Universidade Federal de Pernambuco

Centro de Informática

Uma Arquitetura para Aplicações em Processamento de Imagens: um Estudo em Hardware/Software

Pablo Viana da Silva

Dissertação de Mestrado apresentada para obtenção do grau de
Mestre em Ciência da Computação

Abril de 2002

Orientador: Dr. Manoel Eusebio de Lima

Co-orientador: Dr. Alejandro C. Frery

Agradecimentos

Realmente é uma grande satisfação poder declarar a minha gratidão a todas as pessoas que contribuíram para a realização de um dos mais audaciosos sonhos da minha vida. Sem dúvida, o apoio que recebi desde o início permanecem em minha memória.

Obrigado Senhor, pela sua presença, trazendo saúde, paz e integridade na minha vida e na vida de meus familiares, fortalecendo-me a cada dia em busca da vitória. Graças a Deus por todas as pessoas que tiveram seus corações motivados a colaborar de alguma forma, orientando e estimulando meu crescimento a cada conquista.

Agradeço a compreensão de meus familiares, ao abandoná-los por uns tempos, cheio de sonhos na cabeça, comprometido a um feliz retorno, repleto de realizações. Obrigado a cada colega de trabalho que se dispôs a suprir a minha ausência na UFAL, bem como a liderança daquela universidade por acreditar e apostar no meu sucesso.

Uma eterna gratidão a professora Edna, quem primeiro percebeu a viabilidade de minha admissão aqui no CIn, me acompanhando e orientando cada passo em direção ao êxito e por suas aulas em arquitetura e codesign que desmistificaram minha curiosidade nesta área do conhecimento.

Ao professor Manoel, que com uma simplicidade raramente encontrada nos grandes homens da ciência, excedeu o papel de um orientador, tornando-se um amigo pessoal, um companheiro infalível no desenvolvimento do sistema, resolvendo ainda pormenores do programa de mestrado que eu nem sabia que existiam.

E muito obrigado ao professor Alejandro, que completa a turma que embarcou neste projeto, assumindo papel fundamental na orientação dos primeiros passos do processamento digital de imagens, no uso abençoado do LaTeXe na formatação final do documento. Sua amizade também é valiosa.

Quero agradecer a todos os outros professores, que tive oportunidade de conhecer e trabalhar e a todos os colegas de laboratório que minimizaram seus documentos para prestar socorro às minhas dificuldades de instalação, compilação

e impressão dos projetos aqui no Centro, e pela companhia diária, inclusive nos finais de semana na UFPE nas refeições e nos cafezinhos.

Por outro lado, devo a felicidade estampada em meu rosto todos os dias pelo apoio carinhoso, espiritual e algumas vezes chocante da minha querida Karina. A louríssima amada que esteve ao meu lado em cada momento de desespero quando eu visualizava montanhas de dificuldades.

Desta forma, poderia citar cada nome que teve influência na implementação deste projeto, mas temo ser demasiadamente extenso. Tenho uma vida inteira para lembrar e retribuir com os frutos deste empreendimento, que foi estimulado pela Serttel Engenharia e suportado pelo apoio fomentador da Capes.

A todos vocês, muito obrigado.

Pablo Viana da Silva, MSc.

Resumo

Este trabalho apresenta uma arquitetura Hardware/Software para aplicações em processamento de imagens. O sistema tem como intuito a implementação de um sistema de visão computacional direcionado ao controle de tráfego urbano, o qual visa detectar a presença de veículos em uma área de interesse, dentro do campo visual capturado por uma câmera de vídeo digital instalada em uma via pública.

A metodologia de trabalho contempla o desenvolvimento inicial do algoritmo de processamento de imagens digitais através de ferramentas de alto nível de abstração (IDL - Interactive Data Language), explorando as alternativas de implementação com experimentos e técnicas de realce e análise das imagens.

Na seqüência do fluxo do projeto adotado, a etapa seguinte constitui-se na tradução das funções que compõem o algoritmo desenvolvido em linguagens de médio nível (C/C++), desenvolvendo um código executável que implementa o algoritmo e agregando controle ao usuário do sistema acerca dos ajustes funcionais e resultados obtidos no processamento. Dentro da metodologia de projeto hardware/software, trechos do algoritmo que representam grande demanda do tempo de processamento, tais como filtragens por convolução foram migradas para uma implementação em hardware do processo, mapeando em um dispositivo de lógica programável a síntese lógica da descrição de hardware (VHDL - Very high speed integrated circuit Hardware Description Language), no intuito de satisfazer os requisitos temporais do sistema.

Abstract

This document presents a Hardware/Software architecture for image processing applications. The system aims a computer vision system implementation directed to urban traffic control, which intends to detect the vehicles presence in an interest area, inside of visual field captured by a digital video camera installed at a public way. The work methodology aims the initial development of the image processing algorithm, through high level abstraction tools (IDL – Interactive Data Language), exploring the implementation alternatives with experiences and techniques of image enhance and analysis. Following the adopted design flow, the next stage constitutes in the translate of the functions that composes the developed algorithm to medium level languages (C/C++), building an executable code and attaching user's interface to processing controls and results. In agreement with the hardware/software design methodology, huge time processing modules of the algorithm, like as convolution filters has been migrated to a hardware implementation, mapping the VHDL (Very high speed integrated circuit Hardware Description Language) description logic synthesis in a programmable logic device, attending the system timing constraints.

Sumário

Lista de Figuras

Lista de Tabelas

Glossario de Termos

Bitmap Tipo de arquivo gráfico usado para salvar uma imagem digital. O arquivo contém a informação numérica da cor de cada pixel na imagem.

Bitstream Arquivo de configuração de dispositivos de lógica reconfigurável gerado por ferramentas automáticas de síntese lógica.

Convolução É uma operação matemática em que, dadas duas matrizes numéricas, normalmente de tamanhos diferentes mas de mesma dimensão (1D, 2D, etc.), obtém-se uma terceira matriz de saída onde cada elemento é uma combinação linear (soma de produtos) dos dados das matrizes de entrada.

Crominância Informação de cores da imagem

Device driver Software que possibilita o computador comunicar-se com um dispositivo periférico

FFT (Fast Fourrier Transform) Representação dos dados de uma imagem em termos de seus componentes de freqüência.

FIR (Finite Impulse Response) Resposta de uma transformação a um impulso de entrada com número finito de dados, implementado por uma convolução.

Flag Um bit de estado indicando a condição de uma unidade de processamento

Fps Frames por segundo. Taxa de imagens capturadas por unidade de tempo

Frame Cada imagem completa de uma seqüencia de vídeo

Histograma Gráfico que apresenta a quantidade de pixels em cada nível da escala de cores da imagem.

Kernel Pequena matriz de coeficientes numéricos usada em operações de convolução.

Luminância Brilho da imagem.

MAC (Multiplier / Accumulate) Operação composta da soma de produtos, utilizada em operações de convolução.

Offset Deslocamento em relação ao endereço base do espaço de memória.

Off-the-shelf Componentes comerciais.

Overhead Processamento adicional de controle para a execução de uma tarefa.

pDSP Processador programável específico para aplicações de processamento digital de sinais.

Pixel (Picture element) Cada um dos pontos que formam a imagem na tela do computador.

Taps Tamanho da amostra de entrada. Dígitos ou número de coeficientes de ponderação de uma operação matemática como a convolução.

Time-to-market Tempo decorrido para o lançamento de um produto no mercado, a partir de sua idealização.

Capítulo 1

Introdução

Processamento digital de imagens é uma das áreas de rápido crescimento na comunidade científica, justificado, em parte, pelo avanço e disponibilidade tecnológica de plataformas computacionais cada vez mais poderosas.

Processadores mais velozes e maior espaço em memória disponível têm contribuído para a implementação de muitas soluções que requerem grande performance para diversas aplicações em processamento de imagens, tais como processamento de imagens médicas, jogos, controle de tráfego, etc.

Além das arquiteturas tradicionais de computação, a possibilidade de integração de componentes de *hardware* adicional ao sistema principal tem permitido a aceleração na execução de novas abordagens para o tratamento digital de dados, permitindo o processamento em tempo real de aplicações mais complexas.

Aplicações em processamento digital de sinais (*DSP - Digital Signal Processing*), como o processamento de imagens, demandam em geral uma grande quantidade de memória e processadores de alta performance. Sistemas convencionais baseados em arquiteturas não dedicadas ao tratamento de sinais, em geral, não conseguem atender aos requisitos de performance das aplicações.

A utilização de hardware específico adicional favorece a otimização de recursos tecnológicos, adequando o sistema às exigências de performance necessárias ao correto funcionamento dos algoritmos, atendendo a restrições de tempo, muitas vezes de fundamental importância no projeto.

O elemento mais usual para a implementação de um projeto DSP é um *programmable DSP* (`pDSP`). Este elemento é um componente *off-the-shelf*, con-

stituído essencialmente por um processador ajustado para aplicações específicas DSP. Tais componentes são altamente flexíveis, pois permitem sucessivas reprogramações de seu código executável através de uma linguagem popular como C. Isto permite rápidas iterações de refinamento do projeto, reduzindo o *time-to-market*.

No entanto, um algoritmo codificado em um programa para ser executado em um pDSP está limitado a taxa máxima de processamento, baseada no número de MACs (Multiplier-Accumulator) no dispositivo. Por exemplo, um filtro FIR com 8-*taps* (amostra de entrada com 8 elementos) mostrado na Figura 1.1, requer 8 multiplicações seguidas da adição dos 8 produtos para cada amostra de dado. A implementação deste filtro pode demandar 8 ou mais ciclos em um processador DSP.

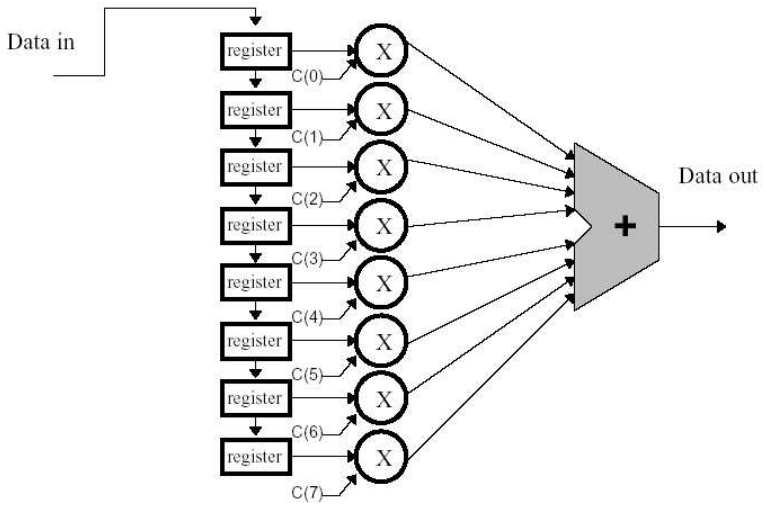

Figura 1.1: Filtro FIR de 8-taps

Intrinsicamente,os *chips* `pDSP` são limitados em performance pelo seu fluxo sequencial de execução, pois quanto maior for o processamento que deva ser realizado com uma certa amostra de dados, mais ciclos de máquina serão necessários e portanto mais lentamente os dados serão processados.

Em contraste ao pDSP, a tecnologia baseada em circuitos integrados digitais para aplicação específica (ASIC – *Application Specific Integrated Circuits*), como os *Gate Array* oferecem a habilidade de execução paralela de tarefas. Um

Gate Array é um *chip* customizado, que permite a implementação específica de circuitos digitais.

Este tipo de abordagem, no entanto, implica em um alto preço de implementação, bem maior que a opção em software, uma vez que a tecnologia empregada exige um processo de fabricação de alto custo inicial. ASICs são resultado de um fluxo de projeto que compreende desde a especificação da funcionalidade do sistema em alto nível passando por etapas de síntese em níveis mais baixos de abstração, em nível de transferência entre registradores (RTL - *Register Transfering Level*), *placement and routing* até chegar ao *lay-out* final do circuito. Após as etapas de síntese e validação virtual o projeto do circuito deve ser encaminhado a fundição de silício e, posteriormente, temos a montagem física do circuito completo em placas de circuito impresso (PCB – *Printed Circuit Board*).

Devido à natureza do processo envolvido, além do alto custo, a flexibilidade de um sistema em hardware é muito menor que uma implementação equivalente em software, inibindo correções de erros e atualizações na sua funcionalidade.

Novas tecnologias disponíveis, tais como dispositivos de lógica programável e hardware reconfigurável, têm sido consideradas na implementação de sistemas exigentes em tempo, custo e flexibilidade, uma vez que tais recursos tecnológicos permitem a configuração de circuitos eletrônicos (hardware) através de sistemas de software, e sua sucessiva reconfiguração adequando o sistema digital a diferentes requisitos funcionais.

FPGAs (*Field Programmable Gate Array*) e CPLDs (*Complex Programmable Logic Device*) são estilos de projetos baseados em arrays de células lógicas reconfiguráveis, que visam oferecer ao projetista uma opção rápida de implementar seus projetos no campo [21]. Estes dispositivos são adequados para prototipação rápida de circuitos integrados digitais, desde que todos os seus parâmetros podem ser reconfigurados, sem a necessidade de retirada de componentes da plataforma de prototipação. Cada FPGA ao ser programado interliga adequadamente todas as conexões internas de acordo com o circuito a ser implementado, comportando-se como um circuito integrado desenvolvido para determinada aplicação.

Neste trabalho, é apresentado um estudo de caso em hardware/software codesign específico para a implementação de um algoritmo de processamento de imagens. O algoritmo proposto funciona como um detector de veículos por imagens.

Os atuais sistemas de monitoramento de trânsito utilizam sensores magnéticos embutidos no calçamento das rodovias, ativando os demais equipamentos de atuação e controle (câmeras, lombadas eletrônicas, etc.). Estes detectores demandam mão-de-obra na instalação e uma manutenção periódica especializada, o que eleva o custo da utilização deste tipo de equipamento.

A solução proposta é substituir o sistema de detecção magnético pelo processamento das imagens que são coletadas por câmeras já instaladas, utilizadas atualmente para o monitoramento do fluxo de veículos, detectando veículos que passam por uma determinada área da via.

A plataforma computacional utilizada é composta por um microcomputador PC, equipado com uma placa de prototipação conectada ao barramento PCI (*Peripheral Component Interface*), onde está instalado o hardware adicional utilizado.

A utilização de dispositivos FPGA (*Field Programmable Gate Arrays*) em uma plataforma de prototipação de hardware possibilita a especificação de uma parte do algoritmo de processamento de imagens, para ser executado em conjunto com a implementação em software do sistema digital proposto.

O particionamento hardware/software foi desenvolvido manualmente com base nas características funcionais tais como comunicação e velocidade de cada modulo do sistema. Este particionamento direcionou para hardware o módulo de processamento das imagens, enquanto que a parte relativa a interface com o usuário e tratamento de arquivos é implementada em software.

O estudo de caso abordado neste projeto possui restrições de tempo real, vinculado ao processamento de frames de vídeo em questão, requerendo taxas de amostragens na ordem de 15 a 30 fps (*frames per second*). O tempo para processamento completo de cada imagem está restrito ao período entre cada quadro capturado (33 a 67 ms).

Considerando a plataforma computacional que existe atualmente para o controle de trânsito, constituída basicamente por computadores PC, o sistema digital proposto compreende o protótipo de um equipamento com baixo custo adicional, referente à inserção de uma placa de expansão, destinado a detecção da presença de veículos sobre faixas de pedestres.

A identificação de veículos se dá através da captura, processamento e análise

de imagens em tempo real de uma câmera instalada em uma via pública e conectada ao computador.

Através de algoritmos de processamento de imagens, as cenas capturadas serão analisadas quadro a quadro, e o sistema indicará a presença ou não de veículos sobre uma determinada área de interesse na cena. Esta aplicação poderá ser utilizada para a detecção de veículos sobre a faixa de pedestres, onde o algoritmo irá detectar a presença de objetos invasores. A funcionalidade obtida servirá como detector de infratores que permanecem com o veículo sobre a faixa durante um período de condições pré-estabelecidas, ou ainda como um contador do fluxo de veículos numa determinada via pública.

O objetivo funcional estabelecido pelo presente trabalho insere-se em um projeto mais abrangente, que estende as características até aqui propostas, agregando recursos adicionais a um sistema de controle de tráfego. Algumas dessas características são a medição da velocidade de veículos, a classificação por tamanho, modelo e a identificação automática do licenciamento (placa) do veículo junto ao órgão regulamentador de trânsito.

1.1 Apresentação

Esta dissertação apresenta no Capítulo 2, o estado-da-arte em trabalhos relacionados a Hardware / Software Codesign para aplicações de processamento de imagens, a utilização de FPGAs para a prototipação de sistemas reconfiguráveis e referências de pesquisas em sistemas de codesign para DSP.

No Capítulo 3, são apresentados fundamentos do processamento digital de imagens que serão aplicados no desenvolvimento do trabalho. Conceitos básicos da área, apresentados com formalismo matemático, fornecem o embasamento teórico das aplicações práticas citadas no final do capítulo.

Um estudo de caso, é desenvolvido no Capítulo 4, apresentando o processo de desenvolvimento do algoritmo detector de veículos, principal objeto de estudo deste trabalho.

A implementação do algoritmo desenvolvido na arquitetura computacional proposta por um PC equipado com uma placa de prototipação PCI é descrito no Capítulo 5, apresentando as principais fases do desenvolvimento do projeto

em hardware/software.

O estudo da plataforma de prototipação utilizada, apresentando a arquitetura da placa, características específicas de disponibilidade lógica de dispositivos e limitações de projetos são analisadas no Capítulo 6, servindo como referência técnica para o desenvolvimento do sistema.

O Capítulo 7 é dedicado a uma análise crítica dos resultados obtidos na implementação, justificando algumas limitações tecnológicas encontradas na utilização da plataforma.

Finalmente, as conclusões obtidas pelo desenvolvimento deste trabalho são apresentados no Capítulo 8. As dificuldades encontradas são analisadas sugerindo futuros trabalhos, em aplicações e arquiteturas de baixo custo com alta performance.

Capítulo 2

Estado da Arte

Sistemas de visão computacional para análise de cenas naturais externas têm tido muitas aplicações úteis em problemas relacionados ao controle de trânsito rodoviário [29], [14]. Os congestionamentos de vias públicas e problemas relacionados com detectores existentes têm gerado interesse em novas tecnologias de detecção de veículos como o processamento de imagens de vídeo [11], [30].

Análise automática de cenas de tráfego é essencial para muitas áreas de IVHS (*Inteligent Vehicle Highway Systems*) [20]. A informação destas imagens pode ser usada para otimizar fluxo de tráfego, identificar veículos enguiçados ou acidentes, e auxiliar a tomada de decisão de um controlador autônomo [40].

Melhorias na tecnologia de sistemas de visão computacional para supervisão têm habilitado o desenvolvimento de sistemas detalhados e confiáveis. A Divisão de Ciência da Computação da Universidade da Califórnia, no desenvolvimento do projeto California PATH (*Partners for Advanced Transit and Highways*) [4] aplicou um rastreador baseado em correlação e um modelo de movimento usando um filtro *Kalman* [53] para extrair trajetórias de veículos numa seqüência de imagens de trânsito [36]. Redes neurais tem sido aplicadas no processo de aprendizado de estimativas recursiva do estado de objetos aplicando extensões desse filtro [32], [18].

No Brasil, equipamentos para monitoramento de rodovias têm sido utilizados para auxiliar o controle na cobrança de pedágios. No complexo rodóviário "Linha Azul", em Santa Catarina, são utilizados equipamentos de tecnologia importada baseada em micro-ondas para a detecção de veículos [2].

A melhoria da tecnologia utilizada está relacionada com a complexidade dos algoritmos de processamento de imagens e seu desempenho submetido a restrições de custo e tempo real. O desenvolvimento de produtos e aplicações nesta área tem utilizado paradigmas de hardware/software codesign na implementação de sistemas de computação reconfigurável, para viabilizar soluções que atendam às especificações propostas. Esta abordagem de codesign tem proporcionado inúmeras vantagens, principalmente no que se refere à adoção de recursos de lógica reconfigurável, tais como:

i) A possibilidade de implementação de algoritmos diretamente em hardware, minimizando o *overhead* de controle devido a decodificação de instruções a cada passo de execução, pode oferecer como resultado uma densidade computacional mais alta que em processadores convencionais.

ii) Implementação de algoritmos em hardware significa o uso de paralelismo. Como o espaço de computação é amplo e reconfigurável, altos graus de paralelismo e eficiência de implementações são facilmente obtidos.

Nem todas estas vantagens estarão necessariamente presentes em qualquer aplicação. No estudo de caso apresentado nesta dissertação, por exemplo, a primeira característica teve maior influência que a segunda na otimização do desempenho do sistema.

Acredita-se que em algum ponto no futuro, projetistas rotineiramente estarão utilizando programas como o principal e talvez a única descrição de sistemas completos. Dentro desta visão, muitos sistemas *hardware/software* serão criados por programadores ao invés de equipes mistas compostas de programadores e engenheiros eletrônicos [42]

Sistemas de codesign para projetos heterogêneos de hardware/software como Ocapi [24], ou PISH [13] oferecem um ambiente para a modelagem do sistema digital em linguagens de alto nível de abstração como C++, Occam, ou extensões destas linguagens como a iniciativa SystemC [47], [5]. Ainda dentro desta linha de pesquisa, podem ser citados os ambientes Chinook [17] que têm como principais tópicos o problema do interfaceamento entre componentes de hardware e software, escalonamento sob restrições de tempo e particionamento da funcionalidade e Polis [38] que traduz a descrição inicial em um modelo interno durante as fases de refinamento do sistema.

Também, considerando as características específicas das aplicações em DSP (*Digital Signal Processing*) como a ênfase do processamento no fluxo de dados, sistemas específicos para modelagem em alto nível de tais aplicações tem sido disponibilizadas. Ferramentas como a Simulink(R) [3], uma ferramenta interativa para modelagem, simulação e análise de sistemas dinâmicos, que permite a construção gráfica de diagramas de blocos pré-definidos ou customizados, simulação e avaliação de desempenho no projeto de sistemas. A integração deste tipo de ambiente com fabricantes de dispositivos reconfiguráveis como a Xilinx Inc. tem permitido o refinamento de sistemas e seu mapeamento a partir de blocos funcionais, em *cores* pré-sintetizados, para as famílias de FPGAs [27]. A Figura 2.1 mostra um exemplo de fluxo de projeto da ferramenta Simulink com o ambiente Xilinx.

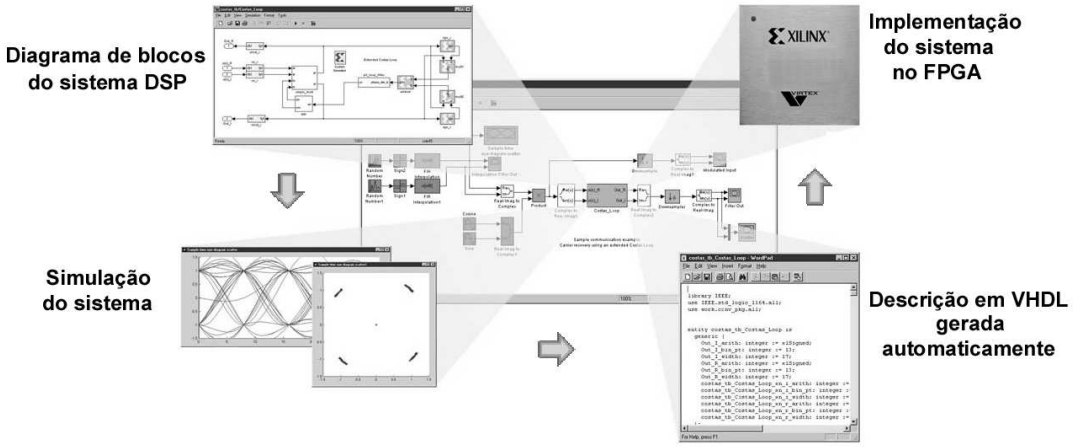

Figura 2.1: Simulação funcional de blocos e geração de código VHDL para Xilinx.

