Universidade de Aveiro
Departamento de
Eletrónica, Telecomunicações e Informática
2013

Sérgio Julião
Silva Alves

Coprocessador Baseado em FPGA para a Visão de um Robô Futebolista

Sérgio Julião Silva Alves

Coprocessador Baseado em FPGA para a Visão de um Robô Futebolista

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Arnaldo Oliveira, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e do Doutor José Nuno Panelas Nunes Lau, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Doutor Tomás António Mendes Oliveira e Silva
Professor Associado da Universidade de Aveiro (por delegação da Reitora da Universidade de Aveiro)

vogais / examiners committee

Doutor José Carlos dos Santos Alves
Professor Associado da Faculdade de Engenharia da Universidade do Porto

Doutor Arnaldo Silva Rodrigues de Oliveira
Professor Auxiliar da Universidade de Aveiro (orientador)

Doutor José Nuno Panelas Nunes Lau
Professor Auxiliar da Universidade de Aveiro (co-orientador)

agradecimentos / acknowledgements

A realização desta dissertação é uma realidade devido, sobretudo, à orientação prestada pelo Doutor Arnaldo Oliveira e Doutor Nuno Lau. A ambos agradeço os sucessivos votos de confiança, assim como a disponibilidade e auxílio constantes. Para mim, foi um privilégio a possibilidade de trabalharmos em conjunto e, mais uma vez, poder usufruir da partilha dos seus conhecimentos.

A minha família também teve um papel preponderante, no apoio e motivação ao longo de todo o meu percurso académico e, por isso, um sentido obrigado!

palavras-chave

Sistemas de visão robótica, coprocessamento, algoritmos de visão, hardware reconfigurável, processamento de imagem, FPGA, VHDL.

resumo

A área da robótica encontra-se em constante evolução, assumindo, nos dias de hoje, uma elevada importância nas mais diversas áreas. Uma grande parte das soluções desenvolvidas neste campo tem como requisito o reconhecimento visual do meio envolvente, possibilitando ao sistema robótico reconhecer determinados objetos ou formas, e/ou tomar decisões e reagir perante determinadas situações. Por outro lado, a visão computacional assume-se como uma ciência de ampla aplicação, o que implica a necessidade de optar pelo sistema de visão mais adequado a cada realidade.

A competição RoboCup surge como uma iniciativa internacional com o objetivo principal de promover a investigação e desenvolvimento da inteligência artificial, robótica e demais áreas relacionadas. Embora existam outras, as principais competições presentes nesta iniciativa envolvem a organização de encontros anuais, onde equipas de robótica de todo o mundo jogam futebol entre si. Estas competições encontram-se agrupadas em determinadas categorias, sendo que a mais relevante no âmbito deste projeto é a liga de robôs médios (RoboCup MSL), onde a equipa CAMBADA representa a Universidade de Aveiro.

Neste tipo de competição, o sistema de visão assume uma importância vital, uma vez que a perceção da bola, das linhas brancas e obstáculos, bem como o seu consequente processamento, de forma rápida e eficiente, são uma mais-valia para a tomada de decisão de um robô, face a uma determinada situação. No entanto, os algoritmos associados ao processamento de imagem são tipicamente muito exigentes, quer em termos de recursos, quer em termos de processamento, podendo o tempo necessário ao seu processamento comprometer uma resposta eficaz e, por conseguinte, todo o sistema.

Esta dissertação resulta da necessidade de melhorar continuamente a prestação desta equipa nas competições em que participa, através do desenvolvimento e implementação de um coprocessador baseado em FPGA, o que se apresenta como uma mais-valia, dado permitir a redução do peso computacional da visão por computador da CAMBADA. Este coprocessador será responsável pela execução de uma parte dos algoritmos de visão, nomeadamente, segmentação de cor, compressão de imagem e deteção de bolhas de cor, baseando-se em diversos núcleos de processamento, quer genéricos, quer especializados, explorando, desta forma, o paralelismo normalmente disponível neste tipo de algoritmos. Este sistema possui ainda capacidade de adquirir de forma autónoma as imagens, processando-as individualmente e disponibilizando os resultados ao computador principal.

O presente projeto pretende demonstrar a aplicabilidade e a supremacia da implementação de tarefas computacionalmente intensivas em *hardware* reconfigurável.

keywords

Robotic vision systems, coprocessing, vision algorithms, hardware reconfigurable, image processing, FPGA, VHDL.

abstract

The field of robotics is evolving, assuming, nowadays, a high importance in several areas. A large part of the solutions developed in this field have as a requirement, the visual recognition of the surrounding environment, allowing the robotic system to recognize certain objects or shapes, and/or make decisions and react to certain situations. Moreover, computer vision is assumed as a science of wide application, which implies the need to choose the most appropriate vision system to each situation.

The RoboCup competition emerges as an international initiative with the main objective to promote research and development of artificial intelligence, robotics and other related areas. Although there are other competitions, major competitions present in this initiative involve the organization of annual meetings, where robotics teams from around the world play soccer between them. These competitions are grouped into certain categories, where the most relevant in the context of this project is the Middle Size League of robots (RoboCup MSL), where CAMBADA team represents the University of Aveiro.

In this type of competition, the vision system is of vital importance, since the perception of the ball, the white lines or obstacles, as well as their subsequent processing quickly and efficiently, are an added value for decision making of a robot to a specific situation. However, the algorithms associated with the image processing are typically very demanding, both in terms of resources, as in terms of processing, which enables the overall processing time can compromise an effective answer and therefore, the whole system.

This work stems from the need to continuously improve the performance of this team in the competitions in which it participates, through the development and implementation of an FPGA-based coprocessor, which is presented as an advantage, since it allows a weight reduction of vision computation in each computer of CAMBADA. This coprocessor is responsible for executing a portion of vision algorithms, namely, color segmentation, image compression and detection of color bubbles, based on various processing cores, either generic or specialized, thus exploiting the parallelism usually available in this type of algorithms. This system also has the ability to autonomously acquire the images, processing them individually and providing the results to the main computer.

This project aims to demonstrate the applicability and supremacy of the implementation of computationally intensive tasks on reconfigurable hardware.

Conteúdo

Lista de Figuras

Lista de Abreviaturas

AOI	Area of interest.
ASIC	Application Specific Integrated Circuits.
CAD	Computer Aided Design.
CAMBADA	Cooperative Autonomous Mobile roBots with Avanced Distributed Architecture.
CAN	Controller Area Network.
DSP	Digital Signal Processors.
FPGA	Field Programmable Gate Array.
IEETA	Instituto de Engenharia Eletrónica e Telemática de Aveiro.
MCL	Monte Carlo Localization.
MSL	Liga de Robôs Médios.
NSVP	Non Single View Point.
RGB	Red, Green, Blue.
ROI	Region of Interest.
ROP	Robotic Open Platform (website: http://www.roboticopenplatform.org/).
RTDB	Real-Time Database.
SoC	System on Chip.
SVM	Support Vector Machine.
SVP	Single View Point.
TURTLE	Tech United RoboCup Team: Limited Edition.
VHDL	ou VHSIC Hardware Description Language (consulte também VHSIC).
VHSIC	Very High Speed Integrated Circuits.
VLSI	Very Large Scale Integration.

Capítulo 1

Introdução

Esta dissertação consiste no estudo, desenvolvimento e implementação de um coprocessador baseado em *hardware* reconfigurável. No sistema a desenvolver, o *hardware* reconfigurável, baseado numa *Field Programmable Gate Array* (FPGA), funcionará como um coprocessador, libertando o computador principal de cada robô das tarefas computacionalmente intensivas e disponibilizando ao computador principal os dados já tratados. Este coprocessador será especialmente dedicado à execução dos algoritmos de visão, procurando explorar o paralelismo normalmente disponível neste tipo de algoritmos.

1.1 Enquadramento

Nos últimos anos, temos assistido a um crescimento exponencial na integração de sistemas robóticos nas mais diversas áreas, como sendo a indústria ou aplicações militares. Este crescimento, entre outros motivos, surge como uma resposta às mais variadas necessidades emergentes de mercado e, sobretudo, como uma constante necessidade de fazer mais, melhor e com um menor custo. Considerando igualmente o enorme esforço e atenção que têm sido empregues na investigação e avanço, quer da tecnologia, quer da inteligência artifical ou demais áreas relacionadas, avizinham-se inevitavelmente tempos em que a realidade será fiel à ficção científica de filmes como "2001, Odisseia no espaço", "Guerra das estrelas" ou outros, onde seja patente a interação de seres humanos com sistemas inteligentes, qualquer que seja a sua forma. De facto, uma parte do esforço de investigação tido nos últimos tempos, visa sobretudo, tornar estes sistemas mais naturais e mais amigáveis do ponto de vista do ser humano, de modo a permitir, tanto a sua fácil inserção no nosso dia-a-dia, como a torná-los sistemas confiáveis e colaborativos nas mais diversas tarefas.

Neste momento é possível encontrar com relativa facilidade, sistemas de limpeza, telepresença e corte de relva como o *iRobot Roomba*[1] (fig. 1.1), os sistemas *iRobot Ava* ou *RP-Vita*[2] (fig. 1.2) e o *Honda Miimo*[3] (fig. 1.3). Embora estes sistemas já possuam um nível de sofisticação elevado, a tecnologia, os métodos e a inteligência tendem a aumentar e melhorar progressivamente, e com eles, necessariamente, a diversidade e o número de sensores incluídos. Neste sentido e analogamente ao ser humano, os sensores poderão ser vistos como os sentidos do robô, enquanto que a inteligência simboliza o seu conhecimento e a sua capacidade de

[1] `http://www.irobot.com/en/us/learn/home/roomba.aspx`, acedido em: 08/07/2013
[2] `http://www.irobot.com/us/learn/commercial/`, acedido em: 08/07/2013
[3] `http://world.honda.com/news/2012/p120820Honda-Miimo/`, acedido em: 08/07/2013

aprender e reagir perante novas situações (conseguindo relacionar informação recolhida com experiências passadas).

Figura 1.1: *iRobot Roomba 760* (fonte: [Iro]).

Figura 1.2: *iRobot RP-VITA* (fonte: [Iro]).

Figura 1.3: *Honda Miimo e ASIMO* (fonte: [Hon]).

Deste ponto de vista, podemos imediatamente depreender que, quer os sistemas robóticos atuais, quer os futuros, deverão possuir capacidades em tudo semelhantes às do ser humano, como por exemplo o sistema de visão. Este sistema permitirá ao robô tomar decisões autonomamente, com base no reconhecimento de cores e formas de objetos, bem como de outros detalhes do meio ambiente circundante.

Uma aplicação prática destes dois conceitos (integração de sistemas robóticos e sistemas de visão) é o evento RoboCup, realizado anualmente e no qual a Universidade de Aveiro possui uma representação especial, através da equipa CAMBADA.

1.1.1 RoboCup

O RoboCup[4] é uma iniciativa internacional e tem como principal objectivo promover o desenvolvimento da Inteligência Artificial, Robótica e áreas relacionadas. Esta iniciativa con-

[4] `http://www.robocup.org/`, acedido em: 12/05/2013

siste na organização anual de competições em que equipas robóticas de todo o mundo jogam futebol entre si. A ideia base surgiu em 1992, por Alan Mackworth, professor na Universidade British Columbia, Canadá, sendo que o primeiro evento oficial RoboCup ocorreu no ano de 1997 em Nagoya, Japão, contando com a participação de 40 equipas (competição real e simulação combinadas) e 5.000 espectadores. Contudo, o sonho que serve de base a esta iniciativa e promove um estudo e desenvolvimento constantes, consiste em que, em meados do século XXI, uma equipa de robôs humanóides consiga vencer (segundo as regras da FIFA), um jogo de futebol contra a equipa humana vencedora da Taça Mundial.

Contudo e embora a competição de futebol seja a mais conhecida e, por isso mesmo, imediatamente associada ao termo RoboCup, existem na verdade quatro competições principais:

- ***RoboCup Soccer***: tal como já referido anteriormente, esta é a competição mais conhecida, sendo o seu foco principal, o jogo de futebol. Os objetivos de pesquisa compreendem a cooperação de sistemas multi-robô e multi-agente em ambiente altamente dinâmicos e adversos;

- ***RoboCup Rescue***: esta competição permite desenvolver, demonstrar e avaliar capacidades robóticas avançadas numa situação de resposta a uma determinada emergência.

- ***RoboCup @Home***: esta competição visa desenvolver serviços e tecnologia que permitam que sistemas robóticos possam futuramente colaborar ou desempenhar tarefas domésticas;

- ***RoboCup Junior***: esta competição surge como uma iniciativa sobretudo educacional, que permite a jovens estudantes dos mais diversos níveis de ensino (que não possuam ainda os recursos suficientes para participar noutras ligas superiores), participar em projetos e eventos de robótica locais, regionais e internacionais. Estes projetos e eventos incluem desafios como futebol, dança ou salvamento.

Com a exceção da competição *RoboCup @Home*, todas as restantes são compostas por várias ligas. Na Liga de Robôs Médios (MSL) da competição *RoboCup Soccer*, na qual participa a equipa CAMBADA (figura 1.4), cada equipa é constituída por um máximo de 5 robôs, cada um com 50 cm de diâmetro, 80 cm de altura e 40 kg de peso (valores máximos). O comportamento destes robôs é totalmente autónomo, podendo haver comunicação entre os vários robôs da equipa, ou entre os diversos robôs em jogo e um sistema remoto (denominado *Base Station*). Quaisquer outras comunicações com o meio exterior ou formas de interação com seres humanos são absolutamente proibidas (exceto, claro está, casos especiais previstos no regulamento, como por exemplo o caso de uma avaria). A competição tem lugar num recinto com uma dimensão aproximada de 18 m × 12 m. O ambiente de jogo encontra-se estruturado com base em cores e pontos de referência. Assim, o campo é verde, com balizas, linhas e marcas circulares de centro e penálti brancas. Adicionalmente existem 6 marcas circulares para reinício de jogo e consequente recolocação da bola, podendo, opcionalmente, a cor destas ser preta. A bola é especificada antes de uma competição, com uma antecedência mínima de um mês, podendo, por isso, assumir qualquer cor (exceto, claro está, cores predominantemente verde, preto ou branco). Assim sendo, o sistema de visão de cada robô deve ser suficientemente robusto, por forma a identificar correctamente a bola, sem a necessidade de qualquer tipo de recalibração. Cada robô dispõe de uma ou mais câmaras digitais, podendo,

Figura 1.4: Equipa CAMBADA durante a competição RoboCup 2012, México (fonte: [Ytb]).

deste modo, detetar e localizar os objetos que se encontram no campo e, inclusivamente, determinar corretamente a sua localização.

Informação mais detalhada sobre os regulamentos e regras aplicados a cada género de competição poderá ser encontrada no site oficial[5].

1.1.2 CAMBADA

A CAMBADA[6] (*Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture*) é a equipa que representa a Universidade de Aveiro[7] na Liga de Robôs Médios do RoboCup. O projeto teve início em 2003 e envolve pessoas das mais diversas áreas, que contribuem para o desenvolvimento da estrutura mecânica do robô, o seu *hardware*, arquitetura e controladores, bem como o desenvolvimento de *software* em áreas como o processamento e análise de imagens, sensores, raciocínio e controlo, cooperação baseada numa Base de Dados em Tempo Real (*Real-Time DataBase - RTDB*), a comunicação entre robôs e o desenvolvimento de uma estação base eficiente (*basestation*). Os robôs operam em tempo real, num ambiente altamente dinâmico, multiobjetivo, parcialmente cooperativo e parcialmente adverso, sempre de forma autónoma, comunicando exclusivamente entre si e com a estação base. A figura 1.5 ilustra uma fotografia de um robô da equipa CAMBADA.

Arquitetura geral dos robôs

A arquitetura adotada para o desenvolvimento dos robôs baseia-se num paradigma biomórfico, isto é, a construção dos robôs é inspirada nos princípios dos sistemas biológicos

[5] `http://www.robocup.org/about-robocup/regulations-rules/`, acedido em: 12/05/2013
[6] `http://robotica.ua.pt/CAMBADA/`, acedido em: 13/05/2013
[7] `http://www.ua.pt`, acedido em: 13/05/2013

Figura 1.5: Fotografia de um robô da equipa CAMBADA (fonte: [Nev+09]).

presentes nos seres humanos ou animais [Nev+10b]. Assim, cada robô possui uma unidade de processamento principal (um computador portátil), que desempenha o papel de cérebro, sendo responsável pela coordenação e comportamento do robô. Esta unidade de processamento principal lida diretamente com os sensores que exigem uma elevada largura de banda (como o sistema de visão) e coordena a comunicação externa com os outros robôs. A mesma unidade processa ainda, a informação recebida pelo sistema distribuído de baixo nível de deteção (o sistema nervoso), enviando comandos por forma a controlar o comportamento e atitude do robô. O "sistema nervoso" é composto por vários microcontroladores interligados entre si através de uma rede CAN (*Controller Area Network*). Apesar da complexidade subjacente, podemos ver este sistema como o responsável por quatro funções básicas: movimento, odometria (método que permite estimar a posição do robô), chuto e monitorização do sistema (por exemplo, o nível das baterias e o estado de cada microcontrolador).

Sistema de visão dos robôs

Ao nível da visão [Nev+10a], os robôs usam um sistema catadióptrico, frequentemente denominado sistema de visão omnidirecional. Este sistema é baseado na utilização de uma câmara de vídeo digital instalada verticalmente no topo dos robôs e apontada a um espelho hiperbólico. Este tipo de espelho reflete a 360° a área envolvente, permitindo, desta forma, que o robô possa localizar e reconhecer a bola, as balizas, os obstáculos e as linhas brancas delimitadoras do campo. Adicionalmente, o guarda-redes possui uma câmara frontal, por forma a permitir detetar as bolas num espaço 3D.

No que diz respeito à calibração do sistema de visão, a configuração do mapa de distâncias e a definição dos valores inerentes a cada cor são efetuados manualmente. Durante a competição e de forma autónoma, a execução contínua de um algoritmo permite configurar os parâmetros principais da câmara, nomeadamente a exposição, o ganho, o brilho e o balanço de cores. Deste modo, é possível detetar e corrigir qualquer tipo de alteração durante uma competição

(como por exemplo, a luminosidade).

Antes de se proceder a qualquer tratamento ou análise de cada imagem capturada, um algoritmo encarrega-se de converter o plano do campo visual da imagem em coordenadas do plano real, utilizando o robô como centro deste sistema. Desta forma, é possível ter uma perceção da real distância que separa o robô dos diversos objetos (assumindo que os mesmos se encontrem no chão).

De modo a conseguir cumprir os requisitos de processamento de toda a informação relativa à imagem em tempo real, foram implementadas estruturas de dados eficientes, bem como explorado o paralelismo entre a extração e a classificação de cor, duas tarefas computacionalmente exigentes. Deste modo, torna-se possível tirar o máximo partido dos processadores *dual core* presentes nos portáteis utilizados.

1.2 Motivação

Os computadores convencionais podem ser utilizados para uma ampla gama de aplicações. No entanto, um computador de uso geral é, como o próprio termo indica, uma ferramenta generalista, possuindo uma arquitetura e um conjunto de instruções fixos, o que obriga a que aplicações diferentes sejam sempre programadas segundo restrições predefinidas. A sua programação é bastante simples e relativamente rápida, se considerarmos a enorme disponibilidade de ferramentas e a curva de aprendizagem, requerida para o estudo de grande parte das linguagens de alto nível. Esta abordagem traduz-se num elevado nível de flexibilidade, visto que qualquer alteração num dado algoritmo é facilmente incorporável no código e, em geral, o resultado alcançado possui um bom desempenho, com um custo razoável. Contudo, se considerarmos uma aplicação específica (por exemplo, o processamento de imagem), esta abordagem não assegura o melhor desempenho.

Uma alternativa consiste em desenvolver ou utilizar circuitos orientados à aplicação ou tarefa em questão, que sejam capazes de assegurar um bom desempenho. Este tipo de circuitos, normalmente designados como *Application Specific Integrated Circuit* (ASIC), possuem um controlo fixo e unidades funcionais personalizadas e otimizadas para uma dada aplicação, utilizando menos recursos de *hardware*. Porém, os custos de projeto e de implementação são demasiado elevados e apenas justificáveis quando produzidos em grande quantidade. Esta alternativa torna-se ainda menos viável quando comparada com os computadores de uso geral, pois possui um tempo de desenvolvimento longo, enquanto o desempenho dos computadores de uso geral aumenta muito rapidamente. Isto significa que este tipo de solução, baseada em ASIC, pode tornar-se obsoleta num curto espaço de tempo, sendo ainda completamente inflexível, dado que a sua funcionalidade não pode ser modificada após o fabrico.

A computação reconfigurável surge como uma abordagem que combina as duas técnicas descritas anteriormente, possibilitando, deste modo, eliminar as desvantagens associadas ao *software* "puro" (implementado num computador de uso geral) e *hardware* "puro". Esta solução baseia-se em dispositivos lógicos reprogramáveis que podem atingir um desempenho elevado e, ao mesmo tempo, fornecer a flexibilidade da programação ao nível de portas lógicas.

Um exemplo destes dispositivos lógicos reprogramáveis são as FPGAs, sendo estas, os dispositivos lógicos reprogramáveis de maior capacidade disponível hoje em dia. Uma das maiores vantagens das FPGAs é a sua arquitetura flexível, sendo, por isso, bastante versátil

para uma ampla gama de aplicações, das quais podemos destacar, a título de exemplo, a implementação de controladores de dispositivos, circuitos de codificação, lógica arbitrária, prototipagem e emulação de sistemas. Contudo, a frequência de relógio suportada nas FPGAs recentes é significativamente menor do que a frequência utilizada em computadores de uso geral, devendo-se este facto, em parte, à necessidade de acomodar interligações programáveis. Quer isto dizer que o mapeamento direto de uma aplicação de *software* para uma FPGA não irá, por si só, assegurar um desempenho mais elevado que o atingido num computador de uso geral. Portanto, a fim de se conseguir um bom desempenho, surge a necessidade de mudar igualmente o paradigma de programação, recorrendo normalmente a três técnicas:

- **largura de banda elevada no acesso à memória:** contrariamente aos computadores de uso geral que organizam a memória como um conjunto de palavras de tamanho fixo, numa FPGA é possível organizar a memória de acordo com a dimensão atual dos dados, permitindo que estes sejam processados numa única operação;

- **uso de unidades funcionais específicas e otimizadas:** o conjunto de instruções básicas utilizado em computadores de uso geral é desenvolvido por forma a servir uma ampla gama de aplicações. Uma desvantagem associada é o facto de operações específicas precisarem de muitas instruções para serem expressas e, consequentemente, necessitarem de muitos ciclos de relógio para serem concluídas. As FPGAs permitem criar unidades funcionais optimizadas a certas operações e tamanho de dados, o que se traduz no facto da unidade ser capaz de executar, num único ciclo de relógio, algumas operações que requerem vários ciclos num computador de uso geral;

- **exploração de paralelismo e de *pipelining*:** o desenvolvimento e uso de unidades funcionais específicas e otimizadas é normalmente simples e ocupa poucos recursos de *hardware*, possibilitando a sua reprodução por forma a explorar o paralelismo. Além disso, as FPGAs típicas incorporam um elevado número de *flip-flops* e LUTs distribuídos pelo dispositivo, podendo ser utilizados para permitir uma gestão flexível de registos, construídos com base nestes elementos de memória. Isto permite a implementação de técnicas de escalonamento (*pipelining*) eficientes que resultam num melhor desempenho, aumentam o nível de utilização dos recursos de *hardware* e reduzem os acessos à memória externa.

- **memória distribuída:** os recursos de memória são fundamentais para a maioria das aplicações de maior complexidade sobre FPGAs. As FPGAs integram dois tipos primários de memória: distribuída e blocos de memória. Enquanto que esta última é agrupada em bancos e baseada em lógica específica, a memória distribuída é conseguida utilizando uma parte da lógica reconfigurável disponível (no caso, LUTs). Esta característica acrescenta um elevado nível de flexibilidade ao nível da arquitetura de soluções, cujo processamento sobre dados possa ser repartido em subconjuntos mais pequenos, localizados proximamente e diretamente acessíveis às entidades que os processam.

Com o objetivo de atingir melhores resultados, os sistemas reconfiguráveis podem ainda ser compostos por lógica reconfigurável interligada a um processador de uso geral. Deste modo, o processador executa operações que geralmente não podem ser realizadas eficientemente em lógica reconfigurável, enquanto as tarefas computacionalmente intensivas da aplicação são mapeadas em *hardware*.

Assim sendo, as FPGAs revelam-se um excelente aliado para a implementação de um sistema de processamento de imagem eficiente, isto porque o tipo de operações geralmente utilizadas em processamento de imagem são altamente paralelizáveis. Além disso e tal como referido anteriormente, a solução final é uma implementação em *hardware*, especialmente otimizada ao problema em questão, o que permitirá aliviar o peso computacional ao processador de uso geral.

1.3 Objetivos

O principal objetivo desta dissertação consiste na redução do peso computacional da visão por computador da equipa CAMBADA (atualmente implementada no computador principal), através da implementação de parte dos algoritmos de visão em *hardware* reconfigurável.

No sistema a desenvolver, o *hardware* reconfigurável, baseado em FPGAs, funcionará como coprocessador, libertando o computador principal de cada robô das tarefas computacionalmente intensivas, adaptadas à implementação em *hardware*, e disponibilizando ao computador principal os dados já tratados.

O coprocessador será baseado em múltiplos elementos de processamento (*cores*), quer genéricos, quer especializados, os quais permitirão explorar o paralelismo normalmente disponível nos algoritmos de processamento de imagem/vídeo. Além destes, por questões de eficiência de comunicação, o coprocessador deverá incluir também, memória e interfaces, de forma a permitir o interface direto com as câmaras do módulo *VmodCAM*, para que as tramas sejam capturadas, armazenadas e processadas sem qualquer intervenção do computador principal.

Assim sendo, o projeto foi desenvolvido com base na seguinte sequência de tarefas:

- Estudo do sistema de visão da equipa CAMBADA (arquitetura, algoritmos e limitações);

- Revisão do estado de arte sobre a implementação em FPGA de algoritmos de processamento de imagem;

- Estudo dos modos de interligação entre o módulo *VmodCAM* da Digilent, a FPGA e o computador principal;

- Especificação das características do coprocessador de suporte à execução dos algoritmos de visão;

- Conceção do coprocessador baseado em FPGA, contendo diversos elementos de processamento, memória e interfaces para ligação entre o módulo *VmodCAM*, os núcleos de processamento e o computador principal;

- Modelação em VHDL, síntese e implementação em FPGA das tarefas de aquisição de imagem, segmentação, imagem de máscara, compressão e deteção de bolhas de cor;

- Teste e avaliação do coprocessador.

1.4 Contributo

Tal como mencionado no capítulo 1.3, o principal objetivo do presente projeto visa reduzir o peso computacional do sistema de visão, no computador principal de cada robô. Por outro lado, não podemos esquecer que os robôs operam em tempo real, num ambiente altamente dinâmico, multiobjetivo, parcialmente cooperativo e parcialmente adverso, dependendo fundamentalmente, da informação adquirida pelo seu sistema de visão. Assim, importa sobretudo, assegurar que o tempo de processamento de toda a informação relativa à imagem é o menor possível, de modo a que todas as decisões possam ser tomadas o mais rapidamente possível e principalmente, em tempo útil de reação face a um evento.

Tendo em conta os objetivos delineados, esperamos que este projeto possa contribuir para:

- implementação e redefinição de algoritmos de processamento de imagem paralelizáveis;
- definição de estruturas auxiliares de dados, simples e eficientes, no sentido de agilizar o acesso à informação por parte dos algoritmos de processamento de imagem;
- processamento em tempo real e, tanto quanto possível, numa perspetiva *raster scan*, por forma a reduzir o número de recursos utilizados (eventualmente necessários para a implementação de outros módulos) e a disponibilizar a informação processada, tão breve quanto possível;
- implementação de uma solução o mais parametrizável possível, por um lado, de modo a suprimir a necessidade sistemática de sintetizar o projeto e, por outro, visando igualmente que o sistema possa ser personalizado e adaptado a diferentes cenários.

1.5 Estrutura da dissertação

Nesta secção será apresentado um breve resumo do conteúdo dos restantes capítulos que compõem esta dissertação. Assim sendo, este documento encontra-se organizado como:

Capítulo 2 - Conceitos fundamentais: Neste capítulo são apresentados alguns conceitos teóricos considerados relevantes para o restante trabalho. De uma forma mais detalhada será introduzida a visão computacional, bem como alguns exemplos de sistemas de visão, sobretudo no âmbito da competição RoboCup MSL. Serão igualmente apresentadas algumas considerações base sobre conceitos, aplicações e métodos relacionados com o processamento de imagem, como espaços de cor e compressão de dados. Seguidamente serão introduzidas as FPGAs enquanto plataforma de desenvolvimento para sistemas de coprocessamento, bem como apresentados alguns trabalhos relacionados, que interliguem os sistemas de visão e soluções implementadas em *hardware* reconfigurável.

Capítulo 3 - Sistema atual de visão da equipa CAMBADA: Neste capítulo será apresentada a equipa CAMBADA com um maior nível de detalhe, bem como a arquitetura do seu atual sistema, com especial ênfase no sistema de visão e nos algoritmos de calibração do referido sistema e deteção de objetos.

Capítulo 4 - Arquitetura do *Vision System Processor*: Neste capítulo serão descritas as principais características tidas em consideração para a definição da arquitetura do sistema de coprocessamento *Vision System Processor*. Em seguida, apresentaremos a referida arquitetura, detalhando e descrevendo sumariamente o fluxo de dados pretendido. Finalmente, serão apresentadas as interfaces do sistema.

Capítulo 5 - Implementação do *Vision System Processor*: Neste capítulo será apresentada e detalhada a implementação da arquitetura apresentada no capítulo anterior, bem como descritos detalhadamente, cada um dos módulos implementados. Posteriormente será apresentada e detalhada a arquitetura base de um projeto disponibilizado pelo fabricante e que visa demonstrar a interligação e o funcionamento da placa *Atlys* e o módulo *VmodCAM*.

Capítulo 6 - Resultados: Este capítulo pretende apresentar os aspectos a avaliar no presente sistema, bem como testes realizados de modo a comprovar a funcionalidade de cada um dos módulos desenvolvidos, assim como uma síntese dos resultados obtidos.

Capítulo 7 - Conclusão e trabalho futuro: Neste capítulo serão discutidas algumas considerações sobre o trabalho desenvolvido, dificuldades e problemas encontrados. De igual forma, serão elaboradas e apresentadas conclusões que permitam avaliar este projeto sob um ponto de vista crítico. Finalmente, serão abordados problemas e soluções que necessitem ser reformulados, sendo adicionalmente sugeridas melhorias contínuas ao trabalho desenvolvido, bem como eventuais implementações em falta.

Anexo A - Plataforma de desenvolvimento: Neste anexo será apresentada resumidamente a plataforma de desenvolvimento *Atlys* e o módulo de aquisição de imagem *VmodCAM*.

Anexo B - Resultado do módulo *ProcRunUnit*: Este anexo inclui uma listagem que ilustra o resultado obtido após o processamento do módulo *ProcRunUnit*. A listagem apresenta uma lista de instruções inferida após a análise e deteção de sequências de uma mesma cor, bem como, a sua relação de vizinhança com outras sequências encontradas anteriormente. Estas instruções serão posteriormente utilizadas pelo módulo *PipelineUnit*, de modo a definir operações sobre os diversos objetos presentes em memória e o cálculo das suas características.

Anexo C - Resultado do módulo *PipelineUnit*: Este anexo inclui uma listagem que ilustra o resultado obtido após o processamento do módulo *PipelineUnit*. A listagem apresenta uma lista de objetos e as diversas características associadas a cada um, inferida após a execução das instruções processadas pelo anterior módulo (*ProcRunUnit*).

Anexo D - Diagrama de simulação do módulo *PipelineUnit*: Este anexo inclui um *screenshot* do diagrama de simulação do módulo *PipelineUnit*, que pretende ilustrar um momento temporal no processamento dos dados (nomeadamente, o cálculo das características para um determinado objeto).

Capítulo 2

Conceitos fundamentais

2.1 Introdução

Neste capítulo serão apresentados alguns conceitos teóricos considerados relevantes e implícitos no restante trabalho. De uma forma mais detalhada, apresentaremos o conceito de visão computacional e sistemas de visão, suas possíveis e pontenciais aplicações, sobretudo no âmbito da competição RoboCup MSL. Serão igualmente apresentadas algumas considerações sumárias sobre conceitos e métodos relacionados com o processamento de imagem, como espaços de cor e compressão de dados. Seguidamente pretende-se abordar as FPGAs enquanto plataforma de desenvolvimento para sistemas digitais, focando especialmente, soluções de coprocessamento. De igual forma, apresentaremos ainda alguns trabalhos relacionados que interliguem, por um lado, os sistemas de visão e, por outro, soluções de coprocessamento implementadas sobre FPGAs.

2.2 Visão computacional

A visão computacional é a ciência que procura desenvolver teoria e tecnologia para a construção de sistemas artificiais, que necessitem de obter informação através da imagem, ou quaisquer dados multidimensionais. Esta ciência pode ser igualmente descrita como um complemento da visão biológica. Na visão biológica procura-se estudar a perceção visual dos seres humanos e outros animais, obtendo modelos que descrevem como estes sistemas funcionam em termos de processos fisiológicos. De forma análoga, a visão computacional estuda e descreve sistemas de visão artificial implementados, quer por *software*, quer por *hardware*.

Apesar de existirem atualmente uma grande quantidade de trabalhos reconhecidos nesta área, somente após o final da década de 1970 surgiram os primeiros estudos aprofundados, quando a evolução dos computadores permitiu processar grandes quantidades de dados (como imagens).

Contudo, estes estudos tiveram origem em outros campos de pesquisa, pelo que não existe uma formulação padrão para o problema da visão computacional, ou para a forma como os problemas desta devem ser resolvidos. O que podemos encontrar atualmente são diversos métodos que permitem resolver várias tarefas bem definidas, sendo estes métodos bastante especializados, não podendo, na maioria das vezes, ser generalizados para várias aplicações.

2.2.1 Aplicações

Uma das aplicações mais conhecidas da visão computacional é a medicina ou o processamento médico de imagens. Esta área caracteriza-se pela extração de informação a partir da imagem, com o objetivo de realizar um diagnóstico ao paciente. Neste caso a imagem pode ser proveniente de uma microscopia, radiografia, angioplastia, ultrasonografia, tomografia ou ressonância magnética.

As aplicações militares são talvez uma das áreas de maior aplicação da visão computacional, embora somente uma pequena parte desse trabalho se encontre disponível ao público. Exemplos simples nesta área incluem a deteção de unidades inimigas ou mísseis teleguiados (dos quais, em sistemas mais avançados, o próprio míssel seleciona o alvo preciso numa determinada área, baseado no processamento da imagem do local).

Também na indústria podemos encontrar diversas aplicações. Nesta área, a informação da imagem auxilia processos, como por exemplo o controlo de qualidade e o cálculo de posição e orientação de um braço robótico.

Mais recentemente, também a indústria automóvel passou a incorporar a visão computacional nos denominados veículos autónomos, para os quais podemos identificar dois níveis de autonomia: parcial e total. Na primeira, o sistema de visão computacional é mais simples e serve meramente como auxílio à condução em determinadas situações, como por exemplo, visualizar a área disponível ou eventuais obstáculos durante um estacionamento. Por sua vez, o sistema de visão implementado para uma autonomia total permite ao veículo obter a sua localização, produzir mapas do ambiente que o rodeia e detetar obstáculos. A figura 2.1 apresenta uma fotografia do veículo protótipo utilizado no âmbito do projeto AtlasCar[1], desenvolvido pela Universidade de Aveiro, o qual, entre outros sensores, utiliza um sistema de visão que lhe permite total autonomia. Embora os resultados alcançados sejam bastante animadores, quer por este projeto, quer por outros projetos semelhantes, esta tecnologia ainda não atingiu maturidade suficiente de modo a ser comercializada massivamente.

Figura 2.1: Projeto AtlasCar, desenvolvido pela Universidade de Aveiro (fonte: [Atlb]).

[1] `http://atlas.web.ua.pt/atlascar.html`, acedido em 2/06/2013

2.2.2 Espaços de cor

A cor de um objeto encontra-se relacionada com o espectro de luz refletida (ou emitida). Contudo, importa salientar que diferentes objetos podem induzir a sensação de cores iguais, razão pela qual é importante conseguir descrever a cor de uma forma objetiva.

O espaço de cores é um modelo matemático abstrato que permite descrever as cores de uma forma objetiva, recorrendo a sequências de números, tipicamente compostas por três ou quatro valores (frequentemente denominados componentes de cor).

A combinação destas componentes permite a representação de qualquer cor. Podem definir-se dois tipos de espaços de cor: aditivo e subtrativo. O primeiro espaço é tipicamente utilizado em sistemas de visualização de imagem, como por exemplo, monitores e televisores. Habitualmente neste tipo de equipamentos, a ausência de luz permite obter a cor preta, enquanto que as restantes cores são formadas pela adição de luz a cada uma das componentes de cor, sendo que o valor máximo da combinação das componentes permite obter a cor branca. Por sua vez, o espaço subtrativo é utilizado, sobretudo, nos sistemas de impressão, onde se assume que o suporte de impressão (papel) é, em si, um elemento refletor da luz ambiente. Deste modo, estes sistemas combinam as componentes de cor (azul claro, magenta, amarelo e preto), por forma a que a cor resultante permita subtrair a luz ao suporte de impressão.

Embora existam vários espaos de cores (tais como RGB, YUV, HSV ou CMYK), a sua especificidade permite que uns espaos sejam mais adequados do que outros, quando considerados os sistemas de visão computacional e a sua finalidade.

Espaço de cores RGB

Definido como tipo aditivo, este espaço é baseado no modelo de cores RGB. Uma cor contida neste espaço pode ser descrita através da indicação da quantidade das componentes vermelho, verde e azul (figura 2.2). Este modelo foi criado com base na anatomia do olho humano, sendo um dos modelos mais utilizados em diversas aplicações, nas quais se incluem os já mencionados sistemas de visualização.

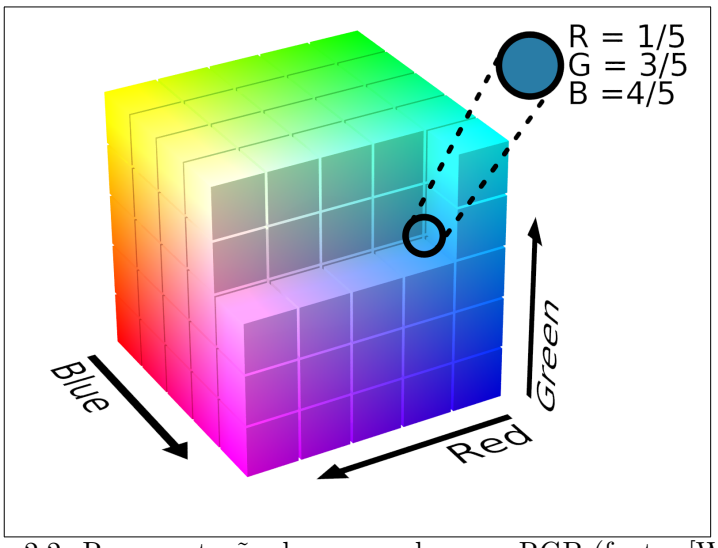


Figura 2.2: Representação do espaço de cores RGB (fonte: [Wikd]).

