar circuitos existentes. Em adicional, o suporte à

orientação a objetos da linguagem de programação Java tem permitido a criação de pequenas bibliotecas parametrizadas, macros e *cores*. Finalmente, esta API pode ser utilizada como base para construção de outras ferramentas. Isto inclui as tradicionais ferramentas de projeto que executam tarefas como posicionamento e roteamento de circuitos, assim como ferramentas para aplicações específicas, onde seja necessária a execução de uma tarefa extremamente definida ou a busca de melhor desempenho e flexibilidade, como em sistemas dinamicamente reconfiguráveis.

Embora ainda tenha-se muito que desenvolver nesta ferramenta, principalmente quanto à automação dos recursos de posicionamento e roteamento dos circuitos, dada à extrema complexidade que tal nível de granularidade e flexibilidade gera, é possível afirmar que esta é uma poderosa ferramenta, e assim como toda forma de poder, esta também pode ser utilizada para inúmeros fins. É possível o uso de JBits para gerar dados de configuração que podem danificar tanto os dispositivos Virtex da Xilinx como o *hardware* contendo estes dispositivos [66]. Por esta ser uma poderosa ferramenta é fundamental que os projetistas conheçam muito bem seu circuito e os assuntos que envolvem a configuração e reconfiguração na arquitetura Virtex. Uma possibilidade que surge com o advento dos dispositivos dinamicamente reconfiguráveis e das ferramentas de suporte a esta tecnologia, como JBits, é a confecção por pessoas mal intencionadas de códigos que danifiquem propositalmente o *hardware* de um sistema. Por exemplo, torna-se possível a existência de vírus com poder de danificar fisicamente o dispositivo [6]. É claro que os benefícios, vantagens e perspectivas que esta tecnologia apresenta, como foi visto no capítulo anterior e ainda será discutido, são maiores que os problemas e as dificuldades encontradas.

A principal meta do projeto JBits é obter uma rápida e simples interface para a família de FPGAs Virtex da Xilinx. A motivação original para isto foi o suporte a aplicações que requerem rápida reconfiguração dinâmica de circuitos. Usando ferramentas tradicionais, a geração de circuitos a partir de um esquemático ou uma especificação VHDL pode levar vários minutos ou horas para produzir uma *bitstream*, sendo que até mesmo alguns segundos podem ser muito tempo para uma aplicação reconfigurável aguardar.

JBits tem várias características que a torna única e permite solucionar problemas não cobertos por ferramentas de projeto tradicionais. Algumas destas

características são:

- compilação de 10.000 linhas por segundo (Compilador Java Microsoft J++);
- suporte a reconfiguração dinâmica;
- suporte a parametrização de circuitos em tempo de execução;
- suporte a macros pré-posicionadas e pré-roteadas;
- modelo de programação em matrix de CLBs;
- suporte à família Virtex; e
- completamente implementado em Java.

A habilidade de criar configurações Virtex rapidamente e dinamicamente, tem possibilitado a exploração de novas aplicações para as famílias de FPGAs Xilinx existentes. Algumas destas novas aplicações são:

- ferramentas CAD personalizadas: ferramentas de aplicação específica, particularmente ferramentas usando reconfiguração dinâmica;
- ferramentas CAD tradicionais: dado que JBits fornece um modelo e uma interface em linguagem de programação para o *hardware* dos FPGAs Xilinx, JBits pode ser usado como uma base para produção de ferramentas CAD tradicionais;
- geração de *cores*: *cores* pré-posicionados e pré-roteados podem ser construídos utilizando JBits, estes *cores* podem ser parametrizados para produzir outros *cores*; e
- computação reconfigurável: projetos podem ser construídos e modificados dinamicamente, isto pode variar da modificação dinâmica de parâmetros e *cores* a um projeto completo de um circuitos.

JBits permite um nível de flexibilidade nunca visto em ferramentas estáticas de projeto como as de captura de esquemático e de HDLs, porém também apresenta inúmeros desafios.

5.3. Fluxo de Desenvolvimento e Execução

A seguir são apresentadas as diferenças entre o fluxo de desenvolvimento e execução de ferramentas estáticas e o fluxo de desenvolvimento e execução de JBits.

5.3.1. Fluxo de Desenvolvimento e Execução Utilizando Ferramentas Estáticas

O fluxo de desenvolvimento de projetos nas atuais ferramentas de desenvolvimento, tradicionalmente difere o *hardware* (estático) de sua aplicação, figura 5.1.

FIGURA 5.1: FLUXO DE DESENVOLVIMENTO E EXECUÇÃO UTILIZANDO FERRAMENTAS ESTÁTICAS.

Este fluxo limita-se a obter uma *bitstream* a partir de uma descrição comportamental, RTL (*Register Transfer Logic*), ou lógica. Uma vez obtida a *bitstream*, o máximo que uma aplicação alocada no *host* pode realizar é acessar as funcionalidades do *hardware* estático implementado.

5.3.2. Fluxo de Desenvolvimento Estático e Execução Utilizando JBits

JBits também pode ser utilizado para o desenvolvimento estático. Neste fluxo,

JBits é usado para desenvolver um projeto estático de uma *bitstream* para um dispositivo Virtex. O fluxo de desenvolvimento da figura 5.2 é diferente do fluxo de desenvolvimento de uma ferramenta tradicional. Ao contrário do fluxo de desenvolvimento convencional, que utiliza HDLs e/ou esquemáticos para a entrada de projetos, com JBits um projeto é especificado em um programa Java usando a JBits API e/ou JBits *cores*. A aplicação utiliza como entrada um arquivo contendo uma *bitstream* e extrai os dados de configuração do dispositivo. Estes dados são modificados, usando a JBits API, de acordo com o projeto especificado, gerando um arquivo de saída contendo a *bitstream* modificada que será utilizada para configurar o *hardware* Virtex. Uma vez efetuado o *download* para o *hardware* Virtex, o projeto poderá ser depurado usando o depurador de *hardware* Boardscope que ainda será apresentado neste capítulo.

FIGURA 5.2: FLUXO DE DESENVOLVIMENTO ESTÁTICO E EXECUÇÃO UTILIZANDO JBITS.

A *bitstream* de entrada pode ser uma *bitstream* nula ou uma de um projeto existente. Quando uma *bitstream* nula é adotada, a ferramenta JBits é utilizada para construir um novo projeto. JBits pode também ser utilizada para modificar porções da *bitstream* de um projeto existente. Isto permite aos projetistas trabalharem com *bitstreams* geradas por ferramentas convencionais de projeto da Xilinx, como uma *bitstream* gerada inicialmente pela ferramenta Foundation ou Alliance.

O modelo de execução para este fluxo utiliza a *bitstream* modificada pela

ferramenta JBits, que é carregada de uma memória como uma PROM, EPROM, Flash e etc. Esta *bitstream* é então utilizada para configurar o FPGA Virtex em uma placa *standalone*. A aplicação alocada no *host* tem as mesmas limitações de uma aplicação estática desenvolvida pelas ferramentas tradicionais.

5.3.3. Fluxo de Desenvolvimento Dinâmico e Execução Utilizando JBits

O poder real da ferramenta JBits é visto no desenvolvimento de aplicações de reconfiguração dinâmica, figura 5.3, também conhecidas como aplicações RTR (*Run Time Reconfiguration*).

FIGURA 5.3: FLUXO DE DESENVOLVIMENTO DINÂMICO E EXECUÇÃO UTILIZANDO JBITS.

Neste fluxo, os circuitos podem ser configurados e reconfigurados dinamicamente pela execução de uma aplicação Java, que comunica com o circuito da placa contendo o dispositivo Virtex. Isto é possível usando JBits para especificar o projeto e a API XHWIF para efetuar o *download* e *readback* da *bitstream*. Adicionalmente, esta *bitstream* de entrada para a aplicação Java também pode ser uma *bitstream* nula ou uma *bitstream* de um projeto existente, além da *bitstream* ativa no dispositivo.

No modelo de execução para este fluxo o computador *host* executa uma aplicação JBits e utiliza a API XHWIF para comunicar com o *hardware* reconfigurável

Virtex. Por exemplo, em uma típica plataforma PC o microprocessador *host* executa as aplicações JBits que, utilizando a API XHWIF, reconfigura o *hardware* Virtex localizado em uma placa com barramento PCI conectada em um dos *slots* da placa mãe. Assim é possível obter um sistema dinâmico para reconfiguração dos dispositivos Virtex.

Embora o conceito de fluxo dinâmico de desenvolvimento integre o desenvolvimento do *hardware* com a aplicação do usuário, a efetiva utilização deste conceito esbarra nas limitações das ferramentas de desenvolvimento dinâmico existentes. O modelo de programação JBits é muito poderoso no que tange a utilização dos recursos de baixo nível dos FPGAs, porém ainda é muito carente quanto aos recursos de posicionamento e roteamento em alto nível, o que não é diferente nas demais ferramentas que propõem o desenvolvimento dinâmico.

Vale observar que a capacidade de posicionamento do projeto na área desejada do *chip* é de extrema importância para a utilização das demais áreas para outras aplicações. Nas tradicionais ferramentas de desenvolvimento não existe este requisito, sendo almejado no posicionamento e roteamento apenas a otimização da densidade lógica e do desempenho.

Alguns detalhes do modelo de programação e alguns métodos de JBits são apresentados no apêndice A.

5.4. O Depurador de *Hardware* Boardscope

Boardscope é uma ferramenta gráfica interativa de depuração de projetos em FPGAs Virtex, ou no simulador de dispositivos VirtexDS. Esta ferramenta provê o controle do *hardware*, como o carregamento de *bitstreams* para os dispositivos Virtex, leitura dos dados de configuração, simples e multi-ciclos de *clock* para os dispositivos e o acionamento do *reset* dos dispositivos.

Os dados de configuração são mostrados de diferentes formas pelo Boardscope. O *State View* mostra a informação dos estados de configuração das CLBs, IOBs e BRAMs. O *Core View* mostra a localização dos *cores* do projeto, assim como informações detalhadas sobre eles. Outras informações dos dados de configuração são

mostradas pelas formas de visualização *Power View* e *Routing Density*. O *Waveform Viewer* monitora o estado dos *flip flops* e mostra as alterações da forma de onda a cada ciclo de *clock*. Há também um painel que mostra mais detalhes sobre os CLBs, como os dados das LUTs e conexões externas. A janela gráfica do depurador Boardscope pode ser visualizada na figura 5.4.

FIGURA 5.4: INTERFACE GRÁFICA DO BOARDSCOPE.