O ambiente permite que o projetista desenvolva e simule sistemas em alto nível de abstração, gerando automaticamente o código sintetizável da especificação em HDL (*Hardware Description Language*), otimizado para as famílias Xilinx Virtex(R)-II, Virtex e Spartan(R)-II.

Aplicações e produtos vêm sendo desenvolvidos por grupos de pesquisa e empresas, sempre no intuito de obter um melhor desempenho de aplicações em processamento de imagens, assim como, prover melhores técnicas para o mapeamento de algoritmos em plataformas reconfiguráveis. Dentro deste contexto, o departamento de tecnologia e medicina do *Imperial College of Science*

(Londres, Inglaterra) em parceria com a *Sony Broadcast & Professional Europe* (Basingstoke, Inglaterra) apresenta a plataforma SONIC [45], uma arquitetura de computação reconfigurável projetada para acelerar o processamento de imagens de vídeo. Esta plataforma, cuja arquitetura é apresentada na Figura 2.2, integra múltiplas unidades de processamento (PIPE), composta por 2 FPGAs Altera, sendo um para implementação da funcionalidade e outro para roteamento, e 4 Mbytes de memória RAM, funcionando como um *plug-in* para aplicações comerciais de processamento de vídeo como o Adobe Premiere(R), um software profissional para edição não-linear [15].

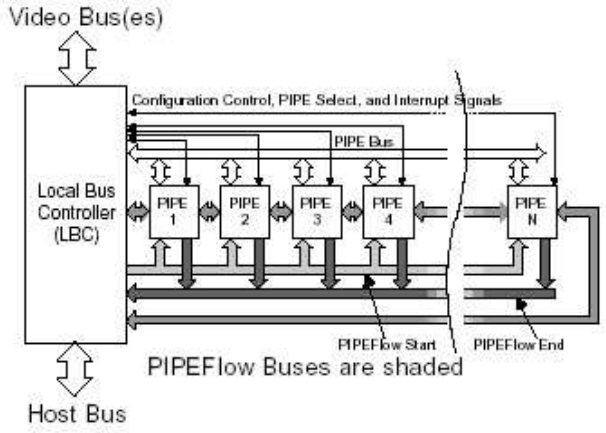

Figura 2.2: A arquitetura SONIC

A arquitetura SONIC permite a implementação paralela de diversos *plugins* (um em cada PIPE) (Figura 2.2), habilitando a flexibilidade e otimização de *pipelines*, utilizado por múltiplas aplicações simultaneamente.

Uma outra plataforma customizada para aplicações específicas de processamento de imagens que tem sido utilizada para implementação de filtros morfológicos e de média espacial em tempo real é a SPLASH-2. Sua arquitetura é adequada para computações repetitivas e altas taxas de transferência de dados que caracterizam a maioria dos problemas de processamento de imagens [48]. Desenvolvida pelo Departamento de Engenharia Elétrica Bradley da *Virginia Polytechnic Institute and State University* (Virginia, USA), esta plataforma é formada por placas processadoras (1 a 15 placas), contendo 17 FPGAs Xilinx XC4010 cada uma. Operando com um clock de 10MHz, a implementação de

um filtro de média espacial (ver Secção 3) pode ser configurado para realizar o processamento numa taxa de 30 imagens por segundo.

A composição de tecnologias DSP e FPGA numa mesma plataforma é proposto por [33] e [35], a qual utiliza uma estratégia de alocação dos processos de controle nos dispositivos de lógica reconfigurável e o processamento intenso de dados, baseado em operações MAC e de ponto flutuante, nos processadores DSP. Estas plataformas, como indicam seus resultados experimentais, têm obtido alta precisão e performance de tempo real [39].

Apesar da performance garantida tanto pela SONIC quanto pela SPLASH-2, entre outras arquiteturas reconfíguráveis [44], a alta complexidade do *harware* disponível nestas plataformas podem implicar num alto custo de aplicação. Além disso, a eficiência de elementos individuais não é o único fator essencial para atingir os requistos de funcionamento em tempo real, mas o gerenciamento do fluxo de dados e a reação rápida a eventos e conflito entre elementos são também de fundamental importância [39]. A integração dos principais recursos necessários a aplicações específicas reunidas em um único *chip* (SOC - *System-on-Chip*) pode ser uma boa alternativa para a redução da complexidade de interconexão entre elementos de processamento.

Outras arquiteturas mais modestas de computação reconfigurável tem sido desenvolvidas para a implementação de sistemas em hardware/software com baixo custo. Um exemplo é a plataforma da `HOT-I` [49], da VCC (Virtual Computer Corporation). Esta plataforma dispõe de um FPGA Xilinx `XC6200` para desenvolvimento de projetos e um Xilinx `XC4010` com um *Core* [10] que implementa a interface PCI da placa com o PC, além de dois bancos de 1MBytes de memória RAM disponíveis. Neste sistema, o FPGA que implementa a interface PCI é configurado com o LogiCORE PCI da Xilinx durante a inicialização do computador (*host*).

O XC6200 é um FPGA de granularidade fina, desenvolvido para sistemas reconfiguráveis *on-the-fly*. Sua produção foi interrompida pela Xilinx e as ferramentas de desenvolvimento para esta família não sofreu evolução. Devido ao fato de que esta família de FPGAs é popular apenas na área de pesquisa, a mesma não é bem suportada por ferramentas de projeto comerciais [25].

Por estas razões, a plataforma `HOT-I` foi comercialmente substituída pela

HOT-II, agora equipada com um FPGA Xilinx Spartan-II XCS40, compatível com as modernas ferramentas de síntese lógica em uso (Xilinx Foundation Series, Synopsys FPGA Express, etc.) [6].

Plataformas como a Cesys XC2S_Eval utilizam um ASIC como interface, reduzindo o custo relacionado a IPs (*Intelectual Property*) dos Cores PCI. Esta placa disponibiliza basicamente um FPGA Xilinx Spartan-II com capacidade para 200k *gates* para o desenvolvimento de projetos de hardware, além de um banco de memória RAM de 512kB. O baixo custo de aquisição desta placa, além da elevada disponibilidade lógica reconfigurável favoreceram sua utilização neste trabalho.

2.1 Conclusão

Uma visão geral dos trabalhos relacionados ao escopo desta dissertação apresentou aplicações de processamento de imagens para o controle de trânsito urbano, incluindo suas plataformas de desenvolvimento e execução.

Neste capítulo foi discutido como a disponibilidade dos recursos tecnológicos da computação reconfigurável tem possibilitado avanços em sistemas de visão computacional, localizando o nosso trabalho no contexto atual desta categoria de aplicações.

A variedade de plataformas de prototipação mencionandas foram avaliadas de acordo com seu desempenho na execução de algoritmos fundamentais do processamento digital de imagens, os quais serão apresentados no próximo capítulo.

Capítulo 3

Elementos de Processamento de Imagens

3.1 Imagem Digital

Este capítulo tem por objetivo fornecer definições dos principais objetos de interesse para este trabalho, as imagens digitais. Para tanto será utilizado o contexto formal da área, fornecido, dentre outras referências por [12]. As operações sobre imagens são bem definidas em [28].

3.1.1 Definição

Imagens digitais são os principais objetos de estudo relacionados com o processamento de imagens. A definição inicial restringir-se-á ao estudo das imagens em **tons de cinza**. Para os propósitos desta dissertação será suficiente tratar com este tipo de dados, reservando-se o uso de cor para outras aplicações.

Consideraremos dois diferentes tipos de conjuntos de elementos, o primeiro deles pode ser denotado por E, um conjunto de quadrados adjacentes arrumados ao longo de um certo número de linhas e colunas, e formando uma superfície retangular. Os quadrados são identificados por suas posições (linha e coluna). Este conjunto é denominado o *suporte da imagem*, e fica caracterizado pelo produto cartesiano de dois intervalos de número inteiros, isto é, $\mathsf{E} = \{0, \ldots, m-1\} \times \{0, \ldots, n-1\}$. Intuitivamente E pode ser visualizado como o conjunto das

posições (livres de valor) de uma matriz de tamanho $m \times n$.

A figura 3.1 apresenta um conjunto de 110 quadrados arrumados ao longo de 10 linhas e 11 colunas, onde um pequeno x foi marcado em cada quadrado para indicar que ele não é colorido com branco.

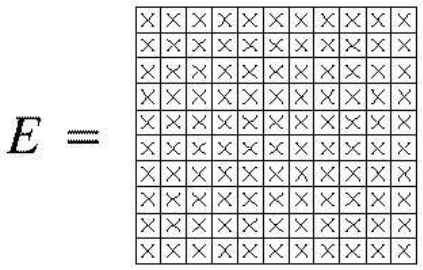

Figura 3.1: Ilustração do suporte de uma imagem E.

O segundo conjunto, o qual é denotado K, é um conjunto de tons de cinza, também chamado de escala de cinza. A figura 3.2 apresenta uma escala de cinza com quatro níveis.

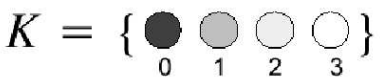

Figura 3.2: Ilustração do conjunto K, o contradomínio de uma imagem.

Cada nível de cinza do conjunto K, pode ser identificado numericamente pela sua posição (0 a $k-1$, numa escala de k tons de cinza). Se $k = 2$ a imagem é chamada "*binária*".

Cada quadrado cinza, denominado *Pixel* (Picture element), localizado em E e com um nível de cinza em K, é um elemento do produto Cartesiano de E e K. A Figura 3.3 apresenta o produto cartesiano $E \times K$, onde E é o conjunto mostrado na Figura 3.1 e K o conjunto da Figura 3.2. Outros autores utilizam a palavra "pixel" para se referir quanto às coordenadas $x \in E$.

Pela definição do produto Cartesiano, um pixel localizado em E e com um nível de cinza em K é um par ordenado (x, s) cujos elementos são um quadrado x (não colorido) de E e um nível de cinza s de K.

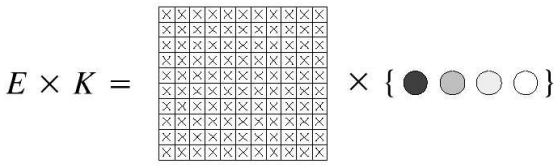

Figura 3.3: Produto Cartesiano de um conjunto de quadrados (suporte E) e uma escala de cinza (K).

Uma imagem digital em tons de cinza é um *grafo de mapeamento* de E para K. Uma imagem I com domínio E e escala de cinza K é um subconjunto de pixels localizados em E e com nível de cinza em K. Assim, cada quadrado de E é colorido com um único tom de cinza de K (Figura 3.4).

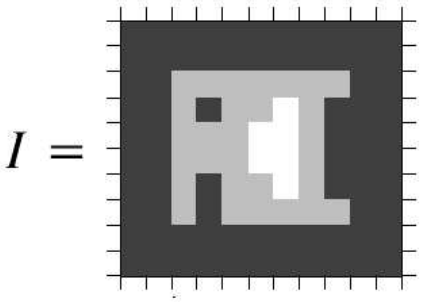

Figura 3.4: Exemplo de uma imagem com valores em K definida sobre E.

O número de linhas de E é $\mathtt{m}$ e o número de colunas é $\mathtt{n}$. Desta forma, a imagem e seu domínio são ditos serem de tamanho $\mathtt{m}$ por $\mathtt{n}$ ($\mathtt{m} \times \mathtt{n}$). A imagem da Figura 3.4 é de tamanho 10 por 11.

Uma imagem pode, então, ser definida por uma matriz $\mathtt{m} \times \mathtt{n}$ onde cada um dos elementos representa um pixel da imagem. Desta forma, o valor numérico de cada elemento está associado ao tom de cinza que coloriza o pixel, tal como ilustrado na matriz 3.1 que representa a imagem da Figura 3.4.

$$I = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 3 & 3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 3 & 3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.1}$$

3.1.2 Formatos de Imagens Digitais

Para uma imagem em tons de cinza (monocromática), uma matriz $m \times n$ descreve através dos valores numéricos de cada célula, a coloração s (onde, $s \in K = \{0, 1, ...k-1\}$) atribuída a cada pixel.

O valor k define o número de níveis de cinza da escala. O valor mínimo (0) está associado à cor preta e o valor máximo ($k-1$) à cor branca. Desta forma, a faixa de valores intermediários são mapeados na variedade de tonalidades de cinza permitida.

Na escala proposta na Figura 3.2, k é igual a 4, ou seja, o conjunto K possui quatro tonalidades de cinza (do preto ao branco), de modo que o valor numérico s pode ser representado na base binária (formato digital), por 2 bits. Da mesma forma, os pixels de uma imagem mapeada em um escala de 256 tons de cinza ($k = 256$) podem ser representados digitalmente por 8 bits (1 Byte).

O formato de imagem assim descrito é conhecido como *Bitmap* e seu tamanho em Bytes pode ser avaliado pela expressão

$$T = m \times n \times \log_2 k \tag{3.2}$$

onde T é o tamanho da imagem em Bytes.

Apesar de ocupar grande espaço de memória se comparado com outros formatos de imagem que possuem algum esquema de compactação, o Bitmap oferecem o acesso direto aos dados, facilitando sua manipulação e permitindo um

tratamento mais eficiente dos arquivos que armazenam as imagens a serem processadas.

3.1.3 Imagens em Cores

Um *espaço de cores* é a representação matemática de características físicas de percepção do sistema visual humano que pode ser aplicado a um sistema computacional. [16]

Dentre os principais espaços de cores comumente utilizados, podem ser citados o RGB (*Red, Green, Blue*) e o CYM (*Cyan, Yellow, Magenta*), entre outros. A adoção de um sistema em particular depende da área de aplicação. O RGB, por exemplo, é utilizado nas telas dos monitores de vídeo, enquanto o CYM é bastante utilizado em sistemas de impressão em cores.

O conjunto K, pode ser atribuído não só a uma escala de tons de cinza variando do preto ao branco, como também pode ser associado a uma escala de qualquer cor, variando do preto até atingir a máxima intensidade daquela cor.

A associação de três imagens de mesmo tamanho ($\mathfrak{m} \times \mathfrak{n}$), uma com escala em tons de veremelho (R-red), uma outra em tons de verde (G-green) e uma terceira em tonalidades de azul (B-blue) sobrepostas, forma a representação de uma imagem multi-colorida (Figura 3.5) no espaço RGB.

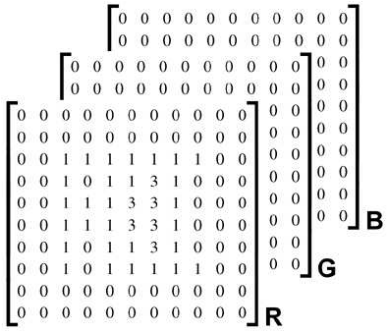

Figura 3.5: Superposição de matrizes formando uma imagem colorida (RGB)

Analogamente, o tamanho digital de uma imagem bitmap apresentada na

Figura 3.5 pode ser obtida pela expressão:

$$T = 3 \times (m \times n \times \log_2 k)$$
$$T = 3 \times (10 \times 11 \times \log_2 4)$$
$$T = 3 \times 220 = 330bits$$

onde m e n definem o suporte E (Figura 3.1), e k o contradomínio da imagem.

A utilização de cores na apresentação de imagens habilita uma maior percepção de informações visuais. Enquanto o olho humano pode perceber somente poucas dezenas de níveis de cinza [28], o mesmo tem habilidade para distinguir entre milhares de cores, as quais possuem atributos, tais como: *brilho, matiz* e *saturação*. O brilho representa a luminância percebida e a matiz de uma cor refere-se ao seu "avermelhamento", "esverdeamento", etc. Para fontes luminosas monocromáticas, diferenças em matiz são manifestadas pela diferença em comprimento de onda. Finalmente, saturação é aquele aspecto de percepção que varia quando mais e mais luz branca é adicionada à luz monocromática.

Experiências prévias demostraram que, para a aplicação corrente, existe informação suficiente na *luminância* (Y) das imagens, de modo que a *crominância* (C_b e C_r) pode ser descartada no intuito de reduzir a informação a ser processada e armazenada.

Para a conversão das imagens originalmente coloridas em RGB (*Red, Green, Blue*) para a informação de luminância (Y), utilizou-se uma transformação, de acordo com o padrão NTSC (*National Television Systems Committee*)

Esta transformação é dada por:

$$Y = 0.30R + 0.59G + 0.11B \tag{3.3}$$

Os coeficientes que multiplicam cada componente de cor são justificados pela percepção visual humana, que possui função de eficiência luminosa relativa (V) com curva no formato de um "sino", como mostra a Figura 3.6. A cor verde ($\lambda = 546.1nm$) localiza-se na faixa central da curva, região dos mais altos valores da função. A cor vermelha ($\lambda = 700nm$) localiza-se à direita do pico da função e a cor azul ($\lambda = 435.8nm$), à esquerda, correspondendo a baixos valores para a função $V(\lambda)$. [28].

Funções de eficiência luminosa são a base da fotometria e foram introduzidas pelo CIE (*Commission Internationale de L'Eclairage*) [1] para prover uma análise psico-física da radiação denominada luminância. A função fornece a proporção de energia de um espectro luminoso para o qual o olho humano é mais sensível [46].

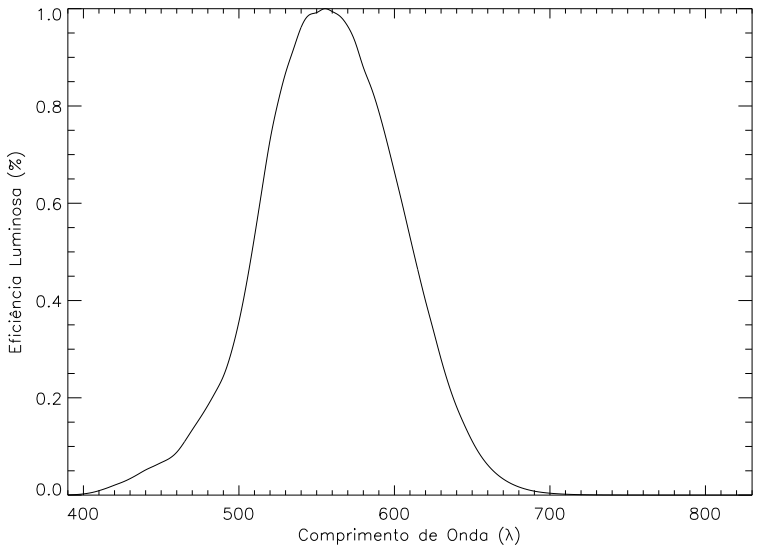

Figura 3.6: Função de eficiência luminosa relativa típica

3.2 Realce em Imagens Digitais

O realce de imagens é o conjunto de técnicas e procedimentos que tem por objetivo a acentuação de características desejadas como bordas, arestas ou contraste para tornar uma imagem mais proveitosa para apresentação ou análise.

Este processo não aumenta a informação inerente contida nos dados, mas acentua certas características já presentes na imagem digital para sua análise. Alguns exemplos de realce são: ajuste de contraste, destaque de arestas, fitragem de ruídos etc. [28]. Pode-se classificar tipos de realce de imagens em duas categorias principais: operações pontuais e operações espaciais.

A seguir serão descritas, neste contexto, as operações que se mostraram úteis para a aplicação pretendida.

3.2.1 Operações Pontuais

As operações pontuais, também chamadas de operações *zero-memory*, são transformações onde um dado nível de cinza $u \in [0, L]$ é diretamente mapeado em um nível de cinza $v \in [0, l]$ de acordo com uma transformação

$$v = f(u) \tag{3.4}$$

que depende exclusivamente de u.

Neste trabalho, a operação pontual denominada *limiarização* será utilizada para tornar binária ($k = 2$) uma imagem em tons de cinza ($k = 256$). Esta transformação filtra substancialmente a quantidade de dados desnecessários da imagem. Esta técnica deve ser aplicada quando diferentes características de uma imagem estão contidas em diferentes níveis de cinza.

$$v = \begin{cases} L, & u \geq \text{limiar} \\ 0, & \text{c.c.} \end{cases} \tag{3.5}$$

3.2.2 Operações Espaciais

Operações espaciais de realce são baseadas nos valores dos pixels vizinhos ao pixel sob transformação. A noção de vizinhança usualmente empregada está relacionada à distância entre pixels. Neste caso, a imagem pode por exemplo ser convoluída com um filtro FIR (*Finite Inpulse Response*) caracterizada por uma *máscara espacial*.

$$\begin{array}{|c|c|c|} \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \end{array} \tag{3.6}$$

máscara de média espacial 3 x 3

A expressão (3.6) apresenta um exemplo de máscara espacial, também denominada *kernel*, a qual é utilizada para a delimitação da vizinhança que envolve o pixel submetido à transformação.

A convolução de uma imagem com a máscara (3.6) substitui o valor de cada pixel da imagem pela média dos valores dos seus oito pixels vizinhos. Esta

operação é conhecida como *Média Espacial Ponderada* que é um caso especial da *Filtragem Espacial Passa-Baixa* (3.7).

$$v(m,n) = \sum \sum_{(k,l)\in W} a(k,l)y(m-k,n-l) \tag{3.7}$$

onde $y(m,n)$ e $v(m,n)$ são as imagens de entrada e a saída, respectivamente, W é um janela adequadamente escolhida (máscara), e $a(k,l) \geq 0$ são coeficientes de ponderação do filtro (elementos de W).

Um Filtro de média espacial possui todos os coeficientes da máscara espacial iguais e com soma total igual a 1, resultando na transformação apresentada na equação 3.8.

$$v(m,n) = \frac{1}{N_w} \sum \sum_{(k,l)\in W} y(m-k,n-l) \tag{3.8}$$

onde $a(k,l) = \frac{1}{N_W}$ e N_w é o número de elementos do kernel de convolução (máscara espacial) W.

Média espacial é utilizado para suavização de ruído, e neste trabalho será necessária a sua aplicação, conforme a abordagem mais específica apresentada no capítulo seguinte.

3.3 Subtração de Imagens (Detecção de mudanças)

De acordo com [28], um problema de grande significância em análise de imagens é a detecção de mudanças ou presença de um objeto em uma determinada cena. Estes problemas ocorrem em áreas como sensoriamento remoto para monitoramento de desenvolvimento urbano, previsão meteorológica, diagnose de doenças em imagens médicas, detecção de alvos em radares, automação utilizando visão computacional, entre outras aplicações do gênero.

Mudanças em uma cena dinâmica observada como $u_i(m,n), i = 1,2,...$são dadas por

$$e_i(m,n) = u_i(m,n) - u_{i-1}(m,n) \tag{3.9}$$

onde o índice i representa a passagem do tempo.

A equação (3.9) atribui a imagem e_i, a diferença ocorrida entre as cenas subsequentes u_{i-1} e u_i. Com isso cada pixel de e_i que possui valor não-nulo indica mudança localizada ocorrida entre as cenas.

$$e_i(m,n) \neq 0 \Rightarrow u_i(m,n) \neq u_{i-1}(m,n) \tag{3.10}$$

Uma outra variação para esta técnica é subtrair de cada cena dinâmica $u_i(m,n), i = 1,2,...$ uma imagem de referencia $r(m,n)$ que serve como base comparativa para detectar a presença de um objeto "invasor" em uma imagem. Deste modo, a imagem $r(m,n)$ deve ser tal que represente um ambiente (cenário), livre de objetos invasores, para que sejam percebidas mudanças no momento da presença de tais objetos, atraves da equação (3.11).

$$e_i(m,n) = u_i(m,n) - r(m,n) \tag{3.11}$$

A subtração de imagens será utilizada no capítulo seguinte, durante a fase de desenvolvimento do estudo de caso desta dissertação.

3.4 Conclusão

Este capítulo apresentou alguns dos principais tópicos em processamento digital de imagens, que serão fundamentalmente utilizados nos capítulos seguintes deste documento.

