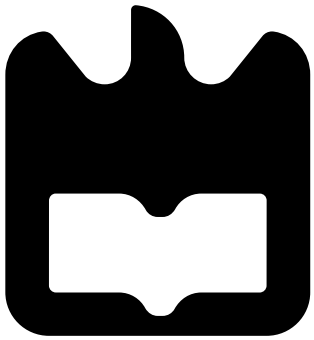
Universidade de Aveiro
Departamento de
Electrónica, Telecomunicações e Informática
2013

Tiago Emanuel Urze Afonso

FPGA e sistemas embutidos multi-core para processamento de vídeo

FPGA and multi-core embedded systems for video processing



Tiago Emanuel Urze Afonso

FPGA e sistemas embutidos multi-core para processamento de vídeo

FPGA and multi-core embedded systems for video processing

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor Pedro Nicolau Faria Fonseca (orientador), professor auxiliar do departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Manuel Bernardo Salvador Cunha (coorientador), professor assistente do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Prof. Doutor Arnaldo Silva Rodrigues de Oliveira
Professor Auxiliar da Universidade de Aveiro (por delegação da Reitora da Universidade de Aveiro)

vogais / examiners committee

Prof. Doutor Hélio Mendes de Sousa Mendonça
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto (Arguente Principal)

Prof. Doutor Manuel Bernardo Salvador Cunha
Professora Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro (Co-orientador)

agradecimentos / acknowledgements

Gostava de agradecer em primeiro lugar aos meus pais e irmão, que me apoiaram incondicionalmente ao longo de todo este processo de formação no ensino superior e que constituiram um pilar muito importante na minha educação, acreditando sempre nas minhas capacidades. À Natasa, por toda a paciência que teve nas horas sem fio que me ouviu a falar do tema, e por todo o apoio, incentivo e confiança que sempre depositou em mim. Aos meus colegas e amigos em geral que me acompanharam ao longo deste percurso académico e comigo partilharam horas a fio de trabalho. Ao meu orientador, co-orientador e colaborador pela ajuda na escrita deste documento, nomeadamente pela sua contribuição científica, demonstrando sempre disponibilidade para ajudar. Por fim, a todos os outros a quem a memória me esteja a falhar.

palavras-Chave

FPGA, visão artificial, deteção de contornos, Canny, análise morfológica de imagem, VHDL, CAMBADA, dispositivos lógicos programáveis, lógica reprogramável, vídeo, processamento em paralelo, ipCores

resumo

O presente trabalho apresenta técnicas de processamento digital de sinal, nomeadamente em processamento de vídeo, recorrendo à tecnologia FPGA. Consiste numa introdução teórica sobre tópicos tais como o papel da visão artifical nos dias de hoje, reconhecimento de imagem, e técnicas matemáticas de processamento e análise morfológica de imagem. Aborda o tema do papel das FPGAs na tecnologia actual, e as suas vantagens quando utilizadas no processamento digital de sinal. Finalmente é demonstrado e explicado o algoritmo implementado na FPGA para deteção de contornos no processamento de vídeo, concluindo com uma análise a nível da sua eficiência, e discussão de melhorias a fazer num possível trabalho futuro em termos de otimização de recursos utilizados e velocidade de processamento.

Keywords FPGA, artificial vision, Canny, morphological image analysis, VHDL, CAMBADA, programmable logic devices, reprogrammable logic, video, parallel processing, ipCores

Abstract The present work presents techniques of digital signal processing, namely in video processing, using FPGA technology. It consists of a theoretical introduction about topics such as the role of artificial vision nowadays, image recognition and mathematical techniques of image processing and morphological analysis. It discusses the role of an FPGA in today's technology and its advantages when used in digital signal processing. Finally, it is demonstrated and explained the algorithm that was implemented in the FPGA for edge detection in video processing, concluding with an analysis in terms of efficiency, and discussion of improvements to do in a possible future work regarding the optimization of used resources and also of its processing speed.

Contents

List of Figures

Acronyms

ALU *Arithmetic Logic Unit*

ASIC *Application-specific Integrated Circuit*

BRAM *Block Random Access Memory*

CAD *Computer-Aided Design*

CAGR *Compound Annual Growth Rate*

CAMBADA *Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture*

CLB *Configurable Logic Block*

CPLD *Complex Programmable Logic Device*

CPU *Central Processing Unit*

DSP *Digital Signal Processor*

FF *Flip-Flop*

FPGA *Field-Programmable Gate Array*

HDL *Hardware Description Language*

IC *Integrated Circuit*

LTE *Long-Term Evolution*

LUT *Look-Up Table*

NRE *Non-recurring engineering*

PLD *Programmable Logic Device*

RAM *Random Access Memory*

SMD *Surface Mounted Device*

SRAM *Static Random Access Memory*

UART *Universal Asynchronous Receiver/Transmitter*

USB *Universal Synchronous Bus*

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

Chapter 1

Introduction

1.1 Computer Vision: A modern reality

As artificial intelligence technology advances, computer vision is becoming more and more important and present in our everyday life. Nowadays we have plenty of applications on this area embracing various markets, namely health, personal security and robotics. Besides those areas, computer vision also plays an important role in all the industrial sectors which aim the optimization of quality standards. This optimization is provided through the inspection at nano-level in various sectors like mechanical, textile, electronics, pharmaceutical, plastic packaging and more, detecting problems which the human eye is oblivious to. Examples of real case situations are the following: monitoring of urban areas, *Surface Mounted Device* (SMD) pins inspection of its positioning, fast and precise evaluation of food color/shape/condition, inspection of leaks in liquid barrels, among other applications.

When we talk about this technology, we have to talk about a common computer system which allows to implement it, performing dedicated functions: the embedded systems. The heart of this computer systems are the processing cores, which in the case of video processing are named *Digital Signal Processor*s (DSPs) and enable the processing of individual or a sequence of frames on-the-fly.

CAMBADA is a middle-size robotic football team project developed in the University of Aveiro, which was created aiming the participation in the RoboCup middle-size league - an international initiative with the goal of promoting the development of artificial intelligence in the robotics area and related fields. CAMBADA project, among other technologies, has adopted an embedded computer vision system to provide their robots with the ability to see, being a great example of successful use of this technology.

In this competition, during a game in the playing field, besides the obstacles and the points-of-reference, there is also a very fast-changing scenery with the teammates, opponents and the ball itself moving quickly and unpredictably, requiring a fast processing system to operate heavy quantities of information on-the-fly, through an omnidirectional camera.

1.2 Goals

This is a very good situation to study the advantage of relegating the processing of heavy information to the power of reconfigurable hardware, through its parallel processing capacity, integrated in an embedded system. In this case we will be using an FPGA to process on-the-fly data. But real-time processing is not the only challenge we have to face.

Year after year, RoboCup competition has been changing in some aspects, namely on its environment. Initially, this environment was robot friendly, where conditions such as controlled lighting or easy to recognize color coded objects were taken for granted. Today, with the advance of the technology, this environment has become increasingly more difficult with all these conditions being relaxed or completely suppressed. This implies an important step on the evolution of the robots vision system which has to adapt to the strong lightning changes and also to the change of ball-type colour from game to game. One solution to this problem is to make a vision system not color based but, instead, able to recognize boundaries and consequently geometric forms. This systems, based on edge detection, becomes independent from the light and colour conditions, detecting object by its form, regardless of its colour.

Edge detection is in fact, one of the most common operations in the world of image analysis, and as a consequence with various applications and big relevance in the computer vision field of study. According to CAMBADA experiments, among several edge detection algorithms, Canny Edge algorithm was the one chosen for the role of edge detection because, despite of being the most demanding regarding processing time, it was fast enough for real-time processing, also providing the most effective contours [12].

In this project, we will test and evaluate the power and possible advantages of the use of reconfigurable hardware parallel processing, in the step of "Edge Detection" using the Canny algorithm, within the vision system of CAMBADA. This will involve the following points:

- The study of reconfigurable hardware technology, namely FPGA technology and its different phases of the project;
- The study of a computer vision algorithm within the scope of CAMBADA vision system,

for the purpose of boundary detection and its inherent steps;

- The study of the versatility of hardware description languages, VHDL, namely for arithmetic operations purposes;
- Test, optimization of size and performance, and validation of a DSP embedded in an FPGA;

Chapter 2

Reconfigurable Computing

2.1 Introduction

The concept of reconfigurable computing has existed since the 1960s [37], when the concept of a computer constituted of a standard processor and an array of "reconfigurable" hardware was proposed. The idea was to have the processor controlling the reconfigurable hardware, which would be tailored to do a specific task - such as image processing - as quickly as a dedicated piece of hardware. Once the task was done, the hardware could be reconfigured to do some other task, resulting in a hybrid computer structure which combined the flexibility of software with the speed of hardware. However, in that time, this idea was far ahead of its needed state-of-art electronic technology.

Today's computationally intensive applications, such as video streaming, image recognition and processing, and highly interactive services, are placing new demands on the computation units that implement these applications, requiring a higher processing power than ever before. Facing these performance requirements, the power consumption targets, the acceptable packaging and manufacturing costs, and the time-to-market requirements of these computation units are all decreasing rapidly, especially in the embedded devices market. In fact, in 2004, studies already reported that moving critical software loops to reconfigurable hardware were resulting in average energy savings of 35% to 70% with an average speed up of 3 to 7 times, depending on the particular device used [5].

Hence, and thanks to the advance of technology, reconfigurable computing finally came up and is rapidly establishing itself as a major discipline. Indeed, there is evidence that embedded systems developers show a growing interest in reconfigurable computing systems [6], especially with the introduction of soft cores which can contain one or more instruction

processors [7, 8, 9, 10, 11, 20].

2.2 Programmable Logic vs Fixed Logic

The vast world of digital electronics can be divided in three categories, memory devices, microprocessors and logic devices, where the latter can be classified in two different categories: fixed and programmable. It is very important to clarify the difference between them. As the name suggests, the circuits in a fixed logic device are permanent and they are customized to perform a set of functions, can't be changed once manufactured (for eg., *Application-specific Integrated Circuit* (ASIC)). The time required to go from design, to prototypes, to a final manufacturing run, can take from several months to more than a year, depending on the complexity of the device, and if the requirements change there is no flexibility at all so a new design must be developed [2].

On the other hand, programmable logic devices, as already mentioned, offer users a wide range of logic capacity, features and speed, and they can be changed and configured at any time. With this technology, designers appeal to inexpensive tools being able to quickly program it into a device, and immediately test it in a live circuit. Another key benefit of using programmable logic devices is that in some of them, circuit can be reconfigured as many times we want, whenever we want, until we are satisfied with the optimization, due to the fact they are based on re-writable memory technology.

So let us resume the main advantages of each technology. Fixed logic devices is better shapped for big volume applications since their mass productions is cheaper, and when high performance is required. However, programmable logic devices provide us with:

- More flexibility during the design cycle – we can change the programming file and the results of the design changes can be seen immediately in working parts;
- Faster prototype building;
- The chance to add new features from time to time, upgrading it with the upload of a programming file, directly via the internet or from a hard-drive, creating new hardware logic in the system;
- Customers don't have to pay for large *Non-recurring engineering* (NRE) costs since these costs will be amortized over the multi-year lifespan of a given line of the *Programmable Logic Device* (PLD).

2.3 FPGA vs CPLDs

The two major types of programmable logic devices are FPGAs and *Complex Programmable Logic Device*s (CPLDs), each having different characteristics. CPLDs, whilst offering much smaller amounts of logic than FPGAs, provide predictable timing characteristics which make it ideal for critical control applications, and also require extremely low amounts of power becoming an optimal choice for cost-sensitive, battery-operated, portable applications like digital phones. On the other hand, FPGAs offer the highest amount of logic density, highest performance and several features, providing substantial amounts of memory, built-in hard-wired processors, and other means. These devices are used mostly for data processing and storage, telecommunications, digital signal processing, among other applications. But how did they emerge? How do they function? These are questions we'll be discussing in the next topic.

2.4 FPGA state-of-art

FPGAs were introduced in the market in 1984 by Xilinx with a central idea: the capacity to produce a personalized integrated circuit, without the economic risk usually associated to other technologies. Today FPGAs are present in fields as diverse as automation, consumer electronics, biomedicine, night vision, telecommunications, security, special investigation, among others. This technology is basically a device with an application in all the industries which require high-speed and/or heavy processing computation. Since the emergence of integrated circuits, there are two alternatives to carry out a digital hardware: algorithm codification in a microprocessor or direct mapping of the algorithm in the hardware. Microprocessor together with microcontrollers and DSPs allow to successfully solve most of the electronic challenges.

But what is an FPGA and why is it such a great device? As mentioned earlier in this chapter, an FPGA is a structure of logic cells and connections, which is under the control of the user/costumer, being able to design and program it, and make changes to the circuit whenever necessary. Basically, they are prefabricated silicon chips that can be programmed electrically to implement any digital design.

The elementary architecture of FPGA consists of three major components: programmable logic blocks (which implement the logic functions), programmable routing (interconnects) to implement the functions and I/O blocks to make off-chip connections, while they may also

integrate extra blocks like dedicated memories (BlockRAM). An illustration of a typical FPGA architecture is shown in figure 2.1.

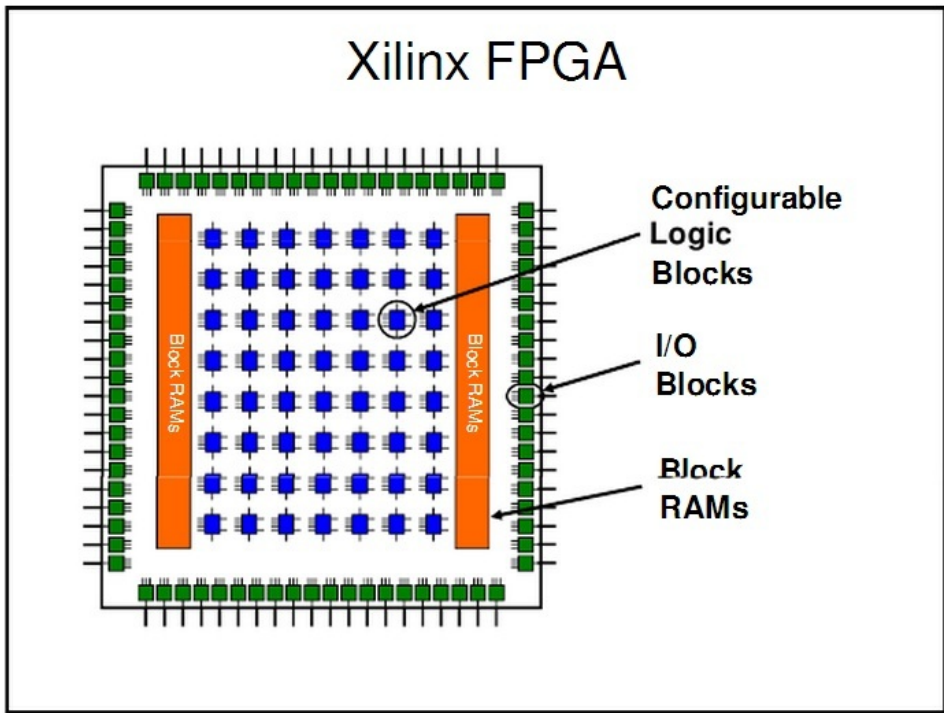


Figure 2.1: Built-in overview of an FPGA

2.4.1 Programmable Logic blocks

Logic blocks can be named the main elements of an FPGA because its purpose is to provide the basic computation and storage elements used in digital systems. Each basic logic element is composed by programmable combinational logic, *Flip-Flop*s (FFs) or latches and fast carry logic to reduce area and delay cost (detailed information will be provided in other topics regarding the FPGA model in use). Furthermore, FPGA has as well other *Configurable Logic Block*s (CLBs) with specific functions like dedicated memory blocks or multipliers.

2.4.2 Programmable Interconnect

For communication to be feasible within the FPGA, through programmable routing connections are established among logic blocks and I/O blocks to construct the design defined by the user. This is provided by several multiplexers, pass transistors and tri-state buffers, which together form the desired connection. Logic elements are connected by the combination of

pass transistors and multiplexers whilst these latter two with the tri-state buffers form global routing structures.

2.4.3 Programmable I/O

FPGA dispose of I/O pads which provide the interface between its logic blocks and routing architectures and the several external components. These I/O pads along with surrounding supporting logic circuitry form I/O cells which are important components consuming approximately 40% of FPGAs area [3].

2.4.4 Block RAM vs Distributed RAM

Regarding the storing capability of the FPGA, usually, depending on the amount of RAM required for a specific function, we're working with one of two types of this memory. *Block Random Access Memory*s (BRAMs) are located in specific areas of the FPGA and they are dedicated, hence they cannot be used for anything but RAM functions. For small amounts of memory, FPGA logic-cells are used as tiny RAMs providing a very flexible RAM distribution (although inefficient in terms of area), which is called Distributed RAM.

RAM operation is affected by many parameters, where the main one is the number of agents that can access it simultaneously. While in a Single-Port RAM only one agent can read/write the RAM, in a "dual-port" RAM two agents can read/write. In fig.2.2 is illustrated a simplified scheme of a dual-port RAM.

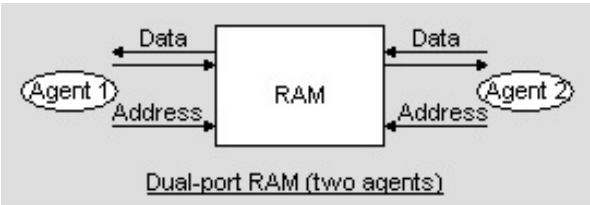


Figure 2.2: Dual-port RAM scheme

Eventually, technology improves and FPGA architecture has not been an exception to that. Nowadays, more specialized programmable functional blocks were added, namely embedded memory (BlockRAMs), *Arithmetic Logic Unit*s (ALUs), multiplexers and even microprocessors, due to a frequent need of such resources for application in a wide variety of areas.

A good example of this fact is the FPGA used in this project, Virtex 5 by Xilinx, which provides various resources, as we can see below.

FPGA Virtex-5 XC5VLX50T typical blocks and its corresponding logic resources: [13]

- 120 x 30 CLB, each with two slices (figure 2.3) with 4 *Look-Up Table*s (LUTs), 4 storage elements (FFs), wide-function multiplexers and carry-logic;

- Some slices support two additional functions: storing data using distributed RAM and shifting data with 32-bit registers. These slices are called SLICEM (fig. 2.4). Others are named SLICEL (fig. 2.5);

- CLBs which have SCLICEMs contain 256 bits of distributed RAMs and 128 bits of Shift Registers;

- Total of 2160 Kb True dual-port RAM blocks [14];

- Clock control units;

- 550 MHz Clock technology;

- 48 x 550 MHz DSP48 Slices, each containing 25x18 multiplier, an adder and an accumulator;

- Ethernet MACs;

- Flexible and high performance I/O interface standards (up to Gbps);

- Hardwired PowerPCs, PCI Express modules;

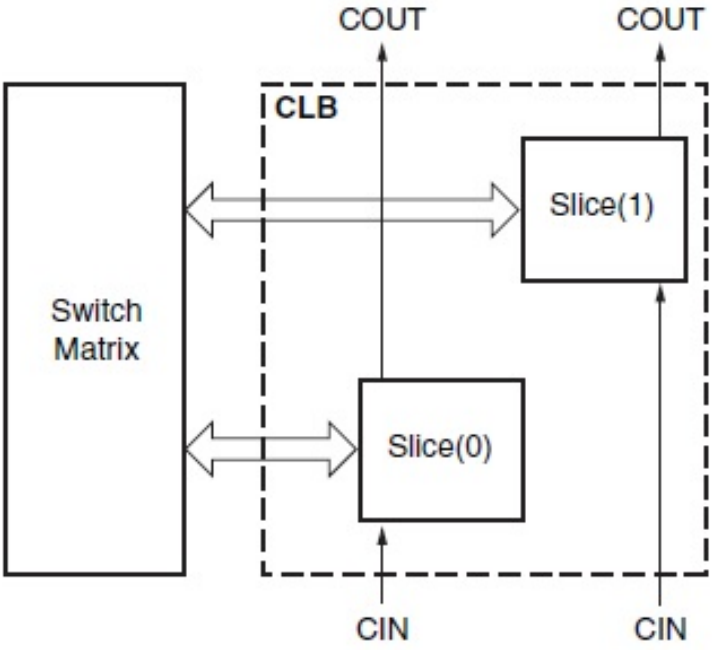


Figure 2.3: Arrangement of a slice within the CLB [13]

