Instituto de Computação
Universidade Estadual de Campinas

Uma Implementação em FPGA de um Processador de Vizinhança para Aplicação em Imagens Digitais

Alexandro Magno dos Santos Adário[1]

FEVEREIRO DE 1997

Banca Examinadora:

- Mario Lúcio Côrtes, PhD (Orientador) [2]
- Sérgio Bampi, PhD [3]
- Paulo Lício de Geus, PhD [2]
- Cândido Ferreira Xavier de Mendonça Neto, PhD (Suplente) [2]

Dissertação apresentada ao Instituto de Computação da UNICAMP como requisito parcial para obtenção do título de Mestre em Ciência da Computação

1. O autor é Bacharel em Tecnologia em Processamento de Dados, formado pelo Centro de Ensino Superior de Juiz de Fora (MG)
2. Professor do Instituto de Computação - UNICAMP
3. Professor do Instituto de Informática - UFRGS

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Ad19i

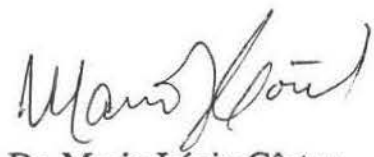
Adário, Alexandro Magno dos Santos
Uma implementação em FPGA de um processador de vizinhança para aplicação em imagens digitais / Alexandro Magno dos Santos Adário. -- Campinas, [SP : s.n.], 1997.

Orientador : Mario Lúcio Côrtes
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Processamento de imagens. 3. Circuitos integrados digitais. 4. Hardware - Linguagens descritivas. I. Côrtes, Mario Lúcio. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida pelo Sr. Alexandro Magno dos Santos Adário e aprovada pela Comissão Julgadora.

Campinas, 05 de março de 1997.



Prof. Dr. Mario Lúcio Côrtes

Dissertação apresentada ao Instituto de Computação - UNICAMP, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

Tese de Mestrado defendida e aprovada em 28 de fevereiro de 1997 pela Banca Examinadora composta pelos Professores Doutores

Profº. Drº. Sérgio Bampi

Profº. Drº. Paulo Lício de Geus

Profº. Drº. Mario Lúcio Côrtes

AGRADECIMENTOS

Acima de tudo, à Deus, pela inspiração, força, coragem e persistência na busca de meus objetivos.

À minha mãe, Dinorá, e meu pai, Nadyr, por seu amor e sua dedicação e esforço incessantes, que me permitiram conquistar o que sou hoje.

Ao meu irmão, João Vicente, e aos meus amigos, em especial a Aline, o Marcus Paulo e a Marianne, que sempre me incentivaram, acreditando em meu potencial.

À Profa. Heloisa V. da Rocha, pelo valioso aconselhamento na minha chegada e por ajudar-me reconhecer a importância de lutar por um ideal.

Ao Prof. Mário L. Cortes, meu orientador, e ao Prof. Neucimar J. Leite, meu co-orientador, pela confiança depositada e grande apoio durante todo o desenvolvimento deste trabalho.

Ao Prof. Paulo C. Centoducatte, por suas idéias, auxílio e empenho na instalação e configuração do ambiente de desenvolvimento deste trabalho, sem o qual não seria possível.

Ao Eng. Carlos G. Kruger, do CPqD-Telebrás, e suas valiosas informações sobre a linguagem VHDL e utilização do ambiente Mentor Graphics.

Aos valiosos e incontáves amigos do Instituto de Computação e do IMECC, por sua solidariedade, companheirismo e carisma, tanto nas horas de trabalho quanto nos momentos de descontração. Não poderia aqui citar todos, mas à alguns, um agradecimento especial: Roseli, Vera, Delano, Alessandra, Cereja, Vanderli, Petty, Rosemeire, Cristiane, Edwin, Fernando e Rui.

À Mentor Graphics e à Altera pela doação dos softwares e equipamento que foram de importância fundamental para este trabalho.

À FAPESP e ao CNPq pelo apoio financeiro.

Dedico este trabalho à minha Mãe
e ao meu Pai, pela grande força
e apoio em todos os momentos,
e ao meu Irmão, artista do traço e das
cores e companheiro de todas as horas.

GENTE QUE PODE E GENTE QUE FAZ

Alexandro Magno dos Santos Adário

Gente que pode
e gente que faz
possuem algumas diferenças.

Gente que pode
se apaixona.
Gente que faz
ama.

Gente que pode
dorme.
Gente que faz
sonha.

Gente que pode
compra, vende, troca.
Gente que faz
se doa.

Gente que pode
morre.
Gente que faz
deixa saudades.

Gente que pode
sorri.
Gente que faz
chora pra fazer sorrir.

Gente que pode
escreve.
Gente que faz
vive.

RESUMO

Este trabalho propõe uma metodologia de projeto de circuitos digitais envolvendo o uso do modelamento comportamental e síntese de alto nível visando o mapeamento tecnológico em componentes reprogramáveis do tipo FPGA. Apresenta uma arquitetura de processador de vizinhança aplicada a imagens digitais e os resultados de sua simulação e da implementação utilizando a metodologia apresentada. Os objetivos principais do trabalho são a validação da metodologia, fazendo um estudo das limitações das ferramentas envolvidas no ciclo de projeto e o impacto na concepção e implementação dos modelos. Também são apresentadas novas contribuições ao modelo da arquitetura proposta.

ABSTRACT

This work proposes a digital circuit design methodology using behavioral modelling and high-level synthesis for technological mapping in FPGA devices. Also, this work introduces an archictecture for neighboorhood processors for digital image applications and the results of its simulation and implementation using the proposed methodology. The main goals of the work are validation of the methodology, including a study on the limitations of the tools used in the design cycle and its impact on model design and implementation. New enhancements to the proposed architecture are also presented.

SUMÁRIO

AGRADECIMENTOS I

RESUMO IV

ABSTRACT V

SUMÁRIO VI

LISTA DE FIGURAS IX

LISTA DE TABELAS XI

LISTA DE SIGLAS XII

1. INTRODUÇÃO 1

1.1. Objetivo do Trabalho 2

1.2. Organização do Trabalho 2

2. ARQUITETURAS PARA PROCESSAMENTO DIGITAL DE IMAGENS 4

2.1. Tipos de Paralelismo de Arquiteturas para PDI 4

2.1.1. Grau de Paralelismo 5

2.1.2. Influência das dimensões de paralelismo 6

2.2. Exemplos de Processadores Matriciais 6

2.2.1. CLIP - *Cellular Logic Image Processor* 7

2.2.2. MPP - *Massively Parallel Processor* 9

2.2.3. DAP - *Distributed Array Processor* 10

2.3. O processador de vizinhança NP9 11

2.3.1. Os Processadores Elementares do NP9 11

2.3.2. O Grafo de Fluxo de Dados 12

2.3.3. Potencialidades 14

2.4. Conclusão 14

3. METODOLOGIA DE PROJETO 15

3.1. Objetivos do Projeto 15

3.2. Fluxo do Projeto ... 16
3.2.1. Descrição das Etapas de Projeto ... 16
3.3. Técnicas de Implementação ... 18
3.4. Ferramentas de Projeto ... 19
3.5. Conclusão ... 20

4. MODELAMENTO EM VHDL E SÍNTESE DE ALTO NÍVEL ... 21
4.1. Classificação das HDLs ... 21
4.2. A linguagem VHDL ... 22
4.2.1. Modelamento comportamental ... 23
4.2.2. Modelamento estrutural ... 23
4.3. Síntese de alto nível ... 24
4.4. Modelamento direcionado à síntese ... 26
4.4.1. Lógica Combinacional e Seqüencial ... 27
4.4.2. Sinais e Variáveis ... 28
4.4.3. Tipos de dados ... 28
4.4.4. Hierarquia de modelos ... 29
4.4.5. Bibliotecas de componentes ... 29
4.5. Conclusão ... 29

5. COMPONENTES FPGA ... 30
5.1. Estrutura Básica ... 30
5.2. Tipos e exemplos de FPGAs ... 31
5.3. Reconfigurabilidade dos Componentes FPGA ... 33
5.4. Mapeamento Tecnológico ... 34
5.4.1. Síntese e Mapeamento ... 35
5.4.2. Recursos Limitados ... 35
5.5. Conclusão ... 36

6. ESPECIFICAÇÕES DO NP9 ... 37
6.1. Considerações Preliminares ... 37
6.1.1. Análise de Escalabilidade ... 37
6.1.2. Alternativas de Projeto ... 38
6.2. Estrutura Geral do NP9 ... 39
6.2.1. Funcionamento do NP9 ... 40

6.3. Os processadores elementares 41
6.3.1. O Multiplicador 41
6.3.2. UCA (Unidade de Comparação e Adição) 42
6.3.3. Programação dos Processadores Elementares 42
6.4. Organização de um Sistema de Processamento de Imagem 43
6.5. Modelamento em VHDL do NP9 44
6.6. Conclusão 45

7. RESULTADOS EXPERIMENTAIS 46
7.1. Estudo de Caso 1: Síntese de PEs do NP9 46
7.1.1. Resultados obtidos 47
7.1.2. Análise dos Resultados 48
7.2. Estudo de Caso 2: Síntese e Mapeamento do NP9 49
7.2.1. Ambiente de Trabalho 49
7.2.2. Procedimentos 49
7.2.3. Resultados Obtidos 50
7.2.4. Alternativa para Melhoria de Desempenho 53
7.2.5. Conclusão 55

8. CONCLUSÃO E TRABALHOS FUTUROS 56

APÊNDICE A - MODELOS VHDL UTILIZADOS 58

APÊNDICE B - EXEMPLOS DE ALGORITMOS DE PDI IMPLEMENTADOS NO NP9 ... 64

REFERÊNCIAS BIBLIOGRÁFICAS 69

LISTA DE FIGURAS

1 Evolução do projeto do NP9 2
2 Arquitetura de um processador matricial. 6
3 Organizações hexagonal (a) e quadrada (b) do CLIP4 8
4 Esquema do processador elementar do CLIP4 8
5 Estrutura do processador elementar do MPP 9
6 Diagrama do processador elementar do DAP 10
7 Esquema de varredura da imagem 11
8 Representação de uma vizinhança 3x3 11
9 Modelo dos processadores elementares do NP9 12
10 Quatro exemplos de configuração do processador elementar do NP9. 12
11 Algoritmo de transposição ímpar-par 13
12 Grafo de Fluxo de Dados do processador 13
13 Diagrama de fluxo de projeto de circuitos integrados 16
14 Cronologia de desenvolvimento de algumas HDLs. 21
15 Exemplo de modelo comportamental 23
16 Exemplo de modelo estrutural 24
17 Representação do Projeto de Circuitos (Diagrama-Y). 25
18 Modelo VHDL de um decodificador 2x4 27
19 Modelo VHDL de um flip-flop 28
20 Exemplos de atribuição a sinal (a) e a variável (b) 28
21 Estrutura de um FPGA. 31
22 Alternativas de Implementação da Arquitetura do Processador de Vizinhança 38
23 Comparação do grau de paralelismo das alternativas de implementação 39
24 Estrutura do NP9 40
25 Estrutura de operação do processador elementar do NP9 41
26 Interface (a) e Estrutura Interna do Multiplicador (b) 41
27 Unidade de Comparação e Adição. 42

28 Organização de um Sistema de Processamento de Imagem 43
29 Esquema de funcionamento do gerador de vizinhança 43
30 Descrição VHDL da entidade CELULA_HL 44
31 Descrição VHDL da entidade PIPELINE_HL 44
32 Detalhe da descrição VHDL da arquitetura do pipeline 45
33 Descrição VHDL da entidade NP9_HL 45
34 Esquema simplificado do modelo 3x3 51
35 Esquema particionado do NP9 53
36 Gráfico da Relação Quadros/s x Pixels Processados 54
37 Grafo do detector morfológico de contorno 65
38 Grafo de fluxo de dados da transposição ímpar-par 65
39 Filtro da mediana separável 66
40 Exemplo da aplicação do detector morfológico de contorno (b), da dilatação (c) e da erosão (d) sobre uma imagem de tamanho 512x512 (a)67
41 Exemplo de aplicação dos filtros de mediana (b) e mediana separável (c) sobre uma imagem com ruído aleatório68

LISTA DE TABELAS

1 Tabela-Verdade de funções Booleanas implementadas no NP9 12
2 Classificação das tecnologias de implementação..... 19
3 Características dos Componentes da Família FLEX10K..... 32
4 Características dos Componentes da Família FLEX8000 32
5 Resumo das Classes de Programabilidade..... 34
6 Funções realizáveis pelos processadores elementares do NP9 42
7 Resultados da Síntese com Otimização por Área 47
8 Resultados da Síntese com Otimização por Desempenho..... 48
9 Resultados do mapeamento do modelo 3x3..... 51
10 Resultado do mapeamento das instâncias crescentes do modelo..... 52
11 Relação Quadro/s x Tamanho da Imagem para freqüência de 12,34 MHz 52
12 Relação Quadro/s x Tamanho da Imagem para freqüência de 17,30 MHz 54

LISTA DE SIGLAS

BES	Bloco de Entrada e Saída
BI	Bloco de Interconexão
BLE	Bloco Lógico Elementar
CLIP	Cellular Logic Image Processor
CPLD	Complex Programmable Logic Device
DAP	Distributed Array Processor
DIS	Dinamyc Instruction Set
EAB	Embedded Array Block
EDA	Eletronic Design Automation
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HLS	High Level Synthesis
ICL	International Computers Ltd
INP20	Image Neighborhood Processor 20
LAB	Logic Array Block
LE	Logic Element
MCU	Master Control Unit
MPP	Massively Parallel Processor
NP9	Neighbourhood Processor 9
PDI	Processamento Digital de Imagens
PE	Processador Elementar
PIMM1	Processeur Intégré de Morphologie Mathématique 1
PPR	Partitioning, Placement and Routing
SIMD	Single Instruction Multiple Data
VHDL	VHSIC HDL
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale of Integration

RTL	Register Transfer Level
RTR	Run-Time Reconfiguration
UCA	Unidade de Comparação e Adição

1. INTRODUÇÃO

Atualmente, conceber um circuito digital significa também consegui-lo no menor tempo possível. Para tanto, é preciso trabalhar dentro de uma metodologia que busque a simplificação e a automatização do processo de projeto, incluindo ainda aumento na produtividade e garantia da qualidade do produto final. Isso é mais importante ainda quando o projeto envolve circuitos de alta complexidade, com um grande número de estruturas funcionais.

Concentrar esforços na funcionalidade e na aplicação do circuito é uma alternativa a ser considerada. O projetista deve colocar em primeiro plano a aplicação proposta e não ocupar a maior parte do tempo em estudos de tecnologia de implementação. Especificação em alto nível (modelamento comportamental) e prototipação rápida do sistema são técnicas que preenchem esses requisitos.

O modelamento comportamental confere maior reusabilidade ao projeto e flexibilidade de mapeamento do circuito final. Uma vez que a preocupação principal volta-se à solução do problema, e não ao modo como essa será implementada, reduz-se consideravelmente o tempo de projeto. A prototipação possibilita validar mais cedo, em ambiente de hardware, a aplicação desenvolvida.

Para o modelamento comportamental, é importante o uso de uma linguagem de descrição de hardware, HDL (*Hardware Description Language*), aliada à síntese em alto nível desses modelos. Direcionar essa síntese para uma técnica de implementação que permita obter um retorno de resultados num espaço de tempo bastante curto, como os componentes FPGA, é também fator imprescindível. Entre outros, dois fatores básicos tornam a técnica de FPGAs um atraente objeto de estudo:

a) é uma técnica bastante moderna e necessita ser explorada mais criteriosamente a fim de apontar-se as suas potencialidades, vantagens, restrições e limitações.

b) custo de produção de poucas unidades muito reduzido em comparação ao de outras técnicas, como o projeto *full-custom* em VLSI, que, sobretudo, demanda grande tempo em desenvolvimento e implementação.

A fim de verificar a validade do emprego do modelamento comportamental e dos componentes FPGA, este trabalho propõe o estudo de uma metodologia baseada nessas duas técnicas, tendo como base para a avaliação a implementação de um processador aplicado a imagens digitais. O Processamento Digital de Imagens (PDI) caracteriza-se por possuir uma elevada carga computacional, envol-

vendo a manipulação de um grande volume de dados, com extensa repetição de operações elementares e ser, muitas vezes, limitado a restrições temporais, exigindo execução em tempo real.

Portanto, aplicações de PDI necessitam, em geral, de arquiteturas altamente eficientes, capazes de realizar um elevado número de operações em um curto espaço de tempo. O NP9, baseado na arquitetura proposta por Leite [40], tem esse objetivo e demonstra ser uma arquitetura funcionalmente simples, utilizando uma estrutura com alto grau de paralelismo espacial. É importante ressaltar que esse trabalho constitui a primeira tentativa de implementação da arquitetura proposta por Leite [40].

1.1. Objetivo do Trabalho

O objetivo deste trabalho é apresentar o estudo do modelamento comportamental e síntese de alto nível de processadores matriciais usando a linguagem VHDL, tendo como tecnologia alvo o mapeamento em componentes FPGA. Baseando-se na implementação do processador de vizinhança (figura 1) NP9 (*Neighborhood Processor 9*), aplicado a imagens digitais, esse estudo aborda as limitações das ferramentas envolvidas no desenvolvimento e o impacto na concepção e implementação dos modelos. Através dessa experiência, foram realizadas novas contribuições baseadas no modelo da arquitetura proposta, visando adaptar e viabilizar a sua implementação em componentes FPGAs.

Figura 1: Evolução do projeto do NP9

1.2. Organização do Trabalho

O capítulo 2 trata dos problemas de PDI, discute os tipos de paralelismo de arquiteturas para PDI, apresenta os principais processadores matriciais existentes e apresenta a definição em alto nível do

NP9. O capítulo 3 faz uma descrição geral acerca da metodologia de projeto utilizada neste trabalho e as ferramentas empregadas. O capítulo 4 trata do modelamento comportamental de circuitos em VHDL e dos requisitos necessários a uma síntese de alto nível mais eficiente. O capítulo 5 apresenta os componentes FPGA, sua estrutura e seus tipos, e faz algumas considerações sobre o relacionamento entre síntese de alto nível e mapeamento tecnológico. O capítulo 6 faz uma análise sobre as alternativas preliminares de implementação do NP9, descreve a estrutura da versão implementada, seus processadores elementares e funcionamento, e traz uma breve introdução a um sistema de processamento de imagens utilizando esse processador. O capítulo 7 apresenta e analisa os resultados da metodologia proposta e propõe alternativas para possíveis melhorias no desempenho do projeto NP9. O capítulo 8 faz as considerações finais dessa dissertação, concluindo-a. O apêndice A contém os modelos VHDL utilizados na implementação e o apêndice B, os resultados da simulação do processamento de uma imagem utilizando o NP9.

2. ARQUITETURAS PARA PROCESSAMENTO DIGITAL DE IMAGENS

Os problemas relativos ao processamento digital de imagens (PDI) estão agrupados em duas classes: *processamento de baixo nível* e *processamento de alto nível*. No domínio dos problemas de processamento de baixo nível temos a aquisição, gerenciamento, codificação, restauração e melhoramento de imagens, segmentação e extração de características. Neste tipo de processamento, a imagem de saída possui o mesmo formato da imagem de entrada. O processamento de alto nível de imagens consiste da classificação, interpretação e avaliação daquelas características e não atua diretamente sobre os *pixels* da imagem, mas sobre as primitivas de descrição da mesma.

Algoritmos de processamento de baixo nível operam uniformemente sobre uma imagem a partir de uma função de transformação, baseada numa vizinhança de um *pixel* (seção 2.3). Esta vizinhança depende, entre outros, do domínio da aplicação, das características da imagem e da complexidade de cálculo. Um algoritmo dessa classe pode realizar centenas de operações por vizinhança e necessita que isso seja executado num curto espaço de tempo.

Para ilustrar o problema, considere o processamento de imagens com resolução de 1024 x 1024 pontos sendo transmitidas a 30 quadros (*frames*) por segundo, onde 1 byte corresponda a um *pixel*. Neste caso, cada imagem possui 1 Mbyte e deve ser processada em cerca de 33,3 ms, com a taxa de transmissão do sistema atingindo cerca de 30 Mbytes/s. É importante lembrar que, em algumas aplicações, as imagens podem atingir resoluções tais como 4000 x 4000 *pixels* (aplicações militares) e 15000 x 15000 *pixels* (imagens de satélites).

Sendo assim, é importante que um sistema de PDI, sobretudo no que concerne ao processamento de baixo nível, tenha uma estrutura regular, com alto grau de paralelismo espacial, e grande desempenho. Este capítulo apresenta os tipos de paralelismo encontrado nas arquiteturas de PDI de baixo nível, as principais arquiteturas para esse tipo de aplicação e alguns exemplos, e descreve a estrutura do processador de vizinhança no qual está fundamentado o NP9.

2.1. Tipos de Paralelismo de Arquiteturas para PDI

Danielsson e Levialdi [17] apresentam uma taxonomia para os tipos de paralelismo de arquiteturas para processamento de imagens, baseada em quatro classes (ou dimensões):

a) paralelismo de operador;

b) paralelismo de imagem;

c) paralelismo de vizinhança;

d) paralelismo pixel-bit.

O **paralelismo de operador** é equivalente ao esquema de pipeline utilizado em algumas arquiteturas. Nesse modo, um determinado tipo de processamento é quebrado em múltiplas operações, arranjadas em estágios subseqüentes. O primeiro estágio recebe a primeira imagem, processa-a e passa ao próximo estágio, que inicia o processamento dessa entrada, enquanto o primeiro estágio já se ocupa com uma nova entrada. Essa situação repete-se em todos os estágios da linha de processamento.

No **paralelismo de imagem**, diversos processadores atuam sobre a imagem ao mesmo tempo. Cada um desses processadores atua sobre uma área da imagem, em geral um pixel e seus vizinhos, e todos operam simultaneamente, executando a mesma instrução. Esse tipo de processamento é o que chamamos de arquitetura SIMD (*Single Instruction, Multiple Data*).

O **paralelismo de vizinhança**, considera a ação do processador sobre um dado pixel e seus vizinhos. Este tipo de paralelismo indica o tamanho da vizinhança que pode ser processada a cada instante. Ou seja, uma arquitetura com maior paralelismo de vizinhança é capaz de executar algoritmos mais complexos (no que se refere a vizinhança) em menos tempo do que outra com menor paralelismo.

O **paralelismo pixel-bit**, refere-se ao processamento de cada pixel da imagem, considerando a cadeia de bits que o formam. Um alto paralelismo pixel-bit numa arquitetura, por exemplo, indica que ela pode processar imagens com maior variação de tons de cinza.

2.1.1. Grau de Paralelismo

Considerando essas quatro dimensões - operador, imagem, vizinhança e pixel - Danielsson sugeriu [17] uma equação que representa o **grau de paralelismo** de uma determinada arquitetura:

$$K = k_o \cdot k_i \cdot k_v \cdot k_p \qquad (1)$$

onde k_o, k_i, k_v e k_p são, respectivamente, as dimensões da arquitetura mencionadas acima e K é o paralelismo total. Através do grau de paralelismo, é possível afirmar que dois sistemas com mesmo K tenham potencialmente o mesmo desempenho, mas apenas a freqüência de trabalho e as características de projeto do processador determinam a capacidade real do sistema.

Como exemplo, suponha um processador que possui um pipeline com 2 estágios ($k_o = 2$), processa um pixel da imagem por vez ($k_i = 1$), considera uma vizinhança de tamanho 3x3 ($k_v = 9$) e opera sobre imagens de 256 níveis de cinza ($k_p = 8$ bits). Segundo a equação 1, esse processador tem um

paralelismo potencial (K) igual a 144 ($K=2 \times 1 \times 9 \times 8$). Esse número, por si só nada indica, mas é importante numa eventual comparação com outro processadores.

2.1.2. Influência das dimensões de paralelismo

Os processadores com ênfase no paralelismo de imagem, são indicados para o processamento genérico de imagens. Esses processadores possuem, em geral, uma arquitetura bastante simples e uma relação custo-benefício melhor que outras arquiteturas. Os processadores matriciais, que serão vistos na seção 2.2, são o exemplo mais característico.

