UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de São José do Rio Preto

Jacqueline Gomes Mertes

Implementação em FPGA de um sistema para processamento de imagens digitais para aplicações diversificadas

São José do Rio Preto
2012

Jacqueline Gomes Mertes

Implementação em FPGA de um sistema para processamento de imagens digitais para aplicações diversificadas

Dissertação de Mestrado elaborada junto ao Programa de Pós-Graduação em Ciência da Computação – Área de Concentração em Sistemas de Computação, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Norian Marranghello

São José do Rio Preto
2012

Mertes, Jacqueline Gomes.
Implementação em FPGA de um sistema para processamento de imagens digitais para aplicações diversificadas / Jacqueline Gomes Mertes - São José do Rio Preto : [s.n.], 2012.
117 f. : il. ; 30 cm.

Orientador: Norian Marranghello
Dissertação (mestrado) - Universidade Estadual Paulista, Instituto de Biociências, Letras e Ciências Exatas

1. Processamento digital de imagens. 2. Implementação de circuitos digitais. 3. FPGA. I. Marranghello, Norian. II. Universidade Estadual Paulista, Instituto de Biociências, Letras e Ciências Exatas. III. Título.

CDU – 004.932

Jacqueline Gomes Mertes

Implementação em FPGA de um sistema para processamento de imagens digitais para aplicações diversificadas

Dissertação de Mestrado elaborada junto ao Programa de Pós-Graduação em Ciência da Computação – Área de Concentração em Sistemas de Computação, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Banca Examinadora

Prof. Dr. Norian Marranghello
UNESP – São José do Rio Preto
Orientador

Prof. Dr. Furio Damiani
UNICAMP – Campinas

Prof. Dr. Alexandre C. Rodrigues da Silva
UNESP – Ilha Solteira

São José do Rio Preto
13 de dezembro de 2012

RESUMO

Este trabalho descreve um sistema para o processamento de imagens digitais coloridas. Este sistema possui um conjunto de filtros, o qual aliado a um controlador pode ser configurado pelo usuário através de um arquivo de configuração, buscando a melhor adequação do sistema às imagens a serem tratadas. O conjunto de filtros é composto por filtros que desempenham as tarefas de suavização, deteção de borda, equalização de histogramas, normalização de cores e normalização de luminância. O sistema foi descrito utilizando a linguagem de descrição de *hardware* System Verilog e implementado em um FPGA. Devido à sua característica reconfigurável, este sistema mostrou-se capaz de processar diversos tipos de imagens coloridas, ajustando-se facilmente às mais diferentes aplicações.

Palavras-chave: Processamento digital de imagens, implementação de circuitos digitais, FPGA.

ABSTRACT

This work describes a colored digital images processing system. This system has a set of filters, which in junction with a controller can be configured by the user through a setup file, in order to adapt the system to the images to be treated.This set is composed by several filters that perform tasks such as smoothing, edge detection, histogram equalization, color normalization and luminance normalization. The system was described using hardware description language (System Verilog), and implemented in an FPGA. Due to its reconfigurable caracteristic, this system showed capable of processing several types of colored images, easily fitting to a broad set of applications.

Keywords: Digital image processing, digital circuits' implementation, FPGA.

AGRADECIMENTOS

Agradeço à minha família pelo incentivo direto e indireto na conclusão deste trabalho.

Obrigada ao professor Norian, pela oportunidade de retomar o mestrado após alguns anos de afastamento e por acreditar na possibilidade de finalizá-lo da melhor maneira possível.

Agradeço ao meu namorado Jorge Enrique pelo apoio nos momentos decisivos no decorrer deste projeto.

Agradeço às instituições de pesquisa em que trabalhei nos últimos anos que possibilitaram o meu desenvolvimento profissional. Além do CNPq pelo apoio financeiro que custeou o meu trabalho de pesquisa nos últimos anos. Além do grupo Brazil-IP por auxiliar-me nas primeiras etapas deste projeto.

Por fim, agradeço aos amigos que conquistei nos últimos anos que de alguma contribuíram para minha formação pessoal e profissional.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ASIC - *Applicantion Specific Integrated Circuit*

CLB - *Configurable Logic Block*

CMY - C*yan*, *Magenta*, *Yellow*

CPLD - *Complex Programmable Logic Device*

DUV - *Design Under Verification*

EDA - *Eletronic Design Automation*

FPLD - *Field-Programmable Logic Devices*

FPGA - *Field Programmable Gate Array*

HDL - *Hardware Description Language*

HSV - *Hue, Saturation, Value*

IEEE - *Institute of Electrical and Electronics Engineers*

IP - *Intellectual Property*

LCD - *Liquid Cristal Display*

LUT - *Lookup Table*

OVI - *Open Verilog International*

pixel - picture element

PCB - *Printed Circuit Board*

PLD - *Programmable Logic Device*

PAL - *Programmable Array Logic*

PLA - *Programmable Logic Array*

PLL - *Phase Locked Loop*

RAM - *Random Access Memory*

ROM - *Read Only Memory*

RGB - *Red*, *Green*, *Blue*

RTL - *Register Transfer Level*

SRAM - *Static Random Access Memory*

SPLD - *Simple Programmable Logic Device*

ULA - Unidade Lógico-aritmética

VGA - *Video Graphics Array*

VHDL - *Very High Speed Integrated Circuits Hardware Description Language*

CAPÍTULO 1 – Introdução

Uma das maneiras mais simples de transmitir uma informação é através de uma imagem. Uma imagem informa sobre o tamanho, posição e relação entre diferentes objetos. Por meio destes dados, o olho humano é capaz de reconhecer e identificar diferentes objetos contidos em uma mesma imagem. Desta forma, o processamento de imagens tem por objetivo auxiliar na visualização para que haja uma melhor identificação e interpretação destas imagens, podendo ser realizada de forma humana ou automática, com o auxílio de máquinas.

Com este objetivo, uma das primeiras aplicações de técnicas de processamento de imagens surgiu com a melhoria de ilustrações de jornais enviadas por meio de cabo submarino de Londres para Nova Yorque no início da década de 20 [Gonzalez e Woods, 2002].

Com o avanço tecnológico, o processamento digital de imagens aumentou sua gama de aplicações, que incluem as mais diversas áreas, como, por exemplo, análise de recursos naturais por meio de imagens de satélites, obtenção e análise de imagens médicas, análise de imagens de vegetação, aplicações em automação industrial envolvendo o uso de sensores visuais em robôs, etc.

Uma das mais novas utilizações de técnicas de processamento de imagens surgiu com a necessidade de processamento de imagens em tempo real. Uma destas utilizações é na área agrícola, com o intuito de extrair informações quantitativas de imagens de plantações, com diversos

graus de infestações por diferentes agentes biológicos. [Mertes, Marranghello e Pereira, 2008] apresenta a implementação de um conjunto de filtros para auxiliar uma rede neural artificial na detecção de plantas daninhas em plantações de soja. [Docusse et al, 2008] mostra a aplicação de técnicas de processamento de imagens para o tratamento de imagens médicas visando auxiliar no diagnóstico das microcalcificações detectadas em mamografias. [Sandeep e Rajagopalan, 2002] destaca aplicações em sistemas de reconhecimento baseado na detecção de faces humanas.

Tendo em vista a ampla gama de abordagens possíveis, neste trabalho, foram estudadas técnicas de processamento de imagens para a criação de um conjunto de filtros, que aliados a um controlador, são capazes de tratar as mais diversas imagens digitais coloridas. Esse conjunto de filtros contém duas versões de filtros para suavização de ruídos, um para detecção de bordas, um para equalização de histograma, um para normalização de cores e um para normalização de luminância. Estes algoritmos foram criados, implementados utilizando a linguagem SystemVerilog, e foram integrados e testados em uma placa contendo um FPGA, após diversas simulações em ferramentas específicas para este propósito, como o ModelSim da Mentor Graphics.

No segundo capítulo, desta dissertação, são apresentados conceitos sobre processamento de imagens digitais. No terceiro capítulo são discutidos conceitos utilizados para o desenvolvimento de circuitos digitais. O quarto capítulo detalha o processo de desenvolvimento do conjunto de filtros e do controlador. No quinto capítulo são apresentados os resultados obtidos com o sistema desenvolvido. No sexto e último capítulo são apresentadas as conclusões sobre o desenvolvimento do módulo e possibilidades de trabalhos futuros.

CAPÍTULO 2 - Processamento Digital de Imagens

Por se tratar de detalhes de técnicas importantes dentro do campo de processamento digital de imagens, as informações contidas neste capítulo são baseadas em [Gonzalez e Woods, 2002], salvo nas referências devidamente identificadas. O processamento digital de imagens abrange uma grande escala de conceitos teóricos. A seguir são discutidos os passos fundamentais para realizar uma tarefa de processamento de imagem, desde sua aquisição até o resultado final passando por algumas etapas que podem ser vistas na figura 2.1.

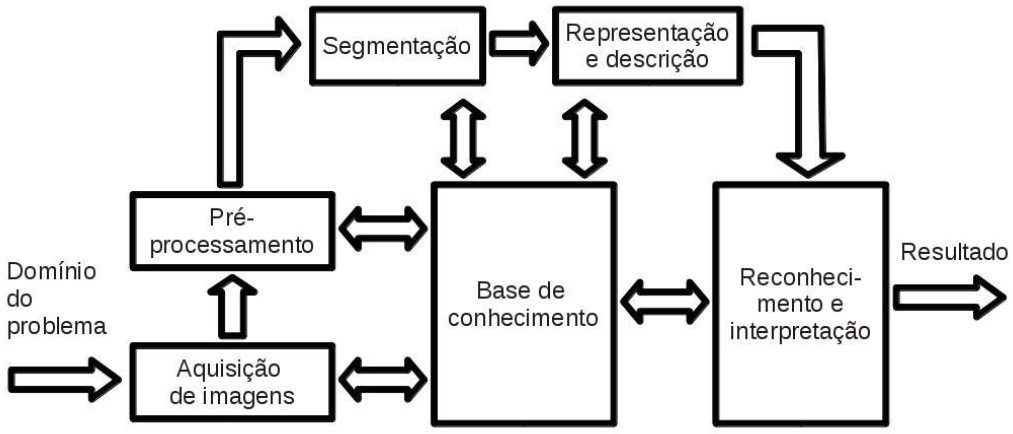


Figura 2.1: Passos fundamentais em processamento de imagens digitais [Gonzalez e Woods, 2002].

A figura 2.1 mostra uma sequência padrão para o reconhecimento de imagens, descritas a seguir:

- *Domínio do problema*: profunda análise do problema a ser tratado para se especificar o objetivo a ser alcançado;
- *Aquisição de imagens*: consiste em utilizar um equipamento para adquirir a imagem, como uma câmera de vídeo, uma câmera fotográfica digital, ou um digitalizador de imagens. Esse processo pode ser também chamado de digitalização, pois consiste na conversão de um sinal da forma analógica para a forma digital;
- *Pré-processamento*: refere-se ao processamento inicial da imagem para correção de distorções geométricas e remoção de ruídos, modificando a imagem para evidenciar características a serem utilizadas nas próximas etapas;
- *Segmentação*: consiste na etapa de processamento em que se analisa a imagem com relação à informação nela presente. A imagem é dividida em diferentes regiões que são, posteriormente, analisadas por algoritmos em busca de informações que a caracterizem. A segmentação consiste em extrair da imagem apenas as áreas que interessam para a resolução do problema;
- *Base de conhecimento*: representa o conhecimento adquirido pelo ser humano, o qual direciona o processamento da imagem para o melhor reconhecimento possível;
- *Representação e descrição*: no processo de representação, as características principais devem ser enfatizadas e extraídas. O processo de descrição é também conhecido como seleção de características, onde são extraídas as características principais que resultam em informação quantitativa, capaz de separar classes de objetos importantes para o reconhecimento dos padrões;
- *Reconhecimento e interpretação*: consiste em atribuir rótulos aos objetos, classificando-os a partir das informações encontradas na imagem na etapa anterior. Esta etapa tenta aproximar o desempenho computacional ao do ser humano ao reconhecer padrões dentro de uma imagem;

- *Resultado*: Objetivos alcançados após o processamento da imagem, como por exemplo, a identificação de um objeto ou a melhoria na visualização de uma cena.

Para que se possa realizar um estudo mais aprofundado sobre as etapas que são relevantes para o desenvolvimento do projeto é necessário fazer uma introdução sobre alguns conceitos a elas relacionados, os quais estão descritos a seguir.

2.1 Conceitos Iniciais

2.1.1 Imagem

Uma imagem pode ser definida como uma função bidimensional *f(x,y)*, a qual representa a intensidade (brilho) da imagem no ponto com coordenadas x e y. A imagem digital representa a discretização da função *f(x, y)*, podendo ser considerada como uma matriz cujos índices de linhas e colunas (x, y), identificam um ponto na imagem, e o correspondente valor do elemento da matriz identifica o nível de cinza naquele ponto. Os elementos dessa matriz digital são chamados de elementos da imagem (pixel).

A natureza básica de *f(x, y)* pode ser caracterizada por dois componentes: a quantidade de luz incidindo na cena sendo observada e a quantidade de luz refletida pelos objetos da cena. Sendo denominados, respectivamente, de iluminação, representado por *i(x, y)* e reflectância, representado por *r(x, y)*. Logo, a imagem pode ser descrita como:

$$f(x,y) = i(x,y)r(x,y) \quad (1)$$

onde

$$0 < i(x,y) < \infty \quad (2)$$

e

$$0 < r(x,y) < 1. \quad (3)$$

Ressaltando que, por a luz ser uma forma de energia, a *f(x,y)* deve ser positiva e finita, ou seja,

$$0 < f(x,y) < \infty. \quad (4)$$

A equação (3) indica que a reflectância é limitada entre 0 (absorção total) e 1 (reflectância total). A natureza de *i(x,y)* é determinada pela fonte de luz, e *r(x, y)* é determinada pelas características dos objetos na cena.

2.1.2 Amostragem e quantização

Como foi dito no item anterior, uma imagem pode ser descrita a partir de uma função contínua. Entretanto, para que uma imagem possa ser tratada computacionalmente é necessário que a imagem tenha cada um dos pontos (x, y) digitalizados. Assim, o processo para trazer uma função contínua para o computador é discretizando-a, ou digitalizando-a, ou seja, tomando valores pontuais ao longo de x e guardando o valor de f(x) correspondente. O processo de discretização do eixo x é chamado de amostragem, o do eixo f(x) é chamado de quantização [Scuri, 2002].

A amostragem mais comum e mais popular é a chamada de uniformemente espaçada, onde cada amostra é tomada em intervalos iguais. Embora existam outras técnicas de amostragem que utilizam menos ou mais amostras de acordo com a imagem.

A quantização mais comum consiste em tomar o valor máximo e o valor mínimo dos pixels da imagem, e dividir este segmento em intervalos iguais de acordo com o número de *bits* definido para armazenar uma amostra. Assim, o número de valores possíveis será 2^n, onde *n* representa o número de *bits* de cada valor a ser armazenado. Comumente n=8, o que equivale a um *byte*, o que representa uma variação de 0 a 255, nos tons da imagem.

Os métodos de quantização visam minimizar os erros entre a imagem real e a imagem quantizada, ou seja, procuram definir variações de tons de cinza que representem da melhor maneira possível os níveis de cinza de uma imagem real.

A partir da figura 2.2, pode-se observar os processos de amostragem e quantização. Na figura 2.2(a), observa-se uma imagem contínua *f(x,y)*, que será convertida para a forma digital. A figura 2.2(b) representa o gráfico da amplitude (tons de cinza) dos valores contínuos da imagem ao longo do segmento AB.

No processo de amostragem, são tomadas amostras igualmente espaçadas ao longo do segmento AB (figura 2.2(c)). A escala do lado direito da figura 2.2(c) representa os níveis de cinza divididos em oito níveis discretos, tal quantidade de níveis é determinado de forma diferente para cada imagem. Os valores contínuos dos níveis de cinza são quantizados

assumindo cada um dos oito níveis discretos para cada amostragem. Isso é realizado dependendo da proximidade de uma amostra com um dos oito níveis de quantização, determinados para essa imagem. As amostras digitais resultantes da amostragem e da quantização podem ser observadas na figura 2.2(d).

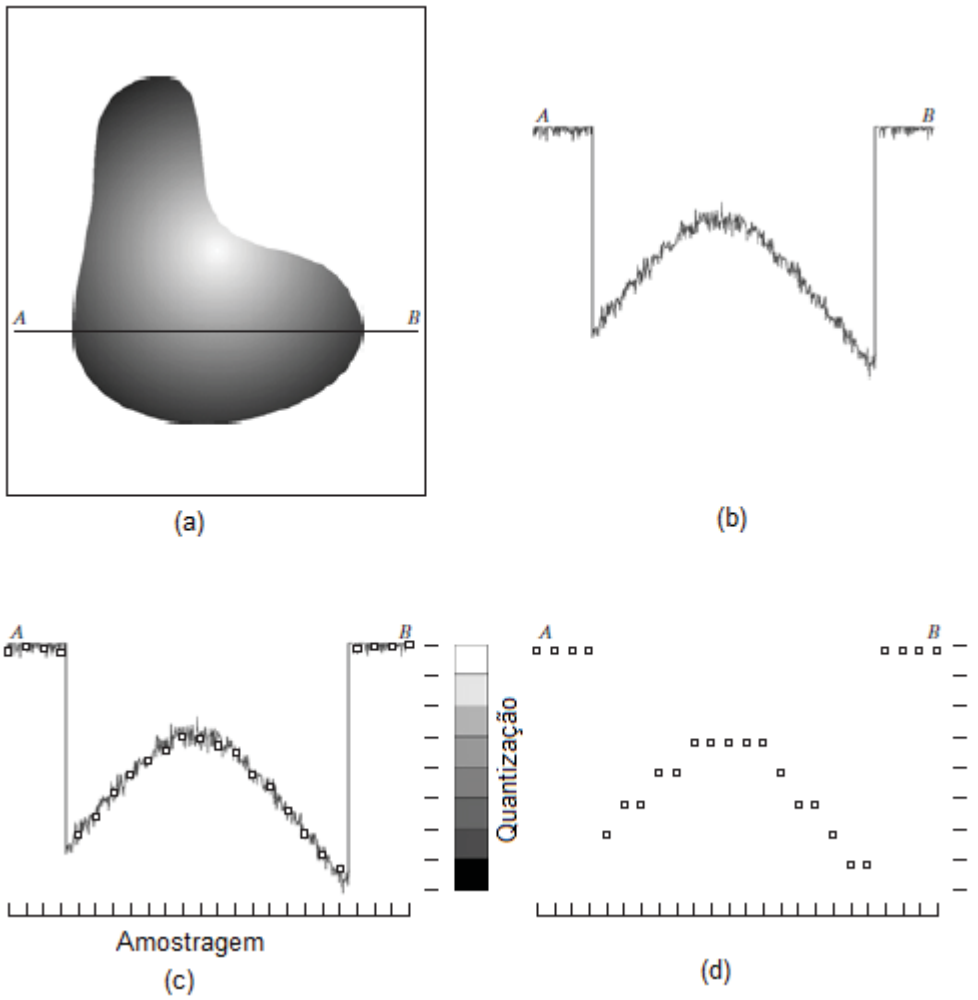
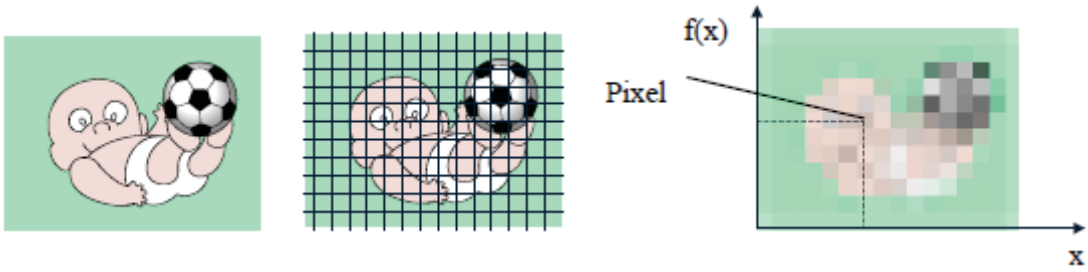


Figura 2.2: Processo de amostragem e quantização (a) Imagem contínua. (b) Gráfico da amplitude dos valores do segmento AB. (c) Amostragem e Quantização. (d) Segmento AB digitalizado [Gonzalez e Woods, 2002].

A figura 2.3(a) ilustra uma imagem contínua captada por um sensor. A figura 2.3(b) ilustra a imagem amostragem e a figura 2.3(c) mostra a imagem quantizada.



(a) (b) (c)

Figura 2.3 Exemplo de amostragem e quantização: (a) Imagem contínua, (b) amostragem e (c) quantização [Scuri, 2002].

2.1.3 Vizinhança de um pixel

Um pixel *p* nas coordenadas (x, y) possui quatro vizinhos horizontais e verticais, dispostos conforme a figura 2.4. Esse conjunto de pixels formam a vizinhança de 4 de *p*, representada por $N_4(p)$, nesse tipo de vizinhança, cada pixel está a uma unidade de distância de (x, y).

Os quatro vizinhos diagonais de *p*, representada por $N_d(p)$ possuem a seguinte distribuição em relação ao pixel *p* como pode ser visto na figura 2.4. Dessa forma, os pixels compartilham apenas um vértice. Esse conjunto de vértices, junto com a vizinhança de quatro, são chamados de vizinhança de 8 de *p*, representada por $N_8(p)$.

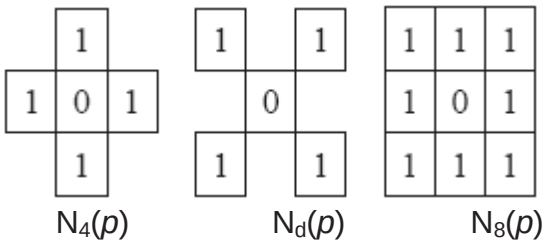


Figura 2.4: Vizinhanças $N_4(p)$, $N_d(p)$ e $N_8(p)$ do pixel central $p = 0$.

2.1.4 Operadores locais

Os operadores locais mais empregados na área de processamento de imagens são os chamados operadores de vizinhança. Ou seja, as operações ocorrem dentro de certa vizinhança de um determinado pixel. Essa vizinhança é determinada por um intervalo quadrado ao redor de um pixel central *p*, formando, dessa maneira, uma janela quadrada sobre a imagem, centralizada no pixel *p*.

As máscaras atuam sobre um intervalo de pixels na imagem, intervalo este com a mesma dimensão da matriz quadrada que são formados. Na aplicação de uma máscara sobre uma imagem, cada elemento da sua matriz é multiplicado pelo valor do pixel correspondente. A soma de todos os resultados é armazenada como o novo valor do pixel na imagem de saída, sendo a posição desse pixel correspondente à posição do elemento central da máscara. Um exemplo da aplicação de uma máscara de tamanho 3x3 pixels sobre uma imagem pode ser observado na figura 2.5.

Em uma máscara com dimensões pares, o elemento central, o qual recebe a soma dos resultados dos pixels vizinhos, é o primeiro elemento da matriz. Para máscaras com dimensões ímpares, o elemento central corresponde ao elemento localizado no centro da matriz, ou seja, com mesmo número de elementos acima, abaixo, à direita e à esquerda.

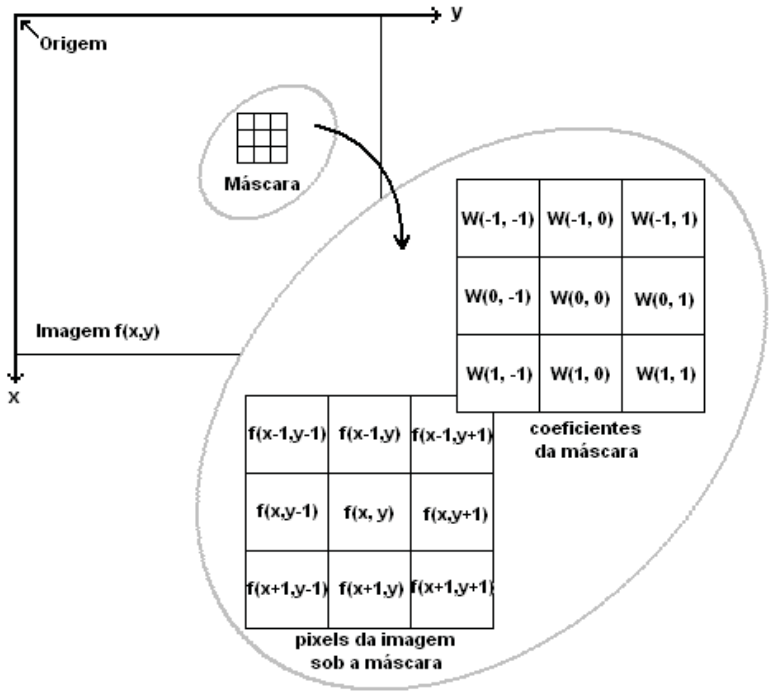


Figura 2.5: Aplicação de uma máscara 3x3 em uma imagem.

2.2 Etapas relevantes do processamento de imagens

Nesta seção serão abordados passos importantes no processamento de imagens para o desenvolvimento da etapa de pré-processamento conforme ilustrado na figura 2.1. Dentre estes passos, ressaltam-se as

técnicas de filtragem de imagens, de segmentação de imagens e suas principais técnicas.

2.2.1 Realce de imagens

As técnicas de realce de imagens visam uma aplicação específica, para qual a imagem realçada seja mais adequada ou mais eficaz que a original. Os métodos de realce de imagens podem ser divididos em dois grupos: um grupo baseado no domínio da frequência e outro grupo baseado no domínio espacial. Técnicas de realce no domínio da frequência são baseadas, principalmente, na modificação das transformadas de Fourier das imagens em processamento. O realce de imagens no âmbito do domínio espacial é baseado na manipulação direta dos pixels da imagem.

2.2.1.1 Realce no domínio da frequência

No domínio da frequência, um dos princípios basea-se no cálculo da transformada de Fourier da imagem a ser aprimorada, multiplicar o resultado pela função de transferência de um filtro e tomar a transformada de Fourier inversa do resultado para produzir a imagem aprimorada.

O conceito da filtragem linear é mais direta e mais simples no domínio da frequência, uma vez que procura-se intensificar ou atenuar as baixas e altas frequências. No entanto, na prática o emprego da transformada de Fourier sobre uma imagem exige um grande esforço computacional, tornando-se uma operação demorada e na maioria dos casos sendo substituída por filtros no domínio espacial.

2.2.1.1.1 Filtragem no domínio da frequência

A utilização de filtros no domínio da frequência é baseada no teorema da convolução. Seja *i(x,y)* uma imagem formada a partir da convolução entre uma imagem *f(x,y)* e um operador linear *h(x,y)*, dada por:

$$i(x,y) = h(x,y) * f(x,y) \tag{5}$$

Assim, pelo teorema da convolução, tem-se a seguinte relação no domínio de frequência:

$$I(u,v) = H(u,v)F(u,v) \tag{6}$$

onde I, H e F são as transformadas de Fourier de i, h e f, respectivamente.

Desta forma, dado um filtro H(u,v) obtém-se a imagem filtrada fazendo-se a transformada inversa de Fourier:

$$i(x,y) = \Im^{-1}[H(u,v)F(u,v)] \tag{7}$$

O problema é escolher um filtro H(u,v) que produza I(u,v) através da atenuação ou aguçamento dos componentes de alta frequência de F(u,v).

Um filtro de passa-baixa usual é o chamado filtro de Butterworth definido por:

$$H(u,v) = \frac{1}{1+[D(u,v)/D_0]^{2n}} \tag{8}$$

onde $D(u,v) = (u^2 + v^2)^{1/2}$, n é a ordem do filtro e D_0 é um limite de corte desejado para o filtro.

O também filtro de Butterworth passa-alta é definido por:

$$H(u,v) = \frac{1}{1+[D_0/D(u,v)]^{2n}} \tag{9}$$

2.2.1.2 Realce no domínio espacial

O termo domínio espacial refere-se ao agregado de pixels que compõem uma imagem. Métodos no domínio espacial operam diretamente sobre os pixels. As funções de processamento de imagens no domínio espacial podem ser expressas como:

$$g(u,v) = T[f(u,v)] \tag{10}$$

onde f(u,v) é a imagem de entrada, g(u,v) é a imagem processada e T é um operador que age sobre f , definido sobre uma vizinhança (u,v) . A abordagem principal para definir uma vizinhança em torno de (u,v) consiste em usar uma máscara quadrada ou retangular centrada em (u,v), para obter *g* naquela posição. Na figura 2.6 observam-se alguns tipos de máscaras quadradas.

w1	w2	w3
w4	w5	w6
w7	w8	w9

3x3

w1	w2	w3	w4	w5
w6	w7	w8	w9	w10
w11	w12	w13	w14	w15
w16	w17	w18	w19	w20
w21	w22	w23	w24	w25

5x5

w1	w2	w3	w4	w5	w6	w7
w8	w9	w10	w11	w12	w13	w14
w15	w16	w17	w18	w19	w20	w21
w22	w23	w24	w25	w26	w27	w28
w29	w30	w31	w32	w33	w34	w35
w36	w37	w38	w39	w40	w41	w42
w43	w44	w45	w46	w47	w48	w49

7x7

Figura 2.6: Típicos arranjos de máscaras para filtragem espacial.

2.2.1.2.1 Filtragem espacial

As operações de filtragem dividem-se entre operações de suavização (*smoothing*) e realce (*sharpening*). Em geral operações de suavização procuram atenuar o nível de ruído nas imagens, removendo pequenos detalhes e suavizando os contornos. Já as operações de realce procuram destacar detalhes da imagem, principalmente cantos e arestas.

2.2.1.2.1.1 Filtros de suavização

Filtros de suavização são usados para borramento e redução de ruídos. O borramento é utilizado para o pré-processamento das imagens a serem tratadas, tais como remoção de pequenos detalhes de uma imagem antes da extração de objetos e conexão de pequenas descontinuidades.

2.2.1.2.1.1.1 Filtragem espacial passa-baixas

Para o desenvolvimento de um filtro passa-baixa, a máscara deve conter todos os valores de seus elementos positivos e a soma desses valores deve ser igual a 1. Outra característica importante desse tipo de filtro é a realização da média da imagem dentro do intervalo da máscara. Dessa maneira, o filtro passa-baixa pode provocar o borramento da imagem (figura 2.7) ou a redução de ruídos (figura 2.8), devido à atenuação de um intervalo específico de pixels contidos na máscara.

$$\frac{1}{N^2}\begin{bmatrix}1 & \cdots & 1\\ \vdots & \ddots & \vdots\\ 1 & \cdots & 1\end{bmatrix}_{NxN}$$

(a) (b) (c)

Figura 2.7: Exemplo de filtro passa-baixa: (a) imagem original, (b) filtro passa-baixa, (c) imagem resultante da aplicação de (b) com N = 15.

$$\frac{1}{25}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1\end{bmatrix}$$

(a) (b) (c)

Figura 2.8: Filtro passa-baixa: (a) imagem original, (b) filtro passa-baixa (c) imagem resultante da aplicação de (b).

2.2.1.2.1.1.2 Filtragem por mediana

O objetivo do uso de filtros por mediana é a remoção de ruídos. Nesse tipo de filtro, o nível de cinza de cada pixel é substituído pela mediana dos níveis de cinza na vizinhança daquele pixel, ao invés da média. Esse método é particularmente efetivo quando o padrão de ruídos apresenta muitos picos, preservando das bordas da imagem filtrada.

Para calcular a filtragem por mediana em uma vizinhança de um pixel, primeiramente selecionamos os valores do pixel e de seus vizinhos, determinamos a mediana e atribuímos este valor ao pixel. Sendo que, a mediana *m* de um conjunto de valores é tal que metade dos valores no conjunto são menores do que *m* e metade são maiores do que *m*.

Na figura 2.9 apresenta-se o resultado da aplicação do filtro por mediana e a comparação da filtragem com o filtro da média ou passa-baixa.

Observam-se diferentes resultados obtidos com a filtragem por mediana em relação à filtragem por média, pois na primeira o nível de cinza de cada pixel é substituído pela mediana dos níveis de cinza na vizinhança, ao invés da média.

(a) (b) (c) (d)

Figura 2.9: Filtro de realce : (a) Imagem original. (b) imagem corrompida por ruído; (c) resultado da média por vizinhança 5x5 de (b); (d) resultado de filtragem por mediana 5x5 de (b).

2.2.1.2.1.2 Filtros de realce

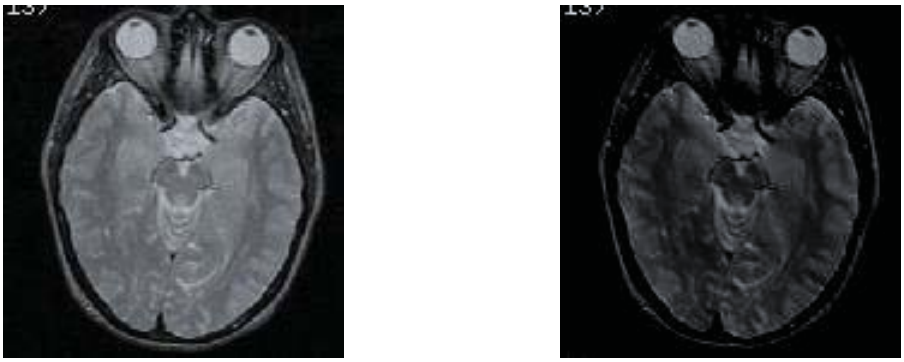
O objetivo principal do realce é enfatizar detalhes finos em uma imagem ou realçar detalhes que tenham sido borrados, em consequência de erros ou como efeito natural de um método particular de aquisição de imagens.

