Universidade do Minho
Escola de Engenharia

Fábio Duarte Rodrigues Magalhães

Hardware Accelerated Real-Time Linux Video Anonymizer

janeiro de 2021

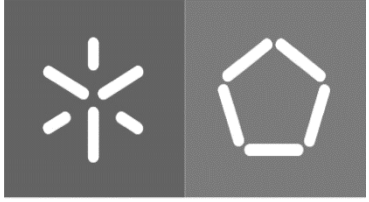
Hardware Accelerated Real-Time Linux Video Anonymizer
Fábio Magalhães
UMinho | 2021

Universidade do Minho
Escola de Engenharia

Fábio Duarte Rodrigues Magalhães

Hardware Accelerated Real-Time Linux Video Anonymizer

Guimarães, January 2021



Universidade do Minho
Escola de Engenharia

Fábio Duarte Rodrigues Magalhães

Hardware Accelerated Real-Time Linux Video Anonymizer

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Jorge Cabral

Guimarães, January 2021

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Acknowledgements

I would like to thank all the teachers and colleagues that contributed for my development as an engineer and as a man. But mostly I would like to thank my family for every sacrifice and compromise made, so that I could follow my dreams in the Engineering world.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Os Sistemas Embebidos estão presentes atualmente numa variada gama de equipamentos do quotidiano do ser humano. Desde *TV-boxes*, televisões, routers até ao indispensável telemóvel.

O Sistema Operativo *Linux*, com a sua filosofia de distribuição "*one-size-fits-all*" tornou-se uma alternativa viável, fornecendo um vasto suporte de *hardware*, técnicas de depuração, suporte dos protocolos de comunicação de rede, entre outros serviços, que se tornaram no conjunto *standard* de requisitos na maioria dos sistemas embebidos atuais.

Este sistema operativo torna-se apelativo pela sua filosofia *open-source* que disponibiliza ao utilizador um vasto conjunto de bibliotecas de *software* que possibilitam o desenvolvimento num determinado domínio com maior celeridade e facilidade de integração de *software* complexo.

Os algoritmos de *Machine Learning* são desenvolvidos para a automização de tarefas e estão presentes nas mais variadas tecnologias, desde o sistema de foco de imagem no *smartphone* até ao sistema de deteção dos limites de faixa de rodagem de um sistema de condução autónoma.

Estes são algoritmos que quando compilados para as plataformas de sistemas embebidos, resultam num esforço de processamento e de consumo de recursos, como o footprint de memória, que na maior parte dos casos supera em larga escala o conjunto de recursos disponíveis para a aplicação do sistema, sendo necessária a implementação de componentes que requerem maior poder de processamento através de elementos de *hardware* para garantir que as métricas temporais sejam satisfeitas.

Esta dissertação propõe-se, por isso, à criação de um sistema de anonimização de vídeo que adquire, processa e manipula as frames, com o intuito de garantir o anonimato, mesmo na transmissão.

A sua implementação inclui técnicas de Deteção de Objectos, fazendo uso da combinação das tecnologias de aceleração por hardware: paralelização e execução em hardware especializado. É proposta então uma implementação restringida tanto temporalmente como no consumo de recursos ao nível do *hardware* e *software*.

Palavras-Chave: Linux Embebido, Deteção de Objectos, Aceleração por Hardware, Machine Learning

Abstract

Embedded Systems are currently present in a wide range of everyday equipment. From TV-boxes, televisions and routers to the indispensable smartphone.

Linux Operating System, with its "one-size-fits-all" distribution philosophy, has become a viable alternative, providing extensive support for *hardware*, debugging techniques, network communication protocols, among other functionalities, which have become the standard set of requirements in most modern embedded systems.

This operating system is appealing due to its open-source philosophy, which provides the user with a vast set of *software* libraries that enable development in a given domain with greater speed and ease the integration of complex *software*.

Machine Learning algorithms are developed to execute tasks autonomously, i.e., without human supervision, and are present in the most varied technologies, from the image focus system on the smartphone to the detection system of the lane limits of an autonomous driving system.

These are algorithms that, when compiled for embedded systems platforms, require an effort to process and consume resources, such as the memory footprint, which in most cases far outweighs the set of resources available for the application of the system, requiring the implementation of components that need greater processing power through elements of *hardware* to ensure that the time metrics are satisfied.

This dissertation proposes the creation of a video anonymization system that acquires, processes, and manipulates the frames, in order to guarantee anonymity, even during the transmission.

Its implementation includes Object Detection techniques, making use of the combination of hardware acceleration technologies: parallelization and execution in specialized hardware. An implementation is then proposed, restricted both in time and in resource consumption at *hardware* and *software* levels.

keywords: Linux, Object Detection, Hardware Acceleration

Table of Contents

Resumo iii

Abstract iv

List of Listings xiv

Acronyms List xv

1 Introduction 1

1.1 Contextualization 1

1.2 Motivation and Goals 4

1.3 Contributions 4

1.4 Dissertation Structure 4

2 Theoretical Foundations 6

2.1 Embedded System 6

2.1.1 Generic Embedded System's Hardware 7

2.1.2 Storage 8

2.1.3 Real-Time Systems 8

2.2 Computer Architecture 10

2.2.1 Data-Level Parallelism 10

2.2.2 Instruction-Level Parallelism 11

2.2.3 Pipeline 12

2.2.4 Cache 13

2.3 Data Representation 14

2.3.1 Floating-Point Representation 14

2.3.2 Fixed-Point Representation 16

2.3.3 Quantization . . . 16
2.4 Linux and Embedded Systems . . . 18
2.4.1 Embedded Linux . . . 18
2.4.2 Linux and Memory Management . . . 20
2.4.3 Linux and Real-Time applications . . . 21
2.4.4 Latency . . . 21
2.4.5 Preemption . . . 22
2.4.6 Real-Time Kernel Patch . . . 23
2.5 Image Processing . . . 24
2.5.1 Image Decoding . . . 24
2.5.2 Color Space Conversion . . . 26
2.5.3 Image Resize . . . 26
2.5.4 Normalization . . . 29
2.5.5 Gaussian Function . . . 31
2.6 Machine Learning . . . 31
2.6.1 Feature . . . 32
2.6.2 Feature Descriptor . . . 32
2.6.3 Supervised Learning . . . 32
2.6.4 Unsupervised Learning . . . 33
2.6.5 Computer Vision . . . 34
2.7 Histogram of Oriented Gradients . . . 34
2.7.1 Image Gradients . . . 35
2.8 Support Vectors Machine . . . 37
2.8.1 Support Vectors . . . 37
2.8.2 Linear Classification . . . 38
2.9 Hardware Acceleration . . . 39
2.9.1 Hardware Descriptor Languages . . . 39
2.9.2 FPGA . . . 42
2.10 Hardware-Software Co-Design . . . 44
2.10.1 System Modeling . . . 44
2.10.2 Software Parallelization . . . 44

2.10.3 Profiling . . . 44
2.10.4 Hardware Design . . . 45
2.10.5 Validation . . . 45
2.10.6 Metrics Confirmation/Implementation . . . 45
2.11 Hardware Accelerated Embedded System Simulation . . . 46
2.11.1 Full System RTL Simulation . . . 46
2.11.2 RTL Simulation with Host Software . . . 47
2.11.3 RTL-Software Co-Simulation . . . 47
2.11.4 Full System Software Simulation . . . 48

3 System Design 49

3.1 Detection System . . . 49
3.2 HARVA System . . . 50
3.2.1 HOG . . . 54
3.2.2 SVM . . . 56
3.3 HOG Features Extraction . . . 58
3.3.1 Convolution Stage . . . 59
3.3.2 Gradient Calculation Stage . . . 60
3.3.3 Histogram Stage . . . 63
3.3.4 Normalization Stage . . . 64
3.3.5 Component Parallelism . . . 66
3.4 SVM Classification . . . 66
3.4.1 Quantization Stage . . . 68
3.4.2 Linear Combination Stage . . . 68
3.4.3 Component Parallelism . . . 69

4 Experimental Results 71

4.1 Test Environment . . . 71
4.1.1 Hardware . . . 73
4.1.2 Communication Interfaces . . . 74
4.1.3 Testbench . . . 75
4.2 Test Cases . . . 76

4.2.1 Block Diagram 76
4.2.2 HSYNC 77
4.2.3 Cache Miss 78
4.2.4 Detecting Window 79
4.3 Software Model Architecture 80
4.3.1 Preprocessing Thread 81
4.3.2 HOG Thread 82
4.3.3 SVM Thread 86
4.4 Results 87
4.4.1 Software Model Profiling 87
4.4.2 HOG Component 89
4.4.3 SVM Component 91
4.5 Detection Performance 92
4.5.1 False Detection 92
4.5.2 Detection Rate 93

5 Conclusion **94**
5.1 Developed Work 94
5.2 Future Work 95

References **102**

List of Figures

1.1 Embedded System's wide range of applications in modern technology. 2

1.2 Machine Learning several application areas in technology. 3

2.1 The Engine Control Unit, introduced in 1970, plays a fundamental role in ensuring the optimal performance of the automobile. 7

2.2 Composition of a Real-Time Embedded System. 10

2.3 Vector Architecture designed to support the processing of several blocks of data within the multiple Processing Element(PE) implemented. Vector Architectures provide an higher throughput at the expense of more hardware for concurrent processing. 11

2.4 Simple RISC pipeline designed for a five-cycle instruction executed. The processor performance is increased by five times, compared with a processor not pipelined. The pipeline is defined within the stages: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB =write-back. 12

2.5 Memory hierarchy in modern Computer Systems, targeting a more optimized data access through implementation of faster and smaller memories between the processor and the Main Memory. 13

2.6 Single Precision Floating Point representation, as defined by the IEEE Standard for Floating-Point Arithmetic. 15

2.7 Double Precision Floating Point representation as defined by the IEEE Standard for Floating-Point Arithmetic. 15

2.8 The graphic on the left side shows the voltage applied to and ADC, ranging from 0V to 10V. The graphic on the right side shows the the quantization results for the different possible values applied to the ADC. 18

2.9 Importance of connectivity in modern devices. 19

2.10 Layer hierarchy for device access in Linux OS. 20
2.11 Real-Time system response to a critical event. 22
2.12 Detailed Real-Time system's response to a critical event. 22
2.13 Image perception by a playback device. The synchronization signals dictate the active are of the image. 25
2.14 The graph (a) describes the disposition of negatively skewed data, with the median and mode placed on the right side of the mean value. Graph (b) describes data disposition without skew, which mean, mode and median values are the same. Graph (c) describes the disposition of positively skewed data, where the median and mode are placed on the left side of the mean value. 30
2.15 Machine Learning Classification problem. 33
2.16 Machine Learning Regression problem. 33
2.17 Machine Learning Clustering problem. 34
2.18 The Convolution is applied in the horizontal, (a), and vertical directions, (b) of the image. The target pixel, represented in yellow, is centered with the convolution kernel and the neighbors are subtracted, calculating the derivative. 36
2.19 The graph represents the distribution of two different classes of data, balls and stars. The blue points represent the support vectors, mapped the closest to the separating gap. 37
2.20 Block diagram of a general HDL module, representing the interaction between the Control and Data path. 40
2.21 Architecture of a UVM-based testbench for validation of HDL designs. 41
2.22 Basic configurable block present in FPGA fabric. 42
2.23 Zynq-7000 SoC's block diagram. 43
2.24 Full system RTL simulation diagram. 47
2.25 RTL simulation with host software diagram. 47
2.26 RTL-Software co-simulation diagram. 48
2.27 Full system software simulation diagram. 48

3.1 General architecture for a Detection System. The system is mainly divided between the Acquisition and the Processing of the surrounding environment's data. 50

3.2 High-Level Block Diagram view of the HARVA system. 51
3.3 HARVA system divides the input image within several subsets, computing for each the intensity variation in a form of an histogram. 51
3.4 Grayscale transformation routine for Image Processing applications. 51
3.5 Detecting Window algorithm for multiple locations detection. 52
3.6 Image Pyramid algorithm for multiple scale detection. 52
3.7 Multiple Pedestrian Detection. 53
3.8 Bluring filter for anonymization of target object in Object Detection algorithms. 53
3.9 HOG's components 2-level memory hierarchy. 54
3.10 HOG's components Data Cache memory layout. The memory is divided in several section for the different required accesses. 55
3.11 HOG CONTROL register bit description. 56
3.12 IMG_DIM register bit description. 56
3.13 HOG's components Data Cache memory layout. The memory is divided in several section to support the different required accesses. 57
3.14 SVM CONTROL register bit description. 57
3.15 HOG component's block diagram. The component is designed as a 4-stage pipeline, aiming an increased data throughput. 58
3.16 $2D\_CONV$ component's state machine. 59
3.17 Block Diagram of the logic implemented for the computation of each value of the G_x and G_y derivatives. 60
3.18 MAG_CALC block diagram, illustrating the pipeline behavior implemented for the Magnitude Gradient computation. 61
3.19 MAG_CALC component's state machine. 61
3.20 BIN_CALC component's block diagram. 62
3.21 Tangent's trignometric upper half of the unit circle. 63
3.22 BIN_CALC component state machine. 63
3.23 $HIST\_CREAT$ component's state machine. 64
3.24 $HIST\_CREAT$ component's block diagram, illustrating the histogram update operation. 64

3.25 Block Histogram Normalization Stage's block diagram. The red signal symbolize the control signal generated by the Control Path to manipulate the behavior of the implemented modules. The blue signals represent the signals that impact the control signals generation. The green signals are used to represent the data manipulated within the stage. 65
3.26 $HIST\_NORM$ component's state machine. 65
3.27 SVM component's block diagram. The red signal symbolize the control signal generated by the Control Path to manipulate the behavior of the component. The blue signals represent the signals that impact the control signals value. The green signals are used to represent the data manipulated within the SVM. . 67
3.28 SVM component's state machine. 67
3.29 $QUANT$ component's block diagram. 68
3.30 LIN_COMB component's block diagram. 69
3.31 SVM component's implementation as a two-stage pipeline, targeting increased throughput by exploiting data-parallelism. 70

4.1 Vivado Simulator flow for HDL development. 72
4.2 Example of INRIA dataset positive samples. 73
4.3 ZC706 Evaluation Board. 73
4.4 ZC706 Evaluation Board High-Level Block Diagram. 74
4.5 Fixed-Point error compensation hardware. 76
4.6 Block Diagram test case for the INRIA dataset. 77
4.7 HSYNC event for the HARVA system. 77
4.8 HSYNC test case for the INRIA dataset. 78
4.9 Detecting Window test case for the INRIA dataset. 79
4.10 Object detection application as Pipeline multi-threaded Model. 80
4.11 Preprocessing component flowchart for the HARVA system. 81
4.12 *imread* API's documentation. 82
4.13 OpenCV *resize*'s documentation. 82
4.14 Convolution Stage software implementation for the HARVA's profiling. 83
4.15 Gradient Calculation Stage software implementation for the HARVA profiling. . 84
4.16 Histogram Stage software implementation for the HARVA profiling. 85

4.17 Normalization Stage software implementation for the HARVA profiling. 86
4.18 SVM thread's diagram. 87
4.19 Oprofile results for the profiling of the HARVA software implementation. 88
4.20 Hardware profiling chart for the HOG HDL design. The chart provides information about the latency variation for different number of cores configurations of the HOG. 89
4.21 HOG HDL design's synthesis results. 90
4.22 HOG HDL design's synthesis results for DSP slices. 90
4.23 HOG HDL design's synthesis results for BRAM blocks. 90
4.24 Hardware profiling chart for the SVM HDL design. The chart provides information about the latency variation for different number of cores configurations of the SVM. 91
4.25 HOG HDL design's synthesis results. 91
4.26 HOG HDL design's synthesis results for BRAM blocks. 92
4.27 False Positives within a Object Detection System. 92
4.28 False Negatives within a Object Detection System. 93

List of Equations

2.1 Conversion from IEEE Floating Point to Real number. 15
2.2 Uniform Step size formula. 17
2.3 Uniform Quantization formula. 17
2.4 Image scaling formula. 27
2.5 Pixel Selection formula. 28
2.6 L1 Normalization formula. 30
2.7 Gaussian Function formula. 31
2.8 Convolution kernels for computation of image Horizontal and Vertical derivatives. . . 36
2.9 Image Magnitude Gradient formula. 36
2.10Image Angle Gradient formula. 36
2.11SVM formula for mapping new data point into the its hyperplanes. 38
2.12SVM formula for classification of new data points. 38
3.1 XILINX CORDIC Fixed Point converting factor. 61
3.2 Bin assignment formula. 62
3.3 Extended Bin assignment formula. 62
3.4 L1 Norm multiplication factor. 65

Acronyms List

ADC Analog-to-Digital Converter.

ALU Arithmetic Logic Unit.

API Application Programming Interface.

ASIC Application Specific Integrated Circuit.

AXI Advanced eXtensible Interface.

BRAM Block Random-Access Memory.

CORDIC COordinate Rotation DIgital Computer.

CPU Central Processing Unit.

DLP Data-Level Paralellism.

DMA Direct Memory Access.

FIFO First In First Out.

FPGA Field-Programmable Gate Array.

FPU Floating-Point Unit.

GPL General Public License.

GPU Graphics Processing Unit.

HARVA Hardware Accelerated Real-Time Linux Video Anonimyzer.

HDL Hardware Description Languages.

HOG Histogram of Oriented Gradients.

HSYNC Horizontal Synchronization.

IEEE Institute of Electrical and Electronics Engineers.

ILP Instruction-Level Parallelism.

INRIA Institut National de Recherche en Informatique et en Automatique.

IP Intellectual property.

LuT Look-Up Table.

MMU Memory Management Unit.

OS Operating System.

PLD Programmable Logic Device.

POSIX Portable Operating System Interface.

RAM Random-Access Memory.

RTL Register-Transfer Level.

RTOS Real-Time Operating System.

SIMD Single Instruction Multiple Data.

SoC System On Chip.

SVM Support Vector Machine.

VHDL Very High Speed Integrated Circuits Hardware Description Language.

VSYNC Vertical Synchronization.

YOLO You Only Look Once.

Chapter 1

Introduction

In this chapter the Hardware Accelerated Real-Time Linux Video Anonimyzer (HARVA) Dissertation's scope and goals are addressed, as well as the motivation for the development of the proposed system.

The chapter is concluded with an brief explanation of the document's organization and the contents addressed.

1.1 Contextualization

Embedded Systems are an aspect of Computer Systems technologies, presenting themselves as an efficient solution where both software and hardware are co-designed, aiming the optimization of the SWaP-C (Size, Weight, Power and Cost) metrics.

This is a challenging task and one that requires a great knowledge of the technologies present in the system, requiring research and in-depth study, presenting itself as an great source of technological development.