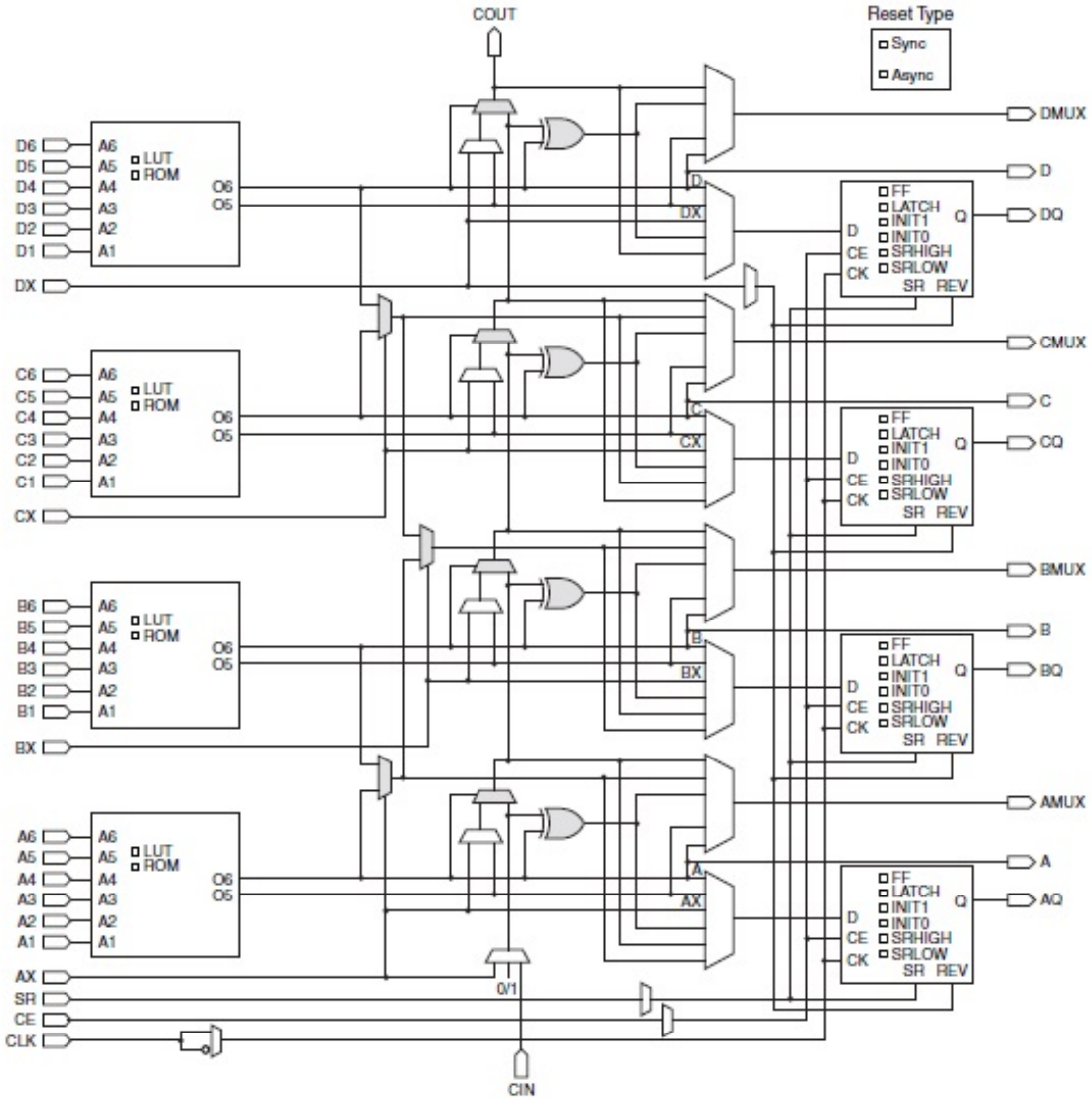


Figure 2.4: Diagram of a SLICEM [13]

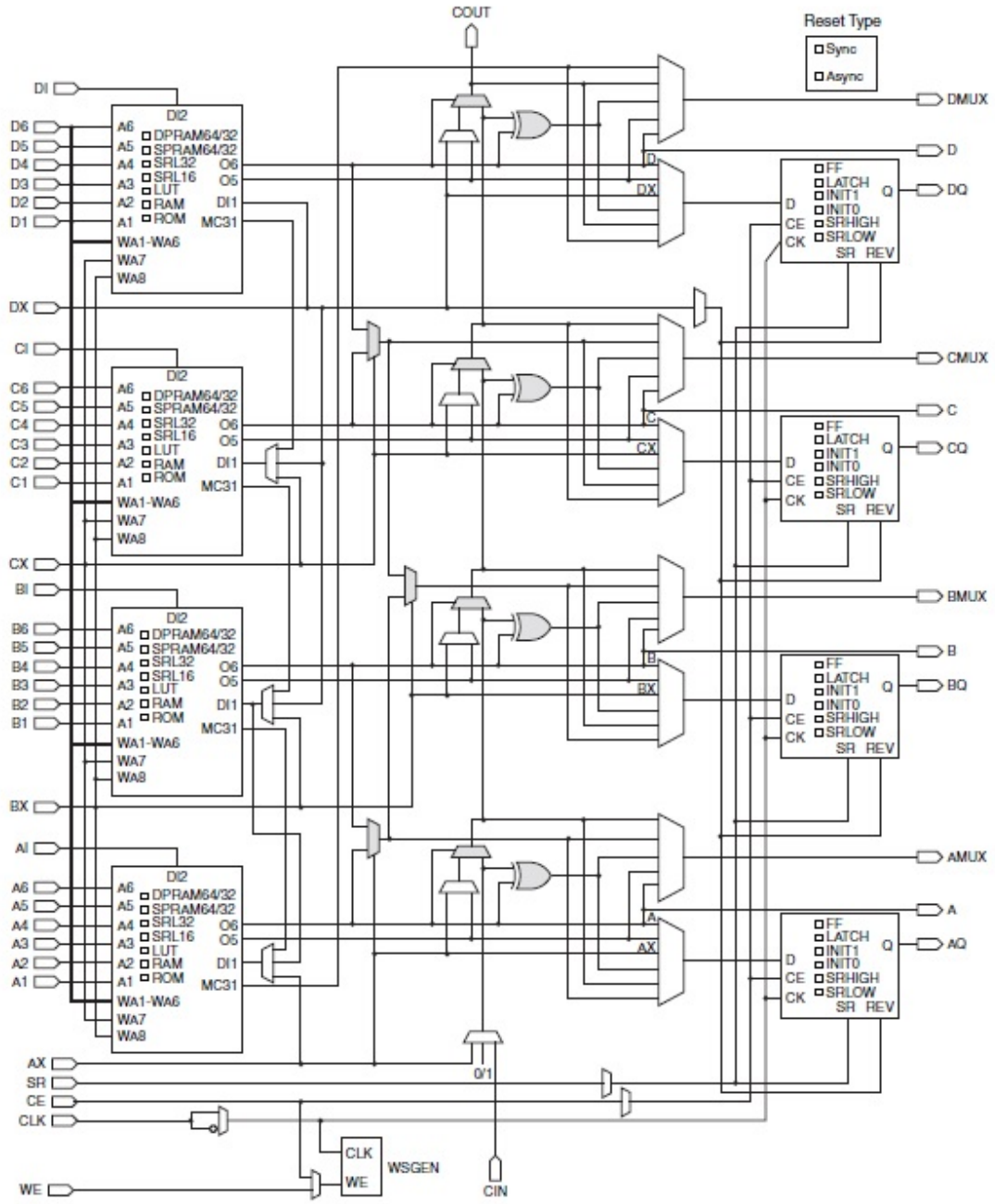


Figure 2.5: Diagram of a SLICEL [13]

But what are the advantages of FPGAs when supporting embedded systems processing power?

We have several, namely:

- Low cost of development despite of its high complexity (for small and medium markets);

- Easy to debug;
- Error tolerant;
- Reduced size;
- High reliability;
- Dedicated and reconfigurable system;
- They are recommended for the processing of parallel operations thanks to its density of logic cells, more precisely Look-up Tables and multiplexers (for non-high transfer rates), being able to run multiple channels simultaneously;
- Fast prototyping;
- Faster cycles, lower risk;
- Dynamic upgrades of the system.

However, there are also some **limitations:** which are important to have in mind when we pretend to use this technology, namely:

- Overhead due to reconfiguration time and/or additional hardware circuit delays;
- Long software compiling times due to its complexity;

Figure 2.6 presents the characteristics of some Virtex-5 family FPGA devices.

2.4.5 Modern market of FPGA

FPGA companies are no longer just trying to architect the best logic module, interconnect, and place-and-route algorithm, but they continue refining it over time, whilst their main core competency is to expand this technology to new markets. These markets have been segmented, based on application, into consumer electronics, automotive, industrial, data processing, military and aerospace, telecommunications, and others. The telecommunications field is expected to present the fastest growth, with an estimated *Compound Annual Growth Rate* (CAGR) of 9.0% from 2013 to 2019 [2], mostly due to the usage of FPGA in base stations in the updates from 2G to 3G and from 3G to *Long-Term Evolution* (LTE), and high demand of bandwidth in wireless networks. Big usage of this technology in imaging equipment, high performance computing in vehicular communications and flat panel displays, is also a growing reality, and consequently FPGAs will have a significant popularity in smartphones, tablets

Virtex-5 FPGA Family Members

Device	Configurable Logic Blocks (CLBs)			DSP48E Slices[2]	Block RAM Blocks			CMTs[4]	PowerPC Processor Blocks	Endpoint Blocks for PCI Express	Ethernet MACs[5]	Max RocketIO Transceivers[6]		Total I/O Banks[6]	Max User I/O[7]
	Array (Row x Col)	Virtex-5 Slices[1]	Max Distributed RAM (Kb)		18 Kb[3]	36 Kb	Max (Kb)					GTP	GTX		
XC5VLX30	80 x 30	4,800	320	32	64	32	1,152	2	N/A	N/A	N/A	N/A	N/A	13	400
XC5VLX50	120 x 30	7,200	480	48	96	48	1,728	6	N/A	N/A	N/A	N/A	N/A	17	560
XC5VLX85	120 x 54	12,960	840	48	192	96	3,456	6	N/A	N/A	N/A	N/A	N/A	17	560
XC5VLX110	160 x 54	17,280	1,120	64	256	128	4,608	6	N/A	N/A	N/A	N/A	N/A	23	800
XC5VLX155	160 x 76	24,320	1,640	128	384	192	6,912	6	N/A	N/A	N/A	N/A	N/A	23	800
XC5VLX220	160 x 108	34,560	2,280	128	384	192	6,912	6	N/A	N/A	N/A	N/A	N/A	23	800
XC5VLX330	240 x 108	51,840	3,420	192	576	288	10,368	6	N/A	N/A	N/A	N/A	N/A	33	1,200
XC5VLX20T	60 x 26	3,120	210	24	52	26	936	1	N/A	1	2	4	N/A	7	172
XC5VLX30T	80 x 30	4,800	320	32	72	36	1,296	2	N/A	1	4	8	N/A	12	360
XC5VLX50T	120 x 30	7,200	480	48	120	60	2,160	6	N/A	1	4	12	N/A	15	480
XC5VLX85T	120 x 54	12,960	840	48	216	108	3,888	6	N/A	1	4	12	N/A	15	480
XC5VLX110T	160 x 54	17,280	1,120	64	296	148	5,328	6	N/A	1	4	16	N/A	20	680
XC5VLX155T	160 x 76	24,320	1,640	128	424	212	7,632	6	N/A	1	4	16	N/A	20	680
XC5VLX220T	160 x 108	34,560	2,280	128	424	212	7,632	6	N/A	1	4	16	N/A	20	680
XC5VLX330T	240 x 108	51,840	3,420	192	648	324	11,664	6	N/A	1	4	24	N/A	27	960
XC5VSX35T	80 x 34	5,440	520	192	168	84	3,024	2	N/A	1	4	8	N/A	12	360
XC5VSX50T	120 x 34	8,160	780	288	264	132	4,752	6	N/A	1	4	12	N/A	15	480
XC5VSX95T	160 x 46	14,720	1,520	640	488	244	8,784	6	N/A	1	4	16	N/A	19	640
XC5VSX240T	240 x 78	37,440	4,200	1,056	1,032	516	18,576	6	N/A	1	4	24	N/A	27	960
XC5VTX150T	200 x 58	23,200	1,500	80	456	228	8,208	6	N/A	1	4	N/A	40	20	680
XC5VTX240T	240 x 78	37,440	2,400	96	648	324	11,664	6	N/A	1	4	N/A	48	20	680
XC5VFX30T	80 x 38	5,120	380	64	136	68	2,448	2	1	1	4	N/A	8	12	360
XC5VFX70T	160 x 38	11,200	820	128	296	148	5,328	6	1	3	4	N/A	16	19	640
XC5VFX100T	160 x 56	16,000	1,240	256	456	228	8,208	6	2	3	4	N/A	16	20	680
XC5VFX130T	200 x 56	20,480	1,580	320	596	298	10,728	6	2	3	6	N/A	20	24	840
XC5VFX200T	240 x 68	30,720	2,280	384	912	456	16,416	6	2	4	8	N/A	24	27	960

Figure 2.6: VirteX-5 FPGA Feature Summary [13]

and phablets with advanced touchscreen functionality, within the same period. Besides, integrating 3D acpIC with FPGAs, processor, SerDes (Serializer or Deserializer), and memory controllers are expected to act as an opportunity for market growth in the near future. Among the different FPGA technologies - SRAM, flash, and antifuse - SRAM is expected to be the largest and fastest growing technology dominating the market over the next few years, accounting for 76.1% of the overall revenue in 2012 [2, 3, 4]. In terms of value and according to a new market report published by Transparency Market Research, "Field-Programmable Gate Array (FPGA) Market - Global Industry Analysis, Size, Share, Growth, Trends and Forecast, 2013 - 2019", the market was valued at USD 5.08 billion in 2012 (EUR 3.68 billion), which is expected to reach USD 8.95 billion by 2019 (EUR 6.48 billion), growing at a CAGR of 8.5% from 2013 to 2019 [3]. Asia Pacific industry, which dominated the global market in 2012, was valued at USD 2.04 billion in that year (EUR 1.48 billion)[4], occupying 40.2% of the overall market share [2], and is expected to remain the largest market in the coming years. Growing opportunities in this segment were driven by countries such as Japan, India, South Korea, and of course, China. However, North America also occupied significant market share thanks to technological advancements coupled with sophisticated equipment across target applications.

The industry is highly consolidated, where Xilinx and Altera have been the biggest competitors in the marketing game for years, dominating the programmable logic field since the 1980s, and holding together about 90% of the overall market share in 2012 with combined revenues in excess of USD 4.5 billion (EUR 3.26 billion) [3, 4, 5]. In fact, for each company, the other is the only real competition, since other key participants such as Lattice Semiconductor, Achronix, e2v, Tabula, Microsemi and others [5], represent a much smaller slice of the market, with the strategy of producing specific applications to sub-markets.

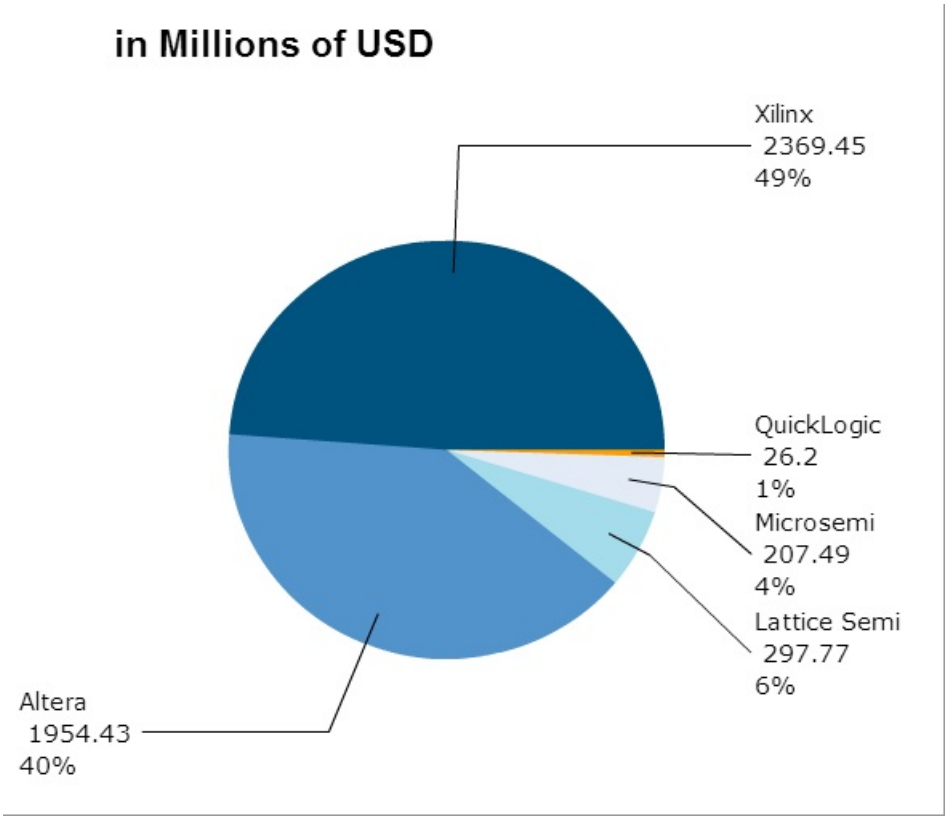


Figure 2.7: FPGA Market (July 2011) [6]

The leader in FPGAs for many years, Xilinx, has the largest programmable logic portfolio in the industry and a good variety of FPGAs in terms of cost and performance. In recent years, the popular Spartan series has covered the low-to-mid-end market (oriented to servers and medium complexity systems) while the Virtex series has covered the high-end (oriented to communications)[6]. Also, Xilinx released the Series-7 family of FPGAs (Artix-7, Kintex-7, Virtex-7) built on 28-nm process, winning the "Highly Commended Prize" Semiconductor of the year award for 2011, and the next most recent Series-6 device family (Virtex-6, Spartan-6). However, most recently they announced their next generation SoC – the "Zynq-7000" –

claiming that developers can use this platform to design smarter systems with tightly coupled software based control and analytics with real time hardware based processing and optimized system interfaces.

Achronix, a company founded in 2004, is one of the emerging suppliers for specialized, high end programmable logic and its parts are built with Intel foundry services using Intel's 22nm 3-D Tri-Gate FinFET process. They have a family of SRAM based FPGAs called "Speedster22i", which have peak performance up to 1.5GHz and densities up to 1.7 million effective acpLUT [38].

Altera has a wide and deep portfolio of programmable logic, with its FPGAs covering the low, mid and upper end markets. For the upper end markets they present the "Stratix" series of FPGAs, for the mid-range market is their "Aria" series, and covering their low cost offer is the "Cyclone" series, all of them build on 28-nm process technology. Altera has made great progress in winning market share in recent years. Many people would say that their software tools are better than those of Xilinx which has likely been an important factor in their success [38].

Lattice Semiconductor tackles the low-power and low-cost market for FPGAs. They commercialize their products as the "high-value FPGAs" of the industry, providing best performance per cost. Lattice claims to have the industry's lowest power and price SERDES-capable FPGA: LatticeECP3.

Microsemi specializes in low-power and mixed-signal FPGAs. Here are some of Microsemi's claims: The industry's lowest power FPGA, the IGLOO, and the industry's only FPGA with hard 32-bit ARM Cortex-M3 microcontroller: the SmartFusion. In 2005 this technology constituted 60% of the technology used on video processing industry being easily integrated in the embedded processors market due to the fact that the FPGA free silicon areas allow to integrate in one chip, together with FPGA logic, core processors that are usually located outside of the chip [21]. Despite the higher cost, its design flexibility is very important. In the next picture we can observe the significant growth of the FPGAs already noticeable in the market of the 90's, from 1991 until 1999 specifically. Although being old statistics, this graphic gives us an idea about the strong growth this technology already had in about two decades ago.

Today, and within the focus of this dissertation, XILINX offers new Hardwired PowerPC's devices, DSP cells, PCI Express modules and Ethernet MACs, and hence we start talking not about a FPGA but the concept of a platform/development board (Atlys/Genesys).

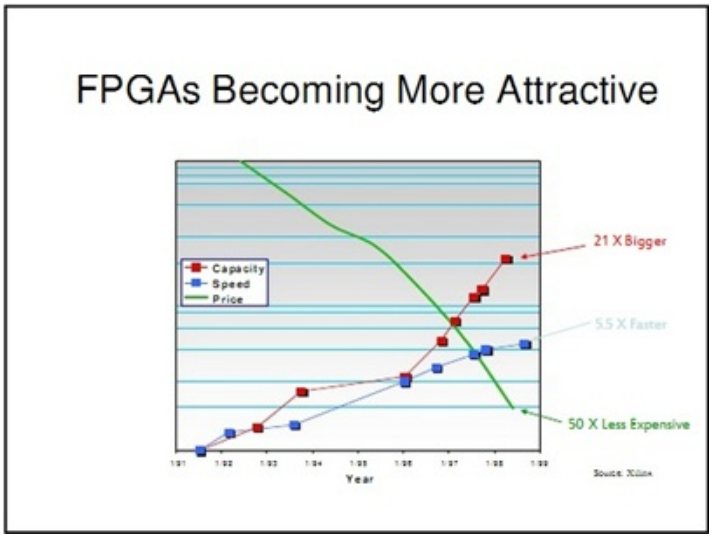


Figure 2.8: Evolution of price (green line), capacity (red line) and speed (blue line) of FPGAs from 1991 to 1999

Graph in figure 2.9 demonstrates the most usual applications. In the scheme presented

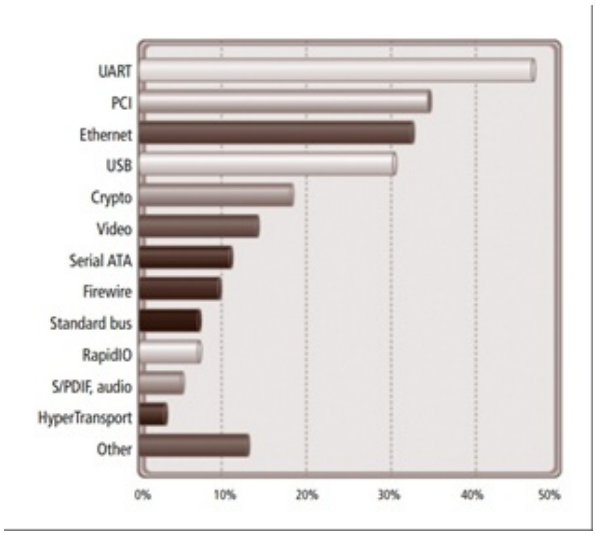


Figure 2.9: Most usual applications in 2005, where embedded processors were required [21]

on figure 2.10 we can have an idea of the steps taken in the market by Xilinx, from the wafer fabrication till the End Market, centered on communications, servers and general consume:

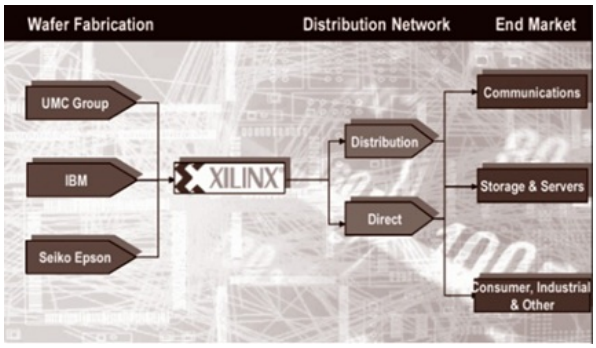


Figure 2.10: XILINXMarket [21]

2.5 FPGA Design Flow and Design methods / approaches

2.5.1 Methods

When we want to design a circuit in an FPGA we can opt between two styles of designing: using a schematic, or using a *Hardware Description Language* (HDL). Basically, when we use the schematic method, we draw our design using various elements among gates and wires, which is very practical for small designs because it is an easily readable format, despite of the difficulty in migrating between vendors and technologies. On the other hand, HDL design entry is the most common for big designs, while requiring learning and understanding at least one of the two main HDL languages, VHDL and Verilog. The first will be used along this entire project and will be discussed further in another section.