2.2.1.2.1.2.1 Filtragem espacial passa-altas

Um filtro passa-alta deve ter seus coeficientes positivos próximos ao centro e coeficientes negativos na periferia, sendo que a soma de todos os coeficientes deve ser igual a 0. Assim quando a máscara está sobre uma área de níveis de cinza constante ou de pequena variação, a saída da máscara é zero ou muito pequena. Pode-se ainda, gerar a imagem filtrada por passa-altas como a diferença entre a imagem original e a filtrada por passa-baixa da mesma imagem, ou seja:

Passa-altas = original – Passa-baixas.

A figura 2.10 apresenta um exemplo da aplicação de um filtro passa-alta, nota-se que na imagem resultante, o contraste é bastante reduzido, além das bordas estarem realçadas sobre um fundo bastante escuro.



(a) (b)

Figura 2.10: Filtro passa-alta: (a) imagem original; (b) imagem filtrada por passa-alta.

2.2.1.2.1.2.2 Filtragem de alto reforço (high boost)

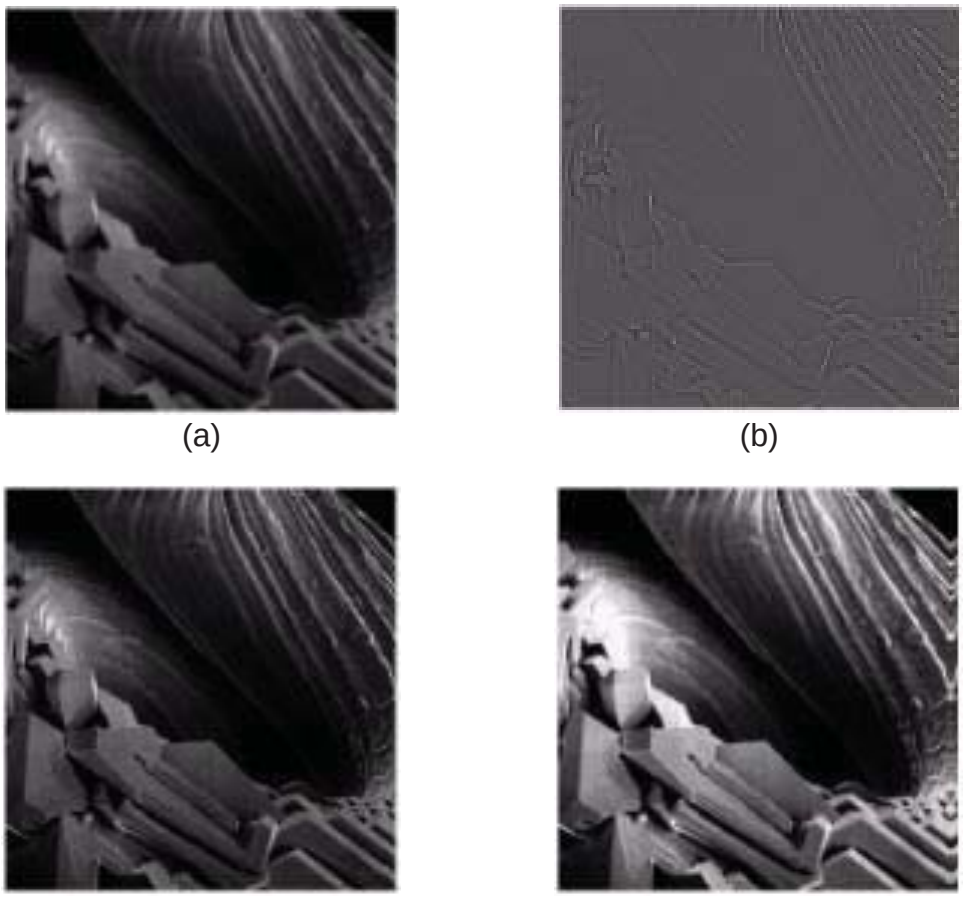
Como foi especificado para o filtro passa-alta, o filtro de alto reforço ou de ênfase de altas frequências pode ser descrito como:

Alto reforço = (A)(Original) – Passa-baixas

Alto reforço = (A – 1)(Original) + Original – Passa-baixas

Alto reforço = (A – 1)(Original) + Passa-altas

O valor A = 1 produz o resultado padrão passa-altas. Quando A > 1, parte do original é adicionado de volta ao resultado passa-altas, o que restaura parcialmente os componentes de baixa-frequência perdidos na operação de filtragem passa-altas. O resultado é que a imagem alto reforço assemelha-se mais à imagem original, com relativo grau de realce de bordas o qual depende do valor de A. A figura 2.11 apresenta um exemplo de aplicação do filtro de alto reforço.



(a) (b)

(c) (d)

Figura 2.11: Filtro de alto reforço: (a) imagem original, (b) imagem filtrada por alto reforço com A = 0;
(c) imagem filtrada por alto reforço com A = 1; (d) imagem filtrada por alto reforço com A = 1,7.

2.2.1.2.1.2.3 Filtro por derivadas

Os detectores de borda formam um conjunto de métodos de pré-processamento utilizados para detectar mudanças abruptas na intensidade dos brilhos. Uma borda ou aresta corresponde a uma descontinuidade na intensidade dos tons de cinza dos pixels que formam a imagem. Os operadores mais utilizados para detectar esta descontinuidade são os que fazem uso da primeira e segunda derivadas.

As principais etapas no processo de detecção de bordas são: a filtragem, o realce e a limiarização. A filtragem é utilizada na diminuição ou eliminação de ruídos presentes na imagem, visto que os ruídos podem produzir alterações no gradiente resultando na identificação de bordas falsas. A etapa de realce é caracterizada pelo cálculo da magnitude do

gradiente e a limiarização consiste na determinação das bordas que serão consideradas [Roosevelt, 2007].

A detecção de bordas por operações de gradiente tende a funcionar bem em situações envolvendo imagens com transições agudas de intensidade e ruído relativamente baixo. Cruzamentos por zero oferecem uma alternativa nos casos em que as bordas forem borradas ou quando um alto conteúdo ruidoso estiver presente. Os cruzamentos por zero permitem o posicionamento confiável das bordas, e as propriedades suavizantes de ∇^2h, reduzem os efeitos do ruído. O custo por essas vantagens é o aumento da complexidade e do tempo computacional [Roosevelt, 2007]. Uma vez que a primeira derivada da imagem convolucionada da função gaussiana, é equivalente a imagem convolucionada da primeira derivada da função de Gauss, é possível combinar os estágios de detecção e suavização em uma simples convolução em 1-D, cada convolução com a primeira derivada da função de Gauss e considerando os picos, ou com a segunda derivada considerando os cruzamentos dos zeros [Sonka et al, 1998].

2.2.1.2.1.2 Processamento do Histograma

O histograma de uma imagem digital descrita em tons de cinza é a distribuição da quantidade de cada um desses tons dentro da imagem. Assim, o histograma fornece a informação sobre quantos pixels na imagem possui um determinado valor de tom de cinza. No caso de imagens de 8 bits, esses valores variam de 0 a 255.

Ao se observar o histograma de uma imagem, tem-se uma noção instantânea sobre as características da mesma. Dessa forma, através do histograma obtemos informações sobre o contraste de uma imagem, ou seja, quanto maior o espalhamento ao longo do eixo dos tons de cinza, maior o contraste da imagem.

É comum a normalização de histogramas por meio da divisão de cada valor de cinza pelo número total de pixels da imagem, denotado por *n*. Assim, o histograma normalizado é dado por $p(r_k) = n_k / n$, para $k = 0, 1, ..., L-1$, onde *rk* corresponde ao tom de cinza da imagem. Dessa forma, o histograma retrata a probabilidade de ocorrência de cada nível de cinza dentro dessa imagem.

2.2.1.2.1.2.1 Equalização de Histogramas

A equalização de histograma é uma técnica baseada na redistribuição dos valores de tons de cinza em uma imagem visando à obtenção de um histograma uniforme com valores parecidos para todos os tons de cinza [Marques e Vieira Neto Filho, 1999]. Para essa redistribuição, utiliza-se uma função de distribuição acumulada (*Cumulative Distribution Function* - CDF) da distribuição de probabilidade original, a qual pode ser expressa por:

$$s_k = T(r_k) = \sum_{j=0}^{k} \frac{n_j}{n} = \sum_{j=0}^{k} p_{r(r_j)} \quad (11)$$

Onde, $0 \leq r_k \leq 1$ e $k = 0, 1, ..., L-1$.

Na figura 2.12 observa-se a aplicação da equalização de histogramas em imagens em tons de cinza.

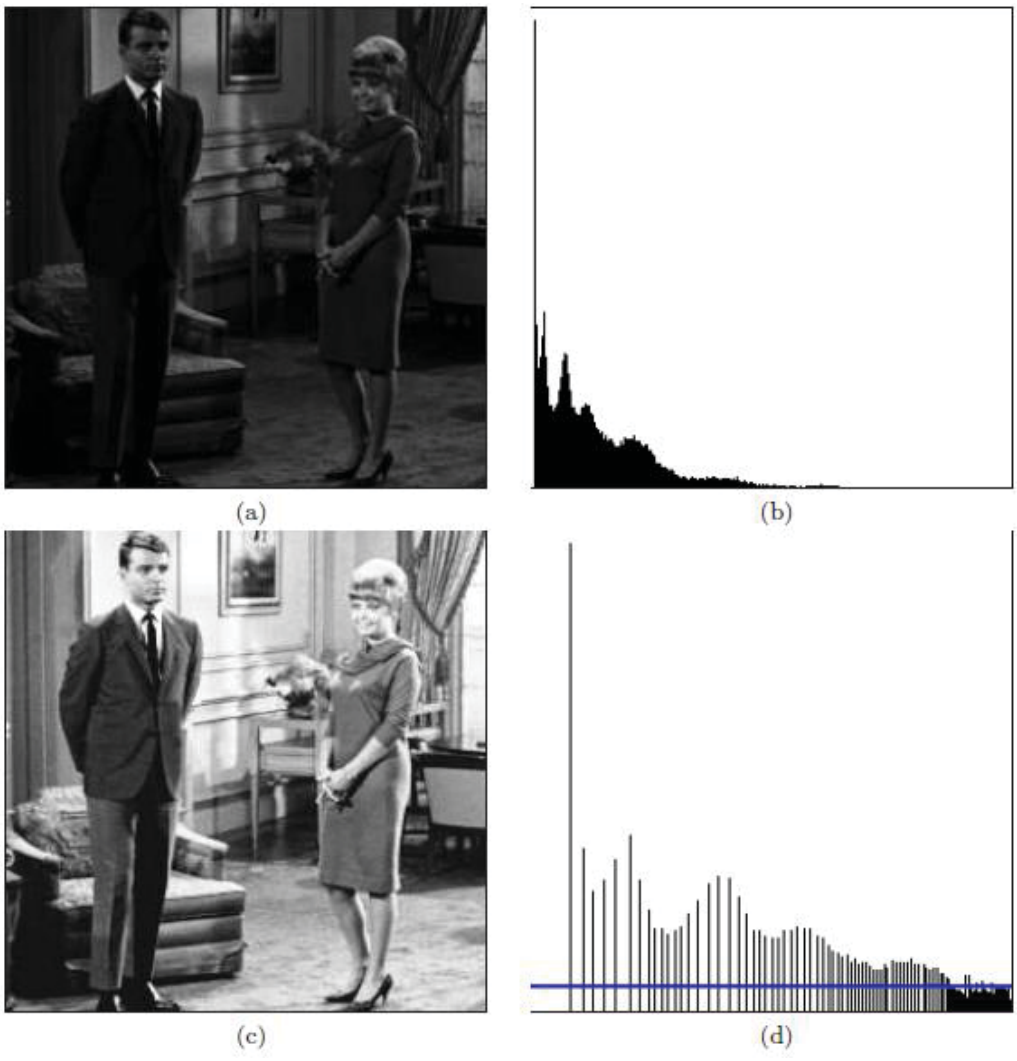


Figura 2.12: Ilustração sobre equalização de histogramas. (a) Imagens originais. (b) Histograma da imagem original. (c) Imagem com histograma equalizado. (d) Histograma da imagem processada, onde a linha azul indica uma distribuição uniforme [Menotti, 2008].

2.2.2 Segmentação

O objetivo das técnicas de segmentação é dividir a imagem em suas diversas partes ou segmentos (objetos e regiões). O nível ou quantidade de divisões aplicadas na imagem varia conforme a aplicação de segmentação, e em geral é realizada até atingir um nível de separação suficiente entre os objetos de interesse na cena analisada [Santos, 2002].

Os algoritmos de segmentação baseiam-se principalmente em duas propriedades do nível de intensidade luminosa das imagens: a descontinuidade e a similaridade. Quanto à descontinuidade, a idéia está em dividir a imagem em regiões de acordo com as mudanças abruptas do nível de intensidade luminosa em seus pontos, por exemplo, cantos e arestas de objetos na imagem. Já por similaridade, a imagem é dividida em regiões de acordo com algum padrão de semelhança entre estas regiões, como por exemplo, o nível de intensidade luminosa, a cor e a textura.

Na segmentação de imagens procura-se distinguir as partículas umas das outras e as de fundo. Esta distinção permitirá interpretar pixels contíguos e agrupá-los em regiões. Não existe um modelo formal para a segmentação, o processo é essencialmente empírico e deve se ajustar a diferentes tipos de imagem.

Dentre as técnicas de segmentação mais conhecidas destacam-se:

- a detecção de descontinuidades;
- a detecção de limiares (*thresholding*);
- a identificação de regiões;
- a caracterização de divisores de água (*watersheds*).

2.2.2.1 Detecção de Descontinuidades

A segmentação por detecção de descontinuidades procura regiões de transição abrupta do nível de intensidade luminosa dos pontos da imagem para realizar as divisões. Os algoritmos para detecção de descontinuidades se prestam a [Roosevelt, 2007]:

a) Detecção de pontos: um ponto será identificado em função da mudança drástica do valor de cinza em relação aos seus vizinhos;

b) Detecção de linhas: basicamente reconhece pontos semelhantes e verifica se pertencem a uma mesma linha;

c) Detecção de bordas: é uma das técnicas básicas utilizadas pela visão humana no reconhecimento de objetos, compreende a localização e realce dos pixels de fronteira dos objetos, aumentado o contraste entre seus limites e o fundo.

2.2.2.1.1 Detecção de pontos

A detecção de pontos isolados em uma imagem pode ser obtida de maneira direta utilizando-se máscaras que evidenciam a diferença do tom de cinza de um dado pixel dos de seus vizinhos. A figura 2.13 mostra uma máscara usada para a detecção de pontos isolados a partir de um fundo constante.

-1	-1	-1
-1	8	-1
-1	-1	-1

Figura 2.13. Máscara para detecção de pontos.

2.2.2.1.2 Detecção de linhas

A detecção de linhas em uma imagem é mais complexa do que a detecção de pontos nessa mesma imagem. Para a detecção de linhas há a necessidade de uma máscara específica para o realce de uma determinada reta, ou seja, a composição da máscara dependerá da inclinação da reta. Na figura 2.14 observam-se algumas máscaras para a detecção de retas horizontais, com ângulo de +45º, verticais e com ângulo de -45º, respectivamente.

-1	-1	-1
2	2	2
-1	-1	-1

Horizontal

-1	-1	2
-1	2	-1
2	-1	-1

+45º

-1	2	-1
-1	2	-1
-1	2	-1

Vertical

2	-1	-1
-1	2	-1
-1	-1	2

-45º

Figura 2.14. Máscara para deteccção de linhas.

2.2.2.1.3 Detecção de bordas

Bordas indicam variações significantes dentro da imagem em âmbito local e são de grande importância para a análise dessas imagens. Tipicamente as bordas estão localizadas nas fronteiras entre duas ou mais regiões na imagem, destacando cenas ou objetos. A detecção de bordas é frequentemente o primeiro passo visando à obtenção de informações nas imagens a serem tratadas [Sonka et al, 1998].

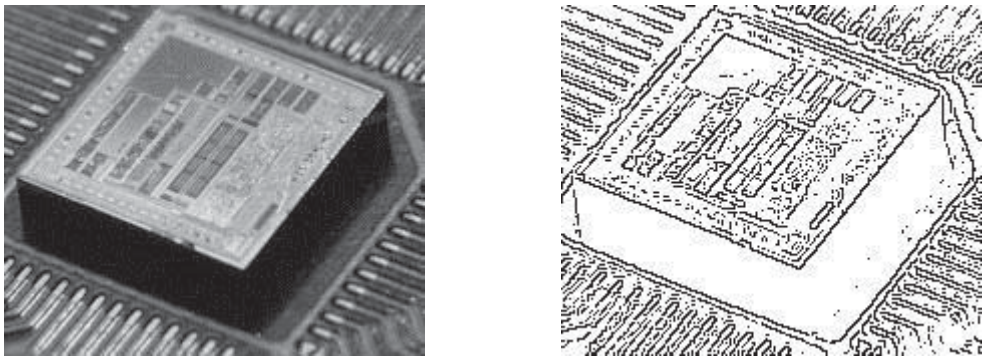
As principais etapas no processo de detecção de bordas são: a filtragem, o realce e a limiarização. A filtragem é utilizada na diminuição ou eliminação de ruídos presentes na imagem, já que os ruídos podem produzir alterações no gradiente, o que pode resultar na identificação de bordas falsas. A etapa de realce é caracterizada pelo cálculo da magnitude do gradiente, já a limiarização consiste na determinação das bordas que serão consideradas.

2.2.2.1.3.1 Operador de Roberts

O operador de Roberts é um dos operadores de gradiente mais tradicionais no processamento de imagens digitais. Sua implementação é bastante simples, através da aplicação do operador 1. Como resultado de sua aplicação, obtém-se uma imagem com altos valores de nível de cinza, em regiões de limites bem definidos e valores baixos em regiões de limites suaves, sendo 0 para regiões de nível de cinza constante.

$$G'_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad G'_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \qquad \text{Op. 1}$$

Na figura 2.15 observa-se o resultado da aplicação do operador de Roberts.



(a) (b)

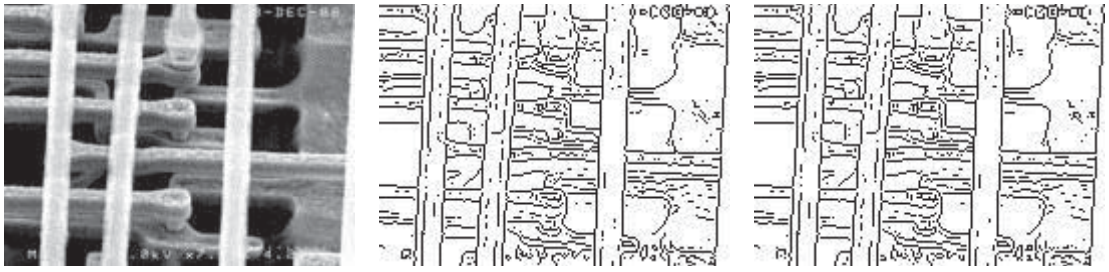
Figura 2.15: Aplicação de filtro de Roberts: (a) imagem original; (b) imagem resultante da aplicação do operador de Roberts em (a).

2.2.2.1.3.2 Operador de Sobel

O operador de Sobel é frequentemente usado para detecção de bordas horizontais e verticais, através do cálculo os gradientes da função em relação ao elemento central da máscara. Sua implementação computacional é simples, a qual pode ser especificada a partir do operador 2.

$$G'_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad G'_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \qquad \text{Op. 2}$$

Na figura 2.16 observa-se o resultado da aplicação dos operadores Sobel e Prewitt.



(a) (b) (c)

Figura 2.16: Aplicação de filtro detectores de borda: (a) imagem original; (b) imagem resultante da aplicação do operador Sobel em (a); (c) imagem resultante da aplicação do operador Prewitt em (a).

2.2.2.1.3.3 Operador de Prewitt

O operador de Prewitt funciona de maneira similar ao operador de Sobel, porém sem enfatizar os pixels próximos ao centro da máscara.

$$G'_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad G'_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ -1 & -1 & -1 \end{bmatrix} \qquad \text{Op. 3}$$

2.2.2.1.3.4 Operador Laplaciano

Com o intuito de visualizar a construção de uma máscara do operador laplaciano , dado pelo operador 4, utilizou-se o modelo mais simples deste tipo de máscara, ou seja, uma máscara 3x3 com vizinhança de 4 pixels para seu elemento central [Sonka et al, 1998].

$$\nabla^2 f = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad \text{Op. 4}$$

2.2.2.1.3.5 Operador de Nevatia e Babu

O conjunto de máscaras utilizadas por Nevatia e Babu é construído de tal forma a apresentar maior resposta para linhas com diversos ângulos de inclinação. O operador de Nevatia e Babu utiliza 12 máscaras para a detecção de linhas com inclinações variando de 30º em 30º (0º a 330º).

Neste operador, é realizado um processo de convolução, sendo calculada uma magnitude através de cada máscara e, para cada pixel a magnitude é dada pela maior resposta calculada e a direção é dada pelo ângulo associada à máscara de maior resposta.

2.2.2.2 Detecção de Limiares (*Thresholding*)

A detecção de limiares é uma das técnicas de segmentação mais simples e fáceis de aplicar, sendo utilizada constantemente em qualquer aplicação de processamento de imagens. A técnica procura agrupar os diferentes objetos e regiões da imagem conforme a similaridade de tonalidades (nível de intensidade luminosa) entre os mesmos [Santos, 2002].

A operação de limiarização simples corresponde à definição de um valor *T* de limiar para o nível de intensidade luminosa ao qual se deseja realizar o ponto de corte (binarização) da imagem. Em uma imagem de 256 tons de cinza, caso *T* =50, todas as tonalidades entre 0 e 50, inclusive, se tornarão informação de fundo da imagem (valor binário 0, ou cor preta). Já

os demais valores a partir deste limiar se tornarão informação correspondente aos objetos da imagem (valor binário 1, ou cor branca).

Dessa forma, a simples definição de um valor de limiar ou de corte já é suficiente para dividir a imagem em duas regiões: fundo e objetos, sendo útil quando já existe bom contraste entre estas regiões e apresentando resultados pobres em situações mais adversas. É possível ainda definir múltiplos limiares de corte (*multilevel thresholding*), e definir faixas de valores de intensidade luminosa para cada entidade da imagem. Por exemplo, caso existam para uma imagem $f(x,y)$ dois valores distintos de limiar ($T1$ e $T2$), define-se que todos os valores em que $f(x,y)<T1$ correspondem ao fundo da imagem, enquanto os valores em que $T1 \geq f(x,y) > T2$ correspondem a um determinado objeto, e quando $f(x,y) \geq T2$ há um outro objeto correspondente na cena. É possível definir mais valores de limiar, porém, situações que demandam diferentes valores de limiar para realizar sua segmentação são normalmente melhor solucionadas por técnicas de segmentação baseadas em regiões.

A figura 2.17 ilustra estas duas situações descritas acima. A imagem original à esquerda têm seus tons de cinza quantizados no histograma mostrado ao centro; e a esquerda observa-se a imagem binarizada a partir do ponto ou pontos de limiar escolhido no histograma.

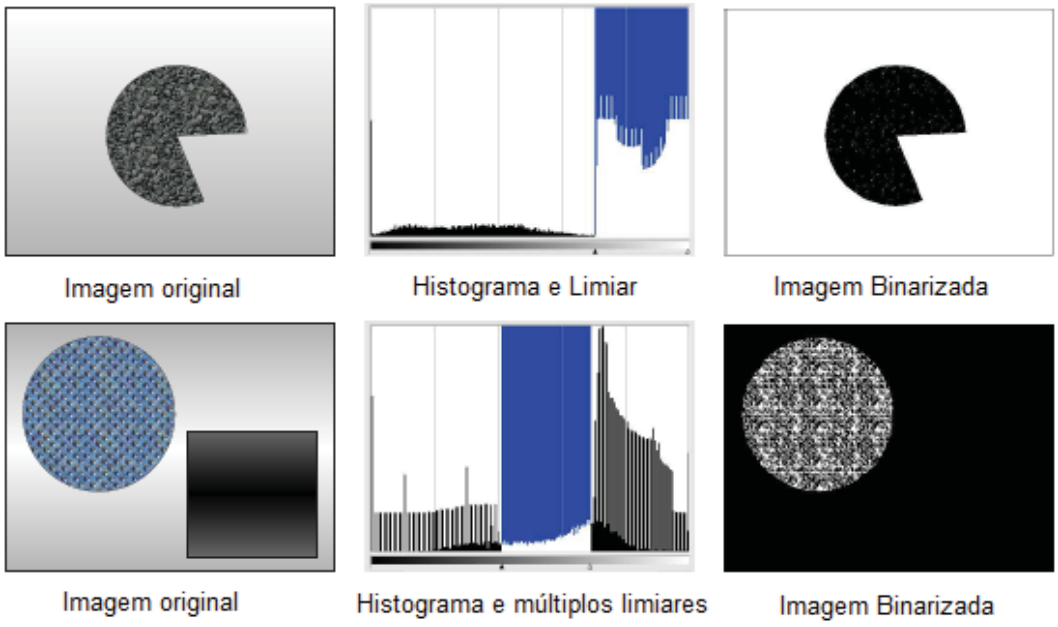


Figura 2.17: Técnicas de detecção de limiares.

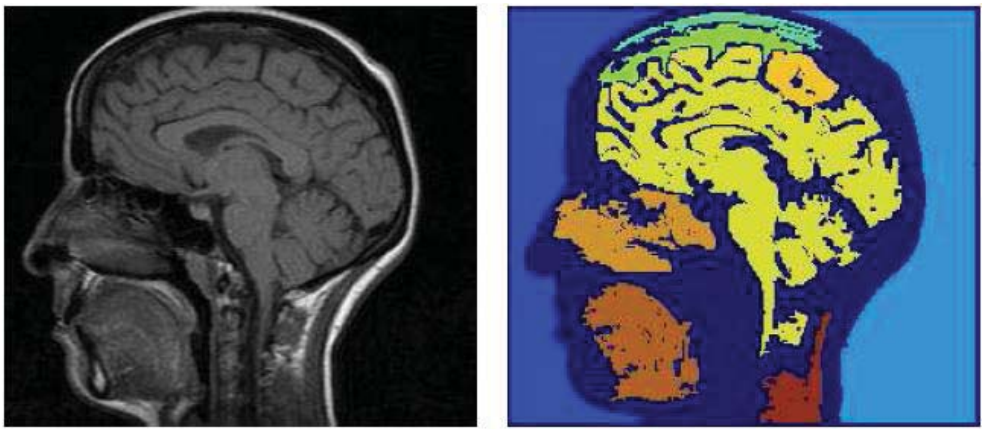
2.2.2.3 Segmentação Baseada em Regiões

Este tipo de segmentação divide a imagem procurando por regiões que atendam a algum tipo de similaridade. Em geral, uma imagem segmentada é formada por um número *n* de regiões de pontos na imagem, sendo que a união destas regiões compõe a imagem completa. As regiões devem sempre ser disjuntas e atender a um determinado critério para o agrupamento de seus pontos. Duas técnicas são bem conhecidas: segmentação por crescimento de regiões (*region growing*) e por divisão e agrupamento de regiões (*split and merge*).

2.2.2.3.1 Segmentação por crescimento de regiões

A segmentação por crescimento de regiões é um procedimento que agrupa pontos e pequenas sub-regiões da imagem em maiores regiões, de acordo com um critério de semelhança pré-definido. Em geral, parte-se de um conjunto de pontos ditos sementes, e a partir destes pontos inicia-se um processo de crescimento de regiões, agrupando todos os pontos vizinhos que respeitem o critério de semelhança com a semente. É comum que para cada aplicação tenha-se uma ligeira noção de quais pontos serviriam como boas sementes para o início do algoritmo.

A escolha pelo critério de similaridade entre os pontos varia conforme a aplicação e o tipo de imagem sendo usada. Dentre os critérios mais utilizados destacam-se a intensidade luminosa, a textura, e a cor dos pontos. Alguns outros critérios fornecem ao algoritmo certa inteligência para a reunião dos pontos, como analisar as propriedades de adjacência e conectividade dos mesmos enquanto forma-se a região. Os critérios para a parada do algoritmo costumam ser o tamanho e a forma das regiões, assim como a similaridade de características de novos pontos para adentrar uma determinada região. A segmentação por crescimento de regiões é ilustrada na figura 2.18(b).



(a) (b)

Figura 2.18: Segmentação por crescimento de regiões: (a) imagem original; (b) imagem segmentada.

2.2.2.3.2 Segmentação por divisão e agrupamento

A segmentação por divisão e agrupamento aborda uma solução diferente para classificar e agrupar regiões da imagem. A idéia consiste em iniciar uma série de divisões sistemáticas a partir da imagem original, até alcançar inúmeras divisões distintas que realmente formem regiões na imagem. Ou seja, nestas divisões todos os pontos devem obedecer a um critério de similaridade para poderem manter-se agrupados. A estrutura de dados mais utilizada para representar as divisões na imagem são as árvores do tipo *quadtree*, que divide sempre a imagem em quatro partes iguais. Após uma divisão, caso a condição de similaridade não se verifique para todas as partes separadas, cada parte prossegue sendo dividida obedecendo esta mesma estrutura até atingir diversas regiões, que reunidas formam a imagem completa.

Quando não há mais possibilidade de realizar divisões, ou seja, todas as divisões já formam regiões distintas, inicia-se o processo de agrupamento das regiões vizinhas que atendam ao mesmo critério de similaridade de agrupamento dos pontos, para realmente formar um grupo de regiões maiores que caracterizem de maneira mais visível as diferentes informações da imagem. O procedimento termina quando não é mais possível agrupar nenhuma região da imagem. O processo de segmentação por divisão e agrupamento é ilustrado na figura 2.19.

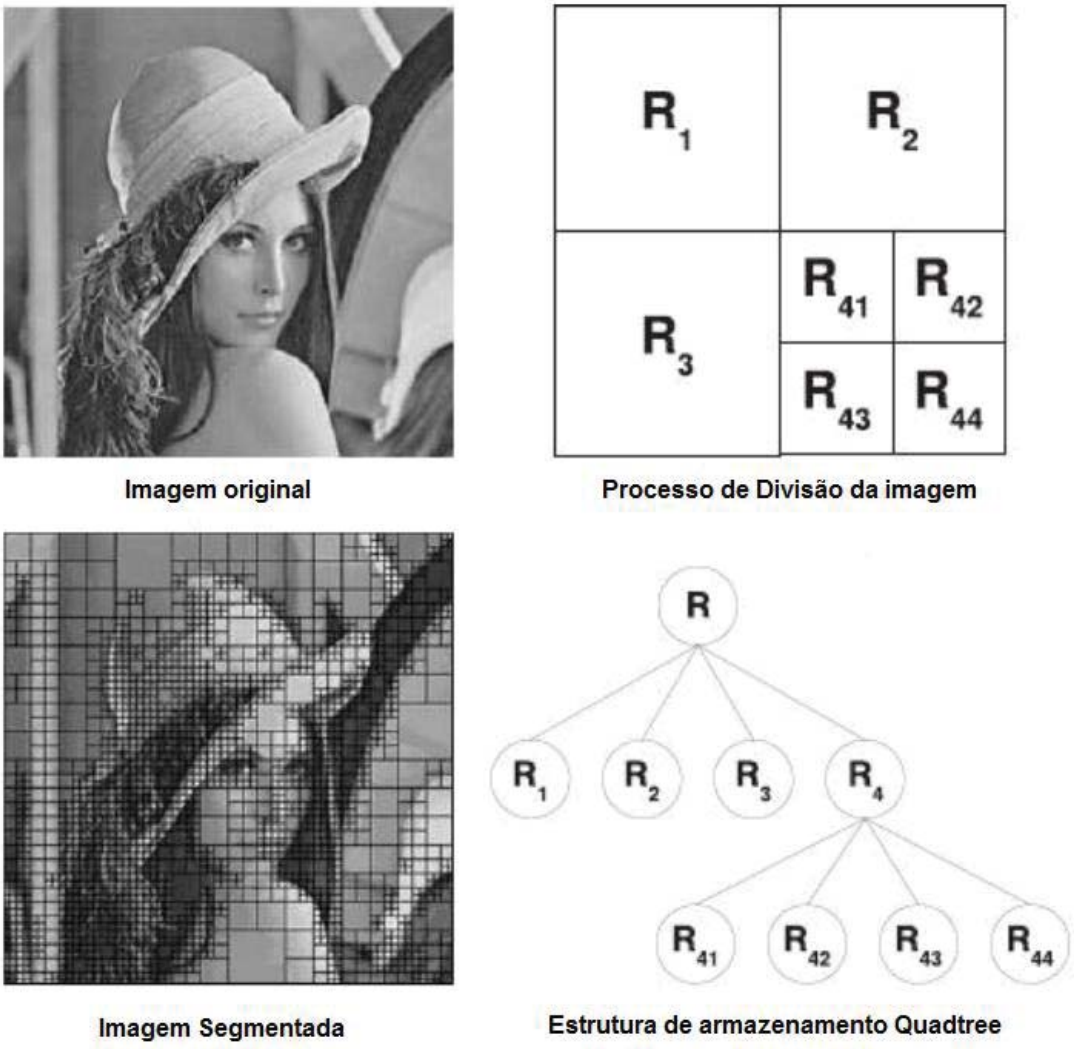


Figura 2.19: Processo de segmentação por divisão e agrupamento.

