UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO MESQUITA FILHO”

FACULDADE DE ENGENHARIA DE ILHA SOLTEIRA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA – DEE
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ALGORITMOS DE DETECÇÃO DE BORDAS IMPLEMENTADOS EM FPGA

PATRÍCIA SALLES MATURANA

Orientador: Eng. Eletr. Alexandre César Rodrigues da Silva

Ilha Solteira – SP

Novembro de 2010

unesp
UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Ilha Solteira

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

"Algoritmos de detecção de bordas implementados em FPGA"

PATRÍCIA SALLES MATURANA

Orientador: Alexandre César Rodrigues da Silva

Dissertação apresentada à Faculdade de Engenharia - UNESP – Campus de Ilha Solteira, para obtenção do título de Mestre em Engenharia Elétrica.

Área de Conhecimento: Automação.

Ilha Solteira – SP

Novembro de 2010

FICHA CATALOGRÁFICA

. Elaborada pela Seção Técnica de Aquisição e Tratamento da Informação
Serviço Técnico de Biblioteca e Documentação da UNESP - Ilha Solteira

Maturana, Patrícia Salles.
M445a Algoritmos de detecção de bordas implementados em FPGA / Patrícia Salles Maturana. -- Ilha Solteira : [s.n.], 2010
155 f. : il.

Dissertação (mestrado) - Universidade Estadual Paulista. Faculdade de Engenharia de Ilha Solteira. Área de conhecimento: Automação, 2010

Orientador: Alexandre César Rodrigues da Silva
Inclui bibliografia

1. Processamento de imagens. 2. NIOS. 3. Operadores de bordas. 4. Roberts. 5. Prewitt. 6. Sobel.

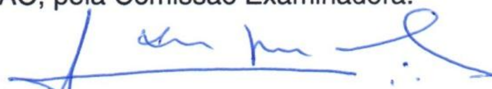
unesp UNIVERSIDADE ESTADUAL PAULISTA
CAMPUS DE ILHA SOLTEIRA
FACULDADE DE ENGENHARIA DE ILHA SOLTEIRA

CERTIFICADO DE APROVAÇÃO

TÍTULO: ALGORITMOS DE DETECÇÃO DE BORDAS IMPLEMENTADOS EM FPGA

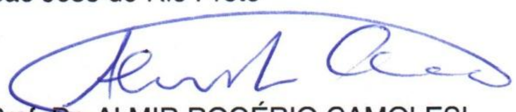
AUTORA: PATRICIA SALLES MATURANA
ORIENTADOR: Prof. Dr. ALEXANDRE CESAR RODRIGUES DA SILVA

Aprovada como parte das exigências para obtenção do Título de Mestre em Engenharia Elétrica , Área: AUTOMAÇÃO, pela Comissão Examinadora:



Prof. Dr. ALEXANDRE CESAR RODRIGUES DA SILVA
Departamento de Engenharia Elétrica / Faculdade de Engenharia de Ilha Solteira

Prof. Dr. ALEDIR SILVEIRA PEREIRA
Departamento de Cienc Comp e Estatistica / Instituto de Biociências, Letras e Ciências Exatas de São José do Rio Preto



Prof. Dr. ALMIR ROGÉRIO CAMOLESI
Coordenadoria de Informática / Instituto Municipal de Ensino Superior de Assis (IMESA)

Data da realização: 26 de novembro de 2010.

Dedico este trabalho à minha família, ao meu namorado e amigos que sempre me apoiaram no meu trabalho e nas minhas decisões.

Agradecimentos

Há muitas pessoas que desejo agradecer por acreditarem em meu trabalho e me apoiarem.

Primeiramente agradeço a minha família que sempre esteve do meu lado, mesmo nas horas mais difíceis, em especial aos meus pais, Miguel e Mary, que mesmo ocorrendo momentos difíceis em minha família se mantiveram ao meu lado, me dando força e apoio.

Ao meu namorado Eduardo de Souza, por estar sempre ao meu lado me ajudando, até mesmo nas madrugadas de estudo, me apoiando para não desanimar dizendo que tudo terminaria bem.

Agradeço imensamente ao meu orientador Alexandre César Rodrigues por acreditar em mim e por sua paciência e apoio, e a quem devo tanto no meu crescimento acadêmico, quanto meu crescimento pessoal.

Aos meus companheiros de laboratório em especial ao Tércio Alberto dos Santos que me ajudou e me ensinou uma grande parte dos meus estudos com muita paciência.

À CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pelo auxílio financeiro no desenvolvimento deste trabalho.

Por fim, agradeço à todos os amigos que me ajudaram, e em especial a um grande amigo Murilo (Et) que me ajudou em meu projeto e pela amizade dedicada, e minhas amigas de república Raiane, Cristiane, Ana Alice, Juliana e Silvia, por momentos de carinho e amizade, mantendo a nossa casa um lugar agradável de se morar.

RESUMO

O processamento de imagens é uma área promissora na automação, por poder ser aplicado nas mais variadas atividades da tecnologia como, por exemplo, na medicina, na agricultura de precisão, dentre muitas outras. Este trabalho consiste na aplicação da área de processamento de imagens, voltada a área de segmentação, com os operadores de bordas *Roberts*, *Prewitt* e *Sobel*. Tendo também muita contribuição na área de processamento de imagens e sistemas embarcados, implementando os detectores de bordas na placa FPGA (*Field Programmable Gate. Array*), por meio de software e de simulações do hardware. A configuração do processador NIOS permitiu a instalação do sistema operacional uClinux e de um software descrito na linguagem C ANSI com a imagem em níveis de cinza particionada em quatro sub-imagens. O hardware gerado, foi modelado com a linguagem de descrição de hardware VHDL (*VHSIC – Hardware Description Language*). Para serem comparadas às imagens geradas, detectores de bordas no ambiente MATLAB foram aplicado por ser uma ferramenta conhecida, usual, com funções para aplicações na área de processamento de imagens. Para ter um melhor entendimento sobre os algoritmos de detecção de bordas, foram criados algoritmos na linguagem C ANSI.

Palavras-chaves: Processamento de imagem. NIOS. FPGA. *Roberts*. *Prewitt*. *Sobel*.

ABSTRACT

Image processing is a promising area for automation, because it can be applied in a variety of technology activities, for example, in medicine, precision agriculture, among many others. This work is the application of image processing area, facing the segmentation area, with the operators of edges Roberts, Prewitt and Sobel. Also having a lot of contribution in the field of image processing and embedded systems, implementing the edge detectors in the FPGA board by means of simulation software and hardware. The NIOS processor configuration allowed the installation of the uClinux operating system and software described in the ANSI C language with the image in grayscale partitioned into four sub-images. The hardware generated, was modeled with the hardware description language VHDL (VHSIC - Hardware Description Language). To be compared to the images generated, edge detectors were implemented in MATLAB, a tool known, usual, with functions for applications in image processing. To get a better understanding of the edge detection algorithms were created algorithms in ANSI C language.

Keywords: Image processing. NIOS. FPGA. Roberts. Prewitt. Sobel.

Lista de Figuras

Figura 1. 1. Fluxograma da metodologia empregada. 17
Figura 2. 1. Convenção dos eixos para representação de imagens digitais (GONZALES; WOODS, 2000). 21
Figura 2. 2. Sistema *G (x, y)* de Transmissão de imagens em 1921(GONZALES; WOODS, 2002). 22
Figura 2. 3. Sistema *G (x, y)* de Transmissão de imagens em 1922(GONZALES;WOODS, 2002). 23
Figura 2. 4. Sistema *G (x, y)* de Transmissão de imagens para 15 níveis de cinza distintos(GONZALES; WOODS, 2002). 23
Figura 2. 5. Primeira foto tirada da lua por um veículo espacial U.S (Ranger 7) (Cortesia da NASA) (GONZALES; WOODS, 2002). 24
Figura 2. 6. Passos fundamentais em processamento de imagens digitais. 25
Figura 2. 7. Conceitos de vizinhança de 4, vizinhança diagonal e vizinhança de 8. 26
Figura 2. 8. Máscara de *Roberts.* 32
Figura 2. 9. Máscaras de *Prewitt.* 33
Figura 2. 10. Máscaras de *Sobel.* 33
Figura 2. 11. (a) Região de uma imagem 3 x 3; (b) Máscara usada para o cálculo de G_x no ponto central da região 3 x 3; (c) Máscara usada para o cálculo de G_y no ponto central da região 3 x 3. 34
Figura 2. 12. Imagem original. 35
Figura 2. 13. Imagem com a aplicação do operador de *Sobel* processada no algoritmo sobel.m no MATLAB. 35
Figura 2. 14. Máscara usada para a detecção de pontos isolados, a partir de um fundo constante. 37
Figura 2. 15. (a) Máscara para linha horizontal. (b) Máscara para linha a +45º. (c) Máscara para linha vertical. (d) Máscara para linha a -45º. 37
Figura 2. 16. Imagem Original. 38
Figura 2. 17. Detecção de Bordas com o operador de *Roberts.* 39
Figura 2. 18. Detecção de Bordas com o operador de *Prewitt* 39
Figura 2. 19. Detecção de Bordas com o operador de *Sobel.* 39
Figura 3. 1. Imagem original. 41
Figura 3. 2. Imagem I redimensionada. 42
Figura 3. 3. Imagem em níveis de cinza e redimensionada. 42
Figura 3. 4. Imagem em níveis de cinza no formato txt. 43
Figura 3. 5. Imagem em níveis de cinza sem o delimitador. 43
Figura 3. 6. Imagens em níveis de cinza. 45
Figura 3. 7. Imagem processada com o operador de *Roberts.* 45
Figura 3. 8. Imagens com o operador de *Prewitt.* 46
Figura 3. 9. Imagens com o operador de *Sobel.* 47
Figura 3. 10. Arquivo com formato txt produzido por programa desenvolvidos na linguagem C. 48
Figura 3. 11. Imagem em níveis de cinza. 49

Figura 3. 12. Imagem processada por um programa em C e visualizada no MATLAB. 49
Figura 3. 13. Arquivo com formato txt..50
Figura 3. 14. Imagem processada com o programa nios_sobel.c. 50
Figura 3. 15. Arquivo tbl gerado pela simulação do modelo VHDL. 51
Figura 3. 16. Arquivo txt alterado a partir do resultado da simulação. 51
Figura 3. 17. Arquivo contendo o resultado da simulação, formatado para a visualização no MATLAB. 52
Figura 3. 18. Imagem em níveis de cinza. 52
Figura 3. 19. Imagem processada pelo modelo VHDL. 53
Figura 3. 20. Imagem resultado de 5 ovos processada com o operador de *Sobel*. 54
Figura 3. 21. Gráfico de *Hough* para a imagem da Figura 3.20. 54
Figura 3. 22. Imagem resultado de 3 ovos processada com o operador de *Sobel*. 55
Figura 3. 23. Gráfico de *Hough* para a imagem da Figura 3.22. 55
Figura 4. 1. Imagens em níveis de cinza pré-processada pelo MATLAB. 59
Figura 4. 2. Imagens processada pelo operador de *Roberts*. 59
Figura 4. 3. Imagens processada pelo o operador de *Prewitt*. 60
Figura 4. 4. Imagens processada pelo o operador de *Sobel*. 61
Figura 4. 5. Imagem em níveis de cinza e redimensionada. 63
Figura 4. 6. Resultado da imagem processada pelo operador de *Roberts* no processador NIOS. 64
Figura 4. 7. Resultado da imagem processada pelo operador de *Prewitt* no processador NIOS. 64
Figura 4. 8. Resultado da imagem processada pelo operador de *Sobel* no processador NIOS. 64
Figura 5. 1. Arquivo em níveis de cinza obtido pelo algoritmo sobel.m. 67
Figura 5. 2. Arquivo em níveis de cinza alterado pelo algoritmo alteração_de_arquivos.c para o vetor do modelo VHDL 67
Figura 5. 3. Simulação do algoritmo de *Sobel* com imagem de dimensão de 148x148 *pixels*. 73
Figura 5. 4. Simulação com extensão tbl 73
Figura 5. 5. Arquivo nulo1.txt alterado pelo algoritmo vhdl_mat.m......................................74
Figura 5. 6. Arquivo alterado pelo algoritmo teste_nulo.c. 74
Figura 5. 7. Arquivo redimensionado pelo algoritmo img_quartus.c.....................................75
Figura 5. 8. Imagem em níveis de cinza. 75
Figura 5. 9. Imagem processada pelo detector de bordas de *Roberts* modelada em VHDL. ...76
Figura 5. 10. Imagem processada pelo detector de bordas de *Prewitt* modelada em VHDL. ..76
Figura 5. 11. Imagem processada pelo detector de bordas de *Sobel* modelada em VHDL. 77
Figura 6. 1– Placa DE2 – 2C35 da Altera (ALTERA, 2010). 78
Figura 6. 2. Esquemático – Bloco1.bdf 82
Figura 6. 3. Componentes do SOPC-Builder 83
Figura 6. 4. Ambiente computacional QUARTUS. 84
Figura 6. 5. Terminal *uClinux*. 84
Figura 6. 6. Acesso ao cartão de memória SD_Card. 85
Figura 6. 7. Arquivos gerados. 86
Figura 6. 8. Resultado da imagem processada pelo operador de *Roberts*. 86

Figura 6. 9. Resultado da imagem processada pelo operador de *Prewitt*. 86
Figura 6. 10. Resultado da imagem processada pelo operador de *Sobel*. 87
Figura 7. 1. Comparação entre os operadores de bordas de *Roberts*, *Prewitt* e *Sobel* – (a) *Roberts*; (b) *Prewitt*; (c) *Sobel*. 89
Figura 7. 2. Comparação entre os resultados para o operador de *Roberts* – (a) MATLAB; (b)C ANSI. 90
Figura 7. 3. Comparação entre os resultados para o operador de *Prewitt* – (a) MATLAB; (b)C ANSI. 90
Figura 7. 4. Comparação entre os resultados para o operador de *Sobel* – (a) MATLAB; (b)C ANSI. 90
Figura 7. 5. Comparação entre os resultados para o operador de *Roberts* – (a) MATLAB; (b)VHDL. 91
Figura 7. 6. Comparação entre os resultados para o operador de *Prewitt* – (a) MATLAB; (b)VHDL. 91
Figura 7. 7. Comparação entre os resultados para o operador de *Sobel* – (a) MATLAB; (b)VHDL. 92
Figura 7. 8. Comparação entre os resultados para o operador de *Roberts* – (a) MATLAB; (b)NIOS. 92
Figura 7. 9. Comparação entre os resultados para o operador de *Prewitt* – (a) MATLAB; (b)NIOS. 93
Figura 7. 10. Comparação entre os resultados para o operador de *Sobel* – (a) MATLAB; (b)NIOS. 93
Figura 7. 11. Comparação entre os ambientes estudados para o operador de *Roberts* - (a) MATLAB; (b) C ANSI; (c) VHDL; (d) NIOS 93
Figura 7. 12. Comparação entre os ambientes estudados para o operador de *Prewitt* - (a) MATLAB; (b) C ANSI; (c) VHDL; (d) NIOS 94
Figura 7. 13. Comparação entre os ambientes estudados para o operador de *Sobel* - (a) MATLAB; (b) C ANSI; (c) VHDL; (d) NIOS 94

Lista de Tabelas

Tabela 2. 1. Operadores utilizados para estimar a amplitude do gradiente através de uma borda. 38
Tabela 5. 1. Máscaras do operador de *Roberts*. 75
Tabela 5. 2. Máscaras do operador de *Prewitt*. 76
Tabela 5. 3. Máscaras do operador de *Sobel*. 77

Sumário

CAPÍTULO 1 Introdução 14
1. 1. Metodologia empregada 16
1. 2. Descrição dos passos da Metodologia 16
1. 3. Descrição dos capítulos 19

CAPÍTULO 2 Teoria Sobre Processamento De Imagem Utilizada 21
2. 1. Imagem Digital 21
2. 2. A origem do processamento da imagem digital 22
2. 3. Os passos fundamentais utilizados no processamento de imagens 25
2. 4. Algumas relações básicas entre os *pixels* 26
2. 5. Realce de imagens 27
2.5. 1. Métodos no Domínio Espacial 27
2.5. 2. Operações de Convolução com Máscaras 28
2.5. 3. Filtro por derivada 31
2.5. 4. Operadores de gradiente 33
2. 6. Segmentação de imagens 35
2.6. 1. Detecção de Pontos 36
2.6. 2. Detecção de Linhas 37
2.6. 3. Detecção de Bordas 37
2. 7. Limiarização 40
2. 8. Transformada de *Hough* 40

CAPÍTULO 3 Estudo dos Detectores de Borda Empregando o MATLAB 41
3. 1. Aplicação dos operadores de bordas utilizando como ambiente computacional o MATLAB. 43
3.1. 1. Operador de *Roberts*. 44
3.1. 2. Operador de *Prewitt* 46
3.1. 3. Operador de *Sobel*. 46
3. 2. Visualização das imagens processadas em outros ambientes 47
3. 3. Transformada de *Hough* 53

CAPÍTULO 4 Estudo dos Detectores de Bordas Utilizando a Linguagem C. 56
4. 1. Algoritmos de detecção de borda programados em linguagem C. 56
4.1. 1. Operador de *Roberts* 58
4.1. 2. Operador de *Prewitt* 59
4.1. 3. Operador de *Sobel* 60
4. 2. Implementação dos algoritmos para detecção de bordas no processador NIOS 61
4.2. 1. Algoritmo mat_inicial.c 62
4.2. 2. Algoritmos mat1.c, mat2.c, mat3.c e mat4.c 62
4.2. 3. Algoritmo mat_final.c 63

CAPÍTULO 5 Estudo dos Detectores de Bordas Utilizando a Linguagem VHDL. 66
5. 1. Operador de *Roberts* 75
5. 2. Operador de *Prewitt* 76
5. 3. Operador de *Sobel* 77

CAPÍTULO 6 Configuração do Processador NIOS e o Sistema Operacional *uClinux* .. 78
6. 1. Dispositivo FPGA 78
6. 2. Kit de desenvolvimento DE2 78
6. 3. Processador NIOS 79
6. 4. Sistema operacional *uClinux* 80
6. 5. Aplicação do projeto 80
6.5. 1. Construção do Hardware 81
6.5. 2. Implementação do *Software* 85

CAPÍTULO 7 CONCLUSÃO 88
7. 1. Comparação entre os resultados 88
7. 2. Trabalhos Futuros 94

Referências 96
APÊNDICE A. Algoritmos criados no ambiente computacional MATLAB. 98
APÊNDICE B. Algoritmos criados no ambiente computacional Dev-C++ 104
APÊNDICE C. Algoritmos criados no ambiente computacional QUARTUS 154
APÊNDICE D. Passos do projeto aplicado na placa FPGA DE2 – 2C35 162

CAPÍTULO 1

Introdução

O objetivo deste trabalho é o estudo e a implementação dos operadores de bordas utilizando-se as máscaras de *Roberts*, *Prewitt* e *Sobel*. Os algoritmos foram implementados em MATLAB, C ANSI e VHDL. A implementação em MATLAB serviu como referência na validação das outras implementações. Como a meta deste projeto é o desenvolvimento de sistemas embarcados, utilizou como tecnologia alvo a FPGA EP2C-35F672C6 da Altera.

Com este dispositivo foi possível avaliar o desempenho do *hardware* através da simulação gerada a partir do VHDL produzido. Assim como, para configurar de um processador NIOS executando o *uClinux* e o C ANSI.

A literatura que trata do processamento de imagem utilizando-se dispositivos FPGAs está bastante difundida atualmente, em decorrência da importância do emprego deste tipo de tecnologia em sistemas embarcados.

No trabalho intitulado "*An Enviroment for Generating FPGA Architectures for Image Algebra-based Algorithms*" (CROOKES, 2002) fala-se sobre os algoritmos de processamento de imagem de alto desempenho aplicada na FPGA, série Xilinx XC6200. Utiliza um processamento paralelo, com o software aplicado descrito em linguagem Prololog. Empregou-se máscaras de 3x3, implementando na placa FPGA.

No trabalho de Zongqing (ZONGQING, 2008) intitulado "*An Embedded System with uClinux based on FPGA*" utilizou-se a FPGA com o sistema operacional *uClinux*, onde implementou-se um MPEG2-player, trabalhando com o decodificador de áudio, vídeo, SD_Card, Ethernet, monitor LCD e USB. A comparação entre a FPGA e a ASIC foi apresentada.

No artigo intitulado "*A Proposed FPGA Based Architecture for Sobel Edge Detection Operator*" (ABBASI, 2007) trabalhou-se com o software FPGA Advantage 6.3 da Mentor Graphics e com o simulador Modelsim SE 5.8c, sintetizado no Leonardo Spectrum 2004, ambos da Mentor Graphics. Este trabalho apresenta uma arquitetura para a placa FPGA Xilinx Spartan 3 XC3S50-5PQ208 baseada no operador de detecção de bordas de *Sobel*, processando os algoritmos do operador de *Sobel* em tempo real.

No artigo de Benkrid (BENKRID, 1999) intitulado "*A High Level Software Environment for FPGA Based Image Processing*" descreve-se sobre a implementação de operações de processamento de imagens na FPGA, com ênfase nas operações de vizinhança, como o algoritmo de detecção de borda de *Sobel* e a linguagem usada é a linguagem C++. Neste trabalho o autor implementa o algoritmo de *Sobel* separando as duas máscaras na fase de convolução, ou seja, realiza as operações com a máscara vertical e com a máscara horizontal, separadamente para depois soma dos resultados entre duas operações.

No artigo "*An Efficient Reconfigurable Architecture and Implementation of Edge Detection Algorithm using Handle-C*" (RAO, 2007) utiliza a linguagem de hardware Handle-C para aplicar o detector de borda de *Canny*. O *hardware* modelado foi implementado usando a ferramenta PK2 IDE na placa FPGA Vertex Xilinx RC1000. Antes de aplicar o detector de borda de *Canny* foi aplicado a máscara de convolução *Gaussian* para a suavização dos ruídos na imagem.

No trabalho intitulado "*Hardware Implementation of Sobel-Edge Detection Distributed Arithmetic Digital Filter*" o autor Sorawat (APREEACHA, 2004) trata da aplicação do operador de borda de *Sobel* na placa FPGA, realizando a convolução entre a máscara de *Sobel* e a imagem de entrada. Foi utilizada a FPGA EPF10K20RC240-4 na Flex10k. Comparou-se os resultados obtidos na implementação com os resultados gerados pelo MATLAB.

No trabalho intitulado "*FPGA Based Sobel as Vehicle Edge Detector In VCAS*" (XUE, 2003) descreve sobre o detector de bordas de *Sobel*, no sistema de anti-colisão de veículos. E para criar este sistema foi aplicado o operador de *Sobel* na placa FPGA. Neste trabalho inicialmente fala sobre a tecnologia de identificação de imagem introduzidas para VCAS (sistema anti-colisão de veículo) para determinar se os veículos estão correndo ou não, se estes veículos oferecem riscos ou não.

O nosso trabalho assemelha-se ao trabalho de Benkrid, contudo a imagem original foi particionada em quatro sub-imagens antes do processamento. Isto foi necessário devido à limitação de memória disponível no dispositivo FPGA utilizado. Assemelha-se também ao trabalho de Apreeacha, pois utiliza o MATLAB para o pré-processamento das imagens e para a visualização da imagem processada.

1. 1. Metodologia empregada

A metodologia deste trabalho iniciou-se pelo passo de aquisição de imagens para a implementação na placa FPGA com os operadores de bordas de *Roberts*, *Prewitt* e *Sobel*. Para ocorrer esta implementação foram necessários alguns passos mais básicos em outros ambientes computacionais, para poder aprender e descrever melhor os algoritmos dos operadores de bordas. O primeiro passo foi adquirir as imagens por meio de máquinas fotográficas, para em seguida trabalhar com elas no ambiente computacional MATLAB, pois os resultados deste serviu como base de comparação com os outros ambientes computacionais trabalhados. Em seguida trabalhou-se com o ambiente computacional Dev-C++, para entender melhor os algoritmos e assim poder implementar na placa FPGA. O terceiro passo foi trabalhar com a linguagem de descrição de *hardware* VHDL no ambiente QUARTUS, e com a simulação pode-se visualizar a imagem com as bordas destacadas pelos operadores. E por fim, implementou-se os operadores de bordas de *Roberts*, *Prewitt* e *Sobel* na placa FPGA com o *software* descrito na linguagem C ANSI, com o processador NIOS II e o sistema operacional *uClinux*. Visualiza-se os passos desta metodologia no fluxograma da Figura 1.1.

1. 2. Descrição dos passos da Metodologia

Cada cor representa um processo diferente, ou seja:

- **Preto:** Passos primários e a parte inicial de todos os processos;
- **Vermelho:** Operadores de bordas aplicados no ambiente computacional Dev-C++;
- **Azul:** Operadores de bordas aplicados no ambiente computacional QUARTUS;
- **Roxo:** Operadores de bordas aplicados no ambiente computacional MATLAB;
- **Verde:** Operadores de bordas aplicados na placa FPGA;

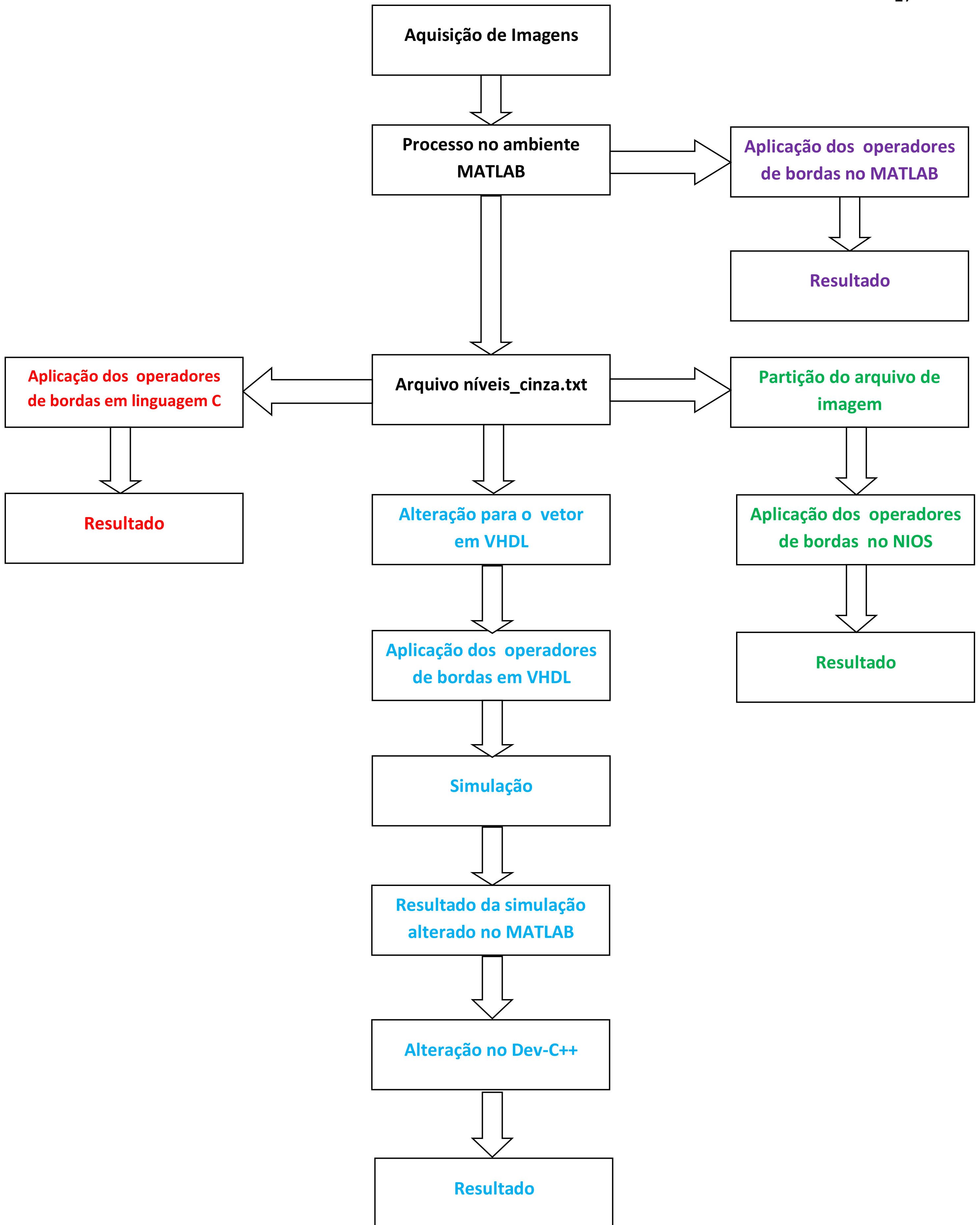


Figura 1. 1. Fluxograma da metodologia empregada.

Passos:

- **Aquisição de imagens:** As imagens foram obtidas por meio de máquina fotográfica *General Electric Company Digital Câmera X5*;
- **Processo no ambiente MATLAB:** Converte a imagem para níveis de cinza e a redimensiona;
- **Aplicação dos operadores de bordas no MATLAB:** Aplicou os operadores de bordas de *Roberts*, *Prewitt* e *Sobel* na imagem convertida;
- **Arquivo níveis_cinza.txt**: Gerou-se o arquivo da imagem em níveis de cinza com o formato txt para trabalhar nos outros ambientes computacionais, ou seja, o Dev-C++ e o QUARTUS e na aplicação na placa FPGA;
- **Aplicação dos operadores de bordas em linguagem C**: Aplicar os operadores de bordas de *Roberts*, *Prewitt* e *Sobel* no arquivo de imagem em níveis de cinza;
- **Alteração para o vetor em VHDL**: Alterar o arquivo em níveis de cinza para gerar um vetor na linguagem de descrição de *hardware* VHDL.
- **Aplicação dos operadores de bordas em VHDL**: Aplicar os operadores de bordas de *Roberts*, *Prewitt* e *Sobel* descrito em VHDL;
- **Simulação**: Simular para gerar a forma de onda no *waveform editor* depois da compilação do algoritmo descrito em VHDL dos operadores de bordas de *Roberts*, *Prewitt* e *Sobel* no ambiente computacional QUARTUS;
- **Resultado da simulação alterado no MATLAB:** Alterar o arquivo obtido pela simulação para utilizar somente o nível lógico obtido quando ocorre a transição do *clock* para nível baixo, colocando o número 2 no lugar do nível lógico de transição do *clock* para nível alto;
- **Alteração no Dev-C++**: Alterar o arquivo da simulação para a retirada do número 2;
- **Partição do arquivo de imagem**: Particionar a imagem em quatro partes, processando cada parte da imagem separadamente para implementá-la na placa FPGA com o processador NIOS;
- **Aplicação dos operadores de bordas no NIOS**: Implementar os algoritmos com as imagens processadas particionadas na FPGA com o processador

NIOS juntamente com o sistema operacional *uClinux*, mandado o resultado para um Sd_Card (cartão de memória);

- **Resultado:** Visualizar a imagem no ambiente computacional MATLAB.