Embedded systems have gained importance with the growth of technology, evolving from bare metal applications to fully supporting Operating System (OS)s and complex software, as depicted in the figure 1.1. [1]

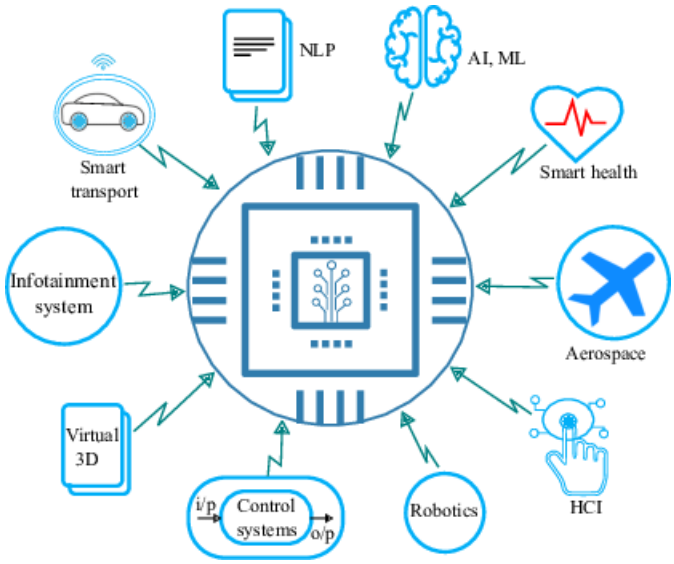


Figure 1.1: Embedded System's wide range of applications in modern technology.

Modern technology is developed for increasingly complex systems, with the rising of concepts such as Internet of Things and Industry 4.0 boosting a demand for connectivity and intelligence in a great variety of devices. The everyday life of the human being is full of smart technological devices, from smart appliances, which facilitate the home chores, to the more complex systems such as autonomous driving. The exponential growth of technology has also boosted the industry, with the implementation of production lines mainly controlled by robotics systems, increasing the daily number of units produced.

Machine Learning algorithms are designed to enable the realization of human tasks, using surrounding environment data to make decisions independently.[2] This algorithms are present in almost every device and industries, as represented in figure 1.2.[3] One popular Machine Learning problem is the Object Detection in images, or video streaming, implemented by several devices used on a daily basis, such as in the smartphone for image focus.

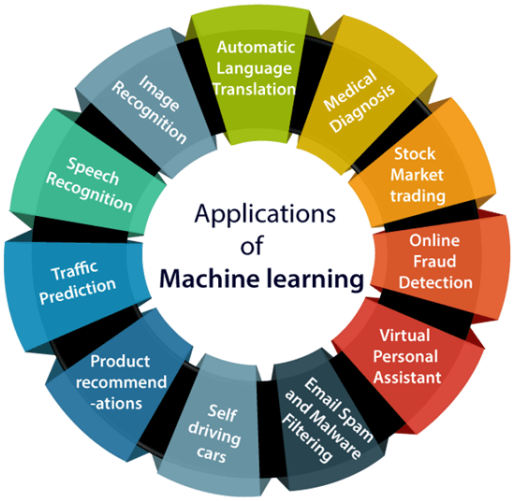


Figure 1.2: Machine Learning several application areas in technology.

The porting of Machine Learning algorithms to Embedded Systems is a challenging task, due to the high complexity and computational cost associated with these algorithms and the fact that these systems present limitations in terms of memory and performance, which are two key factors for Machine Learning applications. A pure software implementation might present itself as not reliable, or even feasible for more constrained systems. The solution is to develop the software application in the host system, where all resources are available to execute the algorithm normally, and analyze its behavior for bottlenecks, i.e., parts of the algorithm which are limiting the performance. After the analysis of the application, the most computational demanding parts of the algorithm are implemented as hardware components, decreasing the latency and the dependency of the CPU performance. In terms of memory requirements, there are multiple solutions provided by open source libraries to tackle this problem, either by decreasing the coefficients required or by mapping the Floating Point values to smaller integers.

In this Digital Age where personal information is used as a major currency for companies, which gather all sorts of data about to wide its customer base, even systems designed to a simpler task can be used to compute important data. Object Detection Systems can be exploited to trace which day and hour has the most people on the set area, an important data for possible investors interested in creating a business in the same area. In the same manner, Detection Systems targeting pedestrians might be used to gather the identification of the persons, by storing the detected images and run a recognition software. Anonymity in this Digital Age is

fundamental for the protection of human rights, providing safety and privacy. ref[https://digital-rightswatch.org.au/2021/04/30/explainer-anonymity-online-is-important/]

1.2 Motivation and Goals

Embedded Systems was always the major area of interest of the author since the start of his academic path. The possibility to combine Embedded System and Machine Learning, which is one of the most popular topics in current technology, sparked great interest and joy from the author. Porting the Linux OS to embedded systems enabled faster software development, due to the availability of several open source libraries targeting a vast range of applications, such as robotics and mobile devices.

The possibility of using open source implementations, which offer several solutions for problems across a wide range of areas, increased the importance of intelligence and connectivity in modern devices. One of the areas that benefited most with this demand was Computer Vision, which is used in a wide range of systems for several applications.

The development of software and hardware components to produce efficient solutions covers all the necessary skills for modern engineering requirements.

1.3 Contributions

This dissertation addresses the development process of an Embedded System and the challenges an engineering team is presented with.

The hardware components were designed to enable several levels of parallelism, i.e., a different number of values processed in parallel, providing the user the possibility to choose a design more latency or resource-oriented, and to enable different configurations without requiring major changes in the implemented Register-Transfer Level (RTL).

1.4 Dissertation Structure

This dissertation addresses the development of the Hardware Accelerated Real-Time Linux Video Anonimyzer system and is structured into five chapters, which will be explained in this section.

The Introduction chapter provides an introduction to the dissertation as well as motivation and main contributions.

The second chapter addresses the theoretical foundations necessary for the development of a Linux-based Embedded System and the required modifications to the kernel for targeting real-time applications. A light introduction to Machine Learning and Computer Vision areas is included to promote the necessary knowledge for understanding the developed system. The chapter finishes with topics regarding hardware development through Hardware Description Languages (HDL) and the several available methodologies to test a system with software and HDL modules.

The third chapter details the implemented system and the decisions on which its development was based. The possible configurations for the developed modules are discussed and the advantages and disadvantages are explained. The interfaces between the software and the HDL components are also addressed.

The fourth chapter presents the case of study for the HARVA system, which is based on a popular dataset in the Computer Vision community targeting pedestrian detection. The chapter finishes with a discussion of the results from the software and HDL implementations and the conclusions taken by the author related to the developed work.

The fifth chapter addresses the developed work for the HARVA system focusing on the conclusions, and evaluates possibilities of future work.

Chapter 2

Theoretical Foundations

This chapter addresses the technology and concepts used in the development of the HARVA, starting with the definition of Embedded Systems and their placement in the real-time spectrum.

Concepts from the Computer Architecture area supporting the system's memory hierarchy and data parallelism are addressed for support of the system implementation in the later chapters.

Linux is introduced with an examination of the key features that make it the developer's choice for the OS in embedded systems, as well as the necessary modifications to make it suitable for Real-Time applications.

A light introduction to Machine Learning and Image Processing is included, explaining the different algorithms and their applications. This information is useful for a better understanding of the system developed.

The chapter finishes with the description of the Hardware-Software Co-Design methodology and the processes associated with the development of user-defined hardware.

2.1 Embedded System

Embedded systems are computers designed in hardware and software to meet specific constraints, targeting the realization of a specific task within a bigger system.

Embedded systems are optimized solutions for size and power consumption and increase the system's reliability and performance.

These systems are present in a wide range of devices, from portable ones like the popular smartwatches to more complex systems like the Engine Control Unit present in every modern automobile, represented in the figure 2.1.[4]

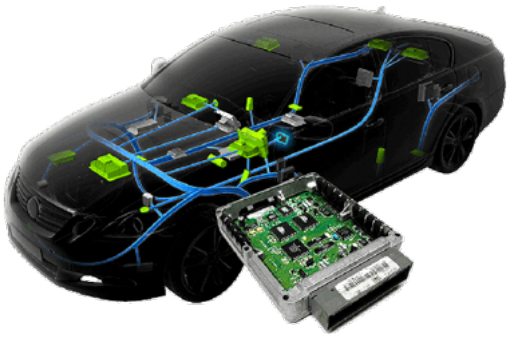


Figure 2.1: The Engine Control Unit, introduced in 1970, plays a fundamental role in ensuring the optimal performance of the automobile.

2.1.1 Generic Embedded System's Hardware

An Embedded System is designed for a specific use case, implementing only the necessary components for the task realization. Some components are generic for a wide range of applications, providing core functionalities. These components are dimensioned to fit the needs and constraints of the target system.

The generic components present in almost every computer system are:

- CPU - Responsible for fetching, decoding and executing the programmed instructions that comprise the user-defined application. Multiple architectures are available in the market suitable for different use cases. This unit is the key factor to both the system's power consumption and instruction throughput, requiring a well-thought tradeoff;

- Memory banks - Logical units of data storage used in electronics devices. In most computers, there are two types of memory present: Volatile and Non-Volatile. Volatile memories require power to retain the stored data and are used for the heap and the stack of the different software components. Non-Volatile Memories retain the stored data even if power off event happens, which is why it is used for storing the application code as well as constant data.

- Real-Time Clock - Precise integrated circuit that runs independently of the CPU and is essential for any device that needs accurate time functionalities;

- Communication ports - Essential for communication with external devices, such as sensors or monitoring systems.

Computers often use processors, which together with the CPU implement hardware components designed for specific tasks. One popular example, present in most modern computers, is the Direct Memory Access (DMA) designed for handling data exchange between different components within the system. The same concept is applied to hardware accelerators, where a task is implemented as a user-defined hardware component and integrated into the processor.

2.1.2 Storage

In embedded systems, the storage capacity is dimensioned to meet the needs of the system, due to a decreased need for storage when compared with other computers. The hard drive solution, which is the solution for computers where large blocks of data are needed to be stored, is not optimal for embedded applications since only a small part of the available storage would be used.

Using hard drives for storage in embedded systems would also be impracticable since these systems are often required to be of reduced volume and power consumption.

In most Embedded Systems Flash memories are used since this technology is available in smaller storage sizes and volume, provide better power efficiency, consuming around 50% less than the Hard drive solution, and greater robustness, i.e., less prone to physical damage and unexpected failures, which is fundamental for critical applications [5].

The Flash solution is also beneficial in terms of memory access latency, providing faster read/write operation than the Hard drive implementation.

2.1.2.1 Flash Memory

2.1.3 Real-Time Systems

Real-Time systems are computer systems that operate by strict timing constraints within it must acknowledge and handle critical events. Real-Time applications must be implemented respecting the core concepts of Functional and Time correctness [6].

Functional correctness requires that all computations must be precise and correct, even if it implies that the logic implemented is less efficient. Data integrity is more important than the system throughput.

Time correctness defines a time frame within which all necessary computations must be concluded. If the time frame is not respected, the data is considered compromised and discarded.

Real-time systems can be classified into two types: Hard and Soft Real-Time. The difference between a Hard and a Soft Real-Time System lies in the severity of the consequences of missing a deadline [7].

In Hard Real-Time systems violating the time frame means catastrophe and can result in great economic damage, or the worst-case scenario in the loss of lives. One example of a Hard Real-Time system is the braking system used in the automotive and aircraft industries, which must ensure that the system actuates within a stipulated time or else the safety of the passengers is liable to danger.

In Soft Real-Time applications, missing a deadline does not endanger the user or the system itself. This scenario can present itself as economic damage as well, due to customer dissatisfaction. A common example of Soft Real-Time Systems is the multimedia applications, like the video players present in every streaming platform, where the consequence of missing the timing deadline is the user experiencing lag in the video reproduction, which can be very unpleasant but not dangerous in any kind.

The structure of a Real-Time system [6] is composed of two types of components, as it is depicted in figure 2.2.

The Controlling component is responsible for acknowledging and handling the Controlled components, whose function is to interact with the external environment and raise an event when certain conditions are met.

The communication between the controlling and controlled components can be done in two manners:

- Periodic - The communication is initiated by the controlling system, pooling every controlled component for events raised. The interaction is predictable and the execution flow of the controlling components is not interrupted;
- Aperiodic - The communication is initiated by the controlled component, through interrupt or events raised, and impose the immediate handling to the controlling system. This type of interaction, if poorly designed, might compromise the system due to the possibility of a high frequency of events raised, which can happen on a defective component.

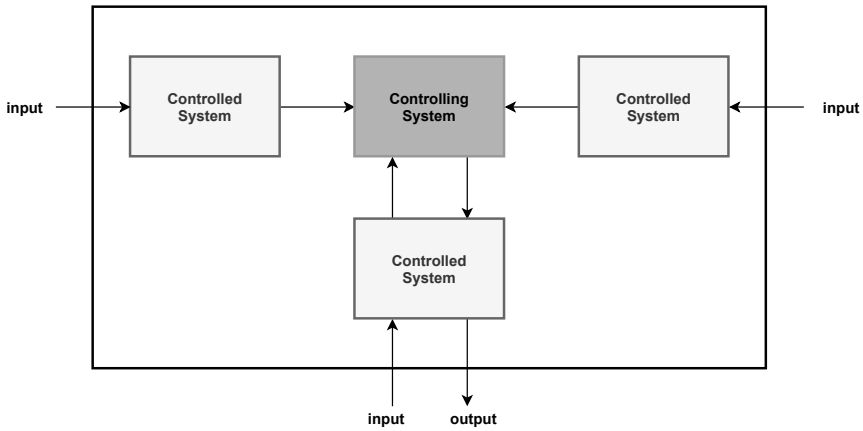


Figure 2.2: Composition of a Real-Time Embedded System.

Real-time systems are denominated deterministic because the response time to a specific and critical event is bounded and known before the implementation.

The levels of determinism and robustness of a system have to be balanced because a highly deterministic system is less adjustable to a mutable environment, therefore becoming less robust.

2.2 Computer Architecture

Computer Engineering is the branch of engineering directed to the development of computer hardware and software, integrating multiple fields of Computer Science, which studies algorithmic processes and computational systems, and Electrical and Electronic Engineering, a discipline oriented to the development of electronic devices.

In Computer Engineering, Computer Architecture is the design of a computational system through a set of rules and criteria describing the functionality, organization and implementation of a computer. This process involves the analysis of the necessary attributes for the computer and the maximization of the performance and energy efficiency while remaining within a price range.

2.2.1 Data-Level Parallelism

Data-Level Paralellism (DLP) is the concept of distributing blocks of data within different processing elements within the system, decreasing the overall latency of the operation through the increment of the system's parallelism.

Modern processors implement Vector Architectures, represented in figure 2.3[8], to process instructions, or operations, in several blocks of data concurrently.

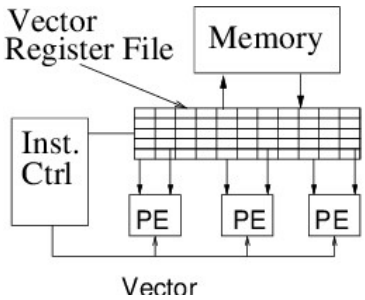


Figure 2.3: Vector Architecture designed to support the processing of several blocks of data within the multiple Processing Element(PE) implemented. Vector Architectures provide an higher throughput at the expense of more hardware for concurrent processing.

Vector Architectures are very useful for multimedia applications, where data can be processed concurrently, increasing considerably the system throughput.

With the growing popularity of Machine Learning applications, the Graphics Processing Unit (GPU) component has gained popularity with the implementation of hundreds of executing units, enabling the processing of large batches of data concurrently.

2.2.2 Instruction-Level Parallelism

Instruction-Level Parallelism (ILP) is the parallel processing of multiple instructions from a thread of execution and can be implemented at Software and Hardware level [9].

The hardware implementation of ILP is considered Dynamic Parallelism, where the processor selects a set of instructions to be processed in parallel.

The software implementation of ILP is described as Static Parallelism, where the compiler analyze the program code and decide which instructions should be processed in parallel. Modern compilers and processors try to take as much advantage as possible from the ILP, since the majority of the programs are coded in sequential blocks of instructions.

In Computer Architecture the ILP is exploited through the following techniques:

- Pipelining - Instruction execution is partialized, each stage of the pipeline handles a part of the instruction enabling multiple instructions to be processed in parallel;

- Superscalar execution - Implementation of multiple execute units, i.e., CPU elements that process data as coded by the software application, to enable several instructions to be executed in parallel;

- Out-of-order execution - The sequence of the instructions is modified, without violating data dependencies, to enable parallel execution of a block of instructions;

- Speculative execution - This technique is used for control flow instructions (e.g. branch), in which the result is predicted and the respective part of the instruction is executed before the control flow is computed itself. This method avoids the CPU stall, waiting for the control flow instruction to be executed, but if the prediction is wrong the CPU has to discard the current data and start executing the correct instructions from the start [10].

2.2.3 Pipeline

The pipeline implementation yields a reduction in the average execution time per instruction in the computational system. Although the CPU instruction throughput is increased, the execution time of each instruction increases slightly due to overhead in the control of the pipeline and the fact that each instruction must pass by all stages. The increment in the throughput is due to the exploitation of the parallelism among instructions in a sequential instruction stream, as represented in the figure 2.4[11], where each pipeline stage executes different actions of an instruction, enabling multiple instructions to be processed in parallel.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Figure 2.4: Simple RISC pipeline designed for a five-cycle instruction executed. The processor performance is increased by five times, compared with a processor not pipelined. The pipeline is defined within the stages: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB =write-back.

Pipeline designing faces hazards, situations that prevent the next instruction from the instruction stream to be executed in its designated clock cycle, which impacts, negatively, the CPU throughput [11]. The three classes of hazard are:

- Structural hazard: A structural hazard refers to a situation where different instructions require access to the same resource simultaneously. The pipeline stalls the next instruction until the resource is available, delaying the execution of the instruction stream.
- Data hazard: A data hazard occurs when the next instruction to enter the pipeline depends on the result of the computations of the previous instruction. The pipeline stalls the execution of the new instruction until the completion of the current one.
- Control hazard: A control hazard happens when the pipeline misses the branch prediction and consequently pulls instructions that should not be executed. The whole pipeline data has to be discarded and the correct instructions fetched.

Compilers take into account the pipeline implemented by the underlying processor and reorganize the instructions, when necessary and possible, to try to avoid pipeline stalls.

2.2.4 Cache

Modern computer memory is implemented through hierarchy levels, represented in figure 2.5[11], to optimize memory access and the overall performance of the computer system. Cache memories, a faster type of memory, are implemented closer to the CPU to enable more efficient data access, by avoiding constant direct fetches from main memory and its inherent high latency.

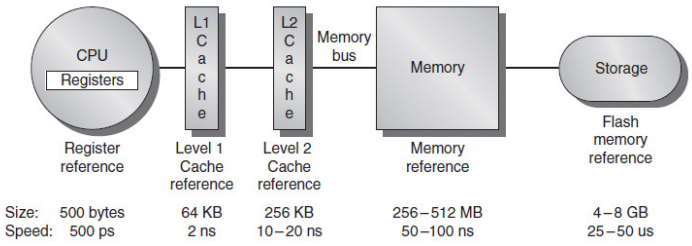


Figure 2.5: Memory hierarchy in modern Computer Systems, targeting a more optimized data access through implementation of faster and smaller memories between the processor and the Main Memory.

