Uma Arquitetura Sistólica para Solução de Sistemas Lineares Implementada com circuitos FPGAs

Antônio Carlos de Oliveira Souza Aragão

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 - Área: Ciências de Computação e Matemática Computacional.

São Carlos
Dezembro - 1998

Aos meus pais e à Andrea,
minhas fontes de incentivo e carinho.

Agradecimentos

À Deus, pelas oportunidades e dádivas em minha vida.

Aos meus pais pelo amor e dedicação em minha formação.

À Andrea pela paciência, carinho e incentivo.

Ao Prof. Eduardo Marques, mais que um orientador um grande amigo.

Ao Prof. Donald Hung e à WSU pela cooperação técnica.

Ao Prof. Jorge Luiz Silva, Karl Hillesland, Omar Cortes, Marcos Roberto, Alencar e a todos que colaboraram para a realização deste trabalho.

Aos amigos e companheiros de mestrado que caminharam juntos nessa batalha.

Aos funcionários do ICMC, sempre prestativos e eficientes.

Ao CNPq pelo apoio financeiro recebido.

Índice

1. INTRODUÇÃO 1

1.1. CONSIDERAÇÕES INICIAIS 1
1.2. ORGANIZAÇÃO DO TRABALHO 2

2. METODOLOGIAS AVANÇADAS PARA PROJETO DE HARDWARE 3

2.1. EVOLUÇÃO DA TÉCNICA DE DESENVOLVIMENTO DE HARDWARE 3
2.2. FERRAMENTAS DE AUTOMAÇÃO DE PROJETOS ELETRÔNICOS.......................................5
2.3. METODOLOGIA DE PROJETO EM ALTO NÍVEL...7
2.3.1. Entrada do Projeto 9
2.3.2. Linguagens de Descrição de Hardware - HDLs...10
2.3.2.1. A Linguagem VHDL 10
2.3.2.2. Verilog HDL...12
2.3.2.3. A Aceitação das HDLs como Ferramentas de Projeto 12
2.3.3. Síntese 12
2.3.3.1. Síntese Comportamental..13
2.3.3.2. Síntese HDL 13
2.3.3.3. Síntese Lógica...13
2.3.4. Simulação..14
2.3.5. Síntese de Testes 16
2.4. CONSIDERAÇÕES FINAIS..17

3. A TECNOLOGIA FPGA 19

3.1. EVOLUÇÃO DOS DISPOSITIVOS LÓGICOS PROGRAMÁVEIS 19
3.2. ARQUITETURA DE FPGAS 21
3.2.1. Tecnologias de Programação 22
3.2.1.1. A Tecnologia de Programação SRAM 23
3.2.1.2. A Tecnologia Antifuse...24
3.2.1.3. A Tecnologia de Gate Flutuante 24
3.2.2. Arquitetura de Blocos Lógicos..26
3.2.2.1. Blocos de Granulosidade Fina 26
3.2.2.2. Blocos de Granulosidade Grossa 27
3.2.2.3. Lógica Sequencial...27
3.2.3. Arquitetura de Roteamento 27
3.2.4. Arquiteturas Comerciais de FPGAs 29
3.2.4.1. Os FPGAs da Xilinx 30
3.2.4.2. Os FPGAs FLEX8000 e FLEX10000 da Altera 36
3.2.4.3. Os FPGAs da Actel...38
3.3. DESENVOLVIMENTO E IMPLEMENTAÇÃO DE PROJETOS USANDO FPGAS............................41
3.3.1. Especificação e Entrada do Projeto 43
3.3.1.1. Editores de Esquemático 43
3.3.1.2. Entrada do Projeto em Baixo Nível...44
3.3.1.3. Entrada através de Linguagens de Descrição de Hardware - HDLs......................44
3.3.2. Síntese Lógica e Mapeamento 45
3.3.2.1. Mapeamento na Tecnologia...46
3.3.3. Posicionamento e Roteamento...47
3.3.3.1. Posicionamento..48

3.3.3.2. Roteamento 48
3.3.3.3. Capacidade de Roteamento e Recursos de Roteamento 49
3.3.3.4. Atrasos de Roteamento 49
3.3.4. Verificação e Testes 50
3.3.4.1. Verificação do Projeto 50
3.3.4.2. Testes de FPGAs 51
3.3.5. Configuração do FPGA 52
3.4. APLICAÇÕES DE FPGAS 52
3.4.1. Hardware Reconfigurável: Uma Nova Abordagem Computacional 53
3.5. CONSIDERAÇÕES FINAIS 54
3.5.1. Escolha das Ferramentas 55

4. ARQUITETURAS SISTÓLICAS 58

4.1. CONSIDERAÇÕES INICIAIS 58
4.2. CONCEITOS E CARACTERÍSTICAS 59
4.3. APLICAÇÕES 62
4.4. TÉCNICAS DE PROJETO E MAPEAMENTO 63
4.5. ASPECTOS RELATIVOS À IMPLEMENTAÇÃO 64
4.5.1. Granulosidade dos Elementos de Processamento 64
4.5.2. Propósito Geral x Propósito Específico 65
4.5.3. Particionamento de Algoritmos 65
4.5.4. Blocos Construtivos Universais 65
4.5.5. Integração em Sistemas já Existentes 66
4.6. CONSIDERAÇÕES FINAIS 66

5. SISTEMAS LINEARES 68

5.1. CONSIDERAÇÕES INICIAIS 68
5.2. SISTEMAS DE EQUAÇÕES LINEARES 68
5.3. MÉTODOS PARA SOLUÇÃO DE SISTEMAS LINEARES 69
5.3.1. Métodos Diretos 69
5.3.2. Método Iterativos 70
5.3.2.1. Método Iterativos Estacionários 71
5.3.2.1.1. Método RF 72
5.3.2.1.2. Método de Jacobi 72
5.3.2.1.3. Método de Gauss-Seidel 73
5.3.2.1.4. Método Baseado no Rede Neural Analógica de Wang e Li 73
5.4. ASPECTOS COMPUTACIONAIS DOS MÉTODOS 75
5.5. CONSIDERAÇÕES FINAIS 76

6. ARQUITETURA PROPOSTA E DESENVOLVIMENTO DO PROJETO 77

6.1. CONSIDERAÇÕES INICIAIS 77
6.2. ARQUITETURA PROPOSTA 77
6.2.1. Versão 1: Arquitetura para Sistemas de tamanho igual ao número de EPs no anel (n = N) 79
6.2.2. Versão 2: Arquitetura para Sistemas maiores que o número de EPs no anel (n > N) 85
6.3. IMPLEMENTAÇÃO DAS ARQUITETURAS 91
6.3.1. Simulação em Software 91
6.3.2. Projeto do Elemento de Processamento 92
6.3.2.1. Projeto do Multiplicador-Acumulador (MAC) 92
6.3.2.2. Projeto da FIFO 93

6.3.3. Projeto do Controlador *93*
6.3.4. Simulação Funcional *94*
6.3.5. Implementação em Circuitos FPGAs *94*
6.4. Considerações Finais 95

7. ANÁLISE E DISCUSSÃO DOS RESULTADOS 96

7.1. Considerações Iniciais 96
7.2. Complexidade do Algoritmo 96
7.3. Resultados da Implementação em circuitos FPGAs 97
7.4. Considerações Finais 100

8. CONCLUSÕES 102

8.1. Considerações Iniciais 102
8.2. Contribuições deste Trabalho 103
8.3. Sugestões para Desenvolvimento Futuro 104

REFERÊNCIAS BIBLIOGRÁFICAS 105

Lista de Figuras

FIGURA 2.1 TECNOLOGIAS DE PROJETO DE CIRCUITOS INTEGRADOS DIGITAIS 4
FIGURA 2.2 EVOLUÇÃO DAS FERRAMENTAS EDA AO LONGO DAS ÚLTIMAS DÉCADAS. 6
FIGURA 2.3 TÉCNICA DE PROJETO TOP-DOWN...8
FIGURA 2.4 ETAPAS DO PROJETO DE HARDWARE COM A METODOLOGIA DE PROJETO EM ALTO NÍVEL. 9
FIGURA 2.5 CÓDIGO VHDL DE UM MULTIPLEXADOR 4-1...11
FIGURA 2.6 SÍNTESE LÓGICA..14
FIGURA 2.7 PROCESSO DE SIMULAÇÃO COM VÁRIAS ITERAÇÕES AO LONGO DO PROJETO..............16
FIGURA 3.1 ESTRUTURA DE UM FPGA. 22
FIGURA 3.2 TECNOLOGIA DE PROGRAMAÇÃO SRAM. 23
FIGURA 3.3 ANTIFUSE PLICE DA ACTEL. 24
FIGURA 3.4 COMUTADOR PROGRAMÁVEL BASEADO EM EPROM. 25
FIGURA 3.5 ARQUITETURA GERAL DE ROTEAMENTO DE UM FPGA. 28
FIGURA 3.6 ARQUITETURAS COMERCIALMENTE DISPONÍVEIS. 30
FIGURA 3.7 BLOCO LÓGICO DA FAMÍLIA XC4000. 31
FIGURA 3.8 SEGMENTOS SIMPLES E COMUTADORES DE ROTEAMENTO DA SÉRIE XC4000. 33
FIGURA 3.9 SEGMENTOS DUPLOS E LINHAS LONGAS DA SÉRIE XC4000......................................33
FIGURA 3.10 UMA CÉLULA BÁSICA DA FAMÍLIA XC6200. 34
FIGURA 3.11 ARQUITETURA HIERÁRQUICA DA FAMÍLIA XC6200. 35
FIGURA 3.12 ARQUITETURA DO FPGA DA SÉRIE FLEX 8000 DA ALTERA. 36
FIGURA 3.13 BLOCO LÓGICO (LE) DA SÉRIE FLEX 8000 DA ALTERA. 37
FIGURA 3.14 ARQUITETURA DA SÉRIE FLEX 10000 DA ALTERA. 38
FIGURA 3.15 ARQUITETURA DOS FPGAS DA ACTEL. 39
FIGURA 3.16 ARQUITETURA DE ROTEAMENTO DA ACTEL. 40
FIGURA 3.17 SISTEMA EDA PARA PROJETOS COM FPGAS. 42
FIGURA 3.18 EXEMPLO DE MAPEAMENTO NUM FPGA DA XILINX. 47
FIGURA 4.1 PRINCÍPIO BÁSICO DE ARQUITETURAS SISTÓLICAS. 59
FIGURA 4.2 TOPOLOGIAS SISTÓLICAS MAIS COMUNS...61
FIGURA 4.3 ARQUITETURA DO ARRANJO SISTÓLICO (4X4 PES) DO CHIP GENES 62
FIGURA 6.1 TOPOLOGIA EM ANEL DA ARQUITETURA PROPOSTA. 78
FIGURA 6.2 ESTRUTURA BÁSICA DO EP. 79
FIGURA 6.3 DIAGRAMA DE FLUXO DE DADOS PARA N=3, N=3..81
FIGURA 6.4 DIAGRAMA RTL DO EP NA ARQUITETURA PARA N=N 82
FIGURA 6.5 DIAGRAMA DE FLUXO DE DADOS RTL PARA O MAC. 82
FIGURA 6.6 FLUXOGRAMA DA ARQUITETURA PARA N=N..84
FIGURA 6.7 FLUXO DE DADOS PARA O CASO ONDE N=6, N=3 (Q=2). 87
FIGURA 6.8 DIAGRAMA DE FLUXO DE DADOS (RTL) DO EP DA ARQUITETURA PARA N>N. 88
FIGURA 6.9 ESTÁGIO DE INICIALIZAÇÃO DO FLUXOGRAMA DA ARQUITETURA PARA N>N 89
FIGURA 6.10 PARTE ITERATIVA DO FLUXOGRAMA DA ARQUITETURA PARA N>N 90
FIGURA 6.11 ESTÁGIO DE FINALIZAÇÃO DO FLUXOGRAMA DA ARQUITETURA PARA N>N 91
FIGURA 7.1 TEMPO DE PROCESSAMENTO (EM SEGUNDOS) X NRO. DE EPS NO ANEL (N)..............98
FIGURA 7.2 TAMANHO DO SISTEMA (N) X TEMPO DE PROCESSAMENTO..99
FIGURA 7.3 ALGORITMO PARA MULTIPLICAÇÃO MATRIZ-VETOR...99

Lista de Tabelas

Tabela 3.1 Sumário das Tecnologias de Programação. 25
Tabela 6.1 Sinais de Controle para os EPs da Arquitetura para n=N.............................83
Tabela 6.2 Sinais de Controle para os EPs da Arquitetura para n>N. 88
Tabela 6.3 Recursos do FPGA utilizados no mapeamento 95
Tabela 7.1 Tempo de Processamento (em segundos, clock= 27 MHz) de 10000 iterações, para várias configurações do anel (N) e tamanhos do sistema (n). 98
Tabela 7.2 Tempos de Processamento de 10000 iterações do algoritmo de Multiplicação Matriz-Vetor para vários tamanhos de matriz (n), em um microcomputador Pentium II 100

Lista de Abreviações

ANN	Artificial Neural Network
ASIC	Application Specific Integrated Circuit
ATE	Automatic Test Equipment
ATM	Asynchronous Transfer Mode
ATPG	Automatic Test Pattern Generator
CI	Circuito Integrado
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processing
EAB	Embedded Array Block
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
EEPROM	Electrically Erasable Programmable Read Only Memory
EP	Elemento de Processamento
EPROM	Erasable Programmable Read Only Memory
FIR	Finite Impulse Response
FPAA	Field-Programmable Analog Array
FPGA	Field-Programmable Gate Array
FPIC	Field-Programmable Interconnect Component
HDL	Hardware Description Language
I/O	Input/Output
IEEE	Institute of Electrical and Electronic Engineers
IIR	Infinite Impulse Response
IOB	Input/Output Block
ISP	In System Programmability
LAB	Logic Array Block
LCA	Logic Cell Array
LE	Logic Element
LPM	Library of Parameterized Modules

LUT	Look-Up Table
MOS	Metal-Oxide Semiconductor
MPGA	Mask-Programmable Gate Array
OTP	One Time Programmable
PAL	Programmable Array Logic
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RNN	Recurrent Neural Network
SPLD	Simple Programmable Logic Device
SRAM	Static Random Access Memory
UV	Ultra Violeta
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
XNF	Xilinx Netlist Format

Resumo

Neste trabalho de mestrado foi desenvolvido o projeto de uma máquina paralela dedicada para solução de sistemas de equações lineares. Este é um problema presente em uma grande variedade de aplicações científicas e de engenharia e cuja solução torna-se uma tarefa computacionalmente intensiva, à medida em que o número de incógnitas aumenta. Implementou-se uma Arquitetura Sistólica, conectada numa topologia em anel, que mapeia métodos de solução iterativos. Essa classe de arquiteturas paralelas apresenta características de simplicidade, regularidade e modularidade que facilitam implementações em hardware, sendo muito utilizadas em sistemas de computação dedicados à solução de problemas específicos, os quais possuem como requisitos a grande demanda computacional e a necessidade de respostas em tempo real. Foram adotadas metodologias e ferramentas avançadas para projeto de *hardware* que aceleram o ciclo de desenvolvimento e para a implementação foram utilizados circuitos reconfiguráveis FPGAs (*Field Programmable Gate Arrays*). Os resultados de desempenho são apresentados e discutidos, indicando que a abordagem e metodologia adotada é viável e eficiente para solução deste tipo de problema.

Abstract

This dissertation presents the project of a parallel machine dedicated for solving linear systems. This is a problem that appears in a great variety of scientific and engineering applications with a solution that becomes a computationally intensive task, measured by the increasing number of unknown variables. An Systolic Architecture was implemented, connected in a ring topology, mapping an iterative solution method. This class of parallel architectures presents characteristics of simplicity, regularity and modularity that facilitate hardware implementations, being very used in dedicated computation systems to the solution of specific problems, which possess as requirements to handle great computational demand and real-time response. Advanced methodologies and tools for hardware project were adopted to accelerate the development cycle. The architecture has been implemented and verified on FPGAs (Field Programmable Gate Arrays). The performance results are presented and discussed, indicating the feasibility and efficiency of the adopted approach and methodology for this kind of problem.

1. Introdução

1.1. Considerações Iniciais

A Comunidade de Computação de Alto Desempenho tem observado que é necessário desenvolver soluções personalizadas para seus problemas. Soluções genéricas muitas vezes não são adequadas, pois sacrificam muito o desempenho quando aplicadas a um caso específico de interesse.

Na busca de alto desempenho, a abordagem mais adotada é a utilização de técnicas de programação concorrente para paralelização de algoritmos a serem executados em máquinas paralelas de propósito geral. A construção de arquiteturas dedicadas tem sido uma abordagem pouco explorada, devido ao elevado custo de projeto e tempo de desenvolvimento.

A evolução das metodologias de projeto de *hardware*, apoiadas em poderosas ferramentas de *software* que aceleram o ciclo de desenvolvimento, e especialmente o surgimento de dispositivos reconfiguráveis como os FPGAs (*Field-Programmable Gate Arrays*) abriram um novo horizonte entre os extremos da computação de finalidade geral e o *hardware* dedicado.

Hoje, é possível desenvolver rapidamente um projeto de sistema digital empregando-se novas metodologias como linguagens de descrição de *hardware* (HDLs), ferramentas de síntese lógica e simulação. Utilizando-se os circuitos reconfiguráveis pode-se implementar em campo, sem necessidade de processos de fabricação de *chips*, um protótipo ou a versão final do projeto desenvolvido.

Neste trabalho de pesquisa apresenta-se "Uma Arquitetura Sistólica para Solução de Sistemas Lineares implementada com os circuitos FPGAs". Com essa forma de implementação tem-se como objetivo explorar essas novas metodologias de projeto e adquirir o *know-how* dessa nova tecnologia de dispositivos semicondutores.

Também pretende-se demonstrar a viabilidade e eficiência desse tipo de abordagem, tomando-se como exemplo a solução de uma classe de problema presente em uma variedade enorme de aplicações científicas e de engenharia e que muitas vezes possui requisitos de desempenho.

Os métodos de solução de sistemas lineares envolvem cálculos matriciais que são computacionalmente intensivos. As arquiteturas sistólicas são as mais adequadas para manipular operações matriciais, pois exploram o paralelismo de granulosidade fina presente nessas operações e possuem um baixo *overhead* de comunicação e sincronismo (Moreno & Lang, 90). Muitos algoritmos bem conhecidos para manipulação de matrizes já foram mapeados em arquiteturas sistólicas. Além disso, essa classe de arquiteturas paralelas também é muita apropriada para implementação em *hardware*, pois apresenta características de regularidade e modularidade que facilitam o projeto.

A pesquisa a que se refere este projeto foi desenvolvida pela Universidade de São Paulo - USP, Instituto de Ciências Matemáticas e de Computação - ICMC, Laboratório de Sistemas Digitais - LaSD, em conjunto com a Washington State University - WSU, School of Electrical Engineering and Computer Science - EECS, USA. Coube à WSU a concepção e especificação da arquitetura sistólica para solução de sistemas de equações lineares. Coube à USP a implementação em *hardware* e a validação do sistema especificado. Os coordenadores locais são: Prof. Dr. Donald Hung (WSU) e Prof. Dr. Eduardo Marques (USP), estando envolvidos nesta pesquisa alunos de mestrado de ambos os países e o Prof. Dr. Jorge Luiz e Silva (UFSCar).

1.2. Organização do Trabalho

Este trabalho consiste de oito capítulos. No capítulo 2 apresenta-se um panorama das tecnologias de implementação de circuitos digitais e as novas metodologias de projeto, enfocando as ferramentas de síntese lógica e as linguagens de descrição de *hardware*. O capítulo 3 apresenta a tecnologia FPGA, descrevendo sua arquitetura, principais características e o ciclo de projeto com esses dispositivos. Também discute-se as principais aplicações dessa tecnologia. No capítulo 4 são apresentados os conceitos relativos às Arquiteturas Sistólicas. Os aspectos mais importantes dessa classe de arquiteturas, suas aplicações e técnicas de projeto são descritos. O capítulo 5 apresenta alguns métodos de solução de sistemas lineares, destacando os métodos iterativos. No capítulo 6 são apresentadas duas arquiteturas sistólicas que implementam um método iterativo para solução de sistemas lineares. A topologia da arquitetura e seu funcionamento são explanados e são apresentados detalhes do projeto do *hardware*. No capítulo 7 discute-se o desempenho obtido e compara-se a outras formas de implementação. Finalmente, o capítulo 8 apresenta as conclusões e sugestões para trabalhos futuros.

2. Metodologias Avançadas para Projeto de Hardware

2.1. Evolução da Técnica de Desenvolvimento de Hardware

Várias evoluções no projeto de circuitos digitais têm ocorrido nas últimas décadas. As constantes mudanças na tecnologia têm transformado radicalmente o processo de projeto. Os componentes dos circuitos têm evoluído de transistores individuais a circuitos integrados VLSI (*Very large scale integration*). Ferramentas EDA (*Electronic Design Automation*) têm acelerado o ciclo de projeto. Não é mais necessário desenhar portas lógicas individuais e montar diferentes componentes. As linguagens de descrição de *hardware* (HDLs) estão se consolidando como forma padrão de descrição de projetos. Ferramentas de síntese lógica automática estão disponíveis para mapear circuitos em diversas tecnologias. Além das mudanças na tecnologia, o ciclo de vida dos produtos modernos está tornando-se mais curto que os tradicionais ciclos de projeto, exigindo uma rápida prototipação.

A implementação de um sistema digital não é independente do estilo de projeto. Circuitos digitais podem ser construídos sob diferentes tecnologias dependendo de fatores como tamanho, função, desempenho requerido e também o custo do projeto. As implementações de circuitos podem ser agrupadas em duas categorias principais (Chan & Mourad, 94): circuitos completamente customizados e semicustomizados, como ilustrado na figura 2.1. A categoria de circuitos semicustomizados consiste de várias abordagens, as quais têm facilitado o projeto e a fabricação de circuitos digitais.

a) CIs customizados (ASICs-*Application Specific Integrated Circuits*): necessitam de um processo de fabricação especial que requer máscaras específicas para cada projeto. Também, o tempo de desenvolvimento é longo e os custos extremamente altos. Para aplicações que requerem grande volume de produção, o alto custo do projeto e testes pode ser amortizado.

b) *Mask-Programmable Gate Arrays* (MPGAs). 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 usualmente facilitado por uma biblioteca de células, oferecendo um tempo de desenvolvimento mais curto e custos mais baixos que CIs customizados.

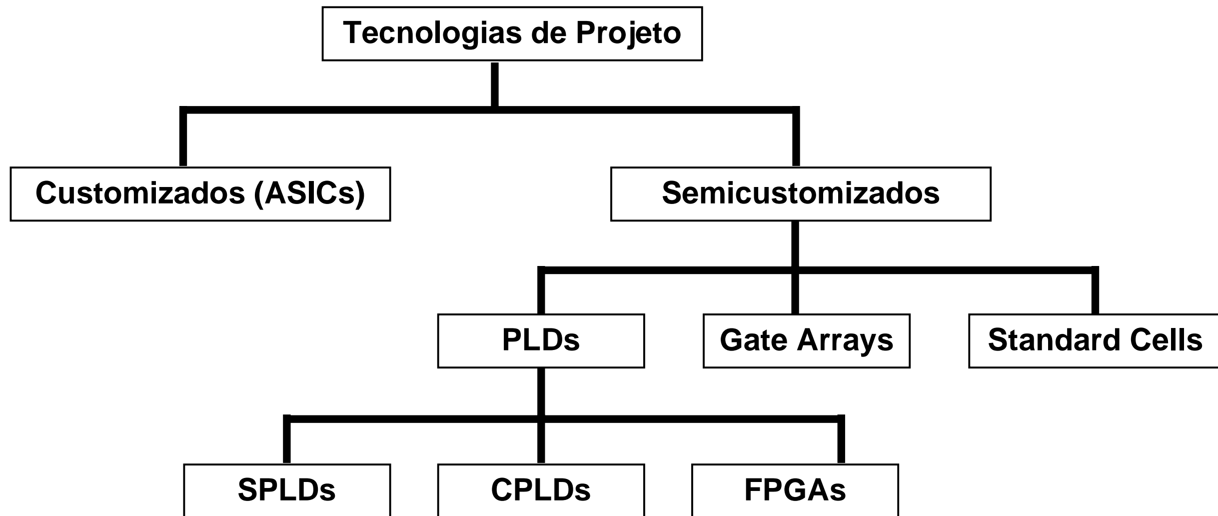


Figura 2.1 Tecnologias de Projeto de Circuitos Integrados Digitais

c) *Standard Cells*. Nesta abordagem, como no caso de MPGAs, a tarefa de projetar é facilitada pelo uso de módulos pré-projetados. Os módulos, *standard cells*, são geralmente salvos em um banco de dados. Os projetistas selecionam as células do banco de dados para realizar o projeto. Comparado aos CIs customizados, os circuitos implementados em *standard cells* são menos eficientes em tamanho e desempenho, entretanto, seu custo de desenvolvimento é baixo.

d) *Programmable Logic Devices* (PLDs). Possuem como principal característica a capacidade de programação (configuração) pelo usuário, eliminando o processo de fabricação e facilitando as mudanças em projetos. Possibilitam um curto ciclo de projeto a baixos custos. São classificados em várias categorias conforme características arquiteturais e capacidade lógica.

O mercado para PLDs tem crescido exponencialmente na última década de modo que há uma grande variedade de dispositivos atualmente disponíveis. O próximo capítulo apresenta um breve histórico da evolução desses dispositivos, os quais têm revolucionado o processo de projeto de

hardware digital, enfocando com maiores detalhes a tecnologia FPGA, que foi empregada na implementação do presente projeto.

2.2. Ferramentas de Automação de Projetos Eletrônicos

A evolução da automação de projeto eletrônico, *Electronic Design Automation* (EDA), começou nos anos 70 com as ferramentas de projeto auxiliado por computador, *Computer-Aided Design* (CAD), que davam assistência aos projetistas na geração de desenhos e *layout* de circuitos. Ferramentas de verificação física verificavam o trabalho dos projetistas.

Nos anos 80, as ferramentas CAE (*Computer-Aided Engineering*) proporcionaram aos engenheiros um nível de abstração mais alto no projeto. Os engenheiros agora possuíam capacidade computadorizada de captura esquemática e roteamento. A simulação lógica também foi introduzida, o que ajudou os engenheiros a verificarem a funcionalidade de seus projetos. Entretanto, simuladores em nível de portas lógicas provaram ser inadequados para grandes sistemas (tais como aqueles com mais do que 50.000 portas lógicas) onde as falhas arquiteturais são difíceis de serem detectadas.

A partir do CAD nos anos 70, o propósito de todos os sistemas EDA tem sido manipular a complexidade do projeto. Os objetivos permaneceram constantes: aumentar a produtividade, a qualidade, e a previsibilidade de um projeto.

As ferramentas EDA agora evoluíram para o projeto em alto nível. O projeto em alto nível está baseado em três poderosas ferramentas: síntese, simulação e teste, sendo a síntese a tecnologia principal. A figura 2.2 ilustra esse processo evolutivo que as ferramentas experimentaram nas últimas décadas.

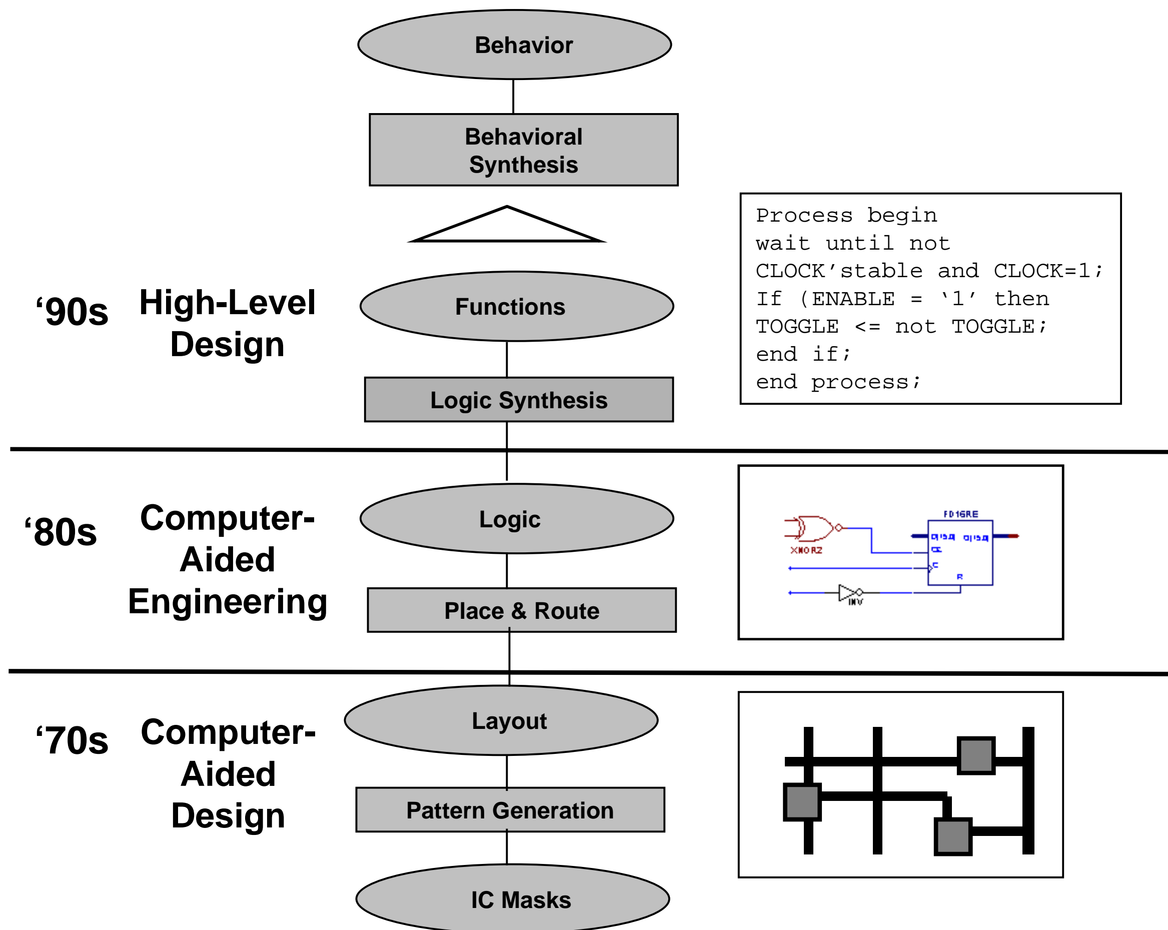


Figura 2.2 Evolução das Ferramentas EDA ao longo das últimas décadas.

A síntese proporciona a tradução automática das abstrações de nível mais alto em descrições em nível de portas lógicas. Antes da síntese, a simulação poderia ser feita a partir de uma descrição do projeto em uma linguagem de descrição de *hardware* e o projeto poderia ser verificado, mas o código não poderia ser traduzido em um projeto manufaturável. A síntese representa a diferença chave entre o projeto em alto nível e o CAE tradicional. As formas de entrada e a simulação nos projetos em alto nível são similares ao CAE na aparência, mas há várias diferenças fundamentais.

Assim como no CAE, no projeto em alto nível a simulação é utilizada para validação e verificação do projeto. Entretanto, diferentemente dos simuladores em nível de portas lógicas no CAE, os simuladores para projeto em alto nível suportam uma mistura de vários níveis de representação funcional do projeto, abrangendo desde os níveis estruturais mais baixos até os

níveis comportamentais mais altos. O projeto no CAE é limitado pela captura esquemática em nível de portas lógicas. Em contraste, a entrada de projeto em alto nível suporta uma variedade de representações, tais como diagramas de bloco, máquinas de estado e descrições textuais em linguagens de descrição de *hardware* tais como o VHDL ou Verilog. Os projetos são descritos em nível comportamental já na entrada do sistema, ao invés de tradicionalmente serem desenhados em nível de portas lógicas. Deste modo, as tendências arquiteturais são muito mais fáceis de serem implementadas e a capacidade de simulação é aumentada, uma vez que o projeto é representado em uma forma mais compacta.

Atualmente, ambientes de automação de projetos em alto nível combinam linguagens de descrição de *hardware*, ferramentas de síntese e de simulação de sistemas, para verificação completa da funcionalidade de um circuito antes de comprometer o silício. Nas próximas seções discute-se o processo de projeto em alto nível, destacando alguns aspectos como as linguagens de descrição de *hardware* e a etapa de síntese.