2.5.2 Design Flow

The project of an FPGA is constituted by 3 main steps:

- Design entry/synthesis;
- Design implementation;
- Device programming;

The achievement of these 3 main steps is possible due to tools called CAD (Computer-Aided Design). Here, design entry can be done either by schematic diagram method, by HDL system to describe the behavior of the circuit, or with a mixture of design entry methods. In this project, the CAD tool we'll use is named Xilinx ISE, appealing to HDL method. This software allows users to design an FPGA, gathering all these steps in the same tool, both using schematics and HDL method.

2.5.3 Design Entry / Synthesis

Using the first method, i.e. schematics, we can draw the structure of the device. This is a traditional method that is supported by a graphical tool where we can specify the required gates and how they are interconnected. This method consists in four main steps:

- Select a specific schematic tool, a device library and build the circuit. The user must choose a specific vendor and a device family at this time;
- Connect the gates using nets or wires, configuring it the way we want or that best fits the application;
- Define the I/O package pins for the device, adding and labeling the input and output buffers;
- Generate the netlist.

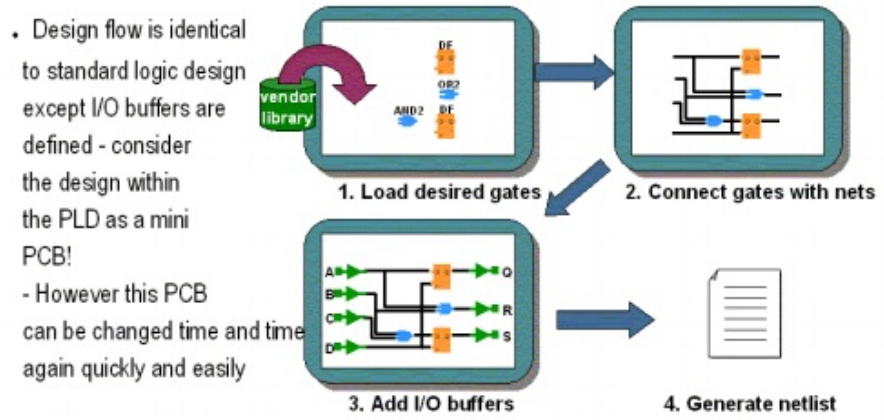


Figure 2.11: FPGA graphic design flow[15]

The netlist will be the text equivalent of the circuit allowing other programs to understand what gates are in the circuit and how they will be connected, and the names of the I/O pins. In fig. 2.12 we can see an example of the design specification netlist. Now let's imagine we need to design (draw) a circuit with approximately 10,000 equivalent gates. If a typical schematic page contains around 200 gates, we would need around 50 pages to create the design, adding all the components with its interconnections, I/Os, and generating a netlist. This would be heavily time-consuming, not to talk about 30k or 50k gate designs which would take a lifetime. This big disadvantage together with the (already mentioned) problem regarding migration between vendors and technologies, makes most of the reconfigurable

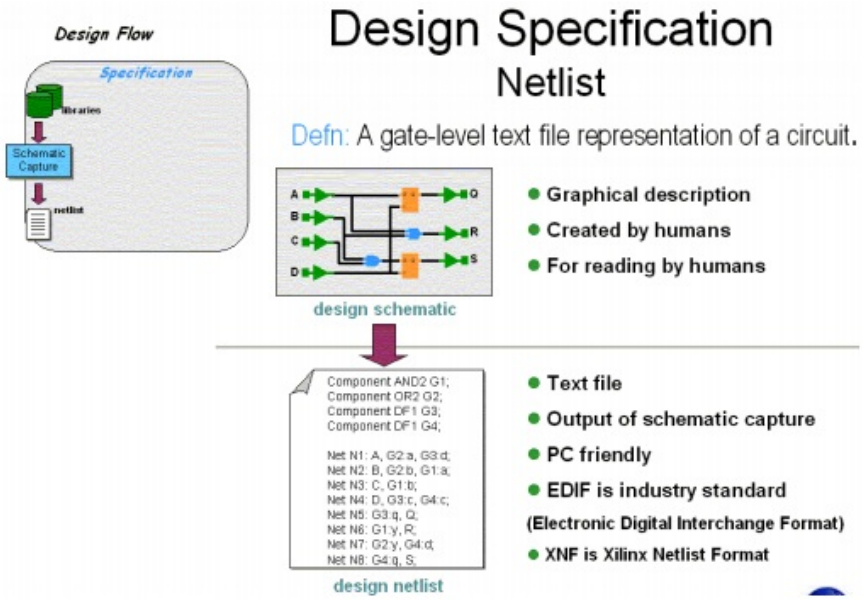


Figure 2.12: Design Specification netlist [15]

hardware designers opt for high-level design. HDL is indeed much simpler, as we can see in the example of fig. 2.13, where the design of a 16x16 multiplier is made by both methods and compared [15]. In this case, the design software Xilinx ISE WebPACK, based on the high-level description file, has to figure out what gates to use and how to interconnect it, also providing the specification of the optimization criteria, i.e. we can optimize the design for the least number of gates, optimize a certain section of the design for fastest speed and use the best gate configuration to minimize power. In opposition to schematic approach, the user can explore different solutions trying it with different vendors, device families and optimization constraints. In the final, synthesis tool automatically generates netlist file which uses the primitives proposed by the vendor in order to satisfy the logic behavior specified in the HDL file. This file is the text equivalent of the circuit with the hardware components and interconnections optimized, implementing the modeled behavior and/or structure. At this point, a behavioral/functional simulation can be made using tools provided by this *Computer-Aided Design* (CAD) software. This simulation tool verifies the functionality or timing of a circuit, allowing the user to identify functional errors and correct them before stepping to a next phase, although not providing a timing simulation, which will be done a little later in the design flow. If there are in fact any problems, it is possible to go back to the HDL file, making changes, regenerating the netlist and rerunning the simulation. Usually, until the design fulfils the desired requirements, designers spend approximately 50% of their

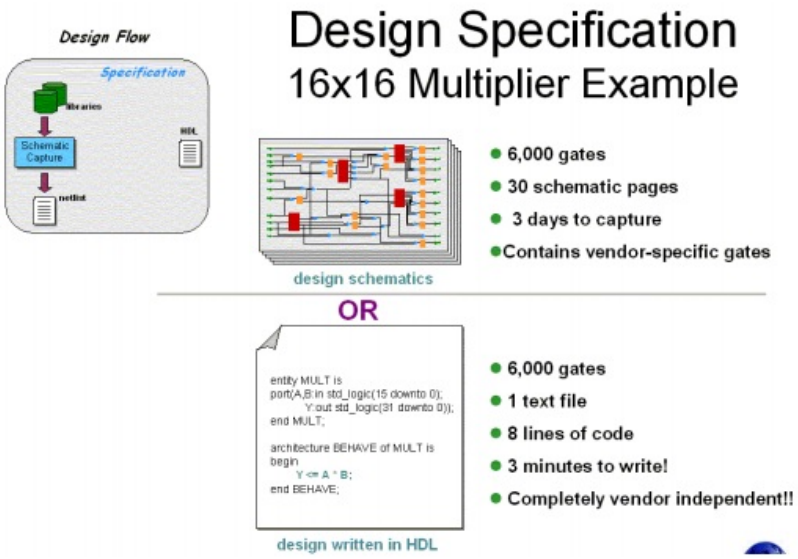


Figure 2.13: Design Specification 16x16 multiplexer - HDL vs Schematics [15]

development time going through this loop.

This method is so flexible that we can change to a different vendor or technology, whilst just a new library selection is required, and we can also try different design tools from different vendors and select the best option depending on the results.

In the final of the synthesis step, besides the netlist, resource usage and performance estimation reports are also outputted. In fig. 2.14 we can see a diagram of the design entry flow.

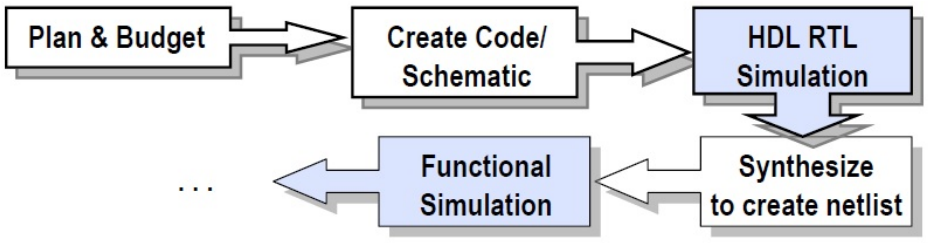


Figure 2.14: FPGA Design Entry flow [15]

2.5.4 Design Implementation

After the netlist is created fulfilling our requirements, it is time to implement this in a chip, which is referred to as device implementation. This step is constituted itself by

3 phases: Translation, Mapping and Place & Routing. In the translation phase, various programs(which can vary among vendors) are used so the designed netlist is imported and prepared for layout. During this process, it is accomplished the optimization and verification of the device associated rules, whilst a verification is made to ensure that the choice of the device was the best, depending on the device itself and the required I/O utilization. After this, comes the phase where logical symbols from the netlist are grouped into physical components (slices and IO Blocks). To finish the implementation the netlist is mapped into the FPGA primitives, i.e. a selection is made of specific modules or logic blocks where the design gates will reside. This process is called "Place", and is complemented by other process named "Route", which is the physical routing of the interconnect between the logic blocks. This is the phase that requires the longest time to successfully be completed since it is a very complex task to, mostly on large designs, determine its locations and ensure that everything is correctly connected while, at the same time, meeting the desired performance. Despite of this being an automatic tool, some vendors provide tools to manually place and route the most critical parts when we aim to achieve better performance than that obtained by automatic tools.

2.5.5 Device downloading / programming

This step consists in the download of a bitstream file which contains all the information to define the logic and interconnect of the design into the device memory of the FPGA. Since we will be working with a *Static Random Access Memory* (SRAM) based FPGA device which loses its configuration when the device is turned off, the bitstream file has to be stored somewhere, for example, in a serial PROM. When programming is applied with all non-volatile PLD, it performs the same function as the download does, with the difference that information is retained in the device even after power is removed from it. In the case of antifuse devices, programming is done once per device. In figure 2.15 [15] we can see the Xilinx FPGA typical design flow.

2.5.6 Programming approaches

When we are up to program the FPGA, we can think about three main types of architecture approaches, depending upon our time and resource requirements of the application.

When minimum resource utilization (lower cost) is demanded and long execution time is possible, serial approach is preferred being a minimalist implementation which takes more

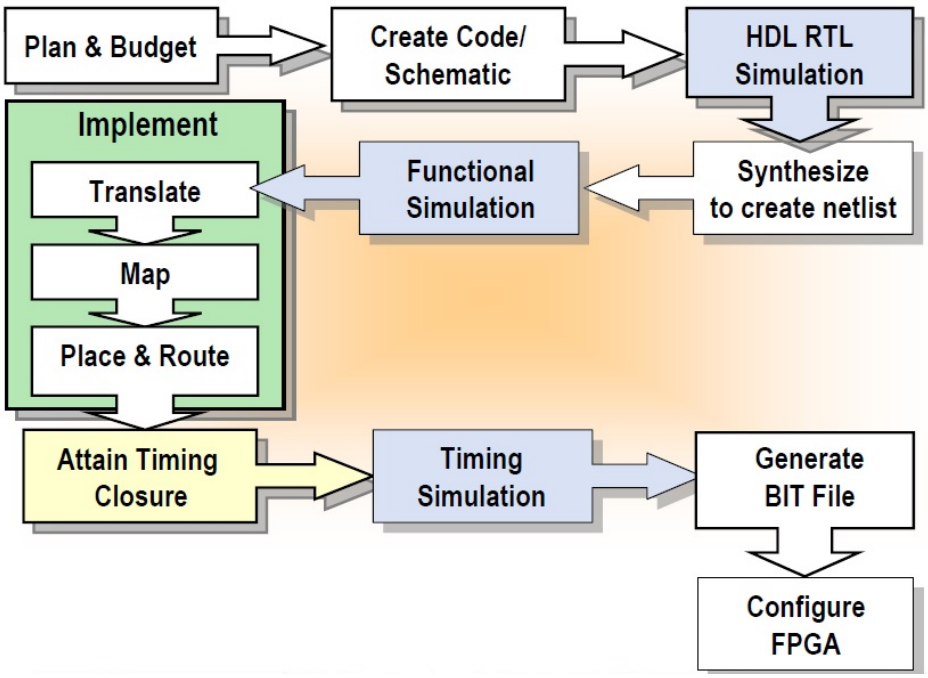


Figure 2.15: Xilinx FPGA Design Flow

clock cycle to generate a result.

On the other hand, if faster computation is the most important issue, regardless of the hardware resource utilization, then parallel approach is the best option, also called systolic approach. This style is better suited for example in the case of DSP industries where parallelism capability should be exploited, i.e., replication of hardware functions that operate concurrently in different parts of the chip. Hence this approach will naturally be the one under study in this project.

Besides these two, we also have an approach which covers both of the styles previously mentioned, suiting all applications which demand optimization of both resource utilization and execution time, and improvement of area/speed metric.

In figure 2.16 [16] we can see a scheme of all the three approaches illustrating the speed/area trade-off in FPGAs.

2.5.7 IP (Intellectual Property) Cores

IP core is basically a complex pre-tested system-level function. It simplifies the design specification step by abstracting designers from gate-level details and it also helps reducing development time. In fact, we take advantage of various benefits when using ipCores, such

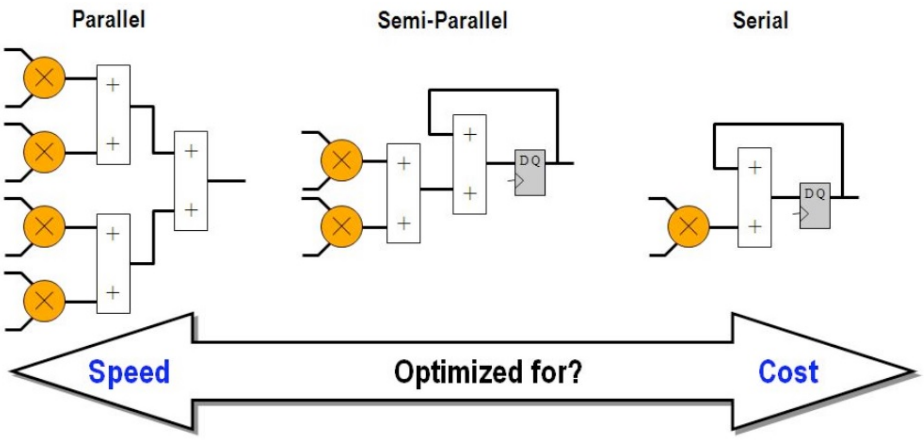


Figure 2.16: Illustration of the speed/area trade-off in FPGAs

as:

- Faster time to market;
- Simple development process;
- Minimal design risk;
- Reduced software compile time;
- Reduced verification time;
- Predictable performance/functionality.

IP Cores can have several uses namely DSP filters, Block RAMs, PCI bus interface, *Universal Asynchronous Receiver/Transmitter*s (UARTs), *Central Processing Unit*s (CPUs), Ethernet controllers, among others. They are very reliable due to the fact they are extensively tested before launched to the market, and are part of the growing electronic design automation industry trend towards repeated use of previously designed components.
They can be subdivided in 3 main categories: Hard cores, firm cores and soft cores. Hard cores are physical units based in IP design and are more commonly used for plug-and-play applications, hence not as flexible and portable as the others two types. Firm cores are similar to hard cores but configurable to various applications. Soft cores are the most flexible, and exist as a list of logic gates and its interconnections featuring an integrated circuit (netlist) or as HDL code. The latter will be used in this project integrated in the FPGA and provided by Xilinx technology, and will be discussed further with more detail.

2.6 Development board (Genesys Virtex-5 LX50T FPGA Development Board)

Genesys Virtex-5 FPGA circuit board (figure 2.17) is a complete digital circuit development platform based on a Xilinx Virtex-5 LX50T FPGA [17] already discussed in earlier sections. It includes various resources, namely Gbit Ethernet, HDMI Video, 64-bit DDR2 memory array, audio ports and *Universal Synchronous Bus* (USB) ports. This platform is compatible with all Xilinx CAD tools, including ChipScope, EDK and the free Webpack. It includes Digilent's Adept USB2 system, which offers device programming, real-time power supply monitoring, automated board tests, virtual I/O, and simplified user-data transfer facilities. There is also built into the board, a second USB programming port, based on the Xilinx programmable cable.

Figure 2.17: Genesys Virtex-5 FPGA Development board [17]

The Virtex5-LX50T is optimized for high-performance logic and offers [17]

- 7200 slices, each containing four 6-input LUTs and eight FFs;
- 1.7 Mbits of fast block RAM;

- 12 digital clock managers;
- Six phase-locked loops;
- 48 DSP slices;
- 500 MHz+ clock speeds.

It has also the following features:

- Xilinx Virtex 5 LX50T FPGA, 1136-pin BGA package;
- 256Mbyte DDR2 SODIMM with 64-bit wide data;
- 10/100/1000 Ethernet PHY and RS-232 serial port;
- Multiple USB2 ports for programming, data, and hosting;
- HDMI video up to 1600x1200 and 24-bit colour;
- AC-97 Codec with line-in, mic, and headphone;
- Real-time power monitors on all power rails;
- 16Mbyte StrataFlash for configuration and data storage;
- Programmable clocks up to 400MHz;
- 112 I/O's routed to expansion connectors;
- GPIO includes eight LEDs, two buttons, two-axis navigation switch, eight slide switches, and a 16x2 character LCD;
- Ships with a 20W power supply and USB cable;

2.6.1 Adept System Serial Port

Genesys board host two 2-wire RS-232 Serial ports (figure 2.19), one with a DB9F connector (for DTE connection), and one with a three-pin 100-mil header connector (including TX, RX and GND). A ST3232 level-shifting buffer is used to provide RS-232 signal levels on both ports. The former is the one that will be used initially, for serial communication between the computer and the FPGA.

2.6.2 Oscillators/Clocks

Genesys development board includes several clock sources available (fig. 2.20), including a 3.3V 100 MHz crystal oscillator, a socket for a user-supplied half-size DIP oscillator, and two high-speed and highly stable differential clocks sources produced by a programmable clock

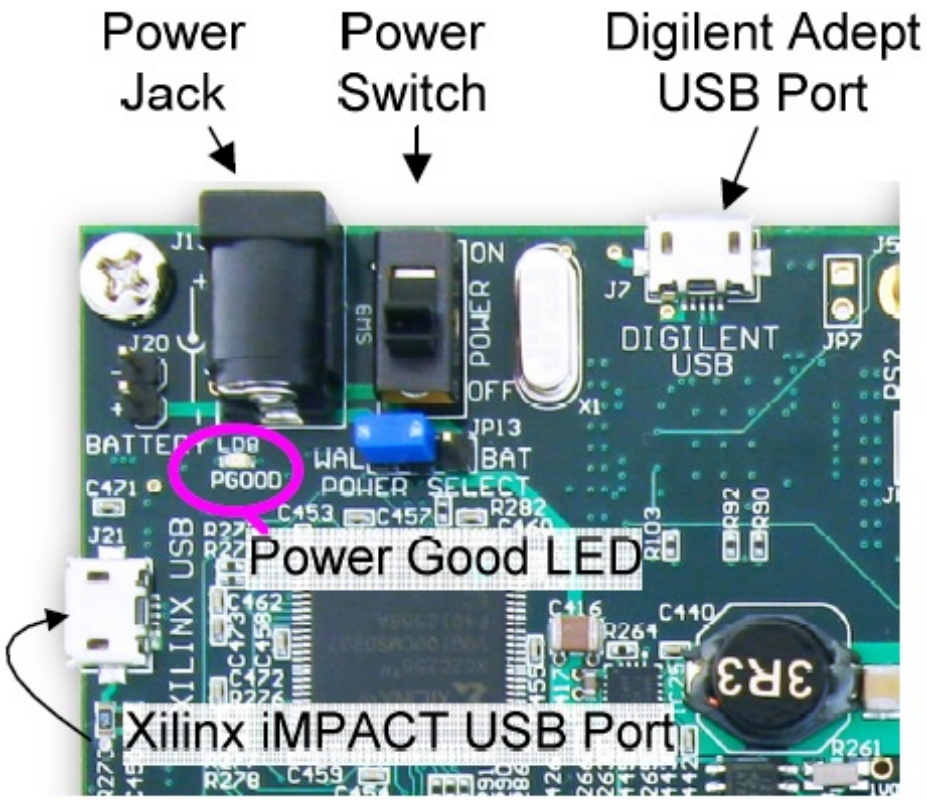


Figure 2.18: Digilent Power plug-in, on button and USB programming port [17]

generator (IDT 5V9885). This clock generator is programmed by origin to produce several clocks required by the Genesys platform, including a 25MHz clock for the Ethernet PHY, a 24.576MHz clock for the Audio codec, a 12MHz clock for the USB circuit, and two differential clocks (100MHz and 200MHz) for use by user circuits in the FPGA.

2.7 VHDL

2.7.1 History

VHDL is the acronym for Very-High Speed Integrated Circuits Hardware Description Language and is one of the two most important HDL languages of the Programmable Logic Devices world industry, together with Verilog HDL, also common in PLD design projects. This behavioral language was developed in the 80s by the defense department of USA to substitute the complex manuals which used to describe ASICs operation, an important technology which constituted some equipments sold to the american military forces. By that time, the adopted

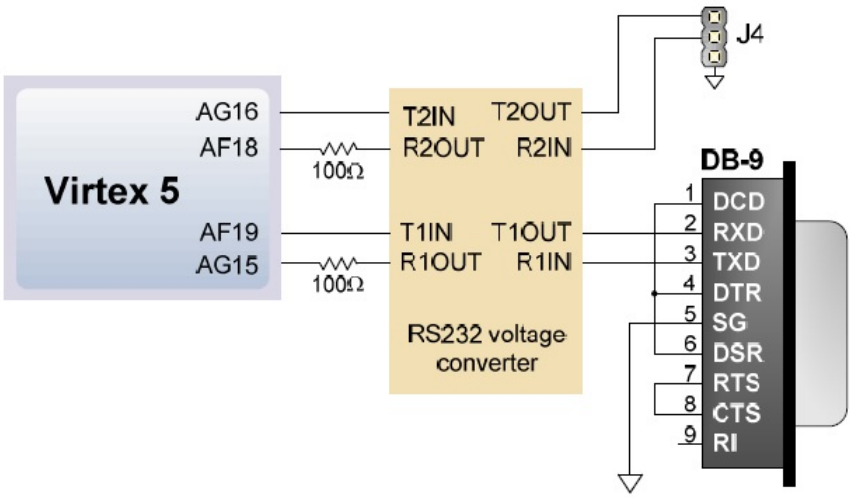


Figure 2.19: Adept system Serial port

methodology for projecting circuits was based on diagram schematics, and the department of defense aimed to develop a language that, regardless, of the original format of the circuit, could work as an efficient description of the circuit, allowing different vendors and participants to understand the overall operation, standardizing the communication. Also, in order to avoid re-inventing concepts that had been already thoroughly tested in the development of Ada programming language, the Department of Defense wanted this new language to have as much as of the syntax as possible based on Ada.