1. 3. Descrição dos capítulos

No capítulo 2 faz-se uma breve introdução sobre a teoria de processamento de imagem digital e a história de suas primeiras aplicações realizadas e os passos para processá-las, como por exemplo, a segmentação de imagens, mais especificamente as operações de convolução com operadores de bordas *Roberts*, *Prewitt* e *Sobel* (GONZALEZ; WOODS, 2002).

Com esses operadores realiza-se a convolução com a imagem original, e em seguida ocorre a limiarização da imagem, ou seja, os *pixels* ("*Picture Elements*") cujo valor for superior a um limite pré-estabelecido terão o valor lógico 1, e corresponderão as bordas do objeto, e aqueles cujos valores fossem inferiores a este limite terão o valor zero e corresponderão ao fundo da imagem obtendo uma imagem processada com as bordas do objeto destacadas do fundo da imagem.

No capítulo 3 trata-se sobre a aplicação dos operadores de bordas de *Roberts*, *Prewitt* e *Sobel* no ambiente MATLAB, desenvolvendo um algoritmo para cada operador estudado. Ainda no ambiente MATLAB desenvolveu-se algoritmos para visualizar as imagens obtidas em outros ambientes, como por exemplo, no ambiente MATLAB gerou arquivos com formato txt para processá-los nos ambientes computacionais Dev-C++ e QUARTUS. Visualizam-se os resultados obtidos no ambiente MATALB.

No capítulo 4 apresenta-se duas formas de aplicação com os operadores de bordas *Roberts*, *Prewitt* e *Sobel*, utilizando-se o ambiente computacional Dev-C++. A primeira aplicação é processar as imagens no formato txt com os operadores de bordas. A segunda aplicação com a linguagem C ANSI é particionando a imagem e implementando na placa FPGA com o processador NIOS e com o sistema operacional *uClinux*.

Ainda no capítulo 4 trata-se também de algoritmos para auxiliar com a aplicação realizada no ambiente computacional QUARTUS para o vetor da linguagem VHDL e para visualizar a imagem obtida a partir da simulação.

No capítulo 5 apresenta-se a aplicação com os operadores de bordas de *Roberts*, *Prewitt* e *Sobel* no ambiente computacional QUARTUS descrevendo os algoritmos com a linguagem VHDL, alterando a dimensão em *pixels* para obter o melhor resultado. Para visualizar os resultados, precisou-se do auxílio dos ambientes MATLAB e Dev-C++.

No capítulo 6 trata-se da aplicação dos operadores de bordas *Roberts*, *Prewitt* e *Sobel* na placa FPGA, criando o *hardware* na ferramenta SOPC-Builder do ambiente computacional QUARTUS e criando a imagem do projeto no *uClinux* para também aplicá-la na placa FPGA, para assim, executar o *software* dos operadores descrito na linguagem C ANSI.

No capítulo 7 apresenta-se o que concluiu-se com as aplicações dos operadores de bordas *Roberts*, *Prewitt* e *Sobel* e a comparação entre estes operadores e suas aplicações. Também trata-se da comparação entre os operadores de bordas *Roberts*, *Prewitt* e *Sobel* e entre as aplicações realizadas nos capítulos anteriores, tendo como base os resultados obtidos no ambiente computacional MATLAB. Fala-se também sobre trabalhos futuros e próximas aplicações.

CAPÍTULO 2

Teoria Sobre Processamento De Imagem Utilizada

O processamento de imagens digitais pode ser empregado em áreas como medicina, cinema, indústrias, entre outras. O seu objetivo é a melhoria da informação visual para interpretação humana e o processamento de dados de cenas para percepção automática, ou seja, com auxílio de máquinas.

2. 1. Imagem Digital

Uma imagem digital pode ser definida como uma função bidimensional, *f(x, y)*, em que *x* e *y* são coordenadas espaciais, e a amplitude de *f*, em qualquer par de coordenadas *(x, y)* é chamada de intensidade também denominada de nível de cinza da imagem no determinado ponto. Quando *x* e *y* e os valores da amplitude de *f* são todos finitos, quantidades discretas, a imagem é chamada de imagem digital. A área de processamento digital de imagens refere-se ao processamento de imagens por meio de um computador digital.

A imagem na Figura 2.1 pode ser representada por uma função bidimensional de *f(x, y)*, onde *x* e *y* indicam as coordenadas espaciais e *f(x, y)* indica a intensidade do nível de cinza da imagem na dada coordenada *(x, y)*.

Figura 2. 1. Convenção dos eixos para representação de imagens digitais (GONZALES; WOODS, 2000).

A imagem digital é uma imagem *f(x, y)* discretizada, tanto em coordenadas espaciais quanto em brilho, e pode ser representada computacionalmente como uma matriz MxN, onde o cruzamento de linha com coluna indica um ponto na imagem, e o valor contido naquele

ponto indica a intensidade de brilho, ou seja, o nível de cinza contido naquele ponto. Os pontos em uma imagem são chamados de *pixels* (“*Picture Elements*”).

A Matriz 1 abaixo representa um exemplo de um formato descritivo de uma imagem digital.

$$f(x,y) \cong \begin{bmatrix} f(0,0) & f(0,1) & \dots & f(0,N-1) \\ f(1,0) & f(1,1) & \dots & f(1,N-1) \\ \vdots & \vdots & \dots & \vdots \\ f(M-1,0) & f(M-1,1) & \dots & f(M-1,N-1) \end{bmatrix}$$

Matriz 1

2. 2. A origem do processamento da imagem digital

Uma das primeiras aplicações utilizando processamento de imagens ocorreu em Londres, quando as imagens digitalizadas eram enviadas para jornais da cidade de Nova York através de um cabo submarino.

No início da década de 1920, com o sistema *G (x, y)* de transmissão de imagens via cabo submarino, reduziu-se o tempo de envio de imagens, de mais de uma semana para menos de três horas, pelo Oceano Atlântico. Quando a imagem chegava ao seu destino era codificada para transmissão a cabo, por um equipamento de impressão, e posteriormente era reconstruída em um terminal receptor. Em uma impressora telegráfica, as imagens codificadas eram reconstruídas contendo apenas caracteres para simulação de padrões de tons intermediários. A Figura 2.2 foi transmitida e reproduzida desta maneira, sendo uma das primeiras imagens enviadas pelo sistema *G (x, y)* em 1921.

O método de impressão utilizado para obter a Figura 2.2 foi abandonado no final de 1921, em favor de uma técnica baseada na reprodução fotográfica feita a partir de fitas perfuradas no telégrafo do terminal receptor. A Figura 2.3 mostra uma imagem obtida com este método. Comparando a Figura 2.2 com a Figura 2.3, nota-se uma melhora tanto na qualidade quanto na resolução.

Figura 2. 2. Sistema *G (x, y)* de Transmissão de imagens em 1921(GONZALES; WOODS, 2002).

Figura 2. 3. **Sistema *G (x, y)* de Transmissão de imagens em 1922(GONZALES;WOODS, 2002).**

Os primeiros sistemas *G (x, y)*s exemplificados nas Figuras 2.2 e 2.3 eram capazes de codificar imagens em até 5 níveis de cinza distintos, já a Figura 2.4 mostra que essa capacidade foi aumentada para 15 níveis distintos. Embora os exemplos citados anteriormente envolvam imagens digitais, não são considerados como processamento digital de imagens, pois não envolvia computadores no processamento da imagem.

A história do processamento digital está ligada ao desenvolvimento do computador digital ao longo dos anos, pois para se poder armazenar e processar as imagens digitais necessita-se de muito poder computacional. Sendo assim, o progresso na área de processamento digital de imagens depende do desenvolvimento dos computadores digitais e de tecnologias para ajudar em relação a armazenamento, visualização e transmissão de dados.

Figura 2. 4. **Sistema *G (x, y)* de Transmissão de imagens para 15 níveis de cinza distintos(GONZALES; WOODS, 2002).**

No início dos anos 60, surgiram os primeiros computadores mais potentes com capacidade de realizar as tarefas de processamento de imagem. A combinação dessas máquinas com o início do programa espacial resultou no processamento digital de imagem dos dias de hoje.

Em 1964, imagens da lua foram transmitidas por *Ranger 7*, um veículo espacial lançado para a lua, e processadas por um computador para corrigir distorções da imagem. A Figura 2.5 mostra a imagem transmitida da lua.

Figura 2. 5. Primeira foto tirada da lua por um veículo espacial U.S (Ranger 7) (Cortesia da NASA) (GONZALES; WOODS, 2002).

Em paralelo com o programa espacial, no final dos anos 60 e início dos anos 70, deu-se o início do processamento digital na área médica, com a invenção da tomografia computadorizada, sendo de grande avanço e de extrema importância na área da medicina e significativo progresso na área de processamento de imagens.

De 1960 até os dias de hoje, o campo de processamento de imagem tem tido acelerado progresso, usando mais técnicas além das aplicações utilizadas no campo espacial e na medicina, como por exemplo, procedimentos para aumentar o contraste da imagem. Uma aplicação que pode ser dada como exemplo é quando geógrafos usam técnicas de processamento de imagens para estudar padrões de poluição a partir de imagens aéreas e de satélite. Estes exemplos são relacionados com o tratamento das imagens para a interpretação humana.

A área de aplicação relacionada com a percepção da máquina é voltada aos procedimentos para a extração de uma informação da imagem de forma adequada para o processamento do computador. Na maioria das vezes, essa informação tem pouca semelhança com os recursos visuais utilizados pelos humanos para interpretar o conteúdo de uma imagem. Alguns dos exemplos usados para a percepção da máquina são momentos estatísticos, coeficientes de Fourier, multidimensional e medidas de distância. As técnicas de processamento de imagem utilizadas na percepção de máquina permitem: o reconhecimento automático de caracteres, de visão de máquina industrial para a montagem do produto e de inspeção, reconhecimento militar, processamento automático de impressões digitais, análise de raios-X e coleta de sangue, e máquinas de processamento de imagens aéreas e de satélite para a previsão do tempo e avaliação ambiental.

2. 3. Os passos fundamentais utilizados no processamento de imagens

Os processos a serem realizados em processamento digitais de imagens são:

- **Aquisição de imagens:** Obtenção de uma imagem discretizada;
- **Pré-Processamento:** Melhora a imagem, envolvendo técnicas de realce e contrastes;
- **Segmentação:** Particiona uma imagem em unidades mais significativas;
- **Representação e descrição**: Representa os dados como regiões completas ou fronteiras;
- **Reconhecimento e interpretação:** Reconhecimento do objeto de acordo com as informações do autor e atribuição de um significado ao objeto reconhecido;
- **Base de conhecimento:** O domínio do problema está codificado sob o conhecimento em um sistema de processamento de imagens na forma de uma base de conhecimento.

Na Figura 2.6, visualiza-se o diagrama de fluxo dos passos realizados no processamento de imagens, desde a aquisição de imagens até os resultados obtidos.

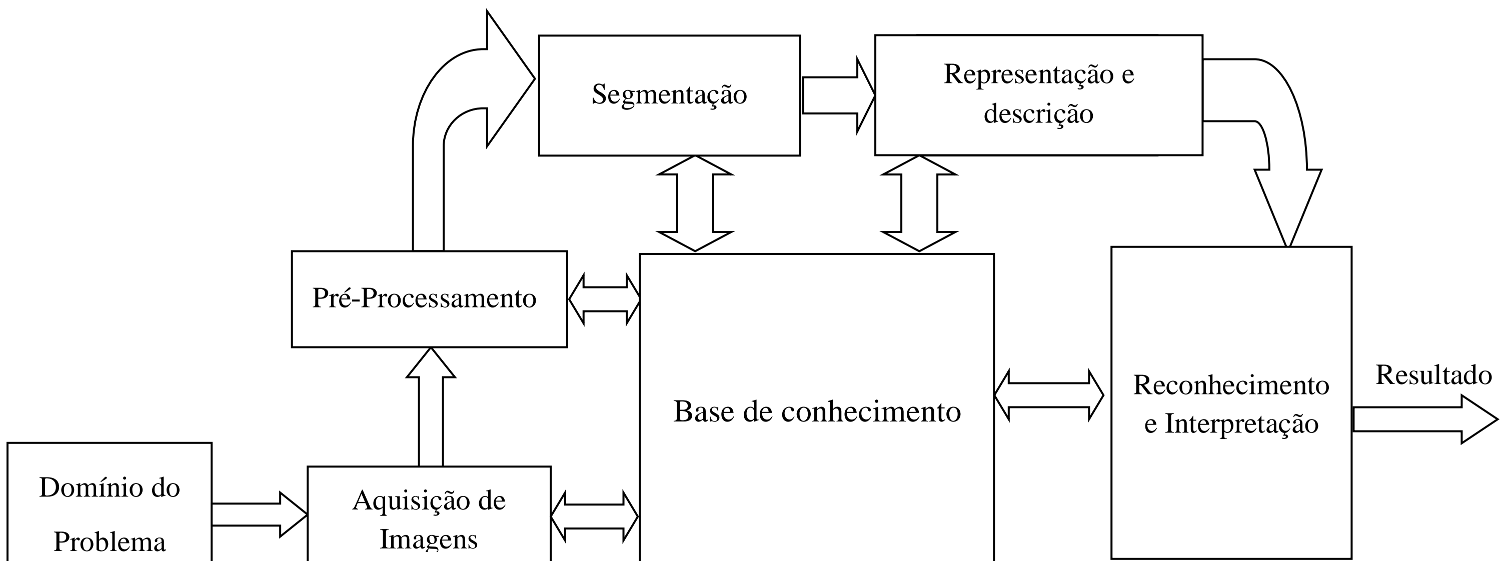


Figura 2. 6. Passos fundamentais em processamento de imagens digitais.

2. 4. Algumas relações básicas entre os *pixels*

Há várias relações importantes entre os *pixels* de uma imagem digital. Uma imagem digital é denotada por *f (x, y)*, e cada *pixel* em particular está sendo denotado como *p* e *q*.

Um *pixel* é um elemento de dimensões finitas na representação de uma imagem digital; a organização de uma imagem é sob a forma de uma matriz de *pixels* feita em uma simetria quadrada parecendo um tabuleiro de xadrez.

Cada *pixel p* nas coordenadas *(x, y)* tem dois vizinhos na horizontal e dois vizinhos na vertical, cujas coordenadas são mostradas na Equação 2.1.

$$(x+1, y), (x-1, y), (x, y+1), (x, y-1) \tag{2.1}$$

Esse conjunto de *pixels*, chamado vizinhança de 4 de *p*, é representado por N4(*p*), onde cada *pixel* está a uma unidade de distância de *(x, y)* sendo que alguns dos vizinhos de *p* ficarão fora da imagem digital se *(x, y)* estiver na borda da imagem.

As coordenadas dos quatro vizinhos diagonais de *p* são mostradas na equação (2.2).

$$(x+1, y+1), (x+1, y-1), (x-1, y+1), (x-1, y-1) \tag{2.2}$$

Essa equação é representada por ND(*p*), todos esses pontos, com a vizinhança de 4, são nomeados como vizinhança de 8 de p, representada por N8(*p*). Na Figura 2.7 visualizam-se os conceitos de vizinhança.

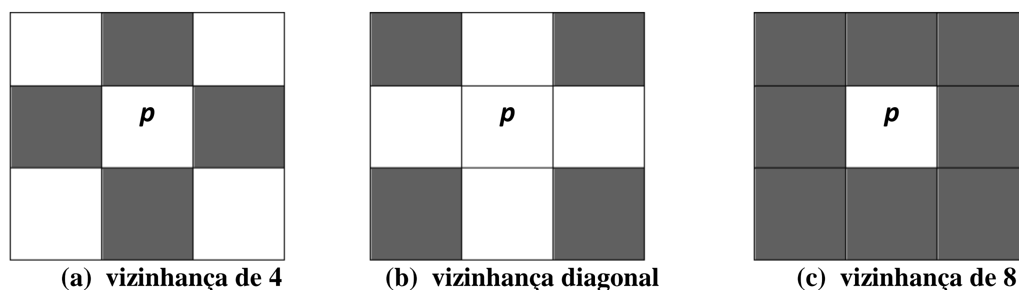


Figura 2. 7. Conceitos de vizinhança de 4, vizinhança diagonal e vizinhança de 8.

2. 5. Realce de imagens

O objetivo principal das técnicas de realce consiste em realçar as bordas e detalhes de uma dada imagem.

Através de técnicas de processamento de imagens, obtém-se uma melhoria nas imagens, com técnicas como o melhoramento de contraste e a técnica de filtragem. Tais técnicas são aplicadas com finalidades para realçar características de interesse ou na recuperação de imagens que sofreram algum tipo de degradação, por causa dos ruídos, perda de contraste ou borramento.

A aplicação dessas técnicas, denominadas como realce de imagem, é uma transformação radiométrica que modifica o valor dos níveis de cinza dos pontos da imagem.

A técnica utilizada neste projeto foi a de filtragem. Um dos processos desse método tem por objetivo extrair informações, como por exemplo, as bordas das imagens, utilizando filtros de detectores de bordas, que são caracterizadas por mudanças abruptas de descontinuidade na imagem.

Os detectores de bordas baseiam-se na idéia de derivadas. Estas medem a taxa de variação de uma função. Em imagens, a variação é maior nas bordas e menor em áreas constantes.

Uma das operações para detectar bordas é conhecida como convolução com máscara. Tal operação é o processo combinado de duas imagens, sendo uma delas uma máscara de convolução, em que os *pixels* da máscara realizam o deslocamento, a multiplicação e a adição com os *pixels* da imagem original.

2.5. 1. Métodos no Domínio Espacial

Domínio espacial refere-se ao conjunto de *pixels* dispostos em um sistema de coordenadas cartesianas que compõem uma imagem, e métodos no domínio espacial são procedimentos que operam diretamente sobre os *pixels*. As funções de processamento de imagens no domínio espacial podem ser representadas como mostrado na equação (2.3), onde, $f(x, y)$ é a imagem de entrada, $g(x, y)$ é a imagem processada, T é um operador sobre f, definido sobre alguma vizinhança de (x, y). O operador T também pode operar sobre um conjunto de imagens de entrada, como na operação *pixel* a *pixel* de M imagens para redução de ruído.

Uma abordagem importante nessa formulação baseia-se na utilização de máscaras, também denominada moldes, janelas ou filtros.

$$g(x, y) = T[f(x, y)] \quad (2.3)$$

2.5. 2. Operações de Convolução com Máscaras

O processo pelo qual duas imagens são combinadas através de operações de deslocamento, multiplicação e adição é chamado de convolução. Usualmente uma das imagens é menor que a outra, sendo assim chamada de máscara de convolução ou simplesmente de janela. As máscaras podem ser projetadas para realizar uma ampla gama de funções de filtragem, por exemplo, os filtros passas-alta e passas-baixa, ou o filtro por derivada, o qual foi aplicado neste trabalho.

As máscaras têm dois tipos de formação, a formação par, por exemplo, máscaras de 2x2 ou 4x4, e a formação ímpar, como por exemplo, máscaras de 3x3 ou 5x5. O resultado do cálculo com a matriz principal da formação par é colocado sobre o primeiro *pixel*, e o da formação ímpar é colocado sobre o *pixel* central.

O resultado da imagem final do cálculo da convolução pode ser menor ou não do que a imagem original processada. Existem duas formas possíveis de resultados, que são a convulução periódica e a convolução aperiódica, sendo que existem dois métodos para este tipo de convolução:

- **Convolução periódica:**
  1. É realizada com o deslocamento da máscara sobre todos os *pixels* da imagem original, como se as bordas externas fossem conectadas.
- **Convolução aperiódica:**
  1. Em que se atribui o valor “0” para os *pixels* da imagem cujos resultados que não puderam ser calculados;
  2. Em que se coloca a máscara centralizada com o primeiro *pixel* da imagem, atribuindo-se o valor “0” aos valores que não existem na imagem.

Neste trabalho foi aplicada a convolução aperiódica pelo segundo método, pois esta é a forma utilizada nos algoritmos de detectores de bordas.

A operação de convolução unidimensional entre dois vetores A e B, denotada A*B, pode ser entendida como um conjunto de somas de produtos entre os valores de A e B, considerando B como uma máscara e após cada soma de produtos, é deslocado espacialmente de uma posição. Para o melhor entendimento desse conceito será mostrado a seguir, passo a passo, a convolução do vetor A= {0,1,2,3,2,1,0} com a máscara dada pelo vetor B= {1,3,-1}.

Inicialmente, o vetor B é alinhado com o primeiro valor de A e colocado em A*B na posição correspondente ao centro do conjunto B.

O resultado da convolução será 1, pois de acordo com a convolução aperiódica, os valores em branco fora de A assumem o valor zero.

A	0	0	1	2	3	2	1	0	
B	-1	3	1						
A*B		1							

O conjunto B é deslocado uma posição. O resultado da convolução A*B será 5.

A		0	1	2	3	2	1	0	
B		-1	3	1					
A*B		1	5						

O conjunto B é deslocado uma posição. O resultado da convolução A*B será 8.

A		0	1	2	3	2	1	0	
B			-1	3	1				
A*B		1	5	8					

O conjunto B é deslocado uma posição. O resultado da convolução A*B será 9.

A		0	1	2	3	2	1	0	
B				-1	3	1			
A*B		1	5	8	9				

O conjunto B é deslocado uma posição. O resultado da convolução A*B será 4.

A		0	1	2	3	2	1	0	
B					-1	3	1		
A*B		1	5	8	9	4			

O conjunto B é deslocado uma posição. O resultado da convolução A*B será 1.

A		0	1	2	3	2	1	0	
B						-1	3	1	
A*B		1	5	8	9	4	1		

O conjunto B é deslocado uma posição. O resultado da convolução A*B será -1.

A		0	1	2	3	2	1	0	0
B							-1	3	1
A*B		1	5	8	9	4	1	-1	

Outra aplicação é para o caso bidimensional, cuja a imagem processada é dada por uma matriz bidimensional mostrada na Matriz 2, correspondente ao mesmo conjunto A do exemplo anterior e a máscara mostrada na Matriz 3, correspondente ao conjunto B.

$$\begin{pmatrix} 1 & 0 & 1 & 3 & 2 & 10 & 0 \\ 4 & 1 & 0 & 0 & 2 & 3 & 4 \\ 5 & 1 & 2 & 0 & 2 & 3 & 10 \\ 40 & 5 & 7 & 15 & 10 & 5 & 2 \\ 4 & 2 & 1 & 1 & 1 & 4 & 8 \\ 3 & 3 & 5 & 1 & 7 & 51 & 10 \\ 1 & 5 & 2 & 7 & 1 & 5 & 1 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 0 \\ 1 & 5 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Matriz 2 **Matriz 3**

Após a máscara B ter sido convoluida com a matriz principal, tanto na horizontal quanto na vertical, também estará varrendo todos os pontos da matriz A deslocando-se ao longo de cada linha, varrendo da esquerda para a direita e também de cima para baixo, até que se possa ser processado o último elemento da matriz imagem. Deste modo, após toda varredura, o resultado obtido será armazenado em uma outra matriz de mesmas dimensões

que a imagem original processada, a qual é mostrada na Matriz 4. (GONZALEZ; WOODS, 2002).

$$\begin{pmatrix} 10 & 4 & 5 & 17 & 34 & 33 & 4 \\ 18 & 18 & 19 & 25 & 35 & 58 & 10 \\ 75 & 58 & 93 & 73 & 44 & 33 & 2 \\ 23 & 20 & 23 & 27 & 85 & 56 & 8 \\ 30 & 32 & 25 & 89 & 281 & 110 & 10 \\ 30 & 27 & 39 & 24 & 78 & 20 & 1 \\ 5 & 2 & 7 & 1 & 5 & 1 & 0 \end{pmatrix}$$

Matriz 4

2.5. 3. Filtro por derivada

O cálculo da média dos *pixels* sobre uma região borra os detalhes de uma imagem, e como esse cálculo é análogo à integração, a diferenciação vai ter o efeito oposto na imagem.

Para o processamento de imagem, o método mais comum de diferenciação é o gradiente. Para uma função f(x, y) o gradiente de f nas coordenadas (x, y) é definido como o vetor mostrado na equação (2.4).

$$\nabla \mathrm{f} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{2.4}$$

A magnitude desse vetor, apresentada na equação (2.5), é a base para várias abordagens de diferenciação de imagens. Na Matriz 5, mostra-se os z's que denotam os valores dos níveis de cinza. A equação (2.5) pode ser aproximada do ponto z5 (ponto central) de várias maneiras. A mais simples é combinando as diferenças (z5 – z8) na direção do eixo x e (z5–z6) na direção do eixo y mostrada na equação (2.6). A equação (2.7) mostra o uso dos valores absolutos ao invés de usar quadrados e raízes quadradas, obtendo-se resultados similares.

$$\begin{bmatrix} z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 \\ z_7 & z_8 & z_9 \end{bmatrix}$$

Matriz 5

$$\nabla f = mag(\nabla \mathrm{f}) = \left[\left(\frac{\partial f}{\partial x} \right)^2 + \left(\frac{\partial f}{\partial y} \right)^2 \right]^{1/2} \quad (2.5)$$

$$\nabla f = \left[(z_5 - z_8)^2 + (z_5 - z_6)^2 \right]^{1/2} \quad (2.6)$$

$$\nabla f = |z_5 - z_8| + |z_5 - z_6| \quad (2.7)$$

Outra maneira para a aproximação da Equação (2.5) é usar as diferenças cruzadas, conforme mostrado na equação (2.8), ou usar os valores absolutos como mostrado na equação (2.9).

$$\nabla f = \left[(z_5 - z_9)^2 + (z_6 - z_8)^2 \right]^{1/2} \quad (2.8)$$

$$\nabla f = |z_5 - z_9| + |z_6 - z_8| \quad (2.9)$$

As equações (2.6) ou a (2.7) e (2.8) ou (2.9) podem ser implementadas através do uso de máscaras de tamanho 2 x 2. Por exemplo, a equação (2.9) pode ser implementada tomando-se o valor absoluto das respostas das duas máscaras mostradas na Figura 2.8 e somando-se os resultados. Essas máscaras são chamadas de operadores cruzados de *Roberts*.

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

Figura 2. 8. Máscara de *Roberts*.

Usando uma máscara 3x3 para a implementação, é feita uma aproximação da equação (2.7) ainda no ponto z5, como é mostrada na equação (2.10).

$$\nabla f = |(z_7 + z_8 + z_9) - (z_1 + z_2 + z_3)| + |(z_3 + z_6 + z_9) - (z_1 + z_4 + z_7)| \quad (2.10)$$

A diferença entre a terceira e a primeira linha da região 3 x 3 aproxima a derivada na direção do eixo x, e a diferença entre a terceira e a primeira coluna da região 3 x 3 aproxima a derivada na direção do eixo y. As máscaras chamadas operadores de *Prewitt,* mostradas na

Figura 2.9, podem ser usadas para implementar a equação (2.10). Outro par de máscaras para a aproximação da magnitude do gradiente, chamadas operadores de *Sobel,* é mostrado na Figura 2.10.

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Figura 2. 9. Máscaras de *Prewitt.*

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Figura 2. 10. Máscaras de *Sobel.*

2.5. 4. Operadores de gradiente

As primeiras diferenciações no processamento de imagem digital são implementadas utilizando-se a magnitude do gradiente para a função f(x, y); o gradiente das coordenadas (x, y) é dado pelo vetor descrito na equação (2.11).

$$\nabla \mathrm{f} = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \tag{2.11}$$

A partir de análise vetorial, o vetor gradiente aponta na direção de mudança mais rápida de f na posição (x, y). Em relação à detecção de bordas, a magnitude desse vetor, chamado de gradiente e denotado por $\nabla \mathrm{f}$, é descrita na equação (2.12).

$$\nabla f = mag(\nabla \mathrm{f}) = \left[G_x^{\ 2} + G_y^2 \right]^{1/2} \tag{2.12}$$

Essa quantidade equivale à maior taxa de aumento de f(x, y) por unidade de distância na direção de $\nabla \mathrm{f}$, e aproxima o gradiente com valores absolutos, descritos na equação (2.13), a qual é mais fácil de ser implementada, particularmente em *hardware* dedicado.

$$\nabla f \approx |G_x| + |G_y| \tag{2.13}$$

A partir das equações (2.11) e (2.12), o cálculo do gradiente baseia-se na obtenção das derivadas parciais $\partial f / \partial x$ e $\partial f / \partial y$ na posição de cada *pixel*. A derivação pode ser implementada de maneira digital de formas diferentes. Por outro lado, os operadores de *Sobel* fornecem os efeitos de diferenciação e de suavização, os quais constituem uma característica atrativa dos operadores de *Sobel*. Tais efeitos destacam mais as bordas, mas também aumentam o ruído da imagem.