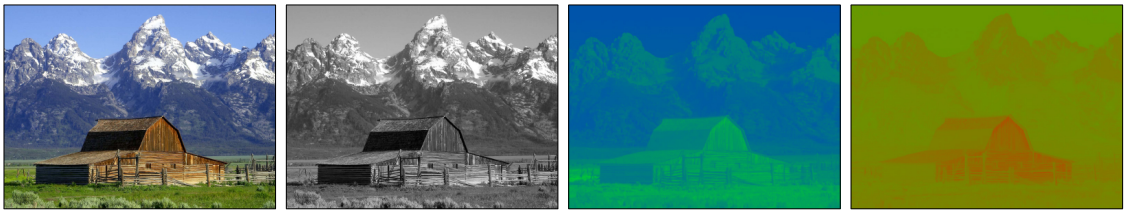
Cada componente é normalmente representada por 8 *bits*, possibilitando um máximo de

256 níveis de cor diferentes. A combinação das três componentes permite obter 16.777.216 cores diferentes (256x256x256). O espaço ocupado em memória por uma imagem RGB depende, essencialmente, do tamanho da imagem (medido em número de píxeis) e da resolução de cores associada a cada píxel. Uma imagem 1024x768 píxeis representada utilizando os 8 *bits* mencionados anteriormente, ocupa 2.359.296 *bytes* (18.874.368 *bits*). De modo a reduzir o espaço ocupado em memória, algumas soluções adotam variantes de representação das componentes em menos *bits*, reduzindo o número de cores definidas e, consequentemente, o tamanho da imagem. O número total de *bits* utilizados para representar uma determinada cor é denominado profundidade de cor, sendo que as codificações de 1, 2, 4, 5, 8 e 16 *bits* por cor são as mais frequentes. Importa ainda salientar que, para uma determinada profundidade de cor podem existir diversas variantes de codificação das componentes, alterando somente o número de *bits* que representa cada componente (por exemplo, para 16 *bits* de cor, podemos ter as variantes RGB565, RGB656 ou RGB555 - esta última com um *bit* adicional de transparência).

Espaço de cores YUV

O modelo YUV define um espaço de cores baseado numa componente de luminância ou intensidade luminosa (Y) e duas componentes de crominância ou cor (U e V). Este modelo é utilizado sobretudo nos sistemas de vídeo e sinal analógico e digital de televisão, devido à compatibilidade com as imagens em tons de cinzento (componente Y). Contudo, também se pode encontrar em equipamentos de fotografia.

Dado que a sensibilidade do olho humano é maior à intensidade luminosa do que à cor, algumas câmaras de video efetuam uma subamostragem espacial das componentes de crominância, a fim de reduzir o volume de dados. Assim, é comum encontrar muitos sistemas de aquisição de vídeo que implementam variantes de subamostragem YUV, como o YUV 4:2:2, YUV 4:2:0 e YUV 4:1:1). A figura 2.3 ilustra o exemplo de uma imagem representada em função de cada uma das componentes YUV.



(a) Imagem original. (b) Componente *Y*. (c) Componente *U*. (d) Componente *V*.

Figura 2.3: Exemplo de imagem representada no espaço de cores YUV. (fonte: [Wike]).

Espaço de cores HSV

O modelo HSV (também denominado HSB) é formado pelas componentes tonalidade (*hue*), saturação (*saturation*) e valor ou brilho (*value*).

Este sistema utiliza uma representação cilíndrica de coordenadas dos pontos presentes no modelo RGB (figura 2.4). Esta representação visa reorganizar a geometria encontrada no modelo RGB, de modo a torná-la mais intuitiva e percetivamente relevante, em relação à representação cartesiana (cubo). No cilindro representativo de cor, o ângulo em torno do

eixo vertical central corresponde à componente *hue*, enquanto que a distância desde o eixo determina a componente *saturation* e a distância ao longo do eixo define à componente *value*.

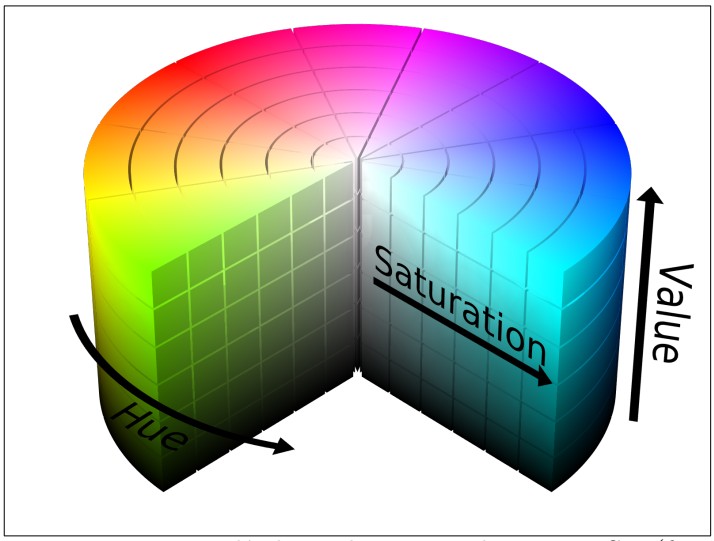


Figura 2.4: Representação cilíndrica do espaço de cores HSV (fonte: [Wikb]).

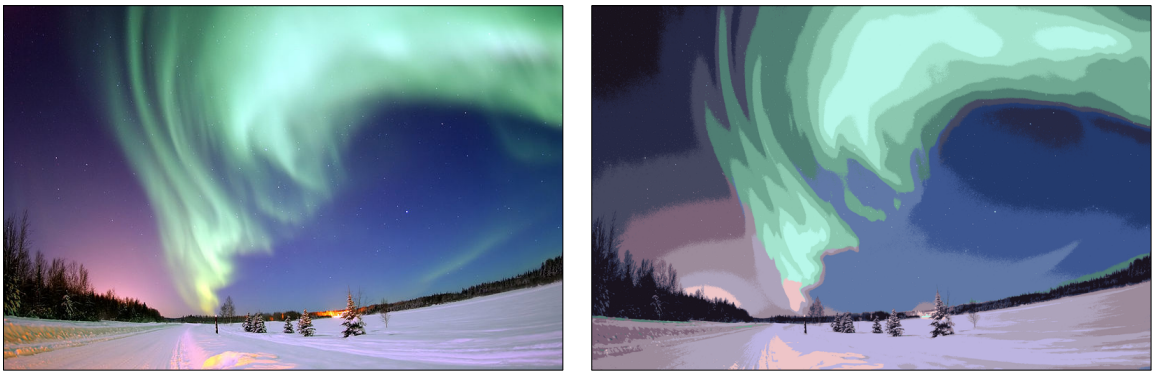
2.2.3 Segmentação de imagem

Em visão computacional, o conceito de segmentação diz respeito ao processo de subdividir uma imagem em múltiplas regiões ou objetos, com o objetivo de simplificar e/ou alterar a sua representação, de modo a facilitar a sua posterior análise. Como resultado da segmentação de uma imagem, cada região ou objeto consiste num conjunto de píxeis que partilham entre si uma característica visual, como a cor, a intensidade luminosa, a textura ou a continuidade.

A classificação de uma cor não depende somente do sistema sensorial. Por exemplo, a perceção de cor no sistema visual humano depende, essencialmente, do cérebro e não somente da retina ou do olho em si. Por outro lado, sabemos que as cores de um objeto variam, sobretudo, em função das condições de iluminação. Por esse motivo, a sua classificação pode revelar-se uma tarefa complicada para um sistema computacional (como é o caso dos robôs em ambiente de competição RoboCup MSL).

Existem vários algoritmos e técnicas passíveis de serem utilizados na segmentação de uma imagem. De modo a que possam ser corretamente aplicadas e os resultados possam ser igualmente úteis, estas técnicas devem ser combinadas com algum conhecimento prévio sobre as imagens e o ambiente da sua aquisição. Uma técnica simples consiste em classificar uma imagem em tons cinzentos (monocromática) numa representação binária, utilizando um valor predefinido (as cores definidas acima desse valor são representadas com o valor "1"; todas as restantes assumem o valor "0" - zero). Outras possíveis técnicas poderão recorrer, por exemplo, a histogramas, deteção de contornos ou procurar tirar partido da própria imagem, adotando uma classificação de cor que maximize a compressão da imagem resultante. Em [Wikc] são descritos, sumariamente, outros possíveis métodos. A figura 2.5 ilustra o exemplo de uma imagem segmentada.

Um método utilizado frequentemente na classificação de cor nos sistemas de visão computacional consiste na utilização de uma LUT (*Look-Up Table*). A LUT pode considerar-se como uma tabela que permite representar o mapeamento do espaço de cores utilizado, nas



(a) Imagem original. (b) Imagem segmentada.

Figura 2.5: Exemplo de imagem segmentada. (fonte: [Wikc]).

cores que se pretende representar. Simplificando, a cada valor resultante da combinação das componentes de cor de um determinado espaço de cores é atribuído um valor predefinido, que será usado para a sua representação. Esta tabela pode ser utilizada de uma forma estática (sendo construída, inicialmente, pelo mesmo ou por outro sistema e mantendo-se inalterada durante a operação do sistema onde se encontra) ou dinâmica. Um aspecto negativo desta abordagem prende-se com o espaço que esta ocupa em memória, o que, em sistemas com poucos recursos disponíveis, poderá ser bastante limitativo.

2.3 Compressão de dados

A compressão de dados consiste em reduzir o espaço que a informação de representação de uma determinada imagem ocupa. Comprimir os dados permite ainda, retirar a redudância que normalmente pode ser encontrada neste tipo de dados, substituindo eventuais sequências do mesmo símbolo, por uma representação semelhante que ocupe menos espaço (por exemplo, uma sequência como "AAAA" pode ser simplesmente representada por "4A").

A compressão de dados justifica-se, sobretudo, quando surge a necessidade de economizar espaço em dispositivos de armazenamento com capacidade limitada, ou, eventualmente, reduzir o tempo de transmissão de informação entre processos ou sistemas, visando melhorar o seu desempenho.

A compressão pode ser classificada como "com perdas" e "sem perdas". No primeiro caso, a informação resultante não descreve, exatamente, os dados originais. Estes métodos são habitualmente aplicados em situações que permitem a perda de alguns dados considerados irrelevantes. São tipicamente exemplos práticos, a compressão de aúdio e imagem, explorando o facto de os sistemas auditivo e visual humano não permitirem a perceção de sons com frequências muito baixas ou muito altas, ou a distinção de variações de cor muito subtis e sequências de imagem muito rápidas.

Por outro lado, diz-se que um método de compressão é "sem perdas" quando os dados comprimidos descrevem, de forma idêntica, os dados originais. Estes métodos são normalmente indicados para sistemas sensíveis à informação, para os quais, uma pequena perda de dados implica um não funcionamento ou torna os dados incompreensíveis. Um exemplo prático da aplicação deste tipo de método poderá consistir na compressão de uma base de dados de um sistema bancário, que contenha o saldo atual das contas de todos os seus clientes.

Como se compreende, a perda de informação, nesta situação, poderia ser catastrófica, quer para a entidade bancária, quer para os clientes.

No caso prático dos sistemas de visão computacional (nos quais se inclui o presente projeto) é privilegiada a qualidade da imagem relativamente ao tamanho que esta ocupa. Contudo, dados os recursos limitados normalmente associados aos sistemas de suporte, torna-se necessário atender a certos compromissos. Por outro lado, é igualmente importante não descurar os tempos de compressão (e, eventualmente, descompressão, se esta for necessária), durante o processamento de uma imagem.

O RLE (*Run-length encoding*) surge como uma técnica de codificação associada a métodos de compressão, possuindo como principal vantagem a sua simplicidade de implementação e, consequentemente, tempos de processamento e recursos reduzidos. Esta codificação é particularmente indicada para sequências de dados nas quais ocorram frequentemente, repetições de dados com o mesmo valor, podendo estas ser descritas numa única posição de codificação (como comprimento e valor).

Naturalmente, para situações em que não existe repetição de dados, a dimensão final do resultado da aplicação desta técnica supera facilmente o tamanho ocupado pelos dados originais, sendo nesse caso, desvantajoso optar pela codificação. Assim, não teremos especial interesse na aplicação direta desta sobre uma imagem original representada num determinado espaço de cores. No entanto, se considerarmos a sua aplicação sobre uma imagem segmentada, onde a menor variação de dados implica uma maior repetição dos mesmos, esta técnica torna-se realmente interessante.

2.3.1 MS-Windows BMP RLE8

Embora existam outras, uma variação desta técnica de codificação é aplicada na compressão de ficheiros BMP (*Microsoft Bitmap*). No caso específico do RLE8, esta variação é exclusivamente vocacionada para *bitmaps* representados por 8 *bits* de cor (máximo de 256 cores). A figura 2.6 ilustra o esquema de codificação RLE8. Esta variante permite distinguir entre "modo codificado de dados" (contendo sequências codificadas de dados) e "modo absoluto" (contendo os dados originais).

Modo codificado de dados

Neste modo, os dados são constituídos por contador e valor de cor. Se o valor inicial estiver compreendido entre 1 e 255 (inclusivé), este valor é inferido como contador e o valor sucessor identifica o valor de cor. Se o valor inicial for 0 (zero), e dependendo do valor que o sucede, os próximos dados dizem respeito a um marcador especial ou a dados em modo absoluto. Caso o valor que sucede o 0 (zero) inicial seja 0 (zero) ou 1 (um), estamos perante um marcador especial definido como "fim de linha" ou "fim de frame", respetivamente, que como os próprios nomes indicam, sinalizam fim de linha e fim de frame. Se este valor for 2 (dois), este marcador especial indica um salto de posição em relação à atual posição. Os dois próximos valores quantificam esse salto relativamente ao eixo X e Y, respetivamente. As sequências posteriores dizem respeito a essa nova posição, sendo que o espaço intermédio assume uma cor definida como fundo (*background color*).

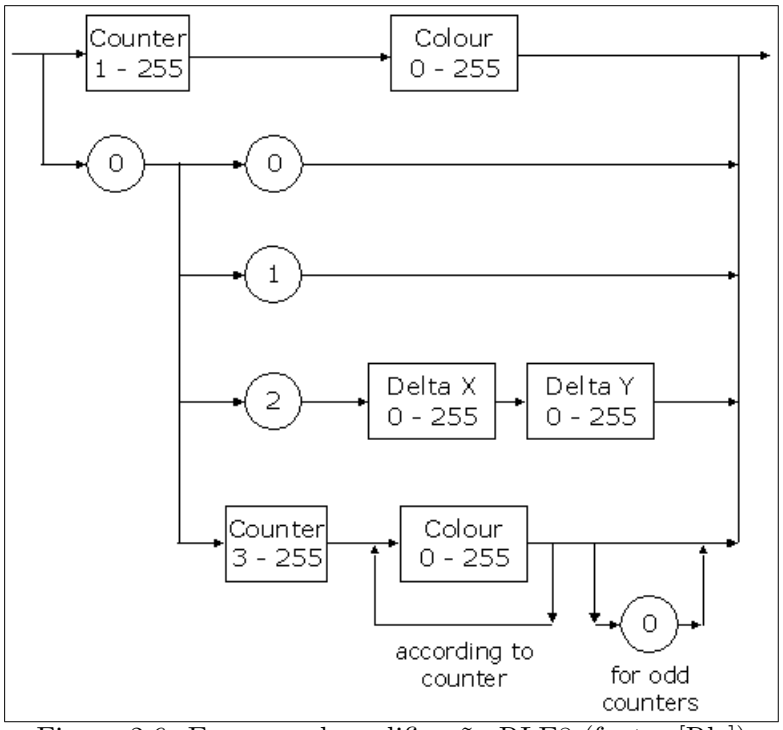


Figura 2.6: Esquema de codificação RLE8 (fonte: [Rle]).

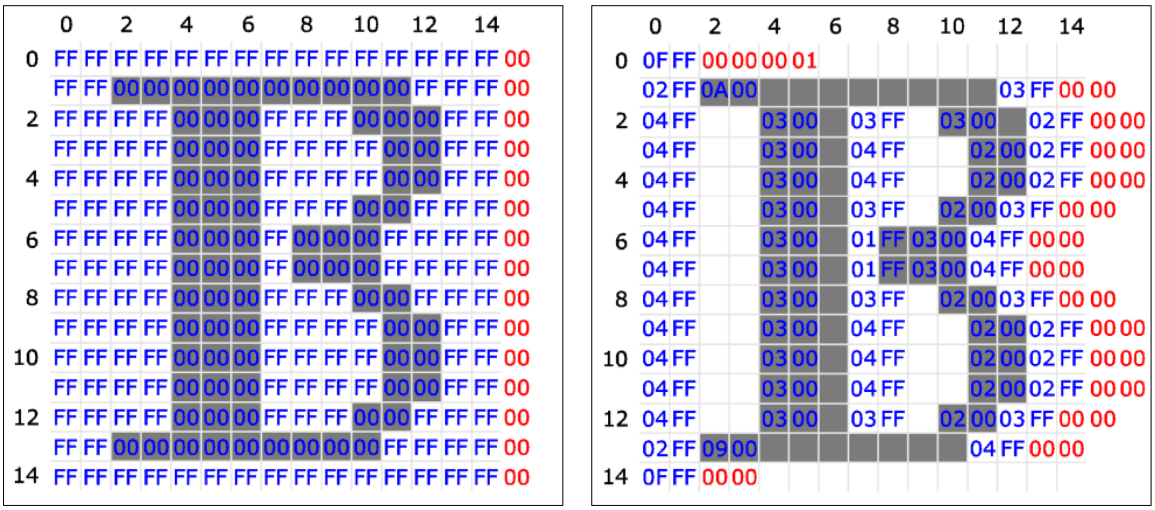
Modo absoluto

Quando o valor inicial é 0 (zero) imediatamente seguido de um valor compreendido entre 3 e 255 (inclusivé), este último valor é assumido como um contador e identifica quantos valores distintos seguintes devem ser considerados como diferentes cores. Este modo é especialmente útil para sequências em que os dados variam com elevada frequência. Se o contador for ímpar, os índices de cor que o sucedem devem terminar com um 0 (zero), por forma a manter um alinhamento de palavras, como no restante esquema. Sequências de apenas um ou dois píxeis (valores), não podem ser codificadas em modo absoluto, sendo necessário proceder à sua codificação no modo normal. Na prática, não existe perda de eficiência, pois ambas as codificações ocupam o mesmo tamanho.

Exemplo prático

A figura 2.7a ilustra um exemplo sugerido em [Rle] de uma imagem *bitmap* não codificada. Apesar de cada linha ser apenas constituída por 15 píxeis, a especificação da *Microsoft* estende a 16 píxeis. Não considerando cabeçalhos e inclusão de tabelas de cor, esta imagem ocupa 240 *bytes*. Por sua vez, a figura 2.7b ilustra o mesmo exemplo de imagem, utilizando codificação RLE8. De igual modo, não considerando cabeçalhos e inclusão de tabelas de cor, esta imagem ocupa somente 158 *bytes*, o que representa uma redução de 34.17 % face ao tamanho ocupado pelos dados originais. Naturalmente, este tipo de codificação depende do tipo de dados a codificar.

Mais detalhes e exemplos práticos poderão ser encontrados em [Rle].



(a) Dados originais. (b) Codificação RLE8.

Figura 2.7: Exemplo de imagem *bitmap* (fonte: [Rle]).

2.4 Sistemas de visão em robótica

Do ponto de vista de organização, um sistema de visão computacional é dependente da aplicação. Também a sua implementação específica, com vista a resolver um determinado problema, depende do facto da funcionalidade ser pré-especificada ou de existir um processo de aprendizagem durante a operação. Embora na maioria das aplicações de visão computacional, os computadores sejam pré-programados para resolver uma tarefa particular, têm surgido, recentemente, vários métodos baseados em aprendizagem.

No entanto, podemos observar algumas funções típicas encontradas em vários sistemas de visão computacional:

- **aquisição de imagem:** uma imagem digital é produzida por um ou vários sensores (vulgarmente denominados câmaras). Dependendo do tipo de sensor, esta imagem pode ser bidimensional, tridimensional ou uma sequência de imagens. O valor de cada píxel indica, tipicamente, a intensidade de luz em uma ou várias faixas de cor, podendo ainda, indicar valores físicos, como profundidade e absorção ou reflexão de ondas eletromagnéticas;

- **pré-processamento:** com o objetivo de assegurar a aplicação e consequente eficácia dos métodos de visão computacional, é, por norma, necessário pré-processar a imagem, de modo a que esta cumpra alguns requisitos. Alguns exemplos compreendem a redução de ruído (de forma a garantir uma maior nitidez e a veracidade da informação), aumento do contraste (assegurando que as informações relevantes possam ser detetadas) e o remapeamento (por forma a assegurar o sistema de coordenadas);

- **extração de características:** de acordo com a finalidade, várias características matemáticas da imagem, em vários níveis de complexidade, podem ser extraídas. A título exemplificativo, podemos salientar a deteção de contornos ou determinados pontos, texturas, formatos ou mesmo movimentos;

- **deteção e segmentação:** em determinado momento, importa decidir quais as regiões de interesse para posterior processamento; tipicamente, como exemplo, encontramos a seleção de regiões de interesse que possam conter um determinado objeto;
- **processamento de alto nível:** quando comparado com a quantidade de dados da imagem adquirida, geralmente, nesta fase, o conjunto de dados é menor. Doravante, processos de alto nível encarregar-se-ão de tratar este conjunto de dados de acordo com a finalidade pretendida.

Um sistema de visão é habitualmente constituído por:

- **uma ou mais câmaras**, podendo ser analógicas ou digitais, mono ou policromáticas;
- **ótica**: uma ou mais lentes para o correto condicionamento da imagem (luz);
- **iluminação**, por forma a realçar os atributos desejados;
- **interface de transmissão/digitalização** de imagens (conhecido como "*framegrabber*"); pode utilizar-se uma interface de transmissão como *firewire* ou USB;
- **processador** (normalmente baseado em PCs) ou DSP (processadores digitais de sinais);
- ***software*** para processar as imagens e detetar as características relevantes;
- **sensor de sincronismo** para a deteção de peças (e/ou disparo de captura de imagem);
- **dispositivo de entrada/saída**, de modo a interagir com partes mecânicas (como atuadores pneumáticos).

2.4.1 Sistemas de visão na competição MSL

Uma parte significativa da inteligência artificial lida com sistemas que realizam ações mecânicas, como movimentar um robô num determinado ambiente. Este tipo de processamento necessita que o sistema de visão computacional se comporte como um sensor de visão, fornecendo informação sobre o ambiente envolvente. Estes sensores de visão assumem a forma física de câmaras.

A competição RoboCup MSL é um exemplo prático perfeito, onde o sistema de visão assume especial importância. O ambiente característico desta competição impõe que o sistema de visão deva ser o mais robusto possível, por forma a detetar corretamente a bola e os diversos obtáculos (não esquecendo a adversidade de algumas variáveis, como a luminosidade). Para além da necessidade de representar com precisão e fiabilidade todo o meio envolvente, o sistema de visão necessita ser igualmente eficiente, por forma a disponibilizar em tempo considerado útil, toda a informação já tratada aos processos de alto nível, responsáveis por decidir o comportamento do robô. De outra forma, a informação torna-se simplesmente obsoleta e errónea, impelindo o robô a responder tardiamente perante uma determinada situação já ocorrida.

Tal como referido em [Rib09], um sistema de visão idealizado para este género de competição deve conter os seguintes processos:

- **aquisição de imagem**

- **calibração da(s) câmara(s)** (incluindo parâmetros da câmara e mapas de distância)

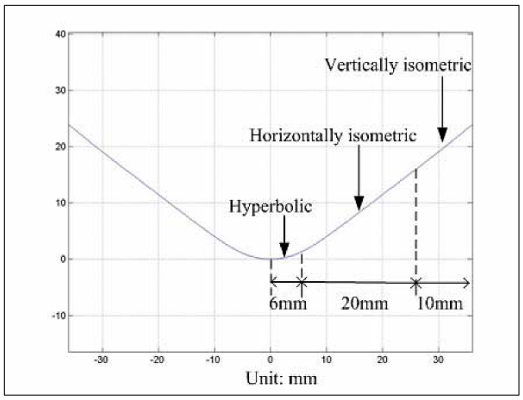
- **segmentação de cor**

- **deteção de objetos** (bola, obstáculos e linhas)

De uma forma geral, quase todas as equipas participantes na competição MSL [Li+13], usam um sistema de visão omnidirecional. Este sistema é constituído por um espelho convexo e por uma câmara fixa na parte superior do corpo do robô, apontada ao espelho. Este sistema permite ao robô adquirir informação visual em todo o seu redor, numa única imagem. Após o processamento dessa imagem e podendo ou não correlacionar essa informação com outros sensores, o robô conhece a sua localização e consegue reconhecer e localizar obstáculos. Contudo, algumas equipas complementam este sistema com um sistema de perspetiva, adicionando uma câmara frontal. Esta câmara é normalmente instalada no guarda-redes, permitindo detetar os objetos a uma maior distância e, dado que complementa o sistema omnidirecional, disponibilizando ao robô informação mais precisa. O uso de duas câmaras permite igualmente ao sistema de visão, a possibilidade de utilizar uma técnica de triangulação, de modo a obter a localização 3D de um dado objeto.

As características do sistema omnidirecional são determinadas, sobretudo, pela forma e a panorâmica do espelho utilizado. Tendo em conta diversos príncipios de imagem, este sistema pode ainda dividir-se em *single-viewpoint* (SVP) e *non-single-viewpoint* (NSVP). O sistema de visão omnidirecional SVP pode ser construído utilizando um espelho hiperbólico, parabólico ou elipsoidal. Já os espelhos cónicos, esféricos, isométrico horizontal ou isométrico vertical, permitem construir o sistema NSVP. Detalhes mais técnicos, características e teoria de imagem podem ser consultados em [Ben+01].

O espelho mais utilizado na competição RoboCup MSL é o espelho hiperbólico. A principal desvantagem em utilizar este espelho deve-se ao facto de a resolução diminuir gradualmente, quando a distância do robô aos objetos aumenta (sendo que a imagem refletida destes, diminui consideravelmente e em função desse mesmo aumento). Assim sendo, este espelho não é indicado para a deteção de objetos localizados a uma distância considerável. Em [Lim+01] é descrita a construção de um espelho constituído por um espelho isométrico horizontal, um espelho de curvatura constante e um espelho plano. A equipa NuBot desenhou igualmente um espelho panorâmico, que consiste num espelho hiperbólico, um espelho isométrico horizontal e um espelho isométrico vertical. A figura 2.8 ilustra o perfil da curvatura, bem como, uma fotografia do novo espelho. Tal como referido em [Lu+09; Lu+11], o sistema omnidirecional que utiliza este novo espelho mantém a resolução constante de objetos próximos ao robô e apresenta uma baixa distorção na imagem, para objetos distantes. Para além disso, melhora a definição da imagem refletida em toda a área adjacente ao robô, incluindo o próprio robô. A figura 2.9 ilustra o exemplo de uma imagem refletida pelo espelho desenvolvido, a qual permite constatar as melhorias descritas anteriormente. As circunferências assinaladas a vermelho pretendem assinalar os limites dos três subespelhos considerados, permitindo perceber visualmente, a forma como os mesmos influenciam diretamente a imagem refletida.



(a) Perfil da curva do espelho.

(b) Fotografia do espelho.

Figura 2.8: Espelho desenvolvido pela equipa NuBot (fonte [Lu+09]).

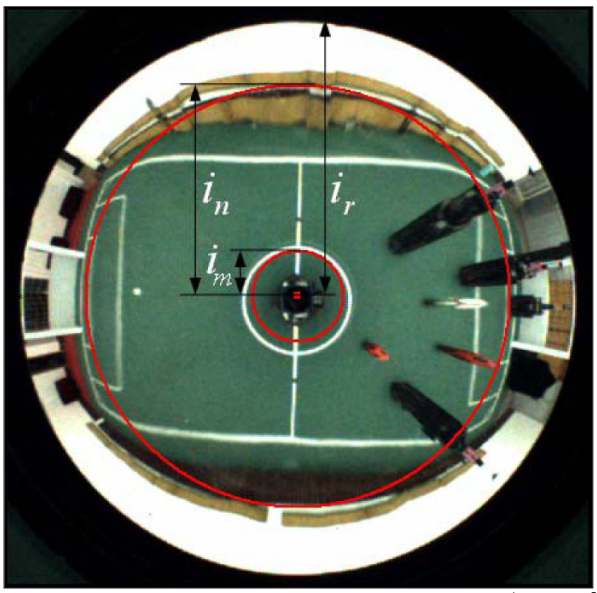


Figura 2.9: Imagem refletida pelo novo espelho (fonte [Lu+09]).

2.4.2 Descrição sumária de alguns sistemas de visão atuais

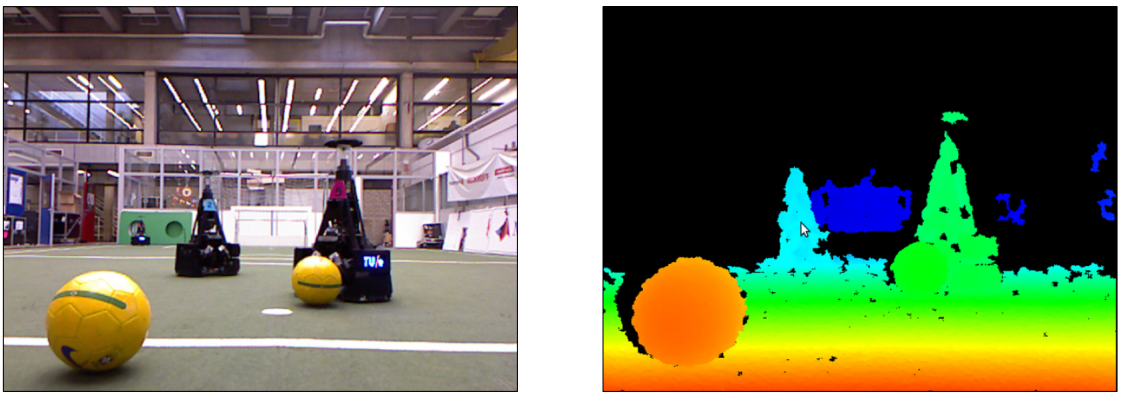
Seguidamente, iremos abordar os sistemas de visão específicos de algumas equipas participantes na competição RoboCup MSL. Apesar de existirem mais participantes, importa somente referir as equipas que mais se destacaram, quer pelo trabalho e conhecimento desenvolvidos, quer pelos prémios obtidos. Relativamente ao sistema de visão da equipa CAMBADA, dedicaremos inteiramente o capítulo 3 a explorar, de uma forma mais detalhada, a arquitetura do seu sistema de visão, pelo que, por forma a evitar repetições desnecessárias, iremos omitir o mesmo nesta abordagem.

Tech United

O sistema de visão presente nos robôs TURTLE (acrónimo de *Tech United RoboCup Team: Limited Edition*) da equipa Tech United[2], que representa a Universidade de Eindhoven[3], localizada no sul dos Países baixos, é constituído por duas câmaras, sendo uma frontal e outra omnidirecional [Aan+09; Bes+10; Kan+11; Hoo+12].

A câmara omnidirecional é composta por uma câmara *Prosilica GC750C*[4], operando a uma taxa de 30fps, para uma resolução de 752x480 píxeis, permitindo recolher informação em todo o redor do robô.

Com base no artigo [Sch+13], a câmara frontal foi alterada no último ano para o modelo *Kinect* desenvolvido pela *Microsoft* em parceria com a empresa *Prime Sense*. A sua função principal consiste em detetar e localizar a bola com precisão, quando esta se encontra no ar. Esta câmara possui conexão USB e foi idealizada para as plataformas *XBox 360* e *XBox One*, bem como PCs com o sistema operativo *Windows*, permitindo no caso concreto das consolas de jogos, interagir com estas sem a necessidade de um comando, utilizando uma interface natural (gestos e voz humanos). No que ao sistema de visão diz respeito, esta câmara incorpora dois sensores de imagem. O primeiro sensor permite capturar imagens com uma resolução até 1280x1024 píxeis em diversos formatos como UYVY. Contudo e por forma a aumentar a taxa de imagens adquiridas, as definições mais comuns contemplam RGB em 8 *bits*, com uma resolução 640x480 píxeis e uma taxa de 30fps. O segundo sensor permite adquirir imagens monocromáticas com profundidade, numa resolução 640x480 com 11 *bits* por píxel, permitindo distinguir 2048 níveis de sensibilidade. A profundidade de imagem é conseguida com apoio de um emissor laser infravermelho, permitindo detetar objetos colocados a diferentes distâncias, até um máximo de 6 m. Este segundo sensor possui um ângulo de visão horizontal de 57° e vertical de 43°. A figura 2.10 ilustra o exemplo de uma imagem adquirida por esta câmara.



(a) Imagem normal (b) Imagem de profundidade

Figura 2.10: Exemplo de imagem adquirida pela câmara Kinect (fonte: [Sch+13]).

A fim de detetar corretamente a bola, o algoritmo desenvolvido explora a segmentação de cor de modo a conseguir distinguir a bola do restante ambiente. Este método requer, necessariamente, que seja efetuada uma calibração de cor antes de cada competição. O

[2] `http://www.techunited.nl/en`, acedido em: 14/05/2013

[3] `http://www.tue.nl/en/`, acedido em: 14/05/2013

[4] `http://www.alliedvisiontec.com/us/products/cameras/gigabit-ethernet/prosilica-gc/gc750.html`, acedido em: 14/05/2013

conjunto de possíveis candidatos a bolas é refinado, correlacionando o seu tamanho na imagem adquirida com o tamanho esperado, considerando a distância detetada (associada ao nível de sensibilidade).

Segundo o mesmo artigo, somente o guarda-redes dispõe desta câmara, embora a intenção passe por alargar a sua inclusão aos demais robôs em campo. Tendo em conta os reduzidos ângulos de visão do segundo sensor e a limitação da resolução (de modo a obter taxas de aquisição superiores), o tempo de reação do guarda-redes é limitado. Esta inclusão permitirá que outros robôs possam igualmente detetar a bola (sobretudo quando em voo), auxiliando a informação do guarda-redes e permitindo que este possa reagir atempadamente.

Com base no artigo [Hoo+12], a anterior câmara frontal consistia numa câmara VC-4458[5], recolhendo imagens a uma taxa de 200fps, para uma resolução de 640x480 píxeis. A elevada taxa de aquisição de imagens implica uma elevada velocidade para o correto processamento de cada imagem adquirida. No entanto, esta câmara dispõe de uma unidade de processamento interna *Texas Instruments* a operar a 1 GHz, bem como 64 MB SDRAM (ou 128 MB em opção) disponíveis para guardar os dados relativos ao processamento de imagem. De acordo com o *datasheet* disponibilizado pelo fabricante, a unidade de processamento é responsável pelo processamento de imagem, tendo sido especialmente otimizada e suportando a execução paralela de diferentes processos durante a aquisição de imagem (por exemplo, processamento de uma obturação, exposição e leitura/transmissão dos dados).

Com base no mesmo artigo, a calibração da câmara é efetuada com recurso a uma ferramenta desenvolvida pela equipa, a qual permite selecionar a bola numa imagem e obter os valores HSV ideais. Estes valores são posteriormente enviados para a FPGA e o resultado do ajuste é imediato. A fim de evitar que estes valores se percam, por exemplo, por falta de energia, os mesmos são salvaguardados numa memória EEPROM incorporada na FPGA.

A localização de cada robô permite-lhe, não só orientar-se no campo de jogo, como inferir a posição e velocidade da bola. Esta localização é obtida correlacionando a informação adquirida pelo sistema de visão e pelos sensores de movimento presentes em cada robô. De modo a tornar esta informação o mais fiável possível, o algoritmo é baseado num filtro *Kalman* e um previsor, o qual permite estimar a posição com base na análise de duas amostras de imagem. O filtro visa, sobretudo, reduzir o erro proporcionado pelo reposicionamento dos robôs durante uma competição.

A utilização da FPGA e do processamento intrínseco a esta serão apresentados com maior detalhe na página 35, no âmbito do coprocessamento na competição RoboCup MSL.

MRL

O sistema de visão adotado pela equipa MRL[6], representante da Universidade de Qazvin[7], é constituído por duas câmaras uEye UI-2210-C, uma frontal e outra colocada no topo do robô, apontada ao espelho hiperbólico [Gho+11]. Ambas utilizam um protocolo de comunicação USB.

[5] `http://www.vision-components.com/en/products/smart-cameras/vc-optimum/`, acedido em: 14/05/2013

[6] `http://www.mrl.ir/`, acedido em 15/05/2013

[7] `http://www.qiau.ac.ir/en/`, acedido em 15/05/2013

O processamento das imagens inclui, primeiramente, a aplicação de um filtro mediano, por forma a reduzir o ruído na imagem, sendo posteriormente atribuído a cada píxel uma de quatro cores possíveis, com base numa tabela de pesquisa de cores (do inglês *Color Lookup Table - CLT*). Esta tabela é previamente preenchida por um processo que recebe imagens amostrais de diferentes bolas e, com base nas cores aprendidas, classifica o espaço de cores HSV em quatro cores. Esta CLT é usada pelo algoritmo de processamento de imagem a fim de detetar a bola, o campo e obstáculos, em tempo real, operando a uma taxa de 50fps.

A localicação do robô é conseguida correlacionando a informação das linhas brancas encontradas na imagem adquirida, com um modelo em memória. As linhas brancas são detetadas por análise de linhas radiais, identificando grupos de píxeis com a cor branca. As coordenadas polares são posteriormente convertidas em coordenadas cartesianas, convertendo igualmente o sistema de representação dos píxeis para um sistema métrico.

Com o objetivo de proceder ao reconhecimento da bola, primeiramente são segmentadas as cores da bola. Em seguida, o algoritmo procura identificar a bola através da sua forma, detetando segmentos circulares. A cada círculo identificado é aplicado um coeficiente de erro que permite saber o grau de similaridade desse círculo com um círculo perfeito, sendo escolhido no final, o círculo com menor coeficiente de erro. Eventuais segmentos de cor preta detetados são tratados como obstáculos. Esta análise é efetuada para a imagem recolhida pelo sistema omnidirecional.

A câmara frontal é utilizada no guarda-redes e permite calcular a altura da bola quando se encontra no ar, melhorando a precisão de deteção sobretudo quando esta se encontra distante do robô. Por forma a tornar o método mais robusto perante situações em que só uma parte da bola é visível nas imagens, são utilizados algoritmos de ajuste de dimensão da bola.

Water CN

O sistema de visão adotado pela equipa Water CN[8], representante da Universidade Ciência da Informação e Tecnologia de Beijing[9], é constituído por uma câmara *Point Grey*, com interface 1394 e uma resolução de 640x480, a uma taxa de 60fps [Hua+12].

A imagem é capturada utilizando o espaço de cores HSV, sendo que a distribuição de píxeis sobre a escala H, determina a cor dos objetos. Deste modo, a bola é vermelha, ao campo é atribuída a cor verde, as linhas delimitadoras de campo são representadas por azul claro e os obstáculos são identificados por roxo.

A localização do próprio robô é conseguida com base numa primeira correção à imagem adquirida, seguida de identificação de linhas brancas. Estas linhas são posteriormente comparadas com um modelo de marcação do campo de jogo, obtendo a posição relativa do robô. As coordenadas absolutas são calculadas com recurso à transformada da matriz ângulo e distância.

Brainstormers Tribots

O sistema de visão adotado pela equipa Brainstorms Tribots[10], representante da Universidade de Osnabrück[11], é constituído por duas câmaras [Haf+09]. A primeira é uma câmara

[8] `http://jdgcxy.bistu.edu.cn/robocup/index_robocup.asp?RoboCup_a=0`, acedido em 15/05/2013
[9] `http://english.bistu.edu.cn/`, China, acedido em 15/05/2013
[10] `http://ml.informatik.uni-freiburg.de/pub/dfg/doku.php?id=en:start`, acedido em 14/05/2013
[11] `http://www.uni-freiburg.de/start-en.html`, Alemanha, acedido em 14/05/2013

omnidirecional que recolhe imagens a uma taxa de 30fps e que permite reconhecer a bola, as linhas brancas delimitadoras, as balizas e os restantes robôs presentes em campo. A segunda é uma câmara de perspetiva, usada para reconhecer a bola de um ponto de vista diferente.

A posição tridimensional da bola é obtida recorrendo a uma razão geométrica, usando a posição da bola detetada em cada câmara. A velocidade vertical da bola é estimada com base numa componente adicional de um módulo que calcula o movimento da bola. O modelo de análise de movimento aplicado por este módulo considera igualmente, a gravidade e os saltos da bola.

1. RFC Stuttgart

O sistema de visão adotado pela equipa 1. RFC Stuttgart[12], representante da Universidade de Stuttgart[13], é constituído por uma câmara omnidirecional e uma de perspetiva [Zwe+09].

Ambas as câmaras são utilizadas para o reconhecimento da bola, sendo que com o sistema de perspetiva torna-se possível detetar a bola a uma distância de 3 m, 5 m ou 7 m. Contudo, utilizando apenas o sistema omnidirecional, a bola é corretamente detetada somente à distância de 3 m e 5 m, sendo que a uma distância de 7 m, a probabilidade de deteção correta diminui para 57 %. O método de deteção é dividido em duas fases. Na primeira, a imagem adquirida é analisada usando a transformada de *Hough*, pesquisando eventuais círculos com um determinado diâmetro mínimo. Na segunda fase são extraídas características de cada círculo (nomeadamente o histograma de cores) e comparadas com as características da bola calibrada.