O paralelismo de operador é mais indicado em aplicações de processamento de imagens específicas, aplicações para as quais a arquitetura esteja bem ajustada. Como, nesse caso, há vários estágios atuando sobre a imagem, é necessário que o maior número de estágios possível esteja sendo utilizado e cada estágio seja dimensionado para atender aos requisitos de desempenho. Do contrário, o processador atuará de maneira ineficiente ou estará sub-utilizado.

O paralelismo pixel-bit e de vizinhança estão relacionados ao modo e à capacidade de operação do processador. Quando $k_v = 1$, o processador realiza apenas operações pontuais (à exceção dos processadores matriciais). Se $k_v = 5, 7$ *ou* 9, o processador atua, respectivamente, sobre vizinhanças ortogonais, hexagonais e completas. O valor k_p indica a faixa de valores dos pixels da imagem; se o processador possui memória local, é possível estender as operações a uma faixa de valores maior.

2.2. Exemplos de Processadores Matriciais

A arquitetura mais utilizada no processamento de baixo nível de imagens é a de processadores matriciais. Esses processadores possuem arquitetura paralela (figura 2), composta de processadores mais simples, denominados células ou processadores elementares (PEs). Conforme Fountain [22], possuem, em geral, algumas características principais:

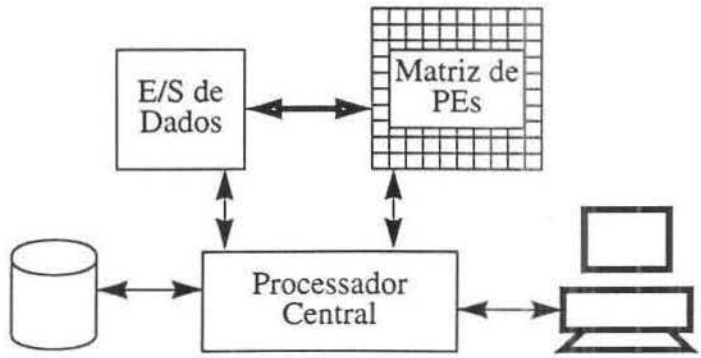


Figura 2: Arquitetura de um processador matricial.

a) os processadores elementares são organizados numa matriz (duas dimensões);

b) cada PE opera, geralmente, em modo bit-serial (os bits de uma palavra são processados um a um sequencialmente);

c) cada PE tem acesso a um bloco de memória local;

d) cada PE está conectado a seus vizinhos mais próximos na matriz;

e) num dado momento todos os PEs executam a mesma instrução;

Esta qualificação, feita em 1987, hoje não é completamente correta e algumas extensões podem ser consideradas. A complexidade dos processadores elementares, por exemplo, não se restringe à operação a nível de bits, alguns processadores operam a nível de palavras, variando entre si no tamanho da palavra (8, 16, 32). O NP9 (*Neighbourhood Processor 9*), por exemplo, é um processador que trabalha externamente com palavras de tamanho 8 e, internamente, com tamanho 9, como veremos nas especificações da implementação do projeto, na seção 6.3.

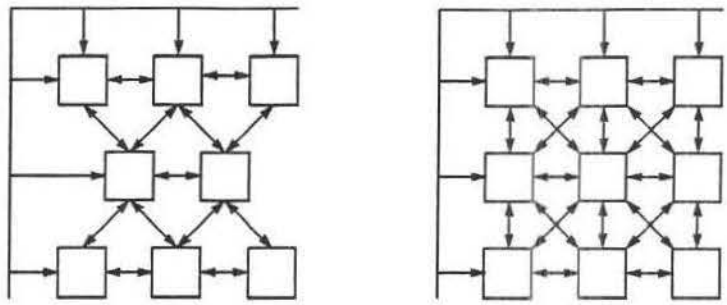
Existe ainda um Processador Central, responsável pelo endereçamento, coordenação e controle dos PEs. Em geral, o número de PEs da matriz é inferior ao número de *pixels* da imagem, o que demanda uma reconfiguração da matriz e/ou uma estrutura especial de endereçamento. Por razões econômicas, a estrutura da matriz pode ser reduzida a apenas uma dimensão (linha ou coluna).

A seguir, este capítulo apresenta alguns dos mais importantes processadores matriciais aplicados à área de processamento de imagens. Estes processadores, foram desenvolvidos durante o início da década de 80 e têm uma contribuição relevante para esse trabalho por representarem as primeiras iniciativas no campo de projeto de arquiteturas paralelas para processamento de imagens. Os exemplos aqui relacionados são o CLIP (seção 2.2.1), o DAP (seção 2.2.2) e o MPP(seção 2.2.3). Uma comparação entre os exemplos apresentados aqui pode ser vista em [58].

2.2.1. CLIP - *Cellular Logic Image Processor*

CLIP é o nome de uma família de processadores desenvolvidos no University College London pelo grupo liderado por Duff [19]. O CLIP4 [21] surgiu em 1980 e foi a primeira versão desses processadores a ser desenvolvida com fins comerciais, após o desenvolvimento de três protótipos experimentais.

O elemento central do sistema CLIP4 é uma matriz bidimensional de 96x96 PEs (totalizando 9216), cada PE comunicando-se com seus 6 ou 8 vizinhos mais próximos, originando uma conexão hexagonal (figura 3a) ou quadrada (figura 3b). Nessa matriz, todos os processadores elementares executam a mesma instrução, simultaneamente, e estão submetidos a um controlador central. Existe ainda um barramento de controle passando por todos os PEs.



(a) (b)

Figura 3: Organizações hexagonal (a) e quadrada (b) do CLIP4

Cada PE (figura 4) possui três registradores internos de 1 bit (A, B e C) e uma memória local de 32 bits (D). O registrador A é usado como acumulador e para E/S de dados, armazenando dados referentes a imagem local. B é utilizado em processamentos com duas imagens. C contém o resultado da função de propagação (*carry*) do processador booleano 2; e D, o novo valor da imagem produzido pelo processador 1. De fato, os dois processadores booleanos são parte de uma mesma estrutura que tem como entradas o conteúdo de A e P, um valor resultante da combinação de B, C e T, onde T é o resultado da função de entrada responsável pela seleção dos vizinhos do PE.

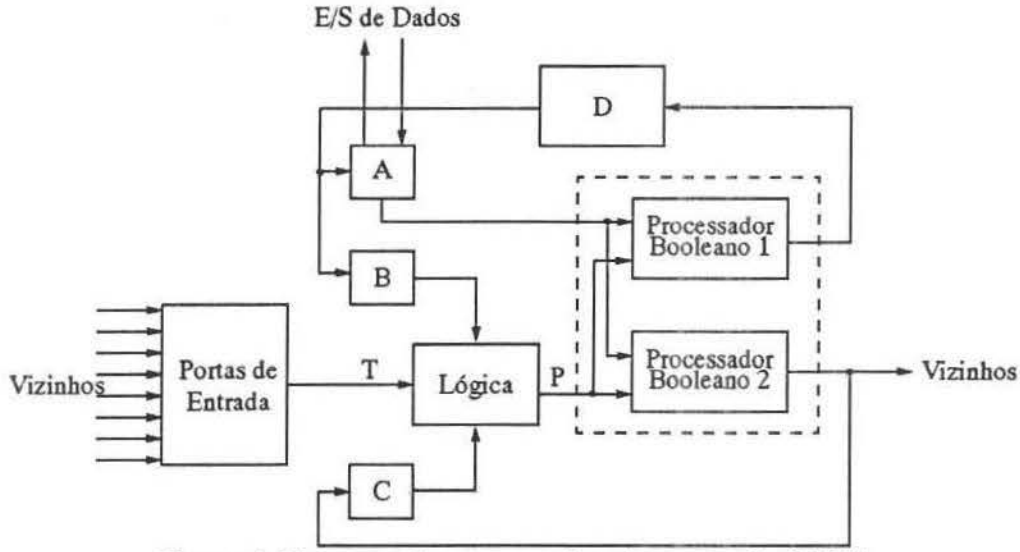


Figura 4: Esquema do processador elementar do CLIP4

A matriz do CLIP4 está implementada em 1152 pastilhas LSI (1056, numa versão reduzida [21]), contendo 8 processadores elementares cada uma. O sistema trabalha com uma freqüência de 1 MHz, limitado pela tecnologia de fabricação, apesar do projeto inicial especificar 2,5 MHz.

O CLIP4 sofreu novas extensões e melhorias [19] que culminaram no desenvolvimento do CLIP7 [23]. Os objetivos principais do projeto CLIP7 eram: produzir uma máquina capaz de processar imagens de alta resolução (512x512 pixels) com taxas comparáveis às do CLIP4 (que operava sobre

imagens de 96x96 pixels); estudar alternativas para possibilitar a autonomia local dos processadores elementares e estudar novas arquiteturas, inclusive estruturas 3-D, visando novos modelos de interconexão entre os processadores elementares.

2.2.2. MPP - *Massively Parallel Processor*

O MPP [10][57] nasceu de um projeto financiado pela NASA durante a década de 70. Este projeto consistia do estudo de métodos para se obter máquinas que alcançassem altíssimo desempenho computacional. Lançado pela *Goodyear-Aerospace* em abril de 1983, é até hoje um dos maiores processadores matriciais já construídos [63].

Sua matriz de processadores elementares possui 16384 processadores bit-seriais, organizados no formato 128x128. Todo o sistema engloba 2048 pastilhas, contendo 8 PEs cada uma, e trabalha a 40 MHz. Cada processador elementar possui uma memória externa de 1 Kbit (por princípios de projeto, pode ser ilimitada) e conecta-se a seus quatro vizinhos mais próximos (ortogonais).

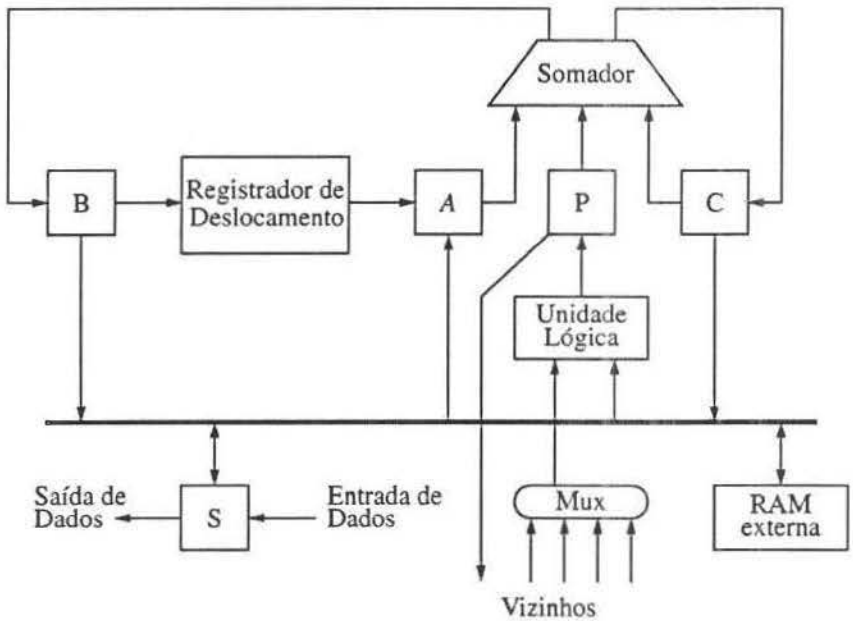


Figura 5: Estrutura do processador elementar do MPP

A estrutura de cada PE (figura 5) inclui um registrador de deslocamento de tamanho variável (máximo de 32 bits) com alto desempenho, 5 registradores de propósito específico (A, B, C, P e S). B e C armazenam, respectivamente, o resultado da adição e o carry. S é um registrador dedicado a E/S de dados. É importante notar que no MPP as funções aritméticas e lógicas são implementadas por estruturas diferentes, permitindo o paralelismo dessas operações.

2.2.3. DAP - *Distributed Array Processor*

O primeiro protótipo do DAP, contendo uma matriz de 32x32 processadores elementares, apareceu em 1976 [22], a partir do trabalho de Reddaway, Hunt e Parkinson na ICL (*International Computers Ltd.*). A primeira versão comercial [27] possuía 4096 PEs (64x64) com 4Kbits de memória cada uma. Os processadores elementares estão conectados a seus 4 vizinhos ortogonais e sob controle da *Master Control Unit* (MCU), que difunde a seqüência de instruções a ser executada.

A estrutura do processador elementar do DAP é bastante simples e, conforme visto na figura 6, conta com os seguintes componentes:

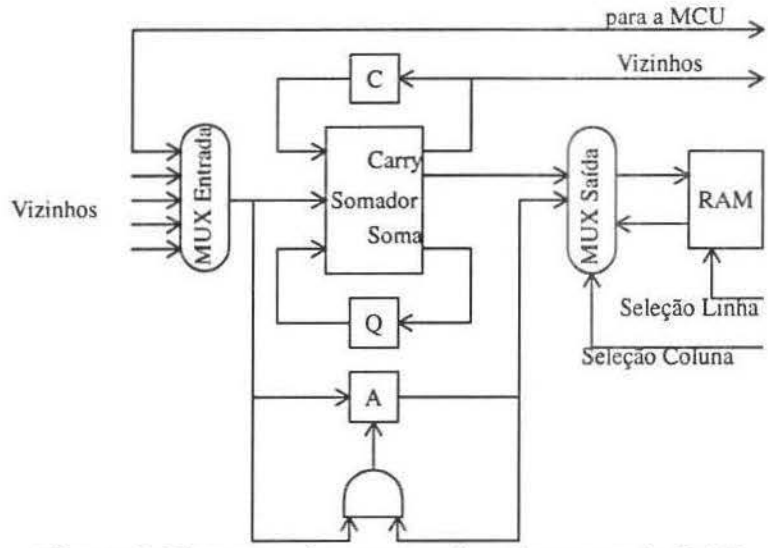


Figura 6: Diagrama do processador elementar do DAP

a) um multiplexador de entrada, que seleciona um dos 4 vizinhos ou a saída do PE;

b) um somador de 1 bit;

c) um registrador de carry (C), que também serve de entrada para o somador;

d) um acumulador (Q), que é outra entrada para o somador;

e) um registrador de propósito geral (A), cuja principal função é controlar o armazenamento de resultados na RAM;

f) uma RAM de 4 Kbits;

g) um multiplexador de saída.

Inicialmente projetado para o processamento numérico e outras aplicações como análise de elementos finitos, modelagem matemática e análise de problemas meteorológicos, o DAP foi naturalmente associado ao processamento de imagens, devido ao arranjo espacial de seus PEs.

2.3. O processador de vizinhança NP9

Um processador de vizinhança é um dispositivo que processa uma imagem, produzindo outra, onde cada *pixel* resultante é gerado em função de seu correspondente na imagem de entrada e dos seus vizinhos mais próximos. Este processador simula um processador matricial e atua sobre a imagem fazendo uma varredura em suas linhas (figura 7), utilizando uma vizinhança de tamanho padrão, em geral, 3x3, 5x5 ou 7x7.

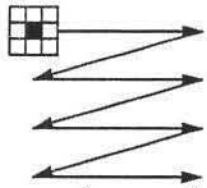


Figura 7: Esquema de varredura da imagem

Alguns exemplos de processadores de vizinhança que podem ser citados são o PIMM1 [52], e o INP20 [55]. O modelo do processador definido por Leite [39] considera, inicialmente, uma vizinhança de tamanho 3x3 (9 pixels, como visto na figura 8) e é a base para a implementação do NP9. Como veremos nas próximas seções, o NP9 é altamente reconfigurável, modular e extensível, capaz de executar um grande número de algoritmos de PDI a partir da mesma estrutura básica. Isso é devido à flexibilidade de programação embutida em seus processadores elementares.

X_1	X_2	X_3
X_4	X_5	X_6
X_7	X_8	X_9

Figura 8: Representação de uma vizinhança 3x3

2.3.1. Os Processadores Elementares do NP9

Os processadores elementares do NP9 são simples do ponto de vista funcional e realizam apenas duas operações básicas: adição (ADD), função que representa a classe de algoritmos lineares, e máximo (MAX), representando a classe de algoritmos não-lineares. Num modelo simplificado (figura 9), em alto nível de abstração, o PE possui duas entradas - os pixels X_1 e X_2 - às quais são atribuídos pesos inteiros - W_1 e W_2 - e uma saída S, cuja valor é dado pela função:

$$S = f(\varepsilon, X_1 W_1, X_2 W_2) \tag{2}$$

ε representa o estado lógico do PE e indica a função que está sendo realizada, conforme a relação:
$ADD((X_1 W_1, X_2 W_2) \rightarrow S;$ se $\varepsilon = 0)$ e $MAX((X_1 W_1, X_2 W_2) \rightarrow S;$ se $\varepsilon = 1)$

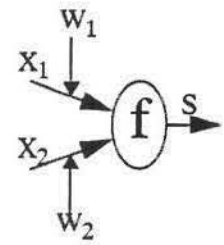


Figura 9: Modelo dos processadores elementares do NP9

Os pesos atribuídos aos valores dos pixels de entrada são usados para estender o número de funções lógicas realizáveis pelo PE (figura 10). Como exemplo dessas extensões, atribuindo o valor -1 como peso a cada uma das entradas, multiplicando S também por -1 e programando o PE para a operação MAX, obtem-se a operação de mínimo (MIN) das entradas (figura 10d). Além disso, a programabilidade dos PEs permite a implementação de funções booleanas, como pode ser visto na tabela 1.

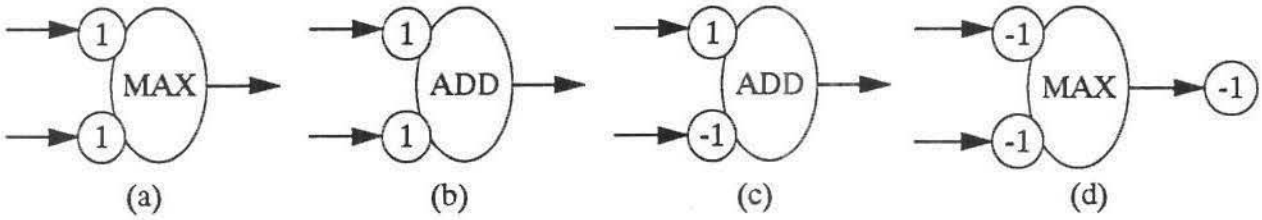


Figura 10: Quatro exemplos de configuração do processador elementar do NP9

Tabela 1: Tabela-Verdade de funções Booleanas implementadas no NP9

X1	X2	Not X Add(1,-X)	X1 or X2 Max(X1,X2)	X1 and X2 -Max(-X1, -X2)
0	0	1	0	0
0	1	1	1	0
1	0	0	1	0
1	1	0	1	1

2.3.2. O Grafo de Fluxo de Dados

A matriz de interconexão entre os processadores elementares do NP9 segue um grafo de fluxo de dados definido por uma classe de filtros não-lineares. Esta classe, muito utilizada em processamento de baixo nível, é representada pela seguinte equação [56]:

$$y = f\left(\sum_{i=1}^{N} a_i (b_i g(x_i))_{(i)} \right) \tag{3}$$

onde y é o novo valor para o *pixel* central da vizinhança; x_i são os *pixels* da vizinhança; $g(\cdot)$ e $f(\cdot)$ são funções não-lineares usadas para reduzir o ruído da imagem, e a_i e b_i (onde i=1,...,N) são os coeficientes

do filtro em questão. Os valores $b_i g(x_j)_{(i)}$ correspondem aos valores $b_i g(x_j)$ em ordenação crescente.

A equação 3 engloba várias classes de filtros não-lineares e detectores de contorno utilizados em aplicações de processamento de imagem: filtros de mediana, filtros de estatística de ordem, operações de dilatação e erosão, detectores de intervalo de contorno, detectores de dispersão, entre outros. É importante salientar ainda que o modelo dos PEs e a capacidade de programação embutida possibilita a execução de um grande número de outros algoritmos, além dos definidos pela equação 3 [38].

Analisando a equação, é possível verificar que o núcleo da estrutura de dados é um algoritmo de ordenação ($b_i g(x_j)_{(i)}$). O grafo de fluxo de dados da arquitetura do NP9 baseia-se no algoritmo de transposição ímpar-par (figura 11) [34]. Este algoritmo não apresenta dificuldades para implementação em hardware. Akl [2] descreve um algoritmo paralelo desta ordenação e faz uma análise do mesmo. A estrutura assim definida (figura 12) alcança um bom compromisso entre complexidade, paralelismo, custo e tempo de execução.

```
Seja x1,...,xn um conjunto de n elementos a ser ordenado
Para k = 1,..., n faça
  Se k é ímpar então
    Para i = 1,3,..., 2⌊n/2⌋-1 faça
      Se xi > xi+1 então xi ↔ xi+1 fimse
    fimpara
  senão
    Para i = 2,4,..., 2⌊(n-1)/2⌋ faça
      Se xi > xi+1 então xi ↔ xi+1 fimse
    fimpara
  fimse
fimpara
```

Figura 11: Algoritmo de transposição ímpar-par

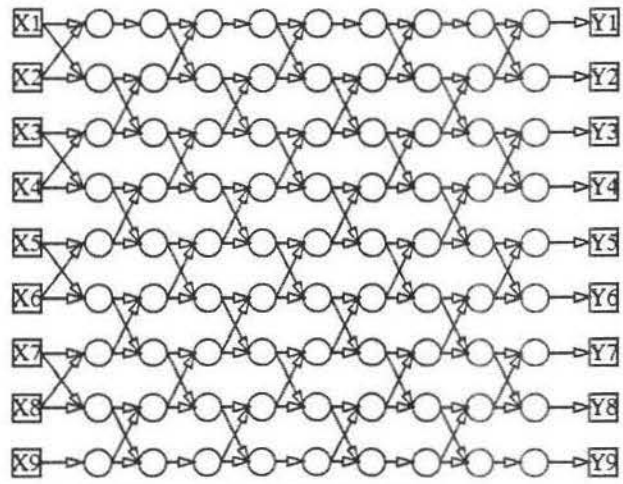


Figura 12: Grafo de Fluxo de Dados do processador

2.3.3. Potencialidades

Considerando as funções primitivas definidas e a flexibilidade de programação inerente à rede de processadores elementares, existe um grande número de algoritmos de transformação de imagens que podem ser mapeados no grafo da figura 12 (além daqueles representados pela equação 3. Entre eles, podemos citar [38]: filtros lineares (convolução), não-lineares (representados pela equação 3) e híbridos (filtro de Wilcoxon, por exemplo), operações morfológicas binárias e de níveis de cinza (erosão, dilatação, algoritmos de afinamento e espessamento, transformada em tudo ou nada, detectores morfológicos de contorno, gradiente, operações "*top hat*", operações geodésicas binárias e de níveis de cinza (dilatação e erosão geodésicas, reconstrução de imagens etc), transformada de distância, etc. O apêndice B apresenta alguns exemplos e resultados desses algoritmos implementados no NP9.

2.4. Conclusão

O PDI possui uma elevada carga computacional, por trabalhar com um grande volume de dados e executar inúmeras repetições de operações elementares que poderiam ser implementadas em paralelo. Este capítulo apresentou uma breve análise sobre o paralelismo das arquiteturas de PDI e alguns exemplos de processadores matriciais, um tipo de arquitetura que se aplica adequadamente a tal aplicação. Por fim, foi apresentado o NP9, um modelo de processador de vizinhança baseado numa arquitetura reconfigurável.

3. METODOLOGIA DE PROJETO

O atual mercado de sistemas eletrônicos exige máquinas cada vez mais velozes e eficientes e num tempo menor que o habitual em décadas passadas. Para se adequar a essa realidade, os projetistas necessitam, cada vez mais, de ferramentas ágeis e produtivas, que permitam conceber um sistema com maior precisão, qualidade e rapidez. A especificação em alto nível e a prototipação rápida do sistema devem estar disponíveis, por serem técnicas que, além de suprir as necessidades de projeto mencionadas acima, adicionam outras vantagens também desejáveis.