As noções iniciais como o suporte de uma imagem digital e seu mapeamento em uma escala de cores, além das técnicas básicas de realce através de operações pontuais e espaciais definem o escopo do trabalho apresentado.

Apesar da vasta diversidade desta área do conhecimento científico, não comentadas neste texto, as noções no tratamento de imagens digitais discutidas até o momento constituem um conhecimento suficiente para a compreensão de todo o processo de desenvolvimento da aplicação específica em estudo.

Capítulo 4

Estudo de Caso

4.1 Trabalho Proposto

O estudo de caso desta dissertação foi proposto por uma empresa privada colaboradora, responsável pelo monitoramento do tráfego rodoviário metropolitano do Recife (SERTTEL Engenharia Ltda.), e consiste no desenvolvimento de um Detector de Veículos por Imagens (DVI).

A necessidade deste projeto surge das dificuldades encontradas pela empresa na instalação e manutenção de equipamentos sensores para detecção de veículos em rodovias. Os sistemas de detecção atuais utilizam técnicas de enlace magnético, que consistem na percepção da variação do campo magnético em um espiral elétrico no momento em que uma massa metálica considerável (carros pequenos, médios ou grandes) passa sobre o referido sensor. Uma breve análise deste tipo de tecnologia fornece subsídios suficientes para motivar o desenvolvimento de uma solução alternativa, utilizando uma nova tecnologia, que contemple sensoriamento óptico. A instalação de enlaces magnéticos requer a quebra e remoção de parte do asfalto ou calçamento da rodovia, causando transtornos no tráfego local, demandando tempo e mão-de-obra. Além disso, a manutenção deste tipo de tenologia é prejudicada pelas limitações mecânicas do sistema, ocasionando freqüentes quebras dos condutores que compõem o sensor.

A solução proposta foi de se utilizar uma câmera de vídeo, do tipo que já é comumente aplicada em monitoramento remoto de trânsito, adaptada para servir como "foto-sensor" de veículos que passam por determinado trecho da

pista. Uma das aplicações para este tipo de sistema é a detecção de veículos que estacionam sobre a faixa de pedestres durante o semáforo vermelho, implicando em infração grave de acordo com o código nacional de trânsito em vigor.

Neste caso, a câmera que captura o campo visual que inclui a faixa de pedestres deve monitorar aquelas mudanças ocorridas na cena que impliquem na invasão de algum veículo sobre a área de interesse que, para este exemplo, estamos considerando como sendo a faixa de pedestres.

De uma forma geral, o escopo deste trabalho visa a detecção da presença de um objeto invasor (veículo) em uma área de interesse delimitada, não necessariamente uma faixa de pedestres, dentro do campo visual da imagem capturada pela câmera. Esta detecção será realizada por um sistema digital de software e hardware composto por um computador PC equipado com uma placa de prototipação baseada em FPGAs.

4.2 Aquisição de Dados

Através do sistema de monitoramento de trânsito cedido pela SERTTEL (empresa colaboradora do projeto), foram capturados trechos de vídeo digital do tráfego de veículos em uma das vias de trânsito de sua jurisdição. As imagens coloridas foram capturadas como arquivos, gravados no formato `.AVI` (vídeo para Windows), numa taxa de captura de 15 quadros por segundo (*fps - frames per second*), entrelaçado. Cada quadro (frame) está definido por uma imagem RGB de tamanho 640×480 pixels que corresponde a 921600Bytes ou 900kBytes ($1\text{kByte} = 1024\text{Bytes}$).

A partir destes arquivos, foram extraídos as sequências de quadros, utilizando-se para esta operação um software gratuito denominado: "XtraConverter (AVI - BMP Extractor)" [7]. Cada frame (quadro) extraído apresenta-se em formato BMP (Bitmap para Windows) de 24-bit (3×8-bit).

Apesar de terem sido ajustadas as propriedades do software de captura, para uma taxa de 15 frames por segundo, o computador que foi utilizado para a tarefa de coleta de vídeo possuia baixa velocidade de processamento. Isto fez com que o sistema fosse incapaz de fornecer aquela taxa de amostragem oferecendo, no máximo, uma taxa em torno de 2 fps. Consequentemente, o software de

captura replicou uma média de 7 frames entre cada cena inédita, como ilustra a Figura 4.1.

Figura 4.1: Seqüência de frames capturados com redundância indesejável.

Durante a extração dos frames de vídeo, percebeu-se visualmente esta redundância de informação e por esta razão foi realizado o "corte" de frames desnecessários.

O corte destes frames redundantes deu-se através de uma pequena rotina descrita em linguagem IDL (*Interactive Data Language*) [22], que compara cada quadro e armazena apenas os distintos, numa estrutura de array ($640 \times 480 \times$ Número de Frames)(ver código em IDL no Anexo 1).

Esta mesma rotina em IDL, além de descartar frames redundantes, elimina também a informação de cores (RGB), convertendo as imagens em monocromáticas através do cômputo da informação de luminância (Y), pixel a pixel de acordo com a Transformação (3.3).

A linguagem IDL mostrou ser uma ferramenta adequada para esta fase do projeto. A facilidade na exploração de diversas alternativas de implementação, através da disponibilidade de funções pré-definidas em alto nível de abstração permitiu rápidas iterações e experimentos com diversos parâmetros durante o desenvolvimento dos algoritmos de processamento de imagens, dispensando a necessidade de implementação em baixo nível do código associado a cada tentativa de abordagem do problema.

Os quadros obtidos após o descarte de redundâncias e cor foram armazenados em um arquivo no formato de dados do IDL (`.dat`), para utilização posterior durante os testes e experimentos descritos a seguir.

4.3 Desenvolvimento do Algoritmo

Considerando o objetivo de detectar a presença de veículos em uma área de interesse, selecionada dentro do campo visual da cenas capturadas, o algoritmo

utilizou a técnica de subtração de imagens, descrita pela equação (3.11) do Capítulo 3. A subtração é aplicada entre as várias imagens (frames em seqüência), utilizando-se como imagem de referência uma cena apresentando uma área de interesse, livre de veículos.

Nas imagens capturadas, conforme descrição da seção anterior, selecionou-se uma área retangular, de dimensões 250 x 150 pixels, na parte inferior direita da imagem, determinando o que seria uma possível região de interesse em monitoramento (Figura 4.2). A área em destaque (área de interesse) constitui uma imagem de referência válida por apresentar um cenário estático (imóvel) e livre de veículos (objetos invasores). Esta imagem, denominar-se-á "imagem de referência" $\mathtt{r}(\mathtt{m}, \mathtt{n})$, e será utilizada pelo algoritmo de detecção.

Figura 4.2: Área de interesse (em destaque) na imagem.

Percebe-se que apenas a área delimitada pela janela em destaque desperta interesse na análise. Portanto, o algoritmo deverá realizar o processamento de cada nova imagem somente nesta porção da cena, uma vez que o processamento da imagem completa representaria "desperdício" de recursos computacionais. A imagem completa é composta por 307.200 pixels, enquanto que a área selecionada possui apenas 37.500 pixels. Desta forma, a restrição do processamento da imagem na área selecionada, para este exemplo, corresponde a uma economia de 87,8% do processamento.

A Figura 4.3 apresenta uma seqüência de imagens mostrando os seis primeiros quadros que foram utilizados para o desenvolvimento do algoritmo. Esta sequência mostra um veículo gradativamente invadindo a área de interesse e posteriormente retirando-se. Este trecho servirá como uma amostra do vídeo capturado e orientará o processo de desenvolvimento do algoritmo neste capítulo.

A área de interesse de cada frame da sequência (Figura 4.3) será comparado (subtraído), com a imagem de referência (Figura 4.2), de acordo com a Expressão (3.11).

Em cada comparação *frame - referência* obtém-se uma imagem em tons de cinza que deverá ser convertida em uma imagem binária ($k = 2$) através de uma limiarização do histograma dada pela equação (4.1).

$$e_i(m,n) = \begin{cases} 1, & [u_i(m,n) - r(m,n)] \geq \text{limiar} \\ 0, & \text{c.c.} \end{cases} \tag{4.1}$$

para $m = \{300 : 550\}$, $n = \{0 : 150\}$, $i = \{1, 2, ...\#Frames\}$.

O valor escolhido para o limiar é empírico, decorrente de sucessivos experimentos executados na ferramenta de desenvolvimento IDL. Este parâmetro deve ser ajustado de acordo com as condições ambientais da cena, tais como a iluminação natural, o ofuscamento causado pelo faróis dos veículos, chuva, etc. Após o refinamento de tentativas com valores na faixa de 0 a L.(onde, $L = 255$) com os dados de imagens disponíveis, o valor 80 mostrou-se adequado para a correta funcionalidade do algoritmo e por esta razão foi adotado como o valor do limiar.

Isto significa que, as diferenças entre os valores pixel a pixel são consideradas iguais a zero se o resultado da subtração for menor ou igual a 80. Caso contrário, o resultado é atribuído a 1, formando deste modo como resultado, imagens binárias, as quais têm o conjunto K resumido aos valores 0 e 1.

O resultado da equação (4.1) aplicada aos sucessivos frames (Figura 4.3) é apresentado na seqüência da Figura 4.4.

O resultado apresentado na Figura 4.4 é insatisfatório para o objetivo do projeto, uma vez que a presença de " ruído" nas imagens impossibilita uma clara distinção entre os objetos de interesse (veículos).

A presença deste ruído deve-se ao fato das condições adversas da captura

Figura 4.3: Seqüência de 6 frames do vídeo.

Figura 4.4: Seqüência de frames resultantes da diferença entre originais e referência

das imagens, já que o ambiente está sujeito a variações de luminosidade natural, reflexos, vibrações da câmera, etc.

Para a minimização deste efeito, aplicou-se uma etapa de filtragem linear da imagem, através de um processo de convolução entre a imagem e um kernel (Equação 4.2) formado por uma estrutura matricial 3×3 de soma igual a 1 e origem central (célula 2,2).

$$\mathtt{Kernel} = \begin{bmatrix} 0.111 & 0.111 & 0.111 \\ 0.111 & 0.111 & 0.111 \\ 0.111 & 0.111 & 0.111 \end{bmatrix} \tag{4.2}$$

A utilização de uma filtragem linear sobre a imagem antes de submetê-la à etapa de subtração causou um melhoramento na qualidade do resultado obtido na diferença entre cenas. A Figura 4.5 apresenta a seqüência de frames processadas, utilizando a filtragem linear.

Visualmente, pode-se perceber nesta seqüência da Figura 4.5 a ausência do ruído binário presente na seqüência anterior (Figura 4.4), enfatizando o sucesso na aplicação do filtro.

O filtro linear, implementado por uma convolução com uma máscara de média espacial 3x3, suavizou as imagens antes de subtraí-las, minimizando diferenças pontuais entre os pixels. É importante entender que a sua aplicação após a subtração das imagens simplesmente causaria um "borramento" do ruído presente na seqüência da Figura 4.4.

As imagens binárias obtidas destacam as regiões da cena que sofreram modificação em relação à imagem de referência $r(m, n)$. As regiões com brilho máximo ($k = 1$), de acordo com a equação (4.1), correspondem aos pixels de $u_i(m, n)$ (Seção 3.3) que tiveram seu valores consideravelmente modificados em relação à imagem de referência $r(m, n)$. Enquanto que que as regiões escuras ($k = 0$) correspondem a valores de pixels que não se alteraram ou que sofreram mudanças desprezíveis.

São denominadas *mudanças desprezíveis* aquelas diferenças na imagem causadas pelo efeito espacial da suavização do filtro linear e sombreamentos causados por objetos na imagem fora da área de interesse mas que, devido a posição da iluminação ambiental da cena, acabam influindo na área selecionada, como por

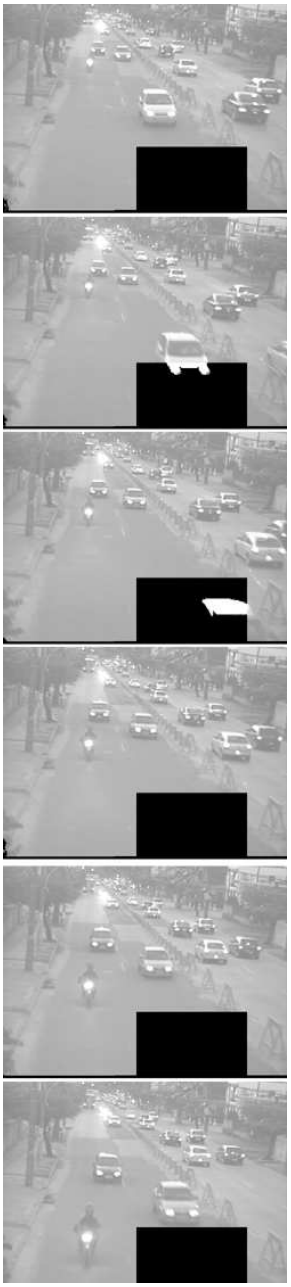

Figura 4.5: Seqüência de frames resultantes da diferença entre imagens filtradas e a referência

exemplo a sombra do automóvel da Figura 4.6.

Figura 4.6: Sombra causada por um veículo.

Será denominado *Grau de Invasão* (G_{inv}) o número total de pixels que sofreram *modificações consideráveis*, ou seja, diferenças superiores ao valor do limiar parametrizado na aplicação (equação 4.1), dentro da área de interesse. Isto resulta no número de pontos brancos dentro na janela de seleção.

Para cada frame $e_i(m, n)$ processado, isto é, filtrado e comparado com a referência, pode-se obter o seu respectivo grau de invasão G_{inv}.pela expressão (4.3).

$$G_{inv}\left[e_i(m,n)\right] = \sum_{(m,n)\in W} e(m,n) \tag{4.3}$$

onde W é a janela que delimita a área de interesse na imagem.

Pequenos valores do grau de invasão, inferiores a um valor mínimo pré-estabelecido pelo parâmetro de área, são atribuídos a zero. Esta operação ajusta a sensibilidade do algoritmo em relação ao tamanho do objeto a ser detectado, descartando falsas ocorrências como pequenos animais e pedestres. Para possibilitar a detecção de veículos pequenos, médios e grandes, inclusive de motocicletas nas imagens utilizadas neste estudo de caso, a área mínima considerada foi ajustada para 1000 pixels.

O grau de invasão associado a cada frame para o exemplo considerado neste capítulo (Figura 4.5) está apresentado na Tabela 4.1.

Frame	Grau de invasão
0	0
1	0
2	2573
3	5727
4	0
5	0
6	0

Tabela 4.1: Grau de invaso detectado em cada frame

O grau de invasão plotado em um plano cartesiano (Frames × Grau de Invasão) mostrando o comportamento do Grau de invasão com o passar do tempo é apresentado na Figura 4.7.

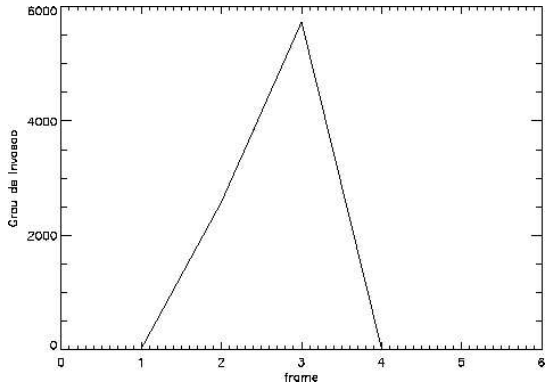

Figura 4.7: Comportamento do grau de invasão nos seis frames analisados

Intuitivamente, o comportamento do gráfico desta função discreta pode indicar a entrada, a permanência e a saída de objetos invasores na área de interesse. Os pontos de *máximos locais* do gráfico do Grau de invasão serão empregados para determinar a ocorrência de um veículo no interior da área monitorada. Neste pequeno exemplo, a detecção do ponto de máximo no Frame 3 indica a passagem de um veículo pela área de interesse.

O algoritmo que determina o ponto de máximo compara os valores atual, anterior e posterior ($\mathsf{G}_{\mathsf{inv}}(\mathsf{i}), \mathsf{G}_{\mathsf{inv}}(\mathsf{i}-\mathsf{1})$ e $\mathsf{G}_{\mathsf{inv}}(\mathsf{i}+\mathsf{1})$) do grau de invasão, localizando assim todos os máximos locais da função, os quais fornecem um indicativo da presença de objetos invasores.

$$G_{inv}(i-1) < G_{inv}(i) > G_{inv}(i+1) \Longrightarrow \text{Veículo detectado} \qquad (4.4)$$

O algoritmo de detecção discutido neste capítulo pode ser resumidamente apresentado no fluxograma da Figura 4.8.

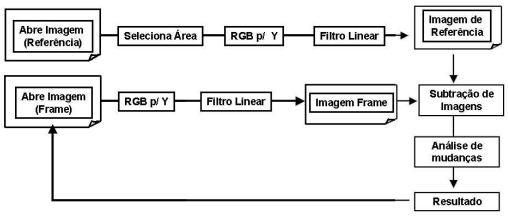

Figura 4.8: Fluxograma do algoritmo de detec ção de veículos por imagens.

Um trecho de 34 frames foi submetido ao algoritmo definido e o comportamento do grau de invasão obtido é apresentado na Figura 4.9. Os pontos de máximo do gráfico são ilustrados por amostras das imagens binárias correspondentes.

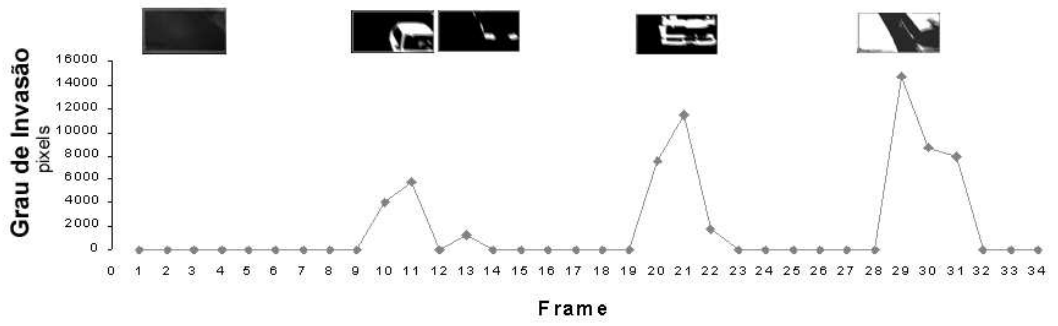

Figura 4.9: Comportamento do grau de invas ão no tempo.

A implementação em IDL deste algoritmo constitui um protótipo para a validação de seu funcionamento e está limitado a realizar simulações para obtenção de parâmetros de sensibilidade tais como o limiar a ser empregado e a área mínima a ser considerada como ocorrência de invasão. Esta implementação serve como especificação do sistema digital a ser desenvolvido usando técnicas de hardware/software co-design, como descreve o próximo capítulo.

4.4 Complexidade do Algoritmo

O estudo da implementação em software ou hardware do algoritmo de detecção de veículos será baseada na complexidade do processamento. Esta complexidade pode ser calculada em relação ao número de operações envolvidas e fundamenta a etapa de particionamento durante o codesign.

Na estimativa da complexidade, desconsiderar-se-á a abertura das imagens bitmap, admitindo que as imagens, objeto do processamento, já estão alocadas na memória do PC.

4.4.0.1 Conversão de cores

A conversão das imagens no padrão de cores RGB para imagens monocromáticas (tons de cinza) envolve operações pontuais baseadas na equação (3.3), que consiste em 3 (três) produtos e 2 (duas) somas para cada pixel da matriz $m \times n$. Neste trabalho, a imagem a ser considerada na implementação do protótipo executável será uma janela retangular de 128×256 pixels, portanto, a complexidade computacional desta etapa é equivalente a:

$$128 \times 256 \times 3 = 98.304 \text{ multiplicações}$$
$$128 \times 256 \times 2 = 65.536 \text{ adições}$$

4.4.0.2 Filtragem por convolução

Esta operação é responsável por grande parte da complexidade do algoritmo, sendo necessário 9 (nove) produtos e 8 (oito) somas para cada pixel da imagem em convolução com um *kernel* de 9 (nove) elementos. Mas, no caso descrito neste trabalho, considerando as características do *kernel* utilizado (expressão 3.6), a convolução foi realizada pela equação (3.8), que equivale a 8 (oito) somas e 1 (um) produto para cada pixel da imagem. Assim, para 128×256 *pixels* teremos:

$$128 \times 256 \times 8 = 262.144 \text{ adições}$$
$$128 \times 256 \times 1 = 32.668 \text{ multiplicações}$$

4.4.0.3 Subtração entre imagens

A comparação entre cada frame e a referência estabelece uma sutração e uma operação do módulo da diferença para cada pixel da imagem. Considerando separamente estas operações:

$$128 \times 256 \times 1 = 32.768 \text{ subtrações}$$
$$128 \times 256 \times 1 = 32.768 \text{ módulos}$$

De acordo com a equação 4.1, cada resultado é comparado com o valor do parâmetro de limiarização da imagem, de modo que contabilizamos:

$$128 \times 256 \times 1 = 32.768 \text{ comparações}$$

4.4.0.4 Quantificação de mudanças

A quantificação de mudanças, tem como resultado o grau de invasão ocorrido no frame e é obtido pela soma do valor associado (1 ou 0) a cada um dos pixels da imagem $m \times n$, limiarizada pela operação anterior.

$$128 \times 256 - 1 = 32767 \text{ adições}$$

Como a quantificação de mudanças representa o grau de invasão de um objeto na área de interesse, e um parâmentro de sensibilidade (Area) descarta pequenas invasões, comparações com a área mínima são computadas para cada frame.

$$1 \text{ comparação}$$

4.4.0.5 Detecção de máximos locais

Esta última operação do algoritmo demanda duas comparações a cada imagem: Uma entre o grau de invasão $G_{inv}(i-2)$ e $G_{inv}(i-1)$, e outra entre $G_{inv}(i-1)$ e $G_{inv}(i)$.

$$2 \text{ comparações}$$

Resumindo, temos para cada imagem processada:

$$
\begin{aligned}
98.304 + 32.668 = \quad & 130.972 && \text{multiplicações} \\
65.536 + 262.144 + 32767 = \quad & 360.447 && \text{adições} \\
& 32.768 && \text{subtrações} \\
& 32.768 && \text{módulos} \\
& 32.770 && \text{comparações}
\end{aligned}
$$

Estes resultados fornecem estimativas de métricas que deverão ser implicitamente consideradas na implementação do algoritmo em software e hardware.

4.5 Conclusão

Neste capítulo foram descritas todas as fases de desenvolvimento do algoritmo de detecção de veículos. O processo compreende desde a captura de dados reais das imagens de trânsito, sua análise e síntese de informações relevantes até a implementação em uma linguagem de alto nível de abstração do algoritmo com a funcionalidade proposta.

Alguns dos conceitos discutidos no Capítulo 3 foram utilizados como fundamentação técnica da manipulação dos dados, no desenvolvimento do algoritmo.

Os testes com seqüências de frames apresentadas nas Figuras 4.5 e 4.9 indicam o correto funcionamento do sistema, de acordo com os parâmetros de limiar e área experimentados.

Finalmente, uma análise da complexidade do algoritmo proposto oferece estimativas métricas das operações algébricas, as quais deverão ser consideradas na sua implementação em software ou em hardware.

Capítulo 5

Implementação em um Modelo de Hardware/Software

Pesquisas em Hardware/Software codesign tratam o problema do projeto de sistemas heterogêneos. O principal objetivo do codesign consiste em desenvolver sistemas digitais que satisfaçam às restrições de projeto através da utilização de componentes comerciais (common-off-the-shelf) e componentes de aplicação específica, reduzindo o tempo de desenvolvimento de produtos (*time-to-market)* pela minimização do esforço e custo de implementação.