NuBot

O sistema de visão adotado pela equipa NuBot[14], representante da *National University of Defense Technology*[15], é constituído por uma câmara omnidirecional e uma câmara de visão estéreo [Zen+13]. Esta última é produzida pela *Point Grey Research Company*, contendo duas câmaras que recolhem imagem e transmitem ao PC, através de uma interface IEEE-1394. Inicialmente é aplicado o método de *Canny* por forma a detetar a informação de contornos presentes na imagem panorâmica, obtendo 15 pontos relevantes do contorno como vetores de suporte para cada direção pré-definida [Lu+11]. Em seguida, é calculado o mapa de distâncias para cada ponto, utilizando a interpolação de *Lagrange*. Numa primeira fase, a interpolação funciona numa direção radial, obtendo um novo vetor de suporte em cada direção. Numa segunda fase é adotada uma direção rotacional, usando os vetores de suporte obtidos na primeira fase. A bola é detetada com base nas duas imagens adquiridas pela câmara de visão estéreo. Para tal, é reconstruído um cenário 3D, de modo a calcular as coordenadas 3D da bola. Uma vez que a bola se encontre no ar são guardados alguns pontos-coordenada, a partir dos quais se tenta criar uma parábola, procurando prever a trajetória de voo da bola.

[12] `http://robocup.informatik.uni-stuttgart.de/rfc/www/`, acedido em 15/05/2013
[13] `http://www.uni-stuttgart.de/home/index.en.html`, acedido em 15/05/2013
[14] `http://www.nubot.com.cn/homeen.htm`, acedido em 14/05/2013
[15] `http://english.nudt.edu.cn/`, China, acedido em: 14/05/2013

Por sua vez, o reconhecimento de uma bola arbitrária é conseguido em duas fases: um treino *offline* e um reconhecimento *online*. No treino *offline*, o sistema de visão omnidirecional captura várias imagens panorâmicas de diferentes bolas. Posteriormente, são extraídas várias subimagens, contendo ou não a bola, que serão usadas para definir amostras positivas e amostras negativas. Em seguida, para cada imagem são calculados vetores de características *Haar*[16] e combinados numa matriz, à qual será aplicado o algoritmo de aprendizagem *AdaBoost*[17]. Na fase de reconhecimento online são definidas uma série de áreas retangulares, ao longo de direções radiais sobre a imagem panorâmica. A dimensão destas áreas depende das características do sistema de visão omnidirecional, variando ao longo da direção radial. Em seguida, toda a imagem é pesquisada utilizando estas áreas retangulares, ao longo das direções radial e rotacional. Finalmente, um processo de classificação é aplicado, por forma a decidir se uma dada área contém a bola ou não.

De modo a que cada robô possa autonomamente autolocalizar-se foi implementado um algoritmo de otimização de localização, o qual correlaciona informação recolhida por sensores de movimento e pelo sistema de visão. Os sensores permitem obter nomeadamente a orientação do robô. No entanto e dado que este pode encontrar-se localizado em qualquer ponto do campo, são definidas 315 amostras de imagens de 315 posições distribuídas uniformemente. Com base na imagem adquirida pelo sistema de visão são calculadas funções de erro para cada amostra, sendo que um menor valor da função de erro representa uma maior probabilidade de a posição associada à amostra corresponder à localização atual do robô. Um algoritmo de otimização de localização permite posteriormente refinar as 5 amostras de menor erro, obtendo novos valores de erro. Se a nova função de erro de menor valor for inferior a um valor limite definido, a posição associada a essa amostra é definida como a posição do robô. Caso a função de erro de menor valor não seja inferior ao valor limite definido, o processo de localização é repetido.

Carpe Noctem

O sistema de visão adotado pela equipa Carpe Noctem[18], representante da Universidade de Kassel[19], é composto por uma câmara omnidirecional *PointGrey Flea2*[20], recolhendo imagens a uma taxa de 30fps, para uma resolução de 640x480 píxeis [Amm+12]. No momento, somente o guarda redes incorpora uma segunda câmara frontal, embora a equipa planeie a sua extensibilidade de inclusão aos demais robôs.

A atualização das definições de ganho da câmara omnidirecional é efetuada com recurso a um regulador de ganho, baseado em estimativas de iluminação na lente da câmara e em diversas áreas em redor do robô. Este processo é ainda complementado com informação recolhida pelo módulo de localização, de modo a permitir verificar se determinado ajuste é apropriado ou não.

Com o objetivo de evitar o tempo de processamento das tarefas de calibração para a segmentação de cor, a maioria dos cálculos são efetuados sobre imagens monocromáticas. A

[16] `http://en.wikipedia.org/wiki/Haar-like_features`, acedido em 14/05/2013
[17] `http://en.wikipedia.org/wiki/AdaBoost`, acedido em 14/05/2013
[18] `http://carpenoctem.das-lab.net/`, acedido em 14/05/2013
[19] `http://www.uni-kassel.de/uni/index.php?id=17`, acedido em 14/05/2013
[20] `http://www.ptgrey.com/products/flea2/flea2_firewire_camera.asp`, acedido em 14/05/2013

tarefa de deteção da bola é a exceção, a qual é baseada num vetor de cores contidas em regiões de interesse (do inglês *Regions Of Interest - ROI*). Neste método, valores altos representam cores "interessantes", enquanto valores baixos representam cores "menos interessantes" ou mesmo desprezáveis. Estes valores podem ser ajustados manualmente, especificando áreas de interesse no espaço de cores YUV, ou através de imagens (amostras) de bolas. Neste caso, é calculado um histograma de cores a partir destas imagens, sendo que os valores do histograma podem ser usados para calcular o vetor ROI. Com base nas cores mais importantes a avaliar, a bola é detetada calculando o gradiente de imagem para o vetor ROI. Esta abordagem permite reconhecer bolas com cores arbitrárias.

De modo a estimar a posição da bola num espaço 3D, é aplicado um método de teste a várias hipóteses de localização, correlacionando a informação recolhida pelas câmaras omnidirecional e frontal (apenas presente no guarda-redes).

Todo o processamento de imagem é realizado pelo denominado dispositivo de controlo, baseado num PC com um processador *Intel Core i7 Dual Core*.

Hibikino Musashi

O sistema de visão adotado pela equipa Hibikino Musashi[21], representante da Universidade de Kitakyushu, Japão, é composto por uma câmara omnidirecional e uma câmara monocular [Nas+11]. A câmara omnidirecional recolhe imagens no formato YUV, sendo posteriormente convertida para o espaço de cores HSV. O sistema de visão é baseado, simultaneamente, nos espaços de cores YUV e HSV. A câmara monocular encontra-se instalada somente no guarda-redes, possibilitando em conjunto com a câmara omnidirecional, reconhecer e localizar a bola quando esta se encontra no ar. Para tal, foi estabelecida uma expressão matemática que permite estimar o raio, por forma a correlacionar a informação recolhida pelas duas câmaras, sendo que a interseção de duas linhas indica a posição da bola.

Com o objetivo de reconhecer corretamente as diversas cores, mesmo considerando diferentes condições de luz, foi implementado um algoritmo usando SVM (*Support Vector Machine*). Este método permite classificar as cores e calcular um plano que maximize a margem entre as diversas classes de cores. O algoritmo é treinado com base nos valores das classes de cor YUV e a média dos valores da imagem YUV obtida. Após adquirir as imagens com base no espaço de cores YUV, estas são binarizadas, definindo um valor limite máximo e mínimo para a distribuição de cada classe, detetando deste modo cada cor.

As linhas do campo são reconhecidas com base na análise de linhas radiais, identificando pontos tipicamente brancos que as intercetem. A fim de estimar a localização do robô é utilizado o método de *Monte Carlo*, sobretudo devido à sua robustez e performance em tempo real. O método consiste em duas fases, sendo que na primeira procura-se prever a posição do robô com base numa distribuição condicional, enquanto que na segunda fase, procede-se à atualização desta distribuição tendo em atenção os dados recolhidos pelos sensores. O resultado final assemelha-se a uma concentração de partículas em torno do valor/ponto mais provável de localização. Dado que devido à simetria do campo, o método pode ainda assim falhar, é usado um sensor de direção em cada robô.

[21] `http://www.lsse.kyutech.ac.jp/~robocup/robocup2013.htm`, acedido em 14/05/2013

WinKIT

O sistema de visão omnidirecional adotado pela equipa WinKIT[22], que representa o Instituto de Tecnologia Kanazawa[23], Japão, é constituído por um espelho *Seiwa-Pro Panorama Eye*[24] e duas câmaras *PointGrey DragonFly2*[25] [Kom+09]. As câmaras utilizam uma interface IEEE1394a, possibilitando taxas de transferência até 400 Mbit/s.

Embora não sejam apresentados maiores detalhes sobre o seu sistema de visão, o artigo analisado salienta que o foco da sua investigação mais recente incidiu, sobretudo, no melhoramento dos métodos de autolocalização, deteção das balizas e como evitar obstáculos.

Para a autolocalização, o seu método baseia-se na aquisição da imagem, extraindo informação sobre as linhas brancas. A essa informação é aplicada a transformada de *Hough*, após a qual os robôs criam um modelo representativo, quer das linhas brancas e resultado da transformada, como do modelo conhecido de campo de jogo. Por fim, é correlacionada a informação de localização e navegação utilizando o método de *Monte Carlo*.

A fim de detetar a baliza, o robô cria uma imagem que contenha somente a informação de cor da estrutura da baliza. Em seguida verifica-se a existência de linhas perpendiculares ao campo de jogo, as quais poderão representar os postes da baliza. Posteriormente é averiguada a existência de uma linha transversal superior que interligue as linhas candidatas a postes e que possua a mesma cor da estrutura. O centro da baliza é decidido com base em apenas duas linhas (uma representativa de poste e uma representativa de barra superior).

No sentido de evitar a colisão com obstáculos foi adotado um método semelhante ao método potencial, o qual procura estimar o ângulo de direção do robô em função do objetivo que pretende alcançar e de eventuais obstáculos detetados que intercetem o seu caminho.

Consideremos a figura 2.11, que pretende representar graficamente e de uma forma analítica, a presença de um obstáculo que interceta o caminho do robô em relação a um dado objetivo. Consideremos ainda a representação no eixo X como sendo o plano de visão do robô em relação ao seu objetivo, sendo que, no caso, o próprio robô encontra-se posicionado frontalmente com o seu objetivo. No eixo Y encontra-se representada, em termos de valor relativo, a distância ao objetivo e ao obstáculo. Assim, verificamos a presença de um obstáculo (representado a azul) compreendido entre os ângulos $-40°$ e $25°$. O ponto mais próximo do obstáculo situa-se, sensivelmente, no ângulo $-15°$. Considerando a interceção das linhas que definem o ângulo do nosso objetivo (representado a rosa) com o obstáculo, é possível obter o ângulo de desvio que o robô necessita efetuar, por forma a contornar o obstáculo (representado a vermelho).

Um pormenor interessante relativamente a esta equipa prende-se com o facto de o trabalho desenvolvido no âmbito deste projeto ser inteiramente proveniente de jovens estudantes não graduados, funcionando o presente projeto, bem como toda a experiência e conhecimentos adquiridos, como um complemento à formação académica. Não obstante, os resultados alcançados permitiram um segundo lugar nas competição RoboCup, edições 2002, 2003 e 2004, mantendo igualmente uma classificação relativamente constante na competição *RoboCup Japan Open*, onde, na edição 2012, obteve o segundo lugar.

[22] `http://www2.kanazawa-it.ac.jp/robocup/`, acedido em: 11/07/2013
[23] `http://www.kanazawa-it.ac.jp/ekit/`, acedido em: 11/07/2013
[24] `http://www.seiwaopt.co.jp/seiwaopt.english/index.html`, acedido em: 11/07/2013
[25] `http://ww2.ptgrey.com/firewire/dragonfly-2`, acedido em: 11/07/2013

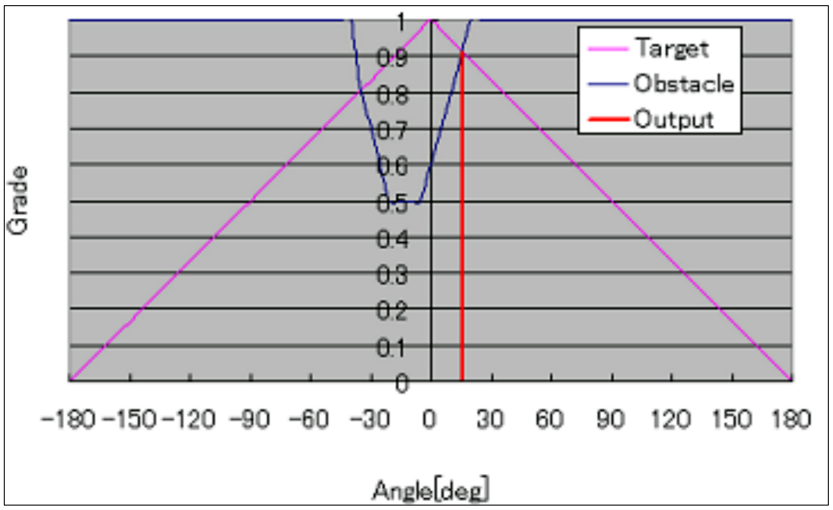


Figura 2.11: Relação entre um objetivo, um obstáculo e o ângulo de desvio do robô (fonte [Kom+09]).

2.5 Utilização de FPGAs para coprocessamento em sistemas de visão

As FPGAs podem ser encontradas em áreas tão diversificadas como: processamento de sinal digital, sinal de rádio definido por *software*, ASIC, prototipagem, imagiologia médica, visão computacional, reconhecimento de voz, criptografia, bioinformática, emulação de *hardware* de computador, radioastronomia, bem como uma emergente infinidade de outras áreas. Para este sucesso, contribuem a sua evolução em termos de arquitetura (área, capacidade e velocidade), aliada à eficiência e versatibilidade de implementação de soluções complexas, providenciando, atualmente, um baixo custo e um baixo consumo de energia, quando comparadas com outras soluções específicas (como DSPs).

Seguidamente, será apresentada uma pequena descrição sobre a arquitetura de uma FPGA e quais as suas principais vantagens, como plataforma de implementação de coprocessamento.

2.5.1 Conceitos gerais sobre FPGAs

Uma FPGA é, numa visão simples, uma matriz de blocos lógicos programáveis (*Configurable Logic Blocks - CLBs*), envolta por blocos de entrada/saída (*Input Output Blocks - IOBs*), interligados entre si, por ligações também elas programáveis. As figuras 2.12 e 2.13 ilustram, respetivamente, a estrutura interna de uma FPGA *Xilinx* e o pormenor das interligações programáveis.

A implementação de um circuito lógico sobre uma FPGA é feita através da distribuição desta lógica entre os blocos programáveis. Importa salientar que os atrasos resultantes são influenciados, em grande medida, pela distruibuição desta lógica e pela estrutura de encaminhamento. Isto quer dizer, que o desempenho de um circuito sobre uma FPGA depende grandemente da eficácia das ferramentas CAD (*Computer Aided Design*) utilizadas para a sua síntese e implementação.

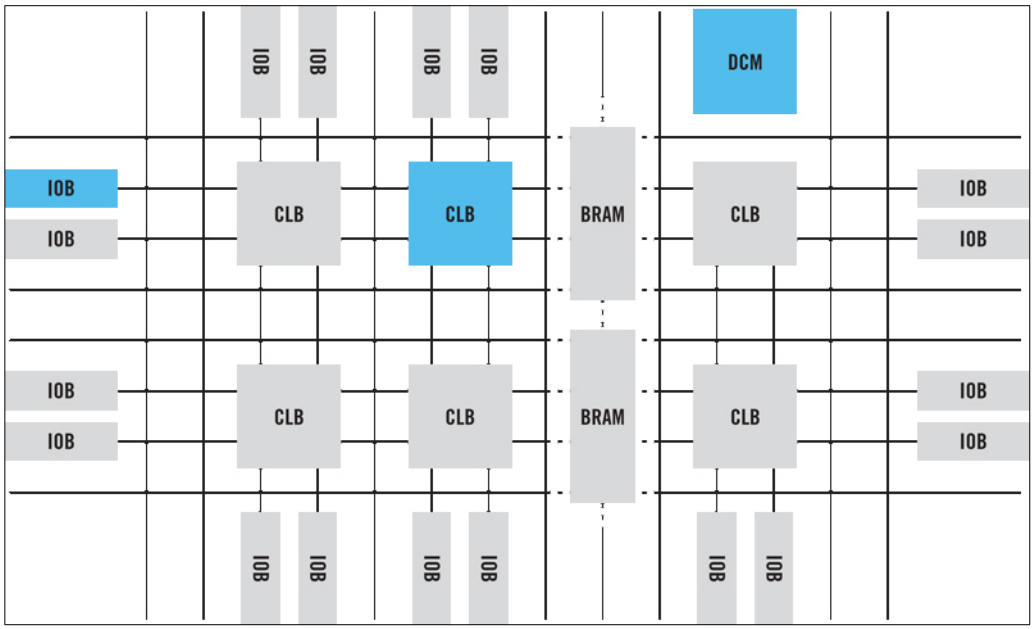


Figura 2.12: Estrutura de blocos interna de uma FPGA *Xilinx* (fonte [Ori]).

A sua evolução nos últimos anos permitiu sobretudo, aumentos de desempenho, densidade e capacidade de processamento, sendo que o consumo de energia diminuiu significativamente.

Atualmente, encontram-se disponíveis FPGAs com capacidade lógica equivalente a milhões de portas lógicas, sob a forma de recursos quer programáveis, quer fixos, tais como LUTs, memórias, macro-células do tipo DSP, processadores de uso geral (por exemplo, *MicroBlaze* e *PowerPc*), controladores para protocolos de comunicação (como *Ethernet*, *PCI Express*) e blocos de entrada/saída totalmente versáteis.

As FPGAs assumem-se como plataformas de implementação de sistemas reconfiguráveis, pois permitem que o comportamento de determinado sistema possa ser alterado, recorrendo somente a tranformações lógicas, não havendo necessidade de alterar fisicamente o seu circuito. Este tipo de alteração de comportamento pode ainda ser definido como estático ou dinâmico, consoante seja necessário interromper a operação do sistema ou não, a fim de aplicar a alteração pretendida.

2.5.2 Coprocessamento

Um processador de uso geral permite efetuar um conjunto de tarefas bastante diversificado e, na maior parte dos casos, difícil ou impossível de prever durante o seu projeto. Por este motivo, a extensibilidade e a flexibilidade destes sistemas são bastante elevadas, permitindo abranger a generalidade das situações. Contudo, mesmo considerando o elevado desempenho que os atuais processadores permitem, existem tarefas que dependem fundamentalmente de operações muito específicas ou demasiado exigentes em termos de processamento, como por exemplo, operações de cálculo em vírgula flutuante e processamento e renderização gráficos. Este tipo de tarefas impõem, normalmente, um elevado peso computacional sobre o processador, limitando ou mesmo atenuando a sua capacidade de aceitar e processar outras tarefas. Por outro lado, aspectos como a arquitetura de um processador de uso geral, o tipo e conjunto de instruções ou a *pipeline* de execução das mesmas, nem sempre permitem o melhor

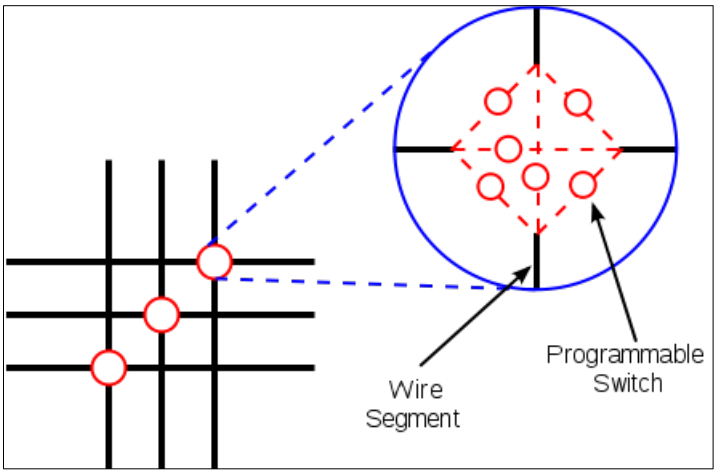


Figura 2.13: Pormenor de uma interligação programável (fonte [Wika]).

desempenho possível, introduzindo, por exemplo, latência ou ciclos de relógio inúteis ou desnecessários no que ao processamento se refere. Em geral e admitindo que as perdas ou mau desempenho não se devam a uma programação ou planeamento da aplicação ineficientes (em termos de *software*), a execução destas aplicações por núcleos de processamento dedicados e específicos, poderá representar um ganho de desempenho substancial.

Um coprocessador permite a especialização de determinadas funções, como por exemplo, operações em vírgula flutuante, processamento de sinal, gráficas e encriptação. Esta especialização implica que as funções sejam conhecidas durante a fase de projeto. Por outro lado, permite igualmente que possam ser introduzidas determinadas otimizações, que visem melhorar a sua eficiência e determinismo na realização das tarefas tidas como objetivo, nomeadamente ao nível da quantidade de recursos necessários, o desempenho ou a potência consumida. O motivo para a adoção de uma solução de coprocessamento visa sobretudo, reduzir a carga computacional de outros sistemas, podendo, na maioria das situações, reduzir igualmente e de forma significativa, o tempo de processamento.

As FPGAs, devido essencialmente ao paralelismo inerente aos recursos lógicos presentes na sua arquitetura, permitem elevadas taxas de processamento, usufruindo sobretudo dos mecanismos de paralelismo e *pipelining*, normalmente aplicáveis a este tipo de funções. Além disso e como já referido anteriormente, são bastante versáteis, permitindo a implementação de uma determinada solução, com um custo relativamente reduzido e com a vantagem de poderem ser futuramente reconfiguráveis.

2.5.3 Trabalhos relacionados

Devido sobretudo à sua evolução nos últimos anos, as FPGAs apresentam-se como uma plataforma de eleição para a implementação dos mais variados projetos. Atualmente, o seu domínio encontra-se em grande evolução, razão pela qual, existe um número cada vez maior de trabalhos e documentação disponíveis. Embora seja possível encontrar documentação diversificada relacionada com este assunto, esta pode ser classificada com alguma facilidade em:

- trabalhos teóricos

- desenvolvimento e teste de pequenos algoritmos de processamento de imagem

- projetos específicos a determinadas tarefas

Da documentação que se refere especialmente à implementação de algoritmos de processamento de imagem sobre FPGA, uma grande parte revela um nível de detalhe insuficiente sobre a solução adotada (ou algumas vezes, inexistente), não permitindo uma compreensão adequada, pelo que, esta documentação não será referida. Importa ainda salientar que a informação disponível é maioritariamente proveniente do meio académico.

Processamento de imagem em geral

Em [McB+03] é apresentada uma arquitetura paralela e programável que poderá ser utilizada para o pré-processamento de imagem em sistemas de visão embutidos. O artigo foca, essencialmente, a camada de pré-processamento de imagem que contém o canal DMA e um *array* paralelo de 16 elementos de processamento. O canal DMA permite o acesso à informação guardada em memória, segundo 24 modos de endereçamento escolhidos por forma a cobrir os algoritmos mais usados para processamento de imagem. A imagem é dividida pelo *array* de elementos de processamento, os quais, com base num algoritmo programável, processam o conjunto de píxeis atribuído. O algoritmo de divisão da imagem deteta eventuais sobreposições de regiões, por forma a reduzir o número de leituras da memória. Os resultados dos blocos de processamento que compõem a imagem pré-processada são guardados numa zona de memória acessível ao sistema de processamento externo, o qual, depois de receber um sinal de processamento concluído, lê a imagem e continua o processamento. Cada elemento de processamento do *array* pode ser visto como um pequeno DSP aceitando 16 *bits* de entrada e produzindo resultados em 32 *bits*. Estes elementos recebem o tipo de operação através de um controlador principal.

Em [Gon+09] e [Bra+06] são descritas abordagens muito semelhantes. Na primeira são somente apresentadas mais considerações e mais detalhes (utilizando o mesmo tipo de abordagem), enquanto que a segunda sugere uma arquitetura ligeiramente diferente mas que assenta essencialmente nos mesmos aspectos (imagem em memória, algoritmos de processamento de imagem genéricos e controlo via PC).

A principal desvantagem neste tipo de abordagens reside no facto de ser necessário manter em memória a *frame* completa original, assim como blocos repetidos desta ao longo dos vários elementos de processamento. Para imagens com maior resolução e FPGAs com menores recursos, esta abordagem pode tornar-se problemática. De igual modo, os algoritmos abordados e presentes em cada elemento de processamento baseiam-se sobretudo em operações e transformações de píxeis, sem que, a um nível global, haja uma consciência que permita correlacionar os resultados dos vários elementos.

Em [Tre+07; Tre+08] é descrita a implementação de um algoritmo de análise de bolhas de cor utilizando somente uma passagem.

O algoritmo sugerido pressupõe que os dados possam ser diretamente capturados pela câmara, seja efetuado o processamento da imagem e posteriormente, enviados os resultados ao computador principal. Para tal, os dados adquiridos pela câmara são inicialmente guardados na memória SDRAM. O algoritmo de análise de bolhas de cor procura adotar uma técnica

de *raster scan*, processando em tempo real e individualmente todos os píxeis, de forma a calcular as características dos objetos. Por forma a facilitar o processamento de análise, os dados são lidos da memória e comprimidos utilizando o método de codificação RLE. A cada sequência resultante da compressão é atribuído um número de objeto e são calculadas, parcialmente, as características para o objeto correspondente. Um mecanismo de interação sobre os objetos visa pesquisar eventuais objetos que não tenham sido alvo de atualização recente, disponibilizando-os para envio.

Os resultados apresentados sugerem que o algoritmo possa lidar com imagens contendo um elevado ruído, mantendo um *frame rate* elevado. No entanto, a imagem a analisar é convertida para um formato monocromático. Outro problema apresentado prende-se com a relação entre os ciclos mínimos de processamento de uma sequência comprimida (em média 6 ciclos) e a compressão da imagem original. Relembrando que o RLE depende da imagem a comprimir, uma imagem com muitas variações origina uma baixa taxa de compressão, produzindo mais resultados (que o módulo de análise poderá não conseguir processar em tempo útil). Contudo, esta abordagem, quando comparada com as anteriores, possui claramente a vantagem de utilizar menos recursos, dado que o processamento é efetuado em modo *raster scan*.

Em [Joh+04] são enunciadas algumas considerações teóricas sobre a implementação de algoritmos de processamento de imagem sobre FPGAs. O artigo pretende somente apresentar algumas considerações, como por exemplo, os requisitos de tempo que determinadas abordagens poderão implicar, tendo em conta o tamanho da imagem, os recursos ocupados pela abordagem e os tempos de comunicação, quando o sistema se destina a coprocessamento. Embora as considerações feitas sejam sobretudo superficiais, as conclusões encontram-se de acordo com a análise efetuada a cada uma das abordagens descritas anteriormente.

Em [Boc+09] foi apresentado um trabalho que procurava aplicar a deteção de bolhas de cor sobre imagens adquiridas por *streaming* de um leitor de DVD. Contudo, são relatados vários problemas que impuseram que a solução só pudesse ser testada com uma imagem carregada em memória. Para além disso, a imagem é convertida para formato monocromático. O algoritmo de análise é baseado na luminosidade dos píxeis e um valor limite definido. Na prática, o sistema é configurado para pesquisar um certo número de bolhas, para as quais os píxeis se encontrem acima de um determinado valor.

Esta solução pressupõe que se conheçam algumas características do tipo de dados a processar (como a luminosidade ou o número de bolhas/objetos a encontrar). Contudo, num cenário mais genérico não é aceitável limitar a luminosidade dos píxeis da imagem, pois a aquisição desta depende, sobretudo, das características do *hardware* (por exemplo, qualidade da câmara, lentes e espelho), bem como de outros fatores como a luz natural e o tipo de cores. De igual modo, a restrição do número de bolhas poderá limitar negativamente o número de objetos detetados num determinado cenário, perante condições imprevisíveis (por exemplo, variações de luminosidade e número de objetos não constante).

Em [Chi+11] é descrita a implementação de um algoritmo de deteção de contornos utilizando o operador de *Sobel*. Esta solução pressupõe a conversão da imagem para um ficheiro texto e o processo inverso, utilizando *Matlab*. Entre as conversões, os algoritmos de deteção de contornos e segmentação serão executados sobre uma FPGA. A solução adotada para o operador de *Sobel* consiste, essencialmente, em dividir o processamento por quatro núcleos

que calculam a convolução, explorando desta forma o paralelismo de processamento. O resultado deste operador alimenta, por sua vez, um módulo de segmentação binária que atribui a cada píxel o valor 0 (zero) ou 255. A segmentação é baseada no histograma da imagem original, identificando de forma automática os valores máximos e definindo valores limite para as regiões a detetar.

Contudo, o tamanho da região varia de acordo com a imagem e deve ser definido pelo utilizador.

Em [Sud+11] é apresentado um estudo comparativo de quatro implementações distintas de algoritmos de deteção de contornos (operador de *Sobel*, *Robert*, *Prewitt* e *Compass*, este último utilizando várias máscaras). Foram realizados testes com resoluções até 1024x768, utilizando uma porta DVI (entrada e saída). Os resultados apresentados permitem perceber que o operador *Compass* com a máscara *Kirsch* detetou o maior número de píxeis de contorno, embora a sua frequência máxima, bem como a utilização de recursos, sejam as piores do comparativo. A implementação pode ser extensível a imagens de cor e a taxas maiores.

Em [Ven+11] é descrita uma implementação de um algoritmo de deteção de contornos, utilizando o método de *Canny* e explorando o mecanismo de *pipelining*. A arquitetura da implementação foi desenvolvida utilizando uma arquitetura reconfigurável e *hardware* modelado, utilizando *Handle-C*. Esta solução permite obter um píxel de contorno a cada ciclo de relógio, com uma frequência de relógio de 16 MHz. O mesmo algoritmo foi implementado utilizando *Visual C++ v6.0* e a sua execução sobre um Pentium III 1.3 GHz revelou um tempo de processamento de 47 ms (contra 4.2 ms sobre a FPGA).

Em [Ẃ+11] é apresentado um projeto de um sistema de vigilância e deteção de trânsito implementado sobre FPGA. O seu objetivo visa, essencialmente, detetar objetos em movimento pela análise de imagens recolhidas por uma câmara (monocromática), identificando algumas características como o número de veículos, a sua direção e a sua velocidade. O algoritmo implementado assegura sobretudo, uma eficiência de *frame rate*, utilizando o mecanismo de *pipelining*. O algoritmo de segmentação adotado converte a imagem original para um formato binário, a qual é posteriormente dividida e analisada por blocos que aplicam o algoritmo de deteção de contornos (temporal e espacialmente). O algoritmo é ainda complementado pela transformada de *Hough*, de forma a melhorar a deteção de contornos dos objetos.

Competição RoboCup MSL

Como já referido, uma dificuldade comum às equipas participantes é o tempo de processamento. Assim, importa que os processos de análise e captura de imagem pela câmara possam ser executados o mais rapidamente possível, permitindo que posteriormente, os processos de alto nível disponham de mais tempo para tomar uma decisão, possam ser mais complexos e, eventualmente, mais completos. Com base nesta ideia, algumas equipas estudam já a possibilidade de implementar os algoritmos de análise e processamento de imagem, utilizando processadores dedicados, como DSPs (*Digital Signal Processor*) ou processadores desenvolvidos em FPGAs.

Em [Kan+11], a equipa Tech United apresentou uma solução para o problema de deteção de uma bola arbitrária em tempo real. O algoritmo descrito procura detetar a bola com base

na sua forma física. À semelhança do sistema de visão humano, que dispõe de células especiais bastante sensíveis a identificar contornos de formas e objetos presentes numa imagem, o algoritmo procura píxeis na vizinhança de um ponto que sejam "sensíveis" numa determinada direção (no caso da bola, considera-se o centro). A sensibilidade, neste caso, é modelada pela derivada em direção ao centro. A soma de valores obtidos para os píxeis pertencentes a um certo circulo de raio r, permite obter a noção de confiança de que para um determinado píxel P_{ij}, existe um circulo de raio r presente. Após repetir o método para diferentes valores de raio, é atribuído o grau de confiança ao píxel, assim como, o valor do melhor raio encontrado. Este método é repetido para cada píxel, sendo que, no final, o píxel que registar o valor de confiança mais alto representa o centro da bola na imagem. Apesar do algoritmo ser simples, a sua implementação é computacionalmente complexa, exigindo um tempo de processamento considerável. Por forma a contornar este problema, esta solução foi implementada sobre *hardware* reconfigurável, mais concretamente sobre uma *Xilinx Virtex5 SX50T*. Contudo e apesar da intenção de expandir esta solução aos restantes robôs da equipa, tanto quanto é possível perceber pelos artigos mais recentes, [Hoo+12] e [Sch+13], esta solução parece continuar implementada somente no guarda-redes.

Esta implementação permite tirar partido da otimização do envio de dados e da paralelização de blocos de processamento. A título exemplificativo, tanto o gradiente da imagem como a derivada em direção ao centro são calculados sobre o *streaming* de dados, recorrendo a alguns blocos de Multiplicação&Adição e apenas alguns dados em memória (no caso, linhas de imagem). Isto permite que após um número reduzido de ciclos de relógio, a posição final da bola e o raio sejam conhecidos. Uma solução implementada num PC implicaria guardar uma frame completa e mais tempo de processamento.

De igual modo, também a equipa Brainstormers Tribots manifestou o seu interesse em desenvolver uma camada de baixo nível, contendo os algoritmos de processamento de imagem diretamente sobre *hardware* [Haf+09]. Mais uma vez, o objetivo consiste em, por um lado, aumentar a eficiência de processamento e, por outro, reduzir significativamente esse mesmo tempo de processamento. Os algoritmos serão especificados em linguagem VHDL, sendo executados sobre um SoC (*System on Chip*) contido numa FPGA. Este sistema será desenvolvido numa primeira fase para os robôs humanóides, com forte possibilidade de, posteriormente, ser usado também nos robôs presentes na competição RoboCup MSL.

2.6 Conclusão

Embora ao longo de todo o capítulo tenham sido apresentadas considerações resultantes da análise aos artigos e trabalhos apresentados (considerações essas, que o leitor é convidado a ler), importa sintetizar algumas observações de foro geral.

Tal como supra citado, o sistema de visão assume uma especial importância no âmbito da competição RoboCup MSL. Para além da elevada importância, acrescem ainda o espírito de competição e o gosto e necessidade de inovação presentes em cada uma das diversas equipas envolvidas, razões que justificam o esforço movido e contínuo na investigação e teste de novos métodos. Como vimos, este mesmo sistema deve ser o mais robusto possível, por forma a permitir ao robô perceber visualmente e o mais fielmente possível, o ambiente que o rodeia,

considerando e lidando eficazmente com as adversidades desse mesmo ambiente. Por outro lado, importa que todo o processamento associado à informação desse meio seja efetuado o mais rapidamente possível, permitindo tomar decisões em tempo útil face aos diversos eventos possíveis. Contudo, esta é precisamente a principal dificuldade sentida pelas equipas e uma das especificidades que mais atenção tem merecido.

Com exceção da equipa *Tech United*, observamos que a generalidade das equipas aqui analisadas, opta por delegar a execução das diversas tarefas associadas ao processamento de imagem, ao processador principal. Contudo e como se depreende, esta decisão compromete o desempenho geral do sistema, razão pela qual alguns métodos e algoritmos são "simplificados", operando tipicamente sobre conjuntos de dados monocromáticos ou assumindo outros compromissos no seu processamento. Na verdade e como se pode observar, comum a todas as equipas, parece ser limitação de resolução da imagem adquirida, de modo a diminuir o conjunto de dados a processar. Contudo, este aspecto deriva em alguns casos da própria limitação da câmara utilizada, pelo que requer algum cuidado e discernimento quando considerado.

Dependendo do *hardware* em questão e do sistema de visão adotado, algumas equipas procuram delegar uma pequena parte do processamento de imagem ao microprocessador da câmara (quando disponível). A equipa *NuBot* destaca-se em parte, pela capacidade e preocupação demonstradas em desenvolver um sistema previsivo e não somente reativo a uma situação (relembramos que o algoritmo implementado para deteção de bola procura antever a posição final da bola, quando esta se encontra em voo). A equipa *Tech United* é notoriamente a equipa que mais tarefas conseguiu delegar ao microprocessador da câmara, sobretudo, devido à possibilidade de este permitir ser programado e personalizado para determinadas tarefas. Assim, parte das tarefas computacionalmente intensivas são libertadas do processador principal, permitindo que este se possa dedicar a outras tarefas (incluindo necessariamente, a gestão da informação de controlo e envio de dados para/do microprocessador). No entanto, não podemos esquecer que, ainda que versátil e com um grau de flexibilidade maior do que as restantes implementações, a capacidade de programação e desempenho deste microprocessador poderá ser um fator limitativo, quer na inclusão de mais tarefas de processamento, quer no próprio desempenho de operações mais complexas.

A computação reconfigurável surge como uma alternativa mais viável, tanto em possível desempenho, como na diversidade ou infinidade de operações que podem ser implementadas. Naturalmente que o desempenho depende, significativamente, da possibilidade de os algoritmos implementados para cada uma das tarefas em questão, conseguir tirar proveito dos mecanismos de paralelismo e *pipelining*. Contudo e como já referido, a generalidade dos algoritmos associados ao processamento de imagem gozam desta propriedade.

Ainda assim, apesar da evidente vantagem, a complexidade e a quantidade de trabalhos existentes que visam incorporar algoritmos ou um processamento mais complexo de imagem, revela uma lacuna. Dos trabalhos analisados, evidenciam-se essencialmente duas abordagens ao problema, as quais poderão ser tidas em conta para este trabalho. A primeira, e talvez a mais comum, visa implementar um pequeno núcleo de processamento de imagem, com capacidade para realizar um número limitado de operações, sobre um conjunto de dados igualmente limitado. O paralelismo é depois explorado, replicando este núcleo, tanto quanto necessário, por forma a conseguir processar pequenos subconjuntos de uma imagem. No entanto e como se depreende, esta abordagem necessita salvaguardar em memória, quer a imagem completa

(e eventualmente, pequenos blocos desta repetidos), quer os pequenos conjuntos de dados que resultam dos diversos tipos de processamento. Para além de um cuidado especial no acesso limitado ao recurso partilhado (a memória), existe ainda o problema do fluxo de processamento de dados. Dependendo da plataforma adotada para a prototipagem ou teste da solução (nomeadamente, dos recursos disponibilizados por esta), do conjunto e dimensão dos dados a processar e, necessariamente, do tipo de operações, poderão surgir duas situações: todo o processamento pode ser realizado de forma sequencial e progressiva, sendo que em cada ciclo de relógio, todos os núcleos de processamento conseguem processar um conjunto de dados diferente e produzir, quer individualmente, quer no seu conjunto, um resultado; ou, não havendo possibilidade de replicar o número suficiente de núcleos, ser necessário compartilhar um mesmo conjunto de núcleos por diferentes etapas de processamento. A primeira situação, apesar do elevado número de recursos utilizado, permite (em teoria) processar e obter um resultado a cada ciclo de relógio. A segunda situação, embora mais comedida nos recursos utilizados, introduz necessariamente latência no processamento dos dados, podendo obrigar a descartar, ainda que ocasionalmente, algumas *frames* de imagem.

A segunda abordagem, procura (tanto quanto possível), processar os dados em modo *raster scan*, ou seja, os dados são processados assim que disponíveis. Quando comparada com a primeira abordagem, esta permite reduzir significativamente, o total de recursos utilizados, sobretudo, porque o volume de dados a guardar entre diferentes tipos de processamento é, tipicamente, menor. Contudo, a especificidade ou complexidade inerentes a uma determinada tarefa, poderá limitar ou impedir a adoção desta abordagem.

Independentemente da abordagem e tal como supra citado, o conjunto de tarefas de processamento de imagem implementado nos diferentes trabalhos é normalmente muito reduzido (quando comparado com a lista de possíveis tarefas relacionadas com o objetivo deste trabalho). Tanto quanto seja possível e numa perspetiva de reduzir o número de recursos utilizados, procuraremos adotar esta abordagem para o maior número de tarefas computacionalmente intensivas e passíveis de serem implementadas sobre *hardware* reconfigurável.