5.5. A API XHWIF

XHWIF (*Xilinx Hardware Interface*), é uma interface padrão para *hardwares* baseados em FPGAs Xilinx. Esta interface permite que novas plataformas de *hardware* sejam suportadas por aplicações JBits. Uma vez definida uma interface XHWIF, para um tipo particular de *hardware*, ferramentas como o Boardscope irão rodar sem

nenhuma recompilação ou modificação. Com isso, outra aplicação JBits que utiliza a interface XHWIF também irá rodar no novo *hardware*, usualmente sem modificação ou recompilação. Finalmente, parte da interface XHWIF é um pacote, baseado em TCP/IP, que suporta acesso remoto ao *hardware*. Uma vez definida a interface XHWIF para suportar um novo *hardware*, o servidor remoto e o acesso remoto ao *hardware* também serão automaticamente suportados.

5.5.1. XHWIF Interface

XHWIF é uma interface Java para comunicação com *hardwares* baseados em FPGAs Xilinx. Ela possui métodos para leitura e escrita de *bitstreams*, métodos para identificar o tipo e quantidade de FPGAs na placa do *hardware*, métodos para leitura e escrita de memórias, métodos de manipulação de *clocks* e etc. Essencialmente, esta interface descreve o *hardware* contido na placa, permitindo as aplicações JBits o total controle, manipulação e utilização do *hardware*.

Esta interface unifica o modo que aplicações comunicam com o *hardware*, de forma que usando a mesma interface, aplicações como Boardscope, por exemplo, podem comunicar com uma variedade de placas. Todas informações de especificação do *hardware* são guardadas dentro de uma classe que implementa a interface XHWIF. Usando métodos nativos da linguagem de programação Java, XHWIF pode comunicar diretamente com o *hardware*, por bibliotecas ou por um *driver*.

A vantagem desta abstração é que um novo *hardware* pode ser rapidamente e facilmente suportado pelas aplicações JBits. Tudo que é necessário é que uma interface XHWIF para o novo *hardware* seja implementada. Uma vez construída esta interface, todos *softwares* que utilizam a interface XHWIF rodarão no novo *hardware*, sem a necessidade de modificar ou recompilar o *software*.

5.5.2. XHWIF Server

Uma vez implementada a interface XHWIF para um novo sistema, aplicações poderão acessar remotamente o novo *hardware* utilizando o pacote XHWIF Server.

Tudo que é necessário é uma conexão TCP/IP para o *hardware* remoto. Este acesso remoto é útil, por exemplo, em aplicações com o Boardscope. Na figura 5.5 é mostrado como a interface XHWIF Server opera.

Utilizando acesso remoto o *hardware* pode ser compartilhado entre diversos usuários, de diferentes localizações, sem a necessidade de um *hardware* dedicado para cada usuário do sistema.

FIGURA 5.5: ACESSANDO REMOTAMENTE O *HARDWARE* UTILIZANDO A INTERFACE XHWIF.

A interface XHWIF provê acesso direto ao *hardware* baseado em FPGAs Xilinx. No modo local, o *hardware* é utilizado pelas aplicações JBits através da interface XHWIF, para executar todas as funções requeridas pelo sistema. No modo remoto, entretanto, a interface XHWIF, que tem acesso ao *hardware*, é acessada através da interface XHWIF Server, que por sua vez fornece acesso remoto ao sistema por uma conexão TCP/IP. As aplicações JBits podem acessar a interface XHWIF Server por intermédio da interface remota XHWIFnet que, obviamente, também necessita de uma conexão TCP/IP para acessar a interface XHWIF Server. A interface XHWIFnet, na visão das aplicações JBits, opera de forma idêntica a uma interface XHWIF que acessa diretamente um *hardware* local.

Para conectar remotamente a um sistema, a interface XHWIF Server deve estar rodando em um sistema contendo *hardware* baseado em FPGAs Xilinx, ou utilizar o simulador VirtexDS. A interface XHWIF Server é uma aplicação Java que aguarda por uma conexão remota de uma aplicação JBits. A operação remota da interface XHWIF é

completamente transparente, exceto possivelmente em conexões muito lentas. Não deveria haver nenhuma diferença entre utilizar a interface XHWIF de forma local ou remota. A figura 5.6 apresenta a janela gráfica da interface XHWIF Server.

FIGURA 5.6: ACESSO REMOTO À INTERFACE XHWIF SERVER.

Na figura 5.6, a opção de listar as mensagens recebidas foi manualmente habilitada. Isto permite uma visão detalhada das mensagens que são trocadas entre cliente e servidor. Neste caso, a ferramenta de depuração Boardscope está estabelecendo comunicação com alguma plataforma de *hardware*. Mensagens do cliente para o servidor também controlam vários aspectos do *hardware* como configuração de FPGA, modificação, *readback*, controle do *clock* e etc.

5.5.3. Sistemas Suportados pela API XHWIF

A interface XHWIF, atualmente fornecida com o JBits 2.8, apresenta suporte nativo aos seguintes sistemas de *hardware*:

- PametteVDC - DEC/Compaq PCI Pamette Virtex Daughtercard (XCV300);
- Pamettevdc800 - DEC/Compaq PCI Pamette Virtex Daughtercard (XCV800);
- Rc1000pp - ESL's RC1000PP Virtex Card (XCV1000); e
- Xsv - XESS Virtex Card (XCV800).

5.6. O Simulador VirtexDS

O simulador de dispositivos Virtex (VirtexDS – *Virtex Device Simulator*) proporciona aos usuários a habilidade de testar arquivos contendo *bitstreams* de dispositivos Virtex, sem a necessidade do *hardware*. VirtexDS provê esta versatilidade implementando a interface XHWIF, permitindo uso imediato com a ferramenta de depuração BoardScope e qualquer outra aplicação que utilize a interface XHWIF. Os arquivos contendo a *bitstream* a ser simulada podem ser criados por ferramenta convencionais de projeto da Xilinx, ou com aplicações JBits, entretanto, existem inúmeras limitações, que possivelmente serão sanadas com a continuidade no desenvolvimento do sistema JBits.

5.7. O Waveform Viewer

Waveform Viewer, figura 5.7, é uma ferramenta gráfica e interativa de depuração de FPGAs Virtex. Ela também pode ser utilizada com o VirtexDS.

FIGURA 5.7: INTERFACE GRÁFICA DO WAVEFORM VIEWER.

Esta ferramenta pode ser chamada pelo depurador Boardscope ou ser executada de forma isolada. Waveform Viewer é uma ferramenta similar a Boardscope que permite a definição e monitoração de estados a cada ciclo de *clock*, a diferença é a visualização dos estados por formas de onda, como mostrado na figura 5.7. As formas

de onda que são mostradas são especificadas em um arquivo SYM. O arquivo SYM especifica ligações e barramentos a serem monitorados.

5.8. Dispositivos Suportados por JBits

A atual versão disponível pela Xilinx, JBits 2.8, define na classe *Devices* os dispositivos suportados por ela, sendo eles:

- Virtex: XVC01, XVC04, XVC50, XVC100, XVC150, XVC200, XVC300, XVC400, XVC600, XVC800 e XVC1000;
- Virtex E: XVC50E, XVC100E, XVC200E, XVC300E, XVC400E, XVC405E, XVC600E, XVC812E, XVC1000E, XVC1600E, XVC2000E, XVC2600E e XVC3200E;
- Virtex II: não suportado; e
- Virtex II Pro: não suportado.

5.9. IRL

Na indústria da comunicação é comum que os equipamentos sejam digitais e interconectados por redes. Esta tecnologia pode habilitar o desenvolvimento de um novo conceito: *upgrades de hardware*. Surge a possibilidade de efetuar virtualmente o *upgrade de hardware*, melhorando ou trocando a funcionalidade sem a necessidade de enviar um técnico especializado ao local. Isto permitiria corrigir erros de projetos do *hardware* a uma fração do custo de hoje e também incrementos de novas características. Assim, estenderia a vida útil do equipamento significativamente. Por exemplo, novos protocolos padrões poderiam ser incorporados virtualmente. Claro que poderiam ser aplicadas virtualmente versões aperfeiçoadas de *hardwares* a muitas formas de equipamentos digitais como: estações de base celulares digitais, roteadores, *switches*, PABX, equipamento de rádio, satélites, modems, decodificadores de TV digital, arquiteturas de computação reconfigurável e etc. Para esta tecnologia ser possível são exigidos alguns elementos mostrados na figura 5.8 [72].

FIGURA 5.8: ELEMENTOS DE UMA IRL.

A rede digital é utilizada pelo *host* para enviar a nova *bitstream* ao *hardware* baseado em FPGAs. O *hardware* pode receber a nova *bitstream* via algum servidor local, como por exemplo, o XHWIF Server que pode permitir até mesmo uma reconfiguração dinâmica. Outra possibilidade é a utilização de uma conexão TCP/IP no próprio *hardware*, implementada em um processador embutido ou no *hardware* dedicado. Como resultado, a funcionalidade de *hardware* pode ser completamente reconfigurada virtualmente. Este conceito é denominado chamado de IRL (*Internet Reconfigurable Logic*).

Em um sistema de IRL o projetista pode simplesmente partir do projeto digital original em VHDL, por exemplo, e adicionar as novas características e correções de *hardware*. Depois de testar o novo projeto completamente, a nova *bitstream* pode ser enviada eletronicamente ao *hardware*. Este processo de atualização de *hardware* é semelhante ao processo de atualização de *software* ou *firmware*.

5.10. Considerações Finais

Embora JBits forneça poderosos recursos para configuração de cada CLB dos dispositivos Virtex, abstraindo do usuário a árdua tarefa de compreender a função de cada *bit* da *bitstream* para cada modelo de dispositivo, é fundamental que o usuário conheça em detalhes a arquitetura e organização dos dispositivos, pois todos os recursos dos dispositivos devem ser explicitamente relacionados ao código fonte da aplicação. Esta característica torna-se crítica pelo fato dos recursos de posicionamento e

roteamento de JBits ainda serem muito pobres, necessitando serem realizados praticamente de forma manual. Este problema é atenuado quando utilizados macros e *cores* parametrizados. Daí a necessidade de uma ferramenta de alto nível para automatizar esse processo.

Considerando que JBits, se não for a mais madura ferramenta de desenvolvimento de *hardware* dinâmico para aplicações RTR, com certeza está entre as mais maduras ferramentas para tal fim. Fica claro notar o quanto esta área de pesquisa ainda tem a desenvolver-se, abrindo inúmeras frentes de pesquisas inéditas. Adicionalmente, nem sequer pode-se afirmar que JBits seja uma ferramenta para o usuário final, pois ela caracteriza-se mais como uma ferramenta de suporte ao desenvolvimento de uma nova classe de ferramentas para aplicações dinâmicas e remotas.

Implementações

“Queremos fazer pelos FPGAs o que os browsers fizeram pela Internet. Queremos torná-los amplamente utilizados.”