2.2.2.4 Segmentação por Divisores de Água (Watersheds)

A segmentação por divisores de água abrange conceitos de todas as demais técnicas de segmentação para realizar a divisão da imagem em partes distintas. O nome dado à técnica provém do seu princípio de funcionamento, que consiste em visualizar qualquer imagem em uma representação tri-dimensional da mesma, considerando as coordenadas espaciais *x* e *y* nos planos horizontal e vertical, respectivamente, e ainda o nível de intensidade luminosa dos pontos nestas coordenadas no plano perpendicular aos demais, provendo a idéia de profundidade na imagem [Soon, 2007].

De acordo com esta representação topográfica da imagem, consideram-se três elementos:

(a) pontos da imagem que pertençam a um mínimo local;

(b) pontos intermediários, nos quais se colocada uma gota d'água sobre o ponto, esta escorreria em direção de um mínimo local;

(c) pontos que pertençam a máximos locais, nos quais, se colocada uma gota d'água sobre o ponto, a gota teria a possibilidade de escorrer para mais de um mínimo local.

Os pontos que satisfazem a condição (b) na imagem são chamados de represa ou divisores de águas. Já os pontos que satisfazem a condição (c) são chamados de linhas divisórias de águas, e o algoritmo está interessado justamente em encontrar estas linhas divisórias, pois elas limitam e representam as bordas de objetos na imagem.

2.2.3 Tratamento de Imagens Coloridas

O uso das cores no processamento de imagens é fortemente motivado, principalmente, por dois fatores. Primeiro, a cor auxilia na identificação de um objeto e sua extração de uma determinada cena. E, segundo, na capacidade humana de distinguir milhares de nuances de cores comparadas à capacidade de perceber apenas duas dezenas de tons de cinza.

Dessa forma, o processamento de imagens coloridas é dividido em duas grandes categorias: o processamento de imagens coloridas ou de imagens pseudo-coloridas. Na primeira categoria, as imagens são adquiridas através de um sensor colorido, como uma câmera ou um digitalizador. Na segunda categoria, o problema é especificar uma cor a uma determinada intensidade monocromática ou uma variedade de intensidades. Porém, atualmente, com a popularização das câmeras e dispositivos similares, o processamento de imagens está baseado nas imagens coloridas e não nas pseudo-coloridas, como acontecia décadas atrás [Gonzalez e Woods, 2002].

2.2.3.1 Fundamentos das cores

As cores que os olhos humanos percebem são determinadas pela luz que é refletida por um determinado objeto. Logo, a caracterização da fonte de luz é o cerne do estudo das cores. Porém, antes de se pensar de conceito de cores, deve-se abordar o conceito de luz. Pois em dados momentos a luz se comporta como uma partícula e ora como onda. Assim, para estudar as cores, deve-se observar a luz quando ela se comporta como onda [Scuri, 2002].

Dependendo do comprimento de onda apresentado pela luz em dado instante, o olho humano percebe um espectro diferente de luz, que representa uma determinada cor. O olho humando percebe a luz entre aproximadamente 380*nm* e 780nm. Luz de 380*nm* até 500*nm* parece azul, de 500*nm* a 600*nm* parece verde, e de 600*nm* a 780*nm* parece vermelho. O restante do espectro de cores é ocupado por outros tipos de ondas que não podem ser visualizadas pelos seres humanos.

Na figura 2.20, observa-se alguns comprimentos de ondas e as cores reconhecidas pelos olhos humanos.

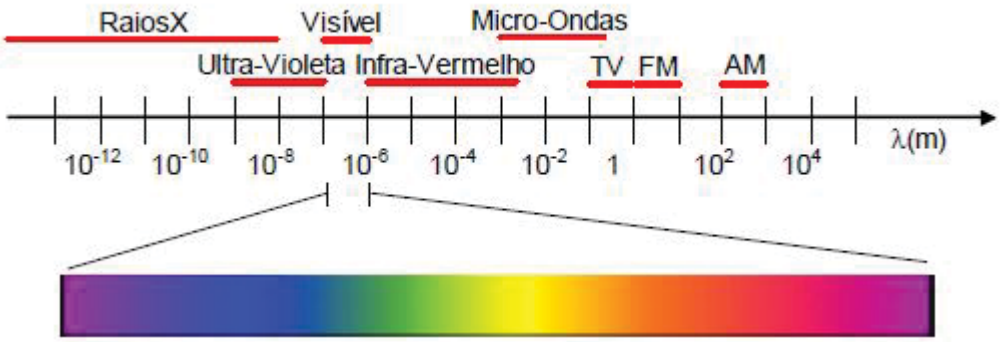


Figura 2.20: Comprimentos de ondas do espectro eletromagnético [Scuri, 2002].

Existem três medidas utilizadas para descrever a qualidade de uma fonte de luz: radiância, luminância e brilho. Radiância é a quantidade de energia que flui da fonte de luz, sendo medida em *watts* (*W*). Luminância, medida em *lumens* (*lm*), é a energia que o observador percebe da fonte de luz. E o brilho é um descritor subjetivo difícil de ser medido, que engloba a noção de intensidade da cor, um dos principais fatores na descrição das cores.

Entretanto, as características mais comumente utilizadas para distinguir as cores são o brilho, o matiz e a saturação. Como dito anteriormente, o brilho engloba a noção de intensidade. O matiz é o atributo associado à forma de onda dominante, que representa a cor dominante percebida pelo observador. E a saturação refere-se à pureza da cor ou à quantidade de branco misturada ao matiz daquela cor. Então, cores como rosa e lilás são variações menos saturadas do vermelho e do roxo, respectivamente, onde o grau de saturação é inversamente proporcional à quantidade de branco adicionado à cor original.

2.2.3.2 Modelos de Cores

A finalidade de um modelo de cor é a padronização e especificação das cores segundo critérios definidos e de aceitação geral. O modelo de cor consiste, basicamente, de um sistema de coordenadas tridimensionais e um subespaço deste sistema de coordenadas onde cada cor é representada por um único ponto. Os sistemas de cores atualmente em uso estão orientados ou direcionados segundo seu emprego em termos de *hardware*.

O sistema RGB (*red*, *green*, *blue*) é principalmente destinado à visualização em monitores e vídeos, sejam de tubos catódicos ou de LCD (*Liquid Cristal Display*). O sistema CMY (*cyan*, *magenta*, *yellow*) está vinculado às impressoras coloridas. Destacam-se ainda o modelo HSI (matiz, saturação e intensidade) e o modelo HSV (matiz, saturação e valor). Na figura 2.20 e 2.21 observa-se uma representação dos modelos RGB e HSV, respectivamente.

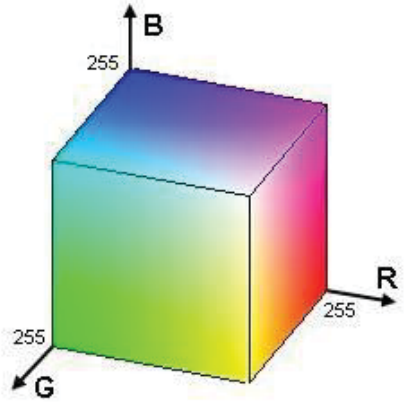


Figura 2.21: Modelo RGB [Brys, 2008].

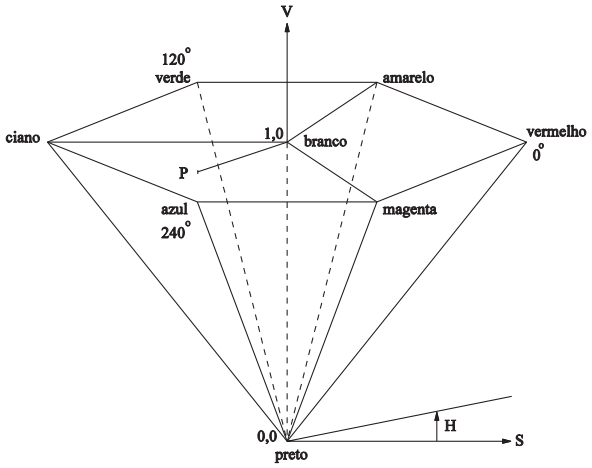


Figura 2.22: Modelo HSV [Brys, 2008].

2.2.3.3 Conversões entre Modelos de Cores

As imagens podem estar descrita nos mais diversos modelos de cores. Dessa forma, em alguns casos é necessária a conversão entre um modelo e outro para evidenciar alguma característica da imagem. A seguir será descrita a conversão entre os modelos RGB e HSV.

2.2.3.3.1 Conversão RGB para HSV

Dada uma cor definida por RGB, onde esses parâmetros estão entre 0.0 e 1.0, sendo o menor e maior valor possível para cada parâmetro, respectivamente. A transformação para os parâmetros HSV dessa cor pode ser determinada pelas equações abaixo.

Seja MAX e MIN, os valores máximo e mínimo, respectivamente, dos valores RGB:

H = 60.[(G-B)/(MAX-MIN)] + 0, se MAX=R e G>=B

H = 60.[(G-B)/(MAX-MIN)] + 360, se MAX=R e G<B

H = 60.[(B-R)/(MAX-MIN)] + 120, se MAX=G

H = 60.[(R-G)/(MAX-MIN)] + 240, se MAX=B

S = (MAX-MIN)/MAX

V = MAX

Os resultados dão a tonalidade variando de 0º a 360º, indicando o ângulo no círculo onde a tonalidade H está definida, e a saturação e o brilho

variando de 0.0 a 1.0, representando o menor e o maior valor possível, respectivamente.

2.2.3.3.2 Conversão HSV para RGB

Seja uma cor definida por HSV, onde H varia de 0.0 a 360.0, informando o ângulo em graus no círculo onde esse parâmetro está definido, e com S e V variando de 0.0 a 1.0. A transformação para os parâmetros RGB dessa cor podem ser calculados conforme as equações abaixo.

Primeiramente, se S = 0, o resultado será cinza. Para esse caso, os valores de RGB são iguais a V, e o valor do parâmetro H é irrelevante.

Para S diferente de zero, as equações abaixo são aplicáveis:

Hi = (H/60) mod 6

f =(H/60) – Hi

p = V.(1-S)

q = V.(1- f.S)

t = V.(1-(1-f).S)

R = V, G= t e B = p, se Hi=0

R = q, G= V e B = p, se Hi=1

R = p, G= V e B = t, se Hi=2

R = p, G= q e B = V, se Hi=3

R = t, G= p e B = V, se Hi=4

R = V, G= p e B = q, se Hi=0

Estas equações dão RGB variando de 0.0 a 1.0.

CAPÍTULO 3 – Metodologia de Desenvolvimento de Circuitos Digitais

Houve um grande avanço no nível de densidade e no desempenho dos circuitos integrados nas duas últimas décadas. Nos anos 60, Gordon Moore, até então funcionário da Fairchild Corporation e mais tarde co-fundador da Intel, previu que o número de transistores que poderiam ser integrados em um único *chip* iria crescer exponencialmente com o tempo. Essa previsão, chamada Lei de Moore, foi comprovada ao longo do tempo e pode ser observada através do gráfico na figura 3.1. Neste gráfico pode-se observar a quantidade de transistores em alguns computadores famosos. Observa-se ainda que a complexidade nos circuitos duplica a cada 1 ou 2 anos [Habaye et al, 2003].

Esta evolução apresenta um profundo impacto em como os circuitos são desenvolvidos, pois com a rápida mundança das tecnologias, o tempo de desenvolvimento dos circuitos é um fator crucial para o sucesso do componente. Por esse motivo, os projetistas estão aderindo a metodologias e estratégias que facilitem o processo de desenvolvimento dos circuitos.

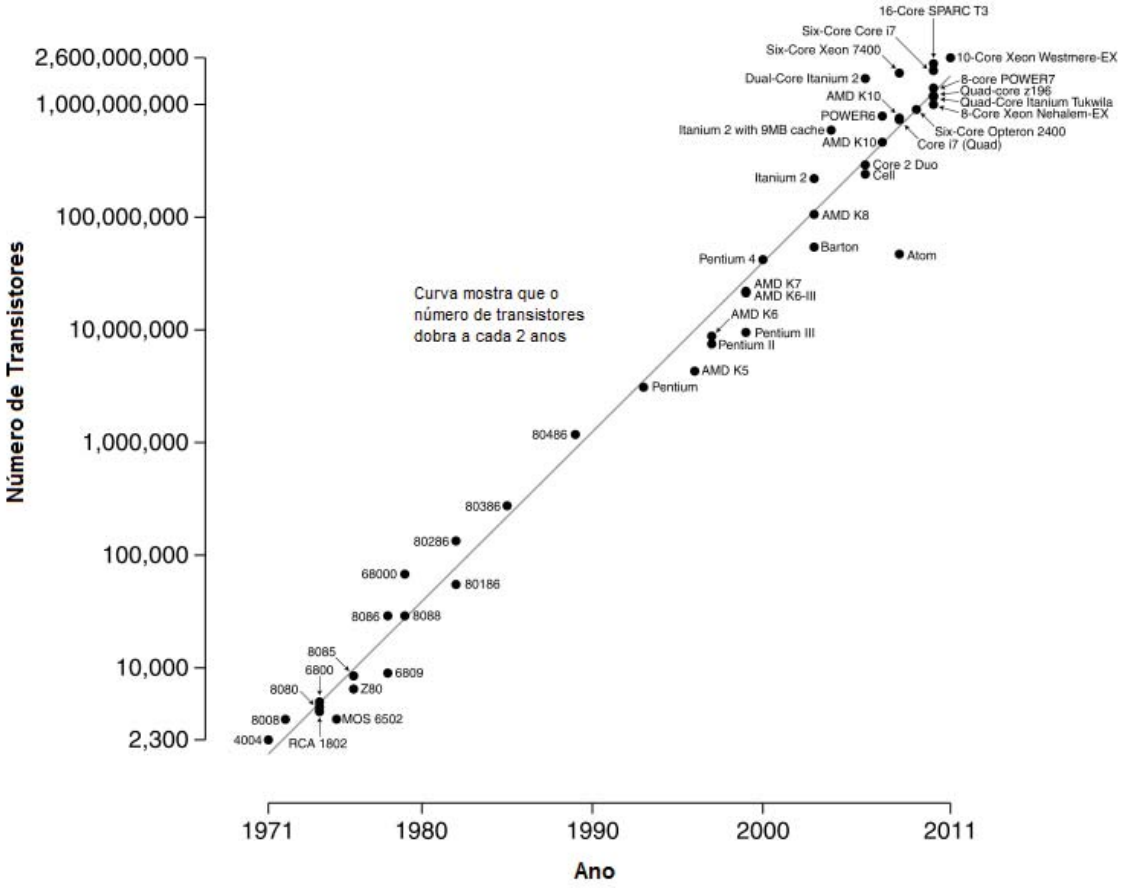


Figura 3.1. Lei de Moore. [Habaye et al, 2003]

Uma das metodologias que podem ser utilizadas é a metodologia *top-down*. Nesta metodologia, inicialmente, um modelo comportamental do sistema deve ser escrito e simulado. Os próximos passos seguem com o intuito de diminuir o nível de abstração do modelo inicial, até chegar a um circuito descrito somente com transistores. A metodologia *top-down* é melhor aplicada a sistemas com requisitos de funcionamento bem definidos.

Outra metodologia utilizada é a metodologia *bottom-up*. Nesta metodologia as metas de funcionamento do sistema são modificadas ao longo do processo de desenvolvimento do circuito, visando obter o melhor desempenho possível com o menor custo. Neste tipo de sistema, um componente novo é desenvolvido somente se o custo para desenvolvê-lo for menor do que o custo de agregar um componente já existente ao sistema. Assim, a viabilidade de um componente afeta diretamente a arquitetura desse tipo de produto [Munden, 2005].

As técnicas acima descritas podem ser utilizadas para o desenvolvimento de um sistema contendo um circuito integrado de aplicação específica (*Application-Specific Integrated Circuits* – ASICs), ou para um dispositivo lógico programável (*Programmable Logic Devices* – PLDs).

3.1 Dispositivos para Circuitos Integrados

Os dispositivos eletrônicos encontrados no mercado possuem determinadas funcionalidades definidas pelo seu fabricante. Em contrapartida, os ASICs e dispositivos lógico-programáveis podem ter suas funcionalidades determinadas pelo projetista de acordo com a aplicação. Porém, os ASICs requerem um processo final de manufatura para tornarem-se operacionais, enquanto os PLDs tornam-se funcionais logo após o seu desenvolvimento.

3.1.1 ASICs

Um circuito integrado para uma aplicação específica (ASIC) é um dispositivo eletrônico desenvolvido para um uso determinado, ao contrário dos dispositivos para uso geral. Os ASICs mais modernos incluem microprocessadores, blocos de memória como RAM, ROM e *Flash*, entre outros.

Os ASICs desenvolvidos no início da década de 80, utilizavam uma tecnologia baseada no arranjo de portas lógicas (*gate array*) pré-fabricadas. Nesse tipo de tecnologia, as células básicas ficavam distribuídas ao longo da pastilha de silício sendo conectadas por trilhas de metais dedicados à sua comunicação. As células básicas eram compostas por um número determinado de transistores. Os circuitos que utilizavam esse tipo de tecnologia variavam entre poucas centenas a dezenas de milhares de portas lógicas para cada aplicação devido à dificuldade de roteamento dessas trilhas de metais.

Na metade da década de 80, com o surgimento de ferramenta de síntese, os ASICs passaram a utilizar o conceito de células padronizadas (*standard cells*). Assim, as ferramentas de síntese transcrevem circuitos descritos utilizando linguagens de descrição de *hardware*, como Verilog e

VHDL, em circuitos descritos utilizando *standard-cells*. Uma biblioteca de *standard-cells* é composta por portas lógicas pré-caracterizadas, como portas NOR, NAND, inversores, etc. Com essa tecnologia, o fabricante responsável pela manufatura final do ASIC, cria máscaras específicas para cada circuito, utilizando o silício de maneira muito mais eficiente do que na tecnologia de *gate array*. Grande parte dos fabricantes oferece um conjunto de blocos funcionais pré-especificados para auxiliar os projetistas no desenvolvimento dos circuitos. Dentre os blocos mais utilizados estão os mais diversos tipos de blocos de memórias, desde SRAM (*Static Random Access Memory*) até *Flash*.

Na década de 90, o desenvolvimento de ASICs utilizando a tecnologia das células padronizadas consolidou-se definitivamente. Dessa forma, os ASICs atuais atingiram um alto nível de complexidade, chegando a milhões de portas lógicas em cada circuito.

3.1.2 Dispositivos Lógico-Programáveis (PLDs)

Os dispositivos lógico-programáveis (PLDs) surgiram como uma nova opção para satisfazer a necessidade de flexibilidade nos circuitos lógicos digitais. A habilidade de adequar-se às diferentes aplicações com necessidades de estruturas computacionais e de comunicação específicas é um dos grandes atrativos desse novo segmento.

Essa nova classe de arquiteturas pode ser subdividida em três categorias: *Simple Programmable Logic Devices* (SPLDs), *Complex Programmable Logic Devices* (CPLDs) e *Field-Programmable Logic Devices* (FPLDs).

Os SPLDs referem-se aos PLDs com funções lógicas mais simples, como por exemplo, funções *AND* e *OR*. Fazem parte desse grupo os primeiros circuitos lógicos programáveis, lançados em meados dos anos 70. O *Programmable Logic Array* (PLA) consiste de dois níveis de portas lógicas, um contendo portas *AND* e outro com portas *OR,* sendo apropriados para aplicações com funções de soma de produtos de elementos.

A dificuldade de aumentar a capacidade lógica dos PLAs, assim como a necessidade de inserir mais níveis lógicos fez surgir uma nova categoria: os CPLDs, os quais possuem vários blocos lógicos de SPLDs em um único

chip, interligados de forma sofisticada. Um CPLD possui capacidade lógica equivalente a 50 SPLDs.

O mesmo ocorreu no processo de evolução dos CPLDs para os FPLDs, mas foram necessárias ferramentas para dar suporte ao crescimento da complexidade. Os FPLDs mais utilizados comercialmente são os *Field-Programmable Gate Arrays* (FPGAs). Os quais são formados por um arranjo de blocos lógicos que podem ter suas conexões programadas levando os sinais resultantes aos blocos de entrada e saída localizados em todo seu perímetro. Essa programação pode ser feita através de RAM estática, transistores ou EPROM/EEPROM dependendo do modelo do FPGA.

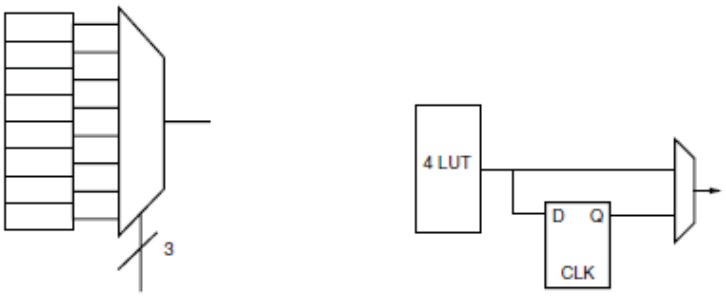
3.1.2.1 FPGAs

Conceitualmente, um FPGA pode ser visto como uma matriz de blocos lógicos interligados por estruturas de conexão. Os blocos lógicos podem conter elementos de processamento visando implementar desde uma simples lógica combinacional, até blocos de memórias. A estrutura de conexão é capaz de rotear cada bloco lógico de forma a conectá-los da melhor maneira possível para garantir o funcionamento do circuito implementado.

Cada bloco lógico pode conter uma *lookup table* (LUT) simples de três entradas (Figura 3.2(a)) ou um bloco lógico composto por essa mesma LUT e um *flip-flop* D (Figura 3.2(b)). Essa diferença é referida como granulosidade do bloco lógico. Sendo a LUT de 3 *bits* um exemplo de um elemento com granulosidade fina e o bloco lógico, um elemento de granulosidade relativamente grossa. Os blocos de granulosidade fina são úteis na manipulação de *bits*, enquanto os com granulosidade grossa são empregados em aplicações onde o fluxo de dados, como um todo, é mais relevante [Hauck and DeHon, 2008].

Para conectar os blocos lógicos, o FPGA conta com um grande número de estruturas de interconexão. Os blocos lógicos, com suas estruturas baseadas em LUTs, ficam rodeados por recursos de encaminhamento, os quais são responsáveis por encaminhar os sinais resultantes aos outros blocos lógicos, através de um conjunto de fios de cobre. Como pode ser visto na Figura 3.3 os blocos lógicos têm seus sinais

de entrada/saída ligados a estruturas de conexão por meio dos recursos de encaminhamento, sendo as estruturas de conexão as responsáveis por interligá-los, conectando os terminais dos blocos lógicos.



(a) (b)

Figura 3.2: LUT de 3 entradas (a) e bloco lógico composto por uma LUT e flip-flop D (b) [Hauck and DeHon, 2008].

As estruturas de conexão podem receber recursos de encaminhamento de blocos lógicos vizinhos como de blocos distantes. Essa distância está intimamente ligada à velocidade de transmissão entre os blocos lógicos, ou seja, quanto mais distantes estiverem os blocos, maior o atraso. Tentando amenizar esse atraso, algumas arquiteturas possuem canais globais, nos quais sinais globais, como de *clock* ou de *reset*, podem trafegar em alta velocidade.

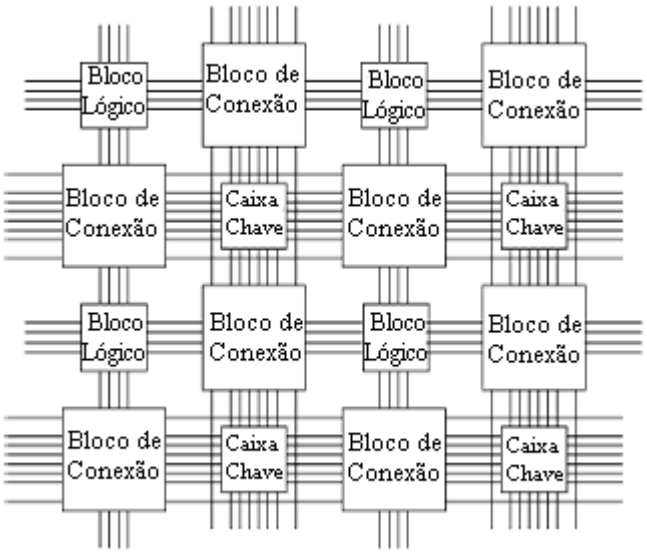


Figura 3.3: FPGA genérico com recursos de encaminhamento.

3.1.3 A Escolha entre ASIC e FPGA

Os custos envolvidos para o desenvolvimento de um ASIC podem chegar a centenas de milhares de reais. Entretanto, após esse investimento inicial, o custo para a produção de uma unidade do circuito pode chegar a apenas centavos de reais. Tal valor é muito inferior ao custo de produção de um circuito contendo um FPGA.

Porém, o processo de desenvolvimento de um circuito num FPGA é mais ágil, quando comparado ao circuito para ASICs. Ainda, as placas de desenvolvimento contendo os FPGAs, apresentam elementos para complementar o sistema para o qual o circuito foi desenvolvido, como por exemplo, osciladores, memórias ou saídas de vídeo. Assim, o sistema pode ser rapidamente testado e validado. Caso existam falhas no sistema desenvolvido, o circuito pode ser ajustado e o FPGA pode ser reprogramado.

Por essas razões, muitos circuitos são desenvolvidos inicialmente em FPGAs. Após serem testados e validados, estes circuitos são reprojetados para serem fabricados como ASICs, para serem produzidos em larga escala.

3.2 Linguagens de Descrição de *Hardware*

Muitas das ferramentas existentes no mercado para auxiliar no projeto de circuitos, tanto para FPGAs quanto para ASICs, utilizam linguagens de descrição de *hardware* (*Hardware Description Languages* – HDLs). As HDLs são utilizadas para descrever de maneira abstrata o funcionamento comportamental de um circuito, além de serem capazes de modelar estruturalmente o *hardware* que irá compor esse mesmo circuito. O comportamento do *hardware* pode ser modelado e representado em vários níveis de abstração durante o processo de desenvolvimento do circuito. Níveis mais altos de abstração são utilizados para descrever o funcionamento comportamental do circuito, enquanto os níveis mais baixos de abstração, por conter mais detalhes, são utilizados para a descrição estrutural do circuito.

Atualmente, existe um grande número de linguagens de descrição de *hardware*, porém, as mais utilizadas são a linguagem VHDL (*Very High Speed Integrated Circuit Hardware Description Language)* e a Verilog.

3.2.1 VHDL (*VHSIC Hardware Description Language)*

A linguagem VHDL foi criada a partir da iniciativa do Departamento de Defesa dos Estados Unidos, dentro do programa VHSIC (*Very High Speed Integrated Circuit*), em meados da década de 80, para facilitar a descrição dos ASICs que compunham os equipamentos vendidos às forças armadas dos Estados Unidos.

Assim, o desenvolvimento da VHDL serviu inicialmente aos propósitos de documentação do programa VHSIC. Entretanto, nesta época buscava-se uma linguagem que facilitasse o projeto de um circuito; ou seja, a partir de uma descrição textual, um algoritmo, desenvolver o circuito, sem a necessidade de especificar explicitamente as ligações entre os componentes. Dessa maneira, a VHDL pode ser utilizada em diversas etapas do fluxo de desenvolvimento de circuitos digitais, desde tarefas de documentação até tarefas de verificação formal [Chu, 2006].

Após o sucesso inicial do uso da VHDL, a sua definição foi posta em domínio público, o que levou a ser padronizada pelo *IEEE (Institute of Electrical and Electronic Engineers)* em 1987. O fato de ser padronizada e de domínio público ampliou ainda mais a sua utilização, novas alterações foram propostas, e a linguagem sofreu uma revisão e um novo padrão mais atualizado foi lançado em 1993. Pequenas alterações foram feitas em 2000 e 2002. Em setembro de 2008 foi aprovada pelo REVCOM a mais recente versão, IEEE 1076-2008.

3.2.2 Verilog

A linguagem de descrição de *hardware* Verilog foi criada em 1983, por Phil Moorby e Prabhu Goel, na empresa Gateway Design Automation. Em 1990, essa empresa foi comprada pela Cadence Design Systems, a qual possui os direitos sobre a linguagem Verilog, assim como pelo compilador desenvolvido para essa linguagem.

Com o sucesso da linguagem VHDL, em 1995, a Cadence decidiu padronizar a linguagem Verilog e torná-la de domínio público sob a supervisão da organização *Open Verilog International* (OVI). Mais tarde, a

linguagem Verilog foi submetida à padronização do IEEE e recebeu a certificação IEEE 1364-1995.

Assim como ocorreu com a VHDL, a linguagem Verilog sofreu pequenas alterações, e novas padronizações pela IEEE surgiram em 2001 e 2005.

A partir de sua última padronização, o padrão IEEE 1364-2005, surgiu a linguagem SystemVerilog. Essa nova linguagem é um conjunto da linguagem Verilog-2005, aumentado com algumas novas características capazes de modelar tanto sistemas para verificação quanto para sistemas de *hardware* em si. Em 2009, a linguagem SystemVerilog foi padronizada pelo IEEE (padrão IEEE 1800-2009) e passou a ser amplamente utilizada no desenvolvimento de circuitos integrados tanto para ASIC quanto para FPGA.

3.3 A Metodologia *top-down*

Em um cenário ideal, a metodologia *top-down* para desenvolvimento de sistemas seria utilizada para desenvolver sistemas completos da maneira mais otimizada possível. Iniciando com uma descrição abstrata dos circuitos utilizando HDLs e sendo complementados por ferramentas para automação de projetos eletrônicos (*Electronic Design Automation* – EDA) até chegar a sua implementação final, em placas de circuito impresso (*Printed Circuit Board* – PCB).

Porém, a realidade atual está longe da ideal, e as ferramentas EDA são constantemente atualizadas para aproximar o fluxo de desenvolvimento de circuitos integrados dos fundamentos da metodologia *top-down*. Isso significa que os projetistas precisam sempre se atualizar e aprender como lidar com as novas características dessas ferramentas. Isso inclui um tempo maior desenvolvendo modelos em HDLs, considerando as novas arquiteturas existentes, aliado à necessidade de aumentar as formas de testar os circuitos desenvolvidos. Assim, está cada vez menor o tempo gasto no modelamento de circuitos utilizando portas lógicas propriamente ditas.

Com os avanços tecnológicos nos últimos anos, houve um aumento considerável na complexidade dos sistemas desenvolvidos. Com sistemas cada vez mais complexos, fica evidente a utilização de um bom fluxo de

projeto, como o descrito pela metodologia *top-down*, aliado às boas ferramentas de EDA.

Na metodologia *top-down*, o desenvolvimento do circuito começa em alto nível, com o modelamento do circuito utilizando uma linguagem HDL. Assim, inicialmente, é descrito um modelo abstrato do circuito, que pode ser comportamental, descrito em linguagem de mais alto nível, como C, por exemplo; ou já na forma de algoritmo, em linguagens mais próximas das linguagens que serão utilizadas na descrição do circuito, como SystemC, por exemplo. Em seguida, descendo a níveis inferiores, no desenvolvimento do sistema, onde o circuito será descrito utilizando transistores, conforme pode ser observado na Figura 3.4.

Figura 3.4: Pirâmide de nível de abstração comportamental [Smith, 1996].

O termo comportamento representa a função que o circuito deveria desempenhar e independe do nível de abstração em que ele foi modelado. O circuito representado com portas lógicas apresenta a mesma função do circuito modelado inicialmente utilizando uma HDL. À medida que os modelos do circuito são levados para níveis mais baixos de abstração, eles se tornam mais complexos e com uma estrutura mais detalhada. Assim, o projetista não precisa se preocupar com a real complexidade do circuito ao modelar os circuitos utilizando níveis mais altos de abstração, garantindo as mesmas funcionalidades do sistema.

A estrutura do circuito é ignorada quando estamos nos níveis mais altos de abstração. Entretanto, ao modelar o circuito utilizando registradores

(*Register Transfer Leve*l – RTL) é essencial manter a funcionalidade do circuito em mente o tempo todo. Na figura 3.5 observa-se como os diferentes níveis de abstração comportamental sobrepõem-se aos diferentes tipos de modelamento do circuito: comportamental, estrutural e físico.

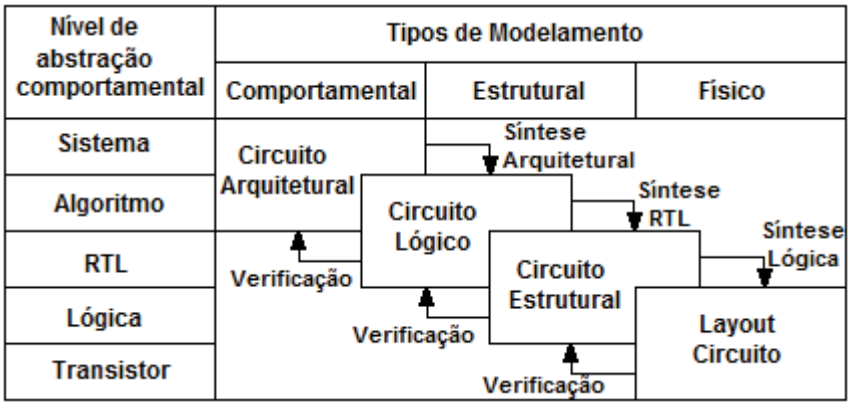


Figura 3.5: Tipos de modelamentos para diferentes níveis de abstração.