2.3. Metodologia de Projeto em Alto Nível

O projeto em alto nível é uma estratégia *top-down* (figura 2.3) baseada na criação de um modelo de operação para cada nível de abstração antes de o decompor para níveis de implementação mais baixos. O processo deve incorporar informações técnicas *bottom-up* na implementação dos parâmetros que podem afetar os requisitos, tais como área e desempenho.

Uma das maiores diferenças entre o projeto em nível de portas lógicas e o projeto em alto nível é a separação clara entre a exploração e a criação da implementação. Captura, exploração e implementação estão fundidas em um único processo no projeto em nível de portas lógicas. No projeto em alto nível, a implementação é realizada ao sintetizar o HDL capturado, e é direcionada pelas metas atribuídas pelo projetista (velocidade, área, potência, etc.). A exploração de múltiplas alternativas de implementação podem ser executadas rapidamente pela alteração das metas do projeto. Em outras palavras, os métodos em nível de portas lógicas forçam a implementação antes de explorar as possibilidades do projeto. O projeto em alto nível permite que se explore várias alternativas antes da implementação.

A exploração das alternativas de projeto requer uma mistura do projeto *top-down* e *bottom-up*. Métodos de projeto em alto-nível, que incluem síntese, tornam possível explorar essas várias implementações rapidamente.

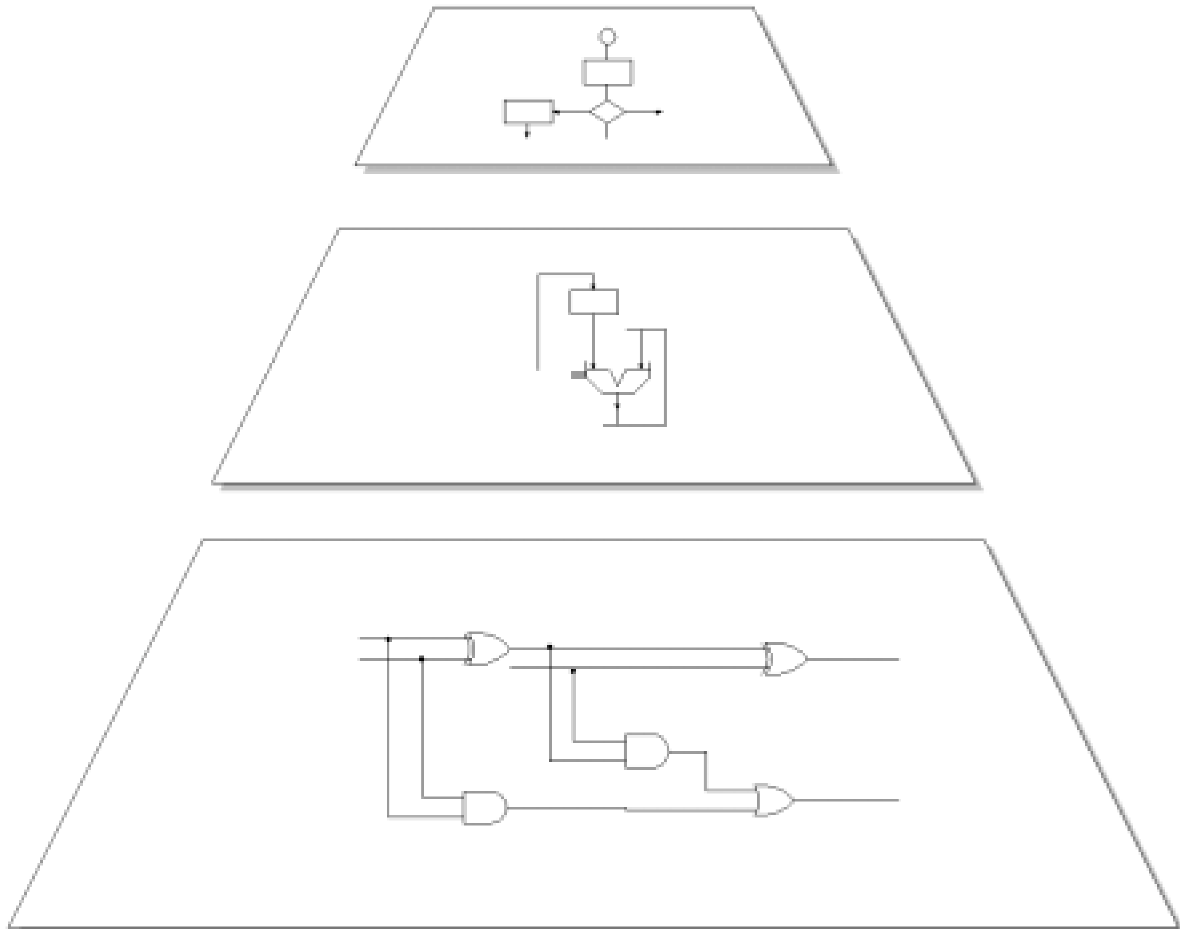


Figura 2.3 Técnica de Projeto Top-down.

Projetos em alto nível são independentes da tecnologia de fabricação e também podem ser reutilizados. Um projetista pode inicialmente ter como tecnologia alvo um FPGA, por exemplo, para em seguida migrar para um ASIC.

A automação de projeto em alto nível inclui ferramentas para entrada do projeto, simulação em alto nível e em nível de portas lógicas, sínteses comportamental e lógica, verificação, e geração

de vetores de teste. A figura 2.4 apresenta um panorama geral da metodologia de projeto em alto nível.

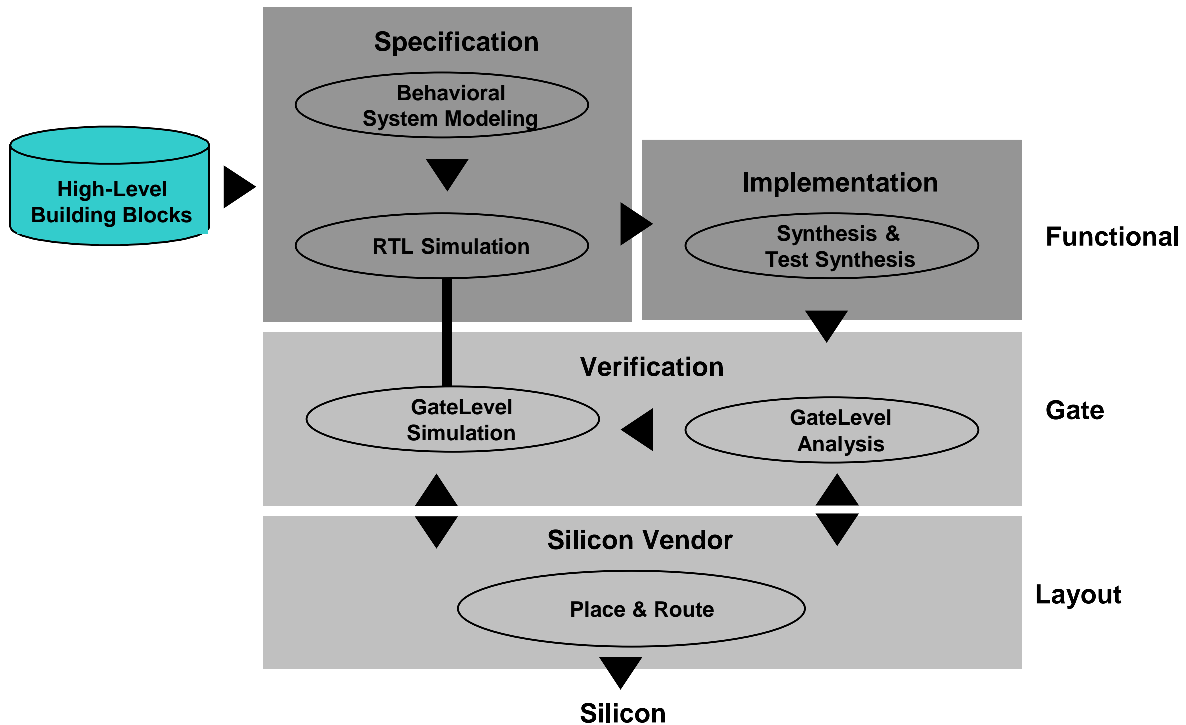


Figura 2.4 Etapas do Projeto de Hardware com a Metodologia de Projeto em Alto Nível.

2.3.1. Entrada do Projeto

Do mesmo modo que a metodologia de projeto tradicional, o projeto em alto nível se inicia com as especificações do projeto baseadas em função, qualidade (cobertura a falha), custo, limitações de tempo e área. Mas ao invés de construir o circuito utilizando somente a entrada esquemática em nível de portas lógicas, utiliza-se também linguagens de descrição de *hardware*, tais como Verilog ou VHDL.

A maioria dos projetos em nível de portas lógicas é capturada via um editor esquemático, enquanto alguns contam com um editor de texto para criar um *netlist*[1]. A captura de projetos de

[1]Uma *netlist* é a representação textual de um projeto. Ela contém uma descrição mais estruturada da funcionalidade do projeto. Por exemplo, quais componentes estão conectados e com quais entradas e saídas. Essa representação remove qualquer redundância associada com a representação em forma de esquemático.

alto nível, por outro lado, conta com uma mistura de entrada do tipo esquemática, para partes estruturais do projeto (interconexões entre blocos), e entrada textual para as funções no interior dos blocos.

2.3.2. Linguagens de Descrição de *Hardware* - HDLs

A captura de esquemático não é o modo ideal para projeto de grandes circuitos, e é uma tarefa muito tediosa quando se projeta grandes blocos regulares repetidamente. À medida que os projetos ficam mais complexos, as descrições em nível de portas tornam-se inviáveis, fazendo com que seja necessário descrever os projetos em modos mais abstratos

As linguagens de descrição de *hardware* (HDLs) foram desenvolvidas para auxiliar os projetistas a documentarem projetos e simularem grandes sistemas, principalmente em projetos de ASIC, mas também são adequadas para projetos em outras tecnologias, como os FPGAs (Donlin, 96; Harding, 90).

Há muitas linguagens de descrição de *hardware* disponíveis. As duas mais comumente usadas são a VHDL e a Verilog. Essas linguagens permitem a descrição comportamental de circuitos para serem sintetizados (compilados) em circuitos destinados a diferentes tecnologias. Isto é feito convertendo-se a descrição comportamental (como uma adição de duas variáveis) em um circuito contendo os blocos corretos para conseguir esse comportamento (como um somador conectado a dois registradores). Como o projeto é especificado de forma comportamental, não há uma arquitetura específica e a maioria das ferramentas de síntese são capazes de produzir *netlist* para diferentes tecnologias de projetos (ASIC, MPGA, FPGA, etc.) a partir de uma única especificação. Ou seja, as grandes vantagens das HDL, são sua independência da tecnologia e portabilidade, além de uma robustez matemática que as linguagens de programação propiciam.

2.3.2.1. A Linguagem VHDL

A linguagem VHDL surgiu como um ramo do programa VHSIC (*Very-high-speed integrated circuits*) que foi fundado pelo Departamento de Defesa dos Estados Unidos, no final dos anos 70 e começo dos anos 80. O objetivo do programa VHSIC era produzir a próxima geração de

circuitos integrados. Os participantes do programa foram desafiados a expandir os limites da tecnologia em cada fase do projeto e fabricação de circuitos integrados (Perry, 91).

A nova linguagem de descrição de *hardware* foi proposta em 1981 e chamada de *VHSIC Hardware Description Language*, ou como é atualmente conhecida, VHDL. Os objetivos dessa nova linguagem foram dois. Primeiro, os projetistas queriam uma linguagem que pudesse descrever os circuitos complexos que estavam tentando projetar. Segundo, eles queriam uma linguagem que fosse um padrão para que todos os participantes do programa VHSIC pudessem distribuir projetos entre eles num formato padrão.

Em 1986, a VHDL foi proposta como um padrão IEEE. Foram feitas várias revisões e mudanças até ser adotada como o padrão IEEE 1076 em dezembro de 1987. Atualmente, em sua nova versão (IEEE 1076-1993) (IEEE, 94), a linguagem VHDL é a linguagem utilizada como padrão da indústria para a descrição de *hardware* num nível abstrato (Coelho, 95; Lispsett, 93; Perry, 91; Ashenden, 1996). Os desenvolvedores de *software* EDA estão padronizando a entrada e saída de suas ferramentas em VHDL, que incluem ferramentas de simulação, de síntese, de *layout* e etc. Um pequeno exemplo de código em VHDL é apresentado na figura 2.5.

```
entity mux41 is
     port (SEL: in STD_LOGIC_VECTOR(1 downto 0);
          A,B,C,D: in STD_LOGIC;
          MUX_OUT: out STD_LOGIC);
end mux41;

architecture BEHAV of mux41 is
begin
     IF_PRO: process (SEL,A,B,C,D)
     begin
          if    (SEL="00") then    MUX_OUT <= A;
          elsif (SEL="01") then    MUX_OUT <= B;
          elsif (SEL="10") then    MUX_OUT <= C;
          elsif (SEL="11") then    MUX_OUT <= D;
          else                     MUX_OUT <= '0';
     end if;
end process; -- END IF_PRO

end BEHAV;
```

Figura 2.5 Código VHDL de um Multiplexador 4-1

2.3.2.2. Verilog HDL

Antes da iniciativa de padronização do VHDL, o único sério concorrente como uma HDL padrão da indústria foi o Verilog HDL (Harding, 90). Foi desenvolvida, pela Divisão de Engenharia Avançada da Cadence Design Systems, como uma linguagem de alto nível para uso com o simulador Verilog-XL. A família Verilog conquistou um grande número de desenvolvedores de ASIC devido ao seu excelente simulador.

2.3.2.3. A Aceitação das HDLs como Ferramentas de Projeto

Muitos engenheiros ainda usam esquemáticos para descrição de projetos e sentem-se desconfortáveis com a mudança para linguagens de alto nível devido a falta de conhecimentos de conceitos de programação concorrente. Porém, com o crescente aumento da complexidade dos projetos, essa transição parece ser inevitável. Projetos com mais de 8000 portas são inviáveis utilizando-se apenas esquemáticos (Donlin, 96b).

Para amenizar esse processo de transição, os fabricantes de ferramentas EDA estão desenvolvendo ferramentas gráficas para geração de código HDL a partir de esquemáticos e máquinas de estado, que facilitam o aprendizado da linguagem (Donlin, 96a). Além dessas ferramentas, os fabricantes já oferecem bibliotecas com modelos pré-definidos das funções mais utilizadas.

A eficiência do código gerado é discutível. Muitos projetistas consideram o código desenvolvido manualmente por engenheiros experientes mais eficiente, mesmo porque há uma dependência da tecnologia alvo. Por mais automáticas que pareçam, alguns engenheiros ainda precisarão modificar o projeto para satisfazer restrições, principalmente quando há requisitos de desempenho. Entretanto, essas ferramentas podem ser usadas como tutoriais por projetistas inexperientes ou facilitar a escrita do código HDL para os projetistas experientes. Algumas delas também auxiliam o gerenciamento e a comunicação de idéias durante o ciclo de projeto. Além disso, são apropriadas para acelerar o projeto quando o *time-to-market* é um fator importante.

2.3.3. Síntese

A síntese consiste de três processos distintos: síntese comportamental, síntese HDL, e síntese lógica.

2.3.3.1. Síntese Comportamental

Constrói o *datapath*, memória, e elementos de controle pelo uso de métodos automáticos de escalonamento, alocação de *hardware*, compartilhamento de registrador, memória, e controle de inferência — tarefas que o projetista costumava executar manualmente utilizando a metodologia RTL. O projetista é liberado de especificar a arquitetura exata de um projeto, podendo explorar muitas implementações alternativas para obter uma arquitetura otimizada.

2.3.3.2. Síntese HDL

Converte a entrada HDL (tanto VHDL como Verilog) em equações booleanas não otimizadas ou em um *netlist* otimizado. Esse processo analisa a linguagem, verifica a sintaxe e a semântica, propaga constantes, elimina código extinto, aloca e compartilha recursos.

2.3.3.3. Síntese Lógica

Converte equações lógicas em um *netlist*, e então o otimiza para uma tecnologia em particular. A síntese lógica consiste de duas fases distintas:

- Otimização lógica para minimizar equações Booleanas.
- Mapeamento na tecnologia para converter equações em células da biblioteca da tecnologia alvo.

Ambas as fases são direcionadas por limitações de área e velocidade. A primeira etapa no processo de síntese é criar limitações de tempo e de área. O processo de criação das limitações em síntese é semelhante ao de desenvolver estímulos para a simulação. Limitações de tempo incluem caminho crítico, período de *clock*, condições de *setup* e de *hold*, etc. As limitações criadas podem levar em conta o ambiente, incluindo fatores como condições de operação e temperatura.

A otimização em nível de portas lógicas executada pela ferramenta de síntese requer suporte para uma grande variedade de bibliotecas de tecnologias. A figura 2.6 ilustra a etapa de síntese a partir de um código em VHDL.

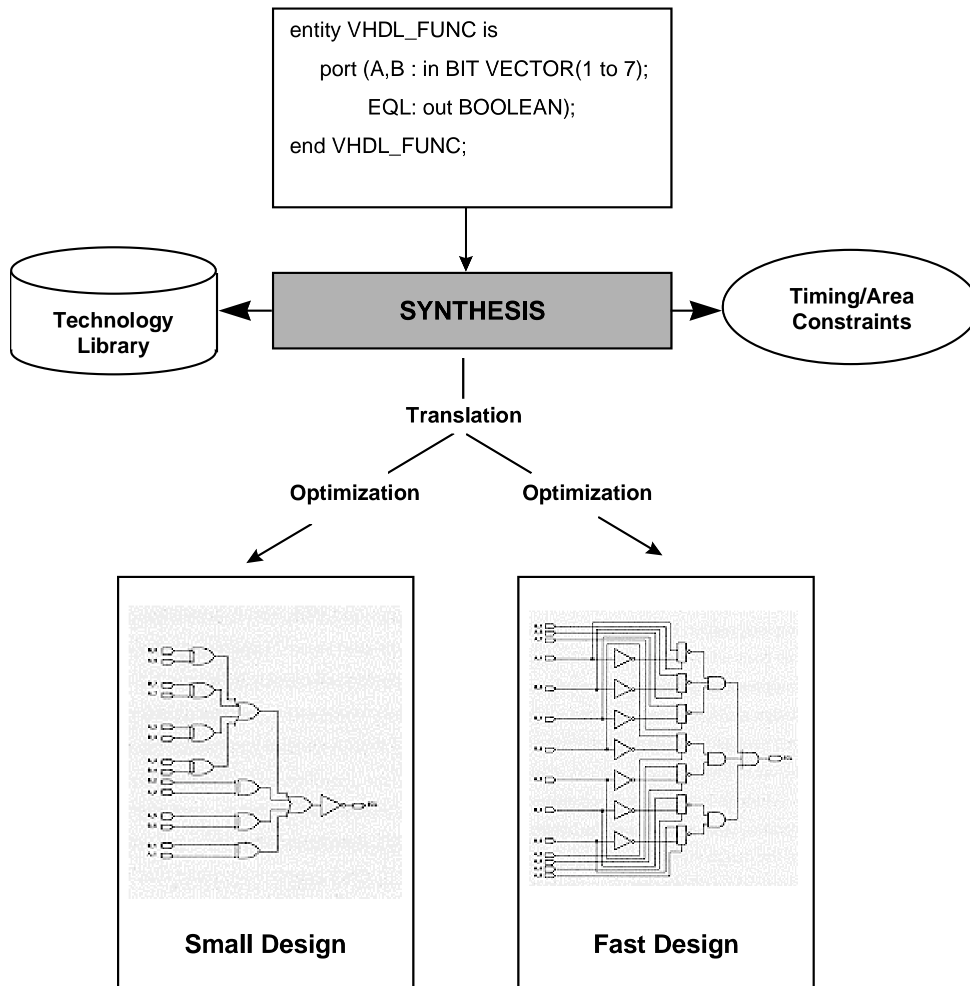


Figura 2.6 Síntese Lógica.

Por causa do poder e velocidade da tecnologia de síntese, pode-se analisar alternativas de implementação, medindo-se os efeitos das mudanças na área e na velocidade. Em contraste, utilizando-se métodos em nível de portas lógicas, há raramente tempo suficiente para ir além de um único caminho de implementação.

2.3.4. Simulação

A simulação é um processo que imita a funcionalidade ou comportamento de um projeto digital num computador. É utilizada principalmente para identificar possíveis erros no projeto ou

problemas de atrasos no circuito. Originalmente, a simulação era usada para prototipação, mas agora também é muito utilizada para depurar e validar um projeto.

O projeto em nível de portas lógicas combina validação e verificação. No projeto em alto nível esses processos são tratados como duas atividades separadas. A separação da validação da funcionalidade e da verificação do desempenho leva a ciclos de simulação menores, reduzindo assim o tempo global do projeto quando se utiliza projeto em alto nível.

Pode-se validar a funcionalidade de um projeto via simulação de HDL, logo após a etapa de descrição. Já a verificação da velocidade e área pode ser feita somente depois da criação de uma implementação em nível de portas lógicas, realizada pela ferramenta de síntese. A verificação pós-síntese examina também a funcionalidade. Aliás, a maior parte do trabalho feito nesse estágio enfocará a verificação funcional, porque as ferramentas de síntese automatizam muito a análise da velocidade e da área. No projeto em alto nível as ferramentas devem ser capazes de comparar a descrição HDL e sua implementação em nível de portas lógicas gerada pela ferramenta de síntese, verificando a equivalência funcional.

Como a simulação tradicional, a simulação de projetos em alto nível também envolve projeto iterativo e procedimentos de *debug*, mas em um alto nível de abstração. Para facilitar o *debugging*, bons relatórios de erro são essenciais; as mensagens devem identificar de maneira clara as áreas de problemas no projeto original. Sempre que necessário o projetista deve retornar a etapa inicial de descrição do projeto. A figura 2.7 ilustra esse processo em que são necessárias várias iterações de simulação até atingir a etapa de verificação final.

Os resultados da simulação no ambiente de projeto em alto nível podem ser visualizados em vários formatos.

- Gráficos de Forma de Onda. Mostram os resultados graficamente conforme eles são determinados pelo simulador.

- Listagens Tabulares. Boa para visualizar dutos e simulações baseadas em ciclos de *clock*. Deve-se ser capaz de customizar o formato das listagens para alcançar os próprios requisitos e salvar o arquivo de resultados para usar posteriormente.

- Amostras Especializadas. Normalmente específicas do projeto, as amostras especializadas são muito úteis para analisar comportamentos complexos.

- Amostras Interativas. Permite a visualização dos sinais diretamente no esquemático. A amostra no esquemático está ligada ao visualizador de forma de onda, apresentando os resultados dos sinais em um certo instante.

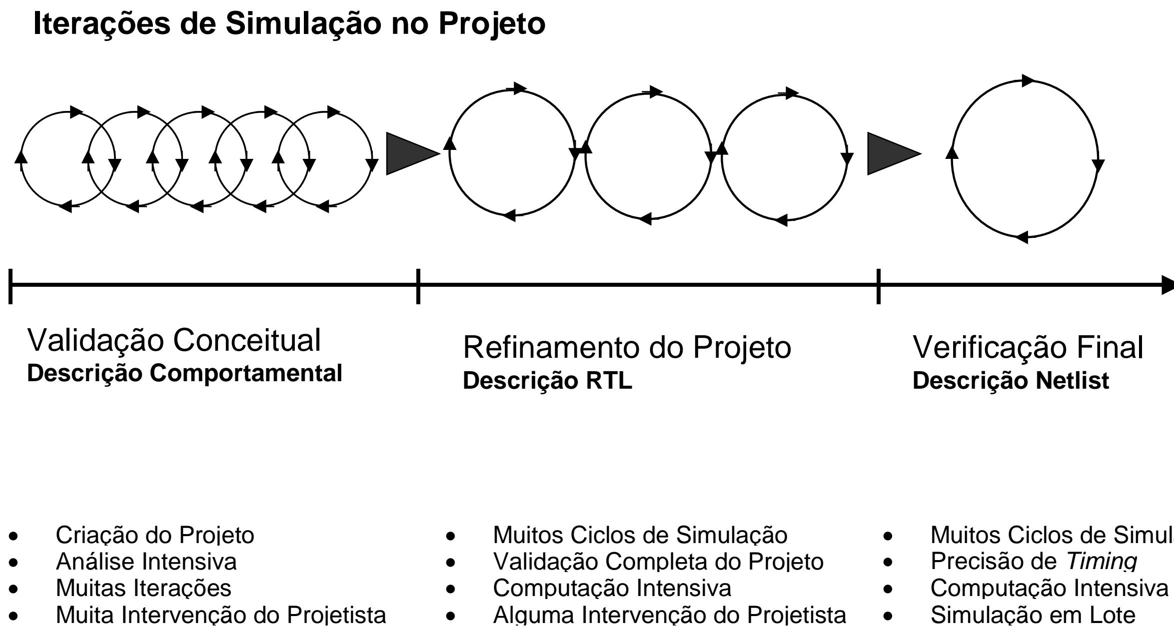


Figura 2.7 Processo de Simulação com várias iterações ao longo do projeto.

2.3.5. Síntese de Testes

Ferramentas de síntese de testes automatizam o processo de maneira que se possa conseguir testabilidade no projeto, à medida que ele está sendo criado. A síntese de testes possibilita minimizar o custo dos testes durante o ciclo de projeto e evitar o reprojeto. Além disso, as ferramentas de síntese de testes proporcionam a geração automática de padrões de teste (ATPG) para obter uma alta cobertura de falhas para o projeto em teste.

A síntese de testes modifica automaticamente os circuitos visando a testabilidade enquanto conserva os objetivos de desempenho e de área. A síntese de testes também faz o seguinte:

- Remove redundâncias lógicas do projeto

- Identifica possíveis lógicas não testáveis

- Dá advertência em concordância com as regras
- Insere estruturas de testabilidade automaticamente
- Re-otimiza o projeto para área e velocidade
- Gera vetores de teste

Como resultado temos projetos amplamente testáveis e de alta qualidade.

2.4. Considerações Finais

As ferramentas de projeto de alto nível estão proporcionando aos projetistas de sistemas um novo patamar de produtividade e independência tecnológica. As vantagens dessa nova metodologia são muitas e evidentes. Com essas ferramentas é possível manipular com facilidade projetos mais complexos. A metodologia de alto nível permite explorar alternativas de implementação mais rapidamente, à medida que as ferramentas de *software* executam as tarefas tediosas e iterativas. Os projetistas podem tomar melhores decisões, corrigir defeitos antes da fabricação, e tornar seus projetos mais portáveis e reutilizáveis. Conseqüentemente, o ciclo de projeto é encurtado de forma significativa, possibilitando um rápido lançamento de produtos no mercado .

A transição para essa metodologia geralmente é realizada em uma série de etapas. Os projetistas começam pelo uso de ferramentas de projeto em alto nível em uma porção pequena do projeto, algo modesto e manipulável. À medida que eles se tornam mais hábeis com as ferramentas e confiantes que os resultados são comparáveis — e muitas vezes superiores — aos que eles obtinham fazendo o projeto tradicional em nível de portas lógicas, eles utilizam as metodologias de projeto em alto nível em porções cada vez maiores de seus sistemas (Synopsys, 94).

Deve-se destacar a importância das HDLs, especialmente a linguagem VHDL que deve se consolidar como o método padrão para o desenvolvimento de projetos de *hardware*. Os esquemáticos em nível de portas lógicas estão dando lugar às linguagens de descrição de *hardware* do mesmo modo que as linguagens de programação montadoras deram lugar às linguagens de programação de alto nível. Realmente, as HDL irão fazer para os projetistas de

hardware o mesmo que C, Pascal e Ada fizeram para os desenvolvedores de *software* (Harding, 90).

A síntese lógica de descrições comportamentais apresenta ainda um campo vasto de pesquisa. Há necessidade do aprimoramento de suas técnicas, condição crucial para a ampliação e consolidação do projeto em alto nível.

3. A Tecnologia FPGA

3.1. Evolução dos Dispositivos Lógicos Programáveis

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 lógico e as linhas de dados como saídas. As funções lógicas, entretanto, raramente requerem mais que alguns termos produto, e uma PROM contém um decodificador completo para seus endereços de entradas. As PROMs, portanto, são uma arquitetura ineficiente para a realização de circuitos lógicos, e são raramente utilizadas na prática para esse fim (Brown, 96a).

O primeiro dispositivo desenvolvido a seguir especificamente para a implementação de circuitos lógicos foram os PLAs (*Programmable Logic Arrays*). Um PLA consiste de dois níveis de portas lógicas: um plano de portas wired-AND seguido por um plano de portas wired-OR, ambos programáveis. Uma PLA é estruturada de tal forma que cada saída do plano AND pode corresponder a qualquer termo produto das entradas. Similarmente, cada saída do plano OR pode ser configurada para produzir a soma lógica de quaisquer saídas do plano AND. Com essa estrutura, os PLAs são bem adequados para a 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.

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 dois 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. Para compensar a falta de generalidade devido ao plano OR fixo, diversos modelos de PALs são produzidos, com diferentes número de entradas e saídas, e vários tamanhos de portas OR.

Os PALs geralmente contêm flip-flops conectados às saídas das portas OR para que circuitos sequenciais possam ser feitos. Dispositivos PAL são 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 mais importantes características são 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 na arquitetura de SPLDs é integrar múltiplos SPLDs em um único *chip* e prover interconexões programáveis para conectar os blocos SPLDs. Muitos produtos PLDs comerciais existem no mercado atualmente com essa estrutura básica, e são coletivamente chamados de CPLDs (*Complex PLDs*).

Os CPLDs foram introduzidos pela Altera Corp., inicialmente com sua família de *chips* chamada Classic EPLDs (*Erasable PLDs*), e em seguida com três séries adicionais, chamadas MAX5000, MAX7000 e MAX9000. Devido ao rápido crescimento do mercado para grandes PLDs, outros fabricantes desenvolveram dispositivos na categoria CPLD e há atualmente muitas opções disponíveis. Os CPLDs provêem 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.

Os MPGAs motivaram o projeto de dispositivos programáveis equivalentes: os FPGAs (*Field-Programmable Gate Arrays*). Como MPGAs, os FPGAs incluem um arranjo de elementos de circuito não conectados, chamados blocos lógicos, e recursos de interconexão, mas a configuração do FPGA é realizada através de programação pelo usuário final. Sendo o único tipo de PLD que suporta capacidade lógica muito alta, 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 companhia chamada Xilinx Inc. surgindo no mercado em 1985, e 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 desses dispositivos programáveis de alta capacidade (Brown 92; Tuck, 92a).

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 serão apresentados os aspectos mais relevantes relativos à arquitetura de FPGAs, detalhando suas principais características e alguns modelos comercialmente disponíveis. Também será explanado o processo de desenvolvimento de projetos utilizando esta tecnologia.

3.2. Arquitetura de FPGAs

Um *Field Programmable Gate Array* (FPGA) consiste de um um grande arranjo de células configuráveis (ou blocos lógicos) contidos em um único *chip*. Cada uma dessas células contém uma capacidade computacional para implementar funções lógicas e/ou realizar roteamento para permitir a comunicação entre as células. Todas essas operações podem acontecer simultaneamente no arranjo de células inteiro (Rose et al., 93; Brown & Rose, 96; Donachy, 96).

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 eletricamente, assim como os tradicionais dispositivos lógicos programáveis. Os FPGAs combinam a versatilidade dos *gate arrays* e a capacidade de programação dos PLDs (Chan & Mourad, 94).

A arquitetura básica de um FPGA, ilustrada na figura 3.1, consiste de arranjo 2-D de blocos lógicos. A comunicação entre os 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).