Shackelford (HP Labs), 2000

6.1. Considerações Iniciais

O conceito de aplicações dinâmicas apresentado no capítulo anterior descarta qualquer possibilidade de utilizar-se ferramentas convencionais de projeto, dada a natureza dinâmica do sistema, tais aplicações também descartam os convencionais métodos de configuração e reconfiguração dos dispositivos.

Para conseguir-se a reconfiguração dinâmica ou mesmo parcial de dispositivos FPGAs, não é possível utilizar-se apenas de equipamentos como as placas, cabos e *softwares* convencionais de reconfiguração, atualmente fornecidos pelos fabricantes dos dispositivos. Para a reconfiguração dinâmica é necessário que o próprio sistema que venha a utilizar o dispositivo, seja capaz de configurá-lo e reconfigurá-lo sob o controle de algum *hardware/software* do próprio sistema, principalmente se for um sistema multi-FPGAs.

Este trabalho não tinha como objetivo nenhuma implementação, dado a vasta tarefa de estudar os diversos dispositivos FPGAs do mercado, suas técnicas e recursos de reconfigurabilidade dinâmica e ainda projetar um controlador de reconfiguração.

Porém, motivado pelas descobertas e perspectivas de tal tecnologia, foram utilizados alguns dos recursos do Laboratório de Computação Reconfigurável – LCR para implementar os seguintes elementos:

- uma interface em *hardware* para configuração e reconfiguração de sistemas multi-FPGAs;

- um *driver* para PCs que controla a interface em hardware;
- um *software* de controle local e remoto do sistema de reconfiguração; e
- uma placa de prototipação multi-FPGAs.

Na placa de prototipação multi-FPGAs foram utilizados dispositivos XC4025E, por serem, dentre os dispositivos disponíveis no LCR, os que mais se adequaram às necessidades. Embora tais dispositivos não sejam dinamicamente reconfiguráveis, seus recursos de *readback*, reconfigurabilidade parcial e divisão da *bitstream* em *frames*, muito se assemelham com as características dos dispositivos Virtex, que são reconfiguráveis dinamicamente.

Como não era prevista nenhuma implementação, não havia recursos financeiros, nem tempo previstos para a aquisição de dispositivos dinamicamente reconfiguráveis. Os únicos dispositivos dinamicamente reconfiguráveis até então existentes no LCR eram apenas dois dispositivos XC6216, que mesmo assim eram componentes de duas placas H.O.T. Works da VCC (*Virtual Computer Corporation*) [73], sendo que a necessidade era de dispositivos avulsos para o início da construção completa de um sistema.

Vale lembrar que um sistema pode ser reconfigurável dinamicamente mesmo que seus FPGAs individualmente não suportem tal recurso. Um sistema pode ser reconfigurável dinamicamente se permitir que dispositivos possam se reconfigurados independentemente, sem prejudicar a operação de todo o sistema. Deste modo, um sistema deste terá granularidade muito grossa.

6.2. Sistema de Reconfiguração

A figura 6.1 ilustra a concepção geral do sistema projetado e implementado. Os elementos de *hardware* do sistema foram implementados em um *protoboard* para *wire-wrap*, figura 6.2, onde foi montada a interface de reconfiguração com um barramento de controle. Neste barramento foram interligados 4 FPGAs XC4025E. Adicionalmente foi instalado um *driver* de tensão para uma interface RS-232, sendo que a lógica para a comunicação serial foi implementada nas FPGAs.

Os diagramas esquemáticos com detalhes das interligações e componentes

utilizados são apresentados no apêndice B. No apêndice C são encontradas tabelas com a descrição detalhada dos pinos e endereços de I/O da porta paralela e da interface de reconfiguração. Estas informações foram fundamentais para a implementação do *Driver* do sistema.

FIGURA 6.1: DIAGRAMA DO SISTEMA IMPLEMENTADO.

FIGURA 6.2: FOTOS DA PLACA CONFECCIONADA EM WIRE-WRAP.

Para efetuar o controle de baixo nível do *hardware* foi escrito um *driver* nas linguagens de programação Pascal e Assembler implementando as seguintes funções:

- `HardwarePresente()`: esta função recebe o endereço da porta paralela e retorna *true* caso a interface de reconfiguração esteja nesta porta;
- `HardwareLigado()`: recebe o endereço da interface de reconfiguração e retorna *true* caso ela esteja ligada;
- `ResetTodos()`: coloca todos FPGAs do sistema em modo de configuração;
- `ResetDispositivo()`: recebe um inteiro colocando o dispositivo desejado em modo de configuração;
- `InformacoesBitstream()`: recebe um arquivo de *bitstream* e retorna o tipo de dispositivo, esta função atua como um filtro para aceitar apenas *bitstreams* compatíveis com os FPGAs do sistema; e
- `ConfiguraDispositivo()`: recebe o arquivo da *bitstream* e envia serialmente os *bits* de configuração para o barramento, terminado o envio é verificado se o dispositivo entrou no modo de operação.

Estas funções do *Driver* são utilizadas pelo Controlador desenvolvido em Delphi. Este Controlador além de operar como um *front-end* para o controle local do *hardware*, também atua como uma camada intermediária de *software* entre o *Driver* e o usuário remoto.

O servidor de reconfiguração, além de hospedar o *Driver* e o Controlador, também hospeda um servidor de HTTP, neste caso foi utilizado o Apache 2.0. O servidor Apache hospeda um *site* que redireciona parte do seu código para uma porta TCP/IP no Controlador. Esta porta recebe e envia textos HTML, propositalmente gerados pelo Controlador para serem visualizados e controlados por um navegador comum de internet que esteja no *host* remoto. Neste *site* também foi implementada em PHP uma função para o *upload* de uma *bitstream* remota para o servidor de reconfiguração, que por sua vez pode utilizar qualquer *bitstream* local para configurar ou reconfigurar um FPGA do sistema.

Outra possibilidade de utilização remota do sistema é a de uma aplicação, em algum *host*, utilizar diretamente os textos HTML gerados e recebidos pelo Controlador como algum tipo de protocolo de comunicação. Os códigos fontes do *Driver*, do Controlador e do *site* são apresentados no apêndice D.

6.3. Resultados e Considerações Finais

Com estas implementações obteve-se um sistema capaz de utilizar FPGAs no seu contexto puro como *chip*. Uma vez desenvolvido o projeto do *hardware* e sintetizado-o em uma *bitstream*, este sistema permite a utilização dos *chips* sem depender de outros elementos fornecidos pelos fabricantes, como: placas; gravadores; cabos; *softwares*; *chips* de configuração; elementos de armazenamento de *bitstreams* e etc. Tais aparatos são muito convenientes em aplicações em geral, porém quando se almeja um sistema de computação reconfigurável dinamicamente é preciso obter maior controle do sistema.

Adicionalmente, os recursos de reconfiguração remota implementados permitem que de um simples navegador de internet um usuário possa fazer o *upload* do seu projeto. Este projeto, sintetizado em uma *bitstream*, pode ser carregado em qualquer uma das FPGAs disponíveis no sistema.

Caso algum fabricante de *hardware* baseado em FPGAs utilize-se desse sistema, ou algum semelhante, será possível o *upgrade* remoto de seus produtos, como discutido no capítulo V quando foi apresentado o conceito de IRL. É claro que a parte em *software* deste sistema, implementada no servidor de reconfiguração, poderia ser implementada em *hardware* dedicado, ou ser executada em um processador embutido no equipamento ou no próprio FPGA.

Como resultado final deste experimento, a tabela 6.1 mostra os tempos médios gastos para a reconfiguração dos dispositivos XC4025E. Para a coleta dos tempos foram utilizadas três máquinas distintas, conforme é apresentado na tabela seguinte.

Cabe observar, que estes tempos dependem apenas da capacidade do *host* em gerar pulsos de *clock* para o sistema e enviar os dados da *bitstream*, visto que os dispositivos XC4025E suportam tempos bem menores que estes. No modo em que os

dispositivos XC4025E estão sendo reconfigurados (serial *slave*), eles poderiam ser carregados a uma frequência máxima de 10MHz [35]. Como as *bitstreams* do FPGA XC4025E possuem 422.176 *bits*, estes dispositivos poderiam ser reconfigurados totalmente ao tempo mínimo de 42,2 ms, o que é cerca de 66 vezes mais rápido que o tempo médio obtido nas reconfigurações utilizando-se o Pentium III.

TABELA 6.1: TEMPOS MÉDIOS DE RECONFIGURAÇÃO DOS DISPOSITIVOS XC4025E.

Máquina	Freq. do Proc. (MHz)	Tempo (ms)
Pentium	233	7.578
Pentium II	266	5.670
Pentium III	866	2.773

Uma alternativa para melhorar estes tempos de reconfiguração é implementar na interface de reconfiguração um *buffer* capaz de armazenar toda a *bitstream*. Quando os dispositivos XC4025E são colocados no modo de configuração eles suspendem suas operações, voltando a estar ativos somente ao término da reconfiguração. A utilização de um *buffer* e um *clock* dedicado na interface de reconfiguração poderá permitir que o *host* envie todos os dados da *bitstream* para o *buffer*, e só inicie a reconfiguração quando todos os *bits* de configuração estiverem no *buffer*. Uma vez iniciado a reconfiguração, a transferência dos dados do *buffer* para o FPGA poderá ser realizada pelo *hardware* da interface, na máxima frequência suportada pelo dispositivo. Assim, será possível atingir tempos mínimos de reconfiguração, que se dependa da capacidade de processamento do *host*.

Conclusões

"Ou nós encontramos um caminho, ou abrimos um."

Aníbal

7.1. Considerações deste Trabalho

Este trabalho teve inicialmente o objetivo de investigar os recursos e aplicações da reconfigurabilidade dinâmica de dispositivos FPGAs. Nas pesquisas do LCR muito se discutia sobre tais recursos, porém pouco se conhecia das reais aplicações e dificuldades que tal área da computação reconfigurável oferecia. Adicionalmente havia especulações sobre a utilização de tal tecnologia no projeto ARMOSH. Porém, mais uma vez se deparava com o abismo entre o *marketing* dos fabricantes e o *know-how* do LCR em utilizar tal tecnologia.

A implementação do sistema de reconfiguração, apresentado no capítulo VI, não era prevista neste trabalho. Porém, este sistema muito pode vir a contribuir em trabalhos futuros de reconfiguração dinâmica de FPGAs, assim como na construção da arquitetura de *hardware* do projeto ARMOSH.

7.2. Contribuições deste Trabalho

Os estudos realizados neste trabalho revelou a atual condição das arquiteturas de FPGAs e das ferramentas de desenvolvimento em implementar plataformas de *hardware* dinâmico, virtual, evolutivo, etc. Embora sejam poucos os fabricantes de FPGAs que oferecem recursos de reconfigurabilidade dinâmica em seus dispositivos, ficou claro que a grande limitação em obter-se *hardware* virtual está por conta das ferramentas para tal fim. O próprio fato da granularidade, discutido no capítulo IV, depende muito mais da capacidade das ferramentas de criar uma abstração em

elementos de mais alto nível, do que dos dispositivos. Como foi visto os FPGAs podem apresentar até mesmo um único *bit* como elemento atômico reconfigurável.