In that time, there was a big need for a language with the capability to produce a circuit without having to explicitly specify connections between components, from a textual description. Consequently, VHDL was a success and, by 1987, when its definition came to the public domain, it was standardized by IEEE, amplifying even more its utilization. This initial version of VHDL included a wide range of data types, including numerical (integer and real), logical (bit and Boolean), character and time, plus arrays of bit called bit_vector and character called string. However, VHDL naturally suffered a review, being upgraded several times during the last few years, with the last upgrade happening in 2008[18].

2.7.2 Design Cycle

VHDL is a HDL that covers all levels of the circuit design cycle, as it is confirmed in the following quote, taken from the preface of the Language Reference Manual (IEEE-1076, 2008):

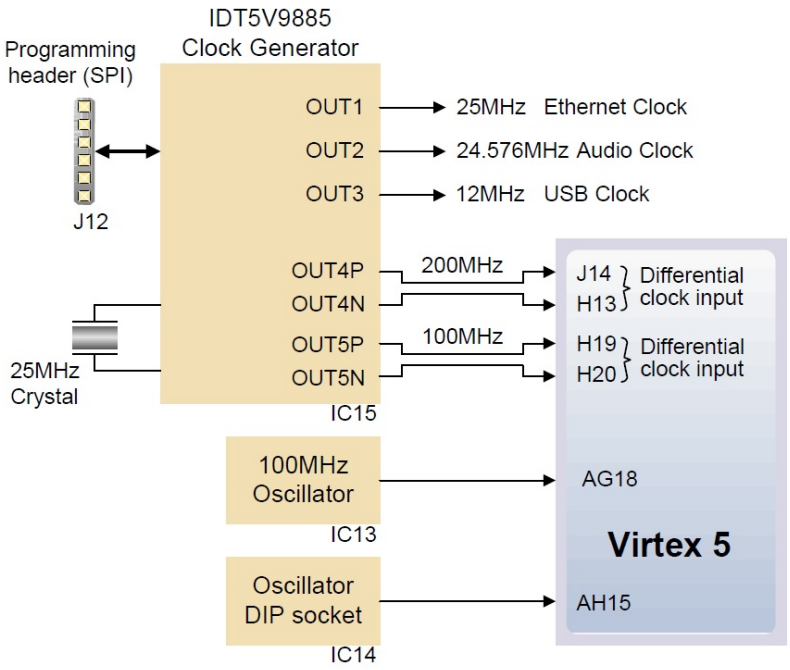


Figure 2.20: Genesys Clock System [17]

"VHDL is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development, verification, synthesis, and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware."

Notice in the keywords of this quote, "all phases", which means that VHDL is indeed intended to cover every level of the design cycle, from the system specification to the netlist, resulting in a language rather large and puzzling, although not necessarily difficult to learn. In fact, we can think in VHDL as a hybrid language which contains features that fit better one or more stages of the design cycle. Basically each stage is covered by one of the three subsets of the whole language, one for system modeling (specification phase), one for register-transfer level (RTL) modeling (design phase), and another for netlist (implementation phase). The first two stages are carried out by human designers, while the last stage is largely performed by synthesis. In figure 2.21 we can see the illustration of the idealized design cycle. [19]

In the first stage, the purpose of the VHDL system model is to be used as a formal specification of the design which can be simulated to check its functionality, and also can be used to confirm with a customer that the requirements have been fully understood, which is a very important market-oriented feature of this language. In the second phase of the design

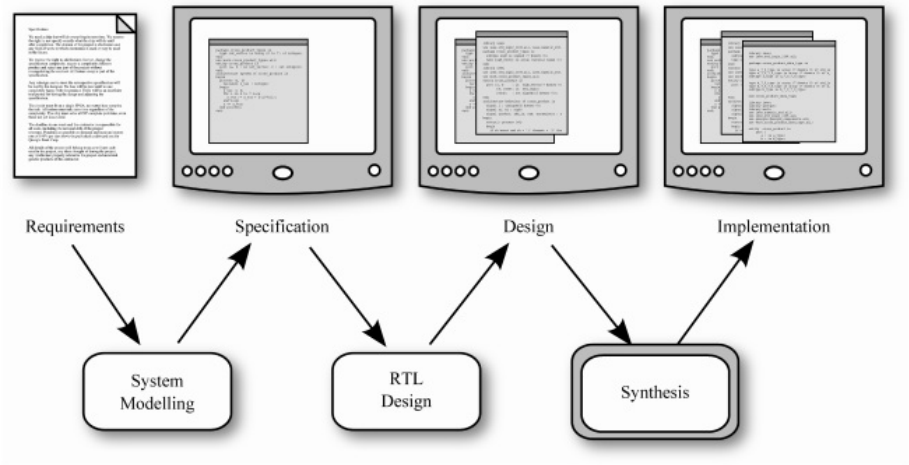


Figure 2.21: The VHDL-based hardware design cycle

cycle, this system model is converted into a RTL design in preparation for synthesis. At this stage of the design, there is a specification of the timing at the clock cycle level, and of the particular hardware resources to be used at the block level. The third and last stage of the design constitutes the synthesis of the RTL design to produce a netlist, which should meet the area constraints and timing requirements of the implementation. Of course this is in theory since, in practice, as we'll see in this project, this may not be the case and most of the times modifications are required, which will impact on the earlier stages of the design process.

2.7.3 The Standardisation Process

Part of this process requires the language to be upgraded periodically, which was established to happen each 5 years. However, in practice, and due to the industry demand, updates have been happening irregularly and hence, it is important to differentiate between the changed versions over the years. For example, the original standard created in 1987, is usually referred to VHDL-1987, making subsequent versions to be referred in a similar way, i.e., according to their year of ratification.

Below we can see a summary of the different versions and changed features of the language, for synthesis [19]

- VHDL-1987 - The original standard;
- VHDL-1993 - Added extended identifiers, xnor and shift operators, direct instantiation of components, improved I/O for writing test benches. Most of the synthesis subset of

VHDL is based on VHDL-1993;

- VHDL-2000/2002 - Nothing of relevance to synthesis;
- VHDL-2008 - Added fixed-point and floating-point packages. Added generic types and packages, enabling the use of generics to define reusable packages and subprograms. Enhanced versions of conditionals. Reading of out ports. Improved I/O for writing test benches. Unification of VHDL standards.

As we can see VHDL-2008 was the first significant change in 15 years since most of what was added has some relevance for synthesis, although some of those features are still not implemented by synthesis tools. In effect, synthesis tools are still using VHDL-1993 and will continue to do so for the foreseeable future [19].

2.7.4 Portability

Synthesizable RTL designs can have a long life span because the same design can be targeted at different technologies, revised and again targeted at a newer technology and so on, so it is an independent technology. Therefore, it is a very good practice to write using a safe, common style of VHDL which is expected to be supported for a reasonable amount of years, instead of using tool-specific tricks that might be discontinued in a short-term future.

It is also of great importance to write using a portable subset of synthesizable VHDL that will work with many different tools, because it is not unusual that a company changes its preferred tools, or that a designer is obliged to use a different synthesis tool because a different technology is being targeted.

Concluding, it is recommended that a designer uses a subset common to all synthesis tools, avoiding the more obscure tool-specific features of VHDL, unless they are really necessary. All these facts will therefore be taken into account when developing the design of this project.

2.8 Conclusions

FPGAs are PLDs which provide not only powerful processing capabilities but also a big flexibility to users in terms of design, power consumption, and development time. Due to these reasons, together with the relation low cost/increasing in logic density and performance, FPGA market is expanding as the technology evolutes.

This technology starts being a growing preferable option to implement designs which require heavy calculations, rather than other technologies such as high-end DSPs and mi-

croprocessors, since they provide the capability of parallel processing, meeting performance requirements in a single device.

VHDL and Verilog are the two most common HDL languages when designing theses devices, which together with the CAD softwares produced by big enterprises such as Altera and Xilinx, supply designers with a lot of solutions and tools when it comes to prototype a new project.

Chapter 3

Computer Vision

3.1 Introduction

Computer Vision is an extensive area of study with a wide variety of applications in various industries. It is based on real-time video processing and consequently digital signal processing, which usually requires heavy computations with complex arithmetic operations. In fact, this is a characteristic of this field of study involving different mathematical algorithms and image processing techniques, which vary depending on the final target.

In this chapter, techniques of image segmentation and edge detection will be discussed, namely the Canny Edge Detector, based on a real example of successful use of this technique.

3.1.1 Image - The base of vision

First of all, we should discuss what is the basic element which provides the existence of computer vision technology, the image. An image is an artifact that depicts or records visual perception. It can be represented in monochromatic form, which is a 2D light intensity function, f(x,y), where x and y are spatial coordinates and the value of "f" at (x,y) is proportional to the brightness of the scene at that point.

In a digital world, we have digital images which are a numerical representation of a two-dimensional image. Basically it is an image that has been discretized in spatial coordinates and in brightness and colour, and is most commonly represented by a 2D integer array, or a series of 2D arrays, one for each colour band. The digitized brightness value is called grey level. In this array, each element is commonly called "pixel", derived from the term "picture element", which can represent several grey levels. Each pixel corresponds to a fraction of a

real object in the 3D world, which reflects part of the light that hits it, and absorbs the other part. The reflected light reaches the array of sensors used to image the scene and one of these sensors makes up a pixel which in turn corresponds to a small patch in the entire image scene. The value that is recorded by the sensor depends on its sensitivity curve. Let's explain it more clearly: when a photon of a certain wavelength reaches the sensor, its energy is multiplied with the value of the sensor's sensitivity curve at that wavelength and is accumulated. The total energy collected by the sensor, during the exposure time, is then used to compute the grey value of the pixel.

3.1.2 Computer vision backbone – image processing

The reason there is so much research around image processing field of study, is that it has multiple useful purposes. Let's name some of them:

- Image enhancement – improves the quality of an image in a subjective way, usually by increasing its contrast;
- Image compression – using as few bits as possible to represent the image, with minimum deterioration in its quality;
- Image restoration – image improvement in an objective way, for example by reducing its blurring;
- Feature extraction – highlighting certain characteristics of the image that can be used to identify its contents.

All these techniques basically constitute a transformation in the image, which is performed using operators, as we will discuss in further topics.

When we talk about image processing, we should mention the parameter directly implied in it, the image quality. This is the most important issue when processing an image, and, in fact, it is also a complicated, largely subjective concept. However, there are 4 very important factors which determine quality factor:

- It isn't noisy – noise is the result or random variation of brightness or color information, and it can be produced by the sensor and circuitry of the camera;
- It is not blurred – image blurring is caused by incorrect image capturing conditions, for example, out of focus or relative motion of the camera;

- It has high resolution (fig. 3.3 [21]) - the resolution of an image is the amount of detail in it and it directly depends on the number of pixels used to represent a scene, and the number of grey levels used to quantize the brightness values;
- It has good contrast – a good contrast means that the grey values of the image range from black to white, making use of the full range of brightness to which the human vision system is sensitive.

3.2 Image Enhancement

Image enhancement is the process of improving an image so its looks subjectively better to our eyes, even without any specific requirement, but only by consideration of its detail or contrast or even unwanted flickering. Images can be enhanced by removing additive noise and interference, increasing its contrast, and decreasing its blurring. All these procedures can be achieved through several techniques like smoothing and low pass filtering, sharpening or high pass filtering, histogram manipulation and algorithms that remove the noise, while avoiding blurring the image.

3.2.1 Anisotropic diffusion

In image processing and computer vision, anisotropic diffusion is a very important non-linear and space-invariant technique that generalizes Gaussian filtering, and has the target of reducing image noise while maintaining significant part of the image content, namely edges, lines or other details which are relevant for the image interpretation. In this technique, an image generates a parameterized family of increasingly blurred images, based on a diffusion process. Each resulting image of this process is a single convolution between a 2D isotropic Gaussian filter with the image, where the width of the filter successively increases. Each result of this process is a linear and space-invariant transformation which reduces noise, smoothing the image and also reducing the detail. As an example, we can see in fig. 3.1 [21] an application of this technique where, as the scale increases, more and more features disappear, only remaining the most prominent image features, despite of being very blurred.

We can notice that the spots where the borders of the distinct image regions meet, turn progressively blurred, and the gradient magnitude which measures the contrast of the image in those spots gradually diffuses, only leaving the borders with the strongest contrast. In figure 3.2 [21] we can see the corresponding gradient magnitude images to those shown in

figure 3.1, where borders between forms are enhanced.

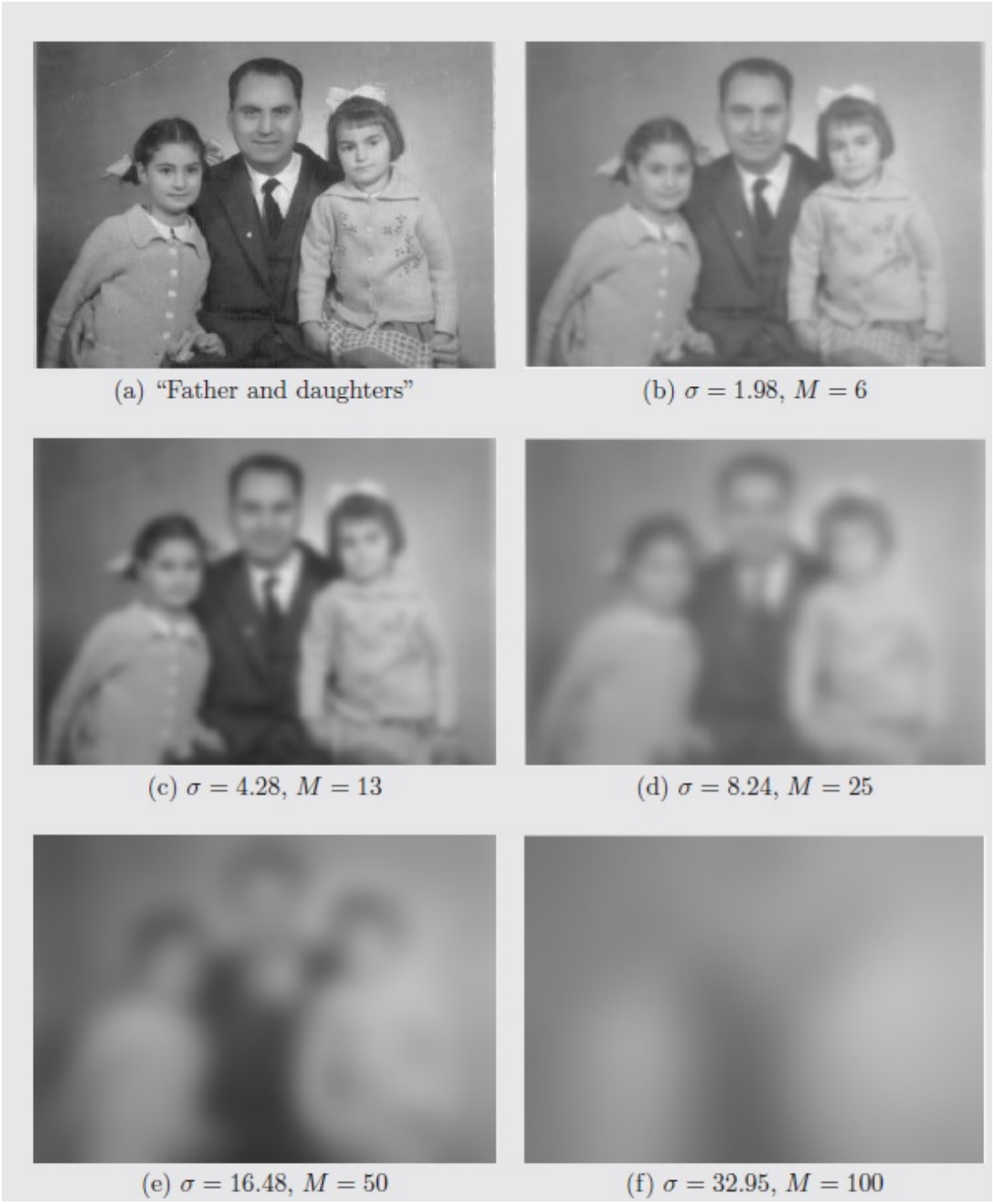


Figure 3.1: As filter scale increases, less information remains. Filter used has size (2M+1)x(2M+1)

Figure 3.2: As the scale of the filter increases, transition between regions becomes less and less sharp

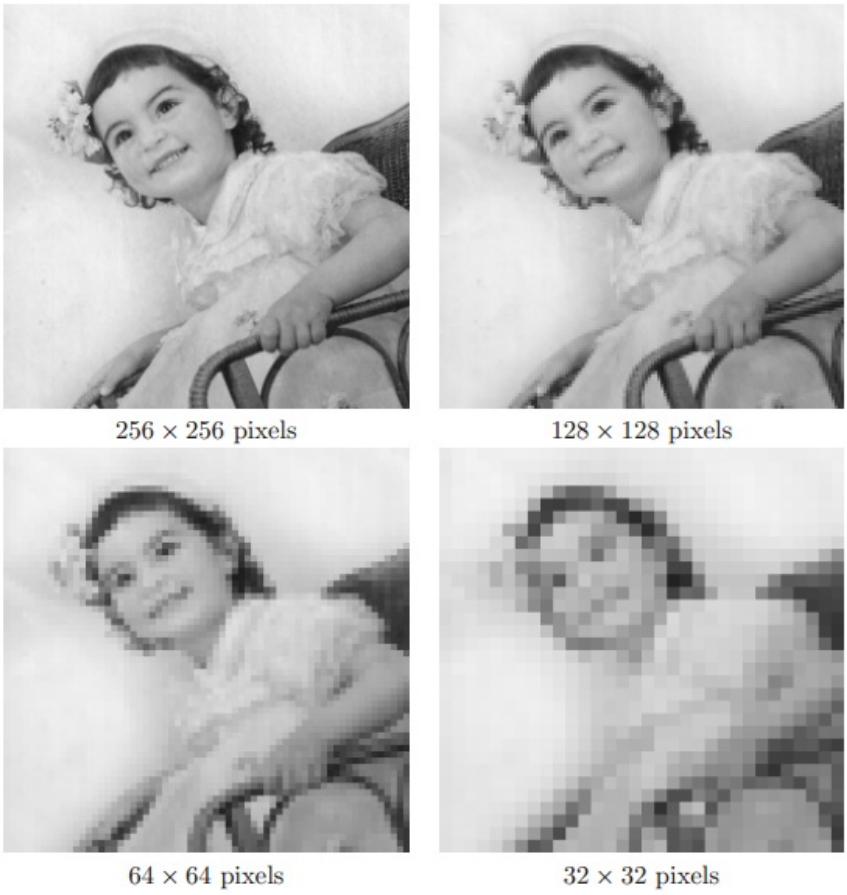


Figure 3.3: Graphical result of keeping the number of grey levels while decreasing the number of pixel of an image

3.3 Image Segmentation and Edge detection

When we are working with computational vision systems, there are a set of techniques being used to prepare an image as an input of this system.These techniques aim the extraction of information from an image, in such a way that a lot of information is discarded, obtaining an output image with a lot less information than the original, but where the remaining data is much more relevant for the modules of the automatic vision system than the discarded one. In this project we'll focus on two of these techniques: image segmentation and edge/boundary detection.

The purpose of image segmentation and edge detection is basically to extract the outlines of different regions in the image, i.e., to divide the image into regions constituted by pixels

which have something in common, as for example brightness or color, which may indicate that they belong to the same object or part of it. Figure 3.4 is an example of image segmentation [23].

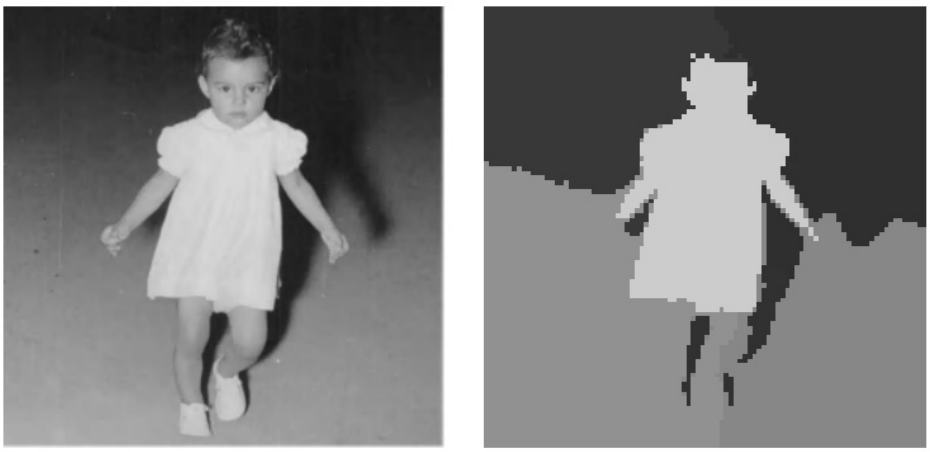


Figure 3.4: Example of Image Segmentation

To divide the image into different regions we can use simple methods like histogramming and thresholding. To create the histogram of the picture, we plot a function representing the number of pixels with a specific grey value, versus the grey value itself. Let's give an example: imagine we have a picture where there is a bright object on a dark background, and we want to extract the object out of the image. As we can see in figure 3.5a, the histogram of the picture has two peaks and a valley between them. To enhance the object, firstly we determine a threshold level that corresponds to the "valley" of the histogram, and after, we can label all pixels which are over grey level "t0" as object pixels, while the pixels with grey level under "t0" will be labeled as background pixels.