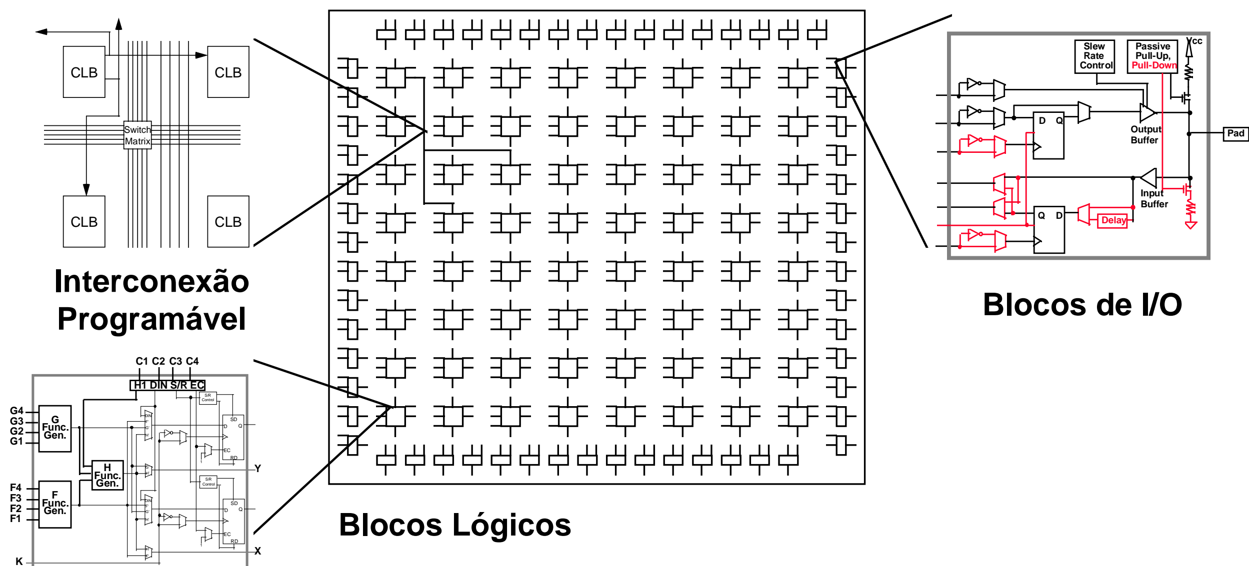


Figura 3.1 Estrutura de um FPGA.

Há três aspectos principais que definem a arquitetura de um FPGA: o tipo de tecnologia de programação utilizada, a arquitetura dos blocos lógicos e a estrutura de sua arquitetura de roteamento. Esses 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 (Rose et al., 93).

As próximas seções descrevem com maiores detalhes as diferentes tecnologias de programação, complexidades dos blocos lógicos, arquiteturas de roteamento e os principais modelos de FPGAs comercialmente disponíveis.

3.2.1. Tecnologias de Programação

Um FPGA é programado usando comutadores programáveis eletricamente. As propriedades desses comutadores tais como tamanho, resistência, capacitância, tecnologia afetam principalmente a 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 maiores 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:

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

3.2.1.1. A Tecnologia de Programação SRAM

A tecnologia de programação SRAM, ilustrada na figura 3.2, usa uma célula de RAM estática para controlar transistores de passagem ou multiplexadores. Comercialmente, essa tecnologia é utilizada pela Xilinx, Altera e AT&T.

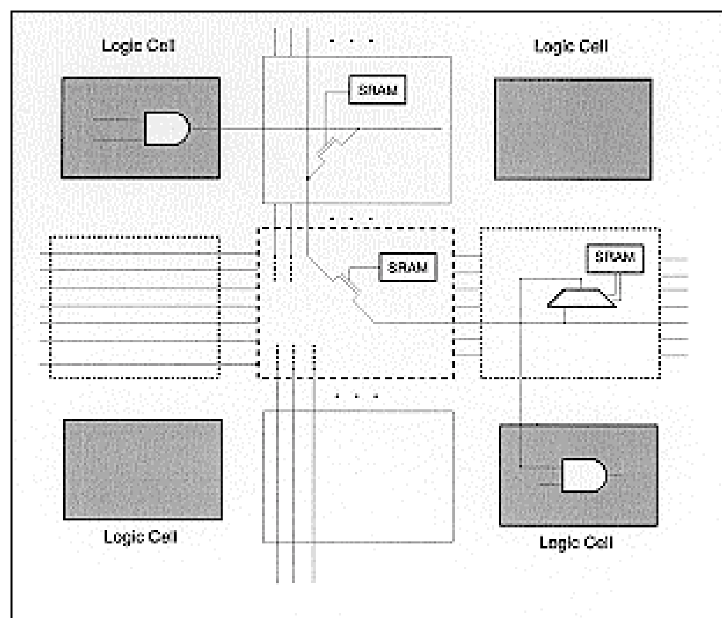


Figura 3.2 Tecnologia de Programação SRAM.

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*: para cada comutador estão associados pelo menos 6 transistores. No entanto, a SRAM apresenta duas grandes vantagens: é rapidamente reprogramável e requer apenas a tecnologia padrão de circuitos integrados para a sua fabricação.

3.2.1.2. A Tecnologia Antifuse

Usado pela Actel e Quicklogic e recentemente pela Xilinx, 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.As vantagens do *antifuse* são o seu tamanho reduzido, por consequência, sua baixa capacitância quando não programado e a baixa resistência quando programado. Porém, o *antifuse* não permite a reprogramação. Além disso, para programá-los é necessária a presença de transistores com larga área a fim de suportar as altas correntes de programação (~5mA). Finalmente, é necessário gastar um espaço extra para conseguir a isolação dos circuitos de programação, pois os mesmos trabalham com tensão de até 20 V. A figura 3.3 abaixo mostra a estrutura do *antifus*e PLICE da Actel (ACTEL 95,Brown & Rose, 96).

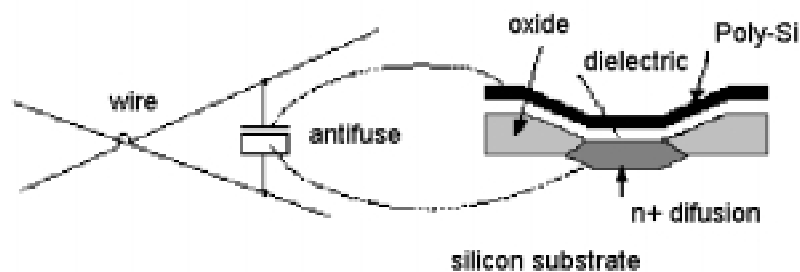


Figura 3.3 **Antifuse** ***PLICE da Actel.***

3.2.1.3. A Tecnologia de Gate Flutuante

Nessa tecnologia, os comutadores programáveis são baseados em transistores com *gate* flutuante iguais aos usados nas memórias EPROM (*Erasable PROM*) e EEPROM (*Electrical Erasable*

PROM). Comercialmente, a EPROM é usada pela Altera e a EEPROM é usada pela AMD e Lattice. A figura 3.4 ilustra um comutador programável do tipo EPROM.

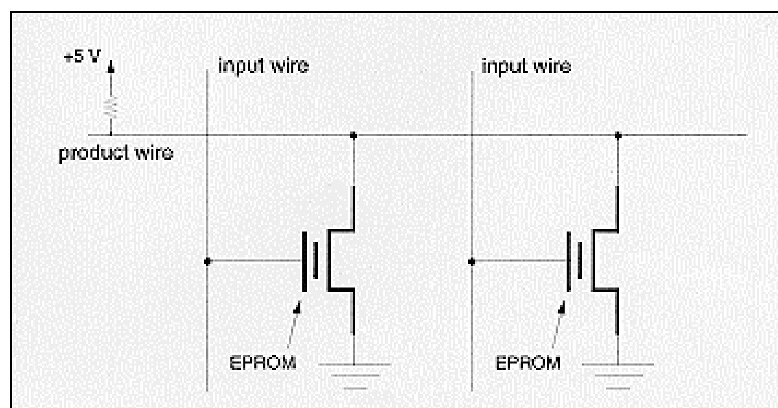


Figura 3.4 Comutador Programável baseado em EPROM.

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 Programmability*).

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*.

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

Tabela 3.1 Sumário das Tecnologias de Programação.

Nome	Reprogramação	Volatilidade	Tecnologia
EPROM	sim, fora do circuito.	não	UVCMOS
EEPROM	sim, no circuito	não	EECMOS
SRAM	sim, no circuito	sim	CMOS
Antifuse	não	não	CMOS+

Na tabela 3.1 encontra-se resumidas as mais importantes características das tecnologias de programação apresentadas.

3.2.2. 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 implementar 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-OR de múltiplas entradas.

A fim de classificar os FPGAs quanto ao bloco lógico, foram criadas duas categorias: a granulosidade fina e a granulosidade grossa. A primeira categoria designa os blocos simples e pequenos e a segunda os blocos mais complexos e maiores (Rose et al., 93).

3.2.2.1. Blocos de Granulosidade Fina

O melhor exemplo para um bloco de granulosidade fina seria um bloco contendo alguns transistores interconectáveis ou portas lógicas básicas (por ex. NAND). A principal vantagem no uso de blocos com granulosidade fina é que estes são quase sempre totalmente utilizados. 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 conexão e comutadores programáveis. Um roteamento desse tipo de FPGA se torna lento e ocupa grande área no *chip*.

3.2.2.2. Blocos de Granulosidade Grossa

Geralmente são baseados em multiplexadores ou *look-up tables*. Os blocos lógicos baseados em multiplexadores têm a vantagem de fornecer um alto grau de funcionalidade com um número relativamente pequeno de transistores. No entanto, eles possuem muitas entradas necessitando de muitos comutadores, o que sobrecarrega o roteamento. Logo, a tecnologia *antifuse* é mais adequada para a fabricação desse tipo de FPGA, devido ao tamanho reduzido dos comutadores *antifuse*.

Uma *look-up table* (LUT) é uma pequena memória de um bit de largura, onde suas linhas de endereçamento funcionam como entradas do bloco lógico e sua saída fornece o valor da função lógica. A tabela verdade para uma função de *k* entradas é armazenada em uma SRAM de 2^K x 1. 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. A desvantagem é que são inaceitavelmente grandes para mais que cinco entradas. Apesar do número de funções que podem ser implementadas aumentar muito rapidamente com o aumento do número de entradas, essas funções adicionais não são geralmente utilizadas em projetos lógicos e são difíceis de serem manipuladas por uma ferramenta de síntese lógica.

3.2.2.3. Lógica Sequencial

A maioria dos blocos lógicos apresenta 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.

3.2.3. Arquitetura de Roteamento

A arquitetura de roteamento de um FPGA é a maneira pela qual os comutadores programáveis e segmentos de trilha são posicionados para permitir a interconexão das células lógicas. As arquiteturas de roteamento podem ser descritas a partir do modelo geral (Rose et al.,93),

conforme a figura 3.5. São necessários alguns conceitos para um melhor entendimento desse modelo:

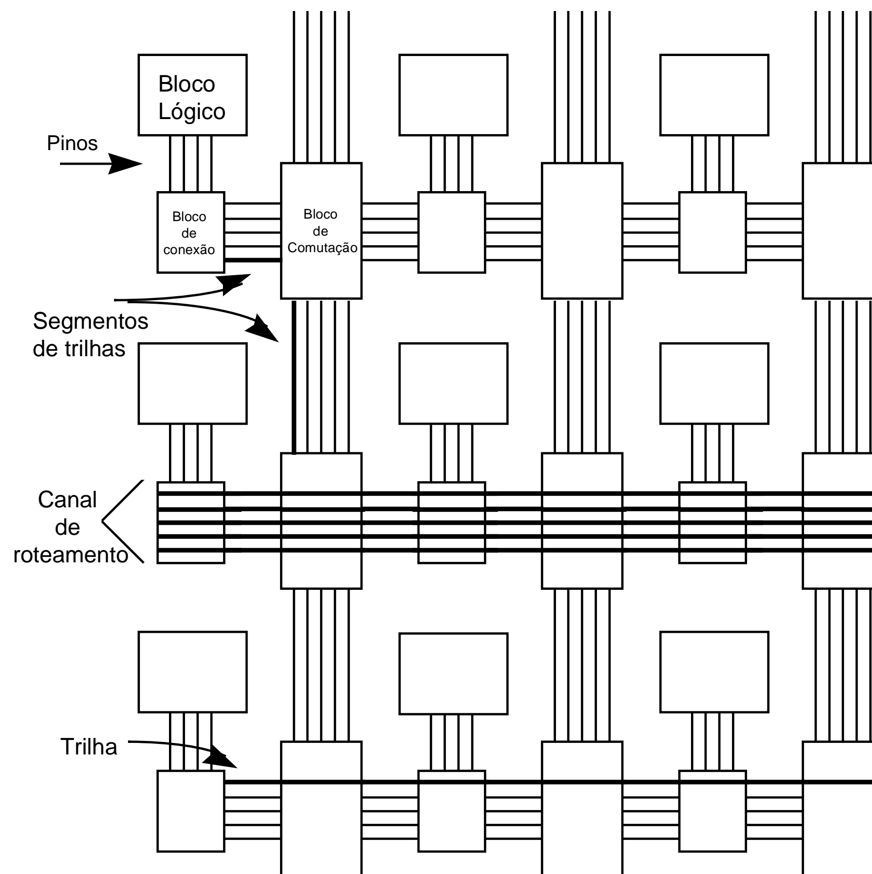


Figura 3.5 Arquitetura Geral de Roteamento de um FPGA.

- Pinos: são as entradas e saídas dos blocos lógicos.
- Conexão: ligação elétrica de um par de pinos de blocos lógicos.
- Rede: é um conjunto de pinos de blocos lógicos que estão conectados. Uma rede pode ser composta de uma ou mais conexões.
- Comutador de roteamento (*switch*): é um dispositivo utilizado para conectar eletricamente dois segmentos de trilha.

- Segmento de trilha: é um segmento não interrompido por comutadores programáveis. Cada terminação de um segmento possui um comutador associado.

- Trilha: é uma sequência 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.

- Canal de roteamento: é a área entre duas linhas ou colunas de blocos lógicos. Um canal contém um grupo de trilhas paralelas.

O modelo contém duas estruturas básicas. A primeira é o bloco de conexão que aparece em todas as arquiteturas. O bloco de conexão permite a conectividade das entradas e saídas 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 conexão são distintos, em outras estão combinados numa mesma estrutura. Nem todas as arquiteturas seguem esse modelo: há arquiteturas que apresentam uma estrutura hierárquica e outras que possuem somente canais horizontais, como veremos na seção seguinte.

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 estabeleceram uma relação entre a flexibilidade da arquitetura de roteamento, a capacidade de roteamento e uso eficiente da área (Rose et al., 93).

3.2.4. Arquiteturas Comerciais de FPGAs

Atualmente existem quatro categorias principais de FPGAs comercialmente disponíveis: as de arranjo simétrico, as baseadas em linhas, as hierárquicas e as de estrutura conhecida como mar de portas (*sea-of-gates*), como ilustra a Figura 3.6.

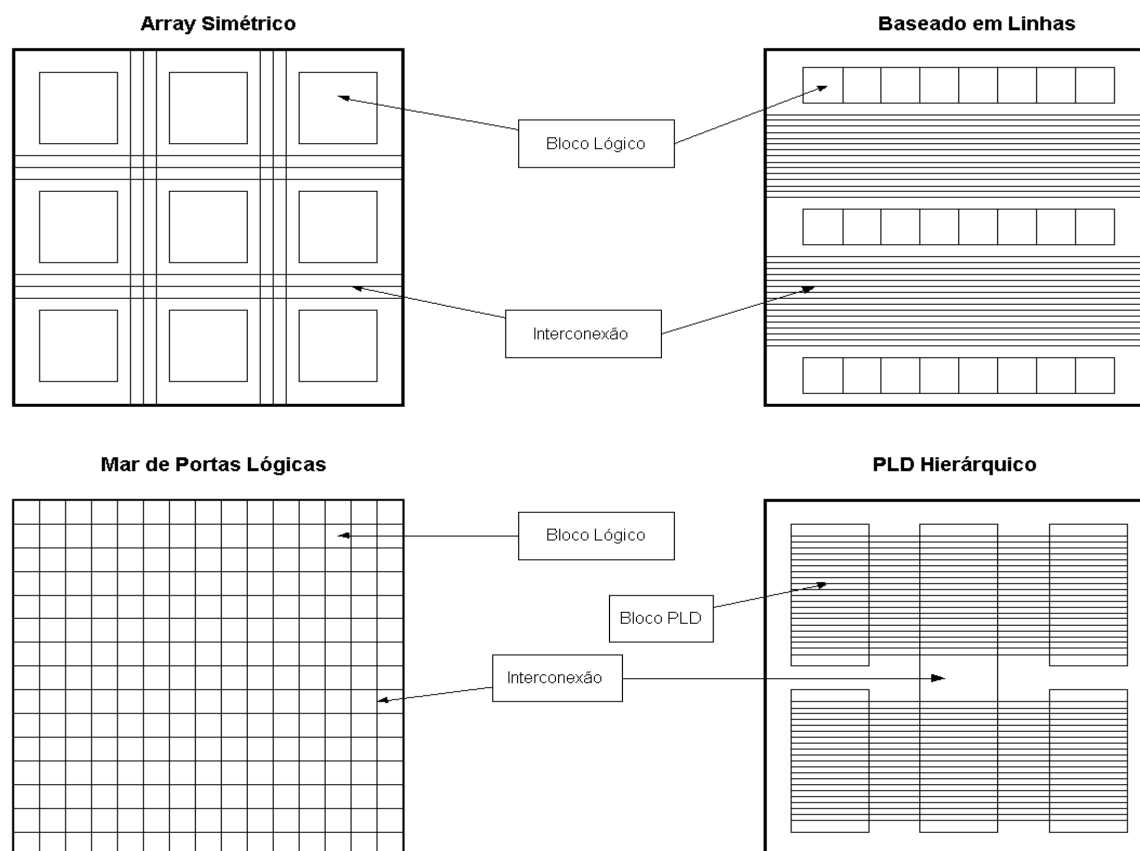


Figura 3.6 Arquiteturas Comercialmente Disponíveis.

Em todas esses FPGAs a estrutura de roteamento, os blocos lógicos e a forma de programação variam. Dependendo da aplicação, uma arquitetura de FPGA pode ter características mais desejáveis para a aplicação em questão. O mercado oferece uma ampla gama de dispositivos de diversos fabricantes, entretanto iremos descrever apenas a arquitetura dos FPGAs da Xilinx, Altera e Actel, para exemplificar os aspectos arquiteturais discutidos.

3.2.4.1. Os FPGAs da Xilinx

A estrutura básica das principais famílias de FPGAs da Xilinx é baseada em *array* e é conhecida como LCA (Logic Cell Array). Cada *chip* inclui um arranjo 2-D de blocos lógicos que podem ser interconectados através de canais horizontais e verticais, como foi ilustrado na figura 3.1.

A Xilinx lançou a primeira família de FPGAs, chamada série XC2000, em 1985 e a cada ano oferece novas gerações, introduzindo inovações tecnológicas, que aumentam a capacidade lógica e a velocidade dos dispositivos que já ultrapassam o patamar de 500.000 portas lógicas (XIL98). A seguir, descreveremos somente sua família mais popular, a XC4000 e a família XC6200, que possui capacidade de reconfiguração dinâmica.

O bloco lógico da Xilinx, chamado de CLB (Configurable Logic Block), é baseado em uma memória do tipo SRAM funcionando como uma look-up table (LUT). O CLB da série XC4000 contém duas LUTs de quatro entradas que alimentam uma LUT de três entradas, como mostra a figura 3.7. Nesse bloco, todas as entradas são distintas e disponíveis externamente ao bloco, possibilitando a implementação de muitas funções lógicas de até 9 entradas, duas funções separadas de 4 entradas ou outras possibilidades. Cada CLB também contém dois flip-flops (Xilinx, 98).

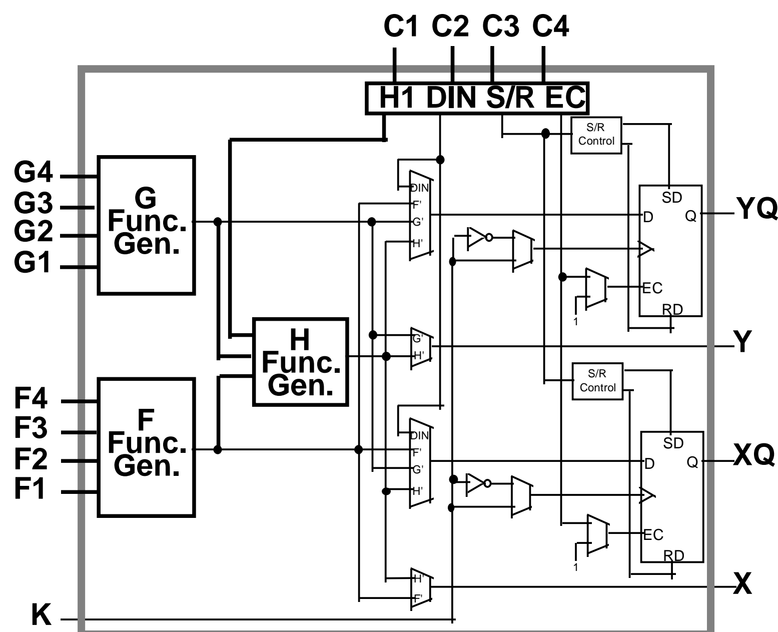


Figura 3.7 Bloco Lógico da família XC4000.

Nesse bloco, dois diferentes tamanhos de LUT são utilizados: 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 significantemente 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.

O bloco 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 configuradas como células RAM, para a implementação 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 como decodificadores. Os dispositivos dessa família variam em capacidade lógica de 3000 atingindo atualmente 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 arranjados em canais verticais e horizontais. Cada canal contém uma quantidade de segmentos de trilhas curtos que se estendem por apenas um CLB, segmentos duplos que atravessam dois CLBs e segmentos longos que atravessam a largura ou altura inteira do *chip*, como mostram as figuras 3.8 e 3.9. 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.

Um importante ponto a ser notado 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 utilizado. Assim, o desempenho de um circuito implementado depende em parte de como os segmentos de trilha são alocados aos sinais individuais pelas ferramentas EDA.

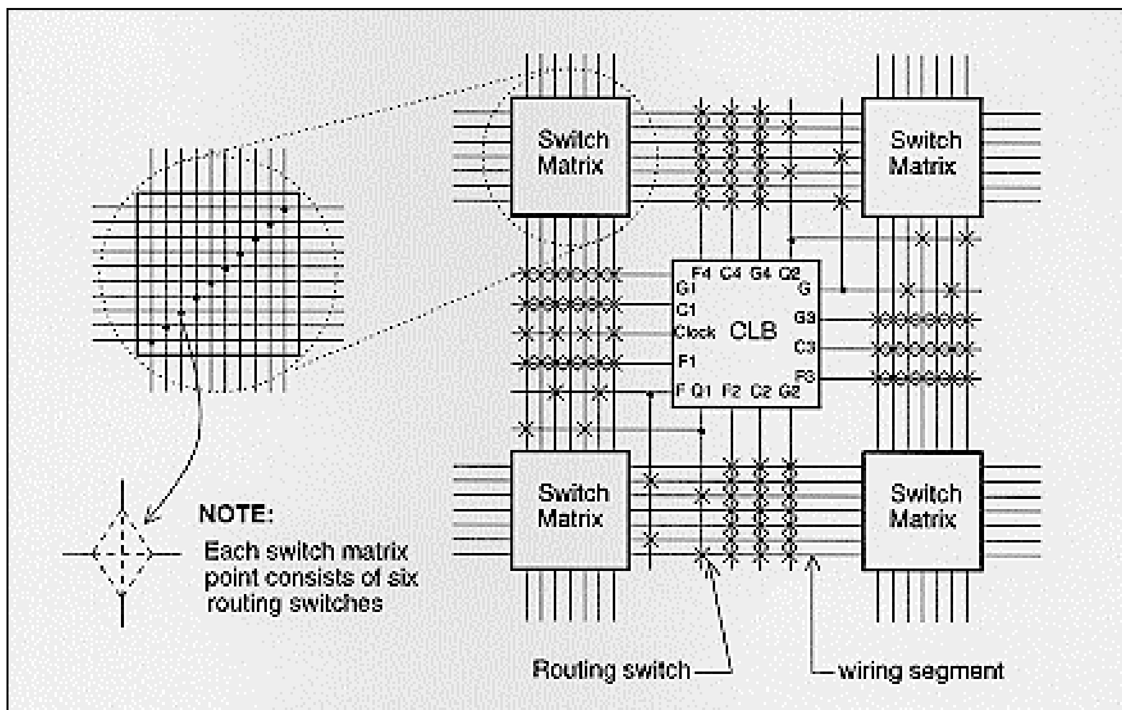


Figura 3.8 Segmentos simples e comutadores de roteamento da série XC4000.

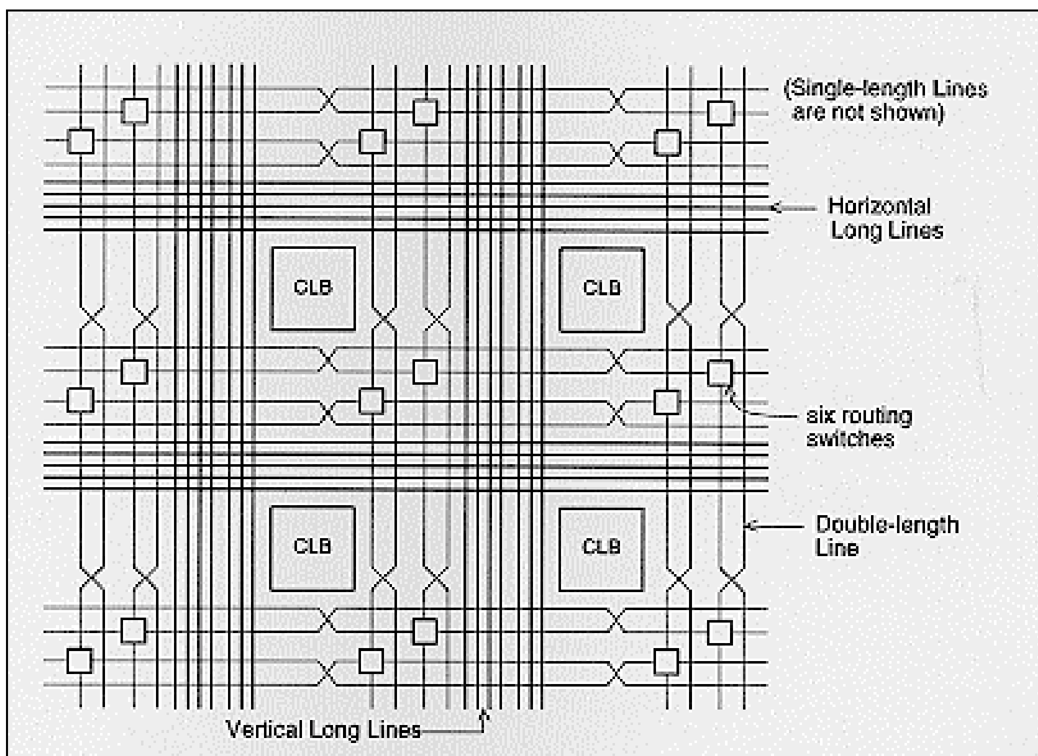


Figura 3.9 Segmentos duplos e linhas longas da série XC4000.

Uma outra família recentemente lançada é a XC6200 (Xilinx, 98). É uma família de FPGAs de granulosidade fina baseada em *sea-of gates*. Seus dispositivos foram projetados para operar em conjunto com microprocessadores ou microcontroladores para implementar funções normalmente efetuadas por coprocessadores ASIC. Sua característica mais interessante é a capacidade de reconfiguração dinâmica, isto é, a reconfiguração total ou parcial do *chip* durante sua operação num sistema. Isto é possível devido a um controle de armazenamento SRAM altamente estável, que permite a rápida reconfiguração do dispositivo inúmeras vezes e pode ser mapeado no espaço de endereçamento de um processador hospedeiro, através da interface paralela com a CPU chamada FastMap™. Esses recursos possibilitam que a família XC6200 dê suporte a *hardware* virtual, permitindo inclusive trocas de contexto (*swap*). Operando como um coprocessador, esse dispositivo pode ser compartilhado por diversos processos em execução no processador hospedeiro.

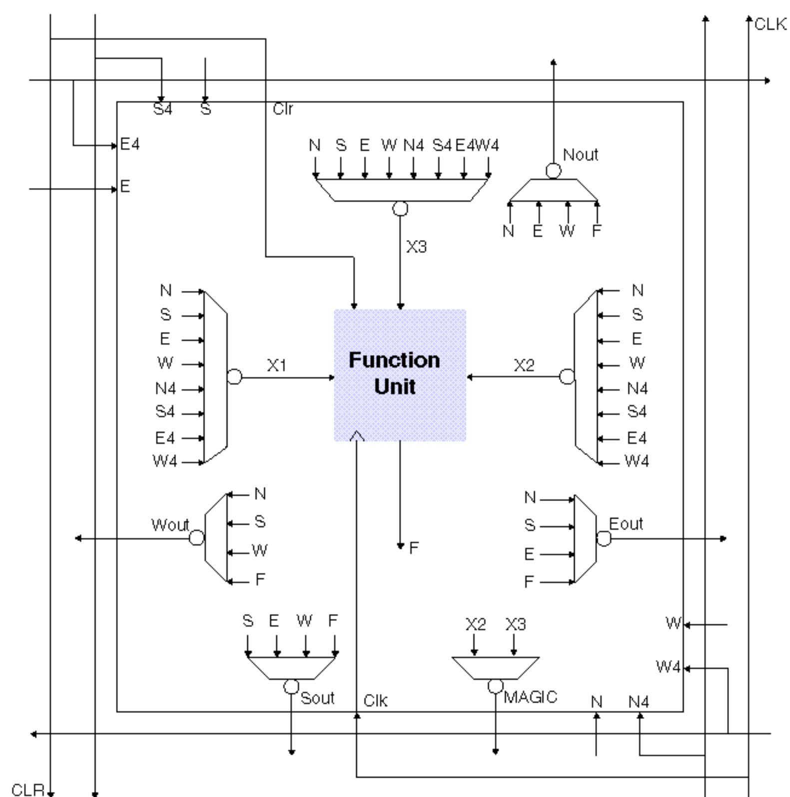


Figura 3.10 Uma Célula Básica da família XC6200.

A arquitetura da família XC6200 é hierárquica. Possui um grande arranjo de células simples, conhecido como *sea-of-gates*. Cada célula (figura 3.10) é programada individualmente podendo implementar, por exemplo, um registrador e uma função lógica simples, tal como um multiplexador. As células também podem ser configuradas para implementar uma função lógica puramente combinacional (sem registrador) e são utilizadas para o roteamento. Células vizinhas são agrupadas em blocos com 4x4 células (figura 3.11.a), formando a primeira hierarquia. Esses blocos comunicam-se com seus vizinhos, formando blocos maiores de 16x16 células. No dispositivo XC6216, como ilustra a figura 3.11.b, um arranjo 4x4 desses blocos de 16x16 células forma 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 trilha de comprimento 1, ou com células distantes utilizando segmentos de trilha longos que atravessam 4, 16 ou 64 células. Esses segmentos longos são denominados FastLANEs^{TM}.

a) Bloco de células 4x4

b) Dispositivo XC6216

Figura 3.11 Arquitetura Hierárquica da família XC6200.

3.2.4.2. Os FPGAs FLEX8000 e FLEX10000 da Altera

A série FLEX 8000 da Altera consiste de uma hierarquia de três níveis muito parecida à encontrada em CPLDs. Entretanto, o nível mais baixo da hierarquia consiste de um conjunto de LUTs, ao invés de blocos como os SPLDs, portanto a série FLEX 8000 pertence à categoria dos FPGAs. Deve ser notado, entretanto, que FLEX 8000 é uma combinação das tecnologias de FPGAs e CPLDs (BRO96a). A FLEX 8000 é baseada em SRAM e possui uma LUT de quatro entradas como seu bloco lógico básico. Sua capacidade lógica está na faixa de aproximadamente 2500 a 15000 portas lógicas (ALT96a). A arquitetura do FLEX 8000 é ilustrada na figura 3.12. O bloco lógico básico, chamado LE (Logic Element) contém uma LUT de quatro entradas, um flip-flop, e circuitos de *carry* de finalidade especial para circuitos aritméticos (similares ao Xilinx XC4000). O LE também inclui circuitos de cascata que permite a implementação eficiente de funções AND de várias entradas. Detalhes do LE estão ilustrados na figura 3.13.