Entre os FPGAs reconfiguráveis dinamicamente presentes no mercado, os dispositivos do fabricante Xilinx melhor se enquadram às necessidades das pesquisas desenvolvidas pelo LCR, conseqüentemente ao ARMOSH.

Como visto no capítulo IV, os fabricantes Atmel e Lattice também oferecem FPGAs reconfiguráveis dinamicamente, sendo que os únicos a fornecerem completa solução SoC em FPGAs são os fabricantes Xilinx e Altera. Como também foi apresentado, o fabricante Altera não oferece dispositivos reconfiguráveis dinamicamente. Vale observar que o fabricante Atmel possui completa solução SoC, porém não é baseada em FPGAs. No capítulo III foi visto que o FPGA é apenas um elemento do sistema SoC do fabricante Atmel.

Quanto aos FPGAs da Lattice, estes não oferecem muita credibilidade para dedicar-se esforços de pesquisas sobre eles. Os dispositivos da família ORCA a alguns anos oferecem a mesma arquitetura e capacidade lógica. Antes estes dispositivos eram produzidos pela Lucent Technologies e desde janeiro deste ano passaram a ser fabricados pela Lattice Semiconductor. Neste período, nenhum investimento ou melhoria foi apresentado nesta família, não sendo assim uma boa opção contar com estes dispositivos para o futuro.

O sistema implementado pode, mesmo com suas atuais limitações, ser utilizado para inúmeras pesquisas que necessitem de emulação em *hardware*. Muitas das atuais pesquisas do LCR utilizam plataformas de *hardware* fornecidas pelos fabricantes, o que oferece pouca flexibilidade aos projetos. A continuidade do desenvolvimento deste sistema pode resultar em uma arquitetura bem mais completa e flexível que as atualmente utilizadas no LCR. Algumas sugestões para desenvolvimento futuro são apresentadas a seguir.

7.3. Sugestões para Desenvolvimento Futuro

Para o sistema de reconfiguração implementado não foi definido nenhum barramento ou topologia para os pinos de dados, até então os objetivos focaram apenas

a reconfiguração do sistema e não uma aplicação específica. Porém, assim como em qualquer sistema, é muito importante que as operações de entrada e saída de dados sejam bem definidas. Uma sugestão de desenvolvimento futuro é a implementação de um barramento de alto desempenho compatível com PCs. Assim, será possível utilizar este sistema como um elemento de *hardware* genérico em plataformas de computação reconfigurável, onde o *host* poderá ser um PC qualquer.

Uma vez definido um canal de comunicação de dados entre o sistema e um *host*, bastará substituir ou adicionar FPGAs Virtex no barramento de controle, para que este sistema forneça elementos reconfiguráveis dinamicamente. O barramento de controle utiliza o modo serial *slave* para configurar os dispositivos XC4025E. Este modo de configuração é compatível com os atuais dispositivos da família Virtex, tornando muito fácil adicionar tais dispositivos nesse barramento.

Antes de tentar realizar reconfigurações dinâmicas de dispositivos Virtex será necessário um estudo detalhado da arquitetura e formato das *bitstreams* destes FPGAs. Uma vez dominado a estrutura das *bitstreams* e os mecanismos de *readback* e configuração por *frames*, será possível realizar reconfigurações parciais até mesmo dos atuais dispositivos XC4025E. Dominado estes elementos e a reconfiguração parcial, restará um pequeno passo para o domínio da reconfigurabilidade dinâmica destes FPGAs. Bastará não efetuar o *reset* e/ou a inicialização dos dispositivos, pois estes devem permanecer ativos durante a reconfiguração. Entretanto, neste instante estará dominado apenas as técnicas de utilização dos recursos de reconfigurabilidade dinâmica dos FPGAs. Porém, vale lembrar que as maiores dificuldades para a utilização desta tecnologia estão nas limitações das ferramentas de desenvolvimento para tal fim.

Uma vez dominado as técnicas de reconfigurabilidade dinâmica e tendo disponível uma plataforma de *hardware* que suporte tais recursos, será possível partir para pesquisas com ferramentas e aplicações dinâmicas. Uma alternativa é utilizar as classes JBits, para isso será necessário a construção de uma interface XHWIF para a plataforma de *hardware* disponível. Implementado a interface XHWIF para o *hardware* em questão, todo o sistema JBits estará disponível para esta plataforma.

Outra alternativa para chegar a este ponto é adquirir uma das quatro placas apresentados no capítulo V que são suportadas de forma nativa pela API XHWIF. Entretanto, voltaremos a questão discutida anteriormente de utilizar sistemas prontos,

pouco flexíveis e limitados. A necessidade do LCR é de uma arquitetura flexível que possa ser remodelada para uma dada aplicação específica, como por exemplo, a construção de uma arquitetura para o ARMOSH.

A utilização da ferramenta JBits é uma possibilidade, porém existem outras alternativas como o emprego da ferramenta JHDL ou mesmo a implementação de uma nova ferramenta. Em qualquer caso será necessário que seja implementado um *driver* que descreva detalhadamente todo o sistema em *hardware* para as aplicações no *host*, assim como faz a interface XHWIF para a ferramenta JBits.

Desenvolvidos os pontos até agora levantados, o que não é nada trivial, será possível concentrar esforços de pesquisa em ferramentas e aplicações dinâmicas para a computação reconfigurável.

Referências Bibliográficas

- [1] LYSAGHT, P. & DUNLOP, J.; *Dynamic Reconfiguration of Field Programmable Gate Arrays*; in Proceedings of the 3rd International Workshop on Field Programmable Logic and Applications (FPL'93), Oxford, UK, Abingdon EE&CS Books, p. 82-94, Set. 1993;
- [2] SIDHU, R. et al.; *A Self-Reconfigurable Gate Array Architecture*; in Proceedings of the 10th International Workshop on Field Programmable Logic and Applications (FPL'2000), Villach, Austria, Springer (LNCS 1896), p. 106-120, Set. 2000;
- [3] SIMMLER, H. & LEVINSON, L. & Männer, R.; *Multitasking on FPGA Coprocessors*; in Proceedings of the 10th International Workshop on Field Programmable Logic and Applications (FPL'2000), Villach, Austria, Springer (LNCS 1896), p. 121-130, Set. 2000;
- [4] FALLSIDE, H. & SMITH, M. J. S.; *Internet Connected FPL*; in Proceedings of the 10th International Workshop on Field Programmable Logic and Applications (FPL'2000), Villach, Austria, Springer (LNCS 1896), p. 48-57, Set. 2000;
- [5] ROBINSON, D. & LYSAGHT, P.; *Verification of Dynamically Reconfigurable Logic*; in Proceedings of the 10th International Workshop on Field Programmable Logic and Applications (FPL'2000), Villach, Austria, Springer (LNCS 1896), p. 141-150, Set. 2000;
- [6] HADZIC, I. & UDANI, S. & SMITH, J.; *FPGA Viruses*; in Proceedings of the 9th International Workshop on Field Programmable Logic and Applications (FPL'99), Glasgow, Scotland, Springer (LNCS 1673), p. 291-300, Set. 1999;
- [7] DEHON, A. & WAWRZYNEK, J.; *Reconfigurable Computing: What, Why, and Design Automation*; in Proceedings of the 36th ACM IEEE Design Automation Conference (DAC'99), New Orleans, ACM Press, p. 610-615, Jun. 1999;
- [8] GOLDBERG, I. & WAGNER, D.; *Architectural Considerations for Cryptanalytic Hardware*; disponível em <http://www.cs.berkeley.edu/~iang/isaac/hardware/paper.ps>, University of California, Berkeley, 1996;
- [9] WOLF, Denis F.; *Projeto de Uma Rede Neural Usada no Reconhecimento de Gestos por Robôs Móveis*; Dissertação de mestrado, ICMC – USP, 96 p., Set. 2001;
- [10] LUDWIG, S. & SLOUS, R. & SINGH, S.; *Implementing PhotoShop^{TM} Filters in Virtex^{TM}*; in Proceedings of the 9th International Workshop on Field

- Programmable Logic and Applications (FPL'99), Glasgow, Scotland, Springer (LNCS 1673), p. 233-242, Set.1999;
- [11] MACVICAR, D. & PATTERSON, J. W. & SINGH, S.; *Rendering PostScript^{TM} Fonts on FPGAs*; in Proceedings of the 9th International Workshop on Field Programmable Logic and Applications (FPL'99), Glasgow, Scotland, Springer (LNCS 1673), p. 223-232, Set.1999;
- [12] Star Bridge Systems Inc.; *Overview of SBSs Reconfigurable Computing Technology*; disponível em <http://www.starbridgesystems.com/tech-over.html>, Midvale, Utah, 2001;
- [13] VILLASENOR, J. & MANGIONE-SMITH, W. H.; *Configurable Computing*; in Scientific American Magazine, New York, p. 66-71, Jun. 1997;
- [14] BROWN, S. & ROSE, J. & FRANCIS, R. & VRANESIC, Z.; *Field Programmable Gate Arrays*; Kluwer Academic Publishers, 224 p., Jun. 1992;
- [15] CHAN, P. K. & MOURAD, S.; *Digital Design using Field Programmable Gate Arrays*; Prentice Hall, 256 p., Abr. 1994;
- [16] RAZDAN, R. & BRACE, K. & SMITH, M. D.; *PRISC Software Acceleration Techniques*; in Proceedings of the IEEE International Conference on Computer Design (ICCD'94), Cambridge, MA, IEEE Computer Society Press, p. 145-149, Out. 1994;
- [17] WIRTHLIN, M. J. & HUTCHINGS, B. L.; *A Dynamic Instruction Set Computer*; in Proceedings of the 3rd IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'95), Napa Valley, California, IEEE Computer Society Press, p. 99-107, Abr. 1995;
- [18] HAUSER, J. R. & WAWRZYNEK, J.; *Garp: A MIPS Processor with a Reconfigurable Coprocessor*; in Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97), Napa Valley, California, IEEE Computer Society Press, p. 24-33, Abr. 1997;
- [19] YE, A. & SHENOY, N. & BANERJEE, P.; *A C Compiler for a Processor with a Reconfigurable Functional Unit*; in Proceedings of the 8th ACM Symposium on Field Programmable Gate Arrays, Monterey, California, disponível em <http://www.ece.nwu.edu/cpdc/Match/Pubs/fpga2000.ye.pdf>, Fev. 2000;
- [20] WAINGOLD, E. et al.; *Baring it all to Software: The Raw Machine*; MIT Laboratory for Computer Science, TR-709, Cambridge, MA, <ftp://ftp.cag.lcs.mit.edu/pub/raw/documents/Waingold:LCS-TR:1997.pdf>, 28 p., Mar. 1997;
- [21] BABB, J. et al.; *The RAW Benchmark Suite: Computation Structures for General Purpose Computing*; in Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97), Napa Valley, California, IEEE Computer Society Press, Abr. 1997;