A especificação do sistema em alto nível de abstração leva em consideração a descrição de uma aplicação bastante específica, com uma abordagem algorítmica. Essa técnica, também chamada de modelamento comportamental, confere ao projeto grande reusabilidade e flexibilidade de mapeamento do circuito final. Uma vez que a preocupação principal está voltada à solução do problema, e não ao modo como essa será implementada, a fase de implementação do circuito é automatizada e o tempo de projeto é reduzido consideravelmente. Contudo, é necessário que a especificação seja feita visando viabilizar a síntese de alto nível do circuito.

Este capítulo apresenta os objetivos do projeto (seção 3.1), descreve as principais etapas englobadas no fluxo do projeto (seção 3.2), mostra uma visão geral das técnicas de implementação de circuitos existentes (seção 3.3) e comenta as ferramentas utilizadas no desenvolvimento do projeto (seção 3.4). Os capítulos 4 e 5 complementam as informações constantes das seções aqui apresentadas, dando uma visão mais detalhada do modelamento comportamental em VHDL, da síntese de alto nível e dos componentes FPGA.

3.1. Objetivos do Projeto

O objetivo deste projeto é o estudo do modelamento e síntese de processadores matriciais usando VHDL, com ênfase no mapeamento em componentes FPGA, através da implementação de um processador de vizinhança para aplicações em imagens digitais. Esse estudo considera e analisa as limitações das ferramentas envolvidas no desenvolvimento e o impacto na concepção e implementação dos modelos. O enfoque principal é na metodologia de desenvolvimento e projeto, e espera-se realizar também novas contribuições ao modelo da arquitetura proposta, visando à adaptação e viabilidade da implementação em dispositivos programáveis do tipo FPGA.

3.2. Fluxo do Projeto

Caracteristicamente o fluxo de projeto de um circuito integrado [16], utilizando modelamento comportamental e síntese de alto nível, segue as etapas que podem ser vistas na figura 13: Especificação Funcional, Modelamento em VHDL, Síntese Lógica, Mapeamento de Tecnologia, Simulação Lógica, Implementação/Fabricação e Testes.

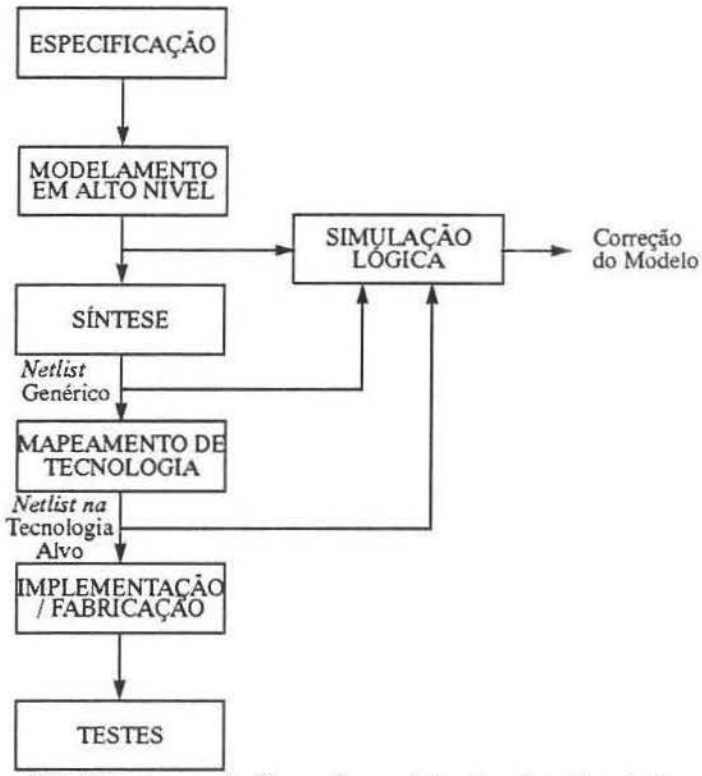


Figura 13: Diagrama de fluxo de projeto de circuitos integrados

3.2.1. Descrição das Etapas de Projeto

A etapa de **Especificação** responde a duas questões básicas: o que e por que fazer. É o momento de determinar algumas características do circuito a ser projetado, as funções a serem executadas, sua estrutura básica e modo de funcionamento. Outras características também podem ser estimadas: freqüência máxima, encapsulamento, número de portas lógicas utilizadas, número de pinos de E/S, tensão de alimentação, etc. As finalidades destas estimativas são: auxiliar na simulação parametrizada do circuito; conduzir a síntese e o mapeamento, de forma a obter resultados satisfatórios; verificar e avaliar os resultados finais de desempenho e utilização de recursos, em comparação às pré-especificações. Segundo alguns especialistas [26], decisões tomadas nos primeiros 5% do ciclo de projeto são responsáveis por 85% do custo total de desenvolvimento.

O **Modelamento em Alto Nível** corresponde à etapa na qual são descritas as especificações definidas anteriormente, utilizando-se uma linguagem de descrição de hardware (HDL - *Hardware Description Language*). O capítulo 4 contém maiores detalhes sobre as HDLs e o modelamento em alto nível, em especial utilizando a VHDL (*VHSIC Hardware Description Language*).

A **Síntese** [13][14][36] consiste na compilação dos modelos em alto nível e a geração automática, através de ferramentas de síntese, de um *netlist* genérico do CI. O *netlist* pode ser entendido como um lista de interconexões entre as células e/ou as portas lógicas do circuito. Esta fase do projeto, praticamente, não possui nenhuma interferência direta do projetista, mas depende muito do estilo de codificação utilizado [1]. Detalhes sobre a Síntese de Alto Nível estão no capítulo 4.

Uma vez especificadas as características do CI, os modelos criados e a etapa de síntese concluída, chega o momento do mapeamento do *netlist* genérico, para a **tecnologia alvo**. Ou seja, é a fase onde os modelos de alto nível são implementados fisicamente em alguma técnica: *gate array, standard cell*, PLA, FPGA, etc. Este mapeamento é realizado através de bibliotecas pré-definidas por cada fabricante para a técnica escolhida.

A **Simulação Lógica** é uma etapa auxiliar dentro do ciclo de projeto e possui importância fundamental. É a principal técnica que possibilita a validação do sistema a cada etapa do projeto. A validação garante que uma especificação de um projeto confere com os requisitos e objetivos do projetista. Desta forma, pode-se realizar alterações ainda em projeto, evitando a descoberta tardia de anomalias ou erros de especificação e/ou modelagem. A simulação pode ser realizada após cada uma das fases de projeto (figura 13), a fim de avaliar tanto a funcionalidade da especificação quanto o resultado dos processos automatizados, como a síntese e o mapeamento. A cada etapa, a simulação tem uma função um pouco distinta. Após o mapeamento, por exemplo, já estão associadas ao circuito todas as características de temporização (atrasos combinacionais, freqüência máxima, tempo de subida e descida, etc) relativas à tecnologia de implementação.

Nas fases anteriores, encerra-se a concepção do circuito, e a próxima etapa é a **Implementação ou Fabricação** do circuito. Agora o CI é finalmente um produto acabado, pronto para operar na aplicação para a qual foi concebido, mas não sem antes passar pela última etapa: a de **Testes.** O teste de um circuito [15] consiste na aplicação de uma seqüência adequada de sinais nas suas entradas, e na observação das respostas obtidas nas suas saídas. O CI é conectado a um sistema igual ou similar ao qual onde será utilizado e submetido a uma série de "testes em bancada". Os testes costumam ser divididos em dois tipos: teste de protótipo e teste de produção. O primeiro é executado sobre um protótipo e tem a finalidade de verificar, ao final do projeto, se o CI cumpre as especificações e a de auxiliar na localização de possíveis erros de projeto. O segundo é realizado em unidades fabricadas já em linha de produção e visa a separação de circuitos bons de defeituosos. Com os componentes FPGA, os testes

de produção são dispensáveis, pois assume-se que os componentes são corretos por construção quando colocados à venda, sendo necessária apenas a validação funcional dos modelos e os testes de protótipo.

3.3. Técnicas de Implementação

Uma vez descrito o fluxo de projeto, é necessário apresentar as técnicas de implementação que podem ser utilizadas na implementação final do circuito. Estas opções são divididas em duas classes fundamentais [26]: **dedicados** (*fullcustom - FC*) e **semi-dedicados** (*semicustom*).

Os circuitos dedicados oferecem a melhor escala de integração e possuem melhor desempenho. Durante a etapa de construção de um chip, várias máscaras [64] são elaboradas para possibilitar a superposição dos materiais que vão compor o circuito (difusão, polissilício, metal, etc). Nesta técnica, o projetista é responsável pelo desenho de todas as máscaras do processo de fabricação. Estas máscaras são detalhadas uma a uma e enviadas para a fabricação final do CI. O número de máscaras varia entre 10 e 13, dependendo do número de camadas de metalização e polissilício e da tecnologia de fabricação (CMOS, bipolar, etc).

Os circuitos semi-dedicados têm apenas algumas destas máscaras desenhadas, ou poucas destas estão sob o controle do projetista. Ou seja, o circuito é fabricado até um nível considerado "padrão" e, a partir daí, o projetista é responsável por definir as máscaras restantes. Os tipos de circuitos semi-dedicados mais comuns são:

a) ***Gate Arrays*** **(GA)** - Esta técnica consiste de *wafers* pré-processados contendo elementos lógicos (*gates*) e necessitam de apenas de uma a três máscaras de metalização. Esta metalização corresponde à conexão entre os gates pré-modelados. Os GAs podem conter até aproximadamente 200.000 (duzentos mil) *gates* num único chip, mas na prática, apenas 70 a 90% são utilizados, devido à necessidade de roteamento e à própria forma de aproveitamento dos gates. Têm um rápido retorno no desenvolvimento de protótipos e são aconselháveis para uso na produção de até alguns milhares de unidades.

b) ***Standard Cells*** **(SC)** - Um circuito utilizando SC é construído utilizando-se células "prontas", com esquema pré-definido e tamanho padrão. Ou seja, existem bibliotecas contendo funções lógicas padrão (AND, OR, INVERTER, XOR, etc) que são interconectadas de forma a compor o esquema desejado. Essa técnica é relativamente rápida, mas, por requerer um número de máscaras para fabricação semelhante ao dos circuitos dedicados, possui um custo fixo de produção mais alto. Contudo, o custo total de produção é inferior ao de outras técnicas.

c) **FPGA** - É a técnica mais moderna, ainda não explorada em todas as suas particularidades e será vista em detalhes no capítulo 5.

A escolha da técnica mais adequada ao projeto depende de uma série de fatores, entre outros: tempo de desenvolvimento, desempenho esperado, densidade de integração, uso de estruturas especiais e custo total de implementação. O custo total da implementação é dado pela equação

$$\text{Custo} = \text{Custos de NRE} + (\text{Custo por peça}) \cdot (\text{No. de peças}) \quad (4)$$

onde custos de NRE (*NonRecurring Engineering*) [26] representam os custos fixos totais da produção e envolvem os custos fixos de fabricação propriamente dita e os custos de projeto/desenvolvimento. Os custos de NRE independem do número de peças, uma vez que estão relacionados somente aos valores que são fixos dentro da produção, como o custo de produção das máscaras de fabricação, custo homem-hora de projeto, entre outros.

A tabela 2 sumariza a classificação das tecnologias de implementação com relação aos critérios de escolha mencionados.

Tabela 2: Classificação das tecnologias de implementação

Critérios		FC	SC	GA	FPGA
Densidade de Integração		alta	média/alta	média	baixa
Tempo de Desenvolvimento		longo	longo/médio	médio	curto
Desempenho Esperado		alto	bom	médio	baixo
Custos NRE	Desenvolvimento	alto	baixo	baixo	baixo
	Fabricação	alto	alto	baixo	nenhum
Custo por Peça		baixo	baixo	médio	alto

3.4. Ferramentas de Projeto

O projeto do NP9 foi desenvolvido utilizando o sistema de EDA (*Electronic Design Automation*) [43] da *Mentor Graphics Corporation* e o sistema de mapeamento *Max+Plus II* [6], da *Altera Corporation*. Estes sistemas aliados cobrem todas as etapas do fluxo de projeto. O sistema Mentor é totalmente modularizado, composto de diversas ferramentas com funções bem definidas. De todo o sistema Mentor, o projeto do NP9, numa primeira fase, fez uso das principais aplicações: *System-1076* [44], *QuickSim II* [45] e *AutoLogic* [46]. A segunda fase utilizou uma nova geração de ferramentas: *QuickHDL* e *Autologic II* [47].

System-1076 é um compilador VHDL baseado no padrão IEEE Std 1076-1987 definido para esta linguagem. **QuickSim II** é um simulador lógico interativo e auxilia na verificação da funcionalidade de projetos eletrônicos descritos em termos de portas lógicas ou VHDL. Essa ferramenta é muito útil durante todo o ciclo de projeto.

A porta de acesso para o **AutoLogic** é o **AutoLogic VHDL**. Ele aceita os modelos VHDL compilados e sintetiza projetos lógicos ao nível de transferência de registradores (RTL - *Register Transfer Level*). O AutoLogic VHDL é independente de tecnologia e permite capturar e sintetizar todo o projeto antes de mapeá-lo na tecnologia alvo. O AutoLogic otimiza o projeto e produz *netlists* específicos para uma tecnologia a partir daqueles genéricos produzidos pelo AutoLogic VHDL, otimizando-os ao nível de portas lógicas, a fim de satisfazer os requisitos em termos de área e/ou desempenho.

O **QuickHDL** é um conjunto de ferramentas que permite gerenciar, compilar e simular projetos baseados em linguagens de descrição de hardware (HDLs) e aceita tanto modelos VHDL quanto Verilog. Os principais itens desse conjunto são: **qhlib**, para a criação de bibliotecas de modelos compilados; **qhmap**, para associar a designação lógica da biblioteca a um caminho físico; **qvhcom**, compilador VHDL; **qhmake**, gerador de makefile padrão, que permite gerenciar as atualizações da biblioteca via **make** do UNIX; e **qhsim**, simulador lógico.

O **AutoLogic II** tem a mesma função do AutoLogic, mas possui uma interface mais amigável, aumentando a facilidade de uso e, conseqüentemente, a produtividade do projetista. Além disso abrange um subconjunto da linguagem VHDL maior que o seu antecessor, permitindo um nível ainda maior de abstração no projeto.

O **Max+Plus II** é um sistema integrado [7], multi-plataforma, que realiza diversas funções, desde a edição de modelos e captura de diagramas esquemáticos, passando pela síntese lógica, compilação, particionamento, simulação funcional, análise de temporização, verificação e localização de erros, chegando até à programação de componentes.

3.5. Conclusão

O trabalho proposto visa a implementação de um processador de vizinhança baseada no uso de modelamento comportamental em VHDL e síntese de alto nível, com mapeamento em componentes FPGA. A implementação segue um fluxo de projeto composto de 6 etapas: Especificação, Modelamento Comportamental, Síntese de Alto Nível, Mapeamento Tecnológico, Simulação e Implementação. Neste capítulo também foram apresentadas outras tecnologias de implementação além dos componentes FPGA: gate arrays, standard cells e full-custom. Foram discutidas algumas comparações entre estas tecnologias e uma análise de custos de projeto. Por fim, estão descritas neste capítulo, as ferramentas utilizadas no projeto do processador de vizinhança.

4. MODELAMENTO EM VHDL E SÍNTESE DE ALTO NÍVEL

O uso de linguagens de descrição de hardware (HDLs) no projeto de circuitos digitais reduziu o tempo de desenvolvimento e as dificuldades encontradas nos projetos orientados por diagramas esquemáticos, como dificuldade de manipulação de estruturas muito complexas e portabilidade. Projetos complexos, como processadores algorítmicos e estruturas de processamento paralelo, são modelados e compreendidos mais facilmente através de uma linguagem que descreva "o que fazer" e não "como fazer". As HDLs permitem ao projetista preocupar-se com a solução do problema em alto nível, sem necessitar conhecer a fundo a tecnologia de construção de integrados.

Esse capítulo não pretende ser um tratado completo sobre HDLs, mas visa apenas dar uma noção geral. A seção 4.1 traz uma classificação das HDLs; a seção 4.2 apresenta a VHDL (*VHSIC Hardware Description Language*), objeto de nosso estudo e elemento fundamental da metodologia de projeto; a seção 4.3 trata de alguns conceitos relacionados à Síntese de Alto Nível; a seção 4.4 faz considerações importantes no modelamento em VHDL direcionado à síntese.

4.1. Classificação das HDLs

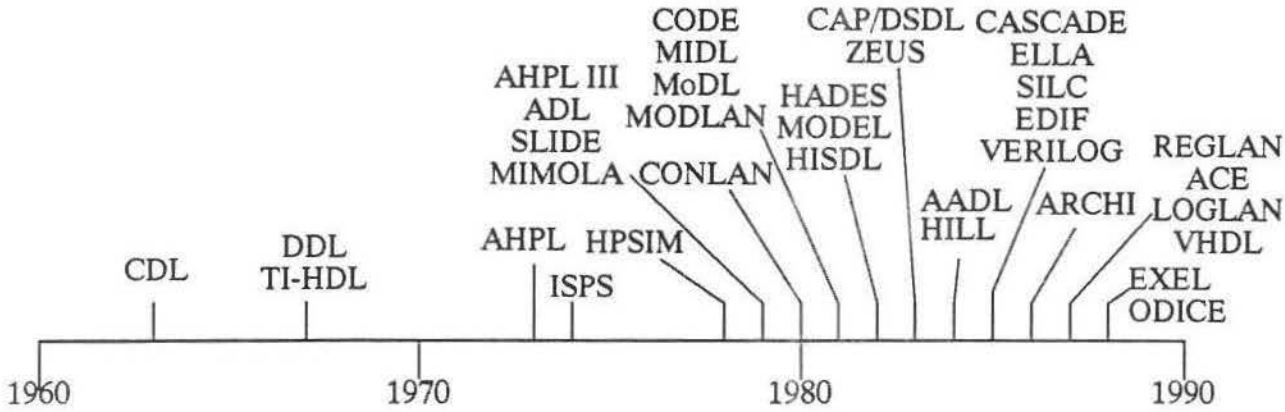


Figura 14: Cronologia de desenvolvimento de algumas HDLs

Conforme pode ser visto na figura 14 [35], as HDLs se multiplicaram rapidamente na década de 80 e dominaram o projeto automatizado de circuitos. Atualmente, duas HDLs se destacam e detém um grande percentual de utilização no mercado mundial: a VHDL e a Verilog [62]. A VHDL é um padrão IEEE desde 1987 (IEEE std 1076-1987), tendo uma atualização em 1993. A linguagem Veri-

log, no entanto, permaneceu como um padrão proprietário até 1990, quando foi colocada em domínio público. O primeiro padrão da Verilog data de 1995 (IEEE std 1364-1995).

A escolha de uma HDL não se baseia somente nas capacidades técnicas da mesma, há ainda outros fatores: preferências pessoais, disponibilidade de ferramentas e indicadores de mercado. As HDLs diferenciam-se por sua maior ou menor adequação às várias etapas do ciclo de projeto [12]: documentação, validação (através de simulação), verificação e síntese. Brage [11] ressalta os principais aspectos que as HDLs devem ter:

a) **Linguagens para Validação**: para a validação uma HDL deve ser de fácil compreensão, pois como a validação serve para garantir a correção da especificação, é importante que a compreensão seja intuitiva; deve também seguir um modelo algorítmico, sem comandos implícitos, o que é requisito básico para a simulação; e possuir um comportamento definido formalmente, caso provas formais sejam usadas na validação.

b) **Linguagens para Verificação:** a verificação compara uma implementação com a sua especificação para assegurar que o processo de projeto está correto. O único método verdadeiro para a verificação envolve o uso de provas formais. Para tanto, a linguagem deve ter um comportamento formalmente definido. Isso implica numa linguagem baseada num modelo matemático e compatível com algum provador formal. Esses requisitos tendem a tornar uma HDL de difícil compreensão para os projetistas.

c) **Linguagens para Síntese:** o aspecto mais importante das linguagens direcionadas à síntese é que o seu comportamento deve ser claro e bem-definido, ou seria impossível predizer os efeitos da síntese. Na prática, é necessário que a HDL seja apropriada tanto para a especificação, quanto para a implementação.

4.2. A linguagem VHDL

A VHDL (*VHSIC Hardware Description Language*) [53] foi proposta em 1981 como parte do programa VHSIC (*Very High Speed Integrated Circuit*) do Departamento de Defesa dos EUA. A primeira versão divulgada (VHDL 7.2) foi desenvolvida a partir de um consórcio das empresas IBM, Texas Instruments e Intermetrics. Já em 1986, os direitos foram cedidos ao IEEE como forma de acelerar o processo de torná-la um padrão nas indústrias. A publicação do primeiro padrão da linguagem VHDL pela IEEE ocorreu em 1987, com o nome de VHDL1076-1987 [32], tendo sofrido ainda uma revisão em 1993.

Através da VHDL pode-se descrever clara e rapidamente [9] o comportamento e estrutura de sistema de hardware digitais, mesmo os mais complexos. O seu uso abrange todas as fases do ciclo de projeto: modelamento, documentação, verificação, síntese, teste e implementação. A metodologia de

trabalho [24][33][61] usando uma linguagem de descrição de hardware em alto nível, como a VHDL, permite um enfoque maior no sistema a ser desenvolvido do que na tecnologia de implementação a ser utilizada, que deve ser uma preocupação da ferramenta de síntese.

Maior produtividade, clareza na descrição, facilidade de manutenção e manuseio de projetos complexos, independência de tecnologia e redução no tempo de projeto são vantagens inerentes aos projetos com VHDL. A portabilidade entre diferentes ambientes de desenvolvimento também pode ser citada como uma vantagem, entretanto ela só é assegurada nas etapas de modelamento e simulação em ferramentas que sigam o padrão VHDL1076 do IEEE. Modelos voltados para a síntese necessitam, em muitas vezes, ser reescritos para atender às restrições das ferramentas.

O usuário também não fica restrito a apenas um estilo de descrição, podendo utilizar a que melhor se adeque aos seus propósitos. O grande atrativo da VHDL reside na possibilidade de se modelar sistemas em vários níveis de abstração diferentes, desde o mais alto nível (conceitual ou comportamental) até o nível de portas lógicas. Basicamente, há duas abordagens de modelamento suportadas pela VHDL: Comportamental e Estrutural.

4.2.1. Modelamento comportamental

Esse tipo de modelamento expressa apenas a funcionalidade do circuito. É uma descrição (figura 15) mais concisa e enxuta, geralmente de fácil escrita, compreensão e manutenção, uma vez que se apresenta como uma descrição algorítmica do problema a ser modelado. É também menos suscetível a erros, possui um ciclo de desenvolvimento menor e maior nível de independência da tecnologia. Essa modelagem dá maior flexibilidade ao sintetizador, por não especificar como devem ser implementadas as construções.

```
ENTITY mux IS
  PORT (d0, d1, sel: IN bit; q: OUT bit);
END MUX;

ARCHITECTURE comportamento OF mux IS
BEGIN
  WITH sel SELECT
    q <= d0 WHEN '0',
         d1 WHEN '1';
END comportamento;
```

Figura 15: Exemplo de modelo comportamental

4.2.2. Modelamento estrutural

É o equivalente a uma versão escrita da representação (figura 16) por esquemáticos e descreve o componente como uma série de interconexões entre sub-componentes, e os componentes de

mais baixo nível na hierarquia são descritos em termos de portas lógicas. Os modelos do projeto em geral são maiores, mais numerosos e mais suscetíveis a erros, mas têm a vantagem de estarem praticamente sintetizados, já descritos em componentes primitivos da implementação. Além disso, estão mais distantes da definição conceitual e mais próximos da tecnologia. Um modelo estrutural assemelha-se a um *netlist* (lista de interconexão de componentes primitivos).

```
ENTITY mux IS
   PORT (d0, d1, sel: IN bit; q: OUT bit);
END mux;

ARCHITECTURE estrutura OF mux IS
   COMPONENT and2
      PORT (a, b: IN bit; z : OUT bit);
   END COMPONENT;
   COMPONENT or2
      PORT (a, b: IN bit; z : OUT bit);
   END COMPONENT;
   COMPONENT inv
      PORT (i: IN bit; z: OUT bit);
   END COMPONENT;

   SIGNAL aa, ab, nsel, bit;
   FOR u1: inv USE ENTITY invrt(comp);
   FOR u2, u3: and2 USE ENTITY and_gt(comp);
   FOR u4: or2 USE ENTITY or_gt(arq);
BEGIN
   u1: inv PORT MAP (sel, nsel);
   u2: and2 PORT MAP (nsel, d1, ab);
   u3: and2 PORT MAP (d0, sel, aa);
   u4: or2 PORT MAP (aa, ab, q);
END estrutura;
```

Figura 16: Exemplo de modelo estrutural

4.3. Síntese de alto nível

Inicialmente a utilização da VHDL restringia-se ao modelamento e à simulação. Com a difusão do seu uso, a síntese também fez uso da VHDL [14]. Contudo, os modelos criados são descritos em alto nível de abstração, conforme o modelamento comportamental. A **Síntese de Alto Nível** (HLS - *High-Level Synthesis*) [13] é um método automático de conversão de uma descrição comportamental para uma descrição a nível de portas lógicas.