The Cache concept is based on the principle of locality, which is defined in computer science as the propensity of a processor to access a set of memory locations repetitively within a period of time. The principle is divided into two types: temporal and spatial locality. The temporal locality states that a block of data is likely to be reused within a short period of time and the spatial locality refers to the reuse of blocks of data close in storage locations [11].

In a computer system, there are two types of cache that can be implemented: Instruction and Data caches.

Instruction cache fetches chunks of instructions from code memory, based on the concept that most programs will execute the instructions sequentially.

Data cache stores frequently used data for future computations, avoiding the costly access to main memory. One popular example is the loop constructs of the programming languages, where the same data is used consecutively.

2.3 Data Representation

Data representation in a computational system refers to how data is stored and process within the computer circuitry. In modern computers, there are several data formats available fitting different use cases, such as the floating-point format that targets the representation of real numbers.

2.3.1 Floating-Point Representation

In a computer system, the Floating-Point data type is used for the representation of an approximation of a real number. This data type is very used in systems where very small or very large numbers are used and great precision is required.

One popular example is Machine Learning applications, which requires the use of precise coefficients to achieve the desired performance.

The Institute of Electrical and Electronics Engineers (IEEE) defines two Floating Point formats: Single and Double Precision. The Single Precision standard represents the real value within 32 bits, divided into 23 bits for the Fraction, 8 bits for the exponent and 1 bit sign, as represented in figure 2.6.[12]

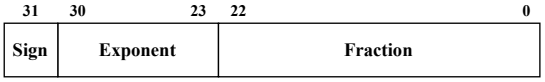


Figure 2.6: Single Precision Floating Point representation, as defined by the IEEE Standard for Floating-Point Arithmetic.

The Double Precision standard represents the real value within 64 bits, divided into 52 bits for the Fraction, 11 bits for the exponent and 1 bit sign, as represented in figure 2.7.[12]

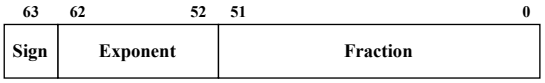


Figure 2.7: Double Precision Floating Point representation as defined by the IEEE Standard for Floating-Point Arithmetic.

The Fraction consists of the significant bits of the number and depending of the notation of the exponent it might be an integer or a fraction. The Fraction section is stored in the Two's Complement by the IEEE Standard for Floating-Point Arithmetic. [12]

The Exponent section stores the coefficient value used to compute the real number and is stored as the excess of 127, for Single Precision, and of 1023, for Double Precision. The calculation of the real number x from Single and Double Precision, respectively, is given by the equation (2.1), where S represents the signal bit, F the Two's Complement Fraction and E the excess exponent.

$$x = (-1)^S * (1.F) * 2^{(E-127)}$$
$$x = (-1)^S * (1.F) * 2^{(E-1023)} \tag{2.1}$$

Floating-Point operations within a computer system can not be handled by the same circuitry which processes other data formats. The Floating-Point Unit (FPU) is a co-processor designed for handling Floating-Point data, including addition, subtraction, multiplication and division. Software implementations that rely heavily on Floating-Point computations face performance problems since in a computer there is a limited number of FPU available therefore multiple calls to the FPU will generate a bottleneck on the system.

2.3.2 Fixed-Point Representation

Fixed-point is a representation of real numbers, similar to the floating-point format, but with a constant number of digits before and after the radix point, the separation between the integer part and the fractional part of the real number. This representation is used when the processor does not implement a Floating Point Unit or when it is necessary to accelerate the computation of real numbers.

A Fixed-Point value is simply a real number scaled by a pre-determined factor and truncated to an integer. The scaling factor is recommended to be a power of 2 for computation efficiency or a power of 10 for human convenience.

Addition and Subtraction of two fixed-point values, with the same precision, can be done by simply computing their respective integer arithmetic operation. Multiplication can also be executed through the respective integer operation, but the scaling factor is squared.

The division operation requires the re-scaling of the numerator value, i.e., multiplication of the numerator twice by the scaling factor. After this step, the division can be computed as it would be with integer representation.[13] Fixed-Point representation also presents advantages in terms of hardware development, since the circuit for fixed-point is less complex when compared to floating-point, resulting in smaller chip size and therefore less manufacturing cost. Fixed-Point representation is also simpler to implement in combinational circuits, such as the Arithmetic Logic Unit (ALU).

Another benefit of implementing a component using fixed-point instead of floating-point for real data representation is that fixed-point hardware is more power-efficient.

2.3.3 Quantization

Quantization is a mathematical process that maps high precision fractional values to smaller integer values. This process is very beneficial in terms of storage and performance, due to faster computation of the integers and better power energy efficiency when compared with floating-point.

The Quantization process has been a subject of great interest in the porting of machine learning models for embedded deployment since these systems often have limited memory and power consumption is a very relevant factor. Machine learning frameworks, like Tensorflow or PyTorch, have popularized the use of 8-bit integers for the implementations of optimized models,

reducing considerably the memory usage and providing faster execution times. The quantization process defines a range of values within the input data that can be mapped, called compression layers. The number of layers is given by $2^{bitwidth}$. Each layer multiplied by the scale factor, or most commonly known step size, produces the decompressed value, an approximation of the original value. Quantization is a lossy compression, which means it will introduce an error in computations [14]. Quantization can be of two types: Uniform and Non-Uniform.

In the Uniform Quantization, the compression layers are uniformly spaced, i.e., the step size is constant.

In the Non-Uniform Quantization, the step size has different values for different layers.

Non-Uniform Quantization produces a smaller quantization error but demands more complex computations. The Uniform Quantization, which is used in this dissertation, defines the step size value by using the extreme values of the input data and the desired bitwidth for the representation. The operation is defined in the equation 2.2.

$$\Delta = \frac{max(x) - min(x)}{2^{bitwidth} - 1} \ , \ x = [\ x_0 \ , \ x_N \ ] \ \in \ \mathbb{R} \tag{2.2}$$

The compression layers are the range of values possible to be represented with the desired bit width and each input value must be mapped to one of those values. The function that transforms an input value into a quantized value is commonly called the kernel, and the Uniform Quantization is defined by 2.3.

$$Q(x) = \lfloor \frac{x}{\Delta} + bias \rfloor \tag{2.3}$$

One well known example of quantization is the Analog-to-Digital Converter (ADC) component, used to convert the voltage applied in the input to integer values. In a system where the voltage sampled by the ADC ranges from 0 to 10 and it is required to be represented the values within a 4-bit bitwidth, the quantizer is composed by 16 compression layers, with a step size of 0.66(6).

The result of the quantization is represented in the graphics of the figure 2.8, where it is visible the loss of precision associated with the quantization error.

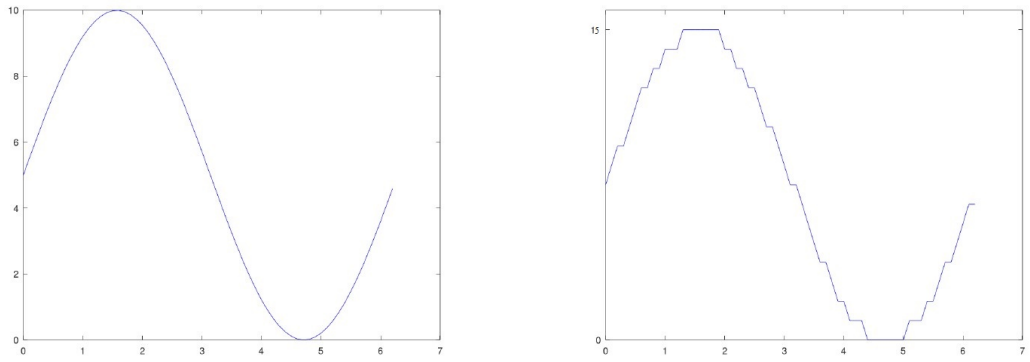


Figure 2.8: The graphic on the left side shows the voltage applied to and ADC, ranging from 0V to 10V. The graphic on the right side shows the the quantization results for the different possible values applied to the ADC.

2.4 Linux and Embedded Systems

Linux OS began as a homemade project in 1991, developed by a young computer science student at the University of Helsinki named *Linus Benedict Torvalds*, who was a strong admirer of the Unix and smaller Unix-based operating systems. Several programmers contributed to the project through a *mailing list* forum in the early stages of the project and on 5th October of 1991, the first release was announced [15]. Since its first release Linux has been widely adopted between the programming community, for the ease of use and freedom given to the user, as well as the fast software development to support new hardware and the fact that its insertion in the system can be made in a modular way, without having to reboot the system.

Another reason why Linux became so popular is the fact it is distributed under the General Public License (GPL) license [16], which grants the user with the rights to execute, modify and deliver without any cost associated.

With a vast community supporting and constantly updating the Linux OS kernel it has become more robust and reliable and nowadays is present in a wide range of devices and application areas, more than any other OS.

2.4.1 Embedded Linux

Embedded systems have grown from simple bare-metal applications, running in a very resource-limited environment, to supporting OS and all their functionalities [17]. The porting of Linux OS

has been a major factor of why the embedded systems are nowadays present in so many devices [18].

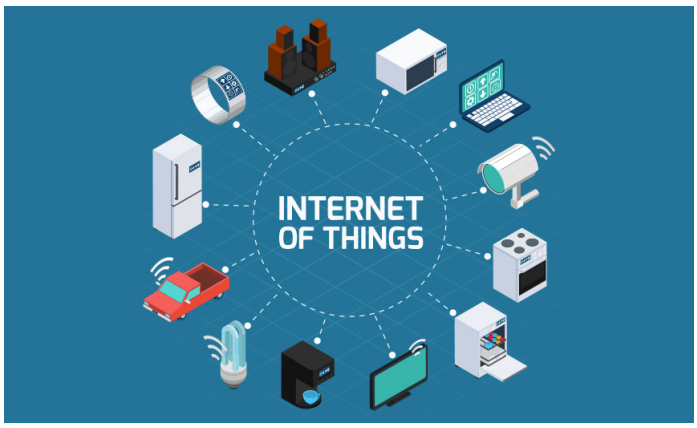


Figure 2.9: Importance of connectivity in modern devices.

The key features that make Linux OS so desirable for embedded systems are:

- Robustness - Offers robust features for communication, file management and graphics, making it suitable for critical-mission applications. Linux systems can operate without having to reboot for a prolonged time, even if an application program crashes it will not compromise any system features;
- Portability - Applications developed in embedded Linux can be used in different target platforms without any major reworks. This feature enables the re-usage of the system-level features for several platforms.
- Ease of customization - Easy to configure accordingly to the target system needs. Software to support new hardware can easily be added, as well as other system-level features;
- Ready Support - Has a great community that actively contributes with updates and support, so any developer can get help with existing issues by accessing a Linux User Group forum.

Several industries have been including Linux-based OS in their systems, due to the current demand for increased connectivity and growing popularity of the IoT concept, as represented by the figure 2.9.[19] One well-known example is the automotive infotainment system, present in almost every recent model.[20]

2.4.2 Linux and Memory Management

Linux is a Unix-based OS, therefore the key philosophy concerning memory management is "'Everything is a file"' [21]. In this concept, every resource and inter-process communication is considered as a stream of data. Each resource is accessible through a filesystem, which differs from device to device. The Virtual Filesystem is a kernel software layer that provides an interface for communication between the user space and the respective filesystem, through system calls [22]. The purpose of this layer is to allow interaction with every resource's uniformly, as illustrated in figure 2.10[22].

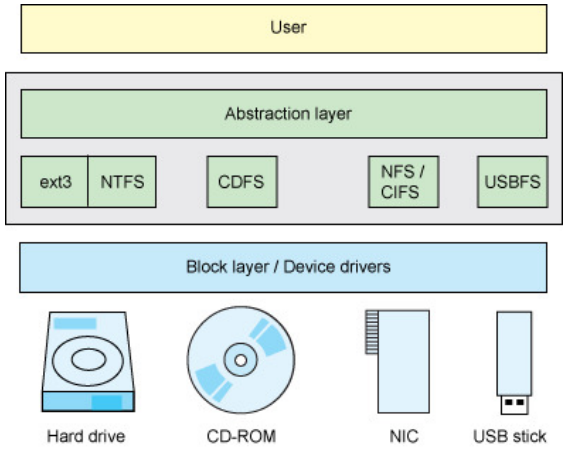


Figure 2.10: Layer hierarchy for device access in Linux OS.

Another important implementation detail, concerning memory management in Linux OS, is that a process does not access physical memory directly, instead it operates between a range of virtual addresses, denominated Virtual Address Space, which are allocated by the OS.

The Virtual Address Space starts at a low address, normally 0, and extends to the highest value defined by the system, as well as supported by the physical memory. The Virtual Address Space allocation is advantageous in terms of:

- Memory allocation: The OS is responsible for the memory assignment, reducing the overhead at the start of the process execution, as well as avoiding address collisions between running entities;
- Memory protection: Each process has its own isolated virtual address, preventing access from unauthorized entities;

- Memory fragmentation: Virtual memory enables the use of the paging technique, allowing non-contiguous blocks from physical memory to be mapped as a contiguous range of virtual addresses. The paging technique also gives the possibility to the system to allocate more memory than physically available, using a portion of the hard disk as a RAM extension, named Swap Space.

Virtual memory handling, from the translation of the virtual to physical addresses, as well as the implementation of the paging architecture, is performed by the Memory Management Unit (MMU) hardware component.

2.4.3 Linux and Real-Time applications

Linux was designed for a desktop environment, focusing on providing an enjoyable experience to the user on top of a time-sharing preemption concept, where every process is guaranteed to have a minimum of execution time, promoting a high throughput. This scheduling policy differs from the ones used in real-time applications, where the system throughput is not the key factor, instead the focus is set on the priority of the processes, which is the determining factor in the schedule routine for selecting the next entity to run. Although Real-Time concepts are not present in the Linux kernel, there are several active researches to make it suitable for mission-critical systems. The PREEMPT_RT patch is a popular solution to adapt some features of the kernel, more specifically the preemption mechanism, to Real-Time applications.

For the system to be suitable for Real-Time it is necessary to ensure the system is capable of acknowledging and servicing a critical event within a bounded latency.

2.4.4 Latency

The time difference between the event occurrence and the system response, described in figure 2.11, is denominated Real-Time latency and, as explained in the previous section, is part of the time correctness concept. Real-time latency results from a series of steps the system executes to handle critical events.

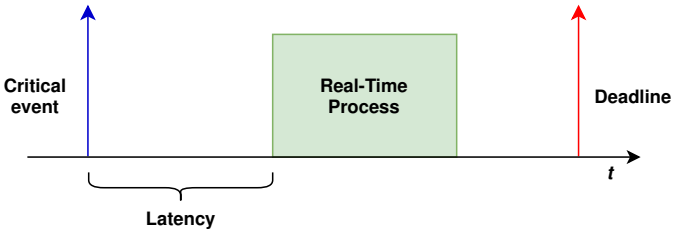


Figure 2.11: Real-Time system response to a critical event.

The Real-Time process, implemented as the system response for a critical event, remains in a Sleep state until the preemption is requested. The latency of the system response is composed of the interrupt and the scheduler executions, which are described in detail in image 2.12.[23]

The Interrupt latency is the required time for the system to acknowledge the critical event and trigger the execution of the defined handler. The handler is generally a small routine, designed for fast execution to avoid suspending the normal execution flow longer than required. Before exiting the handler routine a context switch is requested to transfer control to the respective Real-Time process.

The Scheduler latency is the time delay between the context switch request and the scheduler routine to start its execution. The Scheduler component selects the next schedulable entity to transfer control.

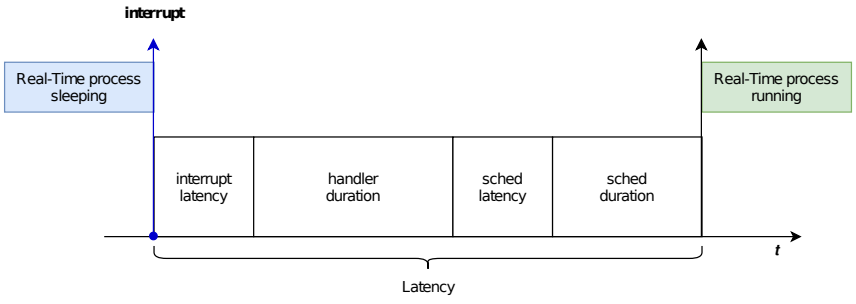


Figure 2.12: Detailed Real-Time system's response to a critical event.

2.4.5 Preemption

Preemption is a concept from the OS area which avoids the lock of a CPU core when an entity is waiting on a resource or other entities are required to run.

In Real-Time Systems, preemption enables the interruption of a running process, which is transitioned to a Sleep state, for the critical event response to be executed. A preemptive kernel enables running kernel tasks, i.e., system calls or the abbreviated form syscall, to be interrupted and suspended, avoiding the unpredictability associated with system calls, which can be very time consuming due to their dependability on resource availability.

A preemptive kernel also avoids a system call to crash the system, in case it enters an infinite loop or an unrecoverable state.

Preemptive kernels are highly suitable for systems where device drivers are implemented in the kernel space.

2.4.6 Real-Time Kernel Patch

A patch is a set of changes in the application files or data, used to fix, improve or add new functionalities to the system.

Patching makes it possible to modify a compiled system without access to the source code. This requires a thorough understanding of the object code to avoid the insertion of bugs. The PREEMP_RT patch was developed for transforming the Linux kernel into a preemptible kernel, eliminating the possibility of a higher priority process to be blocked by a lower priority process running kernel code. A preemptible kernel approximates Linux to an RTOS behavior, making it suitable for Real-Time applications [24] . The PREEMP_RT patch introduces the following changes to the Linux kernel:

- Kernel locks - The Linux kernel implements two types of locking mechanisms: spinlocks and mutexes. The spinlock entity will try the lock until it is available, while the mutex causes the process to be transitioned to the Sleep state.

  The PREEMP_RT patch redefines the spinlock implementation with mutexes, a mutex-like entity that implements the priority inheritance protocol.

- Priority inheritance - The Priority Inheritance protocol tries to avoid lower-priority jobs to blocks high-priority ones. This is done by increasing the blocking process's priority to unblock as fast as possible the higher-priority dependent.

  This protocol is added to the implementations of semaphores and spinlocks entities;

The drawback of these modifications on the Linux kernel is the increased complexity and latency of the source code. The PREEMP_RT patch allows the creation of non-preemptible kernel sections through the use of raw_spinlock_t. The PREEMP_RT patch also introduces two new possible configurations for the interruptions services requests, giving the possibility to define them as hard or soft real-time:

- CONFIG_PREEMPT_HARDIRQ - In this configuration, an ISR is run in process context, which makes it a schedulable entity. This means that an ISR can have an assigned priority, as well as being preempted by higher-priority interrupts.
- CONFIG_PREEMPT_SOFTIRQ - An ISR is run within the context of the kernel's daemon ksoftirqd, which is promoted to real-time making it a schedulable entity for the SCHED_FIFO.