- [22] CARDOSO, João M. P. & VÉSTIAS, Mário P.; *Architectures and Compilers to Support Reconfigurable Computing*; ACM Crossroads Student Magazine, disponível em <http://acm.org/crossroads/xrds5-3/rcconcept.html>, 1999;
- [23] LEVINSON, L. et al.; *Preemptive Multitasking on FPGAs*; in Proceedings of the 8th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00), Napa Valley, California, IEEE Computer Society Press, p. 301-302, Abr. 2000;
- [24] MERINO, P. & LÓPES, J. C. & JACOME, M.; *A Hardware Operating System for Dynamic Reconfiguration of FPGAs*; in Proceedings of the 8th International Workshop on Field Programmable Logic and Applications (FPL'98), Tallin, Estonia, Springer (LNCS 1482), p. 431-435, Set. 1998;
- [25] FARIA, Gedson; *Explorando o Potencial de Algoritmos de Aprendizado com Reforço em Robôs Móveis*; Dissertação de mestrado, ICMC – USP, 68 p., Abr. 2000;
- [26] BROWN, S. & ROSE, J.; *Architecture of FPGA and CPLDs: A Tutorial*; IEEE Design and Test of Computers, v. 13, n° 2, p. 42-57, 1996;
- [27] Actel Corp.; *FPGA Data Book and Design Guide*; 1995;
- [28] ROMERO, Roseli A.; *Aprendizado em Robôs Móveis Via Software e Hardware – ARMOSH*; Projeto de Pesquisa, ICMC – USP, Mar. 2000;
- [29] SALCIC, Z. & SMAILAGIC, A.; *Digital Systems Design and Prototyping: Using Field Programmable Logic and Hardware Description Languages*; Kluwer Academic Publishers, Second Edition, 621 p., 2000;
- [30] WAINGOLD, E. et al.; *Baring it all to software: Raw Machines*; IEEE Computer, New York, v. 30, n° 9, p.86–93, Set. 1997;
- [31] CALLAHAN, T. J. & HAUSER, J. R. & WAWRZINEK, J.; *The Garp architecture and C compiler*; IEEE Computer, v. 33, n° 4, p.62–69, Abr. 2000;
- [32] SIDA; *FIPSOC Mixed Signal System-on-Chip*; disponível em <http://www.sidsa.com/FIPSOC/fipsoc1.pdf>; 2002;
- [33] Atmel; *AT94K Series Field Programmable System Level Integrated Circuit*; disponível em <http://www.atmel.com/atmel/acrobat/doc1138.pdf>; Jun. 2002;
- [34] Xilinx Inc.; *Virtex-II 1.5V Field-Programmable Gate Arrays*; disponível em <http://www.xilinx.com/partinfo/ds031.pdf>; Out. 2001;
- [35] Xilinx Inc.; *XC4000E and XC4000X Series Field Programmable Gate Arrays*; disponível em <http://www.xilinx.com/partinfo/4000.pdf>; Mai. 1999;
- [36] Xilinx Inc.; *XC6200 Field Programmable Gate Array*; disponível em http://dec.bournemouth.ac.uk/drhwh_lib/archive/xc6200.pdf; Jan. 1997;

- [37] Xilinx Inc.; *Virtex 2.5V Field Programmable Gate Array*; disponível em <http://www.xilinx.com/partinfo/ds003.pdf>; Abr. 2001;
- [38] Xilinx Inc.; *Virtex 2.5V Field Programmable Gate Array*; disponível em <http://www.xilinx.com/partinfo/ds003.pdf>; Abr. 2001;
- [39] Xilinx Inc.; *Virtex II 1.5V Field Programmable Gate Array*; disponível em <http://www.xilinx.com/partinfo/ds031.pdf>; Out. 2001;
- [40] Xilinx Inc.; *Virtex II Pro Platform FPGAs: Introduction and Overview*; disponível em <http://www.xilinx.com/partinfo/ds083.pdf>; Jun. 2002;
- [41] Altera Corp.; *FLEX 8000 Programmable Logic Device Family*; disponível em <http://www.altera.com/literature/ds/dsf8k.pdf>; Jun. 1999;
- [42] Altera Corp.; *FLEX 10K Embedded Programmable Logic Family*; disponível em <http://www.altera.com/literature/ds/dsf10k.pdf>; Mar. 2001;
- [43] Altera Corp.; *APEX 20K Programmable Logic Device Family*; disponível em <http://www.altera.com/literature/ds/apex.pdf>; Fev. 2002;
- [44] Altera Corp.; *Excalibur Backgrounder White Paper*; disponível em http://www.altera.com/literature/wp/excal_bkgrnd.pdf; Jun. 2000;
- [45] Altera Corp.; *APEX II Programmable Logic Device Family*; disponível em http://www.altera.com/literature/ds/ds_ap2.pdf; Mai. 2002;
- [46] Altera Corp.; *Stratix Programmable Logic Device Family*; disponível em http://www.altera.com/literature/ds/ds_atx.pdf; Abr. 2002;
- [47] Atmel Corp.; *Coprocessor Field Programmable Gate Arrays - AT6000(VL) Series*; disponível em <http://www.atmel.com/atmel/acrobat/doc0264.pdf>; 1999;
- [48] Atmel Corp.; *AT40K FPGAs with FreeRAM*; disponível em <http://www.atmel.com/atmel/acrobat/doc0896.pdf>; 1999;
- [49] Lattice Semiconductor Corp.; *ORCA Series 4 FPGAs*; disponível em <http://www.latticesemi.com/lit/docs/datasheets/fpga/ds01-174ncip.pdf>; Abr. 2002;
- [50] SMITH, M. J. S.; *Aplication-Specific Integrated Circuits*; disponível em <http://www.dacafe.com/DACafe/EDATools/EDAbooks/ASIC/Book/>; Jun. 1997;
- [51] BREBNER, G. & DIESSEL, O.; *Chip-Based Reconfigurable Task Management*; in Proceedings of the 11th International Workshop on Field Programmable Logic and Applications (FPL'2001), Belfast, Northern Ireland, Springer (LNCS 2147), p. 182-191, Ago. 2001;
- [52] MACBETH, J. & LYSAGHT, P.; *Dynamically Reconfigurable Cores*; in Proceedings of the 11th International Workshop on Field Programmable Logic

- and Applications (FPL'2001), Belfast, Northern Ireland, Springer (LNCS 2147), p. 462-472, Ago. 2001;
- [53] MESQUITA, Daniel G.; *Contribuições para Reconfiguração Parcial, Remota e Dinâmica de FPGAs*; Dissertação de Mestrado, PUC – RS, p. 103, Mar. 2002;
- [54] Actel, Corp.; *The Antifuse Advantage in FPGAs – Single Chip High Performance Solutions*; disponível em <http://www.actel.com/products/antifuse.html>; 2001;
- [55] DEHON, A.; *DPGA – Coupled Microprocessors: Commodity ICs for the Early 21st Century*; in Proceedings of the 2nd IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'94), Napa Valley, California, IEEE Computer Society Press, p. 31-39, Abr. 1994;
- [56] Atmel Corp.; *Implementing Cache Logic with FPGAs*; disponível em <http://www.atmel.com/atmel/acrobat/doc0461.pdf>; 1999;
- [57] National Semiconductor; *Navigator*; disponível em <http://www.national.com/appinfo/milaero/files/f61.pdf>; 1998;
- [58] Algotronix; *Cal1024 datasheet*; disponível em http://dec.bournemouth.ac.uk/drhwh_lib/archive/cal1024.ps.gz; 1991;
- [59] ATHANAS, P. M. & SILVERMAN, H. F.; *Processor Reconfiguration through Instruction-Set Metamorphosis*; IEEE Computer, v. 26, n° 3 p.11-18, Mar. 1993;
- [60] GOKHALE, M. & et. al.; *SPLASH: A Reconfigurable Linear Logic Array*; in Proceedings of the International Conference on Parallel Processing (ICPP'90), v. 9, p. 219-314, Ago. 1990;
- [61] LEWIS, D. M. et. al.; *The Transmogripher-2: A 1 Million Gate Rapid Prototyping System*; IEEE Transactions on VLSI Systems, p.188-198, Jun. 1998;
- [62] DUBOIS, M. et. al.; *Rapid hardware prototyping on RPM-2*; IEEE Design and Test of Computers, v. 15, n° 3, p. 112-118, Set. 1998;
- [63] SANCHEZ, E. et. al.; *Static and Dinamic Configurable Systems*; IEEE Transactions on Computers, p. 556-564, 1999;
- [64] ARM; *AMBA 2.0 Specification*; disponível em http://www.arm.com/armtech/AMBA_Spec?OpenDocument; Mai. 1999;
- [65] HAUCK, Scott; *The Roles of FPGAs in Reprogrammable Systems*; IEEE Computer, v. 86, n° 4, p. 615-639, New York, Abr. 1998;
- [66] Xilinx Inc.; *JBits SDK*; disponível em <http://www.xilinx.com/products/jbits/>; 2002;

- [67] LYSAGHT, Patrick & STOCKWOOD, J; *A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays*; IEEE Transactions on VLSI Systems, p. 381-390, Set. 1996;
- [68] JHDL; *BYU JHDL, Open Source FPGA CAD Tools*; Brigham Young University; disponível em <http://www.jhdl.org>; 2002;
- [69] BELLOWS, P. & HUTCHINGS, B.; *JHDL - An HDL for Reconfigurable Systems*; in Proceedings of the 6th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'98), Napa Valley, California, IEEE Computer Society Press, p. 14-17, Abr. 1998;
- [70] Xilinx; *The Xilinx Lava HDL*; disponível em <http://www.xilinx.com/labs/lava/>; 2002;
- [71] SINGH, Satnam & JAMES-ROXBY, Phil; *Lava and JBits: From HDL to Bitstream in Seconds*; in Proceedings of the 9th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01), Napa Valley, California, IEEE Computer Society Press, 10 p., Abr. 2001;
- [72] Xilinx; *Architecting Systems for Upgradability with IRL (Internet Reconfigurable Logic)*; disponível em www.xilinx.com/xapp/xapp412.pdf; Jun. 2001;
- [73] VCC; *HOT Works Development System based upon the XC62000 RPU*; disponível em <http://www.vcc.com/Hotworks.html>; 2002;
- [74] IBM; *IBM, Xilinx announce multi-million dollar manufacturing deal*; disponível em http://www-3.ibm.com/chips/news/2002/0304_xilinx.html; Mar. 2002;
- [75] Altera Corp.; *Excalibur Device Overview*; disponível em http://www.altera.com/literature/ds/ds_arm.pdf; Mai. 2002;
- [76] CARDOSO, João M. P.; *Compilação de Algoritmos em Java^{TM} para Sistemas Computacionais Reconfiguráveis com Exploração do Paralelismo ao Nível das Operações*; Tese de Doutorado, Universidade Técnica de Lisboa, Instituto Superior Técnico, 2000;
- [77] GONÇALVES, Richard A. et al.; *Architect: Um Sistema de Computação Reconfigurável*; in Proceedings of the Workshop de Computação Reconfigurável (CORE'00), p. 42-45, Marília, Ago. 2000;
- [78] SINGH, S. & JAMES-ROXBY, P.; *Rapid Construction of Partial Configuration Datastreams from High-Level Constructs Using JBits*; in Proceedings of the 11th International Workshop on Field Programmable Logic and Applications (FPL'2001), Belfast, Northern Ireland, Springer (LNCS 2147), p. 346-356, Ago. 2001;