Tem-se a partir da Figura 2.11, as derivadas baseadas nas máscaras do operador de *Sobel*, as quais são descritas nas equações (2.14) e (2.15).

$$\begin{bmatrix} z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 \\ z_7 & z_8 & z_9 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

(a) **(b)** **(c)**

Figura 2. 11. (a) Região de uma imagem 3 x 3; (b) Máscara usada para o cálculo de G_x no ponto central da região 3 x 3; (c) Máscara usada para o cálculo de G_y no ponto central da região 3 x 3.

$$G_x = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3) \tag{2.14}$$

$$G_y = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7) \tag{2.15}$$

Nestas equações os z's são os níveis de cinza dos *pixels* sobrepostos pelas máscaras em qualquer posição da imagem, o cálculo do gradiente na posição central da máscara utiliza a equação (2.12) ou a (2.13), fornecendo o valor do gradiente. Para obter o próximo valor, as máscaras são deslocadas para o próximo *pixel* e o procedimento é repetido; portanto, quando completado para todas as possíveis posições, o resultado obtido é uma imagem de gradiente do mesmo tamanho da imagem original. As operações de máscara nas bordas de uma imagem são implementadas utilizando-se as vizinhanças parciais apropriadas. (GONZALEZ; WOODS, 2002).

Na Figura 2.12 visualiza-se a imagem original tirada de ovos e na Figura 2.13, a imagem processada pelo operador de *Sobel* no algoritmo sobel.m, o qual será explicado no capítulo 3.

Figura 2. 12. Imagem original.

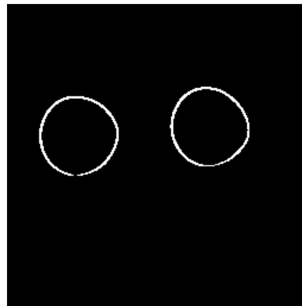


Figura 2. 13. Imagem com a aplicação do operador de *Sobel* processada no algoritmo sobel.m no MATLAB.

2. 6. Segmentação de imagens

A segmentação da imagem é responsável por subdividir a imagem em partes ou objetos constituintes. Para se definir até qual nível essa subdivisão deve ser realizada, depende da análise do problema a ser resolvido, e deve terminar quando os objetos de interesse na aplicação tiverem sido isolados.

Os algoritmos de segmentação para imagens monocromáticas são baseados em duas propriedades básicas de valores de níveis de cinza a saber:

Descontinuidade: A idéia é particionar a imagem com base em mudanças bruscas nos níveis de cinza; funciona como um detector de mudanças bruscas. As principais áreas de interesse, discutidas nessa categoria são a detecção de pontos isolados, de linhas e de bordas em uma imagem;

Similaridade: Baseia-se em limiarização, trabalhando com o crescimento de regiões, divisão e fusão de regiões.

Há três tipos básicos de descontinuidades em imagens digitais, são eles: pontos, linhas e bordas, e a maneira mais utilizada, neste trabalho, pela procura de detecção de descontinuidade foi feita através da varredura da imagem utilizando uma máscara.

Visualiza-se na Matriz 6, um exemplo de uma máscara 3x3. O procedimento do cálculo é a soma dos produtos dos coeficientes pelos níveis de cinza que se encontram na matriz pela região que é englobada pela máscara. Ou seja, a resposta da máscara em qualquer ponto da imagem ou matriz é descrito na equação (2.16), em que zi é o nível que corresponde à imagem em cinza, e wi é o coeficiente da máscara.(GONZALEZ; WOODS, 2002).

$$\begin{bmatrix} w1 & w2 & w3 \\ w4 & w5 & w6 \\ w7 & w8 & w9 \end{bmatrix}$$

Matriz 6

$$R = w1z1 + w2z2 + ... + w9z9 = \sum_{i=1}^{9} wi.zi \qquad (2.16)$$

A resposta à aplicação da máscara dá-se ao ponto central da mesma, quando a máscara é posicionada em um *pixel* da borda; a resposta é computada, utilizando-se a vizinhança parcial apropriada.

2.6. 1. Detecção de Pontos

A detecção de pontos em uma imagem pode ser obtida usando-se máscara, e um ponto é detectado na posição da máscara se: **|R| > T**, onde T é um limiar não-negativo e R é descrito pela equação (2.16). De modo geral, o que essa equação faz é medir e ponderar as diferenças entre o ponto central e seus vizinhos, ou seja, o nível de cinza de um ponto isolado será diferente do nível de cinza de seus vizinhos.

Para detecção de pontos, uma máscara 3x3 é usada para a detecção de pontos isolados, a partir de um fundo constante. Na Figura 2.14 visualiza-se a máscara de pontos isolados.

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Figura 2. 14. Máscara usada para a detecção de pontos isolados, a partir de um fundo constante.

2.6. 2. Detecção de Linhas

Na Figura 2.15, visualizam-se máscaras para detecções de linhas em uma imagem que podem ser feitas nas direções horizontal, vertical e também nos ângulos de + 45º e - 45º.

Para a detecção de linhas horizontais é aplicado uma máscara orientada horizontalmente, ou seja, as linhas da imagem, que são horizontais, são totalmente destacadas em relação às linhas verticais apresentadas na mesma imagem varrida.

O mesmo ocorre para aplicações em imagens usando máscara orientada verticalmente, assim como para +45º ou -45º.

$$\begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix} \quad \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}$$

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

Figura 2. 15. (a) Máscara para linha horizontal. (b) Máscara para linha a +45º. (c) Máscara para linha vertical. (d) Máscara para linha a -45º.

2.6. 3. Detecção de Bordas

Uma borda é a fronteira entre duas regiões de níveis de cinza diferentes. Para a detecção e realce de bordas, é aplicado o filtro por derivada utilizando-se máscaras de convolução, também chamadas de operadores de 2x2 ou de 3x3.

Alguns exemplos destas máscaras são os operadores de *Roberts*, *Prewitt* e *Sobel*, presentes na Tabela 2.1.

Tabela 2. 1. Operadores utilizados para estimar a amplitude do gradiente através de uma borda.

Operador	Vertical	Horizontal
Roberts	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$
Prewitt	$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$
Sobel	$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

Para exemplificar a aplicação das máscaras de *Roberts*, *Prewitt* e *Sobel* utilizou-se foto de ovos reais que pode ser visualizada na Figura 2.16.

Figura 2. 16. Imagem Original.

A partir daí obtêm-se as imagens, com as bordas detectadas para cada operador estudado.

No operador de *Roberts* calcula as diferenças de níveis de cinza por diferenças cruzadas. Este operador é o menos indicado para este projeto, pois o resultado contém muitos ruídos, e como as máscaras têm dimensão 2X2, ele não destaca muito a visualização da borda quando comparado com às das outras máscaras estudadas. Na Figura 2.17 é possível visualizar a detecção de borda com o operador de *Roberts* aplicado na imagem original.

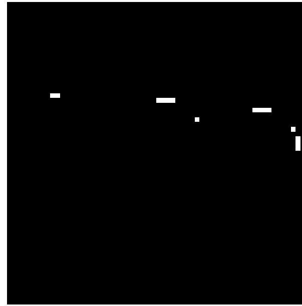


Figura 2. 17. Detecção de Bordas com o operador de *Roberts*.

Diferente do operador de *Roberts*, o operador *Prewitt* diferencia nas direções vertical e horizontal. Por ser uma máscara com dimensão 3X3, há uma melhora significativa na detecção das bordas em relação à máscara de *Roberts*. Na Figura 2.18, visualiza-se a detecção de borda com o operador de *Prewitt* aplicado na imagem original.

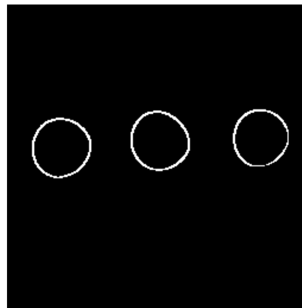


Figura 2. 18. Detecção de Bordas com o operador de *Prewitt*

O operador de *Sobel* é similar ao operador de *Prewitt*, porém com mais peso nos pontos próximos ao *pixel* central. Por esse motivo, a máscara de *Sobel* obtém as bordas mais destacadas em relação ao operador de *Prewitt*. Comparando as máscaras analisadas, a de *Sobel* mostrou melhor destaque das bordas. Na Figura 2.19, visualiza-se a detecção de borda com o operador de *Sobel* aplicado na imagem original.

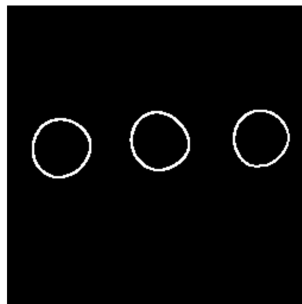


Figura 2. 19. Detecção de Bordas com o operador de *Sobel*.

2. 7. Limiarização

A limiarização é de grande importância para as abordagens de segmentação de imagens.

Um dos métodos para definir o limiar é o do Vale, o qual é definido pela escolha do ponto de limiar. Em que o limiar é conhecido como a variável T, e quando o pixel for maior que o limiar ele assume o valor de nível lógico 1 e quando menor ele assume o valor de nível lógico 0.

Portanto, uma imagem limiarizada g(x, y) é definida na equação (2.17):

$$g(x,y) = \begin{cases} 1 & se \quad f(x,y) > T \\ 0 & se \quad f(x,y) \leq T \end{cases} \tag{2.17}$$

Assim, em nível de rotulação, os *pixels* tendo valor 1 ou com qualquer outro nível de cinza conveniente correspondem aos objetos, enquanto os que receberem 0 como valor corresponderão ao fundo da imagem.(GONZALEZ; WOODS, 2002).

2. 8. Transformada de *Hough*

A transformada de *Hough* foi desenvolvida, no início dos anos 40, por Paul Hough. É uma técnica de reconhecimento, utilizada em imagens digitais, para que sejam parametrizadas facilmente, ou seja, que possuam uma equação com fórmula conhecida como retas, círculos e elipses, entre outras.

Para realizar a aplicação da transformada de Hough é necessário aplicar o processo de detecção de borda e de limiarização, o método sugerido de detecção de borda é o método de *Canny*, porém neste projeto foi aplicado os operadores de *Roberts*, *Prewitt* e *Sobel*.

Neste trabalho, os objetos usados para a detecção das bordas e contagem dos objetos foram os ovos, portanto, no formato de elipses.(SILVA, 2010).

CAPÍTULO 3

Estudo dos Detectores de Borda Empregando o MATLAB

Neste capítulo apresenta-se a aplicação da detecção de bordas na identificação de ovos em uma esteira e na comparação entre os operadores de bordas aplicados e estes operadores aplicados foram os de *Roberts*, *Prewitt* e *Sobel* utilizando como ambiente computacional o MATLAB. Para cada um dos operadores utilizados desenvolveu-se três algoritmos denominados respectivamente de roberts.m, prewitt.m e sobel.m. Foram utilizadas funções disponíveis no MATLAB.

Para avaliar os algoritmos desenvolvidos utilizaram-se imagens reais obtidas com a máquina fotográfica de marca *General Electric Company* modelo *Digital Câmera X5*, tiradas em ambiente doméstico. Colocou-se os ovos sobre uma cartolina preta para simular que os ovos estavam em cima de uma esteira.

O MATLAB também foi utilizado no redimensionamento e na conversão de imagens coloridas para imagens em níveis de cinza. Obteve-se dessa forma, arquivos com formato txt para poder trabalhar com as aplicações em C ANSI e VHDL. Na Figura 3.1, visualiza-se uma das imagens dos ovos obtida com a máquina fotográfica.

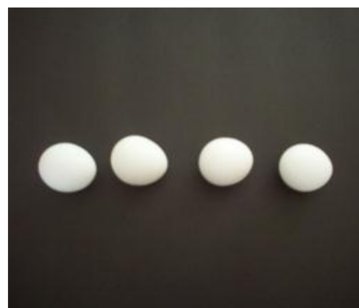


Figura 3. 1. Imagem original.

Para redimensionar a imagem, converter a imagem em níveis de cinza e o arquivo do formato bmp para o formato txt, foram utilizados os seguintes comandos próprios do MATLAB:

```
E = imresize(I, [256 256]);

B=rgb2gray(E);
```

```
dlmwrite('niveis_cinza.txt',B,' ');
```

Sendo:

I: Imagem original obtida com máquina fotográfica;

E: Imagem I (original) redimensionada;

B: Imagem E em níveis de cinza;

níveis_cinza.txt: Arquivo txt com a matriz imagem B gravada;

Com a função imresize redimensiona-se a imagem I (original) no tamanho desejado. Em alguns dos estudos realizado a imagem original foi redimensionada para 256x256 *pixels*. Na Figura 3.2, visualiza-se a imagem I (original) redimensionada.

Figura 3. 2. Imagem I redimensionada.

A instrução rgb2gray transforma a imagem E (imagem original redimensionada), que está colorida (R (*Red*) G (*Green*) b (*Blue*)), ou seja, nas cores primárias, em uma imagem em níveis de cinza (*gray*). Na Figura 3.3, visualiza-se a imagem B (imagem original redimensionada e em níveis de cinza).

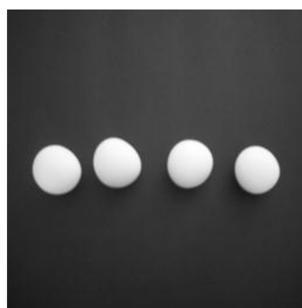


Figura 3. 3. Imagem em níveis de cinza e redimensionada.

A instrução dlmwrite grava dados de uma matriz em um arquivo com determinado

delimitador, no caso grava a matriz imagem B no arquivo níveis_cinza.txt, utilizando como delimitador um espaço em branco. Na Figura 3.4, visualiza-se os números que representam os pixels da imagem em níveis de cinza no formato txt.

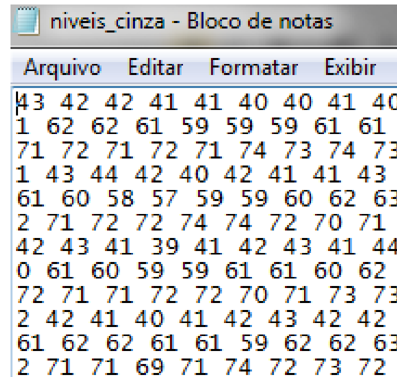


Figura 3. 4. Imagem em níveis de cinza no formato txt.

Pode-se notar que o delimitador (‘ ’) significa o espaço entre os números da matriz, sem ele, os números ficariam entre vírgulas, como visto na Figura 3.5. O espaço é colocado para o arquivo ficar de acordo com a leitura do vetor de números inteiros utilizado na linguagem C ANSI.

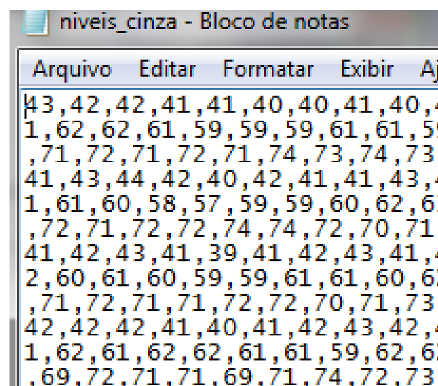


Figura 3. 5. Imagem em níveis de cinza sem o delimitador.

3. 1. Aplicação dos operadores de bordas utilizando como ambiente computacional o MATLAB.

O estudo dos algoritmos de detecção de borda no ambiente computacional MATLAB têm uma única estrutura. No estudo dos algoritmos alterou-se somente as máscaras de *Roberts*, *Prewitt* e *Sobel*. Nessa estrutura realizou-se o cálculo de convolução entre a imagem e a máscara aplicada, e o cálculo da limiarização, ambos apresentados no capítulo 2:

```
GX(i,j) = (abs(hor (1,1) * double(B (i-1, j-1)) + hor(1,2) * double(B (i-1, j)) + hor(1,3) * double(B (i-1, j+1)) + hor(2,1) * double(B (i, j-1)) + hor(2, 2)  * double(B (i, j)) + hor(2,3)  * double(B (i, j+1)) + hor(3,1) * double(B (i+1,j-1)) + hor(3, 2) * double(B (i+1, j)) + hor(3,3)  * double(B (i+1, j+1))));

GY(i,j) = (abs(ver(1,1) * double(B (i-1, j-1)) + ver(1,2) * double(B (i-1, j)) + ver(1,3) * double(B (i-1, j+1)) + ver(2,1)  * double(B (i, j-1)) + ver(2,2)  * double(B (i, j)) + ver(2,3)  * double(B (i, j+1)) + ver(3,1) * double(B (i+1, j-1)) + ver(3,2) * double(B (i+1, j)) + ver(3,3)  * double(B(i+1,j+1))));

 G(i,j) = (abs((double(GX(i,j)) + (double(GY(i,j))))));

if (G(i,j)>254)

   H(i,j) = (1);

else

  H(i,j) = (0);

end
```

Sendo:

GX(i,j): A variável de convolução no eixo horizontal;

GY(i,j): A variável de convolução no eixo vertical;

G(i,j): A soma das variáveis de convolução;

(G(i,j)>254): Este comando para fazer a limiarização;

H(i,j): Variável que recebe o resultado da limiarização.

Para diferenciar a aplicação entre os operadores de bordas, *Roberts*, *Prewitt* e *Sobel*, explicados no capítulo 2, alteram-se somente as máscaras. As máscaras são representadas pelos vetores denominados: hor e ver. Os algoritmos serão explicados respectivamente nas seções 3.1.1, 3.1.2 e 3.1.3.

3.1. 1. Operador de *Roberts*.

Para o operador de *Roberts* foi criado o algoritmo roberts.m, alterando-se somente a máscara na estrutura apresentada na seção 3.1. Para poder usar esta mesma estrutura foi necessário alterar a máscara de *Roberts* de uma matriz 2x2 para uma matriz 3x3, colocando o

número ‘0’ para completar a matriz 2x2. Essa alteração é mostrada a seguir:

```
%Máscaras do operador Roberts

hor =  [0 0 0;
        0 1 0;
        0 0 -1];
ver =  [0 0 0;
       0 -1 0;
        0 0 1];
```

Na Figura 3.6, visualiza-se a imagem em níveis de cinza processada pelos comandos: imresize, rgb2gray e dlmwrite. Na Figura 3.7, visualiza-se o resultado da imagem processada pelo operador de *Roberts*.

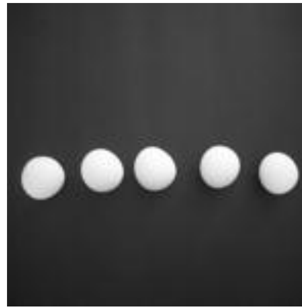


Figura 3. 6. Imagens em níveis de cinza.

Figura 3. 7. Imagem processada com o operador de *Roberts*.

O resultado obtido com a máscara de *Roberts* não gerou uma visualização satisfatória para este estudo com as imagens de ovos.

3.1. 2. Operador de *Prewitt*.

Para o operador de *Prewitt* foi criado o algoritmo prewitt.m e assim como o operador de *Roberts*, alterou-se somente a máscara na estrutura explicada na seção 3.1. A máscara utilizada é mostrada a seguir:

```
%Máscaras do operador Prewitt

hor =  [-1 -1 -1;
         0  0  0;
         1  1  1];
ver =  [1  0 -1;
        1  0 -1;
        1  0 -1];
```

Na Figura 3.8, visualiza-se o resultado aplicando o operador de *Prewitt*.

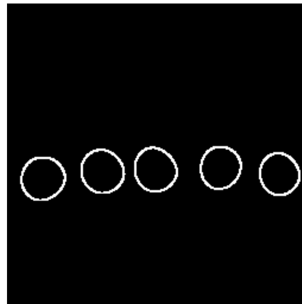


Figura 3. 8. Imagens com o operador de *Prewitt*.

O resultado obtido com a máscara de *Prewitt* foi melhor do que o resultado da máscara de *Roberts* em relação às imagens estudadas.

3.1. 3. Operador de *Sobel*.

Para o operador de *Sobel*, foi criado o algoritmo sobel.m e, assim como no operador de *Roberts* e de *Prewitt*, alterou-se somente a máscara na estrutura explicada na seção 3.1. Essa alteração é mostrada a seguir:

```
%Máscaras do operador Sobel

hor =  [-1 -2 -1;
         0   0  0;
         1   2  1];
ver =  [-1  0  1;
        -2  0  2;
        -1  0  1];
```

Na Figura 3.9, visualiza-se a imagem em que foi aplicada o operador de *Sobel*.

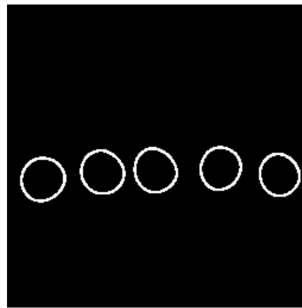


Figura 3. 9. Imagens com o operador de *Sobel.*

A máscara de *Sobel* mostrou um melhor resultado em relação às outras máscaras discutidas anteriormente para o caso estudado. Um dos motivos é que esta máscara tem um peso maior no pixel central do que a máscara de *Prewitt*.

3. 2. Visualização das imagens processadas em outros ambientes

Os algoritmos estudados em MATLAB também foram programados em linguagem C, para serem implementados no processador NIOS executando sob o sistema uClinux. Foram também descritos em VHDL visando a implementação em *hardware* gerando, dessa forma, um cenário adequado às comparações.

O ambiente MATLAB foi utilizado como ferramenta para visualizar as imagens processadas em programas desenvolvidos em linguagem C ou descritos na linguagem de descrição de hardware VHDL.

Cabe lembrar que o foco da pesquisa é avaliar os algoritmos de detecção de borda, por isso utilizou-se os recursos do MATLAB para a visualização das imagens processadas.

No ambiente computacional Dev-C++ programou-se os operadores de borda *Roberts*, *Prewitt* e *Sobel*, dando origem aos arquivos roberts.c, prewitt.c e sobel.c, respectivamente. Estes programas serão explicados com mais detalhes na seção 4.1.

Estes programas geram como resultado arquivos no formato txt, conforme o modelo apresentado na Figura 3.10.

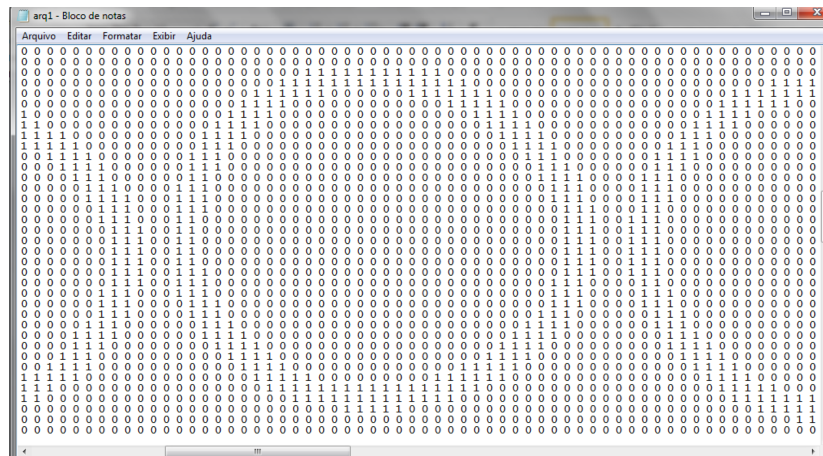


Figura 3. 10. Arquivo com formato txt produzido por programa desenvolvidos na linguagem C.

Com alguns comandos do MATLAB foi possível transforma o arquivo arq1.txt em uma imagem, sendo possível a visualização das bordas em destaque. Estes comandos são descritos a seguir:

```
a = dlmread(‘arq1.txt’);

imwrite(a, ‘resultado.bmp’, ‘bmp’);

imshow resultado.bmp
```

Sendo:

a: A imagem a ser visualizada;

arq1.txt: O arquivo txt obtido após a execução dos algoritmos robert.c, prewitt.c e sobel.c;

dlmread: Comando para ler o arquivo txt e deixá-lo com formato de matriz, possível para visualização da imagem.

imwrite: Comando para gravar a imagem a no formato bmp como resultado.bmp;

resultado.bmp: Imagem gerada;

imshow: Comando para visualizar a imagem.

Na Figura 3.11 visualiza-se a imagem em níveis de cinza. Neste estudo também houve o pré-processamento de imagem pelos comandos imresize, rgb2gray e dlmwrite.

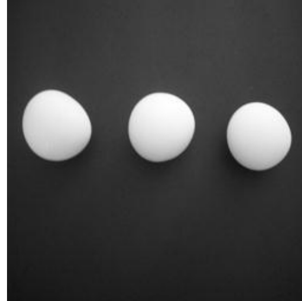


Figura 3. 11. Imagem em níveis de cinza.

Após o uso destes comandos, o resultado é convertido para a extensão bmp, tendo como resposta uma imagem com as bordas destacadas, a qual visualiza-se na Figura 3.12.

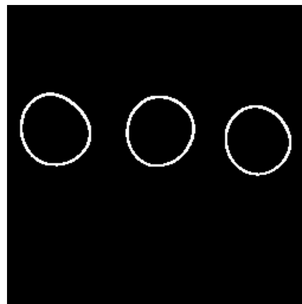


Figura 3. 12. Imagem processada por um programa em C e visualizada no MATLAB.

Para realizar a implementação no FPGA e criar um sistema com tecnologia embarcada com os detectores de bordas *Roberts*, *Prewitt* e *Sobel*, foram criados os algoritmos na linguagem C ANSI, nios_roberts.c, nios_prewitt.c e nios_sobel.c, os quais serão explicados com mais detalhes na seção 4.2.

A implementação desses operadores em FPGA obteve como resultado arquivos com extensão txt. Visualiza-se um exemplo de um dos resultados com a aplicação do operador de *Sobel*, na Figura 3.13.

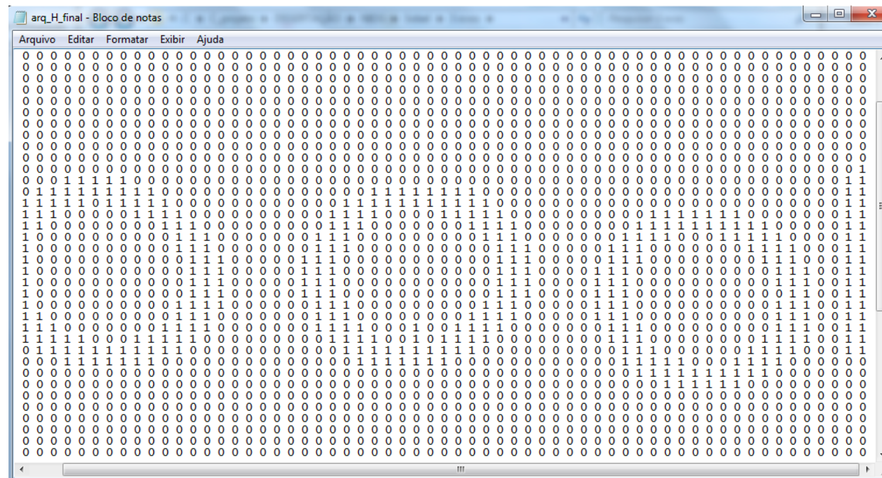


Figura 3. 13. Arquivo com formato txt.

Para visualizar esses arquivos com o formato txt e obter as imagens, utilizaram-se os comandos dlmread, imwrite e imshow. Na Figura 3.14, apresenta-se a imagem processada com o programa nios_sobel.c e visualizada com os comandos do MATLAB.

Figura 3. 14. Imagem processada com o programa nios_sobel.c.

No ambiente QUARTUS também gerou-se os operadores de bordas de *Roberts*, *Prewitt* e *Sobel* descritos com a linguagem VHDL, sendo criados os algoritmos roberts.vhd, prewitt.vhd e sobel.vhd respectivamente. Os modelos VHDL e a simulação serão explicados com mais detalhes no capítulo 5.

Para a visualização das imagens produzidas pela simulação dos modelos VHDL foram necessárias algumas modificações nos arquivos.

O resultado da simulação no QUARTUS foi direcionada para um arquivo, cujo formato esta apresentado na Figura 3.15. Trata-se de um recurso disponível no ambiente QUARTUS, que envia o resultado da simulação para um arquivo texto com extensão tbl. Nota-se na Figura 3.15 três colunas diferentes, a primeira coluna mostra o tempo da

simulação, a segunda coluna mostra a transição do nível de *clock* e a terceira coluna mostra a simulação gerada.

Nível de transição do clock

Tempo

Simulação

```
Arquivo   Editar   Formatar   Exibir
 213130.0> 0 = 1
 213135.0> 1 = 1
 213140.0> 0 = 1
 213145.0> 1 = 1
 213150.0> 0 = 1
 213155.0> 1 = 1
 213160.0> 0 = 1
 213165.0> 1 = 1
 213170.0> 0 = 1
 213175.0> 1 = 1
213178.91> 1 = 0
 213180.0> 0 = 0
 213185.0> 1 = 0
 213190.0> 0 = 0
 213195.0> 1 = 0
 213200.0> 0 = 0
```

Figura 3. 15. Arquivo tbl gerado pela simulação do modelo VHDL.

O MATLAB também foi utilizado para transformar o arquivo contendo o resultado da simulação em um formato apropriado para a visualização da imagem processada.

O programa vhdl_mat.m substitui as informações contidas no arquivo com extensão .tbl relacionadas ao nível de transição de subida do *clock*, que são desnecessárias para a visualização da imagem pelo número 2. Os detalhes sobre esta substituição será abordado no capítulo 5. O arquivo alterado pode ser visualizado na Figura 3.16.

```
Arquivo   Editar   Formatar   Exibir
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1
0
2
0
2
0
2
0
```

Figura 3. 16. Arquivo txt alterado a partir do resultado da simulação.

Para a visualização da imagem simulada foram necessárias mais algumas modificações no formato do arquivo. Essas modificações referem-se a substituição do símbolo ‘2’ por espaço vazio e o redimensionamento do tamanho das linhas. Este ajuste no formato foi realizado pelo programa quartus_c.c. O arquivo no formato correto para ser visualizado pelo MATLAB é apresentado na Figura 3.17.