3.3.1 Fluxo para o Desenvolvimento de Circuito Integrado

Na figura 3.6 observa-se o fluxo para o desenvolvimento de circuitos integrados, baseado nos fundamentos da metodologia *top-down*, o qual pode ser utilizado tanto para ASICs quanto para FPGAs.

Conforme citado anteriormente, o desenvolvimento de circuitos integrados inicia-se pela especificação do sistema. Nessa etapa, todas as funções que o circuito irá desempenhar devem ser minuciosamente descritas utilizando as formas que o projetista julgar útil, como por exemplo, textos, gráficos ou *softwares* especializados em descrição de projetos. Essa etapa é muito importante, pois a partir dessa especificação, os projetistas iniciarão o desenvolvimento da arquitetura que resultará no circuito físico no final do projeto, assim como no desenvolvimento das rotinas para teste do sistema.

Ainda na etapa de especificação, são determinadas as funções que devem ser externas ao circuito, ou seja, quais são os estímulos que o circuito receberá quando estiver inserido no seu ambiente de funcionamento. Com base nessas informações, os engenheiros de sistema são capazes de projetar as placas necessárias para o teste do circuito integrado. No caso de ASICs, desenvolve-se uma PCB que receberá o circuito integrado fabricado,

assim como os circuitos auxiliares para o seu funcionamento, como, por exemplo, osciladores de cristal, resistores, capacitores e memórias externas. Já no caso de FPGAs, utiliza-se uma placa de desenvolvimento que contenha as características necessárias para emular o sistema onde o circuito, a ser desenvolvido, estará inserido.

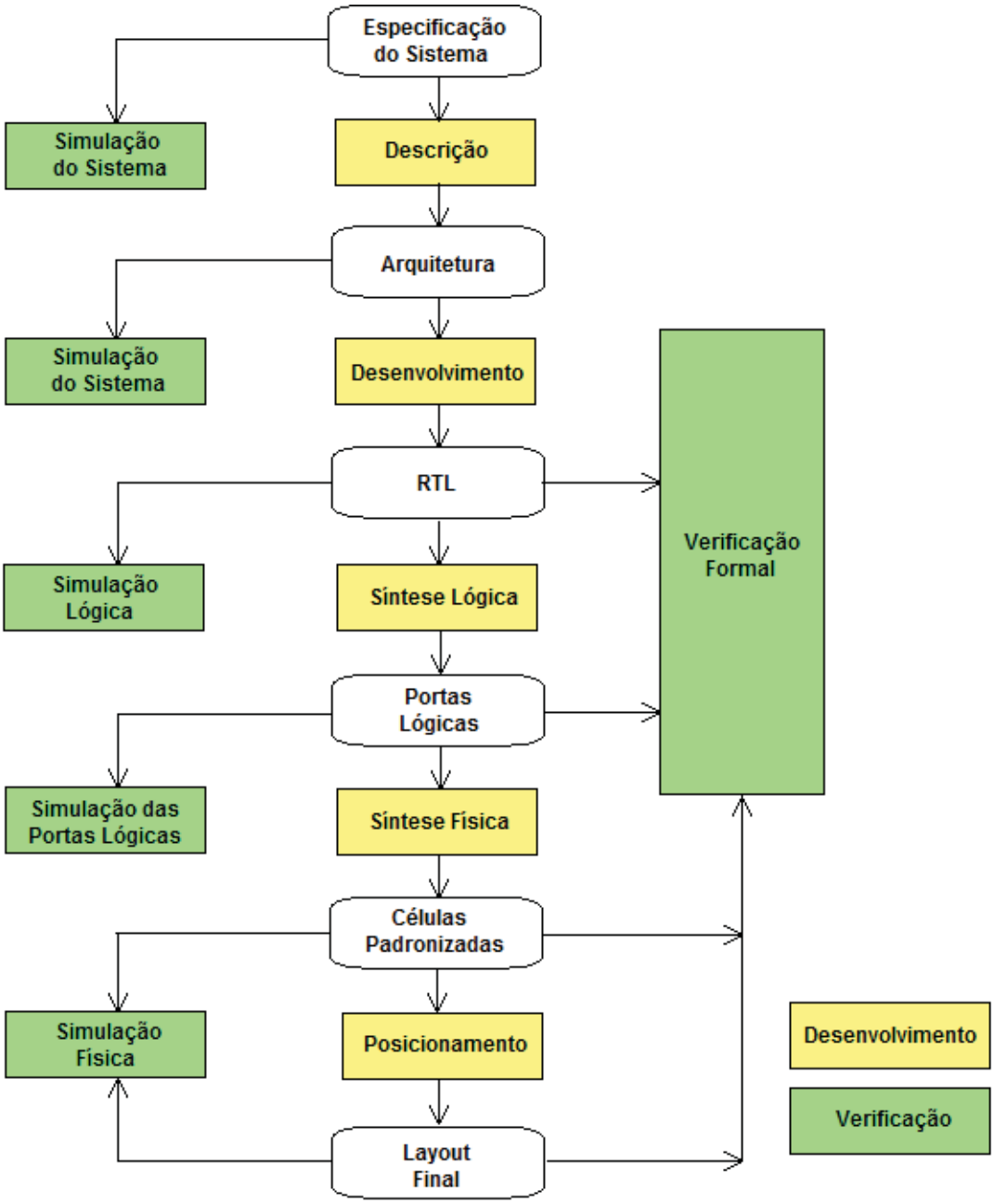


Figura 3.6: Fluxo para desenvolvimento de circuitos integrados.

A partir de uma boa especificação, inicia-se a descrição da arquitetura do circuito. A descrição da arquitetura pode ser feita com os recursos que o

projetista achar conveniente, como textos, gráficos, fluxogramas ou a implementação inicial do circuito.

Baseado na arquitetura proposta, o projetista começa o desenvolvimento da descrição do circuito utilizando registradores. Essa fase consiste em descrever o funcionamento do circuito levando em consideração a transferência dos dados utilizando registradores, o chamado RTL (*Register Transfer Level*). O RTL é a ponte entre a arquitetura inicial e a implementação final do circuito. O RTL é manualmente criado pelos projetistas utilizando linguagens HDLs baseado no que foi determinado na arquitetura e na especificação previamente desenvolvidos. Sendo assim, uma boa descrição em RTL é crucial na busca por um circuito que atinja todas as funções especificadas com o menor consumo, menor área e melhor frequência possíveis.

A descrição em RTL é uma etapa comum em fluxos para circuitos desenvolvidos para ASICs e para FPGAs. Porém, nessa etapa os projetistas utilizam blocos disponibilizados pelos fabricantes, de ASICs ou de FPGAs, para aperfeiçoar o circuito, os chamados IPs (*Intellectual Property*). Dentre os IPs mais comumente utilizados estão os blocos de memórias de dados.

De posse da descrição do circuito em RTL, o projetista é capaz de executar a síntese lógica do circuito. Esse processo é realizado com o auxílio de ferramentas de EDA, diferentes para ASICs e FPGAs. As ferramentas de EDA convertem o circuito descrito em RTL para um circuito descrito utilizando as portas lógicas existentes na tecnologia especificada para o desenvolvimento do circuito em ASIC ou em FPGA.

De maneira similar ao processo de síntese lógica, ocorre o processo para a síntese física. Porém, nesse processo as portas lógicas são traduzidas para as células padronizadas (*Standard Cells*) específicas para a tecnologia do fabricante do ASIC que produzirá o circuito desenvolvido. Para o fluxo de desenvolvimento de FPGAs, essa etapa equivale à tradução do circuito em blocos lógicos específicos para cada família de FPGA.

Após a transcrição do circuito para células padronizadas, no caso de ASICs, ou blocos lógicos, no caso de FPGAs, ocorre a etapa de posicionamento dessas células da melhor maneira possível para garantir a menor área e a melhor frequência possível. Para os ASICs, essa etapa

também é chamada de roteamento, pois com o auxílio de ferramentas de EDA, o projetista distribui as trilhas de metais, disponibilizado pela tecnologia de fabricação, de forma a conectar as células seguindo algumas regras determinadas pelo fabricante sem se esquecer das especificações de área e de frequência determinadas para o circuito. Dessa forma, é gerada a disposição final das células e trilhas que compõem o circuito, chamado de *layout*. No caso de circuitos desenvolvidos para ASICs, a partir do *layout* é gerado um arquivo específico que será enviado ao fabricante a partir do qual será produzido o circuito final. Para os circuitos desenvolvidos para FPGAs, o projetista realizará a programação do FPGA com o auxílio de ferramentas específicas e iniciará as fases finais de teste do sistema.

Após cada etapa do processo de desenvolvimento de circuitos integrados, o circuito é verificado para garantir que suas funcionalidades não fujam ao que foi descrito na especificação. Devido à importância do processo de verificação no desenvolvimento de circuitos integrados, existem diferentes técnicas que podem ser aplicadas a cada uma das etapas do fluxo de desenvolvimento de circuitos integrados.

3.3.1.1 Verificação para Circuitos Integrados Digitais

A etapa de verificação é uma das mais custosas dentro do processo de desenvolvimento de circuitos integrados digitais. Pois, somente através da verificação pode se garantir que todas as funcionalidades previstas na especificação foram corretamente implementadas no circuito desenvolvido.

Dessa forma, durante essa etapa, todas as características do circuito são minuciosamente verificadas para emular todas as possíveis interações do circuito com o meio externo. O processo de recriação do meio exterior para prever possíveis falhas do circuito, chega a consumir 70% dos recursos disponíveis para o desenvolvimento do circuito [Bergeron, 2002]. Um outro parâmetro dessa atividade é a proporção de engenheiros de verificação alocados em um projeto de um ASIC complexo. Normalmente, são necessários dois engenheiros de verificação para cada engenheiro que descreve o circuito em RTL. Outro parâmetro interessante, é que cerca de 80% do total de código produzido é gerado ao longo do processo de verificação devido à grande quantidade de módulos criados para emular o

sistema completo buscando cobrir o maior número de características de funcionamento possíveis.

Devido ao tamanho do esforço demandado pela etapa de verificação, novas metodologias são desenvolvidas, além das ferramentas de EDA serem constantemente aprimoradas visando reduzir o tempo gasto na verificação. Tanto as ferramentas quanto as novas metodologias priorizam o paralelismo do esforço gasto durante o processo de verificação, subsidiados por técnicas que envolvem a automação dos códigos criados com o mais alto nível de abstração possível.

Dessa forma, para paralelizar a tarefa de verificação é necessário um número maior de engenheiros no projeto, pois parte deles trabalhará no desenvolvimento de estruturas de teste enquanto outros engenheiros dedicam-se à implementação dos códigos descritos em RTL.

Ao trabalhar com um alto nível de abstração, o projetista é capaz de criar estruturas de testes mais eficientes sem ater-se aos detalhes da implementação física do circuito. Reduzindo, assim, o grau de controle sobre o circuito a ser verificado (*Design Under Verification* – DUV).

Além disso, através da criação de processos padronizados onde os resultados podem ser facilmente resgatados, o projetista pode automatizar o processo de verificação. Dessa forma, por meio de rotinas pré-determinadas, o projetista pode inserir estímulos e resgatar os resultados esperados com a mínima interação com o DUV, diminuindo a possibilidade de inserção de erros humanos nesse processo.

A paralelização das atividades, a geração de estruturas de verificação em alto nível de abstração e a automatização do ambiente de verificação são os pilares das metodologias de verificação atuais. Entretanto, a forma como verificar um circuito pode seguir duas grandes vertentes: a verificação formal e a verificação funcional, ambas descritas a seguir.

3.3.1.1.1 Verificação Formal

Na verificação formal o circuito é testado para garantir que ele foi corretamente desenvolvido, ou seja, para constatar se a funcionalidade proposta está corretamente implementada. Assim, a partir de determinados

estímulos de entrada, e sabendo-se quais as saídas esperadas, verifica-se o circuito implementado através de um modelo ou de um circuito equivalente.

3.3.1.1.1.1 Verificação de um Modelo

A verificação de um modelo ocorre através da especificação formal das características descritas para aquele circuito. Assim, o projetista desenvolve um modelo formal para o circuito implementado buscando detectar possíveis erros ou condições indefinidas que poderão causar uma parada no sistema, como, por exemplo, estados inalcançáveis ou isolados em uma máquina de estados. Por meio de um modelamento adequado, o projetista é capaz de caracterizar o circuito determinando como a ativação de determinadas entradas influenciará cada saída. O grande desafio nesse tipo de verificação é entender a especificação tão profundamente a ponto de criar um modelo formal completo o suficiente.

3.3.1.1.1.2 Verificação de Circuito Equivalente

Esse procedimento de verificação prova matematicamente que duas instâncias diferentes de um mesmo circuito são equivalentes. Assim, após serem realizadas operações de síntese em ferramentas de EDA é necessário comprovar que o circuito resultante seja equivalente ao original, pois durante a etapa de síntese ocorre a transcrição do circuito original, descrito utilizando um conjunto de portas lógicas básicas, em um circuito descrito utilizando um outro conjunto de portas lógicas pertencentes à tecnologia alvo. Dessa maneira, garante-se que o circuito original foi corretamente sintetizado e que as funcionalidades especificadas para o RTL foram mantidas. Salientando que esse procedimento deve ser aplicado após o circuito descrito em RTL ter sido devidamente verificado.

3.3.1.1.2 Verificação Funcional

O propósito principal da verificação funcional é garantir que o circuito implementado apresente as funcionalidades pretendidas na especificação. Vale ressaltar a diferença entre a verificação formal, a qual prova que a funcionalidade foi corretamente implementada, mas não que essa

funcionalidade estivesse de acordo com a especificação. Assim, sem a verificação funcional não se pode confiar que a transcrição do documento de especificação para a descrição do código em RTL ocorreu corretamente, o que pode prejudicar todo o restante do processo de desenvolvimento do circuito. Portanto, a verificação funcional mostra através de simulações do sistema, que aquela funcionalidade atende à especificação do circuito, enquanto que a verificação formal, prova, através de métodos formais, que aquela funcionalidade foi implementada corretamente.

3.3.1.1.2.1 *Testbenches*

A base da verificação funcional é a simulação do circuito emulando o sistema no qual esse circuito estará inserido. Para isso, é necessária a criação de um ambiente para a verificação, comumente nomeado *testbench*. O *testbench* tem a função de enviar estímulos ao DUV e receber os sinais de saída desse DUV e compará-los com os sinais esperados para esse circuito. Os sinais esperados podem ser determinados por um modelo do circuito implementado ou por uma sequência pré-definida baseada na especificação do circuito, dependendo do estilo de verificação escolhida pelo engenheiro de verificação.

O *testbench* deve ser descrito utilizando rotinas de alto nível de abstração que possam ser facilmente automatizadas permitindo a execução de várias simulações em paralelo. Assim, ao utilizar rotinas em alto nível, o projetista poderá utilizar melhor os recursos da linguagem escolhida, não se preocupando com detalhes da implementação física do circuito. Além disso, as rotinas de verificação devem ser escritas de maneira que possam ser reutilizadas para a criação de outros *testbenches*. Com a padronização das rotinas de verificação, o projetista pode criar um ambiente que possibilite a automatização das simulações, utilizando linguagens específicas para criação de arquivos que manipulem tanto os arquivos contendo essas rotinas quanto as ferramentas de EDA que realizam a simulação do *testbench*, os chamados *scripts*. As linguagens para criação de *scripts* mais utilizadas são a Perl, Phyton, Tcl e JavaScript.

CAPÍTULO 4 – Desenvolvimento do Sistema

4.1 Desenvolvimento do conjunto de filtros

Para o conjunto de filtros foram implementados seis algoritmos diferentes buscando abranger as mais diversas aplicações. Dessa forma, foram implementados dois tipos de filtros de suavização de ruídos, um filtro detector de bordas utilizando o algoritmo de Sobel, um filtro para equalização de histograma, um filtro para normalização de cores e um filtro para normalização de luminância.

Para garantir uma adequada implementação na placa de FPGA, as imagens tratadas pelos filtros foram redimensionadas para 60x40 pixels. Sendo que, primeiramente, os algoritmos de cada filtro, foram testados utilizando o *software* Matlab, por este apresentar rotinas que facilitam a rápida visualização das imagens resultantes. Em seguida foi realizada a implementação desses algoritmos utilizando a linguagem de descrição de *hardware* SystemVerilog, visando a prototipação do circuito resultante em FPGA. Assim, para facilitar a entrada de dados no sistema descrito em FPGA, as imagens iniciais foram decompostas em três vetores com 2400 posições cada correspondendo a cada uma das cores da imagem, RGB.

A seguir é detalhado o desenvolvimento de cada um dos filtros acima mencionados.

4.1.1 Filtros de Suavização

Estes filtros têm a função de suavizar a imagem visando diminuir os ruídos contidos nela. Foram implementados dois tipos de filtros: um contendo a filtragem por mediana em uma vizinhança de 3x3 pixels e outro contendo a filtragem por média em uma vizinhança de 3x3 pixels. Em ambos os filtros, as máscaras de suavização são aplicadas conforme descrito na seção 2.2.1.2.

Após a primeira implementação do algoritmo desses filtros, em Matlab, partiu-se, assim, para a implementação do modelo desses algoritmos na linguagem SystemVerilog, realizando as devidas alterações por conta das características próprias da linguagem. Esse modelo inicial serviu para adequar os tipos de dados utilizados no Matlab para a linguagem SystemVerilog. Pois, a linguagem utilizada pelo Matlab possibilita o tratamento das imagens como matrizes compostas por números inteiros e ou em ponto flutuante, tal fato não ocorre na linguagem SystemVerilog. A qual não é capaz de tratar números em ponto flutuante, fazendo com que seja necessário aumentar a quantidade de *bits* para tratar um número, para que não se perca a precisão do valor calculado. Outro ponto relevante, da linguagem SystemVerilog, é o tratamento das imagens não como matrizes, mas sim, como vetores. Ao se ler os *bits* da imagem a ser tratada de um vetor tem-se a possibilidade de ler a quantidade necessária para cada operação, como no caso das operações com máscaras onde se deve ler uma quantidade fixa de pixels. Entretanto, é necessário fazer um cálculo baseado na quantidade de pixels em cada linha e em cada coluna da imagem para ler o pixel na posição correta dentro da máscara aplicada. Esse cálculo é realizado através da equação 12:

$$k = i*NumColunas+j \quad (12)$$

Na equação 12, *k* é a posição a ser lida do vetor, *i* a posição da linha, *j* a posição da coluna dentro da imagem e *NumColunas* corresponde à quantidade fixa de colunas da imagem, que nesse projeto é igual a 60.

O detalhe a respeito da linguagem SystemVerilog sobre a utilização de matrizes, que fez com que se optasse pelo armazenamento das imagens na forma de vetores e não de matrize,s é o mecanismo de armazenamento dessas matrizes utilizando essa linguagem. Na linguagem SystemVerilog as

matrizes são representadas como vetores bidimensionais, ou seja, cada posição do vetor é composta por uma palavra de um número determinado de *bits*. Assim, ao se fazer uma operação utilizando esse vetor bidimensional, é necessário acessar essa palavra com um número determinado de *bits* e depois escolher os *bits* desejados. Ao armazenar as imagens como vetores unidimensionais, reduziu-se o tratamento apenas à posição a ser lida no vetor, conforme descrita na equação 12.

Levando em consideração todos os detalhes acima descritos, a partir de um teste inicial para relembrar o funcionamento dos filtros de suavização no Matlab, foi gerado um modelo utilizando construções não sintetizáveis da linguagem SystemVerilog. Esse modelo foi utilizado como referência na etapa de verificação da versão sintetizável de cada filtro e foi implementado na forma de uma *task* (tarefa) em SystemVerilog. Assim, essa *task* foi adicionada ao código do monitor, o qual utilizou essa tarefa para calcular a imagem resultante esperada a partir da imagem original lida. O modelo de referência implementado para o filtro da mediana está descrita no Apêndice A : A.1 - *task* ref_mod_mediana, onde pode-se observar o laço para a leitura da imagem como uma matriz nas linhas 15 a 17.

```
15          for(i=0;i<numLinhas;i++)
16            begin
17              for(j=0;j<numColunas;j++)
```

Também se observa a conversão desses valores para a leitura da posição do vetor na linha 20 através da equação 12.

```
20                  k=i*numColunas+j;
```

A partir da linha 21 até a linha 71, observa-se o ajuste do vetor de saída para a realização da convolução periódica nas bordas da imagem. Assim, as bordas da imagem serão lidas como se estivessem conectadas, o que torna a imagem resultante mais suave nas bordas, facilitando o processamento pelos próximos filtros.

```
21                  //aplica a convolução periódica nos extremos da janela
22                  if(i==0 || j==0 || i==numLinhas-1 || j==numColunas-1)
23                    begin
24                      if ((i==0)&(j==0))
25                        begin
26                          aux1=(numColunas*numLinhas)-1;
27                         Vet_outR[k]= Vet_inR[aux1];
28                          Vet_outG[k]= Vet_inG[aux1];
29                          Vet_outB[k]= Vet_inB[aux1];
30                        end
```

```
31                  else
32                      if (i==0)
33                        begin
34                          aux1=(a * numColunas) + b +(numColunas-1);
35                          Vet_outR[k]= Vet_inR[aux1];
36                          Vet_outG[k]= Vet_inG[aux1];
37                          Vet_outB[k]= Vet_inB[aux1];
38                        end
39                      else
40                        if (j==0)
41                          begin
42                            aux1 = (a * numColunas) + b + (numLinhas-1);
43                            Vet_outR[k]= Vet_inR[aux1];
44                            Vet_outG[k]= Vet_inG[aux1];
45                            Vet_outB[k]= Vet_inB[aux1];
46                          end
47                        else
48                          if ((i==numLinhas-1)&(j==numColunas-1))
49                            begin
50                              aux1 = 0;
51                              Vet_outR[k]=Vet_inR[aux1];
52                              Vet_outG[k]=Vet_inG[aux1];
53                              Vet_outB[k]=Vet_inB[aux1];
54                            end
55                          else
56                            if (i==numLinhas-1 )
57                              begin
58                                aux1 = (a * numColunas) + b - (numColunas-1);
59                                Vet_outR[k]= Vet_inR[aux1];
60                                Vet_outG[k]= Vet_inG[aux1];
61                                Vet_outB[k]= Vet_inB[aux1];
62                              end
63                            else
64                              if (j==numColunas-1)
65                                begin
66                                  aux1 = (a * numColunas) + b - (numLinhas-1);
67                                   Vet_outR[k]= Vet_inR[aux1];
68                                  Vet_outG[k]= Vet_inG[aux1];
69                                  Vet_outB[k]= Vet_inB[aux1];
70                                end
71                  end
```

Em seguida, observa-se a leitura dos pixels correspondente a máscara do filtro, nas linhas 75 a 84.

Após a leitura dos pixels da máscara ocorre a ordenação desses pixels através do método bolha, o qual pode ser observado nas linhas 85 a 111.

```
85                    //Ordena o vetor (METODO: BOLHA)
86                  for (b=0;b<9;b++)
87                    begin
88                      for(a=8;a>b;a--)
89                        begin
90                          c=a-1;
91                           aux=VetauxR[c];
92                          if(VetauxR[c]>VetauxR[a])
93                             begin
94                               aux2=VetauxR[a];
```

```
95                              VetauxR[a]=VetauxR[c];
96                              VetauxR[c]=aux2;
97                            end
98                          if(VetauxG[c]>VetauxG[a])
99                            begin
100                             aux2=VetauxG[a];
101                             VetauxG[a]=VetauxG[c];
102                             VetauxG[c]=aux2;
103                           end
104                         if(VetauxB[c]>VetauxB[a])
105                           begin
106                             aux2=VetauxB[a];
107                            VetauxB[a]=VetauxB[c];
108                            VetauxB[c]=aux2;
109                          end
110                   end
111             end
```

Por fim, nas linhas 113 a 115, observa-se que o valor mediano é enviado para a posição dos vetores RGB que estão sendo acessados no momento.

```
113             Vet_outR[k] = VetauxR[4];
114             Vet_outG[k] = VetauxG[4];
115             Vet_outB[k] = VetauxB[4];
```

Baseado nesse modelo de referência, foi descrito o circuito sintetizável, também em SystemVerilog, porém utilizando uma máquina de estados. Na figura 4.1 pode ser observada a máquina de estados do filtro da mediana.

A máquina de estados descrita na figura 4.1 será melhor explicada juntamente com a máquina de estados do filtro de média, observada na figura 4.2.

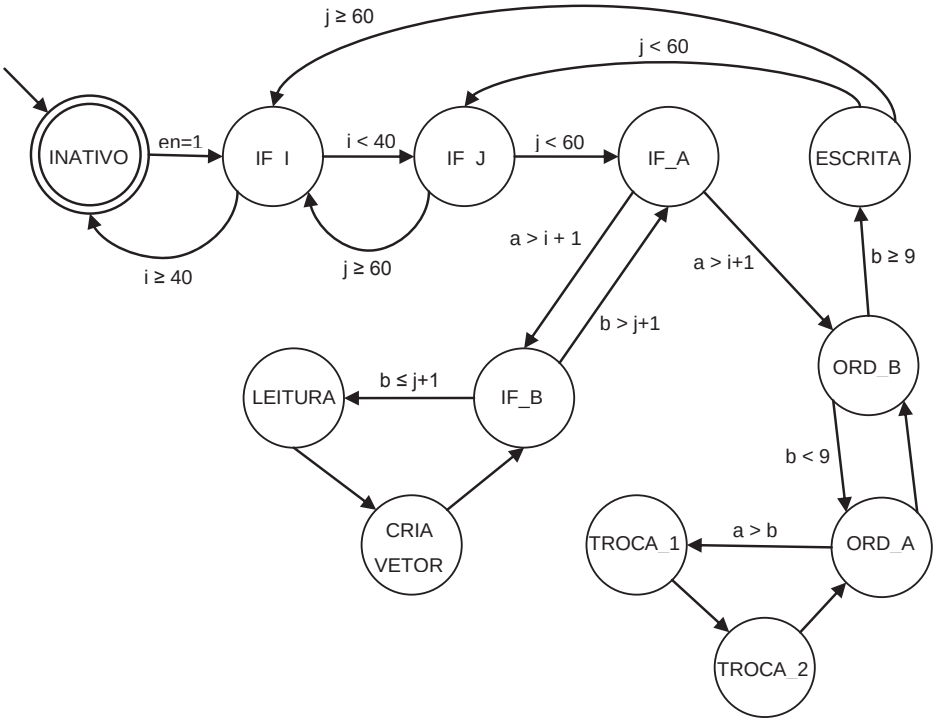


Figura 4.1: Máquina de estados do filtro de mediana.

Assim como foi realizado para o filtro da mediana, foi desenvolvida a *task* ref_mod_media para o filtro da média, descrita no Apêndice A – A.2. A qual também foi utilizada como modelo de referência na etapa de verificação da versão sintetizável.

As *tasks* ref_mod_mediana e ref_mod_media são semelhantes no quesito da leitura da imagem e na aplicação da convolução periódica. Assim, a partir da linha 76 observa-se a leitura dos pixels da máscara e a soma desses valores lidos entre as linhas 91 e 93.

```
76                  sumR = 0;   sumG = 0;   sumB = 0;
77                  for(c=0, a=(i-1); a<=(i+1); a++)
78                    begin
79                      for(b=(j-1); b<=(j+1); b++, c++)
80                        begin
81                          if(i==0 || j==0 || i==numLinhas-1 || j==numColunas-1)
82                            begin
83                              sumR = sumR + 0;
84                              sumG = sumG + 0;
85                              sumB = sumB + 0;
86                              add_aux=0;
87                            end
88                          else
89                            begin
90                              add_aux=a*numColunas+b;
91                              sumR = sumR + Vet_inR[add_aux];
92                              sumG = sumG + Vet_inG[add_aux];
93                              sumB = sumB + Vet_inB[add_aux];
```

Nas linhas 98 a 100, observa-se o cálculo do valor médio do valor resultante, o qual foi substituído pela posição k nos vetores RGB, nas linhas 102 a 104.

```
98                  medR = sumR/9;
99                   medG = sumG/9;
100                  medB = sumB/9;
101                  //substitui o pixel pelo valor médio
102                 Vet_outR[k] = medR;
103                Vet_outG[k] = medG;
104                Vet_outB[k] = medB;
```

A partir desse modelo de referência, foi descrito o circuito sintetizável para o filtro de média, utilizando uma máquina de estados, o qual pode ser observado na figura 4.2.

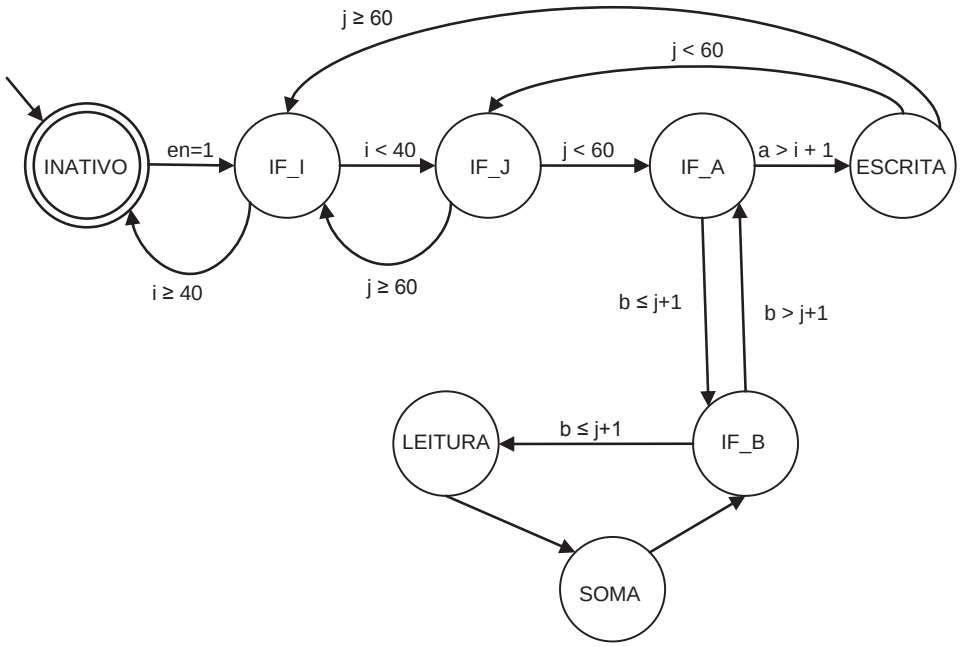


Figura 4.2: Máquina de estados do filtro de média.

A partir das figuras 4.1 e 4.2 observa-se que os filtros de mediana e de média têm grande parte de suas máquinas de estados similares. No estado INATIVO, as variáveis a serem utilizadas são inicializadas, quando o filtro é ativado (en=1), a máquina de estados passa para o próximo estado IF_I. Assim, nos estados IF_I e IF_J são realizados testes com as variáveis que fazem a leitura das imagens nas direções i e j, que correspondem à linha e a coluna, respectivamente. Nos estados IF_A e IF_B são realizados testes com as variáveis que armazenam as coordenadas da máscara de onde são lidos os pixels da imagem. Além disso, no estado IF_A, para o filtro

da média, é realizada a operação de média dos valores dos pixels lidos, antes de ir para o estado ESCRITA. E no estado IF_B, é realizado o cálculo dos endereços dos pixels a serem lidos no estado LEITURA. O cálculo dos endereços é realizado de tal forma e implementar uma convolução periódica sobre a imagem, ou seja, é realizado a leitura dos pixels sobre a máscara como se as bordas externas fossem conectadas. Assim, as bordas da imagem após a aplicação dos filtros de média ou de mediana ficam suavizadas o que não prejudica a aplicação de outros filtros na sequência.

No filtro da mediana, após o estado LEITURA, temos o estado CRIA VETOR, onde os valores dos pixels lidos são armazenados formando um vetor de nove posições. Após o vetor ser lido e preenchido corretamente, a máquina de estados retorna ao estado IF_B, depois ao estado IF_A e segue para o estado ORD_B. O conjunto de estados formados por ORD_B, ORD_A, TROCA_1 e TROCA_2 implementam a ordenação do vetor contendo os valores lidos sob a máscara. Após a ordenação do vetor, o pixel mediano é escrito na posição central da máscara no estado ESCRITA.

No filtro da média, no estado ESCRITA ocorre a escrita do valor médio, calculado no estado IF_A, na posição central da máscara, assim como no filtro da mediana. Em ambos as máquinas de estado, após a operação de escrita, a aplicação do filtro continua até que toda a imagem seja lida.