Capítulo 3

Sistema atual de visão da equipa CAMBADA

3.1 Introdução

Neste capítulo procuraremos apresentar não só a equipa CAMBADA, como o seu trabalho e o seu contributo para a competição RoboCup MSL. Seguidamente, será apresentada a arquitetura geral dos seus robôs, focando sobretudo, especial atenção na arquitetura do sistema de visão e detalhando os diversos métodos utilizados, nomeadamente, na sua calibração e deteção de objetos.

3.2 A equipa CAMBADA

A CAMBADA (*Cooperative Autonomous Mobile roBots with Avanced Distributed Architecture*) é a equipa que representa a Universidade de Aveiro, mais concretamente, a unidade de investigação "Instituto de Engenharia Eletrónica e Telemática de Aveiro"(IEETA), na Liga de Robôs Médios do RoboCup.

O projeto teve início em outubro de 2003 e, desde então, tem vindo a participar em diversas competições robóticas, das quais importa salientar, a nível internacional, a competição RoboCup (edições 2004 a 2012), DutchOpen 2006, GermanOpen 2010 e, a nível nacional, a competição Robótica (edições 2004 a 2013).

A sua classificação e os prémios obtidos são um sinal inequívoco do enorme esforço, dedicação e investigação mantidos por todos os elementos envolvidos neste projeto. Desses prémios poderemos destacar o primeiro lugar na competição nacional Robótica (edições 2007 a 2012), quinto lugar na competição mundial RoboCup 2007, primeiro lugar na RoboCup 2008, segundo lugar no RoboCup German Open 2010, terceiro lugar no RoboCup 2009, 2010 e 2011, terceiro lugar na DutchOpen 2012 e quarto na RoboCup 2012. Recentemente alcançou o segundo lugar na competição portuguesa de robótica, Robótica 2013, revalidando o título de campeão nacional pelo sétimo ano consecutivo.

Este projeto envolve pessoas das mais diversas áreas, que contribuem para o desenvolvimento da estrutura mecânica do robô, o seu *hardware*, arquitetura e controladores, bem como o desenvolvimento de *software* em áreas como o processamento e análise de imagens, sensores,

raciocínio e controlo, cooperação baseada numa Base de Dados em Tempo Real (*Real-Time DataBase - RTDB*), a comunicação entre robôs e o desenvolvimento de uma estação base eficiente (*basestation*).

Os robôs operam em tempo real, num ambiente altamente dinâmico, multiobjectivo, parcialmente cooperativo e parcialmente adverso, sempre de forma autónoma, comunicando exclusivamente entre si. Do seu ponto de vista, o campo de jogo, durante uma competição, é tal como descrito anteriormente, percebido como um cenário altamente dinâmico, onde os restantes elementos da equipa, os adversários e a bola se deslocam com relativa velocidade e de uma forma imprevisível. O ambiente em si é igualmente adverso, já que condições como a luz controlada ou o fácil reconhecimento de objetos, com recurso à codificação de cores, deixaram de ser garantidas.

Assim, o sistema de visão de cada robô é considerado de extrema importância, pois é a sua única forma de perceber o ambiente que o rodeia. Este sistema necessita ser robusto, por forma a detetar eficazmente, e de uma forma precisa, objetos e áreas de interesse e a sua respetiva localização relativamente ao robô, quer perante condições ideais, quer perante condições mais adversas. Tudo isto, claro, em tempo real, havendo necessidade de processar os dados numa pequena fração de tempo, por forma a permitir ao sistema cognitivo do robô, responder adequadamente e em tempo considerado útil.

3.3 Arquitetura geral dos robôs

Como referido no capítulo 1, a arquitetura adotada para o desenvolvimento dos robôs baseia-se num paradigma biomórfico, isto é, a construção dos robôs é inspirada nos princípios dos sistemas biológicos presentes nos seres humanos ou animais. Por analogia, cada robô possui uma unidade de processamento principal (figura 3.1) que desempenha o papel de cérebro, sendo responsável pela coordenação e comportamento do robô. Esta unidade de processamento principal lida diretamente com os sensores, que exigem uma elevada largura de banda (como o sistema de visão) e coordena a comunicação externa com os outros robôs. Esta unidade processa ainda, a informação recebida pelo sistema distribuído de baixo nível de deteção (o sistema nervoso), enviando comandos, por forma a controlar o comportamento e atitude do robô. O "sistema nervoso" é composto por vários microcontroladores interligados entre si através de uma rede CAN. Apesar da complexidade subjacente, podemos ver este sistema como o responsável por quatro funções básicas: movimento, odometria (método que permite estimar a posição do robô), chuto e monitorização do sistema (por exemplo, o nível das baterias e o estado de cada microcontrolador).

De um ponto de vista mais técnico, esta arquitetura foi pensada e implementada de uma forma segmentada por camadas e modular. A construção de cada robô assenta numa estrutura cilíndrica (com 485mm de diâmetro) que acopla os diversos componentes. A camada inferior incorpora os motores, rodas, baterias e sistema de chuto eletromecânico. A segunda camada consiste nos dispositivos de controlo eletrónicos, sendo que a terceira contém a unidade de processamento principal (computador). O sistema de visão consiste numa câmara com interface *firewire*, instalada verticalmente no topo do robô e apontada a um espelho hiperbólico que reflecte 360° da área envolvente ao robô. A arquitetura do sistema de processamento é

Figura 3.1: Detalhe da unidade de processamento principal presente em cada robô da equipa CAMBADA (fonte: [Nev+10a]).

baseada num modelo distribuído, sendo que a maioria das tarefas elementares são processadas por pequenos microcontroladores, interligados entre si pela rede CAN. As tarefas de controlo de alto nível são responsabilidade do computador, por sua vez baseado no sistema operativo Linux. A comunicação entre robôs assenta numa rede *wireless*, utilizando o protocolo IEEE 802.11x. O sistema de deteção baseia-se numa base de dados em tempo real (RTDB), atualizada e replicada por todos os jogadores, servindo como um ponto de partilha da visão do mundo.

3.4 Arquitetura do sistema de visão

O sistema de visão dos robôs baseia-se num sistema catadióptrico, frequentemente denominado como sistema de visão omnidirecional. Este sistema é constituído, tal como referido anteriormente, por uma câmara de vídeo digital direcionada ao espelho hiperbólico (figura 3.2). A câmara utilizada (Point Grey Flea 2 [1], FL2-08S2C com um sensor 1/3"CCD Sony ICX204) permite capturar imagens, com uma resolução até 1024x768 píxeis, em vários formatos de imagem, nomeadamente RGB, YUV 4:1:1, YUV 4:2:2 ou YUV 4:4:4. O espelho hiperbólico foi desenvolvido por IAIS Fraunhofer Gesellschaft [2] (FhG-Ais).

O uso de um sistema de visão omnidirecional permite ao robô estender o seu campo visual a 360° sobre um eixo de rotação vertical, sem a necessidade de se mover ou mover a sua câmara. O sistema catadióptrico, por sua vez, assegura uma perceção integrada de todos os principais objetos na área que circunda o robô, permitindo um maior grau de manobrabilidade. No entanto, para objetos ou áreas de interesse mais distantes do robô, este sistema implica uma maior degradação na resolução, quando comparado com os sistemas não-isotrópicos.

[1] `http://www.ptgrey.com/products/flea2/`, acedido em: 08/05/2013
[2] `http://www.iais.fraunhofer.de/`, acedido em: 08/05/2013

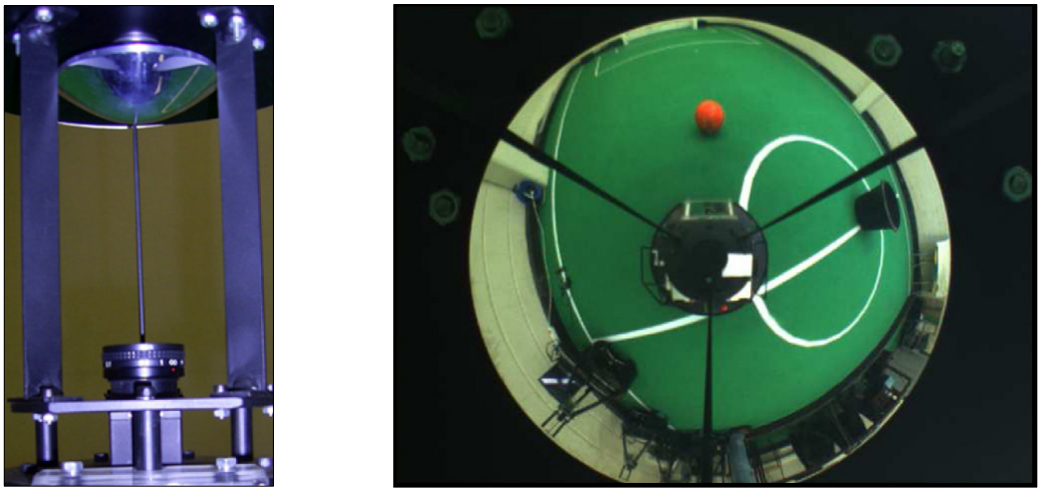


Figura 3.2: À esquerda, detalhe do sistema de visão adotado pela equipa CAMBADA, ilustrando montagem da câmara e espelho hiperbólico; à direita, exemplo de uma imagem adquirida pelo mesmo sistema (fonte: [Nev+10a]).

3.4.1 Calibração do sistema de visão

Como se depreende, por forma a que cada robô possa perceber e modelar adequadamente o ambiente que o envolve, é necessário calibrar corretamente o sistema de visão. Esta tarefa inclui a calibração dos parâmetros intrínsecos à câmara, o correto cálculo do mapa inverso de distância, a deteção do espelho e centro do robô, bem como das regiões da imagem a serem processadas. Para além de ser necessário realizar esta calibração de uma forma quase contínua, já que o ambiente de competição é, como anteriormente referido, altamente dinâmico quanto às suas características, é importante observar que, sempre que surge a necessidade de substituir *hardware* integrante ao sistema de visão (como a câmara ou o espelho), torna-se necessário calibrar e ajustar, de uma forma robusta, todo o sistema.

Auto-calibração dos parâmetros da câmara

Futuramente, um dos objetivos da competição MSL pressupõe que os robôs possam competir sob condições de luz natural e em espaços abertos. Este objetivo introduz novos desafios, uma vez que (e somente a título de exemplo), em espaços abertos, a iluminação depende de aspetos como o movimento do sol, a obstrução ou a interferência de nuvens e outros obstáculos. Isto implica que os robôs necessitem de ajustar, em tempo real, parâmetros da câmara e tabelas de segmentação de cor, por forma a que possam adaptar-se às novas condições.

Em [Nev+10a] é descrita uma proposta de abordagem a este problema, implementada com sucesso na equipa CAMBADA. O algoritmo proposto não necessita de qualquer interação humana, podendo, desta forma, ser executado de modo contínuo, durante uma competição e permitindo configurar autonomamente, os principais parâmetros da câmara (tais como, a exposição, o equilíbrio de brancos, o ganho e a luminosidade). Para tal, o algoritmo utiliza um histograma de intensidades de cor e uma área preta e branca, cuja localização é conhecida. A área branca permite configurar o equilíbrio de brancos, enquanto a área preta permite calibrar a luminosidade da imagem, sendo o histograma, por sua vez, utilizado para ajustar o ganho e

a exposição. Estes parâmetros são configurados de forma iterativa e individualmente, até que todos os parâmetros tenham convergido num valor. Detalhes adicionais sobre este algoritmo podem ser encontrados em [Nev+09].

Calibração do mapa inverso de distâncias

Para uma correta perceção do ambiente envolvente ao robô, nomeadamente a localização de obstáculos e áreas de interesse, é necessário converter o plano de visão obtido pela câmara em coordenadas reais do ambiente onde se encontra o robô, usando este último como centro deste sistema. Por forma a simplificar esta conversão, uma solução frequentemente adotada consiste em criar uma configuração mecânico-geométrica, que permite assegurar uma solução simétrica, através da aproximação de um ponto de vista único [Bak+99]. No entanto, esta abordagem requer um alinhamento rigoroso, sobretudo, entre o foco do espelho, o foco da lente e o centro do sensor de imagem. O sensor deve, ainda, estar perfeitamente paralelo ao pavimento e normal ao eixo do espelho, com o foco do espelho coincidente com a lente da câmara. Como se compreende, uma vez que esta abordagem depende de uma elevada precisão de alinhamento, qualquer pequeno desvio será suficiente para gerar erros de cálculo ou conversão.

A solução adotada pela CAMBADA, a qual pode ser consultada com maior detalhe em [Cun+07a], permite compensar um possível desalinhamento que resulte de um mau ajuste ou do uso de câmaras de video de baixo custo. O método descrito permite, igualmente, extrair grande parte dos parâmetros apenas da imagem adquirida, podendo ser usado para auto-calibração. Para tal, recorre-se a duas ferramentas: a primeira cria um mapeamento inverso da imagem adquirida, relativamente ao mapa de distância real. Posteriormente, um algoritmo integra dados da imagem em áreas exteriores ao mapeamento do plano físico, produzindo uma vista planar superior, na qual é possível observar, visualmente, o paralelismo de linhas e assimetrias circulares. A segunda ferramenta gera uma grelha visual com 0.5 m de distância entre linhas, sobreposta à imagem original, permitindo, imediatamente, perceber se existe necessidade de proceder a correção adicional da distância. Esta ferramenta possibilita, igualmente, determinar outros parâmetros importantes, como o centro do espelho e a área da imagem a ser processada pelos algoritmos de deteção de objetos.

3.4.2 Deteção de objetos baseada na cor

Os algoritmos desenvolvidos para a deteção de objetos podem ser divididos em três módulos principais, tal como representado na figura 3.3:

- **subsistema utilitário**;
- **subsistema de processamento de cor:** neste subsistema são executados processos de extração e classificação de cor, bem como um processo de deteção de objetos que extrai informação da imagem, com base na análise de cor;
- **subsistema de processamento morfológico:** este subsistema permite detetar a bola, independentemente da sua cor.

De modo a conseguir cumprir os requisitos de processamento de toda a informação relativa à imagem em tempo real, foram implementadas estruturas de dados eficientes [Nev+08;

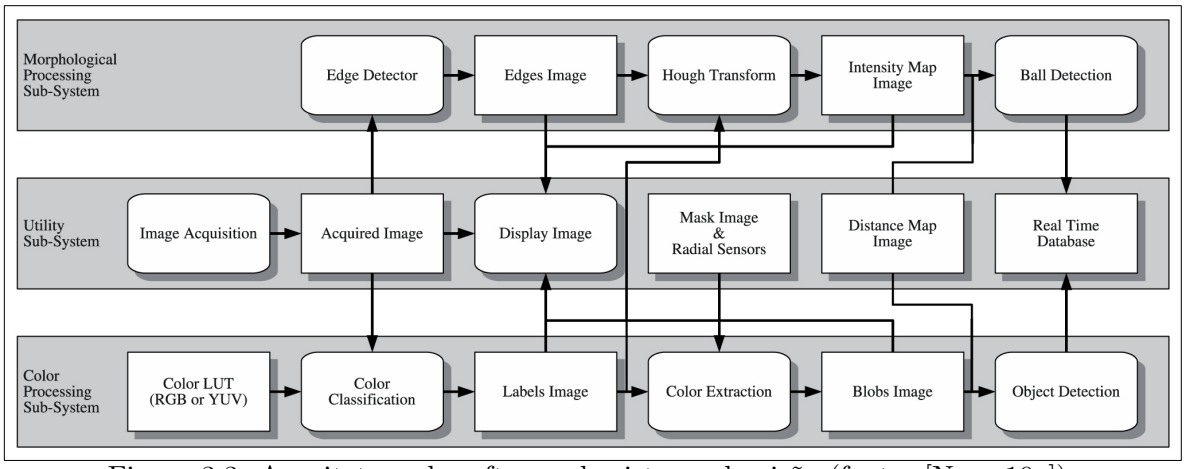


Figura 3.3: Arquitetura de software do sistema de visão (fonte: [Nev+10a]).

Nev+07], bem como explorado o paralelismo entre duas tarefas computacionalmente exigentes, a extração e classificação de cor. Desta forma, torna-se possível tirar o máximo partido dos processadores *dual core* presentes nos portáteis utilizados.

Extração de cor

Se considerarmos o ambiente no qual se desenrola a competição MSL, observamos que robôs pretos procuram jogar com uma bola de uma determinada cor, sobre um campo de jogo verde, com linhas delimitadoras brancas. Este facto evidencia que a cor de determinado píxel numa imagem será uma pista importante para o processo de segmentação de objetos. De modo a obter uma classificação de cor mais rápida é utilizada uma tabela de pesquisa (do inglês *look-up table - LUT*) que contém 16 777 216 entradas (2^{24}, 8 *bits* vermelho, 8 *bits* verde, 8 *bits* azul), sendo que o comprimento de cada entrada é igualmente 8 *bits*, ocupando um total de 16 *MBytes*.

A atribuição de uma classe de cor a cada um dos oito *bits* que compõem cada entrada, tem em conta, tipicamente, as oito principais cores observáveis no cenário de competição, segundo um determinado espaço de cores (por exemplo, poder-se-ão definir as classes como preto, branco, vermelho, azul, verde, amarelo, laranja e rosa). Assim, cada *bit* indica se essa cor pertence a essa classe de cor ou não, podendo haver, no máximo, 256 possíveis classes. Isto significa que uma dada cor pode pertencer, simultaneamente, a várias classes (por exemplo, a um píxel com as componentes R=255, G=255 e B=204, representando a cor amarela clara, poderão ser atribuídas as classes branca e amarela). A fim de classificar um píxel, será necessário somente ler a sua cor, utilizando o seu valor como índice de pesquisa na tabela. O valor obtido representa a classe da cor correspondente, ou seja, a cor segmentada equivalente, a qual será utilizada em todo o posterior processamento. Importa salientar que a classe de cor associada a um determinado *bit* poderá ser facilmente alterada, nomeadamente, quando utilizados outros espaços de cor.

A calibração de cores é efetuada com base no espaço de cores HSV (do inglês *Hue, Saturation, Value* - matiz ou tonalidade, saturação e valor). Este espaço de cores caracteriza-se por ser uma transformação não linear do sistema de cores RGB e assegura um espetro de cores simples e independente. A imagem original pode ser adquirida em formato RGB ou YUV,

sendo posteriormente, convertida numa imagem segmentada, através da LUT.

Algumas áreas da imagem são excluídas da análise, como é o caso do corpo do próprio robô, as hastes de suporte ao espelho ou as áreas exteriores deste. Este processo é simplificado com a utilização de uma máscara, tal como representado na figura 3.4.

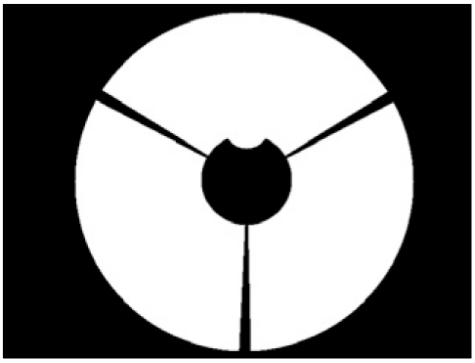


Figura 3.4: Máscara de imagem (fonte: [Nev+10a]).

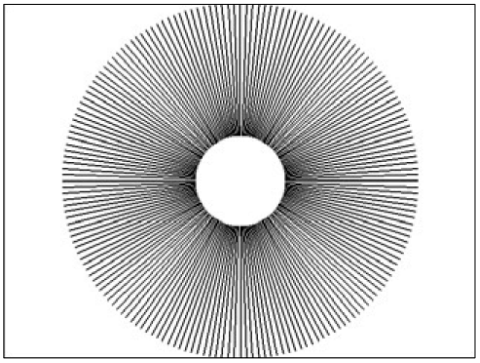


Figura 3.5: Linhas de pesquisa radiais, baseadas no algoritmo de *Bresenham* (fonte: [Nev+10a]).

De modo a evitar processar toda a imagem, são simultaneamente utilizadas linhas de pesquisa radiais (figura 3.5). Uma linha de pesquisa radial caracteriza-se por ser uma linha com origem no centro do robô, que possui um determinado ângulo e que termina no limite da imagem. Estas linhas, baseadas no algoritmo de *Bresenham* [Bre65], são construídas aquando do início da aplicação e, posteriormente, guardadas numa estrutura, por forma a tornar mais rápidos os acessos futuros. Este método permite acelerar o processo de deteção de objetos, pelo facto de apenas ser necessário analisar uma parte dos píxeis considerados válidos. Esta abordagem possui ainda, a vantagem do tempo de processamento ser aproximadamente constante, independentemente da informação adquirida pela câmara. Por outro lado, as coordenadas polares inerentes às linhas radiais de pesquisa facilitam a definição das linhas delimitadoras de um determinado objeto.

A fim de possibilitar, futuramente, a deteção de objetos, e após serem aplicados filtros com o objetivo de reduzir ou eliminar eventuais píxeis isolados, que se assemelhem a ruído na imagem, os algoritmos de pesquisa percorrem cada linha radial em busca de segmentos contínuos de cor e classificam-nos em bolhas de cor.

Uma descrição mais pormenorizada sobre estes algoritmos e o seu funcionamento poderá ser encontrada em [Nev+10a].

Deteção de objetos

No âmbito de uma competição MSL, estamos especialmente interessados em detetar corretamente a bola, os obstáculos e o campo de jogo em si, com fundo verde e linhas brancas delimitadoras. Estes são definidos como objetos de interesse. Os algoritmos desenvolvidos têm por objetivo detetar todos estes objetos, quer eficazmente, quer eficientemente, utilizando as bolhas de cor classificadas anteriormente pelas linhas de pesquisa radiais. Para tal, os algoritmos procuram, essencialmente, transições entre determinadas cores, consoante o tipo de objeto a ser detetado. Estes mesmos algoritmos calculam, ainda, a posição e/ou os limites do objeto, numa representação polar (distância e ângulo). A informação sobre os objetos

detetados é guardada na RTDB, para posteriormente, ser usada pelos processos de alto nível, responsáveis por decidir o comportamento do robô.

Uma descrição mais pormenorizada sobre os algoritmos, a deteção e identificação de obstáculos e os processos de alto nível, poderá ser encontrada, respetivamente, em [Nev+10a], [Sil+09a] e [Lau+09b; Lau+09a; Sil+09a; Sil+09b].

Deteção de bola arbitrária

Nas competições realizadas até ao ano 2009 (inclusive), a bola utilizada assumia, tipicamente, a cor laranja. No entanto e por forma a atingir o objetivo definido para o ano 2050, a tendência passa por, gradualmente, tornar as condições o mais abrangentes possível, pelo que a cor da bola pode assumir uma cor arbitrária para cada competição. Neste sentido e como se pode compreender, a cor da bola deixou de ser um método fiável para o seu reconhecimento. Porém, a sua forma continua a ser esférica, assim como se mantém inalterado um conjunto de outras características.

Este facto sugere que a mesma possa ser reconhecida, recorrendo a uma análise morfológica da imagem. Este tipo de análise permite identificar objetos, com base na sua forma e dimensão, bem como outras características externas.

Para tal, primeiramente precisamos identificar os contornos de possíveis objetos numa determinada imagem. No entanto, esta deteção de contornos deve ser o mais eficiente e precisa possível, por forma a não comprometer a eficiência de todo o sistema. Para além de rápida deve, por um lado, ser tão imune quanto possível ao eventual ruído que a imagem possa conter (retornando contornos bem definidos) e, por outro lado, ser tolerante perante eventuais efeitos "mancha", causados pelo movimento, quer dos robôs, quer da bola. *Sobel* [Zou+06; Zin+07], *Laplace* [Zou+97; Bla+00] e *Canny* [Can86] são alguns dos algoritmos mais conhecidos e frequentemente utilizados para a deteção de contornos. Destes, e tendo em conta os testes realizados pela CAMBADA, o algoritmo de *Canny*, apesar de mais lento quanto ao tempo de processamento, revelou ser o melhor, detetando os contornos da bola de forma correta e nítida, mesmo considerando esta em movimento.

Após a deteção de contornos, obtemos uma imagem binária que contém, unicamente, a informação dos contornos de eventuais objetos. Neste momento, torna-se necessário procurar identificar pontos de interesse, tendo em conta a forma circular da bola. Uma técnica utilizada frequentemente no domínio da análise de imagem, visão computacional e processamento de imagem digital, é a transformada de *Hough*. Esta técnica permite identificar formas geométricas (bem como a sua localização e orientação) que pertençam a uma determinada classe de formas. A transformada de *Hough* utiliza um acumulador que pode ser descrito como uma transformada de um ponto, num plano XY do espaço de parâmetros. Este espaço de parâmetros é definido tendo em conta a forma que pretendemos pesquisar e os parâmetros a considerar. Após calcular todas as possíveis transformadas para os pontos exteriores de uma determinada forma, o espaço de parâmetros contém a frequência de cada píxel como sendo o centro da forma. No caso concreto da bola, dado que a sua forma é circular e tendo em atenção as características da transformada circular de *Hough* [3], o ponto que representa

[3] `http://en.wikipedia.org/wiki/Hough_transform#Circle_Detection_Process`, acedido em: 12/05/2013

o seu centro assume a frequência mais alta. Quando a bola se distancia do robô, o tamanho do seu círculo de contorno diminui, pelo que é necessário correlacionar informação sobre a distância entre a bola e o robô, por forma a ajustar a transformada de *Hough*. No caso da CAMBADA é utilizado o mapeamento inverso, de modo a estimar o raio da bola como uma função da distância [Cun+07b].

Contudo, em algumas situações (como por exemplo, a bola não estando presente no campo de jogo), a transformada de *Hough* "deteta" falsos candidatos, pelo que importa, de alguma forma, descartá-los inteligentemente. Assim, numa primeira fase, um algoritmo procura validar somente os pontos de maior frequência que se encontrem a uma distância máxima pré-definida. Este limite pré-definido encontra-se diretamente relacionado com a distância do ponto ao centro do robô, sendo inversamente proporcional. Numa segunda fase, um outro algoritmo analisa uma pequena área envolvente a cada um dos pontos validados na primeira fase. Caso esta área possua uma percentagem de píxeis verdes acima de um determinado limite obtido experimentalmente, o ponto é descartado. A ideia base deste algoritmo pressupõe que a área analisada funcione como uma caixa, que deverá ter o tamanho ideal, por forma a conter perfeitamente a bola. Se esse espaço contiver mais do que uma certa quantidade de píxeis verdes, então poderemos assegurar, com total certeza, que a bola não se encontra nessa área, sendo esse ponto descartado por se tratar de um falso candidato.

Após a correta deteção da posição da bola, esta informação é guardada na RTDB, conjuntamente com a informação sobre as linhas brancas delimitadoras e obstáculos, a fim de serem acedidas, posteriormente, pelos processos de alto nível, responsáveis pelo comportamento do robô.

Capítulo 4

Arquitetura do *Vision System Processor*

4.1 Introdução

Neste capítulo pretende-se enunciar as várias caraterísticas tidas em conta aquando da definição da arquitetura para o sistema de coprocessamento *Vision System Processor*. Em seguida, será apresentada quer a arquitetura, quer as interfaces de entrada e saída de dados, entre este sistema e o sistema principal de processamento.

4.2 Características

Tendo em conta os objetivos definidos para o presente projeto e após uma análise cuidada do estado de arte, importa definir as características principais para o sistema de coprocessamento a desenvolver.

Por um lado, importa garantir, tanto quanto possível, princípios como a flexibilidade e a escalabilidade de todo o sistema. Assim, pretende-se que a solução a desenvolver seja suficientemente versátil, não se tornando demasiado rígida, considerando somente um número limitado de opções ou dependente de um determinado cenário. A escalabilidade permite assegurar que, futuramente, novas tarefas ou o processamento de dados possam ser incluídos, sem que este processo de atualização implique uma total reestruturação de toda a arquitetura já desenvolvida, ou um custo muito significativo em termos de performance.

Subjacente a estes dois princípios, podemos igualmente salientar a modularidade de todo o sistema. Esta característica permite adotar uma estratégia de dividir para conquistar (do inglês *divide and conquer*), subdividindo um problema ou uma tarefa em subproblemas/subtarefas, visando por um lado, simplificar e concentrar o espírito de resolução somente num objetivo tido como mais simples, e por outro lado, abstraindo o processamento de informação de um determinado módulo, de todo o restante sistema. Deste ponto de vista, cada módulo de processamento poderá ser visto como uma pequena caixa preta, cuja interface de sinais de entrada e saída são necessariamente conhecidos, mas cuja complexidade de implementação é mascarada a todo o restante sistema. Importa ainda salientar que esta abordagem permite que futuras atualizações a este módulo ou, eventualmente, uma total redefinição de todo o seu modelo de processamento, possam ser efetuadas, sendo totalmente independentes

de todo o restante sistema (desde que mantida e respeitada a interface).

No que diz respeito à performance de todo o sistema, a arquitetura desenvolvida procura tirar o máximo partido dos mecanismos de paralelismo e *pipelining*, inerentes às plataformas de desenvolvimento e prototipagem em *hardware* reconfigurável. Para além disso, sempre que possível, a abordagem seguida quer na definição da arquitetura, quer no desenvolvimento dos diversos módulos de processamento da imagem, procura assegurar um processamento contínuo e imediato dos dados logo que disponíveis, procurando desta forma, assegurar um fluxo contínuo de dados processados. Esta abordagem, embora exigente do ponto de vista de definição e implementação da arquitetura, permite sobretudo reduzir o número de recursos associados a elementos de armazenamento. De notar que o uso deste tipo de elementos não é totalmente evitado pois continua a ser necessário assegurar o armazenamento local de pequenos grupos de informação, sobretudo entre módulos que necessitem operar a frequências diferentes devido à complexidade, quer dos dados a processar, quer do processamento em causa. Contudo, optar por guardar somente a informação indispensável, enquanto se assegura um processamento contínuo dos dados, permite reduzir substancialmente o volume da lógica associada a todo o sistema. Este facto é importante no sentido em que as plataformas de desenvolvimento e prototipagem apresentam, normalmente, recursos algo limitados pelo que, a adopção de uma política esbanjadora e imprudente destes recursos, poderá comprometer seriamente a escalabilidade do sistema.

Do ponto de vista de implementação, houve um cuidado especial em tornar todo o sistema o mais autónomo e parametrizável possível. O aspecto de autonomia visa sobretudo reduzir o *overhead* de eventuais sinais ou troca de pacotes de controlo entre o sistema de coprocessamento e o sistema principal. Desta forma, tarefas como a configuração da câmara e a aquisição e posterior processamento da imagem são efetuadas autonomamente pelo sistema de coprocessamento, sem necessidade de controlo por parte do sistema principal. A parametrização relaciona-se de alguma forma com a flexibilidade e versatibilidade da solução, procurando generalizar tanto quanto possível, todo o sistema, permitindo que pequenos aspectos possam facilmente ser alterados e reconfigurados durante uma competição, sem a necessidade de sintetização de todo o sistema ou a amarração a uma solução demasiado específica e talvez inútil na grande maioria dos casos.

4.3 Arquitetura

Tendo em conta as características enunciadas anteriormente, a arquitetura elaborada para o projeto *Vision System Processor* é constituída por vários módulos, tal como representado na figura 4.1. Convém relembrar que a estratégia base para esta arquitetura pressupõe que todos os dados possam ser processados logo que disponíveis. Como curiosidade, este facto permite-nos observar que todo o sistema se assemelha a uma *pipeline*, sendo que os módulos em si, correspondem às diversas etapas de processamento.

Antes de detalharmos a explicação da arquitetura, importa ainda salientar que a figura 4.1 pretende somente apresentar os principais núcleos de processamento, assim como uma simplificação da forma de comunicação entre eles. Nesta fase, julgamos não ser particularmente relevante detalhar quer a lógica de controlo e baixo nível, quer outros elementos auxiliares ao

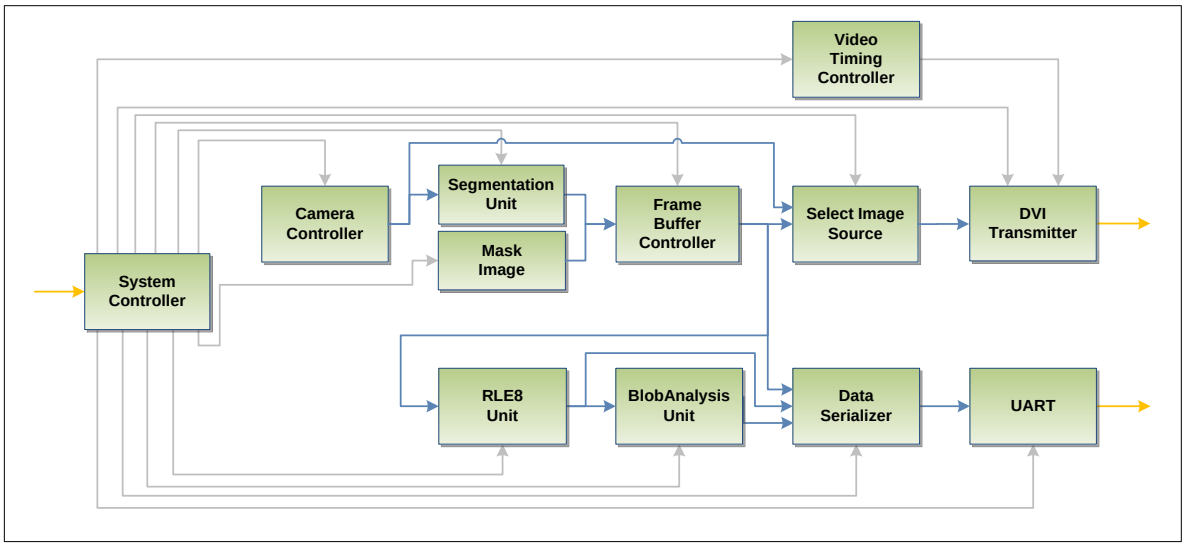


Figura 4.1: Arquitetura simplificada do projeto *Vision System Processor*.

processamento (como por exemplo, FIFOs).

Por outro lado, os módulos responsáveis por controlar e coordenar as interfaces da câmara e saída de video digital (HDMI), bem como o módulo *System Controller* são adaptados de um projeto disponibilizado livremente pelo fabricante. Tanto o referido projeto como os detalhes quer da sua arquitetura e implementação, quer dos aspectos que se relacionam com a sua inclusão no atual projeto, serão devidamente aprofundados no próximo capítulo.

Como se pode observar na figura 4.1, um módulo sobressai imediatamente devido ao elevado número de conexões: trata-se do módulo *System Controller*. Este módulo será responsável por receber os sinais de relógio (*clock*), *reset* e interruptores (*switches*) provenientes do "exterior", desdobrar o sinal de relógio nas frequências tidas como convenientes ao restante processamento e disponibilizar os devidos sinais aos processos em causa, de acordo com as suas exigências. No caso dos sinais *reset* e *switches*, estes são convertidos e disponibilizados de forma síncrona a todo o restante sistema. O sinal indicado a laranja na figura representa os sinais de entrada (*clock*, *reset* e *switches*), enquanto que os diversos sinais a cinzento pretende representar os sinais síncronos e de controlo aos diversos módulos. Os sinais representados a azul pretendem ilustrar as várias dependências de comunicação entre os vários módulos, considerando somente dados a processar.

O módulo *Camera Controller* será responsável por configurar e gerir todo o processo de comunicação com a câmara, disponibilizando à sua saída os dados relativos à imagem adquirida. O módulo de segmentação será responsável, tal como o próprio nome sugere, pelo processo de segmentação da imagem adquirida. A localização deste módulo oferece alguma margem de manobrabilidade, podendo ser colocado após o *frame buffer* (considerando o fluxo de dados). Contudo, essa decisão implica que por cada píxel adquirido, haja necessidade de guardar 16 *bits* na memória, ao passo que, considerando a atual implementação, necessitamos somente guardar 8 *bits*. Admitindo que os dados originais da imagem adquirida não sejam relevantes para o subsequente processamento, o *frame buffer* poderá ser simplificado, libertando eventualmente alguns recursos.

O módulo *frame buffer* representa no fundo a memória DDR, bem como toda a lógica de

acesso subjacente. O seu objetivo é, naturalmente, guardar as imagens adquiridas (segmentadas), facultando essa mesma informação quer aos módulos de processamento de imagem, quer disponibilizando-a para envio para um eventual monitor. Neste último caso, o módulo auxiliar *Select Image Source* será responsável por converter os 8 *bits* de cor segmentada, novamente em 16 *bits*. Se os dados a enviar ao monitor representarem a imagem original (16 *bits*), este módulo transportará somente esses mesmos dados, sem os alterar. O módulo *DVI Transmitter* será responsável por gerir a comunicação com a interface HDMI, convertendo os 16 *bits* recebidos em 24 *bits* (8 *bits* por cada componente RGB). O módulo *Video Timing Controller* será responsável por gerar os sinais de sincronismo para a interface HDMI, de acordo com a taxa e resolução definidos (recorde-se parametrizável).

No que ao processamento dos dados diz respeito e analisando segundo o seu normal fluxo de processamento, aos dados lidos da memória será aplicada a máscara de imagem por forma a verificar se os dados pertencem a uma zona válida para processamento (módulo *Mask Image*). Uma vez validados, os dados serão comprimidos pelo módulo *RLE8 Unit*. Este módulo visa somente encontrar sequências de um mesmo valor para uma determinada linha, ou seja, agrupando somente píxeis de uma mesma côr e que pertençam à mesma linha. O módulo seguinte, *BlobAnalysis Unit* será responsável por detetar bolhas de uma mesma cor, retornando o resultado do calculo das suas características (posição dos seus limites em X e Y, centro de massa e número de píxeis total).

Posteriormente e de acordo com o modo de envio de dados pretendido, o módulo *Data Serializer* será responsável por particionar os diversos tamanhos possíveis dos dados a enviar, de acordo com a capacidade de envio definida pelo tipo de interface de envio (no caso, 8 *bits* por se tratar de uma interface UART/RS232). Finalmente, o módulo *UART* será responsável por configurar e gerir a comunicação com a interface UART.

4.4 Interfaces

Tal como supra citado, as interfaces para o projeto *Vision System Processor* incluem, segundo a sua direcionalidade, os sinais de entrada *clock*, *reset* e *switches*, e como saída, os sinais intrínsecos às interfaces HDMI e UART. Contudo, se admitirmos que os módulos que gerem essas interfaces são eles próprios, parte constituinte da interface, poderemos abstrair e simplificar um pouco mais a arquitetura apresentada na figura 4.1.

Para além destes, importa ainda considerar a interface com a câmara, sem a qual, não seria possível adquirir as imagens, e, em última instância, a memória DDR associada ao módulo *frame buffer controller*. A inclusão desta última como uma interface poderá ser algo discutível, pois o bloco de memória encontra-se incorporado na própria plataforma de desenvolvimento e no fundo, destina-se somente a salvaguardar temporariamente os dados por forma a serem processados. Contudo, numa ótica de escalabilidade e expansibilidade da presente arquitetura, podemos supor que os diversos dados já processados poderão ser salvaguardados nesta memória, sendo desenvolvido um módulo que lhes possa aceder e processar o seu envio ao sistema principal. Embora do ponto de vista externo, esta memória nunca seja acedida de forma direta, esta encontra-se de alguma forma associada aos dados que se pretendem obter, podendo ser vista como inclusa no módulo responsável pelo envio dos dados. A figura 4.2 ilustra as interfaces definidas para o projeto *Vision System Processor*.

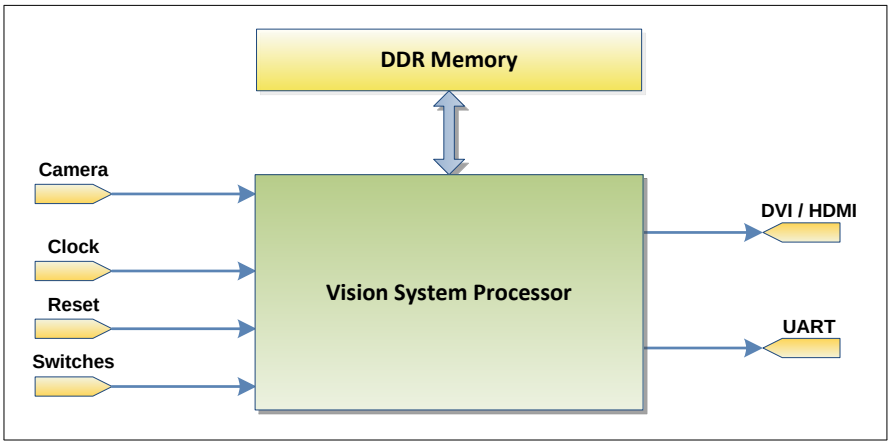


Figura 4.2: Interfaces definidas para o projeto *Vision System Processor*.