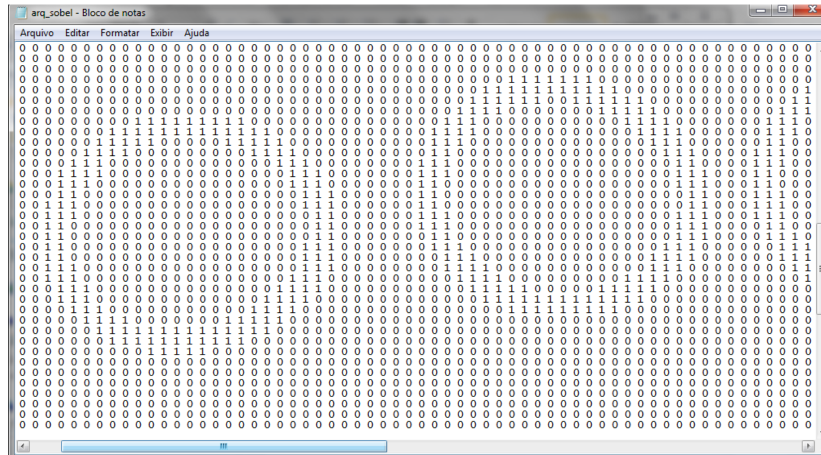


Figura 3. 17. Arquivo contendo o resultado da simulação, formatado para a visualização no MATLAB.

Na Figura 3.18, visualiza-se a imagem em níveis de cinza utilizada para ser processada pelos detectores de bordas no programa descrito pela linguagem VHDL. Neste estudo também houve o pré-processamento da imagem pelos comandos imresize, rgb2gray e dlmwrite.

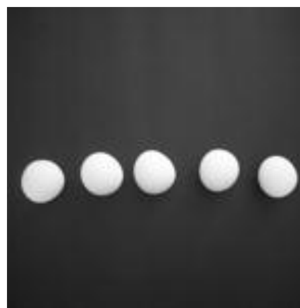


Figura 3. 18. Imagem em níveis de cinza.

A Figura 3.19 apresenta a imagem processada pelo modelo VHDL e visualizada pelo MATLAB, após a transformação no formato dos arquivos.

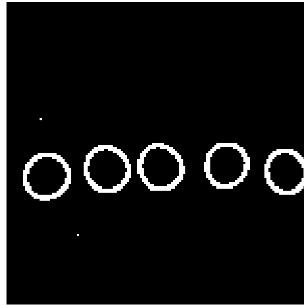


Figura 3. 19. Imagem processada pelo modelo VHDL.

3. 3 – Transformada de *Hough*

Um dos passos realizados nesta pesquisa é a contagem de ovos contidos em uma esteira. Por isso estuda-se também a transformada de *Hough*. Esta transformada pode ser utilizada em vários formatos, como por exemplo, para formas circulares, parábolas ou elipses, Neste estudo, por causa do formato do ovo, foi utilizada a forma de elipses.

O ambiente MATLAB disponibiliza o algoritmo hough.m, porém utiliza a operação de *Canny*. Desta forma o algoritmo hough.m foi modificado para ser utilizado com os operadores de bordas deste trabalho. Portanto o algoritmo hough.m foi modificado para o algoritmo teste_hough.m, essa modificação é visualizada abaixo:

hough.m

```
RGB = imread('gantrycrane.png');

I  = rgb2gray(RGB);

BW = edge(I,'canny');
```

teste_hough.m

```
A=input('Digite o nome:  ','s')

[BW Map]=imread(A);
```

A função de *Hough* disponibilizada pelo MATLAB é apresentada a seguir:

```
[H,theta,rho] = hough(BW);
```

Sendo:

BW – A imagem binária, o resultado da operação de detecção de bordas;

H – A matriz que a função de *Hough* retorna;

Theta e rho – São os valores sobre os quais a matriz de transformação de *Hough* foi gerado.

O algoritmo teste_hough.m, modificado a partir do algoritmo hough.m, quando aplicado à Figura 3.20 gera o gráfico apresentado na Figura 3.21.

Pode-se notar que na esteira apresentada na Figura 3.20 contém 5 ovos e que o algoritmo teste_hough.m apresenta um gráfico, conforme a Figura 3.21, indicando a existência de 5 objetos na imagem processada.

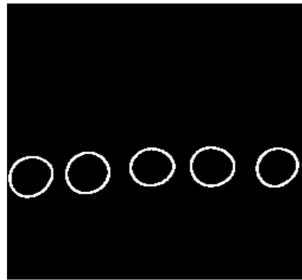


Figura 3. 20. Imagem resultado de 5 ovos processada com o operador de *Sobel.*

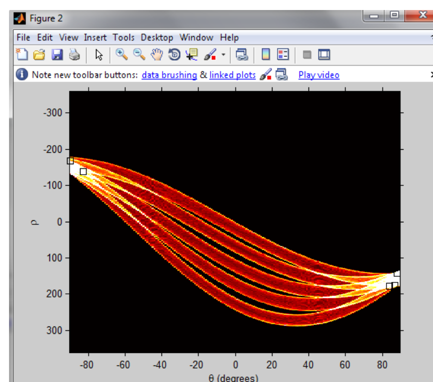


Figura 3. 21. Gráfico de *Hough* para a imagem da Figura 3.20.

Para obter uma comparação foi aplicado à Figura 3.22, que contém 3 ovos, o algoritmo teste_hough.m, gerando o gráfico apresentado na Figura 3.23. Pode-se notar que o gráfico apresenta indica 3 objetos conforme a Figura 3.22 que contém 3 ovos.

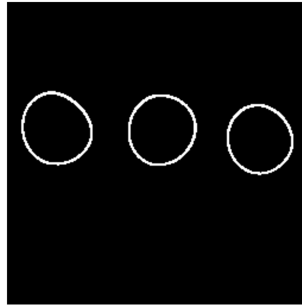


Figura 3. 22. Imagem resultado de 3 ovos processada com o operador de *Sobel.*

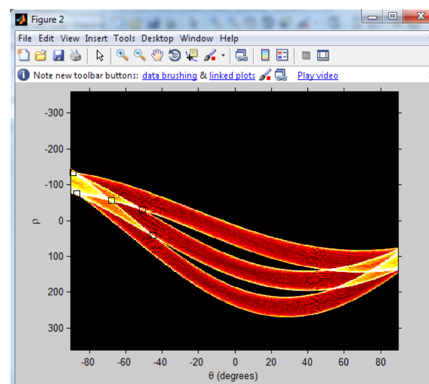


Figura 3. 23. Gráfico de *Hough* para a imagem da Figura 3.22.

Apresenta-se no próximo capítulo os estudos realizados usando a implementação de detecção de bordas empregando as máscaras de *Roberts*, *Prewitt* e *Sobel* no processador NIOS (FPGA).

CAPÍTULO 4

Estudo dos Detectores de Bordas Utilizando a Linguagem C

Neste capítulo, apresenta-se a aplicação da detecção de bordas na identificação de ovos em uma esteira. Os operadores de *Roberts*, *Prewitt* e *Sobel*. Foram programados em linguagem C utilizando o ambiente computacional o Dev-C++. Para cada um dos operadores, desenvolveu-se um algoritmo denominado respectivamente de roberts.c, prewitt.c e sobel.c. Na visualização das imagens processadas utilizou-se o ambiente MATLAB, conforme descrito no capítulo 3.

4. 1. Algoritmos de detecção de borda programados em linguagem C.

Os algoritmos roberts.c, prewitt.c e sobel.c foram programados utilizando-se de uma mesma estrutura. Altera-se somente as máscaras para cada um dos algoritmos.

As imagens sob análise foram pré-processadas utilizando o MATLAB, conforme descrito na seção 3.2. O arquivo denominado níveis_cinza.txt foi gerado pelo MATLAB e o arquivo gerado após o processamento de imagem é denominado de arq1.txt.

Parte do código da estrutura para o cálculo de convolução entre a imagem em níveis de cinza e as máscaras estudadas é descrita a seguir:

```
for(l = 0;l < x;l++)
{
  for(c = 0;c < x; c++)
   {
        GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) + ((MX[0][2]) * (mat[l][c+2])) + ((MX[1] [0]) * (mat[l+1] [c])) + ((MX[1] [1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) + ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) + ((MX[2][2]) * (mat[l+2][c+2]));
   }
}
```

GX [l] [c] representa a variável de convolução no eixo horizontal;

MX[0] [0] até MX[2] [2] representam a máscara no eixo horizontal;

```
GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) + ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) + ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) + ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) + ((MY[2][2])*(mat[l+2][c+2]));
```

GY [l] [c] representa a variável de convolução no eixo vertical;

MY [0] [0] até MY [2] [2] representam a máscara no eixo vertical;

```
if (GX[l][c]<0)
{
  GX[l][c] =GX[l] [c]*(-1);
}
 if (GY[l][c]<0)
{
  GY[l][c] =GY[l] [c]*(-1);
}
  G[l][c] = (GX[l] [c]) + (GY[l] [c]);
```

G[l] [c] representa a soma das variáveis.

A limiarização é realizada através do código a seguir:

```
if (G[l][c]<0)
{
  G[l] [c] =G[l] [c]*(-1);
}
if (G[l][c]>254)
{
  H[l] [c] = (1);
}
```

```
else
{
 H[l] [c] = (0);
}
```

H[l] [c] representa a variável que recebe o resultado da limiarização.

Para diferenciar a aplicação entre os operadores de bordas, *Roberts*, *Prewitt* e *Sobel*, explicados no capítulo 2, alteram-se somente as máscaras. Os algoritmos implementados são apresentados nas seções 4.1.1, 4.1.2 e 4.1.3.

4.1. 1. Operador de *Roberts*

Para o operador de *Roberts*, foi criado o algoritmo roberts.c alterando-se somente a máscara na estrutura explicada no item 4.1 Essa alteração é mostrada a seguir:

```
%Máscaras do operador Roberts

Horizontal
```

MX[0][0] = 0;	MX[1][0] = 0;	MX[2][0] = 0;
MX[0][1] = 0;	MX[1][1] = 1;	MX[2][1] = 0;
MX[0][2] = 0;	MX[1][2] = 0;	MX[2][2] = -1;

```
Vertical
```

MY[0][0] = 0;	MY[1][0] = 0;	MY[2][0] = 0;
MY[0][1] = 0;	MY[1][1] = -1;	MY[2][1] = 0;
MY[0][2] = 0;	MY[1][2] = 0;	MY[2][2] = 1;

Na Figura 4.1, visualiza-se a imagem em níveis de cinza antes de processá-la, e, na Figura 4.2, visualiza-se o resultado da imagem processada pelo operador de *Roberts* através do programa roberts.c.

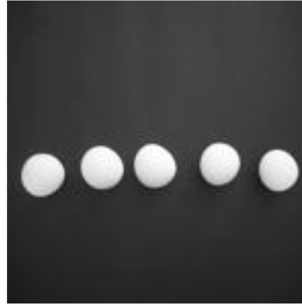


Figura 4. 1. Imagens em níveis de cinza pré-processada pelo MATLAB.

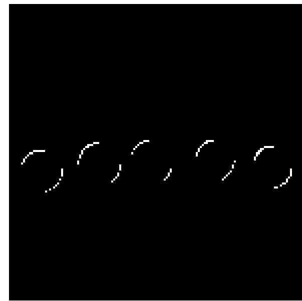


Figura 4. 2. Imagens processada pelo operador de *Roberts*.

Assim como nos estudos realizado no MATLAB, o resultado obtido pelo operador de *Roberts* implementa em linguagem C também obteve um resultado com falhas nas bordas dos ovos.

4.1. 2. Operador de *Prewitt*

Para o operador de *Prewitt* foi criado o algoritmo prewitt.c e assim como no operador de *Roberts*, alterou-se somente a máscara na estrutura explicada no item 4.1. Essa alteração é mostrada a seguir:

```
%Máscaras do operador Prewitt

Horizontal
```

MX[0][0] = -1;	MX[1][0] = -1;	MX[2][0] = -1;
MX[0][1] = 0;	MX[1][1] = 0;	MX[2][1] = 0;
MX[0][2] = 1;	MX[1][2] = 1;	MX[2][2] = 1;

Vertical

MY[0][0] = 1;	MY[1][0] = 0;	MY[2][0] = -1;
MY[0][1] = 1;	MY[1][1] = 0;	MY[2][1] = -1;
MY[0][2] = 1;	MY[1][2] = 0;	MY[2][2] = -1;

Na Figura 4.3, visualiza-se o resultado aplicando o operador de *Prewitt*.

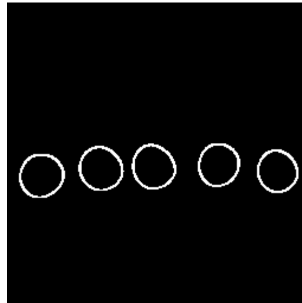


Figura 4. 3. Imagens processada pelo o operador de *Prewitt.*

O resultado obtido com a máscara de *Prewitt* foi bem melhor que o resultado obtido pela máscara de *Roberts* em relação às imagens estudadas. Pode-se notar na Figura 4.3 que os ovos contidos na imagem em nível de cinza foram claramente identificados, o que não ocorreu na imagem da Figura 4.2.

4.1. 3. Operador de *Sobel*

Para o operador de *Sobel* foi criado o algoritmo sobel.c e assim como nos dois outros operadores, alterou-se somente a máscara na estrutura explicada na seção 4.1. Essa alteração é mostrada a seguir:

%Máscaras do operador *Sobel*

Horizontal

MY[0][0] = -1;	MY[1][0] = -2;	MY[2][0] = -1;
MY[0][1] = 0;	MY[1][1] = 0;	MY[2][1] = 0;
MY[0][2] = 1;	MY[1][2] = 2;	MY[2][2] = 1;

```
Vertical
```

MX[0][0] = -1;	MX[1][0] = 0;	MX[2][0] = 1;
MX[0][1] = -2;	MX[1][1] = 0;	MX[2][1] = 2;
MX[0][2] = -1;	MX[1][2] = 0;	MX[2][2] = 1;

Na Figura 4.4, visualiza-se a imagem processada utilizando-se o operador de *Sobel*.

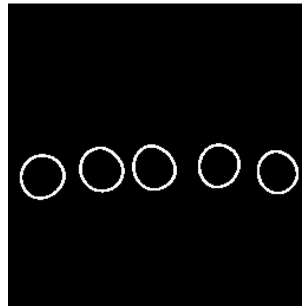


Figura 4. 4. Imagens processada pelo o operador de *Sobel.*

Comparada com as outras máscaras estudadas, a máscara de *Sobel* foi a que mostrou melhor resultado em relação às imagens de ovos.

Como o objetivo principal desta pesquisa é o uso de dispositivos para sistemas embarcados, os algoritmos processados em linguagem C foram modificados para serem executados no processador NIOS.

4. 2. Implementação dos algoritmos para detecção de bordas no processador NIOS

Para a aplicação dos operadores de bordas de *Roberts*, *Prewitt* e *Sobel*, na placa FPGA, foram criados algoritmos no ambiente computacional Dev-C++ com a linguagem C ANSI.

A placa utilizada para este projeto foi a FPGA, com o dispositivo da família Cyclone II, mais especificamente o dispositivo EP2C-35F672C6.

Os algoritmos criados denominados de nios_roberts.c, nios_prewitt.c e nios_sobel.c, relacionados respectivamente com os operadores de bordas *Roberts*, *Prewitt* e *Sobel*, foram implementados no NIOS com o sistema operacional *uClinux.* Os resultados foram armazenados na SD_Card, e com o auxílio da função *dlmread* do MATLAB, explicada na

seção 3.2, foi realizada a leitura dos arquivos com o formato txt relacionados com os resultados obtidos. As imagens processadas foram visualizadas no computador.

A configuração do processador NIOS que permitiu a instalação do *uClinux* e os programas para o processamento das imagens é apresentada no capítulo 6.

Os experimentos com o processador NIOS iniciou-se com imagem de pequenas dimensões, como a de 8x8. Notou-se que imagens com estas dimensões não permitiram uma boa detecção das imagens com os ovos. Continuou-se os testes para dimensões maiores, chegando-se até a dimensão 16x16. Com dimensões maiores o sistema travou em decorrência da falta de memória na FPGA utilizada.

Para resolver o problema de memória insuficiente, os algoritmos foram refeitos para poder particionar a imagem em quatro partes. Os algoritmos criados foram denominados de mat_inicial.c, mat1.c, mat2.c, mat3.c, mat4.c e mat_final.c. Os passos de cada algoritmo são explicados nas seções 4.2.1, 4.2.2, 4.2.3.

4.2. 1. Algoritmo mat_inicial.c

O algoritmo mat_inicial.c particionou a imagem em níveis de cinza com a dimensão 64x64 em quatro sub-imagens com a dimensão de 32x32 cada para poder processar cada partição com os operadores de *Roberts*, *Prewitt* e *Sobel*. A seguir apresenta-se, os passos realizados no algoritmo mat_inicial.c:

1 – Abertura do arquivo com a imagem inteira;

2 – Criação dos arquivos para a gravação das imagens particionadas;

3 – Leitura do arquivo inicial;

4 – Partição da imagem em quatro sub-imagens;

5 – Gravação das imagens particionadas em arquivos diferentes.

4.2. 2. Algoritmos mat1.c, mat2.c, mat3.c e mat4.c

Para trabalhar com as imagens particionadas foram criados algoritmos para processar cada partição, os algoritmos criados foram mat1.c, mat2.c, mat3.c e mat4.c. Os passos

realizados nesses algoritmos são os mesmos, alterando-se somente a imagem de entrada e a imagem de saída.

1 – Abertura do arquivo da imagem particionada;

2 – Criação do arquivo para gravar o resultado da imagem processada;

3 – Leitura da imagem particionada;

4 – Operação com o operador de bordas;

5 – Limiarização;

6 – Gravação do resultado da imagem processada.

4.2. 3. Algoritmo mat_final.c

O algoritmo mat_final.c junta os arquivos criados nos algoritmos em que as imagens particionadas são processadas, gerando o arquivo final contendo a imagem original processada.

Particionando as imagens, a maior dimensão que se conseguiu obter resultados foi com a dimensão de 64x64. Para dimensões maiores, houve problemas de falta de memória na FPGA.

Para cada operador foram criados os mesmos algoritmos, alterando-se somente a máscara de cada detector de borda. Na Figura 4.5, visualiza-se a imagem em níveis de cinza que foi processada com cada operador.

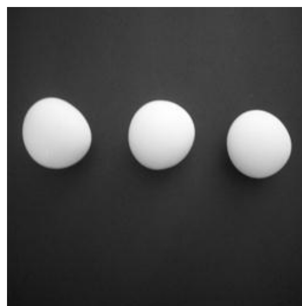


Figura 4. 5. Imagem em níveis de cinza e redimensionada.

Com a aplicação do operador de *Roberts*, obteve-se o resultado visualizado na Figura 4.6.

Figura 4. 6. Resultado da imagem processada pelo operador de *Roberts* no processador NIOS.

Na Figura 4.7, visualiza-se o resultado da imagem processada com o operador de *Prewitt.*

Figura 4. 7. Resultado da imagem processada pelo operador de *Prewitt* no processador NIOS.

Na Figura 4.8, visualiza-se o resultado da imagem processada com o operador de *Sobel.*

Figura 4. 8. Resultado da imagem processada pelo operador de *Sobel* no processador NIOS.

Os algoritmos relacionados aos operadores de bordas de *Roberts*, *Prewitt* e *Sobel* na linguagem C ANSI estão apresentados no Apêndice B.

Os estudos apresentados nos capítulos 3 e 4 utilizaram-se software na implementação dos algoritmos para a detecção de bordas. Com isso, o próximo capítulo descreve a implementação destes algoritmos de detecção de bordas no hardware, com modelos descritos na linguagem de descrição de hardware VHDL.

CAPÍTULO 5

Estudo dos Detectores de Bordas Utilizando a Linguagem VHDL

Assim como os capítulos 4 e 5, neste capítulo também apresenta-se a aplicação de detecção de bordas na identificação de ovos em uma esteira. Utilizam-se os operadores de *Roberts*, *Prewitt* e *Sobel*, ou seja, os mesmos utilizados no capítulo 3 e no capítulo 4, e explicados com detalhes no capítulo 2.

Entretanto, o ambiente computacional utilizado foi o QUARTUS com a linguagem VHDL. Para cada um dos operadores utilizados desenvolveu-se um algoritmo denominado, respectivamente, roberts.vhd, prewitt.vhd e sobel.vhd. Para cada algoritmo foi trabalhado com arquivos com o formato txt obtidos pelo ambiente computacional MATLAB, na forma descrita anteriormente.

A primeira tentativa para processar imagens através de modelos VHDL foi de importar a imagem por manipulação de arquivos, pois com a linguagem VHDL é possível essa manipulação. Mas os testes feitos no ambiente computacional QUARTUS mostraram que não é possível trabalhar com arquivos. Por isso descreveu-se as imagens internamente, ou seja, dentro do modelo descrito em VHDL.

Um único vetor recebe o arquivo descrito internamente, e para isso foi preciso alterar os arquivos das imagens em níveis de cinza com formato txt obtidos no MATLAB com o algoritmo alteracao_de_arquivos.c. Com este algoritmo gerou-se o arquivo arq_148.txt e entre os números que representam os pixels colocou-se vírgula (‘ , ’), no final de cada linha colocou-se ponto e vírgula (‘ ; ’) e no começo e no final do arquivo parênteses (‘(( ))’). Na Figura 5.1, visualiza-se o arquivo da imagem em níveis de cinza com formato txt e na Figura 5.2, visualiza-se o arquivo alterado com o algoritmo alteracao_de_arquivos.c.

```
niveis_cinza - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda
38 39 38 38 39 39 40 40 41 41
6 55 55 58 56 56 58 60 61 60
66 66 69 67 66 67 66 66 68 67
9 57 55 56 55 55 55 52 53 53
55 56 56 57 57 57 58 58 58 58
7 68 67 67 67 67 68 68 68 68
64 63 63 63 63 63 63 61 60 59
3 54 52 52 51 53 53 53 54 53
64 64 63 65 65 65 64 65 67 67
6 68 65 67 67 66 66 65 65 65
49 49 49 50 50 53 52 52 52 52
5 66 67 67 67 66 68 66 67 69
68 67 66 66 66 67 67 66 66 65
```

Figura 5. 1. Arquivo em níveis de cinza obtido pelo algoritmo sobel.m.

```
arq148 - Bloco de notas
Arquivo  Editar  Formatar  Exibir  Ajuda
((38, 39, 38, 38, 39, 39, 40, 40, 41, 41,
(40, 39, 38, 38, 39, 40, 40, 41, 41, 42,
(40, 39, 39, 38, 39, 40, 40, 40, 39, 41,
(41, 39, 39, 40, 39, 39, 41, 41, 39, 41,
(42, 39, 39, 39, 40, 39, 40, 40, 41, 42,
(40, 39, 38, 40, 41, 41, 41, 42, 41, 43,
(40, 41, 39, 41, 41, 41, 42, 42, 41, 43,
(41, 41, 40, 40, 41, 43, 42, 42, 42, 44,
(41, 40, 39, 40, 40, 41, 41, 42, 43, 43,
(41, 40, 41, 40, 41, 42, 42, 43, 43, 44,
(42, 40, 42, 42, 42, 43, 44, 43, 43, 43,
(42, 41, 40, 43, 42, 43, 42, 43, 42, 43,
(43, 40, 41, 42, 43, 43, 42, 43, 45, 45,
```

Figura 5. 2. Arquivo em níveis de cinza alterado pelo algoritmo alteração_de_arquivos.c para o vetor do modelo VHDL

Para obter resultados, foram feitas várias tentativas até chegar aos algoritmos roberts.vhd, prewitt.vhd e sobel.vhd compilados e com simulações geradas.

O primeiro programa compilado, que obteve resultado, foi muito extenso, tendo-se que escrever todas as variáveis do cálculo de convolução, pois não estava simulando todas as variáveis, obtendo-se somente a última variável dos cálculos no resultado final. Alguns passos deste programa extenso, os quais houve alteração no segundo programa é mostrado abaixo:

1 – Declaração das variáveis, neste caso, se a imagem tivesse com dimensão 64x64 *pixels*, teriam 62 variáveis, um exemplo desta declaração até a variável m9 é mostrado abaixo:

```
begin
process
variable m1:integer :=1;
variable m2:integer :=2;
variable m3:integer :=3;
```

```
variable m4:integer :=4;
variable m5:integer :=5;
variable m6:integer :=6;
variable m7:integer :=7;
variable m8:integer :=8;
variable m9:integer :=9;
```

2 – Estrutura de repetição para as variáveis das máscaras do eixo x e do eixo y do operador de borda, os quais variam de acordo com o pulso do clock, quando ele estiver descendo para o nível zero:

```
begin
wait until clk'event and clk='0'; → Espera do pulso do clock para a descida;
for i in 1 to 3 loop
 for j in 1 to 3 loop
  y1(i,j):= x(i,j);
  y2(i,j):=y(i,j);
 end loop;
end loop;
```

3 – Cálculo de convolução com a imagem original e as máscaras do operador de borda, sendo que para cada variável declarada tem o número de cálculo de convolução, tanto para o eixo x, quanto para o eixo y, ou seja, para a variável m1, tem 62 cálculos (hx(1...62)):

```
--variable m1:integer :=1;

hx(1) <= ((img(m1, 1) * y1(1, 1)) + (img(m1, 2) * y1(1, 2)) + (img(m1, 3) * y1(1, 3)) + (img(m1+1, 1) * y1(2, 1)) + (img(m1+1, 2) * y1(2, 2)) + (img(m1+1, 3) * y1(2, 3)) + (img(m1+2, 1) * y1(3, 1)) + (img(m1+2, 2) * y1(3, 2)) + (img(m1+2, 3) * y1(3, 3)));

hx(2) <= ((img(m1, 2)*y1(1, 1)) + (img(m1, 3) * y1(1, 2)) + (img(m1, 4) * y1(1, 3)) + (img(m1+1, 2) * y1(2, 1)) + (img(m1+1, 3) * y1(2, 2)) + (img(m1+1, 4) * y1(2, 3)) + (img(m1+2, 2) * y1(3, 1)) + (img(m1+2, 3) * y1(3, 2)) + (img(m1+2, 4) * y1(3, 3)));

Até a variável hx(62);

--variable m2:integer :=2;
```

```
hx(63) <= ((img(m2, 1) * y1(1, 1)) + (img(m2, 2) * y1(1, 2)) + (img(m2, 3) * y1(1, 3)) +
(img(m2+1, 1) * y1(2, 1)) + (img(m2+1, 2) * y1(2, 2)) + (img(m2+1, 3) * y1(2, 3)) +
(img(m2+2, 1) * y1(3, 1)) + (img(m2+2, 2) * y1(3, 2)) + (img(m2+2, 3) * y1(3, 3)));


hx(64) <= ((img(m2, 2) * y1(1, 1)) + (img(m2, 3) * y1(1, 2)) + (img(m2, 4) * y1(1, 3)) +
(img(m2+1, 2) * y1(2, 1)) + (img(m2+1, 3) * y1(2, 2)) + (img(m2+1, 4) * y1(2, 3)) +
(img(m2+2, 2) * y1(3, 1)) + (img(m2+2, 3) * y1(3, 2)) + (img(m2+2, 4) * y1(3, 3)));

Até a variável hx(124);

--variable m3:integer :=3;

Até a variável hx(186);

.

.

.

--variable m62:integer :=62;

Até a variável hx(3844);
```

O eixo y possui a mesma estrutura do eixo x, alterou-se apenas as variáveis de hx para hy, ou seja, a variável hy(1) até hy(3844).

No segundo programa, o algoritmo foi melhorado com sucesso, pois depois de várias tentativas, em vez de ter várias variáveis, utilizou-se apenas três e que passou a ser fixas, ou seja, independentemente da dimensão da imagem. Por exemplo, o primeiro laço de repetição começa com m1, m2 e m3, recebendo os valores de 1, 2 e 3, respectivamente, depois o segundo laço passa para 2, 3 e 4, respectivamente, e assim por diante.

Alguns passos da melhoria deste programa são mostrados abaixo:

1 – Declaração das variáveis, agora somente com três variáveis fixas, não variando de acordo com a dimensão:

```
shared variable m1 : integer;

shared variable m2 : integer;

shared variable m3 : integer;

shared variable tranz : integer;
```

2 – Atribuição dos valores iniciais antes da estrutura de repetição para as variáveis das máscaras do eixo x e do eixo y do operador de borda, os quais variam de acordo com o pulso do clock, quando ele estiver transitando para o nível zero:

```
begin
wait until clk'event and clk='0';
m1:=1;
m2:=2;
m3:=3;
tranz:=0;
for i in 1 to 3 loop
 for j in 1 to 3 loop
  y1(i,j):= x(i,j);
  y2(i,j):=y(i,j);
 end loop;
end loop;
```

3 – Estrutura de repetição para realizar a convolução com as variáveis do eixo x e do eixo y, variando de 1 até o número máximo de variáveis, e para alterar as variáveis m1, m2 e m3, de 1, 2 e 3, respectivamente para 2, 3 e 4, e assim por diante:

```
for l in 1 to 21316 loop
 hx(l):=0;
 hy(l):=0;
 tranz:=tranz+1;
 if tranz > 146 then
   m1:=m1+1;
   m2:=m2+1;
   m3:=m3+1;
```

```
  tranz:=1;
 end if;
```

4 – Cálculo de convolução com a imagem original e as máscaras do operador de borda:

```
hx(l) := ((img(m1, tranz) * y1(1, 1)) + (img(m1, tranz+1) * y1(1, 2))  + (img(m1, tranz+2) * y1(1, 3)) + (img(m2,tranz) * y1(2, 1)) +  (img(m2, tranz+1) * y1(2, 2))  +  (img(m2, tranz+2) * y1(2,3)) + (img(m3, tranz) * y1(3, 1)) + (img(m3, tranz+1) * y1(3, 2)) +  (img(m3, tranz+2) * y1(3, 3)));

hy(l) := ((img(m1, tranz) * y2(1, 1)) + (img(m1, tranz+1) * y2(1, 2))  +  (img(m1, tranz+2) * y2(1,3)) + (img(m2, tranz) * y2(2, 1)) + (img(m2, tranz+1) * y2(2, 2)) + (img(m2, tranz+2) * y2(2, 3)) + (img(m3, tranz) * y2(3, 1)) + (img(m3, tranz+1) * y2(2, 2)) +  (img(m3, tranz+2) * y2(3,3)));
```

5 – E por fim a limiarização, ou seja, se o resultado for maior que 254, pois os níveis de cinza variam de 0 a 255, será 1, se for menor será 0. E mandará o resultado de cada pixel à medida que o sinal do *clock* transita e for igual à 0 para a saída, gerando a simulação no *waveform editor*.

```
 if n(l)>254 then
 limiar(l):=1;
 else
  limiar(l):=0;
 end if;
end loop;
if clk'event and clk='0' then
 d:=d+1;
 saida<=limiar(d);
 else
  saida<=limiar(d);
  d:=0;
end if;
end process;
end teste;
```

Com a melhora do algoritmo, houve alteração no tempo de compilação. Por exemplo, para a imagem com dimensão de 64x64 *pixels* no primeiro algoritmo demorou 25 horas para compilar, e no segundo algoritmo demorou 15 horas para a mesma dimensão. Houve uma tentativa com a dimensão de 128x128 *pixels*, porém sem sucesso; a compilação durou 35 horas e gerou erro de memória.