  SCHED_FIFO is a simple scheduling algorithm where a running task can only be pre-empted by a higher priority entity, otherwise, it will execute until reaching the end of its execution flow.

Although the real-time patch tries to eliminate most of the unbounded latencies present in the Linux kernel, which dictate a non-deterministic behavior, it still can not be considered a hard Real-Time OSRTOS.

Embedded Linux with the PREEMPT_RT patch applied is still a great solution for critical systems such as robotics or computer that communicate with hard real-time software.

2.5 Image Processing

Image processing is the process of manipulating digital images through an algorithm, altering the original properties of the pixel data. Several systems use image processing algorithms to modify the input image or to process the pixel data to extract relevant information, as is the case of Computer Vision applications.

2.5.1 Image Decoding

An image is a very large data structure containing pixel information and the synchronization signals necessary for a playback device to be able to display it.

The Image synchronization signals, are used to synchronize the image dimensions, and consequently the image active area, for the playback device to be able to display correctly.

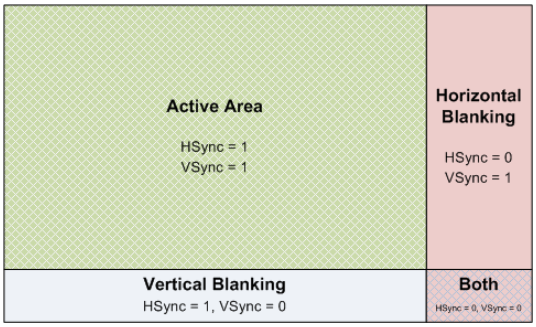


Figure 2.13: Image perception by a playback device. The synchronization signals dictate the active are of the image.

The synchronization signals communicate to the playback device through a rising edge, i.e., the signal changes value from 0 to 1, and a falling edge, i.e., the signal changes value from 1 to 0, of the Horizontal Synchronization (HSYNC) and Vertical Synchronization (VSYNC) signals, illustrated in the figure 2.13. [25]

Image processing can be problematic in terms of storage and transmission latency since the acquisition system that samples the surrounding environment transmits the frames to the system's main memory. To avoid high latency values, in the mentioned transmission, images are encoded, targeting a decreased size, and decoded when viewing or editing is requested. Image compression and decompression are handled by a component denominated Codec, which is developed specifically for each image file format. Codec compression can be Lossless or Lossy.

Lossless compression processes the original data while retaining its information, which results in a decoded image with the same quality as the original.

Lossy compression tries to achieve a higher level of compression, reducing the quality of the original image. Consequently, the decompressed image will have decreased quality, compared with the original data. The development of the codec for the desired file format is complex, time-wasting and error-prone. Since computer vision is a very popular area in recent years, there are multiple open-source codec solutions for all the different file formats, which can easily be integrated into a user defined application.

2.5.2 Color Space Conversion

In image processing applications it is common to convert images from the original to a specific color space, which better fits the needs of the system. A color space is an abstract model that describes how color is represented, enabling its reproduction through a physical device. The most popular color space is the RGB, which is the default for image processing systems, where color is defined by the 3 channels Red, Green and Blue.

HARVA is an Edge Detection model, where the main concept is based on the theory that object edges have higher intensity values than the background. The RGB color space, usually implemented as default in the Codecs, does not suit this use case so a color space conversion to a more appropriate color model is necessary.

The CIELAB color space [26], also known as LAB, expresses color through 3 channels: L, which stands for lightness, i.e, pixel intensity, A which represents the green–red component ratio and B which represents the blue-yellow ratio. The L channel from the CIELAB color space is an image representing the intensity for each pixel across the image and is commonly known as a grayscale image. The grayscale image suits directly the HARVA use case, while also being beneficial in terms of storage and Field-Programmable Gate Array (FPGA) fabric resource utilization since it is composed solely of one channel.

2.5.3 Image Resize

Image resizing is a mathematical operation that transforms an image from the original to a target set of dimensions. It can be used to decrease or to increase the number of pixels of the image. The resizing process can be decomposed into two stages: Pixel selection, where the target pixel location and its neighbors are calculated, and Interpolation, the process where the subset from the Pixel Selection stage is used to calculate the new pixel. The ratio between the original and the resized dimensions is denominated scaling factor, where for values lesser than 1 the resulting image will have a higher pixel grid, while for values higher than 1 the resulting pixel grid will be decreased.

2.5.3.1 Pixel Selection

Pixel Selection is the first stage of an image resizing algorithm, where a subset of pixels from the original image is used for calculating a new pixel, belonging to the newly resized image. There are several methods for interpolating the new image, which might impose some differences in the Pixel Selection routine, but the base concept is similar with the changes lying on the number of neighborhood pixels used.

Calculation of a pixel subset location [27] depends directly on the scaling factor, so the first step is to calculate the dimensions of the new image and for that the scaling factor must already be defined.

Since the scaling factor is simply the ratio between the original and the newly resized image, the new dimensions are calculated through the equation 2.4.

$$\begin{aligned} Width_{resized} &= \frac{Width_{original}}{scaling factor} \\ Height_{resized} &= \frac{Height_{original}}{scaling factor} \end{aligned} \tag{2.4}$$

The second step of the Pixel Selection is to use the calculated dimensions and the scaling factor to calculate the location of the subsets.

This process is expressed mathematically in equation 2.5, where R_o and C_o are the coordinates of the target pixel. Finding the neighborhood of the target pixel is easily done by offsetting R_o and C_o.

The neighborhood subset differs in the several Interpolation implementations, with the Bilinear using only 4 neighbors, while the Bicubic using 16. Despite the different sizes of the neighbor grid, the Pixel Selection routine remains the same.

$$
\begin{aligned}
R &= [\ 1\ ...\ Width_{resized}\ ] \\
C &= [\ 1\ ...\ Height_{resized}\ ]
\end{aligned}
\tag{2.5}
$$

$$
\begin{aligned}
R_o &= \lfloor R * scaling\ factor \rfloor \\
C_o &= \lfloor C * scaling\ factor \rfloor
\end{aligned}
$$

2.5.3.2 Interpolation

Interpolation is a mathematical operation that enables the estimation of unknown points, by trying to fit a polynomial into the known data. Resizing an image is interpolating every unknown pixel of the resulting image, using the neighborhood pixels of the target location [27]. Image resizing is often used in image processing applications, therefore several open-source implementations available offer popular solutions such as:

- Nearest Neighbor: More commonly known as pixel repetition, simply copies selected pixels from the input to the output image. The quality of the resized image decreases considerably, gaining a pixelated aspect.

  Although this approach is the most simplistic, computation-wise, it is not recommendable for systems where the image quality affects directly the system performance.

- Bilinear: In this method, the output pixel is based upon the weighted contribution of the closest 2x2 neighborhood, providing a better result than Nearest Neighbor. Introduces smoothness in the resized image, decreasing the pixel intensity across the image.

  This approach, although it produces resized images with great quality, is not the optimal solution for edge detection models.

- Bicubic: Similar to Bilinear, but with a 4x4 neighborhood grid, where the contribution of each pixel is weighted accordingly to its distance to the output pixel location.

  The key advantage to choose bicubic instead of bilinear is that with this approach the resized image edges are not compromised, not affecting the quality of the edge detection model.

The choice of which interpolation to be used in the image resizing operation must be a well-thought tradeoff, between the quality of the output image and the computational effort. There are 2 use cases where there is an interpolation that better fits the target system needs:

- Resource Prioritization - Systems designed for prioritization of the resource utilization and computational effort instead of the detection performance. In this use case a minimal, and latency-oriented, implementation is chosen. The system is not concerned with the decrease in quality of the output image, as long as the detecting model achieves an acceptable detection rate. The detection rate will decrease at each resize of the same image, which can be problematic for models that process images at several scales.

  A suitable interpolation algorithm is the Nearest Neighbor, because it is simple and understandable, which means faster development and implementation;

- Detection Prioritization - Systems designed for achieving the best detection performance possible, independently of increased resource utilization and, since the complexity of the algorithm is higher, computational effort.

  This approach avoids the quality decreasing at the resize operation. For this use case, 2 popular algorithms can be implemented to better fit the target system. Both options are reliable, image quality-wise and the only difference is on the degradation of the image sharpness introduced by the Bilinear Interpolation, which plays a big factor in the detection performance when an Edge Detection model is used.

2.5.4 Normalization

Normalization is a statistical process that transforms data on different scales to a common scale, where the sum of all values is equal to 1. Normalization is very used in image processing to reduce, or even remove, noise by scaling the original pixel intensity values to a range of more human natural values, improving considerably the detection quality. In image processing applications the normalization is a well-thought tradeoff since there are several algorithms that better fit different use cases. There are two popular implementations, used in a wide range of systems due to their characteristics: L1 Norm and L2 Norm. The normalization robustness of the L1 Norm is superior since the L2 Norm squares the error value, resulting in a more sensible system. The L1

Norm is resistant to outliers and avoids skewed normalized data, represented in the figure 2.14, within the system [28].

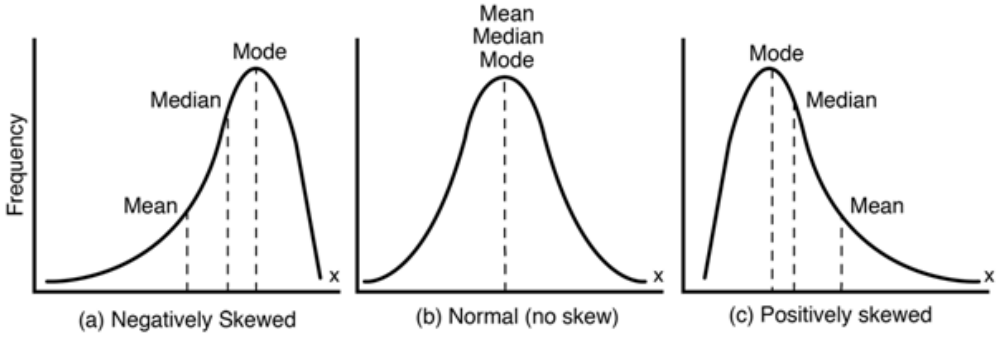


Figure 2.14: The graph (a) describes the disposition of negatively skewed data, with the median and mode placed on the right side of the mean value. Graph (b) describes data disposition without skew, which mean, mode and median values are the same. Graph (c) describes the disposition of positively skewed data, where the median and mode are placed on the left side of the mean value.

In terms of computational efficiency, the L2-Norm has the upper-hand, allowing a more efficient computation of the algorithm due to having an analytical solution, which the L1 Norm does not.

The key difference between the two normalization algorithms is the Built-in feature selection characteristic of the L1 Norm, which tends to be a deciding factor. The L1 Norm produces sparse coefficients, which means a great part of the data will have a value near zero, or equal to zero. These values can be ignored in the posterior calculations, since they will not have any impact, reducing the required memory for storing the data, as well as the number of computations necessary. [29]

The normalization algorithm chosen for this Dissertation was the L1-Norm, a robust algorithm resistant to outliers, therefore avoiding skewed normalized data, represented in the figure 2.14.

The mathematical expression for the L1-norm, specified in equation 2.6, defines the $\|v_1\|$ as the sum of all elements in the block histogram and ε as the smallest value in the block histogram, used as prevention to avoid the denominator of the fraction to be zero, which would cause a divide by zero exception on the system, compromising its normal execution.

$$\frac{v}{\|v_1\|} + \varepsilon \tag{2.6}$$

2.5.5 Gaussian Function

The Gaussian function, named after the mathematician Carl Gauss, is a characteristic '" bell shape"" curve given by the formula (2.7), where A represents the amplitude of the function and (x_0 , y_0) are the coordinates of the Gaussian function center, where the value is equal to the amplitude. The σ variable represents the spread of the blob, i.e, the region where the function value is equal to the amplitude A.

$$Gaussian(x, y) = A * \mathrm{e}^{\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2}} \tag{2.7}$$

The Gaussian function is very used in image processing software as a blurring filter applied to reduce the noise in the input images, with the drawback of impacting the detail and the image sharpness. It is also used in pre-processing routines in some Computer Vision applications to enhance pixel data in different scales.

The σ value can be designed to set the desiring blurring effect, for higher values of σ the image will present a stronger blur effect.

2.6 Machine Learning

Machine Learning is the study of computer algorithms that based on sampled data build a mathematical model able to predict, or classify, new data without being explicitly programmed for that task [30].

The sampled data used in the training phase, where the model is built, is often called dataset. The dataset is organized in a table-like structure where each column matches a variable and each row is an observation of the set of variables. Usually, the dataset is partitioned into training and testing samples.

The dataset variables, which represent a specific characteristic of the system, are divided into Independent and Dependent.

Independent variables are used by the model to make a prediction, or decision, on the data, since they represent the important information about the system sampled. Each Independent

variable has a different impact on the prediction, which is why it is common to test a different set of Independent variables to understand which produces the best performance.

Dependent variables represent the label or the class, a dataset observation belongs to. For new data, this variable is computed by the model inference function.

2.6.1 Feature

A Feature in Machine Learning is a fragment of information, extracted from processed data that is relevant for solving a Classification or Regression problem.

A Feature is defined within the context of the system developed, i.e., for different applications, there will be different types of features. A Feature may be points or edges in input images or the result of a general neighborhood operation on the data pixels.

2.6.2 Feature Descriptor

A Feature descriptor is an algorithm, divided into several steps that process input data by encoding important information in a feature vector, used in a classification, or regression, stage.

There are Feature Descriptor targeting different applications, such as audio and visual content. One popular feature descriptor is the Histogram of Oriented Gradients (HOG) which extracts object edges through the pixel intensity orientation within the image.

2.6.3 Supervised Learning

Supervised Learning models infer the decision function, which can map a set of inputs to the respective output class, through a series of labeled input-output pairs training samples [31].

The most popular examples of Supervised Learning models are the Classification and Regression problems, widely used in a variety of applications.

The Classification problem computes the class, or label, to which new data belongs, through an inferred function built in the training phase.

Classification can be of two types: Binary, represented in image 2.15 [32] where a new data point is either part of stars or squares class, and Multiclass, where new data points can be classified into one of several available classes. Classification can be used for linear and non-linear separable datasets.

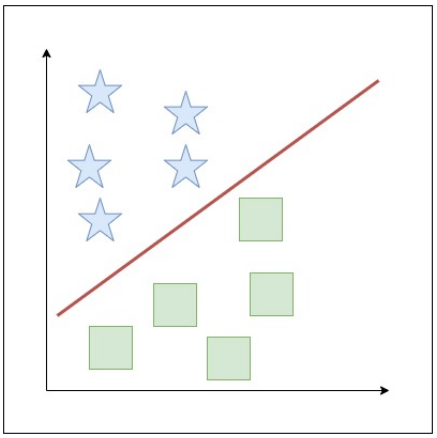


Figure 2.15: Machine Learning Classification problem.

The Regression problem tries to compute the relationship, a continuous value, between the independent and the dependent variables.

The model draws a line intercepting the data, as represented in image 2.16 [33], which has the least mean deviation between every point and the line.

Regression can be applied to a system with one or multiple independent variables, which are called Simple and Multiple Regression, respectively.

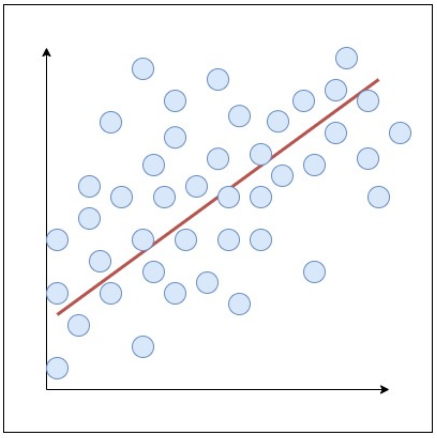


Figure 2.16: Machine Learning Regression problem.

Similar to the Classification problem, Regression can also be implemented with linear and non-linear separable datasets.

2.6.4 Unsupervised Learning

Unsupervised Learning models do not use labeled data, instead, they search for patterns within the dataset, aiming for the creation of clusters [34], represented in figure 2.17 [35]. A cluster is

based on the concept that data points belonging to the same class have similar attributes.

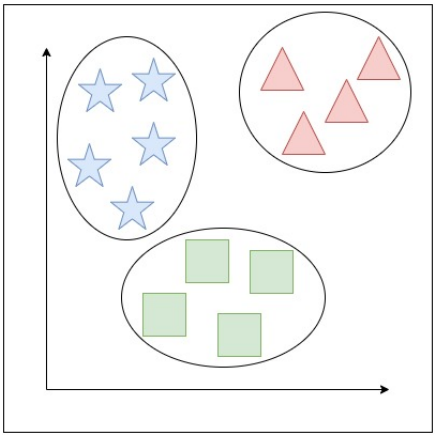


Figure 2.17: Machine Learning Clustering problem.

Clustering is used for data mining tasks and statistical data analysis in several fields, such as computer graphics, data compression and pattern recognition.

2.6.5 Computer Vision

Computer Vision is an interdisciplinary area targeting the acquisition, processing and analysis of digital images and video, to gain a high-level understanding of the surrounding environment.

Computer Vision algorithms seek to automate vision-aided applications, naturally done by the human eye. The most popular example is the Object Detection problem, where instances of the target object are localized in input digital images or video feed.

Vision-aided applications are developed based on two types of algorithms: Feature Descriptors, usually the developer's choice for resource-limited systems, and Deep Learning, which has been harvesting interest in recent years with results comparable or even surpassing human performance.

2.7 Histogram of Oriented Gradients

The HOG is a feature descriptor algorithm implemented for Computer Vision applications, targeting object detection in images and video streaming. [36] The HOG algorithm is based on the concept that object edges have higher pixel intensities than the surrounding areas. The descriptor computes the image gradients which give information about the pixel intensity derivation and

its orientation within the image. The algorithm divides the image into smaller subsets, named blocks, providing more fine-grained detail.

The advantage of implementing the HOG descriptor for object detection applications is that operating on a block-level instead of on the entire image the model becomes invariant to geometric and lightness variations since the general edges of the object will remain constant provided that the object orientation does not change drastically.

The most popular use case for HOG implementations has been pedestrian detection, where the descriptor present a high successful detection rate.

2.7.1 Image Gradients

The Gradient Calculation stage produces the image gradients, magnitude and angle. The magnitude gradient represents the intensity of the horizontal and vertical derivatives, which is relevant for the edge detection technique. The Magnitude allows the algorithm to perceive great alterations on the pixel intensity, i.e, object edges, while the Angle gradient represents the direction of the magnitude. The Gradient Calculation stage is composed, as it is represented in fig:sd4, with 2 major blocks: Control and Data path. The Control path manages the stage state machine implementation and generates the synchronization signals, HSYNC and VSYNC, that control the execution of the components belonging to the Data path block.

Image gradients represent the directional change of the intensity values within the image and are used in a variety of image processing applications, in this Dissertation's theme they are used for edge detection purposes. The gradients calculation is done by computing the horizontal and vertical derivatives of the pixels contained within the image.