Sometimes, pixels near the boundaries of the objects aren't sharply defined, complicating the differentiation between background and object pixels due to similar grey values. This implies that there won't be any clear "valley" in the histogram, which makes us adopt other techniques to overpass this problem, such as hysteresis thresholding. In this method, instead of one threshold level, we use two threshold levels on either side of the "valley", as we can see in fig. 3.5b. Pixels with higher grey values than the upper level t2 are core object pixels, the ones with lower grey values than t1 level compose the background, while pixels between both levels are only object pixels if they're adjacent to a pixel which is a core object pixel.

But what is in fact labeling a pixel? When we want to extract some object or some specific

form from an image, we basically identify the pixels which represent it and label them. All pixels which constitute the object have the same label while the other pixels constituting the background are given a different label. So basically, we create an array of the same size of the original image, but constituted by labeled pixels of our interest, which in practice are pixels with a certain value/class, with a symbolic meaning only. That is labeling.

In the field of image analysis, this technique is an important process with various applications such as analysis of remote sensing data, analysis of medical data, analysis of astronomical data, and many others [29, 30]. It is important to notice that the result of these processes is significantly affected by the results accuracy of this technique.

For a long period of time, different image segmentation methods have been studied and they can be classified into three categories: Edge-based segmentation, Region-based segmentation and characteristic feature thresholding or clustering. In this study, and since our purpose is to highlight objects in an image by the recognition of its boundaries, we'll focus on Edge-based Segmentation, also known as "Edge Detection".

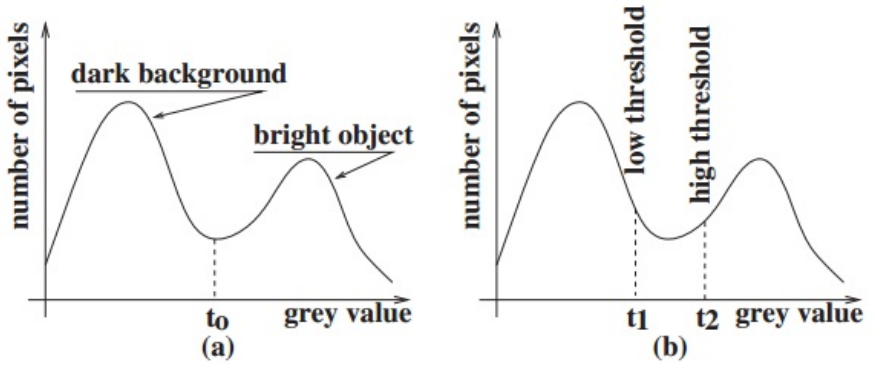


Figure 3.5: Image segmentation histogram with a) one threshold level and b) with hysteresis threshold

3.3.1 Edge Detection

Edge Detection is the name of a set of mathematical methods which have the purpose of identifying sudden discontinuities (figure 3.6 and 3.7) in an image, such as sharpness or brightness changes. It is a fundamental tool in image processing field of study, machine vision and computer vision, particularly in the areas of feature detection and feature extraction.

Edges extracted from 2D or 3D images can be classified as viewpoint dependent or viewpoint independent. The latter usually reflects inherent properties of 3D objects, like surface

markings and shape. A viewpoint dependent edge can change as the viewpoint changes, and typically reflects the geometry of the scene, such as objects occluding one another.

Despite of certain literature consider the detection of ideal step edges, in practice the edges obtained from natural images are not ideal edges, but in fact they are affected by some parameters, namely:

- Focal blur caused by a finite depth-of-field;
- Penumbral blur caused by shadows created by light sources;
- Shading at a smooth object.

But during recent years, substantial (and successful) research has been made, and nowadays researchers use a Gaussian smooth step edge as the simplest extension of the ideal step edge model, for modeling the effects of edge blur in practical applications, as we'll further study with more detail.

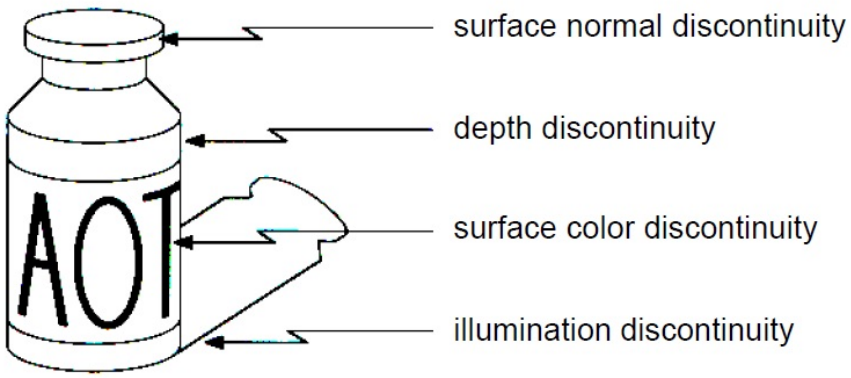


Figure 3.6: Origin of Edges

However, edge detection may be non-trivial when the objects in the scene aren't particularly simple and the illumination conditions can't be controlled. Let's consider the one dimensional signal in fig. 3.8 [33]. At a first look we should say that it clearly exist an edge between the 4th and 5th pixels with grey values of "4" and "152", respectively. Yet, that wouldn't be so clear if the grey values difference between these two pixels was minimal, and the grey value differences between the adjacent neighbouring pixels was higher, leading us to some uncertainty. Hence, it would be subjective to say that there should be an edge in the corresponding region because, actually, this could be a case with the presence of several edges.

In this case, threshold levels have to be carefully analysed to establish trustful levels which can clearly identify true edges. This is indeed, a good example case of why edge detection technique may not be always a trivial problem.

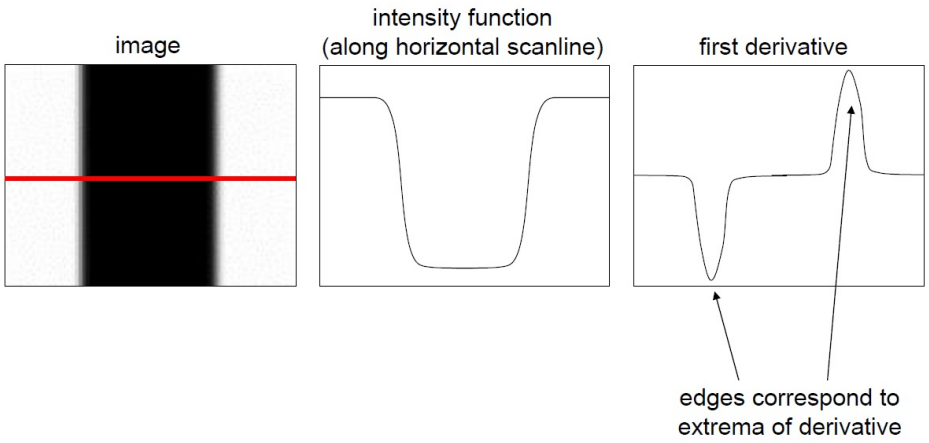


Figure 3.7: Edges characterization. Edges are detected in sudden contrast changes, and through first derivative we can mathematically detect this sudden discontinuities with its local maxima and minima.

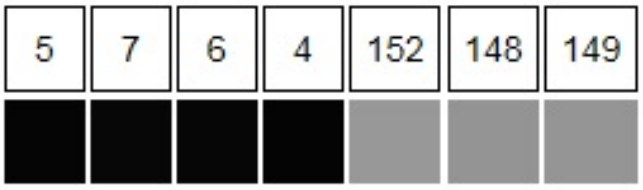


Figure 3.8: Dimensional signal with different intensity pixels

Since the appearance of image processing techniques, the number of edge detector methods has increased continuously, existing several algorithms with different approach characteristics. Most of these algorithms include the three main steps of an edge detector: smoothing, differentiation and labelling. They differ in their smoothing filters, differentiation operators, labelling processes, goals, computational complexity and the mathematical models used to derive them. However, some of these methods can also be categorized separated in two big groups: search-based and zero-crossing based.

Search-based method computationally measures the edges strength, usually through a first-order derivative expression, such as the gradient magnitude, and after it searches for local directional maxima of the gradient magnitude, calculating its direction, so it can estimate the

local orientation of the edge. On the other hand, zero-crossing based method calculates the second-order derivative expression of the image, searching for zero-crossing of the Laplacian or the zero-crossings of a non-linear differential expression. In both methods, the same pre-processing step (smoothing stage) is taken, called, as already mentioned, Gaussian smoothing. In this work, a common search-based step-edge method will be used, called Canny Edge algorithm.

Can edge detection lead to image segmentation?

Edge detection can't lead us directly to image segmentation. In fact, detected edges are usually fragmented and have gaps. Hence, to segment an image using edge detection, we have to follow one of these two ways:

- Use hysteresis edge linking in combination with some further post-processing;
- Use a Laplacian of Gaussian filter to detect edges.

In this study, we'll take the first route, although partially, as a step of Canny Edge Algorithm ("Edge Linking by hysteresis").

3.3.2 Canny Edge Algorithm

Even though it is quite old (it was created in 1986 by John F. Canny) [29], Canny edge algorithm has become one of the standard edge detection methods and it is still used in several applications. Basically it is an operator that uses a multi-stage algorithm to detect a wide range of edges in images, and it was created with the aim of being optimal (for the step edges class) regarding the following criteria:

- Detection: Maximize the probability of detecting real edge pixels while minimizing the probability of detecting non-edge pixels, i.e., maximize the signal-to-noise ratio;
- Localization: The detected edges should be as close as possible to the real edges;
- Number of responses: One real edge should be marked as a detected edge just once, and where possible, false edges shouldn't appear due to noise.

The algorithm is based in 5 separate main steps [33, 34]

- Smoothing: Blurring of the image to remove noise.
- Finding gradients: Edges are marked where the gradients of the image have large magnitudes;

- Non-maximum suppression: Only local maxima are marked as edges;
- Double thresholding: Potential edges are determined by thresholding;
- Edge tracking by hysteresis: Final edges are determined by suppressing all edges that aren't connected to a very certain (strong) edge.

Let's study each of these steps in detail.

3.3.2.1 Smoothing

It is inevitable that all images taken from a camera will contain noise. To prevent that noise to interfere with the acknowledgement of the edges, we have to remove it, smoothing the image through the application of a Gaussian filter. In equation (3.1) [28] it is represented the kernel of a Gaussian filter with a standard deviation of $\sigma = 1.4$.

$$\mathbf{B} = \frac{1}{159} * \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} \tag{3.1}$$

This Gaussian filter, also known as Gaussian blur, is a linear low-pass spatial-domain filter whose impulse response is a Gaussian function (or an approximation to it), that when convolved with an image, achieves the smoothing effect due to the elimination of high-frequency components. In practice, the convolution of this function with the image replaces each pixel of the image with an average of its neighbourhood. We can see the behaviour of this function in fig. 3.9 [25] where a Gaussian curve in a 2D domain is represented.

Also, it is important to notice that smoothing improves as the variance of the Gauss function raises and, the larger is the width of the Gaussian mask, the lower is the detector's sensitivity to noise. Fig.3.10 [25] is an illustration of the shape of a Gaussian filter as a function of the variance.

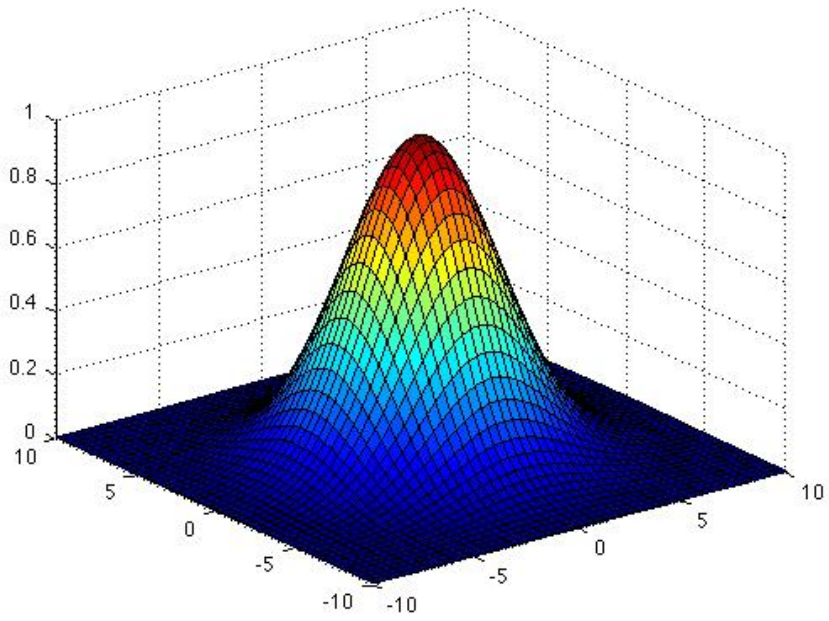


Figure 3.9: Gaussian Curve in 3D domain

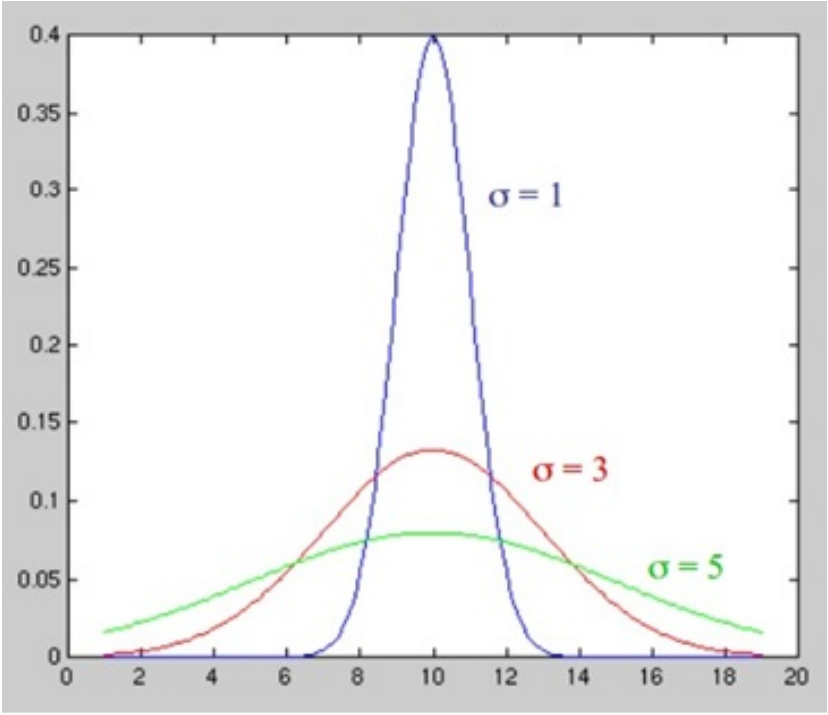


Figure 3.10: Gaussian curve as a function of variance

In the following pictures it is represented a random picture, its monochromatic representation and its result after being processed by this step.

Figure 3.11: Rugby Ball

Figure 3.12: Monochromatic representation of the Rugby Ball

Figure 3.13: Gaussian Smoothed representation of the Rugby Ball

3.3.2.2 Finding Gradients

Canny algorithm basically finds edges where the gray scale intensity of the image changes the most. This is achieved by calculating gradients of the smoothed image through an algorithm called Sobel-operator. Firstly, we have to approximate the gradient in the x- and y-direction by applying the kernels K_{GX} and K_{GY} in equations (3.2) and (3.3) respectively, i.e., convolving these two kernels with the image.

$$\mathbf{K_{GX}} = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \tag{3.2}$$

$$\mathbf{K_{GY}} = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{3.3}$$

The gradient magnitudes, also known as edge strengths, can be calculated as an Euclidean distance measure by applying the law of Pythagoras, as shown in the following equation:

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{3.4}$$

where Gx and Gy are the gradients in the x- and y-direction respectively. To reduce the computation complexity, the equation is simplified by applying Manhattan distance measure

as shown in the following equation:

$$|G| = |G_x| + |G_y| \tag{3.5}$$

Despite of these calculations, edges are typically broad and thus do not exactly indicate where the edges are. Hence, to overpass this problem, the direction of the edges must be determined and stored as shown in equation (3.6).

$$\theta = \arctan(\frac{|G_x|}{|G_y|}) \tag{3.6}$$

In the following pictures we can see the results of the X and Y-direction gradient operators over the image.

Figure 3.14: Output image from X-direction gradient operator

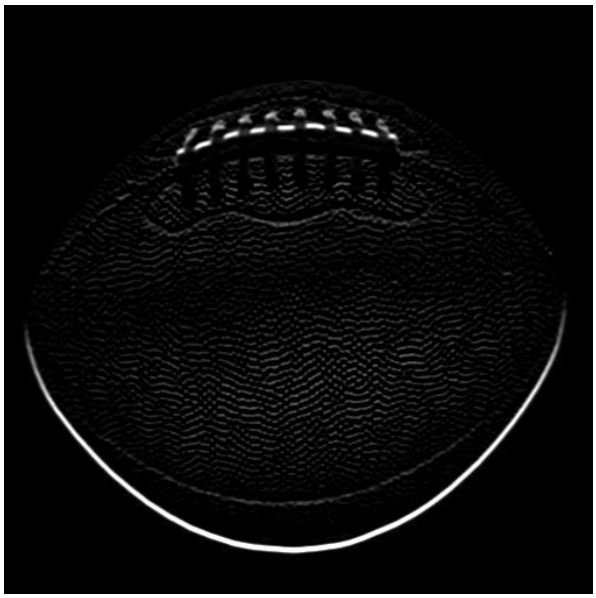


Figure 3.15: Output image from Y-direction gradient operator

3.3.2.3 Non-Maximum Suppression

In this step, as the name indicates, we eliminate the weaker edges in the gradient image and keep the strong ones. Basically this is done by preserving all local maxima in the gradient image, and deleting everything else. The algorithm to be implemented is the following:

- Round the gradient direction to the nearest 45 degrees, corresponding to the use of an 4-connected neighborhood, i.e., there are only four possible directions when describing the surrounding pixels – 0 degrees (horizontal direction), 45 degrees (positive diagonal), 90 degrees (vertical direction), or 135 degrees (negative diagonal). The edge orientation has to be resolved into one of these four directions, depending on which is closest to it. In figure 3.16 [28] is a representation of a semicircle subdivided in these 5 regions. Basically, edge orientation within the yellow range (0 to 22.5 and 157.5 to 180 degrees) is set to 0 degrees, in the green range (22.5 to 67.5 degrees) it is set to 45 degrees, in the blue region (67.5 to 112.5 degrees) it is set to 90 degrees, and in the red interval (112.5 to 157.5 degrees) it is set to 135 degrees;
- Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient direction, i.e., if the gradient direction is north (theta=90 degrees), compare to the pixels in the north and south sides;
- If the edge strength of the current pixel is largest, preserve the value of the edge strength.

If not, suppress the value.

As an example we can observe figure 3.17 where almost all pixels point to the north direction. Consequently, all pixels above and below are compared, to determine the ones with greater gradient value. The pixels which are maximal are marked, and all the others are suppressed. In the picture it is clear that marked pixels are delimited with a white border line. These are the pixels which have resulted from this step.

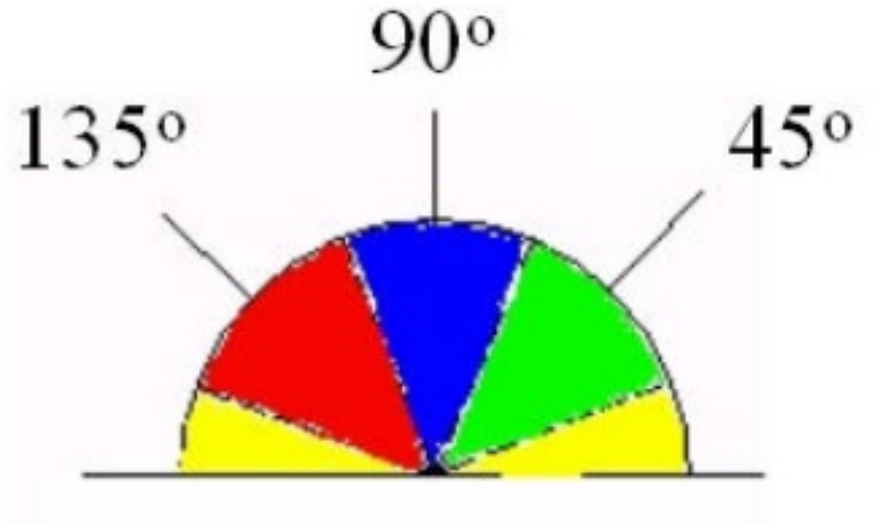


Figure 3.16: Gradient orientation

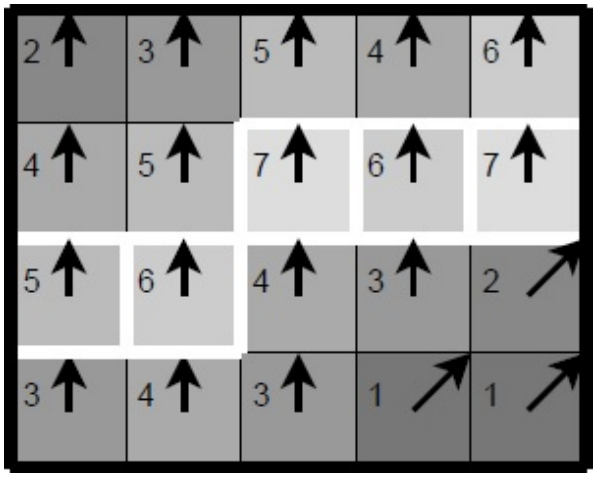


Figure 3.17: Illustration of non-maximum suppression. Edge strengths are marked with numbers and colors, and gradient directions are shown as arrows

3.3.2.4 Double Thresholding

After these 3 steps, many of the resulting pixels are true edges in the picture, but there are still a lot of pixels that are not valid. Some may be caused by noise or color variations for instance due to rough surfaces. So our goal in this step is to discern between "fake" and real true strong edges. This is achieved by the implementation of threshold levels, so that only edges with gradient value higher than a certain pre-determined value will be preserved. In Canny edge algorithm, a double thresholding technique is used, based on the following procedure: edge pixels stronger than the high threshold are marked as "strong", edge pixels with lower value than the low threshold are suppressed, and edge pixels between the two threshold levels are labeled as "weak".