Com esse erro, trabalhou-se somente com a dimensão de 64x64 *pixels* e após pesquisas descobriu-se que não haveria erro de memória com a dimensão de 72x72 *pixels*, obtendo a imagem com sucesso. A última dimensão compilada com êxito foi de 98x98 *pixels*, pois a partir da dimensão de 100x100 *pixels*, ocorreu o mesmo erro de memória, igual ao que ocorreu com a dimensão de 128x128 *pixels*.

Essas tentativas foram feitas com os ambientes computacionais QUARTUS 9.0, e QUARTUS 9.1 sp2. Nas mesmas tentativas com a dimensão de 98x98, o tempo de compilação diminuiu de quase vinte e três horas para doze minutos. Como esse tempo de compilação diminuiu, a próxima idéia foi de realizar novas tentativas com dimensões maiores, mas agora com o QUARTUS 9.1 sp2.

Essa última tentativa obteve compilação com êxito até a dimensão de 148x148 *pixels*, e a partir da dimensão de 150x150 *pixels*, ocorreu o mesmo erro de memória que havia para dimensão de 100x100 *pixels* com o ambiente QUARTUS 9.0.

Após a compilação do algoritmo, cria-se a simulação pelo *waveform editor*, dos sinais de resposta, os quais oscilam com sinal de *clock*.

Essa simulação varia na freqüência e no tempo de simulação, de acordo com a dimensão da imagem. Por exemplo, para a simulação da imagem com dimensão de 64x64 *pixels*, o *clock* teve um tempo de 40 μs. Com a dimensão de 98x98 *pixels* teve o tempo de 93 μs, e com a dimensão de 148x148 *pixels* teve o tempo de 213 μs.

Na Figura 5.3, visualiza-se um exemplo da simulação obtida a partir do *waveform editor* para a imagem com dimensão de 148x148 *pixels*, compilada com o operador de *Sobel*.

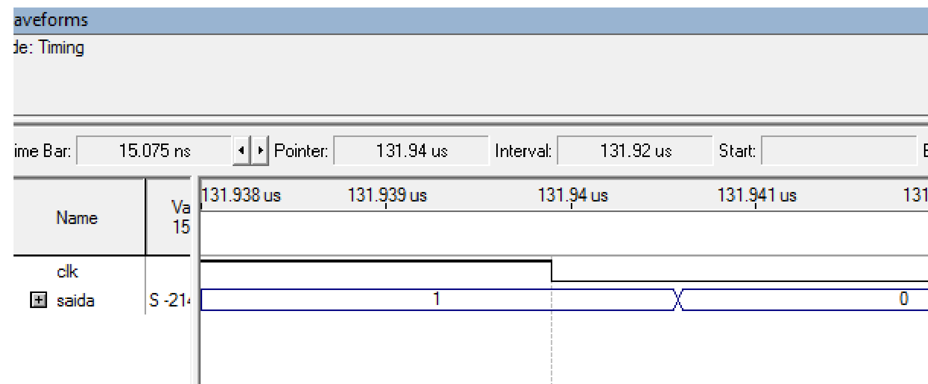


Figura 5. 3. Simulação do algoritmo de *Sobel* com imagem de dimensão de 148x148 *pixels*.

Depois da simulação obtida, o próximo passo foi armazená-la com extensão tbl, ou seja, uma forma de armazenar o arquivo de simulação em forma de texto para dar continuidade no processo de visualização da imagem com o auxílio do ambiente computacionais MATLAB.

Na Figura 5.4, visualiza-se a simulação sobel.tbl armazenada com extensão tbl para a imagem de dimensão 148x148 *pixels*. Nota-se que a Figura 5.4 contém 3 colunas, a primeira coluna mostra o tempo que ocorreu a simulação, na segunda coluna o nível de transição de clock e na terceira coluna o nível lógico da simulação.

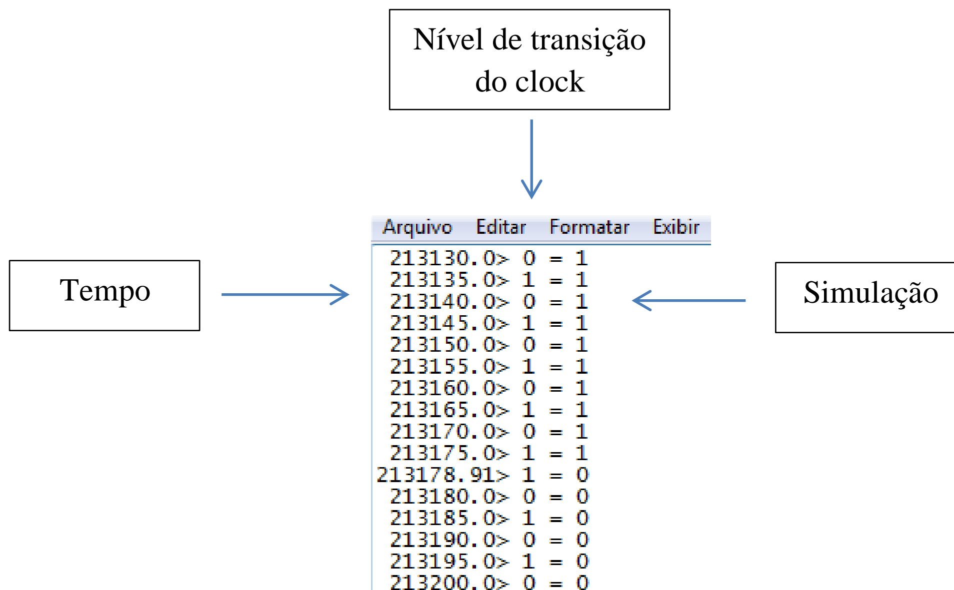


Figura 5. 4. Simulação com extensão tbl

Depois de simulado e gerado o arquivo sobel.tbl, o próximo passo foi importar os dados com o MATLAB e executar o algoritmo vhdl_mat.m criado para diferenciar os números do *clock* com os números dos sinais gerados pela simulação do *waveform editor*. Após a execução do algoritmo vhdl_mat.m obtêm-se o arquivo nulo1.txt explicado na seção 3.4. O arquivo nulo1.txt é visualizado na Figura 5.5.

Figura 5. 5. Arquivo nulo1.txt alterado pelo algoritmo vhdl_mat.m.

O arquivo gerado foi armazenado e manipulado no ambiente computacional Dev_C++. Com o algoritmo teste_nulo.c, substitui o número (‘2’) por um espaço vazio (‘ ’) Com isso gerou-se o arquivo arq_nulo_sobel.txt apenas com os números de nível lógico, ou seja, os ‘0s’ e ‘1s’. O algoritmo img_quartus.c gerou-se o arquivo arq_sobel.txt com o vetor redimensionado para a matriz de 148x148 *pixels*. Visualiza-se o arquivo alterado pelo algoritmo teste_nulo.c na Figura 5.6. Na Figura 5.7, visualiza-se o arquivo alterado pelo algoritmo img_quartus.c.

Figura 5. 6. Arquivo alterado pelo algoritmo teste_nulo.c.

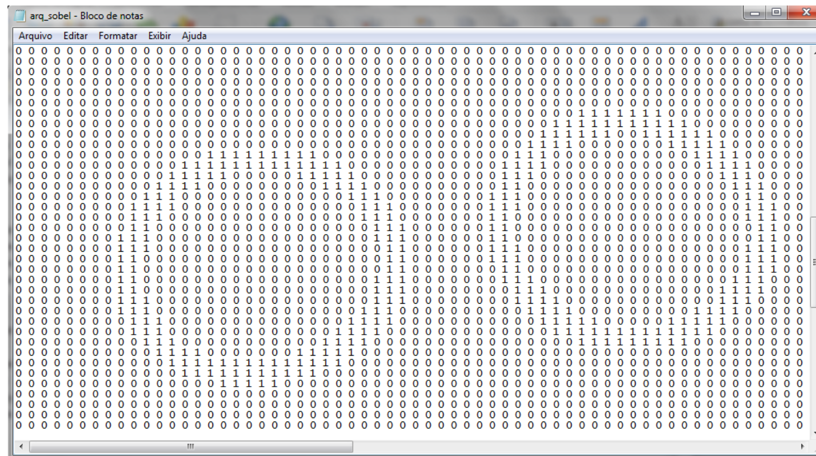


Figura 5. 7. Arquivo redimensionado pelo algoritmo img_quartus.c

Para cada máscara estudada, foi gerado um algoritmo descrito em VHDL, compilado e visualizado, sendo os algoritmos roberts.vhd, prewitt.vhd e sobel.vhd, para os operadores, *Roberts*, *Prewitt* e *Sobel*, respectivamente nas seções 5.1, 5.2, e 5.3. Na Figura 5.8, visualiza-se a imagem dos ovos em níveis de cinza.

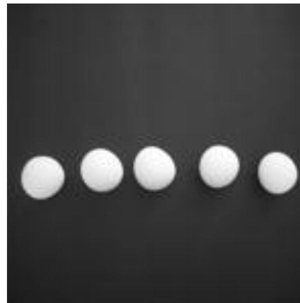


Figura 5. 8. Imagem em níveis de cinza.

5. 1. Operador de *Roberts*

Para o operador de *Roberts*, foi criado o algoritmo roberts.vhd, o qual utiliza uma máscara com a dimensão de 2x2 *pixels*. Estas máscaras são visualizadas nas matrizes, horizontal para o eixo x e vertical para o eixo y na Tabela 5.1.

Tabela 5. 1. Máscaras do operador de *Roberts*.

Horizontal	Vertical
$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$

Na Figura 5.9, visualiza-se a imagem com as bordas destacadas pelo detector de borda de *Roberts*.

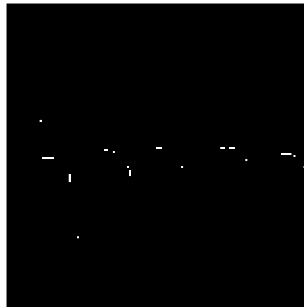


Figura 5. 9. Imagem processada pelo detector de bordas de *Roberts* modelada em VHDL.

5. 2. Operador de *Prewitt*

Para o operador de *Prewitt* foi criado o algoritmo prewitt.vhd, o qual utiliza uma máscara com a dimensão de 3x3 *pixels*. Estas máscaras são visualizadas nas matrizes, horizontal para o eixo x e vertical para o eixo y na Tabela 5.2.

Tabela 5. 2. Máscaras do operador de *Prewitt*.

Horizontal	Vertical
$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$

Na Figura 5.10, apresenta-se o resultado da imagem processada pelo detector de bordas de *Prewitt*.

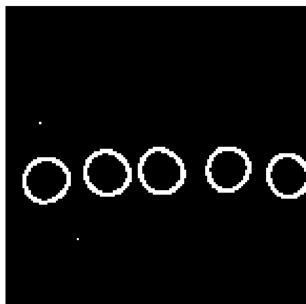


Figura 5. 10. Imagem processada pelo detector de bordas de *Prewitt* modelada em VHDL.

5. 3. Operador de *Sobel*

Para o operador de *Sobel* foi criado o algoritmo sobel.vhd, o qual utiliza uma máscara com a dimensão de 3x3 *pixels*. Estas máscaras são visualizadas nas matrizes, horizontal para o eixo x e vertical para o eixo y na Tabela 5.3.

Tabela 5. 3. Máscaras do operador de *Sobel*.

Horizontal	Vertical
$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$

E na Figura 5.11, o resultado da imagem processada pelo detector de bordas de *Sobel*.

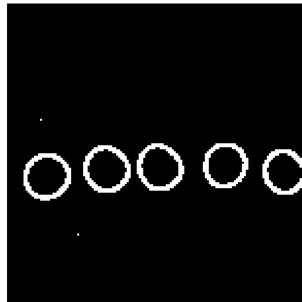


Figura 5. 11. Imagem processada pelo detector de bordas de *Sobel* modelada em VHDL.

Os algoritmos relacionados aos operadores de bordas de *Roberts*, *Prewitt* e *Sobel* modelados na linguagem VHDL e compilados no ambiente computacional QUARTUS estão apresentados no Apêndice C.

Uma das grandes contribuições deste trabalho foi o estudo para o emprego do processador NIOS.

Antes de se utilizar o processador NIOS é necessário configurá-lo com a arquitetura definida. Apresenta-se no próximo capítulo um resumo de como o processador NIOS foi configurado para que os operadores de *Roberts*, *Prewitt* e *Sobel* pudesse ser implementado na FPGA EP2C35F672C6.

CAPÍTULO 6

Configuração do Processador NIOS e o Sistema Operacional *uClinux*

A principal contribuição deste projeto na área de processamento de imagens digitais foi a aplicação dos operadores de bordas na placa FPGA, juntamente com o processador NIOS II e o sistema operacional *uClinux*. Os algoritmos nios_roberts.c, nios_prewitt.c e nios_sobel.c respectivamente foram programados com a linguagem C ANSI, para serem executados no processador NIOS.

6. 1. Dispositivo FPGA

Os dispositivos FPGA são circuitos integrados digitais com blocos lógicos reprogramáveis, os quais recebem o nome de CLB (*Configuration Logical Blocks*), estes são formados por portas lógicas e *flips-flops* que implementam funções lógicas, e também por interconexões entre eles que podem ser rearranjadas.

6. 2. Kit de desenvolvimento DE2

O Kit de desenvolvimento DE2 – 2C35 da Altera foi o Kit utilizado neste projeto para a implementação dos operadores de bordas de *Roberts*, *Prewitt* e *Sobel*. Este Kit possui componentes que permitem vários tipos de aplicações, sendo possível até projetos complexos. Na Figura 6.1, visualiza-se o KIT DE2 – 2C35 de desenvolvimento da Altera.

Figura 6. 1– Placa DE2 – 2C35 da Altera (ALTERA, 2010).

A placa DE2 é composta pelos seguintes componentes:

- Altera Cyclone II EP2C35F672C6 FPGA;
- Altera EPCS16 – Dispositivo de configuração Serial;
- USB Blaster para programação;
- 512 KBytes SRAM (*Static Random Access Memory*);
- 8 Mbytes SDRAM (*Synchronous Dynamic Random Access Memory*);
- 4 Mbytes *Flash Memory*;
- Soquete para SD_Card;
- Botões;
- 4 interruptores de botões;
- 18 interruptores;
- 18 LEDs (*Light-Emitting Diode*) vermelhos;
- 9 LEDs verdes;
- Osciladores de 50 MHz, 27 MHz e conector para oscilador externo;
- CODEC (*Coder-Decoder*) de Áudio de 24 bits;
- VGA (*Video Graphics Array*) DAC (*Digital-to-Analog Converter*) de 10 bits;
- TV *Decoder* (NTSC/PAL);
- Controlador *Ethernet* 10/100;
- Controlador USB com conectores do Tipo A e do Tipo B;
- *Transceiver* RS-232;
- *Transceiver* IrDA (*Infrared Data Association*);
- Conector PS/2;
- Dois barramentos expansores de 40 pinos com proteção de Diodos.

6. 3. Processador NIOS

O processador NIOS II é um dos componentes de hardware configurado na placa FPGA e é um processador RISC (*Reduced Instruction Set Computer*), o qual possui um barramento de dados configurável para 16 ou 32 bits.

A Altera disponibiliza algumas configurações do NIOS II para facilitar o trabalho dos projetistas. O processador pode ser implementado em três tipos de configurações diferentes:

- **NIOS II/f**: É a versão mais rápida para um desempenho superior. Ela tem opções para uma configuração mais ampla. Pode ser usado para otimizar o desempenho do processador;
- **NIOS II/s**: É a versão padrão, requer menos recursos no dispositivo FPGA. Reduz o desempenho;
- **NIOS II/e**: É uma versão mais econômica. Requer uma quantidade menor de recursos da FPGA, mas também limita os recursos para a configuração do usuário.

6. 4. Sistema operacional *uClinux*

O *uClinux* é um sistema operacional desenvolvido para os processadores/microcontroladores que não possue MMU(*Memory Management Unit*), ou seja, unidade de gerenciamento de memória.

Os processadores para os quais aplicam-se o sistema *uClinux* geralmente tem o custo mais baixo e frequentemente são utilizados em sistemas com tecnologia embarcada, ou seja, a combinação entre o hardware e o software, com recursos, como memória e poder de processamento mais escassos. (ALMEIDA, 2003).

6. 5. Aplicação do projeto

Para realizar a aplicação do projeto na placa FPGA foi utilizado o ambiente computacional QUARTUS para criar o projeto, e para isso seguiu como exemplo o projeto DE2_NET fornecido pelo Kit de desenvolvimento DE2.

Após a criação, o projeto system_0.ptf foi compilado e construído a imagem zImage no sistema operacional *uClinux* para aplicar na placa FPGA. A seguir são visualizados os passos realizados para esta aplicação.

6.5. 1. Construção do Hardware

O ambiente computacional utilizado para a construção e compilação do *hardware* foi o QUARTUS 9.1 sp2. Seguiu-se como exemplo o projeto DE2_NET da Altera, alterando para deixa-lo compatível ao necessário para este projeto na aplicação dos operadores de bordas de *Roberts*, *Prewitt* e *Sobel*, ou seja, a retirada dos componentes desnecessários para a aplicação do *software* descritos na linguagem C ANSI dos operadores de bordas e realizar a comunicação com o SD_Card.

Para editar os componentes do system_0.ptf do projeto DE2_NET criou-se um diagrama de bloco chamado Bloco1.bdf, em seguida foi editado o sistema system_0.sopc, deixando apenas os componentes necessários para o desenvolvimento deste projeto.

Os módulos do *hardware* utilizados para o desenvolvimento deste projeto:

- cpu_0: Processador NIOS II – Nios II//f;
- tristage_bridge_0;
- SDRAM;
- epcs_controller;
- jtag_uart_0;
- uart_0;
- uart_1;
- timer_0;
- timer_1;
- PIO – led_red;
- PIO – led_green;
- button_pio;
- switch_pio;
- SRAM;
- DM9000A;
- mmc_spi;
- PIO - pio_0;
- PIO – pio_1.

Após criar e editar os componentes, o *hardware* system_0.sopc é salvo e em seguida gerado o sistema system_0.ptf. O Bloco1 é atualizado de acordo com os componentes editados e inseridos no system_0.sopc. Na Figura 6.2, visualiza-se o esquemático do Bloco 1.

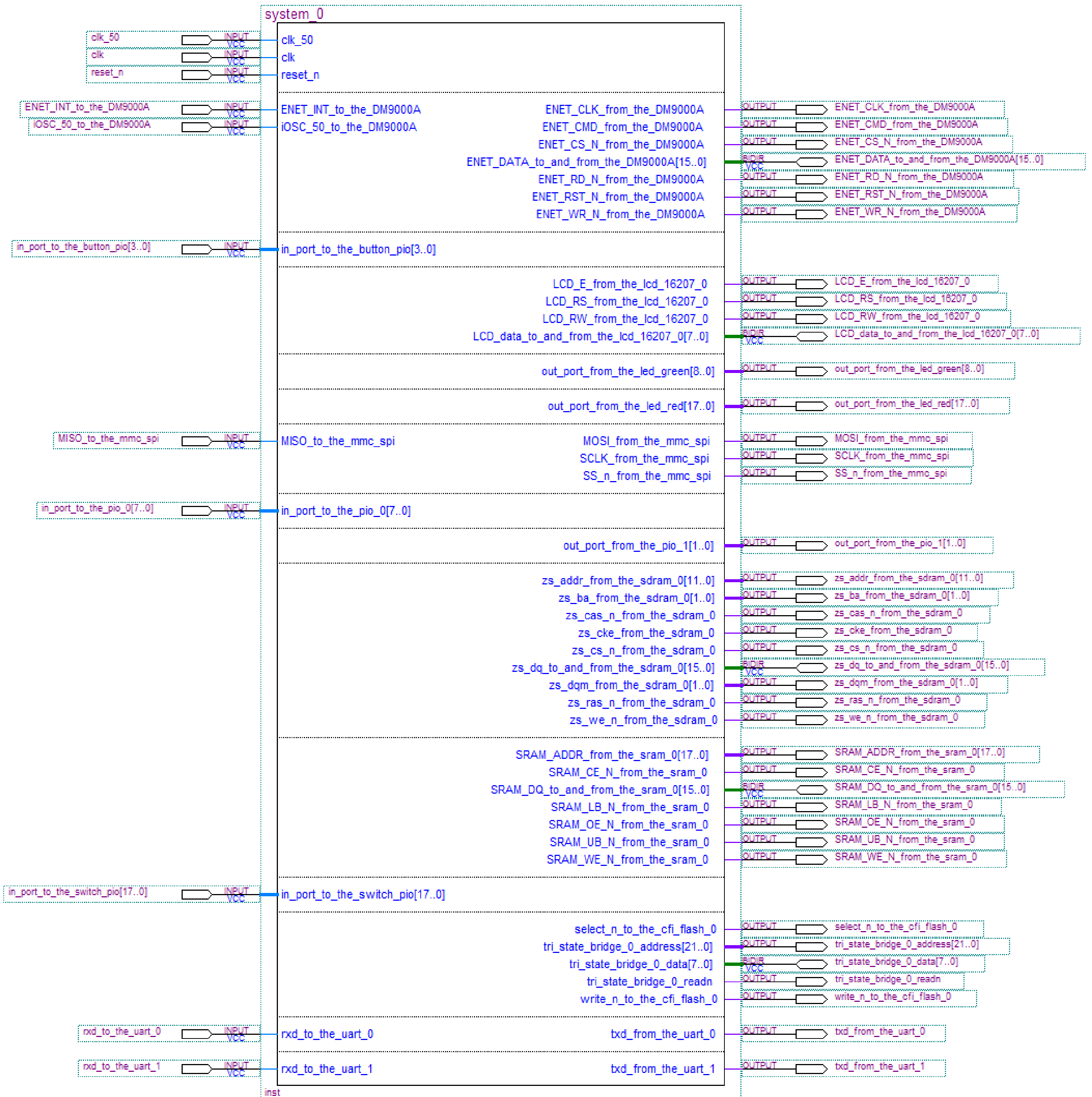


Figura 6. 2. Esquemático – Bloco1.bdf

Em seguida foi preciso editar o algoritmo descrito em Verilog do exemplo DE2_NET. Para isso, foi preciso retirar alguns componentes desnecessários, como por exemplo, o componente da USB, ISP1362, para não haver conflito com o hardware criado do sistema system_0.ptf e a inserção do componente mmc_spi para a comunicação serial com o cartão de memória SD_card. Na Figura 6.3, visualiza-se os componentes do SOPC-Builder.

Use	C...	Module Name	Description	Clock	Base	End	Tags	IRQ
☑		cpu_0	Nios II Processor	clk	0x01901000	0x019017ff		
☑		tri_state_bridge_0	Avalon-MM Tristate Bridge	clk				
☑		cfi_flash_0	Flash Memory Interface (CFI)	clk	0x01400000	0x017fffff		
☑		sdram_0	SDRAM Controller	clk_50	0x00800000	0x00ffffff		
☑		epcs_controller	EPCS Serial Flash Controller	clk	0x01901800	0x01901fff		0
☑		jtag_uart_0	JTAG UART	clk	0x01902120	0x01902127		1
☑		uart_0	UART (RS-232 Serial Port)	clk	0x01902000	0x0190201f		2
☑		uart_1	UART (RS-232 Serial Port)	clk	0x01902060	0x0190207f		3
☑		timer_0	Interval Timer	clk	0x01902020	0x0190203f		4
☑		timer_1	Interval Timer	clk	0x01902040	0x0190205f		5
☑		lcd_16207_0	Character LCD	clk	0x019020a0	0x019020af		
☑		led_red	PIO (Parallel I/O)	clk	0x019020b0	0x019020bf		
☑		led_green	PIO (Parallel I/O)	clk	0x019020c0	0x019020cf		
☑		button_pio	PIO (Parallel I/O)	clk	0x019020d0	0x019020df		6
☑		switch_pio	PIO (Parallel I/O)	clk	0x019020e0	0x019020ef		
☑		sram_0	SRAM_16Bit_512K	clk	0x01880000	0x018fffff		
☑		mmc_spi	SPI (3 Wire Serial)	clk	0x01902080	0x0190209f		7
☑		DM9000A	DM9000A	clk	0x01902128	0x0190212f		8
☑		pio_0	PIO (Parallel I/O)	clk	0x019020f0	0x019020ff		
☑		pio_1	PIO (Parallel I/O)	clk	0x01902100	0x0190210f		

Figura 6. 3. Componentes do SOPC-Builder

Também foi inserido no algoritmo DE2_NET.v a parte relacionada a comunicação serial com o SD_Card, ou seja:

```
.MISO_to_the_mmc_spi(SD_DAT),

.MOSI_from_the_mmc_spi(SD_CMD),

.SCLK_from_the_mmc_spi(SD_CLK),

.SS_n_from_the_mmc_spi(SD_DAT3),
```

Após editar o algoritmo descrito em Verilog, DE2_NET.v, o projeto é compilado e o sistema system_0.ptf é compilado no sistema operacional uClinux para gerar a imagem zImage, ou seja, o sistema operacional uClinux compilado junto com o projeto system_0.ptf. Após gerar a imagem zImage, ela é carregada para placa DE2 da Altera pelo ambiente NiosII 8.1 Command Shell.

Antes de carregar a imagem, o projeto DE2_NET é carregado para a placa pela ferramenta *programmer* - ferramenta do ambiente QUARTUS – com o arquivo DE2_NET_time_limited.sof.

Logo após carregar o arquivo DE2_NET_time_limites.sof para a placa DE2 ficará aparecendo a mensagem que o computador está conectado à placa. Na Figura 6.4, visualiza-se o ambiente QUARTUS com a mensagem de conexão.

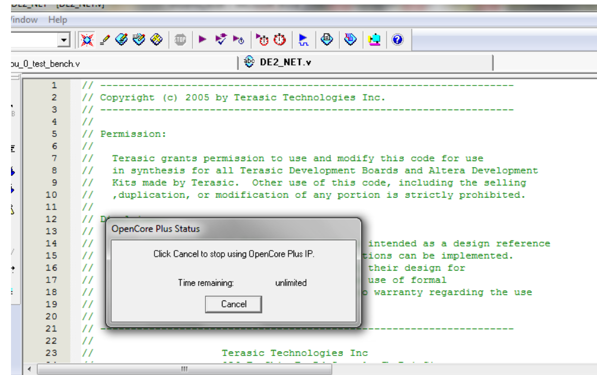


Figura 6. 4. Ambiente computacional QUARTUS.

Em seguida a imagem é carregada para a placa pelo ambiente NiosII 8.1 Command Shell, usando o seguinte comando:

nios2-download –g zImage

E após a imagem ser carregada digita-se um comando para acessar o terminal uClinux, isso ocorre usando o seguinte comando:

nios2-terminal

Na Figura 6.5, visualiza-se a tela do ambiente NiosII 8.1 Comand Shell com acesso ao terminal uClinux.

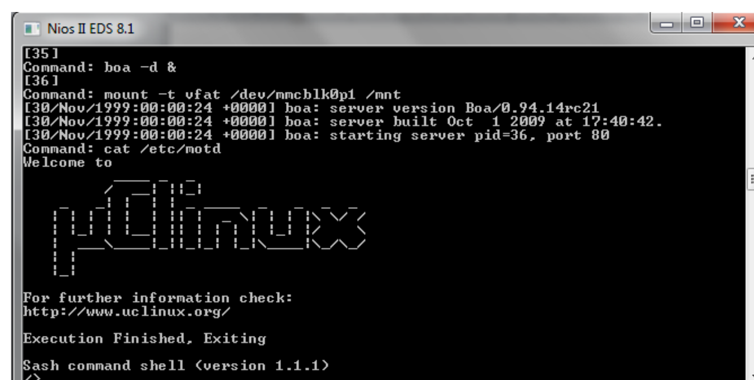


Figura 6. 5. Terminal *uClinux*.

6.5. 2. Implementação do *Software*

Neste projeto inseriu o SD_Card para processar os operadores de borda, para isso compilou-se os algoritmos nios_roberts.c, nios_prewitt.c, nios_sobel.c, respectivamente para os operadores de *Roberts*, *Prewitt* e *Sobel*, no sistema operacional Linux. Para isso transferiu-se a imagem em níveis de cinza com o formato txt, ou seja, o arquivo níveis_cinza.txt e os algoritmos - mat_inicial.c, mat1.c, mat2.c, mat3.c, mat4.c, mat_final.c e nios_sobel.c. A demonstração mostrada em seguida são relacionadas ao operador de *Sobel*. Estes algoritmos são explicados com mais detalhes na seção 4.2. Para compilar os algoritmos no sistema operacional Linux utilizou-se o comando:

```
nios2-linux-uclibc-gcc nios_sobel.c –o card_02 –elf2flt
```

O algoritmo nios_sobel.c compila todos os algoritmos em uma única vez, não havendo a necessidade de compilar um por vez no sistema operacional Linux. Com este comando criou-se o arquivo card_02 com o algoritmo nios_sobel.c compilado. Em seguida este arquivo é transferido para o SD_Card e este é inserido na placa DE2. Na Figura 6.6, visualiza-se a tela do ambiente NiosII 8.1 Command Shell acessando o cartão de memória SD_Card, onde esta contido o arquivo card_02, e o arquivo níveis_cinza.txt.