The derivatives, commonly called G_x and G_y, for the horizontal and vertical directions respectively, are computed through convolution of the image with the kernels represented in (2.8). The $kernel_x$ and $kernel_y$, used for the calculation of G_x and G_y respectively, process the input pixel data as represented in the figure 2.18. [37]

For computation efficiency is it common that each direction is computed sequentially, first G_x is calculated then G_y or vice-versa.

$$kernel_x = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$
$$kernel_y = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \tag{2.8}$$

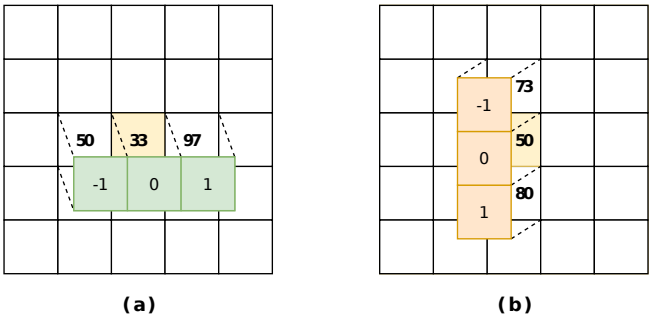


Figure 2.18: The Convolution is applied in the horizontal, (a), and vertical directions, (b) of the image. The target pixel, represented in yellow, is centered with the convolution kernel and the neighbors are subtracted, calculating the derivative.

The Magnitude gradient, which represents the change in intensity within the image, is computed through the expression represented in (2.9).

$$Magnitude = \sqrt{G_x^2 + G_y^2} \tag{2.9}$$

The Angle gradient, which represents the direction of the highest change in intensity, is calculated by using the trigonometric function tangent of the division result between the derivatives, as it is represented in equation (2.10).

$$Angle = \arctan\left(\frac{G_y}{G_x}\right) \tag{2.10}$$

The Angle gradient is used for the histogram creation, to store the magnitude value in the matching angle range, known as bin.

2.8 Support Vectors Machine

The Support Vector Machine (SVM) is a classification algorithm that maps data into multiple high-dimensional planes, commonly named hyperplanes, through a kernel function. SVM uses the training samples to compute the hyperplanes which have the widest gap between data of the different classes since a wider gap translates to better detection [38].

In classification, the SVM maps new data into the high-dimensional space and depending on which side of the gap it is mapped to a class is attributed.

The SVM is a binary classifier, i.e., can classify new data as the target class or not. Although several SVM classifiers can be implemented for a multiclass classification use case, where each classifier handles one class. The SVM algorithm is designed to enable efficient linear and non-linear classification, by changing the kernel function used to map the data to the high-dimensional space. SVM, combined with HOG features, has been a popular case of study in object detection systems, producing interesting results, both in latency and accuracy. SVM classifiers are also used in some deep learning models, in latter layers where the classification is performed.

2.8.1 Support Vectors

SVM separates points from different classes through the widest gap possible, as mentioned in previous sections, to provide the best detection achievable, depicted in figure 2.19.[39]

To understand the concept of support vector the optimization problem used for calculating the widest gap between the planes of each class must be addressed.

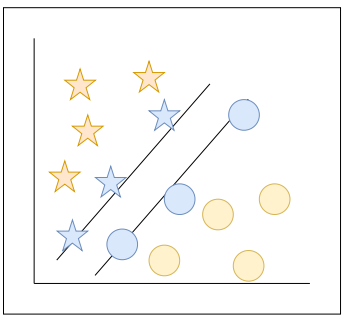


Figure 2.19: The graph represents the distribution of two different classes of data, balls and stars. The blue points represent the support vectors, mapped the closest to the separating gap.

The support vector is the denomination that is given to the points closer to the gap, which has a bigger impact on the classification of new data points.

With this approach, it is avoided the use of data points that will have little, or almost no impact, on the classification increasing the efficiency of the model, since the number of computations is reduced and the memory necessary for storing the weights is decreased as well.

2.8.2 Linear Classification

Linear classification performs the classification through a linear combination between the trained coefficients and the feature vector extracted, in the previous computations. A linear combination is an expression where each vector's values are multiplied by constants and added to form a single result value. One popular example often talked about in the early years of school, is the $y = mx + b$. The linear kernel, which transforms the SVM in a linear classifier, was chosen for this Dissertation due to its simplicity of implementation, lower latency and less training time while producing results comparable to non-linear [40]. The Linear SVM is a variation of $y = mx + b$, where m is the feature vector, containing concatenated block histograms of the entire image, and x is the support vectors calculated in the training phase. The b parameter is often called bias and is the threshold for a new point to be classified as belonging to the target class. So the $mx + b$ turns into the expression (2.11).

$$\sum_{i=1}^{N} w_i * x_i + b \tag{2.11}$$

The result is classified based on the expression (2.12).

$$\begin{cases} 1, & \sum_{i=0}^{N} w_i * x_i \geq b \\ \\ 0, & \sum_{i=0}^{N} w_i * x_i < b \end{cases} \tag{2.12}$$

2.9 Hardware Acceleration

Hardware Acceleration is the use of a hardware components to perform demanding computations, more efficiently than the general-purpose CPU. Mission-critical applications resort to RTOS to ensure that fewer latency-related uncertainties are present. For more complex systems using an RTOS might not suffice to meet the timing constraints, so often most computationally demanding tasks are implemented through hardware acceleration. Hardware acceleration increases the system's parallelism, since the costly part of the algorithm is run in the hardware component, freeing the CPU to execute other tasks and increasing the system's throughput, and determinism because for a set of inputs it is known the respective output and its inherent static latency.

The most popular examples of hardware acceleration in modern computer systems are the GPU. The GPU is designed to execute image processing operations by several parallel hardware threads, decreasing the latency and the CPU load.

In Embedded Systems the GPU is not suitable for the implementation, due to the size and power consumption constraints, instead of other components like Single Instruction Multiple Data (SIMD) are used to execute one of several available operations on a block of data.

Apart from the several available chips for integration in a computer system for parallel execution of specific operations, there is also the possibility of the development of hardware accelerators through Hardware Description Languages (HDL). HDL components can be implemented as Application Specific Integrated Circuit (ASIC) and integrated into any desired system. An HDL module can also be flashed into an FPGA chip, which allows reconfiguration of the circuit it emulates.

2.9.1 Hardware Descriptor Languages

HDL are computer languages dedicated to modeling the structure and behavior of digital logic circuits, as well as for precise simulation of already developed digital components. Hardware development can be done within four levels of abstraction:

- System Level - The component's functional description is outlined through HDL constructs. Some blocks of logic are developed purely for simulation and verification purposes and are not synthesizable.

- RTL - The design is divided into combinational and sequential logic, controlled by the system clock. RTL description is fully synthesizable and can be used for ASIC manufacturing.
- Gate Level - The design is represented as a netlist of logic gates and storage elements. Gate Level description can be used for programming a Programmable Logic Device (PLD).
- Transistor Level - The netlist is placed in different cells of the target technology and connections are routed. After this process is finished, a layout verification is executed to assure that the circuit is ready to be implemented. This level is dependent on the underlying technology, since the route and placement are done accordingly to the cell technology used in the production of the PLD, and so it is normal that the vendor offers the respective tools for this process.

HDL and software programming languages paradigms are different, due to the nature of the hardware where the notion of time must be present and concurrency is applied to all processes. HDL modules are systems that constantly process a set of inputs to generate the respective set of outputs, within a static latency. HDL modules can be implemented as consisting of Control and Data Path blocks, graphically represented in figure 2.20. The Control Path is responsible for generating the control signals, which manipulate the behavior of the module, i.e., how the input data is processed by the component. The Data Path includes the functional logic of the module, i.e, the logic responsible for processing the data and generating the respective outputs.

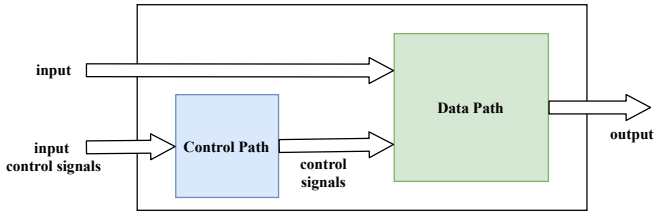


Figure 2.20: Block diagram of a general HDL module, representing the interaction between the Control and Data path.

The modules developed can be tested by creating a testbench, where the target module is instantiated and the stimuli, as well as the clock signal, are generated through non-synthesizable HDL logic. Testbenches can be as simple as stimulating the Design under Test so that the behavior can be analyzed in the simulation tool, or more complex with the use of a standardized

methodology for verifying digital logic circuit designs, such as the Universal Verification Methodology represented in figure 2.21 [41], which provide a set of Application Programming Interface (API) and classes targeting the development of modular, reusable and scalable testbench.

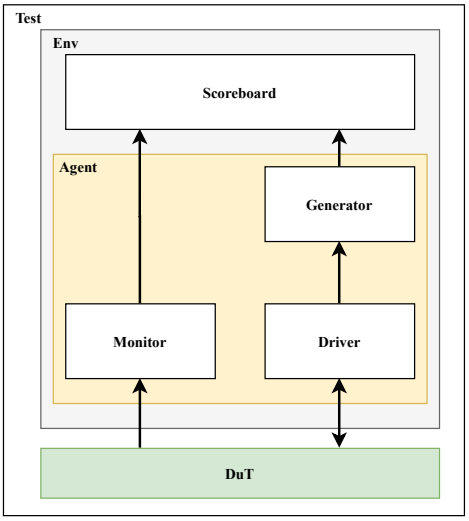


Figure 2.21: Architecture of a UVM-based testbench for validation of HDL designs.

There are two main HDL used by engineering teams for hardware development: Verilog and Very High Speed Integrated Circuits Hardware Description Language (VHDL).

VHDL is based in the ADA programming language, therefore is a rich, strongly typed and also very verbose language, tending to be lengthy and self-documenting. With VHDL it is possible to implement user-defined data types, based on the existing built-in types, as well as overloading the supported operators.

Verilog was designed for developers that were looking for a C-like syntax in the form of an HDL, therefore its constructs are very similar to the C programming language, making it weakly typed and concise.

In Verilog, it is possible to use C-like syntax pre-processing directives, due to the built-in limited pre-processor.

Its similarity, in terms of syntax, with the C language can be an advantage for a team including both software and hardware engineers, making the technical information exchange easier and more understandable.

2.9.2 FPGA

A FPGA is an integrated circuit developed to allow reconfiguration of its hardware after manufacturing. The FPGA fabric can be used to implement any digital circuit. FPGA fabric is built by configurable blocks, composed of Flip-Flops and Look-Up Table (LuT) as it is shown in the figure 2.22.

LuT components emulate the behavior of logic gates, through the creation of a truth table mapping the input to the respective outputs.

Flip-Flop components are used as storage elements for the signals used in the digital circuit.

A multiplexer is used to select the component's output, either the signal received by the LuT or the Flip-Flop.

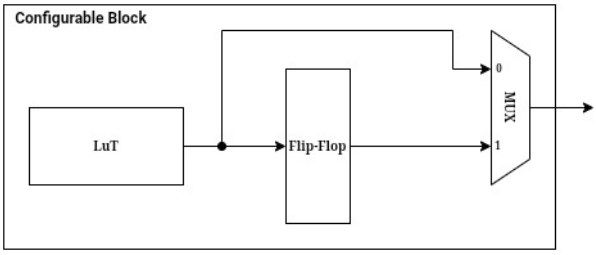


Figure 2.22: Basic configurable block present in FPGA fabric.

The increase of the FPGA fabric usage in Embedded Systems created the need for faster designs and efficient resource utilization, so modern FPGAs implement bigger configurable blocks, containing multiplexers and digital signal processing components, such as adders and multipliers, which have increased power efficiency and performance than the LuT implementation of the same operation.

The configurable blocks are denominated Configurable Logic Blocks by Xilinx and Logic Array Blocks by Altera, the two main vendors in the FPGA market.

The FPGA reconfiguration feature presents itself as an important tradeoff in system design, although it allows any digital circuit to be implemented, LuT-based circuits are slower, dissipate more power and result in larger silicon area compared with their silicon counterparts, the ASIC implementations.

2.9.2.1 System On Chip

As mentioned in the previous section, digital circuits implemented in the FPGA fabric are not as efficient as the ASIC solution, an integrated circuit designed to realize a specific task efficiently.

Implementation in FPGA chips was discarded and instead, the fabric was used for prototyping, due to the ASIC development involving high manufacturing costs and errors in the chip development resulting in the discarding of the faulty printed chip and the restart of the manufacturing process with the new fixed solution.

With FPGA prototyping mistakes in the ASIC development are greatly reduced, since the design can be tested without being printed as an integrated circuit, avoiding error detection upon the manufacturing and consequently avoiding unnecessary expenses. Modern FPGA chips integrate System On Chip (SoC) architectures, where general purposes microprocessors are hardwired in silicon, as well as other peripherals commonly used in embedded applications, as represented in the figure 2.23[42], and a portion of the silicon dedicated to the FPGA fabric, available for the implementation of user-defined hardware components.

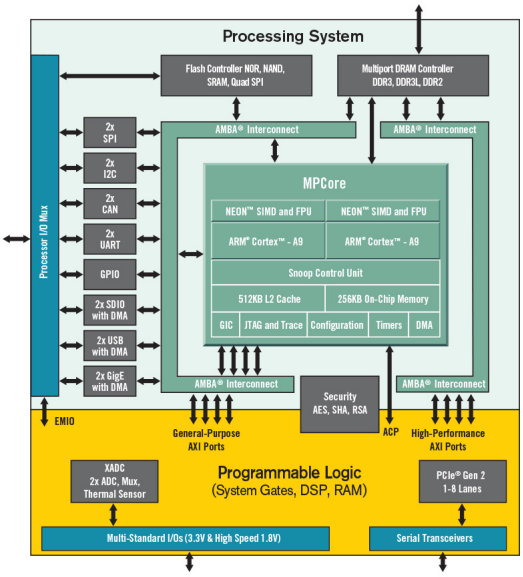


Figure 2.23: Zynq-7000 SoC's block diagram.

In this SoC architecture communication interfaces between the CPU and the programmable logic are already provided to facilitate a faster development cycle.

2.10 Hardware-Software Co-Design

Embedded systems, as explained in previous sections, are designed both in hardware and software to met the wanted performance.

The Hardware-Software Co-Design methodology is an iterative process where both software and hardware parts of the system are concurrently developed to optimize the performance, cost and power consumption of the final product.

The development flow is divided into several stages and is repeated until the system satisfies the defined metrics.

2.10.1 System Modeling

The target system is modeled, in a more flexible functional programming language, providing insights on the behavior and the components required.

System modeling is very useful for testing different algorithms without fully committing to an implementation.

The programming language used for the development of the application must be low level so that the abstraction level is not too high and the algorithm codification stays perceptible. C/C++ is commonly used since it is the developer's choice for software development in embedded systems.

2.10.2 Software Parallelization

The modeled system is implemented as a multithreaded application, where each system task is developed as a thread.

Communication and synchronism between threads are useful to understand the data exchange between the different processing elements of the system, which play a big factor in bottleneck creation.

2.10.3 Profiling

The modeled system is analyzed, for bottleneck detection, through the use of dedicated profiling software that gathers statistical information for specific events of the system.

The most common event is the CPU_CLK_UNHALTED, which counts the number of cycles the CPU is processing the target application enabling the calculation of an estimated percentage of allocated time within the target program. [43]

The profiling process is done in the host machine since the differences of the executions between on target and host are not significant.

2.10.4 Hardware Design

Hardware models are developed through the use of HDL languages, in both development and verification stages.

HDL enables the modeling of digital circuits at the register transfer level by defining the data flow between hardware registers and the logical operations performed on those signals. [44]

The HDL model is processed by a synthesis tool and transformed into a netlist, which can be used by dedicated software to produce an ASIC solution or an FPGA implementation.

2.10.5 Validation

In this stage, the developed RTL model must be validated by the Co-Simulation Methodology, where the software layers of the application and the hardware components are handled as sub-systems.

Co-Simulation accelerates substantially the development cycle of the system, by allowing testing and debugging of the design decisions before committing to the hardware platform. [45]

2.10.6 Metrics Confirmation/Implementation

This step comprises the implementation of the developed system in the target platform, having assured that all established metrics and system functionalities are verified and confirmed.

Through a set of tools like Xilinx ChipScope Pro and Serial I/O Toolkit, which integrates hardware test and measurement components, the target design can be loaded to the FPGA chip and analyzed for specific metrics. [46]

2.11 Hardware Accelerated Embedded System Simulation

Simulation is a big part of hardware development since it is used to validate the behavior of the system and helps to find bugs before implementation.

Modern electronic systems usually are built as tightly coupled, i.e, hardware and software are linked together and the workload is shared between the different components.

In a tightly coupled system, all elements must be tested, since the workload is shared if one component is not working as designed it will affect the entire system.

There are two different kinds of simulations in a Co-Simulation development cycle: instruction accurate, commonly known as a software simulation, and cycle-accurate simulation, also known as RTL simulation.

Software simulators, like QEMU [47], can emulate the target's processor through dynamic binary translation and allow the simulation of user-level applications compiled for the target architecture. QEMU emulates, functionally, all hardware components of the target's processor, allowing the full software stack simulation. This type of simulation enables the testing and validation of the software application, as well as the OS components [48].

The RTL simulators provide a cycle-accurate simulation of the RTL model's behavior, which for more complex models can be a slow process due to each simulation step simulating the state of every logic element in the model.

2.11.1 Full System RTL Simulation

The Full System RTL Simulation, graphically represented in figure 2.24[48], consists of simulating the entire system in the RTL simulator, including both the software application and the RTL model.

This approach is highly ineffective for systems built with several software layers, as is the case for applications running on Operating Systems since the RTL simulator analyzes the state of all logic gates and registers in every integration step.

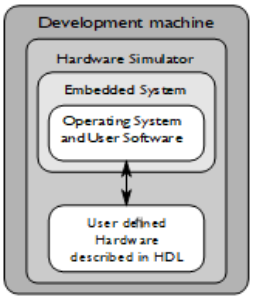


Figure 2.24: Full system RTL simulation diagram.

This approach has extremely high simulation times, due to software emulation in RTL simulators being greatly slower.

2.11.2 RTL Simulation with Host Software

In RTL Simulation with Host Software, the software application is run directly in the host machine and the RTL model is run in the RTL simulator, as it is represented in the diagram of the figure 2.25 [48], solving the Full System RTL Simulation main problem.

Communication interfaces between software and the hardware components must be addressed through the development of API to emulate system bus transactions.

The RTL model must also be altered to be added to an interface that emulates software accesses.

The drawback of this approach is that OS components, such as device drivers, are not validated.

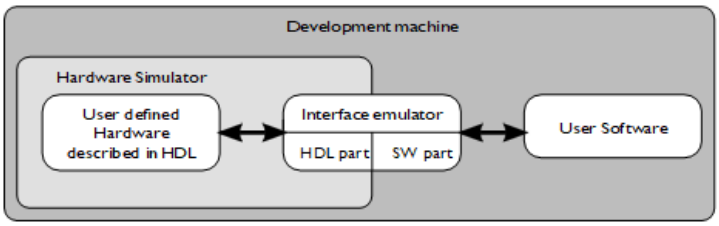


Figure 2.25: RTL simulation with host software diagram.