A implementação de um sistema digital baseado no paradigma Hardware/ Software Codesign segue um fluxo de desenvolvimento em que sua especificação inicial de alto nível, é particionada, e através de etapas de refinamento e síntese alcança o mapeamento numa implementação em componentes de software e de hardware. São considerados como componentes de software a utilização de elementos de propósito geral tais como processadores ou microcontroladores que podem ser adequadamente programados para a execução de um algoritmo, enquanto que componentes de hardware são aqueles elementos de computação que possuem configuração definida para aplicações específicas, como os ASICs (*Application Specific Integrated Circuit*).

Este tipo de abordagem *Top-down*, onde a especificação inicial de projeto é particionada e posteriormente sintetizada, foi adotada neste trabalho. Como ilustra a Figura 5.1, a etapa de especificação do sistema corresponde a definição da funcionalidade do algoritmo a ser implementado, atendendo às restrições im-

postas pelo problema em estudo.

A etapa de particionamento consiste em seccionar a descrição inicial do sistema em duas partes principais. Uma delas será implementada por componentes de software e a outra por componentes de hardware. O processo de particionamento é guiado por métricas de avaliação do custo e desempenho das alternativas de implementação do sistema, levando em consideração fatores de projeto tais como: comunicação entre módulos do algoritmo, espaço necessário em memória, paralelismo no processamento de dados, etc. Em geral, módulos do sistema digital caracterizados por operações seqüenciais, sem rígidas restrições de tempo de processamento são normalmente direcionados para uma implementação em software, enquanto que aqueles módulos, responsáveis por grande demanda de tempo de execução, envolvendo fluxos de dados que podem ser paralelizados são mapeados em componentes de hardware.

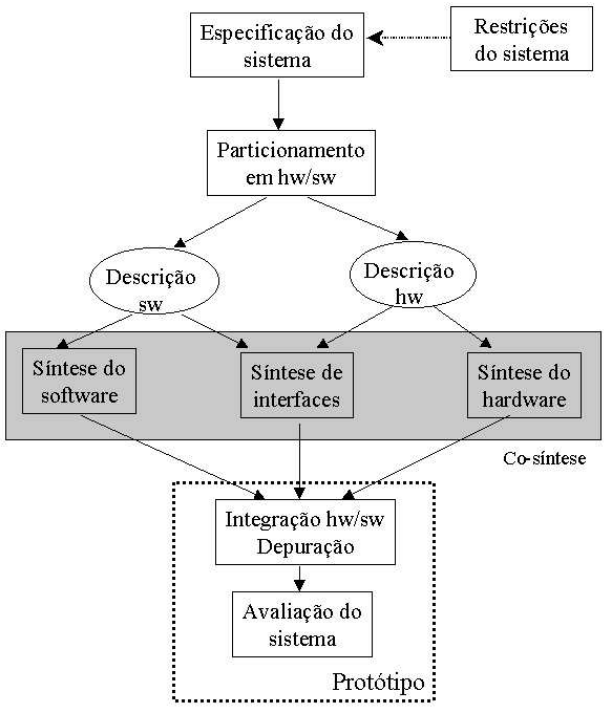

Figura 5.1: Metodologia de hardware/software codesign

A quebra da descrição inicial em duas partições, uma de software outra de hardware, insere no projeto a necessidade da comunicação entre os módulos

do sistema que estão em partições distintas. Por isto, além da síntese das descrições de software e de hardware dos processos contidos na especificação inicial, é necessário a síntese de interface, que implementa a comunicação entre as duas partições.

Após a síntese dos componentes do projeto, o sistema é integrado na plataforma de hardware e software, formando um protótipo, permitindo a avaliação e depuração das funcionalidades do sistema.

Neste capítulo serão apresentados o particionamento do sistema proposto em componentes de hardware e software, realizado manualmente, bem como seus respectivos processos de síntese e implementação em uma arquitetura heterogênea.

Foram adotados como componente de software um computador PC de propósito geral e como componente de hardware uma placa de prototipação baseada em um FPGA. A Figura 5.2 ilustra o arranjo proposto como arquitetura alvo, onde os dados de entrada, provenientes de uma câmera de vídeo são processados por um PC equipado com o hardware adicional implementado em um FPGA.

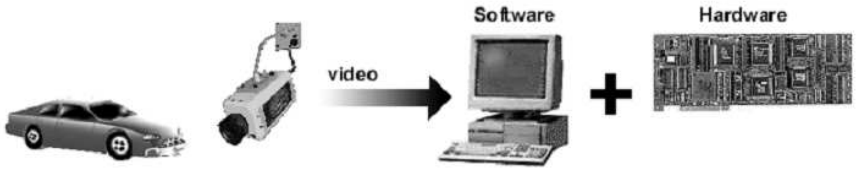

Figura 5.2: Diagrama de blocos do sistema detector de veículos.

No final do Capítulo 4 chegou-se a um algoritmo que implementa a detecção de veículos pela análise de imagens de monitoramento de tráfego rodoviário urbano. O algoritmo, experimentalmente desenvolvido e descrito em linguagem IDL servirá como especificação inicial do sistema digital. Ela será manualmente refinada, em níveis mais baixos de abstração, possibilitando em primeira instância, a implementação por uma linguagem de programação compilada de um protótipo totalmente em software. Em seguida, é apresentada uma abordagem hardware/software do sistema, baseada na metodologia ilustrada pela Figura 5.1.

O objetivo é desenvolver um programa para ser executado em um computador PC, agregando hardware adicional conectado ao barramento PCI para melhorar

a performance do processamento.

O software de acesso à placa de prototipação (*device driver*), fornecido pelo fabricante, foi implementado em C++ e por esta razão adotou-se esta linguagem na programação do componente de software.

5.1 Implementação em Software

Para a construção de uma interface amigável com o usuário do sistema, foi utilizado o ambiente de desenvolvimento Visual C++. A implementação do protótipo em software, como mostra a Figura 5.3, é constituída por uma interface baseada em uma janela principal com menu e barra de ferramentas, permitindo o controle interativo do usuário. Desta forma, a escolha da imagem de referência, posicionamento e seleção da área de interesse, escolha do processamento por software ou hardware, além dos controles deslizantes dos parâmetros de sensibilidade de detecção podem ser facilmente ajustados e isto contribui na depuração e avaliação do sistema completo.

Figura 5.3: Protótipo executável do sistema de detecção de veículos.

Após a construção do *front-end* do sistema, iniciou-se o processo de "tradução" dos comandos utilizados no algoritmo em IDL em linhas de código C++. Pela falta de ferramentas automáticas para esta tarefa, foi realizada uma tradução manual de todo o algoritmo. Percebe-se que algumas das tarefas de refinamento são tão específicas da aplicação que a manipulação direta do desenvolvedor com a escrita do código parece ser indispensável.

5.1.1 Abertura de Imagens Bitmap

A primeira função a ser traduzida corresponde a abertura de arquivos de imagens Bitmap, a qual é feita por um único comando em IDL:

imagem = READ_BMP, nome_do_arquivo.bmp

Em C++ a abertura de um arquivo BMP demanda dezenas de linhas de código e está associada ao método `Serialize` da aplicação C++. Este método deve capturar todas as informações inerentes a imagem, de acordo com o formato estruturado do Windows DIB (*Device-Indepedent Bitmap*). Cada arquivo bitmap contém um *Bitmap File Header*, um *Bitmap Info Header*, um *Color Table* e um *Bitmap Data*.

A estrutura Bitmap File Header contém informações sobre o tipo e o tamanho de um arquivo DIB.

```
struct tagBITMAPFILEHEADER {
 WORD wType;
 DWORD dwSize;
 WORD wReserved1;
 WORD wReserved2;
 DWORD dwOffBits;
};
```

A estrutura Bitmap Info Header contém informações sobre a resolução, compressão, tamanho e número de cores utilizadas refletindo sobre o seu tamanho em bits.

```
struct tagBITMAPINFOHEADER {
```

```
 DWORD dwSize;
 DWORD dwWidth;
 DWORD dwHeight;
 WORD wPlanes;
 WORD wBitCount;
 DWORD dwCompression;
 DWORD dwSizeImage;
 DWORD dwXPelsPerMeter;
 DWORD dwYPelsPerMeter;
 DWORD dwClrUsed;
 DWORD dwClrImportant;
};
```

A tabela de cores (*Color Table*) contém tantos elementos quanto são as cores do bitmap. Esta tabela não está presente para bitmaps com 24-bits por pixel, porque cada pixel é representado por intensidades RGB (*Red*, *Green* e *Blue*) na área Bitmap Data.

```
struct tagRGBQUAD {
 BYTE bBlue;
 BYTE bGreen;
 BYTE bRed;
 BYTE bReserved;
};
```

O Bitmap Data é a matriz de valores que representa as linhas consecutivas (*scan lines*) do bitmap. Cada scan line consiste de bytes consecutivos representando o pixel na linha da esquerda para a direita. O número de bytes representando uma scan line depende do formato de cores e resolução do bitmap. As linhas da imagem estão armazenadas de baixo para cima: a primeira linha no Bitmap Data representa a última linha da imagem e a última linha do array representa a primeira linha da imagem. Ou seja, a imagem é arquivada verticalmente invertida Por isso atenção especial deve ser tomada no momento da implementação do código de abertura destas imagens.

A função de abertura da imagem bitmap deve retornar um array tridimensional representado por 3 matrizes (*Red*, *Green* e *Blue*) de 480 linhas e 640 colunas (`imagem[480][640][3]`) que será manipulado pelas etapas seguintes do algoritmo detector de veículos.

5.1.2 Definição da Área de Interesse

A interface visual do programa permite o usuário escolher a posição desejada da "janela" de dimensões 256×128 pixels na cena, como área de interesse do algoritmo de detecção de veículos. Esta área selecionada da imagem está contornada por um retângulo de cor branca (Figura 5.3) e pode ser movido pelo *mouse* do PC.

As coordenadas $(\mathtt{x}, \mathtt{y})$ da extremidade superior esquerda e inferior direita da janela de seleção são armazenadas em memória em objetos da classe *CPoint* (similar a estrutura *Point* do Windows). Estas instâncias de CPoint parametrizam o suporte das transformações aplicadas a seguir, fazendo com que sejam computados apenas os pixels dentro da área selecionada (ver Seção 4.3).

5.1.3 Conversão RGB para Y

Após a abertura do bitmap, já dispondo das matrizes (R, G e B) que definem o espaço de cores da imagem, o passo seguinte é a conversão da imagem colorida em uma imagem monocromática (tons de cinza).

A imagem resultante da transformação é uma matriz 480 linhas $\times$ 640 colunas de valores inteiros, do tipo `unsigned char`. Ou seja, a soma de produtos da equação (3.3) é arredondado para o menor inteiro ($\mathtt{Y}_{\mathbb{I}}$) maior que o valor real ($\mathtt{Y}_{\mathbb{R}}$) da transformação (5.1).

$$\mathtt{Y}_{\mathbb{I}} = \lceil \mathtt{Y}_{\mathbb{R}} \rceil \tag{5.1}$$

Este arredondamento não causa influência perceptível ao correto funcionamento do algoritmo, uma vez que a precisão matemática das operações de comparação entre imagens está principalmente baseada no valor utilizado para limiarizar a imagem como foi apresentado no Capítulo 4 (equação 4.1).

5.1.4 Filtro Linear

Conforme já justificado em capítulos anteriores, a aplicação do filtro linear nas imagens comparadas (frame - referência) elimina a presença de ruídos pela suavização da imagem. O FIR (*Finite Impulse Response*) de tamanho 3×3 foi obtido pela implementação de dois laços (*loops*) aninhados, operando seqüenciamente a máscara de média espacial com a imagem monocromática.

Como se trata de uma máscara de média espacial 3×3, todos os seus coeficientes são iguais a $\frac{1}{9}$. Por isso, a equação 3.8 foi utilizada na forma:

$$v(m,n) = \frac{1}{9} \sum_{k=-1}^{1} \sum_{l=-1}^{1} y(m-k, n-l) \tag{5.2}$$

o código associado a esta transformação substitui o valor de cada pixel pela soma com seus oito vizinhos e divide o total por 9.

5.1.4.1 Condição de Contorno

A aplicação do algoritmo de convolução expressa na equação (5.2) à imagem $m \times n$ sofre restrições em sua borda, uma vez que os pixels dispostos no contorno não possuem os 8 vizinhos envolvidos na operação (Figura 5.4). Por isto somente o conjunto C dos pixels localizados entre as bordas serão calculados.

$$C = \{y(i,j) \in E \mid 0 < i < n, 0 < j < n\}$$

Por isto, na imagem 128×256 considerada, apenas o núcleo de 126×254 *pixels* será processado.

Como mostra a Figura 5.4, o valor da imagem de saída I_{out} (direita) é computado como a média dos 9 (nove) *pixels* da imagem de entrada I_{in} (esquerda) que estão cobertos pela janela 3×3 com centro em (i,j) apresentando o conceito da operação de um filtro de média.

5.1.5 Subtração de Imagens

Com as imagens (referência e frame atual) suavizadas pela convolução com o kernel (expressão 3.6) armazenadas na memória, foi implementada a subtração

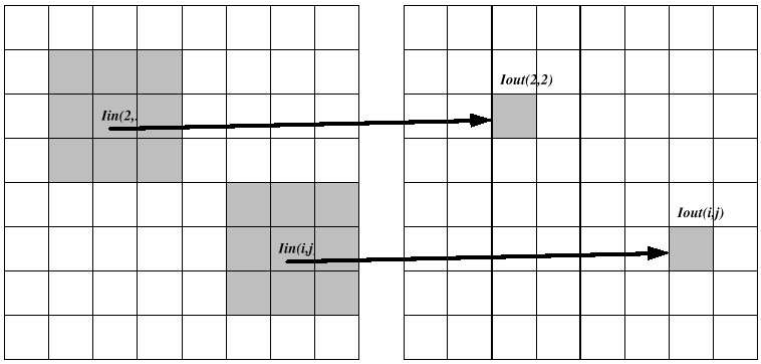

Figura 5.4: Conceito de uma operação baseada em um kernel 3x3.

de imagens. Esta operação toma como resultado de cada pixel o módulo da diferença entre os valores no frame e na imagem de referência. Na mesma linha de código C++, efetua-se a limiarização da imagem, comparando cada resultado com o valor do limiar, parametrizado para a aplicação. Experiências com os dados disponíveis para esta aplicação estabeleceram este limiar entre os valores 20 e 80.

5.1.6 Quantificação de Mudanças

A análise de cada frame é concluída pela quantificação de mudanças na cena. Ou seja, a quantidade de pixels que sofreram modificações consideráveis em relação a imagem de referência. Sabendo-se que o resultado da operação de limiarização é uma imagem binária (valores 1 ou 0), a soma dos pixels desta imagem fornece o valor denominado Grau de Invasão (G_{inv}).

Para cada frame i processado, obtém-se um grau de invasão $G_{inv}(i)$ correspondente. Durante o processamento da seqüência de frames, são mantidos em memória o resultado dos três últimos frames ($G_{inv}(i-2)$, $G_{inv}(i-1)$ e $G_{inv}(i)$), e estes são aplicados a inequação 5.3 para obtenção de resultados a cada frame processado.

$$Evento = \begin{cases} 1 & se \quad G_{inv}(i-2) < G_{inv}(i-1) > G_{inv}(i) \\ 0 & caso\ contrário. \end{cases} \tag{5.3}$$

Neste caso, *Evento* igual a 1 representa o momento de invasão máxima de um objeto na área de interesse, caracterizando a passagem de um veículo pela

região selecionada da imagem. No protótipo desenvolvido, este evento incrementa uma variável inteira que armazena a contagem de veículos durante a captura de frames.

A execução deste protótipo em software permitiu a análise do funcionamento do algoritmo em um nível de abstração mais baixo, avaliando o desempenho na detecção de veículos, e o tempo de processamento.

5.2 Estudo do Particionamento Hardware/Software

Uma implementação em hardware/software do estudo de caso deste trabalho foi feita, utilizando a plataforma de prototipação descrita no Capítulo 6. O particionamento do sistema em dois módulos (hardware e software) não foi baseado em uma metodologia formal, mas levou-se em consideração métricas de custo de comunicação, tempo e área de hardware como estimadores para o particionamento manual do algoritmo.

A análise do diagrama de blocos representando o fluxo de processamento do algoritmo detector de veículos sugere algumas alternativas de particionamento (Figura 5.5).

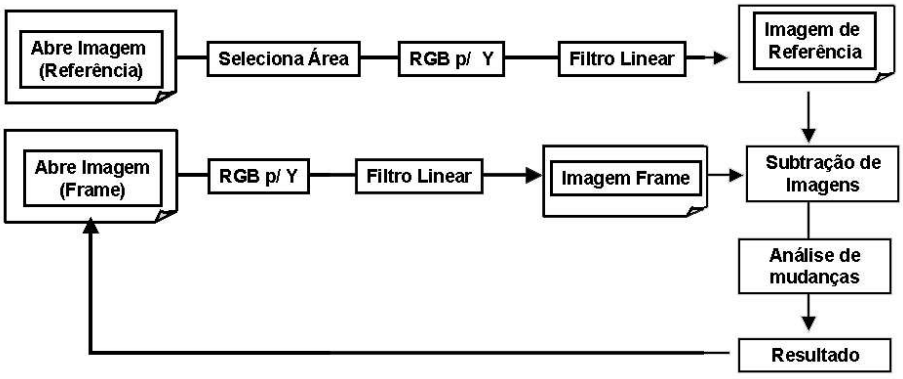

Figura 5.5: Diagrama em blocos do algoritmo de detecção de veículos.

Da forma como está implementado o protótipo do DVI, as imagens capturadas são lidas a partir do disco rígido do PC. Deste modo, a abertura destes bitmaps deverão permanecer nesta partição, sendo executados a partir do método

`serialize` da aplicação em software (ver Secção 5.1.1). Esta alocação é principalmente justificada pelas seguintes razões:

1º) Complexidade do hardware necessário para acessar os arquivos;

2º) Alto custo de comunicação devido ao *overhead* da transação;

3º) O formato serializado do arquivo inibe a utilização do paralelismo de hardware na melhoria do desempenho.

Definindo a área de interesse na imagem, a facilidade do posicionamento da janela com o uso do *mouse* oferece vantagens que justificam sua implementação em software. Além disso, a sua operação não implica no aumento da complexidade do algoritmo de detecção, pois resume-se na tarefa de uma única captura das coordenadas das extremidades da janela de seleção. Por isso, este bloco também deverá permanecer em software.

O processo seguinte refere-se a conversão das imagens coloridas (RGB) em imagens monocromáticas (Y). Este processo merece uma análise especial sobre a opção de sua implementação em software ou em hardware pois a independência de dados das operações pontuais envolvidas permitem o uso de paralelismo na sua execução.

A adoção de uma implementação em hardware, no entanto, implica em um alto custo de comunicação necessário para transferir as 3 matrizes que formam a imagem colorida para a plataforma de hardware.

Além disso, a placa de prototipação adotada, possui apenas um único banco de memória RAM disponível, o que impossibilita o acesso simultâneo a diferentes endereços de memória, necessário para a implementação do paralelismo no processamento.

Por estas razões, a conversão RGB para tons de cinza permaneceu na partição de software do sistema. Assim, uma vez convertida para tons de cinza, a matriz é então transferida do PC para a placa com um custo de comunicação três vezes mais baixo que na etapa anterior, o que já pode ser considerado razoável.

De acordo com a literatura da área [43], na busca pela melhoria de desempenho da execução de um sistema, a atenção do projetista deve se voltar principalmente para a otimização de trechos do código caracterizados por *loops*. Este tipo de processamento ocorre com freqüência na etapa de convolução do filtro adotado no algoritmo de detecção de veículos.

Apesar da limitação do número de bancos de memória na placa impedindo o acesso paralelo de dados na memória, a possibilidade da síntese de um circuito específico para a operação da convolução desta aplicação pode proporcionar um ganho considerável no tempo de execução se comparado com o código gerado pela compilação do programa em C++, executado no ambiente de um sistema operacional multitarefa.

Uma das alternativas consideradas foi transferir o bloco de filtragem para hardware, funcionando como um processador anexado, como é definido em [48], onde a imagem em tons de cinza seria enviada ao hardware e este a retornaria convoluída com o *kernel* 3×3.

Neste caso, existe a necessidade de transferência da imagem do PC para a placa e novamente, após o processamento, a volta da placa para o PC, implicando na duplicação do custo de comunicação.

Para solucionar este problema, decidiu-se tranferir para a partição de hardware as etapas seguintes do processamento, quais sejam, a subtração de imagens, a quantificação e a análise de mudanças. Com isto, o resultado final do processamento seria um *flag*, indicando a ocorrência ou não do veículo na cena.

Este *flag*, representa um custo de comunicação muito baixo e pode ser implementado através do bit de interrupção da arquitetura.

Deste modo, o particionamento hardware/software do algortimo é sugerido pela Figura 5.6, apresentando em uma área hachurada os processos que foram mapeados em hardware.

A filtragem da imagem de referência continuou na partição de software porque este processo é executado somente uma vez, não influindo portanto no desempenho do processamento de cada quadro.

Com isto, a partição de software implementa o controle do algoritmo, abrindo as imagens, convertendo em tons de cinza e enviando ao hardware as imagens de referência filtrada e cada novo frame para ser processado. O hardware filtra cada novo frame e compara com a imagem de referência, contabilizando as mudanças ocorridas na cena, detectando ou não a presença do objeto invasor. Ao final, o resultado informado pelo *flag* de hardware é anunciado pela inteface visual do software.

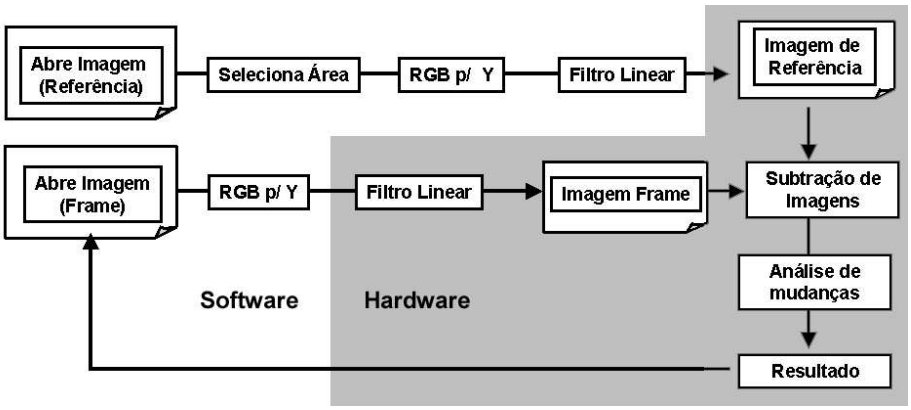

Figura 5.6: Particionamento hardware/software do algortimo de detecção de veículos.

5.3 Interface em Software

Devido ao particionamento do algoritmo, processos de comunicação precisaram ser incluídos nas duas partições. A funcionalidade do software, definido na Secção 5.1, foi sensivelmente modificado, eliminando as etapas descritas nas Subsecções 5.1.4, 5.1.5 e 5.1.6, e adicionando duas outras. Uma referente ao envio de dados de imagens e parâmetros ao hardware e outra para o tratamento de interrupções.

5.3.1 Escrita de Dados

É necessário que o software (PC) forneça à plataforma de hardware os dados das imagens e os parâmetros de sensibilidade do algoritmo (limiar e área). Para isso, são utilizados métodos de classe implementados em C++, fornecidos pelo fabricante da placa de prototipação em sua API (*Application Programming Interface*) [41].

A placa de prototipação é reconhecida pelo software como um objeto da classe XC2S, a qual modela o dispositivo em questão, conectado ao barramento PCI com seus respectivos parâmetros de configuração ajustados como atributos do objeto.

Estão disponíveis para escrita de dados no dispositivo quatro métodos, os quais podem ser invocados a depender da necessidade da aplicação.

5.3.1.1 WriteByte

Este método é utilizado para o envio de um único byte à plataforma de hardware. Recebe como parâmetros o endereço, denominado *Offset,* para onde deseja-se enviar o dado, o espaço de memória (ver Secção 6.4) e o valor do byte a ser escrito.