As tarefas envolvidas na HLS estão divididas em várias etapas [13]:

a) Compilação da descrição comportamental em uma representação interna. Junto à compilação também pode ser realizada uma otimização/simplificação da representação.

b) Escalonamento das operações, associando cada operação a uma fatia de tempo, denomi-

nada passo de controle.

c) Alocação e acoplamento das operações, definindo a estrutura do hardware e a função de cada uma de suas partes.

d) Particionamento do projeto, a fim de melhorar sua qualidade e desempenho, facilitar a manutenção, e auxiliar o processamento nas etapas posteriores, como a síntese lógica.

Dentro da representação de Gajski (figura 17) para o projeto de circuitos digitais, a HLS situa-se onde indica a seta. Esta representação considera duas dimensões como nas coordenadas polares:

a) Quando se percorre o gráfico do exterior para o centro, diminui-se o nível de abstração da descrição: arquitetural (processadores, memórias e barramentos), RT - R*egister Transfer* (somadores, registradores, etc), lógico (portas lógicas, flip-flops) e dispositivos (transistores, capacitores e resistores).

b) Ao longo dos círculos concêntricos, circundando o eixo central estão dispostos os possíveis domínios da abstração: comportamental (abrange as HDLs), estrutural (correspondente aos netlists) e físico (layout do circuito)

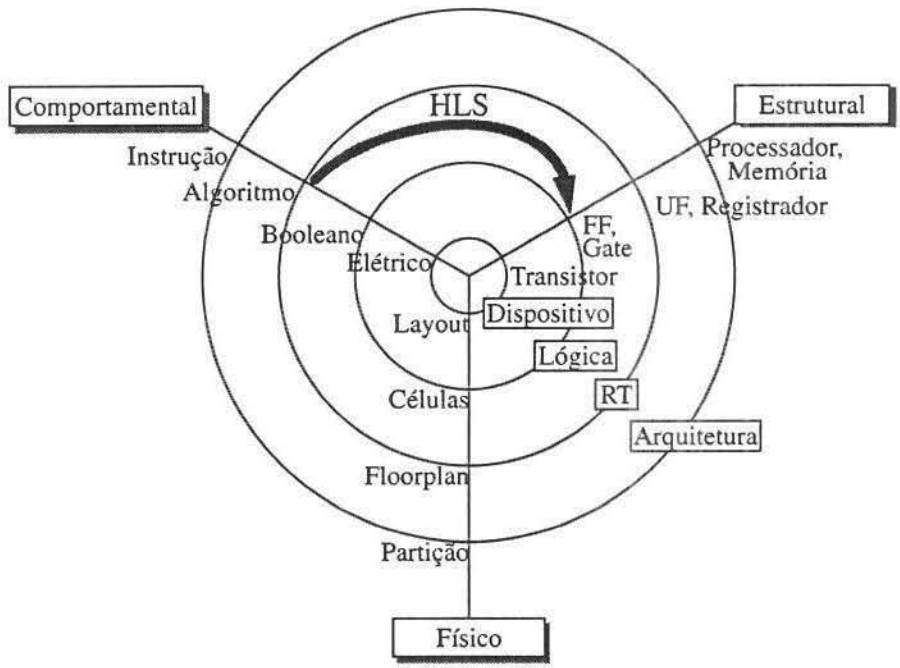


Figura 17: Representação do Projeto de Circuitos (Diagrama-Y)

Uma ferramenta de HLS deve ser capaz de gerar a representação lógica do circuito, um *netlist* genérico, independente da tecnologia de implementação. A qualidade do netlist e seu desempenho na etapa de mapeamento são dependentes do estilo de modelamento e codificação utilizados e da ferramenta de síntese empregada.

É importante mencionar que a portabilidade dos modelos VHDL fica comprometida quando direcionados à síntese. Na HLS, as ferramentas utilizam apenas um subconjunto da VHDL, restringindo as opções disponíveis para o modelamento. Em alguns casos, é preciso reprojetar ou reescrever os modelos para submete-los às ferramentas. Além disso, as ferramentas, de uma forma geral, têm dificuldades de manusear as construções VHDL de maneira eficiente e fazem transformações e otimizações nos circuitos que ficam aquém das que poderiam ser feitas manualmente por um projetista experiente. Fornecedores diferentes também podem implementar semânticas diferentes para a mesma construção.

Existem muitos problemas relacionados à síntese, mas três pontos importantes ainda a distinguem como um método de projeto atraente:

a) hoje em dia, são desenvolvidos projetos muito complexos que, sem a síntese, demorariam muitos meses e requisitariam uma equipe altamente especializada;

b) apesar do resultado não ser muito eficiente, é adequado ao que se presta e facilmente redirecionável a outras tecnologias;

c) podem ser combinadas várias abordagens de modelamento para que se obtenha um resultado mais vantajoso;

d) a rapidez alcançada no desenvolvimento permite testar várias alternativas num projeto.

4.4. Modelamento direcionado à síntese

Não é simples identificar o estilo de modelamento que produz o melhor resultado na síntese. Modelos estruturais costumam apresentar melhores resultados, mas limitam a otimização (por desempenho ou área). Modelos comportamentais resultam em netlists maiores e mais lentos que os feitos manualmente. Sendo assim, aparecem algumas dúvidas quanto ao melhor estilo de codificação, entre outras que podem ser citadas:

a) usar modelos comportamentais ou estruturais ?

b) codificar através de programação concorrente ou seqüencial ?

c) especificar condicionais através de IF-THEN-ELSE ou CASE-WHEN-END ?

Como já mencionado, conforme a ferramenta, certas construções VHDL são ou não aceitas para a síntese. Outra questão variável de uma ferramenta para outra é a otimização do circuito. Dado um modelo e a sua interpretação lógica, é possível otimizá-lo por área, a fim de se economizar recursos (portas lógicas disponíveis, blocos lógicos, blocos de roteamento), ou por desempenho, buscando o ajuste às restrições de freqüência e atraso do sistema. Algumas características inerentes ao modelamento devem ser observadas para evitar-se efeitos indesejáveis no circuito sintetizado ou futuros problemas de projeto, como dificuldade de manutenção.

4.4.1. Lógica Combinacional e Seqüencial

Quando temos uma variável num código-fonte escrito numa linguagem de programação qualquer, se o valor dessa variável não for atualizado numa dada subrotina, o valor antigo (seja o da inicialização ou não) continuará armazenado na memória. Agora, se o mesmo ocorre com um sinal dentro de um modelo VHDL, o valor antigo também deve ser mantido. O único mecanismo que existe para a retenção desse valor é a implementação de uma lógica de estados ou seqüencial para que o valor seja armazenado até o próximo evento que dispare aquela atribuição.

Entretanto, a lógica seqüencial pode ser sintetizada em partes do circuito que deveriam ser estritamente combinacionais, caso não se tome o devido cuidado durante a codificação. Esse problema ocorre na atribuição de sinais dentro de um processo. Um processo, em VHDL, é uma estrutura onde todos os comandos que a compõem são executados sequencialmente.

Levando em consideração o fluxograma de execução de um processo qualquer, existem distintos caminhos lógicos que podem ser percorridos durante sua execução, de acordo com as condições ocorridas. Se em algum desses caminhos lógicos não há modificação de um dado sinal de saída, e este sinal não possui um valor padrão dentro do processo, o sintetizador assume que essa saída deve manter o seu valor e cria um latch.

```
ENTITY dec2x4 IS
PORT (sel: IN bit_vector (1 DOWNTO 0);
      en: IN bit;
      y: OUT bit_vector (3 DOWNTO 0));
END dec2x4;

ARCHITECTURE logica OF dec2x4 IS
BEGIN
  PROCESS (sel, en)
  BEGIN
    IF (en='1') THEN
      CASE sel IS
        WHEN "00" => y(0) <= '0';
        WHEN "01" => y(1) <= '0';
        WHEN "10" => y(0) <= '0';
        WHEN "11" => y(0) <= '0';
      END CASE;
    END IF;
  END PROCESS;
END logica;
```

(a)

```
ENTITY dec2x4 IS
PORT (sel: IN bit_vector (1 DOWNTO 0);
      en: IN bit;
      y: OUT bit_vector (3 DOWNTO 0));
END dec2x4;

ARCHITECTURE logica OF dec2x4 IS
BEGIN
  PROCESS (sel, en)
  BEGIN
    y <= "1111";
    IF (en='1') THEN
      CASE sel IS
        WHEN "00" => y(0) <= '0';
        WHEN "01" => y(1) <= '0';
        WHEN "10" => y(0) <= '0';
        WHEN "11" => y(0) <= '0';
      END CASE;
    END IF;
  END PROCESS;
END logica;
```

(b)

Figura 18: Modelo VHDL de um decodificador 2x4

Esta idéia pode ser compreendida melhor através de dois exemplos: um demonstra uma situação onde o latch é indesejável e outro, onde é imprescindível. O primeiro exemplo (figura 18) descreve um decodificador 2x4. Neste caso (figura 18a), a saída y não tem um valor padrão e quando en for igual a 0, y não terá atribuição de valor; portanto, um latch em y é criado durante a síntese. A figura 18b apresenta a versão devidamente corrigida do modelo. O segundo exemplo (figura 19) é a descrição de um flip-flop. Nesse tipo de componente, a omissão do valor padrão é proposital para que ocorra a geração do latch.

```
ARCHITECTURE logica OF flipflop IS
BEGIN
   PROCESS (Clk) BEGIN
      IF (Clk='1') THEN
         Saida <= Entrada;
      END IF;
   END PROCESS;
END;
```

Figura 19: Modelo VHDL de um flip-flop

4.4.2. Sinais e Variáveis

Tanto sinais quanto variáveis servem para armazenar valores em modelos VHDL. No entanto, existe uma distinção prática que deve ser levada em conta. Um sinal é um objeto que possui uma história de valores e eventos associados a ele. Durante a simulação, um grupo de atributos associados ao sinal são configurados e recebem novos valores a cada iteração. Uma variável é um elemento estático no tempo e sem atributos, armazena o seu valor atual, de maneira instantânea, e nada mais.

As variáveis só podem ser definidas dentro de blocos seqüenciais, como sub-programas (procedimentos e funções) e processos e, portanto, somente esses podem atribuir valores a elas e modificá-las. Além da atribuição convencional de valor a um sinal (figura 20), existe também a atribuição com atraso, especificado em unidades de tempo (normalmente em ns). Na modelagem direcionada à simulação, as variáveis devem ter preferência sobre os sinais, pois causam menor *overhead* e economizam memória, uma vez que sinais necessitam armazenar todos os eventos relacionados a eles.

```
SINAL_A <= '1';                          VAR_A := '1';
SINAL_B <= '0' AFTER 10 ns;

          (a)                                    (b)
```

Figura 20: Exemplos de atribuição a sinal (a) e a variável (b)

4.4.3. Tipos de dados

O pacote padrão std_logic_1164 (padrão IEEE [31]) define um tipo de dados constituído de 9 estados lógicos essenciais para uma simulação de alta precisão. Desses 9, apenas 4 valores possuem

efeitos dentro da síntese: '1' (um lógico), '0' (zero lógico), 'Z' (alta impedância) e '-' (*don't care*), pois somente esses afetam as tabelas verdade que representam o sistema. A utilização de tipos de dados `std_logic` e `std_logic_vector` aumenta a portabilidade do código, por serem um padrão, e são mais eficientes que o tipo `bit`, padrão da VHDL.

4.4.4. Hierarquia de modelos

Alguns estudos [20] tratam da importância da hierarquia de componentes nos modelos e indicam a hierarquização como fator influente na otimização por desempenho e por área. Essas análises vão de encontro à forma de projeto *top-down* utilizada em muitos casos, uma vez que a hierarquia de projeto mais adequada ao entendimento do circuito por parte dos projetistas pode não ser a melhor do ponto de vista da síntese. Projetos grandes com hierarquia "achatada" (*flattened* - com poucos níveis de detalhamento) tendem a ser difíceis de rotear pela alta concentração de estruturas lógicas numa mesma região do componente. Os projetos muito particionados podem não ajustar-se ao componente desejado, pois não possibilitam uma adequada otimização do espaço físico. A experiência sugere a busca de um meio termo, reagrupando a hierarquia e dividindo o projeto em blocos de tamanho "médio", cuja medida varia de tecnologia para tecnologia.

4.4.5. Bibliotecas de componentes

Com a síntese de alto nível, o custo de redirecionamento para outra tecnologia é pequeno. Em muitos casos, basta a aquisição das bibliotecas que descrevem a nova tecnologia. Estas bibliotecas facilitam a portabilidade dos modelos entre ferramentas distintas, pois a utilização de componentes pré-definidos nestas bibliotecas permite a redução e evita a reescrita do código-fonte. Os componentes definidos nelas têm sua estrutura definida de acordo com as possibilidades da ferramenta de síntese e com base nos recursos disponíveis na tecnologia.

4.5. Conclusão

O projeto automatizado sofreu um grande impulso com o surgimento das HDLs. Nenhuma HDL cobre eficientemente todo o ciclo de projeto e cada uma dá ênfase a uma determinada fase, seja ela validação, verificação ou ainda a síntese. Duas das principais HDLs do mercado são a Verilog e a VHDL. Esse capítulo apresentou algumas caracterizações sobre as HDLs, mostrou uma visão geral sobre a VHDL e os diferentes tipos de modelamento existentes, além de apresentar algumas noções sobre a síntese de alto nível e considerações importantes no modelamento comportamental direcionado à síntese. Outras informações sobre VHDL e modelamento comportamental podem ser encontradas em [18] [42] [50] [51] [54] [59] [60].

5. COMPONENTES FPGA

Os FPGAs (*Field Programmable Gate Arrays*) são dispositivos programáveis que permitem a implementação de uma extensa gama de circuitos. Alguns componentes atingem uma densidade de até 100.000 gates equivalentes numa única pastilha, viabilizando o desenvolvimento de circuitos de baixa e média complexidade. A grande vantagem destes dispositivos está na facilidade de programação "na bancada" (*field*). Isto significa que o próprio projetista do sistema tem a possibilidade de mapeá-lo em hardware em seu ambiente de trabalho, utilizando-se de recursos de custo relativamente baixo. Desta forma, FPGAs proporcionam uma grande flexibilidade e permitem uma diminuição considerável nos tempos de projeto e implementação, se comparado com outras técnicas, como *full-custom* e *gate arrays*. Alguns fabricantes utilizam nomes específicos para seus dispositivos programáveis, como os CPLDs (*Complex Programmable Logic Devices*) Altera. Este trabalho utiliza apenas o termo FPGA, por ser bastante difundido.

Aliado ao projeto em alto nível, através de descrições comportamentais em linguagens de descrição de hardware, como a VHDL, o uso dos FPGAs reduz o ciclo de projeto e permite maior garantia da funcionalidade. Para melhor compreensão desta tecnologia, este capítulo apresenta a estrutura básica dos FPGAs (seção 5.1), os tipos de FPGAs existentes e alguns exemplos comerciais (seção 5.2), trata da reconfigurabilidade desses componentes (seção 5.3) e menciona características de mapeamento (seção 5.4).

5.1. Estrutura Básica

Genericamente, os componentes FPGAs possuem uma estrutura matricial (figura 21) composta de 4 elementos básicos:

a) **blocos lógicos elementares (BLEs):** onde são implementadas as funções lógicas do circuito. Conforme o fornecedor, a família e o modelo, variam no número de entradas e saídas e possuem maior ou menor granularidade (compostos por blocos menores). Dentro de um mesmo componente pode haver ainda BLEs distintos e com funções específicas (contadores/somadores, RAMs, operações de ponto flutuante, etc).

b) **blocos de interconexão (BIs):** permitem a implementação de funções lógicas mais complexas com o agrupamento de BLEs e a transmissão de sinais entre os mesmos.

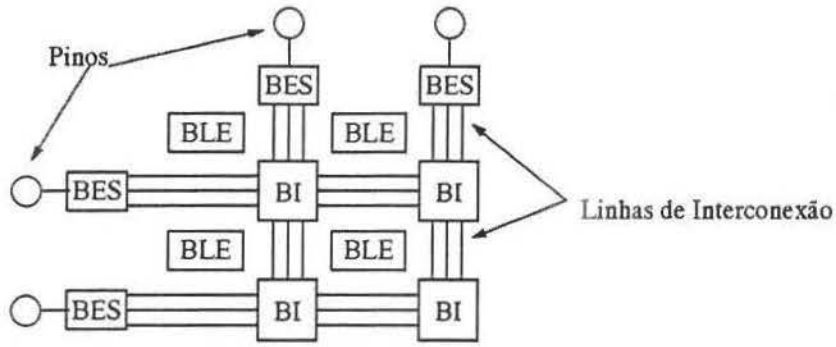


Figura 21: Estrutura de um FPGA

c) **blocos de entrada e saída (BESs):** são conectados diretamente aos pinos do FPGA e realizam a intercomunicação entre a o sistema implementado e o ambiente externo.

d) **linha de interconexão:** fazem a ligação entre os BIs, transportando os sinais ao longo do componente FPGA. Num mesmo componente pode haver vários tipos de linhas de interconexão, hierarquizadas entre si ou não.

5.2. Tipos e exemplos de FPGAs

Atualmente, podem ser encontrados no mercado três tipos de FPGAs classificados de acordo com a tecnologia empregada na programabilidade [8][67]: SRAM (*Static Random Access Memory*), EPROM (*Erasable and Programmable Read Only Memory*) ou EEPROM (*Electrically Erasable and Programmable Read Only Memory*). Os FPGAs baseados em SRAM têm programação volátil, permitindo inclusive reprogramação dinâmica da aplicação [65][66], e conseqüente redução dos custos de hardware.

Entre os fornecedores e famílias de FPGAs há várias classes destes componentes organizadas por número de gates disponíveis, velocidade, voltagem, número de pinos de E/S, encapsulamento, resistência à temperatura entre características. A escolha do componente mais adequado ao circuito é fundamental para o desempenho final.

A família FLEX10K de componentes FPGA da Altera (tabela 3)[8] é um bom exemplo dos FPGAs baseados em SRAM. Seus componentes contém de 10 a 100 mil portas lógicas equivalentes, podem implementar internamente memórias RAM de até 3 Kbytes (24.576 bits). Fazendo um paralelo com a estrutura básica de FPGAs citada anteriormente, esses componentes da Altera, possuem as *Fast-Track*, uma estrutura de roteamento, formada por linhas horizontais e verticais que cruzam o circuito de ponta a ponta, que fazem o serviço das linhas de interconexão.

Nos componentes da família FLEX10K, existem dois tipos de estruturas distintas responsáveis pela implementação de funções no circuito (papel realizado pelos BLEs): os LABs (*Logic Array Blocks*) e os EABs (*Embedded Array Blocks*). Os LABs implementam qualquer tipo de lógica genérica, como contadores, somadores e multiplexadores. Os EABs permitem a implementação de funções de memória e funções especializadas de maior complexidade. Os LABS são, na verdade, um agrupamento de outros elementos mais simples, os LEs (*Logic Elements*). Cada LE possui uma tabela de busca (*look-up-table)* de 4 entradas e 1 saída lógica, e ainda 1 flip-flop interno, que possibilita a implementação de lógicas seqüenciais. Todos os LEs de um mesmo LAB e todos os LABs numa linha estão ligados por "*carry and cascade chains*". Estas "*chains*" permitem a implementação de contadores e somadores (*carry chain*) e funções lógicas (*cascade chain*) mais complexas e com maior número de entradas do que as possíveis com apenas um LE ou um LAB.

Tabela 3: Características dos Componentes da Família FLEX10K

	EPF10K10	EPF10K20	EPF10K30	EPF10K40	EPF10K50	EPF10K70	EPF10K100
Gates Equivalentes	10.000	20.000	30.000	40.000	50.000	70.000	100.000
Gates Utilizáveis	7.000 a 31.000	15.000 a 63.000	22.000 a 69.000	29.000 a 93.000	36.000 a 116.000	46.000 a 118.000	62.000 a 158.000
Logic Elements	576	1.152	1.728	2.304	2.880	3.744	4.992
LABs	72	144	216	288	360	468	624
EABs	3	6	6	8	10	9	12
RAM máxima	6.144	12.288	12.288	16.384	20.480	18.432	24.576
Flip-Flops	720	1.344	1.968	2.576	3.184	4.096	5.392
Max. pinos de E/S	134	189	246	189	310	358	406

Tabela 4: Características dos Componentes da Família FLEX8000

	EPF8282A	EPF8452A	EPF8636A	EPF8820A	EPF81188A	EPF81500A
Gates Utilizáveis	2.500	4.000	6.000	8.000	12.000	16.000
Flip-Flops	282	452	636	820	1188	1500
Logic Elements	208	336	504	672	1.008	1.296
LABs	26	42	63	84	126	162
Tam. Matriz (linhas x colunas)	2 x 13	2 x 21	3 x 21	4 x 21	6 x 21	6 x 27
Max. pinos de E/S	78	120	136	152	184	208

Este projeto utilizou, para estudos de síntese e otimização para componentes FPGAs [1] e para implementação final, a família de FLEX8000, também da Altera. Essa família é muito semelhante à citada anteriormente, FLEX10K, e é base do projeto daquela. A principal diferença entre as duas são

os EABs, adicionados com o propósito de servirem a aplicações mais específicas. Maiores informações sobre os componentes da FLEX8000 podem ser vistos na tabela 4

5.3. Reconfigurabilidade dos Componentes FPGA

Além da classificação dos componentes FPGAs quanto à tecnologia de construção empregada, eles são ainda divididos quanto à sua programabilidade. Essa característica indica se um componente é programável integral ou parcialmente. Explorando essa característica, é possível obter vantagens que outras tecnologias de implementação de circuitos em hardware não permitiriam.

Apesar dos FPGAs possuírem melhor relação custo-benefício quando utilizados na produção de poucas unidades, o uso da flexibilidade de programação melhora ainda mais essa relação. Dessa forma, é possível classificar o projeto de circuitos envolvendo o uso de FPGAs em três **classes de programabilidade**:

a) **projeto estático** - O circuito possui apenas 1 programação, que não é alterada nunca, nem durante e nem após o processamento. Ou seja, o componente é programado, integralmente, para realizar apenas uma função que não é mais alterada durante toda a atividade do sistema. Esse tipo de projeto não explora em nada a flexibilidade de programação, apenas a facilidade para implementação e/ou prototipação.

b) **projeto reconfigurável estaticamente** - Há várias (N) configurações disponíveis para o circuito, entretanto a reprogramação do FPGA ocorre apenas ao final do processamento de uma dada tarefa. Assim, os componentes são melhor aproveitados e o hardware pode ser particionado, visando a reusabilidade de recursos. O FPGA pode ser programado integral (mais comum) ou parcialmente.

c) **projeto reconfigurável dinamicamente** - Nesta modalidade também existem N configurações para o circuito, e a reprogramação ocorre durante a execução. Esse método é conhecido como RTR (*Run-Time Reconfiguration*) [25]. É nesse tipo de aplicação que os FPGAs são utilizados com maior eficiência, extraindo dos dispositivos toda a flexibilidade que eles possuem. A reconfiguração dinâmica possibilitou a concepção de máquinas com conjunto de instruções dinâmicos (DIS - *Dynamic Instruction Set*) [65][66]. O FPGA pode ser programado integral ou parcialmente (mais comum).