Modelo de Programação JBits

Os vários passos do fluxo de aplicações JBits incluem: criação de um objeto JBits; leitura de uma *bitstream* para o objeto JBits; modificação da *bitstream* com os dados de projeto; e escrita do projeto na *bitstream* de saída. Estes passos são discutidos abaixo. Como já citado, a *bitstream* de um projeto existente pode ser utilizada, podendo a aplicação JBits modificar este projeto, ou uma *bitstream* nula pode ser adotada para que seja criado um novo projeto pela aplicação JBits. O projeto pode ser verificado usando o Boardscope, e então carregado para o dispositivo Virtex usando a API XHWIF.

A.1. O Construtor

Um objeto JBits deve ser construído antes que qualquer coisa seja feita. O Construtor é muito simples e necessita de apenas um parâmetro, o tipo do dispositivo. A lista de dispositivos pode ser encontrada na classe *Devices*. Este construtor produz o modelo do dispositivo para o tipo selecionado e inicia várias funções. A sintaxe para a chamada ao Construtor é apresentada na figura A.1.

```
JBits(int deviceType);  
  
Por exemplo,  
  
JBits jbits = new JBits(Devices.XCV100);
```

FIGURA A.1: SINTAXE PARA A CHAMADA AO MÉTODO CONSTRUTOR E EXEMPLO.

O código exemplificado na figura A.1 constrói o modelo para o dispositivo Virtex XCV100.

A.2. O Método *Read Bitstream*

O método *Read Bitstream* necessita também de apenas um parâmetro, uma *string* contendo o nome do arquivo contendo a *bitstream* para ser lida. Este método carrega a *bitstream* no objeto JBits construído e mapeia os dados da *bitstream* no modelo criado do dispositivo. Uma vez carregada a *bitstream*, os dados de configuração, na forma de *bits*, podem ser lidos e escritos. Notem que várias configurações e exceções de entrada e saída podem ocorrer neste método. A sintaxe para a chamada ao método *read()* é apresentada na figura A.2.

```
void JBits.read(String inFileName);
```

Por exemplo,

```
jbits.read("infile.bit");
```

FIGURA A.2: SINTAXE PARA A CHAMADA AO MÉTODO READ() E EXEMPLO.

O código exemplificado na figura A.2 realiza a leitura da *bitstream* contida no arquivo “infile.bit”.

A.3. O Método *Set Bits*

O método *Set Bits* escreve os dados de configuração, de um determinado recurso, em um elemento de configuração. O recurso é identificado por uma linha e uma coluna da CLB. O recurso selecionado na CLB é então identificado por uma constante. Estas constantes são definidas nas classes Java, contendo os objetos configuráveis. Por exemplo, selecionando a configuração do recurso *SLICE0 F1* a entrada é realizada usando a constante *SOF1* na classe *SOF1*, isto é *SOF1.SOF1*. Um *array* de inteiros que provêm os *bits* de configuração é passado como parâmetro final para este método. Assim como o recurso, este dado quase sempre é uma constante pré-definida. Por exemplo, para selecionar a entrada *SOF1.SOF1* para o valor da saída do *SLICE0 X*, a constante usada é *SOF1.SO_X*. Exceções podem ocorrer se dados inválidos forem passados como parâmetros, ou se o dados a serem configurados não forem válidos. Enfim, este método é utilizado para identificar a CLB e o recurso associado com ela e

para especificar o valor constante pré-definido aplicável para aquele recurso. A sintaxe para a chamada ao método *set()* é apresentada na figura A.3.

```
void JBits.set(int row, int column, int[][] resource,  
int[] bits);  
  
Por exemplo,  
  
jbits.set(clbRow, clbCol, S0F1.S0F1, S0F1.S1_X);
```

FIGURA A.3: SINTAXE PARA A CHAMADA AO MÉTODO *SET()* E EXEMPLO.

Devido aos recursos e aos valores dos *bits* serem associados com a arquitetura Virtex, apenas constantes pré-definidas e valores definidos nas respectivas classes Java, devem ser utilizados na chamada *set()*.

A.4. O Método *Write Bitstream*

Similar ao método *Read Bitstream*, o método *Write Bitstream* também necessita de apenas um parâmetro, uma *string* contendo o nome do arquivo a ser escrita a *bitstream*. Este método escreve a *bitstream* do objeto JBits construído em um arquivo. Várias configurações e exceções de entrada e saída também podem ocorrer neste método. A sintaxe para a chamada ao método *write()* é apresentada na figura A.4.

```
int JBits.write(String outFileName);  
  
Por exemplo,  
  
jbits.write("outfile.bit");
```

FIGURA A.4: SINTAXE PARA A CHAMADA AO MÉTODO *WRITE()* E EXEMPLO.

O código exemplificado na figura A.4 escreve os dados de configuração modificados, no arquivo de *bitstream* “outfile.bit”.

A.5. O Método *Get Bits*

O método *Get Bits* é usado para ler a configuração de um determinado recurso em um elemento configurável. O recurso é identificado usando a mesma convenção mencionada no método *set()*. A maioria dos dados obtidos usando o método *get()*, podem ser interpretados e usados por outras partes de aplicações JBits. Este método também pode causar exceções se dados inválidos forem passados como parâmetros, ou se os dados a serem acessados não forem válidos. A sintaxe para a chamada ao método *get()* é apresentada na figura A.5.

```
int[] JBits.get(int row, int column, int[][] resource);

Por exemplo,

int[] theValue = jbits.get(clbRow, clbCol, S0F1.S0F1);
```

FIGURA A.5: SINTAXE PARA A CHAMADA AO MÉTODO *GET()* E EXEMPLO.

O código exemplificado na figura A.5 retorna o valor configurado para o recurso *S0F1.S0F1*, que é o valor do recurso *SLICE0 F1*.

A.6. Fluxo de Desenvolvimento

No caso de aplicações reconfiguráveis, os dados podem ser lidos repetidamente e escritos no *hardware* presente, ao invés de em um arquivo, sendo a estrutura geral essencialmente a mesma. As partes de códigos apresentados nos métodos acima, são apresentadas na figura A.6 para ilustrar uma simples aplicação JBits.

```
JBits jbits = new JBits(Devices.XCV100);

JBits.read("infile.bit");

JBits.set(5, 4, S0F1.S0F1, S0F1.S1_X);

JBits.write("outfile.bit");
```

FIGURA A.6: EXEMPLO DE UMA SIMPLES APLICAÇÃO JBITS.

O código exemplificado na figura A.6 configura a entrada *F1* do *SLICE0 F LUT*, localizado na CLB da linha 5 e coluna 4, para o *SLICE1 X* de saída.

Diagramas Esquemáticos

B.1. Interface de Reconfiguração Multi-FPGAs

FIGURA B.1: DIAGRAMA ESQUEMÁTICO DA INTERFACE DE RECONFIGURAÇÃO MULTI-FPGAs.

B.2. Interconexões dos FPGAs XC4025E

FIGURA B.2: DIAGRAMA ESQUEMÁTICO DE INTERCONEXÕES INDIVIDUAIS DOS FPGAs XC4025E.

Pinagem e Endereçamento

C.1. Pinagem Externa da Interface de Reconfiguração

TABELA C.1: PINAGEM EXTERNA DA INTERFACE DE RECONFIGURAÇÃO.

Pinos do DB25M (HW)	Nome	Descrição
1	A	<i>Bit</i> de entrada LSB do decoder
2	DIN	Entrada serial de dados
3	CLK	Clock de sincronização dos dados na borda de subida
4	TMS_IN	
5	CTRL	Habilita o <i>hardware</i> com nível lógico 0
6	PROG	
7	-	Não utilizado
8	D6	Ligado aos pinos 11 e 12
9	-	Não utilizado
10	-	Não utilizado
11	-	Ligado ao pino 6
12	-	Ligado ao pino 6
13	DONE	
14	B	<i>Bit</i> de entrada do decoder
15	VCC	Nível lógico 1 quando energizado o <i>hardware</i>
16	C	<i>Bit</i> de entrada do decoder
17	D	<i>Bit</i> de entrada MSB do decoder
18	-	Não utilizado
19	-	Não utilizado
20	GND	Referência de tensão
21	-	Não utilizado
22	-	Não utilizado
23	-	Não utilizado
24	-	Não utilizado
25	-	Ligado ao pino 20

C.2. Pinagem e Endereçamento da Porta Paralela de um PC

TABELA C.2: PINAGEM E ENDEREÇAMENTO DA PORTA PARALELA DE UM PC.