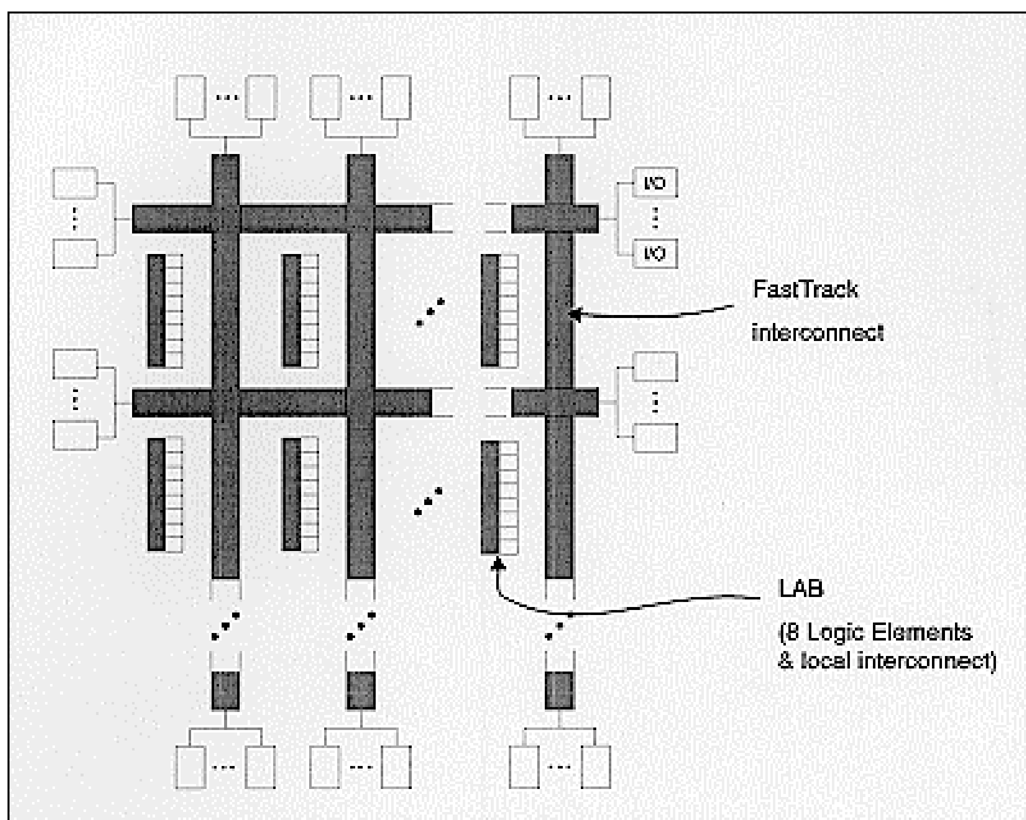


Figura 3.12 Arquitetura do FPGA da série FLEX 8000 da Altera.

Na FLEX 8000, os LEs são agrupados em conjuntos de oito, chamados LABs (Logic Arrays Blocks). Cada LAB contém interconexão local e cada trilha local pode conectar qualquer LE a outro LE no mesmo LAB. A interconexão local também se conecta a interconexão global da FLEX 8000, chamada *FastTrack*.

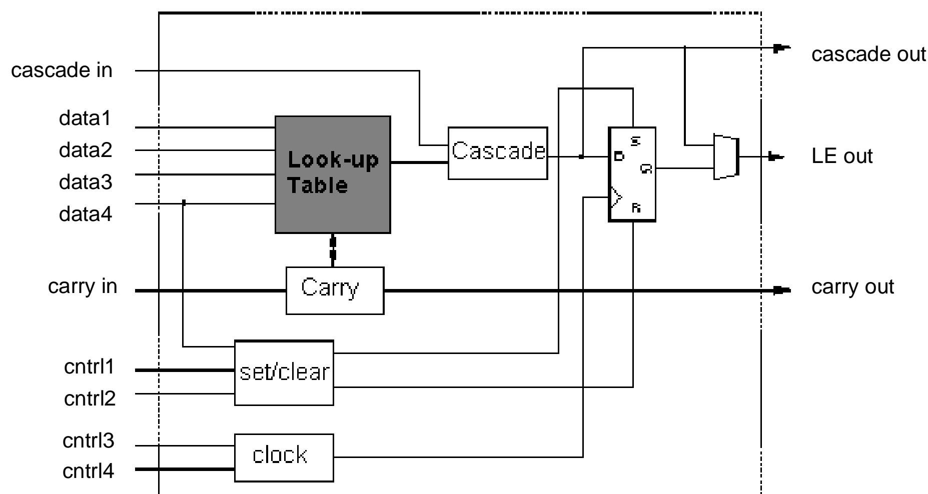


Figura 3.13 Bloco Lógico (LE) da série FLEX 8000 da Altera.

A *FastTrack* é similar às linhas longas 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. Isto facilita a configuração automática feita pelas ferramentas EDA. Todas as *FastTracks* horizontais são idênticas, portanto os atrasos de conexões na série FLEX 8000 são mais previsíveis do que em outros FPGAs que empregam muitos segmentos de comprimento menores, pois há menos comutadores programáveis em caminhos longos (Brown, 96).

A arquitetura da série FLEX 8000 foi estendida na família FLEX 10000. A FLEX 10000 oferece todas as características da FLEX 8000, com a adição de blocos de SRAM de tamanhos variados,

chamados EABs (Embedded Array Blocks). Essa idéia é ilustrada na figura 3.14, que mostra que cada linha no *chip* FLEX 10000 tem uma EAB em um final (Altera, 96).

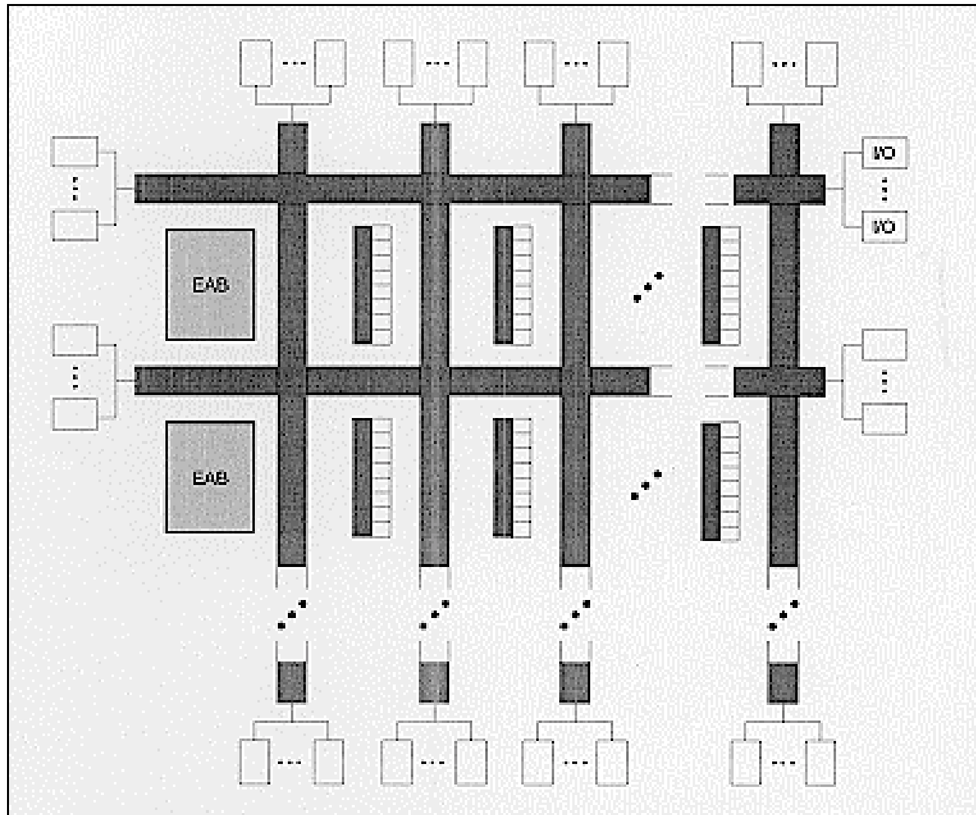


Figura 3.14 Arquitetura da série FLEX 10000 da Altera.

Cada EAB é configurável para servir como um bloco SRAM com tamanhos variáveis entre: 256x8, 512x4, 1Kx2, 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.

3.2.4.3. Os FPGAs da Actel

Ao contrário dos FPGAs descritos anteriormente, os dispositivos fabricados pela Actel são baseados na tecnologia antifuse. A Actel oferece três famílias principais: Act 1, Act 2 e Act 3.

Apesar das três gerações possuírem características semelhantes, será descrita somente a mais recente, por ser atualmente a mais utilizada.

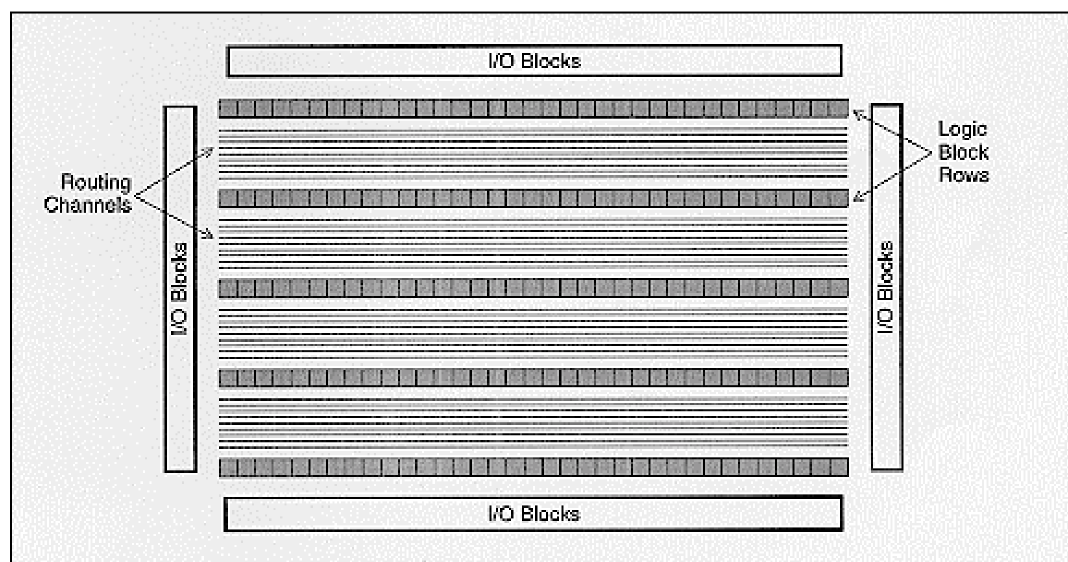


Figura 3.15 Arquitetura dos FPGAs da Actel.

Os dispositivos da Actel são baseados em uma estrutura similar aos tradicionais gate arrays; os blocos lógicos são arranjados em linhas e existem canais horizontais entre linhas adjacentes. Essa arquitetura é ilustrada na figura 3.15 (Actel, 95).

Os blocos lógicos da Actel são baseados em multiplexadores, sendo relativamente menores, comparados aos baseados em LUTs. O bloco lógico da família Act 3 é composto 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 possui um *flip-flop.*

Como já dito, os recursos de roteamento estão organizados em canais de roteamento horizontais. Os canais consistem de segmentos de trilhas de vários comprimentos com comutadores *antifuses* para conectar os blocos lógicos aos segmentos ou um segmento a outro. Também, como mostra a figura 3.16, os *chips* Actel têm trilhas verticais que atravessam os blocos lógicos, para caminhos

de sinais que se estendem por múltiplas linhas. A arquitetura de roteamento da Actel é assimétrica porque há mais trilhas na horizontal do que na vertical.

Não há um bloco de comutação claramente separado nessa arquitetura. A comutação é distribuído por todos os canais horizontais. Dependendo da direção da conexão, diferentes graus de conectividade são disponíveis. Todas as trilhas verticais podem ser conectadas a todas as trilhas horizontais incidentes. Essa forma fornece mais conectividade que o bloco de comutação da Xilinx. Essa flexibilidade permite que o roteamento dos canais horizontais sejam feitos 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 na arquitetura da Xilinx. A flexibilidade simplifica o problema de roteamento, mas em compensação aumenta o número de comutadores necessários.

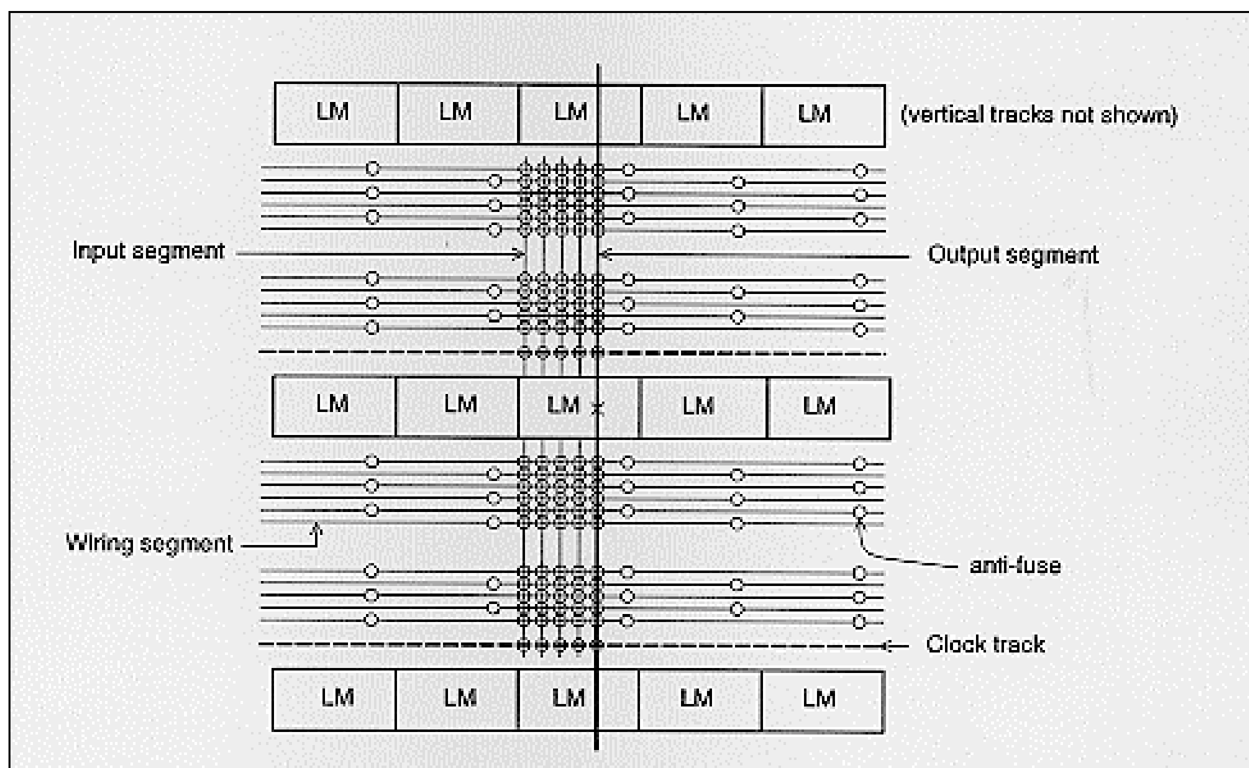


Figura 3.16 Arquitetura de Roteamento da Actel.

Em termos de desempenho, pode parecer que os *chips* Actel não são completamente previsíveis (cálculo dos atrasos), porque a quantidade de comutadores *antifuse* atravessados por um sinal depende de como os segmentos de trilha foram alocados, durante a implementação do circuito,

pelas ferramentas EDA. Entretanto, a Actel possibilita uma rica seleção de segmentos de trilhas de diferentes comprimentos em cada canal e tem desenvolvido algoritmos que garantem limites restritos ao número de comutadores *antifuse* atravessados por uma conexão entre dois pontos de um circuito, o que melhora significantemente o desempenho.

3.3. Desenvolvimento e Implementação de Projetos Usando FPGAs

O processo de projeto com FPGAs é bastante similar aos de outros dispositivos PLD, envolvendo várias etapas que são geralmente automatizadas. Porém, devido a sua maior complexidade, exige o uso de ferramentas EDA mais sofisticadas, como os programas de posicionamento e roteamento (Brown & Rose, 96). O processo envolve as seguintes etapas :

- Especificação e entrada do projeto (desenho de esquemático, HDLs)
- Síntese Lógica e mapeamento na tecnologia
- Posicionamento e roteamento
- Verificação e Testes
- Configuração/programação do FPGA

Apesar de alguns sistemas possibilitarem a intervenção manual em todas as etapas, geralmente o projetista é responsável apenas pela entrada inicial do projeto e verificações durante o processo. As fases mais críticas como otimização lógica, mapeamento, posicionamento e roteamento são realizadas por *software*s específicos, possibilitando que o projetista se concentre nas especificações do projeto e trabalhe em alto nível. Um sistema típico de EDA para FPGAs consiste de vários programas interconectados como ilustra a figura 3.17.

A entrada do projeto pode ser feita criando-se um diagrama esquemático com uma ferramenta gráfica, usando um sistema textual para descrever um projeto em uma linguagem de descrição de *hardware*, ou com uma mistura de métodos de entrada de projeto.

Como a lógica inicial não está geralmente numa forma otimizada, algoritmos são empregados para otimizar os circuitos, manipulando-os para minimizar área, atraso, ou a combinação de área e atraso. Esse passo geralmente realiza o equivalente a uma minimização algébrica de equações

booleanas e é apropriado quando se implementa um circuito em qualquer tecnologia, não somente em FPGAs (Brown, 92).

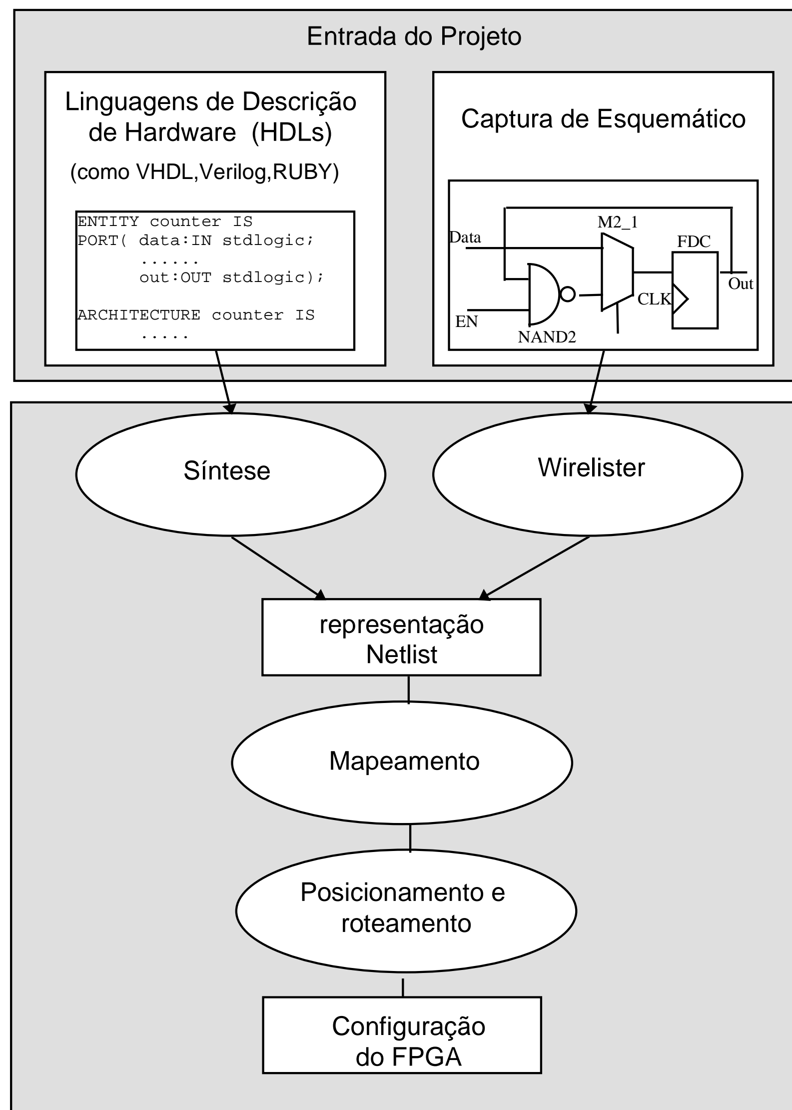


Figura 3.17 Sistema EDA para projetos com FPGAs.

Para transformar as equações booleanas em circuito de células lógicas de FPGA, a rede otimizada alimenta um programa de mapeamento na tecnologia. Esse passo mapeia as equações em células lógicas, e também é uma oportunidade para otimizar, ou minimizar o número total de células lógicas requeridas (otimização de área) ou o número de células lógicas em caminhos críticos (otimização de atraso). O circuito de células lógicas é então passado para um programa

de posicionamento, que seleciona uma localização específica no FPGA para cada célula lógica. Algoritmos típicos de posicionamento geralmente tentam minimizar o comprimento total das interconexões necessárias.

O passo final no sistema de EDA é realizado pelo *software* de roteamento, que aloca os recursos de roteamento do FPGA para interconectar as células lógicas posicionadas. As ferramentas de roteamento devem assegurar que 100% das conexões requeridas são realizadas, e deve procurar maximizar a velocidade das conexões críticas, porém, essa meta nem sempre é alcançada. Finalmente a saída do sistema EDA alimenta a unidade de programação que é utilizada para configurar o FPGA. A seguir são descritas, com maiores detalhes, as diferentes etapas de projeto.

3.3.1. Especificação e Entrada do Projeto

A especificação do projeto é apresentada ou em termos abstratos ou em métodos formais, seguida pela análise da viabilidade da implementação através de simulação de alto nível. Esta é a fase de pensar e escrever os processos em um nível abstrato, em alto nível. Nessa fase é importante que a linguagem utilizada seja o mais próximo possível da linguagem humana. Então a implementação do projeto é expressa por uma descrição que é admissível pelas ferramentas do fabricante da FPGA, por exemplo, em termos de portas lógicas e macros que estão na biblioteca de macros do fabricante ou em linguagens de descrição de *hardware*.

3.3.1.1. Editores de Esquemático

As ferramentas de captura de esquemático ou editores de esquemático permitem que o desenvolvedor especifique o circuito como um diagrama 2-D, conectando componentes lógicos com os recursos de roteamento. Isto pode ser feito usando uma das ferramentas de projeto (como ViewDraw, Design Architect, Orcad, Tango e etc.).

Esses componentes lógicos estão geralmente contidos em bibliotecas de macros fornecidas pelos fabricantes de FPGAs ou podem ser definidos pelo usuário. Tipicamente, as bibliotecas contêm portas lógicas básicas, pinos de I/O, *buffers*, osciladores ou geradores de pulso, *latches*, *flip-flops*, decodificadores, multiplexadores, registradores, contadores, comparadores, registradores de deslocamento, funções aritméticas, memórias e outras funções especiais.

Há também símbolos especiais para controle do mapeamento, posicionamento e roteamento durante a fase de implementação do projeto. Com esses símbolos pode-se definir (estabelecendo restrições) o mapeamento de determinada parte da lógica e o posicionamento de blocos lógicos. As macros podem ser *soft*, que significa que o posicionamento e roteamento ficarão a cargo das ferramentas; ou *hard*, que já estão pré-mapeadas, pré-posicionadas e pré-roteadas. Idealmente, as macros *hard* fornecem garantia de posicionamento e roteamento de um projeto mapeado, e consequentemente garantia de desempenho, entretanto o uso abusivo desse tipo de macro pode gerar conflitos e dificultar a realização de um projeto.

O modo tradicional de especificar um projeto usando um editor de esquemático é melhorado pela introdução de ferramentas orientadas a módulos como LogiBLOX da Xilinx. O LogiBLOX é uma biblioteca de funções lógicas implementadas em VHDL. Ela permite que projetos sejam descritos em nível de blocos, ou seja, um nível de abstração acima do nível de portas lógicas. Uma vez que as rotinas estão descritas em VHDL, muitos parâmetros são criados para uma maior flexibilidade dos blocos. Apenas os parâmetros utilizados serão sintetizados, gerando um circuito final menor.

3.3.1.2. Entrada do Projeto em Baixo Nível

Alguns sistemas permitem que o projetista tenha o absoluto controle da utilização dos recursos disponíveis pelo FPGA. Utilizando editores de projetos, como o XACT da Xilinx, o projetista pode especificar um projeto em baixo nível. Com o XACT, os conteúdos das configurações dos CLBs e IOBs devem ser inseridos manualmente um por um, e as conexões entre os blocos lógicos são determinadas pelo projetista com o auxílio limitado de um roteador. O projeto manual em baixo nível tem o mérito que a utilização do *chip* e o desempenho do projeto são otimizados, porém esse procedimento é muito demorado.

3.3.1.3. Entrada através de Linguagens de Descrição de Hardware - HDLs

Como mencionado no capítulo anterior, as HDLs estão se tornando a forma padrão para entrada de projetos, principalmente para grandes circuitos, devido as vantagens que oferece como a portabilidade e independência da tecnologia.

Assim como as ferramentas, o estilo de descrição das HDLs devem ser adaptados aos requerimentos das arquiteturas de FPGAs para gerar projetos eficientes. Ao contrário de ASIC, os FPGAs oferecem um conjunto fixo de recursos que podem ser utilizados em um projeto. Isto requer que o código fonte em HDL de um projeto seja adaptado para explorar os recursos disponíveis. Enquanto, idealmente, o modelo de VHDL sintetizável pode ser o mesmo para todas as tecnologias alvo, a eficiência do projeto resultante é muito dependente da descrição e tecnologia utilizada. Devido a arquitetura dos FPGAs, problemas de otimização encontrados em ASIC nem sempre podem ser convertidos para FPGAs. Nesse aspecto, principalmente as arquiteturas baseadas em LUTs são diferentes devido à granulosidade grossa de seus blocos lógicos, enquanto arquiteturas de granulosidade fina comportam-se como ASICs. Metodologias mais adequadas para projetos com FPGAs estão sendo propostas (Gschwind & Salapura, 95; Middelhoek & Rajan, 96) visando a geração de modelos de VHDL eficientes.

3.3.2. Síntese Lógica e Mapeamento

Os processos mais dependentes da tecnologia começam aqui. Há várias abordagens para a otimização lógica. A abordagem mais comumente utilizada é quebrar o processo de síntese em duas fases: uma fase independente da tecnologia, seguida pela fase de mapeamento na tecnologia. A fase independente da tecnologia tenta gerar uma representação abstrata otimizada do circuito lógico. A fase de mapeamento na tecnologia seleciona um conjunto de portas lógicas de uma dada biblioteca para implementar as representações abstratas, enquanto otimiza a área, o atraso ou a combinação de ambos, levando em consideração as restrições arquiteturais da tecnologia alvo, nesse caso os FPGAs.

A complexidade de arquiteturas de FPGA torna o mapeamento manual de projetos muito difícil e demorado. As ferramentas de síntese que mapeiam automaticamente um projeto composto de portas simples ou descrito com uma HDL, em portas de uma dada biblioteca são, portanto, muito importantes no processo de desenvolvimento de projetos.

A abordagem mais direta para a síntese de FPGAs é adaptar as ferramentas de síntese desenvolvidas para bibliotecas de MPGAs. Um projeto é inicialmente mapeado em portas simples (como portas NAND de duas entradas), e então grupos de portas simples são substituídos por blocos lógicos do FPGA. Essa abordagem funciona bem para FPGAs com blocos lógicos de granulosidade fina, já que um bloco de granulosidade fina pode implementar

somente uma ou duas portas simples. Entretanto, para os FPGAs mais usados de granulosidade grossa como os da Actel, Altera e Xilinx, essa abordagem não apresenta resultados aceitáveis.

Uma abordagem mais promissora e desafiadora é mapear o projeto diretamente em blocos lógicos. As ferramentas de síntese desenvolvidas para FPGAs empregam tanto a abordagem de mapeamento na biblioteca como a abordagem de mapeamento direto. A maioria dos métodos desenvolvidos otimiza a área de um projeto e somente alguns poucos otimizam o desempenho, explicitamente (Vincentelli et al., 93),(Morgai, 95).

3.3.2.1. Mapeamento na Tecnologia

As funções lógicas geradas por um minimizador lógico independente da tecnologia podem não estar na forma implementável. O mapeamento na tecnologia é o processo de atribuir circuitos dependentes da tecnologia (blocos lógicos) aos circuitos independentes da tecnologia, como ilustra a figura 3.18, onde um esquemático é mapeado em três blocos lógicos de um FPGA da Xilinx.

Minimizadores lógicos tipicamente produzem expressões booleanas de granulosidade muita fina. Um mapeador da tecnologia geralmente agrupa circuitos de granulosidade fina em células de granulosidade maior. Um mapeador para a tecnologia de FPGA para circuitos combinacionais realiza três funções primárias (Chan & Mourad, 94):

i. Decompõe expressões inviáveis em viáveis.

ii. Agrupa pequenas expressões em blocos lógicos para promover o compartilhamento de recursos.

iii. Aloca blocos lógicos para expressões que não podem ser divididas.

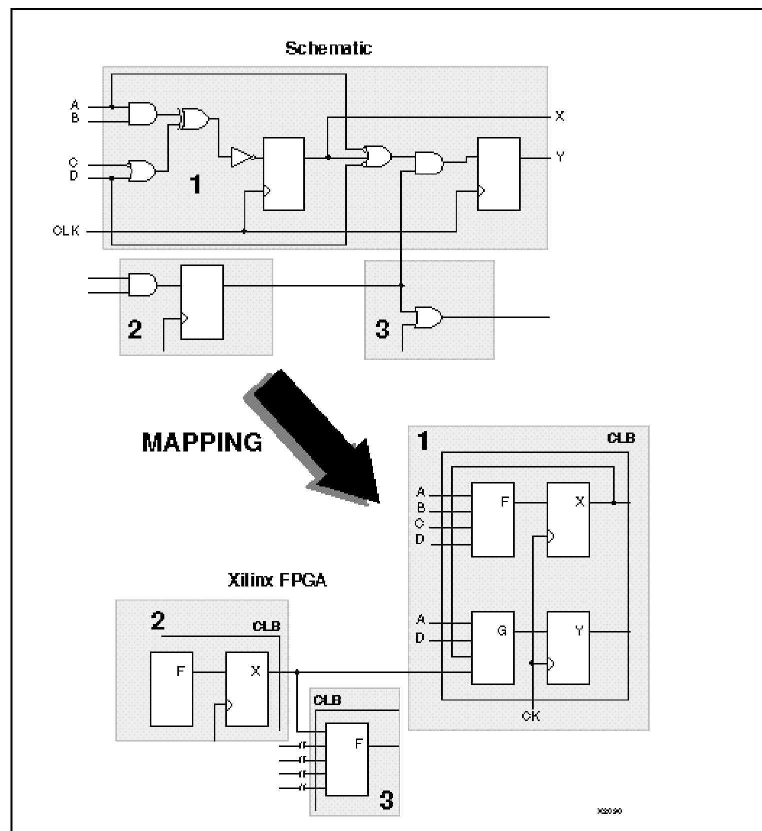


Figura 3.18 Exemplo de Mapeamento num FPGA da Xilinx.

3.3.3. Posicionamento e Roteamento

Após a minimização lógica e o mapeamento da tecnologia, o projeto consiste de uma *netlist* de componentes lógicos a serem designados aos componentes físicos de uma arquitetura de FPGA. Neste ponto, somente os conteúdos das células básicas da tecnologia estão definidos; onde as células devem ser posicionadas e como conectá-las permanece indeterminado.

O posicionamento e roteamento são dois processos mutuamente dependentes. O posicionamento é a atribuição de componentes físicos particulares do *chip* aos componentes lógicos do projeto. A atribuição de trilhas e elementos programáveis, consumindo os recursos disponíveis de interconexão para a comunicação entre os componentes, é conhecida como roteamento. O posicionamento e roteamento de um projeto são tradicionalmente feitos em duas fases independentes, apesar de haver exceções em que são feitos simultaneamente. Essa etapa do processo é crucial, pois afeta diretamente a viabilidade e desempenho do projeto.

3.3.3.1. Posicionamento

Em dispositivos FPGAs, o posicionamento de componentes atribui uma unidade física do dispositivo (blocos lógicos ou de I/O) a uma unidade lógica. Algumas ferramentas de particionamento e posicionamento intencionalmente aumentam a utilização do *chip*, usando blocos lógicos para rotear sinais e, dessa forma, ajudando o roteador.

Algoritmos de posicionamento têm de satisfazer tanto as restrições do dispositivo e as impostas pelo usuário como a minimização de uma função-custo. Restrições típicas de posicionamento são: alocar certos blocos de I/O como requerido pelo usuário; alinhar *buffers tristate*, que pertencem ao mesmo barramento, em uma trilha longa. Por outro lado, as funções-custo mais comuns para o posicionamento são: minimização do comprimento total das conexões; minimização do número total de elementos programáveis; redução da congestão para assegurar a capacidade de roteamento e a minimização das conexões entre diferentes regiões do *chip*.