Os projetos reconfiguráveis dinamicamente são os que melhor aproveitam todas as características dos FPGAs e influem consideravelmente na relação custo-benefício, uma vez que há um maior número de funções sendo realizadas pelo mesmo hardware. Em geral, esses projetos possibilitam atualizações e implementação de novas funções com maior facilidade do que projetos das outras classes.

A tabela 5 apresenta um resumo dessas classes de programabilidade de projetos, levando em consideração três características básicas: o número de configurações possíveis dentro do projeto, o momento em que é requerida a reprogramação do circuito e o tipo de programação que é aplicada, parcial ou integral, conforme a capacidade do componente utilizado na implementação.

Tabela 5: Resumo das Classes de Programabilidade

Projeto	Configurações Possíveis	Instante de Reprogramação	Tipo de Programação
Estático	1	-	Integral
Reconfigurável Estaticamente	N	Fim da execução	Integral ou Parcial
Reconfigurável Dinamicamente	N	Durante a execução	Integral ou Parcial

5.4. Mapeamento Tecnológico

Uma vez conhecidas as principais características dos componentes FPGA e dos projetos que fazem uso dessa tecnologia, é importante também ressaltar algumas características do mapeamento do sistema. Logo após a síntese do circuito, chega o momento do mapeamento do *netlist* genérico para a tecnologia alvo. Os modelos de alto nível são implementados fisicamente nos componentes, através do uso de bibliotecas pré-definidas pelo fabricante.

O mapeamento realiza três tarefas básicas que influenciam decisivamente no desempenho final do sistema:

a) **particionamento (*partitioning*)**: divisão do circuito em duas ou mais instâncias, a serem mapeadas em diferentes componentes, devido aos recursos limitados de cada um;

b) **posicionamento (*placement*)**: essa tarefa é responsável por ajustar as funções do circuito aos BLEs do FPGA, ou seja, organizar o circuito sintetizado ao longo da estrutura matricial dos componentes;

c) **roteamento (*routing*)**: a cada novo particionamento e/ou posicionamento do circuito, é necessário reprogramar os BIs (blocos de interconexão) e os BESs (blocos de E/S) para conectar e sequenciar os BLEs e os múltiplos componentes do circuito (caso o particionamento tenha estipulado mais de um componente) a fim de manter a sua funcionalidade.

Os problemas relativos ao mapeamento estão intimamente ligados às limitações de recursos dos FPGAs, ao tamanho do circuito sintetizado e à qualidade da síntese. Portanto, independente do esquema de projeto utilizado (seção 5.3) e do tipo de aplicação, é importante possuir ferramentas de síntese, otimização e mapeamento adequadas ao tipo de projeto e à tecnologia de implementação.

5.4.1. Síntese e Mapeamento

Algumas tendências [30] indicam que a síntese e o mapeamento não deveriam ser desacoplados, ou melhor, deveriam determinar, desde o princípio do projeto, qual a tecnologia alvo da implementação. No caso dos FPGAs, a simplificação e particionamento de funções realizados durante a síntese e a otimização, considerando o uso de ferramentas genéricas, nem sempre são a melhor resposta para um perfeito ajuste aos BLEs.

Uma ferramenta de síntese genérica oferece a vantagem da flexibilidade de aplicação, pois permite solucionar uma gama maior de problemas. Entretanto, isto ocorre em detrimento da eficiência da solução obtida. Em alguns casos, considerando-se as restrições e limitações da tecnologia de implementação, a ineficiência pode até inviabilizar a obtenção de uma solução. Ferramentas orientadas a uma classe mais restrita de problemas permitem maior otimização dos resultados, já nos níveis mais altos de abstração, devido a simplificações que podem ser adotadas em função de características particulares da aplicação. Além disso, a eficiência da síntese depende também da estrutura e das características elétricas da tecnologia de implementação.

A escolha de uma certa tecnologia como alvo para o mapeamento final da síntese aumenta a possibilidade de se encontrar **funções custo** mais eficientes para guiar o processo de síntese. As funções custo são procedimentos que nos permitem avaliar o custo de uma determinada solução, o que se reflete sobre as escolhas a serem efetuadas no processo de síntese.

5.4.2. Recursos Limitados

A utilização dos FPGAs como tecnologia alvo da implementação ainda enfrenta o problema da limitada disponibilidade de recursos de roteamento local e global, pois o posicionamento de BLEs e roteamento de redes são muito críticos nos projetos em FPGA. Mohanakrishhnan [48] comenta em seus trabalhos as dificuldades em se utilizar programas de PPR (*Partitioning, Placement and Routing*) automático, onde não foi possível obter 100% do posicionamento de forma automática. As ferramentas, como já mencionado, são de uso genérico, baseadas em métodos heurísticos e nem sempre fornecem o mapeamento ótimo para todos os projetos.

Tais experiências [49] levam a crer que o PPR controlado pelo usuário é a chave para alcançar um alto desempenho. O uso ineficiente dos recursos de roteamento resultam em pinos não roteados e/ou atrasos muito grandes. O atraso de roteamento domina em qualquer projeto em FPGA, devido às numerosas matrizes de chaveamento e pontos de interconexão programáveis ao longo dos cruzamentos dos sinais. Em contrapartida, o PPR manual não é uma tarefa trivial, demanda muito tempo e requer um

conhecimento especializado do projetista, que necessita lidar com um conjunto de recursos de roteamento que, embora limitados, ainda são muito numerosos e possuem diversas configurações possíveis.

5.5. Conclusão

Os componentes FPGAs são uma recente e poderosa tecnologia de implementação de sistemas digitais. Possuem uma estrutura básica bastante simples e abrem margem para novos tipos de aplicações até então inviáveis como as dinamicamente reconfiguráveis. Aliados aos processos de síntese de alto nível, os FPGAs conseguem trazer vantagens em sua utilização na substituição de outras tecnologias. Entretanto diversos aspectos devem ser observados: componentes adequados ao tipo de projeto, limitação de temporização, necessidade de ferramentas de mapeamento eficientes, possível necessidade de intervenção manual no processo de mapeamento, entre outros.

6. ESPECIFICAÇÕES DO NP9

Com o intuito de validar a metodologia de modelamento VHDL e síntese de alto nível para mapeamento em componentes FPGA aqui apresentada, o projeto do NP9 procurou realizar uma implementação do modelo de processador definido por Leite [39] e fazer um estudo das influências da forma de codificação, requisitos de síntese e restrições de otimização no mapeamento final.

O presente capítulo traz os detalhes dessa implementação, apresentando as considerações preliminares (seção 6.1) do projeto que culminaram na escolha da melhor estrutura para o projeto. Além disso, apresenta informações sobre a estrutura geral do NP9 (seção 6.2), sobre os processadores elementares do NP9 (seção 6.3) e um exemplo de sistema de processamento de imagens (seção 6.4) utilizando o NP9. Uma breve discussão sobre o modelamento em VHDL (seção 6.5) encerra esse capítulo.

6.1. Considerações Preliminares

Nessa seção são apresentados os resultados das primeiras análises sobre o projeto do NP9 e os motivos que levaram à implementação do processador como é atualmente. A base para esse estudo foi a análise da escalabilidade (seção 6.1.1) apresentada pelas alternativas e o grau de paralelismo envolvido em cada uma das alternativas levantadas (seção 6.1.2).

6.1.1. Análise de Escalabilidade

Antes de apresentar os motivos que conduziram a implementar o NP9 da forma atual, é necessário esclarecer o que vem a ser escalabilidade. Segundo Hwang [28][29], escalabilidade é um conceito multi-dimensional, abrangendo características como recursos, tecnologia e aplicações. Um sistema de computação é dito escalável se ele pode ser expandido para alcançar maior desempenho e suprir a demanda dos usuários ou reduzido para melhorar a relação custo-desempenho ou adaptar-se a condições especiais de processamento (como no caso de sistemas utilizados em satélites). Existem três níveis de escalabilidade a serem explorados: de recursos, de tecnologia e de aplicação.

A **escalabilidade de recursos** diz respeito ao ganho em desempenho e/ou funcionalidade do sistema através do aumento de suas características funcionais: número de processadores, investimento em armazenamento (memória cache, memória principal, discos) e melhoria das aplicações de software.

A **escalabilidade de tecnologia** está ligada à adaptabilidade do sistema às mudanças na tecnologia. Este fator é um pouco mais abrangente e está dividido em três sub-classes: escalabilidade de

geração, heterogeneidade e escalabilidade espacial. O sistema é dito escalável em geração se, ao atualizar alguma de suas partes por uma nova tecnologia, o restante continua funcionando corretamente. A heterogeneidade é a característica do sistema que o permite operar em harmonia possuindo componentes de fornecedores distintos. Já a escalabilidade espacial tem a ver com a necessidade de se utilizar o mesmo sistema em diversos ambientes, obrigando o mesmo a ter suas proporções físicas reduzidas ou expandidas.

Entre todas a **escalabilidade de aplicação** é a que está mais ligada ao software. Segundo ela, o programa deve ser construído de forma a melhorar seu desempenho e utilizar eficientemente os recursos do sistema de computação quando esse é expandido. Esse nível de escalabilidade não é interessante ao estudo aqui apresentado, uma vez que a preocupação fundamental é uma arquitetura algorítmica [63], ajustada ao processamento em questão.

6.1.2. Alternativas de Projeto

Uma análise introdutória identificou duas diferentes implementações possíveis para a arquitetura proposta. A primeira alternativa (figura 22a) considera uma implementação direta da arquitetura proposta e requer a modelagem de uma matriz de N^2 processadores elementares (estrutura quadrada), onde N representa o número de pixels da vizinhança - 3x3, no caso do NP9. A segunda alternativa (figura 22b) é uma implementação simplificada e tenta balancear custo, reconfigurabilidade e velocidade; ela reduz a arquitetura original a apenas uma coluna de N processadores elementares (estrutura linear) e sua estrutura é reprogramada dinamicamente a cada passo do algoritmo.

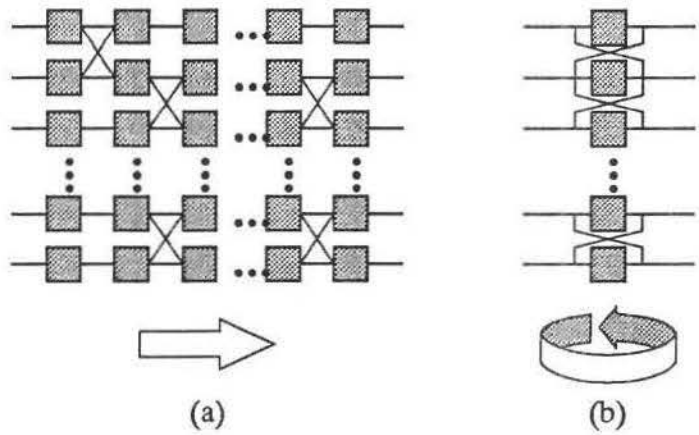


Figura 22: Alternativas de Implementação da Arquitetura do Processador de Vizinhança

A estrutura linear abre maior espaço para a escalabilidade de recursos e para a escalabilidade espacial. Com essa estrutura, é possível ainda simular a matriz de N^2 PEs, reprogramando a estrutura a cada estágio da execução ou conectando N processadores em cascata, a fim de reduzir o tempo de

execução. Há também a flexibilidade de várias configurações possíveis utilizando um número variável de conjuntos de N PEs. A estrutura linear possui outra característica interessante: a reprogramação dinâmica. Com ela, o hardware é utilizado de forma mais eficiente, melhorando o custo-benefício; mas sofre uma sobrecarga muito alta no desempenho devido ao tempo de reprogramação.

Já a estrutura quadrada permite explorar o *pipelining* [28] e alcançar melhores tempos de execução, pois evita a reconfiguração da estrutura durante o processamento (projeto reconfigurável estaticamente, seção 5.3). Esse é um fator muito importante, pois conforme mencionado na introdução deste trabalho, aplicações de processamento digital de imagens podem requerer uma preocupação maior com o tempo de execução. Além disso, o projeto visa estudar o comportamento dos modelos VHDL através da síntese de alto nível, não há uma preocupação direta com o custo. Uma estrutura maior possibilita explorar mais amplamente os processos de síntese e mapeamento e ter estimativas mais precisas quanto ao desempenho em aplicações reais.

Sendo assim, a estrutura escolhida para a implementação é a de NxN PEs, a ser detalhada no decorrer desse capítulo. Analisando as duas alternativas através da fórmula de Danielsson (equação 1, seção 2.1.1), a estrutura quadrada tem um grau de paralelismo 9 vezes maior que a estrutura linear.

Estrutura Linear	Estrutura Quadrada
$k_o = 1, k_i = 1$	$k_o = 1, k_i = 9$
$k_v = 9, k_p = n^*$	$k_v = 9, k_p = n^*$
$K = 1 \cdot 1 \cdot 9 \cdot n = 9n$	$K = 1 \cdot 9 \cdot 9 \cdot n = 81n$

* n é o número de bits do pixel, um fator dependente da implementação e igual para as duas alternativas

Figura 23: Comparação do grau de paralelismo das alternativas de implementação

6.2. Estrutura Geral do NP9

A estrutura geral do NP9 (figura 24), em alto nível de abstração, tem 3 estruturas principais:

a) **Registrador de Programação (PrgReg):** é responsável por armazenar o vetor de programação do NP9. O registrador de deslocamento completo possui 410 bits e contém 5 bits para o multiplexador de seleção da saída e 405 bits para a programação dos processadores elementares do pipeline, num total de 81 PEs com 5 bits para cada um.

b) **Pipeline:** É a estrutura de processamento do NP9. Seus 81 processadores elementares, organizados numa matriz de tamanho 9x9, executam o algoritmo previamente carregado no registrador de programação.

c) **Mux de Saída:** seleciona uma entre as nove saídas do pipeline e faz ou não a inversão do valor final do pixel, conforme o algoritmo que esteja sendo executado.

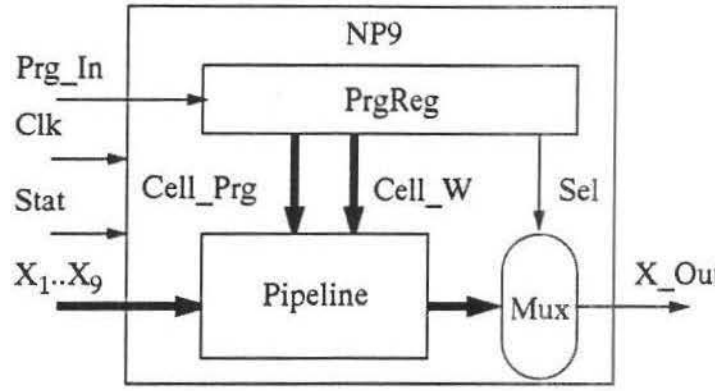


Figura 24: Estrutura do NP9

Na interface externa do NP9 existem 12 entradas e 1 saída:

a) 9 pixels de entrada (X_1 a X_9);

b) 1 sinal de clock (Clk);

c) 1 canal de programação (Prg_In);

d) 1 sinalizador de estado do processador (Stat);

e) 1 pixel de saída (X_Out).

6.2.1. Funcionamento do NP9

O NP9 possui dois estados de operação possíveis: programação (PROG) ou processamento (EXEC), indicados pelo sinalizador de estado (Stat). Na fase de programação, o NP9 recebe pelo canal de programação os valores correspondentes às funções (Cell_Prg) a serem executadas pelos processadores elementares, os pesos (Cell_W) de entrada e a seleção (Sel) de saída e armazena-as num registrador de deslocamento (PrgReg). Quando em processamento, o pipeline executa o algoritmo previamente programado.

O NP9 não realiza operações de E/S ou de controle; toda a programação provém do ambiente externo e os dados para o processamento (pixels de entrada, X_1 a X_9) também são fornecidos através de sua interface. Na sua versão atual, o NP9 opera com pixels de 8 bits, podendo processar imagens em até 256 tons de cinza. Para melhor compreensão do NP9, as próximas seções detalham sua estrutura e o funcionamento das processadores elementares.

6.3. Os processadores elementares

As unidades básicas de composição do NP9 são os processadores elementares; por este motivo, o projeto dos mesmos é um fator crítico dentro da implementação. Um estudo realizado durante o projeto [1] verificou o melhor tipo de modelamento para os processadores elementares do NP9.

Figura 25: Estrutura de operação da processador elementar do NP9

A operação das processadores elementares do NP9 está dividida em duas partes (figura 25): quantificação e qualificação. Na etapa de quantificação, os pixels de entrada são multiplicados pelos respectivos pesos, gerando novos valores que são passados à fase de qualificação, representada pela UCA (Unidade de Comparação e Adição). Os pesos de entrada trabalham apenas com valores inteiros, na faixa de -1 a 1, e permitem ampliar o número de funções lógicas realizáveis pelo processador elementar, originalmente duas (máximo e adição). Os processadores elementares são estruturados para trabalhar com valores de 9 bits (1 bit de sinal) para os pixels de entrada e saída.

6.3.1. O Multiplicador

O multiplicador é o elemento básico da quantificação. Tem uma estrutura (figura 26a) bastante simples, trabalha apenas com 2 entradas (X_in e W) e 1 saída (X_out). Nessa versão do NP9, o peso de entrada (W) só pode assumir três valores: -1, 0 e 1. Portanto, decidiu-se implementar o multiplicador (figura 26b) através do acoplamento de um circuito gerador de complemento de dois (GC2) e um multiplexador 3x1.

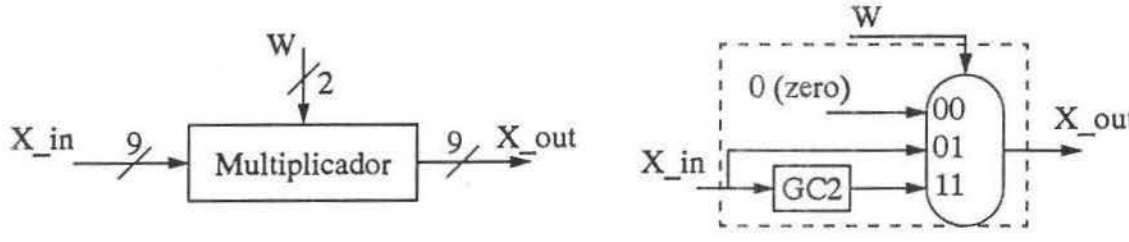


Figura 26: Interface (a) e Estrutura Interna do Multiplicador (b)

6.3.2. UCA (Unidade de Comparação e Adição)

A interface da UCA possui 3 entradas (Op, X_1 e X_2) e 1 saída (Y). Responsável pela etapa de qualificação, a UCA executa duas operações, máximo (MAX) e adição (ADD).

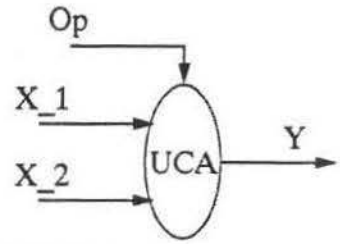


Figura 27: Unidade de Comparação e Adição

6.3.3. Programação dos Processadores Elementares

Cada PE recebe uma programação composta de 5 bits: 2 bits para cada um dos dois pesos de entrada e 1 bit para seleção da função do PE. Com esta estrutura de programação é possível implementar 18 operações, conforme apresenta a tabela 6, que podem ser reduzidas a 16, caso sejam feitas simplificações. A operações 1 e 10 fornecem sempre o mesmo resultado; e a operação 9, soma dos negativos dos pixels, pode ser eliminada e implementada como o negativo da soma dos pixels.

Tabela 6: Funções realizáveis pelas processadores elementares do NP9

Op	Peso 1	Peso 2	Saída	#	Op	Peso 1	Peso 2	Saída	#
ADD	0	0	Zero	1	MAX	0	0	Zero	10
	0	1	X2	2		0	1	Max(0,X2)	11
	0	-1	- X2	3		0	-1	Max(0,-X2)	12
	1	0	X1	4		1	0	Max(X1,0)	13
	1	1	X1 + X2	5		1	1	Max(X1,X2)	14
	1	-1	X1 - X2	6		1	-1	Max(X1,-X2)	15
	-1	0	- X1	7		-1	0	Max(-X1,0)	16
	-1	1	X2 - X1	8		-1	1	Max(-X1,X2)	17
	-1	-1	- (X1 + X2)	9		-1	-1	Max(-X1,-X2)	18

Essa simplificação de operações indica uma implementação de PEs com programação de apenas 4 bits (16 possibilidades). Entretanto, optamos por implementá-la com 5 bits para evitar a inclusão de uma nova estrutura apenas para decodificação da programação, pois a mesma acarretaria um possível aumento no tempo de execução e ocuparia preciosos recursos dos componentes da implementação.

6.4. Organização de um Sistema de Processamento de Imagem

A concepção e futura implementação de um Sistema de Processamento de Imagens são objetivos decorrentes do projeto NP9. O NP9 não executa funções de controle ou E/S. Sendo assim, não há como fazer uma verdadeira análise de seu desempenho em aplicações reais sem a implementação de tal sistema, que permite simulação e execução em ambiente real de hardware. A princípio, o sistema apresenta a organização da figura 28.

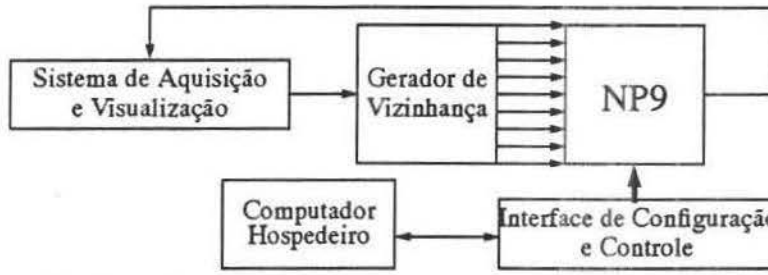


Figura 28: Organização de um Sistema de Processamento de Imagem

Nessa arquitetura, o NP9 é conectado a um sistema de aquisição de imagens e visualização por intermédio de um gerador de vizinhança, responsável por receber os dados na forma serial, linha a linha da imagem, e enviar ao NP9 a vizinhança a ser processada, conforme pode ser visto na figura 29. A interface de configuração e controle liga o NP9 ao computador hospedeiro, e é responsável pela programação e controle do processador, sinalizando o estado de operação do mesmo.

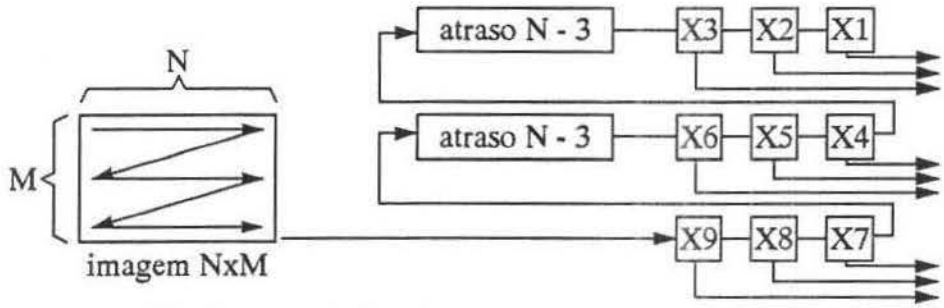


Figura 29: Esquema de funcionamento do gerador de vizinhança

O ideal é que o NP9 alcance um desempenho que permita trabalhar com sistemas de tempo real. Um sistema típico destes trabalha com dispositivos que transmitem cerca de 30 quadros (*frames*) por segundo, onde cada imagem (com resolução de 1024x1024 pixels) deve ser processada em 33,3 ms e a taxa de transmissão fica em torno de 30 Mbytes/s.