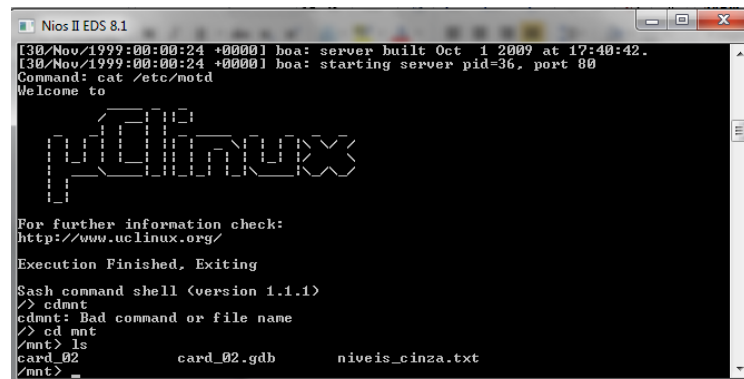


Figura 6. 6. Acesso ao cartão de memória SD_Card.

Em seguida o arquivo card_02 é executado com o comando ./card_02 na placa DE2, gerando os arquivos imagem1.txt, imagem2.txt, imagem3.txt, imagem4.txt, arq_H1.txt, arq_H2.txt, arq_H3.txt, arq_H4.txt e arq_final.txt, sendo este último o arquivo com a imagem final gerada com sucesso. Na Figura 6.7, visualiza-se o ambiente NiosII 8.1 Command Shell com a execução do arquivo card_02 e os arquivos gerados.

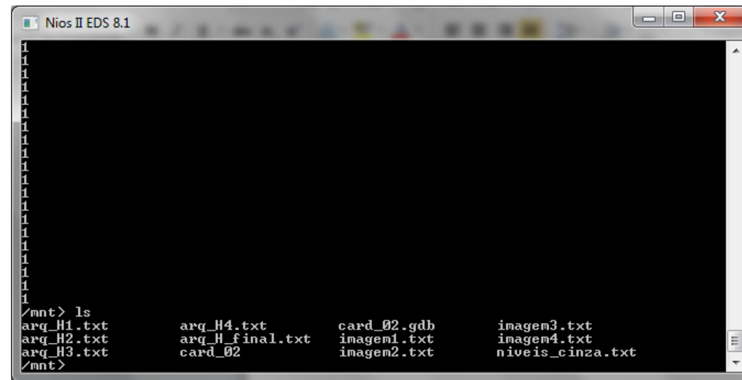


Figura 6. 7. Arquivos gerados.

O arquivo final arq_H_final.txt é visualizado no ambiente computacional MATLAB com a função *dlmread*, que foi explicado com mais detalhe na seção 3.3.

Este processo foi realizado para cada operador de borda estudado, ou seja, os operadores de *Roberts*, *Prewitt* e *Sobel*. Cada imagem foi gerada com um ruído nas bordas das imagens. Na Figura 6.8, visualiza-se a imagem obtida com o operador de *Roberts*.

Figura 6. 8. Resultado da imagem processada pelo operador de *Roberts*.

Na Figura 6.9, visualiza-se a imagem obtida com o operador de *Prewitt*.

Figura 6. 9. Resultado da imagem processada pelo operador de *Prewitt*.

Na Figura 6.10, visualiza-se a imagem obtida com o operador de *Sobel*.

Figura 6. 10. Resultado da imagem processada pelo operador de *Sobel*.

Os comandos executados para a criação da imagem zImage, para a compilação dos algoritmos dos operadores de bordas e a execução na placa DE2 são mostrados no Apêndice D.

A comparação entre os resultados das aplicações nos ambientes computacionais MATLAB, Dev-C++, QUARTUS e a aplicação na placa DE2 e entre os operadores de bordas *Roberts*, *Prewitt* e *Sobel* serão explicadas no capítulo 7.

CAPÍTULO 7

CONCLUSÃO

Para as aplicações feitas neste projeto comprovou-se que o melhor método de detecção de bordas é o método de *Sobel* em relação aos métodos de *Roberts* e de *Prewitt*.

O método de *Sobel* além de destacar-se com mais intensidade as bordas em relação aos outros métodos, também suaviza a imagem ao mesmo tempo. Todos os métodos trabalham com a convolução entre as máscaras e a imagem original e em seguida ocorre a limiarização para assim obter as bordas destacadas do restante da imagem.

As aplicações com os operadores de bordas foram realizadas nos ambientes computacionais MATLAB, Dev-C++, descrito com a linguagem C ANSI, e QUARTUS modelado com a linguagem VHDL. Com a linguagem C ANSI também criou-se algoritmos para particionar imagens e implementar na FPGA com o processador NIOS e o sistema operacional *uClinux*, tendo com essa aplicação, a contribuição para a área de processamento de imagens.

As imagens foram obtidas através de máquina fotográfica, e armazenadas com a extensão jpeg e depois processadas no ambiente computacional MATLAB, primeiro para obter a base de comparação com os outros resultados, pois esse ambiente é conhecido por trabalhar com processamento de imagens, segundopara gerar os arquivos com extensão txt para processar os operadores de bordas de *Roberts*, *Prewitt* e *Sobel* nos outros ambientes computacionais.

Os resultados obtidos em cada aplicação foram comparados entre eles e tendo como base o resultado obtido no MATLAB, os outros resultados são visualmente parecidos, com exceção de ruídos nas imagens obtidas com o processo no ambiente QUARTUS modelado com a linguagem VHDL e na implementação do *software* gerado na linguagem C ANSI na placa FPGA.

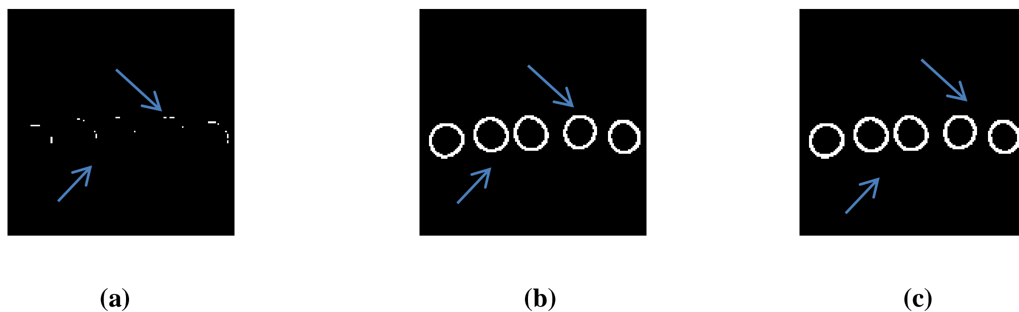
7. 1. Comparação entre os resultados

Nesta seção, apresenta-se a comparação entre as imagens obtidas com os cálculos dos operadores de bordas *Roberts*, *Prewitt* e *Sobel* e entre as imagens obtidas nas aplicações

no MATLAB, com a linguagem C ANSI, com a linguagem VHDL e a implementação na placa FPGA com a linguagem C ANSI.

A comparação entre os operadores prova que o detector de borda *Sobel* é o melhor para a realização do projeto em relação aos detectores de *Prewitt* e *Roberts*. As imagens obtidas em cada algoritmo criado são visualmente parecidas.

Entre os operadores observou-se que o operador de *Roberts* não obteve uma visualização com todas as bordas detectadas para a visualização da detecção de ovos, por ser uma máscara de dimensão 2x2. E o operador de *Prewitt* não obteve um resultado melhor que o operador de *Sobel* para este trabalho pelo fato do operador de *Sobel* ter um peso maior no pixel central que o operador de *Prewitt*. Na Figura 7.1, visualiza-se a comparação entre os operadores de *Roberts*, *Prewitt* e *Sobel*. Observa-se que na comparação enre os operadores, o detector de *Sobel* desta melhor as bordas.



(a) **(b)** **(c)**

Figura 7. 1. Comparação entre os operadores de bordas de *Roberts*, *Prewitt* e *Sobel* – (a) *Roberts*; (b) *Prewitt*; (c) *Sobel*.

O ambiente MATLAB, por ser um software mais usual na área de processamento de imagens, serviu como base para a comparação entre as imagens obtidas com os algoritmos descritos na linguagem C ANSI e na linguagem VHDL e também com a implementação no processador NIOS com a linguagem C ANSI.

Entre o MATLAB e a linguagem C ANSI, os resultados para todos os operadores estudados são visualmente parecidos.

Na Figura 7.2, visualiza-se a comparação entre os ambientes computacionais MATLAB e a linguagem C ANSI para o operador de *Roberts*.

(a) (b)

Figura 7. 2. Comparação entre os resultados para o operador de *Roberts* – (a) MATLAB; (b)C ANSI.

Na Figura 7.3, visualiza-se a comparação entre os ambientes computacionais MATLAB e a linguagem C ANSI para o operador de *Prewitt*.

(a) (b)

Figura 7. 3. Comparação entre os resultados para o operador de *Prewitt* – (a) MATLAB; (b)C ANSI.

Na Figura 7.4, visualiza-se a comparação entre os ambientes computacionais MATLAB e a linguagem C ANSI para o operador de *Sobel*.

(a) (b)

Figura 7. 4. Comparação entre os resultados para o operador de *Sobel* – (a) MATLAB; (b)C ANSI.

As imagens obtidas no MATLAB e no QUARTUS como modelo VHDL são visualmente parecidas, com exceção de dois pontos, os quais são ruídos na imagem, indicadas pelas setas nas imagens obtidas no processo com os operadores de bordas *Roberts*, *Prewitt* e

Sobel no ambiente QUARTUS como modelo VHDL. Os dois pontos estão apontados por setas para ficarem em destaque.

Na Figura 7.5, visualiza-se a comparação entre os ambientes computacionais MATLAB e VHDL para o operador de *Roberts*.

(a) **(b)**

Figura 7. 5. Comparação entre os resultados para o operador de *Roberts* – (a) MATLAB; (b)VHDL.

Na Figura 7.6, visualiza-se a comparação entre os ambientes computacionais MATLAB e VHDL para o operador de *Prewitt*.

(a) **(b)**

Figura 7. 6. Comparação entre os resultados para o operador de *Prewitt* – (a) MATLAB; (b)VHDL.

Na Figura 7.7, visualiza-se a comparação entre os ambientes computacionais MATLAB e VHDL para o operador de *Sobel*.

(a) (b)

Figura 7. 7. Comparação entre os resultados para o operador de *Sobel* – (a) MATLAB; (b)VHDL.

As imagens obtidas no MATLAB e a implementação no processador NIOS com a linguagem C ANSI são visualmente parecidas, com exceção de alguns ruídos indicadas na imagem por setas. Nas imagens obtidas no processo com os operadores de bordas *Roberts*, *Prewitt* e *Sobel* na implementação na placa FPGA.

Na Figura 7.8, visualiza-se a comparação entre os ambientes computacionais MATLAB e a implementação no processador NIOS para o operador de *Roberts*.

(a) (b)

Figura 7. 8. Comparação entre os resultados para o operador de *Roberts* – (a) MATLAB; (b)NIOS.

Na Figura 7.9, visualiza-se a comparação entre os ambientes computacionais MATLAB e a implementação no processador NIOS para o operador de *Prewitt*.

(a) (b)

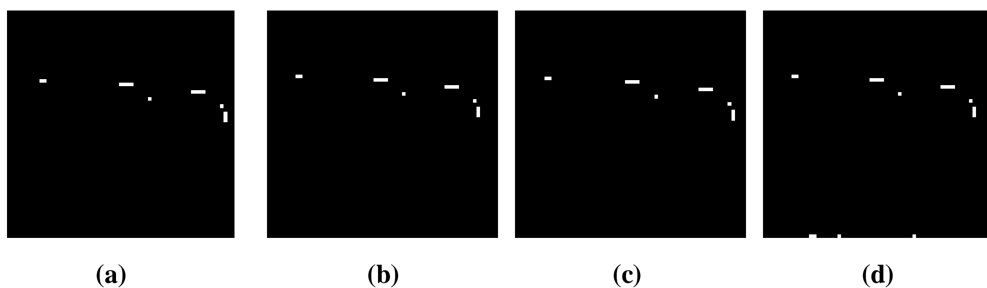
Figura 7. 9. Comparação entre os resultados para o operador de *Prewitt* – (a) MATLAB; (b)NIOS.

Na Figura 7.10, visualiza-se a comparação entre os ambientes computacionais MATLAB e a implementação no processador NIOS para o operador de *Sobel*.

(a) (b)

Figura 7. 10. Comparação entre os resultados para o operador de *Sobel* – (a) MATLAB; (b)NIOS.

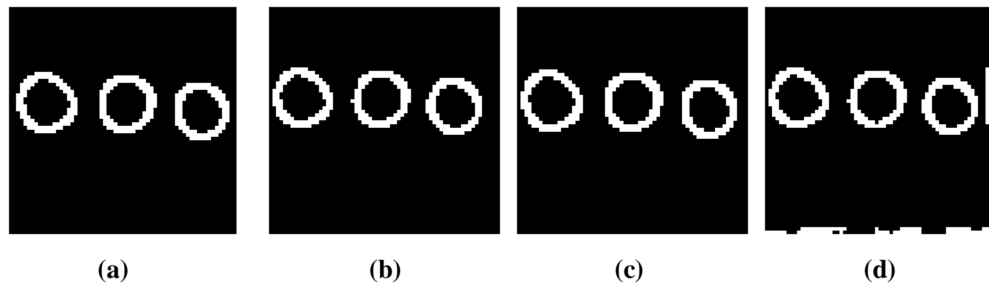
Na Figura 7.11 visualiza-se a comparação entre as imagens geradas para o operador de *Roberts* nos ambientes estudados, ou seja, no MATLAB, nos algoritmos gerados na linguagem C ANSI, no QUARTUS com modelo descrito na linguagem VHDL e a implementação do software gerado na linguagem C ANSI no processador NIOS.



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

Figura 7. 11. Comparação entre os ambientes estudados para o operador de *Roberts* - (a) MATLAB; (b) C ANSI; (c) VHDL; (d) NIOS

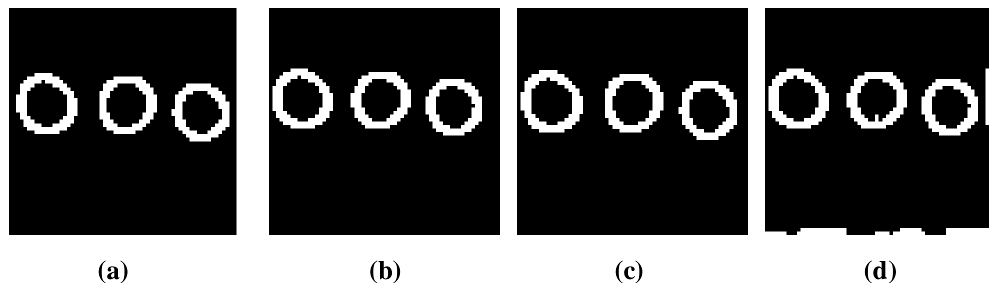
Na Figura 7.12 visualiza-se a comparação entre as imagens geradas para o operador de *Prewitt* nos ambientes estudados.



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

Figura 7. 12. Comparação entre os ambientes estudados para o operador de *Prewitt* - (a) MATLAB; (b) C ANSI; (c) VHDL; (d) NIOS

Na Figura 7.13 visualiza-se a comparação entre as imagens geradas para o operador de *Sobel* nos ambientes estudados.



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

Figura 7. 13. Comparação entre os ambientes estudados para o operador de *Sobel* - (a) MATLAB; (b) C ANSI; (c) VHDL; (d) NIOS

Comparou-se também a porcentagem utilizada na placa FPGA entre o hardware criado modelado com a linguagem VHDL e a implementação no processador NIOS. Com o modelo gerado com a linguagem VHDL teve uma porcentagem menor que 1% e a implementação no processador NIOS teve uma porcentagem de 17%.

A principal contribuição para este trabalho foi a implementação dos operadores de bordas de Roberts, Prewitt e Sobel na placa FPGA e por consequência outras contribuições para a área de processamento de imagens foram obtidas, como as aplicações destes mesmos operadores no MATLAB, na linguagem C ANSI e na linguagem de descrição de hardware VHDL. Outra contribuição foi o estudo da transformada de *Hough* para a contagem de ovos das imagens.

7. 2. Trabalhos Futuros

Um trabalho futuro para continuação deste projeto é melhorar os passos realizados para a implementação na FPGA, agilizando o processo, e descobrir o motivo de estar aparecendo os ruídos nas imagens obtidas como resultado. Aplicar a transformada de *Hough* com a linguagem C ANSI e assim implementar no processador NIOS, e junto com os operadores de bordas de *Sobel* ser aplicado em indústrias para a identificação e contagem de objetos, no caso deste estudo, na contagem e identificação de ovos, agilizando e melhorando o processo.

Referências

ABBASI, T. A.; MOHD, A. U. A proposed FPGA based architecture for *Sobel* edge detection operator. **Journal of Active and Passive Electronic Devices**, v. 2, p. 271 – 277, 2007. Disponível em: <http://74.125.155.132/scholar?q=cache:9YuxiUsP0UAJ:scholar.google.com/+A+proposed+FPGA+based+architecture+for+Sobel+edge+detection+operator&hl=pt-BR&as_sdt=2000&as_vis=1>. Acesso em 20 jun. 2010.

ALMEIDA, M. B. uClinux. **Revista do Linux**. out. 2003. Disponível em : <httpd://RevistaDoLinux.com.br> Acesso em: 1 Out. 2010.

APREEACH, S.C. **Hardware implementation of sobel-edge detection distributed arithmetic digital filter**. p. 284 – 289. 2004. Disponível em: <http://www.a-a-r-s.org/acrs/proceeding/ACRS2004/Papers/AFE04-3.pdf>. Acesso em: 20 jun. 2010.

BENKRID, K. **A high level software environment for FPGA based image processing. Queen´s University of Belfast, UK**. p. 112 – 116. 1999. (Conference Publication No. 465)Disponível em: <http://www.google.com.br/search?q=high+level+software+environment+for+FPGA+based+image+processing.+Queen%C2%B4s+University+of+Belfast&ie=utf-8&oe=utf-8&aq=t&rls=org.mozilla:pt-BR:official&client=firefox-a> Acesso em: 25 jun. 2010.

CROOKES, D; ALOTAIBI, K; BOURIDANE, A; DONACHY P; BANKRID, A. **An environment for generating FPGA architectures for image algebra-based algorithms. Department of Computer Science, The Queen´s University of Belfast**. p. 990 – 994, 2002. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00999082>. Acesso em 20 maio 2010.

GONZALEZ, R. C.; WOODS, Richard E. **Digital image processing**. 2.ed. New Jersey: Prentice-Hall, 2002.

RAO, D. V. An efficient reconfigurable architecture and implementation of edge detection algorithm using Handle-C. University of Nevada Las Vegas. Las Vegas, NV 89154. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY: CODING AND COMPUTING, ITCC´4, 2004, Las Vegas. **Proceedings of the...** Las Vegas: University of Nevada Las Vega, 2004.

SILVA, D. **Aplicação de transformada de hough na detecção de leveduras viáveis e inviáveis.** 2010. Disponível em: <http://www.gpec.ucdb.br/pistori/orientacoes/planos/diogo2010.pdf>. Acesso em: 25 jun. 2010.

XUE, L. FPGA based *Sobel* algorithms as vehicle edge Detector. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01281070>. p. 14-17, 2003. Acesso em: 20 jun. 2010.

ZONGQING, L. **An embedded system with *uClinux* based on FPGA.** p. 691 – 694, 2008. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4756864>. Acesso em: 20 dez. 2009.

APÊNDICE A. Algoritmos criados no ambiente computacional MATLAB.

A. 1. Algoritmo do operador de *Roberts* – roberts.m 99
A. 2. Algoritmo do operador de *Prewitt* – prewitt.m..100
A. 3. Algoritmo do operador de *Sobel* – sobel.m...101
A. 4. Algoritmo para modificar a simulação obtida pelo QUARTUS - vhdl_mat.m 102
A. 5 – Algoritmo da Transformada de *Hough* – teste_hough.m 103

A. 1. Algoritmo do operador de *Roberts* – roberts.m

```
A=input('Digite o nome:  ','s')
[I Map]=imread(A);
E = imresize(I,[256 256]);
B=rgb2gray(E);
%Máscaras dos operadores Roberts
hor = [1 0 0;
       0 -1 0;
       0 0 0];

ver = [0 1 0;
      -1 0 0;
       0 0 0];

for i=1:size(B,1)
 for j=1:size(B,2)
 C(i,j)=double(B(i,j));
 end
end

% Gx = horizontal e Gy = vertical
for i=2:size(B,1)-1
 for j=2:size(B,2)-1

  GX(i,j)    =    (abs(hor(1,1)    *    double(B(i-1,j-1))    +    hor(1,2)    *
  double(B(i-1,j))   +   hor(1,3)   *   double(B(i-1,j+1))   +   hor(2,1)   *
  double(B(i,j-1))    +    hor(2,2)    *    double(B(i,j))    +    hor(2,3)    *
  double(B(i,j+1))   +   hor(3,1)   *   double(B(i+1,j-1))   +   hor(3,2)   *
  double(B(i+1,j))   +  hor(3,3) * double(B(i+1,j+1))));

  GY(i,j)    =    (abs(ver(1,1)    *    double(B(i-1,j-1))    +    ver(1,2)    *
  double(B(i-1,j))   +   ver(1,3)   *   double(B(i-1,j+1))   +   ver(2,1)   *
  double(B(i,j-1))    +    ver(2,2)    *    double(B(i,j))    +    ver(2,3)    *
  double(B(i,j+1))   +   ver(3,1)   *   double(B(i+1,j-1))   +   ver(3,2)   *
  double(B(i+1,j))   +  ver(3,3)* double(B(i+1,j+1))));

  G(i,j) = (abs((double(GX(i,j)) + (double(GY(i,j))))));

 if(G(i,j)>254)
  H(i,j) = (1);
else
  H(i,j) = (0);
end
end
end

imwrite(H,'resultado.bmp','bmp');
imwrite(E,'redimensionado.bmp','bmp');
imwrite(B,'niveis_cinza.bmp','bmp');
figure,imshow(B);
figure,imshow(E);
figure,imshow(H);
dlmwrite('niveis_cinza.txt',B,' ')
dlmwrite('resultado.txt',H,' ')
dlmwrite('redimensionado.txt',E,' ')
```

A. 2. Algoritmo do operador de *Prewitt* – prewitt.m

```
A=input('Digite o nome:  ','s')
[I Map]=imread(A);
E = imresize(I,[256 256]);
B=rgb2gray(E);

%Máscaras dos operadores Prewitt
hor = [1 0 -1;
       1 0 -1;
       1 0 -1];

ver = [-1-1-1;
        0 0 0;
        1 1 1];

for i=1:size(B,1)
 for j=1:size(B,2)
  C(i,j)=double(B(i,j));
 end
end
% Gx = horizontal e Gy = vertical
for i=2:size(B,1)-1
 for j=2:size(B,2)-1

  GX(i,j) = (abs(hor(1,1) * double(B(i-1,j-1)) + hor(1,2) *
  double(B(i-1,j)) + hor(1,3) * double(B(i-1,j+1)) + hor(2,1) *
  double(B(i,j-1)) + hor(2,2) * double(B(i,j)) + hor(2,3) *
  double(B(i,j+1)) + hor(3,1) * double(B(i+1,j-1)) + hor(3,2) *
  double(B(i+1,j)) + hor(3,3) * double(B(i+1,j+1))));

  GY(i,j) = (abs(ver(1,1) * double(B(i-1,j-1)) + ver(1,2) *
  double(B(i-1,j)) + ver(1,3) * double(B(i-1,j+1)) + ver(2,1) *
  double(B(i,j-1)) + ver(2,2) * double(B(i,j)) + ver(2,3) *
  double(B(i,j+1)) + ver(3,1) * double(B(i+1,j-1)) + ver(3,2) *
  double(B(i+1,j)) + ver(3,3) * double(B(i+1,j+1))));

  G(i,j) = (abs((double(GX(i,j)) + (double(GY(i,j))))));

if(G(i,j)>254)
            H(i,j) = (1);
else
 H(i,j) = (0);
end
end
end

imwrite(H,'resultado.bmp','bmp');
imwrite(E,'redimensionado.bmp','bmp');
imwrite(B,'niveis_cinza.bmp','bmp');
figure,imshow(B);
figure,imshow(E);
figure,imshow(H);
dlmwrite('niveis_cinza.txt',B,' ')
dlmwrite('resultado.txt',H,' ')
dlmwrite('redimensionado.txt',E,' ')
```

A. 3. Algoritmo do operador de *Sobel* – sobel.m

```
A=input('Digite o nome:  ','s')
[I Map]=imread(A);
E = imresize(I,[256 256]);
B=rgb2gray(E);

%Máscaras dos operadores Sobel
hor = [-1-2-1;
        0 0 0;
        1 2 1];

ver = [-1 0 1;
       -2 0 2;
       -1 0 1];

for i=1:size(B,1)
 for j=1:size(B,2)
  C(i,j)=double(B(i,j));
 end
end
% Gx = horizontal e Gy = vertical
for i=2:size(B,1)-1
 for j=2:size(B,2)-1

   GX(i,j) = (abs(hor(1,1) * double(B(i-1,j-1)) + hor(1,2) *
   double(B(i-1,j)) + hor(1,3) * double(B(i-1,j+1)) + hor(2,1) *
   double(B(i,j-1)) + hor(2,2) * double(B(i,j)) + hor(2,3) *
   double(B(i,j+1)) + hor(3,1) * double(B(i+1,j-1)) + hor(3,2) *
   double(B(i+1,j)) + hor(3,3) * double(B(i+1,j+1))));

   GY(i,j) = (abs(ver(1,1) * double(B(i-1,j-1)) + ver(1,2) *
   double(B(i-1,j)) + ver(1,3) * double(B(i-1,j+1)) + ver(2,1) *
   double(B(i,j-1)) + ver(2,2) * double(B(i,j)) + ver(2,3) *
   double(B(i,j+1)) + ver(3,1) * double(B(i+1,j-1)) + ver(3,2) *
   double(B(i+1,j)) + ver(3,3) * double(B(i+1,j+1))));

   G(i,j) = (abs((double(GX(i,j)) + (double(GY(i,j))))));

if(G(i,j)>254)
  H(i,j) = (1);
else
 H(i,j) = (0);
end
end
end

imwrite(H,'resultado.bmp','bmp');
imwrite(E,'redimensionado.bmp','bmp');
imwrite(B,'niveis_cinza.bmp','bmp');
figure,imshow(B);
figure,imshow(E);
figure,imshow(H);
dlmwrite('niveis_cinza.txt',B,' ')
dlmwrite('resultado.txt',H,' ')
dlmwrite('redimensionado.txt',E,' ')
```

A. 4. Algoritmo para modificar a simulação obtida pelo QUARTUS - vhdl_mat.m

```
a=textdata;
b=data;
for i=1:size(a,1)
 if (a{i,2}=='0')
  m(i,1)=2;
 else
  if (a{i,2}=='1')
  if (b(i,1)==0)
   m(i,1)=0;
  else
   m(i,1)=1;
  end
  end
 end
end

(dlmwrite('nulo.txt',m,''));
```

A. 5 – Algoritmo da Transformada de *Hough* – teste_hough.m

```
A=input('Digite o nome:  ','s')
[BW Map]=imread(A);
%BW  = imread('ovo5.bmp');
figure, imshow(BW);

%Compute the Hough transform of the image using the hough function.
[H,theta,rho] = hough(BW);
%figure, imshow(H);
c = hough(BW);
%Display the transform using the imshow function.
figure,
imshow(imadjust(mat2gray(H)),[],'XData',theta,'YData',rho,'InitialMagnifica
tion','fit');
xlabel('\theta (degrees)'), ylabel('\rho');
axis on, axis normal, hold on;
colormap(hot)

%Find the peaks in the Hough transform matrix, H, using the houghpeaks
function.
P = houghpeaks(H,5,'threshold',ceil(0.3*max(H(:))));

%Superimpose a plot on the image of the transform that identifies the
peaks.
x = theta(P(:,2));
y = rho(P(:,1));
plot(x,y,'s','color','black');

%Find lines in the image using the houghlines function.
lines = houghlines(BW,theta,rho,P,'FillGap',5,'MinLength',7);
```

APÊNDICE B. Algoritmos criados no ambiente computacional Dev-C++

B. 1 - Algoritmo do operador *Roberts* - roberts.c 105
B. 2 - Algoritmo do operador *Prewitt* - prewitt.c 107
B. 3 - Algoritmo do operador *Sobel* - sobel.c 109
B. 4 - Algoritmos do operador *Roberts* 111
B.4. 1 - nios_roberts.c 111
B.4. 2 - mat_inicial.c 112
B.4. 3 - mat1.c 114
B.4. 4 - mat2.c 116
B.4. 5 - mat3.c 118
B.4. 6 - mat4.c 120
B.4. 7 - mat_final.c 122
B. 5 - Algoritmos do operador *Prewitt* 124
B.5. 1- nios_prewitt.c 124
B.5. 2 - mat_inicial 125
B.5. 3 - mat1.c 127
B.5. 4 - mat2.c 129
B.5. 5 - mat3.c 131
B.5. 6 - mat4.c 133
B.5. 7 - mat_final.c 135
B. 6 - Algoritmos do operador *Sobel* 137
B.6. 1- nios_sobel.c 137
B.6. 2 - mat_inicial.c 138
B.6. 3 - mat1.c 140
B.6. 4 - mat2.c 142
B.6. 5 - mat3.c 144
B.6. 6 - mat4.c 146
B.6. 7 - mat_final.c 148
B. 7 - Algoritmo para alteração de arquivos para a linguagem VHDL – alteração_arquivo.c 150
B. 8 - Algoritmos para alteração da simulação obtida pelo QUARTUS 151
B.8. 1- teste_nulo.c 151
B.8. 2 - img_quartus.c 152
B.8. 3 - quartus_c.c 153