Após a primeira implementação do modelo de referência, a imagem resultante foi levada ao Matlab para ser comparada à imagem resultante do algoritmo desenvolvido para essa ferramenta, para garantir que o modelo funcionava corretamente. Para a visualização da imagem resultante, tanto do modelo de referência quanto do circuito sintetizável, foi desenvolvido um programa que monta a imagem a partir dos vetores armazenados em arquivos texto. Esse programa chamado *vetor2imagem*, desenvolvido utilizando a linguagem de programação própria do *software* Matlab, está descrito no Apêndice A – A.3.

Confirmada a validade do modelo de referência, passou-se para a etapa de verificação do circuito implementado. Assim, foram enviados estímulos ao DUV e ao modelo de referência e durante o processamento das imagens enviadas, os sinais resultantes do DUV eram comparados aos

sinais vindos do modelo de referência para garantir que o circuito foi implementado corretamente.

Assim como para a leitura dos arquivos de saída, foi criado um programa para gerar os arquivos a serem enviados para o DUV e para o modelo de referência. O programa chamado *imagem2vetor*, desenvolvido utilizando a linguagem própria do *software* Matlab, realiza a leitura de uma imagem, a separação dos vetores RGB e os armazena em arquivo texto, os quais são utilizados como estímulos na etapa de verificação do DUV. A descrição do programa *imagem2vetor* é realizada no Apêndice A – A.4.

4.1.2 Filtro para Detecção de Borda

Esse filtro tem por objetivo ressaltar as bordas da imagem utilizando o algortimo de Sobel. Outros filtros foram testados anteriormente em [Mertes, Marranghello e Pereira, 2008], mas optou-se pelo algoritmo de Sobel por este apresentar melhores resultados nas imagens testadas aliado a um baixo custo computacional, devido às características de suas máscaras descritas na seção 2.2.2.1. Durante a aplicação do filtro de Sobel, uma máscara é aplicada após a outra sobre a imagem original, e o resultado é somado formando a imagem final com as bordas realçadas.

Conforme foi realizado com os filtros anteriores, esse filtro foi implementado inicialmente no Matlab para uma análise inicial do algoritmo. Em seguida, esse algoritmo foi implementado na linguagem SystemVerilog, gerando o modelo de referência descrito no Apêndice A : A.5 - *task* ref_mod_borda, onde observa-se inicialmente, nas linhas de 17 a 23, que são armazenados os valores, correspondentes às máscaras descritas na seção 2.2.2.1, nos vetores h1 e h2.

```
17          h1[0]=(-1); h1[1]=(0); h1[2]=(1);
18          h1[3]=(-2); h1[4]=(0); h1[5]=(2);
19          h1[6]=(-1); h1[7]=(0); h1[8]=(1);
20
21          h2[0]=(1);  h2[1]=(2);  h2[2]=(1);
22          h2[3]=(0);  h2[4]=(0);  h2[5]=(0);
23          h2[6]=(-1); h2[7]=(-2); h2[8]=(-1);
```

A seguir são realizadas as etapas de leitura da imagem e a aplicação da convolução periódica, descritas nas linhas 25 a 82. Tais etapas são similares às descritas nos filtros de mediana e de média.

A partir da linha 89, é realizada a leitura dos pixels dentro da máscara de 3x3 pixels, com os quais são calculados os valores resultantes a partir da convolução com os valores dos vetores h1 e h2.

```
89                  for(c=0, a=(i-1); a<=(i+1); a++)
90                    begin
91                      for(b=(j-1); b<=(j+1); b++, c++)
92                        begin
93                          add_aux=a*numColunas+b;
94                          //convolucao com h1
95                          sumR_h1 = sumR_h1 + (h1[c]*Vet_inR[add_aux]);
96                          sumG_h1 = sumG_h1 + (h1[c]*Vet_inG[add_aux]);
97                          sumB_h1 = sumB_h1 + (h1[c]*Vet_inB[add_aux]);
98                          //convolucao com h1
99                          sumR_h2 = sumR_h2 + (h2[c]*Vet_inR[add_aux]);
100                         sumG_h2 = sumG_h2 + (h2[c]*Vet_inG[add_aux]);
101                         sumB_h2 = sumB_h2 + (h2[c]*Vet_inB[add_aux]);
102                         //armazenamento em vetor auxiliar
103                         Vet_auxR[c] = Vet_inR[add_aux];
104                        Vet_auxG[c] = Vet_inG[add_aux];
105                        Vet_auxB[c] = Vet_inB[add_aux];
106                     end
107                 end
```

Nas linhas 109 a 111, ocorre a soma dos valores resultantes após a convolução. Como os valores resultantes podem gerar números negativos, estes são convertidos para números positivos na forma de complemento de 2, assim são armazenados corretamente nos vetores resultantes.

```
109             resR = sumR_h1 + sumR_h2;
110             resG = sumG_h1 + sumG_h2;
111               resB = sumB_h1 + sumB_h2;
```

Por fim, nas linhas 120 a 131, os valores finais são normalizados para estar dentro do intervalo de 0 a 255, intervalo característico das imagens resultantes que são armazenadas utilizando 8 *bits*.

```
120             if (resR > 255)
121                 Vet_outR[k] =  255;
122             else
123                 Vet_outR[k] = resR;
124              if (resG > 255)
125                 Vet_outG[k] =  255;
126              else
127                 Vet_outG[k] = resG;
128               if (resB > 255)
129                Vet_outB[k] =  255;
130              else
131                 Vet_outB[k] = resB;
```

Concluída a implementação do modelo de referência, foi descrito o circuito sintetizável para o filtro de detecção de borda utilizando uma

máquina de estados. Na figura 4.3 observa-se a máquina de estados do filtro para detecção de bordas.

A máquina de estados implementada para o filtro de detecção de bordas é similar à do filtro de média, exceto pelas funcionalidades de cada estado. As diferenças estão nos estados SOMA, IF_A e IF_B. No estado SOMA, é realizada a aplicação de cada das duas máscaras, que compõem o algoritmo de Sobel, ao pixel lido no estado LEITURA, resultando em 2 valores diferentes. No estado IF_B, é realizada a soma desses 2 valores, conforme descrição do algoritmo de Sobel. No estado IF_A é realizada uma operação de complemento de 2, caso o valor resultante seja menor que 0. Assim, o valor final a ser escrito na posição central da máscara será sempre positivo.

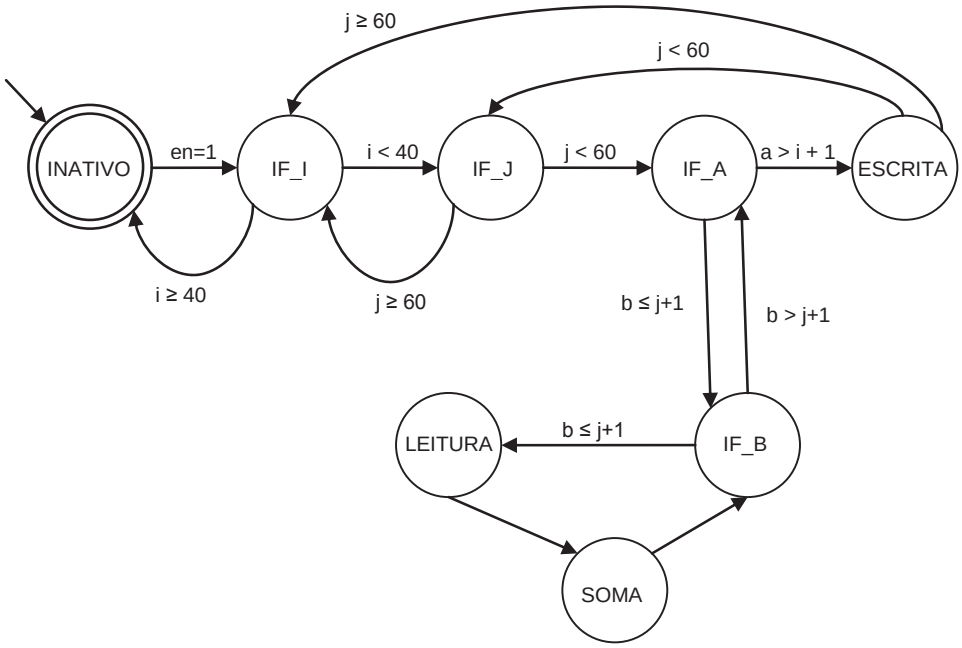


Figura 4.3: Máquina de estados do filtro de detecção de bordas.

Assim como descrito anteriormente, após a validação do modelo de referência, passou-se para a etapa de verificação do circuito implementado. Os estímulos foram gerados com o auxílio do programa *imagem2vetor* e enviados ao DUV e ao modelo de referência. Sendo que os sinais resultantes foram conferidos através do programa *vetor2imagem*, após serem devidamente comparados pelo monitor.

4.1.3 Filtro de Equalização de Histogramas

O filtro de equalização de histograma busca redistribuir os valores do histograma original visando à obtenção de um histograma uniforme com valores parecidos para todos os tons da imagem. Como a imagem original é colorida, a equalização do histograma é realizada sobre o matiz da imagem, variável que representa a cor fundamental de cada pixel. Para poder trabalhar corretamente com o matiz de cada pixel é necessário realizar uma transformação na imagem, a qual está descrita seguindo o modelo RGB, para o modelo HSV. Dessa forma, pode-se equalizar o histograma da componente H, o qual corresponde ao matiz da imagem. O processo de transformação entre os modelos de cores RGB e HSV foi descrito na seção 2.2.3.3. Já o processo de equalização de histograma foi descrito na seção 2.2.1.2, item B.

Um algoritmo inicial para validar as equações referentes à transformação entre os modelos de cores foi implementado no Matlab. Esse algoritmo foi útil para determinar as transformações necessárias para manter a compatibilidade entre os tipos de dados utilizados no Matlab e em SystemVerilog. Tais transformações incluíram a multiplicação por mil para transformar os números com pontos decimais em números inteiros, além da divisão dos valores resultantes por 255 para manter o intervalo utilizado nas imagens. Após validar as operações de transformação entre os modelos de cores, foi adicionado a sequência referente à equalização da variável H. Baseado nesse programa descrito no Matlab, implementou-se o modelo de referência, descrito em SystemVerilog, no Apêndice A – A.6 – *task* ref_mod_histograma, onde pode-se observar todas as etapas do filtro de equalização de histograma. Inicialmente, a partir da linha 20 realiza-se a transformação da imagem do modelo de cor RGB para o modelo de cor HSV.

```
20          for(i=0;i<numLinhas*numColunas;i++)
21            begin
22              r = Vet_inR[i];
23              g = Vet_inG[i];
24              b = Vet_inB[i];
25              min = r;
26              if(g<min)
27                min = g;
28              if(b<min)
29                min = b;
```

```
30              max = r;
31              if(g>max)
32                max = g;
33              if(b>max)
34                max = b;
35              v = max;
36               if (max>0)
37                 begin
38                   delta = max - min;
39                   s =((delta*255)/max);
40                   if(r==max)
41                     if (g>=b)
42                       h = ((((60*(g-b))/delta)));
44                     else
45                       h = ((((60*(g-b))/delta))+360);
46                   else
47                     if(g==max)
48                       h = ((((60*(b-r))/delta))+120);
49                     else
50                       h = ((((60*(r-g))/delta))+240);
51                 end
52               else
53                 begin
54                   delta = 0;
55                   s = 0;
56                   h = 0;
57                 end
58               Vet_H[i] = h;
59               Vet_S[i] = s;
60               Vet_V[i] = v;
61             end
```

Na linha 63 inicia-se o processo de equalização do canal H, após a transformação da imagem inicial. Nas linhas 66 a 69, ocorre a inicialização do vetor que armazenará o histograma original com valores iguais à zero.

```
66          for (i=0;i<361;i++)
67            begin
68              hist_origh[i] = 0;
69             end
```

Seguindo para as linhas 71 a 75, é realizada a soma de cada valor do histograma para compor o histograma acumulado da imagem transformada.

```
71          for (i=0;i<numLinhas*numColunas;i++)
72            begin
73              pixh = Vet_H[i];
74              hist_origh[pixh] = (hist_origh[pixh]+1);
75            end
```

Dentre as linhas 77 e 83, observa-se a criação do vetor auxiliar Y com as informações do vetor do histograma acumulado previamente.

```
77          for (i=0;i<361;i++)
78            begin
79              if (i==0)
80                Yh[i]=hist_origh[i];
```

```
81              else
82                 Yh[i]=(hist_origh[i]+Yh[i-1]);
83           end
```

Finalizando o processo de equalização, da linha 85 a 89, realiza-se a equalização da imagem original, por meio do vetor contendo as informações do canal H e do vetor auxiliar Y. O núcleo dessa equalização pode ser analisada através da linha 88, na qual se observa a multiplicação do valor do pixel pelo número de níveis do canal H, seguida pela divisão pelo valor total de pixels da imagem equalizada.

```
85           for (i=0;i<numLinhas*numColunas;i++)
86             begin
87               pixh = Vet_H[i];
88               Vet_Heq[i] = ((n_levels*Yh[pixh])/(numLinhas*numColunas));
89             end
```

Concluindo o filtro de equalização, a partir da linha 92, observa-se a transformação da imagem equalizada de volta para o modelo de cor RGB.

```
92           for (i=0;i<numLinhas*numColunas;i++)
93             begin
94               h = Vet_Heq[i];
95               s = Vet_S[i];
96               v = Vet_V[i];
97               if(s==0)
98                 begin
99                   r = v;
100                  g = v;
101                  b = v;
102                end
103              else
104                begin
105                  hi=(h/60);
106                  f = (((h*1000)/60)-(hi*1000));
107                  p = ( (v * (255-s)) / 255);
108                  q = ( (v * (255000 - (s*f))) / 255000);
109                  t = (v*(255000-s*(1000-f)))/255000;
110                  if(hi==0)
111                    begin
112                      r = v;
113                      g = t;
114                      b = p;
115                   end
116                  else
117                    if (hi==1)
118                      begin
119                        r = q;
120                        g = v;
121                        b = p;
122                      end
123                    else
124                      if (hi==2)
125                        begin
126                          r = p;
127                          g = v;
```

```
128                         b = t;
129                       end
130                     else
131                       if (hi==3)
132                         begin
133                           r = p;
134                           g = q;
135                           b = v;
136                         end
137                       else
138                         if (hi==4)
139                           begin
140                             r = t;
141                             g = p;
142                             b = v;
143                           end
144                         else
145                           begin
146                             r = v;
147                             g = p;
148                             b = q;
149                           end
150             end
```

A figura 4.4 ilustra a máquina de estados desenvolvida para o filtro de equalização de histograma, baseada no modelo de referência descrito no Apêndice A – A.6.

Na figura 4.4, observa-se o estado INATIVO, que assim como nos demais filtros, é o responsável por inicializar as variáveis e assim que o filtro é habilitado, a máquina de estados passa para o próximo estado.

Inicialmente, a imagem original tem de ser transformada do modelo RGB para o modelo HSV. Essa transformação está descrita nos estados RGB HSV, LEITURA 1, DET MIN MAX e CALCULA HSV. No estado RGB HSV é determinado o endereço do dado a ser lido no estado LEITURA 1. No estado DET MIN MAX, é determinado o valor máximo e o valor mínimo, dentre os valores lidos RGB. E no estado CALCULA HSV, são realizados os cálculos para determinar os canais HSV, de acordo com as equações descritas na seção 2.2.3.3.

Após ser realizada a transformação para o modelo HSV, inicia-se o processo de equalização do histograma do canal H. Assim, no estado INIC HIST, o vetor de 360 posições que receberá o histograma do canal H, é inicializado com zeros. Nos estados, LEITURA PIX 1, LEITURA PIX 2 e SOMA HIST, é realizada a acumulação do histograma, ou seja, a cada valor lido do canal H, esse valor é somado à sua posição respectiva dentro do vetor do histograma, que pode variar de 0 a 360.

A seguir, nos estados LEITURA HIST Y e SOMA Y, os valores do vetor do histograma são passados a um vetor auxiliar Y, que será utilizado no processo de equalização. Assim o vetor Y pode ser representado pela equação Y[i] = hist[i] + Y[i-1], onde i varia de 0 a 360.

Nos próximos estados, LEITURA H, LEITURA Y e CALCULA HIST, o histograma é equalizado. Sendo que no estado LEITURA H, o vetor com os valores do histograma são lidos e no estado LEITURA Y, os valores do vetor Y são lidos. Por fim, no estado CALCULA HIST, o histograma é equalizado de acordo com a equação 11.

Após a equalização do histograma do canal H, a imagem tem que ser transformada de volta para o modelo RGB. Assim, a transformação do modelo HSV para o modelo RGB é descrita nos estados HSV RGB, LEITURA 2, CALCULA RGB e ESCRITA de acordo com a transformação descrita na seção 2.2.3.3.

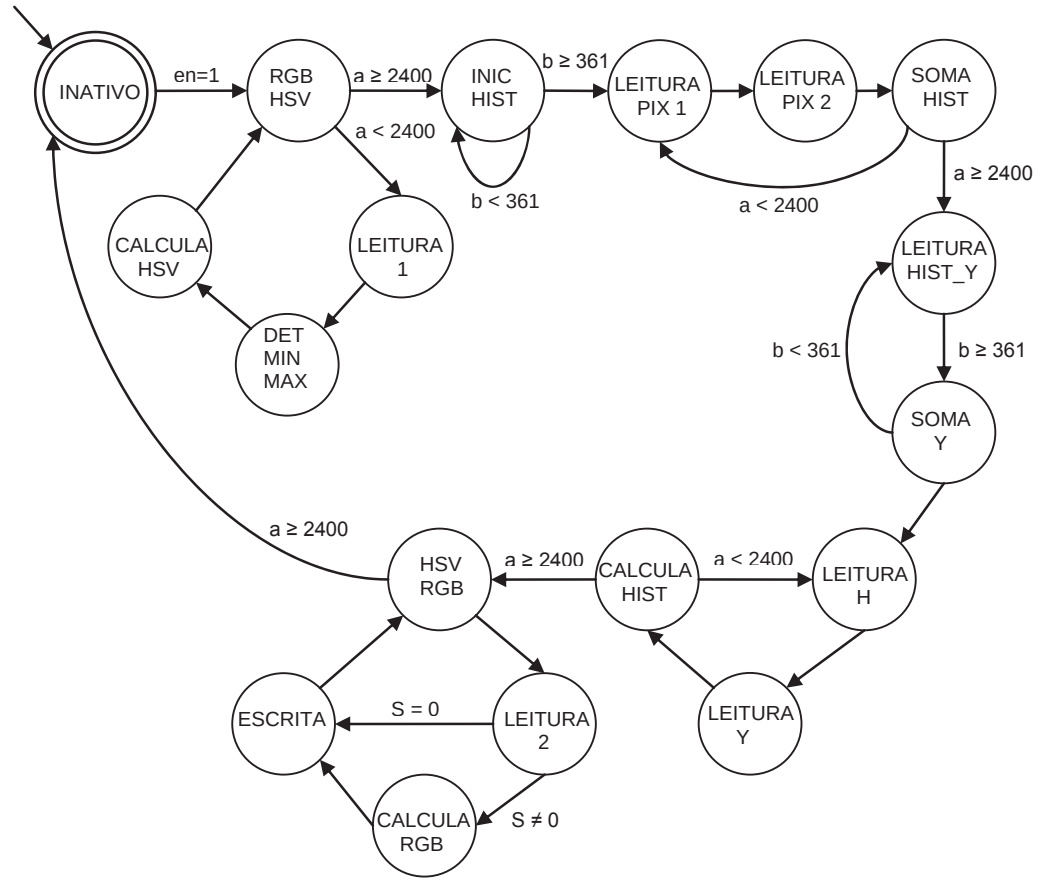


Figura 4.4: Máquina de estados do filtro de equalização de histogramas.

Na implementação do filtro de equalização de histograma em FPGA, assim como para os filtros de normalização de cores e de luminância, foi necessária a utilização de blocos de memórias para o armazenamento dos vetores que contém os canais HSV, e os vetores para armazenar o histograma e o vetor auxiliar Y. Esses blocos de memórias foram descritos utilizando modelos disponibilizados pela ferramenta Quartus II. Assim, esses blocos de memórias são compilados utilizando áreas específicas na FPGA para esse fim.

Então, para cada leitura ou escrita de valores em algum desses vetores, é necessário acessar os blocos de memórias, respeitando o tempo de acesso desses blocos. Dessa forma, as operações de leitura em algum desses vetores são acrescidos de um estado extra na máquina de estado para respeitar o tempo de acesso do bloco de memória correspondente.

4.1.4 Filtro para Normalização de Cores

Existe uma grande variedade de algoritmos para normalização de cores, um dos mais comuns é o algoritmo *Gray World*. O algoritmo *Gray World* assume que a média das cores de uma imagem é cinza [Kyung, 2012]. Dessa forma, o algoritmo *Gray World* estima a média dos canais RGB da imagem a ser tratada, e a corrige baseada na variável cinza. A variável cinza é calculada através da média da soma dos valores dos canais RGB.

O filtro, para normalização de cores, desenvolvido foi baseado no algoritmo descrito em [Li, 2012]. Esse algoritmo já se encontra descrito na linguagem própria para o *software* Maltab, dessa forma, a partir dele implementou-se o modelo de referência do filtro de normalização de cores no Apêndice A : A.7 - *task* ref_mod_norm_cores.

Esta *task* inicia-se, na linha 19, com a normalização da imagem de entrada e com o somatório dos valores dos pixels de cada vetor RGB.

```
19          pos = 0;
20          sumR = 0;   sumG = 0;      sumB = 0;
21          for(i=0;i<numLinhas*numColunas;i++)
22            begin
23                Vet_inR2[i] = ((Vet_inR[i]*1000)/256);
24                Vet_inG2[i] = ((Vet_inG[i]*1000)/256);
25                Vet_inB2[i] = ((Vet_inB[i]*1000)/256);
26                sumR = sumR + Vet_inR2[i];
27                sumG = sumG + Vet_inG2[i];
28                sumB = sumB + Vet_inB2[i];
```

```
29              pos = pos +1;
30            end
```

Os resultados destes somatórios são utilizados para a determinação dos valores médios de cada vetor, além do valor da constante cinza, nas linhas 33 a 36.

```
33          medR = (sumR/pos);
34          medG = (sumG/pos);
35          medB = (sumB/pos);
36          medGray = ((medR+medG+medB)/3);
```

Entre as linhas 39 e 53, ocorre a normalização característica do algoritmo *GrayWorld*. Assim, cada pixel, de cada vetor da imagem, tem seu valor multiplicado pela constante cinza e dividido pela média daquele vetor, como pode ser visualizado na linha 44 para o vetor R.

```
39          for(i=0;i<numLinhas*numColunas;i++)
40            begin
41              if (medR == 0)
42                Vet_outRi[i] = Vet_inR2[i];
43              else
44                Vet_outRi[i] = ((medGray*Vet_inR2[i])/medR);
45              if (medG == 0)
46                Vet_outGi[i] = Vet_inG2[i];
47              else
48                Vet_outGi[i] = ((medGray*Vet_inG2[i])/medG);
49              if (medB == 0)
50                Vet_outBi[i] = Vet_inB2[i];
51              else
52                Vet_outBi[i] = ((medGray*Vet_inB2[i])/medB);
53            end
```

A partir da linha 55, observa-se uma série de ajustes para a correta normalização da imagem resultante. Dentre as linhas 59 e 67, são determinados os pixels com valor máximo de cada vetor RGB.

```
59          for (i=0;i<numLinhas*numColunas;i++)
60            begin
61              if (Vet_outRi[i]>maxR)
62                maxR=Vet_outRi[i];
63              if (Vet_outGi[i]>maxG)
64                maxG=Vet_outGi[i];
65              if (Vet_outBi[i]>maxB)
66                maxB=Vet_outBi[i];
67            end
```

Nas linhas 71 a 80, é determinado o máximo dentre os valores máximos para cada vetor, determinando assim um único valor máximo RGB.

```
71          if (maxR > maxG)
72            if (maxR > maxB)
73              maxRGB = maxR;
```

```
74              else
75                maxRGB = maxB;
76            else
77              if (maxG > maxB)
78                maxRGB = maxG;
79              else
80                maxRGB = maxB;
```

A partir da linha 83, ocorre o ajuste da escala da imagem resultante para garantir que a imagem final esteja dentro do intervalo esperado pelas memórias externas.

```
83          for (i=0;i<numLinhas*numColunas;i++)
84            begin
85              if  (maxRGB > 1000)
86                begin
87                  Vet_outR[i] = ((Vet_outRi[i]*255)/maxRGB);
88                  Vet_outG[i] = ((Vet_outGi[i]*255)/maxRGB);
89                  Vet_outB[i] = ((Vet_outBi[i]*255)/maxRGB);
90                end
91              else
92                begin
93                  Vet_outR[i] = (Vet_outRi[i]*255);
94                  Vet_outG[i] = (Vet_outGi[i]*255);
95                  Vet_outB[i] = (Vet_outBi[i]*255);
96                end
97            end
```

Após a descrição do modelo de referência, passou-se para a implementação do circuito sintetizável. A figura 4.5 ilustra a máquina de estados desenvolvida para esse circuito sintetizável que representa o filtro de normalização de cores.

A máquina de estados descrita na figura 4.5 inicia-se pelo estado INATIVO, onde as variáveis são inicializadas e após a habilitação do filtro, a máquina avança para o próximo estado NORM VETOR 1. Nos estados NORM VETOR 1, LEITURA 1 e SOMA 1 é realizada a normalização da imagem inicial e a soma dos valores de cada canal RGB. Assim, os valores da imagem inicialmente estão entre 0 e 255, e após a normalização estarão entre 0 e 1. Como esses valores normalizados serão armazenados em blocos de memórias, assim como no filtro de equalização de histogramas, os valores de 0 a 1, foram multiplicados por 1000, para poderem ser corretamente armazenados nos blocos de memória.

No estado DET MED, são determinados os valores médios da soma de cada canal RGB. Os valores médios são determinados pela divisão do valor final da soma pela quantidade de pixels da imagem, nesse caso é igual

a 2400 pixels. A determinação da variável cinza, que será utilizada na etapa de normalização das cores de acordo com o algoritmo *GrayWorld*, é realizada no estado NORM VETOR 2. A variável cinza é determinada pela soma dos valores medianos do canal RGB, seguida pela divisão por 3 desse valor.

Através dos estados NORM VETOR 2, LEITURA 2, SOMA 2, DET MAX e ESCRITA VETOR, é realizado o processo de normalização das cores da imagem. Assim, inicialmente, os valores de cada canal RGB, que estão armazenados no vetor normalizado inicialmente, são lidos. Cada valor é normalizado pela multiplicação da variável cinza pelo valor lido do vetor correspondente, seguida pela divisão pelo valor mediano do canal especifíco, ou seja, Vetor normalizado R = (variável cinza * Vetor R) / (média canal R), para o canal R. Sendo que o mesmo processo ocorre também para os canais G e B. Após a normalização de cada vetor, ele é novamente armazenado no bloco de memória correspondente.

Concomitantemente ao processo de normalização dos vetores dos canais RGB, é realizada a determinação do valor máximo e do valor mínimo dentre os valores RGB. O qual se conclui no estado DET MAX RGB. O valor máximo será utilizado no processo de ajuste da escala do vetor de saída para que os valores dos pixels da imagem final fiquem entre 0 e 255. Assim, nos estados LEITURA 3, PARAM VETOR e ESCRITA ocorre o processo de ajuste de escala do vetor de saída e armazenamento dos vetores que compõem a imagem final.

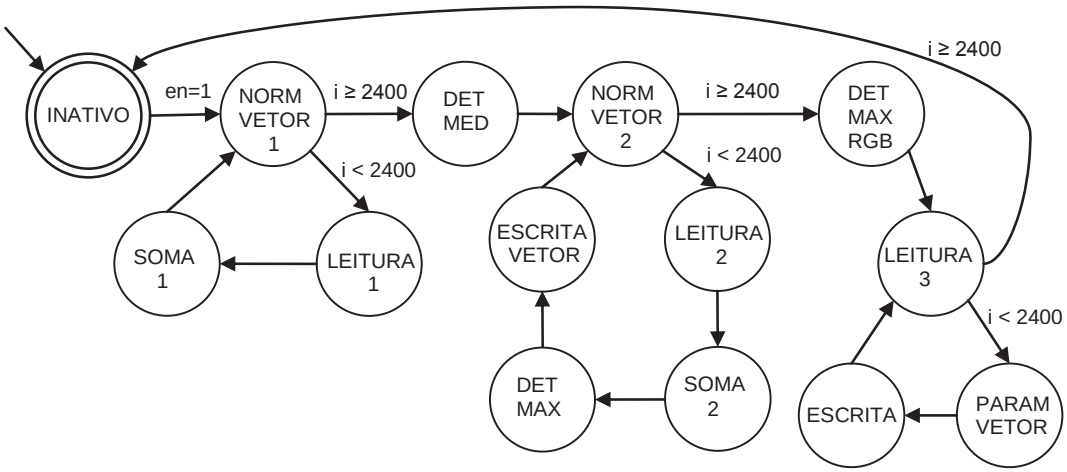


Figura 4.5: Máquina de estados do filtro de normalização de cores.

4.1.5 Filtro para Normalização de Luminância

Na visão humana, a luminosidade é um parâmetro subjetivo difícil de medir, o qual envolve a noção de intensidade, um dos principais fatores na descrição da sensação de cores [Gonzalez e Woods, 2002].

A normalização de luminância consiste na uniformização da variação da luminosidade de uma imagem. A luminosidade de uma imagem pode ser medida através do canal B, de uma imagem descrita utilizando o modelo H, S e B (*Hue, Saturation, Brightness*), sendo que o canal B equivalente ao canal V, no modelo de cores HSV.

Assim, de maneira similar ao filtro de equalização de histograma, o filtro de normalização de luminância utiliza a transformação do modelo RGB para o HSV. Após a transformação da imagem, é aplicada a normalização sobre o canal V, e a imagem é transformada novamente para o modelo RGB. Tal fato abreviou o processo de implementação desse filtro, partindo-se diretamente para a implementação do modelo de referência em SystemVerilog, sem a necessidade de validar o algoritmo no Matlab.

A partir da *task* ref_mod_norm_luminancia, descrito no Apêndice A – A.8, podem-se ver as etapas em comum com o filtro de equalização de histograma, como por exemplo, as transformações entre os modelos de cores que ocorrem no início e no final do processo de filtragem. Dessa forma, a partir da linha 20, realiza-se a transformação da imagem do modelo de cor RGB para o modelo de cor HSV. Na linha 63 inicia-se o processo de normalização da luminância. Nas linhas 65 a 74, ocorre a busca pelos valores máximo e mínimo dentro do vetor V.

```
65          maxV=0;
66          minV=255;
67          for (i=0;i<numLinhas*numColunas;i++)
68            begin
69              pixV = Vet_V[i];
70              if (pixV > maxV)
71                maxV = pixV;
72              if (pixV < minV)
73                minV = pixV;
74            end
```

Seguindo para as linhas 77 a 81, onde é realizada a normalização do vetor V de acordo com os valores máximo e mínimo determinados anteriormente.

```
77          for (i=0;i<numLinhas*numColunas;i++)
78            begin
79              pixV = Vet_V[i];
80              Vet_Vi[i] = (((Vet_V[i] - minV)*255)/(maxV - minV));
81            end
```

Concluindo o processo de filtragem, a partir da linha 83, observa-se a transformação da imagem normalizada, descrita no modelo HSV para o modelo de cor RGB.

Com base no modelo de referência implementado, criou-se o circuito sintetizável para o filtro de normalização de luminância composto pela máquina de estados descrita na figura 4.6.

A partir da figura 4.6, observa-se o estado INATIVO, assim como nos demais filtros, é o responsável por inicializar as variáveis e assim que o filtro é habilitado, a máquina de estados passa para o próximo estado.

Inicialmente, a imagem inicial tem que ser transformada do modelo RGB para o modelo HSV, assim como no filtro de equalização de histograma. Essa transformação está descrita nos estados RGB HSV, LEITURA 1, DET MIN, DET MAX e CALCULA HSV. No estado RGB HSV é determinado o endereço do dado a ser lido no estado LEITURA 1. Nos estados DET MIN e DET MAX, são determinados o valor máximo e o valor mínimo, respectivamente, dentre os valores lidos RGB. No estado CALCULA HSV, são realizados os cálculos para determinar os canais HSV, de acordo com as equações descritas na seção 2.2.3.3. Concomitantemente ao processo de transformação de modelo RGB para HSV, ocorre a determinação do valor máximo e do valor mínimo do canal V. Tais valores serão utilizados no processo de normalização desse canal.

Após ser realizada a transformação para o modelo HSV, inicia-se o processo de normalização do canal V, através do estado NORM V. Nesse estado é enviado o endereço do pixel do canal V a ser lido no estado LEITURA V. Assim, no estado ESCRITA V, é calculado o valor normalizado do canal V a ser enviado para o vetor que armazena os valores desse canal. A normalização do canal V pode ser descrita por: valor normalizado = (valor inicial – valor mínimo de V) / (valor máximo de V – valor mínimo de V).

Com a normalização do canal V concluída, a imagem atual será transformada novamente para o modelo de cores RGB. A transformação do modelo HSV para o modelo RGB é descrita nos estados HSV RGB, LEITURA 2, CALCULA VAR, CALCULA RGB e ESCRITA de acordo com a transformação descrita na seção 2.2.3.3.