A validação da funcionalidade desse sistema foi toda feita através da simulação de modelos VHDL, realizando o processamento de uma imagem, como num ambiente real de hardware. Os resulta-

dos foram satisfatórios e o sistema conseguiu atingir sua funcionalidade, executando os algoritmos especificados; os detalhes da simulação estão no apêndice B.

6.5. Modelamento em VHDL do NP9

Utilizando um alto nível de abstração, com descrições basicamente comportamentais, foi possível modelar o NP9 e seus componentes de uma forma bastante concisa, clara e eficiente. A versão final do projeto consta de 1 biblioteca (*package*) com a definição de alguns tipos e estruturas de dados utilizadas e 3 modelos representando as principais estruturas hierárquicas do NP9: a processador elementar (célula), o pipeline e o processador propriamente dito, o modelo raiz da implementação. Todos os modelos constantes do projeto estão disponíveis no apêndice A.

A biblioteca np9_pkg define 3 tipos básicos a serem usados no projeto: o tipo `PIXEL_INT` (`INTEGER range 0 to 255`), para os pixels de entrada e saída do processador, o tipo `PIXEL_INT_S` (`INTEGER range -255 to 255`), para os pixels de entrada e saída das processadores elementares, e o tipo `PESO_S`, para os pesos de entrada dos pixels. Também estão definidas estruturas vetoriais envolvendo os tipos `PIXEL_INT` e `PIXEL_INT_S`.

```
entity CELULA_HL is
   port ( X1, X2 : in  PIXEL_INT_S;
          W1, W2 : in  PESO_S;
          OPC    : in  STD_LOGIC;
          CLK    : in  STD_LOGIC;
          Y      : out PIXEL_INT_S );
end CELULA_HL;
```

Figura 30: Descrição VHDL da entidade `CELULA_HL`

A entidade `CELULA_HL`, componente que representa as processadores elementares do NP9, tem sua estrutura definida como na figura 30. As operações de multiplicação, adição e máximo são efetuadas por funções internas ao corpo da arquitetura. Em `CELULA_HL` existe ainda um registrador interno de 9 bits para armazenar o resultado final da operação do processador elementar. O conjunto de registradores dos PEs de um estágio do pipeline atuam como um *interstage*, estrutura que possibilita ao pipeline operar em modo sistólico, armazenando os resultados de um estágio a cada pulso de clock.

```
entity PIPELINE_HL is
port ( X   : in  PIXEL_INT_VEC ( 1 to 9) ;
       PRG : in  STD_LOGIC_VECTOR ( 1 to 405 );
       CLK : in  STD_LOGIC;
       Y   : out PIXEL_INT_S_VEC ( 1 to 9) );
end PIPELINE_HL;
```

Figura 31: Descrição VHDL da entidade `PIPELINE_HL`

A entidade PIPELINE_HL é onde está descrito o grafo de fluxo de dados entre os PEs do processador. Um pequeno algoritmo e duas funções de indexação permitem um correto tráfego dos valores dos pixels e da programação dos processadores elementares. Essa entidade foi criada com o intuito de modularizar e simplificar a compreensão da estrutura do NP9. Uma implementação concisa do pipeline como essa, só é possível através do comando GENERATE existente na VHDL. Ao contrário da versão anterior, o AutoLogic II suporta o comando GENERATE, permitindo uma codificação menos extensa e mais simples, conforme pode ser visto na figura 32, na modelagem da arquitetura do pipeline.

```
P1: for COL in 1 to 9 generate
  P2: for LIN in 1 to 9 generate
    CEL: CELULA_HL port map
      ( X_IN(COL, LIN), X_IN(COL, INDICE(COL, LIN)),
      PRG(INDPRG(COL, LIN)+0 to INDPRG(COL, LIN)+1),
      PRG(INDPRG(COL, LIN)+2 to INDPRG(COL, LIN)+3),
      PRG(INDPRG(COL, LIN)+4), CLK, X_IN(COL+1, LIN) );
    end generate P2;
end generate P1
```

Figura 32: Detalhe da descrição VHDL da arquitetura do pipeline

O NP9, representado no modelamento pela entidade NP9_HL (figura 33), possui duas estruturas internas, implementadas através de processos seqüenciais: PRGREG (Registrador de Programação) e MUX (Multiplexador de Saída), além de declarar uma instância de PIPELINE_HL.

```
entity NP9_HL is
   port ( X      : in  PIXEL_INT_VEC (1 to 9);
          PRG_IN : in  STD_LOGIC;
          CLK    : in  STD_LOGIC;
          STAT   : in  STD_LOGIC;
          X_OUT  : out PIXEL_INT );
end NP9_HL;
```

Figura 33: Descrição VHDL da entidade NP9_HL

6.6. Conclusão

Um estudo preliminar indicou duas alternativas de implementação para o NP9. A escolha da melhor alternativa foi feita com base na análise de escalabilidade e do grau de paralelismo das duas arquiteturas. A base de toda a arquitetura do NP9 são seus processadores elementares, cujo projeto é um fator crítico na implementação do processador. A implementação, conforme descrita neste capítulo, foi realizada através da síntese de descrições comportamentais em VHDL. O alto nível de abstração conferido por esta linguagem possibilitou a codificação de uma descrição concisa (aproximadamente 270 linhas de código), clara e eficiente do ponto de vista funcional.

7. RESULTADOS EXPERIMENTAIS

A fim de validar a metodologia apresentada e estudar os fatores que alteram os resultados da síntese em um sintetizador de uso geral, o projeto NP9 realizou algumas experiências envolvendo diversos testes de síntese sobre os modelos VHDL. Os estudos de caso apresentados neste capítulo tratam, respectivamente da síntese de processadores elementares do NP9 (seção 7.1) e da síntese do processador NP9 (seção 7.2). Para a experiência, foram utilizadas as ferramentas já apresentadas no capítulo 3, e o mapeamento foi direcionado para componentes FPGA da Altera, mais especificamente para as famílias FLEX8000 e FLEX10K.

7.1. Estudo de Caso 1: Síntese de PEs do NP9

O estudo de caso realizado consta da descrição, síntese e mapeamento para FPGAs Altera de dois modelos diferentes de descrição de um processador elementar do NP9. Compilados no System1076 e sintetizados em alto nível no AutoLogic, do ambiente de projeto de circuitos da Mentor Graphics, os modelos possuem como principais características:

a) **Modelo 1 (celula_bit)** - os tipos de dados criados utilizam o padrão VHDL pré-definido `std_logic_1164`. Os operadores matemáticos e relacionais usados pertencem às extensões do pacote padrão (`std_logic_1164_extensions`). Esse modelo segue uma linha estrutural, onde são instanciados componentes no modelo de mais alto nível na hierarquia. Os componentes definidos foram três: multiplicador (2 instâncias), somador e comparador (1 instância cada um).

b) **Modelo 2 (celula_hl)** - todos os sinais foram definidos com tipos de mais alto nível que no anterior, utilizando o tipo `integer` (pacote `standard` da VHDL) para os pesos e os pixels de entrada e saída e um tipo enumerado para a função do PE. É um modelo mais compacto, sem instanciação de componentes e apenas um nível de hierarquia.

Os dois modelos incluem um registrador na saída da UCA, que faz parte do *interstage e* permite testar o atraso dos elementos de estado integrantes do FPGA. Após o trabalho no ambiente Mentor, os *netlists*, em padrão EDIF, gerados na síntese foram mapeados para FPGAs Altera, com a ferramenta MAX+Plus II, do mesmo fabricante dos FPGAs. O AutoLogic possui duas opções principais de otimização: por área e por desempenho. O estudo de caso explorou as duas opções a fim de verificar os resultados alcançados na tecnologia Altera.

A otimização por área de um circuito produz um novo, com as seguintes características:

a) número de *gates* reduzido ao mínimo suficiente para o correto funcionamento;

b) introdução de macrocélulas criadas pelos fornecedores para executar determinadas funções lógicas com layout otimizado;

c) eliminação da lógica redundante.

A otimização por desempenho visa alcançar a funcionalidade do circuito com o menor tempo de atraso e com o maior clock possível. Na otimização por desempenho, o AutoLogic faz análise de temporização estática, ou seja, a disponibilidade do sinal de saída depende do tempo de chegada dos sinais de entrada e do atraso de cada *gate* no caminho crítico. A temporização ainda é restringida pelo dispositivo escolhido, que possui suas próprias limitações de atraso e freqüência máxima de operação.

7.1.1. Resultados obtidos

A síntese dos modelos utilizou, simultaneamente, as opções de área e desempenho. O mapeamento baseou-se em outras duas escolhas: a seleção automática do componente pela ferramenta (Max+Plus II) e escolha manual do componente. Na seleção automática, a ferramenta optou pelo componente mais simples da família FLEX8000, o EPF8282LC84. Na seleção manual, o circuito foi mapeado num componente EPF8282ALC84, que possui a mesma organização do anterior - número de LEs (*Logic Elements*), LABs (*Logic Array Blocks*), pinos, etc - embora seja bem mais veloz.

No EPF8282LC84, o período mínimo de clock é de cerca de 7,0 ns, no EPF8282ALC84 esse tempo cai para 3,4 ns. Isso ressalta o que foi dito anteriormente, na seção 5.2, sobre a escolha do componente. Além da seleção manual do componente, foram solicitadas algumas otimizações específicas da tecnologia, como redistribuição global de clock e *carry* e *cascade chains* automáticos. A seção 5.2 traz uma explicação acerca dessas opções [8].

A primeira etapa (otimização por área) obteve os resultados vistos na tabela 7:

Tabela 7: Resultados da Síntese com Otimização por Área

Componente	Modelo	Atraso Máximo (ns)	LEs Utilizados	Taxa de Utilização
EPF8282LC84	celula_bit	132,5	94	45%
	celula_hl	126,3	100	48%
EPF8282ALC84	celula_bit	75,8	91	43%
	celula_hl	54,1	84	40%

A segunda etapa (otimização por desempenho) obteve estes resultados:

Tabela 8: Resultados da Síntese com Otimização por Desempenho

Componente	Modelo	Atraso Máximo (ns)	LEs Utilizados	Taxa de Utilização
EPF8282LC84	celula_bit	125,3	104	50%
	celula_hl	113,8	97	46%
EPF8282ALC84	celula_bit	76,6	109	52%
	celula_hl	73,6	109	52%

7.1.2. Análise dos Resultados

Os resultados apresentados neste estudo de caso demonstram a eficiência da ferramenta de síntese no reconhecimento das funções lógicas implementadas, não havendo grandes distinções entre os dois modelos. A utilização de células dos FPGAs foi quase a mesma em todos os casos, com uma redução um pouco mais acentuada na otimização por área utilizando o EPF8282ALC84 e os recursos disponíveis na tecnologia. A otimização por desempenho não apresentou melhora considerável e, em pelo menos um dos casos, piorou o tempo de atraso combinacional. No entanto, a seleção manual do componente influiu decisivamente na redução dos tempos de atraso, obtendo uma redução média de 43,5%.

A forma com que o sintetizador implementou o *lookahead* da UCA do NP9 parece ser a causadora dos altos tempos de atraso. Verificou-se, nos relatórios de análise de temporização da ferramenta de mapeamento, um crescimento no atraso combinacional no sentido do bit mais significativo para o menos significativo da saída UCA. É possível que exista margem para melhoria usando a metodologia descrita; mas, com base nos resultados desta experiência, não há garantias de que esta abordagem alcance um tempo mínimo para aplicações em tempo real (33ns). Isto reforça o fato de que nem sempre a síntese por si só atinge um resultado satisfatório. Outras metodologias podem obter melhores resultados em projetos simples, como o do processador elementar do NP9:

a) síntese lógica de estrutura *top-down*, com reedição do *floorplanning* do circuito mapeado. Esta opção não segue a metodologia adotada neste trabalho e é bastante arriscada, caso não se conheça bem a tecnologia empregada, pois tais alterações podem quebrar a consistência do circuito.

b) realizar todo o projeto em "baixo nível", elaborando todas as células a serem utilizadas já na tecnologia. Essa alternativa está ainda mais longe da metodologia aqui proposta.

c) a terceira opção, de nível mais intermediário, considera o desenvolvimento em alto nível e a elaboração de macrocélulas otimizadas para substituir as partes críticas do circuito. O

modelo de alto nível utiliza estas macrocélulas como seus componentes.

O problema de atrasos da lógica combinacional é mencionado por Lichtermann [41] em seu trabalho com um processador para rotação de imagens, onde usava componentes Actel e Xilinx na implementação do processador. Em seu trabalho, ele optou pelo desenho manual dos somadores, pois a síntese de alto nível excedeu o período de clock máximo para a aplicação (47ns).

7.2. Estudo de Caso 2: Síntese e Mapeamento do NP9

Esta seção contém os resultados de um estudo de caso envolvendo a compilação, síntese e mapeamento dos modelos comportamentais VHDL do NP9 para componentes FPGA. Tal experiência serviu para validar a metodologia apresentada neste trabalho e ter uma noção preliminar do desempenho do NP9 implementado em tais componentes.

7.2.1. Ambiente de Trabalho

O estudo utilizou diferentes plataformas, equipamentos e softwares, a saber:

a) 1 SPARCstation-20, com 1 cpu de 75 MHz, 128 MB de memória real e 390 MB de memória virtual, rodando sistema operacional SunOS 5.5 e ambiente Solaris 2.5;

b) 1 microcomputador Pentium, 90MHz, 48 MB de memória real, IBM-DOS 6.3 e MS-Windows (96 MB de memória virtual);

c) Ferramentas de EDA da Mentor Graphics (Plataforma Unix), já descritas na seção 3.4;

d) Max+Plus II (da Altera) versão 7.0 para Unix e para Windows.

Durante as experiências, outros equipamentos e versões anteriores desses softwares foram utilizados, até que se alcançou esta configuração final, considerada satisfatória dentro das necessidades do projeto. A utilização de duas versões do sistema Max+plus II (Unix e Windows) foi devida a alguns problemas técnicos, mas contribui com a possibilidade de confrontamento dos resultados de mapeamento realizado em diferentes ambientes. Ao final, ambas as versões obtiveram resultados iguais, a despeito do tempo de execução mais elevado na plataforma Windows.

7.2.2. Procedimentos

Os modelos VHDL do NP9 foram compilados com o QuickHDL (Mentor Graphics) com direcionamento para a síntese (opção `-synth`), que realiza uma "pré-síntese" do modelo e checa a validade de algumas construções VHDL. Em seguida, os modelos compilados foram submetidos ao AutoLogic II, ambiente de síntese e otimização de circuitos, que tem como produto final a geração de netlists. A otimização com o AutoLogic II utilizou duas opções que demonstraram melhor resultado em

desempenho e utilização de recursos: "achatamento" de hierarquia (*hierarchy flatten*) e otimização por área, sem fatorização [47].

A opção de hierarquia achatada incorpora os blocos de um dado componente ao componente de maior nível que o contém. Essa opção não deve ser utilizada quando o componente a ser substituído por seus componentes possui uma estrutura já otimizada, com um bom desempenho. A fatorização (*"factoring"*) utiliza técnicas algébricas para eliminar lógica redundante. Por exemplo, em circuitos como somadores, onde tenha sido utilizada alguma estrutura de lookahead, a fatorização irá suprimir essa estrutura, considerada lógica redundante, embora aumente a velocidade do componente.

Na etapa seguinte, o sistema Max+plus II processou os netlists gerados pelo AutoLogic II e obteve os resultados relativos à utilização de recursos e desempenho preliminar, através da análise de temporização. Toda a escolha de componentes foi feita automaticamente, restringindo o conjunto de possibilidades aos dispositivos mais rápidos de cada família. Esse sistema também fornece informação como redes, pinos de entrada e saída, conexões entre BLEs, entre outras.

Devido ao fato da síntese e otimização do NP9 ser um processo bastante demorado (de 2 a 4 horas no AutoLogic II e cerca de 2 horas no sistema Max+Plus II), alguns testes foram realizados com modelos em menor escala, para evitar sucessivas e lentas repetições. Esses modelos possuem a mesma estrutura e organização do NP9, mas com um número menor de entradas, o que resulta numa matriz de processadores elementares de tamanho inferior à original. As instâncias desses modelos variaram de 3 a 7 entradas. Um resultado positivo dessa alternativa de estudo foi a análise do comportamento do mapeador em relação ao aumento gradativo de complexidade do projeto.

7.2.3. Resultados Obtidos

A primeira parte deste estudo de caso envolveu o modelamento de uma instância em menor escala do NP9, mas com as mesmas características de funcionalidade e a mesma organização desse. O modelo possui apenas 3 entradas, resultando numa matriz 3x3 de processadores elementares (figura 34). A princípio, esse modelo não tem aplicação prática nenhuma, servindo apenas para a realização de uma bateria de testes que tomariam um grande tempo se realizadas num circuito do tamanho do NP9.

Na geração dos netlists a partir do AutoLogic II, utilizou-se apenas a otimização por área, em face aos resultados obtidos no estudo de caso anterior, apresentado na seção 7.1.1, e conforme recomenda o guia de interface Mentor x Altera. A otimização foi feita para três níveis de "esforço" diferentes: *low, medium* e *high*. O esforço na otimização por área é o grau de otimização exigido na redução e compactação das estruturas do circuito. Além disso, foram utilizados netlists a partir de modelos com e sem achatamento da arquitetura. Os resultados do mapeamento desses netlists estão na tabela 9.

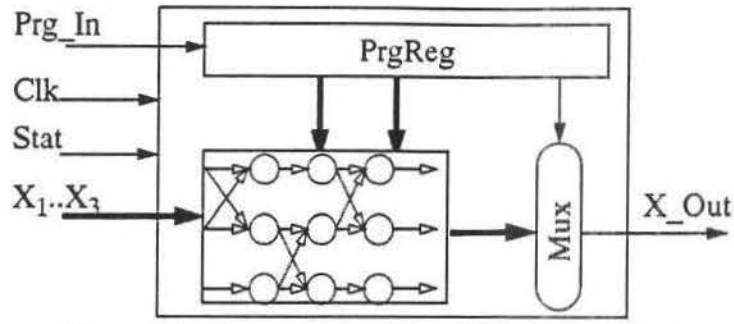


Figura 34: Esquema simplificado do modelo 3x3

Tabela 9: Resultados do mapeamento do modelo 3x3

Esforço de Otimização	Hierarquia Achatada	Freqüência Máxima (MHz)	LEs Utilizados	Taxa de Utilização
LOW	Sim	18,55	799	79%
	Não	17,92	840	83%
MEDIUM	Sim	18,34	785	77%
	Não	17,39	821	81%
HIGH	Sim	17,79	781	77%
	Não	17,54	847	84%

Todos os netlists puderam ser ajustados num mesmo tipo de componente da família FLEX8000: o EPF81188A, que possui cerca de 12.000 gates utilizáveis e 1.008 LEs (*logic elements*). Como resultado esperado, o desempenho geral do circuito decresceu à medida que se utilizou um esforço de otimização mais voltado para a minimização de recursos. O achatamento da hierarquia do circuito contribui positivamente para a melhoria do desempenho.

O próximo passo consistiu em verificar o comportamento do sistema de síntese em presença de um aumento crescente na complexidade do circuito. Todos os modelos foram mapeados em componentes da família FLEX10K. Conforme apresenta a tabela 10, o mapeamento realizado pelo sistema é ineficiente à medida que o circuito aumenta. Durante os testes, o desempenho dos modelos mais complexos acabou por ser pior do que todos aqueles com tamanho inferior. Isso é devido à dificuldade encontrada nos sistemas de mapeamento em gerenciar um número muito grande de variáveis durante o processo de otimização, ocasionando um roteamento de má qualidade, com grande tempo de atraso.

Mohanakrishnan [48][49] já havia comentado sobre essa dificuldade e ressalta a importância da interferência manual do usuário no processo de síntese.

Tabela 10: Resultado do mapeamento das instâncias crescentes do modelo

Tamanho do Modelo	No. de PEs	Freqüência Máxima	LEs Utilizados	Componente(s)	Taxa de Utilização
3x3	9	18,55	799	10K20	79%
4x4	16	18,05	1435	10K30	83%
5x5	25	16,94	2225	10K40	96%
6x6	36	13,58	3180	10K70	84%
7x7	49	12,39	4298	10K100	86%
9x9	81	10,58	6656	10K100/10K100	67%

Por fim, o netlist do NP9 completo foi sintetizado. Os resultados obtidos para esse mapeamento não foram ainda satisfatórios. Numa primeira tentativa, o circuito final foi mapeado em 2 componentes distintos, ambos do modelo FLEX10K100, e utilizou 6.656 LEs, de um total de 9.984 possíveis (total dos dois componentes), e obteve um desempenho de 10,58MHz. Numa segunda tentativa, o mapeamento, agora com restrições de temporização, obteve uma freqüência máxima de 12,34MHz para o circuito (a solicitada foi de 13MHz) e também utilizou 2 componentes: 1 FLEX10K70 (3.744 LEs), onde utilizou 3.135 LEs, e 1 componente FLEX10K100, utilizando 3.391 células desse. Dada a freqüência máxima prevista para o processador, a tabela 11 faz uma relação entre o número de quadros/s, enviados por um dispositivo de aquisição de imagem qualquer, e o tamanho máximo da imagem para a obtenção de uma resposta em tempo real. Os experimentos realizados indicam que esse é o limite de desempenho do circuito seguindo esse estilo de projeto e não há grandes variações que possam ser realizadas sem uma reorganização do projeto e um conseqüente particionamento manual de suas estruturas durante o modelamento inicial.

Tabela 11: Relação Quadro/s x Tamanho da Imagem para freqüência de 12,34 MHz

Número de Quadros/segundo	Tamanho Máximo da Imagem	Total de Pixels da Imagem
10	1110 x 1110	1.232.100
20	785 x 785	616.225
30	641 x 641	410.881
40	555 x 555	308.025

7.2.4. Alternativa para Melhoria de Desempenho

A principal alternativa para melhorar o desempenho do processador é a redefinição do projeto, particionando sua estrutura em blocos menores e ajustar bloco a bloco em cada componente. Isso permite fazer o mapeamento de partes do circuito que possuem complexidade bem inferior à do projeto como um todo. Além disso, é mais fácil detectar os pontos críticos num projeto modularizado e fazer alterações do que manusear uma rede muito grande de componentes.

A figura 35 mostra o projeto particionado em quatro unidades a serem implementadas em separado. O pipeline (figura 24, seção 6.2) foi dividido em 3 estágios, contendo 27 processadores elementares cada um; o registrador de programação foi dividido, e cada estágio possui seu próprio registrador de deslocamento para armazenar o vetor de programação; por fim, o mux de saída foi agregado a uma nova estrutura, denominada Controle, que é responsável por redirecionar o Clock, configurar o status local de cada estágio (LStat) e programar cada um desses estágios.

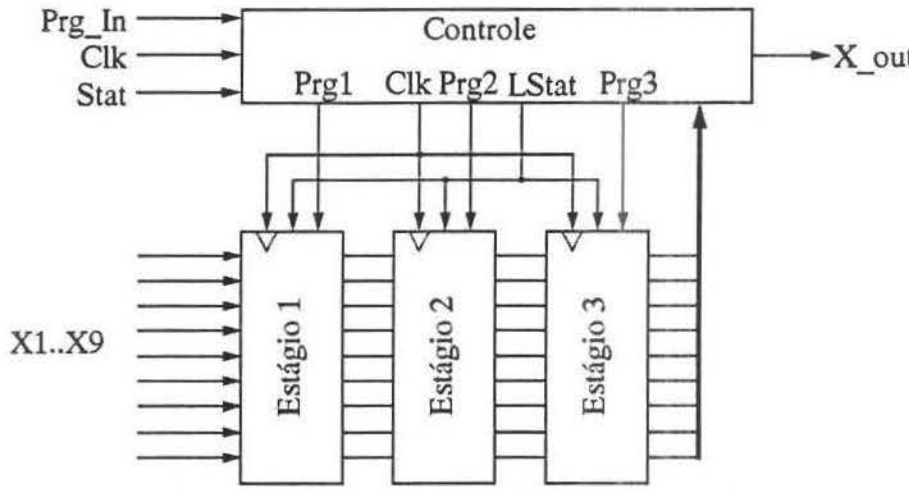


Figura 35: Esquema particionado do NP9

Logo de início, essa estrutura demonstrou algumas vantagens: menor tempo de execução dos processos de síntese e mapeamento, facilidade de manutenção e aumento na freqüência máxima de operação. A estrutura assim definida, alcançou uma freqüência prevista de 17,30 MHz. Cada estágio foi mapeado sobre um FPGA FLEX10K50, utilizando cerca de 2.380 LEs de um total de 2.880 possíveis. A unidade de controle não foi avaliada em suas estatísticas por não representar uma porção crítica do processador, como é o caso do pipeline. Este ainda não é o desempenho ideal para o NP9 trabalhar em tempo real, como esperado no início do projeto, quando se desejava uma desempenho aproximado de 31,5 MHz, o que permitiria processar imagens de 1.024 x 1.024 pixels a uma taxa de 30 quadros/s. Com o desempenho obtido, a relação quadros/s x tamanho da imagem fica como apresentado na tabela 12.