3.3.3.2. Roteamento

Softwares que realizam roteamento automático já existem há vários anos, sendo os primeiros algoritmos destinados ao roteamento de placas de circuito impresso. Com o passar dos anos muitos algoritmos de roteamento foram publicados, portanto o problema é bem definido e entendido.

Devido à complexidade combinatorial envolvida, a solução de um grande problema de roteamento usualmente requer uma estratégia "dividir e conquistar". Seguindo essa filosofia, o roteamento pode ser resolvido por um processo de três etapas (Brown, 92):

i. Partição dos recursos de roteamento em áreas de roteamento que são apropriadas tanto para o dispositivo a ser roteado como para os algoritmos a serem empregados.

ii. Uso de um roteador global para designar cada rede a um subconjunto de áreas de roteamento. O roteador global não escolhe segmentos de trilhas específicos e comutadores para cada conexão, mas cria um novo conjunto restrito de problemas de roteamento.

iii. Uso de um roteador detalhado para selecionar segmentos de trilha específicos e comutadores para cada conexão, com os conjuntos restritos do roteador global.

A vantagem dessa abordagem é que cada uma das ferramentas de roteamento pode resolver mais eficientemente uma parte menor do problema de roteamento. Mais especificamente, como o roteador global não precisa se preocupar com a alocação de segmentos de trilhas e comutadores, ele pode se concentrar em questões mais globais, como o balanceamento do uso dos canais de roteamento. Similarmente, com a redução do número de alternativas disponíveis para o roteador detalhado, devido às restrições introduzidas pelo roteador global, o roteador detalhado pode empenhar-se no problema de selecionar as conexões. Esse escopo limitado possibilita que o roteador detalhado se concentre na solução de disputas pelos recursos de roteamento que podem existir entre diferentes redes.

3.3.3.3. Capacidade de Roteamento e Recursos de Roteamento

Dada uma arquitetura de FPGA, uma ferramenta de posicionamento e roteamento e um projeto, defini-se capacidade de roteamento como a medida da probabilidade em que a ferramenta de posicionamento e roteamento poderá completar com sucesso o roteamento do projeto. É uma probabilidade porque quase todas as ferramentas de posicionamento e roteamento têm algum grau de aleatoriedade. Em outras palavras, a capacidade de roteamento de um projeto é afetada tanto pelos recursos de roteamento disponíveis quanto pelos algoritmos de posicionamento e roteamento utilizados. Os FPGAs têm recursos heterogêneos: linhas diretas, trilhas longas, segmentos de trilha de tamanhos variados. Um FPGA tem recursos lógicos e de roteamento fixos. Não é difícil um projeto que está utilizando bem poucos recursos lógicos exceder os recursos de roteamento. Tanto a teoria como a prática confirmaram que há três fatores dominantes que afetam a capacidade de roteamento de um projeto com FPGA: a flexibilidade do bloco de conexão, flexibilidade do bloco de comutação e a média dos comprimentos das trilhas (Rose et al., 93).

3.3.3.4. Atrasos de Roteamento

Uma vez que as células lógicas estão posicionadas e roteadas, o projeto pode não funcionar apropriadamente devido a problemas de temporização. Como já foi dito, o posicionamento e roteamento são mutuamente dependentes. Algumas vezes pode ser necessário criar um novo posicionamento para atingir os objetivos do projeto: completar o roteamento e satisfazer as restrições de temporização.

O objetivo de um roteamento dirigido a atraso é produzir rotas que reduzam atrasos entre os caminhos que limitam o desempenho do circuito. Esses caminhos são chamados caminhos críticos. Diferentes partes do circuito contribuem para os atrasos nos caminhos. O atraso em circuitos FPGAs podem ser atribuídos a: atraso dos blocos de I/O; atraso dos blocos lógicos; atraso das trilhas e atraso devido aos elementos programáveis. O atraso devido ao roteamento é bastante significante em FPGAs: corresponde entre 40% a 60%, portanto o cálculo do atraso tem um importante papel no projeto com esses dispositivos.

3.3.4. Verificação e Testes

3.3.4.1. Verificação do Projeto

A simulação é o tipo mais comum de verificação utilizado. Em projetos com FPGAs, a simulação é realizada geralmente na fase inicial, para verificação funcional, e antes da programação, para verificação de restrições de temporização.

As ferramentas de verificação de projeto são genéricas considerando-se que o processo de verificação e a interface do usuário não estão geralmente vinculados a fabricantes específicos, embora os desempenhos dos componentes estejam. O processo de verificação pode iniciar tão logo o projeto esteja especificado. Se um modelo de unidade de atraso for estipulado, pode-se verificar a funcionalidade do projeto nesse estágio. Nos FPGAs, a simulação funcional pode ser realizada em nível comportamental ou em nível de portas lógicas.

No caso de simulação temporal, várias abordagens podem ser tomadas. A temporização pode ser verificada com os atrasos nominais da tecnologia empregada, ou pode considerar o pior caso. Há ferramentas que extraem informações de atrasos após a fase de posicionamento e roteamento do projeto, para serem utilizadas na simulação.

Vários simuladores para a tecnologia FPGA estão disponíveis como o *Viewsim* e *Synopsys*. Esses simuladores não são desenvolvidos pelos fabricantes de FPGAs, mas por desenvolvedores de ferramentas EDA.

A verificação pode ser considerada mais importante que os testes do FPGA antes da programação, principalmente para dispositivos OTP, como os da Actel. A validação de um

projeto também pode ser feita através de um método formal. Essa técnica é baseada na prova da correção do projeto usando lógica formal (teoria dos conjuntos), porém é muito complexa e inviável para grandes circuitos.

3.3.4.2. Testes de FPGAs

Os FPGAs geralmente passam, entre a fabricação e a programação, por três estágios de testes: pelo fabricante, antes da programação e após a programação. Os dispositivos ainda não programados são geralmente inteiramente testados pelos fabricantes. Porém, como são dispositivos sensíveis a descargas eletrostáticas, podem ser danificados, se mal manuseados. Logo, é recomendável que o usuário teste o dispositivo antes da programação.

Já que os FPGAs não incluem nenhuma lógica em particular, a abordagem utilizada para seus testes não é realizada do modo tradicional como em outros circuitos integrados. O principal problema em testar esses dispositivos é verificar se seus elementos programáveis estão funcionando. A estratégia de testes varia com o tipo de FPGA, dependendo de sua arquitetura e tecnologia de programação. Seus testes envolvem exame de seus blocos lógicos, elementos programáveis e interconexões. Os FPGAs tem circuitos especiais para facilitar os testes. Por exemplo, os dispositivos da Actel têm o mecanismo de *probe* e os da Xilinx têm procedimentos de *read back.*

Nos *chips* baseados em SRAM da Xilinx, a estratégia de testes inclui: escrita e leitura de toda a configuração de células; testes pseudo-exaustivos de todos os CLBs; teste de continuidade e de curto circuito nos segmentos de metal interconectados. As diferentes configurações de teste testam o dispositivo sem removê-lo do circuito. Testar o dispositivo dessa maneira é equivalente a programá-lo com todas as possíveis configurações. Os usuários podem aplicar alguns desses métodos de teste antes de programar o dispositivo com suas aplicações, por exemplo, conectando todos os flip-flops em uma ou mais configurações de registradores e verificando se eles podem deslocar um padrão conhecido. Um teste desse tipo muito utilizado é o *Boundary Scan*, segundo o padrão IEEE 1149.1-1990 (Xilinx, 98).

Uma abordagem similar é utilizada no caso de outros dispositivos reprogramáveis que utilizam as tecnologias EPROM e EEPROM. Testes de todas as configurações são problemáticos para

EPROM já que o apagamento do dispositivo não é tão simples como o caso de EEPROM e FPGAs baseados em SRAM.

Testes de dispositivos OTP são provavelmente mais difíceis. Os dispositivos da Actel incluem um *hardware* extra dedicado para testes que não requer programação dos *antifuses* exceto aqueles dedicados para os testes. Como no caso de dispositivos baseados em EPROM, não há garantia que todas as potenciais interconexões estão de fato conectadas. O único modo de verificar com certeza é testando o dispositivo depois de programá-lo.

Após a programação, o dispositivo deve ser testado novamente. Para isto, podem ser utilizados os padrões gerados na simulação ou outros métodos como a geração automática de padrões de teste (ATPG), que aplica testes aleatórios gerados por um equipamento ATE (*Automatic Test Equipment*).

3.3.5. Configuração do FPGA

A implementação do projeto é completada neste ponto, mas ainda resta um passo final que é a programação do FPGA. Para um FPGA programável uma única vez (OTP, *One Time Programmable*), como os da ACTEL, é necessário um dispositivo especial para "queimar" os *antifuses* dentro do FPGA para configurá-lo, de modo a implementar o projeto. Para FPGAs reprogramáveis como os da Xilinx, o programa *makebits* é usado para gerar um arquivo de configuração para o FPGA. Um dispositivo FPGA pode ser programado de vários modos. O modo serial é o mais recomendado para a fase inicial de prototipação. Isto é porque o arquivo de configuração pode ser carregado através da porta serial do computador diretamente para o dispositivo por um cabo especial fornecido pelo fabricante. Se o projeto deve ser realizado na placa, pode-se usar PROMs, EPROMs ou EEPROMs como um modo semi-permanente de fornecer o arquivo de configuração para o FPGA.

3.4. Aplicações de FPGAs

Os FPGAs surgiram nos meados dos anos 80 e ainda são dispositivos relativamente novos, mas que já se disseminaram e conquistaram uma significativa fatia do mercado de semicondutores. Ao lado dos CPLDs, são dispositivos que apresentam características e vantagens competitivas que os tornam adequados a uma grande variedade de aplicações.

Dentre suas principais vantagens destaca-se: sua ótima flexibilidade em função da alta capacidade lógica; a facilidade de programação pelo usuário final, que reduz o tempo de fabricação para minutos; o ciclo curto de projeto que possibilita um rápido *time-to-market*, o baixo custo do dispositivo e dos custos fixos do projeto. (Tuck, 92; Donlin, 95).

Atualmente os FPGAs são utilizados nas mais variadas áreas (Brown & Rose, 96), (Linde et al., 93), (Petersen & Hutchings, 95), (Vincentelli, 93), (Fawcet, 94). Uma lista de aplicações inclui: integração de múltiplos SPLDs, lógica *fuzzy*, equipamentos de comunicação (codificadores, filtros, encriptação/decriptação, modems, compressão de dados e etc.), equipamentos industriais (controladores, aparelhos de testes, equipamentos médicos, robótica, emuladores de microprocessadores e etc.), computadores e periféricos (interfaces de memórias, controladores DMA (*Direct Memory Access*), co-processadores, impressoras, scanners, controladoras de vídeo e etc.), telecomunicações (centrais telefônicas, estações de telefonia celular, redes ATM (*Asynchronous Transfer Mode*), interfaces de fibra ótica e etc.), equipamentos de alta confiabilidade (militares, espaciais) e muitas outras.

Outras interessantes aplicações de FPGAs são a prototipação de projetos a serem implementados em outras tecnologias como ASIC e *gate arrays*, e também a emulação de grandes sistemas de *hardware* (Brown & Rose, 96). A primeira dessas aplicações pode ser possível usando-se um único FPGA de alta capacidade (que corresponde a um pequeno *gate array*), e a última poderia vincular muitos FPGAs conectados por algum tipo de interconexão, como o sistema da Aptix Corp., que utiliza a tecnologia FPIC (*Field-Programmable Interconnect Component*) para esse fim.

3.4.1. *Hardware* Reconfigurável: Uma Nova Abordagem Computacional

Outra área promissora para aplicação de FPGAs, que ainda está começando a se desenvolver, é a implementação de máquinas computacionais dedicadas e reprogramáveis dinamicamente (Fawcet, 94; Lysaght & Dunlop, 94; Brown, 96; Lysaght, 95). Isto envolve o uso das partes programáveis para "executar" algoritmos, ao invés de compilá-los para a execução numa CPU.

Os FPGAs combinam a flexibilidade de dispositivos programáveis (como PLDs e microprocessadores de finalidade geral) com o desempenho do *hardware* de finalidade

específica (como ASIC), abrindo um espaço interessante entre os extremos da computação de finalidade geral e *hardware* dedicado (Donachy, 96).

Apesar de sua larga utilização, processadores de finalidade geral não são ideais para a maioria das aplicações que executam. Para quase todas as aplicações, mudanças ou adições que melhorariam de forma significativa o desempenho podem ser sugeridas nas arquiteturas dos microprocessadores. Entretanto, as adições diferem de aplicação para aplicação.

O *hardware* reconfigurável apresenta-se como uma alternativa tecnológica que pode adaptar-se a aplicação com a facilidade de um processador de finalidade geral, enquanto mantém as vantagens de desempenho do *hardware* dedicado. Com essa inerente velocidade e adaptabilidade, a computação reconfigurável tem um grande potencial a ser explorado especialmente em aplicações que necessitam de alto desempenho como arquiteturas paralelas, processamento de imagens e aplicações de tempo real. Sua eficiência computacional já foi demonstrada em projetos como o Splash (Gokhale et al., 91) e muitos outros estão sendo desenvolvidos (Guccione, 95,96; Linde et al.,93). Entretanto, aspectos como arquiteturas de FPGAs mais adequadas, o modelo de programação de FPGAs, métodos de síntese de VHDL mais eficientes e a adaptação de algoritmos ao *hardware* devem ser discutidos e melhorados para a difusão dos sistemas de computação reconfigurável.

3.5. Considerações Finais

Desde sua introdução em 1985, os FPGAs têm rapidamente evoluído e transformado o modo como o *hardware* digital é projetado. Seu enorme potencial para o desenvolvimento de aplicações faz do FPGA uma tecnologia empolgante, que deve se expandir muito nos próximos anos (Fawcet, 94), (Vincentelli,93), (Donlin, 95).

Atualmente dispomos de diversas opções de dispositivos de vários fabricantes, que empregam diferentes arquiteturas e tecnologias de programação e oferecem capacidade lógica já superior a 500 mil portas lógicas. Novos dispositivos estão sendo continuamente lançados, cada vez maiores, mais rápidos, fáceis de utilizar e mais baratos. Entretanto, os FPGAs são dispositivos relativamente novos, portanto muita pesquisa ainda é necessária para melhorar essa tecnologia, principalmente visando otimizar a área do *chip* e o desempenho dos circuitos implementados (Brown, 96).

Na área de arquiteturas alguns tópicos que têm sido estudados são: complexidade dos blocos lógicos, flexibilidade das estruturas de interconexão, segmentação das interconexões, blocos lógicos *hard-wired*, estruturas hierárquicas, estruturas de memórias em FPGAs , dentre outros. As pesquisas nessa área além de propor novas arquiteturas também devem desenvolver ferramentas EDA necessárias para experimentar e avaliar a arquitetura proposta.

Nos próximos anos a capacidade lógica dos FPGAs deve atingir 1 milhão de portas, aumentando sua gama de aplicações e superando algumas desvantagens em relação ao MPGA. Também é possível que o sucesso do FPGA no mundo digital seja reproduzido no domínio analógico, com o *Fiel-Programmable Analog Array* (FPAA).

Observou-se que o processo de desenvolvimento de projetos com FPGAs é similar aos de outras tecnologias como ASIC e *gate arrays,* entretanto as ferramentas e algoritmos foram adaptados em função das características particulares das arquiteturas de. O processo é significantemente mais curto, tanto na fase inicial, facilitada pelo uso de bibliotecas de componentes e/ou HDLs, como na fase final, devido à programação pelo usuário (Tuck, 92a).

Cabe ressaltar que as ferramentas automáticas são essenciais durante o processo, porém é importante que o projetista conheça o dispositivo e tenha o total controle sobre as ferramentas, a fim de alcançar um melhor desempenho do projeto (Tuck, 92b).

3.5.1. Escolha das Ferramentas

O gerenciamento cuidadoso do processo de projeto é muito importante para satisfazer os requisitos, atualizar as metodologias e escolher as ferramentas. O gerenciamento assegura que as ferramentas adequadas são selecionadas e executadas na sequência correta (Baldwin & Chung, 95). Usuários experientes afirmam que possuir as ferramentas corretas é tão importante para o projeto como selecionar o dispositivo adequado. As ferramentas corretas oferecem facilidade de aprendizado, ciclo de desenvolvimento curto, máxima utilização do *chip*, desempenho e suporte ao dispositivo mais adequado ao projeto. Algumas diretrizes são relevantes para a escolha de ferramentas apropriadas. Um bom conjunto de ferramentas deve (Tuck, 92b):

- Dar suporte aos ambientes de projeto EDA existentes (editores de esquemático e simuladores existentes).

- Ser completo. Um conjunto mínimo deve incluir: entrada para todas as metodologias populares de projeto (Palasm/Abel, captura de esquemático, VHDL), assim como tradutores para padrões da indústria como EDIF (*Electronic Design Interchange Format)* e LPM (*Library of Parameterized Modules*); mapeadores; otimização específica para o dispositivo; posicionamento e roteamento automático; um editor gráfico que possa ser usado para pré-posicionamento e roteamento de sinais críticos; simuladores e análise de tempo.

- Possibilitar um curto *time-to-market*.

- Ter capacidade de otimizar o desempenho e utilização de dispositivos existentes. É vital que as ferramentas forneçam rotinas de otimização eficientes para implementar um projeto numa arquitetura específica.

- Possibilitar um controle completo sobre as ferramentas e seus resultados. As soluções automáticas somente irão fornecer resultados satisfatórios se o usuário puder verdadeiramente controlar as ferramentas. O usuário deve ser capaz de: especificar preferências, que incluam restrições físicas como pinagem e *floorplanning*, e requerimentos de tempo, como frequências de *clock*, atrasos e defasagens.

- Ter capacidade controlar a temporização (*timing-driven*). Especificando-se requisitos de tempo exatos (freqüências e atrasos de caminhos), e não somente prioridades de roteamento, os problemas de temporização podem ser eliminados antes mesmo que ocorram.

- Tornar o projeto de FPGAs simples. Não deve exigir que o usuário torne-se um especialista na arquitetura do *chip* para que explore todas as vantagens do dispositivo.

- A independência do dispositivo está se tornando muito importante à medida que os novos fabricantes entram no mercado e as ferramentas devam suportá-los. O usuário não deve comprar e aprender vários conjuntos de ferramentas para usufruir dos novos dispositivos. As ferramentas devem ter a capacidade de reimplementar projetos existentes.

- Dar suporte a uma metodologia de projeto transparente à tecnologia. Isto envolve a capacidade de realizar a captura de esquemático e verificação funcional, independente da tecnologia final de implementação.

- Finalmente, o conjunto de ferramentas deve executar em plataformas populares, como PCs e workstations.

4. Arquiteturas Sistólicas

4.1. Considerações Iniciais

Em 1978 foi introduzido o termo e o conceito "*Systolic Array*" para designar uma classe especial de arquiteturas de computadores. A partir disso, muita pesquisa tem sido realizada sobre algoritmos e arquiteturas suportadas por esse conceito. Cabe salientar que muitas técnicas de arquiteturas sistólicas já eram utilizadas anteriormente pelos projetistas de arquiteturas, sendo que o mérito de Kung e Leirserson foi de formalizar o conhecimento em torno dessa classe de computadores. O termo *Systolic Array* designa uma classe de arquiteturas paralelas e estabelece uma analogia com o sistema vascular humano. O coração envia e recebe uma grande quantidade de sangue como resultado do bombeamento freqüente e ritmado do fluxo sanguíneo através das artérias e de veias. Analogamente à arquitetura, o coração corresponde à fonte e a destinação dos dados (por exemplo, uma memória global), a rede de veias e artérias equivale à rede de processadores e "*links*" que compõem a arquitetura. Outra analogia é que a arquitetura intercala ciclos de admissão e expulsão de dados, da mesma forma que o coração, quando bombeia sangue pelas artérias e veias (Kung, 82; Fortes & Wah, 87; Duncan, 90; Almasi & Gottlieb, 94).

O poder por trás dos arranjos sistólicos vem do modo com que os dados fluem entre os elementos de processamento. Tipicamente um arranjo sistólico é capaz de realizar operações simples como multiplicação de matrizes ou inversão. Eles são, portanto, máquinas de finalidade especial usados principalmente em equipamentos dedicados e não em computadores de finalidade geral.

Atualmente, o conceito de arquitetura sistólica é conhecido e utilizado em várias aplicações e sistemas de computação dedicados à solução de problemas numéricos particulares. Este capítulo aborda os conceitos que envolvem Arquiteturas Sistólicas, suas características, técnicas de projeto e aplicações.

4.2. Conceitos e Características

H. T. Kung (Kung, 82) definiu arquiteturas sistólicas como uma metodologia de mapeamento de computação de alto nível em estruturas de *hardware*. Basicamente, um arranjo sistólico é um sistema em que os dados fluem pela memória do computador ritmicamente, passando por vários elementos de processamento antes de retornar novamente para a memória do sistema. O *array* sistólico possui um conjunto de células interconectadas, capazes de executar operações simples. Por possuírem características como simplicidade, comunicação regular e estruturas de controle, essas células possuem muitas vantagens de implementação sobre outros tipos de elementos de processamento. Em sistemas sistólicos, essas células são organizadas em topologias tipo árvore ou "*array*s". A comunicação com o mundo externo é realizada apenas através das células de fronteira da topologia.

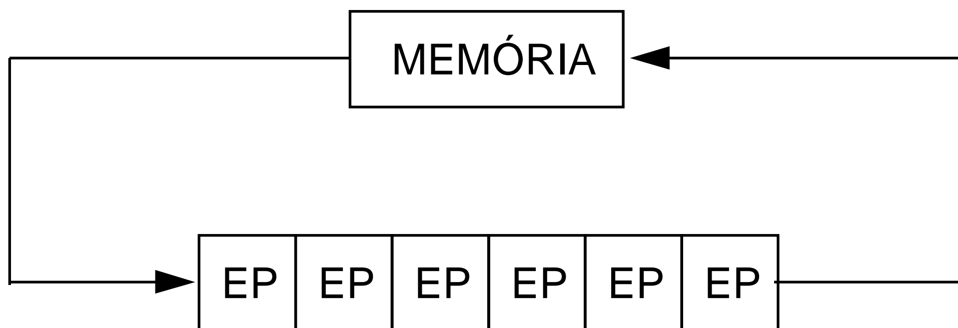


Figura 4.1 Princípio Básico de Arquiteturas Sistólicas.

Pode-se classificar as tarefas computacionais em duas grandes famílias: computação tipo *"compute-bound"* e computação tipo *"I/O-bound"*. Se o número de operações realizadas pelo computador for maior que o número de operações de entrada/saída, a computação é denominada *compute-bound*, caso contrário é denominada *I/O-bound*. Um exemplo de processamento tipo *compute-bound* é uma operação de multiplicação de duas matrizes, onde o número de operações de multiplicação e soma é bem maior que o número de operações de entrada de dados nas matrizes. Um exemplo de processamento *I/O-bound* é a soma de duas matrizes, onde o número de operações tipo soma é bem menor que o número de operações de carga dos dados das matrizes.

As arquiteturas sistólicas visam solucionar problemas computacionais tipo *compute-bound*, visto que os dados são aproveitados ao máximo pelo arranjo sistólico antes de voltar para a memória principal do sistema. Desse modo, restringe-se os acessos à memória principal fazendo com que o desempenho do sistema cresça como um todo. O princípio básico de um sistema sistólico consiste em substituir o elemento de processamento por um arranjo de elementos de processamento (figura 4.1), aumentando significativamente a capacidade de processamento da arquitetura, sem obrigatoriamente ser necessário aumentar a memória. A função da memória nesse caso é similar ao de um coração, ou seja, a memória deve fornecer "pulsos" de dados aos elementos de processamento. O problema dessa arquitetura se resume em garantir que os dados retirados da memória sejam efetivamente utilizados por cada célula de processamento e bombeados de célula para célula através do arranjo sistólico.

A partir das considerações iniciais podemos definir uma arquitetura sistólica como uma classe de arquiteturas que possuem as seguintes características :

- Regularidade Espacial e Localidade: esta característica estipula que uma arquitetura sistólica é composta por um número finito de células básicas de processamento que possuem uma determinada posição e conexão com a rede de acordo com a sua função. Assim, esse conceito está associado com a topologia da arquitetura que define as interconexões e a posição dos processadores. Na figura 4.2 apresenta-se alguns exemplos de topologias utilizadas.

- Regularidade Temporal e Sincronismo: uma característica muito interessante desse tipo de arquitetura é que ela intercala ciclos de comunicação com ciclos de processamento de dados. Desse modo, cada elemento de processamento deve possuir a lógica necessária para sincronização de processamento através de estímulos externos. Isto não exclui a possibilidade do caso de programas diferentes funcionarem em células distintas em diferentes instantes.

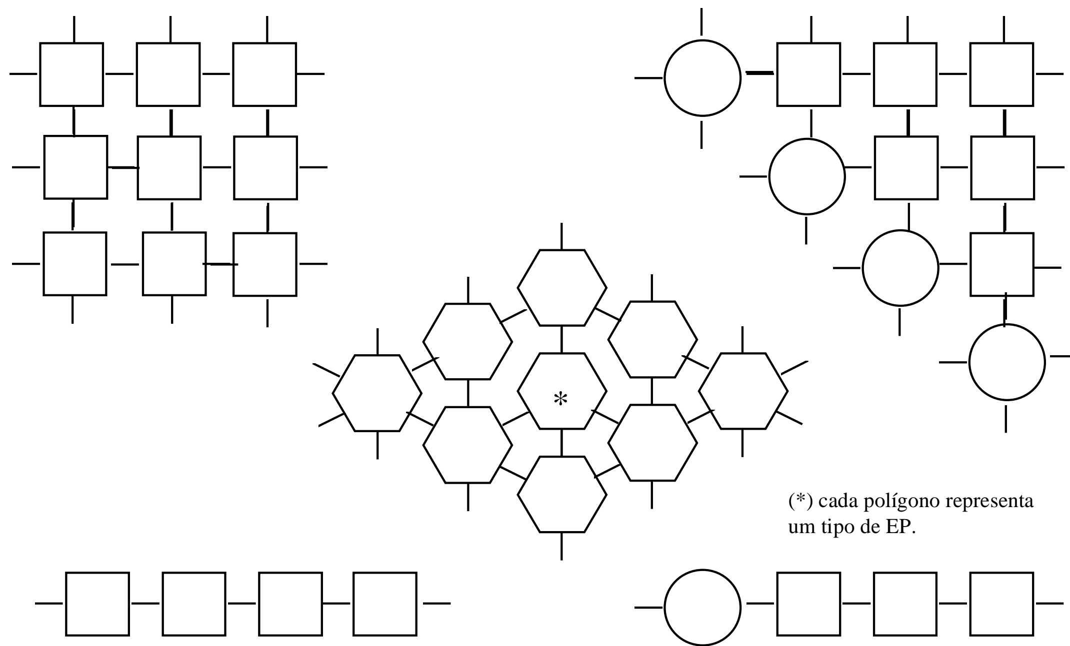


Figura 4.2 Topologias Sistólicas mais comuns.

- Computação *Pipeline* e Concorrência: arquiteturas sistólicas são adequadas a algoritmos que possuem características de computação *pipeline*. Assim, se para um algoritmo qualquer forem disponibilizados N processadores, pode-se ter um desempenho N vezes maior (teoricamente). A potencialidade de processamento de sistemas sistólicos resulta justamente do uso concorrente de várias unidades simples de processamento. Essa concorrência é obtida com o *pipeline* de estágios de processamento envolvendo operações simples.

- Entrada e Saída Fechadas: em arranjos sistólicos apenas as células de fronteira possuem conexão com o mundo exterior. Desse modo, caso se deseje carregar uma célula interna ao arranjo, serão necessários alguns ciclos (pulsos) para que a carga de dados se efetue. No caso de topologias lineares, as células internas também podem realizar operações de I/O.

- Modularidade: pela simplicidade das células básicas, o arranjo sistólico é extremamente modular, permitindo a replicação das células conforme elas sejam necessárias. A característica de modularidade facilita a implementação desses arranjos em VLSI, além de diminuir enormemente os custos de projeto e implementação.

4.3. Aplicações

Uma característica básica da maioria das aplicações que se utilizam de arquiteturas sistólicas é a grande demanda computacional e a necessidade de respostas em tempo real. São descritos a seguir as grandes áreas de aplicação das arquiteturas sistólicas e os principais algoritmos utilizados.

- Processamento de sinais (Drake et al.,87), (Bayomi, 91), processamento de imagens e reconhecimento de padrões: filtros FIR (*Finite Impulse Response*) e IIR (*Infinite Impulse Response*), convolução 1-D, convolução e correlação 2-D, transformada discreta de Fourier, interpolação, ordenação estatística, classificação de distância mínima, cálculo de matriz de covariância, classificação de sinais sísmicos, processamento de sinais de radar, reconhecimento sintático de padrões, detecção de curvas, reamostragem de imagens e etc.

- Álgebra Matricial: multiplicação de matrizes, triangularização de matrizes, operações sobre matrizes esparsas, solução de sistemas lineares triangulares e etc.

- Aplicações não numéricas: estruturas de dados, ordenação de filas e pilhas, algoritmos de grafos, reconhecimento de linguagens, programação dinâmica, arranjos aritméticos, operações sobre base de dados relacionais e etc.

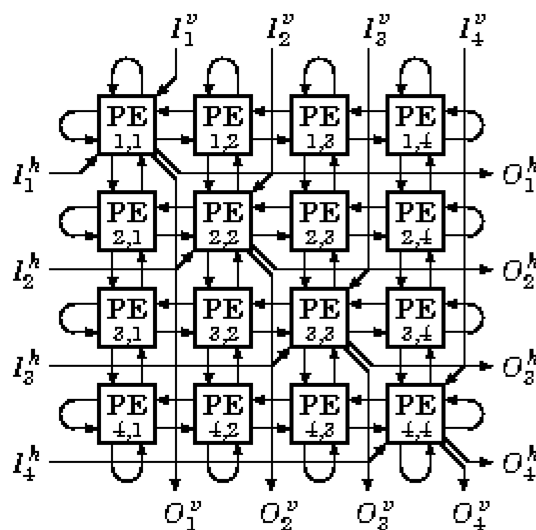


Figura 4.3 Arquitetura do arranjo sistólico (4x4 PEs) do chip GENES IV que implementa algoritmos de Redes Neurais Artificiais (ANN). (Viredaz, 94)

Esta classe de arquitetura tem sido também muito utilizada na implementação de algoritmos de Redes Neurais Artificiais (Viredaz, 94), como o arranjo ilustrado na figura 4.3, e na construção de máquinas denominadas *Neurocomputers*.

4.4. Técnicas de Projeto e Mapeamento

Para o projeto de um arranjo sistólico a partir de algoritmos, o projetista deve estar familiarizado com quatro princípios básicos: computação sistólica; a aplicação; o algoritmo e a tecnologia. Esforços vêm sendo realizados no sentido de desenvolver algumas técnicas de desenvolvimento sistemático para automatizar o projeto de arranjos sistólicos. Essas técnicas, porém, não substituem completamente o projetista. Por outro lado, explorando as características de arquiteturas sistólicas (modularidade, simplicidade, extensibilidade), essas técnicas vêm fornecendo com sucesso ferramentas e subsídios para auxiliar o projetista a propor várias alternativas para um determinado algoritmo. Essas técnicas estão relacionadas à derivação de uma descrição de alto nível do arranjo sistólico a partir da descrição do algoritmo. Geralmente essa descrição inclui o tamanho da topologia do arranjo, o processamento a ser realizado por cada unidade de processamento, a ordem dos dados e entradas e saídas do arranjo. Além disso, essas técnicas incorporam fatores tecnológicos e a inter-relação do arranjo com o resto do sistema. Para o projeto de um arranjo sistólico existe 4 linhas básicas que podem ser seguidas :

- Técnicas Heurísticas: Este é o meio direto de implementação de arranjos sistólicos, visto que a metodologia adotada é a experiência do projetista baseada em projetos anteriores. Para alguns algoritmos como filtros digitais tipo FIR e IIR, a implementação em *hardware* é direta pelas próprias características do algoritmo a ser implementado. Existem muitos métodos heurísticos de projeto de arquiteturas sistólicas para alguns importantes algoritmos. Porém, foi constatado que o projeto dessas arquiteturas é lento. São necessárias muitas simulações para a determinação de erros e essas técnicas heurísticas não têm se mostrado ótimas e nem corretas.