Capítulo 5

Implementação do *Vision System Processor*

5.1 Introdução

Neste capítulo será, primeiramente, apresentada a arquitetura implementada para este projeto, bem como, descritos cada um dos módulos implementados. Posteriormente será abordada a interface de comunicação e a arquitetura base de um projeto de demonstração da interligação e funcionamento da referida placa e respetivo módulo *VmodCAM*.

5.2 Projeto *Vision System Processor*

O projeto *VmodCAM_Ref_HD*, disponibilizado pelo fabricante *Digilent* e o qual detalharemos no final deste capítulo, serviu de suporte ao trabalho tido como objetivo desta dissertação, uma vez que o mesmo permitia testar e demonstrar as capacidades e potencialidades do módulo *VmodCAM* e da plataforma *Atlys* (detalhados no Anexo A, página 97). Embora inspirado na referida implementação, o presente projeto pode facilmente ser exportado e incorporado com qualquer outra plataforma ou câmara (sendo somente necessário especificar a interface da plataforma e o módulo controlador da câmara). Assim e numa primeira fase, importava conhecer a arquitetura implementada por este, por forma a alterar o necessário. Tendo sido detetados alguns sinais supérfluos (presentes na declaração, mas não utilizados), foi realizada uma limpeza ao código, refinados alguns comentários já existentes e adicionados outros, de modo a tornar o código e as ideias nele presentes, mais percetíveis futuramente.

Posto isto, o próximo passo consistiu em retirar o módulo controlador da segunda câmara, assim como o *frame buffer* associado. A figura 5.1 representa a arquitetura base do projeto *Vision System Processor*.

5.2.1 Configuração dos parâmetros da câmara

Como já referido anteriormente, o módulo que controla a câmara é responsável pela sua configuração. Para tal, encontra-se definida uma pequena ROM que explicita os diversos parâmetros a configurar e os correspondentes valores de configuração. Estes valores serão

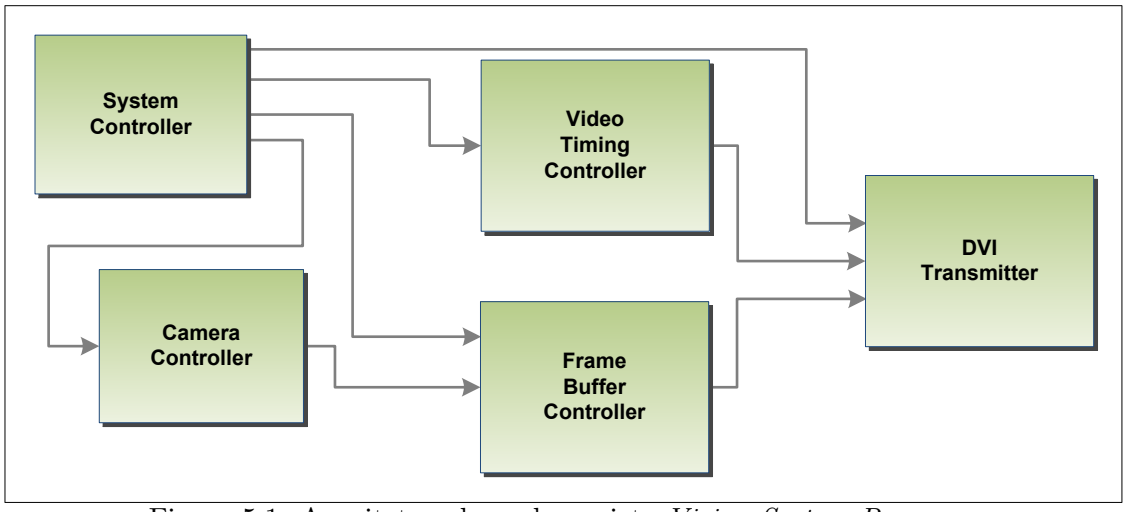


Figura 5.1: Arquitetura base do projeto *Vision System Processor*.

enviados, sequencialmente, à mesma, aquando do início da aplicação ou de um *Reset*. A mesma ROM inclui, ainda, diversos parâmetros, que visam definir os sinais de controlo das máquinas de estado, responsáveis por induzir o comportamento da câmara, de acordo com a especificação técnica referida em [Vmo].

Para este projeto foi definido o formato de saída da câmara como RGB565, o que implica uma representação de cor, para um dado píxel, como 5 *bits* vermelho, 6 *bits* verde e 5 *bits* azul. Quer isto dizer, que a cor de cada píxel é representada em 16 *bits*.

Por omissão, a configuração especificada no projeto inicial define o modo de aquisição de imagem como "obter apenas uma imagem" (*snapshot*), pelo que, por forma a obter um fluxo contínuo de aquisição de imagens, importa redefinir este modo para vídeo. Outros parâmetros passíveis de alteração incluem a definição da resolução de aquisição da imagem. Por outro lado, reconhecemos o interesse de, no futuro, adquirir somente uma determinada região da imagem, de modo a evitar processamento desnecessário. Este processo é denominado de corte da imagem (do inglês *crop*). No caso específico da competição RoboCup MSL, esta área pode ser definida para um quadrado, que contemple somente a área útil do espelho. Isto permite-nos, de imediato, reduzir, quer a carga, quer o tempo de processamento com as restantes áreas (assumindo que elas não tenham interesse). Contudo e durante a fase de desenvolvimento, estes parâmetros permaneceram inalterados relativamente ao projeto inicial (resolução 1600x1200, sem corte na imagem).

Tendo definido a arquitetura base, a estratégia consiste em incorporar, progressivamente, os diversos módulos, realizando os testes tidos como convenientes, de forma a assegurar a sua correta funcionalidade.

5.2.2 Segmentação de cor

O primeiro módulo incorporado diz respeito à segmentação de cor. Este módulo terá como principal função, classificar as cores da imagem em classes previamente definidas.

Como vimos anteriormente, as imagens processadas pela câmara encontram-se no formato RGB565, o que se traduz em 16 *bits* de cor para cada píxel. Isto significa que teremos um

máximo de 65 536 cores possíveis ($2^{16} = 65\,536$). Embora este número seja significativamente menor do que o possibilitado atualmente no sistema CAMBADA (recorde-se $2^{8R+8G+8B} = 16\,777\,216$ cores), os testes realizados no reconhecimento e classificação das principais cores presentes durante uma competição, sugerem ser suficiente.

Por forma a tornar a classificação de cor mais rápida, e à semelhança da atual solução presente na CAMBADA, utiliza-se uma LUT. Esta é composta por 65 536 entradas, cada uma delas com um comprimento de 8 *bits*, ocupando um total de 64 KB.

Para o seu preenchimento foi desenvolvido um pequeno *script* em MATLAB, que cria e percorre todas as combinações possíveis de cor, verifica a composição de cada uma, em termos das componentes R, G e B, e com base em intervalos de valores de cor programados (segundo as características próprias do espaço de cor adotado), atribui uma classe a essa cor. Este *script* permite igualmente, exportar a LUT criada, para um ficheiro de extensão .coe, que será usado, posteriormente, na inicialização da LUT sobre a plataforma Atlys. Embora o método de criação da LUT possa não ser totalmente elegante ou o mais eficiente possível, convém lembrar que este assume um propósito meramente demonstrativo. Atualmente, métodos mais práticos e eficientes já se encontram implementados e em uso, totalmente parametrizados e calibrados para o ambiente de competição, sendo que o seu resultado poderá ser exportado para um ficheiro que, posteriormente, possibilite o seu uso para a inicialização da LUT, presente neste projeto. Tal como mencionado em [Nev+10a], mesmo considerando outros espaços de cor, o tamanho da LUT não é alterado, sendo somente necessário mudar o significado atribuído a cada componente.

Em termos de implementação física, a LUT consiste numa pequena memória ROM, criada com recurso ao assistente de criação de novos *IP Cores*. Na atual implementação, esta memória é inicializada com recurso ao ficheiro exportado pelo *script*, em MATLAB, sendo necessário sintetizar o projeto de modo a refletir eventuais alterações ao conteúdo desse ficheiro. Igualmente, durante a execução do projeto sobre a plataforma, unicamente são permitidos acessos de leitura. Contudo, num trabalho futuro será interessante tornar esta inicialização independente da sintetização, aceitando, por exemplo, uma nova configuração de LUT, via UART.

Em termos de processamento, este módulo recebe os 16 *bits* que compõem a cor de um determinado píxel, utiliza esse valor como endereço de leitura da ROM e devolve como resultado, o valor lido que representa a classe de cor.

5.2.3 Imagem de máscara

Embora seja possível selecionar diretamente uma área de interesse aquando da aquisição pela câmara (recorrendo à configuração desta, como vimos anteriormente), surge ainda a necessidade de descartar algumas pequenas áreas da imagem obtida. Estas áreas incluem, entre outros eventuais pormenores, as zonas onde se encontra a reflexão dos suportes do espelho hiperbólico e do corpo do robô. Do ponto de vista da competição RoboCup, esta informação não é relevante. Por este motivo, continua a fazer sentido utilizar uma imagem binária, cujo objetivo nos permita descartar informação considerada irrelevante. Esta imagem é denominada imagem de máscara.

Para efeitos demonstrativos foi considerada uma imagem binária, com dimensão 700x700 píxeis. Esta imagem, ilustrada na figura 5.2, permite somente descartar os cantos da imagem adquirida, pois, como se pode observar, não contém as áreas relativas aos suportes do espelho. A área assinalada a preto será descartada, considerando-se apenas a área equivalente na imagem original, à área branca. Utilizando o mesmo principio referido na implementação da segmentação, foi utilizado o assistente de criação e configuração de *IP Cores*, de modo a criar uma pequena ROM de tamanho ajustado à dimensão da imagem (490.000 *bits* $\simeq$ 64KB).

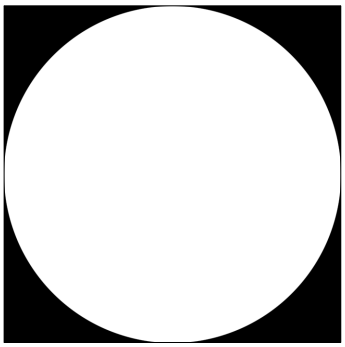


Figura 5.2: Imagem de máscara.

Na prática e em termos de processamento, este módulo pode ser paralelizado com o processo de segmentação. Deste modo, enquanto o módulo de segmentação processa um dado píxel, este módulo verifica se para o mesmo píxel e de acordo com a sua posição, este deve ser considerado válido em termos de processamento futuro ou não. Caso o píxel seja válido, o resultado da segmentação é considerado e salvaguardado no *frame buffer*; caso o píxel não seja válido, o valor de segmentação é substituído por uma cor pré-definida. Naturalmente, esta cor não deverá representar nenhuma das habituais cores presentes numa competição, o que poderia induzir em erro todo o restante processamento. Todavia, se admitirmos que o processo de segmentação ocorreu previamente a este módulo, sabemos que o valor que precisamos processar neste momento é definido pelos 8 *bits*, que representam a classe de cor atribuída pelo processo de segmentação. Assim sendo, essa cor pré-definida poderá assumir um valor que represente uma situação improvável, como por exemplo, uma cor pertencer, simultaneamente, à classe da cor branca, preta e outra cor. Naturalmente, o restante sistema deverá ser totalmente imune a esta cor.

5.2.4 Compressão de dados (RLE)

Como sabemos, o ambiente onde se desenvolve a competição RoboCup MSL é predominantemente dominado por um conjunto de cores: preto para robôs e demais obstáculos, verde para o campo, branco para as linhas delimitadoras e uma outra cor para a bola. Embora possam, obviamente, coexistir outras cores neste ambiente, consideremos por ora, estas como as mais importantes.

Considerando a figura 5.3, outra característica que podemos observar diz respeito à forma como as cores se encontram dispostas. De modo a simplificar esta ideia, consideremos, por exemplo, uma linha imaginária horizontal, que intercete a bola e que compreenda somente

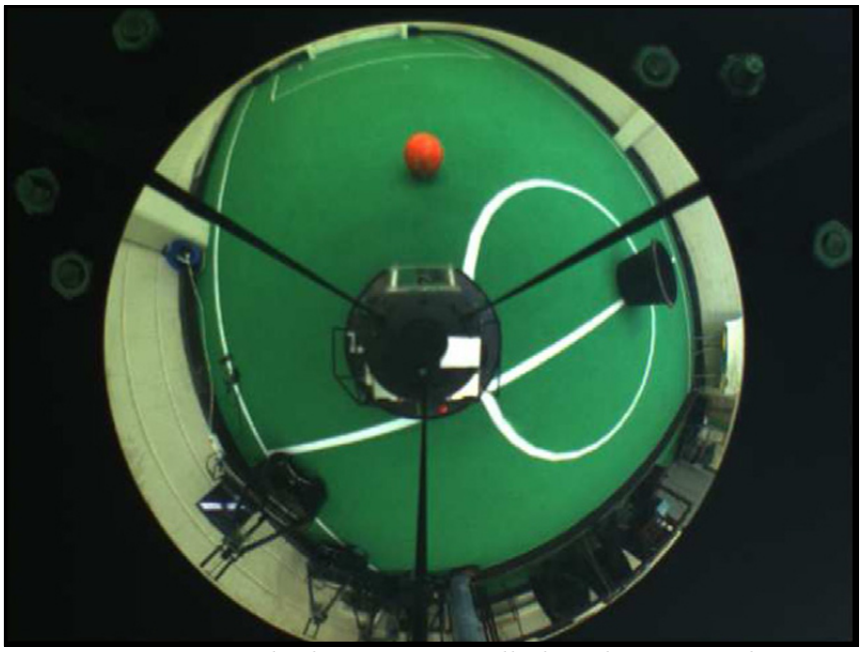


Figura 5.3: Exemplo de imagem recolhida pelo sistema de visão omnidirecional da equipa CAMBADA.

---

a área do campo (na sua medida de largura). A sequência de segmentos de uma mesma cor (independentemente da sua dimensão), permite-nos observar a seguinte ordem de cores: verde, branco, verde, laranja, verde, branco e por fim, verde (respetivamente).

Naturalmente, este exemplo diz respeito a uma área muito específica, não refletindo as demais possibilidades. No entanto, permite-nos identificar a ideia essencial: segmentos de uma determinada cor sucedem segmentos de outra cor, independentemente da sua dimensão. Todavia, não podemos descurar o facto de existirem ligeiras variações na cor, o que, do ponto de vista de análise de dados, significa cores distintas.

Porém, o processo de segmentação de cor permite-nos reduzir o número total de cores, agrupando de certo modo, as cores principais e suas pequenas variações, numa única cor. Assim sendo, é possível agrupar, por exemplo, todas as variações de verde visíveis na imagem, numa única cor verde. Deste modo, a ideia de segmentos de cor torna-se ainda mais interessante.

Por outro lado, até ao momento, foi possível processar a informação de forma contínua. No entanto, alguns processos posteriores possuem uma complexidade de processamento elevada, associada a grandes conjuntos de dados. Se observarmos o processo de análise e deteção de bolhas de cor, começamos a antever problemas, considerando que para cada píxel, precisamos averiguar, não só os seus vizinhos imediatamente anteriores, como os vizinhos da linha anterior. Uma solução, com o objectivo de reduzir essa mesma carga computacional, poderá passar pela adoção da compressão de dados, recorrendo à codificação RLE.

Na secção 2.3 (pág. 16) poderá encontrar uma descrição mais pormenorizada, assim como um exemplo prático sobre esta codificação.

A implementação deste módulo, embora seja baseada na especificação descrita em [Rle],

encontra-se simplificada, não contemplando os modos absoluto e delta descritos. De igual modo, também a deteção e sinalização de "fim de linha" e "fim de *frame*" são efetuadas paralelamente, sendo que a sua sinalização pode ser enviada juntamente com os dados codificados, no momento em que é detetada. Desta forma, é eliminada a necessidade de enviar a informação separadamente, adicionando *overhead* desnecessário de dados. O contador interno de codificação possui 8 *bits*, possibilitando uma contagem até 256 elementos repetidos. Após o processo de segmentação, sabemos igualmente que a cor é representada por 8 *bits*, sendo que a sinalização de "fim de linha" e "fim de *frame*" é representada por 1 *bit* cada.

Assim sendo, o sinal *output* é constituído por: contador 8b + cor 8b + EOL 1b + EOF 1b, perfazendo os 18 *bits*. Adicionalmente, é disponibilizado um sinal *ValidOut*, de modo a validar a informação de saída deste módulo. A figura 5.4 ilustra o diagrama de blocos correspondente à implementação deste módulo, incluindo, ainda, uma representação interna ao sistema do sinal *Output*.

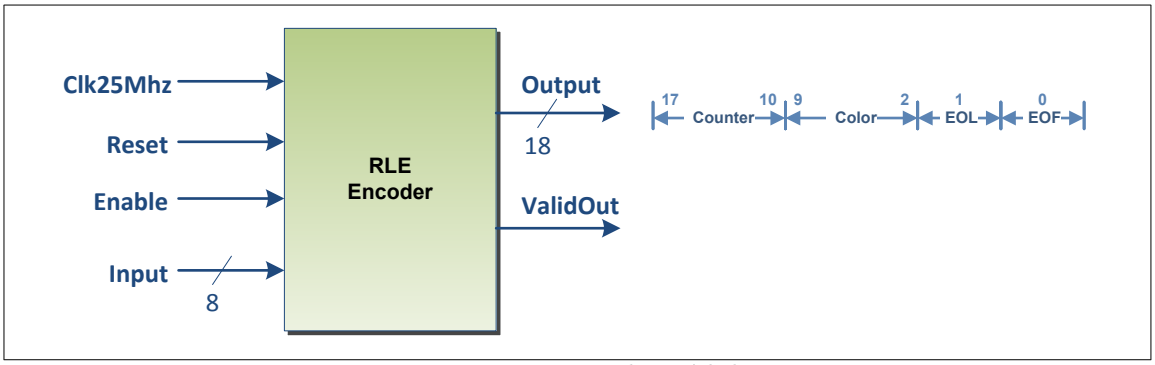


Figura 5.4: Diagrama do módulo RLE.

5.2.5 Deteção de bolhas de cor (*BlobAnalysis*)

De modo a reconhecer obstáculos, bola e linhas brancas, é necessário, primeiramente, identificar as denominadas bolhas de cor. Estas são grupos de píxeis adjacentes entre si e que partilham uma cor. Tendo identificado estas bolhas, é possível calcular algumas características, a fim de podermos classificar cada bolha e obter dados que permitam uma decisão de alto nível.

Contudo, dado que todo este processamento possui uma elevada complexidade subjacente, optou-se por dividir o mesmo em duas fases. Por outro lado e tal como o leitor poderá constatar, a arquitetura e a implementação sugeridas assemelham-se elas próprias a um processador, sendo que numa primeira fase, teremos inerente o conceito de instruções e posteriormente, uma pipeline de processamento das mesmas. Seguidamente, apresentaremos a referida implementação.

Este módulo é constituído por um FIFO e dois módulos de processamento: *ProcRunUnit* e *PipelineUnit*. O FIFO permite salvaguardar, temporariamente, a informação proveniente do módulo de compressão RLE. Na verdade, este FIFO assume tarefas de mediação de informação entre o módulo RLE e o próximo módulo. Até ao momento, os dados têm sido processados sempre em modo sequencial, isto é, quando a informação fica disponível, é enviada a um módulo que efetua um determinado processamento específico, disponibilizando à sua saída

um novo resultado, que, por sua vez, irá imediatamente alimentar outro módulo. Esta solução permite-nos poupar preciosos recursos que seriam utilizados se necessitássemos de armazenar a informação intermédia da imagem.

Contudo, o módulo RLE não produz informação de modo constante. Tal como mencionado anteriormente, o seu funcionamento depende da sequência de dados. Para sequências com muita variação de informação, este módulo terá dados válidos à sua saída, com uma maior frequência. Porém, quando as sequências tiverem pouca variação, este módulo poderá não produzir informação válida durante alguns ciclos de relógio.

Por outro lado, o próximo módulo na cadeia de processamento, apresenta igualmente uma complexidade mais elevada. Como iremos ver adiante, este módulo necessita correlacionar um conjunto de dados atuais, com outros conjuntos anteriores, a fim de verificar se existe vizinhança. Torna-se claro que este processo poderá demorar alguns ciclos de relógio, enquanto processa um conjunto de dados. Se durante esse processamento, eventualmente o módulo RLE enviasse dados, estes seriam perdidos, pois o próximo módulo não os poderia aceitar. O FIFO garante esta mediação de dados entre os dois módulos, permitindo adicionalmente, frequências de operação distintas.

O módulo *ProcRunUnit* terá como principal objetivo receber os dados provenientes do RLE (por intermédio do FIFO) e organizá-los, de tal modo que seja possível processar um conjunto mínimo de informação, como se analisássemos uma parte da imagem original. Este processamento envolve a pesquisa de eventuais vizinhanças verticais de píxeis (na verdade, sequências) de uma mesma cor, que tenham uma relação de adjacência entre si.

Tendo conseguido organizar internamente os dados provenientes do RLE e uma vez bem definidas as operações de deteção de vizinhança para os diferentes casos, assim como a atualização de toda a informação de suporte, necessitamos ainda de calcular as características inerentes a cada objeto. Contudo, quer este cálculo, quer o acesso à memória poderiam colocar-nos problemas de desempenho, necessitando de alguns ciclos de processamento adicionais. De igual modo, as operações matemáticas envolvidas (potência, multiplicação e divisão), se não forem bem estruturadas, poderão reduzir a frequência total de operação do sistema. Este problema seria mais evidente se procurássemos realizar esses mesmo cálculos durante esta etapa de processamento de *Runs*. Com vista a contornar esse problema, procuraremos realizar a tarefa de atualização de objetos e o cálculo das suas características de forma paralela a todo o processamento descrito neste módulo.

Uma possível solução poderá ser a adoção de uma *pipeline*, cujas várias etapas de processamento permitirão realizar estas operações de forma concorrente entre si, com um elevado débito e sem prejudicar a frequência máxima de operação do sistema. No entanto, a *pipeline* necessita saber em cada momento, qual o tipo de operação que deverá realizar sobre os objetos.

Para tal, o módulo *ProcRunUnit* deverá ser capaz de inferir corretamente diferentes instruções, consoante seja possível encontrar vizinhanças para uma dada *Run*, não haja vizinhanças ou, de acordo com a pesquisa de objetos completos, detete objetos que possam ser enviados e libertadas as suas posições em memória e respetivo número associado.

Assim sendo, este módulo irá produzir um novo tipo de dados, semelhante a uma instrução de operação, que será depois enviada à *pipeline*, definindo toda a sua operação para um dado objeto.

Processamento dos dados de uma *Run* (*ProcRunUnit*)

De modo a processar a deteção de bolhas de cor torna-se necessário relacionar os dados recebidos até ao momento. Sabendo que uma bolha de cor é um agrupamento de píxeis adjacentes e que partilham de uma mesma cor, estamos sobretudo interessados em identificar as diversas bolhas, a sua área e a sua localização na imagem.

Assim, este módulo terá como principal função receber os dados provenientes do RLE e transformá-los num conjunto de dados interno, que permita representá-los como parte da imagem (em termos de posição relativa a outros conjuntos de dados). Estes conjuntos de dados serão mantidos como um histórico de dados recentes, permitindo que novos conjuntos possam ser correlacionados com outros mais antigos. O principal objetivo consiste em procurar encontrar eventuais vizinhanças de píxeis adjacentes e com uma mesma cor (bolha de cor).

Como vimos anteriormente, os dados provenientes do módulo de compressão RLE são agrupados em 18 *bits*, onde se incluem 8 *bits* para o contador e valor de cor, respetivamente, e dois *bits*, sinalizando um eventual "fim de linha" ou "fim de frame". No entanto, esta informação por si só, não nos permite saber muito, pois sem um histórico, não a podemos relacionar, quer temporalmente, quer espacialmente, no domínio de uma imagem. Para que tal possa acontecer, precisamos de reconstruir algo que se assemelhe à *frame* da imagem.

Nesse sentido, iremos considerar uma estrutura de dados denominada *Run* e definida como: {*cor*, *posStart*, *posEnd*, *objNumber*}. Em "*cor*" iremos guardar o valor da cor; "*posStart*" e "*posEnd*" dizem respeito às posições de início e fim de uma sequência de píxeis da mesma cor; e "*objNumber*" irá guardar, futuramente, um número de objeto a definir. As posições de início e fim podem facilmente ser inferidas, recorrendo ao contador proveniente do RLE em determinado momento, e incrementando uma variável interna de posição (que numa próxima iteração será utilizada como posição inicial). Os *bits* de sinalização EOL e EOF poderão ser usados para verificação interna da coerência dos dados recebidos. Na verdade, admitindo que não haja perda de dados ou erros de processamento entre os dois módulos, os *bits* de sinalização poderiam mesmo ser desprezados.

Para efeitos de deteção de uma vizinhança, precisamos comparar a atual *Run* com os dados de outras *Runs* recentes, nomeadamente as anteriores mas que possam ainda pertencer à mesma linha que estamos a processar da imagem; e a linha imediatamente anterior. No entanto, atendendo a que o RLE processa sequências de repetição de um mesmo valor, dois valores consecutivos e que pertençam a uma mesma linha, nunca poderão dizer respeito à mesma cor. Ademais, no nosso caso não estamos interessados em comparar dados de uma mesma cor, pertencentes a uma mesma linha, embora não adjacentes entre si, pois não sabemos no atual momento, se ambas se encontram relacionadas. Quer isto dizer que o RLE eliminou a necessidade de averiguar vizinhanças na mesma linha, sendo somente necessário pesquisar a linha imediatamente anterior.

Uma vizinhança existe se para a *Run* que estamos atualmente a processar, existir uma ou mais *Runs* na linha anterior, com a mesma cor da atual e cujas posições se sobreponham ou sejam imediatamente adjacentes às atuais posições. Se não forem detetadas vizinhanças para a atual *Run* (ou no caso de estarmos a processar a primeira linha de uma *frame*), iremos assumir que a sequência de dados se refere a um novo objeto, pelo que atribuiremos o número de um novo objeto (figura 5.5a). Caso exista uma vizinhança, a atual *Run* assumirá

o número do objeto que a sua vizinha contiver, por forma a que possam referir-se ao mesmo objeto (figura 5.5b). Para tal necessitamos de uma pequena lista que contenha números de objetos livres, suscetíveis de serem atribuídos. Esta lista é inicializada automaticamente, no início do nosso sistema (ou aquando de um *Reset*). Quando necessitamos de atribuir um novo objeto, iremos retirar um número que se encontre disponível nesta lista e atribuí-lo à atual *Run*. Pelo contrário, sempre que determinado objeto já não se encontre atribuído, o seu número será novamente inserido nessa lista, podendo ser utilizado posteriormente.

No caso de ser encontrada mais do que uma vizinhança na linha anterior, temos uma situação especial (figura 5.5c). A primeira vizinhança encontrada contém um número de objeto associado, o qual iremos atribuir à atual *Run* (à semelhança do que descrevemos para o caso de uma única vizinhança). As restantes vizinhanças encontradas na linha anterior dizem respeito a *Runs* não adjacentes entre si mas que partilham da mesma cor. Ademais, quando relacionadas com a atual *Run*, sabemos que os seus limites são adjacentes a esta última (por imposição da pesquisa de vizinhança). Então poderemos concluir que estas *Runs* dizem respeito ao mesmo objeto, embora na verdade, ainda refiram números de objetos diferentes. Por forma a associar corretamente futuras *Runs* que detetem vizinhança com estas, precisamos atualizá-las de modo a que respeitem ao mesmo objeto. Para tal, assumiremos que elas se encontram erradamente associadas aos seus objetos e iremos definir um número de objeto comum a todas elas. Este número de objeto será o número associado à primeira vizinhança que havíamos detetado (o qual também atribuímos à atual *Run*). No exemplo ilustrado na figura 5.5c, o objeto 4 associado à segunda vizinhança detetada será então substituído pelo número de objeto 2. Desta forma, futuras *Runs* vizinhas desta serão corretamente referenciadas com o número de objeto 4 (figura 5.5d), pelo que obteremos uma deteção correta (figura 5.5e).

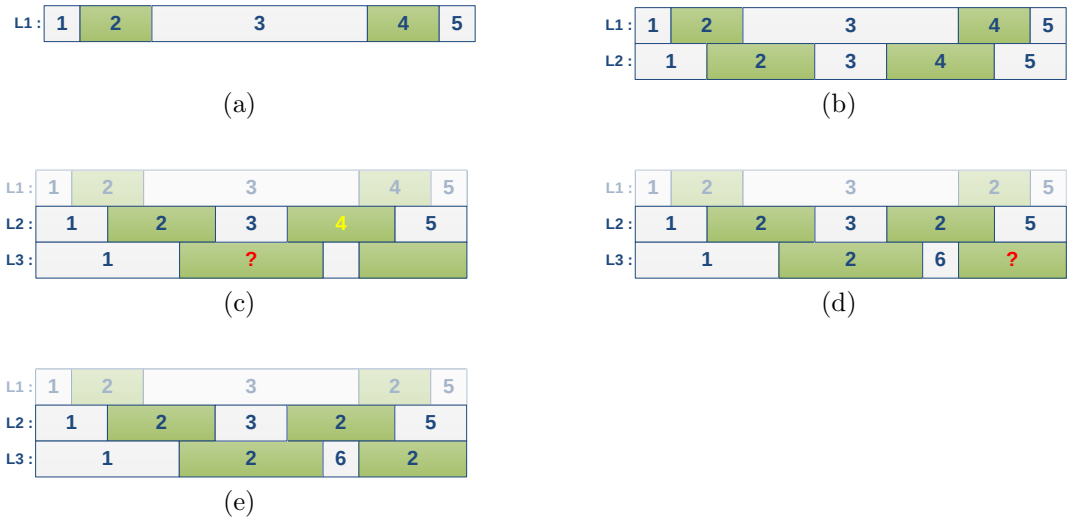


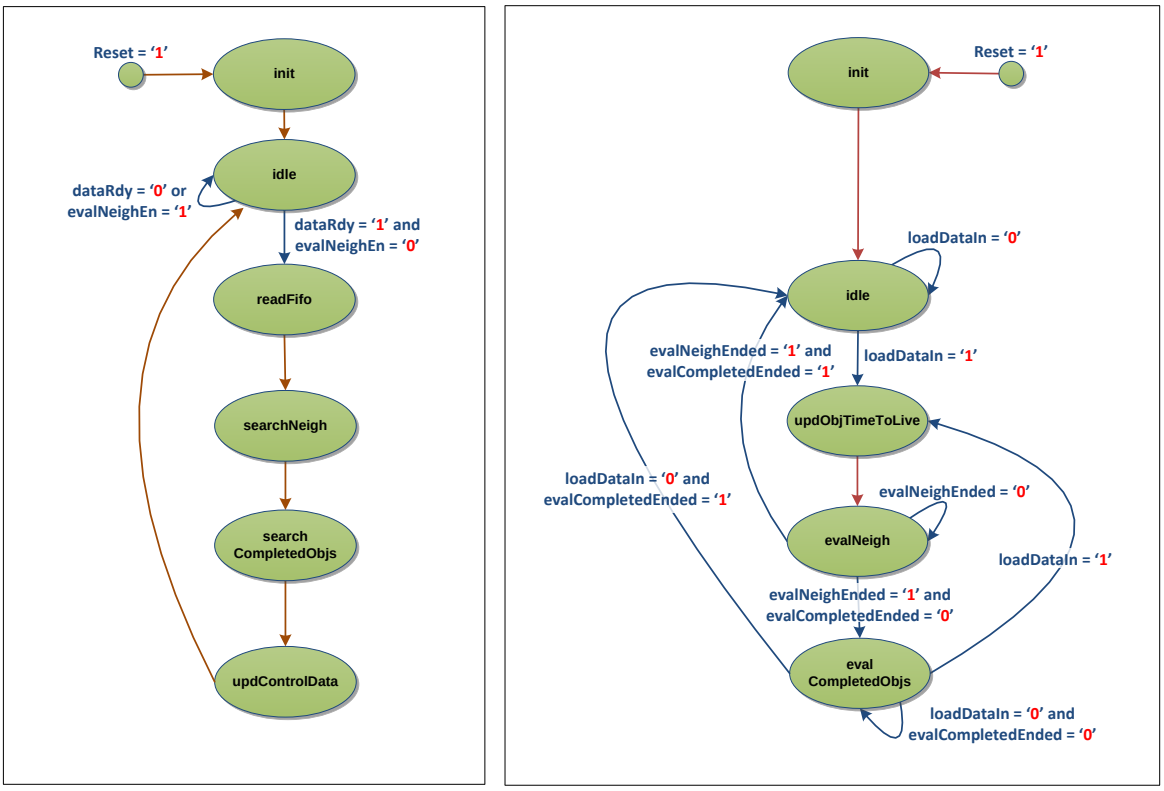
Figura 5.5: Exemplo de atribuição de números de objetos a uma *Run*.

Dado que somente mantemos em memória a informação das *Runs* da linha atual e imediatamente anterior, quando completamos uma linha, necessitamos descartar a informação da que considerávamos nesse momento como anterior e começaremos a guardar novos dados para

a atual linha. Alguns dos objetos associados a essa linha, entretanto descartada, poderão estar associados à nova linha que consideramos como anterior. Naturalmente, esses continuarão a estar utilizados, pelo que não poderemos fazer nada. No entanto, muito provavelmente, alguns objetos da linha descartada já não serão associados a novas *Runs*, por não haver possibilidade de estabelecer uma vizinhança com esses objetos. Estes objetos serão considerados como estando completos, podendo ser novamente colocados na lista de objetos livres, a fim de serem utilizados, se necessário. Por forma a mantermos uma relação dos objetos que havíamos considerado anteriormente e que entretanto foram descartados, necessitamos manter uma lista que nos permita saber o histórico de atualização dos objetos atribuídos. Essa lista irá conter uma espécie de contador para cada objeto atribuído. O contador, por sua vez, será atualizado quando esse objeto for atribuído ou referenciado a uma nova *Run*. Sempre que detetarmos um final de linha ou final de frame, iremos decrementar ou colocar esse contador a zero, respetivamente. A fim de ir mantendo esta lista organizada, precisamos somente de um processo que pesquise por eventuais objetos, com o seu contador de atualização a zero. Quando encontrados, o número de cada objeto será novamente inserido na lista de objetos livres, sendo igualmente marcado como livre.

Na figura 5.6 são apresentados os diagramas que descrevem a implementação de duas máquinas de estados, responsáveis por coordenar as diferentes operações associadas ao processamento de cada *Run*. A presente implementação procura seguir a mesma política de processamento *raster scan* dos módulos anteriores. Contudo, dada a complexidade inerente ao processamento da informação relativa às *Runs*, necessitamos de assegurar um tempo de processamento o mais reduzido possível, de modo a não comprometer a eficiência de todo o sistema. Assim, para além de triplicarmos a frequência de funcionamento deste módulo, relativamente aos anteriores (75 MHz vs 25 MHz), teremos especial atenção em procurar paralelizar todo o processamento o mais eficientemente possível. Um outro aspecto tido em conta, com vista à otimização do número de recursos utilizados, foi a redundância de informação e respetivas estruturas de dados associadas. Contudo, optar por estruturas de dados comuns a várias operações, pode revelar-se nefasto, obrigando a introduzir latência no processamento da informação, uma vez que precisamos de garantir o acesso concorrente a esta (quer em termos de tempo, quer em termos de dependência de atualização de dados, relativamente a operações anteriores). De modo a contornar estes problemas, dividiu-se o processamento por duas máquinas de estados. Embora concorrentes entre si, estas são mutuamente dependentes no que respeita ao processamento da informação associada a cada um dos seus diferentes estados.

A primeira máquina apresentada assume-se como uma máquina simples, com o objetivo principal de coordenar as várias fases de processamento de uma *Run* disponível no FIFO. Esta é composta por seis estados, representados de forma simplificada na figura 5.6a, tendo em conta o tipo de processamento associado a cada um. A transição entre estes é, maioritariamente, assegurada somente pela mudança de ciclo de relógio. O início do sistema ou a ativação do sinal *Reset* induzem o estado *init*, onde basicamente, se inicializam as estruturas de dados auxiliares ao processamento e controlo. No estado *idle* é verificada a existência de dados para processamento no FIFO. Confirmando-se a sua disponibilidade e caso a segunda máquina não se encontre a processar os resultados de uma pesquisa de vizinhanças, poderemos iniciar o processamento de uma nova *Run*, ativando a transição para o próximo estado. No estado *ReadFifo* processamos a leitura dos dados da nova *Run*. Uma vez lidos,



(a) *Finite State Machine 1* (FSM1). (b) *Finite State Machine 2* (FSM2).

Figura 5.6: À esquerda, diagrama da máquina de estados principal; à direita, diagrama da máquina de estados secundária.

poderemos ativar a pesquisa por eventuais vizinhanças com a anterior linha de *Runs*. Esta pesquisa é feita de forma paralela, comparando, simultaneamente, a atual *Run* com todas as *Runs* eventualmente associadas à linha anterior. Tendo concluído esta pesquisa, verificamos a necessidade de procurar eventuais objetos já completos. Esta pesquisa é ativada somente quando detetados os sinais EOL ou EOF, ou seja, "fim de linha" ou "fim de frame". No último estado, *updControlData*, teremos o especial cuidado de atualizar a informação de controlo do processamento.

A segunda máquina visa sobretudo processar os resultados de ambas as pesquisas (vizinhanças e objetos completos), bem como atualizar a informação auxiliar dos objetos atribuídos. Este processamento foi dividido por 5 estados, representados de forma simplificada na figura 5.6b, de acordo com o tipo de processamento associado a cada um. No estado *init*, e à semelhança do que sucede com a primeira máquina, é inicializada a informação de controlo e eventuais estruturas de dados auxiliares, utilizadas por esta máquina ou pelos processos por ela geridos. Este estado é automaticamente imposto quando ativado o sinal *Reset* ou aquando do início do sistema. O estado *idle* é um estado de espera, significando que todo o processamento gerido por esta máquina se encontra completo. A máquina permanecerá neste estado enquanto não for ordenada uma leitura ao FIFO (imposta pela FSM1). Sendo ordenada uma leitura, esta máquina verifica imediatamente se, dos dados lidos do FIFO, é detetado o sinal EOL ou EOF. Esta verificação permite atualizar o tempo de vida de cada objeto, de acordo

com a sua eventual e recente atualização. Isto quer dizer que, para todos os objetos tidos como atribuídos, iremos decrementar o seu contador de tempo. Como supracitado, a FSM1 no seu estado *searchCompletedObjs*, irá ordenar posteriormente e detetando os mesmos sinais, uma pesquisa por objetos tidos como completos (ou não utilizados/referenciados nas últimas duas linhas). Esta pesquisa marcará imediatamente os eventuais objetos cujo tempo de vida se encontre a zero, permitindo que a sua informação possa ser processada e descartada e o seu número possa ser disponibilizado novamente. O estado *updObjTimeToLive* é paralelo ao estado *searchNeigh* da FSM1. No estado *evalNeigh* iremos analisar os resultados da pesquisa de vizinhanças ordenada pela FSM1. Este estado é tido como obrigatório enquanto não forem processados todos os resultados, situação, por exemplo, de várias *Runs* detetadas, para as quais é necessário atualizar o número do objeto associado. Como vimos, esta condição pode limitar o fluxo de execução da FSM1, mas é inevitável, por forma a assegurar um correto processamento e correlação de toda a informação. Ainda assim, esta limitação surge apenas quando para a atual *Run* são encontradas mais do que duas vizinhanças, as quais não podem ser processadas durante o fluxo normal e contínuo da FSM1.