3.3.2.5 Edge Linking by hysteresis

This step is basically a continuous part of the previous stage. Here we consider strong edges as certain edges and they are automatically included in the final edge image. Weak edges are only considered if they are connected to strong edges, otherwise they are suppressed.

The logic is that, with proper adjustment of the threshold levels, noise and other small variations are unlikely to result in a strong edge. Therefore, strong edges will practically be due to true edges in the original image, whilst weak edges can either be due to true edges or noise/color variations, despite of the latter being a minor case. Weak edges due to true edges are much more likely to be connected directly to strong edges.

In fig. 3.18 is represented the design flow of the complete cycle of canny algorithm.

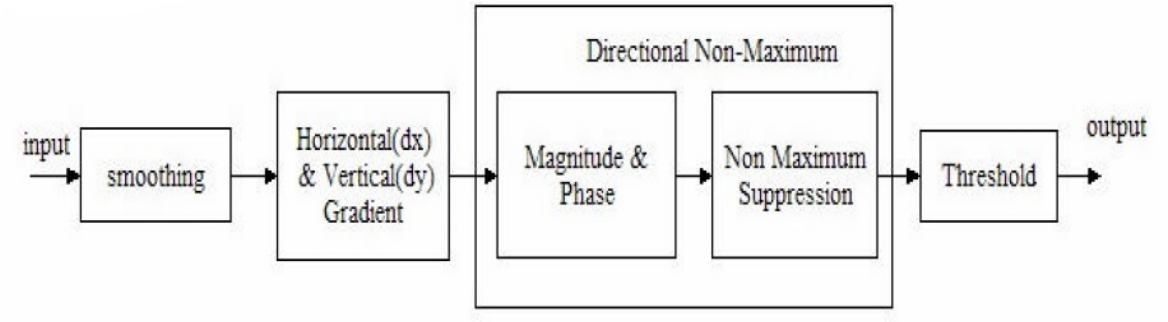


Figure 3.18: Canny Algorithm Design Flow

In figure 3.19 it is illustrated the final result of the Canny Edge Algorithm over the model picture.

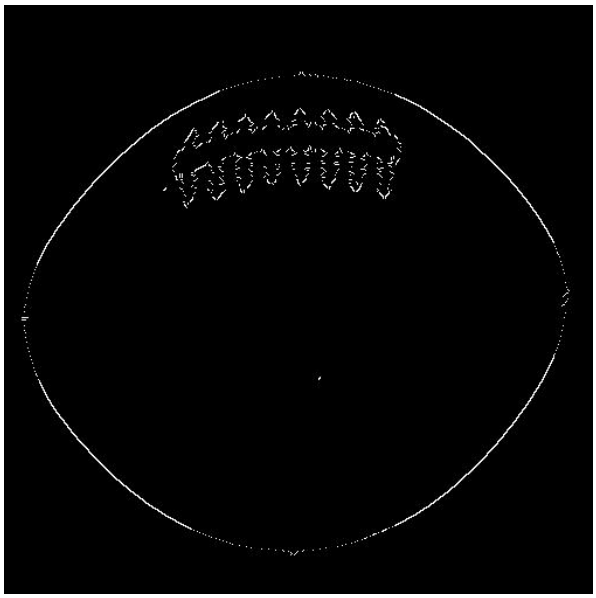


Figure 3.19: Final Output of the Canny Edge Algorithm

As already mentioned in the beginning of this section, Canny Edge Algorithm is one of the most common algorithms used all around for edge detection purposes, mostly when processing speed is an important characteristic to consider. Among various applications, it can be applied in the robotic vision field for different goals. One good example that will be the base for the development of this project, is the real case of the robotic football team of the university of Aveiro. The robots of CAMBADA team, among other techniques, take advantage of image recognition techniques, namely the Canny Edge Detection algorithm, to recognize objects on the fly. This algorithm is actually part of a bigger vision system that we will now analyse with more detail.

3.4 CAMBADA Vision System Software Architecture

The software architecture of the vision system of CAMBADA robots is based on a hybrid system, which in fact is a distributed paradigm that accomplishes several tasks in different modules. This hybrid system is based on 3 main different modules (figure 3.20): Morphological Processing Sub-System, Utility Sub-System and Color Processing Sub-System [32]. As already (lightly) mentioned in the introduction of this work, Morphological processing sub-system is an independent color sub-system, recently developed by CAMBADA, to overpass recent changes in the RoboCup league environment conditions.

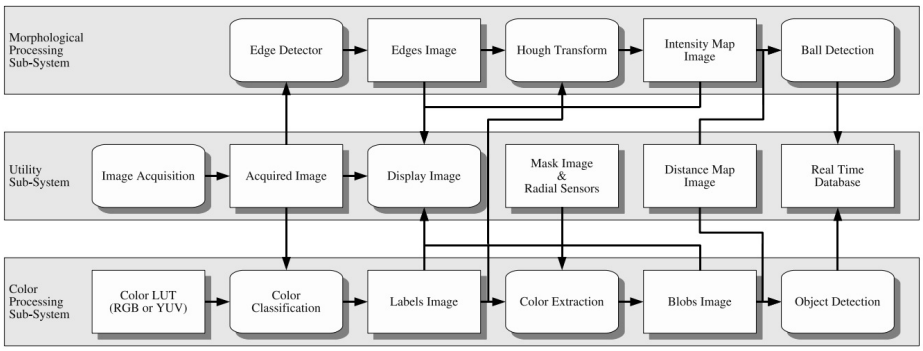


Figure 3.20: Software architecture of the vision system developed by CAMBADA robotic football team

This sub-system does a morphological analysis of the image, with the goal of detecting round objects in the field, aiming, in this case, the detection of the ball. The reason is that the ball which was previously orange, is now going to have an arbitrary color, bringing algorithm complications in the Colour processing sub-subsystem, which in turn can be overpassed with the adoption of this vision method.

As we can verify in figure 3.20, the first step of the Morphological Processing Sub-System is the Edge Detection. In this step, well defined contours must be determined being at the same time tolerant to the motion blur introduced by the movement of the ball and robots. This step must be also efficient, accurate, fast to calculate, and absent of noise as much as possible so it doesn't compromise the efficiency of the whole sub-system.

Chapter 4

FPGA Circuit Design and Development

4.1 Introduction

The development of the image processor within the FPGA involves mainly two topics: FPGA algorithm design, and its communication via serial cable with the computer for test purposes. As we have already seen in the last chapter, Canny edge algorithm is composed by several phases of processing constituted by several calculations and computations each, which means that we'll have different modules implemented on Virtex-5. In this chapter,we will discuss the adopted practices to implement the whole project, divided in the following topics:

- Buffer design for serial communication and state diagram;
- Distributed algorithm implementation in the FPGA – pipeline and parallel approach;
- Approached strategy with arithmetic operations;
- Individual modules and each corresponding timing diagram analysis;
- Final results and statistical values of the FPGA logic;
- Design optimization techniques.

From the beginning of the circuit design, we should always have in mind the main purpose of this study: analyze the processing power of an FPGA when processing video frames. Hence, it is important to think about the design always considering the maximization of its processing power through a proper architecture. So let's think about one of the three primary physical

characteristics of a digital design, the speed. Depending on the context of a problem, speed contemplates three primary definitions:

- Throughput – amount of data that is processed per clock cycle (bits/second);
- Latency – time between data input and processed data output (clock cycles);
- Timing – logic delays between sequential elements (frequency).

This last speed characteristic, timing, will be the most important in this project, since it will determinate the maximum operating frequency of the design. A term often used is that "design doesn't meet timing", i.e., when the critical path, which is the largest delay between FFs, is greater than the target clock period. Timing optimizations will be discussed in further subjects, approaching solutions to reduce the critical path.

Note: For a better comprehension, from this point on signals will be written in *italic*, ports in **bold**, logical states underlined, and modules of the implementation between "quotation marks".

4.2 Serial Communication

The initial task of this project was to develop a module which was able to firstly receive a frame through a UART already designed, store it in a buffer, and send it back to the computer, for test purposes. This module called "RS232_COM" is constituted by an UART, a FIFO BlockRAM based memory buffer, and an address controller as it is illustrated in figure 4.1. When the UART receives a pixel by the serial comunication, "Address Controller"

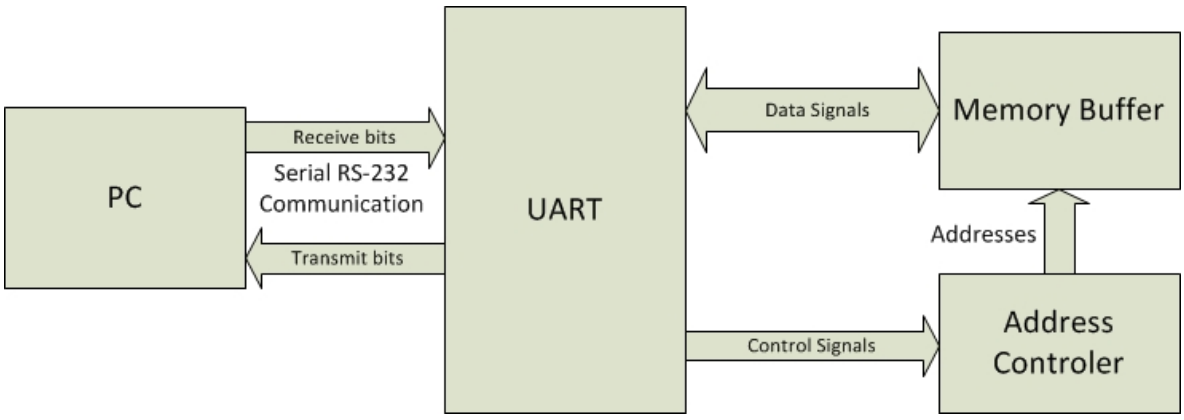


Figure 4.1: Serial Communication algorithm

is responsible to address this pixel to the right position in the memory buffer and increment

its address after the pixel is redirected from the UART directly to the memory buffer. This way, the next pixel will be stored in the next stable address of the memory. Whenever the UART is ready to transmit, "Address Controler" is also responsible for addressing the right pixels to be read in the buffer, based on a FIFO principle.

To control the addresses, the "Address Controler" module is based in a state diagram constituted by for 4 states:

- IDLE: this is the waiting state from where all the processes start, i.e., from here start both the writing processing and the reading process in the memory;
- WR_MEM: in this state the write enable signal of the memory buffer is activated so the pixel can be stored;
- RD_MEM: when the UART is ready to transmit, the "Address Counter" passes from IDLE to RD_MEM, connecting the UART address to the correct read address;
- UART_WRITE: After the read address is stable, the UART buffer loads the 8 bits pixel to the output port;

Figure 4.2 presents this state diagram. After observing it we can verify that the signals *rx_en* and *tx_en* are responsible for the transition from IDLE to WR_MEM and RD_MEM, respectively. The signal *rx_en* is high when the UART is ready to receive and hence to store a pixel in the memory buffer, and *tx_en* is high when the UART is ready to transmit a pixel and hence to read it from the memory buffer. The signal *wren* is responsible for activating the incrementing of the write address, while *rden* activates the incrementing of the read address. *uartwrite* is responsible for loading the pixel that is in the output port of the memory buffer to the buffer of the UART. As we can observe, in IDLE all these signals are low. However, when the address counter shifts to WR_MEM, *wren* enables the writing mode in the memory buffer while it also increments the writing address. In the case of the RD_MEM, *rden* activates the incrementing of the read address and in the following state, UART_WRITE, *uartwrite* is high to send the pixel bit-by-bit.

To better analyse how the whole process works, let's look at its timing diagram presented in figure 4.3. If we take a closer look, we can see that when *rx_en* is high, the input pixel (**bytein**) is available in the input port of the BlockRAM (**dina**) and after half a clock delay, *we_en* is high, enabling the writing on the block RAM, as we can verify in its write enable port (**wea**). Immediately after the writing, the write address (*s_wraddr*) on the "Address Counter" increments, while at the same time *tx_en* turns high, which means that the UART

is ready to transmit. The read address (*s_rdaddr*) on the "Address Controller" increments, respecting the FIFO principle, and in the next clock *uartwrite* turns high, loading the pixel to the UART buffer so it can be transmitted through serial communication.

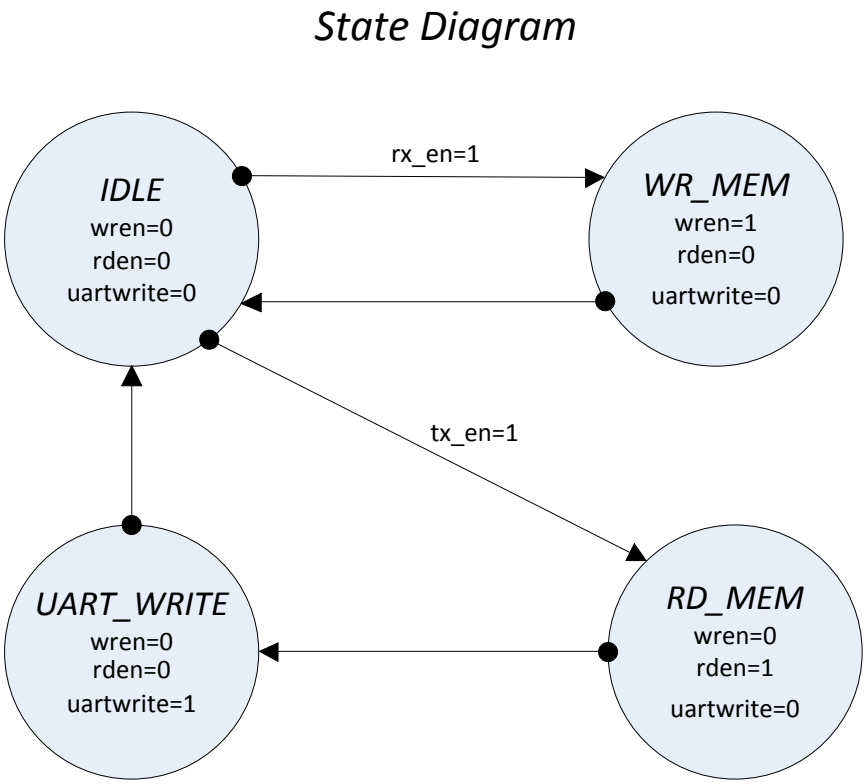


Figure 4.2: Serial Communication State Diagram

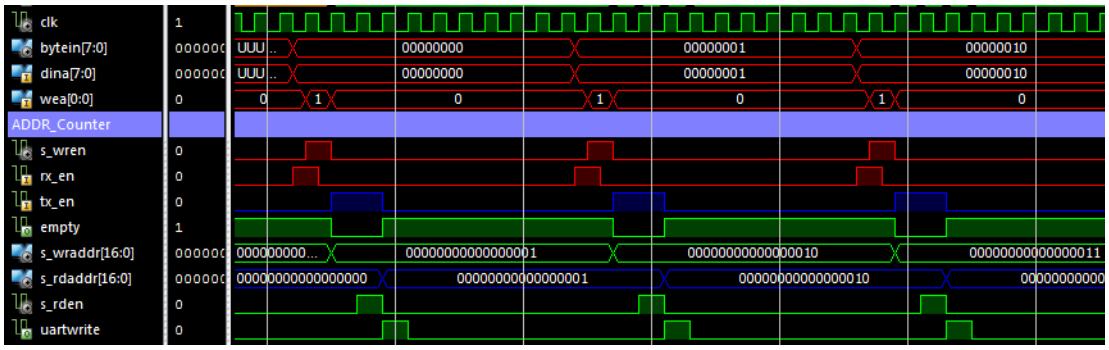


Figure 4.3: Serial communication timing diagram

4.3 Canny's Circuit Design

Recalling Canny edge algorithm theory, its backbone is constituted by 5 main steps: The Smoothing Filter, Finding gradients, Non-Maximum Suppression, Double thresholding, and Edge Tracking. We are in fact in the presence of a multi-stage algorithm that executes several calculations over a frame and hence implying heavy computations. The main reason of selecting the FPGA technology to implement this algorithm is to take advantage of its capability to process information in parallel, aiming the maximization of processing speed to process video on-the-fly. For this to be feasible, this algorithm was designed with a pipeline architecture approach. *But what is in fact a pipeline architecture?*

In the world of digital design where digital data is processed, we use the abstract term "pipeline" to define what is conceptually similar to an assembly line of an automobile manufacturer. In the latter, raw material or data input enters in the front end line, passes through various stages of processing and manipulation, and in the final exits as a finished product or data output. This is in fact what is happening in this algorithm. Due to this pipeline technique, which is used in early all very-high-performance devices [1], new pixels can begin processing before the prior pixel has finished.

To better understand this concept, let's look at a simple software implementation example and the corresponding iterative and pipelined implementations. Below is the iterative algorithm [36] for the calculation of the power of 3 of a number X, which will be executed on a microprocessor (consider a single core processor).

XPower = 1; for (i=0; i < 3; i++) XPower = X * XPower;

In this case, the same variables and addresses are accessed until the computation is complete, and in fact, a parallel approach is not feasible due to the fact that a microprocessor only executes one instruction at a time. Figure 4.4 [36] illustrates a possible hardware implementation of this algorithm.

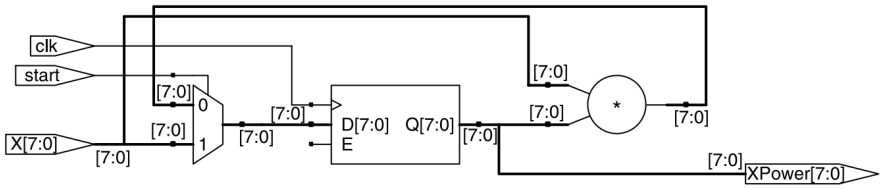


Figure 4.4: Iterative Implementation

In this iterative implementation, a new computation only begins after the previous one is completed, achieving a throughput of 8bits per each 3 clocks, which results in an average of 2.7 bits/clock, while the timing factor is one multiplier delay in the critical path. Let's now look to the pipeline implementation proposed in figure 4.5 [36].

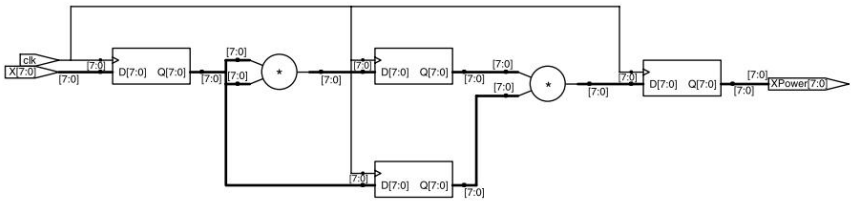


Figure 4.5: Pipelined Implementation

As we can observe in this implementation, while the final power of 3 of the X value is being calculated in the second pipeline stage, a new value of X can be sent to the first pipeline stage. In fact, both the final calculation of X^2 and the calculation of X^3 of the next value occur simultaneously, since the multiply operations are computed by independent resources. This results in a throughput of 8bits/clock, which is a performance 3 times faster than the iterative implementation. Also timing factor didn't change, since it is still equal to one multiplier delay in the critical path.

In the case of this design, in each clock different pixels are under different computations, similarly to an assembly line of an automobile manufacturer, and consequently with a throughput of one entirely processed pixel/clock. Let's now analyse each of these steps with more detail.

4.3.1 Smoothing Filter

Smoothing is the process to capture important patterns and eliminate noise from an image. In practice, a frame with size RxC pixels is convolved with a Gaussian function, i.e. a 5x5 Gaussian smoothing operator, resulting in a blurred image reduced in 2 dimensions, with (R-4)x(C-4) pixels, due to reasons that will be discussed in the next lines. The main "challenge" in the design of this step was the necessity to find a way of multiplying each pixel by the 25 elements of the Gaussian matrix, in order not to compromise the processing speed. The adopted solution to improve timing performance, was hence an optimization technique based on parallel structures (more information about optimization techniques in

chapter 4.4.1), which in this case was the instantiation in parallel of 5 BlockRAMs. Beyond the selection of this architecture is the following idea: make a 5x5 moving window (Gaussian matrix) that will process the whole image, 25 pixels at a time, as it is illustrated in figure 4.6. This process basically changes the value of the pixel P(2,2) of the operating region (central pixel), by convolving the elements of the Gaussian matrix with the whole 25 pixels, resulting in a blurred pixel (more details about this function in chapter 3.3.2.1).

To maximize the design with the maximum processing speed, each of the 5 BlockRAMs sends one pixel/clock to the Gaussian 5x5 Mask. Initially, the same pixels of the original frame are stored in all the 5 memory buffers (figure 4.7). When the first 5 rows of the frame are completely stored in the buffers, each of the buffers starts sending one pixel/clock from the same column but separated by one row, i.e., buffer 1 sends pixels from the 1st row, buffer 2 sends pixels from the 2nd row, and so on. The result is illustrated in figure 4.8, representing the effect of the moving window over the frame. As it is illustrated, in each clock new pixels enter in the last column of the 5x5 Gaussian Mask, where they will be shifted left one column per clock. After the initial 5 rows are sent, buffer1 starts sending the 1st pixel of the 2nd row, buffer2 send the 1st pixel of the 3rd row etc, creating the effect of the moving window.

Using an iterative implementation, the function would evaluate pixels through a serial string of logic, i.e., it would have to "wait" for 5 clocks to get 5 new pixels so it could produce an output. However, with this optimization technique, this serial string of logic can be broken up and evaluated in parallel. In just one clock, it is provided the output of a smoothed pixel, avoiding more complex calculations and consequently the needing for more clocks per output.

However, in terms of results, the loss of two dimensions in the frame is due to the adopted approach for convolving the Gaussian Mask with the frame. As we can conclude from the discussion of this step, and observing figure 4.6, we see that the first element of the frame to be processed is the element P(2,2) (in light blue), i.e., the 3rd element of the 3rd row of the original frame. For this function to be calculated, there must be 24 surrounding pixels available which wouldn't happen if we would try, for example, to process the first 2 elements of the first 2 rows of the frame. The same happens in the other three corners of the image for the same reasons. In fact, this will happen in all the steps of the algorithm, although the frame will only loose 1 dimension / step in the next stages, resulting in a final edge frame with size (R-10)x(C-10) pixels.