- Avaliação do Balanceamento das Operações: Esta técnica considera a implementação de um arranjo sistólico a partir do levantamento estatístico das operações necessárias para a execução de um determinado algoritmo. A topologia do arranjo é gerada de forma que cada célula convenientemente possua as operações necessárias para a execução do algoritmo. Assim, podem ser criadas células genéricas, capazes de

realizar certos tipos de operações (ou classes de processamento), e o projeto se restringe a determinar a ordem dessas células na topologia. Essa técnica foi utilizada no projeto WARP (cada célula da WARP é programada para realizar um tipo de operação do algoritmo).

- Utilização de Regras de Projeto Pré-definidas: Essa técnica consiste em determinar algumas regras que serão obedecidas no decorrer do projeto do arranjo sistólico. Ela é baseada em três pontos principais: uma linguagem para a descrição do algoritmo; um conjunto de regras de transformação e um sistema automático para aplicação dessas regras sobre a linguagem de descrição.

- Utilização de Métodos Semi-automáticos: Esta técnica baseia-se em automatizar a transformação de um algoritmo em sua implementação em *hardware* (Sedukhin & Sedukhin, 94). O princípio básico consiste em eliminar as dependências no fluxo de dados de forma automática. Essa dependência pode ser eliminada utilizando-se a representação de grafos de dependência (McCanny et al., 90).

4.5. Aspectos Relativos à Implementação

Alguns aspectos que devem ser considerados para a implementação de arranjos sistólicos são discutidos a seguir.

4.5.1. Granulosidade dos Elementos de Processamento

A operação básica realizada em cada ciclo por cada elemento de processamento pode ir desde uma operação sobre 1 bit até a execução de um programa completo. A escolha da granulosidade do processamento a ser realizado em cada célula básica depende de dois fatores: a tecnologia e a aplicação. Geralmente, a tecnologia é uma restrição a grandes granulosidades, pois quanto mais complexa for a operação, mais difícil é a sua implementação em *hardware*. Uma implementação flexível é a adoção de células programáveis, nas quais a granulosidade é determinada pelo nível de programação da célula.

4.5.2. Propósito Geral x Propósito Específico

Geralmente, um arranjo sistólico é fruto da implementação de um certo algoritmo que possua características adequadas. Porém é desejável que o arranjo sistólico projetado possa executar eficientemente o maior número possível de algoritmos. Existem dois modos de atacar o problema da "generalidade de propósito" durante a implementação do arranjo sistólico:

a) Um modo é fornecer ao sistema os recursos necessários de *hardware* para que haja a possibilidade de reconfiguração da topologia e modificação das interconexões das células da malha. Um exemplo desse método é a implementação da máquina *Chip* (*Configurable Highly Parallel Computer*).

a) o segundo método utiliza recursos de *software* para mapear diferentes algoritmos em uma topologia fixa. Esse método implica em utilizar linguagens que eficientemente exprimam computação paralela. Um exemplo dessa aplicação é a máquina WARP (Fortes & Wah, 87).

4.5.3. Particionamento de Algoritmos

Uma qualidade dos arranjos sistólicos é que para a solução de um problema não é necessário a implementação de um arranjo extenso. Geralmente, os algoritmos sistólicos podem ser particionados, ou seja, quebrados em pequenos problemas menores solucionáveis pelo arranjo. Entretanto, o particionamento incorreto do algoritmo pode ocasionar um aumento considerável da complexidade do arranjo. Um método para se resolver esse problema é identificar etapas de processamento dentro do algoritmo que podem ser executadas pelo arranjo sistólico. Assim, a solução do problema consiste na solução de vários problemas menores facilmente suportados pelo arranjo (Navarro et al., 87).

4.5.4. Blocos Construtivos Universais

Arranjos sistólicos possuem um custo de implementação menor que outras arquiteturas devido à possibilidade de replicação de suas células básicas e de seu *layout* denso e eficiente. Desse modo, podem ser criados blocos construtivos que são cuidadosamente projetados e otimizados, sendo que seu custo de implementação é amortizado com a reprodução em larga escala desses

blocos. O projeto modular de arranjos sistólicos permite aos projetistas, que desejam uma rápida prototipação, utilizar dispositivos comerciais em suas implementações como memórias, microprocessadores de ponto fixo e ponto flutuante. Por outro lado, esses *chips* podem não ser suficientemente adequados ao arranjo em questão. Muitos processadores comerciais vêm sendo utilizados em arranjos sistólicos como DSPs. Alguns problemas surgem ou outros são solucionados ao se utilizar esses tipos de componentes como: ferramentas para desenvolvimento de programas e suporte para interconexões flexíveis no arranjo.

4.5.5. Integração em Sistemas já Existentes

As implementações sistólicas em geral são máquinas escravas associadas a algum sistema de propósito geral que forneça recursos de *software*, como compiladores e programas de gerenciamento e distribuição de processamento para o arranjo. Apesar disso, um problema que pode acarretar sérias complicações é a integração do arranjo sistólico no sistema hospedeiro. Isto se deve às próprias características da arquitetura sistólica, como banda larga de entrada e saída de dados (principalmente para algoritmos particionados, em que repetidamente é necessário injetar dados no arranjo sistólico). Existem outros problemas como a interconexão do arranjo sistólico com o computador hospedeiro e o subsistema de memória para suportar o arranjo.

4.6. Considerações Finais

Os arranjos sistólicos podem ser vistos como a forma mais natural de se implementar algoritmos em *hardware*. Podem ser usados para resolver muitos problemas regulares contendo operações repetidas em grandes quantidade de dados. As arquiteturas sistólicas são uma classe de arquiteturas que pelas suas características são facilmente implementáveis em VLSI, tanto que são consideradas como uma arquitetura estreitamente ligada a essa tecnologia (Hwang, 84). As vantagens das arquiteturas sistólicas aproveitáveis em VLSI são a simplicidade, regularidade e modularidade que facilitam o projeto. Assim, o projetista só precisa se limitar a projetar uma célula básica e, a partir desta, replicá-la quantas vezes a topologia sistólica exigir. Essas características minimizam consideravelmente o custo de implementações em VLSI.

Os arranjos sistólicos geralmente não são adequados para computadores de finalidade geral, embora com o desenvolvimento de elementos de processamento programáveis mais gerais,

arquiteturas sistólicas programáveis podem ser desenvolvidas (Gokhale et al., 91), (Blickle, et al, 94). Assim, a tecnologia FPGA parece ser uma excelente alternativa para implementação de arranjos sistólicos (Marnane et al., 95). Com essa tecnologia, as vantagens citadas para VLSI podem ser igualmente aproveitadas e, além disso, é possível projetar topologias mais flexíveis que possam executar vários algoritmos, explorando-se as características de reprogramação e da própria arquitetura dos FPGAs.

5. Sistemas Lineares

5.1. Considerações Iniciais

Em diversas áreas do conhecimento humano existem problemas que podem ser modelados com sistemas lineares ou através de modelos linearizados. Os sistemas de equações lineares estão associados com muitos problemas em engenharia e ciência, assim como aplicações de matemática nas ciências sociais, economia e finanças. Com o aumento da complexidade dos problemas, resolvê-los torna-se uma tarefa computacional muito intensa, principalmente quando há requisitos de tempo, no caso de aplicações de tempo real.

Tratando-se de uma classe de problemas com tantas aplicações , é natural que seja objeto de muito estudo e discussão na comunidade científica, a qual tem proposto uma grande variedade de métodos de solução. Entretanto, a eficiência desses métodos depende das características particulares do problema e também da forma de implementação, que pode ser tanto em *software* como em *hardware*.

Neste capítulo serão apresentados alguns métodos de solução. Nosso foco serão os métodos iterativos que em geral são mais simples de implementar e possuem características de paralelismo que podem ser exploradas para se obter uma implementação eficiente.

5.2. Sistemas de Equações Lineares

Um sistema linear é descrito da seguinte maneira:

$$E_1: a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$

$$E_2: a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$$

$$\vdots$$

$$E_n: a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n$$

ou na forma matricial

$$\mathbf{Ax} = \mathbf{b} \qquad (5.1)$$

onde $\mathbf{A} \in \Re^{nxn}$ e representa a matriz de coeficientes $\mathbf{a_{ij}}$ (i, j = 1, 2,...,n) , $\mathbf{x}$ e $\mathbf{b}$ $\in \Re^{nx1}$ e representam os vetores das incógnitas $\mathbf{x_i}$ (i=1,2, ...,n) e dos termos constantes $\mathbf{b_i}$ (i= 1,2, ...,n), respectivamente. Resolver o sistema linear significa determinar as incógnitas x_1, …, x_n que satisfazem as equações, dados os elementos a_{ij} e b_i.

5.3. Métodos para Solução de Sistemas Lineares

Os métodos para solução de sistemas de equações lineares são divididos principalmente em dois grupos (Burden & Faires, 85; Golub & Van Loan, 89):

- Métodos Exatos ou Diretos: são aqueles que forneceriam a solução exata, se não fossem os erros de arredondamento, com um número finito de operações. Como exemplos de métodos exatos pode-se citar: Decomposição LU, Processo de Cholesky e o Método de Eliminação de Gauss, dentre outros.

- Métodos Iterativos: são aqueles que permitem obter as raízes de um sistema com uma dada precisão através de um processo infinito convergente. Os métodos iterativos fornecem uma sequência de aproximações da solução, através da repetição de um processo.

5.3.1. Métodos Diretos

Os métodos diretos são baseados principalmente na decomposição ou fatorização LU, que é uma operação de difícil implementação computacional. Dentre esses, o Método de Eliminação de Gauss é provavelmente o mais conhecido. A seguir é apresentado um resumo das principais etapas desse método.

O primeiro passo do método de Eliminação de Gauss é reduzir o conjunto de equações a forma triangular. Um sistema linear (nxn) na forma triangular apresenta uma equação expressa em termos de somente uma das incógnitas; uma segunda equação com duas incógnitas, umas delas sendo a da primeira equação e assim sucessivamente com cada nova equação tendo uma

incógnita adicional, até a n-ésima equação, que terá n incógnitas. Em termos de uma equação matricial, isto corresponde a multiplicar ambos os lados da equação (5.1) por alguma matriz quadrada, que será chamada $T \in R^{nxn}$, tal que TAx = Tb, e onde TA é uma matriz triangular.

Tipicamente as equações não estão na forma triangular . A forma triangular é alcançada usando-se uma equação (uma linha de Ax = b), que chamamos de E_1 para eliminar uma incógnita, que chamamos de x_1 , de todas as outras equações E_2 a E_n. A incógnita x_1 é eliminada de E_2 multiplicando-se ambos os lados de E_1 por uma constante escolhida de tal forma que quando a equação modificada E_1' é adicionada a E_2 , o termo envolvendo x_1 na equação E_2 será eliminado. Como há n termos em E_1 e E_2 envolvendo incógnitas e uma constante, este processo levará O(n) operações para multiplicar todos os termos de E_1 pela constante apropriada e para somar os temos de E_1 e E_2.

Se há equações que não têm termos para todas as incógnitas , o trabalho requerido para se alcançar a forma triangular é reduzido. Entretanto, a discussão aqui irá focar o caso onde todas as incógnitas aparecem em todas as equações. Neste caso, o processo deve se repetir para o restante das equações, resultando em um total de $O(n^2)$ operações para eliminar uma incógnita de n-1 equações.

O próximo passo é eliminar a segunda incógnita de n-2 equações, de E_3 até E_n. O terceiro passo é eliminar a terceira incógnita de n-3 equações, de E_4 até En, e assim por diante. Portanto, para reduzir as equações para a forma triangular são requiridas $O(n^3)$ operações.

Uma vez reduzidas para a forma triangular, a substituição é utilizada para resolver todas as incógnitas. Este processo não é discutido aqui, mas requer $O(n^2)$ operações. Portanto, o Método de Eliminação de Gauss é de ordem $O(n^3)$.

5.3.2. Método Iterativos

O termo "Método Iterativo" refere-se a ampla gama de técnicas que se utilizam de aproximações sucessivas para obter uma solução mais precisa para um sistema linear a cada iteração ou passo.

Há basicamente dois tipos de métodos iterativos:

- Métodos estacionários: são os mais antigos, simples de entender e implementar, mas geralmente não muito eficientes. O método é dito estacionário pois cada aproximação é obtida da anterior sempre pelo mesmo processo. Os métodos iterativos estacionários mais conhecidos são o método de Jacobi e o método de Gauss-Seidel, embora haja outros mais modernos como o SOR (Sobre-relaxação Sucessiva) e o SSOR (Sobre-relaxação Sucessiva Simétrica).

- Métodos não-estacionários: são de desenvolvimento relativamente recente. Nestes, os processos variam de passo para passo. A maioria é baseada na idéia de sequência de vetores ortogonais. Embora sejam mais complexos de entender e implementar, geralmente são mais eficientes. Podemos citar como exemplo os métodos do Gradiente Conjugado (CG), o do Mínimo Resíduo (MINRES) e o método de Chebyshev, dentre outros.

A taxa em que um método iterativo converge depende muito do espectro da matriz de coeficentes. Por isso, os métodos iterativos normalmente utilizam uma segunda matriz que transforma a matriz de coeficientes numa matriz que apresenta um espectro mais favorável. Esta matriz de transformação é chamada pré-condicionador. Um bom pré-condicionador melhora a convergência do método o suficiente para compensar o custo extra de construí-lo e aplicá-lo. Sem um pré-condicionador o método iterativo pode até mesmo não convergir.

Outro aspecto importante nos métodos iterativos é o estabelecimento do critério de parada. Um bom critério deve identificar quando o erro é suficientemente pequeno ou quando este não está mais sofrendo variações significativas e também deve limitar o tempo máximo gasto nas iterações.

5.3.2.1. Método Iterativos Estacionários

Um método iterativo começa com uma estimativa inicial da solução e com sucessiva iterações melhora esta estimativa inicial até alcançar a precisão requerida. Com n equações, cada uma com n incógnitas, a complexidade para uma única iteração é $O(n^2)$. A relação entre as sucessivas aproximações para os métodos iterativos estacionários pode ser expressa como

$$\mathbf{v}(k+1) = \mathbf{F}\mathbf{v}(k) + \mathbf{g} \qquad (5.2)$$

onde $\mathbf{v}(k) \in \Re^{n x 1}$ é a k-ésima aproximação de x da equação (5.1). A matriz $\mathbf{F} \in \Re^{n x n}$ e o vetor $\mathbf{g} \in \Re^{n x 1}$ são constantes dependentes de **A** e **b** da equação (5.1) e do método iterativo utilizado. A seguir apresentamos um resumo de alguns métodos iterativos estacionários.

5.3.2.1.1. Método RF

A expressão iterativa para o método RF é:

$$\mathbf{v}(k+1) = (\mathbf{I}-\mathbf{A})\mathbf{v}(k) + \mathbf{b}$$

onde A, b e v são os definidos na equação (5.1), e $\mathbf{I} \in \Re^{n x n}$ é a matriz identidade. Em termos da equação (5.2),

$$\mathbf{F} = \mathbf{I} - \mathbf{A} \text{ e } \mathbf{g} = \mathbf{b}.$$

5.3.2.1.2. Método de Jacobi

O método de Jacobi é similar ao método RF, exceto que cada equação é inicialmente dividida por uma constante. Cada constante é escolhida de tal forma que os elementos diagonais da matriz A serão igualados a 1, resultando numa matriz (**I** – **A**) com elementos diagonais nulos. Isto é expresso como:

$$\mathbf{v}(k+1) = (\mathbf{I} - \mathbf{D}^{-1}\mathbf{A})\mathbf{v}(k) + \mathbf{D}^{-1}\mathbf{b} \tag{5.3}$$

onde $\mathbf{D} \in \Re^{n x n}$ é a matriz diagonal contendo todos os elementos diagonais de A. Podemos definir também $\mathbf{L} \in \Re^{n x n}$ como matriz diagonal inferior e $\mathbf{U} \in \Re^{n x n}$ como a matriz triangular superior, tal que

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}.$$

Substituindo a expressão para **A** na equação (5.3), teríamos que

$$\mathbf{v}(k+1) = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{v}(k) + \mathbf{D}^{-1}\mathbf{b}$$

Em termos da expressão geral dada na Equação (5.2) temos:

$$\mathbf{F} = \mathbf{I} - \mathbf{D}^{-1}\mathbf{A} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U}) \text{, e } \mathbf{g} = \mathbf{D}^{-1}\mathbf{b}.$$

A discussão do mapeamento do método de Jacobi em um arranjo linear de elementos de processamento é dada em (Leighton, 92).

5.3.2.1.3. Método de Gauss-Seidel

Os métodos de RF e Jacobi são métodos de correções simultâneas, onde todos os elementos de v(k+1) são calculados antes de proceder com a próxima iteração. Já o método de Gauss-Seidel realiza correções sucessivas, utilizando os valores atualizados de v(k+1) tão logo estejam disponíveis. O método de Gauss-Seidel começa calculando um único elemento de v(k+1). Esse elemento é então usado no lugar do elemento correspondente de v(k) para calcular o próximo elemento de v(k+1) em uma execução sequencial. Esse processo se repete, usando os elementos calculados de v(k+1) para calcular o próximo elemento de v(k+1) até que o vetor inteiro de v(k+1) é determinado. Esse método pode ser expresso como:

$$\mathbf{v}(k+1) = -\mathbf{D}^{-1}\mathbf{L}\mathbf{v}(k+1) - \mathbf{D}^{-1}\mathbf{U}\mathbf{v}(k) + \mathbf{D}^{-1}\mathbf{b}.$$

onde D, L e U são os mesmos definidos para o método de Jacobi e A, b e v são definidos na equação 5.1. Em (Lei92) há alguma orientação de como esse método pode ser implementado num arranjo linear de elementos de processamento. Rearranjando os termos, pode-se expressar o método de Gauss-Seidel nos termos da Equação (5.2), resultando em

$$\mathbf{F} = -(\mathbf{D} + \mathbf{L})^{-1}\mathbf{U}, \ \mathbf{g} = -(\mathbf{D} + \mathbf{L})^{-1}\mathbf{b}.$$

5.3.2.1.4. Método Baseado no Rede Neural Analógica de Wang e Li

Recentemente, Wang e Li (1993) propuseram uma rede neural recorrente (RNN) analógica para a solução de sistemas de equações lineares. Variações dessa RNN pode acomodar a soluções de outros problemas, incluindo inversão de matrizes, ordenação e programação linear. Interessantemente, a forma discretizada desta RNN se enquadra na categoria de métodos iterativos estacionários.

A dinâmica da RNN proposta pode ser descrita como:

$$\mu\,\dot{\mathbf{v}}(t) = -\mathbf{A}^T\mathbf{A}\mathbf{v}(t) + \mathbf{A}^T\mathbf{b}\,, \tag{5.4}$$

onde $\mathbf{v}(t) \in \Re^{n\times 1}$ é o vetor que descreve os estados de cada nó, e $\dot{\mathbf{v}}(t) \in \Re^{n\times 1}$ é taxa de mudança dos estados ao longo do tempo, e **A** e **b** são os definidos na equação (5.1).

Usando a Rede Neural, Wang e Li demonstraram que:

1) O sistema em (5.4) é sempre estável;

2) estado estável do sistema em (5.4) é a solução para o problema em (5.1), isto é , $\mathbf{v}(\infty) = \mathbf{x}^*$ tal que $\mathbf{A}\mathbf{x}^*=\mathbf{b}$.

Usando termos mais familiares a Teoria de Redes Neurais, pode-se definir

$$\mathbf{W} = -\mathbf{A}^T\mathbf{A} \in \Re^{nxn}$$

como a matriz peso de conexão e

$$\boldsymbol{\theta} = \mathbf{A}^T\mathbf{b} \in \Re^{nx1}$$

o vetor de limiar da rede neural proposta.

Assim, pode-se reescrever a equação (5.4) como:

$$\mu\dot{\mathbf{v}}(t) = -\,\mathbf{W}\mathbf{v}(t) + \boldsymbol{\theta}\,. \tag{5.5}$$

Redes Neurais analógicas desse tipo são tipicamente sistemas dedicados e requerem um grande número de interconexões com limitada reconfigurabilidade. Portanto, é desejável encontrar a forma discretizada dessa rede neural. Usando o método de Euler, podemos discretizar a equação (5.5) como:

$$\mu\frac{\mathbf{v}(k+1) - \mathbf{v}(k)}{\Delta t} = -\mathbf{W}\mathbf{v}(k) + \boldsymbol{\theta}\ .$$

Isto nos dá uma relação entre os estados dos neurônios (v) nos instantes de tempo k e k+1. Arranjando termos, pode-se criar uma expressão iterativa:

$$\mathbf{v}(k+1)\ = \left|\ \mathbf{I} - \frac{\Delta t}{\mu}\mathbf{W}\right|\mathbf{v}(k) + \frac{\Delta t}{\mu}\boldsymbol{\theta}$$

Pode-se reescrevê-la em termos da Equação (5.2) definindo:

$$\mathbf{F} = \mathbf{I} - c\mathbf{W}$$

$$\mathbf{g} = c\boldsymbol{\theta}\,.$$

$$\text{onde } c = \frac{\Delta t}{\mu}\,.$$

A constante c é referenciada como um fator de extrapolação. Note que c é uma constante que podemos escolher livremente. A escolha apropriada de c irá garantir a convergência.

Este método é de fato muito similar ao método iterativo RF. A matriz $\mathbf{W}=\mathbf{A}^T\mathbf{A}$ é simétrica e definida positiva se **A** é não-singular, o que implica que todos os auto-valores são reais e positivos.

5.4. Aspectos Computacionais dos Métodos

A resolução eficiente de um sistema linear é muito dependente da escolha apropriada do método iterativo. Nem todo método irá funcionar para todo tipo de problema, logo o conhecimento das propriedades da matriz é o principal critério para a seleção do método iterativo. Entretanto, para se obter um bom desempenho, também devem ser considerados o núcleo computacional do método e o quão eficiente pode ser sua execução na arquitetura alvo. Este ponto é de particular importância em arquiteturas paralelas (Barrett et al., 94).

Os métodos iterativos são muito diferentes dos métodos diretos nesse aspecto. O desempenho dos métodos diretos, tanto para matrizes densas ou esparsas, depende basicamente da fatorização da matriz. Esta operação é ausente nos métodos iterativos, portanto, espera-se para os métodos iterativos uma taxa de operações de ponto flutuante menor do que para os métodos diretos.

Além disso, os métodos iterativos são geralmente mais simples de serem implementados que os métodos diretos, e como não há fatorização para ser armazenada, eles podem tratar sistemas bem maiores que os métodos diretos com uma mesma quantidade de memória.

A seleção do melhor método para uma determinada classe de problema é um exercício de tentativa e erro. A avaliação de aspectos como a quantidade de memória disponível e o custo das operações (como multiplicação matriz-vetor, produtos internos, inversão e etc) contribuem para uma escolha mais apropriada. Baseando-se no problema em particular ou na estrutura de dados, o usuário pode observar qual operação pode ser realizada mais eficientemente na arquitetura alvo.

5.5. Considerações Finais

As aplicações que envolvem solução de sistemas lineares são uma classe típica de problemas que necessitam de soluções personalizadas. Como vimos, há diversas alternativas de métodos que podem ser empregados, levando-se em consideração principalmente as características particulares do sistema em questão. Além disso, quando há requisitos de tempo, é essencial uma arquitetura com alto poder computacional, pois as operações matriciais são computacionalmente intensivas.

A abordagem que tem sido mais utilizada é a adoção de algoritmos e arquiteturas paralelas, como computadores vetoriais (Dongarra et al., 91). Entretanto, muitas vezes é necessário desenvolver arquiteturas dedicadas, sendo as arquiteturas sistólicas as mais adequadas para manipular cálculos matriciais, pois exploram o paralelismo de granulosidade fina presente nessas operações e possuem um baixo *overhead* de comunicação e sincronismo (Moreno & Lang, 90).

6. Arquitetura Proposta e Desenvolvimento do Projeto

6.1. Considerações Iniciais

Neste capítulo são apresentadas duas versões de uma Arquitetura Sistólica para Solução de Sistemas de Equações Lineares. A arquitetura é baseada em uma topologia em anel e execução síncrona, permitindo o uso de um controlador global para todos os elementos de processamento. O arranjo sistólico deverá ser conectado a uma máquina hospedeira, a qual será responsável por algumas tarefas do processamento e controle. Os detalhes de comunicação e interface entre o arranjo sistólico e a máquina hospedeira não são tratados neste projeto, sendo abordados em um outro trabalho de mestrado do LaSD-ICMC.

A primeira versão da arquitetura proposta é para a solução de sistemas de equações onde o número de equações é menor ou igual ao número de elementos de processamento (EPs). A segunda versão foi projetada com algumas alterações no elemento de processamento para acomodar a solução de sistemas com um número de equações maior que o número de EPs. Ambas as versões foram verificadas por simulação em *software* (programa em linguagem C e ferramentas de simulação de *hardware* digital). Os elementos de processamento foram projetados utilizando-se técnicas modernas de desenvolvimento de *hardware* e implementadas em circuitos FPGAs da Xilinx.

Nas próximas seções, o funcionamento das arquiteturas é explanado para em seguida discutir-se detalhes da implementação.

6.2. Arquitetura Proposta

Como visto no capítulo anterior, há diversos métodos de solução de sistemas lineares. A maioria dos métodos envolve basicamente cálculos matriciais como multiplicação de matrizes por constantes, vetores ou outras matrizes, inversão, produtos-internos, etc. A implementação eficiente dos métodos geralmente explora técnicas de paralelismo tanto em *software* como em *hardware*.

O mapeamento de cálculos matriciais em arquiteturas sistólicas é muito natural, sendo esta uma das principais áreas de aplicação dessas arquiteturas. Em particular, o mapeamento da operação conhecida na literatura como Gaxpy[1] (*General Ax plus y*) é bem discutido (Moreno & Lang, 90), (Golub, 89).

Em (Hillesland, 97) foi proposta a realização digital do método de Wang e Li. A forma discretizada deste método, originalmente baseado em redes neurais, enquadra-se como um método iterativo estacionário que é baseado, assim como outros, na operação Gaxpy. A arquitetura sugerida utiliza um arranjo sistólico nx1 conectado em anel. O arranjo sistólico atua como um máquina computacional conectada a uma máquina hospedeira. A máquina hospedeira é responsável pelas seguintes tarefas de pré-processamento:

- calcular a matriz **F** e o vetor **g** da Equação (5.2). Este cálculo varia de acordo com o método empregado.
- Rearranjar os elementos da matriz **F** e do vetor **g** para carregar os dados apropriadamente no arranjo sistólico.
- Iniciar os cálculos do arranjo sistólico.
- Determinar parâmetros para a condição de parada.
- Coletar os resultados calculados pelo arranjo sistólico.

A arquitetura do arranjo sistólico proposto é ilustrada na figura 6.1. Cada elemento de processamento do arranjo sistólico é construído baseado no conceito de MAC (multiplicador acumulador), como mostra a figura 6.2, onde $f_{i,j}$ é o elemento da i-ésima linha,j-ésima coluna da matriz F, g_i é o i-ésimo elemento de g, e $v_i(k)$ é o i-ésimo elemento de v(k).

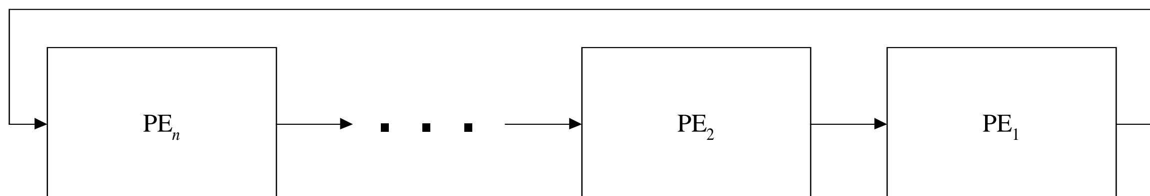


Figura 6.1 Topologia em Anel da Arquitetura Proposta.

[1] Operação na forma z = y + Ax , onde z e y ∈ $\Re^m$, A ∈ $\Re^{mxn}$ e x ∈ $\Re^n$

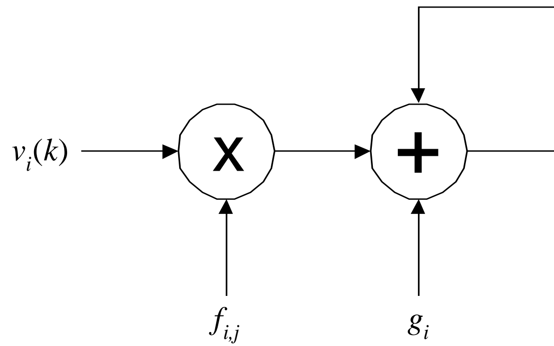


Figura 6.2 Estrutura Básica do EP.

A partir dessa arquitetura geral, duas versões de projeto são apresentadas. A primeira versão é adequada para os casos em que o tamanho do sistema a ser resolvido (denominado n) é menor ou igual ao número de EPs (denominado N). A segunda versão da arquitetura se propõe a acomodar os casos em que o tamanho do sistema é maior que o número de EPs , ou seja, n> N.

6.2.1. Versão 1: Arquitetura para Sistemas de tamanho igual ao número de EPs no anel (n = N)

Nesta arquitetura, cada processador será responsável por calcular um único elemento do vetor v(k+1), apresentado na Equação 5.2. Reescrevendo essa equação para expressar a computação realizada por cada elemento de processamento (p), temos:

$$v_p(k+1) = \mathbf{f}_p \mathbf{v}(k) + g_p \tag{6.1}$$

onde $v_p(k)$ é o p-ésimo elemento de $\mathbf{v}(k)$, $\mathbf{f}_p \in \Re^{n\,x\,1}$ é a p-ésima linha da matriz $\mathbf{F}$, $\mathbf{v}(k) \in \Re^{n\,x\,1}$ é o vetor dado na Equação (5.2), e g_p é o p-ésimo elemento de $\mathbf{g}$. Os elementos de $\mathbf{v}(k)$ circulam de EP em EP através do anel. Cada EP irá receber os termos $\mathbf{f}_p$ e g_p para calcular quando necessário. A ordem em que os termos de $\mathbf{f}_p$ e $\mathbf{v}(k)$ são multiplicados e acumulados é determinada através de como o anel é utilizado.

O p-ésimo EP irá ser usado para computar $v_p(k+1)$. Inicialmente o p-ésimo EP terá disponível o elemento $v_p(k)$. O próximo elemento de $\mathbf{v}(k)$, que é recebido do EP anterior no anel, será o elemento $v_{p \bmod n\,+\,1}$. Após um número i de passos, o p-ésimo EP irá receber o elemento $v_{(p+i-1) \bmod n\,+\,1}$. A ordem em que cada processador recebe os elementos de $\mathbf{v}(k)$ também dita a ordem

em que os elementos do vetor-linha $\mathbf{f}_p$ são utilizados. A máquina hospedeira é responsável por rearranjar os elementos de $\mathbf{f}_\mathrm{p}$ em vetores-linhas $\mathbf{d}$, para cada processador p, de forma que correspondam a ordem em que os elementos de $\mathbf{v}(k)$ serão recebidos pelo EP. A estrutura de dados de $\mathbf{d}$ para o elemento de processamento p pode ser descrita como:

$$d_i = f_{p,(p+i-2)\bmod n+1}$$

Como exemplo, será considerado o caso em que o número de EPs N=3 e o número de equações lineares n=3. O vetor $\mathbf{d}$ para cada EP é dado abaixo.

EP$_1$ EP$_2$ EP$_3$

$$\begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}^T = \begin{bmatrix} f_{1,1} \\ f_{1,2} \\ f_{1,3} \end{bmatrix}^T \qquad \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}^T = \begin{bmatrix} f_{2,2} \\ f_{2,3} \\ f_{2,1} \end{bmatrix}^T \qquad \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}^T = \begin{bmatrix} f_{3,3} \\ f_{3,1} \\ f_{3,2} \end{bmatrix}^T$$

Um diagrama de fluxo de dados do arranjo sistólico é apresentado na figura 6.3. No p-ésimo EP, o bloco rotulado R1 é um registrador contendo o valor de $v_p(k)$ correntemente sendo usado pelo MAC para o cálculo de $v_p(k+1)$. O elementos $v_p(k)$ indicados são aqueles que deveriam estar presentes no início de uma iteração k. O componente ACC é um registrador no qual os termos da Equação (6.1) são acumulados para calcular $v_p(k+1)$. No primeiro passo da iteração, ACC é carregado com g_p. Por exemplo, no EP$_1$, o acumulador será inicialmente carregado com o valor g_1.