O método *WriteByte* é utilizado neste trabalho para o envio dos dados referentes ao limiar e a área mínima a ser considerada na análise e quantificação de mudanças do algoritmo. Experimentalmente, como foi dito, o valor do `limiar` encontra-se na faixa entre 20 e 80, de modo que seu valor é diretamente enviado ao hardware pelo método WriteByte. Entretanto, o parâmetro `area` possui valores experimentais situados entre 1000 e 2000, os quais não podem ser diretamente representados por um único byte (8-bit). Seriam necessários 11-bits para representar tais valores. Por esta razão, este parâmetro é dividido por 32 antes de ser enviado, reduzindo assim a largura da palavra binária para 1 byte (Equação 5.4).

$$\mathrm{AREA}_{\mathrm{enviada}}[8\text{-bit}] = \frac{\mathrm{AREA}_{\mathrm{considerada}}[11\text{-bit}]}{32} \tag{5.4}$$

5.3.1.2 WriteWord e WriteDword

Análogos ao anterior, estes método permitem o envio de palavras de 2 e 4 bytes, respectivamente.

Devido à limitações na interface PCI, específicas da plataforma de prototipação utilizada, para algumas aplicações como o estudo de caso desta dissertação, a utilização destes métodos podem não resolver tão facilmente o problema da largura da palavra necessária para enviar o parâmetro de área. Como será descrito no Capítulo 6, o barramento é limitado a 8-bits, de forma que o *device driver* da placa implementa duas ou quatro escritas de 8-bits para satisfazer a transferência solicitada.

5.3.1.3 WriteBlock

Este método permite o envio de um bloco de dados de até 256 bytes para a placa. Apesar de ser implementado por escritas sucessivas de 1 byte, o modo

burst do barramento PCI reduz o *overhead* asociado ao protocolo de comunicação estabelecido entre o PC e o dispositivo [37].

Recebe como parâmetro uma referência de memória para uma estrutura contendo o bloco de dados e informações sobre o tamanho, o *Offset*, o incremento de endereços entre cada byte escrito, e o espaço de memória mapeado no dispositivo [41].

Neste trabalho, o método WriteBlock foi utilizado para enviar as imagens (256×128 pixel) para a partição de hardware, através de 128 escritas sucessivas, uma para cada *scan line,* de blocos de 256 bytes.

5.3.2 Mapeamento de Instruções em Memória

O parâmetro *Offset* das funções de escrita de dados foi utilizado para definir um conjunto de instruções que pudessem ser compreendidas pela partição de hardware, distinguindo os diferentes tipos de dados enviados para um processamento adequado de cada caso.

Desta forma, todos os dados da imagem de referência são escritos no endereço 00h do espaço de endereçamento do FPGA e os dados relativos a imagens de cada frame, são escritos no endereço 01h.

Os parâmetros de sensibilidade também possuem instruções específicas para a escrita de dados, como mostra a Tabela 5.1.

Endereço (Instrução)	**Descrição**
00h	Imagem de Referência
01h	Imagem de um Frame
02h	Limiar
03h	Área

Tabela 5.1: Conjunto de instrues

É importante salientar que este artifício na comunicação de dados foi proposto para esta aplicação específica. Portanto, é necessário que o projeto de hardware (FPGA) implemente a decodificação deste mapeamento, possibilitando o correto funcionamento do protocolo.

5.3.3 Tratamento de Interrupções

O resultado do processamento em hardware é enviado ao componente de software, através de um sinal (*flag*) portando a informação sobre a ocorrência na detecção de veículos. Esta comunicação de retorno é implementada pelo recurso de interrupção suportado pela interface PCI entre a placa de prototipação (`XC2S_EVAL`) e o PC.

O sinal é complementado por um *buffer* contendo o motivo da interrupção. Do modo como o sistema foi desenvolvido, o sinal de interrupção apenas informa ao software a finalização do processamento de um frame enquanto o *buffer* indica, de acordo com o grau de invasão ($\mathsf{G}_{\mathsf{inv}}(\mathtt{i})$) daquele frame e dois outros imediatamente anteriores ($\mathsf{G}_{\mathsf{inv}}(\mathtt{i}-1)$ e $\mathsf{G}_{\mathsf{inv}}(\mathtt{i}-2)$) a invasão ou não da área de interesse por um veículo. Em caso afirmativo, o buffer contém o valor 1, caso contrário o valor 0.

O tratamento da interrupção pela partição de software é realizado pelo método `NotifyFunc` de uma classe que herda a `NotifyEvent`, como indicado em [41].

A código do método NotifyFunc implementa o incremento de uma variável de contagem de veículos, a qual tem seu valor apresentado ao usuário pela interface visual da aplicação desenvolvida como protótipo (Figura 5.3).

5.4 Implementação da Partição de Hardware

A partição de hardware inclui além da funcionalidade relativa aos módulos de processamento de imagem, os processos relacionados ao interfaceamento com a partição de software. Os primeiros referem-se ao processamento já especificados na descrição inicial do sistema, enquanto que estes últimos são devido a necessidade da síntese de interface entre as duas partições [21], [19].

O algoritmo implementado em hardware foi descrito como uma máquina de estados finitos (*FSM - Finite State Machine*) com 23 estados que abrange os processos de recepção e armazenamento de dados de imagens e parâmetros de sensibilidade, recuperação de dados na memória, convolução, subtração de matrizes, análise de resultados e geração de interrupção.

5.4.1 Processos

5.4.1.1 Transferência Software/Hardware da Imagem de Referência

Os dados da imagem de referência são transferidos do software (PC) para o hardware (FPGA) pela operação de escrita em blocos, segundo o mapeamento da Tabela 5.1. O FPGA recebe seqüencialmente os 256 bytes enviados em 128 blocos, até completar o armazenamento na memória local da placa de toda a imagem de referência (32 kBytes). A memória local da placa de prototipação, controlada pelo FPGA, tem capacidade para armazenar até 512kBytes.

Os dados de referência são mapeados no mesmo endereço (00h) da interface paralela mas, o projeto de hardware deve discernir cada novo byte através do comportamento dos sinais de controle de comunicação ($\overline{Wr}$ e $\overline{CS2}$).

A imagem de referência é armazenada no bloco de endereços 0000h a 7FFFh da memória (Figura 5.7). Como será apresentado no Capítulo 6, o FPGA é um componente intermediário entre a interface PCI e o chip de memória RAM da placa (Figura 6.8). Portanto, todo o controle de acesso à memória é feito através do FPGA.

5.4.1.2 Transferência Software/Hardware de um Frame

Semelhante à transferência da imagem de referência, os dados de cada novo frame provenientes da partição de software são enviados via barramento PCI à placa de prototipação, através de métodos de escrita em blocos no endereço 01h do mapeamento da interface paralela. O hardware captura cada byte da imagem e os armazena na memória local, endereçando o espaço de memória (8000h - FFFFh). Para cada frame a ser processado, os novos dados enviados sobrescrevem o frame anterior, sendo armazenados no mesmo espaço de memória (Figura 5.7).

5.4.1.3 Transferência dos Parâmetros de Sensibilidade do Algoritmo

O parâmetro de `Limiar` é enviado pelo PC ao endereço 02h e este valor é diretamente armazenado na memória interna do FPGA.

Como já foi explicado na Subsecção 5.3.1, o valor do parâmetro `Área` não

Figura 5.7: Mapeamento de memória utilizado

pode ser utilizado diretamente, uma vez que seu valor adequado foi dividido por 32, para viabilizar a comunicação Sw/Hw. Por isto, o byte recebido no endereço 03h (Tabela 5.1) é multiplicado por 32, retornando ao seu valor original por uma operação binária de 5 deslocamentos à esquerda e armazenados internamente no FPGA. Esta operação é realizada pela seguinte linha de código em VHDL, que converte o dado em um sinal de 16-bits.

```
area := ‘‘000’’ & Data_in_pita2 & ‘‘00000’’;
```

A operação descrita, apesar de alterar o valor original do parâmetro de área mínima adotado, pela redução da sua precisão na ordem de 2^5, não causa alterações perceptíveis no correto funcionamento do sistema.

5.4.1.4 Captura dos Pixels para a Convolução

Uma vez armazenados na memória, os dados do novo frame são lidos pelo FPGA, dando início ao processamento previsto na descrição inicial. Cada nove pixels da imagem de frame são seqüencialmente capturados na memória e acumulados em um registrador interno.

Na convolução da imagem com o *kernel* escolhido, o somatório dos valores associados aos nove pixels envolvidos na operação é dividido por 9 para obter a média espacial (equação 5.2). Dada a complexidade associada a implementação de um divisor por 9 numa descrição comportamental [9], ao invés disso, foi tomada uma solução alternativa de comparar o somatório com o valor do pixel de referência multiplicado por 9, compatibilizando assim a faixa de valores a serem comparados.

Com isso, a convolução implementada em hardware além de ter sido simplificada pela natureza dos coeficientes do *kernel* (Secção 5.1.4), dispensou a divisão por 9 efetuada a cada somatório (equação 5.5).

$$v(m,n) = \sum_{k=-1}^{1} \sum_{l=-1}^{1} y(m-k, n-l) \tag{5.5}$$

5.4.1.5 Captura do Pixel de Referência para Comparação

Uma vez que os valores da imagem de referência precisam ser compatibilizados com o resultado da convolução apresentada na equação 5.1.4, o byte coletado da memória, no espaço de imagem de referência, é multiplicado por 9 e subtraído do valor obtido no somatório. O módulo da diferença é classificado (0 ou 1) pelo parâmetro de Limiar e acumulado como grau de invasão (G_{inv}) durante todo o processamento da imagem.

5.4.1.6 Análise de Resultados e Geração de Interrupção

Ao final do processamento, o valor obtido para o grau de invasão é comparado com os resultados de processamento de dois frames anteriores (mantidos em registros), analizando-os de acordo com a inequação 5.3. Desta comparação,o valor associado a *Evento* (0 ou 1) é enviado ao *buffer* identificador (HOST5..0), indicando a razão da interrupção (ver Secção 6.6), que é imediatamente gerada na interface PCI, pelo ajuste do sinal `INT0` do FPGA

Esta interrupção sinaliza à partição de software sobre a finalização do processamento em hardware.

5.4.2 Mapeamento de Memória

Apesar do alto nível de abstração proporcionado pela descrição comportamental (código VHDL) do projeto de *hardware*, existe ainda uma grande diferença em relação a facilidade de acesso às variáveis no contexto de execução, encontrado na implementação em *software*, que é auxiliado pelos aplicativos do ambiente de desenvolvimento (compilador, *linker* e sistema operacional).

Por isto, o mapeamento de variáveis na implementação em *hardware* exige do projetista total controle das operações de acesso a memória e uma efetiva utilização de ponteiros.

A Matriz (5.6) apresenta o mapeamento da imagem de referência na memória de acordo com a alocação proposta (Figura 5.7). Os números referem-se aos endereços de memória onde cada *pixel* da imagem de referência é armazenado.

$$
\begin{array}{cccccccc}
0000\text{h} & 0001\text{h} & 0002\text{h} & \ldots & \longrightarrow & 00\text{FEh} & 00\text{FFh} & \\
0100\text{h} & 0101\text{h} & 0102\text{h} & \ldots & \longrightarrow & 01\text{FEh} & 01\text{FFh} & \\
0200\text{h} & 0201\text{h} & 0202\text{h} & \ldots & \longrightarrow & 02\text{FEh} & 02\text{FFh} & \\
0300\text{h} & 0301\text{h} & 0302\text{h} & \ldots & \longrightarrow & 03\text{FEh} & 03\text{FFh} & \\
0400\text{h} & 0401\text{h} & 0402\text{h} & \ldots & \longrightarrow & 04\text{FEh} & 04\text{FFh} & (5.6) \\
0500\text{h} & 0501\text{h} & 0502\text{h} & \ldots & & \ldots & \ldots & \\
\ldots & \ldots & \ldots & & \searrow & \downarrow & \downarrow & \\
\downarrow & \downarrow & \downarrow & & & 7\text{EFEh} & 7\text{EFFh} & \\
7\text{F00h} & 7\text{F01h} & 7\text{F02h} & \ldots & \longrightarrow & 7\text{FFEh} & 7\text{FFFh} &
\end{array}
$$

Conforme explicado na Subsecção 5.1.4, a condição de contorno adotada restringe o processamento aos *pixels* que não fazem parte da borda da imagem. Ou seja, a partir da segunda coluna da segunda linha (0101h) até a penúltima coluna da penúltima linha (7EFEh).

Analogamente, o mapeamento de um frame (5.7), define os endereços utilizados da memória para armazenar cada pixel dos novos quadros capturados.

$$
\begin{array}{cccccccc}
8000\text{h} & 8001\text{h} & 8002\text{h} & \ldots & \longrightarrow & 80\text{FEh} & 80\text{FFh} & \\
8100\text{h} & 8101\text{h} & 8102\text{h} & \ldots & \longrightarrow & 81\text{FEh} & 81\text{FFh} & \\
8200\text{h} & 8201\text{h} & 8202\text{h} & \ldots & \longrightarrow & 82\text{FEh} & 82\text{FFh} & \\
8300\text{h} & 8301\text{h} & 8302\text{h} & \ldots & \longrightarrow & 83\text{FEh} & 83\text{FFh} & \\
8400\text{h} & 8401\text{h} & 8402\text{h} & \ldots & \longrightarrow & 84\text{FEh} & 84\text{FFh} & (5.7) \\
8500\text{h} & 8501\text{h} & 8502\text{h} & \ldots & & \ldots & \ldots & \\
\ldots & \ldots & \ldots & & \searrow & \downarrow & \downarrow & \\
\downarrow & \downarrow & \downarrow & & & \text{FEFEh} & \text{FEFFh} & \\
\text{FF00h} & \text{FF01h} & \text{FF02h} & \ldots & \longrightarrow & \text{FFFEh} & \text{FFFFh} &
\end{array}
$$

Com base na alocação acima descrita, o algoritmo em *hardware* deverá, após ter armazenado a imagem de referência e um frame, começar a captura dos pixels para a convolução.

Para isto, 3 (três) ponteiros de memória foram implementados: `mem_pointer_ref`, `mem_pointer_frame` e `mem_pointer_neigh`.

O primeiro é utilizado para endereçar os *pixels* do espaço de referência, o segundo acessa o espaço de frames e o último serve para coordenar a coleta dos *pixels* vizinhos de cada elemento.

O primeiro passo da etapa de convolução é adicionar os valores armazenados nos endereços 8000h, 8001h, 8002h, 8100h, 8101h, 8102h, 8200h, 8201h e 8202h como a soma dos vizinhos de 8101h. Em seguida, compara-se com o valor de referência correspondente, armazenado no endereço 0101h, multiplicado por 9. Esta multiplicação é implementada pela soma do valor contido na memória (`data_std`), com seu deslocamento de 3 bits para esquerda ($\times 8$).

```
datax9 <= CONV_INTEGER(data_std + (data_std & ‘‘000’’));
```

Prossegue-se então com a etapa seguinte de limiarização, analisando a diferença encontrada no referido *pixel*, associando-o o valor 1 ou 0, de acordo com a expressão 4.1.

O processo continua coletando os elementos vizinhos de 8102h, e seu correspondente 0102h, analogamente até o último elemento válido da imagem (endereço FEFEh), quando então passa-se para a etapa de análise de resultados e geração de interrupção.

Pode-se perceber que o endereçamento de cada elemento computado não é trivial, sendo necessário um controle completo da atualização dos ponteiros, considerando o final de cada linha da imagem (*scan line*) e do *kernel* de convolução.

5.4.3 Ferramentas de Desenvolvimento

A partição de hardware do algoritmo foi implementada em uma descrição comportamental em VHDL. Para isso, utilizou-se o ambiente integrado de ferramentas Xilinx Foundation 3.1i Series [51].

Este ambiente suporta todas as fases de desenvolvimento do projeto de hardware em FPGAs (Figura 5.8), partindo da descrição textual do código VHDL (*Design Entry*), sua simulação (*Simulation*), implementação (*Implementation*) até a etapa de geração do arquivo de configuração (*bitstream*) do dispositivo.

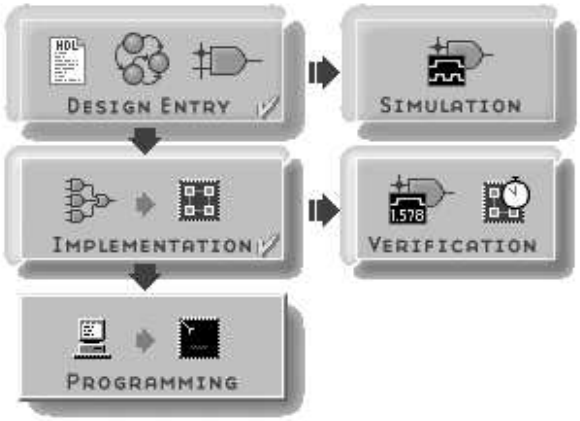

Figura 5.8: Fluxo de desenvolvimento de hardware Xilinx Foundation 3.1i Series

A descrição comportamental é logicamente sintetizada, gerando uma *netlist* estrutural representando a funcionalidade do algoritmo. Esta síntese lógica é armazenada no ambiente de projeto em forma de macrocélula (*Macro*) para que possa ser instanciada nas etapas posteriores do desenvolvimento do *design* digital.

A *macro* gerada (Figura 5.9) constitui o sistema em alto nível implementado no FPGA. O *port* denominado `ADDR_PITA2` é conectado ao barramento de endereços, enquanto que o `DATA_IN_PITA2` ao barramento de dados da interface

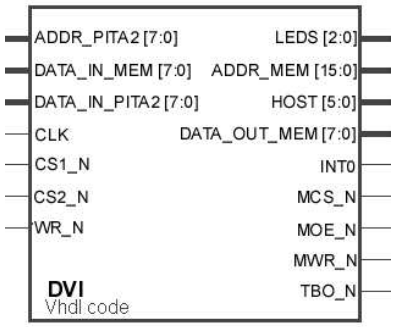

Figura 5.9: Macrocélula do algoritmo detector de veículos em hardware.

PCI. Adicionalmente, os *ports* `CS1_N` (*Chip Select* 1), `CS2_N` (*Chip Select* 2) e `WR_N` (*Write Enable*) controlam a comunicação com o hardware e também são conectados a interface.

O sinal de *clock* provém do oscilador a cristal disponível na placa de prototipação e deve ser ligado ao *port* `CLK`. Para facilitar a depuração do sistema durante a fase desenvolvimento e testes, existe uma conexão `LEDS` disponibilizada para o monitoramento de alguns estados da FSM através dos diodos emissores de luz (LEDs) montados na placa.

O acesso do FPGA à memória SRAM (*Static RAM*) é realizada pelos barramentos `ADDR_MEM` (endereços), `DATA_IN_MEM` (dados de entrada da SRAM para o FPGA) e `DATA_OUT_MEM` (saída de dados do FPGA para escrita na memória), e são controlados pelos sinais dos *ports* `MCS_N` (*Memory Chip Select*), `MOE_N` (*Memory Output Enable*) e `MWR_N` (*Memory Write Enable*).

Para possibilitar a conexão dos *ports* de entrada e de saída de dados da memória nos mesmos pinos (`INOUT`) do FPGA, foi implementado um sinal TBO_N (*Tri-state Bus Output*) controlando o fluxo de dados no dispositivo. Finalmente, o *bit* de interrupção do sistema, sinal `INT0`, é complementado pelo *port* `HOST[5:0]` que identifica o motivo da interrupção para a partição de software.

Através do *Schematic Editor* (Xilinx Foundation Series), a *macro* é estruturalmente conectada aos *Pads* (pinos) do dispositivo, complementando a *netlist* com as informações de `I/O` (*Input/output*) e locação (`LOC`) que descreve o hardware do sistema.

A etapa seguinte é a simulação funcional do projeto de hardware, através da

ferramenta *Logic Simulator* do ambiente. Vetores de teste foram implementados e todo o processo foi validado.

Estando correta a descrição VHDL e por conseguinte sua simulação, a próxima etapa é a implementação da *netlist* no dispositivo alvo (Xilinx Spartan-II XC2S200). Esta etapa é composta por um processo inicial denominado *Translate* que corresponde a reunião de toda a especifição de entrada (*netlist*, restrições, etc.) traduzindo em um modelo interno do ambiente. Em seguida, o *Map* executa o mapeamento da síntese lógica no FPGA especificado, adequando o modelo à disponibilidade lógica do dispositivo. O processo de *Place&Route* posiciona e roteia o projeto no chip e o *Timing* avalia os atrasos nas redes versus as restrições de tempo especificadas. O último processo, denominado *Configure*, finaliza a implementação com a formação do arquivo de configuração (*bitstream*) do FPGA (Figura 5.10).

Figura 5.10: Fluxo de implementação das ferramentas Xilinx Foundation Series.

Devido a necessidade de se trabalhar com um sinal de *clock* em uma freqüência maior ou igual a velocidade do barramento PCI (33MHz), foi utilizado o *clock* pré-ajustado da plataforma de prototipação de 40MHz. Esta freqüência corresponde a um período de tempo de 25ns o que restringe o atraso máximo nas redes do roteamento interno do *chip*. Utilizando recursos de restrição temporal da ferramenta de síntese, conseguiu-se atingir um período de $18,43$ns ($54,259$MHz), satisfazendo os requisitos de comunicação entre os componentes de hardware e software.As heurísticas utilizadas pelo ambiente para a solução de *lay-out* do chip não fazem parte do escopo deste trabalho.

O resultado do processo de síntese dos módulos implementados em hardware é apresentado na Tabela 5.2.

Lógica	Quant.	% do Total disponível
Slices (CLB)	264	11
Flip-Flops	215	4
4-input LUTs	427	9
IOBs	51	36
GCLKs	1	25
GCLKIOBs	1	25
Total (gates)	**5.086**	**2,54**

Tabela 5.2: Resumo da sntese lgica

É importante relatar que estes resultados só foram possíveis através de um processo manual de refinamento do código VHDL especificado, reduzindo ao máximo os níveis lógicos em cada estado da FSM modelada. Orientações do fabricante do FPGA indicam possíveis otimizações no código, substituindo comandos e estruturas equivalentes [23], [8].

5.4.4 Configuração do FPGA

O desenvolvimento do projeto de hardware é finalizado com o *download* do arquivo de configuração no FPGA. O bitstream gerado pela implementação (arquivo .bit) é transformado para o formato PROM EXORmacs (arquivo .exo) utilizando a ferramenta *PROM File Formater* [52], para adequar-se ao padrão recomendado para a utilização da placa de prototipação Cesys `XC2S_Eval` [34]. Este arquivo pode ser "descarregado" no FPGA através dos métodos de classe integrantes da API que acompanha a placa de prototipação.

Estes métodos são invocados pelo aplicativo que implementa a partição de software, permitindo o *download* do arquivo de configuração em tempo de execução do sistema, possibilitando futuras aplicações desta plataforma em sistemas dinamicamente reconfiguráveis.

5.5 Conclusão

Neste capítulo foi apresentado a implementação do algoritmo de processamento de imagens sob duas abordagens diferentes. Inicialmente todos os módulos foram implementados em software, em uma linguagem compilada orientada a objetos

C++, possibilitando a análise do funcionamento completo do sistema, sendo observadas métricas de tempo de execução e custo de desenvolvimento, de acordo com as restrições iniciais impostas pelo problema estudado.

O protótipo em execução apresentou a corretude na sua funcionalidade, detectando e contando os veículos em movimento que atravessam a área de interesse da cena. O aplicativo desenvolvido constitui uma ferramenta de suporte, para o refinamento da implementação, exibindo dados de processamento como parâmetros adotados, resultados obtidos e tempo de execução de tarefas. A análise dos resultados, quando confrontados com as restrições temporais do projeto, sugeriram o mapeamento de funções de processamento complexo em uma implementação DSP de alto desempenho.