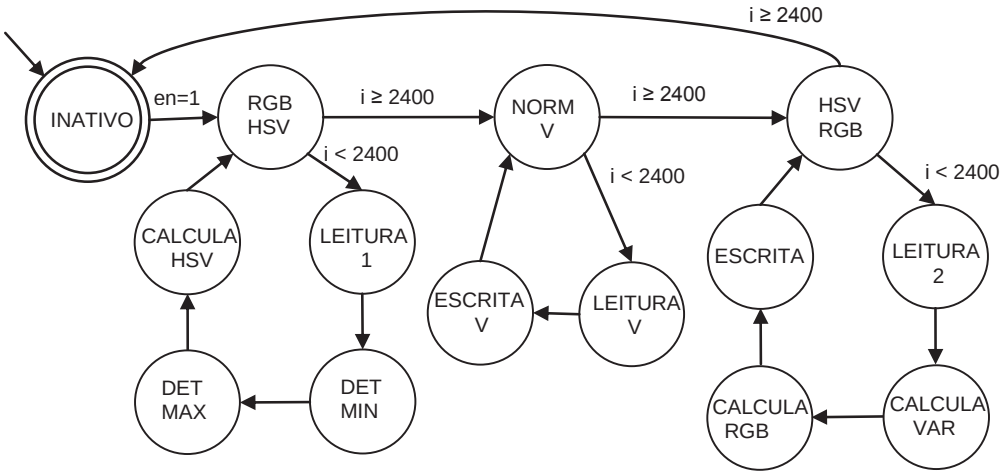


Figura 4.6: Máquina de estados do filtro de normalização de luminância.

4.2 Desenvolvimento do Controlador

O conjunto de filtros descrito na seção 4.1 pode ser aplicado a diferentes imagens devido ao controlador desenvolvido para promover a escolha dos filtros e aplicação de cada um deles de maneira individual para cada aplicação. Assim, pode-se determinar quais os filtros necessários e em que ordem esses filtros deverão ser aplicados dependendo do tipo de imagem a ser tratada.

Então, as informações sobre quais filtros e a ordem de aplicação são passadas ao controlador que habilitará cada filtro na ordem especificada. Essas informações são armazenadas em uma memória de programa devidamente preparada para ser utilizada no sistema composto pelo FPGA. O mecanismo para criação e preenchimento dessa memória está descrito na seção 5.2.

Para o controlador não foi implementado um modelo de referência como ocorreu no processo de desenvolvimento dos filtros. Dessa forma, o circuito sintetizável desenvolvido para o controlador foi implementado

diretamente através de uma máquina de estados, a qual está representada na figura 4.7.

Na figura 4.7, observa-se o estado INATIVO como sendo o inicial dessa máquina de estados. Nesse estado, as variáveis do controlador são inicializadas e é lida a variável *nf* do arquivo contendo as informações de controle. A variável *nf* corresponde ao número de filtros que serão aplicados pelo controlador para aquela aplicação. Ainda, nesse estado é aguardado o sinal (*en=1*) de habilitação para o início daquela aplicação.

Assim, após a variável *nf* ser lida, a máquina de estados passa para o estado LEITURA DADOS, onde o endereço, com a informação sobre qual aplicação será executada, será lido no estado LEITURA NÚMERO. No estado LEITURA APL, a aplicação atual que o controlador ativará é atualizada, e o endereço para o primeiro filtro, a ser aplicado, é enviado e este será lido no estado ATIVA APL. No estado ATIVA FILTRO o filtro especificado é habilitado e aguarda-se até o término de sua execução, indicado pela variável *fim*. Após a aplicação do filtro, as variáveis são atualizadas no estado LEITURA DADO APL, reiniciando o processo para a execução do próximo filtro até que todos os filtros determinados para aquela aplicação sejam aplicados. Ao retornar para o estado INATIVO a próxima aplicação pode ser habilitada pelo sinal *en*, assim tantas aplicações quantas forem determinadas pelo arquivo de controle, poderão ser executadas.

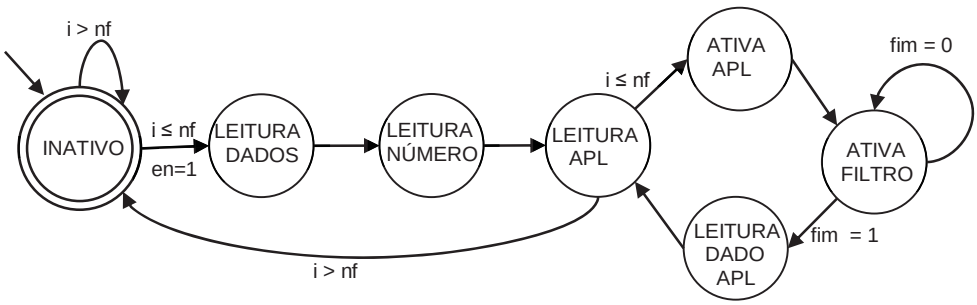


Figura 4.7: Máquina de estados do controlador de filtros.

4.3 Desenvolvimento em FPGA

O sistema contendo o conjunto de filtros e o controlador, descritos nas seções anteriores, foram desenvolvidos visando sua implementação física

em uma placa de FPGA da Altera, especificamente a placa de desenvolvimento DE2-70 [Terasic, 2009].

Conforme pode ser observado na figura 4.8, o sistema do topo contém a instanciação de um conjunto de memórias que armazena a imagem inicial lida a partir da memória *Flash* da placa. Na memória *Flash*, as imagens a serem tratadas são descritas nos vetores RGB. Sendo que os valores para cada vetor são armazenados sequencialmente, ou seja, primeiro temos todo o vetor R, seguido pelo vetor G e por fim o vetor B. Dessa forma, ao iniciar o sistema na placa, é necessário que a imagem a ser processada seja lida e armazenada no conjunto de memórias instanciado no módulo topo.

Ainda no topo, temos a instanciação de um PLL (*Phase Locked Loop*), para a divisão do *clock* vindo da placa de desenvolvimento antes de ser enviado ao sistema de processamento de imagens. O *clock* vindo da placa chega a uma frequência de 50 MHz, sendo enviado ao sistema um *clock* com frequência de 10 MHz.

Além de ser o responsável pela aplicação dos filtros na ordem determinada, o controlador também determina o acesso às memórias durante a aplicação de cada filtro. Assim, o controlador instancia dois conjuntos de três memórias, que armazenam os vetores RGB que compõem as imagens a serem tratadas. No início do processamento, a imagem inicial é armazenada em um desses conjuntos com três memórias e é enviada ao primeiro filtro. A imagem resultante desse filtro é armazenada no segundo conjunto de memórias, que enviará o seu conteúdo ao próximo filtro determinado pela aplicação. Logo, o controlador determina qual conjunto de memórias contém a imagem a ser lida e tratada e qual conjunto de memórias armazenará a imagem resultante. Esse tratamento é importante para não haver sobreposição de imagem durante o processamento com vários filtros.

Dessa forma, a imagem inicial a ser processada é armazenada na memória *Flash* da placa de FPGA e o estágio de processamento inicia-se pela ativação dos pinos de entrada da placa de FPGA. O processamento continua até que todos os filtros da aplicação sejam executados. Então, a

imagem final pode ser lida pelas portas de saída da placa de FPGA, representado pelos sinais de saída na figura 4.8. Na figura 4.8 observa-se um esquema do sistema desenvolvido para a placa de FPGA.

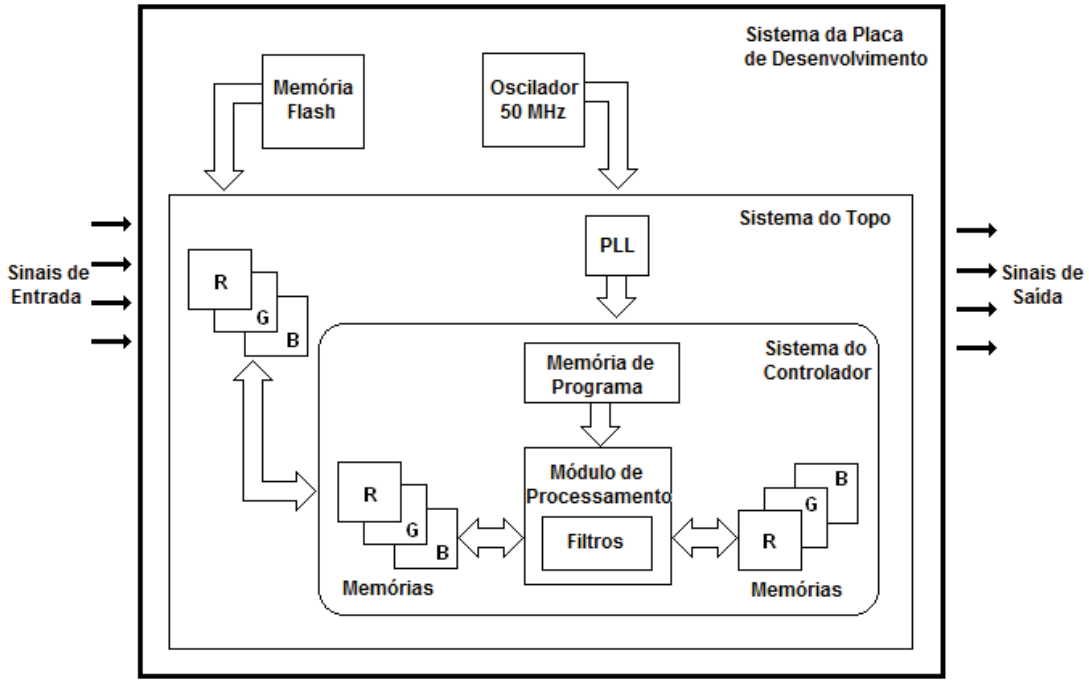


Figura 4.8 : Sistema desenvolvido para a placa de FPGA DE2-70.

Na figura 4.9 observa-se a placa DE2-70 da Altera onde foi implementado o sistema observado na figura 4.8.

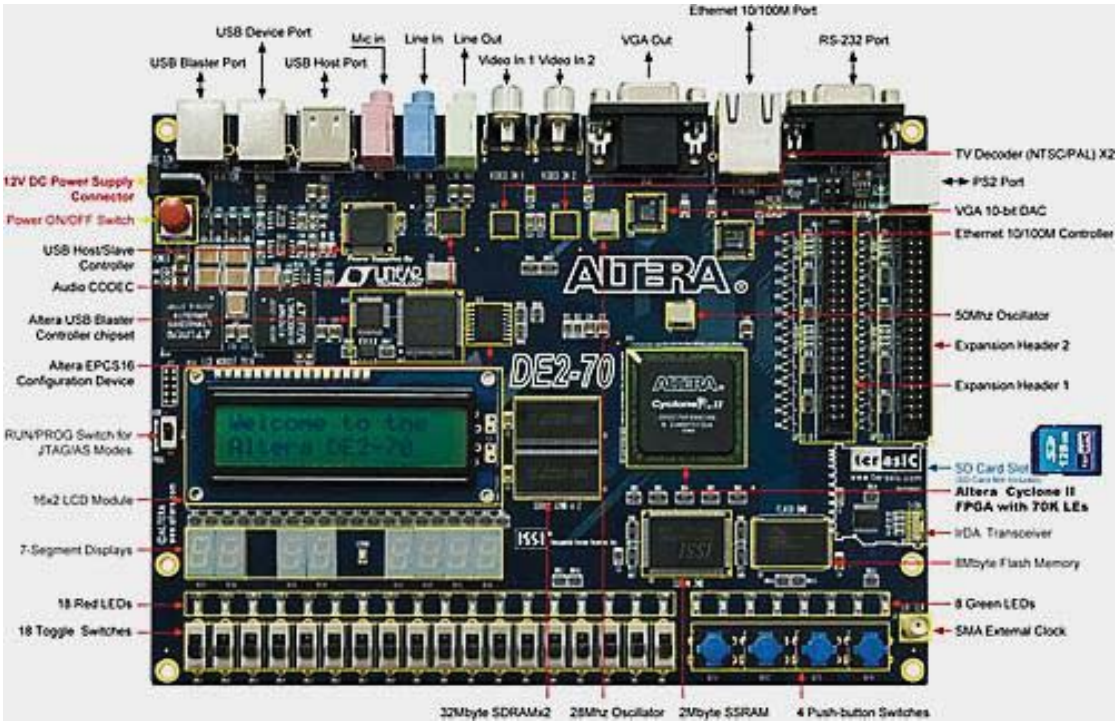


Figura 4.9: Placa DE2-70 da Altera [TERASIC, 2009].

CAPÍTULO 5 - Testes e resultados

5.1 Conjunto de Filtros

5.1.1 Metodologia de Verificação Utilizada

O estágio de verificação foi baseado na Metodologia Universal de Verificação (*Universal Verification Methodology* – UVM) [Accelera, 2001]. Assim, foi criado um módulo *driver* e um módulo *monitor* utilizando rotinas desenvolvidas para maximizar o reuso na verificação de cada um dos módulos. Para cada filtro, um *driver* foi desenvolvido, o qual envia sinais para o DUV, foi desenvolvido também um monitor que compara os resultados, simulando o sistema real no qual o sistema estiver inserido.

Dessa forma, o *driver* envia diferentes tipos de estímulos ao DUV e ao *monitor* simultaneamente. O *monitor* recebe a imagem de entrada e sinais de controle do *driver*, calcula a imagem resultante através do modelo de referência e compara com os resultados vindos do DUV. Durante o processamento, o *monitor* envia mensagens informando sobre as etapas do processo e sobre os resultados. Na figura 5.1, observa-se o diagrama do ambiente de verificação utilizado.

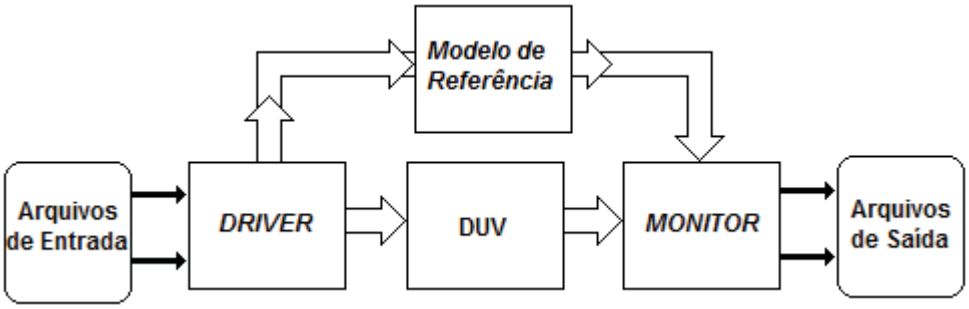


Figura 5.1: Ambiente de verificação.

Dois tipos básicos de estímulos foram criados. O primeiro tipo utiliza números pseudo-aleatórios para testar o filtro de maneira geral. O segundo tipo de estímulo utiliza um conjunto de imagens descritas em arquivos de texto para testar os filtros em diferentes aplicações.

No teste com números pseudo-aleatórios, esses números foram gerados utilizando uma função específica da linguagem SystemVerilog, a função *$random* [Accelera, 2004]. Nesse caso, o *monitor* compara os resultados vindos do modelo de referência com os resultados do DUV e envia mensagens de acordo com o processo de simulação na ferramenta de simulação ModelSim. Nesse tipo de estímulo somente o arquivo de log é armazenado após o final da simulação.

No teste utilizando imagens como estímulos, as imagens criadas são lidas pelo *driver*, através de arquivos de texto contendo os canais RGB separadamente, ou seja, uma imagem consiste em três arquivos de texto com os canais RGB descritos em valores hexadecimais. Durante o teste com imagens, o *monitor* compara os resultados do modelo de referência com os resultados vindos do DUV e guarda a imagem final em arquivos de texto. Então, com os arquivos finais foi possível realizar a comparação com a imagem resultante do algoritmo implementado, utilizando a ferramenta Matlab, uma garantia extra com relação à correta operação do filtro testado.

5.1.2 Vetores de Teste

Os testes com imagens utilizaram alguns tipos de imagens, incluindo imagens comuns, imagens médicas e imagens agrícolas. Para garantir uma adequada implementação na placa de FPGA essas imagens foram redimensionadas para 60x40 pixels. Na figura 5.2 são apresentadas as

imagens que serviram de base para compor os vetores de testes utilizados na etapa de verificação realizada com os filtros desenvolvidos neste trabalho. As imagens médicas foram retiradas de [Dermis, 2012], as imagens agrícolas de [Mertes, Marranghello e Pereira, 2008] e as imagens comuns de [Graci, 2011].

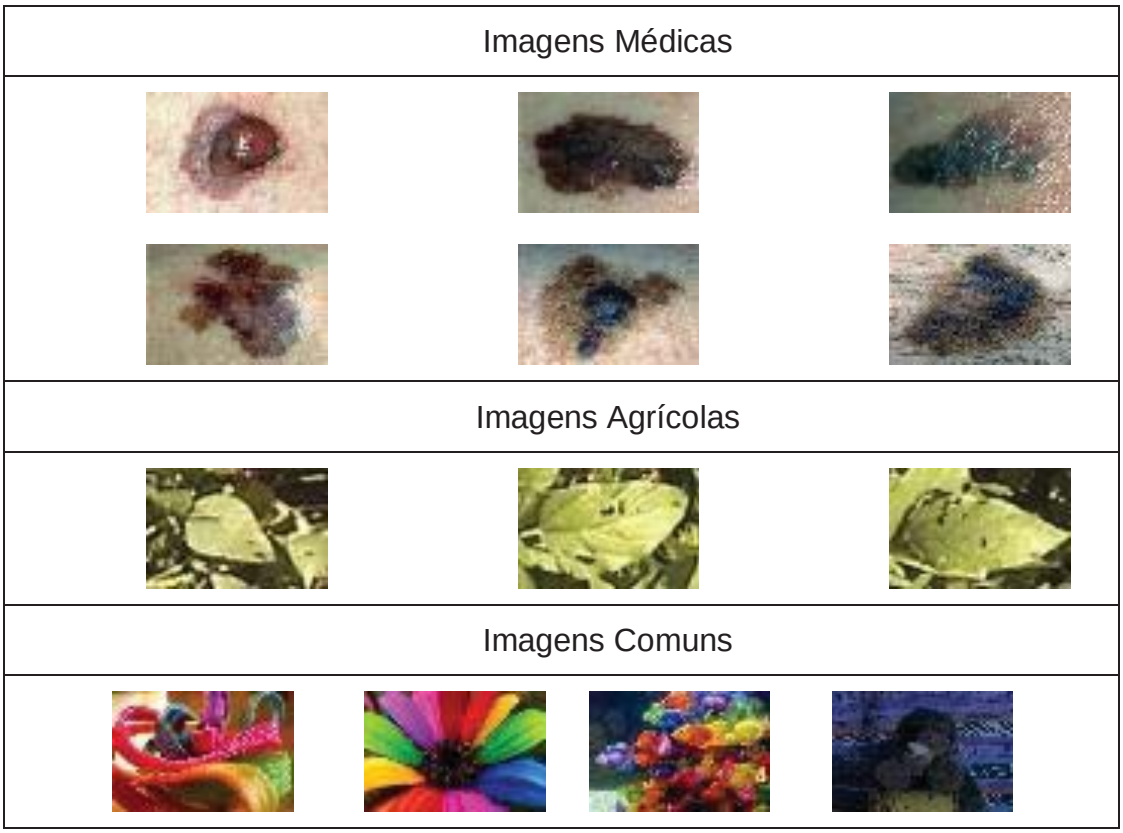


Figura 5.2: Conjunto base para geração de imagens de teste.

5.1.3 Resultados Obtidos

A seguir são apresentados os resultados obtidos, a partir das imagens apresentadas na figura 5.2, para cada filtro implementado.

5.1.3.1 Filtros de suavização

Para ressaltar os resultados dos filtros de suavização foram inseridos ruídos do tipo *salt and pepper* em algumas imagens do conjunto de imagens comuns. Esse tipo muito comum de ruído foi inserido nas imagens com o auxílio do *software* Maltab. Para tanto, foi criado um programa para auxiliar nessa tarefa, tal programa chamado *coloca_ruido* está descrito no Apêndice A – A.9.

A partir do programa *coloca_ruido* foi possível inserir ruídos em cada um dos vetores RGB das imagens comuns, garantindo que os filtros de suavização fossem testados corretamente. Já para o teste das imagens médicas e das imagens agrícolas, as imagens originais foram mantidas para ressaltar a utilização desses filtros em imagens com diferentes tipos de textura. Na figura 5.3 observa-se o resultado da aplicação dos filtros de suavização nos três conjuntos de imagens.

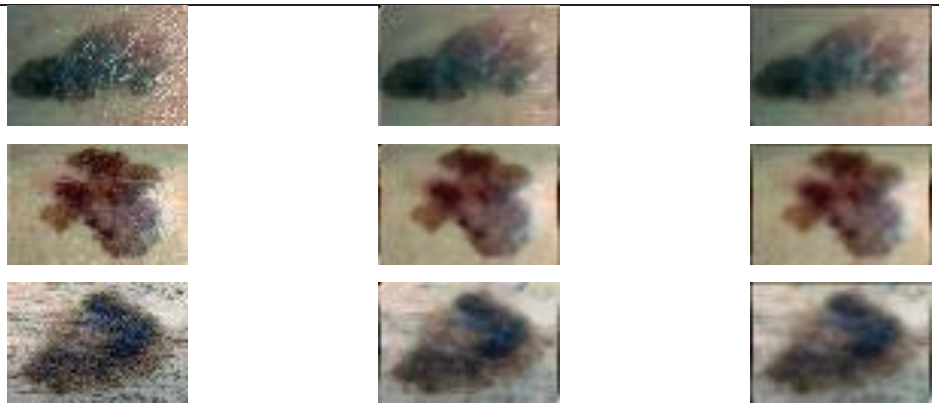
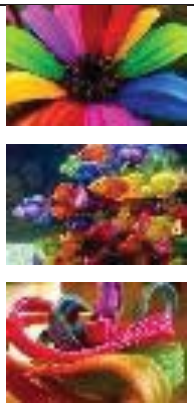
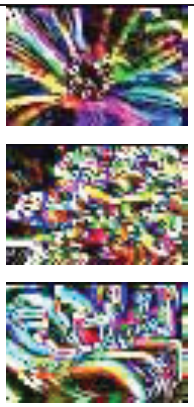
Imagem original	Imagens resultantes da aplicação do filtro	
	Mediana	Média
Imagens Comuns		
		
Imagens Médicas		
		
Imagens Agrícolas		
		

Figura 5.3: Imagens originais (direita), imagens resultantes do filtro de mediana (centro) e imagens resultantes do filtro de média (direita).

Na figura 5.3, observa-se a coluna das imagens originais à esquerda, das imagens resultantes do filtro de mediana ao centro e das imagens resultantes do filtro de média à direita. No grupo de imagens comuns observa-se o nítido efeito dos filtros de suavização a partir das imagens ruidosas. Os resultados do filtro de mediana e de média foram sutilmente diferentes, mas a escolha entre um deles deve ser baseada na aplicação final à qual a imagem original deve ser submetida. Como pode ser observado nos resultados dos grupos de imagens médicas e agrícolas, onde em cada caso obteve-se resultados que ressaltavam detalhes diferentes em cada imagem.

5.1.3.2 Filtro de Detecção de Bordas

Para testar o filtro de detecção de bordas, aplicou-se este filtro nas imagens comuns, nas imagens médicas e nas imagens agrícolas para determinar o grau de precisão do filtro implementado em diferentes objetos. Na figura 5.4, observam-se os resultados obtidos por meio da aplicação do filtro de detecção de borda a esses grupos de imagens.

Imagem original	Imagem resultante
Imagens Comuns	
	
Imagens Médicas	

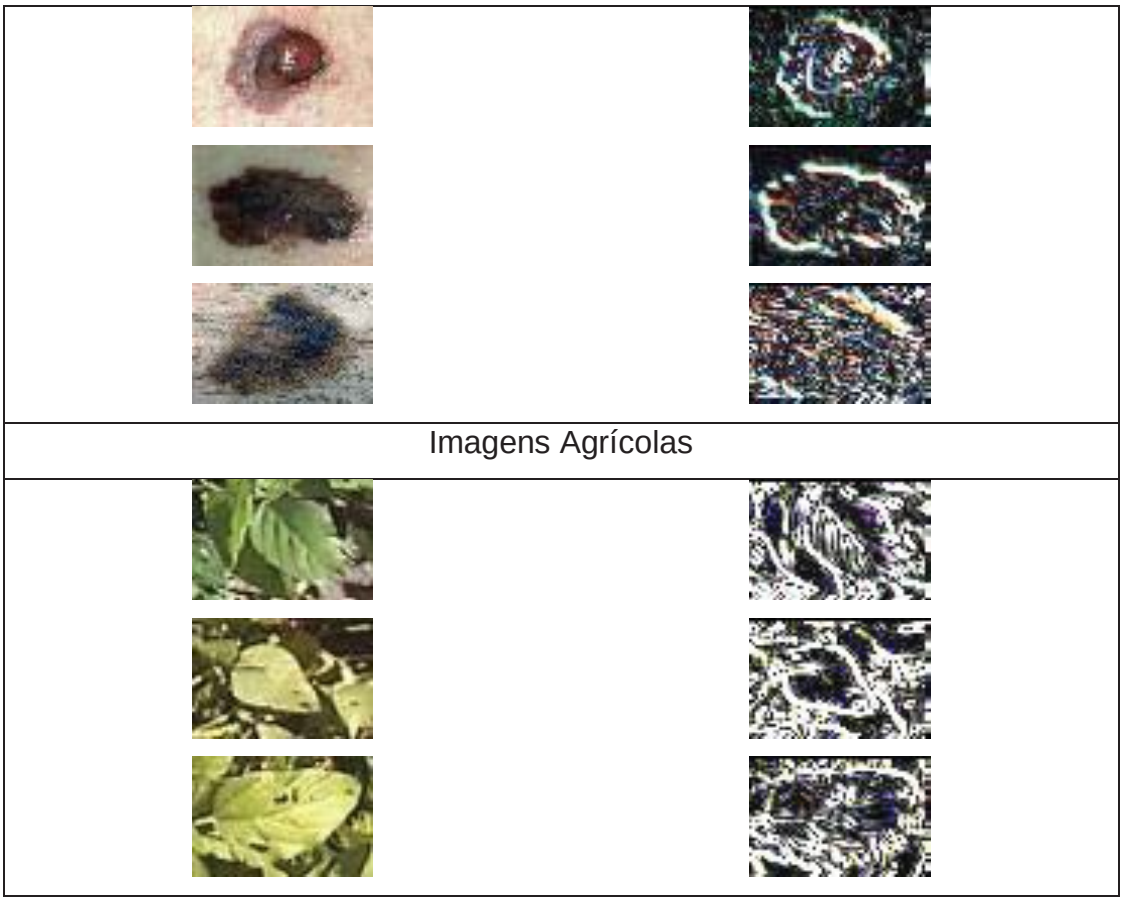


Figura 5.4: Imagens originais (esquerda) e imagens resultante do filtro de detecção de bordas (direita).

A partir da figura 5.4, observa-se que aplicação do filtro de detecção de borda mostrou-se muito sensível à quantidade de cores da imagem original. Dessa forma, as imagens comuns obtiveram as bordas menos destacadas em comparação aos dois outros grupos. Entretanto, os resultados das imagens médicas e das imagens agrícolas mostraram a necessidade da aplicação de um tratamento prévio para melhorar a detecção das bordas pelo filtro implementado. Por esse motivo recomenda-se a aplicação algum tipo de tratamento prévio na imagem para melhorar o resultado do filtro de detecção de bordas, como por exemplo, ajustes nas cores ou na luminosidade da imagem.

5.1.3.3 Filtro de Equalização de Histograma

Assim como no teste para os filtros de suavização, para o teste do filtro de equalização foi necessário alterar as imagens originais para que as imagens resultantes ressaltassem o funcionamento do filtro implementado.

Para tanto, foi criado um programa utilizando a linguagem do *software* Maltab, para ajustar o histograma da componente H das imagens originais. Dessa forma, foi necessária a conversão da imagem entre os modelos de cores RGB e HSV antes e depois do ajuste do histograma. O programa *conv_rgb_hsv_hist* criado para modificar as imagens originais está descrito no Apêndice A – A.10.

Com o auxílio do programa *conv_rgb_hsv_hist* foram criados três grupos para cada imagem original; o grupo das imagens claras, onde o histograma foi deslocado para o intervalo entre 0.5 e 1.0; o grupo das imagens escuras, onde o histograma foi deslocado para o intervalo de 0.0 e 0.5 e, por fim, as imagens medianas, onde o histograma foi centralizado para o intervalo de 0.3 a 0.7. Salientando que o termo clara, escura e mediana está relacionado ao componente H da imagem transformada e não se reflete diretamente na imagem descrita seguindo o modelo de cor RGB.

Na figura 5.5 são apresentados os resultados obtidos a partir da aplicação do filtro de equalização de histograma ao conjunto de imagens comuns, seguindo a divisão criada pelo programa *conv_rgb_hsv_hist*. Na parte superior da figura 5.5, observa-se a imagem modificada e logo abaixo a imagem resultante da aplicação do filtro de equalização de histograma.

Imagem original	Imagem clara	Imagem escura	Imagem mediana
			
			
			

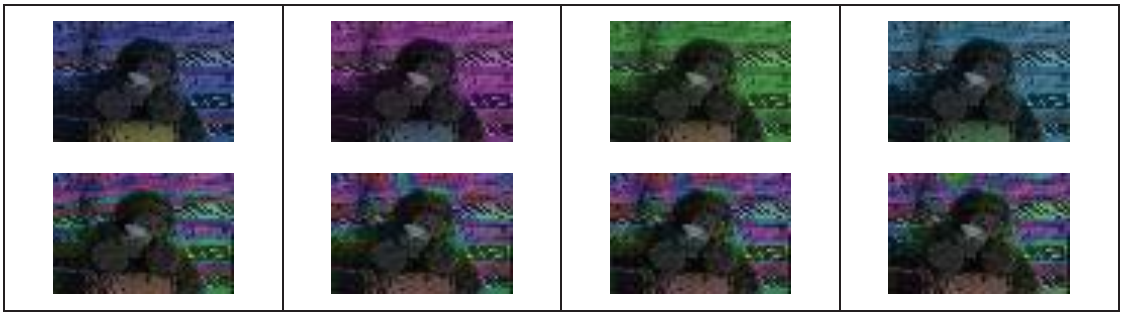


Figura 5.5: Imagens comuns originais (acima) e imagens resultantes do filtro de equalização de histograma (abaixo).

Na figura 5.6 são apresentados os resultados obtidos a partir da aplicação do filtro de equalização de histograma ao conjunto de imagens médicas, seguindo a mesma divisão apresentada na figura 5.5.

Imagem original	Imagem clara	Imagem escura	Imagem mediana

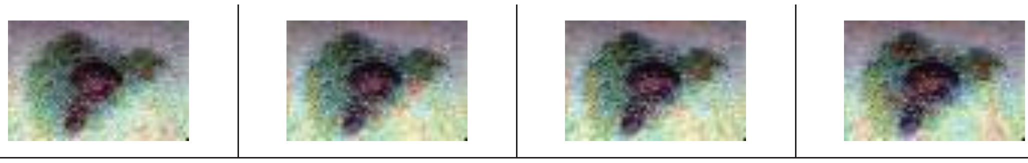


Figura 5.6: Imagens médicas originais (acima) e imagens resultantes do filtro de equalização de histograma (abaixo).

Na figura 5.7 são apresentados os resultados obtidos a partir da aplicação do filtro de equalização de histograma ao conjunto de imagens agrícolas, seguindo a mesma divisão apresentada nas figuras anteriores.

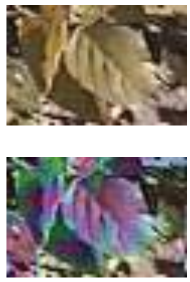
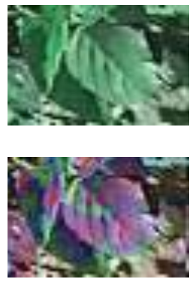
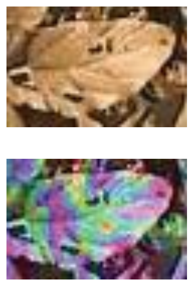
Imagem original	Imagem clara	Imagem escura	Imagem mediana
			
			
			

Figura 5.7: Imagens médicas originais (acima) e imagens resultantes do filtro de equalização de histograma (abaixo).

A partir das figuras 5.5, 5.6 e 5.7 observa-se o funcionamento do filtro de equalização de histograma nos grupos de imagens comuns, médicas e agrícolas, respectivamente. O resultado apresentado em cada um dos grupos de imagens foi diferente devido às características de cada grupo.

Assim, nas imagens com maior quantidade de cores, o resultado do filtro implementado foi mais suave, pois, nesse tipo de imagem, a distribuição das cores no histograma estava um pouco mais uniformizado, facilitando o processo de filtragem. Para as imagens médicas e agrícolas, onde há o predomínio de certos tons nas imagens, a equalização do histograma produziu resultados pouco interessantes, pois o filtro em questão visa redistribuir as cores que estavam concentradas em um determinado tom, produzindo imagens com uma nova distribuição de cores.

Apesar dos resultados nos grupos de imagens terem sido bem diferentes, observou-se que em cada uma das imagens ajustadas, nomeadas como imagem original, clara, escura ou mediana, dependendo da característica do histograma do canal H, as imagens resultantes foram praticamente iguais. Garantindo o bom funcionamento do filtro de equalização de histograma, independente da característica do histograma da imagem a ser tratada.