Tendo sido carregado no acumulador o valor apropriado de g_p, o valor $v_p(k)$ em R1 é multiplicado por um elemento de $\mathbf{f}_p$ e acumulado em ACC. O mesmo $v_p(k)$ é então passado para o registrador R1 do próximo EP. Da mesma maneira, cada EP usa os valores de $\mathbf{v}(k)$ para calcular um elemento de $\mathbf{v}(k+1)$. Desta forma cada iteração levará $O(n)$ passos. Quando a iteração é completada, o acumulador ACC do p-ésimo EP irá conter o p-ésimo elemento de $\mathbf{v}(k+1)$. Este valor é então carregado em R1 , que irá ser o p-ésimo valor de $\mathbf{v}(k)$ para a próxima

iteração. Como todos os EPs são utilizados ao mesmo tempo, há um ganho de desempenho em relação a uma implementação seqüencial em uma máquina com um único processador.

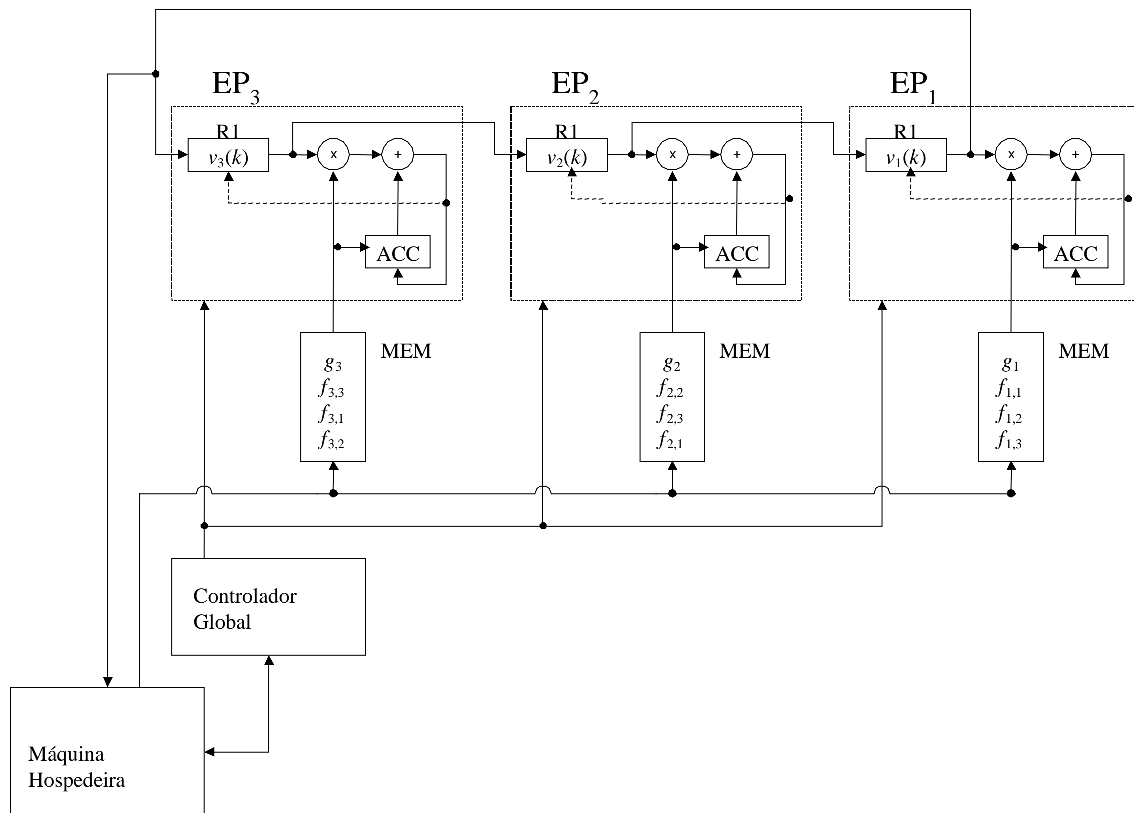


Figura 6.3 Diagrama de Fluxo de Dados para* n=*3,* N=*3.

Os elementos de $\mathbf{f}_p$ e o valor g_p são apresentados em um único elemento de memória para cada EP na figura 6.3. Isto foi feito com a finalidade de mostrar a sequência em que os elementos de $\mathbf{f}_p$ e o valor g_p serão utilizados pelo EP. Também é melhor para observar que somente um caminho (*datapath*) é requerido para os dados dos elementos de $\mathbf{f}_p$ e do valor g_p, e que somente um valor precisa ser acessado a cada instante.

Usando a topologia em anel evita-se o *overhead* associado com o *pipeline*. De fato, n passos seriam necessários para preencher o *pipeline*. As operações realizadas por todos os EPs estão sincronizadas. Isto permite o uso de um único controlador global, evitando o custo de *hardware* adicional para realizar o controle em cada EP.

Estando os cálculos completos, será necessário extrair os valores finais de $\mathbf{v}(k)$ dos EPs. Isto pode ser feito através de um "*tap*" em uma conexão do anel e rotacionando-se a informação

(v(k)) mais uma vez através dos processadores. Um esquema de comunicação detalhado com uma máquina hospedeira não faz parte deste projeto. Um diagrama (*datapath*) em nível de registradores (RTL) para um único EP é mostrado na figura 6.4. Como este processador requer o uso de um MAC, o diagrama (RTL) para o MAC é apresentado na figura 6.5.

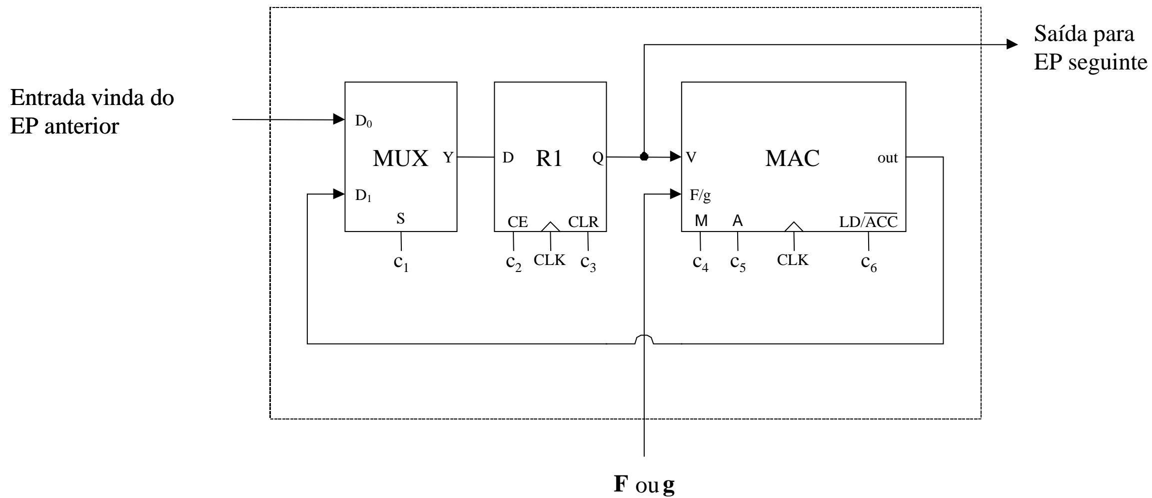


***Figura 6.4 Diagrama RTL do EP na Arquitetura para* N=n .**

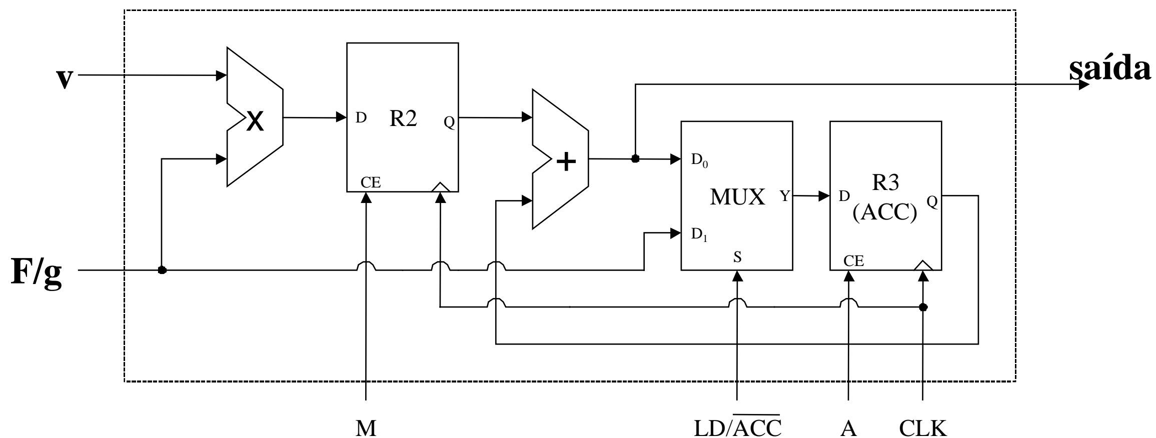
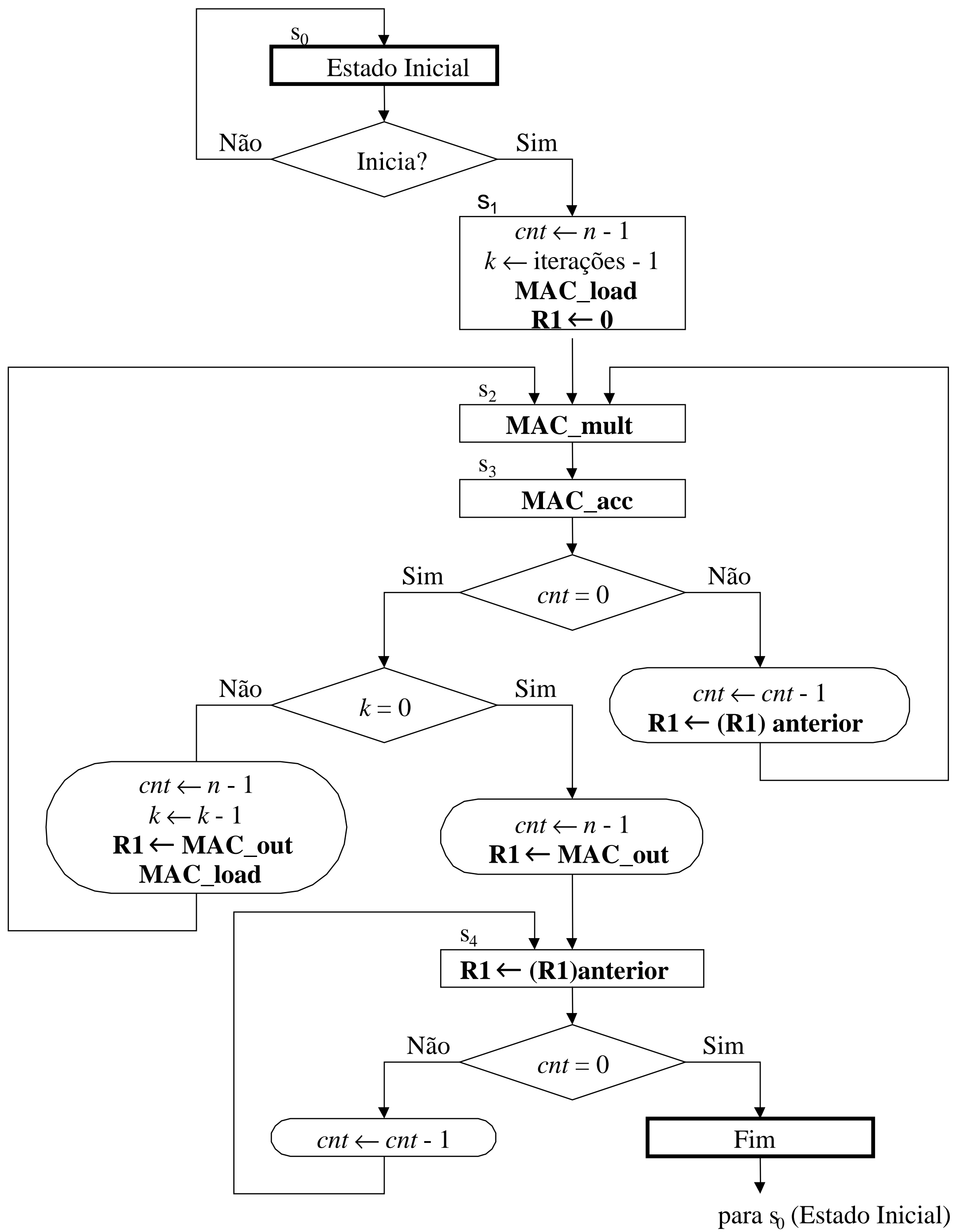


Figura 6.5 Diagrama de Fluxo de Dados RTL para o MAC.

A operação do arranjo sistólico é descrita no fluxograma da figura 6.6. Alguns parâmetros de controle estão inclusos, mas que não fazem parte do EP. Somente as operações em negrito devem ser realizadas em cada EP. O fluxograma inclui os contadores *cnt* e *k* , que fazem parte do controlador. O termo “iterações” é o número total de iterações que serão realizadas. Os sinais de controle denominados como c_1, c_2, ..., c6 figura 6.4, são derivados dos estados apresentados no fluxograma da figura 6.6 como mostra a tabela 6.1.

Tabela 6.1 Sinais de Controle para os EPs da Arquitetura para n=N

Sinais de Controle	*Equações Lógicas*
c_1	$s_3(cnt=0)$
c_2	$s_1 + s_3(k\neq 0) + s_4$
c_3	s_1
c_4	s_2
c_5	$s_1 + s_3$
c_6	$s_1 + s_3(cnt=0)$



MAC_load carrega g_p no registrador ACC do MAC.

Figura 6.6 Fluxograma da Arquitetura para n=N.

6.2.2. Versão 2: Arquitetura para Sistemas maiores que o número de EPs no anel (n > N)

Quando o número de equações a serem resolvidas é maior que o número de EPs, pode-se atribuir a cada EP o cálculo de mais de um elemento de $\mathbf{v}(k+1)$. Por simplicidade, assumi-se que o tamanho do sistema, n, é um múltiplo do número de EPs, N. Se este não for o caso, o tamanho do problema pode ser expandido para o próximo múltiplo de N, adicionando-se zeros a $\mathbf{v}$, $\mathbf{F}$, e $\mathbf{g}$.

Inicia-se definindo um novo valor , $q = n/N$. Este valor representa o número de elementos de $\mathbf{v}(k+1)$ dado na equação (6.1) que deverão ser calculados por cada EP. Cada p-ésimo EP irá calcular os elementos $(p\text{-}1)q+1$ até pq de $\mathbf{v}(k+1)$. Para um dado elemento de processamento p, serão designados q elementos de $\mathbf{v}(k+1)$, cujos índices são dados por $((p\text{-}1)\ q + i)$, i variando de 1 a q. A equação 6.1 é reescrita como

$$v_i(k+1) = \mathbf{f}_i \mathbf{v}(k) + g_i \qquad (6.1')$$

para cada EP, onde aqui $\mathbf{f}_i \in \Re^{nx1}$ é o $((p\text{-}1)q+i)$-ésimo vetor-linha de $\mathbf{F}$, e g_i é o $((p\text{-}1)q+i)$-ésimo elemento de $\mathbf{g}$, $i=1, 2, \ldots q$.

Os elementos de $\mathbf{v}(k)$ se deslocam de EP para EP através do anel. Como somente um elemento de $\mathbf{v}(k)$ é usado pelo MAC de cada EP, uma fila (FIFO) é usada em cada EP para armazenar os outros q-1 valores de $\mathbf{v}(k)$. Além disso, como cada EP deve calcular q elementos de $\mathbf{v}(k+1)$, uma fila adicional é necessária para armazenar os elementos de $\mathbf{v}(k+1)$ à medida que são calculados. As duas filas (FIFO) são denominadas como $FIFO_0$ e $FIFO_1$ e trocam de função ao final de cada iteração, ou seja, quando em uma iteração na $FIFO_0$ circulam os valores v(k) e a $FIFO_1$ armazena v(k+1), na próxima iteração na $FIFO_1$ circulará os valores v(k) e a $FIFO_0$ armazenará os valores de v(k+1), e assim sucessivamente.

Também cada EP irá requerer q elementos de $\mathbf{g}$ e q vetores-linha de $\mathbf{F}$. A ordem em que os termos de $\mathbf{F}$ e $\mathbf{v}(k)$ são multiplicados e acumulados é determinada pela forma em que o anel é utilizado. A ordem em que cada EP recebe os elementos de $\mathbf{v}(k)$ é idêntica ao caso de $n=N$. Entretanto, como cada EP deve processar vários $\mathbf{v}(k+1)$, será necessário mais de uma linha de $\mathbf{F}$, o que é denotado pela notação $\mathbf{f}_i$, onde i varia de 1 a q. Assim, a expressão para a reordenação dos elementos de $\mathbf{F}$ para o elemento de processamento p pode ser escrita como

$$d_{i,j} = f_{(p-1)q+i,\,((p-1)q+j-1)\bmod n+1} \ .$$

Como um exemplo, considera-se o caso onde o número de EPs N=3 e o número de equações n=6. Para cada processador tem-se:

EP$_1$

$$\begin{bmatrix} d_{1,1} & d_{1,2} & d_{1,3} & d_{1,4} & d_{1,5} & d_{1,6} \\ d_{2,1} & d_{2,2} & d_{2,3} & d_{2,4} & d_{2,5} & d_{2,6} \end{bmatrix} = \begin{bmatrix} f_{1,1} & f_{1,2} & f_{1,3} & f_{1,4} & f_{1,5} & f_{1,6} \\ f_{2,1} & f_{2,2} & f_{2,3} & f_{2,4} & f_{2,5} & f_{2,6} \end{bmatrix}$$

EP$_2$

$$\begin{bmatrix} d_{1,1} & d_{1,2} & d_{1,3} & d_{1,4} & d_{1,5} & d_{1,6} \\ d_{2,1} & d_{2,2} & d_{2,3} & d_{2,4} & d_{2,5} & d_{2,6} \end{bmatrix} = \begin{bmatrix} f_{3,3} & f_{3,4} & f_{3,5} & f_{3,6} & f_{3,1} & f_{3,2} \\ f_{4,3} & f_{4,4} & f_{4,5} & f_{4,6} & f_{4,1} & f_{4,2} \end{bmatrix}$$

EP$_3$

$$\begin{bmatrix} d_{1,1} & d_{1,2} & d_{1,3} & d_{1,4} & d_{1,5} & d_{1,6} \\ d_{2,1} & d_{2,2} & d_{2,3} & d_{2,4} & d_{2,5} & d_{2,6} \end{bmatrix} = \begin{bmatrix} f_{5,5} & f_{5,6} & f_{5,1} & f_{5,2} & f_{5,3} & f_{5,4} \\ f_{6,5} & f_{6,6} & f_{6,1} & f_{6,2} & f_{6,3} & f_{6,4} \end{bmatrix}$$

A figura 6.7 ilustra essa configuração. Nessa figura, a FIFO$_1$ é usada para armazenar os elementos de $\mathbf{v}(k+1)$ à medida que são calculados, e a FIFO$_0$ é usada para circular os elementos de de $\mathbf{v}(k)$, dos quais os novos elementos de $\mathbf{v}(k+1)$ são calculados. Os valores de FIFO$_0$ são os mesmos no início e no final de uma iteração. No início de uma iteração, a FIFO$_1$ estará vazia. Ao final do primeiro passo de uma iteração o acumulador (denominado como "ACC") é carregado com o valor apropriado de **g**. Por exemplo, no EP$_1$, o acumulador irá ser carregado com o valor g_1. A figura 6.7 não inclui a máquina hospedeira e o controlador global como na figura 6.3 por questão de simplicidade.

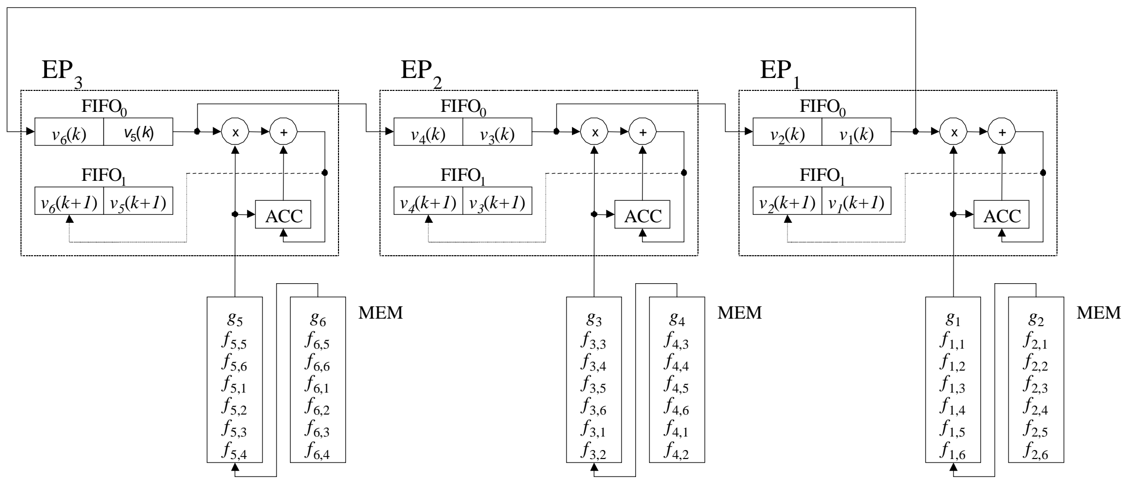


Figura 6.7 Fluxo de Dados para o caso onde n=6, N=3 (q=2).

Após os valores apropriados de g_i terem sido carregados no acumulador de um EP, o elemento $v_i(k)$ no início de $FIFO_0$ é multiplicado por um elemento de $\mathbf{f}_i$ e acumulado em ACC. Este mesmo $v_i(k)$ é então passado para o final da fila $FIFO_0$ do próximo EP. Desta maneira, todos os EPs usam os elementos de $\mathbf{v}(k)$ para calcular um elemento de $\mathbf{v}(k+1)$.

Após um elemento de $\mathbf{v}(k+1)$ ser calculado, por exemplo $v_1(k+1)$ no EP_1, é colocado na $FIFO_1$ deste EP. O processo é então repetido para o próximo elemento de $\mathbf{v}(k+1)$, que no caso do EP_1 deve ser o $v_2(k+1)$. Durante o segundo estágio da iteração, os elemento de **F** e **g** indicados na segunda coluna são utilizados. Os conteúdos da $FIFO_1$ na figura são os valores que devem estar presentes na $FIFO_1$ ao final de uma iteração, quando todos os valores de $\mathbf{v}(k+1)$ tiverem sido calculados.

Novamente, os elementos de $\mathbf{f}_i$ e o valore g_i são apresentados em um único elemento de memória para cada EP na figura 6.7 para ilustrar a sequência em que devem ser utilizados. A figura 6.8 mostra o diagrama (RTL) de um único EP. Um registrador denominado R1 na figura 6.8 é usado para armazenar a saída da FIFO. Como R1 é usado para armazenar uma cópia do valor do início da fila FIFO, ele não está incluso na figura 6.7 por simplicidade.

A operação do arranjo sistólico é descrita nos fluxogramas das figuras 6.9, 6.10 e 6.11. Alguns parâmetros de controle que não precisam fazer parte do EP estão inclusos no fluxograma. Somente as operações mostradas em negrito devem ser realizadas em cada EP. Um único

controlador pode ser utilizado para controlar todos os EPs. O fluxograma inclui os contadores (cnt1, cnt2 e k) e o bit de *toggle* *T* que também fazem parte do controlador.

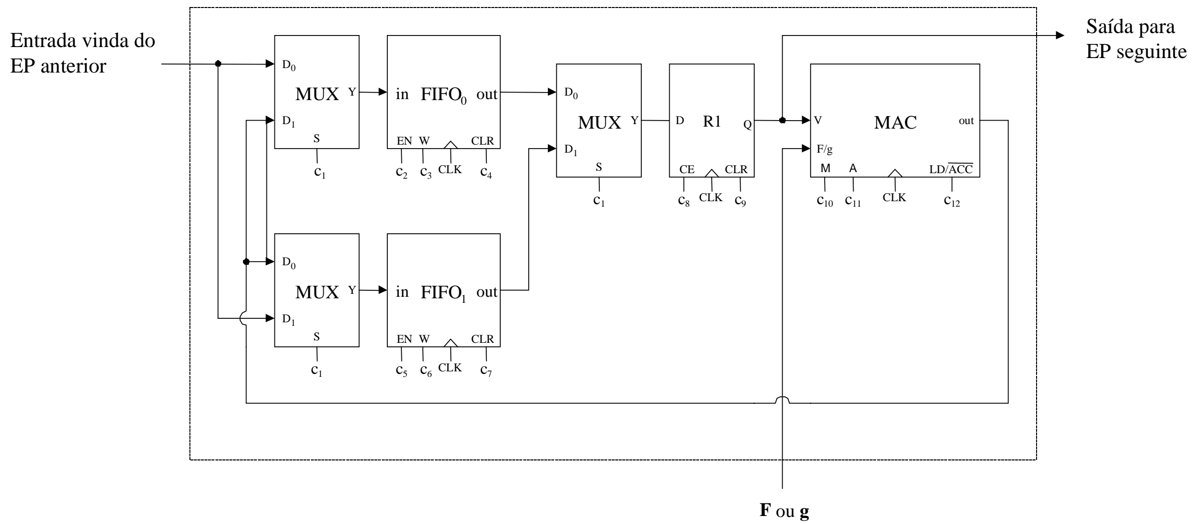


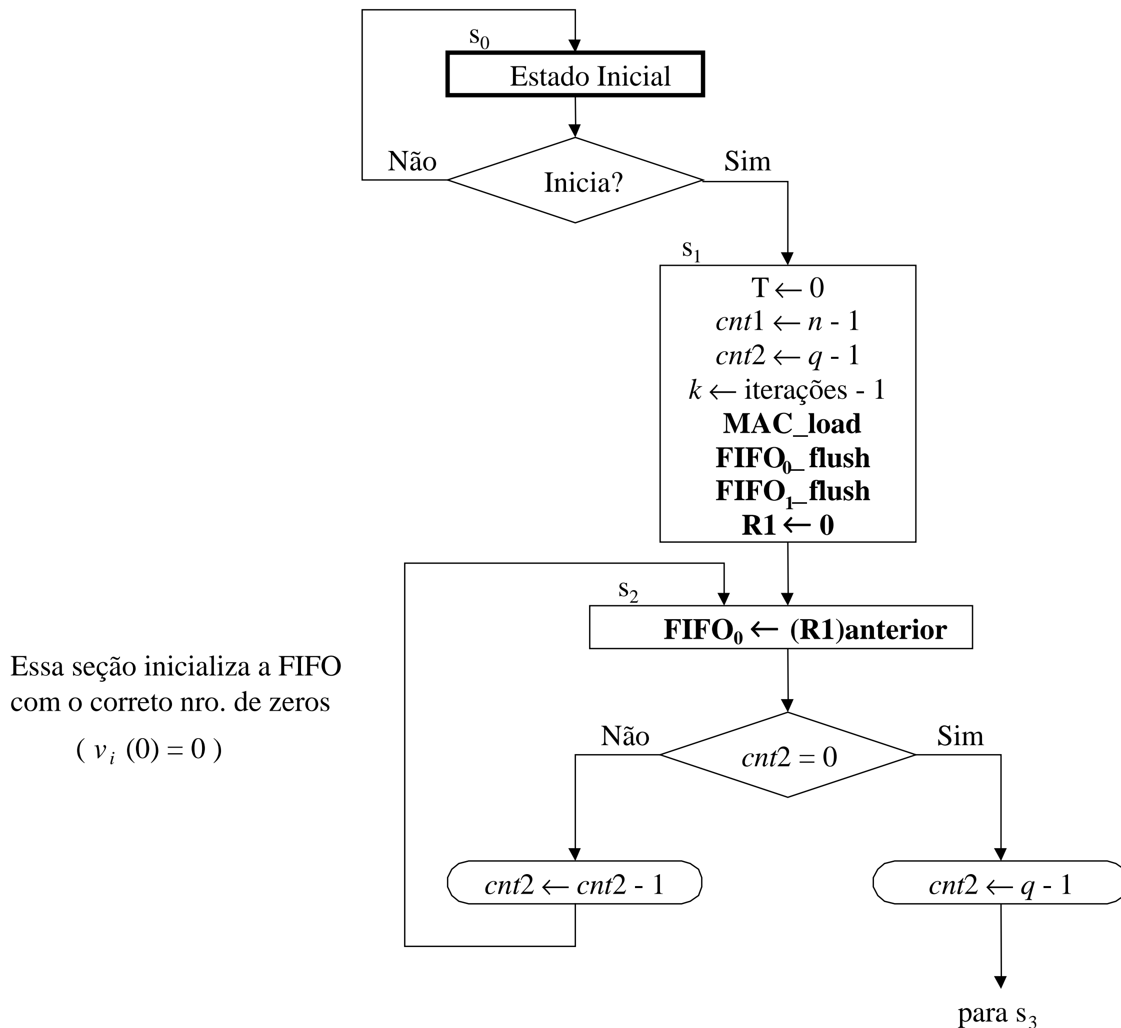
Figura 6.8 Diagrama de Fluxo de Dados (RTL) do EP da Arquitetura para n>N.

Tabela 6.2 Sinais de Controle para os EPs da Arquitetura para n>N.

Sinal de Controle	*Equações Lógicas*
c_1	T
c_2	$s_2 + s_3\overline{T} + s_5(cnt1 \neq 0)\overline{T} + s_5(cnt2 \neq 0)\overline{T} + s_5(cnt1 = 0)T + s_6\overline{T} + s_7\overline{T}$
c_3	$s_2 + s_5 + s_7$
c_4	$s_1 + s_5\overline{T}(cnt1 = 0)(cnt2 = 0)$
c_5	$s_3T + s_5(cnt \neq 0)T + s_5(cnt2 \neq 0)T + s_5(cnt1 = 0)\overline{T} + s_6T + s_7T$
c_6	$s_5 + s_7$
c_7	$s_1 + s_5T(cnt1 = 0)(cnt2 = 0)$
c_8	$s_2 + s_3 + s_6$
c_9	s_1
c_{10}	s_4
c_{11}	$s_1 + s_5$
c_{12}	$s_1 + s_5$

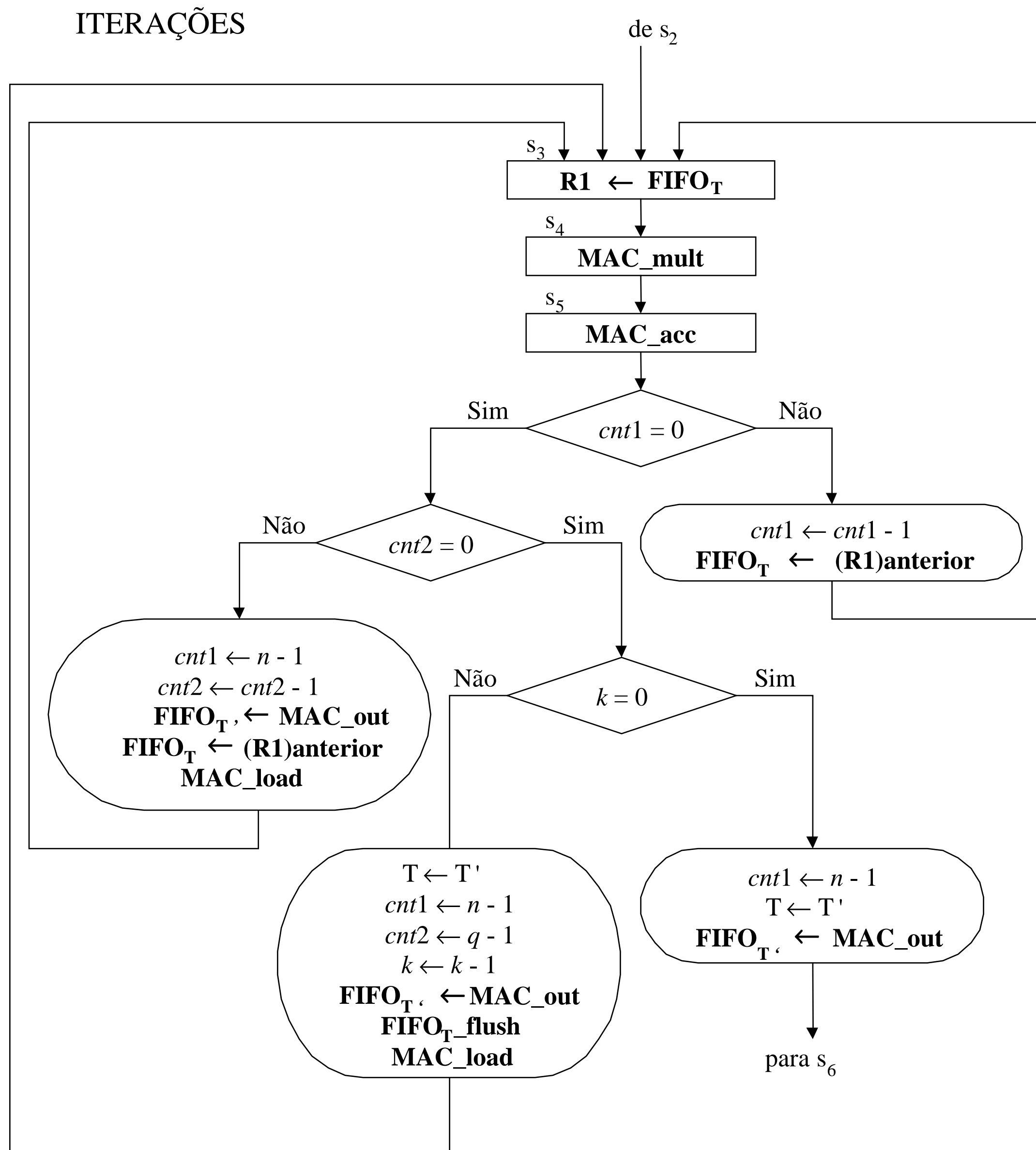
Os sinais de controle para cada EP na figura 6.8 são listados como c_1, c_2, …, c_{12} . Estes sinais são derivados dos estados apresentados no fluxograma como mostra a tabela 6.2. O bit de toggle *T* especifica qual FIFO está sendo usada para armazenar os elementos de **v**(*k*) e qual está sendo usada para armazenar os elementos de **v**(*k*+1).