Tabela 12: Relação Quadro/s x Tamanho da Imagem para freqüência de 17,30 MHz

Número de Quadros/segundo	Tamanho Máximo da Imagem	Total de Pixels da Imagem
10	1.315 x 1.315	1.729.225
20	930 x 930	864.900
30	759 x 759	576.081
40	657 x 657	431.649

A figura 36 apresenta o gráfico da relação quadros/s x pixels processados. A curva de 32 MHz é a curva ideal e representa o desempenho do processador trabalhando numa freqüência baseada no processamento de imagens de 1024x1024 pixels a 30 quadros/s. A curva de 17,30 MHz representa o resultado do desempenho do modelo particionado do NP9, e a curva de 12,34 MHz representa o desempenho alcançado no modelo do NP9 sem particionamento da estrutura do pipeline.

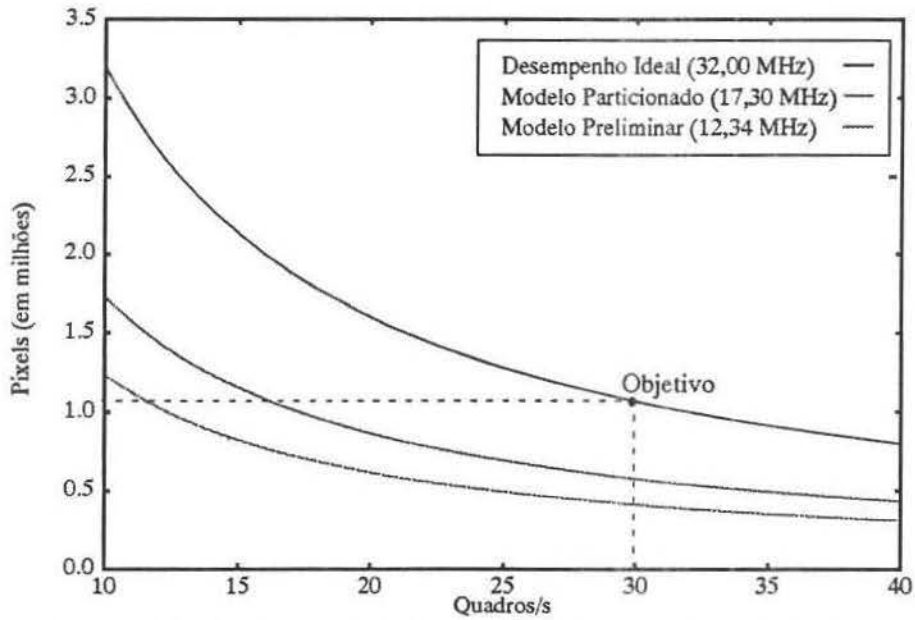


Figura 36: Gráfico da Relação Quadros/s x Pixels Processados

Uma outra alternativa, não explorada neste trabalho, ainda pode melhorar as características do circuito mapeado, tanto em desempenho quanto em área ocupada. Ela consiste, como menciona a seção 7.1.2, em projetar as partes críticas do circuito diretamente sobre as LEs do FPGA. No caso do NP9, a opção mais atraente seria implementar dessa forma os seus PEs, ou ao menos a UCA. A partir daí, essas células passariam a ser componentes pré-definidos, denominados macrocélulas, e poderiam ser instanciados nos modelos de mais alto nível, sem maiores prejuízos para o modelamento.

7.2.5. Conclusão

A experiência adquirida nos estudos de caso apresentados neste capítulo possibilitaram um melhor conhecimento das ferramentas utilizadas no projeto e melhor compreensão dos processos envolvidos no ciclo de projeto de um circuito digital. Também como resultado positivo, pode ser citada a validação da metodologia proposta neste trabalho. O processador NP9, nas primeiras tentativas de implementação, ainda não alcançou o resultado esperado, mas acredita-se que ainda há margens para a melhoria do projeto através de algumas alterações nas especificações iniciais e a combinação da metodologia proposta com alguma outra técnica de projeto, como o projeto de macrocélulas otimizadas. Além dos resultados experimentais relativos à síntese apresentados neste capítulo, o apêndice B apresenta os resultados dos testes de funcionalidade do circuito. Nestes testes, um sistema completo de processamento de imagens utilizando o NP9 foi simulado a partir de sua descrição comportamental em VHDL.

8. CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho foi apresentada uma metodologia de projeto de circuitos digitais baseada no modelamento comportamental em VHDL e síntese de alto nível direcionada a componentes reprogramáveis do tipo FPGA. Como base para a validação dessa metodologia, foi projetado um processador de vizinhança para aplicação em imagens digitais.

Em decorrência disso, o trabalhou apresentou algumas características do processamento de imagem, conceitos aplicados a arquiteturas de processamento de imagens e exemplos de processadores matriciais. Ainda sobre arquiteturas, foi apresentado o processador de vizinhança proposto por Leite [39] e demonstrada a sua funcionalidade, através da implementação de alguns algoritmos de PDI, conforme pode ser visto no apêndice B.

As principais técnicas envolvidas na metodologia, modelamento comportamental e componentes FPGAs também são discutidas. Por fim, o trabalho trata da especificações do projeto do NP9, seu modelamento e dos testes preliminares abrangendo as ferramentas utilizadas no referido projeto e faz uma análise dos resultados desses testes.

A metodologia proposta demonstrou ser bastante eficiente. Seu principal objetivo, o de reduzir o tempo de projeto e possibilitar o estudo de diversas soluções para um mesmo problema, foi cumprido integralmente. O NP9 foi implementado em várias formas, utilizando-se estruturas hierárquicas, modulares e de alto nível.

Quanto ao retorno dos resultados esperados, o baixo desempenho do NP9 demonstrou a existência de um problema mais relacionado às ferramentas utilizadas do que na metodologia propriamente dita. Os componentes FPGA não são a tecnologia mais indicada para aplicações de alta velocidade, pois existe um prejuízo no desempenho para possibilitar a flexibilidade de reprogramação. Há que se considerar a necessidade de ferramentas mais adequadas a cada tipo de aplicação a ser desenvolvida, seja ela processamento de imagens, protocolos de rede, datapaths, coprocessadores numéricos, etc. Outros fatores relevantes para melhoria dos resultados também são a maior integração entre os processos de síntese e mapeamento tecnológico, os quais deveriam estar sempre acoplados, e o estudo de técnicas eficientes de particionamento e modularização de projetos.

A necessidade de reestruturação da arquitetura, caracterizada pelo particionamento do pipeline em três estágios a serem implementados em componentes distintos, foi uma contribuição relevante

à arquitetura do NP9. Isto indica que, em alguns casos, é preciso adaptar a arquitetura à uma estrutura para implementação que seja capaz de aproveitar melhor os recursos da tecnologia e alcançar um melhor desempenho, como foi demonstrado nesse trabalho. O particionamento demonstrou, no projeto do NP9, ser uma técnica bastante vantajosa e alcançou uma melhoria de cerca de 40% na freqüência máxima de operação do processador.

Como prováveis extensões do trabalho apresentado podemos citar alguns projetos a serem considerados:

a) Implementação de um sistema de processamento de imagens utilizando o NP9.

b) Estudo e implementação de sistemas de síntese de alto nível voltada a componentes reprogramáveis para aplicações específicas.

c) Estudo de uma possível implementação do NP9 em VLSI.

d) Análise comparativa entre as versões VLSI e FPGA do NP9. Independente dos resultados alcançados, isso permitirá uma análise mais precisa das vantagens na utilização dos FPGAs, comparando desempenho e custo de ambas.

e) Projeto e implementação de um sistema de programação de algoritmos sobre a estrutura do NP9.

f) Estudo de uma nova arquitetura, baseada no NP9, com processadores elementares de interconexão reprogramáveis.

APÊNDICE A - MODELOS VHDL UTILIZADOS

A.1. np9_pkg.vhd

```
-------------------------------------------------------------------
-- PACKAGE NP9
-- Arquivo : np9_pkg.vhd
-- Autor   : Alexandro Magno dos Santos Adario
-- Local   : Instituto de Computacao
--           Universidade Estadual de Campinas
--           SP - Brasil
-------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

package NP9_PKG is

   subtype PIXEL_INT     is INTEGER range    0 to 255;
   subtype PIXEL_INT_S   is INTEGER range -255 to 255;
   subtype PESO_S        is STD_LOGIC_VECTOR (1 downto 0);
   subtype INT_PADRAO    is INTEGER range 1 to 9;

   type    PIXEL_INT_VEC   is array ( NATURAL range <> ) of PIXEL_INT;
   type    PIXEL_INT_S_VEC is array ( NATURAL range <> ) of PIXEL_INT_S;
   type    PIXEL_INT_S_MTZ is array ( NATURAL range <>, NATURAL range <> )
           of PIXEL_INT_S;

end NP9_PKG;
```

A.2. celula.vhd

```
-------------------------------------------------------------------
-- CELULA
-- Arquivo : celula_hl.vhd
-- Autor   : Alexandro Magno dos Santos Adario
-- Local   : Instituto de Computacao
--           Universidade Estadual de Campinas
--           SP - Brasil
-------------------------------------------------------------------
library IEEE;
library NP9HL;
use IEEE.STD_LOGIC_1164.ALL;
use NP9HL.NP9_PKG.ALL;

entity CELULA_HL is
   port ( X1, X2 : in  PIXEL_INT_S;
```

```
          W1, W2 : in  PESO_S;
          OPC    : in  STD_LOGIC;
          CLK    : in  STD_LOGIC;
          Y      : out PIXEL_INT_S );
end CELULA_HL;

architecture ALTONIVEL of CELULA_HL is

   signal TEMP1, TEMP2, Y_TEMP : PIXEL_INT_S;

   function MULT (VALOR: PIXEL_INT_S; PESO: PESO_S)
   return PIXEL_INT_S is
      variable RESULTADO: PIXEL_INT_S;
   begin
      if PESO = "11" then
         RESULTADO := -VALOR;
      elsif PESO = "00" then
         RESULTADO := 0;
      elsif PESO = "01" then
         RESULTADO := VALOR;
      end if;
      return RESULTADO;
   end;

   function MAX ( V1, V2 : PIXEL_INT_S ) return PIXEL_INT_S is
      variable VMAX : PIXEL_INT_S;
   begin
      if V1 > V2 then
         VMAX := V1;
      else
         VMAX := V2;
      end if;
      return VMAX;
   end;

   function ADD ( V1, V2 : PIXEL_INT_S ) return PIXEL_INT_S is
      variable VSUM : INTEGER;
   begin
      VSUM := V1 + V2;
      if VSUM > 255 then
         VSUM := 255;
      elsif VSUM < -255 then
         VSUM := -255;
      end if;
      return VSUM;
   end;

begin

   TEMP1 <= MULT( X1, W1);
   TEMP2 <= MULT( X2, W2);

   Y_TEMP <= ADD ( TEMP1, TEMP2 ) when OPC = '0' else
```

```
            MAX ( TEMP1, TEMP2 );

   REGISTRADOR : process (CLK)
   begin
      if CLK 'EVENT and CLK = '1' then
         Y <= Y_TEMP;
      end if;
   end process REGISTRADOR;

end;
```

A.3. pipeline.vhd

```
-----------------------------------------------------------------------
-- PIPELINE
-- Arquivo : pipeline_hl.hdl
-- Autor   : Alexandro Magno dos Santos Adario
-- Local   : Instituto de Computacao
--           Universidade Estadual de Campinas
--           SP - Brasil
-----------------------------------------------------------------------
library IEEE;
library NP9HL;
use IEEE.STD_LOGIC_1164.ALL;
use NP9HL.NP9_PKG.ALL;

entity PIPELINE_HL is
port ( X   : in  PIXEL_INT_VEC ( 1 to 9) ;
       PRG : in  STD_LOGIC_VECTOR ( 1 to 405 );
       CLK : in  STD_LOGIC;
       Y   : out PIXEL_INT_S_VEC ( 1 to 9) );
end PIPELINE_HL;

architecture ESTRUTURA of PIPELINE_HL IS

   component CELULA_HL
   port ( X1, X2 : in  PIXEL_INT_S;
          W1, W2 : in  PESO_S;
          OPC    : in  STD_LOGIC;
          CLK    : in  STD_LOGIC;
          Y      : out PIXEL_INT_S );
   end component;

   function INDICE ( COLUNA, LINHA : INT_PADRAO )
   return INTEGER is
      variable IND : INTEGER range 0 to 10;
   begin
      if (LINHA rem 2) = (COLUNA rem 2) then
         IND := LINHA + 1;
      else
         IND := LINHA - 1;
      end if;
```

```
      if IND = 0 then
         IND := 1;
      end if;
      if IND = 10 then
         IND := 9;
      end if;
      return IND;
   end;

   function INDPRG ( COLUNA, LINHA : INT_PADRAO )
   return INTEGER is
      variable IND : INTEGER range 1 to 405;
   begin
      IND:= (COLUNA - 1)*45 + (LINHA - 1)*5 +1;
      return IND;
   end;

   signal X_IN : PIXEL_INT_S_MTZ( 1 to 10, 1 to 9 );

begin

   X_IN(1, 1) <= X(1);
   X_IN(1, 2) <= X(2);
   X_IN(1, 3) <= X(3);
   X_IN(1, 4) <= X(4);
   X_IN(1, 5) <= X(5);
   X_IN(1, 6) <= X(6);
   X_IN(1, 7) <= X(7);
   X_IN(1, 8) <= X(8);
   X_IN(1, 9) <= X(9);

   P1: for COL in 1 to 9 generate
      P2: for LIN in 1 to 9 generate
         CEL: CELULA_HL port map
              ( X_IN(COL, LIN), X_IN(COL, INDICE(COL, LIN)),
                PRG(INDPRG(COL, LIN)+0 to INDPRG(COL, LIN)+1),
                PRG(INDPRG(COL, LIN)+2 to INDPRG(COL, LIN)+3),
                PRG(INDPRG(COL, LIN)+4), CLK, X_IN(COL+1, LIN) );
      end generate P2;
   end generate P1;

   Y(1) <= X_IN(10, 1);
   Y(2) <= X_IN(10, 2);
   Y(3) <= X_IN(10, 3);
   Y(4) <= X_IN(10, 4);
   Y(5) <= X_IN(10, 5);
   Y(6) <= X_IN(10, 6);
   Y(7) <= X_IN(10, 7);
   Y(8) <= X_IN(10, 8);
   Y(9) <= X_IN(10, 9);

end ESTRUTURA;
```

A.4. np9_hl.vhd

```
--------------------------------------------------------------------
-- NP9
-- Arquivo : np9_hl.vhd
-- Autor   : Alexandro Magno dos Santos Adario
-- Local   : Instituto de Computacao
--           Universidade Estadual de Campinas
--           SP - Brasil
--------------------------------------------------------------------
library IEEE;
library NP9HL;
use IEEE.STD_LOGIC_1164.ALL;
use NP9HL.NP9_PKG.ALL;

entity NP9_HL is
   port ( X      : in  PIXEL_INT_VEC (1 to 9);
          PRG_IN : in  STD_LOGIC;
          CLK    : in  STD_LOGIC;
          STAT   : in  STD_LOGIC;
          X_OUT  : out PIXEL_INT );
end NP9_HL;

architecture ESTRUTURA of NP9_HL is

   component PIPELINE_HL
   port ( X   : in  PIXEL_INT_VEC ( 1 to 9) ;
          PRG : in  STD_LOGIC_VECTOR ( 1 to 405 );
          CLK : in  STD_LOGIC;
          Y   : out PIXEL_INT_S_VEC ( 1 to 9) );
   end component;

   signal PRG_TEMP : STD_LOGIC_VECTOR (1 to 410);
   signal CEL_PRG  : STD_LOGIC_VECTOR (1 to 405);
   signal MUX_SEL  : STD_LOGIC_VECTOR (1 to 4);
   signal Y_TEMP   : PIXEL_INT_S_VEC  (1 to 9);
   signal INVERTE  : STD_LOGIC;

   for PIPE : PIPELINE_HL use entity NP9HL.PIPELINE_HL(ESTRUTURA);

begin

   PRGREG: process ( CLK )
   begin
      if CLK'EVENT and CLK = '1' then
         if STAT = '1' then            -- Status de Programacao
            PRG_TEMP(410) <= PRG_IN;
            for I in 409 downto 1 loop
               PRG_TEMP(I) <= PRG_TEMP(I+1);
            end loop;
         end if;
      end if;
```

```
   end process PRGREG;

   INVERTE <= PRG_TEMP(1);
   MUX_SEL <= PRG_TEMP(2 to 5);
   CEL_PRG <= PRG_TEMP(6 to 410);

   PIPE : PIPELINE_HL port map ( X, CEL_PRG, CLK, Y_TEMP );

   MUX : process ( MUX_SEL, INVERTE, Y_TEMP )
      variable X_OUT_TEMP : PIXEL_INT_S;
   begin
      case MUX_SEL is
         when "0000" => X_OUT_TEMP := Y_TEMP(1);
         when "0001" => X_OUT_TEMP := Y_TEMP(2);
         when "0010" => X_OUT_TEMP := Y_TEMP(3);
         when "0011" => X_OUT_TEMP := Y_TEMP(4);
         when "0100" => X_OUT_TEMP := Y_TEMP(5);
         when "0101" => X_OUT_TEMP := Y_TEMP(6);
         when "0110" => X_OUT_TEMP := Y_TEMP(7);
         when "0111" => X_OUT_TEMP := Y_TEMP(8);
         when "1111" => X_OUT_TEMP := Y_TEMP(9);
         when others => X_OUT_TEMP := 0;
      end case;
      if X_OUT_TEMP < 0 then
         if INVERTE = '1' then
            X_OUT <= - X_OUT_TEMP;
         else
            X_OUT <= 0;
         end if;
      else
         X_OUT <= X_OUT_TEMP;
      end if;
   end process MUX;

end ESTRUTURA;
```

APÊNDICE B - EXEMPLOS DE ALGORITMOS DE PDI IMPLEMENTADOS NO NP9

Este apêndice apresenta os resultados de alguns algoritmos implementados no NP9, através da simulação de um sistema de processamento de imagem (figura 28). O sistema, conforme descrito na seção 6.4, foi todo modelado em VHDL e simulado tomando como base uma imagem formato PGM (figura 40a). A programação, controle, leitura da imagem de entrada e gravação da imagem de saída foi toda realizada por modelos VHDL, não tendo sido necessária a utilização de qualquer outro tipo de sistema auxiliar. Os algoritmos foram implementados conforme apresentados por Leite [39].A experiência demonstrou que a arquitetura resolve adequadamente certos problemas de PDI. O sistema executou os algoritmos de dilatação, erosão, filtros de mediana e de mediana separável e detector morfológico de contorno.Para os testes de filtragem de ruídos, foi utilizada uma imagem produzida através da injeção artificial de ruídos sobre a imagem base (figura 41).

B.1. Detector Morfológico de Contorno

O detector morfológico de contorno [37] é um algoritmo executado a partir de uma estrutura em árvore (figura 37) sobre o grafo de fluxo de dados inicial. Esse detector é definido pela equação

$$E(i,j) = g(G_e(i,j), G_d(i,j)) \qquad (5)$$

onde g é a função adição, máximo ou mínimo e $G_e(i,j)$ e $G_d(i,j)$ são, respectivamente, o operador residual de erosão (diferença entre o valor do pixel central e a erosão da vizinhança) e o operador residual de dilatação (diferença entre a dilatação e o valor do pixel central da vizinhança). A escolha da função g depende de algumas características da imagem, como por exemplo a imunidade ao ruído. O resultado desse detector pode ser conferido na figura 40b.

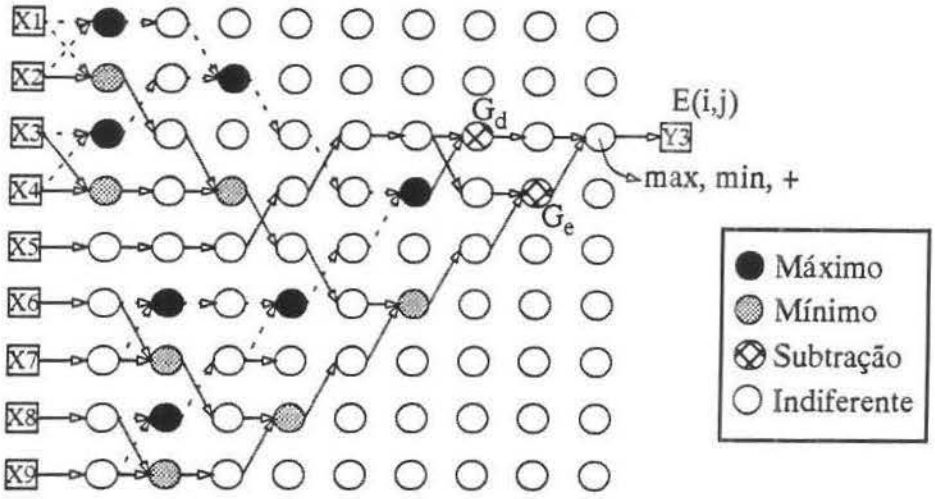


Figura 37: Grafo do detector morfológico de contorno

B.2. Ordenação por Transposição Ímpar-par

Este algoritmo, apresentado em detalhe na seção 2.3.2, produz três valores importantes: o máximo, o mínimo e a mediana da vizinhança. Desta forma a mesma estrutura de programação executa três diferentes tipos de processamento. Obtendo o máximo, a imagem é dilatada (figura 40c); com o mínimo, a imagem é erodida (figura 40d); e, por último, a mediana permite a filtragem de ruídos (figura 41a).

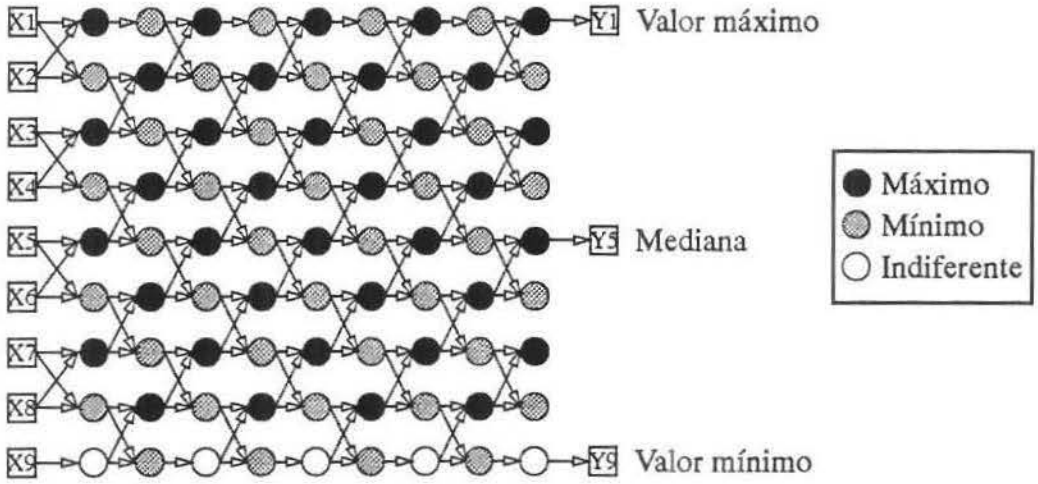


Figura 38: Grafo de fluxo de dados da transposição ímpar-par

B.3. Filtro de Mediana Separável

O filtro de mediana separável é uma versão do filtro de mediana e, em alguns casos, tem desempenho melhor que o original. Sua idéia é subdividir a vizinhança em linhas ou colunas, calcular a

mediana de cada sub-divisão e então calcular a mediana dessas medianas. O grafo desse filtro é representado na figura 39 e os resultados estão na figura 41.