Foi proposto a utilização de uma plataforma de hardware reconfigurável, para a solução dos problemas de execução do algoritmo em uma aplicação de tempo real. Otimizou-se o processamento de módulos através de uma descrição em hardware, obtendo a mesma funcionalidade, porém com um melhor desempenho.

O algoritmo passou então a operar em um ambiente heterogêneo de hardware/software codesign, que reúne as vantagens de cada abordagem, no desenvolvimento de um protótipo de alta performance e baixo custo.

Cada etapa do desenvolvimento do estudo de caso (DVI), explicadas nas diversas secções deste capítulo, constituem uma metodologia de projeto para aplicações em processamento de imagens na arquitetura proposta. Neste trabalho, uma placa de prototipação para projetos em FPGA foi adicionada a um computador PC, como objeto de estudo em sistemas hardware/software codesign para aplicações DSP.

A arquitetura da plataforma reconfigurável utilizada será apresentada no próximo capítulo, onde os detalhes da plataforma justificarão algumas das decisões de projeto adotadas, durante o desenvolvimento do projeto do detector de veículos por imagem.

Capítulo 6

Plataforma de Prototipação

6.1 Descrição do Sistema

O sistema completo do detector de veículos por imagens (DVI) é formado por uma câmera de vídeo conectada a uma placa de captura que converte os sinais analógicos em padrões digitais. A placa A/D de captura de imagens está conectada a um computador PC através de um dos seus barramentos de expansão (PCI ou ISA).

Internamente, foi instalada no PC de uso geral que recebe os sinais provenientes da câmera, uma placa de prototipação baseada em um FPGA, conectado ao barramento PCI.

A Figura 5.2 ilustra o sistema proposto, onde as imagens de trânsito capturadas pela câmera de vídeo são transferidas ao computador equipado com o hardware adicional (placa de prototipação com FPGA).

Consideremos como *plataforma de captura* de imagens o subsistema composto pela câmera de vídeo e a placa de captura, e como *plataforma de execução* do algoritmo a composição do PC com a placa contendo o FPGA.

O PC utilizado neste trabalho possui a seguinte configuração: Processador AMD$^{(R)}$ `K6-II` de `500` MHz, com `128` MBytes de memória RAM e `10` GBytes de disco rigido, com `50`% de sua mídia ocupada.

A placa de prototipação, conectada ao barramento PCI do computador é classificada como um "processador anexado", já que sua conexão se dá através de um barramento de expansão [48]. Esta classificação difere de um co-processador, já

que a placa não está diretamente conectada ao barramento do processador *host*. Neste trabalho a placa de prototipação utilizada é uma Cesys `XC2S_EVAL [34]`

XC2X_EVAL é uma plataforma de desenvolvimento para projetos com FPGAs da família Xilinx Spartan-II [50]. A placa permite aplicações variadas e a conexão com hardware externo através de conectores disponíveis. A placa que foi utilizada para a implementação do estudo de caso desta dissertação foi equipada com um FPGA Xilinx `XC2S200-5PQ208C`, contendo lógica Xilinx disponível equivalente a 200.000 gates. Este dispositivo de lógica programável recebe suas funções internas de acordo com a configuração a partir do PC. Portanto, um ajuste dependente da aplicação de toda a lógica é possível a qualquer instante, sem a necessidade de um `dispositivo programador`. Após o processo de inicialização do PC, o FPGA é carregado com a configuração desejada e através do software da aplicação uma nova configuração é permitida, alterando o projeto sem a necessidade de reiniciar o computador. O diagrama de blocos da XC2S_EVAL é apresentado na Figura 6.1.

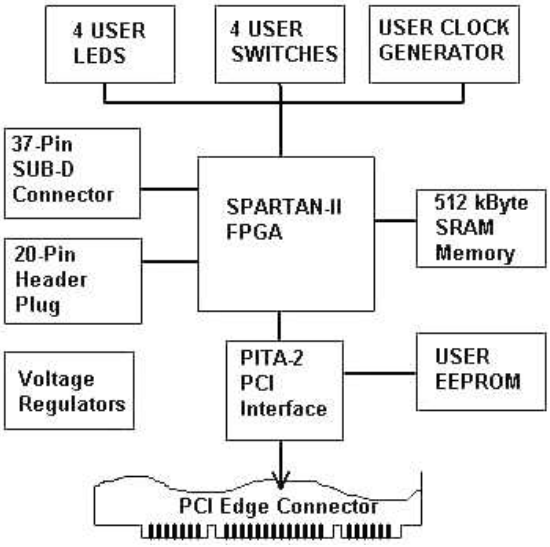

Figura 6.1: Diagrama de blocos da arquitetura da XC2S_EVAL

Um banco de memória SRAM de 512kByte está incluído na placa e seus pinos estão diretamente conectados ao FPGA podendo ser livremente utilizada pelo projeto. A interface PCI da placa é implementada por um dispositivo

adicional, o `PITA-2,` um ASIC para aplicações de dados e telefonia fabricado pela Infineon [31]. Este dispositivo dispensa a necessidade do desenvolvimento ou a utilização de um Core PCI [10] no FPGA, além de não exigir do projetista conhecimentos detalhados sobre o barramento PCI.

6.2 Dispositivo de Hardware Reconfigurável

Um FPGA da Família Spartan-II da Xilinx é utilizado na XC2S_EVAL como dispositivo reconfigurável de prototipação. Este dispositivo possui uma lógica disponível equivalente a 200.000 gates e o arquivo de configuração pode ser descarregado através da interface PCI em suas células internas de memória RAM estática.

A simulação do projeto e a síntese do arquivo de configuração pode ser obtida através das ferramentas de desenvolvimento da Xilinx, tais como Foundation 3.1, ISE, WebPack, etc. que contenha a biblioteca da família Spartan-II. Neste trabalho, utilizou-se a ferramenta Foundation 3.1 com o *service pack* 8 instalado (`Build 3.1.181`).

A família Xilinx Spartan-II de FPGAs possui uma arquitetura programável regular e flexível baseada em CLBs (*Configurable Logic Blocks*), e um anel de células para entrada e saída IOBs (*Input/Output Blocks*) localizadas em todo o perímetro do *chip*. O circuito comporta quatro DLLs (*Delay-Locked Loops*), um em cada canto do chip e duas colunas de block RAM dispostas lateralmente entre os CLBs e os IOBs (Figura 6.2).

Semelhantes a PLLs (*Phased-Locked Loops*), os circuitos DLLs podem eliminar o "escorregamento" do sinal entre o pino de entrada do clock e as entradas internas de clock no dispositivo. Cada DLL monitora o clock de entrada e o clock distribuído e, automaticamente ajusta o elemento de atraso do sinal. Este sistema elimina de maneira efetiva o atraso no clock distribuído, garantindo que a subida do sinal alcance cada `Flip-Flop` interno de forma sincronizada com o clock de entrada.

O bloco básico de um CLB Spartan-II é a LC (*Logic Cell*). Cada LC inclui um gerador de funções de 4 entradas (*look-up table*), lógica *carry*, e elementos de armazenamento (registradores). Cada CLB da Spartan-II contém quatro LCs,

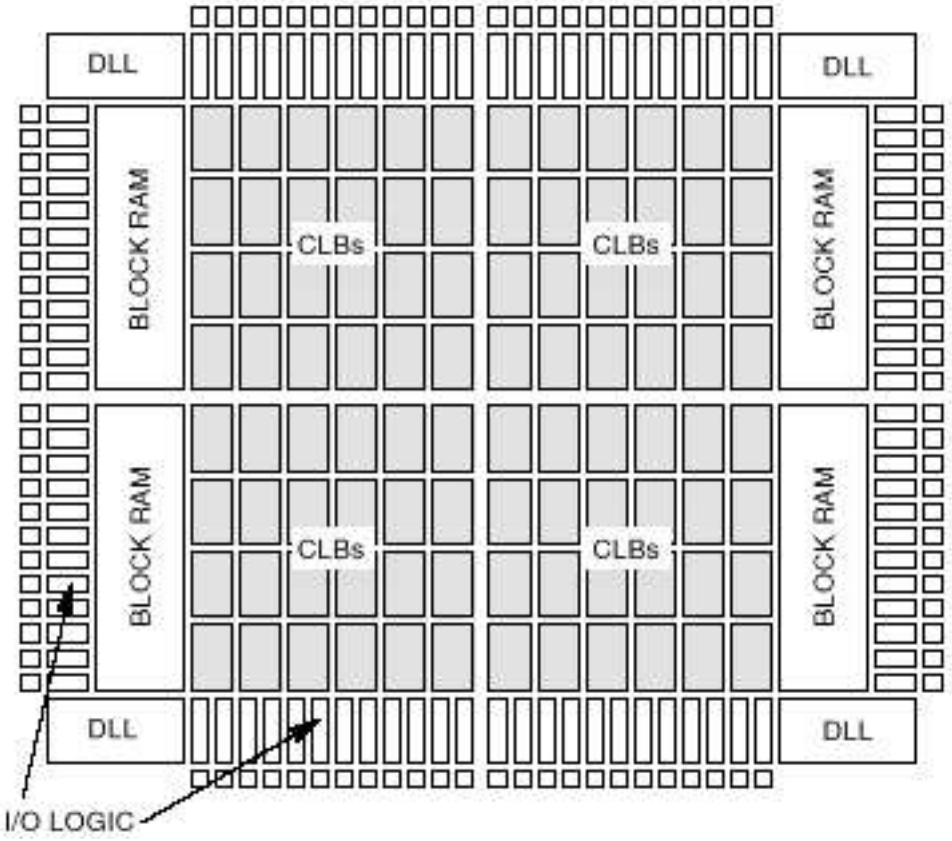

Figura 6.2: Diagrama de blocos da família Spartan-II

organizadas duas a duas em fatias similiares. Uma destas fatias é mostrada na Figura 6.3.

Os IOBs Spartan-II são caracterizados por entradas e saídas de alta velocidade que suportam uma grande variedade de interfaces com barramentos e memórias de última geração. Nestes blocos são sintetizados toda a lógica necessária para a conexão externa do FPGA, provendo *buffers*, resistores *pull-up* e *pull-down*, e ainda registradores de sincronismo de sinais, que serão comentados na seção seguinte.

FPGAs da família Spartan-II incorporam blocos de memória RAM interna que complementam as LUTs (*Look-up-Tables*) existentes nos CLBs distribuídos pelo chip. A quantidade de memória RAM disponível está relacionado à densidade lógica do dispositivo. No XC2S200 (200k) instalado na placa de prototipação, existem 14 destes blocos totalizando o equivalente a 56kbits de memória

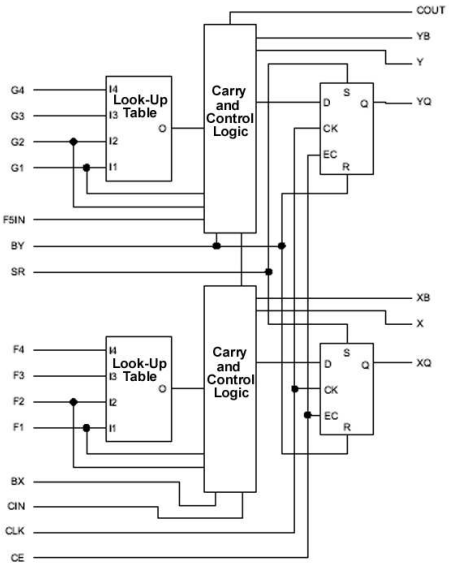

Figura 6.3: Metade de uma CLB Spartan-II

RAM. Para o estudo de caso apresentado neste trabalho, a memória interna do FPGA foi insuficiente para o armazenamento dos dados de imagens, e por isso utilizou-se o chip de memória RAM instalado da placa (Secção 6.5).

6.3 Clock do Sistema

O clock principal da placa para o FPGA provém de um oscilador à cristal com valor padrão de 40MHz (Período = 25 ns). Utilizando este sinal como relógio principal do projeto é importante sincronizar cada sinal assíncrono de entrada com um Flip-Flop antes de utilizá-lo internamente (Figura 6.4). A ausência deste Flip-Flop na entrada de cada sinal de controle pode provocar instabilidades no funcionamento de Máquinas de Estados Finitos (*FSM - Finite State Machine*), as quais podem perder o estado ou entrar em estados ilegais. Portanto, cuidado ao manipular o clock é necessário para se evitar problemas no interfaceamento com o PITA-2.

Através do editor de esquemáticos (*Schematic Editor*), foi inserido no projeto, em cada sinal de controle, um Flip-Flop do tipo `D` entre cada buffer de entrada (IBUF) e o pino correspondente na instância da macrocélula (Figura 5.9) do

detector de veículos.

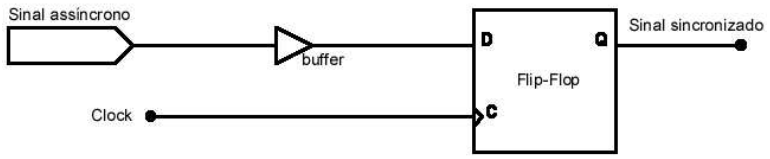

Figura 6.4: Utilização de um Flip-Flop para o sincronismo de sinais de controle.

Um outro ponto importante a considerar é a estabilidade do barramento de dados provenientes do FPGA durante operações de leitura. O dado não deve se alterar durante toda a transação. Neste caso, a solução é a utilização de um `Latch` dentro do FPGA sendo ativado pelo sinal de leitura (Rd). Esta providência é "transparente" para o projetista, desde que a descrição textual (HDL) defina a atribuição de dados às portas de saída em sincronismo com os sinais de controle do sistema.

6.4 Interface Paralela

Para habilitar a comunicação entre o FPGA e o programa sendo executado no PC, é utilizada a interface paralela do PITA-2. Esta é formada por dois barramentos paralelos (dados e endereços) de 8 bits e sinais de controle tais como *Read* (Rd), *Write* (Wr) e *ChipSelect* (CS), que conectam diretamente o PITA-2 ao FPGA, como apresentado na Figura 6.5.

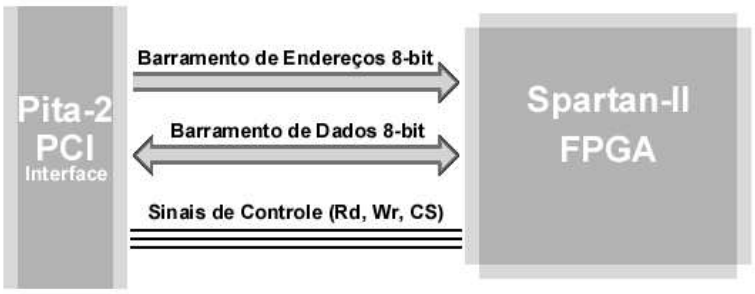

Figura 6.5: Interface paralela entre o Pita-2 e o FPGA

A conexão da interface PCI com todos os elementos que compõem a arquitetura da XC2S_EVAL é feita através do espaço de endereçamento e dos sinais de seleção de recursos (CS), como mostra a tabela 6.1.

Endereço no Barramento PCI	**Chip Select**	**Endereço na Interface Paralela**	**Utilização**
3FFh - 000h	CS0	FFh - 00h	Configuração do FPGA
7FFh - 400h	CS1	FFh - 00h	Configuração do FPGA
BFFh - 800h	CS2	FFh - 00h	Livre para projetos
FFFh - C00h	-	-	Não utilizado

Tabela 6.1: Mapeamento do espao de endereos da placa PCI

A placa é acessada pelo software através de *Device Drivers* como um dispositivo mapeado em memória em um espaço de endereços de 4kByte (000h - FFFh). De acordo com a Tabela 6.1, existe um mapeamento entre o endereço do dispositivo PCI e o endereço interno na interface paralela, associando ao protocolo um CS conforme o endereço acessado. O Pita-2 é responsável por este mapeamento, facilitando a comunicação de dados diretamente aos recursos da XC2S_EVAL.

Para o projetista, esta tradução de endereços é "transparente", pois existem métodos de classe disponíveis na API do dispositivo (XC2S_EVAL), que tomam como parâmetro o endereço da interface paralela, o recurso da placa a ser acessado e o buffer de dados da transação. Estes métodos estão implementados na linguagem C++ e são fornecidos pelo fabricante em conjunto com a placa.

6.5 Banco de Memória

Para o armazenamento local de dados, utilizou-se a RAM estática (SRAM), organizada em 512-kword × 8-bit de tecnologia CMOS (célula de memória: 6-transistor) existente na placa [26]. Este dispositivo possui uma latência de acesso (escrita e leitura) em torno de 50ns (ciclo completo), como mostra o diagrama de tempo abaixo (Figura 6.6).

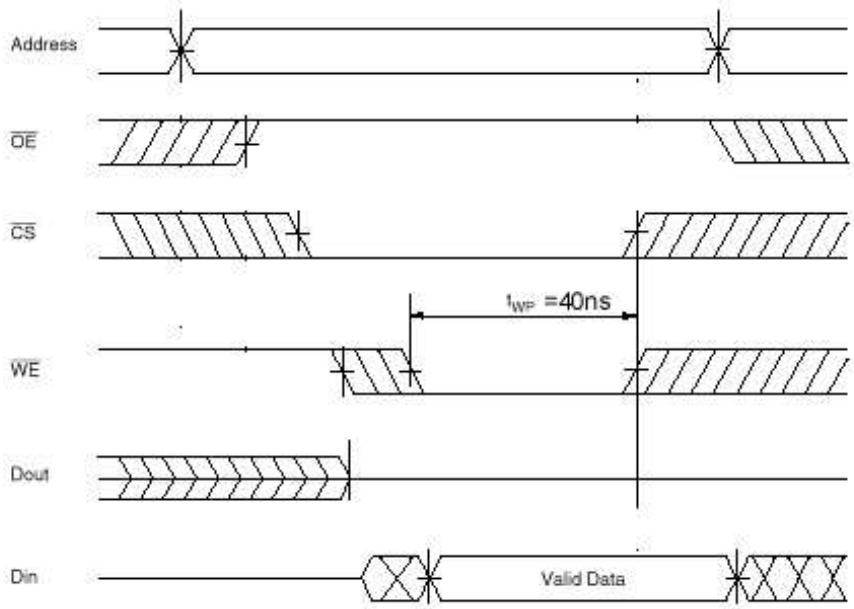

Figura 6.6: Diagrama de tempo de escrita na memória

De acordo com o diagrama da Figura 6.6, o protocolo de escrita na memória pode ser implementado pelo seguinte algoritmo:

i) Dispõe no barramento (*Address*) o endereço onde o dado será escrito;

ii) Ajusta o $\overline{\mathsf{OE}}$ (*Output Enable*) para nível lógico `1`;

iii) Ajusta o $\overline{\mathsf{CS}}$ (*Chip Select*) da memória para o nível lógico `0`;

iv) Disponibiliza no barramento (*Data*) os dados que devem ser escritos

v) Baixa para o nível `0` o sinal $\overline{\mathsf{WE}}$ (*Write Enable*);

vi) Aguarda a latência de 50ns (dois ciclos de clock 40MHz);

vii) Eleva os sinais $\overline{\mathsf{CS}}$ e $\overline{\mathsf{WE}}$ ao nível lógico `1`;

Este algoritmo deverá ser utilizado para a implementação do acesso a memória em situações de escrita de dados. Analogamente, a Figura 6.7 apresenta o diagrama de tempo para o acesso de leitura na memória.

Neste caso, o acesso aos dados guardados na memória deve ser obtido pelo algoritmo abaixo:

i) Dispõe no barramento (*Address*) o endereço de onde o dado será lido;

ii) Ajusta o $\overline{\mathsf{CS}}$ (*Chip Select*) da memória para o nível lógico `0`;

iii) Ajusta o $\overline{\mathsf{OE}}$ (*Output Enable*) para nível lógico `0`;

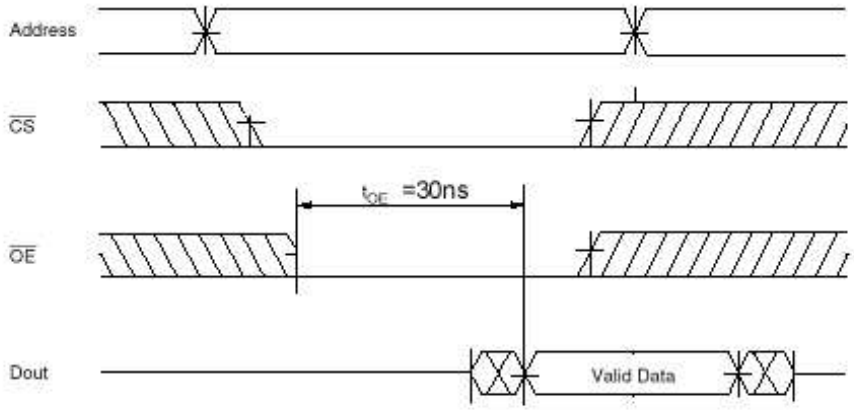

Figura 6.7: Diagrama de tempo de leitura na memória

vi) Aguarda a latência de 50ns (1 ciclo de clock é insuficiente);

vii) Eleva os sinais $\overline{CS}$ e $\overline{WE}$ ao nível lógico 1;

Todos os sinais de dados, endereço e controle da memória estão conectados diretamente ao FPGA da placa, de modo que qualquer acesso à RAM deve ser efetuado através do FPGA, como ilustra a Figura 6.8.

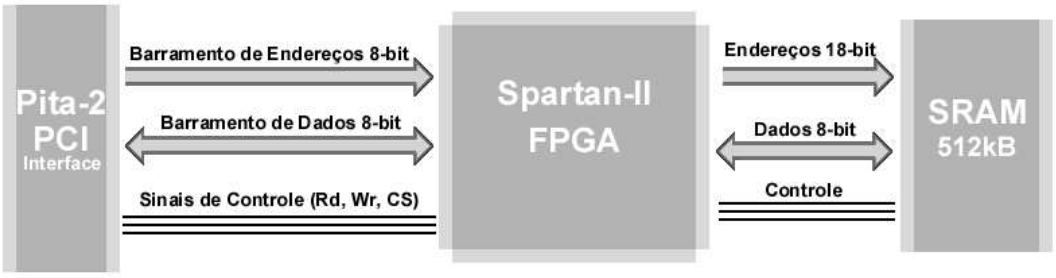

Figura 6.8: Arquitetura da placa: conexão entre a interface, FPGA e memória.

De uma forma geral, de acordo com a arquitetura da placa, a utilização de todos os outros recursos disponíveis tais como conectores, buffers de interrupção e leds, entre outros, é permitida através da pinagem do FPGA, devendo-se portanto, definir adequadamente no projeto de hardware (arquivo de configuração) a ligação interna das portas de entrada e saída de sinais aos recursos.

6.6 Interrupções

O Pita-2 pode gerar uma interrupção no sinal `INTA` do barramento PCI por um evento assíncrono externo. Um desses eventos é o estado do pino `INT0` correspondente ao pino P110 do FPGA. Para gerar uma interrupção, o projeto implementado no FPGA deve utilizar o pino INT0 como saída e ajustá-lo para o nível lógico `1`. Os `drivers` ISR (*Interrupt Service Routine*) [41] irão ler o conteúdo do buffer de interrupções da placa a cada interrupção, que reflete o estado dos pinos `HOST5..0` do FPGA. O projeto deve detectar o acesso ao buffer monitorando o sinal $\overline{\texttt{CS1}}$ (*Chip Select 1*). Quando o sinal é ativado (nivel lógico `0`), o pino de interrupção (P110) deve imediatamente ser desativado, evitando que a interface gere uma interrupção novamente.

O driver informará ao software sobre a ocorrência da interrupção. Juntamente com a mensagem, será enviado o estado dos sinais HOST5..0 que indicará a razão da interrupção.

Nesta dissertação, o projeto desenvolvido como estudo de caso utilizou o recurso de interrupção para informar à partição de software a conclusão do processamento de hardware, resultando na ocorrência ou não de um veículo dentro da área de interesse (Capítulo 4).

6.7 Conclusão

Neste capítulo foram apresentadas as principais características da plataforma de prototipação adotada, para a implementação do algoritmo detector de veículos.