Pinos do DB25F (PC)	Endereço	Bits de D0 (LSB) até D7 (MSB)	Lógica
1	base + 2	D0	Invertida
2	base + 0	D0	Não-invertida
3	base + 0	D1	Não-invertida
4	base + 0	D2	Não-invertida
5	base + 0	D3	Não-invertida
6	base + 0	D4	não-invertida
7	base + 0	D5	não-invertida
8	base + 0	D6	não-invertida
9	base + 0	D7	não-invertida
10	base + 1	D6	não-invertida
11	base + 1	D7	Invertida
12	base + 1	D5	não-invertida
13	base + 1	D4	não-invertida
14	base + 2	D1	Invertida
15	base + 1	D3	não-invertida
16	base + 2	D2	não-invertida
17	base + 2	D3	Invertida
18	-	GND	-
19	-	GND	-
20	-	GND	-
21	-	GND	-
22	-	GND	-
23	-	GND	-
24	-	GND	-
25	-	GND	-

Códigos Fontes

D.1. Código do Driver

```
library RcnfgDrv;

uses
    SysUtils;

type
    TInformacoesBitstream = ^InfBit;
    InfBit = record
        Dispositivo: ShortString;
        Valida: Boolean;
    end;
    TConfiguraDispositivo = ^ConfDisp;
    ConfDisp = record
        Dispositivo: ShortString;
        Informacao: ShortString;
        Sucesso: Boolean;
        Tempo: ShortString;
    end;
    TShortString = ^ShortString;

function HardwarePresente(Base: Word): Boolean; stdcall;
var
    aux_Byte: Byte;
begin
    asm
        PUSH AX
        PUSH DX

        MOV DX, Base
        MOV AL, 12
        OUT DX, AL
        ADD DX, 1
        IN AL, DX
        AND AL, 168
        MOV aux_Byte, AL

        POP DX
        POP AX
    end;

    if aux_Byte = 136 then
        Result := True
    else
        Result := False;
    end;
end;

function HardwareLigado(Base: Word): Boolean; stdcall;
var
    aux_Byte: Byte;
begin
    asm
        PUSH AX
        PUSH DX

        MOV DX, Base
```

```

    MOV AL, 12
    OUT DX, AL
    ADD DX, 1
    IN AL, DX
    AND AL, 184
    MOV aux_Byte, AL

    POP DX
    POP AX
end;

if aux_Byte = 136 then
    Result := True
else
    Result := False;
end;

procedure ResetTodos(Base: Word); stdcall;
var
    aux_Byte: Byte;
begin
    asm
        PUSH AX
        PUSH DX

        MOV DX, Base
        MOV AL, 0
        OUT DX, AL

        POP DX
        POP AX
    end;

    for aux_Byte := 0 to 7 do
        begin
            asm
                PUSH AX
                PUSH DX

                MOV DX, Base
                ADD DX, 2
                MOV AL, aux_Byte
                OUT DX, AL

                POP DX
                POP AX
            end;
        end;

    asm
        PUSH AX
        PUSH DX

        ADD DX, -2
        MOV AL, 28
        OUT DX, AL

        POP DX
        POP AX
    end;
end;

function ResetDispositivo(Base: Word; Dispositivo: Byte): Boolean; stdcall;
var
    aux_Byte: Byte;
begin
    case Dispositivo of
        0: aux_Byte := 11;
        1: aux_Byte := 10;
        2: aux_Byte := 9;
        3: aux_Byte := 8;
        4: aux_Byte := 15;
        5: aux_Byte := 14;
        6: aux_Byte := 13;
        7: aux_Byte := 12;
    end;
end;

```

```

8: aux_Byte := 3;
9: aux_Byte := 2;
10: aux_Byte := 1;
11: aux_Byte := 0;
12: aux_Byte := 7;
13: aux_Byte := 6;
14: aux_Byte := 5;
15: aux_Byte := 4;
else
  aux_Byte := 16;
end;

if aux_Byte <> 16 then
begin
  asm
    PUSH AX
    PUSH DX

    MOV DX, Base
    ADD DX, 2
    MOV AL, aux_Byte
    OUT DX, AL //Endereça o dispositivo

    ADD DX, -2
    MOV AL, 0
    OUT DX, AL //Reseta o dispositivo
    MOV AL, 20
    OUT DX, AL //Habilita a leitura do pino DONE

    ADD DX, +1
    IN AL, DX //Faz a leitura do pino DONE
    MOV aux_Byte, AL

    ADD DX, -1
    MOV AL, 28
    OUT DX, AL //Desabilita o decoder e abre os buffers

    POP DX
    POP AX
  end;
end;

if aux_Byte = 200 then
  Result := True
else
  Result := False;
end;

procedure InformacoesBitstream(NomeArquivo: TShortString; Resultado:
TInformacoesBitstream); stdcall;
var
  Fl: File of Byte;
  Dado, Status: Byte;
  Caracter: Char;
  Limite: Integer;
  aux_ShortString: ShortString;
begin
  aux_ShortString := '';
  Resultado.Valida := False;

  AssignFile(Fl, NomeArquivo^);
  FileMode := 0;
  Reset(Fl);

  Status := 0;
  Limite := 0;
  while (Status = 0) and (not Eof(Fl)) do
  begin
    Read(Fl, Dado);
    if Dado = 97 then
    begin
      Read(Fl, Dado);
      if Dado = 0 then
        Status := Status + 1
      else

```

```

        Status := 6;
    end;
    Limite := Limite + 1;
    if Limite > 100 then
        Status := 6;
    end;

while (Status = 1) and (not Eof(F1)) do
begin
    Read(F1, Dado);
    if Dado = 98 then
        begin
            Read(F1, Dado);
            if Dado = 0 then
                Status := Status + 1
            else
                Status := 6;
            end;
        Limite := Limite + 1;
        if Limite > 150 then
            Status := 6;
        end;
    end;

if not Eof(F1) then
    Read(F1, Dado);
while (Status = 2) and (not Eof(F1)) do
begin
    Read(F1, Dado);
    if Dado <> 0 then
        begin
            asm
                PUSH AX

                MOV AL, Dado
                MOV Character, AL

                POP AX
            end;
            aux_ShortString := aux_ShortString + Character;
        end
    else
        begin
            Status := Status + 1;
        end;
    end;
end;

while (Status = 3) and (not Eof(F1)) do
begin
    Read(F1, Dado);
    if Dado = 99 then
        begin
            Read(F1, Dado);
            if Dado = 0 then
                Status := Status + 1
            else
                Status := 6;
            end;
        end;
    end;
end;

while (Status = 4) and (not Eof(F1)) do
begin
    Read(F1, Dado);
    if Dado = 100 then
        begin
            Read(F1, Dado);
            if Dado = 0 then
                Status := Status + 1
            else
                Status := 6;
            end;
        end;
    end;
end;

while (Status = 5) and (not Eof(F1)) do
begin
    Read(F1, Dado);

```

```

    if Dado = 101 then
    begin
        Read(F1, Dado);
        if Dado = 0 then
        begin
            while (Status < 15) and (not Eof(F1)) do
            begin
                Read(F1, Dado);
                if Dado = 255 then
                begin
                    Resultado.Valida := True;
                    Status := 15;
                end;
                Status := Status + 1;
            end;
        end
        else
        begin
            Status := Status + 1;
        end;
    end;
end;

Resultado.Dispositivo := aux_ShortString;
CloseFile(F1);
end;

procedure ConfiguraDispositivo(Base: Word; Dispositivo: Byte; NomeArquivo: TShortString;
Resultado: TConfiguraDispositivo); stdcall;
var
    Dado: Byte;
    Bitstream: File of Byte;
    InfBitstream: TInformacoesBitstream;
    Limite: Integer;
    Time0, Time1: TDateTime;
    aux_ShortString: ShortString;
begin
    New(InfBitstream);
    Time0 := Time;
    InfBitstream.Dispositivo := '';
    InfBitstream.Valida := False;
    Resultado.Sucesso := False;

    if HardwarePresente(Base) = True then
    begin
        if HardwareLigado(Base) = True then
        begin
            if ResetDispositivo(Base, Dispositivo) = True then
            begin
                InformacoesBitstream(NomeArquivo, InfBitstream);
                if InfBitstream.Valida = True then
                begin
                    AssignFile(Bitstream, NomeArquivo^);
                    FileMode := 0;
                    Reset(Bitstream);

                    Limite := 0;
                    while (Limite < 200) and (not Eof(Bitstream)) do
                    begin
                        Read(Bitstream, Dado);
                        if Dado = 255 then
                        begin
                            Limite := 200;
                        end;
                        Limite := Limite + 1;
                    end;

                    if Dado = 255 then
                    begin
                        for Limite := 0 to 7 do
                        begin
                            asm
                                PUSH AX
                                PUSH DX

```

```

        MOV DX, Base
        MOV AL, 21
        OUT DX, AL //Envia 1 bit
        ADD AL, 2
        OUT DX, AL //Gera 1 pulso de clock

        POP DX
        POP AX
    end;
end;

while not Eof(Bitstream) do
begin
    Read(Bitstream, Dado);

    asm
        PUSH AX
        PUSH BX
        PUSH DX

        MOV DX, Base
        MOV BL, Dado

        MOV AL, BL
        AND AL, 128
        RCR AL, 7 //Bit D7 de 'Dado'
        ADD AL, 20
        OUT DX, AL //Envia bit D7
        ADD AL, 2
        OUT DX, AL //Gera 1 pulso de clock

        MOV AL, BL
        AND AL, 64
        RCR AL, 6 //Bit D6 de 'Dado'
        ADD AL, 20
        OUT DX, AL //Envia bit D6
        ADD AL, 2
        OUT DX, AL //Gera 1 pulso de clock

        MOV AL, BL
        AND AL, 32
        RCR AL, 5 //Bit D5 de 'Dado'
        ADD AL, 20
        OUT DX, AL //Envia bit D5
        ADD AL, 2
        OUT DX, AL //Gera 1 pulso de clock

        MOV AL, BL
        AND AL, 16
        RCR AL, 4 //Bit D4 de 'Dado'
        ADD AL, 20
        OUT DX, AL //Envia bit D4
        ADD AL, 2
        OUT DX, AL //Gera 1 pulso de clock

        MOV AL, BL
        AND AL, 8
        RCR AL, 3 //Bit D3 de 'Dado'
        ADD AL, 20
        OUT DX, AL //Envia bit D3
        ADD AL, 2
        OUT DX, AL //Gera 1 pulso de clock

        MOV AL, BL
        AND AL, 4
        RCR AL, 2 //Bit D2 de 'Dado'
        ADD AL, 20
        OUT DX, AL //Envia bit D2
        ADD AL, 2
        OUT DX, AL //Gera 1 pulso de clock

        MOV AL, BL
        AND AL, 2
        RCR AL, 1 //Bit D1 de 'Dado'
        ADD AL, 20

```



```

        OUT DX, AL //Envia bit D1
        ADD AL, 2
        OUT DX, AL //Gera 1 pulso de clock

        MOV AL, BL
        AND AL, 1 //Bit D0 de 'Dado'
        ADD AL, 20
        OUT DX, AL //Envia bit D0
        ADD AL, 2
        OUT DX, AL //Gera 1 pulso de clock

        POP DX
        POP BX
        POP AX
    end;

end;

end;

asm
    PUSH AX
    PUSH DX

    MOV DX, Base
    ADD DX, 1
    IN AL, DX
    MOV Dado, AL

    POP DX
    POP AX
end;

if Dado = 216 then
begin
    aux_ShortString := 'Dispositivo configurado com sucesso!';
    Resultado.Sucesso := True;
end
else
begin
    aux_ShortString := 'Não houve confirmação de sucesso do
dispositivo!';
end;

asm
    PUSH AX
    PUSH DX

    MOV DX, Base
    MOV AL, 28
    OUT DX, AL //Desabilita o decoder e abre os buffers

    POP DX
    POP AX
end;
    CloseFile(Bitstream);
end
else
begin
    aux_ShortString := 'Bitstream inválida!';
end;
end
else
begin
    aux_ShortString := 'Dispositivo não responde!';
end;
end
else
begin
    aux_ShortString := 'Hardware desligado!';
end;
end
else
begin
    aux_ShortString := 'Hardware não encontrado!';
end;
end;

```

```

Resultado.Informacao := aux_ShortString;
Resultado.Dispositivo := InfBitstream.Dispositivo;
Time1 := Time;
Dispose(InfBitstream);
Resultado.Tempo := FloatToStrF((Time1 - Time0) * 86400, ffGeneral, 7, 4);
end;

exports
  HardwarePresente,
  HardwareLigado,
  ResetTodos,
  ResetDispositivo,
  InformacoesBitstream,
  ConfiguraDispositivo;
end.