B. 1. Algoritmo do operador *Roberts* - roberts.c

```
#include <stdio.h>
#include <stdlib.h>
main()
{
 FILE *p;
 FILE *a;
 int MX[3][3];
 int MY[3][3];
 char i[256];
 int GX[256][256];
 int GY[256][256];
 int H[256][256];
 int G[256][256];
 int n;
 int mat[256][256];
 int l,c;
 //horizontal
  MX[0][0] = 1;
  MX[0][1] = 0;
  MX[0][2] = 0;
  MX[1][0] = 0;
  MX[1][1] = -1;
  MX[1][2] = 0;
  MX[2][0] = 0;
  MX[2][1] = 0;
  MX[2][2] = 0;
 //vertical
  MY[0][0] = 0;
  MY[0][1] = 1;
  MY[0][2] = 0;
  MY[1][0] = -1;
  MY[1][1] = 0;
  MY[1][2] = 0;
  MY[2][0] = 0;
  MY[2][1] = 0;
  MY[2][2] = 0;

  if((p = fopen("imagem2.txt","r")) == NULL)
  {
   printf("Erro ao abrir arquivo!!!\n");
  exit(1);
  }
  if((a = fopen("arq1.txt","w")) == NULL)
  {
   printf("Erro ao abrir arquivo!!!\n");
   exit(1);
  }


  for(l = 0;l < 256;l++)
  {
   for(c = 0;c < 256; c++)
   {
    fscanf(p,"%s",&i);

    n=atol(i);

    mat[l][c]=n;
   }
```

```
}
for(l = 0;l < 256;l++)
{
 for(c = 0;c < 256; c++)
 {

  GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
  ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
  ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
  ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
  ((MX[2][2]) * (mat[l+2][c+2]));

  GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
  ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
  ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
  ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
  ((MY[2][2]) * (mat[l+2][c+2]));

  if (GX[l][c]<0)
  {
   GX[l][c]=GX[l][c]*(-1);
  }
  if (GY[l][c]<0)
  {
   GY[l][c]=GY[l][c]*(-1);
  }

  G[l][c]=(GX[l][c])+(GY[l][c]);

  if (G[l][c]<0)
  {
   G[l][c]=G[l][c]*(-1);
  }
  if(G[l][c]>254)
  {
   H[l][c]=(1);}
   else
   {
    H[l][c]=(0);
   }

   fprintf(a, "%d ", H[l][c]);

   if (c == 255)
   {
    fprintf(a,"%s","\n");
   }
  }
  }

  fclose(p);
  fclose(a);
  }
```

B. 2. Algoritmo do operador *Prewitt* - prewitt.c

```
#include <stdio.h>
#include <stdlib.h>
main()
{
 FILE *p;
 FILE *a;
 int MX[3][3];
 int MY[3][3];
 char i[256];
 int GX[256][256];
 int GY[256][256];
 int H[256][256];
 int G[256][256];
 int n;
 int mat[256][256];
 int l,c;
 //horizontal
 MX[0][0] = 1;
 MX[0][1] = 0;
 MX[0][2] = -1;
 MX[1][0] = 1;
 MX[1][1] = 0;
 MX[1][2] = -1;
 MX[2][0] = 1;
 MX[2][1] = 0;
 MX[2][2] = -1;
 //vertical
 MY[0][0] = -1;
 MY[0][1] = -1;
 MY[0][2] = -1;
 MY[1][0] = 0;
 MY[1][1] = 0;
 MY[1][2] = 0;
 MY[2][0] = 1;
 MY[2][1] = 1;
 MY[2][2] = 1;

 if((p = fopen("imagem2.txt","r")) == NULL)
 {
  printf("Erro ao abrir arquivo!!!\n");
  exit(1);
 }
 if((a = fopen("arq1.txt","w")) == NULL)
 {
  printf("Erro ao abrir arquivo!!!\n");
  exit(1);
 }

 for(l = 0;l < 256;l++)
 {
  for(c = 0;c < 256; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
  }
 }
```

```
for(l = 0;l < 256;l++)
{
 for(c = 0;c < 256; c++)
 {
  GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
  ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
  ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
  ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
  ((MX[2][2]) * (mat[l+2][c+2]));

  GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
  ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
  ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
  ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
  ((MY[2][2]) * (mat[l+2][c+2]));

  if (GX[l][c]<0)
  {
   GX[l][c]=GX[l][c]*(-1);
  }
  if (GY[l][c]<0)
  {
   GY[l][c]=GY[l][c]*(-1);
  }
  G[l][c]=(GX[l][c])+(GY[l][c]);

  if (G[l][c]<0)
  {
   G[l][c]=G[l][c]*(-1);
  }
  if(G[l][c]>254)
  {
   H[l][c]=(1);}
   else
   {
    H[l][c]=(0);
   }

   fprintf(a, "%d ", H[l][c]);

   if (c == 255)
   {
    fprintf(a,"%s","\n");
   }
  }
  }
fclose(p);
fclose(a);
}
```

B. 3. Algoritmo do operador *Sobel* - sobel.c

```
#include <stdio.h>
#include <stdlib.h>
main()
{
 FILE *p;
 FILE *a;
 int MX[3][3];
 int MY[3][3];
 char i[256];
 int GX[256][256];
 int GY[256][256];
 int H[256][256];
 int G[256][256];
 int n;
 int mat[256][256];
 int l,c;
 //horizontal
 MX[0][0] = -1;
 MX[0][1] = -2;
 MX[0][2] = -1;
 MX[1][0] = 0;
 MX[1][1] = 0;
 MX[1][2] = 0;
 MX[2][0] = 1;
 MX[2][1] = 2;
 MX[2][2] = 1;
 //vertical
 MY[0][0] = -1;
 MY[0][1] = 0;
 MY[0][2] = 1;
 MY[1][0] = -2;
 MY[1][1] = 0;
 MY[1][2] = 2;
 MY[2][0] = -1;
 MY[2][1] = 0;
 MY[2][2] = 1;

 if((p = fopen("imagem2.txt","r")) == NULL)
 {
  printf("Erro ao abrir arquivo!!!\n");
  exit(1);
 }
 if((a = fopen("arq1.txt","w")) == NULL)
 {
  printf("Erro ao abrir arquivo!!!\n");
  exit(1);
 }

 for(l = 0;l < 256;l++)
 {
  for(c = 0;c < 256; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
  }
 }
```

```
 for(l = 0;l < 256;l++)
 {
  for(c = 0;c < 256; c++)
  {
   GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
   ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
   ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
   ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
   ((MX[2][2]) * (mat[l+2][c+2]));

   GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
   ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
   ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
   ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
   ((MY[2][2]) * (mat[l+2][c+2]));

   if (GX[l][c]<0)
   {
    GX[l][c]=GX[l][c]*(-1);
   }
   if (GY[l][c]<0)
   {
    GY[l][c]=GY[l][c]*(-1);
   }
   G[l][c]=(GX[l][c])+(GY[l][c]);

   if (G[l][c]<0)
   {
    G[l][c]=G[l][c]*(-1);
   }
   if(G[l][c]>254)
   {
    H[l][c]=(1);
   }
   else
   {
    H[l][c]=(0);
   }

   fprintf(a, "%d ", H[l][c]);

   if (c == 255)
   {
    fprintf(a,"%s","\n");
   }
  }
 }

 fclose(p);
 fclose(a);
}
```

B. 4. Algoritmos do operador *Roberts*

B.4. 1. nios_roberts.c

```
#include<stdio.h>
#include"mat1.c"
#include"mat2.c"
#include"mat3.c"
#include"mat4.c"
#include"mat_final.c"
#include"mat_inicial.c"

int main ()
{
 printf("Gerando as matrizes iniciais:\n");
 roberts_inicial();
 printf("Gerando a primeira matriz:\n");
 roberts1();
 printf("Gerando a segunda matriz:\n");
 roberts2();
 printf("Gerando a terceira matriz:\n");
 roberts3();
 printf("Gerando a quarta matriz:\n");
 roberts4();
 printf("Gerando a matriz final:\n");
 roberts_final();
}
```

B.4. 2. mat_inicial.c

```
#include <stdio.h>
#include <stdlib.h>

int roberts_inicial()
{
 FILE *p;
 FILE *a;
 FILE *b;
 FILE *d;
 FILE *e;
 char i[4];
 short int n;
 short int mat[64][64];
 short int l,c;

 p = fopen("niveis_cinza.txt","r");
 a = fopen("imagem1.txt","w");
 b = fopen("imagem2.txt","w");
 d = fopen("imagem3.txt","w");
 e = fopen("imagem4.txt","w");

 for(l = 0;l < 64;l++)
 {
  for(c = 0;c < 64; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
for(l = 0;l < 34;l++)
{
 for(c = 0;c < 34; c++)
 {
  fprintf(a, "%i ", mat[l][c]);

  if (c == 33)
  {
   fprintf(a,"%s","\n");
  }
 }
}
for(l = 0;l < 34;l++)
{
 for(c = 32;c < 64; c++)
 {
  fprintf(b, "%i ", mat[l][c]);

  if (c == 63)
  {
   fprintf(b,"%s","\n");
  }
 }
}
```

```
for(l = 32;l < 64;l++)
{
 for(c = 0;c < 34; c++)
  {
   fprintf(d, "%i ", mat[l][c]);

   if (c == 33)
   {
    fprintf(d,"%s","\n");
   }
  }
}
for(l = 32;l < 64;l++)
{
 for(c = 32;c < 64; c++)
  {
   fprintf(e, "%i ", mat[l][c]);

   if (c == 63)
   {
    fprintf(e,"%s","\n");
   }
  }
}
fclose(p);
fclose(a);
fclose(b);
fclose(d);
fclose(e);
}
```

B.4. 3. mat1.c

```
#include <stdio.h>
#include <stdlib.h>
#define x 32

int roberts1()
{
 FILE *p;
 FILE *b;
 short int MX[3][3];
 short int MY[3][3];
 char i[32];
 short int GX[x][x];
 short int GY[x][x];
 short int H[x][x];
 short int G[x][x];
 short int n;
 short int mat[34][34];
 short int l,c;
 //horizontal
 MX[0][0] = 1;
 MX[0][1] = 0;
 MX[0][2] = 0;
 MX[1][0] = 0;
 MX[1][1] = -1;
 MX[1][2] = 0;
 MX[2][0] = 0;
 MX[2][1] = 0;
 MX[2][2] = 0;
 //vertical
 MY[0][0] = 0;
 MY[0][1] = 1;
 MY[0][2] = 0;
 MY[1][0] = -1;
 MY[1][1] = 0;
 MY[1][2] = 0;
 MY[2][0] = 0;
 MY[2][1] = 0;
 MY[2][2] = 0;

 p = fopen("imagem1.txt","r");
 b = fopen("arq_H1.txt","w");

 for(l = 0;l < 34;l++)
 {
  for(c = 0;c < 34; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
```

```
 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {

   GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
   ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
   ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
   ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
   ((MX[2][2]) * (mat[l+2][c+2]));

   GY[l][c]=((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
   ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
   ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
   ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
   ((MY[2][2]) * (mat[l+2][c+2]));

   if (GX[l][c]<0)
   {
    GX[l][c]=GX[l][c]*(-1);
   }
   if (GY[l][c]<0)
   {
    GY[l][c]=GY[l][c]*(-1);
   }
   G[l][c]=(GX[l][c])+(GY[l][c]);

   if (G[l][c]<0)
   {
    G[l][c]=G[l][c]*(-1);
   }
   if(G[l][c]>254)
   {
    H[l][c]=(1);}
    else
    {
     H[l][c]=(0);
    }
    fprintf(b, "%i ", H[l][c]);

    if (c == 31)
    {
     fprintf(b,"%s","\n");
    }
   }
  }
 fclose(p);
 fclose(b);
}
```

B.4. 4. mat2.c

```
#include <stdio.h>
#include <stdlib.h>
#define x 32

int roberts2()
{
 FILE *p;
 FILE *b;
 short int MX[3][3];
 short int MY[3][3];
 char i[4];
 short int GX[x][x];
 short int GY[x][x];
 short int H[x][x];
 short int G[x][x];
 short int n;
 short int mat[34][x];
 short int l,c;
 //horizontal
 MX[0][0] = 1;
 MX[0][1] = 0;
 MX[0][2] = 0;
 MX[1][0] = 0;
 MX[1][1] = -1;
 MX[1][2] = 0;
 MX[2][0] = 0;
 MX[2][1] = 0;
 MX[2][2] = 0;
 //vertical
 MY[0][0] = 0;
 MY[0][1] = 1;
 MY[0][2] = 0;
 MY[1][0] = -1;
 MY[1][1] = 0;
 MY[1][2] = 0;
 MY[2][0] = 0;
 MY[2][1] = 0;
 MY[2][2] = 0;

 p = fopen("imagem2.txt","r");
 b = fopen("arq_H2.txt","w");

 for(l = 0;l < 34;l++)
 {
  for(c = 0;c < 32; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
```

```
 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {
   GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
   ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
   ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
   ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
   ((MX[2][2]) * (mat[l+2][c+2]));

   GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
   ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
   ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
   ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
   ((MY[2][2]) * (mat[l+2][c+2]));

   if (GX[l][c]<0)
   {
    GX[l][c]=GX[l][c]*(-1);
   }
   if (GY[l][c]<0)
   {
    GY[l][c]=GY[l][c]*(-1);
   }
   G[l][c]=(GX[l][c])+(GY[l][c]);

   if (G[l][c]<0)
   {
    G[l][c]=G[l][c]*(-1);
   }
   if(G[l][c]>254)
   {
    H[l][c]=(1);
   }
    else
    {
     H[l][c]=(0);
    }
    fprintf(b, "%i ", H[l][c]);

    if (c == 31)
    {
     fprintf(b,"%s","\n");
    }
   }
  }

fclose(p);
fclose(b);
}
```

B.4. 5. mat3.c

```
#include <stdio.h>
#include <stdlib.h>
#define x 32

int roberts3()
{
 FILE *p;
 FILE *b;
 short int MX[3][3];
 short int MY[3][3];
 char i[4];
 short int GX[x][x];
 short int GY[x][x];
 short int H[x][x];
 short int G[x][x];
 short int n;
 short int mat[x][34];
 short int l,c;
 //horizontal
 MX[0][0] = 1;
 MX[0][1] = 0;
 MX[0][2] = 0;
 MX[1][0] = 0;
 MX[1][1] = -1;
 MX[1][2] = 0;
 MX[2][0] = 0;
 MX[2][1] = 0;
 MX[2][2] = 0;
 //vertical
 MY[0][0] = 0;
 MY[0][1] = 1;
 MY[0][2] = 0;
 MY[1][0] = -1;
 MY[1][1] = 0;
 MY[1][2] = 0;
 MY[2][0] = 0;
 MY[2][1] = 0;
 MY[2][2] = 0;

 p = fopen("imagem3.txt","r");
 b = fopen("arq_H3.txt","w");

 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 34; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
```

```
 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {

    GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
    ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
    ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
    ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
    ((MX[2][2]) * (mat[l+2][c+2]));

    GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
    ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
    ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
    ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
    ((MY[2][2]) * (mat[l+2][c+2]));

    if (GX[l][c]<0)
    {
     GX[l][c]=GX[l][c]*(-1);
    }
    if (GY[l][c]<0)
    {
     GY[l][c]=GY[l][c]*(-1);
    }
    G[l][c]=(GX[l][c])+(GY[l][c]);

    if (G[l][c]<0)
    {
     G[l][c]=G[l][c]*(-1);
    }
    if(G[l][c]>254)
    {
     H[l][c]=(1);
    }
     else
     {
      H[l][c]=(0);
     }
     fprintf(b, "%i ", H[l][c]);

     if (c == 31)
     {
      fprintf(b,"%s","\n");
     }
    }
  }

fclose(p);
fclose(b);
}
```

B.4. 6. mat4.c

```
#include <stdio.h>
#include <stdlib.h>
#define x 32

int roberts4()
{
 FILE *p;
 FILE *b;
 short int MX[3][3];
 short int MY[3][3];
 char i[4];
 short int GX[x][x];
 short int GY[x][x];
 short int H[x][x];
 short int G[x][x];
 short int n;
 short int mat[x][x];
 short int l,c;
 //horizontal
 MX[0][0] = 1;
 MX[0][1] = 0;
 MX[0][2] = 0;
 MX[1][0] = 0;
 MX[1][1] = -1;
 MX[1][2] = 0;
 MX[2][0] = 0;
 MX[2][1] = 0;
 MX[2][2] = 0;
 //vertical
 MY[0][0] = 0;
 MY[0][1] = 1;
 MY[0][2] = 0;
 MY[1][0] = -1;
 MY[1][1] = 0;
 MY[1][2] = 0;
 MY[2][0] = 0;
 MY[2][1] = 0;
 MY[2][2] = 0;

 p = fopen("imagem4.txt","r");
 b = fopen("arq_H4.txt","w");

 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
```

```
 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {
   GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
   ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
   ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
   ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
   ((MX[2][2]) * (mat[l+2][c+2]));

   GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
   ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
   ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
   ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
   ((MY[2][2]) * (mat[l+2][c+2]));

   if (GX[l][c]<0)
   {
    GX[l][c]=GX[l][c]*(-1);
   }
   if (GY[l][c]<0)
   {
    GY[l][c]=GY[l][c]*(-1);
   }
   G[l][c]=(GX[l][c])+(GY[l][c]);

   if (G[l][c]<0)
   {
    G[l][c]=G[l][c]*(-1);
   }
   if(G[l][c]>254)
   {
    H[l][c]=(1);
   }
    else
    {
     H[l][c]=(0);
    }
    fprintf(b, "%i ", H[l][c]);

    if (c == 31)
    {
     fprintf(b,"%s","\n");
    }
   }
  }

fclose(p);
fclose(b);
}
```

B.4. 7. mat_final.c

```
#include <stdio.h>
#include <stdlib.h>


int roberts_final()
{
 FILE *p;
 FILE *a;
 FILE *b;
 FILE *d;
 FILE *e;

 char i[4];
 short int H[66][66];
 short int n;
 short int mat[66][66];
 short int l,c;

 p = fopen("arq_H1.txt","r");
 a = fopen("arq_H2.txt","r");
 d = fopen("arq_H3.txt","r");
 e = fopen("arq_H4.txt","r");
 b = fopen("arq_H_final.txt","w");

 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }

 for(l = 0;l < 32;l++)
 {
  for(c = 32;c < 64; c++)
  {
   fscanf(a,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }

 for(l = 32;l < 64;l++)
 {
  for(c = 0;c < 32; c++)
  {
   fscanf(d,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
```

```
for(l = 32;l < 64;l++)
{
 for(c = 32;c < 64; c++)
 {
  fscanf(e,"%s",&i);
  n=atol(i);
  mat[l][c]=n;
  printf("%i\n",mat[l][c]);
 }
}

for(l = 0;l < 64;l++)
{
 for(c = 0;c < 64; c++)
 {
  fprintf(b, "%i ", mat[l][c]);

  if (c == 63)
  {
   fprintf(b,"%s","\n");
  }
 }
}

fclose(p);
fclose(a);
fclose(b);
fclose(d);
fclose(e);
}
```

B. 5. Algoritmos do operador *Prewitt*

B.5. 1. nios_prewitt.c

```
#include<stdio.h>
#include"mat1.c"
#include"mat2.c"
#include"mat3.c"
#include"mat4.c"
#include"mat_final.c"
#include"mat_inicial.c"

int main ()
{
 printf("Gerando as matrizes iniciais:\n");
 prewitt_inicial();
 printf("Gerando a primeira matriz:\n");
 prewitt1();
 printf("Gerando a segunda matriz:\n");
 prewitt2();
 printf("Gerando a terceira matriz:\n");
 prewitt3();
 printf("Gerando a quarta matriz:\n");
 prewitt4();
 printf("Gerando a matriz final:\n");
 prewitt_final();
}
```

B.5. 2. mat_inicial

```
#include <stdio.h>
#include <stdlib.h>

int prewitt_inicial()
{
 FILE *p;
 FILE *a;
 FILE *b;
 FILE *d;
 FILE *e;
 char i[4];
 short int n;
 short int mat[64][64];
 short int l,c;

 p = fopen("niveis_cinza.txt","r");
 a = fopen("imagem1.txt","w");
 b = fopen("imagem2.txt","w");
 d = fopen("imagem3.txt","w");
 e = fopen("imagem4.txt","w");

 for(l = 0;l < 64;l++)
 {
  for(c = 0;c < 64; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
for(l = 0;l < 34;l++)
{
 for(c = 0;c < 34; c++)
 {
  fprintf(a, "%i ", mat[l][c]);

  if (c == 33)
  {
   fprintf(a,"%s","\n");
  }
 }
}
for(l = 0;l < 34;l++)
{
 for(c = 32;c < 64; c++)
 {
  fprintf(b, "%i ", mat[l][c]);

  if (c == 63)
  {
   fprintf(b,"%s","\n");
  }
 }
}
```

```
for(l = 32;l < 64;l++)
{
 for(c = 0;c < 34; c++)
  {
   fprintf(d, "%i ", mat[l][c]);

   if (c == 33)
   {
    fprintf(d,"%s","\n");
   }
  }
}
for(l = 32;l < 64;l++)
{
 for(c = 32;c < 64; c++)
  {
   fprintf(e, "%i ", mat[l][c]);

   if (c == 63)
   {
    fprintf(e,"%s","\n");
   }
  }
}
fclose(p);
fclose(a);
fclose(b);
fclose(d);
fclose(e);
}
```

B.5. 3. mat1.c

```
#include <stdio.h>
#include <stdlib.h>
#define x 32

int prewitt1()
{
FILE *p;
FILE *b;
short int MX[3][3];
short int MY[3][3];
char i[32];
short int GX[x][x];
short int GY[x][x];
short int H[x][x];
short int G[x][x];
short int n;
short int mat[34][34];
short int l,c;
//horizontal
MX[0][0] = 1;
MX[0][1] = 0;
MX[0][2] = -1;
MX[1][0] = 1;
MX[1][1] = 0;
MX[1][2] = -1;
MX[2][0] = 1;
MX[2][1] = 0;
MX[2][2] = -1;
//vertical
MY[0][0] = -1;
MY[0][1] = -1;
MY[0][2] = -1;
MY[1][0] = 0;
MY[1][1] = 0;
MY[1][2] = 0;
MY[2][0] = 1;
MY[2][1] = 1;
MY[2][2] = 1;

p = fopen("imagem1.txt","r");
b = fopen("arq_H1.txt","w");

 for(l = 0;l < 34;l++)
 {
  for(c = 0;c < 34; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
```

```
 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {

    GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
    ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
    ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
    ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
    ((MX[2][2]) * (mat[l+2][c+2]));

    GY[l][c]=((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
    ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
    ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
    ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
    ((MY[2][2]) * (mat[l+2][c+2]));

    if (GX[l][c]<0)
    {
     GX[l][c]=GX[l][c]*(-1);
    }
    if (GY[l][c]<0)
    {
     GY[l][c]=GY[l][c]*(-1);
    }
    G[l][c]=(GX[l][c])+(GY[l][c]);

    if (G[l][c]<0)
    {
     G[l][c]=G[l][c]*(-1);
    }
    if(G[l][c]>254)
    {
     H[l][c]=(1);}
     else
     {
      H[l][c]=(0);
     }
     fprintf(b, "%i ", H[l][c]);

     if (c == 31)
     {
      fprintf(b,"%s","\n");
     }
    }
   }
 fclose(p);
 fclose(b);
}
```

B.5. 4. mat2.c

```
#include <stdio.h>
#include <stdlib.h>
#define x 32

int prewitt2()
{
 FILE *p;
 FILE *b;
 short int MX[3][3];
 short int MY[3][3];
 char i[4];
 short int GX[x][x];
 short int GY[x][x];
 short int H[x][x];
 short int G[x][x];
 short int n;
 short int mat[34][x];
 short int l,c;
 //horizontal
MX[0][0] = 1;
MX[0][1] = 0;
MX[0][2] = -1;
MX[1][0] = 1;
MX[1][1] = 0;
MX[1][2] = -1;
MX[2][0] = 1;
MX[2][1] = 0;
MX[2][2] = -1;
//vertical
MY[0][0] = -1;
MY[0][1] = -1;
MY[0][2] = -1;
MY[1][0] = 0;
MY[1][1] = 0;
MY[1][2] = 0;
MY[2][0] = 1;
MY[2][1] = 1;
MY[2][2] = 1;


  p = fopen("imagem2.txt","r");
  b = fopen("arq_H2.txt","w");

  for(l = 0;l < 34;l++)
  {
   for(c = 0;c < 32; c++)
   {
    fscanf(p,"%s",&i);
    n=atol(i);
    mat[l][c]=n;
    printf("%i\n",mat[l][c]);
   }
  }
```

```
for(l = 0;l < 32;l++)
{
 for(c = 0;c < 32; c++)
 {
  GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
  ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
  ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
  ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
  ((MX[2][2]) * (mat[l+2][c+2]));

  GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
  ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
  ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
  ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
  ((MY[2][2]) * (mat[l+2][c+2]));

  if (GX[l][c]<0)
  {
   GX[l][c]=GX[l][c]*(-1);
  }
  if (GY[l][c]<0)
  {
   GY[l][c]=GY[l][c]*(-1);
  }
  G[l][c]=(GX[l][c])+(GY[l][c]);

  if (G[l][c]<0)
  {
   G[l][c]=G[l][c]*(-1);
  }
  if(G[l][c]>254)
  {
   H[l][c]=(1);
  }
   else
   {
    H[l][c]=(0);
   }
   fprintf(b, "%i ", H[l][c]);

   if (c == 31)
   {
    fprintf(b,"%s","\n");
   }
  }
 }

fclose(p);
fclose(b);
}
```

B.5. 5. mat3.c

```
#include <stdio.h>
#include <stdlib.h>
#define x 32

int prewitt3(){
FILE *p;
FILE *b;
short int MX[3][3];
short int MY[3][3];
char i[4];
short int GX[x][x];
short int GY[x][x];
short int H[x][x];
short int G[x][x];
short int n;
short int mat[x][34];
short int l,c;
//horizontal
MX[0][0] = 1;
MX[0][1] = 0;
MX[0][2] = -1;
MX[1][0] = 1;
MX[1][1] = 0;
MX[1][2] = -1;
MX[2][0] = 1;
MX[2][1] = 0;
MX[2][2] = -1;
//vertical
MY[0][0] = -1;
MY[0][1] = -1;
MY[0][2] = -1;
MY[1][0] = 0;
MY[1][1] = 0;
MY[1][2] = 0;
MY[2][0] = 1;
MY[2][1] = 1;
MY[2][2] = 1;

 p = fopen("imagem3.txt","r");
 b = fopen("arq_H3.txt","w");

 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 34; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
```

```
 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {

    GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
    ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
    ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
    ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
    ((MX[2][2]) * (mat[l+2][c+2]));

    GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
    ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
    ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
    ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
    ((MY[2][2]) * (mat[l+2][c+2]));

    if (GX[l][c]<0)
    {
     GX[l][c]=GX[l][c]*(-1);
    }
    if (GY[l][c]<0)
    {
     GY[l][c]=GY[l][c]*(-1);
    }
    G[l][c]=(GX[l][c])+(GY[l][c]);

    if (G[l][c]<0)
    {
     G[l][c]=G[l][c]*(-1);
    }
    if(G[l][c]>254)
    {
     H[l][c]=(1);
    }
     else
     {
      H[l][c]=(0);
     }
     fprintf(b, "%i ", H[l][c]);

     if (c == 31)
     {
      fprintf(b,"%s","\n");
     }
    }
  }

fclose(p);
fclose(b);
}
```

B.5. 6. mat4.c

```
#include <stdio.h>
#include <stdlib.h>
#define x 32

int prewitt4(){
FILE *p;
FILE *b;
short int MX[3][3];
short int MY[3][3];
char i[4];
short int GX[x][x];
short int GY[x][x];
short int H[x][x];
short int G[x][x];
short int n;
short int mat[x][x];
short int l,c;
//horizontal
MX[0][0] = 1;
MX[0][1] = 0;
MX[0][2] = -1;
MX[1][0] = 1;
MX[1][1] = 0;
MX[1][2] = -1;
MX[2][0] = 1;
MX[2][1] = 0;
MX[2][2] = -1;
//vertical
MY[0][0] = -1;
MY[0][1] = -1;
MY[0][2] = -1;
MY[1][0] = 0;
MY[1][1] = 0;
MY[1][2] = 0;
MY[2][0] = 1;
MY[2][1] = 1;
MY[2][2] = 1;

 p = fopen("imagem4.txt","r");
 b = fopen("arq_H4.txt","w");

 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }




 for(l = 0;l < 32;l++)
 {
```

```
  for(c = 0;c < 32; c++)
  {
   GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
   ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
   ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
   ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
   ((MX[2][2]) * (mat[l+2][c+2]));

   GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
   ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
   ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
   ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
   ((MY[2][2]) * (mat[l+2][c+2]));

   if (GX[l][c]<0)
   {
    GX[l][c]=GX[l][c]*(-1);
   }
   if (GY[l][c]<0)
   {
    GY[l][c]=GY[l][c]*(-1);
   }
   G[l][c]=(GX[l][c])+(GY[l][c]);

   if (G[l][c]<0)
   {
    G[l][c]=G[l][c]*(-1);
   }
   if(G[l][c]>254)
   {
    H[l][c]=(1);
   }
    else
    {
     H[l][c]=(0);
    }
    fprintf(b, "%i ", H[l][c]);

    if (c == 31)
    {
     fprintf(b,"%s","\n");
    }
   }
  }

fclose(p);
fclose(b);
}
```