Uma descrição detalhada foi realizada enfocando a placa que constitui o componente de hardware, caracterizando cada dispositivo com suas respectivas funcionalidades no circuito. A descrição apresentada justifica, em parte, o baixo custo de aquisição da placa, uma vez que seus recursos estão limitados à simplicidade de sua configuração.

Alguns dos detalhes técnicos discutidos neste capítulo constituem aspectos práticos, percebidos durante a fase de desenvolvimento do protótipo do DVI, os quais causaram dificuldades na implementação do projeto.

Tais detalhes técnicos referem-se ao manuseio com os sinais assíncronos e o

sinal de *clock*, além da largura do barramento de dados e endereços suportados pela interface PCI da placa.

De forma geral, este capítulo constitui uma referência técnica para as soluções adotadas no desenvolvimento do estudo de caso desta dissertação, que justificam os resultados obtidos durante os experimentos realizados.

Capítulo 7

Análise de Resultados

Este capítulo apresenta uma síntese da análise de desempenho da arquitetura proposta, submetida ao estudo de caso do Detector de Veículos por Imagens (DVI).

A abordagem usando técnicas de Hardware/Software codesign considerada neste trabalho teve como objetivo principal a viabilização do funcionamento do sistema em tempo real e com custo de implementação limitado à escrita dos códigos de software e descrição de hardware, além da expansão do PC com a placa de prototipação.

Considerando que o protótipo desenvolvido poderá ser futuramente aplicado a um sistema de trânsito urbano, a confiabilidade dos resultados e a tolerância permitida em relação à taxa de acertos na detecção de veículos, inserem no projeto um conjunto de restrições, as quais deverão ser atendidas em futuros trabalhos.

7.1 Taxa de captura das imagens

A taxa de captura de frames no monitoramento do tráfego influi diretamente na velocidade máxima dos veículos que poderão ser detectados pelo sistema. Além das características da câmera utilizada para a captura das imagens, a sua instalação e o trecho escolhido como região de interesse da cena contribuem na limitação da funcionalidade do algoritmo. A Figura 7.1 mostra um exemplo da influência entre algumas variáveis envolvidas no desempenho do DVI.

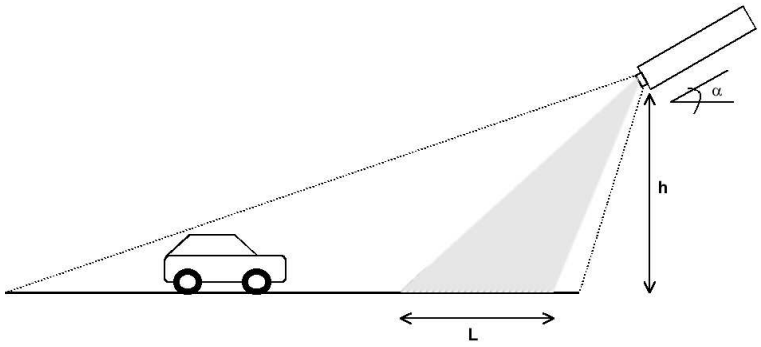

Figura 7.1: Campo visual em relação ao posicionamento da câmera

A altura (h) e o ângulo (α) de posicionamento da câmera determinam o campo visual capturado e proporcionalmente a dimensão (L) da área de interesse. A posição da câmera utilizada para a coleta de dados para o estudo de caso desta dissertação estabeleceu a dimensão (L) em aproximadamente cinco metros.

Com o valor aproximado de L, e com a taxa de captura suportado pelo sistema, é possível calcular uma estimativa da velocidade máxima de veículos detectados.

Neste trabalho, devido à baixa capacidade de processamento do sistema de monitoramento utilizado inicialmente para a captura das imagens de trânsito (computador PC 486 32MB RAM), a amostra obtida foi de aproximadamente 2 quadros por segundo (ver Secção 4.2). Desta forma, existe um intervalo de tempo (Δt) igual a $0,500$ segundos entre cada frame e por isso, um veículo movendo-se a uma velocidade suficiente para percorrer os 5 metros dentro deste intervalo não será percebido pelo DVI. Esta velocidade pode ser estimada por:

$$V_{max} = \frac{L}{\Delta t} \tag{7.1}$$

onde, a V_{max} é a velocidade máxima de veículos detectados em metros por segundo.

Pela equação 7.1, a velocidade máxima detectada pelo protótipo desenvolvido é de aproximadamente 10 m/s, ou 36 km/h. Para uma rodovia de trânsito rápido, este valor limite está abaixo do esperado, já que na realidade é comum a ocorrência de veículos circulando a velocidades superiores a 60 km/h.

Afastando-se a câmera em relação ao solo é possível elevar esta velocidade limite, por conseqüente ampliação do campo de abrangência da área de interesse. No entanto, a perda de qualidade de imagem associada a esta mudança pode prejudicar o funcionamento do algoritmo detector devido a resolução limitada do equipamento de captura.

Uma outra alternativa é a redução do intervalo Δt através do aumento da taxa de captura de imagens. Sistemas atuais de monitoramento estão ajustados para amostragens de 15 fps, equivalente a intervalos de $66,7$ ms entre cenas, o que nos permite detectar veículos a 270 km/h conforme o cálculo mostrado abaixo.

$$V_{max} = \frac{5}{66,7 \times 10^{-3}}$$
$$V_{max} = 75 \text{ m/s } = 270 \text{ km/h}$$

Neste caso, a quantidade de informação na entrada do sistema digital é maior, devendo este estar apto a processar tal volume de dados no intervalo de tempo restrito.

7.2 Qualidade dos Resultados

A heterogeneidade da arquitetura proposta neste trabalho visa a exploração do aumento de desempenho promovido pela utilização de harware específico no processamento digital de imagens. Embora as imagens para teste tenham sido capturadas por um PC de baixa velocidade, o processamento visando atingir as restrições temporais da velocidade máxima de 270 km/h foram realizados em um sistema composto por um computador PC com processador `K6-II` 500 MHz, 128 MBytes de memória RAM e uma plataforma de prototipação Cesys `XC2S_Eval` baseada em um FPGA Xilinx `XC2S200` (200k).

O tempo necessário para realizar o processamento de cada frame através da implementação em software foi avaliado por meio de funções *clock()* e *QueryPerformanceCounter()* (captura o estado atual do relógio do sistema) inseridas no código do programa, informando o usuário sobre o tempo decorrido para a aber-

tura das imagens bitmap (*Open Time*) e o tempo gasto no processamento propriamente dito (conversão, filtragem, comparação, etc.), denominado *Analysis*. A soma total do período é informado em *Run time* (ver Figura 5.3).

Uma amostra de 44 frames foi submetida ao processamento pelo sistema implementado em software, e os resultados temporais obtidos variaram na faixa entre $67,81$ e $72,49$ms, apresentando um valor médio de $70,47$ms por frame.

Esta amostra constitui-se em uma gravação de vídeo capturado no período diurno (17h00) com boas condições meteorológicas, exibindo veículos de pequeno, médio e grande porte.

A mesma amostra, submetida à execução do mesmo trecho de código (listagem abaixo), no sistema implementado em *hardware/software* apresentou melhores resultados temporais, com valores entre $12,30$ e $12,51$ ms, média de $12,37$ ms.

O primeiro ramo (linhas de código 03 a 10) da estrutura condicional (IF) implementa o algoritmo de processamento de imagens em software, realizando através da função `Analise()` toda a conversão de cores, a filtragem por convolução e a quantificação de mudanças na imagem, retornando o valor do grau de invasão associado.

```
01 ok=QueryPerformanceCounter(&inicio); //Instante inicial
02 if(pDoc->firSwHw==0)                 //opcao por software
03 {
04     area=Analise();                    //RGB/Y e Convolucao
05     inv_old=inv_act;
06     inv_act=inv_new;
07     inv_new=area;
08     if((inv_act > inv_old)&&(inv_act >= inv_new))
09       contagem++;
10 }
```

O segundo ramo (linhas 12 a 16) corresponde apenas a implementação da conversão de cores e o envio da imagem de frame à placa. A partir da transferência, todas as funções de processamento das imagens são realizadas pelo componente de hardware. Na Secção 7.4, será demonstrado que este período é seguido de um intervalo de tempo adicional, devido a execução do algoritmo internamente no

FPGA.

```
11 else                                    //opcao por hardware
12 {
13     RgbToY();
14     CopyMemory(pInfo->Buffer, &frame[0][0], BUF_LENGTH);
15     Device.WriteBlock((PPTA_READ_WRITE_BLOCK_INFO)Buffer);
16 }
17 ok=QueryPerformanceCounter(&fim);  //Instante final
```

A compararação direta do tempo gasto em cada elemento de processamento não representa uma idéia clara do ganho proporcionado pelo hardware adicional ao sistema, uma vez que após o envio dos dados de imagens do PC para a placa de prototipação, o processador continua executando tarefas de controle tais como atualização da interface visual do programa concorrentemente com o processamento em execução no FPGA (Figura 7.2).

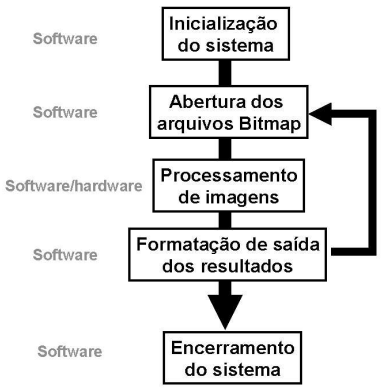

Figura 7.2: Fluxograma simplificado do sistema

Estes resultados estão intimamente relacionados com o desempenho do sistema operacional do ambiente, e este fator pode promover resultados numéricos variantes a cada experimento. Congestionamentos no barramento de comunicação utilizado, configurações na prioridade de interrupção além das tarefas sendo executadas concorrentemente podem corromper alguns dos resultados esperados do sistema variando em $\pm 3\%$ a precisão do tempo gasto no processamento de cada frame.

7.3 Limitações da Plataforma

Embora a placa de prototipação seja uma plataforma PCI, a mesma foi configurada pelo fabricante para suportar transferências de dados de apenas 8-bits. Esta restrição no entanto não prejudicou a performance do sistema.

Por outro lado, por se tratar de uma arquitetura de software para propósito geral, o desempenho do algoritmo em execução no PC apresenta variações no tempo de processamento, uma vez que o sistema compartilha os recursos computacionais com tarefas concorrentes. Como não se trata de um sistema operacional de tempo real, o seu comportamento temporal não-determinístico deve ser contornado e o mapeamento em hardware de processos críticos pode ser uma boa alternativa. A utilização do hardware específico para a aplicação promove alta velocidade de processamento, paralelismo e estimativa do tempo necessário para a execução de processos.

7.4 Cálculo do Processamento em Hardware

O tempo teórico de processamento em hardware (T_{hw}) pode ser calculado como a soma do tempo necessário para o envio dos dados de imagens (t_{send}) e tempo gasto no processamento na placa de prototipação (t_{board}).

$$T_{Hw} = t_{send} + t_{board}$$

Os dados enviados totalizam 32.786 bytes referente aos 128×256 *pixels* de cada frame capturado. A comunicação pelo barramento é realizada através de 128 escritas sucessivas de blocos de 256 bytes a um *clock* de 33MHz, alcançando 3Mbytes/s.

$$\begin{aligned} t_{send} &= \frac{32.768}{3 \times 10^6} \\ t_{send} &= 10,92267 \times 10^{-3} \\ t_{send} &\cong 10,92ms \end{aligned}$$

Enquanto os dados são enviados ao hardware, concorrentemente o FPGA realiza a escrita na memória.

O processamento que ocorre em seguida na placa, corresponde à coleta e análise de dados da memória até a geração da interrupção pelo FPGA para a partição de software.

Para realizar a filtragem por convolução, o hardware coleta 9 elementos para cada pixel da imagem. Cada elemento coletado leva 4 ciclos de relógio, para ajustar o ponteiro de endereço, aguardar a latência da memória e escrever no registrador de soma. Desta forma a execução do somatório (equação 5.5) corresponde a 36 ciclos.

Em seguida, o pixel de referência é carregado e multiplicado por 9, levando 5 ciclos para completar esta operação (ver Secção 5.4.1.5).

A comparação entre o somatório de pixels vizinhos e o elemento de referência seguido da limiarização do resultado demanda mais 2 ciclos de *clock* e finalmente, com mais 2 ciclos, os ponteiros de memória são atualizados para o próximo elemento, verificando se toda a imagem já foi processada.

Deve-se lembrar de acordo com a Secção 5.1.4.1, foi adotada uma condição de contorno que restringe o processamento apenas aos pixels que não fazem parte da borda da imagem. Então para as imagens utilizadas neste trabalho (32.768 *pixels*) apenas ao núcleo de 126×254 *pixels* será aplicado o processamento completo.

Assim, para cada um dos 32.004 *pixels* do núcleo da imagem são necessários $36 + 5 + 2 + 2 = 45$ ciclos, de modo que o frame completo será processado em $45 \times 32.004 = 1.440.180$ ciclos.

Os processos subseqüentes, correspondentes à análise histórica de grau de invasão dos últimos três frames, e geração da interrupção demandam mais 2 ciclos da máquina de estados finitos (FSM).

Como a placa de prototipação trabalha com um *clock* de 40MHz, o tempo de processamento local de um frame pode ser avaliado em:

$$t_{board} = \frac{1.440.182}{40 \times 10^6}$$
$$t_{board} = 36,00455 \times 10^3$$
$$t_{board} \cong 36,00ms$$

Logo, o tempo total para o processamento do algoritmo pela partição de

hardware é estimado em:

$$T_{Hw} = 10,92 + 36,00$$
$$T_{Hw} = 46,92ms$$

Com este resultado, e de acordo com o exposto na Secção 7.1, a implementação em hardware possibilita o processamento de imagens capturadas à taxa de 15 frames/s, garantindo as restrições impostas pela aplicação.

7.5 Conclusão

As restrições temporais impostas ao problema, devido à taxa de captura das imagens foram alcançadas por meio da abordagem proposta por este trabalho.

A comprovação matemática do tempo de execução em hardware, incluindo o custo de comunicação com o software viabilizou a implementação do sistema heterogêneo, validando a arquitetura proposta para aplicações em processamento de imagens.

A medição da performance do sistema pode ser realizada de diversas maneiras, uma vez que a possibilidade de inserção de medidores de tempo (ciclos de relógio) em qualquer trecho do código de software permite a validação do período de tempo estimado para o processamento.

As limitações do protótipo são justificadas pelo caráter experimental da implementação, atendendo a todos os resultados esperados na execução do sistema. Estes resultados estimulam a reutilização da plataforma em novas aplicações, sugeridas como futuros trabalhos.

Capítulo 8

Conclusões e Trabalhos Futuros

8.1 Conclusões

Neste trabalho foi apresentada uma plataforma DSP para processamento de imagens em uma aplicação para detecção e contagem de veículos em uma via pública. Esta aplicação, exposta aqui como estudo de caso surgiu de um problema real, carente de uma solução moderna e de baixo custo, considerando a plataforma computacional existente baseada em PCs.

O projeto baseado em uma arquitetura hardware/software compreende etapas que foram desde o desenvolvimento e especificação do algoritmo de processamento de imagens e suas restrições, em uma linguagem de alto nível de abstração, até a sua implementação em uma arquitetura alvo.

O desenvolvimento do algoritmo de detecção de veículos é precedido pelo estudo dos fundamentos em processamento de imagens digitais, suas aplicações e principais técnicas de realce, assim como o uso de ferramentas de software para o processamento digital de sinais. Ferramentas profissionais como o IDL permitiram a realização de experimentos e análises preliminares com dados reais, através da implementação de rotinas e utilização de funções disponíveis, oferecendo ao projetista um ambiente integrado (gráficos, listagens, imagens, etc.) para a análise da funcionalidade almejada e suporte para a definição da estratégia e das características do algoritmo de processamento de imagens a ser implementado.

A partir da especificação do algoritmo em uma linguagem de alto nível IDL utilizada no desenvolvimento, partiu-se para sua implementação em uma lin-

guagem compilada orientada a objetos, C++, possibilitando futuras integrações com outros sistemas complementares. Esta etapa exigiu o conhecimento detalhado das funções utilizadas e permitiu que fossem feitas otimizações de código pelo projetista.

Devido a baixa performance do atual sistema e das restrições impostas pelo problema, uma solução baseada no paradigma de hardware/software codesign foi proposta. Baseado em um sistema computacional convencional, um PC, que funciona como um componente de software e a plataforma reconfigurável, utilizando um FPGA com interface PCI, foi proposto um particionamento manual de todas as funções inerentes ao problema. Estas funções foram cuidadosamente analizadas levando em conta características tais como tempo de execução, paralelismo e comunicação.

Todo processo foi implementado em software inicialmente, para a análise dos "gargalos" existentes e de suas limitações no processamento de funções críticas que impedem atingir a performance exigida pelo problema.

Assim, uma vez definida os módulos a serem implementados em ambas as partições, e suas respectivas implementações, constatou-se uma melhora no desempenho e a satisfação das restrições temporais do sistema digital. O particionamento foi cuidadosamente realizado, visando a minimização da comunicação entre as partições, agrupando os módulos do algoritmo de acordo com a taxa de transferência de dados entre os módulos numa mesma partição.

A utilização de uma placa de prototipação PCI como hardware adicional ao computador constitui a terceira e última etapa do projeto dentro do escopo proposto. A placa, que possui como unidade de processamento dispositivos reconfiguráveis baseados em FPGA oferece uma solução de alta tecnologia e permite dentro de seu ambiente de projeto, especificação em VHDL e utilização de ferramentas comerciais para síntese e verificação em uma metodologia *top-down*.

Neste projeto duas abordagens foram estudadas para o componente de hardware. A primeira plataforma cogitada para a implementação da partição de hardware [49], também equipada com FPGA, memória RAM e interface PCI, oferecia uma granularidade muito fina de lógica reconfigurável, exigindo um projeto estrutural de baixo nível de abstração, dificultando o mapeamento de funções DSP e a integração com a partição de software. Além disso, as ferramentas de

síntese não permitiam um fluxo de projeto *top-down*, e exigiam grande interação do projetista para que fosse possível o mapeamento lógico.

Esta dificuldade do desenvolvimento em hardware foi solucionada pela aquisição de uma nova plataforma, equipada com FPGA compatível com ferramentas modernas de síntese. Embora, esta placa possua um "gargalo" na sua interface com o PC, pois apesar de utilizar o barramento PCI de 33MHz 32-bits, o *chip* (ASIC) que implementa a interface PCI suporta apenas uma comunicação de 8-bits essa performance foi suficientemente adequada para atender a demanda de dados da aplicação.

A proposta PCI, utilizando um FPGA Spartan-II com 200.000 *gates* conseguiu alcançar as restrições de tempo impostas pelo projeto. A síntese da especificação VHDL de todos os módulos de hardware foi feita na ferramenta Foundation 3.1i da Xilinx, comprometendo apenas $2,54\%$ da capacidade do FPGA.

Nesta abordagem todas as restrições de projeto foram alcançadas com um baixo custo adicional, uma vez que a placa pode ser facilmente acoplada a um microcomputador convencional já existente.

Foi possível implementar toda a funcionalidade do projeto com a detecção da presença de veículos até uma velocidade de 270 km/h, a uma taxa de amostragem de 15 frames/s, bem como oferecer a contagem dos mesmos em períodos determinados.

Uma interface amigável foi desenvolvida em C++, o que facilita ao usuário do sistema a manipulação de parâmetros como a escolha da janela de observação, ajuste da sensibilidade de luminância e grau de invasão. Itens que podem otimizar on-line a resolução e a detecção do veículo. Todo este processo de ajustes em software é facilmente implementado no sistema, dado que as plataformas de processamento são facilmente e rapidamente reconfiguráveis.

Visando uma plataforma com maiores recursos em aplicações que exigem maior performance e *throughput* na interface hardware/software, certamente uma interface PCI convencional com 32-bits de dados permitiria uma vazão bem maior que a atual.

Para o estudo de caso descrito, uma organização formada por 4 (quatro) bancos de memória RAM poderia permitir a execução da convolução do filtro aplicado em apenas um ciclo de relógio para cada *pixel*, promovendo um

processamento 10 vezes mais rápido que a implementação atual. Neste caso, 3 (três) *pixels* vizinhos além do *pixel* de referência poderiam ser coletados simultaneamente, adotando otmizações na convolução, permitidas por esta nova arquitetura, onde os outros 6 (seis) pixels envolvidos na operação por um *kernel* 3×3 já estariam armazenados em registradores [48].

A placa de captura, utilizada para digitalizar o sinal da câmera de vídeo envia os dados da imagem à memória de vídeo do computador. A implementação de um sistema `DMA` (*Direct Memory Access*) possibilitando o acesso direto do componente de hardware aos dados das imagens disponíveis na memória eliminaria a influência do componente de software que tem serializado esta operação pela utilização de arquivos.

Metodologias de desenvolvimento de sistemas hardware/software codesign para aplicações DSP deverão ser estudadas, aproveitando o potencial e caráter prático do estudo de caso apresentado. A grande quantidade de dados, operações algébricas que podem ser efetuadas paralelamente além de restrições de tempo real são características muito comuns em aplicações DSP, e precisam ser adequadamente tratadas. Para tanto, a especificação formal do algoritmo em uma linguagem compatível com ferramentas automáticas de codesign poderão explorar as alternativas de implementação em hardware ou em software visando alto desempenho, menor custo e redução do tempo de projeto.

Linguagens de especificação de sistemas como SystemC [5] encontrarão neste estudo de caso um bom exercício de implementação, onde comparações entre os resultados manuais e automáticos obtidos poderão ser confrontados para a validação de tais metodologias.

Apêndice A

Anexos

Neste capítulo estão apresentados os códigos-fonte utilizados no desenvolvimento do estudo de caso desta dissertação.