2.11.3 RTL-Software Co-Simulation

RTL-Software Co-Simulation approach, represented in figure 2.26[48], implements the entire software stack for the target platform, providing near-native performance, while HDL models are simulated through the RTL simulator.

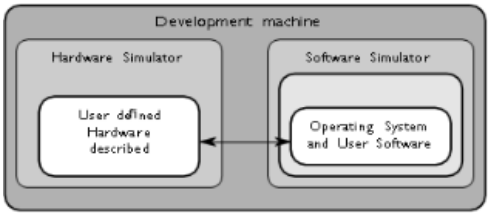


Figure 2.26: RTL-Software co-simulation diagram.

This approach favors OS elements validation, enabling both the custom hardware and their respective device drivers to be validated.

The drawback of this approach is that due to different simulator usage, synchronization may be difficult to implement and can be time-wasting to develop.

2.11.4 Full System Software Simulation

In the Full System Software Simulation approach the entire system is run as software, with the hardware components being functionally emulated and integrated into the software simulator, as represented in the figure 2.27[48].

Although it is ineffective to maintain one HDL for hardware synthesis and one for functional simulation, it can be a very useful feature for design experimentation early in the project, without committing to a model design right away.

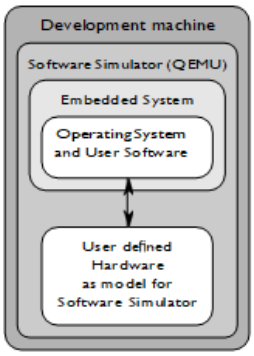


Figure 2.27: Full system software simulation diagram.

This approach enables validation and device driver development before an actual HDL implementation, which increases the design flexibility and also allows concurrent development between software and hardware design teams.

Chapter 3

System Design

In this chapter the HARVA System's architecture is described in greater detail, as well as every component and their respective role.

The chapter is divided between the Feature Extraction and the Classification subcomponents, providing the reader the required information to understand the behavior of the HARVA system.

3.1 Detection System

A Detection System is a system designed to acknowledge the presence of specific objects, or shapes, in a video stream, or still frame, without human supervision (see 2.6.5).

These systems have a wide range of applications, from Security systems that detect suspicious movement in a secured area to Autonomous Driving where pedestrians are detected to avoid collision and ensure human safety. [49]

A Detection System is composed of an Acquisition and Processing subsystems, as illustrated in figure 3.1. The Acquisition handles the frame capture, through dedicated hardware, as well as applying the required pre-processing routines. Once the frame is acquired, and processed, it is transmitted to the Processing subsystem, which implements the Feature Extraction and the Classification routines.

The Feature Extraction implements an algorithm that extracts relevant information, commonly addressed as features (see 2.6.1), from the image, while the Classification routine uses the extracted features and the classification vector, computed in the training routines, to predict if the target object is present in the input frame.

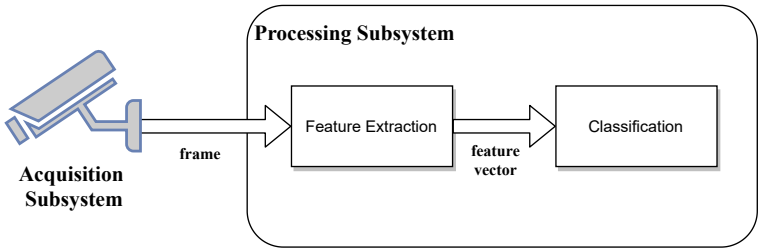


Figure 3.1: General architecture for a Detection System. The system is mainly divided between the Acquisition and the Processing of the surrounding environment's data.

The Object Detection routines can be implemented through Classic Machine Learning or Deep learning algorithms.

Classic Machine Learning (see 2.6) models implement feature vectors (see 2.6.2), algorithms divided in several steps which do not require any training routines. There are multiple Feature Descriptors available in several open-source libraries, that suit different applications in the Object Detection area such as the Haar-Cascades, that although developed for face recognition problems can be adapted for recognition of other patterns in images, the Scale-Invariant Feature Transform(SIFT), designed for identification of specific objects in an image, and the HOG descriptor, introduced as a feature extractor for pedestrian detection but adaptable to detection of other shapes and objects.

Deep Learning models have gained great importance in recent years in computer vision problems, due to the complex features produced, the high detection rates and the robustness of the algorithms. [50]

The biggest drawbacks of Deep Learning models are the time-consuming model training routines and the great memory footprint, required to handle the massive amount of data operated within the model, which is not suitable for more constrained systems.

Models like the You Only Look Once (YOLO) have decreased considerably the latency of the detection process, reaching Classic Machine Learning algorithms' latency or even surpassing.

3.2 HARVA System

The HARVA is an embedded system designed to detect human beings in images and video streaming. Upon the detection of humans in the input data feed, the system must apply a filter to ensure the anonymity of the person.

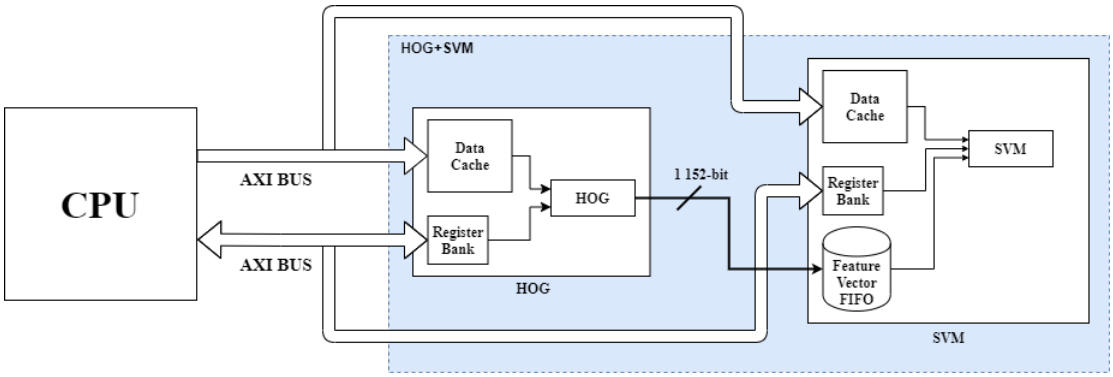


Figure 3.2: High-Level Block Diagram view of the HARVA system.

The HARVA implements the HOG+SVM Object Detection algorithm as co-processor, depicted in 3.2. The HOG+SVM is an Edge Detection algorithm, acknowledging the presence of the target object by processing the pixel intensity variation across the image, as represented in figure 3.3.

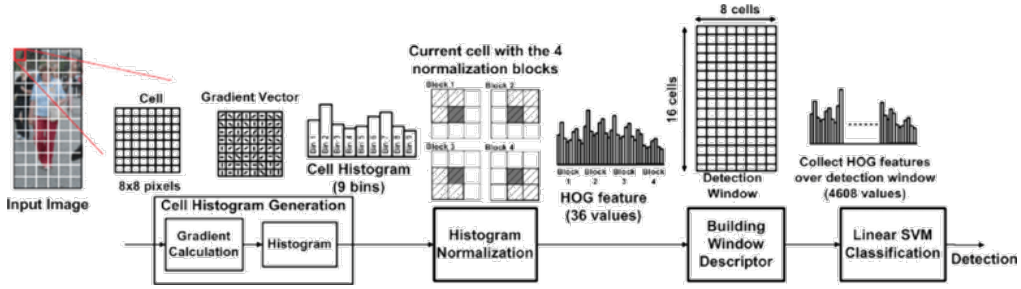


Figure 3.3: HARVA system divides the input image within several subsets, computing for each the intensity variation in a form of an histogram.

Transformation to grayscale, illustrated in figure 3.4, benefits the system providing reduced memory footprint and hardware, since the image is represented only within one color channel.

Figure 3.4: Grayscale transformation routine for Image Processing applications.

The HARVA system processes the grayscale image, splitting it into multiple Detecting Windows, i.e., pixel subsets. This method enables the search of the target object in different locations across the image, as represented in figure 3.5.

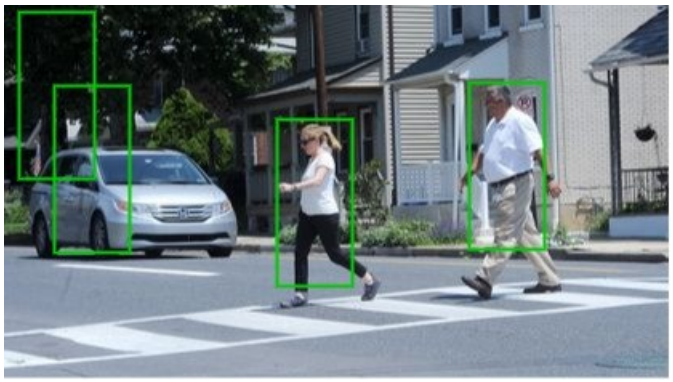


Figure 3.5: Detecting Window algorithm for multiple locations detection.

The Detecting Windows are also extracted at several scales of the grayscale image, by successively downsampling the image until it matches a predefined size. This process, commonly known as Image Pyramid, shown in figure 3.6, enables the detection of the target object in different sizes.

Figure 3.6: Image Pyramid algorithm for multiple scale detection.

These preprocessing routines, implemented as software routines, extract multiple Detecting Windows and transmit them to the HOG+SVM co-processor, which creates the respective Feature Vector and maps it into pre-trained hyperplanes through the SVM kernel. Depending on which side of the gap the point is positioned it is classified as Pedestrian or Not Pedestrian.

The Detecting Windows classified as containing the target object return their coordinates to the main application, depicted in figure 3.7.

Figure 3.7: Multiple Pedestrian Detection.

The main application after retrieving information from the co-processor, about the locations of the Pedestrian across the image, applies a blurring filter ensuring the anonymity of the detected pedestrians as shown in figure 3.8.

Figure 3.8: Bluring filter for anonymization of target object in Object Detection algorithms.

The blurring filter is developed as a software routine, applying the Gaussian function (see 2.5.5) to the location detected by the HOG+SVM co-processor.

The combination of the HOG Feature Descriptor with the SVM Classification has been proved to achieve good performance and became a de-facto standard across many visual perception tasks. The popularization of the HOG+SVM model is largely attributed to the step-change in performance they brought to pedestrian detection [51].

The HARVA system was based in the FPGA-based Real-Time Pedestrian Detection on High-Resolution Images [52], mainly lending the HOG component's architecture and the Bin Assignment algorithm, which removes the requirement for the arctan trignometric operation.

The HARVA, differently from the FPGA-based Real-Time Pedestrian Detection on High-Resolution Images, implements the preprocessing routines in software, enabling a fast and easy implementation through open-source libraries. Another key difference is that the HARVA does not implement

the Gauss Filter, avoiding the implementation of an 1024 byte Memory, dedicated to the storage of Gauss data, as well as the latency of fetching the Gauss coefficients from the HOG Data Cache.

The main focus of the HARVA system is to provide the highest possible configurability of the underlying components, allowing the user to define the number of cores implemented by each hardware module, as well as runtime configuration of specific parameters. Other key difference between the 2 system mentioned is the fact that the HARVA supports compression of data, decreasing the memory footprint as well as the transmission latency.

3.2.1 HOG

The HOG is a Classic Machine Learning Feature Descriptor algorithm, designed to detect object edges within the input image (see 2.7).

The main reason for the HOG descriptor to be used in the HARVA system is the fact that it operates on local cells, which results in the process sustaining invariance to geometric and photometric transformations, as long as the object maintains a somewhat upright orientation. This property of the HOG Descriptor makes it is highly suitable for pedestrian detection and for the HARVA system. [53]

The HARVA system implements the data caching concept by connecting a Data Cache to the HOG component, as it can be seen in the 3.9. The HOG component's Cache is used for storing pixel data, that will be use in future computations, avoiding the overhead of exchanging data with the system's main memory (see 2.2.4).The storage element can be dimensioned for different sizes, balancing the manufacturing cost with the system throughput, since higher memory capacity translates in better performance and, consequently, higher production cost.

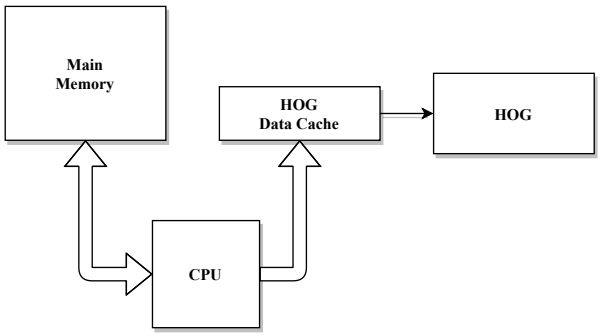


Figure 3.9: HOG's components 2-level memory hierarchy.

The HOG component's memory is divided in Control Register and Pixel Data sections, as illustrated in figure 3.10. The Control Registers enable the user to manipulate the component's behavior, while the Pixel Data section is used to store image data transmitted from the system's Main Memory.

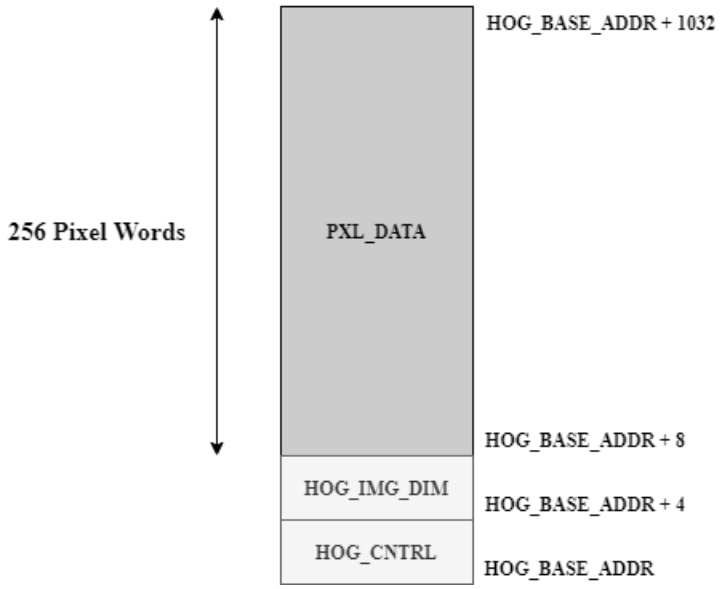


Figure 3.10: HOG's components Data Cache memory layout. The memory is divided in several section for the different required accesses.

The default value of the Pixel Data section is 256 words, which matches the Advanced eXtensible Interface (AXI) transmission size with the best throughput. Multiples of 256 can be used to define the section size, as long as the target technology provides the number of cells required.

The HOG CONTROL register controls the component's behavior, enabling the trigger of the processing of a new image, or halting of the current one through manipulation of the $HogEN$ bit. Runtime information, relative to the status of the configuration of the Gauss Weighting and Quantization processes, is available by accessing the $GaussOK$ and $QuantStepOK$ bits, respectively. The HOG CONTROL register also contains information about the need to update the support vectors, present in the component's local data cache, by accessing the $HogCacheMiss$ bit. The bit information of the $HOGCONTROL$ register is available in the figure 3.11.

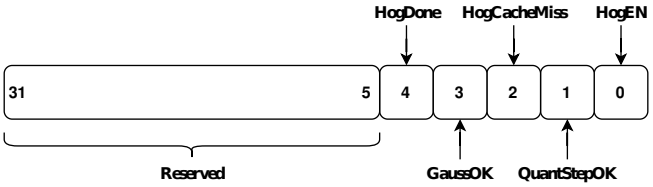


Figure 3.11: HOG CONTROL register bit description.

The IMG_DIM register is used for the configuration of the image dimensions, so that the VSYNC and HSYNC control signals are generated correctly throughout the HOG component. The bit information of the IMG_DIM register is represented in the figure 3.12.

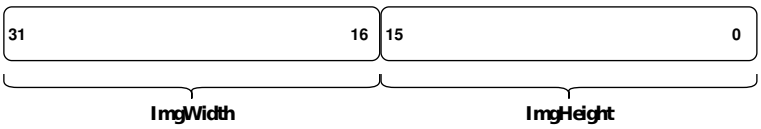


Figure 3.12: IMG_DIM register bit description.

3.2.2 SVM

The SVM is a classification algorithm, offering a good performance for a wide variety of linear and non-linear separable features (see 2.8). The Linear Kernel SVM is often use for object detection problems, where the latency is critical.

The SVM component, similarly to the HOG, implements a Data Cache for storing the SVM support vectors used in the classification, decreasing the overall latency of memory access within the module. A First In First Out (FIFO) memory is also implemented, as a buffering element, to store blocks from the HOG feature vector. The FIFO enables the HOG and SVM components to run at different frequencies, as well as the fetch of new SVM support vectors data without losing feature vector data.

The SVM component's memory is sectioned into Support Vector Data, Feature Vector Data and Control Registers. The Control Registers support configuration and manipulation of the component by the Main application.

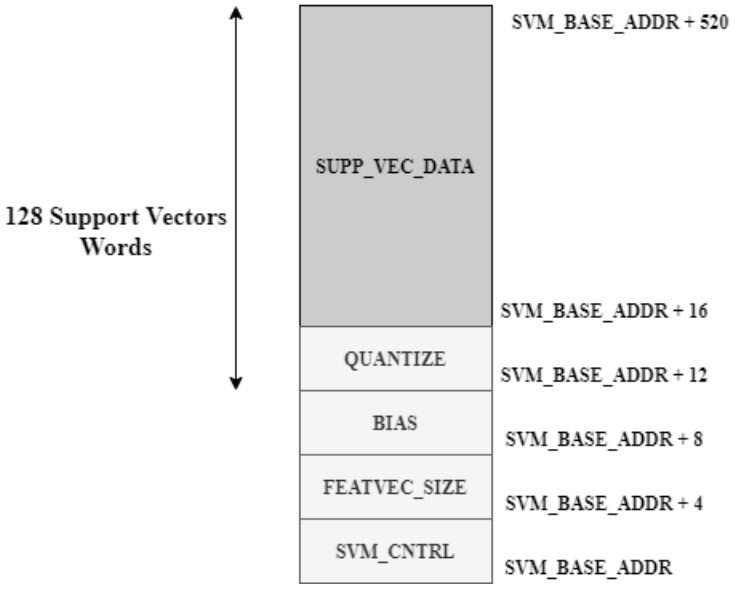


Figure 3.13: HOG's components Data Cache memory layout. The memory is divided in several section to support the different required accesses.

The Support Vector Data section stores SVM coefficients, used for the classification of the Feature Vector. The default size of the section is set to 128 words, although it can also be set to 256.

The Feature Vector Data supports the storage of the HOG component produced data. The section size can be configured to higher values, with the default value being set to 144 words, which translates to 4 block histograms.