Tendo concluído a análise dos resultados da pesquisa de vizinhanças, a FSM2 verifica a necessidade de analisar eventuais resultados da pesquisa de objetos completos. Esta situação poderá ocorrer numa de duas situações: os resultados da última pesquisa ainda não foram todos processados ou uma nova pesquisa foi realizada entretanto, requerendo uma análise aos eventuais resultados. Se for o caso, a FSM2 transita para o estado *evalCompletedObjs* e analisa os ditos resultados. Caso não haja essa necessidade, a máquina transita para o estado *idle*, onde aguardará novo processamento por parte da FSM1. Por sua vez, estando no estado *evalCompletedObjs*, a FSM2 manterá este estado, enquanto não for detetada uma ordem de leitura por parte da FSM1, situação em que necessitará preparar-se para atender ao processamento da informação da nova *Run*. Neste caso, optou-se por não tornar este estado como limitativo ao processamento de uma nova *Run*, uma vez que objetos completos surgem em menor número e é possível processá-los de forma intercalada com novas *Runs*. Contudo, esta decisão impõe um especial cuidado em dotar o sistema de um número de objetos suficientemente grande, por forma a permitir que novas *Runs* sem vizinhanças possam ter um número de objeto atribuído. O pior caso surge após a sinalização EOF ("fim de frame"), a qual implica que todos os objetos atribuídos até esse momento devam ser libertados. De momento, a presente implementação não impõe ainda uma prioridade em despachar estes objetos, podendo mesmo, em último recurso, suspender o processamento de uma (ou várias) *frames*. Esta necessidade precisa de ser convenientemente avaliada perante imagens recolhidas no ambiente natural onde se insere a competição, dado que o módulo RLE depende fortemente do tipo de dados recolhido, o qual, por sua vez, poderá influenciar negativamente este processo. Para os testes realizados até ao momento, com imagens de pequena dimensão a presente implementação responde adequadamente. Tendo processado todos os eventuais objetos detetados como completos e não verificando uma ordem de leitura da FSM1 ao FIFO, a FSM2 regressa ao estado *idle*, onde aguardará o processamento de um nova *Run*.

Até ao momento, sabemos como receber a informação do RLE e convertê-la numa estrutura de dados (*Run*), por forma a estabelecer uma relação espacial e temporal, de acordo com os dados originais da imagem capturada. De igual modo, estabelecendo vizinhanças entre essa informação, podemos descrever e associar eventuais objetos. No entanto, ainda não efetuamos qualquer cálculo de características, como o centro de massa, posição absoluta em termos de coordenadas e área ocupada. Por outro lado, uma vez que esta informação é precisamente

a que desejamos utilizar em decisões de processos de alto nível, necessitamos que ela esteja contida em memória, sendo igualmente acessível a outros processos (nomeadamente, o envio). No entanto, o facto de estar em memória e a necessidade de efetuar constantemente cálculos sobre ela, poderão colocar-nos problemas de performance, nomeadamente, devido aos tempos de acesso para leitura e escrita (sem considerar problemas provocados por eventuais colisões de leitura-escrita num mesmo endereço).

Atualização de dados em memória (*PipelineUnit*)

Por forma a contornar este problema, uma eventual solução poderá passar pela adoção de uma *pipeline*, com o objetivo de procurar desempenhar o máximo número de tarefas em paralelo.

Concetualmente, numa *pipeline* existem várias unidades funcionais (componentes) que operam de forma independente e paralelamente entre si. Porém, para que este conceito possa ser aplicado com sucesso, deve ser possível decompor uma tarefa em várias subtarefas, igualmente independentes entre si. Estas subtarefas serão divididas de uma forma uniforme, ao longo de várias etapas (ou fases), tendo como objetivo, um tempo parcial de processamento igual em todas. Normalmente, o número de fases é igual ao número de subtarefas.

Tendo assegurado estas condições, a tarefa será submetida sequencialmente a cada uma das fases, sendo que, quando as tiver percorrido todas, o processamento de uma tarefa estará concluído. A vantagem em dividir uma tarefa em várias fases reside no facto de, enquanto uma tarefa se encontra numa determinada fase a realizar um trabalho específico, outras tarefas poderão estar escalonadas nas restantes fases (tantas tarefas como o número de fases restantes).

De notar que a *pipeline* não irá diminuir o tempo de processamento para uma determinada tarefa. No entanto, permite aumentar o débito (do inglês *throughput*) se considerarmos uma lista de tarefas. Por outro lado, um mau desenho desta poderá aumentar, desnecessariamente, a latência dos dados, aumentando, igualmente, os recursos utilizados para a sua implementação. De igual modo, a *pipeline* por si só, não garante a melhor solução de desempenho, se eventualmente as operações realizadas em cada etapa não forem igualmente otimizadas, como veremos adiante. Antes, importa descrever quais os dados que o módulo *ProcRunUnit* produz, por forma a determinar o comportamento da *pipeline*.

Os dados provenientes do módulo de processamento de *Runs* encontram-se sob a forma de uma instrução, que dirá à *pipeline* qual o tipo de processamento que deverá efetuar. Existem cinco tipos de instruções definidos: *nop*, *new*, *mrg*, *upd* e *dpt*. As instruções encontram-se definidas em 56 *bits*.

A instrução *nop* significa *no operation*, determinando que a *pipeline* não efetuará qualquer tipo de processamento de dados. Esta instrução não é escalada pelo módulo *ProcRunUnit*, pois seria insensato ocupar memória para agendar uma instrução que não realiza trabalho útil. No entanto, no caso de não haver nenhuma instrução disponível para ser processada, a *pipeline* autoinduz esta instrução ao primeiro registo. No caso do sinal *Reset* ser ativado, todos os registos são forçados a esta instrução.

As instruções *new* e *upd* são muito semelhantes entre si, pois ambas apresentam a mesma estrutura e os mesmos dados. O que as distingue é, claro está, o código de instrução (ou operação). A estrutura desta instrução é definida como:

opCode	objNumber	color	start	end	lineNumber
3 bits	*12 bits*	*8 bits*	*11 bits*	*11 bits*	*11 bits*

Se o *opCode* representar uma instrução *new*, os dados contidos em *color*, *start*, *end* e *lineNumber* serão utilizados para o cálculo das características e, uma vez calculadas, os dados serão guardados em memória, utilizando a posição referida por *objNumber*.

No caso do código da instrução ser definido como *update*, o campo *objNumber* identificará o objeto (e, portanto, a posição de memória) que precisamos atualizar. As suas características serão lidas da memória, procedendo-se aos cálculos (considerando as características obtidas da memória e as descritas na instrução) e, posteriormente, salvaguardam-se os novos dados, utilizando a mesma posição de memória, definida por *objNumber*.

Considerando uma instrução *mrg*, os dados referentes ao objeto indicado por *objNumber2* serão utilizados, por forma a atualizar os dados do objeto *objNumber1*. Esta instrução apresenta a seguinte forma:

opCode	objNumber1	objNumber2	
3 bits	*12 bits*	*12 bits*	

Finalmente, a instrução *dpt* diz respeito aos objetos que se encontram completos e que, por esse motivo, podem ser imediatamente enviados para o exterior (tipicamente, ficam disponíveis a um módulo responsável pelo seu envio a um processo de alto nível). Esta intrução é definida simplesmente por:

opCode	objNumber	
3 bits	*12 bits*	

Neste caso, *objNumber* refere somente qual o objeto que desejamos enviar, sendo que os restantes componentes da *pipeline* limitam-se a propagar a informação desse objeto.

Antes de descrevermos a implementação da *pipeline*, veremos como calcular as características dos objetos e como representá-los em memória.

O cálculo supracitado pode ser dividido em duas partes. Por um lado, as características calculadas até ao momento e guardadas em memória, necessitam ser atualizadas com os dados de novas *Runs*. Por outro lado, quando detetada a união de objetos, as suas características precisam ser adicionadas.

As características que estamos interessados em calcular dizem respeito à área, centro de massa e limites do objeto. A área não é mais do que a soma de todos os píxeis de um mesmo objeto. Uma relação de recorrência, que descreve a atualização do cálculo da área até um dado momento, em função dos dados de uma nova *Run*, pode ser descrita como

$$A^* = A + (e - s + 1) \tag{5.1}$$

onde as letras e e s significam, respetivamente, as posições de início e fim da *Run*. A formula $(e - s + 1)$ permite, então saber o comprimento da *Run*.

O centro de gravidade pode ser determinado como

$$\overline{x} = \frac{1}{A} \sum_{(u,v)\in\mathbf{R}} u \qquad \overline{y} = \frac{1}{A} \sum_{(u,v)\in\mathbf{R}} v \tag{5.2}$$

onde A se refere à área, e u e v traduzem as coordenadas do objeto. Deste modo, a relação de recorrência poderá ser descrita como

$$x^* = x + \frac{e^2 + e - s^2 + s}{2} \qquad y^* = y + (e - s + 1) * l \tag{5.3}$$

onde l se refere ao número da linha onde se encontra a atual *Run*. Naturalmente, x e y devem ser divididos pela área A, de modo a obter as coordenadas finais que desejamos.

Por sua vez, os limites do objeto dizem respeito ao menor retângulo que se pode inferir, de modo a incluir todos os píxeis do objeto em causa. Estes limites podem ser facilmente determinados, recorrendo a comparações.

A estrutura de dados de cada objeto encontra-se definida como:

- *color*
- *boundingBox* (definido em termos de coordenadas x_1, x_2, y_1 e y_2)
- *massCenter* (definido em termos de coordenadas x e y)
- *numPixels*

Embora as nomenclaturas adotadas, já por si sugiram quais os dados a que se referem, não será demais salientar que *color* irá guardar a cor do objeto, *boundingBox* irá descrever em termos de coordenadas x_1, x_2, y_1 e y_2, os limites definidos do retângulo, de modo a incluir todos os píxeis; *massCenter* descreve igualmente em termos de coordenadas x e y, o centro de gravidade (ou massa) do objeto, e finalmente, *numPixels* irá conter o número total de píxeis desse objeto.

Em termos de implementação, a *pipeline* é composta por quatro etapas de processamento. Na primeira etapa, pretende-se ler os dados relativos à instrução que irá ser processada. Na segunda etapa procuraremos tomar decisões de controlo para as restantes etapas e de forma a controlar corretamente os vários componentes. Esta etapa inclui, igualmente, a leitura dos dados de um objeto a partir da memória, assim como a conversão de parte dos dados da instrução, num novo objeto. Na terceira etapa pretende-se calcular as características dos objetos, enquanto que na quarta, esses mesmos dados serão guardados na memória ou enviados a outro processo, consoante o tipo de instrução considerado. A figura 5.7 ilustra o diagrama de blocos simplificado da *pipeline*.

Relativamente à primeira etapa, esta é composta essencialmente por um FIFO de instruções. À semelhança do que foi descrito em relação ao FIFO do módulo *ProcRunUnit* (capítulo 5.2.5, pág. 62), este FIFO recebe, diretamente, os dados provenientes do anterior módulo, salvaguardando-os nas situações em que a *pipeline* se encontre ocupada. Este FIFO age, ainda, como um mediador de informação entre os dois módulos, pois ambos operam a frequências diferentes, sobretudo devido à complexidade de processamento de informação envolvida.

Na segunda etapa, como já vimos, iremos avaliar a instrução recebida e decidir os sinais de controlo dos restantes componentes, de acordo com a situação. Para tal, a unidade de

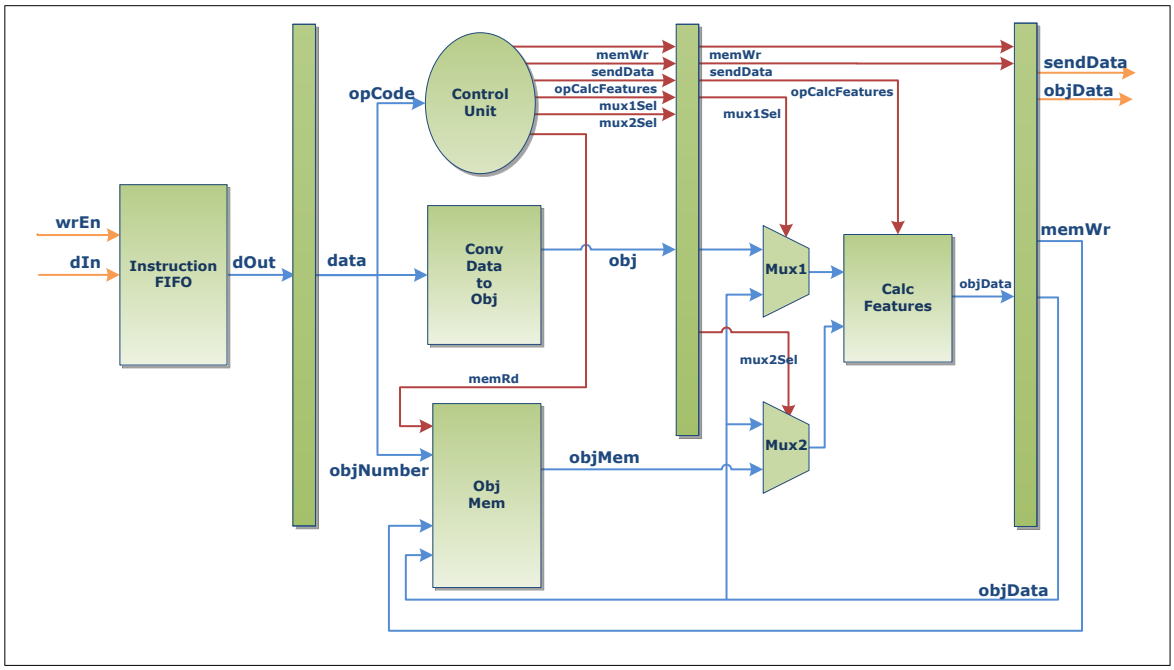


Figura 5.7: Diagrama simplificado de blocos do módulo *BlobAnalysisPipeline*.

controlo avalia, sobretudo, o *opCode* da instrução em causa. Porém, poderão surgir situações em que determinadas instruções se referem a dados de objetos atualizados imediatamente pela instrução anterior. Esta instrução, em termos de visão de *pipeline*, encontra-se na etapa seguinte (etapa de cálculo de características). Nestas situações dizemos estar perante uma dependência de dados.

Dada a forma sequencial de processamento de informação associada a uma *pipeline*, sabemos que, se for ordenada uma leitura do referido objeto, a partir da memória, os seus dados não se encontram atualizados, pois o resultado dos cálculos ainda não foi escrito em memória. Por outro lado, quando a atual instrução se encontrar na fase de cálculo, os dados atualizados pela instrução anterior, estarão somente a ser escritos em memória. No entanto, esta escrita está associada uma latência, pelo que a informação escrita nesse ciclo de relógio (ou ciclo de processamento), somente se tornará efetiva, no próximo ciclo. Isto acontece, porque considerando o mesmo ciclo de processamento e latências diferentes associadas às operações de escrita e leitura em memória, não podemos inferir com certezas, que a operação de leitura apenas acontecerá posteriormente à operação de escrita. Para além desta incerteza, precisamos ainda considerar um tempo de estabilização para os dados.

Uma solução para este problema, consiste em verificar a dependência de dados na unidade de controlo. Caso esta dependência seja verificada, podemos, de imediato, preparar a próxima etapa, por forma a que, em vez de considerar os dados do objeto lidos da memória (relembremos que estes não se encontram ainda atualizados), considere os dados atualizados que se encontrarão disponíveis na última etapa (mesma etapa em que são escritos em memória). Deste modo estamos a assegurar que processamos e produzimos dados corretos.

Nesta etapa, temos ainda um módulo de conversão de dados. Este módulo será responsável por converter os dados presentes na própria instrução da *pipeline*, para uma estrutura temporária do tipo objeto. Esta conversão tem o propósito de facilitar os cálculos posteriores.

Naturalmente, esta conversão só acontecerá para instruções do tipo *new* e *upd*, dado que são as únicas instruções que contêm dados do tipo *Run*.

Outro componente igualmente presente nesta etapa é a memória de objetos. Embora o seu sinal de leitura dependa da unidade de controlo, o endereço de leitura de um objeto é definido pelo campo *objNumber* presente nas instruções *upd* e *dpt*, ou *objNumber1* e *objNumber2*, no caso particular de uma instrução tipo *mrg*. No entanto, nas instruções *upd* e *mrg*, o campo *objNumber* identifica igualmente o endereço de escrita, onde serão guardados os dados atualizados. No caso particular de uma instrução *new*, o campo *objNumber* identifica, exclusivamente, o endereço de escrita. Para a atual instrução presente nesta etapa, a leitura (se necessária) é ativada pela unidade de controlo, de modo a que os dados se encontrem disponíveis para a próxima fase. No entanto, os sinais de escrita (ativação e dados) presentes à entrada da memória dizem respeito à instrução que se encontra na quarta etapa da *pipeline*. Esta situação é expectável e baseia-se nos princípios fundamentais de uma *pipeline*.

Na terceira etapa encontram-se dois *multiplexers* de seleção de dados e a unidade de cálculo das características. Os *multiplexers* permitem seleccionar os dados dos objetos provenientes da anterior fase, ou considerar os dados atualizados recentemente e que se encontram na fase seguinte (escrita em memória). Esta seleção depende dos sinais de controlo determinados pela unidade de controlo, tal como foi mencionado anteriormente.

Na última fase, procede-se à escrita dos dados atualizados para a memória (no caso de instruções *new*, *upd* e *mrg*). No caso específico da instrução *dpt*, os dados previamente lidos da memória são disponibilizados nos sinais de interface com o exterior, ficando deste modo, disponíveis para um outro processo. Esta situação é denominada de "enviar os dados completos".

5.2.6 Envio de dados (*Data Serializer*)

Os dados processados pelo sistema possuem, tipicamente, uma dimensão superior ao tamanho máximo permitido para envio (8 *bits*, sendo que em cada instante de tempo, a UART só permite enviar um bit). Assim, importa dividir os conjuntos de dados a enviar em tamanhos devidamente apropriados. Essa será a função deste módulo.

A implementação deste módulo é muito semelhante à implementação de um *multiplexer* de dados, cuja função consiste em, mediante um sinal de seleção, colocar à sua saída, determinado sinal. No entanto, neste caso específico, este módulo precisa conter alguma lógica adicional de controlo. Isto acontece porque nem todos os sinais possuem tamanhos idênticos e, portanto, é necessário saber como subdividir o sinal, tendo em conta os seus limites.

Os dados a enviar, encontram-se num FIFO. Sempre que houver dados para enviar, este módulo é executado e, de acordo com o sinal de seleção proveniente dos interruptores da FPGA (definidos como "modo de dados"), é dividida a informação disponível em blocos de 8 *bits*. A ordem de envio adotada assume o envio do bloco mais significativo, sucedido do bloco menos significativo (por exemplo, bloco[15-8] seguido de bloco[7-0]). Naturalmente, o número de blocos a enviar depende do tamanho do sinal a enviar (definido pelo "modo de dados"). No entanto e por forma a garantir alguma coerência, todos os sinais são definidos em múltiplos de 8 *bits*. Esta solução permite-nos, por um lado, reutilizar o FIFO, onde são guardados os dados disponíveis para envio (desde que a frequência do módulo que os gera seja,

naturalmente, compatível com a frequência de escrita do FIFO); e, por outro lado, simplificar a lógica de controlo dos módulos posteriores, que processem, eventualmente, estes sinais.

5.3 Comunicação série (UART)

Numa primeira fase, e dado que a ferramenta ISE não dispõe de nenhum *IP Core* base que pudesse ser incorporado e configurado sobre a Atlys, foi criado um módulo de comunicação, com base num tutorial encontrado online. Apesar de nos testes realizados inicialmente, este primeiro módulo ter funcionado corretamente, foi possível observar, posteriormente, comportamentos anómalos e nem sempre consistentes entre si. Em alguns momentos, a informação enviada e recebida era consistente, todavia, repetindo a experiência, em alguns momentos havia perda de informação. Após vários testes realizados e sem uma certeza quanto à origem do problema, foi incorporado um novo módulo, cedido gentilmente pelo orientador Dr. Arnaldo Oliveira. Este módulo permitiu averiguar as causas do problema e despistá-lo com sucesso.

A sua implementação segue a especificação base da comunicação série UART. Adicionalmente a esta interface, encontra-se implementado um FIFO, onde são guardados os dados prontos a enviar (provenientes do módulo *Data Serializer*). De momento, o projeto somente utiliza esta interface para envio de dados, não processando eventuais dados provenientes, no sentido inverso.

A interface encontra-se configurada para uma ligação com velocidade de 115.200 *bps*, 8 *bits* de dados, 1 *stop bit*, sem paridade.

5.4 Projeto base *VmodCAM_Ref_HD*

No site da Digilent[1] é possível encontrar dezenas de projetos, disponíveis para *download* e totalmente gratuitos. O ficheiro descarregado é, habitualmente, um ficheiro comprimido e contém uma lista de ficheiros, que inclui (entre outros), código fonte, configurações, esquemas técnicos e ficheiros de texto com algumas notas importantes a considerar. Na grande maioria dos casos, este ficheiro compreende ainda o ficheiro .bit (*bitstream*), o qual permite, de imediato, programar a plataforma de desenvolvimento em questão e comprovar a sua funcionalidade.

Estes projetos encontram-se normalmente associados aos diferentes produtos comercializados por este fabricante. A sua complexidade pode ser maior ou menor, dependendo sobretudo, da plataforma final para que foi concebido e do componente ou da funcionalidade a demonstrar. Por outro lado e tal como referido no parágrafo anterior, estes projetos permitem comprovar a funcionalidade dos produtos em questão, mas é sobretudo na vertente didática que se revelam verdadeiramente interessantes, sendo uma ajuda e uma forma encorajadora de aprender, através de exemplos, e um convite à experimentação.

No caso concreto do módulo *VmodCAM*[2], para além do diagrama técnico e do manual de referência, são cedidos, ainda, dois projetos de exemplo, disponíveis individualmente nas

[1] `http://www.digilentinc.com`, acedido em 15/05/2013

[2] `http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,648,931&Prod=VMOD-CAM`, acedido em 15/05/2013

versões *Xilinx ISE Design Studio 13* e *Xilinx ISE Design Studio 12* (e versões inferiores). Um dos projetos configura ambas as câmaras presentes no módulo *VmodCAM*, para uma resolução VGA (640x480), armazenando, posteriormente, as imagens na memória (*frame buffer*), e disponibilizando na porta HDMI *Out*, a saída de uma das câmaras, ou ambas, simultaneamente. O outro projeto é em tudo muito semelhante, na medida em que implementa a mesma funcionalidade, com a diferença de configurar as câmaras para a resolução máxima (1600x1200). Este projeto em particular, apenas permite visualizar a saída das câmaras de forma individual, sendo que a imagem é ajustada à resolução máxima implementada do monitor (1600x900).

Importa salientar, que no caso particular do projeto de resolução VGA, a versão disponibilizada para *Xilinx ISE Design Studio 12* encontra-se incompleta. O ficheiro de projeto não inclui os ficheiros base de criação e configuração do IP *Core DCM Fixed*, que permite desdobrar o sinal de relógio base, em frequências mais baixas. Quer isto dizer que, embora seja possível visualizar e testar a funcionalidade, simplesmente programando a plataforma de desenvolvimento com o ficheiro *bitstream*, a alteração do projeto, a fim de testar novas configurações ou mesmo diferentes funcionalidades, implica algum trabalho adicional. Ainda assim, é perfeitamente possível reconstruir o IP *Core* em causa, por exemplo, criando uma nova *DCM Fixed* (utilizando o mesmo nome), com base nas configurações utilizadas pela *DCM Fixed* da versão 13 do mesmo projeto.

Como base para este trabalho, foi utilizado o código disponibilizado para a resolução máxima (1600x1200) e versão ISE 12, uma vez que o ambiente de desenvolvimento é o *Xilinx ISE Design Studio 12.4*. O código incluído neste projeto foi, numa primeira fase, alvo de estudo e análise, com recurso a diversos testes, a fim de comprovar a correta funcionalidade e assimilação de conhecimento. Este código possui alguns sinais declarados que são desnecessários, resultantes de um possível lapso, aquando da limpeza final de código, por parte do programador. Contudo, estes sinais, ainda que declarados e não utilizados, não são preocupantes, na medida em que o próprio ISE não os inclui no processo de sintetização.

5.4.1 Arquitetura simplificada

Seguidamente, será apresentado um esquema simplificado da arquitetura implementada neste projeto, bem como uma descrição sumária dos componentes mais importantes. Posteriormente, serão descritas as alterações realizadas, por forma a torná-lo projeto base à nossa arquitetura.

Na figura 5.8 é apresentada a arquitetura simplificada do projeto *VmodCAM_Ref_HD*. Como podemos observar, este é constituído por 6 módulos principais, os quais, analisaremos seguidamente.

System Controller

Os sinais de entrada *Clock*, *Reset* e interruptores, são diretamente ligados a este módulo, que será responsável por sincronizá-los de forma adequada e, por sua vez, propagar novos sinais de relógio (*Clock*) aos restantes componentes do projeto. Este módulo inclui a instanciação de duas DCMs (*Digital Clock Manager*): DCM Fixed e DCM Recfg. A primeira visa criar

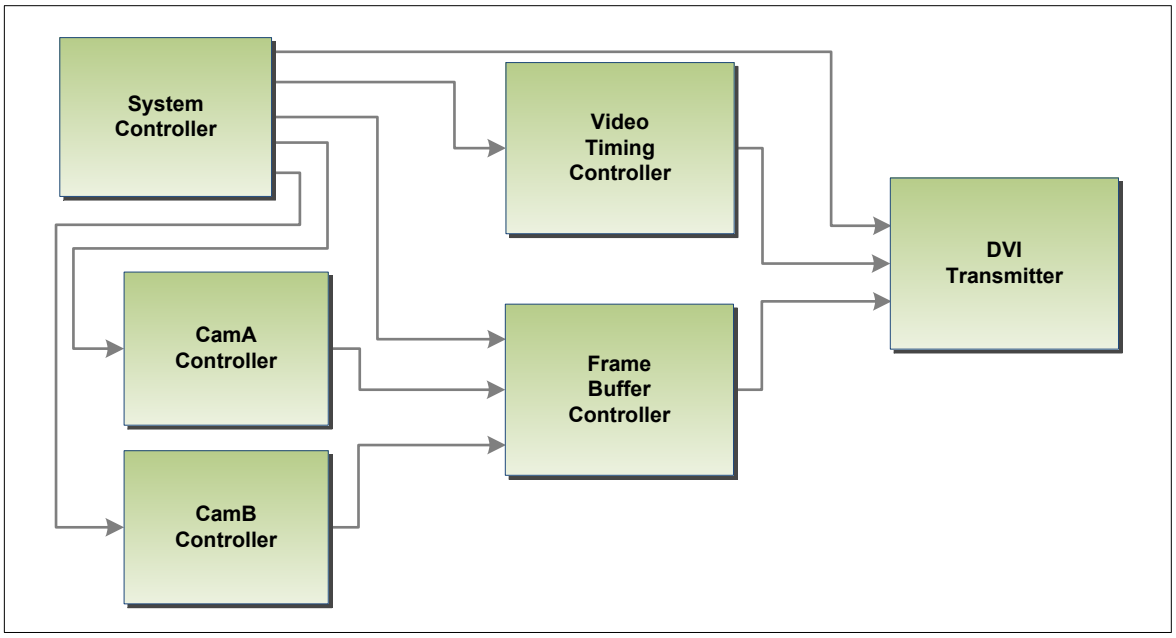


Figura 5.8: Arquitetura simplificada do projeto *VmodCam_Ref_HD*.

novos sinais de relógio para as câmaras, com uma frequência de 24 MHz e, consequente, sinal de relógio para escrita do *frame buffer*. A segunda será responsável por criar novos sinais de relógio para os módulos *Video Timing Controller*, leitura do *frame buffer* e *DVI Transmitter*. Esta DCM é reconfigurável dinamicamente, de modo a permitir diferentes frequências para cada píxel e diferentes formatos de vídeo. Este módulo inclui, ainda, uma PLL (*Phase-Locked Loop*), responsável por gerar um sinal de relógio devidamente alinhado em fase e com uma frequência superior ao sinal de relógio produzido pela DCM Recfg, que irá alimentar especificamente os *serializers* incluídos no módulo DVI Transmitter.

CamA e CamB Controllers

Estes módulos são responsáveis por configurar todos os parâmetros das câmaras e providenciam, igualmente, uma interface simples para leitura da informação recolhida pelas mesmas. Cada módulo é responsável por controlar somente uma das câmaras, permitindo deste modo, e se necessário, aquisições de imagem distintas, mediante configurações diferentes.

Video Timing Controller

Este módulo é responsável por gerar os sinais de sincronização (por exemplo, sincronização horizontal e vertical), para a porta HDMI *Out*, de acordo com a resolução pretendida.

Frame Buffer Controller

Este módulo é responsável por encapsular e gerir convenientemente o espaço destinado a guardar temporariamente a informação recolhida pelas câmaras. Para tal, aloca dois *frame buffers* separados, utilizando a memória DDR2 presente na plataforma Atlys. Cada um destes *frame buffers* inclui um FIFO para escrita da *stream* de dados. A leitura de dados é efetuada

por intermédio de um outro FIFO, comum aos dois *frame buffers*, sendo que um interruptor seleciona qual o *frame buffer* específico a ler.

DVI Transmitter

Este módulo inclui três *TMDS Encoders* (*Transition Minimized Differential Signaling*) e quatro *serializers*. Os primeiros serão responsáveis por codificar os 8 *bits* de uma dada cor, em 10 *bits*. Os segundos serão responsáveis pelo envio e decomposição da informação de cor, num formato adequado à interface DVI (no caso, rácio de 3 sinais de cor e 1 de relógio, que funciona como sinal de sincronização para o recetor). A implementação interna deste módulo encontra-se de acordo com a especificação oficial da interface DVI. Em [Ped10] poderá encontrar um exemplo muito semelhante, implementado em VHDL e seguindo a mesma especificação, o qual contém uma descrição mais pormenorizada, quer da especificação da interface, quer da própria implementação. A figura 5.9 ilustra um diagrama completo da implementação VHDL da interface DVI, sendo que, demarcado a vermelho, encontra-se uma implementação semelhante à arquitetura interna deste módulo. Nesta figura, o bloco *Image Generator* corresponde aos módulos *CamX Controller* e *Frame Buffer*, enquanto o bloco *Control Generator* é idêntico ao módulo *System Controller*. As frequências de relógio, ilustradas na figura para o bloco *TMDS Transmitter*, são coincidentes com as utilizadas no projeto base em discussão.

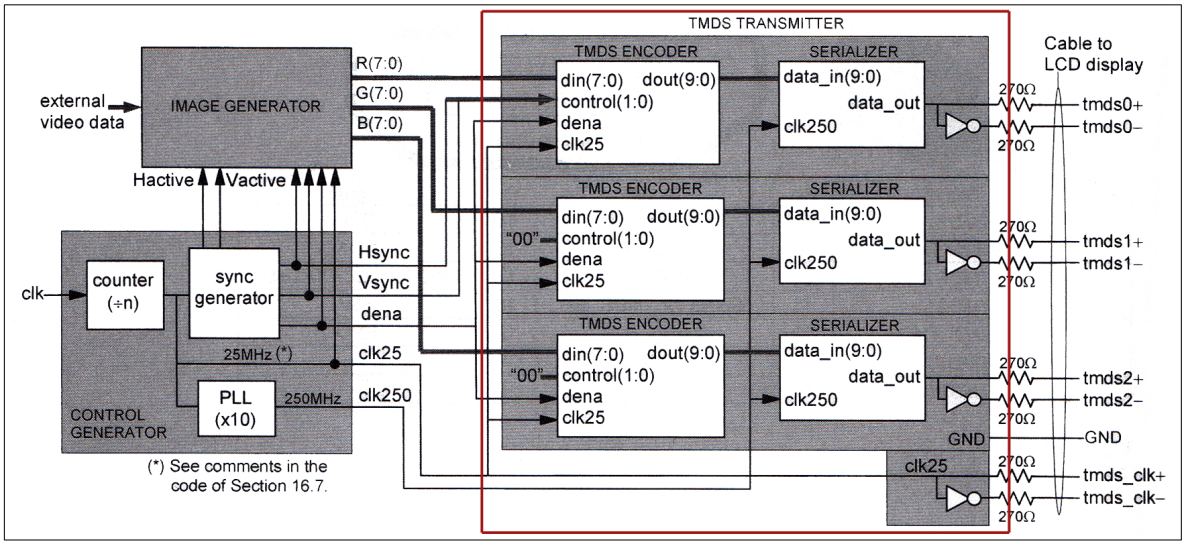


Figura 5.9: Diagrama completo de interface DVI, implementado em VHDL. (fonte: [Ped10], pág. 452)

Capítulo 6

Resultados

6.1 Introdução

Neste capítulo apresentam-se os aspectos a avaliar, assim como os testes realizados e tidos como mais importantes para a valorização e demonstração da funcionalidade dos diferentes módulos que compôem este projeto. De um ponto de vista crítico, pretende-se igualmente, efetuar uma breve discussão dos mesmos.

6.2 Método de teste

Considerando o projeto *VmodCAM_Ref_HD*, o seu teste torna-se mais fácil, pois uma inspeção visual denuncia, de imediato, alguns problemas eventualmente presentes no processamento de imagem. Contudo, no caso concreto dos algoritmos de compressão e deteção de bolhas de cor, este método não se torna eficaz pois, na maioria das vezes, o facto de visualizarmos algo de errado com a imagem, não significa que possamos identificar imediatamente a sua origem. O recurso à simulação possibilita validar o comportamento dos algoritmos desenvolvidos, sendo possível personalizar quais os sinais que desejamos visualizar, por forma a identificar a origem do problema e qual o tipo de simulação que desejamos (comportamental, pós-sintese, pós-mapeamento ou pós-roteamento). No entanto, para grandes conjuntos de dados, a simulação deixa de ser um método totalmente prático. Outros métodos, como o *debug* para ficheiros, com base numa simulação comportamental, são igualmente possíveis, embora na prática não sejam sintetizáveis. O uso de uma interface de comunicação, como a interface de comunicação série, permite que o próprio sistema possa tornar visível os dados internos auxiliares ao processamento e, deste modo, investigar prováveis causas quando nos deparamos com resultados não expectáveis.

O aumento da complexidade do projeto e a dificuldade em identificar os problemas que possam surgir, determinam grandemente o tipo de teste e verificação a aplicar, podendo mesmo resultar numa conjunção de vários métodos.

Ao longo do desenvolvimento deste trabalho foram aplicados vários métodos de teste, dependendo, tal como supracitado, da complexidade do teste, do conjunto de dados a avaliar e deteção de erros. Todos os módulos desenvolvidos foram testados em simulação, ficheiros *log* e sobre a FPGA Atlys. Sobre esta última, excetuando o teste de aquisição e segmentação de imagem, os restantes testes consistiram essencialmente no envio de informação através

da UART. Naturalmente, este tipo de testes implicaram o desenvolvimento de pequenas aplicações responsáveis por receber e formatar convenientemente a informação, de modo a torná-la legível ao utilizador. Para tal, implementaram-se pequenos *scripts* desenvolvidos sobre MATLAB, dado que esta ferramenta revela-se especialmente poderosa quando consideradas operações vetoriais ou matriciais, sendo igualmente extremamente simples de programar para o tratamento de sinais e dados em geral.

6.2.1 *Scripts* MATLAB

Seguidamente, serão apresentados os *scripts* tidos como mais importantes, sendo igualmente os mais utilizados na metodologia de teste ao presente projeto. A sua execução segue as regras gerais de qualquer outro comando ou função invocados em MATLAB. Os *scripts* foram pensados e implementados de forma a complementarem-se entre si, sendo possível desta forma, por exemplo, inicializar a ligação, processar dados oriundos do processamento RLE, BlobAnalysis ou somente imagem segmentada, de uma forma completamente aleatória e autónoma. No presente momento, não existe ainda uma versão mais amigável do utilizador que simplifique as diversas operações, por exemplo através de um menu, permitindo ao utilizador somente selecionar a opção mais adequada. Esta limitação prende-se sobretudo com o facto de os presentes *scripts* servirem um propósito de *debug* e verificação de dados e funcionalidade do sistema, não pretendendo de forma alguma, assumir a tarefa de aplicação final de alto nível para tratamento de dados processados.

Criação de LUT para segmentação (*createLUT.m*)

Tal como já referido no capítulo 4, foi desenvolvido um *script* por forma a criar a LUT utilizada para o processo de segmentação de cores. Este *script* não possui um grande nível de parametrização, em parte porque a sua implementação encontra-se orientada ao problema em questão, assumindo-se com um propósito somente de demonstração e teste. Tal como mencionado anteriormente, métodos mais práticos e eficientes já existem e encontram-se em uso por parte da equipa CAMBADA, podendo os seus resultados ser exportados para um ficheiro, permitindo a posterior inicialização da LUT deste projeto. O formato atual para o ficheiro de inicialização possui extensão *.coe*. Todavia, tanto o formato e tipo de ficheiro exportado, como a própria funcionalidade, poderão facilmente ser adaptados a outras situações tidas como convenientes.

Definição dos parâmetros gerais *globalParameters.m*

Neste ficheiro encontram-se definidos alguns parâmetros gerais e comuns a todos os *scripts*, como sendo a resolução ou as configurações da ligação série. Adicionalmente, existe a possibilidade de ativar ou desativar o registo de todas as comunicações efetuadas durante uma dada ligação. Estes registos encontram-se ativados por omissão e são guardados na pasta *SerialRecordLogs*, sendo que o nome de ficheiro atribuído a cada registo é automático e possui a forma "`<nome da porta COM>_<dia>.<mes>.<ano>_<hora>.<minutos>.txt`" (por exemplo: `COM10_30.04.2013_22.03.txt`).

Suporte da conexão UART (*initCon.m*, *closeCon.m* e *clearReadBuffer.m*)

Tal como sugerido pelo nome de cada *script*, estes visam principalmente gerir a ligação, nomeadamente, o seu estabelecimento, término e limpeza do *buffer* de dados recebidos. Tal como supracitado, os parâmetros afetos à configuração e tamanho do *buffer* de dados são definidos no ficheiro de parametrização *globalParameters.m*. No caso particular do *script* `initCon.m`, este comando não só inicializa a ligação série, como apresenta um pequeno resumo descritivo dos parâmetros utilizados e estado da ligação, como apresentado seguidamente:

```
A tentar inicializar a conexao (COM10)...
Conexao aberta com as seguintes caracteristicas:

   Serial Port Object : COM10ToAtlys

   Communication Settings
      Port:               COM10
      BaudRate:           115200
      Terminator:         'LF'

   Communication State
      Status:             open
      RecordStatus:       on

   Read/Write State
      TransferStatus:     idle
      BytesAvailable:     0
      ValuesReceived:     0
      ValuesSent:         0
```

Validação de dados da imagem segmentada (*rcvSegmentedImg.m*)

Este *script* assume que os dados já existentes no *buffer* ou eventualmente a receber da interface série dizem respeito à imagem segmentada, processando-os de acordo com o formato esperado e, no final, apresentando-os sob a forma de imagem.

Validação de dados da imagem comprimida (*rcvRLE8Data.m*)

Este *script* apresenta-se de forma semelhante ao *script* anteriormente descrito, assumindo que os dados já existentes no *buffer* ou eventualmente a receber da interface série dizem respeito à imagem comprimida no formato RLE8, processando-os de acordo com o formato esperado e, no final, apresentando-os sob a forma de imagem descomprimida.

Validação de instruções sobre objetos (*rcvBlobAnalysis_instruction.m*)

Este *script* assume que os dados já existentes no *buffer* ou eventualmente a receber da interface série dizem respeito ao resultado do processamento do primeiro módulo do processo

de deteção de bolhas de cor, assumindo portanto, a forma de instruções de operação sobre os dados dos objetos.

Validação de dados de objetos (*rcvBlobAnalysis_object.m*)

Este *script* assume que os dados já existentes no *buffer* ou eventualmente a receber da interface série dizem respeito ao resultado do processamento do segundo módulo do processo de deteção de bolhas de cor, representando as características dos diversos objetos.

6.3 Projeto base *VmodCAM_Ref_HD*

Como mencionado no capítulo 4, este projeto, tal como cedido pelo fabricante *Digilent*, visa principalmente demonstrar a funcionalidade e como implementar a interface entre o módulo *VmodCAM* e a FPGA Atlys. Necessariamente, e com o objetivo de validar o seu correto funcionamento, foi adotada uma política contínua de alteração sucedida de respetivo teste. Embora morosa, sobretudo porque os tempos de sintetização do projeto base rondam os 15 minutos, sensivelmente, esta política revelou-se a melhor forma de validar todo o fluxo inicial de trabalho, já que, tal como se depreende, este tipo de alterações são propícias a erros, que não sendo detetados, podem comprometer a eficiência ou o total funcionamento da solução.