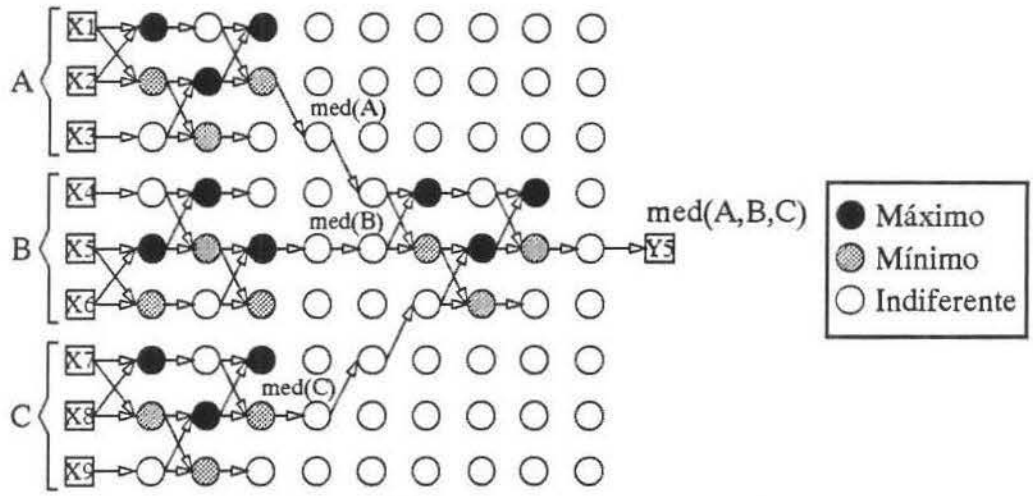
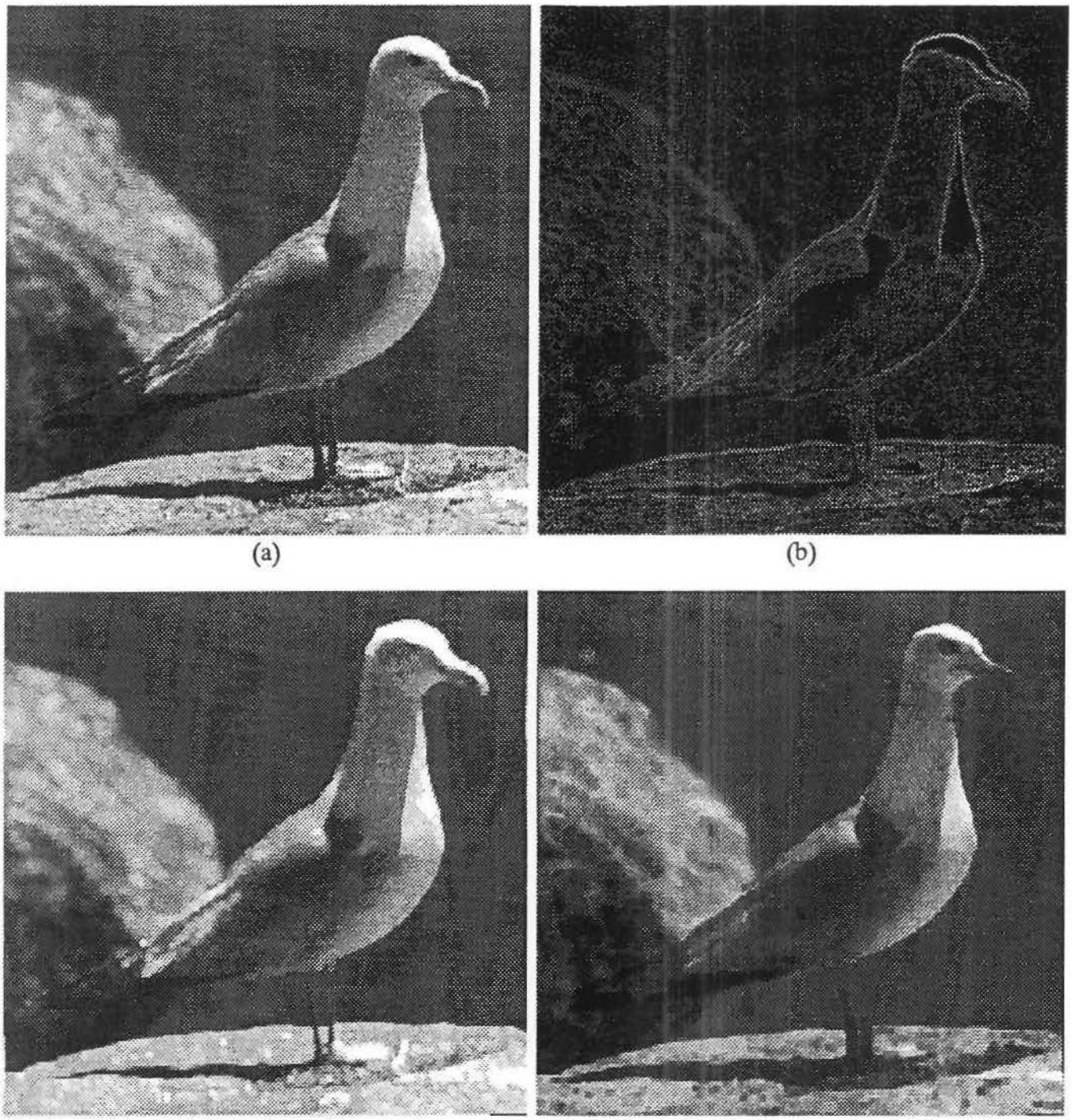


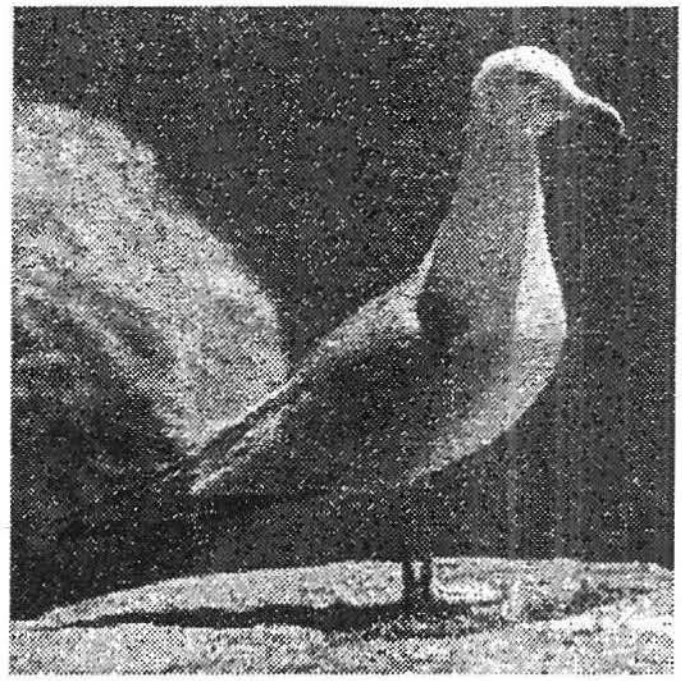
Figura 39: Filtro da mediana separável



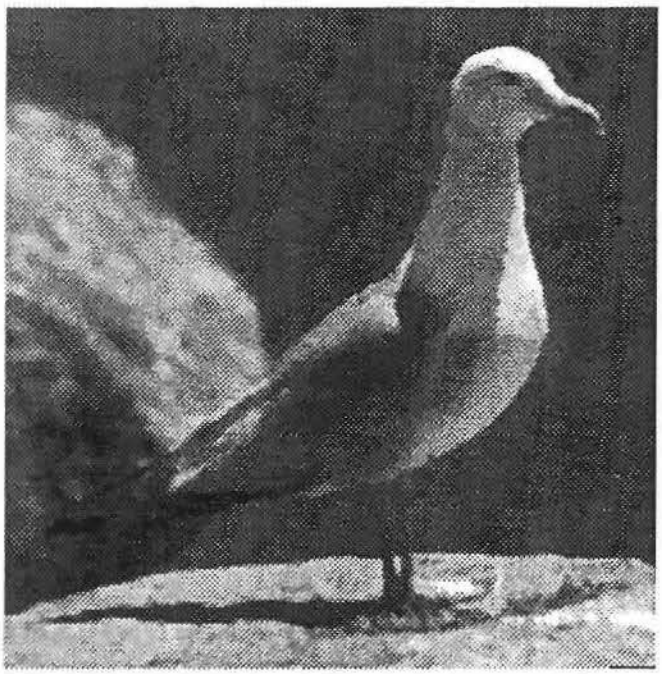
(a) (b)

(c) (d)

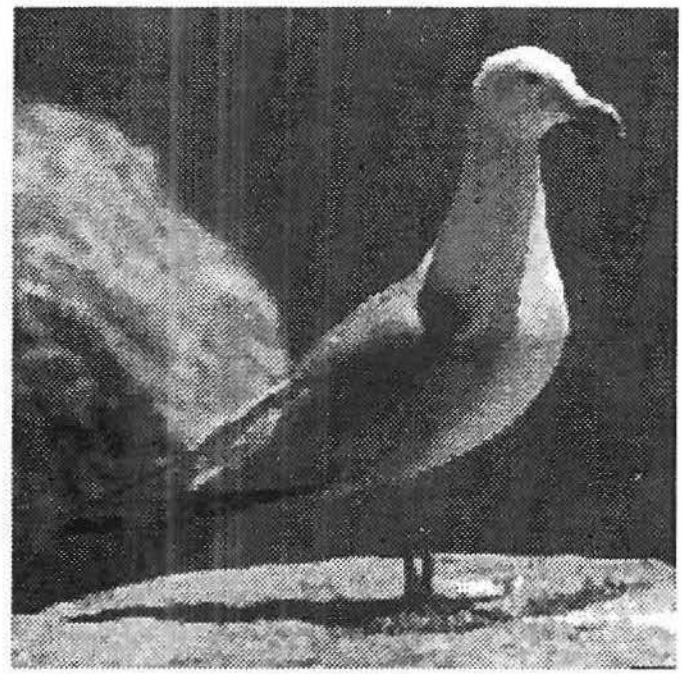
Figura 40: Exemplo da aplicação do detector morfológico de contorno (b), da dilatação (c) e da erosão (d) sobre uma imagem de tamanho 512x512 (a)



(a)



(b)



(c)

Figura 41: Exemplo de aplicação dos filtros de mediana (b) e mediana separável (c) sobre uma imagem com ruído aleatório

REFERÊNCIAS BIBLIOGRÁFICAS

1) ADÁRIO, Alexandro M. S., CÔRTES, Mario L. & LEITE, Neucimar J. ***"Considerações sobre a Síntese em FPGAs de Células de Processadores Matriciais"***. In: XXIII Seminário Integrado de Software e Hardware, Recife, Ago 1996.

2) AKL, Selim G. ***Parallel Sorting Algorithms***. Academic Press, Orlando, Florida, 1985.

3) ALEXANDRINO, Josemir C. ***"Uma Implementação em VLSI para Reconhecimento de Padrões de Imagens"***. Dissertação de Mestrado. Universidade Estadual de Campinas. 1994.

4) ALEXANDRINO, Josemir C. & CÔRTES, Mario L. ***"Uma Implementação em VLSI para reconhecimento de imagens"***. In: VI Conferência Nacional de Inteligência Artificial, Barquisimeto, Venezuela, Out. 1993.

5) ALEXANDRINO, Josemir C. & CÔRTES, Mario L. ***"Uma Implementação em VLSI para reconhecimento de imagens"***. In: XX Seminário Integrado de Software e Hardware (SEMISH), Florianópolis. Set. 1993.

6) ALTERA. ***Max+plus II Getting Started***.Altera Corporation, San Jose, California, 1995.

7) ALTERA. ***Mentor Graphics & Max+plus II Software Interface Guide***. Altera Corporation, San Jose, California, 1995.

8) ALTERA. ***Data Book***. Altera Corporation, San Jose, California, 1996.

9) ASHENDEN, Peter J. ***The Designer's Guide to VHDL***. Morgan Kaufmann, London, 1995

10) BATCHER, K. E. "Design of a Massively Parallel Processor" In: IEEE Transactions on Computers, vol C-29, no. 9. IEEE Computer Society. Jan 1980, pp 836-840

11) BRAGE, Jens P. ***"Foundations of a High-Level Synthesis System"***. PhD Thesis. Center for Integrated Electronics, Department of Computer Science, Technical University of Denmark. July/1993.

12) CALVEZ, Jean P. ***Embedded Real-Time Systems:*** *A Specification and Design Methodology.* Tradução (francês): Alan Wyche & Charles Edmundson. John Wiley, Chichester, England, 1993. Tít. Orig: *Spécification et conception des systémes: une méthodologie.*

13) CAMPOSANO, Raul. ***"High-Level Synthesis"***. In: Encyclopedia of Computer Science and Technology, vol. 38, suppl. 13. Marcel Dekker, New York, 1993. pp. 129-152.

14) CAMPOSANO, Raul. ***"Behavioral Synthesis"***. In: 33rd Design Automation Conference (DAC96), Las Vegas, NV. June 1996

15) CÔRTES, M. L. & MENDONÇA F. Neto, J. ***Introdução ao Teste de Circuitos Digitais***. V Escola Brasileiro-Argentina de Informática. 1991.

16) CÔRTES, Mario L. N. ***Notas de Aula***. Disciplina: Ferramentas de Projeto de Circuitos Integrados, Instituto de Computação, Universidade Estadual de Campinas, 1995.

17) DANIELSSON, Per-Erik & LEVIALDI, Stefano. ***"Computer Architectures for Pictorial Information Systems"***. In: IEEE Computer Magazine, Nov. 1981. pp-53-67

18) EUROPEAN SPACE AGENCY. ***VHDL Modelling Guidelines***. ESA, European Space Research and Technology Center, Noondwijk, The Netherlands, Sept 1994. (ASIC/001).

19) DUFF, Michael J. B. & FOUNTAIN, Terry. J. ***Cellular Logic Image Processing***. Academic Press, London, 1986.

20) FIELDS, Carol A. ***"Proper Use of Hierarchy in HDL-Based High Density FPGA Design"***. In: Field-Programmable Logic and Applications - 5th International Workshop (FPL'95). Oxford. Aug/Sept 1995. pp 168-177.

21) FOUNTAIN, Terry J. ***"CLIP4: A Progress Report"***. In: Languages and Architectures for Image Processing. DUFF, M. J. B. & LEVIALDI, S. (ed.) Academic Press, London, 1981.

22) FOUNTAIN, Terry J. ***Processor Arrays***: Architectures and Applications. Academic Press, London, 1987.

23) FOUNTAIN, Terry J. ***"The CLIP7 Programme"***. In: Multiprocessor Computer Architectures. FOUNTAIN, T. J. & SHUTE, M. J. (ed.). Nort-Holland, Amsterdam, 1990.

24) GSCHWIND, Michael & SALAPURA, Valentina. ***"A VHDL Methodology for FPGAs"***. In: Field-Programmable Logic and Applications - 5th International Workshop (FPL'95). Oxford. Aug/ Sept 1995. pp 208-217.

25) HADLEY, James. D. & HUTCHINGS, Brad L. ***"Design Methodologies for Partially Reconfigured System"***. In: Proceedings. of the IEEE Workshop on FPGAs for Custom Computing Machines. IEEE Computer Society. IEEE Computer Society. Press, Abr 1995.

26) HUBER, J. P. & ROSNECK, M. W. ***Successful ASIC Design the first Time through***. Van Nostrand Reinhold. New York. 1991.

27) HUNT, D. J. ***"The ICL DAP and its Application to Image Processing"***. In: Languages and Architectures for Image Processing. DUFF, M. J. B. & LEVIALDI, S. (ed.) Academic Press, London, 1981.

28) HWANG, Kai. ***Advanced Computer Architecture***: Parallelism, Scalability, Programmability. McGraw-Hill. 1993.

29) HWANG, Kai & XU, Zhuwei. ***"Scalable Parallel Computers for Real-Time Signal Processing"***. In: IEEE Signal Processing Magazine, vol. 13, no. 4. IEEE Signal Processing Society, July/ 1996. pp 50-66.

30) HWANG, Ting-Ting, OWENS, Robert M., IRWIN, Mary J. et alli. ***"Logic Synthesis for Field-Programmab le Gate Arrays"***. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 10. IEEE Computer Society, Oct/1994. pp 1280-1286.

31) IEEE. ***IEEE Standard Multivalue Logic System for VHDL Model Interoperability***. IEEE Standard 1164-1993. IEEE Inc., New York, NY, 1993.

32) IEEE. ***IEEE Standard VHDL Language Reference Manual***. IEEE Standard 1076-1987. IEEE Inc., New York, NY, 1988.

33) KHOSRAVIPOUR, M. & GRÜNBACHER, H. ***"VHDL-Based Rapid Prototyping Using FPGA Technology"***. In: Field-Programmable Logic and Applications - 5th International Workshop (FPL'95). Oxford. Aug/Sept 1995. pp 218-226.

34) KNUTH, Donald E. ***The Art of Computer Programming***, vol. 3. Addison-Wesley. Reading, Massachusetts, 1973.

35) KRÜGER, Carlos G. & CÔRTES, Mario L. ***"Modelamento, Simulação e Síntese com VHDL"***. Departamento de Ciência da Computação, IMECC, UNICAMP. Relatório Técnico 19/93. Set. 1993.

36) JACOBI, Ricardo P. ***Síntese Lógica de Circuitos Combinacionais***. X Escola de Computação, Campinas, 1996.

37) LEE, J. S. J. et alli. " Morphologic Edge Detection", IEEE Journal of Robotics and Automation, vol 3, n. 2, Apr 1987, pp 142-156.

38) LEITE, Neucimar. J. & BARROS, Marcelo A. ***"A Highly Reconfigurable Neigborhood Image Processor Based on Functional Programming"***. IEEE International Conference on Image Processing. Nov 1994. pp. 659-663.

39) LEITE, Neucimar J. ***"Quelques Modèles d'Automates Cellulaires pour le Traitement d'Images"***. Thèse de Doctorat. Université Paris 6, 1993.

40) LEITE, Neucimar J. & BERTRAND, G. ***"A Versatile Image Processing Cellular Automaton Based on Functional Programming"***, In: Proceedings of VII Brazilian Microelectronics Society Congress. Jul. 1992. pp. 541-551.

41) LICHTERMANN, Jan & NEUSTÄDTER, Günter. *"A High-Speed Rotation Processor"*. In: Field-Programmable Logic and Applications - 4th International Workshop (FPL'94). Prague, 1994. pp 123-125.

42) MEERSMAN, C. ***The VHDL Standard***: *An Overview of activities, organizations and european tool efforts*. E2S Software Engineering, Zwijnaarde, Belgium, May 1994. (E2S/ESTEC/ SCADES 2/WO5-D1). FTP: ftp.estec.esa.nl:/pub/vhdl/doc/VHDLReport.ps

43) MENTOR GRAPHICS CORPORATION. ***Mentor Graphics System Overview Manual.*** Mentor Graphics Corporation, Wilsonville, Oregon, 1993.

44) MENTOR GRAPHICS CORPORATION. ***System-1076 Design and Model Development Manual.*** Mentor Graphics Corporation. Wilsonville, Oregon. 1994.

45) MENTOR GRAPHICS CORPORATION. ***Quicksim II User's Manual.*** Mentor Graphics Corporation. Wilsonville, Oregon. 1995.

46) MENTOR GRAPHICS CORPORATION. ***Synthesizing with AutoLogic.*** Mentor Graphics Corporation. Wilsonville, Oregon. 1994.

47) MENTOR GRAPHICS CORPORATION. ***Synthesizing with AutoLogic II.*** Mentor Graphics Corporation. Wilsonville, Oregon. 1995.

48) MOHANAKRISHNAN, Satish & EVANS, Joseph B. ***"Automatic Implementation of FIR Filters on FPGAs"***. In: IEEE Signal Processing Letters, Mar 1995

49) MOHANAKRISHNAN, Satish & EVANS, Joseph B. ***"A Framework for the Design of High Speed FIR Filters on FPGAs"***. In: Proceedings (1994) of International Conference on Signal Processing Applications and Technology.

50) PELLERIN, David. ***An Introduction to VHDL.*** Accolade Design. http://www.acc-eda.com/ h_intro.htm.

51) PELLERIN David & TAYLOR, Douglas. ***VHDL Made Easy.*** Prentice-Hall, 1997.

52) PEYRARD, René & KLEIN, Jean C. ***"Speeding up Mathematical Morphology Processes by Using a New ASIC: PIMM1"***. In: Workshop on Computer Architecture for Machine Perception (CAMP91), Paris, Dec. 1991.

53) PERRY, Douglas. ***VHDL***: Second Edition. McGraw-Hill, New York, 1993.

54) PHOUKAS, Dimitri. ***Logic Synthesis using VHDL.*** University of Windsor. http:// hobbes.engn.uwindsor.ca/~dimitris/VHDLsynthTutorial/

55) PIRSON, A. et alli, ***"A Highly Efficient method for Synthesizing Some Digital Filters"***. In: Signal Processing IV: Theories and Applications. LACOUME, J. L. et alli (eds). Nort-Holland, Amsterdam, 1988, pp. 1481-1484.

56) PITAS, I. & VENETSANOPOULOS, A. N. ***"A New Filter Strucutre for Implementation of Certain Classes of Image Processing Operations"***. IEEE Transaction on Circuits and Systems, Jun 1988. pp 636-647, vol 35, n.6

57) POTTER, J. L. " Image Processing on the Massively Parallel Processor". IEEE Computer Magazine, vol-16 (12), Jan 1983, pp 62-67.

58) PRESTON Jr, K. ***"Comparison of Parallel Processing Machines: A Proposal"***. In:Languages and Architectures for Image Processing. DUFF, M. J. B. & LEVIALDI, S. (ed.) Academic Press, London. 1981.

59) REESE, Bob. ***VHDL Synthesis Tutorial.*** Mississipi State University. http://www.erc.msstate.edu/~reese/vhdl_synthesis/index.html

60) RUSHTON, Andrew. ***WHDL for Logic Synthesis***. McGraw-Hill, 1995.

61) SALAPURA, Valentina & WALECZEK, G. ***"Designing from VHDL Behavioral Description to FPGA Implementation"***. In: Proceedings of AustroChip'94, Brunn am Gebirge, Austria. June. 1994. pp 141-146.

62) SMITH, Douglas J. ***"VHDL & Verilog Compared and Contrasted - Plus Modeled Example Writtem in VHDL, Verilog and C"***. In 33rd Design Automation Conference. Las Vegas, June/ 1996.

63) UHR, Leonard. ***Algorithm-Structured Computer Array and Networks***: Architectures and Processes for Images, Percepts, Models, Information. Academic Press, Orlando, Flórida, 1984.

64) WESTE, N. & ESHRAGIAN, K. ***Principles of CMOS VLSI Design***. Addison-Wesley. 1985.

65) WIRTHLIN, Michael J. & HUTCHINGS, Brad L. ***"A dynamic Instruction Set Computer"***. In: Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines. IEEE Computer Society, IEEE Computer Society Press, Apr 1995.

66) WIRTHLIN, Michael J. & HUTCHINGS, Brad L. ***"DISC: The dynamic instruction set computer"*** In: Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, John Schewel, Editor, 1995. Proc. SPIE 2607, pp. 92-103.

67) XILINX. ***The Programmable Logic Data Book.*** Xilinx Inc. San Jose, CA, 1994.