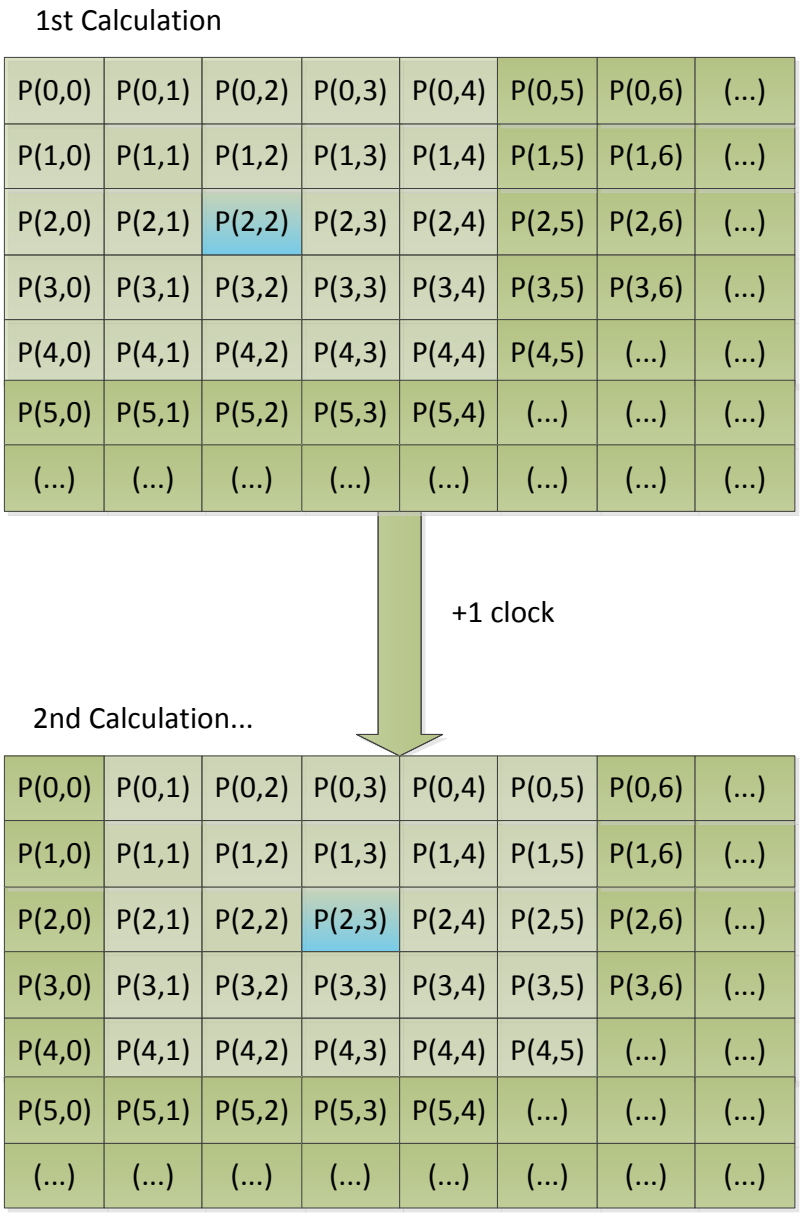


Figure 4.6: First two calculations of the moving window (in light green) over the original frame. Pixels in light blue, P(2,2) and P(2,3), are the pixels under calculation in each clock

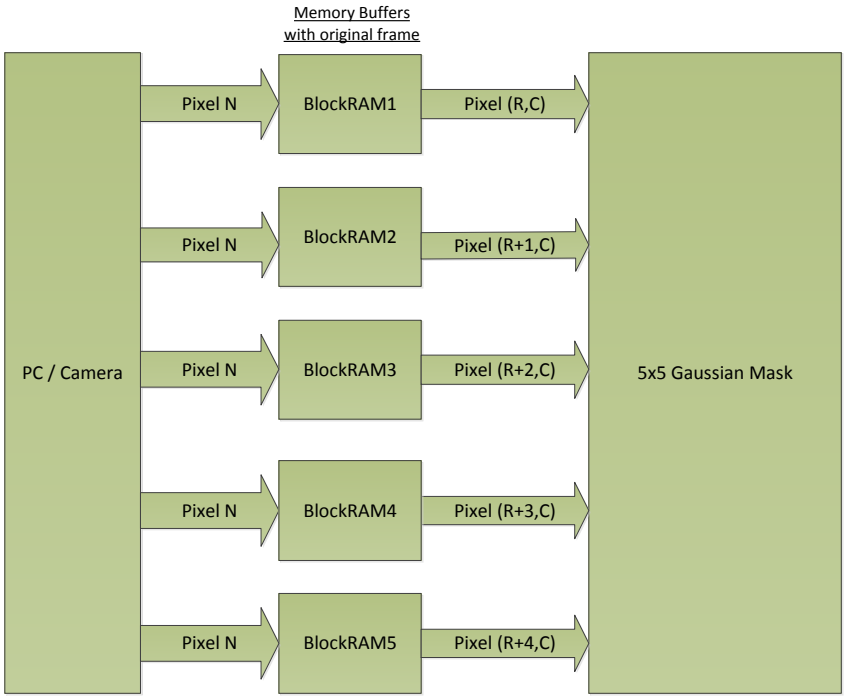


Figure 4.7: Algorithm of the Smoothing Filter step

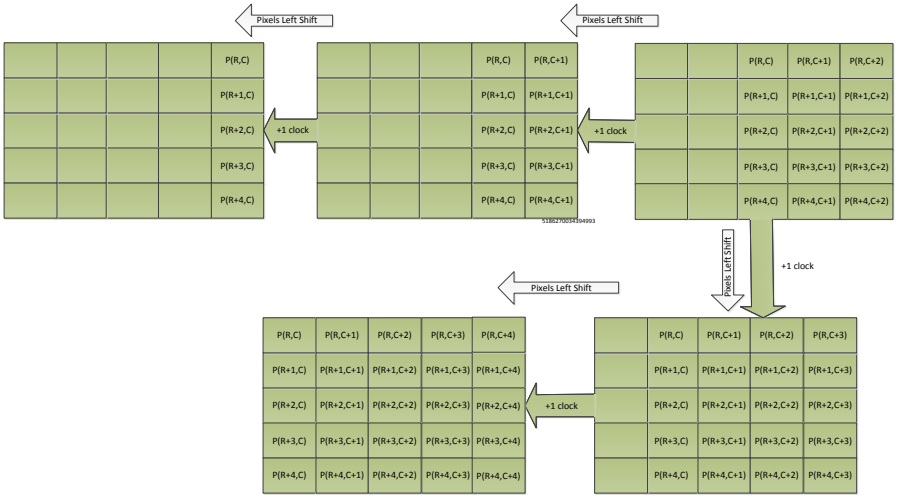


Figure 4.8: Mask shift process in the Gaussian Operator. P = pixel, R = row and C = column

Let's now look to the timing diagram of some of the involved signals in this step. Observing figure 4.9, we can see how the output of the memory buffers with the original frame connect directly to the last column of the 5x5 Mask (signals in blue). Also is it clear the shift left in

each clock of the values within the mask, and the direct connection between the output of the mask and all the 3 memory buffers that will store the smoothed image.

Figure 4.9: Smoothing Filter Timing Diagram

4.3.2 Finding Gradients

This step of the algorithm aims to calculate the gradients of the image through the convolution of two 3x3 matrices with the smoothed frame output by the previous step. In fact, this step is implemented in a very similar way to the Gaussian Smoothing step, in order to improve the processing speed, taking advantage of multiple acpBRAM to send pixels in parallel to the "Gradient Processor" module. Similarly to the first step, the Gradient Processor is based on a moving window principle that will calculate the gradient of the pixels in the x and y-direction each, operating over the local region of the 8 neighbouring pixels. Let's look at the implementation with more detail. Smoothing filter, as we can verify in figure 4.10, sends the same pixels to all the 3 BRAMs. Based on the same principle of the first stage, each memory output pixels in the same columns but separated by one row. The scheme in figure 4.11 represents the effect of moving an entire pixel window through the memory. Looking at it, we can verify that these pixels enter in the last column (column 3) of the 3x3 "Gradient Processor" in each clock, and will be shifted left one column per clock. In each clock, both x and y-direction mask operators will be applied to the pixels since all the pixels in the moving window operator must be accessed at the same time for every clock. These gradient calculations introduce negative signed values represented as 2's complement, with 9bits each.

In figure 4.12 is presented a scheme of the 3x3 moving window with the signals that correspond to each element. Looking at the structure of the multipliers mask module (Gradient 3x3 Mask) in figure 4.13, we can see it has the 9 inputs of the 3x3 mask but, in fact, the

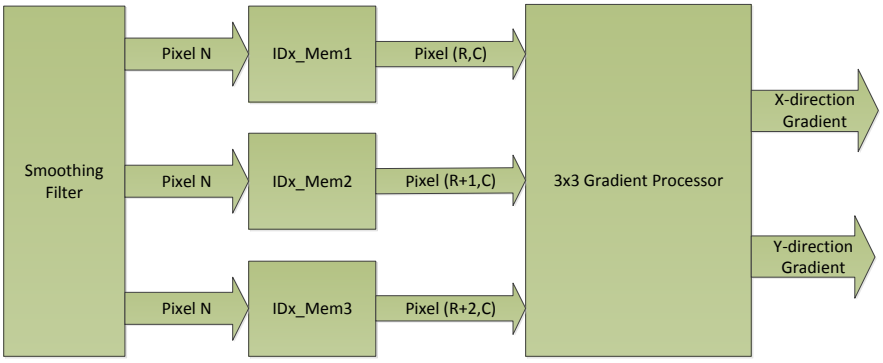


Figure 4.10: Finding Gradients algorithm

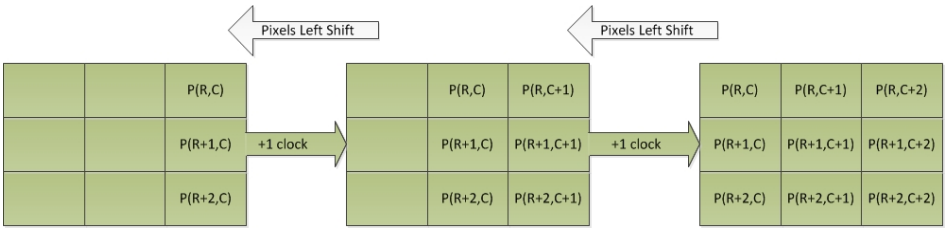


Figure 4.11: Mask shift process in Gradient 3x3 Mask. P = pixel, R = row and C = Column

signals which carry in the pixels output by the distributed memory, are signals *idx02*, *idx12* and *idx22*, while the others just left shift themselves, as already illustrated in figure 4.11. X and y-direction gradients are output through signals *idxx* and *idyy* respectively and they are summed resulting in the total gradient value.

Idx00	*Idx01*	*Idx02*
Idx10	*Idx11*	*Idx12*
Idx20	*Idx21*	*Idx22*

Figure 4.12: 3x3 Moving Window with corresponding signals

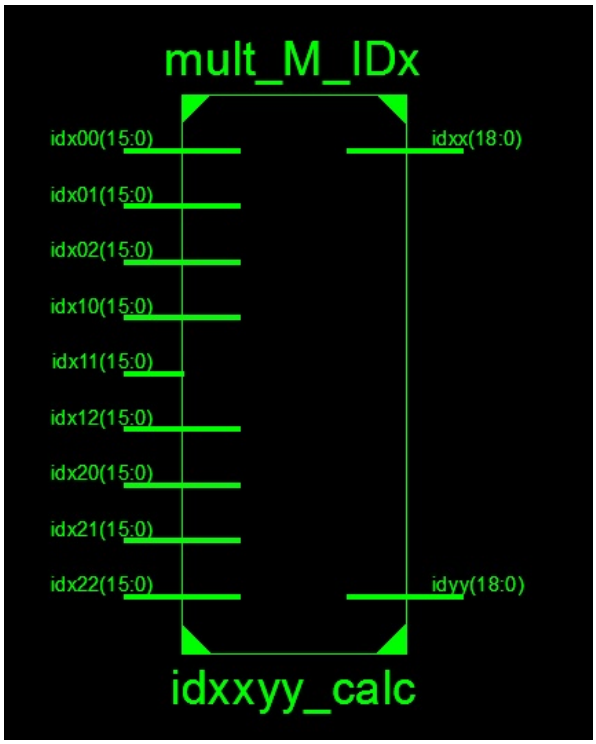


Figure 4.13: Mask multipliers structure

Let's now analyse the timing diagram of this step to understand the behaviour of its signals:

Figure 4.14: Finding Gradients Step Diagram

After 3 complete lines of pixels are stored in the distributed memory, i.e., in all three "IDX_Mem", the signal responsible to start the gradient calculation process, *startidxxyy*, turns high, and the output addresses (ports **MEM1_doutb**, **MEM2_doutb** and **MEM3_doutb**

in figure 4.14) of these memory buffers start incrementing. Observe that while the address in port **MEM1_addrb** starts in 0, in **MEM2_addr** starts in the first element of the 2nd row of the frame, pixel number 512 (in binary, 001000000000), while in **MEM3_addrb** starts in the next row, pixel number 1024 (010000000000 in binary).

In figure 4.15, we can see more clearly the output per clock of the x and y-direction gradients, while the connection between the output data of the memory buffers and the signals connected to the last column of the moving windows, *idx02*, *idx12* and *idx22*, is verified. It is also noticeable the pixels left shift per clock between *idx00* and *idx01*, *idx10* and *idx11*, and between *idx20* and *idx21*.

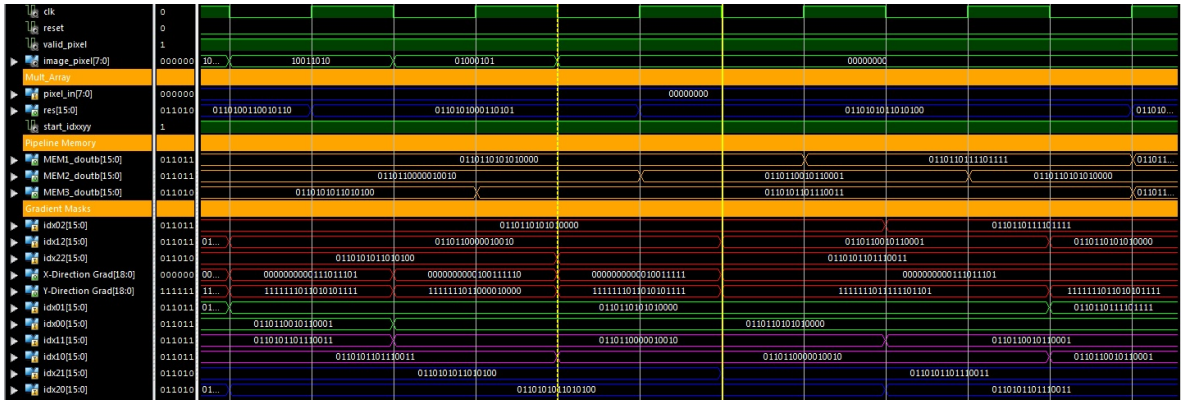


Figure 4.15: "Finding Gradients" step timing diagram 2

4.3.3 Non-Maximum Suppression

At this stage of the algorithm, gradients of the frame are already calculated and so, we need now to calculate its orientation and suppress the non-interesting pixels, as already explained before. Actually, in this step, parallel approach will be again adopted to speed up the process of gradient comparison between pixels, as it is illustrated in figure 4.16. After observing the algorithm, we can see that Gradient values are stored in 3 BRAMs in parallel, so they can be later sent to the non-maximum suppression module, similarly to the previous step where shifting masks were used. At the same time gradient pixels are stored in the BRAMs, its orientation is also calculated and stored in "Theta_RAM", to be used further in the non-maximum suppression module.

For a deeper analysis of this step, let's look to its timing diagrams.

In figure 4.17 we can see that as soon 3 pixels are outputted from the Gradient 3x3 Mask (blue signal), writing on the Gradient BRAMs is enabled (green signal), and input

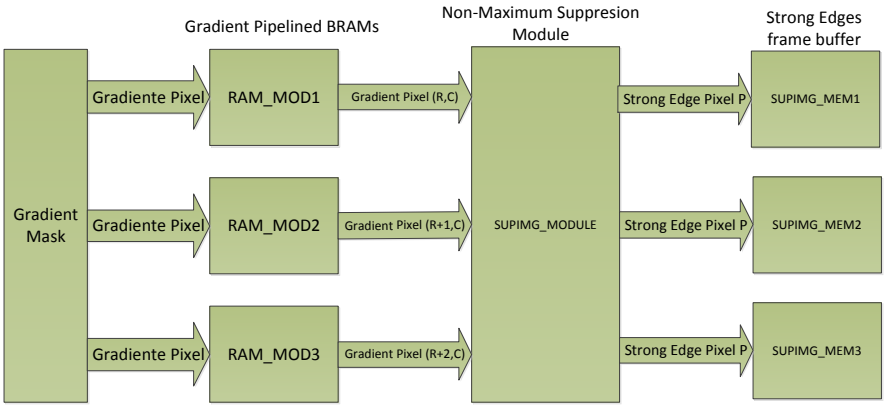


Figure 4.16: Non-Maximum suppression algorithm

addresses of the pipelined RAMs start incrementing (red signal). Since the division ipCore

Figure 4.17: Non-Maximum suppression (memory) timing diagram

has an output latency of 39 clock cycles till the first output, in figure 4.18 we can see that the determination of the theta angles (theta labeling - 0, 45, 90 or 135°) and its storage in the "THETA_RAM", starts when the input address of the pipelined RAM number 1 is equal to 38, i.e., *addra*=000000100110. To avoid calculating the *arctan* of the division, it was used a "frontier" method. Knowing the threshold angle values which will label the gradient orientation into 1 of the four directions mentioned before, it was previously calculated their corresponding tangent value, and then compared with the output of the ipCore (i.e., the division result) so the angles are categorized without the need for arithmetic operations.

Figure 4.18: Non-Maximum suppression (memory) timing diagram

When the storage of 3 complete lines of gradients in the pipelined BRAMs ("RAM_MOD1", 2 and 3) is done, gradients start being sent to the "supimg_module" (figure 4.19) through *mod_rd_data* signals, where they are object of the same procedure as in the 2nd step of the algorithm, i.e., while 3 new gradient pixels (separated by one row) are carried in the module

in each clock, the 6 gradient pixels which entered in the previous two clocks are shifted left. This way, pixels can be compared to the surrounding neighbours to keep the strongest ones. In the timing diagram of figure 4.20, we can notice all this process with more precision.

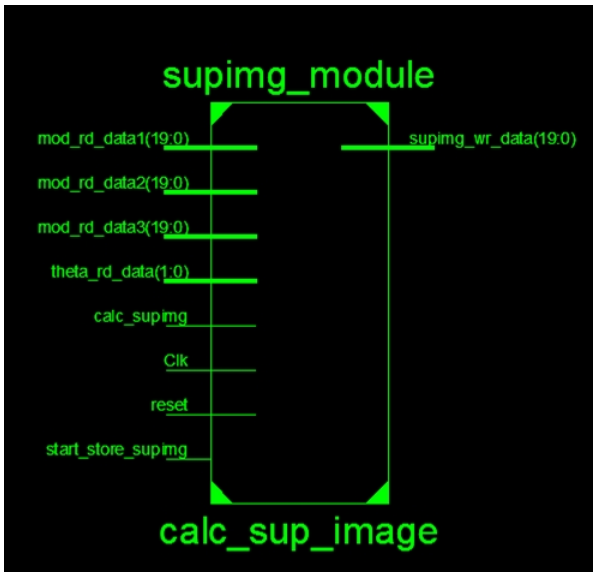


Figure 4.19: "Supimg_module" structure

Figure 4.20: Non-Maximum suppression timing diagram

After analysing the previous figure, we can see that the outputs of the pipelined RAMs (in green) are directly connected to the 3 input signals in the "supimg_module" (in red), and we can also observe the shifting process between signals *mod00* and *mod01*, *mod10* and *mod11*, and between *mod20* and *mod21*.

When "supimg_module" has already 9 pixels available to execute the intensity comparison, *start_store_supimg* turns high (figure 4.21), the output frame from the non-maximum suppression starts being stored in a buffer and the input address (*addra*) of the "SUPIMG_MEM"

starts incrementing.

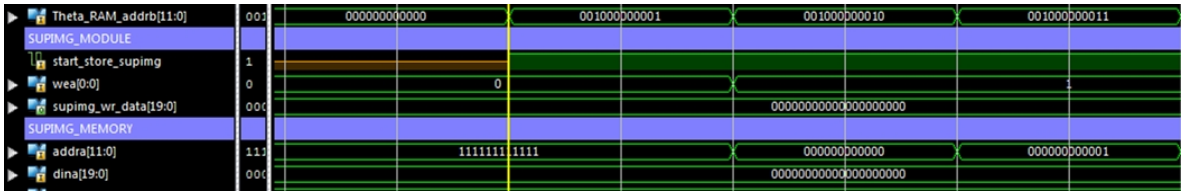


Figure 4.21: Non-Maximum suppression timing diagram

4.3.4 Double Thresholding and Edge Linking

The output obtained from the previous stage contains single edge pixels which contribute to noise. So, in this step we obtain the final frame using two threshold levels only considering the true edges of an image. Basically, pixels which are considered for the final image are:

- Pixels with gradient values over the high threshold level - strong edges;
- Pixels with gradient values between the low and high threshold levels (weak edges) that are directly connected to strong edges.

The weak edges that are not connected to strong edges, or pixels with gradient values under the low threshold level, are unconditionally set to zero.

At an implementation level, while the selection of strong edges is linear, the evaluation of weak edges that are connected to strong edges has hardware implications due to the fact that they must be compared with the surrounding 8 neighbouring pixels, in every single clock. Hence, the design of this logic was, similarly to all the other stages of the algorithm, based on a parallel architecture to provide a throughput of one final pixel/clock. Again, we have 3 BRAMs instantiated in parallel that have stored the pixels output by the previous step. Each of these buffers send the pixels separated by one row to the "Edge Linking" module (figure 4.22), where they will be compared so the double threshold evaluation and edge linking are accomplished. In practice, we have again a 3x3 moving window that will run all over the entire frame resultant from the previous step, and compare the pixels with its neighbours.

Analysing some signals of this step in the timing diagram of figure 4.26, we can see the linking between the pixels that are output by the previous stage buffers (ports in red) and the third column of the 3x3 edge linking mask (signals in blue). The output of this step is the final frame with the detected edges of the image.