In order to trigger the classification of a new image, or halt the current execution, the software must manipulate the $SvmEN$ bit of the SVM CONTROL register. The $SVMCONTROL$ register also enables the verification of the validity of the quantization configuration, by accessing the $QuantStepOK$ bit and if the transmission of new support vectors i required , $SvmCacheMiss$ bit. The $SvmDone$ bit signalizes the end of the SVM component's execution for the input image. The bit information, and disposition, of the $SVMCONTROL$ register is available in the figure 3.14.

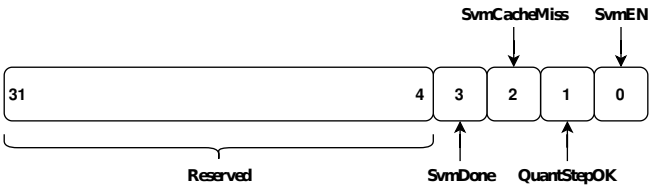


Figure 3.14: SVM CONTROL register bit description.

The configuration of the Quantization step, required for the decompression of the values, is done by accessing the $QUANTIZE$ register and set the respective value. Similarly the $BIAS$ and the $FEATVEC_SIZE$ registers are used to set the classification threshold and number of cycles, respectively.

3.3 HOG Features Extraction

The HOG component is based on Histograms of Oriented Gradients for Human Detection paper by Navneet Dalal and Bill Triggs [36], which divides the input image into small pixel subsets, commonly denominated as blocks, with height and width dimensions set to 16 pixels. Each block is processed individually and the respective histogram is build, representing the intensity distribution within the image. The histogram are all concatenated to form the image's feature vector.

The HOG component's architecture, depicted in figure 3.15, is based on the pipeline concept (see 2.2.3), and divides the algorithm in 4 stages: Convolution, Gradient Calculation, Histogram and Normalization.

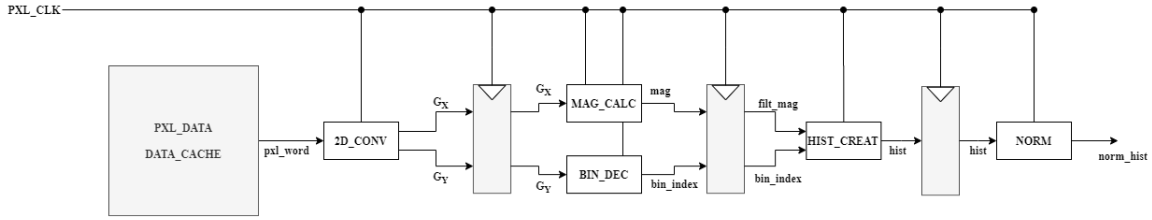


Figure 3.15: HOG component's block diagram. The component is designed as a 4-stage pipeline, aiming an increased data throughput.

The HOG component fetches data from the Data Cache and, when required, formats data for the convolution operation. The convolution component requires, for each input, a composition of 3 vertically sequential Pixel Words, i.e., 32-bit word comprising 4 horizontally sequential pixels. When the Pixel Words are positioned in the image edges, the HOG component interpolates the missing pixels, represented in figure , and formats the Pixel Words accordingly to ensure the convolution behaves as expected.

Concurrently to fetching pixel data from the Data Cache, the HOG component generates the HSYNC and VSYNC signals (see 2.5.1), which are used to control the execution of the pipeline stages. The VSYNC signal signalizes the end of the processing of the input image, concluding

the execution of the stages of the pipeline and transitioning the component into a ready state, waiting for a new request.

3.3.1 Convolution Stage

The convolution stage computes four concurrent vertical and horizontal derivatives values from pixel data of the input image. The vertical derivative, G_y, is produced for each new input, since it is a subtraction of the pixels values from the vertical neighbor words, while the horizontal derivative, G_x, needs a second input, due to the right neighbor of the last element of a Pixel word being the first value from the next target word.

The $2D\_CONV$, which implements the convolution logic, is modeled through the state machine of the image 3.16. The $VCONV$ state executes the vertical derivative and is executed every time a new valid input is available. The $HCONV$ state executes every two cycles, when the required data is available to compute the 4 values of G_x.

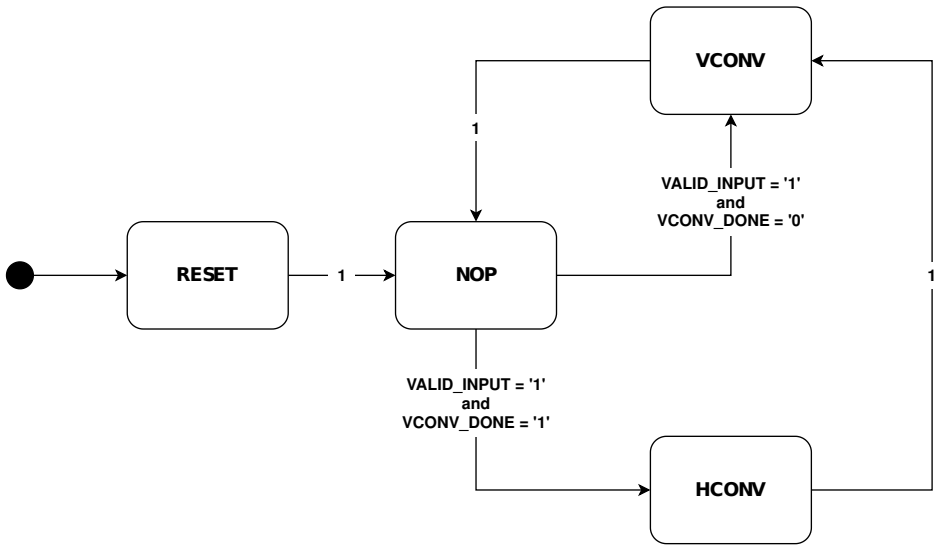


Figure 3.16: $2D\_CONV$ component's state machine.

The $2D\_CONV$ component design takes advantage of the fact that the derivatives computations are done at different cycles to use the same hardware components for both operations, as it is illustrated in figure 3.17 , making the implementation resource-efficient.

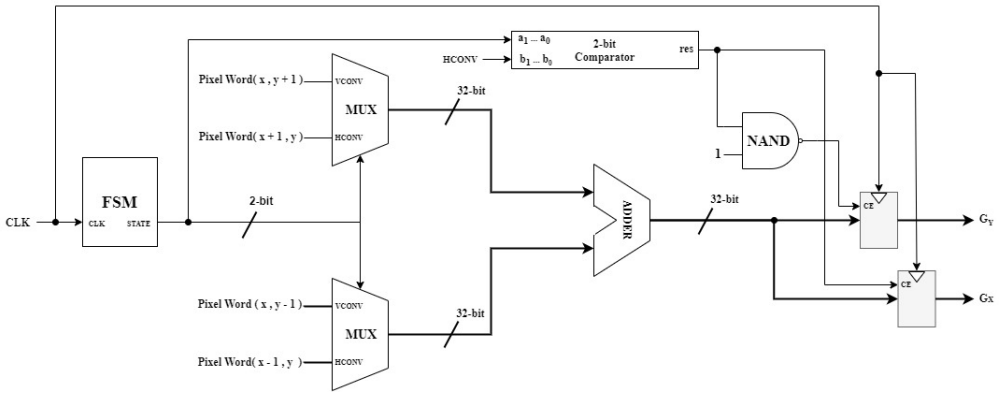


Figure 3.17: Block Diagram of the logic implemented for the computation of each value of the G_x and G_y derivatives.

Although the $2D\_CONV$ component can compute the convolution in two cycles, it requires two inputs for producing 4 valid derivative values for G_x and G_y, therefore the latency depends on the availability of the data, i.e., the number of cycles necessary for fetching the data from Cache memory.

3.3.2 Gradient Calculation Stage

The Gradient Calculation Stage computes, concurrently, the images gradients by processing the G_x and G_y values from the convolution operation. The gradients calculations are implemented by the MAG_CALC and BIN_ASSIGN components.

The Magnitude calculation (2.9) is divided into two stages, as illustrated in the block diagram in figure 3.18. The first stage computes, for each value of G_x and G_y, the power of 2 and, then, performs the addition of the results. The second stage implements the square root on the addition results.

The implementation of a complex algorithm such as the square root requires great mathematical expertise, allied with good experience in hardware design, to reach the desired latency and resource usage for the design, but most importantly it does require a considerable amount of time for the research and development of the algorithm. Taking this into consideration, and because there is no need to re-invent the wheel, a chip off-the-shelf solution was used.

The Xilinx COordinate Rotation DIgital Computer (CORDIC) Intellectual property (IP) [54] offers multiple trigonometric and vector transformations functionalities, using only addition, bitshift and table look-up elements.

Considering that the output from the CORDIC IP is formatted as a fixed-point expression, whose converting factor is given by the equation 3.1, 33-bit input signals were used. With this decision, the output will be divided into 16-bit of fraction part and 16-bit of an integer part, enabling bitshift operation for conversions between fixed-point resolutions, required for other components. The 33-bit width input signals configuration of the CORDIC has a latency of 17 clock cycles.

$$ConvertingFactor = 2^{(}\frac{signal_bitwidth - 1}{2}) \tag{3.1}$$

Between the two stages of the MAG_CALC component there is a buffering element that allows asynchronously run, i.e., both stages can process data in parallel, as soon as there are available results from the first stage and the CORDIC core is free it starts executing new data, increasing the module throughput.

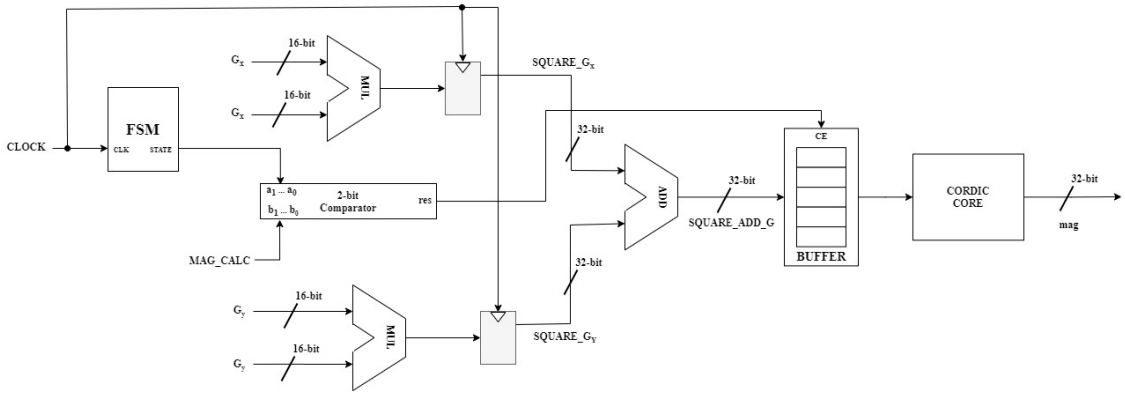


Figure 3.18: MAG_CALC block diagram, illustrating the pipeline behavior implemented for the Magnitude Gradient computation.

The MAG_CALC component is modeled through the state machine represented in image 3.19, where the MAG_PIPE state implements the two-stage logic, responsible for the Magnitude gradient computation.

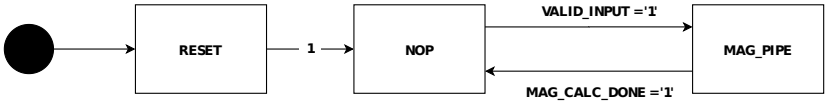


Figure 3.19: MAG_CALC component's state machine.

The Angle gradient calculation avoids the implementation of the CORDIC chip, based on a new algorithm that fuses the angle calculation with the bin assignment operation, removing the

necessity of trigonometric logic [52]. The algorithm computes the matching bin through the use of inequalities and pre-calculated values stored in look-up tables. The algorithm is based in the equation (3.2), where n is given in degrees and is ranged from $0°$ to $180°$.

$$\tan(n) \leqslant \left|\frac{G_y}{G_x}\right| < \tan(n+20) \tag{3.2}$$

To avoid divisions, which result in more complex and expensive hardware, the equation is expanded into the form of (3.3). The values of the tan are static and computed before implementation.

$$\tan(n) * \left|G_x\right| \leqslant G_y < \tan(n+20) * \left|G_x\right| \tag{3.3}$$

Analyzing the tangent unit circle, described in figure 3.21, it is visible that the bins contemplating the degrees from $0°$ to $80°$ and $100°$ to $180°$ have the same absolute values for the tangent, only inverted by the vertical axis.

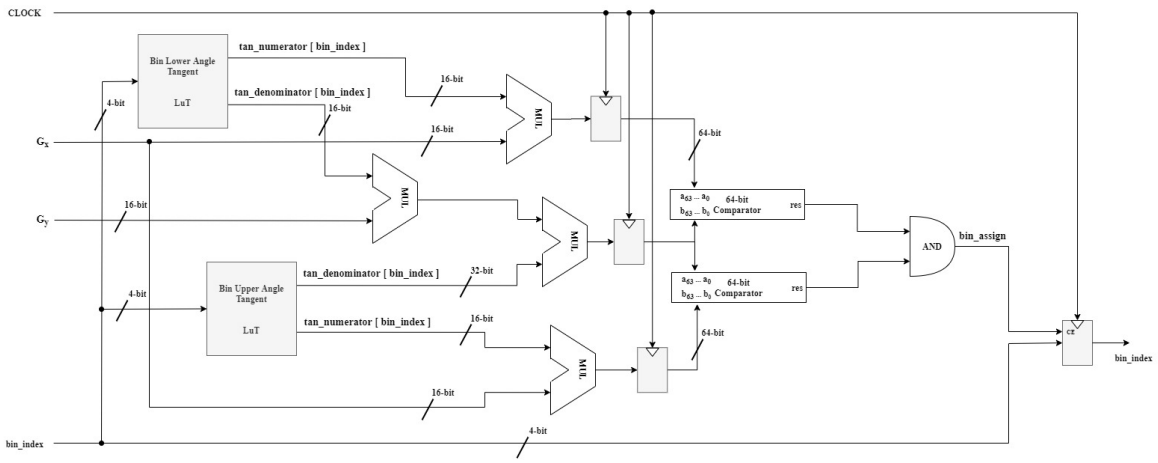


Figure 3.20: BIN_CALC component's block diagram.

This detail enables the BIN_ASSIGN component to checks two bins concurrently, i.e., a bin between $0°$ to $80°$ and its mirror bin between $100°$ to $180°$, reducing the operation latency and enabling the reuse of the same multipliers and comparators for every bin testing, as represented in the figure 3.20. [55]

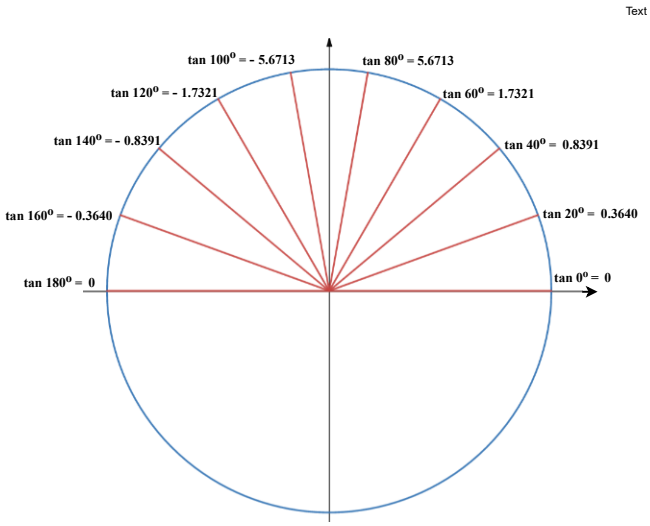


Figure 3.21: Tangent's trignometric upper half of the unit circle.

The state machine that models the BIN_CALC component is represented in the image 3.22. The TRY_BIN state implements the bin assignment algorithm, which sequentially tests the bins inequalities with the G_x and G_y values until the conditions are passed.

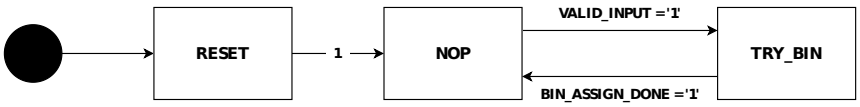


Figure 3.22: BIN_CALC component state machine.

The Gradient Calculation stage produces 8 valid values for the Magnitude gradient and the matching bins in the block histogram. The latency of the stage is bounded to the latency of the MAG_CALC component, since it is the operation with the highest latency within the stage.

3.3.3 Histogram Stage

The Histogram stage is responsible for the iteratively building of the blocks histograms from the input image. This process requires 8 values of the magnitude gradient and the matching bin indexes to be available at each input, which is processed sequentially, since multiples Magnitude values might be assigned to the same histogram bin which would cause data collision if the operation were implemented in parallel.

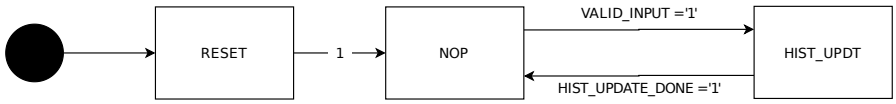


Figure 3.23: $HIST\_CREAT$ component's state machine.

The $HIST\_CREAT$ component implements the normalization logic through the state machine represented in figure 3.23. The $HIST\_UPDT$ state handles, iteratively, the update of the block histogram, by adding the Magnitude gradients present in the input with the values stored in the respective bins from previous computations, as described in the block diagram of the figure 3.24.

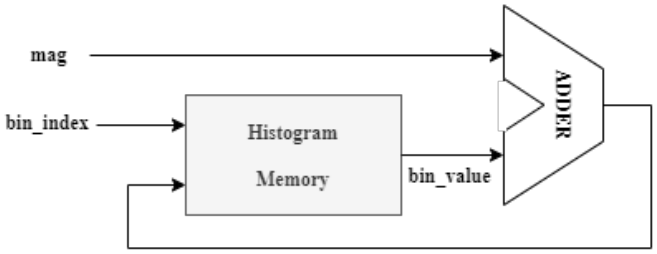


Figure 3.24: $HIST\_CREAT$ component's block diagram, illustrating the histogram update operation.

This implementation, while having the drawback of the imposed sequentiality its is beneficial in terms of resource usage, since the same adder can be used for all the computations.

The latency of the Histogram stage relies on the availability of data, since to build one block histogram the component requires 32 inputs, composed of 8 Magnitude values and the matching bins. The output is the histogram comprising 36 values, resulting from the concatenation of the histograms from the 4 cells.

3.3.4 Normalization Stage

The Block Histogram Normalization stage, represented in the figure 3.25, implements the $HIST\_NORM$ component, which applies the L1-norm (see 2.5.4)to a block histogram for an increased invariance to the contrast. The L1-norm expression, equation 2.6, is implemented in 2 stages.