B.5. 7. mat_final.c

```
#include <stdio.h>
#include <stdlib.h>

int prewitt_final()
{
 FILE *p;
 FILE *a;
 FILE *b;
 FILE *d;
 FILE *e;

 char i[4];
 short int H[66][66];
 short int n;
 short int mat[66][66];
 short int l,c;

 p = fopen("arq_H1.txt","r");
 a = fopen("arq_H2.txt","r");
 d = fopen("arq_H3.txt","r");
 e = fopen("arq_H4.txt","r");
 b = fopen("arq_H_final.txt","w");

 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }

 for(l = 0;l < 32;l++)
 {
  for(c = 32;c < 64; c++)
  {
   fscanf(a,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }

 for(l = 32;l < 64;l++)
 {
  for(c = 0;c < 32; c++)
  {
   fscanf(d,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
```

```
for(l = 32;l < 64;l++)
{
 for(c = 32;c < 64; c++)
 {
  fscanf(e,"%s",&i);
  n=atol(i);
  mat[l][c]=n;
  printf("%i\n",mat[l][c]);
 }
}

for(l = 0;l < 64;l++)
{
 for(c = 0;c < 64; c++)
 {
  fprintf(b, "%i ", mat[l][c]);

  if (c == 63)
  {
   fprintf(b,"%s","\n");
  }
 }
}

fclose(p);
fclose(a);
fclose(b);
fclose(d);
fclose(e);
}
```

B. 6. Algoritmos do operador *Sobel*

B.6. 1. nios_sobel.c

```
#include<stdio.h>
#include"mat1.c"
#include"mat2.c"
#include"mat3.c"
#include"mat4.c"
#include"mat_final.c"
#include"mat_inicial.c"

int main ()
{
 printf("Gerando as matrizes iniciais:\n");
 sobel_inicial();
 printf("Gerando a primeira matriz:\n");
 sobel1();
 printf("Gerando a segunda matriz:\n");
 sobel2();
 printf("Gerando a terceira matriz:\n");
 sobel3();
 printf("Gerando a quarta matriz:\n");
 sobel4();
 printf("Gerando a matriz final:\n");
 sobel_final();
}
```

B.6. 2. mat_inicial.c

```
#include <stdio.h>
#include <stdlib.h>

int sobel_inicial()
{
 FILE *p;
 FILE *a;
 FILE *b;
 FILE *d;
 FILE *e;
 char i[4];
 short int n;
 short int mat[64][64];
 short int l,c;

 p = fopen("niveis_cinza.txt","r");
 a = fopen("imagem1.txt","w");
 b = fopen("imagem2.txt","w");
 d = fopen("imagem3.txt","w");
 e = fopen("imagem4.txt","w");

 for(l = 0;l < 64;l++)
 {
  for(c = 0;c < 64; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
for(l = 0;l < 34;l++)
{
 for(c = 0;c < 34; c++)
 {
  fprintf(a, "%i ", mat[l][c]);

  if (c == 33)
  {
   fprintf(a,"%s","\n");
  }
 }
}
for(l = 0;l < 34;l++)
{
 for(c = 32;c < 64; c++)
 {
  fprintf(b, "%i ", mat[l][c]);

  if (c == 63)
  {
   fprintf(b,"%s","\n");
  }
 }
}
```

```
for(l = 32;l < 64;l++)
{
 for(c = 0;c < 34; c++)
  {
   fprintf(d, "%i ", mat[l][c]);

   if (c == 33)
   {
    fprintf(d,"%s","\n");
   }
  }
}
for(l = 32;l < 64;l++)
{
 for(c = 32;c < 64; c++)
  {
   fprintf(e, "%i ", mat[l][c]);

   if (c == 63)
   {
    fprintf(e,"%s","\n");
   }
  }
}
fclose(p);
fclose(a);
fclose(b);
fclose(d);
fclose(e);
}
```

B.6. 3. mat1.c

```
#include <stdio.h>
#include <stdlib.h>
#define x 32

int sobel1()
{
 FILE *p;
 FILE *b;
 short int MX[3][3];
 short int MY[3][3];
 char i[32];
 short int GX[x][x];
 short int GY[x][x];
 short int H[x][x];
 short int G[x][x];
 short int n;
 short int mat[34][34];
 short int l,c;
 //horizontal
 MX[0][0] = -1;
 MX[0][1] = -2;
 MX[0][2] = -1;
 MX[1][0] = 0;
 MX[1][1] = 0;
 MX[1][2] = 0;
 MX[2][0] = 1;
 MX[2][1] = 2;
 MX[2][2] = 1;
 //vertical
 MY[0][0] = -1;
 MY[0][1] = 0;
 MY[0][2] = 1;
 MY[1][0] = -2;
 MY[1][1] = 0;
 MY[1][2] = 2;
 MY[2][0] = -1;
 MY[2][1] = 0;
 MY[2][2] = 1;

 p = fopen("imagem1.txt","r");
 b = fopen("arq_H1.txt","w");

 for(l = 0;l < 34;l++)
 {
  for(c = 0;c < 34; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
```

```
 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {

   GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
   ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
   ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
   ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
   ((MX[2][2]) * (mat[l+2][c+2]));

   GY[l][c]=((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
   ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
   ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
   ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
   ((MY[2][2]) * (mat[l+2][c+2]));

   if (GX[l][c]<0)
   {
    GX[l][c]=GX[l][c]*(-1);
   }
   if (GY[l][c]<0)
   {
    GY[l][c]=GY[l][c]*(-1);
   }
   G[l][c]=(GX[l][c])+(GY[l][c]);

   if (G[l][c]<0)
   {
    G[l][c]=G[l][c]*(-1);
   }
   if(G[l][c]>254)
   {
    H[l][c]=(1);}
    else
    {
     H[l][c]=(0);
    }
    fprintf(b, "%i ", H[l][c]);

    if (c == 31)
    {
     fprintf(b,"%s","\n");
    }
   }
  }
 fclose(p);
 fclose(b);
}
```

B.6. 4. mat2.c

```
#include <stdio.h>
#include <stdlib.h>
#define x 32

int sobel2()
{
 FILE *p;
 FILE *b;
 short int MX[3][3];
 short int MY[3][3];
 char i[4];
 short int GX[x][x];
 short int GY[x][x];
 short int H[x][x];
 short int G[x][x];
 short int n;
 short int mat[34][x];
 short int l,c;
//horizontal
 MX[0][0] = -1;
 MX[0][1] = -2;
 MX[0][2] = -1;
 MX[1][0] = 0;
 MX[1][1] = 0;
 MX[1][2] = 0;
 MX[2][0] = 1;
 MX[2][1] = 2;
 MX[2][2] = 1;
 //vertical
 MY[0][0] = -1;
 MY[0][1] = 0;
 MY[0][2] = 1;
 MY[1][0] = -2;
 MY[1][1] = 0;
 MY[1][2] = 2;
 MY[2][0] = -1;
 MY[2][1] = 0;
 MY[2][2] = 1;

 p = fopen("imagem2.txt","r");
 b = fopen("arq_H2.txt","w");

 for(l = 0;l < 34;l++)
 {
  for(c = 0;c < 32; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
```

```
 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {
   GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
   ((MX[0][2])  *  (mat[l][c+2]))  +  ((MX[1][0])  *  (mat[l+1][c]))  +
   ((MX[1][1])  *  (mat[l+1][c+1]))  +  ((MX[1][2])  *  (mat[l+1][c+2]))  +
   ((MX[2][0])  *  (mat[l+2][c]))  +  ((MX[2][1])  *  (mat[l+2][c+1]))  +
   ((MX[2][2]) * (mat[l+2][c+2]));

   GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
   ((MY[0][2])  *  (mat[l][c+3]))  +  ((MY[1][0])  *  (mat[l+1][c]))  +
   ((MY[1][1])  *  (mat[l+1][c+1]))  +  ((MY[1][2])  *  (mat[l+1][c+2]))  +
   ((MY[2][0])  *  (mat[l+2][c]))  +  ((MY[2][1])  *  (mat[l+2][c+1]))  +
   ((MY[2][2]) * (mat[l+2][c+2]));

   if (GX[l][c]<0)
   {
    GX[l][c]=GX[l][c]*(-1);
   }
   if (GY[l][c]<0)
   {
    GY[l][c]=GY[l][c]*(-1);
   }
   G[l][c]=(GX[l][c])+(GY[l][c]);

   if (G[l][c]<0)
   {
    G[l][c]=G[l][c]*(-1);
   }
   if(G[l][c]>254)
   {
    H[l][c]=(1);
   }
    else
    {
     H[l][c]=(0);
    }
    fprintf(b, "%i ", H[l][c]);

    if (c == 31)
    {
     fprintf(b,"%s","\n");
    }
   }
  }

fclose(p);
fclose(b);
}
```

B.6. 5. mat3.c

```
#include <stdio.h>
#include <stdlib.h>
#define x 32

int sobel3(){
FILE *p;
FILE *b;
short int MX[3][3];
short int MY[3][3];
char i[4];
short int GX[x][x];
short int GY[x][x];
short int H[x][x];
short int G[x][x];
short int n;
short int mat[x][34];
short int l,c;
//horizontal
 MX[0][0] = -1;
 MX[0][1] = -2;
 MX[0][2] = -1;
 MX[1][0] = 0;
 MX[1][1] = 0;
 MX[1][2] = 0;
 MX[2][0] = 1;
 MX[2][1] = 2;
 MX[2][2] = 1;
 //vertical
 MY[0][0] = -1;
 MY[0][1] = 0;
 MY[0][2] = 1;
 MY[1][0] = -2;
 MY[1][1] = 0;
 MY[1][2] = 2;
 MY[2][0] = -1;
 MY[2][1] = 0;
 MY[2][2] = 1;

 p = fopen("imagem3.txt","r");
 b = fopen("arq_H3.txt","w");

 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 34; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
```

```
 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {

    GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
    ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
    ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
    ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
    ((MX[2][2]) * (mat[l+2][c+2]));

    GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
    ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
    ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
    ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
    ((MY[2][2]) * (mat[l+2][c+2]));

    if (GX[l][c]<0)
    {
     GX[l][c]=GX[l][c]*(-1);
    }
    if (GY[l][c]<0)
    {
     GY[l][c]=GY[l][c]*(-1);
    }
    G[l][c]=(GX[l][c])+(GY[l][c]);

    if (G[l][c]<0)
    {
     G[l][c]=G[l][c]*(-1);
    }
    if(G[l][c]>254)
    {
     H[l][c]=(1);
    }
     else
     {
      H[l][c]=(0);
     }
     fprintf(b, "%i ", H[l][c]);

     if (c == 31)
     {
      fprintf(b,"%s","\n");
     }
    }
  }

fclose(p);
fclose(b);
}
```

B.6. 6. mat4.c

```
#include <stdio.h>
#include <stdlib.h>
#define x 32

int sobel4(){
FILE *p;
FILE *b;
short int MX[3][3];
short int MY[3][3];
char i[4];
short int GX[x][x];
short int GY[x][x];
short int H[x][x];
short int G[x][x];
short int n;
short int mat[x][x];
short int l,c;
//horizontal
 MX[0][0] = -1;
 MX[0][1] = -2;
 MX[0][2] = -1;
 MX[1][0] = 0;
 MX[1][1] = 0;
 MX[1][2] = 0;
 MX[2][0] = 1;
 MX[2][1] = 2;
 MX[2][2] = 1;
 //vertical
 MY[0][0] = -1;
 MY[0][1] = 0;
 MY[0][2] = 1;
 MY[1][0] = -2;
 MY[1][1] = 0;
 MY[1][2] = 2;
 MY[2][0] = -1;
 MY[2][1] = 0;
 MY[2][2] = 1;

 p = fopen("imagem4.txt","r");
 b = fopen("arq_H4.txt","w");

 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }
```

```
 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {
   GX[l][c] = ((MX[0][0]) * (mat[l][c])) + ((MX[0][1]) * (mat[l][c+1])) +
   ((MX[0][2]) * (mat[l][c+2])) + ((MX[1][0]) * (mat[l+1][c])) +
   ((MX[1][1]) * (mat[l+1][c+1])) + ((MX[1][2]) * (mat[l+1][c+2])) +
   ((MX[2][0]) * (mat[l+2][c])) + ((MX[2][1]) * (mat[l+2][c+1])) +
   ((MX[2][2]) * (mat[l+2][c+2]));

   GY[l][c] = ((MY[0][0]) * (mat[l][c])) + ((MY[0][1]) * (mat[l][c+1])) +
   ((MY[0][2]) * (mat[l][c+3])) + ((MY[1][0]) * (mat[l+1][c])) +
   ((MY[1][1]) * (mat[l+1][c+1])) + ((MY[1][2]) * (mat[l+1][c+2])) +
   ((MY[2][0]) * (mat[l+2][c])) + ((MY[2][1]) * (mat[l+2][c+1])) +
   ((MY[2][2]) * (mat[l+2][c+2]));

   if (GX[l][c]<0)
   {
    GX[l][c]=GX[l][c]*(-1);
   }
   if (GY[l][c]<0)
   {
    GY[l][c]=GY[l][c]*(-1);
   }
   G[l][c]=(GX[l][c])+(GY[l][c]);

   if (G[l][c]<0)
   {
    G[l][c]=G[l][c]*(-1);
   }
   if(G[l][c]>254)
   {
    H[l][c]=(1);
   }
    else
    {
     H[l][c]=(0);
    }
    fprintf(b, "%i ", H[l][c]);

    if (c == 31)
    {
     fprintf(b,"%s","\n");
    }
   }
  }

fclose(p);
fclose(b);
}
```

B.6. 7. mat_final.c

```
#include <stdio.h>
#include <stdlib.h>


int sobel_final()
{
 FILE *p;
 FILE *a;
 FILE *b;
 FILE *d;
 FILE *e;

 char i[4];
 short int H[66][66];
 short int n;
 short int mat[66][66];
 short int l,c;

 p = fopen("arq_H1.txt","r");
 a = fopen("arq_H2.txt","r");
 d = fopen("arq_H3.txt","r");
 e = fopen("arq_H4.txt","r");
 b = fopen("arq_H_final.txt","w");

 for(l = 0;l < 32;l++)
 {
  for(c = 0;c < 32; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }

 for(l = 0;l < 32;l++)
 {
  for(c = 32;c < 64; c++)
  {
   fscanf(a,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }

 for(l = 32;l < 64;l++)
 {
  for(c = 0;c < 32; c++)
  {
   fscanf(d,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   printf("%i\n",mat[l][c]);
  }
 }


for(l = 32;l < 64;l++)
```

```
{
 for(c = 32;c < 64; c++)
 {
  fscanf(e,"%s",&i);
  n=atol(i);
  mat[l][c]=n;
  printf("%i\n",mat[l][c]);
 }
}

for(l = 0;l < 64;l++)
{
 for(c = 0;c < 64; c++)
 {
  fprintf(b, "%i ", mat[l][c]);

  if (c == 63)
  {
   fprintf(b,"%s","\n");
  }
 }
}

fclose(p);
fclose(a);
fclose(b);
fclose(d);
fclose(e);
}
```

B. 7. Algoritmo para alteração de arquivos para a linguagem VHDL – alteração_arquivo.c

```
#include <stdio.h>
#include <stdlib.h>


main()
{
 FILE *p;
 FILE *a;
 char i[148];
 int n;
 int mat[148][148];
 int l,c;
 if((p = fopen("niveis_cinza.txt","r")) == NULL)
 {
  printf("Erro ao abrir arquivo!!!\n");
  exit(1);
 }
 if((a = fopen("arq148.txt","w")) == NULL)
 {
  printf("Erro ao abrir arquivo!!!\n");
  exit(1);
 }
 for(l = 0;l < 148;l++)
 {
  for(c = 0;c < 148; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
   fprintf(a, "%d", mat[l][c]);

   if (c == 147)
   {
    fprintf(a,"%s","),\n(");
   }
   else
   {
    fprintf(a,"%s",", ");
   }
  }
 }
}
```

B. 8. Algoritmos para alteração da simulação obtida pelo QUARTUS

B.8. 1. teste_nulo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <direct.h>

int nulo()
{
FILE *p;
FILE *a;
FILE *b;
char i[145];
int d = 9483;
int n;
int mat[44411][1];
int nulo[44411][1];
int l,c;

if((p = fopen("nulo.txt","r")) == NULL)
{
 printf("Erro ao abrir arquivo!!!\n");
 exit(1);
}
if((b = fopen("nulo1.txt","w")) == NULL)
{
 printf("Erro ao abrir arquivo!!!\n");
 exit(1);
}
if((a = fopen("arq_nulo_sobel.txt","w")) == NULL)
{
 printf("Erro ao abrir arquivo!!!\n");
 exit(1);
}
for(l = 0;l < 44411;l++)
{
 for(c = 0;c < 1; c++)
 {
  fscanf(p,"%s",&i);
  n=atol(i);
  mat[l][c]=n;
  fprintf(b, "%i\n", mat[l][c]);

  if (mat[l][c] == 2)
  {
   nulo[l][c] = mat[l][c];
  }
  else
  fprintf(a, "%i\n", mat[l][c]);
  }
 }
fclose(p);
fclose(a);
fclose(b);
system("pause");
}
```

B.8. 2. img_quartus.c

```
#include <stdio.h>
#include <stdlib.h>

int img()
{
FILE *p;
FILE *a;
char i[148];
int d = 146;
int n;
int mat[148][148];
int l,c;

if((p = fopen("arq_nulo_sobel.txt","r")) == NULL)
{
 printf("Erro ao abrir arquivo!!!\n");
 exit(1);
}
 if((a = fopen("arq_sobel.txt","w")) == NULL)
 {
  printf("Erro ao abrir arquivo!!!\n");
  exit(1);
 }

 for(l = 0;l < d;l++)
 {
  for(c = 0;c < d; c++)
  {
   fscanf(p,"%s",&i);
   n=atol(i);
   mat[l][c]=n;
  }
 }
 for(l = 0;l < d;l++)
 {
  for(c = 0;c < d; c++)
  {
   fprintf(a, "%d ", mat[l][c]);
   if (c == 145)
   {
    fprintf(a,"%s","\n");
   }
  }
 }

fclose(p);
fclose(a);
system("pause");
}
```

B.8. 3. quartus_c.c

```
#include<stdio.h>
#include"teste_nulo.c"
#include"img_quartus.c"


int main ()
{
 printf("Alterando o arquivo da simulação:\n");
 nulo();
 system("pause");
 printf("Gerando a matriz imagem:\n");
 img();
 system("pause");
}
```

APÊNDICE C. Algoritmos criados no ambiente computacional QUARTUS

C. 1- Algoritmo do operador *Roberts* - roberts.vhd 155
C. 2 - Algoritmo do operador de *Prewitt* – prewitt.vhd 158
C. 3 - Algoritmo do operador de *Sobel* – sobel.vhd 160

C. 1. Algoritmo do operador *Roberts* - roberts.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;


entity roberts is
  port( saida : out integer;
        clk : in std_logic);
end roberts;

ARCHITECTURE teste OF roberts is
type h is array (1 to 21316) of integer;
type masc IS ARRAY (1 TO 3, 1 to 3) of integer;
TYPE matriz IS ARRAY (1 to 148, 1 to 148) OF INTEGER;
shared variable hx : h;
shared variable hy : h;
shared variable m : h;
shared variable d : integer:=0;
shared variable tranz : integer;
shared variable z : h;
shared variable j : h;
shared variable conv : h;
shared variable n : h;
shared variable limiar : h;
shared variable x : masc := ((1,0,0),(0,-1,0),(0,0,0));
shared variable y : masc := ((0,1,0),(-1,0,0),(0,0,0));
shared variable mat : matriz;
shared variable y1 : masc;
shared variable y2 : masc;
shared variable img : matriz := ((matriz_imagem));

shared variable m1 : integer;
shared variable m2 : integer;
shared variable m3 : integer;
begin
process
--variable z : integer := 0;
begin
wait until clk'event and clk='0';

m1:=1;
m2:=2;
m3:=3;
tranz:=0;

for i in 1 to 3 loop
 for j in 1 to 3 loop
  y1(i,j):= x(i,j);
  y2(i,j):=y(i,j);
  end loop;
end loop;


for l in 1 to 21316 loop
      hx(l):=0;
      hy(l):=0;
      tranz:=tranz+1;
```

```
      if tranz > 146 then
            m1:=m1+1;
            m2:=m2+1;
            m3:=m3+1;
            tranz:=1;
      end if;


      hx(l):= ((img(m1,tranz) * y1(1,1)) + (img(m1,tranz+1) * y1(1,2)) +
      (img(m1,tranz+2) * y1(1,3)) + (img(m2,tranz) * y1(2,1)) +
      (img(m2,tranz+1) * y1(2,2)) + (img(m2,tranz+2) * y1(2,3)) +
      (img(m3,tranz) * y1(3,1)) + (img(m3,tranz+1) * y1(3,2)) +
      (img(m3,tranz+2) * y1(3,3)));


      hy(l):= ((img(m1,tranz) * y2(1,1)) + (img(m1,tranz+1) * y2(1,2)) +
      (img(m1,tranz+2) * y2(1,3)) + (img(m2,tranz) * y2(2,1)) +
      (img(m2,tranz+1) * y2(2,2)) + (img(m2,tranz+2) * y2(2,3)) +
      (img(m3,tranz) * y2(3,1)) + (img(m3,tranz+1) * y2(2,2)) +
      (img(m3,tranz+2)*y2(3,3)));


if hx(l) < 0 then
 z(l):=(hx(l)*(-1));

 else
 z(l):=hx(l);

end if;

 if hy(l) < 0 then
 j(l):=(hy(l)*(-1));

 else
 j(l):=hy(l);

 end if;
 conv(l):=((z(l))+(j(l)));

if conv(l)<0 then
n(l):=conv(l)*(-1);

else
n(l):=conv(l);
end if;

if n(l)>254 then
 limiar(l):=1;
 else
 limiar(l):=0;
end if;

end loop;

if clk'event and clk='0' then

d:=d+1;
saida<=limiar(d);

else
```

```
saida<=limiar(d);
d:=0;
end if;
end process;

end teste;
```

matriz_imagem→ Matriz da imagem a ser processada de dimensão 148x148;

C. 2 - Algoritmo do operador de *Prewitt* – prewitt.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;


entity prewitt is
 port( saida : out integer;
 clk : in std_logic);
end prewitt;

ARCHITECTURE teste OF prewitt is
type h is array (1 to 21316) of integer;
type masc IS ARRAY (1 TO 3, 1 to 3) of integer;
TYPE matriz IS ARRAY (1 to 148, 1 to 148) OF INTEGER;
shared variable hx : h;
shared variable hy : h;
shared variable m : h;
shared variable d : integer:=0;
shared variable tranz : integer;
shared variable z : h;
shared variable j : h;
shared variable conv : h;
shared variable n : h;
shared variable limiar : h;
shared variable x : masc := ((-1,-1,-1),(0,0,0),(1,1,1));
shared variable y : masc := ((1,0,-1),(1,0,-1),(1,0,-1));
shared variable mat : matriz;
shared variable y1 : masc;
shared variable y2 : masc;
shared variable img : matriz := ((matriz_imagem));

shared variable m1 : integer;
shared variable m2 : integer;
shared variable m3 : integer;
begin
process
--variable z : integer := 0;
begin
wait until clk'event and clk='0';

m1:=1;
m2:=2;
m3:=3;
tranz:=0;

for i in 1 to 3 loop
 for j in 1 to 3 loop
  y1(i,j):= x(i,j);
  y2(i,j):=y(i,j);
 end loop;
end loop;

for l in 1 to 21316 loop
 hx(l):=0;
 hy(l):=0;
 tranz:=tranz+1;
```

```
 if tranz > 146 then
  m1:=m1+1;
  m2:=m2+1;
  m3:=m3+1;
  tranz:=1;
end if;

 hx(l):= ((img(m1,tranz) * y1(1,1)) + (img(m1,tranz+1) * y1(1,2)) +
 (img(m1,tranz+2) * y1(1,3)) + (img(m2,tranz) * y1(2,1)) +
 (img(m2,tranz+1) * y1(2,2)) + (img(m2,tranz+2) * y1(2,3)) +
 (img(m3,tranz) * y1(3,1)) + (img(m3,tranz+1) * y1(3,2)) +
 (img(m3,tranz+2) * y1(3,3)));

 hy(l):= ((img(m1,tranz) * y2(1,1)) + (img(m1,tranz+1) * y2(1,2)) +
 (img(m1,tranz+2) * y2(1,3)) + (img(m2,tranz) * y2(2,1)) +
 (img(m2,tranz+1) * y2(2,2)) + (img(m2,tranz+2) * y2(2,3)) +
 (img(m3,tranz) * y2(3,1)) + (img(m3,tranz+1) * y2(2,2)) +
 (img(m3,tranz+2) * y2(3,3)));


if hx(l) < 0 then
 z(l):=(hx(l)*(-1));
else
 z(l):=hx(l);

end if;

if hy(l) < 0 then
 j(l):=(hy(l)*(-1));
 else
 j(l):=hy(l);

end if;
conv(l):=((z(l))+(j(l)));

if conv(l)<0 then
n(l):=conv(l)*(-1);
else
n(l):=conv(l);
end if;

if n(l)>254 then
limiar(l):=1;
else
limiar(l):=0;
end if;

end loop;

if clk'event and clk='0' then
d:=d+1;
saida<=limiar(d);
else
saida<=limiar(d);
d:=0;
end if;
end process;

end teste;
```

matriz_imagem → Matriz da imagem a ser processada de dimensão 148x148;

C. 3. Algoritmo do operador de *Sobel* – sobel.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;


entity sobel is
port( saida : out integer;
      clk : in std_logic);
end sobel;

ARCHITECTURE teste OF sobel is
type h is array (1 to 21316) of integer;
type masc IS ARRAY (1 TO 3, 1 to 3) of integer;
TYPE matriz IS ARRAY (1 to 148, 1 to 148) OF INTEGER;
shared variable hx : h;
shared variable hy : h;
shared variable m : h;
shared variable d : integer:=0;
shared variable tranz : integer;
shared variable z : h;
shared variable j : h;
shared variable conv : h;
shared variable n : h;
shared variable limiar : h;
shared variable x : masc := ((-1,-2,-1),(0,0,0),(1,2,1));
shared variable y : masc := ((-1,0,1),(-2,0,2),(-1,0,1));
shared variable mat : matriz;
shared variable y1 : masc;
shared variable y2 : masc;
shared variable img : matriz := ((matriz_imagem));
shared variable m1 : integer;
shared variable m2 : integer;
shared variable m3 : integer;
begin
process
--variable z : integer := 0;
begin
wait until clk'event and clk='0';

m1:=1;
m2:=2;
m3:=3;
tranz:=0;

for i in 1 to 3 loop
 for j in 1 to 3 loop
  y1(i,j):= x(i,j);
  y2(i,j):=y(i,j);
  end loop;
end loop;


for l in 1 to 21316 loop
      hx(l):=0;
      hy(l):=0;
      tranz:=tranz+1;
```

```
      if tranz > 146 then
       m1:=m1+1;
       m2:=m2+1;
       m3:=m3+1;
       tranz:=1;
      end if;

   hx(l):= ((img(m1,tranz) * y1(1,1)) + (img(m1,tranz+1) * y1(1,2)) +
   (img(m1,tranz+2) * y1(1,3)) + (img(m2,tranz) * y1(2,1)) +
   (img(m2,tranz+1) * y1(2,2)) + (img(m2,tranz+2) * y1(2,3)) +
   (img(m3,tranz)*y1(3,1)) + (img(m3,tranz+1) * y1(3,2)) +
   (img(m3,tranz+2) * y1(3,3)));

   hy(l):= ((img(m1,tranz) * y2(1,1)) + (img(m1,tranz+1) * y2(1,2)) +
   (img(m1,tranz+2) * y2(1,3)) + (img(m2,tranz) * y2(2,1)) +
   (img(m2,tranz+1) * y2(2,2)) + (img(m2,tranz+2) * y2(2,3)) +
   (img(m3,tranz) * y2(3,1)) + (img(m3,tranz+1) * y2(2,2)) +
   (img(m3,tranz+2) * y2(3,3)));

if hx(l) < 0 then
 z(l):=(hx(l)*(-1));
 else
 z(l):=hx(l);
end if;


if hy(l) < 0 then
 j(l):=(hy(l)*(-1));
 else
 j(l):=hy(l);
end if;
conv(l):=((z(l))+(j(l)));

if conv(l)<0 then
 n(l):=conv(l)*(-1);
 else
 n(l):=conv(l);
end if;

 if n(l)>254 then
 limiar(l):=1;
 else
 limiar(l):=0;
 end if;

 end loop;

if clk'event and clk='0' then
d:=d+1;
saida<=limiar(d);
else
saida<=limiar(d);
d:=0;
end if;
end process;

end teste;
```

matriz_imagem → Matriz da imagem a ser processada de dimensão 148x148;

APÊNDICE D. Passos do projeto aplicado na placa FPGA DE2 – 2C35

1 – Abre o QUARTUS 9.1 sp2;

2 – Projeto DE2_NET do kit DE2 alterado;

3 – Abrir a ferramenta programmer e carregar o projeto para a placa;

4 – Abrindo o comando para o sistema operacional *uClinux* – NiosII 8.1 Command Shell;

5 – Digitar o comando - nios2-download –g zImage - para carregar a imagem;

6 – Digitar o comando - nios2-terminal - para abrir o terminal do *uClinux*;

7 – Na máquina com o sistema operacional Linux – compilar os algoritmos dos operadores de bordas – nios_roberts.c, nios_prewitt.c, nios_sobel.c – com o comando – nios2-linux-uclibc-gcc nios_sobel.c –o card_02 –elf2flt – criado o arquivo card_02;

8 – Transferir o arquivo card_02 e o arquivo níveis_cinza.txt para o cartão de memória SD_card;

9 – Inserir o cartão de memória SD_card na placa DE2;

10 – Executar o arquivo card_02 na pasta mnt – pasta do cartão de memória SD_Card - para executar o algoritmo nios_sobel.c com o comando - ./card_02;

11 – Transferi o arq_H_final.txt gerado do SD_Card para o computador;

12 – Visualizar o arquivo arq_H_final.txt no ambiente MATLAB.