In figure 4.23 it is represented an original image obtained from the CAMBADA omnidirectional cam, and in figure 4.24 is the corresponding picture processed by Canny Edge Detector. As we can see, both footballs (delimited with red circles) and the white lines of the field are pretty clear. In figure 4.25 it is presented the complete algorithm of the Canny Edge Detector.

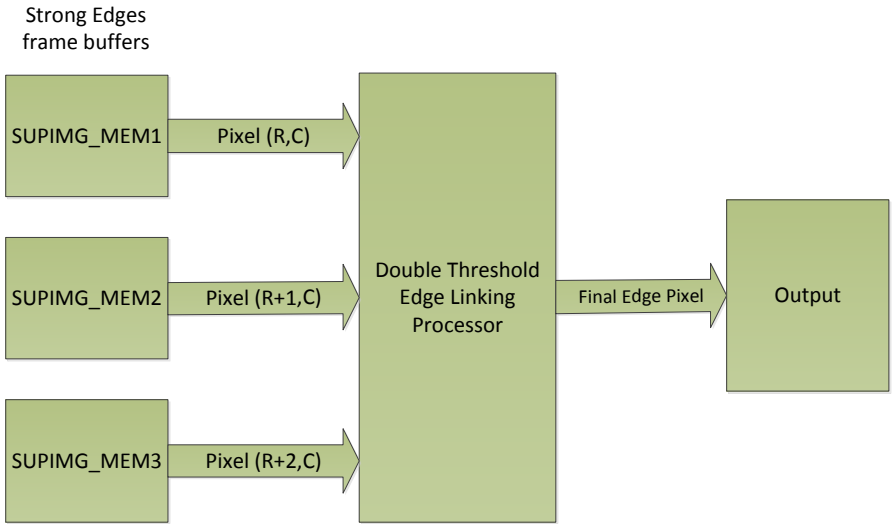


Figure 4.22: Algorithm of the Edge Linking with double threshold stage

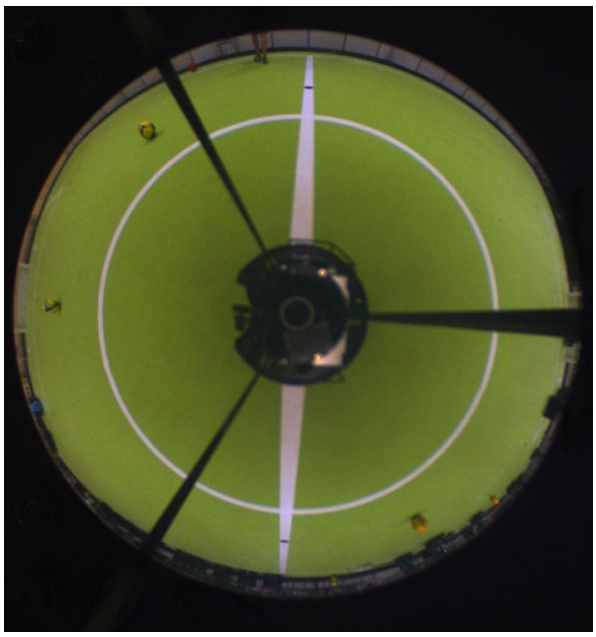


Figure 4.23: Image obtained from CAMBADA omnidirectional camera

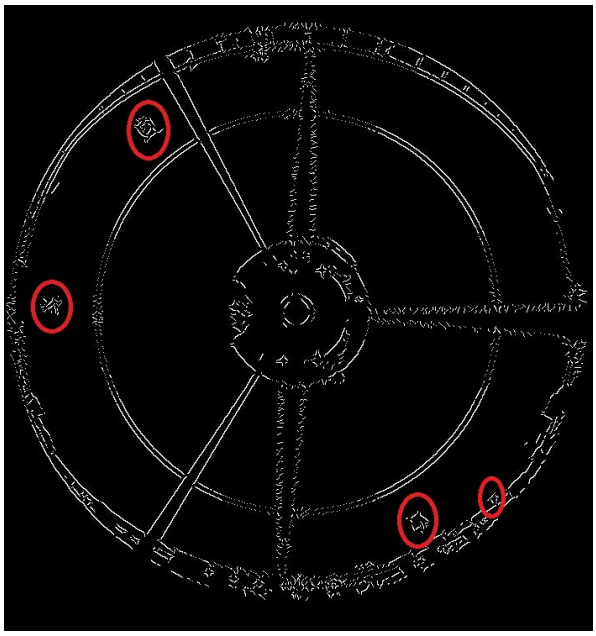


Figure 4.24: Previous image processed by the designed Canny Edge Algorithm

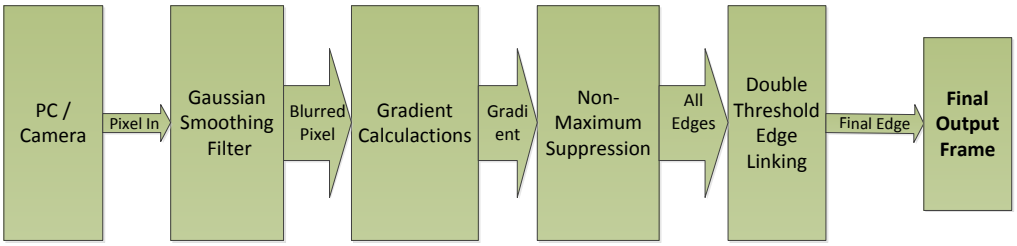


Figure 4.25: Canny Edge complete algorithm, step-by-step

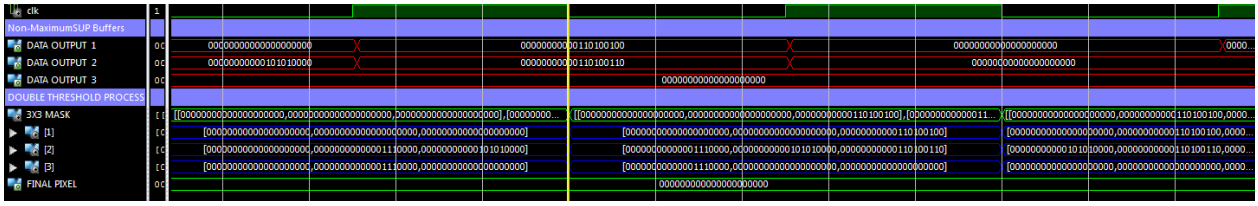


Figure 4.26: Edge Hysteresis Processor timing diagram

4.4 FPGA Statistics and Design optimization techniques

The proposed Canny Edge Detector algorithm is implemented on a XILINX Virtex 5-XC5VLX50T FPGA chip. ISE Project Navigator 13.4 was used to synthesize VHDL modeling code and a 708x752 gray-scale image was used as a test-bench image for evaluating the proposed processor. The maximum frequency achieved is equal to 30.412 MHz (minimum period of 32.882 ns), so it can process around 30 million pixels per second, which for a 708*752 video frame represents a processing power of approximately 57fps (frames per second). The logic resource utilization for the proposed design is illustrated in table of figure 4.27. There

Device Utilization Summary (estimated values)			FPGA Logic
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	3390	28800	11%
Number of Slice LUTs	2153	28800	7%
Number of fully used LUT-FF pairs	1102	4441	24%
Number of bonded IOBs	34	480	7%
Number of Block RAM/FIFO	29	60	48%
Number of BUFG/BUFGCTRLs	1	32	3%
Number of DSP48Es	25	48	52%

Figure 4.27: FPGA Resources Utilization Statistics

are indeed techniques to improve resource utilization, and others to improve processing speed. Within the scope of this project, speed is the main target of the design and hence, we will below discuss some of the possible techniques to improve this physical characteristic.

4.4.1 Optimization Techniques

Since processing speed is the main issue of the project, we should be looking to improve its timing, i.e., its clock speed. As already mentioned in previous topics, the maximum delay between any two sequential elements in a design will determine its maximum clock speed. Theoretically, the maximum working speed/frequency is determined by equation (4.1)

$$F_{max} = \frac{1}{T_{clk-q} + T_{logic} + T_{routing} + T_{setup} - T_{skew}} \tag{4.1}$$

where F_{max} is the maximum allowable frequency for clock, T_{clk-q} is the time from clock arrival until data arrives, T_{logic} is the propagation delay through logic between FFs, $T_{routing}$ is the

routing delay between FFs, T_{setup} is the minimum time data arrive at D before the next rising edge of clock, and T_{setup} is the propagation delay between the launch FF and the capture FF. This topic aims to give the reader a brief overview of the possible techniques that may be used over a critical path of a design to improve timing performance. However, for a deeper understanding of these methods, the reading of the book [36] is advised.

Methods

The first technique we'll discuss is the ***adding register layers*** technique which is adding intermediate layers of registers to the critical path. It is appropriated to highly pipelined designs where it is not a major problem to have and additional clock cycle latency, and it mustn't affect the overall functionality. Considering registers *X1*, *X2* and Y and signals A, B, C and X, let's look to the following example:

$$
\begin{aligned}
&process(clk)\\
&\quad begin\\
&\qquad X1 <= X;\\
&\qquad X2 <= X1;\\
&\qquad Y <= A * X + B * X1 + C * X2;\\
&end\, process;
\end{aligned}
$$

Architecturally, all multiply/add operations occur in one clock cycle. If the critical path of one multiplier and one adder is greater than the minimum clock period requirement, we can initially add a pipeline layer between the multipliers and the adder, reducing the critical path, as in the following example (consider now registers *prod1*, *prod2* and *prod3*):

```
process(clk)
    begin
        if valid ='1' then
            X1 <= X;
            X2 <= X1;
            prod1 <= A * X;
            prod2 <= B * X1;
            prod3 <= C * X2;
            Y <= prod1 + prod2 + prod3;
        end if;
end process;
```

This way, the adder is separated from the multiplier by a pipeline stage of registers, dividing the critical path into two paths of smaller delay. As we can see, multipliers are good candidates for pipelining because the calculations can be easily broken up into different stages. We could continue optimizing the design by breaking the multipliers and adders up into stages that can be individually registered.

Adding ***parallel structures*** to the design, was a method adopted in the development of this work to evaluate information that would normally be evaluated through a serial string of logic. In fact, the use of several BRAMs in the implementation of Canny Edge algorithm is a direct application of this technique, and it is discussed in detail in previous sections.

Related to this technique, ***flatten logic structures*** is a similar method but applies specifically to logic that is chained due to priority encoding. Consists of manually breaking up logic structures that are coded in a serial fashion, since typically synthesis and layout tools don't have enough information relating to the priority requirements of the design, and consequently aren't smart enough to do it [36]. Let's consider the following example of an

address decode where a 4-bit control signal ($ctrl$) controls the writing of four registers ($rout$):

$$
\begin{aligned}
&process(clk)\\
&\quad begin\\
&\qquad if\ ctrl(0) <= 1\ then\ rout(0) <= in;\\
&\qquad elsif\ ctrl(1) <= 1\ then\ rout(1) <= in;\\
&\qquad elsif\ ctrl(2) <= 1\ then\ rout(2) <= in;\\
&\qquad elsif\ ctrl(3) <= 1\ then\ rout(3) <= in;\\
&end\ process;
\end{aligned}
$$

In this example, the control signals are coded based on a priority principle but, in fact, this priority is not vital to its function. Hence, to flatten the logic structure and reduce the path delay, we can remove the priority encoding in the following way:

$$
\begin{aligned}
&process(clk)\\
&\quad begin\\
&\qquad if\ ctrl(0) <= 1\ then\ rout(0) <= in;\\
&\qquad if\ ctrl(1) <= 1\ then\ rout(1) <= in;\\
&\qquad if\ ctrl(2) <= 1\ then\ rout(2) <= in;\\
&\qquad if\ ctrl(3) <= 1\ then\ rout(3) <= in;\\
&end\ process;
\end{aligned}
$$

With this approach, each of the control signals acts independently and controls its corresponding rout bits independently.

In another situation, whenever logic is highly imbalanced between the critical path and an adjacent path, ***register balancing*** is the best option of optimization. The idea is to improve timing by moving this logic from one path to the other, i.e., by redistributing logic evenly between registers to minimize the worst-case delay between any two registers. Imagine we have three registers, rA, rB and rC, three input signals A, B and C and one output signal

Sum. Consider the following VHDL code excerpt of an adder:

$$
\begin{aligned}
&process(clk)\\
&\quad begin\\
&\qquad rA <= A;\\
&\qquad rB <= B;\\
&\qquad rC <= C;\\
&\qquad Sum <= rA + rB + rC;\\
&\quad end\,if;\\
&end\,process;
\end{aligned}
$$

Register rA, rB and rC constitute the first register state, while the second register stage consists of the Sum. As we can see, the logic between stages 1 and 2 is the adder, whereas the logic between the input and the first register stage contains no logic. If the critical path of this implementation is defined through the adder, to a successful application of this technique in the above implementation, a possible solution would be to move the logic back a stage balancing the logic load between the two registers, as follows:

$$
\begin{aligned}
&process(clk)\\
&\quad begin\\
&\qquad rABSum <= A + B;\\
&\qquad rC <= C;\\
&\qquad Sum <= rAB + rC;\\
&end\,process;
\end{aligned}
$$

Here we have moved one of the add operations back one stage between the input and the first register stage, balancing the logic between the pipeline stages and reducing the critical path.

The last case we may face is whenever multiple paths combine with the critical path. In this case, we should adopt a strategy to ***reorder the combined path***, moving the critical path closer to the destination register. With this approach, we are only concerned with the logic paths between any given set of registers, aiming the minimization of the critical path and consequently the maximization of design's operating frequency.

Chapter 5

Conclusions and Future Work

To study the capabilities of an FPGA for video processing purposes, the project was based on the analysis of an image edge detector called Canny Edge Detector, designed with a hardware description language, VHDL, for an FPGA implementation. Canny Edge Detector algorithm is adaptable to various environments and its parameters allow it to be tailored to recognize edges of different characteristics, depending on the particular requirements of a given implementation. In this work, the study was based on the real case of the robotic football team of the University of Aveiro, CAMBADA. Aiming a better performance of their hybrid vision system, namely its Morphological Processing Sub-system, an analysis was made to test and evaluate the functionality and processing speed capabilities of the Canny Edge algorithm integrated in a Virtex-5 device.

The main advantage of the FPGA technology for digital signal processing is its capability of allowing large-scale parallel processing and pipelining of data flow. Hence, a satisfactory hardware design was achieved through partitioning and pipelining of the different sub-algorithms which compose the Edge Detector, using multiple distributed BRAM buffers which provided significant improvements in performance, i.e., a throughput of 1 processed edge pixel/clock, with a maximum operating frequency of 30MHz.

The implementation of this design, as already mentioned, was developed giving priority to the speed characteristic, which despite of its effectiveness in power processing, it has also disadvantages in terms of area consumption. A good example of this issue is the parallel design which requires the use of more BRAMs, contributing to the increase of used logic.

5.1 Future Work

As a future work, there are some topics which can be studied to improve this design. Because the FPGA is essentially custom programmable hardware, is possible to trade off area utilization against speed power and also power consumption. For speed purposes, a key analysis in the design is the evaluation of the critical path. After the study of the discussed optimization techniques in chapter 4.4.1, a solution might be achieved by applying one or more of these methods over this critical path, so it improves the processing power. It is very important to have consciousness that these techniques have implications in terms of area utilization and power consumption, and so, depending on the design requirements, it is interesting to approach resources utilization techniques and/or power techniques, which were out of the scope of this thesis. However, one recommended improvement to do in terms of area consumption, is the reduction of resources consumption when using BRAMs, storing only one row of the frame per buffer instead of several rows.

In order to make processing system more functional, and regarding object recognition, it is also proposed the implementation of the Hough Transform, which based on the detected edges is able to recognize geometric forms in the digital image.

Bibliography

[1] Luk, W.: "Customizing processors: design-time and run-time opportunities", Lect. Notes Comput. Sci., 2004, 3133;

[2] Xilinx, (2013), "What is Programmable Logic"[online]. Available: http://www.xilinx.com/company/about/programmable.html. Last access on: 06/09/2013;

[3] V. Patki (2012, May 1st),"FPGA Basics and Architecture"[online]. Available: http://vedikapatki.wordpress.com/2012/05/01/fpga-basics-and-architecture/. Last access on: 08/08/2013;

[4] Nicolle J.P., (2009, September 30),"How FPGAs work". [online]. Available: http://www.fpga4fun.com/FPGAinfo2.html. Last access on: 28/12/2012;

[5] Stitt, G., Vahid, F., and Nematbakhsh, S.: 'Energy savings and speedups from partitioning critical software loops to hardware in embedded systems', ACM Trans. Embedded Comput. Syst., 2004, 3, (1), pp. 218–232;

[6] Vereen,L.:'Soft FPGA cores attract embedded developers', Embedded.Syst. Program., 2004, 23 April 2004, http://www.embedded.com//showArticle.jhtml?articleID=19200183;

[7] Altera Corp., Nios II Processor Reference Handbook, May 2004;

[8] Fidjeland, A., Luk, W., and Muggleton, S.: 'Scalable acceleration of inductive logic programs'. Proc. IEEE Int. Conf. on Field-Programmable Technology, 2002;

[9] Leong, P.H.W., and Leung, K.H.: 'A microcoded elliptic curve processor using FPGA technology', IEEE Trans. Very Large Scale Integr. (VLSI) Syst., 2002, 10, (5), pp. 550–559;

[10] Seng, S.P., Luk, W., and Cheung, P.Y.K.: 'Flexible instruction processors'. Proc. Int. Conf. on Compilers, Arch. and Syn. For Embedded Systems (ACM Press, 2000);

[11] Seng, S.P., Luk, W., and Cheung, P.Y.K. : Run-time adaptive ?exible instruction processors, Lect. Notes Comput. Sci., 2002, 2438;

[12] Neves AJR et al. CAMBADA soccer team: from robot architecture to multiagent coordination. Transverse Activity on Intelligent Robotics, IEETA / DETI, University of Aveiro;

[13] Xilinx. (2012, March 16th). "Virtex-5 FPGA User Guide(v5.4)" [online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug190.pdf. Last access on: 13/08/2013;

[14] Xilinx. (2009, February 6th). "Virtex-5 Family Overview(v5.0)" [online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf. Last access on: 13/08/2013;

[15] K. Parnell, N. Mehta. (2004, April). "Programmable Logic Design Quick Start Handbook"[online]. Available: http://uglyduck.ath.cx/PDF/Xilinx/Spartan3/BeginnnersBook-screen.pdf. Last access on: 06/10/2013;

[16] J. Serrano. "Introduction to FPGA design"[online]. Available: http://cds.cern.ch/record/1100537/files/p231.pdf. Last access on: 06/10/2013;

[17] Digilent (2013, September 5th). "Genesys Board Reference Manual"[online]. Available: http://www.digilentinc.com/Data/Products/GENESYS/Genesys_RM_VC.pdf. Last access on: 06/10/2013;

[18] Wikipedia. (2013, September 10th). "VHDL"[online]. Available: http://en.wikipedia.org/wiki/VHDL. Last access: 06/10/2013;

[19] VHDL for Logic Synthesis, Third Edition. Andrew Rushton. 2011 John Wil.ey and Sons, Ltd. Published 2011 by John Wiley and Sons, Ltd. ISBN: 978-0-470-68847-2;

[20] Xilinx, Inc., Microblaze Processor Reference Guide, June 2004;

[21] E.B. Scalvinoni, “Estado del arte de la tecnologia FPGA”, Instituto Nacional de Tecnologia Industrial, Proyecto Mejora de la Eficiencia y de la Competitividad de la Economia Argentina, October 2005;

[22] M. Petrou and C. Petrou,”Image Enhancement”, in Image Processing: The Fundamentals, 2nd ed. 2010 John Wiley and Sons, Ltd. ISBN: 978-0-470-74586-1;

[23] Wikipedia. (2013, June 5th). “Anisotropic Diffusion” [online]. Available: http://en.wikipedia.org/wiki/Anisotropic_diffusion. Last access on: 07/10/2013;

[24] M. Petrou and C. Petrou,”Image Segmentation and Edge Detection”, in Image Processing: The Fundamentals, 2nd ed. 2010 John Wiley and Sons, Ltd. ISBN: 978-0-470-74586-1;

[25] Marr, David, and Ellen Hildreth. ”Theory of edge detection.” Proceedings of the Royal Society of London. Series B. Biological Sciences 207.1167 (1980): 187-217;

[26] “Basic Filters” [online]. Available: http://www.swarthmore.edu/NatSci/mzucker1/e27/filter-slides.pdf. Last access on: 07/10/2013;

[27] M. Venkatesan and D. V. Rao, “Hardware acceleration of Edge Detection Algorithm on FPGAs”, University of Nevada Las Vegas, Las Vegas, NV 89154;

[28] “Canny Edge Detection” [online]. Available: http://homepage.cs.uiowa.edu/ cwyman/classes/spring08-22C251/homework/canny.pdf. Last access on: 07/10/2013;

[29] Wikipedia. (20th January,2013). “Canny Edge Detector” [online] Available: http://en.wikipedia.org/wiki/Canny_edge_detector. Last access on: 07/10/2013;

[30] L. Spirkovska, A summary of image segmentation techniques, NASA Technical Memorandum 104022, 1993;

[31] P.G. van Dokkum, Cosmic-ray rejection by Laplacian edge detection, The Publications of the Astronomical Society of the Pacific, vol.113, no.789, pp.1420-1427, 2001;

[32] Neves AJR et al. An efficient omnidirectional vision system for soccer robots: From calibration to object detection. Mechatronics (2010), doi:10.1016/j.mechatronics.2010.05.006;

[33] Wikipedia. (2013, September 26th). “Edge Detection” [online]. Available: http://en.wikipedia.org/wiki/Edge_detection. Last access on: 07/10/2013;

[34] John Canny. A computational approach to edge detection. Pattern Analysis and Machine Intelligence, IEEE Transactions on, PAMI-8(6):679-698, Nov. 1986;

[35] Thomas B. Moeslund. Image and Video Processing. August 2008;

[36] Advanced FPGA Design – Architecture, Implementation, and Optimization, John Wiley and Sons, Inc., Hoboken, New Jersey;

[37] Wikipedia.(2013, November 8th). "Reconfigurable Computing"[online]. Available: http://en.wikipedia.org/wiki/Reconfigurable_computing. Last access: 06/10/2013

[38] SourceTech411, (2013), "Top FPGA Companies for 2013"[online]. Available: http://www.fpgadeveloper.com/2011/07/list-and-comparison-of-fpga-companies.html. Last access: 06/09/2013