A.1 Códigos em IDL (*Interactive Data Language*)

A.1.1 Função de ajuste de contraste

Função utilizada para clarear a porção da imagem externa à área de interesse. Recebe como argumentos a imagem a ser manipulada, o menor valor (a) do subconjunto $K_1 = [a, 255]$ de tonalidades de cinza da nova paleta atribuída a área a ser clareada (figura 7) e uma máscara que definirá a área de interesse para ser preservada. Retorna a imagem com brilho alterado.

```
FUNCTION CONSTRETCH, imagem, a, mask
  saida=imagem
  FOR x=0, 639 DO BEGIN
    FOR y=0, 479 DO BEGIN
      IF(mask(x,y) EQ 0) THEN BEGIN
        saida(x,y)=a+imagem(x,y)*(255-a)/255
      ENDIF
    ENDFOR
```

```
  ENDFOR
  RETURN, saida
END
```

A.1.2 Função para conversão RGB para Y

Função utilizada para converter os componentes de cor RGB da imagem em informação de luminância (tons de cinza). Recebe como argumento a imagem colorida a ser convertida (array 3 dimensões) e retorna a imagem monocromática (array de 2 dimensões).

```
FUNCTION rgb2y, imargb
  imay = REFORM(BYTE(0.299 * imargb(0,*,*) + 0.587 * imargb(1,*,*) $
       + 0.114 * imargb(2,*,*)))
  RETURN, imay
END
```

A.1.3 Função de descarte de frames

Função utilizada para descartar os frames redundantes, resultado da baixa taxa de captura das imagens. Recebe como argumento o valor do limiar o qual julga a mudança de cena frame a frame. Chama internamente a função de monocromatização das imagens (`rgb2y`) e retorna um de array de 3 dimensões

$$[imagem\ (largura \times altura) \times frame(tempo)]$$

```
FUNCTION separabmp, limiar
  raiz = 'FrameNo'
  destino = BYTARR(640, 480, 1)
  destino(*,*,0) = rgb2y(READ_BMP('FrameNo1.bmp'))
  FOR ima = 1, 51 DO BEGIN
    nome = STRCOMPRESS(raiz + STRING(ima) + '.bmp', /REMOVE_ALL)
    temp1 = rgb2y(READ_BMP(nome))
    temp2 = SIZE(destino)
    ultima_imagem = temp2(3)
```

```
    IF (TOTAL(ABS(destino(*,*,ultima_imagem - 1) - temp1)) $
        GE limiar) THEN BEGIN
      destino = [[[destino]], [[temp1]]]
      PRINT, 'Armazenando imagem ' + STRING(ultima_imagem)
    ENDIF
  ENDFOR
  RETURN, destino
END
```

A.1.4 Visualização da área selecionada

Procedimento auxiliar, utilizado para visualizar a mascara que delimita a area de interesse. Recebe como argumento a seqüência de frames (saída da função separabmp) e a máscara de delimitação.

```
PRO exibe, i, m
  dim=SIZE(i)
  FOR c=0, dim(3)-1 DO BEGIN
    tvscl, i(*,*,c)+10*m
  ENDFOR
END
```

A.1.5 Máscara de seleção na imagem

Função utilizada para a construção da máscara que delimita a área de interesse. Recebe como parâmetros os limites da esquerda, direita, bem como os limites inferior e superior da janela retangular. Retorna um array de tamanho 640×480 com valores iguais a 1 na região de interesse das imagens e 0 na área restante.

```
FUNCTION mascara, esquerda, direita, inferior, superior
  mascara=REPLICATE(0, 640, 480)
  FOR m=esquerda, direita-1 DO BEGIN
    FOR l=inferior, superior-1 DO BEGIN
      mascara [m,l]=1
    ENDFOR
```

```
  ENDFOR
  tvscl, mascara
  RETURN, mascara
END
```

A.1.6 Detecção de veículos por imagens

*Função que realiza a detecção da passagem do objeto invasor pela área selecionada de interesse. Recebe como argumentos a seqüência de frames, o kernel de convolução, o valor do limiar a ser considerado na subtração das imagens, a máscara que determina a área selecionada e a sensibilidade quanto a porção em área invadida que deve ser considerada.

```
FUNCTION sobre_faixa, imagens, k, limiar, mask, sens
  invasao_old = 0
  invasao_act =0
  invasao_new = 0
  contagem = 0
  dim = SIZE(imagens)
  referencia=CONVOL(FLOAT(imagens(*,*,0)),k)
  FOR i = 1, dim(3) - 2 DO BEGIN
    diferenca= mask AND ( CONVOL ( FLOAT (imagens (*,*,i) ),k)$
             -CONVOL ( FLOAT ( imagens(*,*,0) ) , k) ge limiar)
    invasao_old=invasao_act
    invasao_act=invasao_new
    invasao_new= TOTAL(diferenca AND mask)
    TVSCL,(imagens(*,*,i) +10*mask)
    IF((invasao_act GT invasao_old) AND (invasao_act GT invasao_new)$
           AND (invasao_act GT sens)) THEN BEGIN
      contagem=contagem+1
      print, contagem
    ENDIF
  ENDFOR
  RETURN, STRCOMPRESS('Total: ' + STRING(contagem) + ' veiculos', $
```

```
          /REMOVE_ALL)
END
```

A.2 Códigos implementados em C++

Os principais trechos de código header (*.h*) e fonte (*cpp*) das classes *CDvicppView* e *CDvicppDoc* do aplicativo implementado em software são listados abaixo.

A.2.1 DvicppDoc.h

Header da classe que implemanta a manipulação (abertura e salvamento) de arquivos em disco.

```
class CDvicppDoc : public CDocument
{
  public:
    unsigned char *bits, *c;
    unsigned char copy[480][640][3];
    unsigned char ref[480][640][3];
    BITMAPFILEHEADER bmfh;
    BITMAPINFO bmi;
    CDvicppDoc();
    // Operations
  public:
    int firSwHw;
    bool reference;
    virtual ~CDvicppDoc();
    #ifdef _DEBUG
      virtual void AssertValid() const;
      virtual void Dump(CDumpContext& dc) const;
    #endif
};
```

A.2.2 DvicppDoc.cpp

Código fonte da classe que implementa o acesso a documentos (imagens bitmap) armazenadas em disco.

```
// dvicppDoc.cpp : implementation of the CDvicppDoc class
CDvicppDoc::CDvicppDoc()
{
  reference=TRUE;
  bits=(unsigned char *)malloc(
    (long int)(640*3*480)*sizeof(unsigned char)
  );
}
void CDvicppDoc::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  {
    //Saving the file
    // This code is for 24 Bit Bitmaps only.
  }
  else
  {//Opening the file
    int i,j,k;
    ar>>bmfh.bfType; \   \  //bmfh:BitMap File Header
    ar>>bmfh.bfSize ;
    ar>>bmfh.bfReserved1;
    ar>>bmfh.bfReserved2;
    ar>>bmfh.bfOffBits; \   \
    ar>>bmi.bmiHeader.biSize; \  //bmi: Bit Map Info
    ar>>bmi.bmiHeader.biWidth;
    ar>>bmi.bmiHeader.biHeight;
    ar>>bmi.bmiHeader.biPlanes;
    ar>>bmi.bmiHeader.biBitCount;
    ar>>bmi.bmiHeader.biCompression;
```

```
ar>>bmi.bmiHeader.biSizeImage;
ar>>bmi.bmiHeader.biXPelsPerMeter;
ar>>bmi.bmiHeader.biYPelsPerMeter;
ar>>bmi.bmiHeader.biClrUsed;
ar>>bmi.bmiHeader.biClrImportant;
if (bmi.bmiHeader.biBitCount==24)
{
  // 24 bit bitmap
  long int cnt=0;
  for(i=bmi.bmiHeader.biHeight-1; i>=0; i--)
  {
    for(j=0; j<bmi.bmiHeader.biWidth; j++)
    {
      for (k=0; k<3;k++)
      {
        ar>>bits[cnt];
        if(reference)
          ref[i][j][k]=bits[cnt];
        else
          //Note the inverse referencials i,j
          copy[i][j][k]=bits[cnt];
        cnt++;
      }
    }
    for (j=0;j<bmi.bmiHeader.biHeight%4;j++)
    {
      ar>>bits[cnt];
      cnt++;
    }
  }
  reference=TRUE;
}
else
```

```
    {
      MessageBox(NULL,''Sorry, this is not a 24 Bit Bitmap.'',
                 ''File Open Error'',MB_ICONSTOP|MB_OK);
    }
  }
}
```

A.2.3 DvicppView.h

Header da classe que implementa as funcionalidades de visualização e controle do algoritmo de processamento de imagens.

```
//Interrupt treatment class
class Countcar:public NotifyEvent{
public:
  CDvicppView *init_;
  int cont_hw;
  bool inbit; //indicates the interrupt's arrived
  Countcar(CDvicppView *init);
  ~Countcar(void);
  virtual void NotifyFunc(unsigned long IntContext);
  void ResetInbit(void);
  bool GetInbit(void);
};

//Image processing class
class CDvicppView : public CView
{
protected: // create from serialization only
  friend class Countcar;
  Countcar func;
  DECLARE_DYNCREATE(CDvicppView)
  CDvicppView();
public:
```

```
 \  CDvicppDoc* GetDocument();
 \  RGBQUAD *bmicolor;
 \  BITMAPINFOHEADER bmih;
 \  BITMAPINFO bmi;
// Implementation
public:
  double an_time;
  clock_t timeInt, timeAn;
  LARGE_INTEGER inicio, fim;
  int massa;
  int limiar;
  int contagem;
  unsigned char ConvSoft( int x, int y);
  void CreateTrackbar();
  BOOL stop;
  void CreateProgressBar();
  long int area;
  void RgbToY();
  virtual ~CDvicppView();
protected:
  XC2S Device;
  unsigned char refer[128][256];
  unsigned char frame[128][256];
  long int Analise();
  CFont m_font;
  BOOL m_bIsTracking;
  CPoint m_pointLast;
  CPoint m_pointOrigin;
  CProgressCtrl m_progressBar;
  CSliderCtrl m_trackbar1;
  CSliderCtrl m_trackbar2;
};
```

A.2.4 DvicppView.cpp

Código fonte da classe que implementa a visualização das imagens e o processamento do algoritmo de detecção de veículos. Esta classe também implementa a interface em software com a partição de hardware.

```
#include ''stdafx.h''
#include ''dvicpp.h''
#include ''dvicppDoc.h''
#include ''dvicppView.h''
#include ''MainFrm.h''
#include <winioctl.h>
#include <time.h>
#include ''PTAif.h''
#include ''util.h''
#include ''afxmt.h''
#include ''XC2S.h''

#define XEXTENTS_LOGICALUNITS 1400
#define YEXTENTS_LOGICALUNITS 1400
#define XORIGIN_LOGICALUNITS 200
#define YORIGIN_LOGICALUNITS 400
#define XMAXCLIP_LOGICALUNITS 2000
#define YMAXCLIP_LOGICALUNITS 800
#define BUF_LENGTH 32768

Countcar::Countcar(CDvicppView *init): init_(init)
{
  cont_hw = 0;
  inbit = FALSE;
}
void Countcar::NotifyFunc(unsigned long IntContext){
  if(IntContext != 0){
    cont_hw++;
```

```
  }
  inbit = TRUE;
  return;
}
void CDvicppView::OnRunProcessing()
{
  //Getting a handle to the document (bitmap that has been openned)
  CDvicppDoc* pDoc = GetDocument();
  CDC *pDC = GetDC();
  stop=0;
  contagem=0; //vehices counter variable
  //Filtering and converting the reference image
  for(int a=m_pointOrigin.x; a<=m_pointLast.x;a++){
    for(int b=m_pointOrigin.y; b<=m_pointLast.y;b++){
      int element=0;
      for(int l=-1; l<2; l++){
        for(int m=-1; m<2; m++){
          element+=(int)((0.299*pDoc->ref[b+l][a+m][1])
          +(0.587*pDoc->ref[b+l][a+m][2])
          +(0.114*pDoc->ref[b+l][a+m][0]));
        }
      }
      //refer[*][*][0] monocromatic / filtered reference frame
      refer[b-m_pointOrigin.y][a-m_pointOrigin.x]=
        (unsigned char)(element/9);
    }
  }
  //setting the bitstream to be downloaded
  char ExoBitFile[256];
  char gCurrDeviceName[100];
  sprintf(gCurrDeviceName, ''\\\\.\\%s%d'', PTA_DEVICE_NAME, 0);
  //Bitstream file design to be dowloaded
  wsprintf(ExoBitFile,
```

```
''C:\\Xilinx\\active\\projects\\dvihw\\xproj
    \\ver1\\rev1\\dvihw.exo'');
BOOL ok = Device.OpenWithName(gCurrDeviceName);
//downloads the design now...
if (pDoc->firSwHw==1){ //hardware
  //Downloads the bitstream to the FPGA (.exo file)
  if(ConvertExoAndDownload(ExoBitFile,&Device))
  {
    Sleep(10);
    //get FPGA status and check if it is configured correctly
    int Status = Device.IsFpgaConfigured();
  }
  Device.IsDeviceInterruptEnabled(&ok);
  if(ok){ //disables any interrupt wait
    Device.StopInterruptCapture();
  }
  ok = Device.StartInterruptCapture(&func); \   \
  ok = Device.Open();
}
//Capture the user's parameters
limiar=m_trackbar2.GetPos();
unsigned char limiar_char = (unsigned char)limiar;
massa=m_trackbar1.GetPos();
unsigned char massa_32=(unsigned char)(massa/32);
//send this parameters to the hardware
ok=Device.WriteByte(2, SPACE_PARALLEL_3, limiar_char);
//setting the block of bytes which will be sent by PCI
ULONG Size = BUF_LENGTH + sizeof(PTA_READ_WRITE_BLOCK_INFO) -1;
unsigned char *Buffer = (unsigned char *) malloc(Size); \
PPTA_READ_WRITE_BLOCK_INFO pInfo=(PPTA_READ_WRITE_BLOCK_INFO)Buffer;
pInfo->Length = BUF_LENGTH;
pInfo->Offset = 0;
pInfo->IncrementAddress = 0; \
```

```
pInfo->Location = SPACE_PARALLEL_3;
//Reference image block assembling
CopyMemory(pInfo->Buffer, &refer[0][0], BUF_LENGTH);
ok=Device.WriteBlock((PPTA_READ_WRITE_BLOCK_INFO)Buffer);
//set the paralell address to send frames
pInfo->Offset = 1;
//Processing each frame, opening, filtering and comparing
CFile cena;
char local[32];
//Variables to hold the mensured execution time
double open_time, run_time;
//software analisys variables
long int inv_old = 0;
long int inv_act = 0;
long int inv_new = 0;
//Log file
FILE *file_pointer;
file_pointer=fopen(''logtime.txt'', ''w'');
ok=Device.WriteByte(3, SPACE_PARALLEL_3, massa_32);
//Processes each bitmap
for(int x=1; x<45/*&&(!stop)*/; ++x)
{
  //Elapsed time control
  clock_t timeStart, timeEnd;
  //Capture the initial time
  timeStart=clock();
  //open the bitmap file
  wsprintf(local,''c:\\transito\\cap%03d.bmp'', x);
  cena.Open(local, CFile::modeRead );
  CArchive cenar( &cena, CArchive::load, 925696, NULL );
  pDoc->reference=FALSE;//openning frames
  pDoc->Serialize(cenar);
  cena.Close();
```

```
//Capture the main window's program to use the status bar
CMainFrame *mainWnd =(CMainFrame*) AfxGetMainWnd();
CStatusBar *statusBar = &(mainWnd->m_wndStatusBar);
CStatusBarCtrl & sbCtrl = statusBar->GetStatusBarCtrl( );
//Refresh the window
RedrawWindow(NULL,NULL,RDW_INVALIDATE|RDW_UPDATENOW|RDW_ERASE);
Device.Open();
//Catch the opening finished time
timeAn=clock();
ok=QueryPerformanceCounter(&inicio);
//Image processing: RGB to Y converting and convolution filtering
if(pDoc->firSwHw==0){ //by software
  area=Analise();
  //Information analysis
  inv_old=inv_act;
  inv_act=inv_new;
  inv_new=area;
  if((inv_act > inv_old)&&(inv_act >= inv_new)){
    contagem++;
  }
  timeEnd=clock();
}
else{
  RgbToY();
  CopyMemory(pInfo->Buffer, &frame[0][0], BUF_LENGTH);
  ok=Device.WriteBlock((PPTA_READ_WRITE_BLOCK_INFO)Buffer);
}
ok=QueryPerformanceCounter(&fim);
if(pDoc->firSwHw==1){
  contagem = func.cont_hw;
}
//Results presentation
char resultado[30];
```

```
    char invasao[20];//busy area
    ltoa(area, invasao, 10);
    sbCtrl.SetWindowText(invasao);
    char carros[20];
    _ltoa(contagem, carros, 10);
    wsprintf(resultado,''#Cars:  %03d  Area: %d  '',contagem,area);
    sbCtrl.SetWindowText(resultado);
    //time elapsed by frame (image analysis, no opening time)
    an_time = (double)(fim.LowPart - inicio.LowPart)/CLOCKS_PER_SEC;
    //time elapsed by frame (to open each)
    open_time = (double)(timeAn - timeStart) / CLOCKS_PER_SEC;
    //time elapsed by frame (full processing)
    run_time = (double)(timeEnd - timeStart) / CLOCKS_PER_SEC;
  }
  fclose(file_pointer);
  Device.StopInterruptCapture();
  free(Buffer);
  func.cont_hw=0;
  Device.Close();
}

long int CDvicppView::Analise()
{
  CDvicppDoc* pDoc = GetDocument();
  long int total = 0;
  unsigned char imag=0;
  //Converts the image from RGB to Y (luminance) information
  RgbToY();
  //Compare the frame with the reference image at each pixel
  for(int by=1; by<127;by++){
    for(unsigned int ax=1; ax<255;ax++){
      imag=ConvSoft(by, ax);
      //comparing and thresholding the image
```

```
      total+=(imag-refer[by][ax])>limiar;
    }
  }
  if(total<massa)
  total=0;
  return total;
}

void CDvicppView::RgbToY()
{
  CDvicppDoc* pDoc = GetDocument();
  for(int by=m_pointOrigin.y; by<=m_pointLast.y;by++) \  {
    for(int ax=m_pointOrigin.x; ax<=m_pointLast.x;ax++) {
      frame[by-m_pointOrigin.y][ax-m_pointOrigin.x]=
        (unsigned char)((0.299*pDoc->copy[by][ax][1])
        +(0.587*pDoc->copy[by][ax][2])
        +(0.114*pDoc->copy[by][ax][0]));
    }
  }
}
//Convolution filtering by software/////
unsigned char CDvicppView::ConvSoft(int cy, int cx){
  CDvicppDoc* pDoc = GetDocument();
  int element=0;
  unsigned char imag;
  for(int l=-1;l<=1;l++){
    for(int m=-1; m<=1; m++){
      element+=(frame[cy+l][cx+m]/*pDoc->copy[cy+l][cx+m][0]*/);
    }
  }
  imag=(unsigned char)(element/9);
  return imag;
}
```

Referências Bibliográficas

[1] CIE. Available URL: http://www.cie.co.at/cie/home.html.

[2] Combitech traffic systems - PREMID TS3100. Available URL: http://www.trafficsystems.com.

[3] The MathWorks: Developers of MATLAB and simulink for techinical computing. Available at URL: http://www.mathworks.com/.

[4] PATH - partners for advanced transit and highways. http://www-path.eecs.berkeley.edu/.

[5] SystemC community. www.systemc.org.

[6] VCC - virtual computer corporation. URL: http://www.vcc.com.

[7] XtraConverter: AVI-BMP extractor. Available at URL: hhtp://remote-security.co.uk/freebees/xavi2bmp.zip.

[8] *The Programmable Logic Data Book*. Xilinx, San Jose, California, third edition edition, 1994.

[9] Pipelined divider v2.0. November 2000.

[10] LogiCORE PCI32 interface v3.0 interface data sheet, 2002.

[11] P. M. B. Coifman, D. Beymer and J. Malik. A real-time computer vision system for vehicle tracking and traffic surveillance. *Transportation Research: Part C*, 6(4):271–288, 1998.

[12] G. J. F. Banon. Formal introduction to digital image processing. INPE, São José dos Campos, Brazil, July 2000. Available URL: http://iris.sid.inpe.br:1912/ rep/dpi.inpe.br/ banon/1998/07.02.12.54.

[13] E. Barros and A. Sampaio. Towards provably correct Hardware/Software partitioning using occam. *IEEE Computer Society Press*, pages 210–217, 1994.

[14] A. Broggi and S. Bertè. Vision-based road detection in automotive systems: A real-time expectation-driven aproach. *Journal of Artificial Intelligence Reasearch*, 3:325–348, 1995.

[15] S. E. Browne. *Nonlinear Editing Basics: A Primer on Electronic Film and Video Editing*. Focal Press, Boston, Oxford, 1998.

[16] H. E. Burdick. *Digital Imaging: Theory and Applications*. McGraw-Hill, 1997.

[17] P. Chou and G. Borriello. Software architecture synthesis for retargetable real-time embedded systems. *Proc. Codes/CACHE*, pages 101–105, March 1997.

[18] J. G. D. Bullock. Vehicle detection using a hardware-implemented neural net. *IEEE Computer*, 1997.

[19] C. C. de Araújo. InterfPISH - uma ferramenta para geração automática de interfaces em hardware/software co-design. Master's thesis, Universidade Federal de Pernambuco, 2001.

[20] I. Development and P. Interim. IVHS america, 1994.

[21] E. B. et al. *Hardware/Software Co-Design: Projetando Hardware e Software Concorrentemente*. IME-USP, São Paulo, 2000.

[22] D. W. Fanning. *IDL Programming Techniques*. Fanning Software Consulting, Fort Collins, January 1998.

[23] R. Fulton. The xilinx HDL advisor: Using nested if statements. *Xcell Journal*, 30:21 – 23, 1998.

[24] S. V. M. I. B. G. Vanmeerbeeck, P. Schaumont. Hardware/Software partitioning of embedded system in OCAPI-xl. *SIGDA Proceedings Archives CODES 2001*, 2001.

[25] F. Gilbert. *Development of a Design Flow and Implementation of Example Designs for the Xilinx XC6200 FPGA Series*. PhD thesis, Universität Kaiserslautern, mai 1998.

[26] Hitachi. *HM62V8512C Series*. Hitachi, Ltd. Semiconductor and Integrated Circuits, April 2001.

[27] P. Holmberg and A. Mascarin. The MathWorks and xilinx take FPGAs into mainstream DSP. *Xcell Journal*, 37:14–15, Third Quarter 2000.

[28] A. K. Jain. *Fundamentals of Digital Image Processing*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[29] B. T. C. Jung Soh and M. Wag. Analysis of road image sequences for vehicle counting. *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pages 679 – 683, October 1995.

[30] X. Z. . J. Z. Junwen Wu. Vehicle detection in static road images with PCA-and-wavelet-based classifier. *Proceedings of IEEE ITSC*, pages 742–746, 2001.

[31] P. Karmazin. *PCI Interface for Telephony/Data Applications*. Infineon Technologies, 2000. PITA-2.

[32] W. Kasprzak and H. Niemann. Applying a dinamic recognition scheme for vehicle recognition in many object traffic scenes. *Workshop on Machine Vision Applications*, pages 212–215, 1996.

[33] G. Knittel. A reconfigurable processing system for DSP applications. *Proceedings of the IEEE Midwest Symposium on Circuits and Systems*, 1996.

[34] M. Kraus. *XC2SEVAL User Manual*. Cesys GmbH, Germany, 2001.

[35] M. N. Mahmoud Meribout and T. Ogura. Accurate and real-time image processing on a new PC-compatible board. *Real-Time Imaging*, 8:35–51, 2002.

[36] J. Malik and S. Russel. A machine vision based survellance system for california roads. Technical report, University of California.

[37] R. M. Manoel E. de Lima. Uma interface PCI para um comutador ATM. pages 333–344, 1996.

[38] A. S.-V. Marco Sgroi, Luciano Lavagno. Formal models for embedded system design. *IEEE Design & Test of Computers*, 17, 2:14–27, June 2000.

[39] J. P. Marek Gorgon. FPGA based controller for heterogenous image processing system. 2001.

[40] A. F. Massimo Bertozzi, Alberto Broggi. Vision-based inteligent vehicles: State of the art and perspectives. *Robotics and Autonomous Systems*, 32:1–16, 2000.

[41] E. ÖZALP. *XC2SEval Evaluation Board Programmer's Guide*. Cesys, 2001.

[42] I. Page. Compiling video algorithms into hardware. July 1997.

[43] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design - The Hardware / Software Interface*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998.

[44] W. L. P.I. Mackinlay, P.Y.K. Cheung and R. Sandiford. Riley-2: A flexible platform for design and dynamic reconfigurable computing research. *Field-Programmable Logic and Applications*, LNCS 1304:91–100, 1997.

[45] W. L. J. S. Simon D. Haynes, Peter Y. K. Cheung. SONIC - a plug-in architecture for video processing.

[46] A. Stockman and L. T. Sharpe. Color & vision database, January 2000. Available URL: http://www-cvrl.ucsd.edu/database/text/intros/introvl.htm.

[47] Synopsys Inc. *SystemC Reference Manual*.

[48] A. Tarmaster, P. M. Athanas, and A. L. Abbott. Accelerating image filters using a custom computing machine, 1995.

[49] Virtual Computer Corporation, Reseda, CA. *H. O. T. Works: Hardware Object Technology Development System (User's Guide V. 1)*, September 1997. Available URL:http://www.vcc.com.

[50] Xilinx. *Spartan-II 2.5V FPGA Family: Functional Description*. Xilinx, Inc., March 2001.

[51] Xilinx Inc, San Jose, CA. *Foundation Series 3.1i*.

[52] Xilinx Inc. *PROM File Formater Guide*, 4.1i edition.

[53] P. Zarchan and H. Musoff. *Fundamentals of Kalman Filtering: A Practical Approach*. AIAA, 2000.