5.1.3.4 Filtro de Normalização de Cores

Para ressaltar o funcionamento do filtro de normalização de cores foi necessário alterar as imagens originais com o auxílio de um programa criado utilizando a linguagem do *software* Maltab. Tal programa ajusta cada vetor RGB, tornando a imagem mais avermelhada, mais azulada ou mais esverdeada. O programa *ajuste_cor_imagem* criado para modificar as imagens originais está descrito no Apêndice A – A.11.

A partir do programa *ajuste_cor_imagem* foram geradas imagens divididas em vermelho, verde e azul, representando o incremento de cada vetor RGB, respectivamente. Na figura 5.8 são apresentados os resultados obtidos a partir da aplicação do filtro de normalização de cores ao conjunto de imagens comuns, seguindo a mesma divisão criada pelo programa *ajuste_cor_imagem*. Na parte superior de cada coluna da figura 5.8, observam-se as imagens modificadas e logo abaixo as imagens resultantes da aplicação do filtro de normalização de cores.

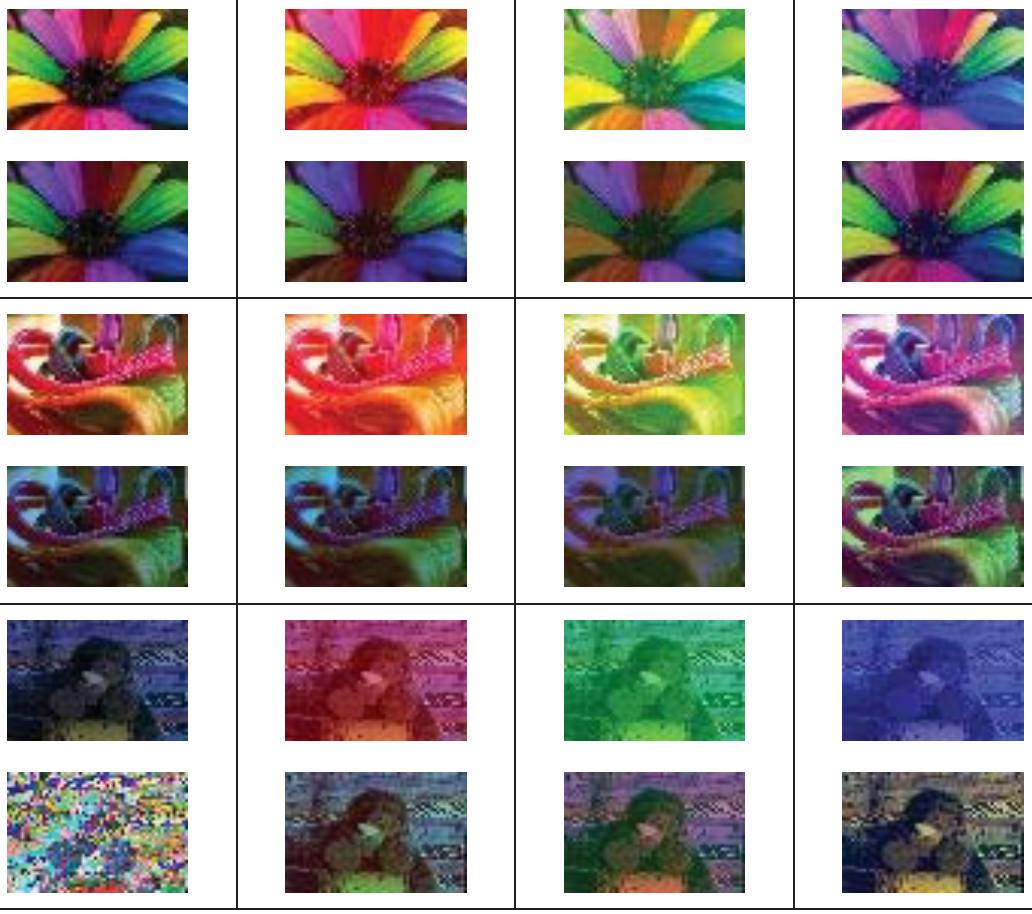
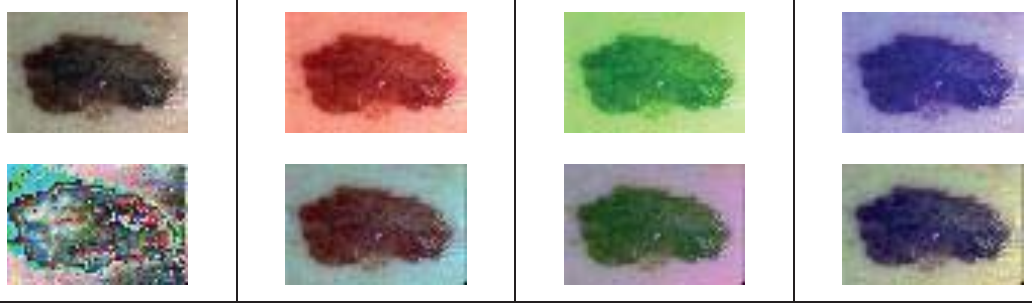
Imagem original	Imagem vermelha	Imagem verde	Imagem blue
			

Figura 5.8: Imagens comuns originais (acima) e imagens resultantes do filtro de normalização de cores (abaixo).

Na figura 5.9 são apresentados os resultados obtidos a partir da aplicação do filtro de normalização de cores ao conjunto de imagens médicas, seguindo a mesma divisão apresentada na figura 5.8.

Imagem original	Imagem vermelha	Imagem verde	Imagem azul
			

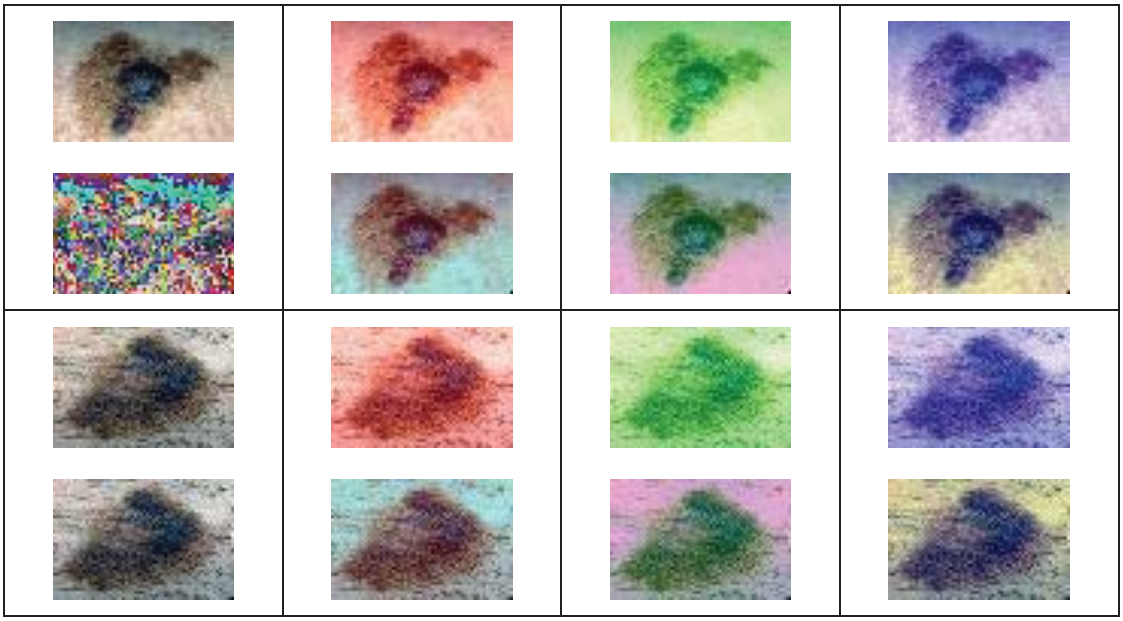


Figura 5.9: Imagens médicas originais (acima) e imagens resultantes do filtro de normalização de cores (abaixo).

Na figura 5.10 são apresentados os resultados obtidos a partir da aplicação do filtro de normalização de cores ao conjunto de imagens agrícolas, seguindo a mesma divisão apresentada anteriormente.

Imagem original	Imagem vermelha	Imagem verde	Imagem azul
			
			
			

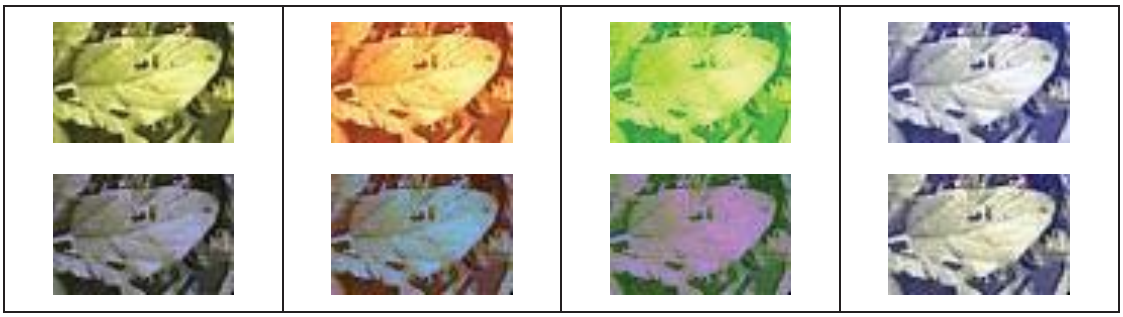


Figura 5.10: Imagens agrícolas originais (acima) e imagens resultantes do filtro de normalização de cores (abaixo).

A partir das figuras 5.8, 5.9 e 5.10 observa-se o funcionamento do filtro de normalização de cores nos grupos de imagens comuns, médicas e agrícolas, respectivamente. O resultado apresentado em cada um dos grupos de imagens não foi tão diferente quanto os resultados do filtro de equalização de histograma. Com exceção de algumas imagens originais que apresentaram resultados discrepantes, a maioria das imagens apresentaram resultados condizentes com o esperado para o filtro de normalização de cores. Os resultados mais perceptíveis, novamente, ocorreram com as imagens comuns, por apresentarem uma gama maior de cores em cada cena. Tanto no grupo de imagens médicas quanto no grupo de imagens agrícolas, as imagens com o vetor B mais acentuado, apresentaram resultados mais sutis. Enquanto que, as imagens com os vetores R e G realçados, apresentaram imagens resultantes com maior diferença em relação à imagem original.

5.1.3.5 Filtro de Normalização de Luminância

Na figura 5.11, observam-se as imagens resultantes após a aplicação do filtro de normalização de luminância aplicada a imagens comuns, médicas e agrícolas.

Imagem original	Imagem resultante
Imagens Comuns	
	

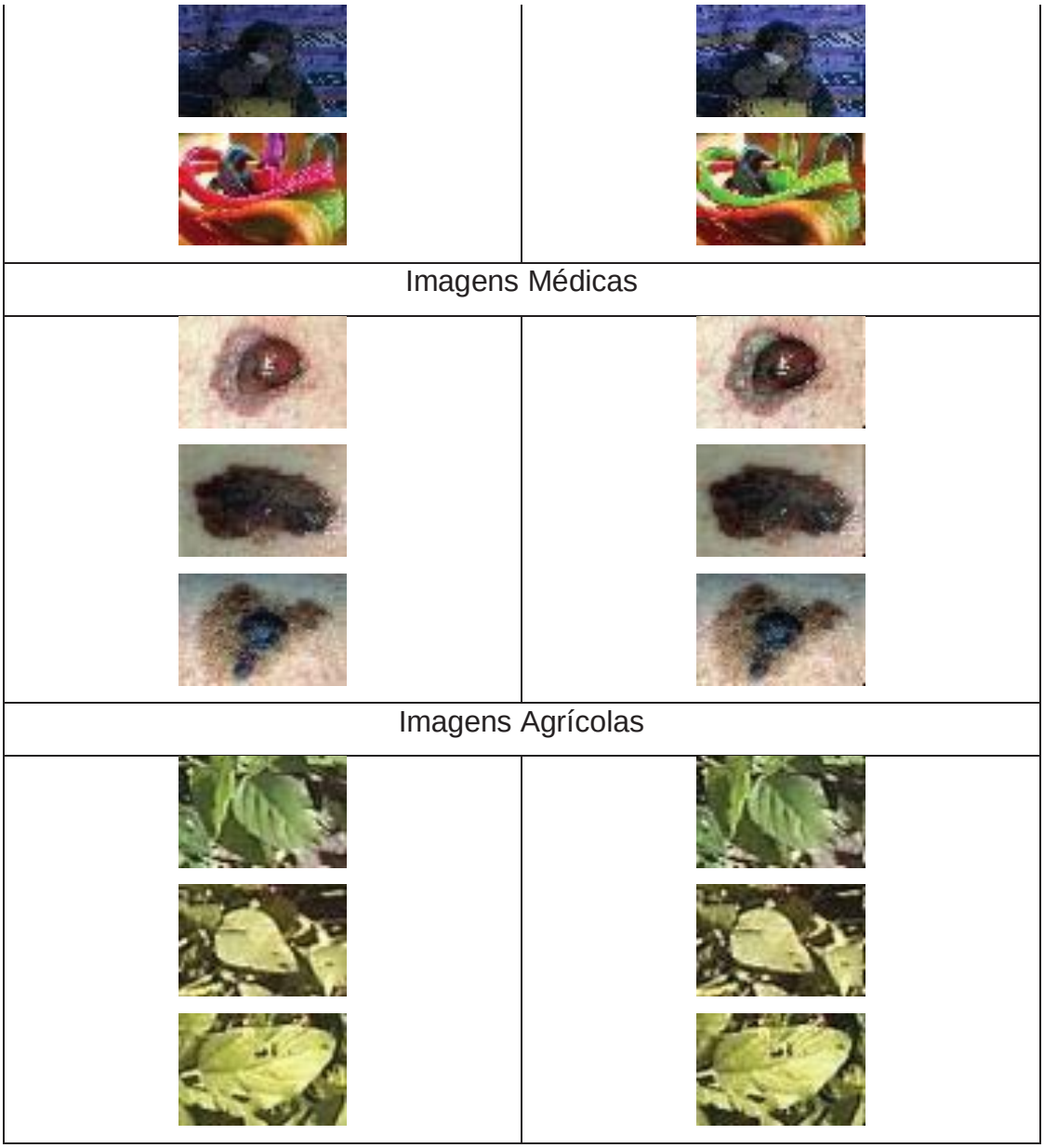


Figura 5.11: Imagens originais (esquerda) e imagens resultante do filtro de normalização de luminância (direita).

A partir da figura 5.11, observa-se que as imagens resultantes condizem com o esperado para o funcionamento do filtro de normalização de luminância. Tal resultado fica evidenciado de forma diferente em cada um dos grupos de imagens. No grupo de imagens comuns, as imagens resultantes apresentam-se mais claras, pois as imagens originais estão um pouco escurecidas. Para as imagens médicas, observou-se o correto ajuste da luminância em cada imagem. Assim, no primeiro caso, onde a imagem original está clara, observou-se que a imagem resultante está mais escura, evidenciando os detalhes da cena. No segundo e terceiro casos, onde a

imagem original está um pouco escurecida, as imagens resultantes apresentam-se levemente mais claras. Por fim, para o grupo de imagens agrícolas, as imagens resultantes apresentam-se sutilmente corrigidas.

5.2 Controlador

O controlador é o responsável por aplicar os filtros de acordo com a programação realizada na memória de programa do sistema, como ilustrado na figura 4.8. Assim, após determinar-se os campos para cada aplicação no arquivo ctrl_data.hex, o qual será carregado na memória de programa, o controlador realizará a leitura desses campos e aplicará os filtros na ordem determinada.

A seguir é feita uma descrição de como preencher os campos do arquivo ctrl_data.hex para que o sistema funcione adequadamente, ana tabela 5.1. Em seguida, um exemplo de arquivo utilizado para testar o controlador, na tabela 5.2.

Endereço Inicial	Endereços sequenciais							
	+00	+01	+02	+03	+04	+05	+06	+07
00	Quantidade de filtros da 1ª. aplicação	1º. filtro	2º. filtro	3º. filtro	4º. filtro	5º. filtro	6º. filtro	Reser-vado.
08	Quantidade de filtros da 2ª. aplicação	1º. filtro	2º. filtro	3º. filtro	4º. filtro	5º. filtro	6º. filtro	Reser-vado.
16	Quantidade de filtros da 3ª. aplicação	1º. filtro	2º. filtro	3º. filtro	4º. filtro	5º. filtro	6º. filtro	Reser-vado.

Tabela 5.1: Descrição de cada campo do arquivo de configuração ctrl_data.hex.

Endereço Inicial	Endereços sequenciais							
	+00	+01	+02	+03	+04	+05	+06	+07
00	06	1	2	3	4	5	6	0
08	03	2	6	3	0	0	0	0
16	04	1	4	5	6	4	0	0
24	02	0	0	0	0	0	0	0
32	00	3	5	0	0	0	0	0
40	02	1	6	0	0	0	0	0

Tabela 5.2: Arquivo ctrl_data.hex contendo, um exemplo das aplicações testadas.

Os filtros foram ordenados de 01 a 06 para facilitar a composição do arquivo de programação. Dessa forma, os filtros numerados de 01 a 06, equivalem respectivamente aos filtros de média, de mediana, de detecção de borda, de equalização de histogramas, de normalização de cores e de normalização de luminância.

Assim, cada linha do arquivo de configuração representa uma aplicação diferente, determinada especificamente para um tipo de imagem, que pode ser uma imagem médica, agrícola ou comum.

Logo, na linha descrita a partir do endereço inicial 00, observa-se uma aplicação com 6 filtros, na ordem crescente, ou seja, o processamento iniciou pelo filtro 01 e terminou no filtro 06. Com essa aplicação simples, testaram-se as conexões internas do controlador e a habilitação de todos os sinais internos de cada filtro seguindo a sequência determinada.

Na linha descrita a partir do endereço inicial 08, observa-se uma aplicação com 3 filtros, iniciando pelo filtro de mediana, seguindo para o filtro de equalização de luminância e finalizando com o filtro de detecção de bordas. Assim como no caso anterior, essa aplicação serviu para verificar as conexões internas do controlador e a habilitação de cada filtro seguindo uma determinada sequência de filtros.

Na linha descrita a partir do endereço inicial 16, observa-se uma aplicação que deve executar apenas 4 filtros, porém foram especificados 5 filtros. Observou-se que a aplicação iniciou com o filtro de média, seguido pelos filtros de equalização de histogramas, de normalização de cores e de normalização de luminância. Sendo que essa aplicação terminou sem executar o filtro 04, demonstrando o seu correto funcionamento com relação ao número de filtros a serem aplicados, independente de quantos filtros foram especificados.

Na linha descrita a partir do endereço inicial 24, verificou-se um caso que possivelmente poderia causar erros, a não especificação do número do filtro a ser aplicado apesar de ter sido indicado que deveriam ser aplicados 2 filtros. Nesse caso, a aplicação termina devido à leitura do valor 00, no campo que deveria conter o número do filtro a ser aplicado. A finalização da aplicação pela leitura do valor 00 no arquivo de programação é um

mecanismo de controle eficiente, o qual evita que a memória de dados seja lida indefinidamente.

Algo similar foi testado na aplicação que se inicia na linha com endereço 32, para a qual não foi determinado o número de filtros a serem aplicados, apesar de ter sido especificada a aplicação dos filtros 03 e 05, nessa ordem. Como no caso anterior, a aplicação terminou pela leitura do valor 00, no campo que deveria conter o número de filtros a serem aplicados.

A última linha, descrita a partir do endereço inicial 40, representa a aplicação que produziu a imagem 5.12, na qual foram aplicados dois filtros, sendo o primeiro o filtro de média, seguido pelo filtro de normalização de luminância.

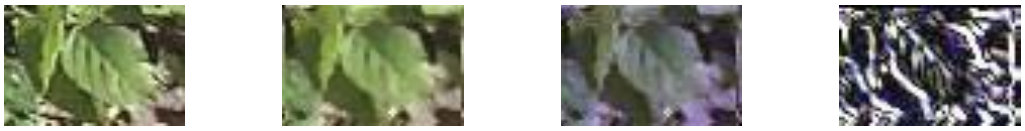
Nas demais aplicações não se levou em consideração a qualidade da imagem resultante, apenas verificou-se o correto funcionamento do sistema, pois a definição do conjunto de filtros mais adequados ao tratamento de cada imagem é muito dependente da aplicação pretendida. Dessa forma, as imagens resultantes das aplicações testadas nem sempre apresentavam resultados interessantes.

A figura 5.12 apresenta o resultado de uma aplicação médica, na qual alguns filtros foram aplicados a uma imagem de melanoma. O objetivo é melhorar a imagem para auxiliar no diagnóstico, com a melhor visualização da extensão e do formato da lesão.

(a) (b) (c)

Figura 5.12: Exemplo de aplicação médica: (a) imagem original, (b) imagem resultante da aplicação do filtro de média e (c) imagem resultante da aplicação do filtro de normalização de luminância.

Na figura 5.13 observa-se a aplicação dos filtros de mediana, de normalização de cores e de detecção de borda a uma imagem agrícola.



(a) (b) (c) (d)

Figura 5.13: Exemplo de aplicação agrícola: (a) imagem original, (b) imagem resultante da aplicação do filtro de mediana, (c) imagem resultante da aplicação do filtro de normalização de cores e (d) imagem resultante do filtro de detecção de bordas.

Por fim, na figura 5.14, observa-se a aplicação dos filtros de mediana, de equalização de histograma e de normalização de luminância em uma imagem colorida.

(a) (b) (c) (d)

Figura 5.14: Exemplo de aplicação comum: (a) imagem original, (b) imagem resultante da aplicação do filtro de mediana, (c) imagem resultante da aplicação do filtro de equalização de histograma e (d) imagem resultante do filtro de normalização de cores.

A partir dos exemplos das aplicações observados nas figuras 5.12, 5.13 e 5.14, conclui-se que a determinação da ordem de execução dos filtros para uma específica aplicação é crucial para o bom processamento da imagem original. Entretanto, uma vez determinada a ordem dos filtros a ser aplicada, a reaplicação dessa sequência torna-se simples com o apoio do sistema desenvolvido.

5.3 Sistema em FPGA

Após a descrição e a validação de cada filtro, com suas memórias auxiliares quando necessário, foi realizada uma síntese preliminar voltada para a placa de FPGA a ser utilizada. Essa síntese foi utilizada para verificar a quantidade de elementos lógicos e a quantidade de *bits* de memória utilizados por cada filtro. Foi ainda detectada a frequência máxima de execução de cada filtro. A tabela 5.3 mostra o resultado da síntese realizada para cada filtro, assim como do sistema completo.

Filtros	**Resultados da Síntese**		
	Número de Elementos Lógicos (% utilizada do total disponível)	*Números de Bits de Memória (% utilizada do total disponível)*	*Frequência Máxima (MHz)*
Média	750 (1%)	0 (0%)	41.97
Mediana	922 (1%)	0 (0%)	113.83
Detector de Borda	499 (<1%)	0 (0%)	118.74
Equalização de Histograma	8 746 (12%)	249 856 (22%)	8.58
Normalização de Cores	8 352 (12%)	258 048 (23%)	8.44
Normalização de Luminância	8 542 (12%)	196 608 (17%)	6
Sistema Completo	30 427 (44%)	903 168 (78%)	10

Tabela 5.3: Resultados da síntese lógica para FPGA.

A partir da tabela 5.3, observa-se que a quantidade de elementos lógicos do sistema completo foi otimizada em relação à quantidade de elementos lógicos em cada filtro. Isso ocorre devido ao uso de técnicas de otimização oferecidas pela própria ferramenta Quartus II v11.1. Através da utilização dessas técnicas, o projetista habilita a ferramenta a implementar certas rotinas que otimizam passos específicos, como, por exemplo, temporização ou roteamento. Com a ajuda dessas técnicas de otimização aplicadas à temporização, associado a um arquivo de restrições devidamente escrito, foi possível alcançar a frequência final de 10 MHz, apesar de alguns filtros apresentarem frequências inferiores a esse valor.

CAPÍTULO 6- Conclusões

6.1 Conclusões

O conjunto de filtros desenvolvido nesta dissertação mostrou-se capaz de processar diferentes tipos de imagens coloridas de forma simples, a partir de uma configuração previamente inserida em um arquivo de controle específico. Assim, o usuário será capaz de escolher quais filtros são mais interessantes para uma imagem, aplicar esses filtros na ordem desejada e analisar o resultado. Caso o resultado não seja o esperado, o sistema pode ser reconfigurado e novos filtros em uma nova ordem podem ser aplicados. Logo, após a determinação da ordem de aplicação dos filtros, o sistema poderá processar grandes conjuntos de imagens com características similares produzindo resultados parecidos para aquele tipo de imagem. Desta forma, por exemplo, o usuário poderá determinar uma ordem específica de filtros para processar imagens médicas e aplicar esse processamento em um grande banco de imagens de maneira automática, sem se preocupar em readequar os filtros a cada nova imagem. Esse tipo de abordagem representa uma eficiente estratégia para processamento em bancos de imagens, pois visa à uniformização do processamento e dos resultados obtidos a partir das características das imagens a serem analisadas.

Ademais, com o intuito de garantir uma boa capacidade de adaptação, o sistema foi implementado utilizando a placa de FPGA DE2-70 da Altera. Atualmente, esse tipo de sistema mostrou-se uma excelente

alternativa de baixo custo e alto desempenho para o desenvolvimento de circuitos integrados digitais.

6.2 Trabalhos Futuros

A partir do conjunto de filtros desenvolvido, observou-se a necessidade de um estudo mais aprofundado para a escolha dos filtros a serem aplicados para um determinado tipo de imagem. Dessa forma, um estudo sobre quais características são mais favorecidas por cada filtro pode introduzir mais confiança no momento de especificar quais filtros e a ordem dos filtros a serem aplicados dependendo do tipo de imagem.

Além disso, a partir do sistema implementado para a placa de desenvolvimento de FPGA DE2-70, observou-se a possibilidade de utilizar a saída VGA para enviar as imagens pré e pós-processamento para um monitor externo. Possibilitando, assim, a visualização da imagem original e da imagem resultante após a aplicação da sequência de filtros especificados, o que beneficiaria algumas aplicações.

Referências Bibliográficas

ACCELLERA, 2001. Accellera Organization Inc., **Universal Verification Methodology (UVM) 1.1 Class Reference**, CA: Napa, 2001.

ACCELLERA, 2004. Accellera Organization Inc., **System verilog 3.1a Reference Manual**, CA: Napa, 2004, pp.144 -182.

BERGERON, J. **Writing Testbenches**. EUA: Kluwer Academic Publishers, 2002.

CHU, PONG P. **RTL Hardware design using VHDL**. EUA: John Wiley & Sons, INC, 2006.

DERMIS, 2012. Dermis. T. L. Diepgen, G. Yihune et al, **Dermatology Information System – DermIS**. Disponível em in Atlas Dermatology Online: http://www.dermis.net/dermisroot/en/home/index.htmL. Acesso em julho de 2012.

DOCUSSE, T. A.; FURLANI, J. R.; ROMANO, R. P.; GUIDO, R. C.; SHI-HUANG CHEN; MARRANGHELLO, N.; PEREIRA, A. S. Microcalcification enhancement and classification on mammograms using the wavelet transform. **International Joint Conference on Neural Networks (IJCNN)**, Hong Kong. p. 3181-3186, 2008.

GONZALEZ, R. C.; WOODS, R. E. **Digital Image Processing**. 2.ed. New Jersey: Prentice Hall, 2002.

GRACI, 2011. **Fotos Coloridas**. Disponível em <http://www.essaseoutras.com.br/lindas-fotos-e-imagens-coloridas-cores-do-arco-iris-e-objetos-flor/>. Acesso em julho de 2012.

JAIN, R.; KASTURI, R.; SCHUNCK, B. G. **Machine Vision**. Cingapura: McGraw-Hill, Inc, 1995.

HABAYE, J. M.; CHANDRAKASAN, A.; NIKOLIC, B. **Digital Integrated Circuits A Design Perspective.** 2.ed. New Jersey: Prentice Hall, 2003.

HAUCK, S.; DEHON, A. **Reconfigurable computing: the theory and practice of FPGA-based computation**. EUA: Morgan Kaufmann Publishers, 2008.

KYUNG, W.; KIM, D.; HA H.; HA, Y. Color enhancement for faded images based on multi-scale gray world algoritm. I**EEE Internation Symposium on Consumer Electronics**, 2012, in press.

LI, H. **Gray World Algorithm**, Disponível em <scien.stanford.edu/pages/labsite/2000/psych221/projects/00/trek/GWimages.html>. Acesso em: 12 jul. 2012.

MARQUES FILHO, O.; VIEIRA NETO, H. **Processamento Digital de Imagens**, Rio de Janeiro: Brasport, 1999.

MENOTTI, G. D. **Realce de Contraste em Imagens Digitais usando Equalização de Histograma**. 2008. 127f. Tese (Doutorado) – Universidade Federal de Minas Gerais, Belo Horizonte, 2008.

MERTES, J. G.; MARRANGHELLO, N.; PEREIRA, A. S. Implementation of Filters for Image Pre-processing for Leaf Analyses in Plantations. **International Conference on Computational Science (ICCS)**, 2008, Kraków. LNCS – Computational Science – ICCS 2008 – Proceedings – Part II. Heidelberg : Springer-Verlag, v. 5102. p. 153-162, 2008.

MUNDEN, R. **Asic and FPGA Verification: a guide to component modeling.** San Francisco: Elsevier, 2005.

ROOSEVELT, L. S. JR., **Extração Automática de Pontos de Apoio para Integração de Imagens Aéreas Digitais e Dados de Perfilamento Laser Aerotransportado**. 2007. 151 p. Dissertação (Mestrado) - Universidade Federal do Paraná, Curitiba , 2007.

SANDEEP, K.; RAJAGOPALAN, A. N. Human Face Detection in Cluttered Color Images Using Skin Color and Edge Information. **Proceedings of the Third Indian Conference on Computer Vision, Graphics & Image**

Processing (ICVGIP), 2002, Ahmadabad, India. Allied Publishers Private Limited 2002.

SANTOS, D. R. **Extração Semi-Automática de Edificações com Análise do Modelo Numérico de Elevações.** 2002. 166p. Dissertação (Mestrado) – Faculdade de Ciências e Tecnologia, UNESP, Presidente Prudente, 2002.

SCURI, A. E. Fundamentos da Imagem Digital. Disponível em: www.tecgraf.puc-rio.br/~scuri/download/fid.pdf. Acesso em: 28 jul. 2012.

SMITH, D. J. **HDL Chip Design**. EUA: Doone Publications, 1ed, 1996.

SONKA, M., HLAVAC, V., BOYLE, R. **Image Processing, Analysis, and Machine Vision**. EUA: PWS Publishing, 2. ed.,1998.

SOON, S. H. **Segmentation.** Disponível em: http://www.ntu.edu.sg/home/ashsseah/cvip.html. Acesso em: 03 jul. 2012.

TERASIC, 2009. Terasic Technologies, **DE2-70 Development and Education Board User Manual**, v1.09, 2009.

.

Apêndice A – *Tasks* e programas

Nesse apêndice estão descritos os modelos de referência utilizando a linguagem SystemVerilog. Assim como os programas desenvolvidos utilizando a linguagem própria do *software* Matlab, ambos mencionados na seção 4.