```

D.2. Código do Controlador

```

unit Control;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ScktComp, Spin, ComCtrls, FileCtrl;

type
  TForm1 = class(TForm)
    Memo1: TMemo;
    ServerSocket1: TServerSocket;
    Button1: TButton;
    OpenDialog1: TOpenDialog;
    SpinEdit1: TSpinEdit;
    SaveDialog1: TSaveDialog;
    Button2: TButton;
    Button3: TButton;
    StatusBar1: TStatusBar;
    Label1: TLabel;
    Label2: TLabel;
    FileListBox1: TFileListBox;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure ServerSocket1Accept(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure ServerSocket1ClientRead(Sender: TObject;
      Socket: TCustomWinSocket);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

type
  TInformacoesBitstream = ^InfBit;
  InfBit = record
    Dispositivo: ShortString;
    Valida: Boolean;
  end;
  TConfiguraDispositivo = ^ConfDisp;
  ConfDisp = record
    Dispositivo: ShortString;
    Informacao: ShortString;
    Sucesso: Boolean;
    Tempo: ShortString;
  end;
  TShortString = ^ShortString;
  TLibResetTodos = procedure(Base: Word); stdcall;
  TLibResetDispositivo = function(Base: Word; Dispositivo: Byte): Boolean; stdcall;

```

```

    TLibInformacoesBitstream = procedure(NomeArquivo: TShortString; Resultado:
TInformacoesBitstream); stdcall;
    TLibConfiguraDispositivo = procedure(Base: Word; Dispositivo: Byte; NomeArquivo:
ShortString; Resultado: TConfiguraDispositivo); stdcall;

var
    Form1: TForm1;
    EnderecoBase, Porta: Word;
    IP, Root: String;
    N: Integer;
    procedure ResetTodos(Base: Word); stdcall;
    function ResetDispositivo(Base: Word; Dispositivo: Byte): Boolean; stdcall;
    procedure InformacoesBitstream(NomeArquivo: TShortString; Resultado:
TInformacoesBitstream); stdcall;
    procedure ConfiguraDispositivo(Base: Word; Dispositivo: Byte; NomeArquivo:
TShortString; Resultado: TConfiguraDispositivo); stdcall;

implementation

{$R *.dfm}

procedure ResetTodos(Base: Word);
    external 'RcnfgDrv.dll' name 'ResetTodos';

function ResetDispositivo(Base: Word; Dispositivo: Byte): Boolean;
    external 'RcnfgDrv.dll' name 'ResetDispositivo';

procedure InformacoesBitstream(NomeArquivo: TShortString; Resultado:
TInformacoesBitstream);
    external 'RcnfgDrv.dll' name 'InformacoesBitstream';

procedure ConfiguraDispositivo(Base: Word; Dispositivo: Byte; NomeArquivo: TShortString;
Resultado: TConfiguraDispositivo);
    external 'RcnfgDrv.dll' name 'ConfiguraDispositivo';

procedure TForm1.Button1Click(Sender: TObject);
var
    aux: TConfiguraDispositivo;
    aux_Byte: Byte;
    aux_ShortString: ShortString;
begin
    New(aux);
    if OpenFileDialog1.Execute then
        begin
            aux_Byte := SpinEdit1.Value;
            aux_ShortString := OpenFileDialog1.FileName;
            ConfiguraDispositivo(EnderecoBase, aux_Byte, @aux_ShortString, aux);
            Mem1.Lines.Clear;
            Mem1.Lines.Append(aux.Dispositivo);
            Mem1.Lines.Append(aux.Tempo + 's');
            Mem1.Lines.Append(aux.Informacao);
            Mem1.Lines.Append('');
            Mem1.Lines.Append('Listening...');
        end;
    Dispose(aux);
end;

procedure TForm1.FormCreate(Sender: TObject);
var
    F1, F2, F3: TextFile;
    aux: String;
begin
    try
        try
            AssignFile(F1, 'Controlador.ini');
            Reset(F1);
            Readln(F1, aux);
            EnderecoBase := StrToInt(aux);
            Readln(F1, aux);
            IP := aux;
            Readln(F1, aux);
            Porta := StrToInt(aux);
            ServerSocket1.Port := Porta;
            Readln(F1, aux);
            Root := aux;
        end;
    end;
end;

```

```

    Readln(F1, aux);
    N := StrToInt(aux);
    AssignFile(F2, Root + '\index.html');
    Reset(F2);
    AssignFile(F3, Root + '\index.tmp');
    Rewrite(F3);
    while not Eof(F2) do
    begin
        Readln(F2, aux);
        Writeln(F3, aux);
    end;
    Rewrite(F2);
    Reset(F3);
    while not Eof(F3) do
    begin
        Readln(F3, aux);
        if Copy(aux,1,19) = '<frame src="http://' then
            Writeln(F2, '<frame src="http://' + IP + ':' + IntToStr(Porta) + '/'>')
        else
            Writeln(F2, aux);
        end;
        Rewrite(F3);
        Memol.Lines.Append(IntToStr(EnderecoBase));
        Memol.Lines.Append(IP);
        Memol.Lines.Append(IntToStr(ServerSocket1.Port));
        Memol.Lines.Append(Root);
        Memol.Lines.Append('');
        ServerSocket1.Active := True;
        if ServerSocket1.Active = True then
            Memol.Lines.Append('Listening...')
        else
            Memol.Lines.Append('Stopped');
        finally
            CloseFile(F3);
            CloseFile(F2);
            CloseFile(F1);
        end;
    except
        //Emitir mensagem de erro
    end;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
    aux_Boolean: Boolean;
    aux_Byte: Byte;
begin
    aux_Byte := SpinEdit1.Value;
    aux_Boolean := ResetDispositivo(EnderecoBase, aux_Byte);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    ResetTodos(EnderecoBase);
end;

procedure TForm1.ServerSocket1Accept(Sender: TObject;
    Socket: TCustomWinSocket);
begin
    Memol.Lines.Clear;
    Memol.Lines.Append(Socket.RemoteAddress);
end;

procedure TForm1.ServerSocket1ClientRead(Sender: TObject;
    Socket: TCustomWinSocket);
var
    strHtml, Arquivo, Dispositivo: String;
    j: Integer;
    Bitstream: TInformacoesBitstream;
    aux_ShortString: ShortString;
    aux: TConfiguraDispositivo;
    aux_Byte: Byte;
begin
    strHtml := Socket.ReceiveText;
    Memol.Lines.Clear;

```

```

Memor1.Lines.Append(strHtml);

if (Copy(strHtml,1,16) <> 'GET /?bitstream=') and (Copy(strHtml,1,13) <> 'GET
/?device=') then
begin
    strHtml := '<p><b><font face="Arial" size="4">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& Seleccione uma
<i>bitstream</i></font></b></p>';
    Socket.SendText(strHtml);
    FileListBox1.Update;
    for j := 0 to FileListBox1.Items.Count - 1 do
    begin
        Arquivo := FileListBox1.Items.Strings[j];

        New(Bitstream);
        aux_ShortString := Root + '\' + Arquivo;
        InformacoesBitstream(@aux_ShortString, Bitstream);
        if Bitstream.Valida = True then
            Dispositivo := Bitstream.Dispositivo
        else
            Dispositivo := 'Bitstream inválida!';
            Dispose(Bitstream);

            if Dispositivo <> 'Bitstream inválida!' then
                strHtml := '<p>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& <font face="Arial">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& - <a
href="http://' + IP + ':' + IntToStr(Porta) + '?bitstream=' + Arquivo + '">' + Arquivo +
'</a> (Dispositivo ' + Dispositivo + ')</font></p>'
            else
                strHtml := '<p>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& <font face="Arial">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& - '
+ Arquivo + '</a> (<i>Bitstream</i> inválida)</font></p>';

            Socket.SendText(strHtml);
            end;

        strHtml := '<p><font face="Arial">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& <a href="http://' + IP +
'/upload.htm">Envie uma <i>bitstream</i></a></font></p>';
        Socket.SendText(strHtml);

    end
    else
    begin

        if Copy(strHtml,1,16) = 'GET /?bitstream=' then
            begin

                j := 17;
                Arquivo := '';
                while Copy(strHtml,j,1) <> '.' do
                begin
                    Arquivo := Arquivo + Copy(strHtml,j,1);
                    j := j + 1;
                end;
                Arquivo := Arquivo + '.bit';

                strHtml := '<p><b><font face="Arial" size="4">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& Seleccione um
dispositivo:</font></b></p>';
                Socket.SendText(strHtml);

                for j := 0 to N - 1 do
                begin
                    strHtml := '<p>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& <font face="Arial">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& -
<a href="http://' + IP + ':' + IntToStr(Porta) + '?device=' + IntToStr(j) +
'&bitstream=' + Arquivo + '">FPGA_' + IntToStr(j) + '</a></font></p>';
                    Socket.SendText(strHtml);
                end;
                strHtml := '<p>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& <font face="Arial">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& - <a
href="http://' + IP + ':' + IntToStr(Porta) + '?device=all&bitstream=' + Arquivo +
'">FPGA_Todos</a></font></p>';
                Socket.SendText(strHtml);
                strHtml := '<p><font face="Arial">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;& <a href="http://' + IP +
'">Voltar</i></a></font></p>';
                Socket.SendText(strHtml);

            end
            else

```

[illegible]

```
        Dispose(aux);

        strHtml := '<p><font face="Arial">&nbsp;&nbsp;&nbsp;&nbsp;<a href="http://' +
IP + '">Voltar</i></a></font></p>';
        Socket.SendText(strHtml);

    end;

end;

end;

Socket.Close;
Memor1.Lines.Append('Listening...');
end;

end.
```

D.3. Código do Site

```
<html>

<head>
<title>Controlador de Reconfiguração</title>
</head>

<frameset>
<frame src="http://IP:Port/">
</frameset>

</html>
```