INICIALIZAÇÃO



MAC_load carrega g_i no registrador ACC do MAC de cada EP.
FIFO_flush esvazia a FIFO

Figura 6.9 Estágio de Inicialização do Fluxograma da Arquitetura para n>N .



MAC_load carrega g_i no registrador ACC do MAC.
FIFO_flush esvazia a FIFO

Figura 6.10 Parte Iterativa do Fluxograma da Arquitetura para $n>N$.

FINALIZAÇÃO

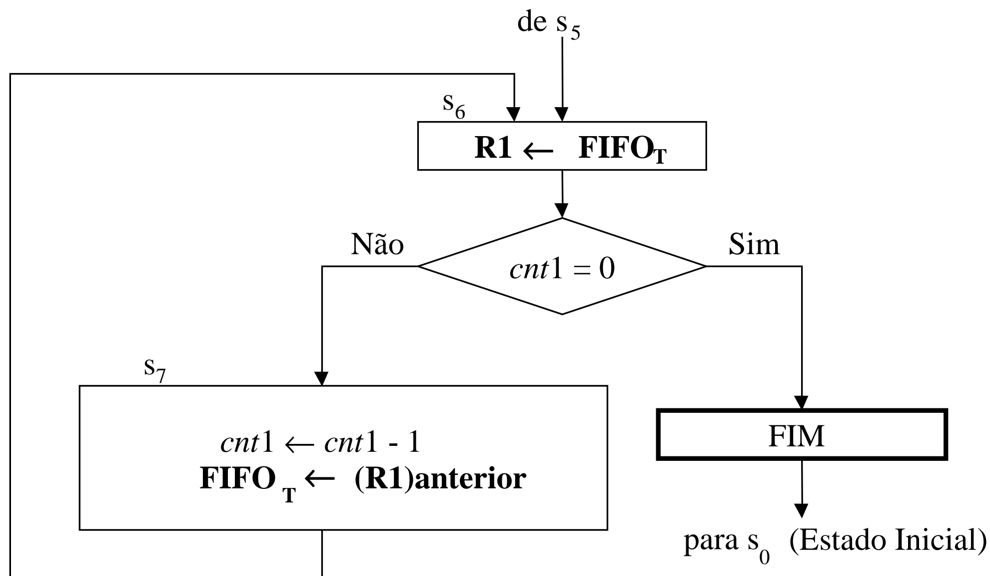


Figura 6.11 Estágio de Finalização do Fluxograma da Arquitetura para n>N .

6.3. Implementação das Arquiteturas

A seguir são descritas as etapas realizadas na implementação das arquiteturas propostas.

6.3.1. Simulação em *Software*

As arquiteturas foram inicialmente simuladas em *software*, escrevendo-se algoritmos seqüenciais na linguagem C coerentes com os fluxogramas apresentados. Além de constatar a eficácia do método, esta etapa foi importante para determinar alguns requisitos de projeto, como, por exemplo, a forma de representação dos dados.

Foi observado que os valores da matriz de **F** e **g**, para o método de Wang e Li, estão na faixa de –1 a 1. Para facilitar a implementação em *hardware* foi decidido utilizar números inteiros com sinal. Para representar os valores reais na forma de números inteiros, estes valores são multiplicados por uma constante *m* para então ser desprezada a parte decimal. Através da simulação em *software* estimou-se que o valor desta constante deveria ser no mínimo da ordem

de 3 x 10^4, para que esta operação não afetasse a convergência do método. Reescrevendo-se a equação 5.2 para acomodar tal transformação tem-se:

$$m[m\mathbf{v}(k+1)] = [m\mathbf{F}][m\mathbf{v}(k)] + m[m\mathbf{g}]$$

ou

$$\hat{\mathbf{v}}(k+1) = \frac{\hat{\mathbf{F}}\hat{\mathbf{v}}(k) + m\hat{\mathbf{g}}}{m},$$

onde $\hat{\mathbf{v}}(k) = m\mathbf{v}(k)$, $\hat{\mathbf{F}} = m\mathbf{F}$ e $\hat{\mathbf{g}} = m\mathbf{g}$. A máquina hospedeira é encarregada de mais esta operação fornecendo à arquitetura os valores de $\hat{\mathbf{F}}$ e $\hat{\mathbf{g}}$. Entretanto, cada EP deve executar mais uma multiplicação de $\hat{\mathbf{g}}$ pela constante *m* e também realizar, ao final de cada iteração, a divisão do valor acumulado (no registrador ACC) pela constante *m*, operações não previstas inicialmente na arquitetura. Por simplicidade escolheu-se para o valor de *m* um valor de potência de 2, pois desta forma essas operações adicionais são executadas com simples deslocamentos de bits, sem acrescentar *hardware* e sem afetar o número de estados do algoritmo.

Nesta implementação, o valor escolhido para *m* foi 32768 (2^{15}) implicando na utilização de 17 bits para a representação dos dados na forma de inteiros com sinal.

6.3.2. Projeto do Elemento de Processamento

Conforme apresentado há duas versões de arquitetura implicando em duas versões de elementos de processamento e alterações nos sinais de controle. O elemento de processamento, de ambas as versões, é basicamente um MAC, tendo como principal função realizar multiplicações e a totalização dos resultados. O EP da segunda versão difere do primeiro pela introdução das duas filas FIFO e mais um multiplexador, acrescentando um pouco de complexidade ao controlador.

6.3.2.1. Projeto do Multiplicador-Acumulador (MAC)

Particularmente, a operação de multiplicação binária representa o "gargalo" do sistema. O desempenho da arquitetura depende da execução de forma eficiente desta operação.

Inicialmente foi projetado um multiplicador *Shift-Add*, mas que não apresentou um desempenho satisfatório, pois requeria *b-1* ciclos de *clock* (*b* = número de bits dos dados de entrada) para realizar uma multiplicação. Realizou-se em seguida uma pesquisa para avaliar alternativas de

algoritmos de aritmética binária para o projeto do multiplicador. O algoritmo escolhido para a implementação foi o de Baugh-Wooley (Hwang, 79), utilizado para multiplicação direta de números binários com sinal e representação complemento de dois. Este algoritmo requer uma quantidade de *Full-Adders* (FA) igual a $(b-1)b+3$, onde b representa o número de bits dos dados de entrada, ou seja, há um custo elevado de *hardware* para se obter desempenho.

A representação complemento de dois foi escolhida pois facilita o estágio seguinte (Acumulador), sem onerar muito o *hardware* em relação a multiplicação de números na representação Sinal-Magnitude que é mais simples. Além disso é a forma de representação mais utilizada em circuitos de aritmética binária.

Embora este algoritmo já apresentasse um desempenho bem superior em relação ao *Shift-Add*, foi introduzido um *pipeline* interno, visando torná-lo mais eficiente. Foram acrescentados no multiplicador 3 conjuntos de registradores intermediários, os quais armazenam resultados parciais e permitem que novos dados sejam inseridos nas entradas antes do término de uma multiplicação. Dessa forma o MAC pode utilizar *clocks* mais rápidos.

O projeto foi desenvolvido em esquemático e simulado funcionalmente com o *software* Active CAD 2.5 da Aldec Inc., o qual faz parte do pacote *Foundation Series* da Xilinx Inc..

6.3.2.2. Projeto da FIFO

Para o projeto deste componente do EP da segunda versão da arquitetura foram utilizadas FIFOs baseadas em memória. Inicialmente foi adotada uma memória do tipo RAM síncrona, mas em seguida optou-se pela RAM do tipo *dual-port* que permite leituras e escritas simultâneas, possibilitando a eliminação de estados e do registrador R1. Em ambas as versões da FIFO utilizou-se memórias de 16x17.

6.3.3. Projeto do Controlador

O controlador é composto de contadores e de uma máquina de estados que gera os sinais de controle para os componentes dos EPs. Consequentemente, há também uma versão de controlador para cada EP. As máquinas de estado foram projetadas em esquemático segundo a técnica One-Hot Bit (Katz , 94) e também em VHDL. Constatou-se as vantagens do VHDL para este tipo de circuito. A implementação de máquinas de estado em VHDL é muito produtiva e eficiente.

6.3.4. Simulação Funcional

Cada bloco e componente projetado foi simulado com o *software Logic Simulator* da Aldec Inc., que faz parte do pacote *Foundation Series* da Xilinx Inc. Para cada versão da arquitetura foi simulado um anel de tamanho reduzido. Para a primeira versão utilizou-se um anel com 3 EPs, um controlador e elementos de memória ROM para cada EP. As memórias foram pré-gravadas com dados correspondentes aos elementos da matriz **F** e **g** devidamente rearranjados de um problema de tamanho 3x3. Os resultados finais da simulação foram similares aos resultados com o programa em C. Para a segunda arquitetura simulou-se um anel com somente 2 EPs resolvendo um sistema 4x4. Os dados do sistema (**F** e **g**) também foram gravados em memórias ROM de forma similar. Também foram obtidos resultados coerente com o programa em C. Apesar de reduzido, as simulações destes anéis consumiram consideráveis recursos computacionais e provavelmente as limitações da ferramenta ou do computador utilizado não permitiriam a simulação de anéis muito maiores. Esta etapa foi fundamental para validar o projeto dos EPs e dos controladores.

6.3.5. Implementação em Circuitos FPGAs

Após projetar e simular cada bloco e componente, os projetos dos elementos de processamento foram implementados no circuito FPGA Xilinx XC4025E-2. Foi utilizada a ferramenta **M1** do pacote *Foundation Series* Versão 1.4 , a qual realizou as etapas de síntese, mapeamento e roteamento dos circuitos, resultando em um arquivo de configuração do dispositivo. Foram inseridas restrições de *timing* para que a ferramenta otimizasse o desempenho, diminuindo os atrasos nos caminhos críticos do circuito. Os resultados de desempenho são discutidos no capítulo seguinte.

A tabela 6.3 apresenta alguns dados do relatório de mapeamento que demonstram a utilização dos recursos (CLBs e *Flip-Flops*) do FPGA no projeto do multiplicador (com e sem *pipeline*) e no projeto das duas versões do EP. Pode-se observar que somente o multiplicador em *array* consome mais de 80% dos recursos utilizados pelo EP. Também observa-se que a introdução do *pipeline* no circuito do multiplicador não altera o número de CLBs utilizadas, somente o número de *flip-flops* internos aos CLBs. Portanto, a capacidade dos CLBs é melhor aproveitada com o uso do *pipeline*.

Tabela 6.3 Recursos do FPGA utilizados no mapeamento .

	CLBs	Flip-Flops
Mult 17	382	0
Mult 17p	382	198
EP-1	422	81
EP-2	453	278

A diferença da quantidade de CLBs para os EPs é devido principalmente às FIFOs introduzidas na segunda versão. A quantidade de CLBs para o caso do EP-2 representa cerca de 44% dos recursos do *chip* XC4025E-2 que possui 1024 CLBs. Portanto, pode-se acomodar até dois EPs neste modelo de FPGA da Xilinx.

6.4. Considerações Finais

No arranjo sistólico apresentado, todos os elementos de processamento são utilizados continuamente, atingindo um alto grau de paralelismo. A arquitetura possui um baixo *overhead* de comunicação e sincronismo. Além disso, utiliza somente um tipo de elemento de processamento, sendo facilmente escalonável para acomodar problemas de tamanhos variáveis.

A utilização de metodologias avançadas, como o uso de simuladores e linguagem de descrição de *hardware*, acelera bastante o ciclo de desenvolvimento, possibilitando a exploração de alternativas de implementação.

A implementação em circuitos FPGAs permite a construção de protótipos rapidamente e obtenção de dados reais de desempenho. O projetista tem um *feedback* imediato, não dependendo de processos de fabricação demorados para ter o circuito projetado funcionando, dentro de certos parâmetros.

7. Análise e Discussão dos Resultados

7.1. Considerações Iniciais

Neste capítulo discute-se a complexidade dos algoritmos das duas versões da arquitetura e apresenta-se os resultados de desempenho obtidos na implementação dos elementos de processamento em circuitos FPGAs da Xilinx. Também são apresentados dados de desempenho de um microcomputador Pentium II 300MHz na execução de um algoritmo sequencial equivalente, com o intuito de prover um referencial e não uma análise comparativa aprofundada, já que esta envolve diversos fatores que vão além do escopo deste trabalho.

7.2. Complexidade do Algoritmo

Analisando-se a complexidade do algoritmo da primeira versão da arquitetura, apresentado na Figura 6.6, observa-se que, para o problema de tamanho n, o tempo total requerido do início do estado S1 até todos os dados terem sidos recuperados do anel, assumindo-se k iterações é

$$T_T = T_w(1 + 2nk + n\text{-}1) = T_w\, n(2k + 1) \tag{7.1}$$

onde T_w é o período do *clock*, e considerando-se também que a operação de multiplicação seja executada em um único ciclo de *clock*. Inicialmente foi implementado um multiplicador *Shift-Add* que requer (b-1) ciclos (b = número de bits dos dados de entrada) para completar esta operação. Em seguida foi implementado um Multiplicador em *Array* que executa a multiplicação em apenas um ciclo de *clock*, mas seu período mínimo torna-se longo em função do número de bits dos dados de entrada. Para reduzir o período do *clock* utilizou-se o recurso de introduzir um *pipeline* interno com registradores intermediários. No caso particular dessa implementação utilizou-se 3 conjuntos de registradores, e acoplou-se a operação de Acumulação, resultando em um *pipeline* de 5 estágios (p = 5). Portanto, com essas modificações, o tempo de processamento pode ser expresso como:

$$T_T = T_w(1 + (n + p)k + n\text{-}1) = T_w\,(n(k{+}1) + pk) \qquad (7.2)$$

A complexidade para a esta versão da arquitetura é então de ordem O(n).

A segunda arquitetura ($n>N$) requer mais tempo para a iteração pois há um fator adicional de q passos para calcular todos os valores de $\mathbf{v}(k+1)$, onde $q = n/N$, assumindo-se q um múltiplo de N. Adicionando-se os q passos necessários para iniciar as FIFOs com $\mathbf{v}(0)$ e os n-1 passos requeridos para recuperar os resultados do anel, o tempo total seria

$$T_T = T_w(1 + q + 3nqk + n\text{-}1) = T_w\,(q + n(3qk + 1)) \qquad (7.3)$$

Como q é proporcional a n, o termo de mais alta ordem é

$$3nqk = \frac{3kn^2}{N},$$

que corresponde a uma complexidade de ordem $O(n^2)$. Entretanto, como o termo de mais alta ordem é inversamente proporcional ao número de EPs, a arquitetura alcança um *speedup* linear à medida que o número de EPs é aumentado. Novamente, levando em conta o *pipeline* introduzido nessa implementação, o tempo total é expresso como

$$T_T = T_w(1 + q + (n{+}p)qk + n\text{-}1) = T_w\,(q + n(qk + 1) + pqk) \qquad (7.4)$$

7.3. Resultados da Implementação em circuitos FPGAs

Ambas as versões dos elementos de processamento foram implementadas usando-se circuitos reconfiguráveis FPGAs da Xilinx . O dispositivo utilizado foi o XC4025E-2 HQ240, que é o maior disponível no LaSD-ICMC. A análise temporal, após as fases de mapeamento, posicionamento e roteamento (*post-layout timing*), indicaram um atraso máximo de 37ns, correspondendo a um *clock* de 27MHz. Utilizando-se dispositivos mais rápidos da família XC4000XL, pode-se atingir taxas de *clock* acima de 50MHz. A tabela 7.1 apresenta os dados de tempo de processamento de 10.000 iterações, considerando-se o *clock* de 27 MHz. Cada linha da tabela representa uma configuração do anel (N) e cada coluna apresenta um determinado tamanho de problema (n). Para a tabulação desses dados foi considerada somente a parte iterativa do algoritmo, correspondente ao termo $(n+p)qk$ da equação (7.4), com p=5 e k=10000. Os dados da diagonal em cinza representam os valores para o caso de $n = N$, ou seja $q = 1$.

Tabela 7.1 Tempo de Processamento (em segundos, clock= 27 MHz) de 10000 iterações, para várias configurações do anel (N) e tamanhos do sistema (n).

n / N	16	32	64	128	256	512	1024	2048
4	0,031	0,110	0,409	1,576	6,187	24,510	97,564	389,310
8	0,016	0,055	0,204	0,788	3,093	12,255	48,782	194,655
16	**0,008**	0,027	0,102	0,394	1,547	6,127	24,391	97,327
32		**0,014**	0,051	0,197	0,773	3,064	12,196	48,664
64			**0,026**	0,099	0,387	1,532	6,098	24,332
128				**0,049**	0,193	0,766	3,049	12,166
256					**0,097**	0,383	1,524	6,083

Pelo gráfico 7.1 observa-se que para um dado tamanho de problema (*n*) o tempo de processamento cai pela metade à medida que dobra-se o número de processadores no anel, indicando uma eficiência de 100%. Isto se deve ao baixo *overhead* de comunicação e o uso contínuo de todos os elementos de processamento paralelamente nessa arquitetura. Pelo gráfico 7.2 nota-se que, para um dado número de processadores no anel, o tempo de processamento quadruplica à medida que o tamanho do problema dobra.

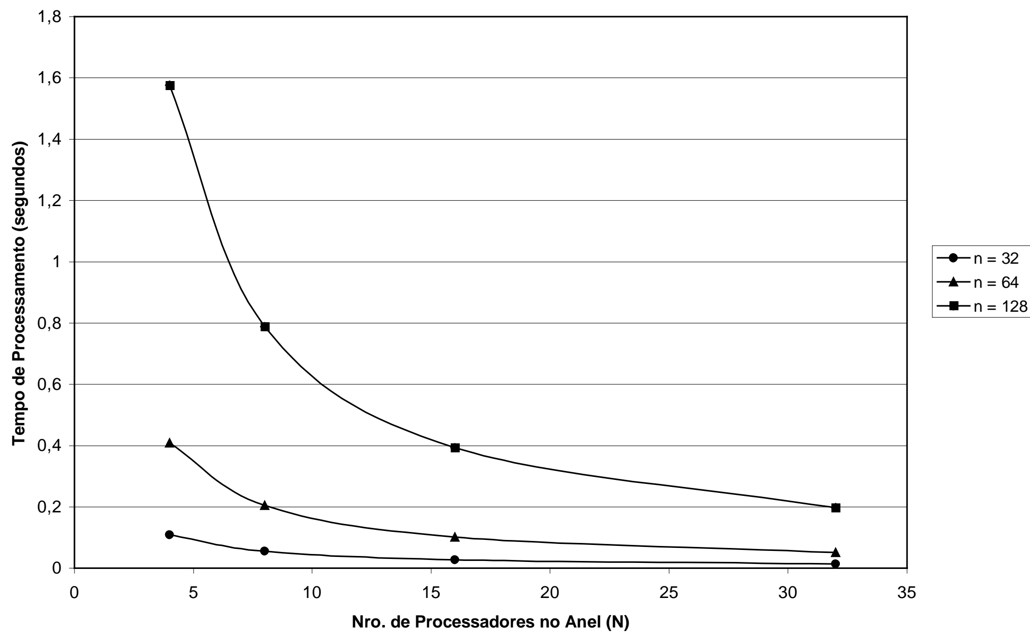


Figura 7.1 Tempo de Processamento x Nro. de EPs no anel (N).

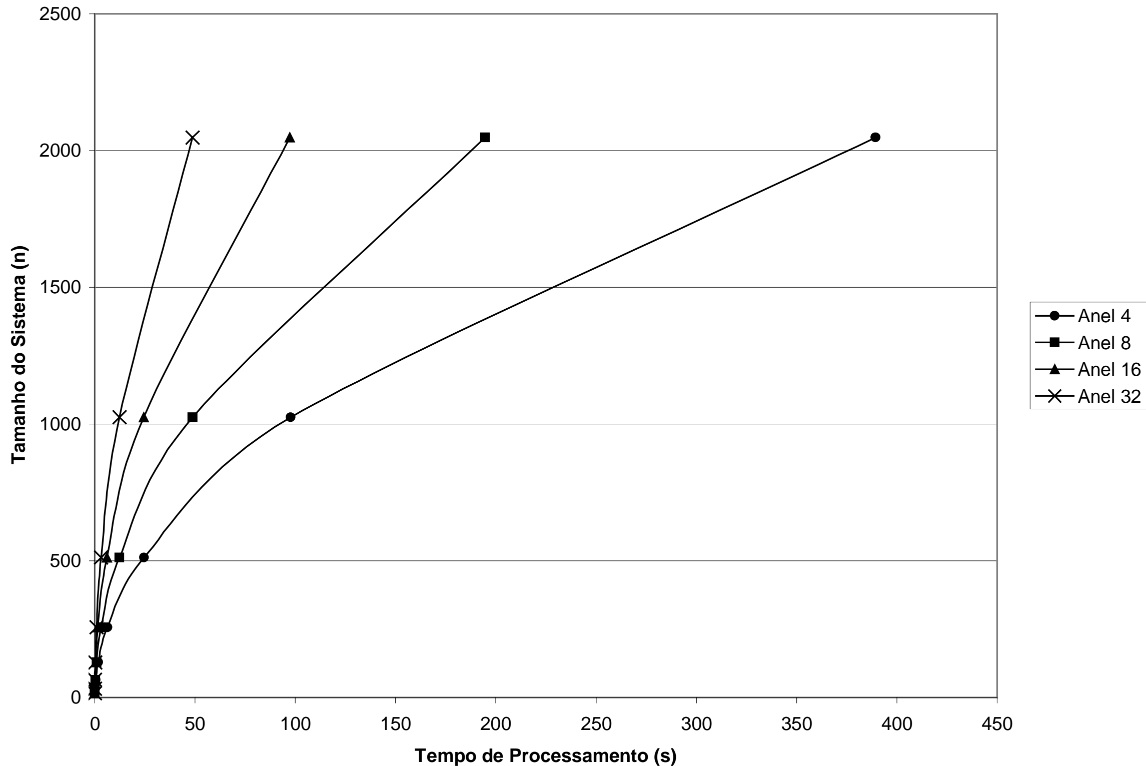


Figura 7.2 Tamanho do Sistema (n) x Tempo de Processamento

Há *benchmarks*, baseados em solução de sistemas lineares, como o LINPACK, para avaliação de desempenho de máquinas computacionais (Dongarra et al., 91, Hwang, 84). Entretanto, este tipo de *benchmark* não se aplica para um hardware dedicado. Para se ter um referencial de comparação, embora rústico, escreveu-se um algoritmo seqüencial que implementa a operação de multiplicação matriz-vetor, que corresponde à operação básica realizada pela arquitetura sistólica. Este algoritmo (figura 7.3), de complexidade de ordem $O(n^2)$, constitui-se de dois laços para a multiplicação de uma matriz A (*n* x *n*) por um vetor b (*n* x 1) resultando um vetor c (*n* x 1) , com valores inteiros.

```
for (i = 0; i<n ; i++) {
         for (j = 0; j<n ; j++) {
                c(j) = c(j) + A(i)(j) * b(j);
         }
}
```

Figura 7.3 Algoritmo para Multiplicação Matriz-Vetor

Este algoritmo foi compilado na ferramenta C++ Builder 3.0 da Borland e executado em um processador Pentium II 300MHz com 64Mbytes de memória RAM e 512KBytes de cache. Os tempos de execução de 10000 iterações para diferentes tamanhos de matriz são apresentados na tabela 7.2

Tabela 7.2 Tempos de Processamento de 10000 iterações do algoritmo de Multiplicação Matriz-Vetor para vários tamanhos de matriz (n), em um microcomputador Pentium II .

n	Tempo(s)
16	0,15
32	0,58
64	2,41
128	10,61
256	42,17
512	232,91
1024	939,99
2048	3695,21

Comparando-se os dados apresentados nas tabelas 7.1 e 7.2, observa-se que o *hardware* projetado tem um desempenho bem superior em relação a um processador seqüencial na execução da operação *Gaxpy*. O anel com apenas quatro elementos de processamento possui um tempo de processamento cerca de 5 a 10 vezes menor dependendo do tamanho da matriz. Aumentando-se o número de EPs no anel este ganho pode chegar acima de 600 vezes para o caso de 256 EPs e um sistema de 2048 equações.

7.4. Considerações Finais

O projeto de *hardware* dedicado para algoritmos, tais como os métodos iterativos de solução de sistemas lineares, é uma abordagem que mostrou-se viável e eficiente.

Confirmou-se a expectativa de se obter um desempenho satisfatório que respalde e justifique a construção do *hardware* na tecnologia FPGA. Obviamente, com a evolução desses dispositivos pode-se obter, em breve, resultados ainda melhores.

Para a construção da arquitetura completa com circuitos FPGAs sugere-se um anel de 16 elementos de processamento. Com essa configuração é possível obter resultados de desempenho até 38 vezes mais rápidos que o Pentium II 300MHz, dependendo do tamanho do problema. Se esta arquitetura fosse implementada no dispositivo XC4025E seriam necessários 8 *chips*, pois este dispositivo acomoda até dois EPs. Esta seria uma configuração razoável em termos de

custo/benefício e em termos de realização prática, já que são previstos problemas de atrasos de propagação dos sinais de controle à medida que acrescenta-se EPs ao anel. Para o armazenamento dos dados da matriz **F** e vetor **g**, recomenda-se o uso de *chips* de memória externos. Apesar de se poder implementar a memória dentro do próprio circuito FPGA a demanda por esse recurso cresce de forma quadrática com o aumento do tamanho do problema.

8. Conclusões

8.1. Considerações Iniciais

Neste trabalho de pesquisa adotou-se uma abordagem pouco utilizada para solução de problemas computacionais: a construção de um hardware dedicado. Foi desenvolvido o projeto de uma Arquitetura Sistólica para Solução de Sistemas Lineares, procurando-se explorar novas metodologias de projeto de hardware e adquirir o *know-how* da tecnologia de dispositivos reconfiguráveis FPGAs.

Tratando-se da primeira pesquisa no ICMC nessa área, boa parte do trabalho constituiu-se do aprendizado de ferramentas de projeto e linguagens de descrição de hardware e da realização de um amplo levantamento bibliográfico sobre essa tecnologia ainda recente. Este trabalho certamente será útil para futuras pesquisas nesse campo de conhecimento.

A implementação de hardware dedicado sem dúvida alguma é mais eficiente que a execução de algoritmos em microprocessadores de propósito geral. Entretanto, essa abordagem tem sido pouco utilizada, pois os custos de projeto e o tempo de desenvolvimento certamente são maiores que a programação em software.

Com a evolução das metodologias de projeto de hardware o ciclo de projeto é realmente reduzido de forma significativa. As poderosas ferramentas de síntese e simulação possibilitam explorar alternativas de implementação rapidamente. Entretanto, a síntese lógica ainda é uma área que necessita ser aperfeiçoada para suportar descrições em alto nível de sistemas mais complexos.

Aliado a isso, as novas tecnologias de dispositivos reconfiguráveis possibilitam a implementação do projeto sem necessidade de processos de fabricação caros e demorados. Essa tecnologia pode ser empregada numa gama enorme de aplicações desde a integração de circuitos digitais LSI até o desenvolvimento completo de sistemas computacionais. Dentro dessa nova perspectiva, a construção de hardware dedicado é uma opção a ser considerada com maior atenção, sendo uma alternativa viável para um nicho de aplicações específicas, que requerem desempenho e não necessitam de toda a funcionalidade de uma máquina computacional de finalidade geral.

8.2. Contribuições deste Trabalho

Foram descritas duas versões de Arquitetura Sistólica para solução de Sistemas Lineares, explicando-se o seu funcionamento através de diagramas RTL e fluxogramas e discutindo-se detalhes de implementação. A primeira versão da arquitetura proposta é para a solução de sistemas de equações lineares onde o número de equações é menor ou igual ao número de EPs. Cada elemento de processamento consiste de um MAC (multiplicador acumulador), um registrador e um multiplexador. A segunda versão da arquitetura acomoda problemas com número de equações superior ao número de EPs. Cada elemento de processamento consiste de um MAC, duas filas FIFO e três MUXs. Ambas as versões foram simuladas com software e ferramentas de simulação de hardware e os elementos de processamento foram implementados em circuitos FPGAs.

A Arquitetura foi implementada segundo a proposta do trabalho de (Hillesland, 97), que , embora enfoque o método de solução de Wang e Li, pode utilizar outros métodos iterativos de solução de sistemas lineares baseados na operação Gaxpy. O arranjo sistólico apresentado também pode ser aplicado na solução de outros problemas baseados no trabalho de Wang (1993).

O projeto foi desenvolvido com ferramentas modernas utilizando descrições em esquemático e VHDL. Os blocos e componentes projetados poderão ser reutilizados, pois podem ser encorporados como bibliotecas em outros projetos. Aliás, essa é uma das grandes vantagens dessa metodologia: a reutilização de projetos e a portabilidade do código VHDL.

Os dados de desempenho da implementação do elemento de processamento da arquitetura em circuitos FPGAs da Xilinx foram apresentados para que possam ser confrontados com outras

alternativas de solução. A arquitetura apresentou um desempenho satisfatório, indicando que o projeto de *hardware* dedicado para este tipo de algoritmo é uma abordagem viável e eficiente.

8.3. Sugestões para Desenvolvimento Futuro

No projeto da arquitetura ainda remanescem as questões relativas à comunicação com a máquina hospedeira e o armazenamento eficaz dos elementos da matriz **F** e **g** para a utilização pelo arranjo sistólico. Mais especificamente sobre este projeto poderiam ser feitas algumas melhorias como: codificação completa em VHDL; projeto do multiplicador e acumulador com ponto-flutuante; projeto de outros blocos funcionais para as operações mais comuns utilizadas nos algoritmos de Álgebra Linear e outras áreas.

Há também algumas possibilidades de expandir a arquitetura para resolver outros problemas similares. A arquitetura da rede neural recorrente (RNN) analógica proposta por Wang e Li (1993) é similar a outras RNNs propostas para resolver problemas mais sofisticados. Dessa maneira, a implementação digital apresentada aqui poderia, talvez, ser expandida de tal maneira a resolver estes outros problemas. Aliás, a área de Redes Neurais Artificiais oferece uma gama de algoritmos que podem ser implementados em arquiteturas sistólicas, ou outras classes de arquitetura, desenvolvendo-se hardware dedicado para *Neurocomputers.*

Referências Bibliográficas