Um problema sentido logo de início prende-se com o excessivo aquecimento do *chip* principal da FPGA. Este aquecimento deve-se ao projeto em si, nomeadamente à forma contínua de processamento de dados e recursos utilizados constantemente. A implementação de pequenas aplicações de cálculo sobre a FPGA revelou, igualmente, um aquecimento, embora, não tão exagerado como com o projeto *VmodCAM_HD*. Torna-se portanto óbvio, que o dissipador passivo utilizado na referida FPGA é claramente insuficiente, caso se desejem testar projetos mais complexos em termos de processamento. Uma solução, por forma a contornar este problema, passa por instalar um *cooler* ativo, apesar de ser necessário acautelar a sua fonte de alimentação.

Um outro problema sentido aquando dos primeiros testes com a câmara, prende-se com a sua sensibilidade à iluminação ambiente. As imagens adquiridas revelam uma enorme granularidade e ruído, bem como cores pouco fiéis. Embora nos testes iniciais, esse aspecto não fosse propriamente comprometedor, torna-se, obviamente, um aspecto a ter em conta para o trabalho posterior e respetivos testes.

6.4 Projeto *Vision System Processor*

O projeto base deste trabalho, tal como já mencionado no capítulo 4, baseia-se na implementação do projeto *VmodCAM_Ref_HD*. Para além de diversos testes e alterações com o objetivo de testar a funcionalidade e, igualmente, permitir uma perceção mais profunda da solução implementada, sob um ponto de vista pedagógico, foi igualmente revisto o código e respetivos comentários.

Idealmente, toda a posterior implementação assentará sobre este novo projeto base. Na prática e durante a realização deste trabalho e dos respetivos testes, somente o processo de

segmentação foi implementado e testado sobre este projeto base. Embora o problema da iluminação e do aquecimento sejam realmente algo limitativos, o referido módulo foi testado, validando corretamente os seus resultados. A implementação dos restantes módulos de processamento encontra-se num projeto à parte. Esta decisão permite, não só, contornar temporariamente os referidos problemas durante a fase de implementação e teste dos diversos módulos de processamento da imagem, como torna mais fácil o *debug* de eventuais problemas de implementação. Neste projeto auxiliar, o módulo de aquisição da imagem é substituído por uma pequena memória de inicialização e um processo responsável por ler e alimentar os restantes processos, com a informação contida nessa memória.

6.4.1 Segmentação de cor

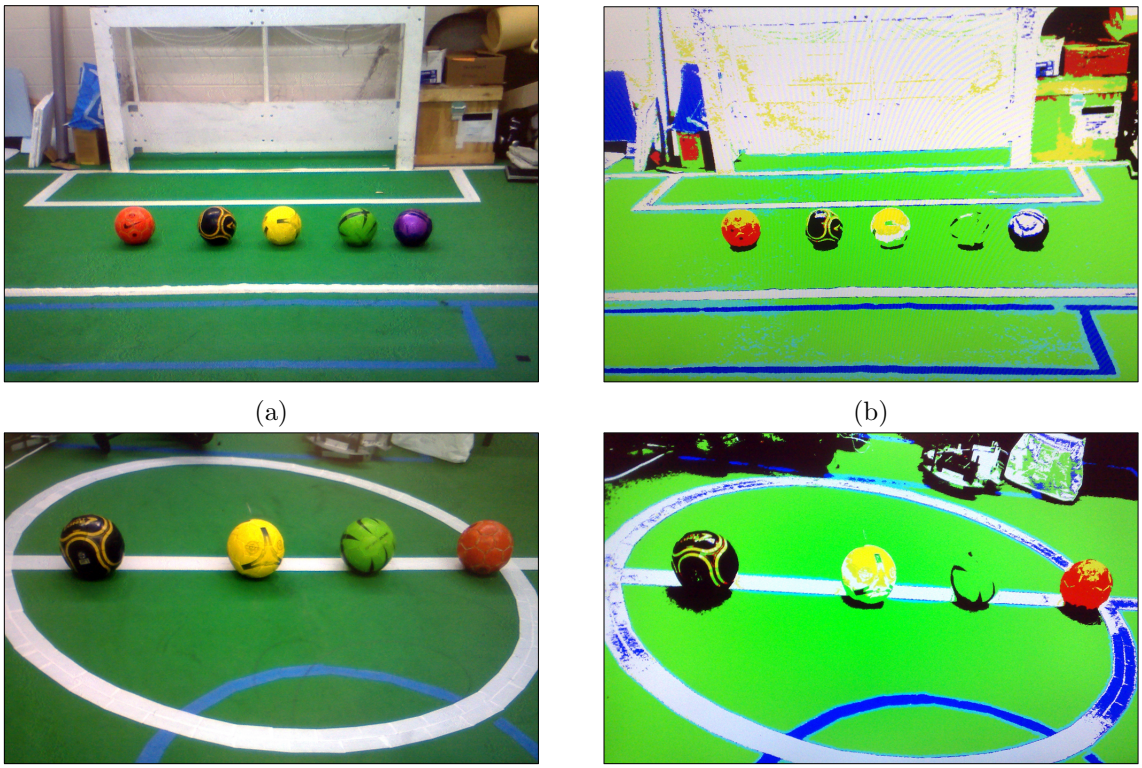
O módulo de segmentação de cor, devido à sua simplicidade de implementação, foi diretamente desenvolvido e testado sobre o projeto base. Devido ao problema supracitado, relacionado com a iluminação ambiente, foram inicialmente realizados vários testes, revelando quase todos eles, alguns problemas no correto reconhecimento das cores. Embora frustada, foi efetuada uma tentativa de alterar os diversos parâmetros da câmara, com vista a procurar obter melhores resultados. Contudo, em diferentes instantes de tempo e perante condições de iluminação igualmente variáveis, não foi possível encontrar uma solução que permitisse, de uma forma geral, obter os melhores resultados.

Para além destes problemas, também o problema do aquecimento revelou a sua existência, acrescentando a possibilidade de afetar definitivamente o *hardware*, pelo que se tornou necessário adotar medidas adicionais, bem como algum cuidado.

A figura 6.1 ilustra alguns exemplos de aquisição de imagem, considerando diversos momentos. Do lado esquerdo são apresentadas as imagens originais, sendo que do lado direito é apresentado o respetivo processo de segmentação aplicado no mesmo momento da aquisição da imagem original. Estas imagens não foram alvo de tratamento gráfico, tendo sido recolhidas pela câmara e posteriormente fotografadas. Embora esta operação introduza um nível de erro que deve ser considerado, foi o único meio de adquirir as imagens, dada a inexistência de implementação de qualquer interface de comunicação.

6.4.2 Compressão de dados (RLE)

Tendo em vista o teste do módulo de compressão RLE8 implementado, foi codificada a imagem original correspondente à figura 6.2, de forma extensa (15 píxeis x 15 píxeis) e inicializada em memória, sendo posteriormente, enviada a este módulo. Naturalmente, foram realizados outros testes de menor dimensão, de modo a assegurar o correto processamento perante outras situações. Esta imagem é particularmente interessante, pelo facto de já apresentar a codificação RLE8 correspondente. Para efeitos de comparação com os resultados obtidos por este módulo, deve-se somente considerar a informação apresentada a azul, já que a restante informação a vermelho se refere à implementação da *Microsoft*. Os resultados obtidos, incluindo os *bits* de sinalização EOL e EOF foram corretamente validados, perante diferentes imagens.



(a) (b)

(c) (d)

Figura 6.1: Exemplo de aquisição de imagem em diversos momentos, com aplicação de processo de segmentação.

6.4.3 Deteção de bolhas de cor

Como já mencionado, este processamento foi dividido em dois subprocessos, procurando isolar, por um lado, a complexidade associada ao processamento das *Runs* e suas eventuais vizinhanças e, por outro lado, o processamento e cálculo das características dos eventuais objetos encontrados. Seguidamente, apresentaremos os resultados dos diferentes processamentos.

Módulo *ProcRunUnit*

Com o objetivo de testar o módulo *ProcRunUnit* e à semelhança do módulo de compressão de dados RLE8, entre outros testes menores, foi utilizada a imagem correspondente à figura 6.2. Apesar da sua reduzida escala, esta imagem serve o propósito de testar todas as funcionalidades presentes por este módulo, validando corretamente a informação de saída e, por conseguinte, as operações de cálculo e processamento das características de cada objeto. O Anexo B, página 103, ilustra o resultado obtido para o teste referido.

Módulo *PipelineUnit*

Apesar de corretamente testado em ambiente de simulação e validado em termos comportamentais, o teste deste módulo sobre a FPGA Atlys revelou alguns problemas de operação

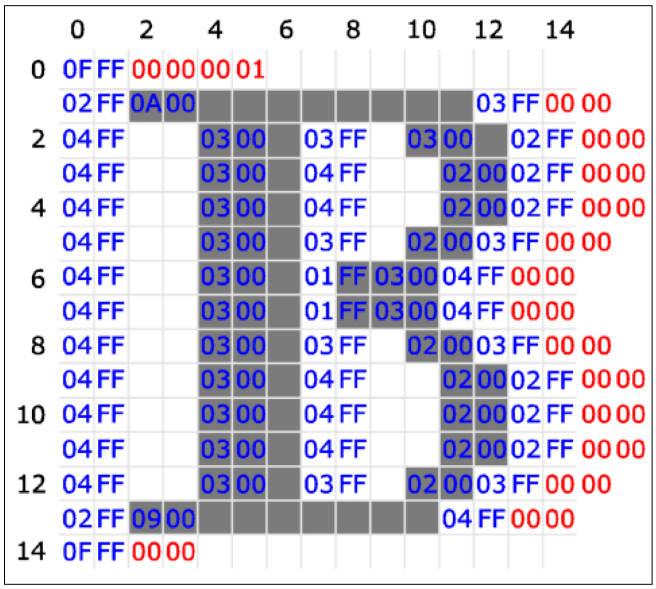


Figura 6.2: Exemplo de imagem codificada usando RLE8 (fonte [Rle]).

com dados incorretos. O problema surge sempre que uma dada instrução gerada pelo anterior módulo necessita de dados que hajam sido atualizados pela instrução imediatamente anterior (tipicamente instruções *upd* e *mrg*). Nestes casos, os dados de um dos objetos necessários encontram-se disponíveis nos registos da fase 4, sendo que a unidade de controlo determina que um dos *multiplexers* presentes na fase 3, encaminhe estes dados e os alimente à unidade de cálculo das características. Contudo, parece haver um problema temporal no acesso a esta informação que, apesar dos elevados esforços, ainda não foi possível resolver. Testes individuais, realizados sobre a FPGA, a cada *multiplexer*, unidade de cálculo e registos não revelam problemas; no entanto, quando interligados, os sinais aparentam instabilidade, o que pode justificar o facto de os dados utilizados não serem corretos. Para instruções que operam sobre dados contidos na própria instrução (instruções *new* e *upd*, esta última sobre um objeto diferente do atualizado por outra instrução anterior), são executadas corretamente e validados os seus resultados. Também o acesso aos dados dos objetos contidos em memória foi corretamente verificado para objetos cujas características haviam sido previamente bem calculadas. Tanto quanto possível e porque este problema limita grandemente a validação de todo o trabalho e a apresentação dos resultados, tal como disponibilizados ao sistema de processamento principal, serão movidos esforços para resolver o presente problema.

A figura D.1 incluída no Anexo D (página 109) ilustra a simulação e validação em termos comportamentais. Os dados apresentados referem-se à segunda instrução considerada para a imagem de teste "B", a qual determinada a atualização dos dados do objeto 0 (zero) com os dados da atual *run* (`UPD objnumber(0) color(255) start(0) end(1) lineNumber(1)`). Como se pode observar, a unidade de cálculo determina corretamente as várias características. Contudo, como ilustrado no Anexo C, página 107, quando executado o mesmo método sobre a Atlys, o objeto possui os dados incorretos.

Capítulo 7

Conclusão e trabalho futuro

7.1 Conclusão

O presente projeto visa a redução do peso computacional da visão por computador da equipa CAMBADA, atualmente implementada no computador principal de cada robô, através da implementação de parte dos algoritmos de visão em *hardware* reconfigurável. No sistema desenvolvido, o *hardware* reconfigurável, baseado em FPGA, assume-se como um coprocessador, libertando deste modo, o computador principal de cada robô, das tarefas computacionalmente intensivas, explorando quer o paralelismo, quer os recursos e as demais técnicas associadas às FPGAs.

Para a elaboração de uma solução, foram estudados os diversos sistemas de visão computacional, nomeadamente, o sistema omnidirecional, um dos sistemas mais presentes na competição RoboCup MSL (onde se insere a equipa CAMBADA), bem como as suas características. Complementarmente a este estudo, foram avaliados tanto os diversos sistemas de visão adotados pelas principais equipas presentes na competição, como a sua implementação ou os métodos utilizados para a resolução de determinado problema. Foram igualmente pesquisados e analisados diversos artigos, teses e trabalhos sobre processamento de imagem e, quando existentes, métodos já explorados para a sua implementação sobre *hardware* reconfigurável.

Simultaneamente e de uma forma mais prática, foram igualmente testados o módulo de aquisição de imagem *VmodCAM* e a sua interligação com a FPGA Atlys. O projeto disponibilizado pelo fabricante Digilent, depois de devidamente modificado e comentado, foi adotado como projeto base, sobre o qual foi implementado e testado o módulo de segmentação de cor. Os restantes módulos de compressão de imagem e análise e deteção de bolhas de cor, devido à sua complexidade de processamento e à necessidade de efetuar *debug*, a fim de resolver diversos problemas, foram implementados e testados sobre um projeto auxiliar, o qual, através de um processo de leitura e de uma imagem pré-carregada em memória, se encarrega de alimentar os dados necessários ao consequente processamento. De todos os módulos implementados e testados, a parte final do processamento do módulo de deteção de bolhas de cor apresenta-se como o único problema de implementação conhecido e ainda não resolvido. Embora em ambiente de simulação e em termos comportamentais, a sua funcionalidade tenha sido verificada e validada, o referido módulo, quando sintetizado e programado sobre a FPGA, revela problemas de cálculo das características dos objetos, gerando resultados incorretos. Dado que a funcionalidade no presente momento, ainda não é totalmente correta, uma avaliação de de-

sempenho ou de recursos utilizados, torna-se precária, sendo que pequenas alterações poderão afetar estas medidas. Deste modo, teceremos somente algumas considerações relativamente ao desempenho esperado.

Uma análise informal ao módulo de segmentação de cor permite perceber que este, por adotar um processamento *raster scan*, possibilita o resultado de processamento de um determinado píxel no ciclo de relógio imediatamente seguinte ao da sua disponibilidade como dado de entrada. No caso do módulo de compressão de dados, ainda que o seu processamento adote igualmente uma política *raster scan*, uma vez que este analisa sequências repetidas de uma mesma cor, nem sempre teremos dados à sua saída de forma contínua. Contudo, esta situação não afeta o seu processamento interno, pois do seu ponto de vista, cada píxel que fique disponível à entrada, é processado sincronamente e o seu resultado (ainda que não visível ao exterior por meio da porta de saída) encontra-se disponível no ciclo seguinte. No caso concreto do módulo de deteção de bolhas de cor, nomeadamente do módulo *ProcRunUnit*, considerando um determinado conjunto de dados à sua entrada, o resultado do seu processamento somente estará disponível 5 ciclos de relógio depois. Contudo, para o mesmo conjunto de dados de entrada, este módulo poderá necessitar de mais ciclos de relógio para processar outros resultados. Esta situação acontece no caso de serem detetadas várias vizinhanças para um mesmo objeto, sendo que a primeira é entendida como o objeto a atualizar e as restantes, como objetos tidos até ao momento como diferentes, mas que afinal pertencem ao mesmo objeto considerado. Se o número de vizinhanças detetado for superior a 2, o número de ciclos de relógio necessários ao processamento aumenta consoante o número de vizinhanças adicionais encontradas. No entanto, para que esta situação não comprometa todo o processamento, este módulo utiliza o triplo da frequência do módulo de compressão de dados (cujo resultado é alimentado como entrada de dados do atual módulo). Esta abordagem, conjuntamente com o facto de nem sempre termos dados à saída do RLE8 (e portanto, disponíveis para processar neste módulo), deverão assegurar um nível de desempenho assinalável. A segunda parte do processamento adota um mecanismo de *pipelining*, pelo que, considerando que todas as fases se encontrem a processar um conjunto diferente de dados, o máximo débito encontra-se assegurado. Não sendo o caso, precisamos considerar que cada objeto a ser processado necessita de 4 ciclos de relógio. Necessariamente, importa ainda considerar outros tempos, como os associados aos FIFOS intermédios que permitem guardar e mediar a informação entre diferentes processos.

Relativamente ao módulo *VmodCAM* e à FPGA Atlys, importa ainda considerar os problemas enunciados quanto a estes. O problema de aquecimento merece uma especial atenção, sobretudo pelos danos físicos que poderão advir para o *hardware*. O problema da iluminação e consequente afetação da qualidade de imagem poderá igualmente mostrar-se como limitativo aos posteriores módulos de processamento de imagem. Contudo, este último poderá ser contornado implementando um processo de configuração dos diversos parâmetros da câmara, com base na análise periódica de uma imagem recolhida (à semelhança do que já acontece com a atual solução adotada pela equipa CAMBADA).

No que à implementação diz respeito, um problema detetado e até ao presente momento, ainda não resolvido, prende-se com a segunda parte do processamento da deteção de bolhas de cor, nomeadamente, com o cálculo das características dos objetos sobre a *pipeline*. Tal como já mencionado anteriormente, ainda que corretamente validado em simulação e em termos comportamentais, o cálculo e atualização de dados de objetos revela dados incorretos, para instruções *upd* ou *mrg* que referenciem objetos atualizados pela instrução imediatamente anterior. Este problema limita a avaliação de desempenho da presente implementação, uma

vez que, quase todos os objetos terão as suas características calculadas erradamente.

Embora testes e medidas de desempenho efetivas não possam ser apresentados de momento e, sob o ponto de vista de funcionalidade, o presente projeto apresente ainda o problema de dados calculados incorretamente, a solução, na sua globalidade, e para os módulos testados e validados, permite demonstrar a aplicabilidade e a supremacia da implementação de tarefas computacionalmente intensivas sobre *hardware* reconfigurável, tirando máximo partido dos mecanismos de paralelismo e *pipelining*. De igual forma, quer o trabalho desenvolvido, quer as sugestões e considerações de trabalho futuro, poderão ser uma mais valia para futuras soluções que visem o desenvolvimento de um sistema de visão mais avançado.

7.2 Trabalho futuro

Iremos agora apresentar algumas sugestões de melhorias ao trabalho desenvolvido, bem como eventuais implementações em falta.

Configuração dos vários parâmetros da câmara (módulo *VmodCAM*)

Na atual implementação, os parâmetros de configuração da câmara encontram-se definidos numa pequena memória interna ao módulo *VmodCAM*. Relativamente a estes parâmetros, e por forma a tornar o sistema o mais autónomo possível, importa, sobretudo, implementar um módulo que analise a imagem capturada e ajuste os parâmetros da câmara, de forma automática (à semelhança do atual sistema implementado na equipa CAMBADA). Contudo, por questões de tempo, não foi possível implementar este módulo.

Outra melhoria prevista no presente trabalho visa a configuração remota destes parâmetros. Esta melhoria pretende, sobretudo, permitir a visualização dos atuais valores definidos e a sua alteração (por exemplo, por meio de registos). Deste modo, tarefas como alterar a resolução da câmara ou o formato da imagem, tornam-se mais acessíveis e não implicam a sintetização de todo o projeto.

Segmentação de cor (LUT)

Na atual implementação, esta memória (LUT) é inicializada com recurso ao ficheiro exportado pelo *script* em MATLAB, sendo necessário sintetizar o projeto de modo a refletir eventuais alterações ao conteúdo do mesmo. Igualmente, durante a execução do projeto sobre a plataforma, somente são permitidos acessos de leitura. Contudo, num trabalho futuro será interessante tornar esta inicialização independente da sintetização, aceitando, por exemplo, uma nova configuração de LUT, via UART. Para tal, será necessário mudar o tipo de memória implementada (ROM para RAM), de modo a permitir a escrita de novos dados, durante a execução. É igualmente necessário criar um módulo que receba os dados, através de uma interface de comunicação (por exemplo, UART), os valide e atualize esta memória.

Imagem de máscara

À semelhança do módulo de segmentação, também a memória presente neste módulo é inicializada com recurso a um ficheiro externo, sendo necessário sintetizar novamente o projeto, de modo a refletir eventuais alterações ao conteúdo desse ficheiro. Durante a execução,

não será esperada (em condições tidas como normais) a alteração do conteúdo dessa memória. Todavia, seria igualmente interessante que também este processo de inicialização pudesse ser independente da sintetização do projeto.

A esta situação em particular, poderia acrescer a preocupação do espaço reservado em memória para a imagem não ser suficiente, caso a dimensão da mesma aumentasse. Todavia, é uma situação pouco frequente, dado que nada leva a crer que seja necessário efetuar esse tipo de alteração com regularidade. Dado que o restante sistema já se encontra devidamente parametrizado, basta definir a quantidade de memória alocada em função dos parâmetros já existentes. Por forma a inicializar os dados em memória, o tipo desta deve ser alterado para RAM, devendo ser criado um processo que receba os dados através de uma interface de comunicação (por exemplo, UART), os valide e atualize em memória.

Processamento de dados recebidos via UART

Tal como já mencionado nos pontos anteriores, a implementação de um módulo que pudesse tratar os dados recebidos remotamente, via comunicação série (UART), torna-se uma mais-valia. Deste modo, seria sobretudo possível configurar parâmetros internos e carregar máscaras de imagem ou LUT diferentes.

Bibliografia

[Aan+09] W. Aangenent, J. Best, B. Bukkems, F. Kanters, K. Meessen, J. Willems, R. Merry e M. Molengraft. *Tech united eindhoven team description 2009*. Inglês. Rel. téc. Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven - The Netherlands: Control Systems Technology Group, Eindhoven University of Technology, 2009 (ver p. 23).

[Amm+12] T. Amma, J. Beifuβ, F. Gawora, K. Geihs, S. Jakob, D. Kirchner, N. Kubitza, K. Liebscher, C. Luehring, S. Opfer, R. Reichle, D. Saur, F. Seute, H. Skubch, S. Triller, D. Walden, M. Wetzel e A. Witsch. *Carpe Noctem 2012*. Inglês. Rel. téc. D-34121 Kassel, Germany: Distributed Systems Group, University of Kassel, 2012 (ver p. 27).

[Atla] *Atlys Board Reference Manual*. Inglês. Digilent Inc. 1300 Henley Court, Pullman, WA 99163, dez. de 2011. URL: `http://www.digilentinc.com/` (acedido em 25/05/2013) (ver p. 98).

[Bak+99] S. Baker e S. K. Nayar. ≪A theory of single-viewpoint catadioptric image formation≫. Inglês. Em: *International Journal of Computer Vision* (1999(2)), pp. 175–96 (ver p. 43).

[Ben+01] R. Benosman e S. B. Kang. *Panoramic vision: sensors, theory and applications*. Inglês. New York: Springer-Verlag. 2001 (ver p. 21).

[Bes+10] J. Best, D. Bruijnen, R. Hoogendijk, R. Janssen, K. Meessen, R. Merry, M. Molengraft, G. Naus e M. Ronde. *Tech united eindhoven team description 2010*. Inglês. Rel. téc. Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven - The Netherlands: Eindhoven University of Technology, 2010 (ver p. 23).

[Rle] *Binary Essence*. Inglês. URL: `http://www.binaryessence.com/dct/en000073.htm` (acedido em 21/05/2013) (ver pp. 18, 19, 59, 83).

[Bla+00] T. Blaffert, S. Dippel, M. Stahl e R. Wiemker. ≪The Laplace integral for a watershed segmentation≫. Inglês. Em: *Proc of the international conference on image processing, 2000*. Vol. 3. 2000, pp. 444–7 (ver p. 46).

[Boc+09] A. Bochem, R. Herpers e Kenneth Kent. *Acceleration of Blob Detection Within Images in Hardware*. Inglês. Rel. téc. NB, E3B 5A3; Canada: Faculty of Computer Science, University of New Brunswick Fredericton, 2009 (ver p. 34).

[Bra+06] I. Bravo, P. Jiménez, M. Mazo, J. Lázaro e E. Martín. *Architecture Based on FPGAs for Real-Time Image Processing*. Inglês. Rel. téc. Alcalá de Henares (Madrid), Spain: Electronics Department, University of Alcalá, 2006 (ver p. 33).

[Bre65] J. E. Bresenham. ≪Algorithm for computer control of a digital plotter≫. Inglês. Em: *IBM Systems Journal* 4(1) (1965), pp. 25–30 (ver p. 45).

[Can86] J. F. Canny. ≪A computational approach to edge detection≫. Inglês. Em: *IEEE Transactions on Pattern Analysis and Machine Intelligence vol. 8 (number 6)* (1986) (ver p. 46).

[Chi+11] Priyanka Chikkali e K. Prabhushetty. ≪FPGA based Image Edge Detection and Segmentation≫. Inglês. Em: *(IJAEST) International Journal of Advanced Engineering Sciences and Technologies, Vol. 9, Issue 2* (2011) (ver p. 34).

[Cun+07a] Bernardo Cunha, José Luís Azevedo, Nuno Lau e Luís Almeida. ≪Obtaining the inverse distance map from a non-SVP hyperbolic catadioptric robotic vision system≫. Inglês. Em: *Proc of the RoboCup 2007*. Atlanta (USA), 2007, pp. 417–24 (ver p. 43).

[Cun+07b] Bernardo Cunha, José Luís Azevedo, Nuno Lau e Luís Almeida. ≪Obtaining the inverse distance map from a non-SVP hyperbolic catadioptric robotic vision system≫. Inglês. Em: *Proc of the RoboCup 2007*. Atlanta (USA), 2007, pp. 417–24 (ver p. 47).

[Gho+11] M. Gholipour, S. Ebrahimijam, H. Fard, M. Montazeri, A. Zanjaninejad, S. Moein, B. Eskandariun, M. Mahmoodi, H. Ziarati, A. Zarineh, M. Tabrizi e A. Aryandust. *MRL Middle Size Team: RoboCup 2011 Team Description Paper*. Inglês. Rel. téc. IRAN: Mechatronics Research Laboratory, Islamic Azad University of Qazvin, 2011 (ver p. 24).

[Gon+09] Griselda González e M. Estrada. *FPGA Based Acceleration for Image Processing Applications*. Inglês. Rel. téc. Puebla, Mexico: National Institute for Astrophysics, Optics e Electronics (INAOE), 2009 (ver p. 33).

[Haf+09] R. Hafner, S. Lange, M. Riedmiller e S. Welker. *Brainstormers Tribots Team Description*. Inglês. Rel. téc. 49069 Osnabrück, Germany: Neuroinformatics Research Group, Institute of Computer Science e Institute of Cognitive Science, University of Osnabrück, 2009 (ver pp. 25, 36).

[Hon] *Honda - News Releases 2012*. Inglês. URL: `http://world.honda.com/news/2012/p120820Honda-Miimo/` (acedido em 08/07/2013) (ver p. 2).

[Hoo+12] R. Hoogendijk, G. Naus, F. Schoenmakers, C. Martinez, G. Heldens, J. Hoen, R. Merry e M. Molengraft. *Tech united eindhoven team description 2012*. Inglês. Rel. téc. Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven - The Netherlands: Eindhoven University of Technology, 2012 (ver pp. 23, 24, 36).

[Hua+12] M. Huang, X. Ge, S. Hui, X. Wang, S. Chen, X. Xu, W. Zhang W, Y. Lu, X. Liu, L. Zhao, M. Wang, Z. Zhu, C. Wang, B. Huang, L. Ma, B. Qin, F. Zhou e C. Wang. *Water Team Description Paper 2012*. Inglês. Rel. téc. China, 100192: Beijing Information Science & Technology University, 2012 (ver p. 25).

[Iro] *iRobot Corporation: Robots that make a difference*. Inglês. URL: `http://www.irobot.com/us/` (acedido em 08/07/2013) (ver p. 2).

[Joh+04] C. Johnston, K. Gribbon e D. Bailey. *Implementing Image Processing Algorithms on FPGAs*. Inglês. Rel. téc. Private Bag 11-222, Palmerston North, New Zealand: Institute of Information Sciences & Technology, Massey University, 2004 (ver p. 34).

[Kan+11] F. Kanters, R. Hoogendijk, R. Janssen, K. Meessen, D. Best, D. Bruijnen, G. Naus, W. Aangement, R. Berg, H. Loo, G. Heldens, R. Vugts, G. Harkema, P. Brakel, B. Bukkums, R. Soetens, R. Merry e M. Molengraft. *Tech united eindhoven team description 2011*. Inglês. Rel. téc. Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven - The Netherlands: Eindhoven University of Technology, 2011 (ver pp. 23, 35).

[Kom+09] Yuichi Komoriya, Ryoma Hashimoto, Keisuke Miyoshi, Yuya Ogawa, Daisuke Sakakibara, Jun Sawada, Kazuya Tahara, Hiroaki Fukushima, Kazuki Mitani, Ryo Yamakage, Yasuki Asano e Kosei Demura. *WinKIT 2009 Team Description Paper*. Inglês. Rel. téc. 7-1 Ohgigaoka; Nonoichi; Ishikawa 921-8501, Japan: Kanazawa Institute of Technology, 2009 (ver pp. 29, 30).

[Lau+09a] Nuno Lau, Luís Seabra Lopes, Gustavo Corrente e N. Filipe. ≪Multi-robot team coordination through roles, positioning and coordinated procedures≫. Inglês. Em: *Proc of the IEEE/RSJ international conference on intelligent robots ans systems*. St. Louis (USA), 2009, pp. 5841–48 (ver p. 46).

[Lau+09b] Nuno Lau, Luís Seabra Lopes, Gustavo Corrente e N. Filipe. ≪Roles, positionings and set plays to coordinate a MSL robot team≫. Inglês. Em: *Proc of the 4th international workshop on intelligent robotics, IROBOT'09*. Aveiro, Portugal, 2009, pp. 323–37 (ver p. 46).

[Li+13] X. Li, H. Lu, D. Xiong, H. Zhang e Z. Zheng. ≪A Survey on Visual Perception for RoboCup MSL Soccer Robots≫. Inglês. Em: *International Journal of Advanced Robotic Systems* 10 (2013), p. 110 (ver p. 21).

[Lim+01] P. Lima, A. Bonarini e C. Machado. *Omnidirectional catadioptric vision for soccer robots*. Inglês. Robotics and Autonomous Systems, 36 (1), pages 87-102. 2001 (ver p. 21).

[Lu+11] H. Lu, S. Yang, H. Zhang e Z. Zheng. ≪A Robust Omnidirectional Vision Sensor for Soccer Robots≫. Inglês. Em: *Mechatronics, 21(2)*. 2011, pp. 373–389 (ver pp. 21, 26).

[Lu+09] H. Lu, H. Zhang e J. Xiao. ≪Arbitrary Ball Recognition Based on Omnidirectional Vision for Soccer Robots≫. Inglês. Em: *RoboCup 2008: Robot Soccer World Cup XII, LNAI 5399*. 2009, pp. 133–144 (ver pp. 21, 22).

[McB+03] Stephanie McBader e Peter Lee. *An FPGA Implementation of a Flexible Parallel Image Processing Architecture Suitable for Embedded Vision Systems*. Inglês. NeuriCam S.p.A, Via S M Maddalena 12, 38100 Trento, Italy; University of Kent at Canterbury, Canterbury, Kent, CT2 7NT, UK. 2003 (ver p. 33).

[Nas+11] A. Nassiraei, S. Ishida, N. Shinpuku, M. Hayashi, N. Hirao, K. Fujimoto, K. Fukuda, K. Takanaka, I. Godler, K. Ishii e H. Miyamoto. *Hibikino-Musashi Team Description Paper*. Inglês. Rel. téc. Japan: Kyushu Institute of Technology, Japan. The University of Kitakyushu, Japan, 2011 (ver p. 28).

[Nev+08] António Neves, A. D. Martins e Armando J. Pinho. «A hybrid vision system for soccer robots using radial search lines». Inglês. Em: *Proc of the 8th conference on autonomous robot systems and competitions*. Portuguese robotics open - ROBOTICA'2008, Aveiro, Portugal, 2008, pp. 51–5 (ver p. 43).

[Nev+10a] António Neves, Armando J. Pinho, Daniel A. Martins e Bernardo Cunha. «An efficient omnidirectional vision system for soccer robots: From calibration to object detection». Inglês. Em: *Mechatronics* (2010). doi:10.1016/j.mechatronics.2010.05.006 (ver pp. 5, 41, 42, 44–46, 57).

[Nev+07] António Neves, Gustavo Corrente e Armando J. Pinho. «An omnidirectional vision system for soccer robots». Inglês. Em: *Proc of the 2nd international workshop on intelligent robotics IROBOT 2007*. 2007, pp. 499–507 (ver p. 44).

[Nev+09] António Neves, Bernardo Cunha e Ivo Pinheiro. «Autonomous configuration of parameters in robotic digital cameras». Inglês. Em: *Proc of the 4th Iberian conference on pattern recognition and image analysis, IbPRIA-2009*. Póvoa de Varzim, Portugal, 2009, pp. 80–7 (ver pp. 5, 43).

[Nev+10b] António Neves, José Luís Azevedo, Bernardo Cunha, João Silva, Frederico Santos e Gustavo Corrente. «CAMBADA soccer team: from robot architecture to multiagent coordination». Inglês. Em: Vienna, Austria: I-Tech Education e Publishing, jan. de 2010. Cap. 2 (ver p. 5).

[Ped10] Volnei A. Pedroni. *Circuit Design and Simulation with VHDL*. Inglês. 2ª ed. The MIT Press, 2010 (ver p. 75).

[Atlb] *Projeto Atlas Car, Universidade de Aveiro*. Inglês. URL: `http://atlas.web.ua.pt/atlascar.html` (acedido em 02/06/2013) (ver p. 12).

[Rib09] Bruno M. M. Ribeiro. «Detecção de objectos em robótica usando informação morfológica». Inglês. Tese de mestrado. Universidade de Aveiro, 2009 (ver p. 20).

[Sch+13] F. Schoenmakers, G. Koudijs, C. Martinez, M. Briegel, H. Wesel, J. Groenen, O. Hendriks, O. Klooster, R. Soetens e M. Molengraft. *Tech united eindhoven team description 2013*. Inglês. Rel. téc. Den Dolech 2, P.O. Box 513, 5600 MB Eindhoven - The Netherlands: Eindhoven University of Technology, 2013 (ver pp. 23, 36).

[Sil+09a] J. Silva, Nuno Lau, João Rodrigues, José Luís Azevedo e António Neves. «Sensor and information fusion applied to a robotic soccer team». Inglês. In: RoboCup 2009: robot soccer world cup XIII. 2009 (ver p. 46).

[Sil+09b] João Silva, Nuno Lau, António Neves, João Rodrigues e José Luís Azevedo. «Obstacle detection, identification and sharing on a robotic soccer team». Inglês. Em: *Proc of the 4th international workshop on intelligent robotics, IROBOT'09*. Aveiro, Portugal, 2009, pp. 350–60 (ver p. 46).

[Sud+11] K. C. Sudeep e Jharna Majumdar. «A Novel Architecture for Real Time Implementation of Edge Detectors on FPGA». Inglês. Em: *(IJCSI) Internation Journal of Computer Science Issues, Vol. 8, Issue 1* (2011) (ver p. 35).

[Tre+07] J. Trein, A. Schwarzbacher e B. Hoppe. *Development of a FPGA Based Real-Time Blob Analysis Circuit*. Inglês. ISSC 2007, Derry. 2007 (ver p. 33).

[Tre+08] J. Trein, A. Schwarzbacher e B. Hoppe. *FPGA Implementation of a Single Pass Real-Time Blob Analysis Using Run Length Encoding*. Inglês. Rel. téc. Ireland: Department of Electronic, Computer Science, Hochschule Darmstadt, Germany; School of Electronic e Communications Engineering, Dublin Institute of Technology, 2008 (ver p. 33).

[Ven+11] Muthukumar Venkatesan e Daggu V. Rao. *Hardware Acceleration of Edge Detection Algorithm on FPGAs*. Inglês. Rel. téc. Las Vegas, NV 89154: Department of Electrical e Computer Engineering, University of Nevada Las Vegas, 2011 (ver p. 35).

[Vmo] *VmodCAM Reference Manual*. Inglês. Digilent Inc. 1300 Henley Court, Pullman, WA 99163, jul. de 2011. URL: `http://www.digilentinc.com/` (acedido em 25/05/2013) (ver pp. 56, 101).

[Wika] *Wikipedia: Field-programmable gate array*. Inglês. URL: `http://en.wikipedia.org/wiki/Field-programmable_gate_array` (acedido em 24/05/2013) (ver p. 32).

[Wikb] *Wikipedia: HSV Color Space*. Inglês. URL: `http://en.wikipedia.org/wiki/HSL_and_HSV` (acedido em 25/05/2013) (ver p. 15).

[Wikc] *Wikipedia: Image Segmentation*. Inglês. URL: `http://en.wikipedia.org/wiki/Image_segmentation` (acedido em 25/05/2013) (ver pp. 15, 16).

[Pip] *Wikipedia: Pipeline (computing)*. Inglês. URL: `http://en.wikipedia.org/wiki/Pipeline_(computing)` (acedido em 23/05/2013).

[Wikd] *Wikipedia: RGB Color Space*. Inglês. URL: `http://en.wikipedia.org/wiki/RGB_color_space` (acedido em 25/05/2013) (ver p. 13).

[Wike] *Wikipedia: YUV Color Space*. Inglês. URL: `http://en.wikipedia.org/wiki/Yuv` (acedido em 25/05/2013) (ver p. 14).

[W+11] Marek Wójcikowski, Robert Zaglewski e Bogdan Pankiewicz. *FPGA-Based Real-Time Implementation of Detection Algorithm for Automatic Traffic Surveillance Sensor Network*. Inglês. Gdańsk University of Technology, Gdansk, Poland; Intel Shannon Ltd, Shanon, Ireland. 2011 (ver p. 35).

[Ori] *Xilinx: What is FPGA*. Inglês. URL: `http://www.origin.xilinx.com/fpga/` (acedido em 24/05/2013) (ver p. 31).

[Ytb] *YouTube: RoboCup 2012 MSL Group Stage 2 - Tech United vs CAMBADA*. Inglês. URL: `http://www.youtube.com/watch?v=oKJQkBzbxyw` (acedido em 12/06/2013) (ver p. 4).

[Zen+13] Z. Zeng, D. Xiong, Q. Yu, K. Huang, S. Cheng, H. Lu, X. Wang, H. Zhang, X. Li e Z. Zheng. *NuBot Team Description Paper 2013*. Inglês. Rel. téc. China, 410073: College of Mechatronics e Automation, National University of Defense Technology, 2013 (ver p. 26).

[Zin+07] T. T. Zin, H. Takahashi e H. Hama. «Robust person detection using far infrared camera for image fusion». Inglês. Em: *Second international conference on innovative computing, information and control, ICICIC 2007*. 2007, p. 310 (ver p. 46).

[Zou+06] J. Zou, H. Li, B. Liu e R. Zhang. ≪Color edge detection based on morphology≫. Inglês. Em: *First international conference on communications and electronics, ICCE 2006*. 2006, pp. 291–3 (ver p. 46).

[Zou+97] Y. Zou e W. Dunsmuir. ≪Edge detection using generalized root signals of 2-d median filtering≫. Inglês. Em: *Proc of the international conference on image processing, 1997*. Vol. 1. 1997, pp. 417–9 (ver p. 46).

[Zwe+09] O. Zweigle, U. Käppeler, H. Rajaie, K. Häussermann, A. Tamke, A. Koch, B. Eckstein, F. Aichele, G. Bhalsing e P. Levi. *1. RFC Stuttgart Team Description 2009*. Inglês. Rel. téc. Universitätsstraβe 38, 70569 Stuttgart, Germany: IPVS, University of Stuttgart, 2009 (ver p. 26).