A.1 – Task ref_mod_mediana

A seguir é descrito o modelo de referência utilizado para verificar o filtro de mediana.

```
1   //declaração de parâmetros iniciais
2   parameter numLinhas = 40;
3   parameter numColunas = 60;
4
5   //declaração de  variáveis e vetores
6   int i, j, a, b, c, k, aux, aux1, aux2;
7   int Vet_outR [2400] , Vet_outG [2400], Vet_outB [2400];
8   int Vet_inR [2400], Vet_inG[2400], Vet_inB[2400];
9   int VetauxR [9], VetauxG [9], VetauxB [9];
10
11  //inicio da task
12  task ref_mod_mediana;
13    begin
14      //leitura de todos os pixels da imagem
15      for(i=0;i<numLinhas;i++)
16        begin
17          for(j=0;j<numColunas;j++)
18            begin
19              //transforma a posição i, j em uma única posição do vetor k
20               k=i*numColunas+j;
21              //aplica a convolução periódica nos extremos da janela
22              if(i==0 || j==0 || i==numLinhas-1 || j==numColunas-1)
23                begin
24                  if ((i==0)&(j==0))
25                    begin
26                      aux1=(numColunas*numLinhas)-1;
27                    Vet_outR[k]= Vet_inR[aux1];
28                     Vet_outG[k]= Vet_inG[aux1];
29                     Vet_outB[k]= Vet_inB[aux1];
30                    end
31                 else
32                    if (i==0)
33                      begin
34                        aux1=(a * numColunas) + b +(numColunas-1);
```

```
35                          Vet_outR[k]= Vet_inR[aux1];
36                          Vet_outG[k]= Vet_inG[aux1];
37                          Vet_outB[k]= Vet_inB[aux1];
38                        end
39                      else
40                        if (j==0)
41                          begin
42                            aux1 = (a * numColunas) + b + (numLinhas-1);
43                            Vet_outR[k]= Vet_inR[aux1];
44                            Vet_outG[k]= Vet_inG[aux1];
45                            Vet_outB[k]= Vet_inB[aux1];
46                          end
47                        else
48                          if ((i==numLinhas-1)&(j==numColunas-1))
49                            begin
50                              aux1 = 0;
51                              Vet_outR[k]=Vet_inR[aux1];
52                              Vet_outG[k]=Vet_inG[aux1];
53                              Vet_outB[k]=Vet_inB[aux1];
54                            end
55                          else
56                            if (i==numLinhas-1 )
57                              begin
58                                aux1 = (a * numColunas) + b - (numColunas-1);
59                                Vet_outR[k]= Vet_inR[aux1];
60                                Vet_outG[k]= Vet_inG[aux1];
61                                Vet_outB[k]= Vet_inB[aux1];
62                              end
63                            else
64                              if (j==numColunas-1)
65                                begin
66                                  aux1 = (a * numColunas) + b - (numLinhas-1);
67                                   Vet_outR[k]= Vet_inR[aux1];
68                                  Vet_outG[k]= Vet_inG[aux1];
69                                  Vet_outB[k]= Vet_inB[aux1];
70                                end
71                  end
72                else
73                  begin
74                    //leitura dos pixels da janela 3x3
75                    for(c=0, a=(i-1); a<=(i+1); a++)
76                      begin
77                        for(b=(j-1); b<=(j+1); b++, c++)
78                          begin
79                            aux1=a*numColunas+b;
80                            VetauxR[c] = Vet_inR[aux1];
81                            VetauxG[c] = Vet_inG[aux1];
82                            VetauxB[c] = Vet_inB[aux1];
83                          end
84                      end
85                    //Ordena o vetor (METODO: BOLHA)
86                    for (b=0;b<9;b++)
87                      begin
88                        for(a=8;a>b;a--)
89                          begin
90                            c=a-1;
91                            aux=VetauxR[c];
92                            if(VetauxR[c]>VetauxR[a])
93                              begin
94                                aux2=VetauxR[a];
95                                VetauxR[a]=VetauxR[c];
96                                VetauxR[c]=aux2;
```

```
97                                  end
98                                if(VetauxG[c]>VetauxG[a])
99                                  begin
100                                   aux2=VetauxG[a];
101                                   VetauxG[a]=VetauxG[c];
102                                   VetauxG[c]=aux2;
103                                 end
104                               if(VetauxB[c]>VetauxB[a])
105                                 begin
106                                   aux2=VetauxB[a];
107                                  VetauxB[a]=VetauxB[c];
108                                   VetauxB[c]=aux2;
109                                end
110                         end
111                       end
112                     //substitui o pixel pelo valor mediano
113                     Vet_outR[k] = VetauxR[4];
114                     Vet_outG[k] = VetauxG[4];
115                     Vet_outB[k] = VetauxB[4];
116               end
117             end
118           end
119       end
120   endtask
```

A.2 – Task ref_mod_media

A seguir é descrito o modelo de referência utilizado para verificar o filtro de media.

```
1    //declaração de parâmetros iniciais
2    parameter numLinhas = 40;
3    parameter numColunas = 60;
4
5    //declaração de  variáveis e vetores
6    int i, j, a, b, c, k, add_aux, aux1;
7    int Vet_outR [2400] , Vet_outG [2400], Vet_outB [2400];
8    int Vet_inR [2400], Vet_inG[2400], Vet_inB[2400];
9    int sumR, sumG, sumB;
10
11   //inicio da task
12   task ref_mod_media;
13     begin
14       //leitura de todos os pixels da imagem
15       for(i=0;i<numLinhas;i++)
16         begin
17           for(j=0;j<numColunas;j++)
18             begin
19               //transforma a posição i, j em uma única posição do vetor k
20                k=i*numColunas+j;
21               //aplica a convolução periódica nos extremos da janela
22               if(i==0 || j==0 || i==numLinhas-1 || j==numColunas-1)
23                 begin
24                   if ((i==0)&(j==0))
25                     begin
26                       aux1=(numColunas*numLinhas)-1;
27                      Vet_outR[k]= Vet_inR[aux1];
28                       Vet_outG[k]= Vet_inG[aux1];
29                       Vet_outB[k]= Vet_inB[aux1];
```

```
30                      end
31                  else
32                      if (i==0)
33                        begin
34                          aux1=(a * numColunas) + b +(numColunas-1);
35                          Vet_outR[k]= Vet_inR[aux1];
36                          Vet_outG[k]= Vet_inG[aux1];
37                          Vet_outB[k]= Vet_inB[aux1];
38                        end
39                      else
40                        if (j==0)
41                          begin
42                            aux1 = (a * numColunas) + b + (numLinhas-1);
43                            Vet_outR[k]= Vet_inR[aux1];
44                            Vet_outG[k]= Vet_inG[aux1];
45                            Vet_outB[k]= Vet_inB[aux1];
46                          end
47                        else
48                          if ((i==numLinhas-1)&(j==numColunas-1))
49                            begin
50                              aux1 = 0;
51                              Vet_outR[k]=Vet_inR[aux1];
52                              Vet_outG[k]=Vet_inG[aux1];
53                              Vet_outB[k]=Vet_inB[aux1];
54                            end
55                          else
56                            if (i==numLinhas-1 )
57                              begin
58                                aux1 = (a * numColunas) + b - (numColunas-1);
59                                Vet_outR[k]= Vet_inR[aux1];
60                                Vet_outG[k]= Vet_inG[aux1];
61                                Vet_outB[k]= Vet_inB[aux1];
62                              end
63                            else
64                              if (j==numColunas-1)
65                                begin
66                                  aux1 = (a * numColunas) + b - (numLinhas-1);
67                                  Vet_outR[k]= Vet_inR[aux1];
68                                  Vet_outG[k]= Vet_inG[aux1];
69                                  Vet_outB[k]= Vet_inB[aux1];
70                                end
71                  end
72              else
73                  begin
74                    //leitura dos pixels da janela 3x3
75                    //Realiza a soma dos pixels da janela
76                    sumR = 0;   sumG = 0;   sumB = 0;
77                    for(c=0, a=(i-1); a<=(i+1); a++)
78                      begin
79                        for(b=(j-1); b<=(j+1); b++, c++)
80                          begin
81                            if(i==0 || j==0 || i==numLinhas-1 || j==numColunas-1)
82                              begin
83                                sumR = sumR + 0;
84                                sumG = sumG + 0;
85                                sumB = sumB + 0;
86                                add_aux=0;
87                              end
88                            else
89                              begin
90                                add_aux=a*numColunas+b;
91                                sumR = sumR + Vet_inR[add_aux];
```

```
92                                  sumG = sumG + Vet_inG[add_aux];
93                                 sumB = sumB + Vet_inB[add_aux];
94                                end
95                            end
96                          end
97                          //realiza a média dos valores somados
98                          medR = sumR/9;
99                         medG = sumG/9;
100                        medB = sumB/9;
101                       //substitui o pixel pelo valor médio
102                      Vet_outR[k] = medR;
103                       Vet_outG[k] = medG;
104                       Vet_outB[k] = medB;
105               end
106             end
107           end
108         end
109       endtask
```

A.3 – Programa vetor2imagem

A seguir é descrito o programa *vetor2imagem* utilizado para visualizar as imagens descritas a partir de arquivos textos.

```
%programa para converter arquivos de texto em imagem RGB
clear;
I = imread('margarida.jpg');
x=size(I,1);
y=size(I,2);

%abertura dos arquivos contendo os vetores RGB
fidr = fopen('mem_r_margarida.txt', 'r');
fidg = fopen('mem_g_margarida.txt', 'r');
fidb = fopen('mem_b_margarida.txt', 'r');

%leitura dos valores dos arquivos e armazenando em vetores
for m=1:x*y
   R(m)=fscanf(fidr, '%08x', 1);
   G(m)=fscanf(fidg, '%08x', 1);
   B(m)=fscanf(fidb, '%08x', 1);
end

%montando a imagem a partir dos valores armazenados nos vetores
pos=0;
for i=1:40
    for j=1:60
        if i==1
            pos=j;
        else
            pos=((i-1)*60) + j;
        end
        I1(i,j,1)=uint8((R(pos)));
        I1(i,j,2)=uint8((G(pos)));
        I1(i,j,3)=uint8((B(pos)));
    end
end
imshow(I);
figure, imshow(I1);
%gravando a imagem resultante
```

```
imwrite(I1, 'margarida_resultante.jpg');
%fechando os arquivos iniciais
fclose(fidr);
fclose(fidg);
fclose(fidb);
```

A.4 - Programa imagem2vetor

A seguir é descrito o programa *imagem2vetor* utilizado para converter imagens em arquivos de texto.

```
%programa para converter imagem RGB em arquivos de texto
clear;
I = imread('margarida.jpg');
x=size(I,1);
y=size(I,2);

%lendo a imagem e armazenando os valores em vetores
for i=1:x
    for j=1:y
        if i==1
            pos=j;
        else
            pos=((i-1)*60) + j;
        end
        R(pos) = uint8(I(i,j,1));
        G(pos) = uint8(I(i,j,2));
        B(pos) = uint8(I(i,j,3));
    end
end

%abertura dos arquivos
fidr = fopen('margarida_r.txt', 'w');
fidg = fopen('margarida_g.txt', 'w');
fidb = fopen('margarida_b.txt', 'w');

%armazenando os valores dos vetores RGB nos arquivos
for i=1:(x*y)
    fprintf(fidr, '%08x\n', uint8(R(i)));
    fprintf(fidg, '%08x\n', uint8(G(i)));
    fprintf(fidb, '%08x\n', uint8(B(i)));
end

%fechando os arquivos
fclose(fidr);
fclose(fidg);
fclose(fidb);
imshow(I);
```

A.5 – Task ref_mod_borda

A seguir é descrito o modelo de referência utilizado para verificar o filtro de detecção de borda, o qual utiliza o algoritmo Sobel.

```
1          //declaração de parâmetros iniciais
```

```
2       parameter numLinhas = 6'd40;
3       parameter numColunas = 6'd60;
4
5       //declaração de variáveis e vetores
6       int i, j, a, b, c, k, add_aux, aux1;
7       int Vet_outR [2400], Vet_outG [2400], Vet_outB [2400];
8       int Vet_inR [2400], Vet_inG[2400], Vet_inB[2400];
9       int sumR_h1, sumG_h1, sumB_h1, sumR_h2, sumG_h2, sumB_h2, resR, resG, resB;
10      int h1[9], h2[9];
11      int Vet_auxR [9], Vet_auxG[9], Vet_auxB[9];
12
13       //inicio da task
14      task ref_mod_borda;
15        begin
16          //atribuição dos valores das mascaras
17           h1[0]=(-1); h1[1]=(0); h1[2]=(1);
18           h1[3]=(-2); h1[4]=(0); h1[5]=(2);
19           h1[6]=(-1); h1[7]=(0); h1[8]=(1);
20
21           h2[0]=(1);  h2[1]=(2);  h2[2]=(1);
22           h2[3]=(0);  h2[4]=(0);  h2[5]=(0);
23           h2[6]=(-1); h2[7]=(-2); h2[8]=(-1);
24          //leitura de todos os pixels da imagem
25         for(i=0;i<numLinhas;i++)
26            begin
27                for(j=0;j<numColunas;j++)
28                 begin
29                   //transforma a posicao i,j em uma unica posicao do vetor k
30                     k=i*numColunas+j;
31                   //coloca pixels brancos nos extremos da janela
32                  //aplica a convolucao periodica nos extremos da janela
33                  if(i==0 || j==0 || i==numLinhas-1 || j==numColunas-1)
34                     begin
35                       if ((i==0)&(j==0))
36                         begin
37                           aux1=(numColunas*numLinhas)-1;
38                           Vet_outR[k]= Vet_inR[aux1];
39                           Vet_outG[k]= Vet_inG[aux1];
40                           Vet_outB[k]= Vet_inB[aux1];
41                        end
42                       else
43                         if (i==0)
44                           begin
45                             aux1=(a * numColunas) + b +(numColunas-1);
46                             Vet_outR[k]= Vet_inR[aux1];
47                             Vet_outG[k]= Vet_inG[aux1];
48                             Vet_outB[k]= Vet_inB[aux1];
49                          end
50                         else
51                           if (j==0)
52                             begin
53                               aux1 = (a * numColunas) + b +(numLinhas-1);
54                               Vet_outR[k]= Vet_inR[aux1];
55                               Vet_outG[k]= Vet_inG[aux1];
56                               Vet_outB[k]= Vet_inB[aux1];
57                            end
58                           else
59                             if ((i==numLinhas-1 )&(j==numColunas-1))
60                               begin
61                                 aux1 = 0;
62                                 Vet_outR[k]=Vet_inR[aux1];
63                                 Vet_outG[k]=Vet_inG[aux1];
```

```
64                              Vet_outB[k]=Vet_inB[aux1];
65                            end
66                          else
67                            if (i==numLinhas-1 )
68                              begin
69                                aux1 = (a * numColunas) + b-(numColunas-1);
70                                Vet_outR[k]= Vet_inR[aux1];
71                                Vet_outG[k]= Vet_inG[aux1];
72                                Vet_outB[k]= Vet_inB[aux1];
73                              end
74                            else
75                              if (j==numColunas-1)
76                                begin
77                                  aux1 = (a * numColunas) + b-(numLinhas-1);
78                                  Vet_outR[k]= Vet_inR[aux1];
79                                  Vet_outG[k]= Vet_inG[aux1];
80                                 Vet_outB[k]= Vet_inB[aux1];
81                                end
82                  end
83                else
84                  begin
85                    //leitura dos pixels da janela 3x3
86                    //Realiza a soma dos pixels da janela
87                    sumR_h1 = 0;   sumG_h1 = 0;   sumB_h1 = 0;
88                    sumR_h2 = 0;   sumG_h2 = 0;   sumB_h2 = 0;
89                    for(c=0, a=(i-1); a<=(i+1); a++)
90                      begin
91                        for(b=(j-1); b<=(j+1); b++, c++)
92                          begin
93                            add_aux=a*numColunas+b;
94                            //convolucao com h1
95                            sumR_h1 = sumR_h1 + (h1[c]*Vet_inR[add_aux]);
96                            sumG_h1 = sumG_h1 + (h1[c]*Vet_inG[add_aux]);
97                            sumB_h1 = sumB_h1 + (h1[c]*Vet_inB[add_aux]);
98                            //convolucao com h1
99                            sumR_h2 = sumR_h2 + (h2[c]*Vet_inR[add_aux]);
100                           sumG_h2 = sumG_h2 + (h2[c]*Vet_inG[add_aux]);
101                           sumB_h2 = sumB_h2 + (h2[c]*Vet_inB[add_aux]);
102                           //armazenamento em vetor auxiliar
103                           Vet_auxR[c] = Vet_inR[add_aux];
104                          Vet_auxG[c] = Vet_inG[add_aux];
105                          Vet_auxB[c] = Vet_inB[add_aux];
106                       end
107                    end
108                    //realiza a soma final dos valores convolucionados
109                    resR = sumR_h1 + sumR_h2;
110                    resG = sumG_h1 + sumG_h2;
111                    resB = sumB_h1 + sumB_h2;
112                   //ajuste de valores negativos para valores em complemento de 2
113                   if (resR< 0)
114                         resR = (~(resR))+1;
115                   if (resG< 0)
116                         resG = (~(resG))+1;
117                    if (resB< 0)
118                         resB = (~(resB))+1;
119                    //substitui o pixel pelo valor final normalizado
120                    if (resR > 255)
121                         Vet_outR[k] =  255;
122                    else
123                         Vet_outR[k] = resR;
124                     if (resG > 255)
125                         Vet_outG[k] =  255;
```

```
126                     else
127                         Vet_outG[k] = resG;
128                     if (resB > 255)
129                       Vet_outB[k] =  255;
130                     else
131                         Vet_outB[k] = resB;
132             end
133           end
134         end
135       end
136     endtask
```

A.6 – Task ref_mod_histograma

A seguir é descrito o modelo de referência utilizado para verificar o filtro de equalização de histograma.

```
1       //declaração de parâmetros iniciais
2       parameter numLinhas = 6'd40;
3       parameter numColunas = 6'd60;
4       parameter n_levels = 360;
5
6       //declaração de variáveis e vetores
7       int i, r, g, b, h, s, v, min, max, delta;
8       int Vet_outR [2400], Vet_outG [2400], Vet_outB [2400];
9       int Vet_inR [2400], Vet_inG[2400], Vet_inB[2400];
10      int Vet_H [2400], Vet_S[2400], Vet_V[2400];
11      int Vet_Heq [2400];
12      int hist_origh[361], Yh[361];
13      int pixh, pixs, pixv;
14      int hi, f, p, q, t1, t2, t3, t, hf, hl;
15
16      //inicio da task
17      task ref_mod_histograma;
18        begin
19          //transformação rgb para hsv
20          for(i=0;i<numLinhas*numColunas;i++)
21            begin
22              r = Vet_inR[i];
23              g = Vet_inG[i];
24              b = Vet_inB[i];
25              min = r;
26              if(g<min)
27                min = g;
28              if(b<min)
29                min = b;
30              max = r;
31              if(g>max)
32                max = g;
33              if(b>max)
34                max = b;
35              v = max;
36               if (max>0)
37                 begin
38                   delta = max - min;
39                   s =((delta*255)/max);
40                   if(r==max)
41                     if (g>=b)
```

```
42                      h = ((((60*(g-b))/delta)));
44                    else
45                      h = ((((60*(g-b))/delta))+360);
46                  else
47                    if(g==max)
48                      h = ((((60*(b-r))/delta))+120);
49                    else
50                      h = ((((60*(r-g))/delta))+240);
51                end
52              else
53                begin
54                  delta = 0;
55                  s = 0;
56                  h = 0;
57                end
58              Vet_H[i] = h;
59              Vet_S[i] = s;
60              Vet_V[i] = v;
61            end
62
63          //inicio da aplicação da equalização do histograma
64
65          //inicialização do histograma original
66          for (i=0;i<361;i++)
67            begin
68              hist_origh[i] = 0;
69            end
70          //soma dos valores do histograma original
71          for (i=0;i<numLinhas*numColunas;i++)
72            begin
73              pixh = Vet_H[i];
74              hist_origh[pixh] = (hist_origh[pixh]+1);
75            end
76          //criação do vetor auxiliar Y
77          for (i=0;i<361;i++)
78            begin
79              if (i==0)
80                Yh[i]=hist_origh[i];
81              else
82                Yh[i]=(hist_origh[i]+Yh[i-1]);
83            end
84          //equalização da component H
85          for (i=0;i<numLinhas*numColunas;i++)
86            begin
87              pixh = Vet_H[i];
88              Vet_Heq[i] = ((n_levels*Yh[pixh])/(numLinhas*numColunas));
89            end
90
91          //transformação hsv para rgb
92          for (i=0;i<numLinhas*numColunas;i++)
93            begin
94              h = Vet_Heq[i];
95              s = Vet_S[i];
96              v = Vet_V[i];
97              if(s==0)
98                begin
99                  r = v;
100                 g = v;
101                 b = v;
102               end
103             else
104               begin
```

```
105                 hi=(h/60);
106                 f = (((h*1000)/60)-(hi*1000));
107                 p = ( (v * (255-s)) / 255);
108                 q = ( (v * (255000 - (s*f))) / 255000);
109                 t = (v*(255000-s*(1000-f)))/255000;
110                 if(hi==0)
111                   begin
112                     r = v;
113                     g = t;
114                     b = p;
115                   end
116                 else
117                   if (hi==1)
118                     begin
119                       r = q;
120                       g = v;
121                       b = p;
122                     end
123                   else
124                     if (hi==2)
125                       begin
126                         r = p;
127                         g = v;
128                         b = t;
129                       end
130                     else
131                       if (hi==3)
132                         begin
133                           r = p;
134                           g = q;
135                           b = v;
136                         end
137                       else
138                         if (hi==4)
139                           begin
140                             r = t;
141                             g = p;
142                             b = v;
143                           end
144                         else
145                           begin
146                             r = v;
147                             g = p;
148                             b = q;
149                           end
150               end
151             //armazenamento vetor final
152             Vet_outR[i] = r;
153             Vet_outG[i] = g;
154             Vet_outB[i] = b;
155         end
156      end
157     endtask
```

A.7 – Task ref_mod_norm_cores

A seguir é descrito o modelo de referência utilizado para verificar o filtro de normalização de cores.

```
1         //declaração de parâmetros iniciais
2         parameter numLinhas = 6'd40;
3         parameter numColunas = 6'd60;
4
5         //declaração de variáveis e vetores
6         int Vet_outR [2400], Vet_outG [2400], Vet_outB [2400];
7         int Vet_inR [2400], Vet_inG[2400], Vet_inB[2400];
8         int Vet_inR2 [2400], Vet_inG2[2400], Vet_inB2[2400];
9         int Vet_outRi [2400], Vet_outGi[2400], Vet_outBi[2400];
10        int sumR, sumG, sumB;
11        int medR, medG, medB, medGray;
12        int maxR, maxG, maxB, maxRGB;
13        int i, pos;
14
15        //inicio da task
16        task ref_mod_norm_cores;
17          begin
18            //normalização da imagem inicial
19            pos = 0;
20            sumR = 0;   sumG = 0;      sumB = 0;
21            for(i=0;i<numLinhas*numColunas;i++)
22              begin
23                  Vet_inR2[i] = ((Vet_inR[i]*1000)/256);
24                  Vet_inG2[i] = ((Vet_inG[i]*1000)/256);
25                  Vet_inB2[i] = ((Vet_inB[i]*1000)/256);
26                  sumR = sumR + Vet_inR2[i];
27                  sumG = sumG + Vet_inG2[i];
28                  sumB = sumB + Vet_inB2[i];
29                  pos = pos +1;
30              end
31
32            //determinação dos valores médios
33            medR = (sumR/pos);
34            medG = (sumG/pos);
35            medB = (sumB/pos);
36            medGray = ((medR+medG+medB)/3);
37
38            // normalização GrayWorld
39            for(i=0;i<numLinhas*numColunas;i++)
40              begin
41                if (medR == 0)
42                  Vet_outRi[i] = Vet_inR2[i];
43                else
44                  Vet_outRi[i] = ((medGray*Vet_inR2[i])/medR);
45                if (medG == 0)
46                  Vet_outGi[i] = Vet_inG2[i];
47                else
48                  Vet_outGi[i] = ((medGray*Vet_inG2[i])/medG);
49                if (medB == 0)
50                  Vet_outBi[i] = Vet_inB2[i];
51                else
52                  Vet_outBi[i] = ((medGray*Vet_inB2[i])/medB);
53              end
54
55            // determinação do máximo de cada vetor RGB
56            maxR = 0;
57            maxG = 0;
58            maxB = 0;
59            for (i=0;i<numLinhas*numColunas;i++)
60              begin
61                if (Vet_outRi[i]>maxR)
62                  maxR=Vet_outRi[i];
```

```
63              if (Vet_outGi[i]>maxG)
64                maxG=Vet_outGi[i];
65              if (Vet_outBi[i]>maxB)
66                maxB=Vet_outBi[i];
67            end
68
69          //determinação máximo RGB
70          maxRGB = 0;
71          if (maxR > maxG)
72            if (maxR > maxB)
73              maxRGB = maxR;
74            else
75              maxRGB = maxB;
76          else
77            if (maxG > maxB)
78              maxRGB = maxG;
79            else
80              maxRGB = maxB;
81
82         // ajuste da escala da imagem normalizada
83        for (i=0;i<numLinhas*numColunas;i++)
84          begin
85            if  (maxRGB > 1000)
86              begin
87                Vet_outR[i] = ((Vet_outRi[i]*255)/maxRGB);
88                Vet_outG[i] = ((Vet_outGi[i]*255)/maxRGB);
89                Vet_outB[i] = ((Vet_outBi[i]*255)/maxRGB);
90              end
91            else
92              begin
93                Vet_outR[i] = (Vet_outRi[i]*255);
94                Vet_outG[i] = (Vet_outGi[i]*255);
95                Vet_outB[i] = (Vet_outBi[i]*255);
96              end
97          end
98        end
99      endtask
```

A.8 – Task ref_mod_norm_luminancia

A seguir é descrito o modelo de referência utilizado para verificar o filtro de normalização de luminância.

```
1       //declaração de parâmetros iniciais
2       parameter numLinhas = 6'd40;
3       parameter numColunas = 6'd60;
4       parameter n_levels = 360;
5
6       //declaração de variáveis e vetores
7       int i, r, g, b, h, s, v, min, max, delta;
8       int Vet_outR [2400], Vet_outG [2400], Vet_outB [2400];
9       int Vet_inR [2400], Vet_inG[2400], Vet_inB[2400];
10      int Vet_H [2400], Vet_S[2400], Vet_V[2400];
11      int Vet_Vi [2400];
12      int hist_origh[361], Yh[361];
13      int maxV, minV, pixV;
14      int hi, f, p, q, t1, t2, t3, t, hf, hl;
15
16      //inicio da task
```

```
17      task ref_mod_norm_luminancia;
18        begin
19          //transformação rgb para hsv
20          for(i=0;i<numLinhas*numColunas;i++)
21            begin
22              r = Vet_inR[i];
23              g = Vet_inG[i];
24              b = Vet_inB[i];
25              min = r;
26              if(g<min)
27                min = g;
28              if(b<min)
29                min = b;
30              max = r;
31              if(g>max)
32                max = g;
33              if(b>max)
34                max = b;
35              v = max;
36               if (max>0)
37                 begin
38                   delta = max - min;
39                   s =((delta*255)/max);
40                   if(r==max)
41                     if (g>=b)
42                       h = ((((60*(g-b))/delta)));
44                     else
45                       h = ((((60*(g-b))/delta))+360);
46                   else
47                     if(g==max)
48                       h = ((((60*(b-r))/delta))+120);
49                     else
50                       h = ((((60*(r-g))/delta))+240);
51                 end
52               else
53                 begin
54                   delta = 0;
55                   s = 0;
56                   h = 0;
57                 end
58               Vet_H[i] = h;
59               Vet_S[i] = s;
60               Vet_V[i] = v;
61            end
62
63          //início do processo de normalização da luminância
64          //buscar valor maximo e minimo em V
65          maxV=0;
66          minV=255;
67          for (i=0;i<numLinhas*numColunas;i++)
68            begin
69              pixV = Vet_V[i];
70              if (pixV > maxV)
71                maxV = pixV;
72              if (pixV < minV)
73                minV = pixV;
74            end
75
76          // normalizar V
77          for (i=0;i<numLinhas*numColunas;i++)
78            begin
79              pixV = Vet_V[i];
```

```
80              Vet_Vi[i] = (((Vet_V[i] - minV)*255)/(maxV - minV));
81            end
82
83            //transformação hsv para rgb
84            for (i=0;i<numLinhas*numColunas;i++)
85              begin
86                h = Vet_Heq[i];
87                s = Vet_S[i];
88                v = Vet_V[i];
89                if(s==0)
90                  begin
91                    r = v;
92                    g = v;
93                    b = v;
94                  end
95                else
96                  begin
97                    hi=(h/60);
98                    f = (((h*1000)/60)-(hi*1000));
99                    p = ( (v * (255-s)) / 255);
100                   q = ( (v * (255000 - (s*f))) / 255000);
101                   t = (v*(255000-s*(1000-f)))/255000;
102                   if(hi==0)
103                     begin
104                       r = v;
105                       g = t;
106                       b = p;
107                     end
108                   else
109                     if (hi==1)
110                       begin
111                         r = q;
112                         g = v;
113                         b = p;
114                       end
115                     else
116                       if (hi==2)
117                         begin
118                           r = p;
119                           g = v;
120                           b = t;
121                         end
122                       else
123                         if (hi==3)
124                           begin
125                             r = p;
126                             g = q;
127                             b = v;
128                           end
129                         else
130                           if (hi==4)
131                             begin
132                               r = t;
133                               g = p;
134                               b = v;
135                             end
136                           else
137                             begin
138                               r = v;
139                               g = p;
140                               b = q;
141                             end
```

```
142                 end
143               //armazenamento da imagem final
144               Vet_outR[i] = r;
145               Vet_outG[i] = g;
146               Vet_outB[i] = b;
147           end
148       end
149       endtask
```

A.9 – Programa coloca_ruido

A seguir é descrito o programa *coloca_ruido* utilizado para inserir ruídos do tipo *salt and pepper* em imagens coloridas.

```
%programa para inserir ruidos salt & pepper em imagens coloridas
clear;
I = imread('cadarcos.jpg');
x=size(I,1);
y=size(I,2);

%separação dos vetores RGB
for i=1:x
    for j=1:y
        Ir(i,j)=I(i,j,1);
        Ig(i,j)=I(i,j,2);
        Ib(i,j)=I(i,j,3);
    end
end

%inserção de ruído do tipo salt & pepper com granulosidade 0.1
I2r=imnoise(Ir, 'salt & pepper', 0.1);
I2g=imnoise(Ig, 'salt & pepper', 0.1);
I2b=imnoise(Ib, 'salt & pepper', 0.1);

%construção da imagem ruidosa
for i=1:x
    for j=1:y
        I2(i,j,1)=I2r(i,j);
        I2(i,j,2)=I2g(i,j);
        I2(i,j,3)=I2b(i,j);
    end
end

imshow(I);
figure, imshow(I2);
imwrite(I2, 'cadarcos_ruido.jpg');
```

A.10 - Programa conv_rgb_hsv_hist

A seguir é descrito o programa *conv_rgb_hsv_hist* utilizado para ajustar o histograma da componente H de imagens descritas no modelo HSV.

```
%programa para converter imagem de rgb para hsv e modificar o
histograma
```

```
clear;
I = imread('margarida.jpg');
x = size(I,1);
y = size(I,2);
%conversão da imagem original de RGB para HSV
Ihsv=rgb2hsv(I);
xh = size(I,1);
yh = size(I,2);

%separação dos vetores HSV
for i=1:xh
    for j=1:yh
        Ih(i,j)=Ihsv(i,j,1);
        Is(i,j)=Ihsv(i,j,2);
        Iv(i,j)=Ihsv(i,j,3);
    end
end

%clareia a imagem somente sobre o canal H (hue ou matiz)
I2hc= imadjust(Ih,[],[0.5 1.0]);

%escurece a imagem somente sobre o canal H (hue ou matiz)
I2he= imadjust(Ih,[],[0.0 0.5]);

%centraliza o histograma somente sobre o canal H (hue ou matiz)
I2hm= imadjust(Ih,[],[0.3 0.7]);

I2hsvc=Ihsv;
I2hsve=Ihsv;
I2hsvm=Ihsv;

%montagem das imagens resultantes
for i=1:x
    for j=1:y
        I2hsvc(i,j,1)=I2hc(i,j);
        I2hsve(i,j,1)=I2he(i,j);
        I2hsvm(i,j,1)=I2hm(i,j);
    end
end

%conversão das imagens ajustadas de HSV para RGB
Irgbc=hsv2rgb(I2hsvc);
Irgbe=hsv2rgb(I2hsve);
Irgbm=hsv2rgb(I2hsvm);

imshow(I);
figure,imshow(Irgbc);
figure,imshow(Irgbe);
figure,imshow(Irgbm);

imwrite(Irgbc, 'margarida_clara.jpg');
imwrite(Irgbe, 'margarida_escura.jpg');
imwrite(Irgbm, 'margarida_mid.jpg');
```

A.11 - Programa ajuste_cor_imagem

A seguir é descrito o programa *ajuste_cor_imagem* utilizado para ajustar o cada componente RGB, tornando a imagem mais avermelhadas, mais azuladas ou mais esverdeadas.

```
%programa para modificar imagens alterando os vetores RGB
clear;
I = imread('margarida.jpg');
x=size(I,1);
y=size(I,2);

%separação dos vetores RGB
for i=1:x
    for j=1:y
        Ir(i,j)=I(i,j,1);
        Ig(i,j)=I(i,j,2);
        Ib(i,j)=I(i,j,3);
    end
end

%incremento dos vetores RGB em 100 unidades
for i=1:x
    for j=1:y
        I2r(i,j)=Ir(i,j)+uint8(100);
        I2g(i,j)=Ig(i,j)+uint8(100);
        I2b(i,j)=Ib(i,j)+uint8(100);
    end
end

%armazenamento dos vetores incrementados
for i=1:x
    for j=1:y
        I2rgbr(i,j,1)=I2r(i,j);
        I2rgbr(i,j,2)=Ig(i,j);
        I2rgbr(i,j,3)=Ib(i,j);

        I2rgbg(i,j,1)=Ir(i,j);
        I2rgbg(i,j,2)=I2g(i,j);
        I2rgbg(i,j,3)=Ib(i,j);

        I2rgbb(i,j,1)=Ir(i,j);
        I2rgbb(i,j,2)=Ig(i,j);
        I2rgbb(i,j,3)=I2b(i,j);
    end
end

imshow(I);
figure, imshow(I2rgbr);
figure, imshow(I2rgbg);
figure, imshow(I2rgbb);
imwrite(I2rgbr, 'margarida_red.jpg');
imwrite(I2rgbg, 'margarida_green.jpg');
imwrite(I2rgbb, 'margarida_blue.jpg');
```

Autorizo a reprodução xerográfica para fins de pesquisa.

São José do Rio Preto, 11 de janeiro de 2013.

_____

Assinatura