Anexos

Anexo A

Plataforma de desenvolvimento

A plataforma de desenvolvimento adotada como suporte para este projeto, consiste numa FPGA do fabricante Digilent. De notar que embora a oferta existente no mercado à altura do início deste trabalho contemplasse outras possibilidades mais atrativas, quer do ponto de vista de recursos disponíveis, quer do ponto de vista de desempenho, esta solução apresentou o melhor compromisso. Como principal vantagem surge, inevitavelmente, o facto de possuir uma interface de alto débito (VHDCI) com suporte para diversos módulos. Entre outros, destaca-se o módulo *VmodCAM*, que inclui duas câmaras *Aptina*, dispondo de sensores CMOS a 2 megapíxeis e permitindo uma resolução de 1600x1200 a uma taxa de 15fps. Para além disso, a plataforma em si apresenta-se como um excelente compromisso entre a quantidade de recursos disponíveis e a performance permitida, face ao seu, ainda assim, baixo custo. Na verdade, considerando os aspectos enunciados e o facto de serem ainda disponibilizados pelo fabricante, pequenos projetos que visam demonstrar a funcionalidade da plataforma e do módulo *VmodCAM*, podemos mesmo admitir que se trata de uma solução pronta a usar fora da caixa (do inglês *ready to use out of the box*).

Dado que a arquitetura geral de uma FPGA, bem como algumas considerações sobre a implementação de projetos em ambiente de coprocessamento, foram já introduzidas nos subcapítulos 2.5.1 (pág. 30) e 2.5 (pág. 30), respetivamente, limitaremos as seguintes secções somente à apresentação sumária da placa e módulo *VmodCAM*.

A.0.1 Digilent Atlys

A plataforma de desenvolvimento Atlys (figura A.1) é uma solução pronta a utilizar no que diz respeito ao projeto, implementação e teste de soluções baseadas em sistemas digitais. Esta placa foi desenvolvida pelo fabricante Digilent[1], sendo baseada na FPGA Xilinx Spartan-6 LX45.

Apesar do seu reduzido tamanho, esta plataforma integra uma coleção invejável de tecnologia recente. Das suas principais características, importa destacar:

- FPGA Xilinx Spartan-6 LX45, 324-pin BGA
- 128 Mbytes de memória DDR2, 16 bits
- 10/100/1000 Ethernet PHY

[1] `http://www.digilentinc.com/`, acedido em 15/05/2013

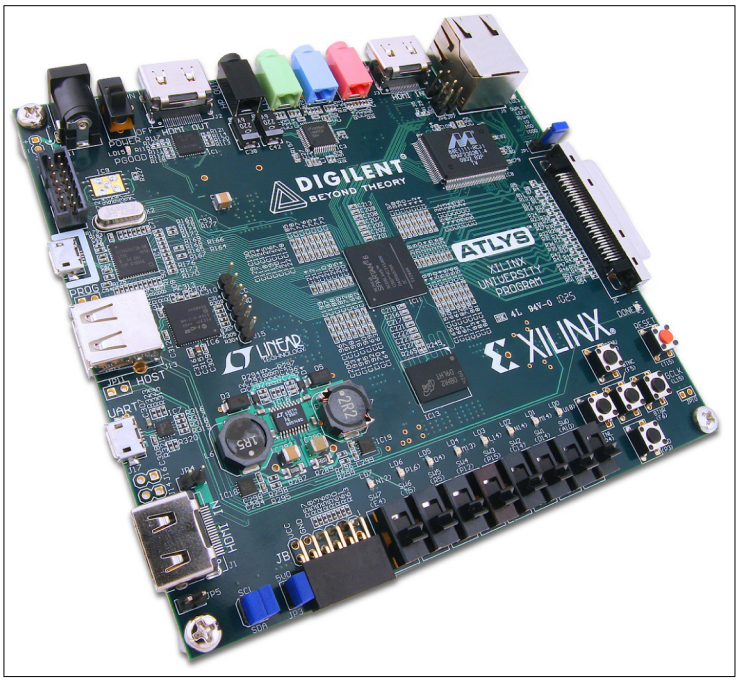


Figura A.1: Placa de desenvolvimento Atlys.

- portas USB2 integradas *onboard* (permitem programação e transferência de dados)
- duas portas HDMI de entrada e duas portas HDMI de saída
- portas audio AC-97 (linhas de entrada, saída, microfone e auscultadores)
- monitores de energia em tempo real em todas as linhas de energia
- 16 Mbytes x4 SPI Flash (para configuração e dados)
- oscilador 100 MHz CMOS
- 48 I/Os interligados com os conectores de expansão
- 8 Leds, 6 botões e 8 interruptores

Esta placa apresenta-se como uma escolha bastante interessante para o desenvolvimento de uma grande quantidade de sistemas digitais, suportando mesmo soluções de processadores integrados, baseados no *MicroBlaze* da *Xilinx*. Também a lista de periféricos disponíveis para as interfaces Pmod e VHDCI, bem como, exemplos de projetos disponíveis no site da *Digilent*, tornam esta solução bastante poderosa, expandindo sobretudo, o seu potencial e versatilidade. Outra vantagem reside no facto de ser compatível com todas as ferramentas CAD da *Xilinx*, incluindo algumas gratuitas, como o *ISE WebPack*.

Seguidamente, será apresentada uma descrição mais pormenorizada das características ou funcionalidades mais importantes e de uso neste projeto. Mais detalhes, poderão ser consultados em [Atla].

Configuração

Esta FPGA suporta três modos de configuração:

- porta JTAG ou porta mini-USB PROG (ferramenta *Adept*)

- SPI Flash ROM

- porta USB HID

A placa dispõe de um *jumper* (JP11) que permite selecionar entre os modos de programação JTAG/USB ou ROM. Se este *jumper* estiver aberto (ou não estiver presente), a FPGA irá autoconfigurar-se a partir da ROM. Se o *jumper* estiver fechado, a FPGA permanecerá em *standby* após ser ligada, a aguardar ser configurada a partir da porta série ou JTAG. A figura A.2 ilustra o pormenor do referido *jumper*, como estando aberto (não presente).

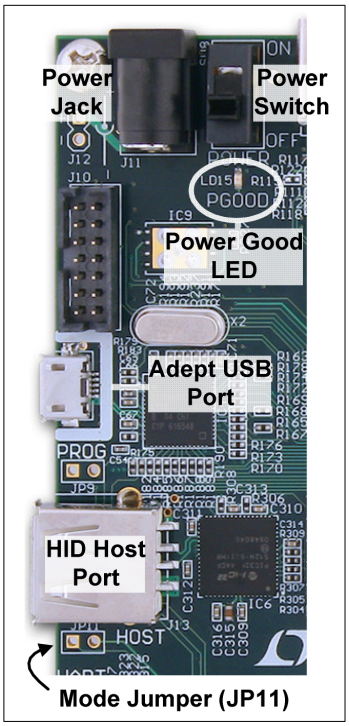


Figura A.2: Pormenor da placa de desenvolvimento Atlys, ilustrando o *jumper* J11.

A FPGA poderá ser configurada recorrendo às ferramentas *Adept* (fornecida pela *Digilent*) ou *iMPACT* (desenvolvida pela *Xilinx*), usando a porta mini-USB (PROG). Como referido anteriormente, é igualmente possível configurar a FPGA, transferindo o ficheiro .bit (única extensão aceite) para uma PEN USB. Neste caso, esse ficheiro deverá ser único e ficar na diretoria raíz.

Configuração via mini-USB (PROG), usando *Adept* ou *iMPACT*

A porta mini-USB (PROG) é compatível com a ferramente *iMPACT* da *Xilinx*. Contudo, para que a mesma possa reconhecer e programar corretamente a FPGA é necessário

instalar um *plugin* denominado *Digilent Plug-In for Xilinx Tools*. Este *plugin* encontra-se disponível no site da *Digilent*, secção de *software*, e permite converter automaticamente os comandos JTAG, gerados pela ferramenta *iMPACT*, num formato compatível com a porta USB, providenciando o mesmo conforto e grau de utilização, como se usasse a porta JTAG. Este funcionamento será transparente a todas as aplicações da *Xilinx*, sendo mesmo permitido o uso simultâneo da interface de programação via *plugin* e do monitor de energia presente na ferramenta *Adept*.

Memória DDR2

Esta FPGA inclui 128 MB de memória DDR2, com um *bus* de dados de 16 *bits*. Testes conduzidos, utilizando esta memória, demonstraram taxas de transferência de dados até 800 MHz.

Portas HDMI

A Atlys possui quatro portas HDMI, incluindo duas portas com *buffer* em modo entrada/saída, uma porta com *buffer* somente de saída e, finalmente, uma porta sem *buffer* em modo entrada/saída. As primeiras três portas que dispõem de *buffer*, são HDMI tipo A, enquanto a última é HDMI tipo D, encontrando-se oculta na parte inferior do conector Pmod. Os dados desta última, são partilhados com este conector, razão pela qual a largura de banda na transmissão de dados de vídeo pode ser estrangulada, não sendo possível enviar ou receber os sinais à máxima frequência (especialmente quando utilizados cabos HDMI longos).

Oscilador/Relógio

A Atlys inclui um oscilador de 100 MHz CMOS. O relógio de entrada pode conduzir unicamente um ou todos os blocos de gestão do sinal de relógio. Cada bloco destes inclui duas DCMs (*Digital Clock Manager*) e quatro PLLs (*Phase-Locked Loops*).

As DCMs podem gerar quatro fases na frequência de entrada (0°, 90°, 180° e 270°) e um sinal de relógio, que resulte de um divisor de sinal de relógio por um valor inteiro, compreendido entre 2 a 16 ou 1.5 a 7.5 (em incrementos de 1). São suscetíveis de gerar, ainda, dois sinais de relógio em fase inversa, que podem ser multiplicados por qualquer valor inteiro entre 2 a 32, e, simultaneamente, dividido por qualquer número inteiro entre 1 a 32.

As PLLs utilizam osciladores controladores de voltagem (do inglês, *Voltage-controlled oscillator*), que podem ser programados para gerar frequências, desde 400 MHz até 1080 MHz, definindo somente três conjuntos de divisores programáveis durante a configuração da FPGA. Os sinais de saída dos VCOs contêm oito valores igualmente espaçados (0°, 45°, 90°, 135°, 180°, 225°, 270° e 315°) que podem ser divididos por qualquer valor inteiro entre 1 e 128.

USB-UART (Porta de comunicação série)

A Atlys inclui uma *bridge* EXAR USB-UART, por forma a permitir a comunicação série com outras aplicações, usando uma porta COM. Encontram-se disponíveis controladores para os sistemas operativos Linux e Windows, que poderão ser descarregados gratuitamente.

USB HID Host

A FPGA Atlys inclui um microcontrolador interligado à porta USB HID HOST. Este microcontrolador permite que possa ser ligado um teclado ou um rato a esta porta (USB tipo A). Dado que a funcionalidade HUB não se encontra disponível, apenas é possível ligar um dispositivo.

Entradas e saídas genéricas

A placa Atlys inclui 6 botões de pressão, 8 interruptores e 8 *leds*, de forma a assegurar operações básicas de entrada e saída de sinais. Estas operações representam normalmente, formas de interação básica entre o sistema digital implementado e o utilizador, seja como entrada de dados e configuração de parâmetros ou como visualização de valores (saída de valores para os *leds*) ou modo *debug*. Um botão em especial, demarca-se pela sua cor vermelha e pela etiqueta *reset*. Este botão não difere dos restantes cinco, na medida em que é usado e acedido exatamente da mesma forma. No entanto, poderá ser usado como entrada de *reset* para sistemas que incluam um processador.

Portas de expansão

A placa Atlys dispõe de uma porta VHDCI de 68 *pins* que permite um alto débito na transmissão de dados. Esta porta suporta entrada e/ou saída de dados, sendo que estes são transferidos de forma paralela. Inclui 40 sinais de dados, 20 para massa (um por cada par de sinais de dados) e 8 sinais de energia. É normalmente utilizada por aplicações que usem SCSI-3, pois permite obter velocidades de transferência na ordem de vários *megahertz* em cada *pin*.

A placa possui ainda um conector de 8 *pins* denominado Pmod, adequado a componentes que necessitem de velocidades mais baixas ou com uma interface simplificada de apenas alguns *pins* (entrada e saída).

Encontram-se disponíveis para venda diversos componentes para ambas as portas, assim como os respetivos cabos de comunicação (disponíveis em vários tamanhos), sendo possível adquiri-los através da Digilent ou outros distribuidores.

A.0.2 Módulo *VmodCAM*

O módulo *VmodCAM*, ilustrado na figura A.3, providencia excelentes capacidades de aquição de imagem digital a qualquer FPGA Digilent que possua uma interface VHDCI [Vmo]. Este módulo possui duas câmaras *Aptina MT9D112* com um sensor digital CMOS de 2 megapíxeis. Estes sensores possibilitam taxas superiores a 15fps (o valor máximo alcançado depende da resolução adotada).

A sua implementação SoC integra um processador de fluxo de imagem, permitindo vários formatos de saída, ampliação e efeitos especiais. A PLL integrada e o seu microprocessador permitem uma interface de controlo flexível e simples.

As suas características principais incluem:

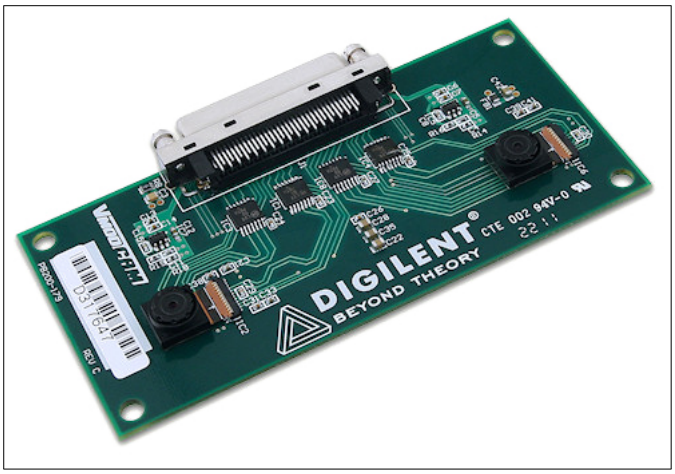


Figura A.3: Módulo *VmodCam*.

- duas câmaras independentes *Aptina MT9D112*, com sensor de 2 megapíxeis CMOS;
- resolução máxima de 1600x1200 píxeis, a uma taxa de 15fps;
- 63 mm de espaço entre câmaras;
- 10 *bit* de profundidade de cor em formato *Raw*;
- formatos de saída Bayer, RGB e YCrCb;
- exposição, ganho e balanço de brancos automáticos;
- algoritmos de correção de imagem;
- ampliação de imagem;
- FIFO de saída.

As câmaras podem ser controladas de forma independente, podendo adquirir duas imagens diferentes, simultaneamente. O seu sensor suporta saltar ou agrupar determinadas linhas/colunas. O microprocessador integrado possibilita a aplicação de algoritmos de correção à imagem, por forma a aumentar a sua qualidade. Este microprocessador consegue converter os dados *raw*, conforme adquiridos pela câmara, no formato RGB ou YCrCb, permitindo mesmo cortar ou ampliar a imagem. O FIFO assegura que os dados resultantes de alguns algoritmos mais complexos não são perdidos (tipicamente, estes algoritmos produzem resultados de forma não constante, sendo esta forma vulgarmente denominada "rajada de dados").

Anexo B

Resultado do módulo *ProcRunUnit*

Seguidamente será apresentada uma listagem que ilustra o resultado obtido após o processamento do módulo *ProcRunUnit*. A listagem consiste num conjunto de instruções inferidas após análise e deteção de sequências de uma mesma cor, bem como a sua eventual relação de vizinhança com outras sequências adjacentes e encontradas anteriormente. Estas instruções destinam-se a definir o tipo de operação sobre um ou mais objetos presentes em memória, definindo nomeadamente, o método de cálculo das suas características.

A lista de instruções gerada diz respeito à imagem de teste "B", cuja representação gráfica foi apresentada na página 19, figura 2.7b.

Considerando a arquitetura apresentada e admitindo que todo o processamento se encontra ativo, sabemos que os dados da imagem, são numa primeira fase, alvo da aplicação da máscara de imagem e de compressão utilizando o método RLE8. Neste exemplo, dadas as reduzidas dimensões da imagem e porque se trata somente de um teste de validação de processamento, não é aplicada a máscara de imagem. Deste modo, os dados de entrada ao módulo *ProcRunUnit* (resultantes do RLE8) são idênticos aos representados a azul na figura 2.7b.

Considerando a referida figura como apoio à explicação e detalhe da lista gerada, verificamos que cada instrução deriva diretamente do processamento de cada uma das diferentes sequências apresentadas (as quais denominamos *Runs*). Deste modo, para o primeiro conjunto de dados (`0F FF`) e dado que não existem ainda dados anteriores que possam ser correlacionados, é induzida a criação de um novo objeto. O número de objeto é atribuído com base na lista de objetos livres, a qual, nesta situação, inclui todos os números, pelo que é obtido o número 0 (zero). A cor, inicio e fim de posição são automaticamente inferidos com base nos dados processados pelo RLE8 (`0F`). A instrução final obtida após o processamento deste primeiro conjunto de dados é a indicada pela linha 1.

Os três próximos conjuntos de dados referem-se às sequências encontradas na segunda linha da imagem. No caso da primeira e terceira sequências, e dado que já é possível averiguar a existência de atuais vizinhanças, é detetada uma adjacência de igual cor na primeira linha, pelo que os objetos são os mesmos. Nesta situação, é induzida uma instrução `UPD` ao objeto já existente e cujas características deverão ser atualizadas com os dados da *Run* processada (instruções na linha 2 e 4 da listagem). O processamento da segunda sequência é idêntico, com a exceção de que não existe nenhuma vizinhança anterior, razão pela qual é induzida a criação de um novo objeto (linha 3 da listagem). O restante processamento deverá ser intuitivo ao leitor, a partir deste momento.

As instruções constantes nas linhas 62 e 65 referem-se aos casos em que uma sequência de uma determinada cor possui mais do que uma vizinhança detetada, situação que indica que encontramos uma junção de píxeis de uma mesma cor, a qual só pode dizer respeito ao mesmo objeto. No nosso caso, o objeto já era conhecido como sendo o mesmo, razão pela qual a instrução repete os números de objeto. Contudo, se a imagem original não possuísse as duas primeiras linhas, todo o processamento teria induzido a um conjunto de objetos diferente e, uma vez detetadas estas junções, os números de objetos seriam diferentes.

Por fim, observamos que quando um determinado objeto deixa de ser atualizado após processar duas ou mais linhas da imagem original, esse objeto é assumido como estando completo e pode imediatamente ser enviado, situação que ocorre com o objeto 2 na linha 66. Tendo processado uma frame completa, podemos igualmente assumir que todos os atuais objetos utilizados se encontram completos. Neste caso, todos os novos conjuntos de dados deverão induzir instruções que refiram novos objetos (situação ilustrada na linha 67), enquanto que todos os objetos tidos como completos, devem ser enviados à medida que for possível, sem prejudicar todo o restante processamento (situação ilustrada pelas linhas 68 e 69).

A figura B.1 ilustra a reconstrução da imagem em MATLAB, utilizando exclusivamente a lista de instruções gerada. Este tipo de reconstrução visa somente atestar a correta funcionalidade do módulo ou eventualmente para efeitos de *debug*.

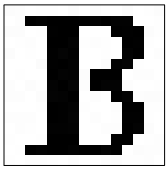


Figura B.1: Imagem reconstruída em MATLAB recorrendo à lista de instruções gerada pelo módulo *ProcRunUnit*.

```
 1:   NEW   objnumber(0)   color(255)   start(0)   end(14)   lineNumber(0)
 2:   UPD   objnumber(0)   color(255)   start(0)   end(1)    lineNumber(1)
 3:   NEW   objnumber(1)   color(0)     start(2)   end(11)   lineNumber(1)
 4:   UPD   objnumber(0)   color(255)   start(12)  end(14)   lineNumber(1)
 5:   UPD   objnumber(0)   color(255)   start(0)   end(3)    lineNumber(2)
 6:   UPD   objnumber(1)   color(0)     start(4)   end(6)    lineNumber(2)
 7:   NEW   objnumber(2)   color(255)   start(7)   end(9)    lineNumber(2)
 8:   UPD   objnumber(1)   color(0)     start(10)  end(12)   lineNumber(2)
 9:   UPD   objnumber(0)   color(255)   start(13)  end(14)   lineNumber(2)
10:   UPD   objnumber(0)   color(255)   start(0)   end(3)    lineNumber(3)
11:   UPD   objnumber(1)   color(0)     start(4)   end(6)    lineNumber(3)
12:   UPD   objnumber(2)   color(255)   start(7)   end(10)   lineNumber(3)
13:   UPD   objnumber(1)   color(0)     start(11)  end(12)   lineNumber(3)
14:   UPD   objnumber(0)   color(255)   start(13)  end(14)   lineNumber(3)
15:   UPD   objnumber(0)   color(255)   start(0)   end(3)    lineNumber(4)
16:   UPD   objnumber(1)   color(0)     start(4)   end(6)    lineNumber(4)
17:   UPD   objnumber(2)   color(255)   start(7)   end(10)   lineNumber(4)
18:   UPD   objnumber(1)   color(0)     start(11)  end(12)   lineNumber(4)
19:   UPD   objnumber(0)   color(255)   start(13)  end(14)   lineNumber(4)
20:   UPD   objnumber(0)   color(255)   start(0)   end(3)    lineNumber(5)
21:   UPD   objnumber(1)   color(0)     start(4)   end(6)    lineNumber(5)
22:   UPD   objnumber(2)   color(255)   start(7)   end(9)    lineNumber(5)
```

```
23:   UPD   objnumber(1)   color(0)     start(10)  end(11)   lineNumber(5)
24:   UPD   objnumber(0)   color(255)   start(12)  end(14)   lineNumber(5)
25:   UPD   objnumber(0)   color(255)   start(0)   end(3)    lineNumber(6)
26:   UPD   objnumber(1)   color(0)     start(4)   end(6)    lineNumber(6)
27:   UPD   objnumber(2)   color(255)   start(7)   end(7)    lineNumber(6)
28:   UPD   objnumber(1)   color(0)     start(8)   end(10)   lineNumber(6)
29:   UPD   objnumber(0)   color(255)   start(11)  end(14)   lineNumber(6)
30:   UPD   objnumber(0)   color(255)   start(0)   end(3)    lineNumber(7)
31:   UPD   objnumber(1)   color(0)     start(4)   end(6)    lineNumber(7)
32:   UPD   objnumber(2)   color(255)   start(7)   end(7)    lineNumber(7)
33:   UPD   objnumber(1)   color(0)     start(8)   end(10)   lineNumber(7)
34:   UPD   objnumber(0)   color(255)   start(11)  end(14)   lineNumber(7)
35:   UPD   objnumber(0)   color(255)   start(0)   end(3)    lineNumber(8)
36:   UPD   objnumber(1)   color(0)     start(4)   end(6)    lineNumber(8)
37:   UPD   objnumber(2)   color(255)   start(7)   end(9)    lineNumber(8)
38:   UPD   objnumber(1)   color(0)     start(10)  end(11)   lineNumber(8)
39:   UPD   objnumber(0)   color(255)   start(12)  end(14)   lineNumber(8)
40:   UPD   objnumber(0)   color(255)   start(0)   end(3)    lineNumber(9)
41:   UPD   objnumber(1)   color(0)     start(4)   end(6)    lineNumber(9)
42:   UPD   objnumber(2)   color(255)   start(7)   end(10)   lineNumber(9)
43:   UPD   objnumber(1)   color(0)     start(11)  end(12)   lineNumber(9)
44:   UPD   objnumber(0)   color(255)   start(13)  end(14)   lineNumber(9)
45:   UPD   objnumber(0)   color(255)   start(0)   end(3)    lineNumber(10)
46:   UPD   objnumber(1)   color(0)     start(4)   end(6)    lineNumber(10)
47:   UPD   objnumber(2)   color(255)   start(7)   end(10)   lineNumber(10)
48:   UPD   objnumber(1)   color(0)     start(11)  end(12)   lineNumber(10)
49:   UPD   objnumber(0)   color(255)   start(13)  end(14)   lineNumber(10)
50:   UPD   objnumber(0)   color(255)   start(0)   end(3)    lineNumber(11)
51:   UPD   objnumber(1)   color(0)     start(4)   end(6)    lineNumber(11)
52:   UPD   objnumber(2)   color(255)   start(7)   end(10)   lineNumber(11)
53:   UPD   objnumber(1)   color(0)     start(11)  end(12)   lineNumber(11)
54:   UPD   objnumber(0)   color(255)   start(13)  end(14)   lineNumber(11)
55:   UPD   objnumber(0)   color(255)   start(0)   end(3)    lineNumber(12)
56:   UPD   objnumber(1)   color(0)     start(4)   end(6)    lineNumber(12)
57:   UPD   objnumber(2)   color(255)   start(7)   end(9)    lineNumber(12)
58:   UPD   objnumber(1)   color(0)     start(10)  end(11)   lineNumber(12)
59:   UPD   objnumber(0)   color(255)   start(12)  end(14)   lineNumber(12)
60:   UPD   objnumber(0)   color(255)   start(0)   end(1)    lineNumber(13)
61:   UPD   objnumber(1)   color(0)     start(2)   end(10)   lineNumber(13)
62:   MRG   objnumber(1)   objNumber(1)
63:   UPD   objnumber(0)   color(255)   start(11)  end(14)   lineNumber(13)
64:   UPD   objnumber(0)   color(255)   start(0)   end(14)   lineNumber(14)
65:   MRG   objnumber(0)   objNumber(0)
66:   DPT   objnumber(2)
67:   NEW   objnumber(3)   color(255)   start(0)   end(14)   lineNumber(0)
68:   DPT   objnumber(0)
69:   DPT   objnumber(1)
```

Anexo C

Resultado do módulo *PipelineUnit*

Seguidamente será apresentada uma listagem que ilustra o resultado obtido após o processamento do módulo *PipelineUnit*. A listagem apresenta uma lista de objetos, bem como a atualização das suas características após a execução de uma determinada instrução.

Tal como já referido, o resultado obtido não é correto do ponto de vista de processamento e cálculo das características. Embora o mesmo algoritmo tenha sido corretamente validado em simulação comportamental, a sintetização e teste da solução sobre *hardware* reconfigurável (no nosso caso, a plataforma *Atlys*), revelou sempre um problema no cálculo das características, o qual não foi possível resolver até ao momento.

O objeto processado na linha 1, bem como as características associadas, encontram-se totalmente corretos. Na verdade, sempre que o cálculo das características de um objeto depende exclusivamente dos dados contidos na própria instrução, o processamento é correto (situação demonstrada igualmente na linha 7). Contudo, quando a operação necessita de características que tenham sido calculadas/atualizadas no ciclo de relógio anterior, existe um problema no acesso aos dados que impõe que o cálculo seja errado.

```
1: color(255) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(0) MCenterX(105) MCenterY(0) NumPixels(15)
2: color(0) BoxX1(0) BoxX2(1) BoxY1(0) BoxY2(1) MCenterX(1) MCenterY(2) NumPixels(2)
3: color(0) BoxX1(2) BoxX2(11) BoxY1(1) BoxY2(1) MCenterX(65) MCenterY(10) NumPixels(10)
4: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(1) MCenterX(39) MCenterY(3) NumPixels(3)
5: color(0) BoxX1(0) BoxX2(3) BoxY1(0) BoxY2(2) MCenterX(6) MCenterY(8) NumPixels(4)
6: color(0) BoxX1(2) BoxX2(11) BoxY1(1) BoxY2(2) MCenterX(80) MCenterY(16) NumPixels(13)
7: color(255) BoxX1(7) BoxX2(9) BoxY1(2) BoxY2(2) MCenterX(24) MCenterY(6) NumPixels(3)
8: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(2) MCenterX(113) MCenterY(22) NumPixels(16)
9: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(2) MCenterX(27) MCenterY(4) NumPixels(2)
10: color(0) BoxX1(0) BoxX2(3) BoxY1(0) BoxY2(3) MCenterX(6) MCenterY(12) NumPixels(4)
11: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(3) MCenterX(128) MCenterY(31) NumPixels(19)
12: color(255) BoxX1(7) BoxX2(10) BoxY1(2) BoxY2(3) MCenterX(58) MCenterY(18) NumPixels(7)
13: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(3) MCenterX(151) MCenterY(37) NumPixels(21)
14: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(3) MCenterX(27) MCenterY(6) NumPixels(2)
15: color(0) BoxX1(0) BoxX2(3) BoxY1(0) BoxY2(4) MCenterX(6) MCenterY(16) NumPixels(4)
16: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(4) MCenterX(166) MCenterY(49) NumPixels(24)
17: color(255) BoxX1(7) BoxX2(10) BoxY1(2) BoxY2(4) MCenterX(92) MCenterY(34) NumPixels(11)
18: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(4) MCenterX(189) MCenterY(57) NumPixels(26)
19: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(4) MCenterX(27) MCenterY(8) NumPixels(2)
20: color(0) BoxX1(0) BoxX2(3) BoxY1(0) BoxY2(5) MCenterX(6) MCenterY(20) NumPixels(4)
21: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(5) MCenterX(204) MCenterY(72) NumPixels(29)
22: color(255) BoxX1(7) BoxX2(10) BoxY1(2) BoxY2(5) MCenterX(116) MCenterY(49) NumPixels(14)
23: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(5) MCenterX(225) MCenterY(82) NumPixels(31)
```

24: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(5) MCenterX(39) MCenterY(15) NumPixels(3)
25: color(0) BoxX1(0) BoxX2(3) BoxY1(0) BoxY2(6) MCenterX(6) MCenterY(24) NumPixels(4)
26: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(6) MCenterX(240) MCenterY(100) NumPixels(34)
27: color(255) BoxX1(7) BoxX2(10) BoxY1(2) BoxY2(6) MCenterX(123) MCenterY(55) NumPixels(15)
28: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(6) MCenterX(267) MCenterY(118) NumPixels(37)
29: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(6) MCenterX(50) MCenterY(24) NumPixels(4)
30: color(0) BoxX1(0) BoxX2(3) BoxY1(0) BoxY2(7) MCenterX(6) MCenterY(28) NumPixels(4)
31: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(7) MCenterX(282) MCenterY(139) NumPixels(40)
32: color(255) BoxX1(7) BoxX2(10) BoxY1(2) BoxY2(7) MCenterX(130) MCenterY(62) NumPixels(16)
33: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(7) MCenterX(309) MCenterY(160) NumPixels(43)
34: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(7) MCenterX(50) MCenterY(28) NumPixels(4)
35: color(0) BoxX1(0) BoxX2(3) BoxY1(0) BoxY2(8) MCenterX(6) MCenterY(32) NumPixels(4)
36: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(8) MCenterX(324) MCenterY(184) NumPixels(46)
37: color(255) BoxX1(7) BoxX2(10) BoxY1(2) BoxY2(8) MCenterX(154) MCenterY(86) NumPixels(19)
38: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(8) MCenterX(345) MCenterY(200) NumPixels(48)
39: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(8) MCenterX(39) MCenterY(24) NumPixels(3)
40: color(0) BoxX1(0) BoxX2(3) BoxY1(0) BoxY2(9) MCenterX(6) MCenterY(36) NumPixels(4)
41: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(9) MCenterX(360) MCenterY(227) NumPixels(51)
42: color(255) BoxX1(7) BoxX2(10) BoxY1(2) BoxY2(9) MCenterX(188) MCenterY(122) NumPixels(23)
43: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(9) MCenterX(383) MCenterY(245) NumPixels(53)
44: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(9) MCenterX(27) MCenterY(18) NumPixels(2)
45: color(0) BoxX1(0) BoxX2(3) BoxY1(0) BoxY2(10) MCenterX(6) MCenterY(40) NumPixels(4)
46: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(10) MCenterX(398) MCenterY(275) NumPixels(56)
47: color(255) BoxX1(7) BoxX2(10) BoxY1(2) BoxY2(10) MCenterX(222) MCenterY(162) NumPixels(27)
48: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(10) MCenterX(421) MCenterY(295) NumPixels(58)
49: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(10) MCenterX(27) MCenterY(20) NumPixels(2)
50: color(0) BoxX1(0) BoxX2(3) BoxY1(0) BoxY2(11) MCenterX(6) MCenterY(44) NumPixels(4)
51: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(11) MCenterX(436) MCenterY(328) NumPixels(61)
52: color(255) BoxX1(7) BoxX2(10) BoxY1(2) BoxY2(11) MCenterX(256) MCenterY(206) NumPixels(31)
53: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(11) MCenterX(459) MCenterY(350) NumPixels(63)
54: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(11) MCenterX(27) MCenterY(22) NumPixels(2)
55: color(0) BoxX1(0) BoxX2(3) BoxY1(0) BoxY2(12) MCenterX(6) MCenterY(48) NumPixels(4)
56: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(12) MCenterX(474) MCenterY(386) NumPixels(66)
57: color(255) BoxX1(7) BoxX2(10) BoxY1(2) BoxY2(12) MCenterX(280) MCenterY(242) NumPixels(34)
58: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(12) MCenterX(495) MCenterY(410) NumPixels(68)
59: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(12) MCenterX(39) MCenterY(36) NumPixels(3)
60: color(0) BoxX1(0) BoxX2(1) BoxY1(0) BoxY2(13) MCenterX(1) MCenterY(26) NumPixels(2)
61: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(13) MCenterX(549) MCenterY(527) NumPixels(77)
62: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(13) MCenterX(1098) MCenterY(1054) NumPixels(154)
63: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(13) MCenterX(1098) MCenterY(1054) NumPixels(154)
64: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(13) MCenterX(50) MCenterY(52) NumPixels(4)
65: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(14) MCenterX(105) MCenterY(210) NumPixels(15)
66: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(14) MCenterX(210) MCenterY(420) NumPixels(30)
67: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(14) MCenterX(210) MCenterY(420) NumPixels(30)
68: color(255) BoxX1(7) BoxX2(10) BoxY1(2) BoxY2(12) MCenterX(280) MCenterY(242) NumPixels(34)
69: color(255) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(0) MCenterX(105) MCenterY(0) NumPixels(15)
70: color(0) BoxX1(0) BoxX2(14) BoxY1(0) BoxY2(14) MCenterX(210) MCenterY(420) NumPixels(30)
71: color(0) BoxX1(2) BoxX2(12) BoxY1(1) BoxY2(13) MCenterX(1098) MCenterY(1054) NumPixels(154)

Anexo D

Diagrama de simulação do módulo *PipelineUnit*

Seguidamente será apresentado um *screenshot* do diagrama de simulação comportamental do módulo *PipelineUnit*, que pretende ilustrar um momento temporal no processamento dos dados, nomeadamente, o cálculo das características para o objeto 0 (zero).

A simulação em causa pretende ilustrar o momento pós-cálculo da segunda instrução gerada pelo módulo *ProcRunUnit*. A instrução em causa impõe uma atualização de dados ao objeto 0 (zero), utilizando os dados constantes na própria instrução e os dados calculados no ciclo de relógio anterior.

Relembrando as equações 5.3 apresentadas na página 69, verificamos que para a primeira instrução gerada pelo módulo *ProcRunUnit*, temos que

$$x^* = x + \frac{e^2 + e - s^2 + s}{2} = 0 + \frac{14^2 + 14 - 0^2 + 0}{2} = 105 \tag{D.1}$$

$$y^* = y + (e - s + 1) * l = 0 + (14 - 0 + 1) * 0 = 0 \tag{D.2}$$

Considerando os dados referentes à segunda instrução e os dados calculados anteriormente para o objeto 0 (zero), temos que

$$x^* = 105 + \frac{1^2 + 1 - 0^2 + 0}{2} = 106 \tag{D.3}$$

$$y^* = 0 + (1 - 0 + 1) * 1 = 2 \tag{D.4}$$

Para um correta análise da figura importa ainda relembrar qual a estrutura de dados utilizada por cada objeto de modo a guardar as diversas características calculadas, a qual foi apresentada na página 69, assim como a arquitetura de implementação para a *pipeline* apresentada na página 70, figura 5.7.

Tendo ainda em conta que todos os valores se encontram em binário, atentemos nos sinais `mux1out` e `mux2out`, os quais contém os dados referentes, respetivamente, à segunda instrução

e os dados referentes ao objeto 0 (zero) atualizados no ciclo anterior. Estes sinais são no fundo, os sinais de entrada ao módulo *calcFeaturesUnit*, o qual terá a responsabilidade de efetuar os cálculos supra citados. O resultado do cálculo é representado pelo sinal `calcFeaturesOut` e, tal como se pode observar, os dados encontram-se de acordo com o esperado.

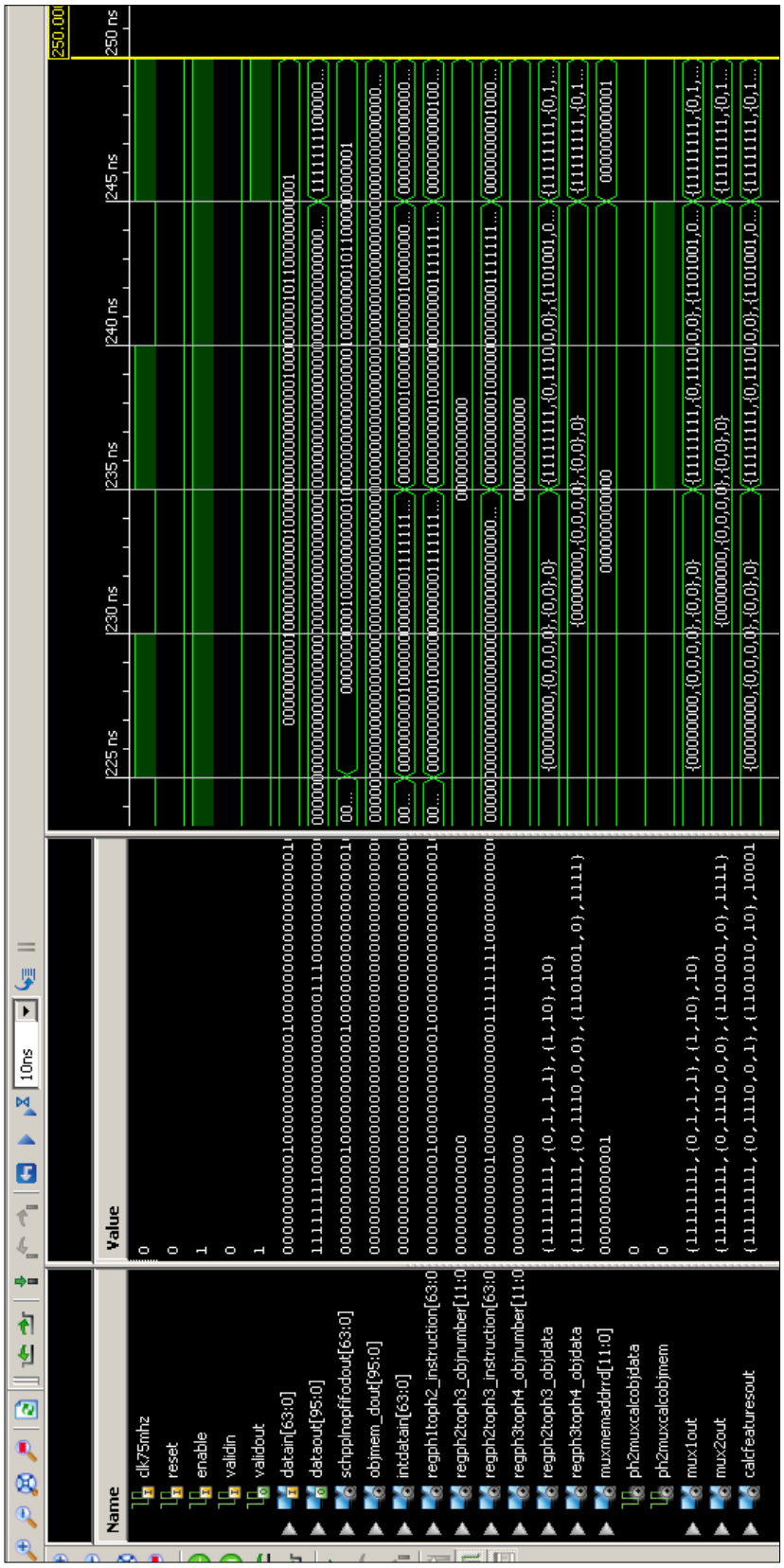


Figura D.1: Simulação comportamental do módulo *PipelineUnit*.