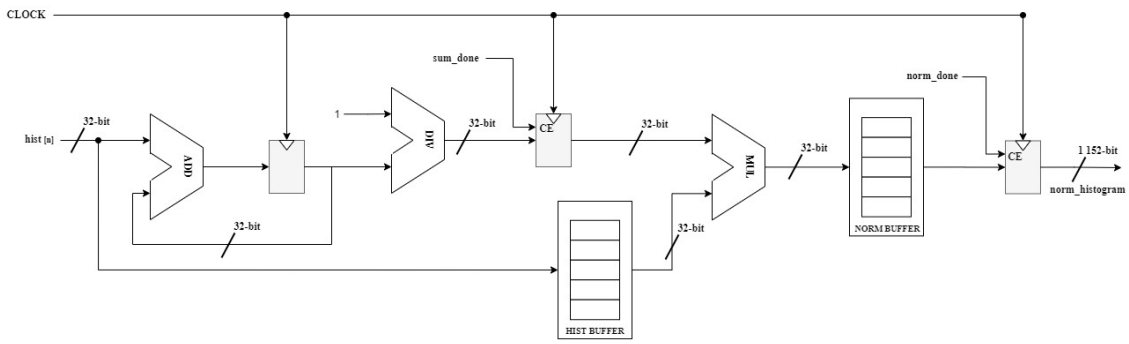


Figure 3.25: Block Histogram Normalization Stage's block diagram. The red signal symbolize the control signal generated by the Control Path to manipulate the behavior of the implemented modules. The blue signals represent the signals that impact the control signals generation. The green signals are used to represent the data manipulated within the stage.

In the first stage, the component computes the sum of all values of the histogram as well as finding the minimum value, added to avoid a division by zero situation.

To avoid division operation, which result in less efficient RTL, with greater silicon area and consequently higher manufacturing cost, the second stage transforms the division into a multiplication, which together with addition and subtraction is easily synthesized. With this approach, in the second stage, each histogram value is multiplied by the expression (3.4).

$$\sqrt{\frac{1}{\|v_1\|} + \varepsilon} \tag{3.4}$$

In the $HIST\_NORM$ component state machine, represented in the figure 3.26, the $HIST\_SUM$ state handles the cumulative sum of the histogram values and the $HISTOGRAM\_NORMA$ state multiplies each value by the inverted sum value, as illustrated in the block diagram in figure 3.25

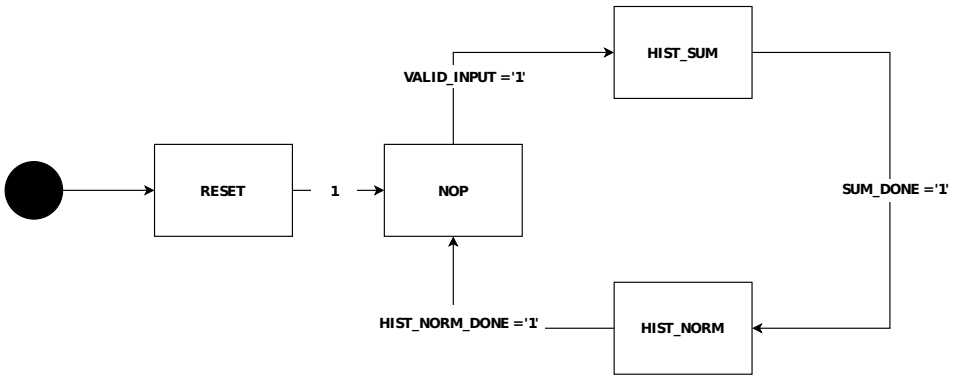


Figure 3.26: $HIST\_NORM$ component's state machine.

The Block Histogram Normalization stage receives as input a block histogram, which is a composition of 36 values. The stage processes the input to produce the normalized histogram, of the same length, which is used by the classification component in the next stage.

3.3.5 Component Parallelism

The HOG component is implemented as a pipeline, similar to in modern computer systems to take advantage of data-parallelism, by processing multiple chunks of data within the pipeline stages, as represented in the pipeline behavior in the figure 3.15, and, consequently, achieving better performance.

The HOG stages run asynchronously and concurrently, provided that there is data available in the input register bank and available space in the output register bank of the stage. This enables stages to execute at different clock frequencies, given that clock domain crossing issues are addressed correctly.

Each stage enable different configurations, where the number of elements processed in parallel at each clock cycle is set to different values, giving the user the choice to decide between an implementation that privileges the stage's performance or resource usage.

The Convolution stage computes, concurrently, 4 values of G_x and G_y, taking advantage of the fact that image data is stored as a 32-bit words. The Gradient Calculation enables the configuration of the different levels of parallelism, supporting the concurrent processing of 1,2,4 or 8 elements. The Histogram stage does not support parallelism configuration, due to the imposed sequentiality of the operation. The Normalization stage enables a wider range of levels of parallelism, supporting concurrent processing of 1, 2, 4, 6, 12, 18 and 36 values.

3.4 SVM Classification

The SVM Classification component processes the feature vector, mapping it into the hyperplanes in which the SVM model was trained. The image is classified depending on which side of the gap the feature vector is mapped onto.

The SVM is comprised of two stages that implement the classification algorithm and two storage elements for communication with the HOG component and data exchange with the system's main memory, as represented in the SVM's block diagram in figure 3.27.

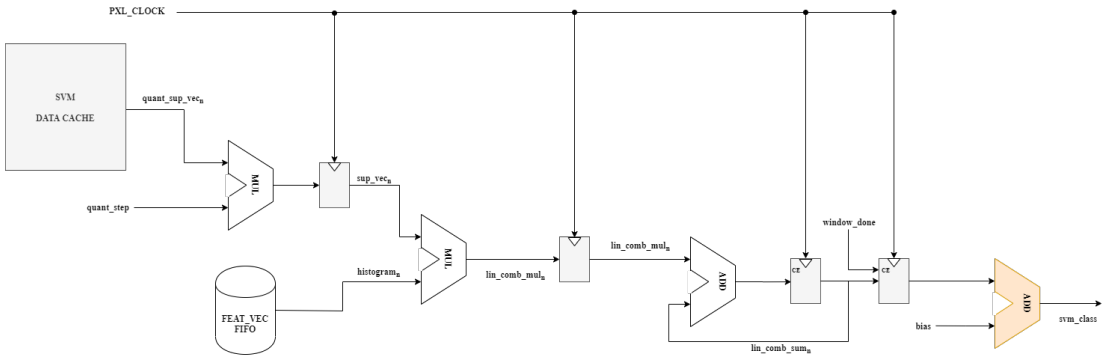


Figure 3.27: SVM component's block diagram. The red signal symbolize the control signal generated by the Control Path to manipulate the behavior of the component. The blue signals represent the signals that impact the control signals value. The green signals are used to represent the data manipulated within the SVM.

The SVM component's behavior was modeled through the state machine of the figure 3.28, where the $RESET$ state is the default state, executed on power-on or system reset, and initializes the required signals to a known value. The NOP state is used for stalling the component until new feature vector and support vectors data are available for the $QUANT$ and LIN_COMB.

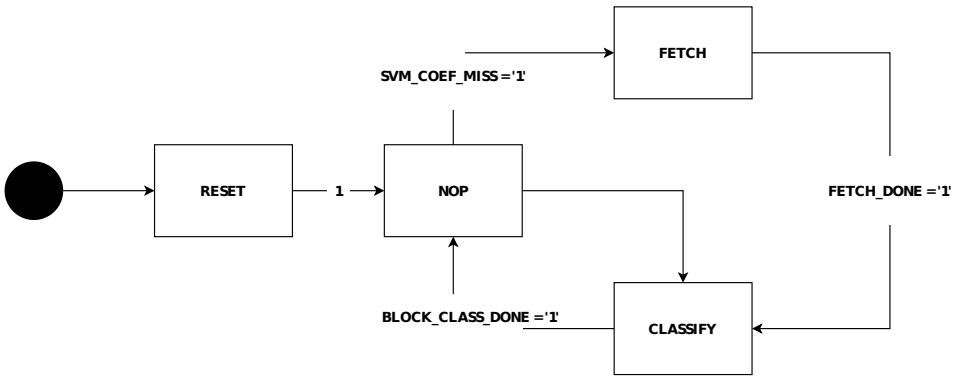


Figure 3.28: SVM component's state machine.

The $FETCH$ state triggers the transmission of the new support vectors values from the main memory to the SVM's local memory, equivalent to a cache miss in modern computer systems. The components wait until the transmission is completed before resuming the normal execution.

The $CLASSIFY$ state executes the linear combination on the support vectors and the feature vector values, iteratively and concurrently while feature data is also computed in the HOG component.

3.4.1 Quantization Stage

The Quantization stage targets an increased memory efficiency in the component by enabling the quantization (see2.3.3) of the SVM support vectors, transforming the double-precision data into smaller integers. With quantization, the support vectors required less storage capacity and their transmission from the main memory to the SVM component's local memory has a better performance (see 2.3.3).

The $QUANT$ component reverses the quantization process, decompressing the values to an approximation of the original real value since the quantization is a lossy operation. The decompressed value is then ready to be processed by the LIN_COMB component.

The $QUANT$ component uses the decompresses data by multiplication between the Quantization Step, configured by the user, and the compressed values, as illustrated in figure 3.29.

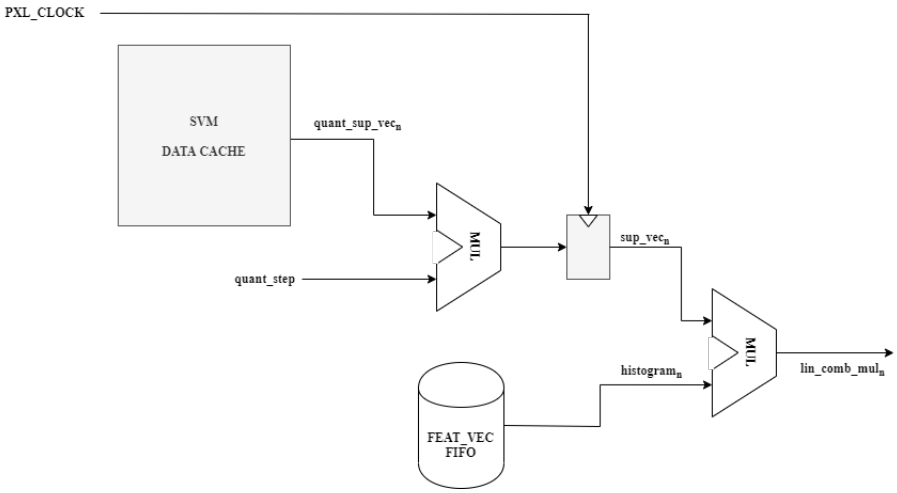


Figure 3.29: $QUANT$ component's block diagram.

The Quantization stage decompresses one Support Vector value per clock cycle. This stage is dependent on the availability of data, suspending the execution when new data is not present in the SVM component's memory.

3.4.2 Linear Combination Stage

The Linear Combination Stage handles the execution of the Linear classification, which the SVM Linear Kernel uses for the mapping of new data points into the hyperplanes.

The component, depicted in figure 3.30, classifies the feature vector, iteratively, processing the data, by multiplication between the feature vector and the Support Vectors values and then the cumulative sum of the results (see 2.8.2).

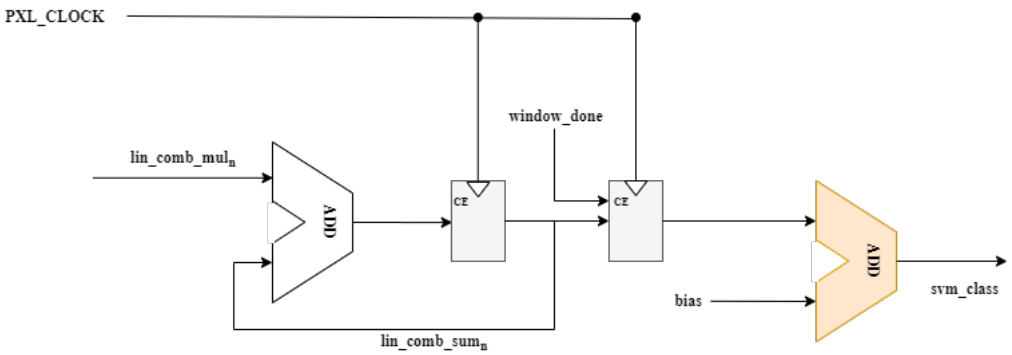


Figure 3.30: LIN_COMB component's block diagram.

The LIN_COMB component checks the VSYNC signal associated with each block histogram to acknowledge an end of image event, in which case the module adds the Bias value to compute the final classification of the image.

3.4.3 Component Parallelism

The SVM implementation emulates a two-stage pipeline, represented in the block design of the figure 3.31, processing valid data while new valid histogram blocks and support vectors values are available, decreasing the component overall latency. The first stage comprises the decompression of the support vectors, transferred from the main memory to the local data cache, and is handled by the $QUANTIZER$ component. In the second stage, the element-wise multiplication between the decompressed support vectors and the feature vector values is computed. The third and final stage handles the cumulative sum, which results in the new mapped point in the SVM hyperplanes. The LIN_COMB component implements the two later stages of the pipeline. Each stage is configurable for different levels of parallelism, enable multiple values to be processed concurrently. The pipeline implementation of the SVM component, similar to the HOG use case, increases the data parallelism and, consequently, the system throughput.

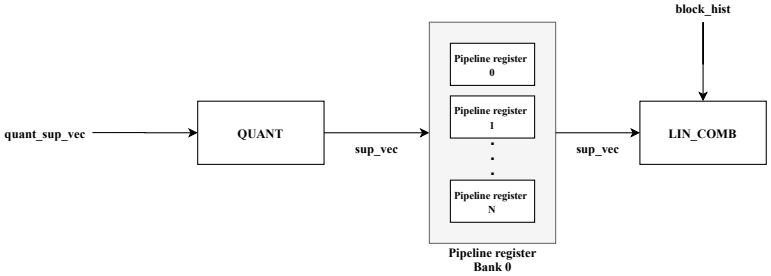


Figure 3.31: SVM component's implementation as a two-stage pipeline, targeting increased throughput by exploiting data-parallelism.

The SVM component requires the support vectors values, stored in the system's main memory, which are transmitted through the AXI protocol. This transmission is costly, latency-wise, and presents itself as a bottleneck in the component execution. Since the SVM component only executes when there is new data available from the HOG component, the transmission can be requested to happen when the component is waiting for new data, reducing the bottleneck impact.

Chapter 4

Experimental Results

This chapter explains the validation and profiling of the HARVA system, intending to test the performance of the components.

The chapter describes how the HARVA system was modeled as a software application for the comparison of performance between software and hardware implementations.The profiling is also addressed, as well as how the bottleneck were tackled from software to hardware.

4.1 Test Environment

Test Environment is a setup of software and hardware, designed to execute test sequences on a developed system. The environment can support test execution in software, hardware, or both since it must be implemented based on the target system. [56]

The Xilinx Vivado Design Suite enables the development and testing of the HARVA HDL's designs, as well as providing useful IPs a faster the AXI BUS integration. The Vivado Simulator, a HDL event-driven simulator that supports functional and timing, is used for the validation of the HARVA system, following the Xilinx development flow of figure 4.1.

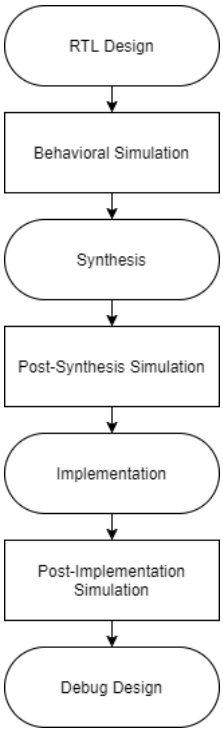


Figure 4.1: Vivado Simulator flow for HDL development.

Vivado Simulator was used until the Post-Synthesis Simulation step, verifying the design RTL as well as ensuring the design meets the functional requirements and behavior.

The HARVA's Test Environment targets the Pedestrian Detection use case, a computer vision task consisting of searching and acknowledging pedestrians in any still image or video streaming frames in computer systems. The Institut National de Recherche en Informatique et en Automatique (INRIA) dataset, which was built as part of Histograms of Oriented Gradients for Human Detection paper[57] targeting the development of a robust system for Feature Extraction to support high accuracy object/non-object decisions was used in HARVA development flow, both on hardware and on software level, to ensure the integrity of the profiling and detection results.

The INRIA dataset is one of the most popular data collection in the Pedestrian Detection community and is comprised of images from different sources, other datasets and personal digital images taken over a period of time, as shown in figure 4.2.

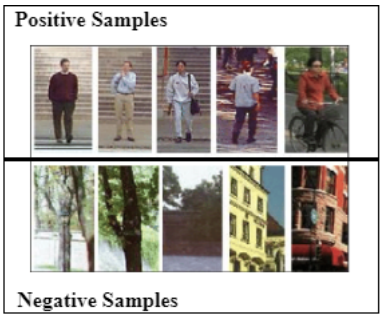


Figure 4.2: Example of INRIA dataset positive samples.

The INRIA dataset is composed of two subsets of original and normalized images. The normalized images are cropped and processed into 64x128 dimensions and are divided into positive and negative samples. The positive samples are images that are labeled as containing a pedestrian, whereas the negative samples are labeled as the opposite. The dataset composition is the default for Supervised Learning (see 2.6.3), where the model takes labeled training samples to infer the required weights, or coefficients, values.

4.1.1 Hardware

The hardware required to implement and run the HARVA tests is comprised within the ZC706 Evaluation Kit, shown in figure 4.3.[58] The ZC706 Evaluation Kit provides the hardware components necessary for the development and evaluation of any Embedded System design.

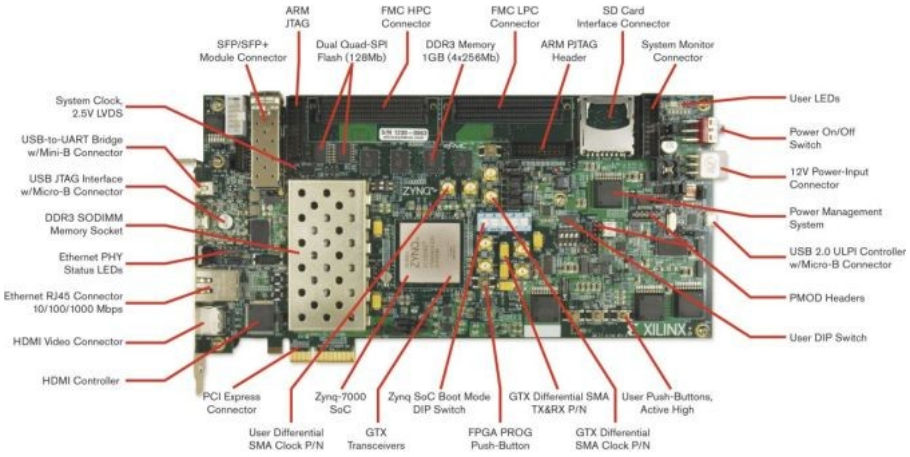


Figure 4.3: ZC706 Evaluation Board.

The ZC706 evaluation board is implemented with the Zynq-7000 XC7Z045 SoC, which combines the software programmability of an ARM-base processor with the hardware programmability of an FPGA, providing the best performance-per-watt and price in the current market. The

XC7Z045 SoC is composed of an integrated Processing System and Programmable Logic within the same die, as represented in the block diagram in figure 4.4(see 2.9.2.1).

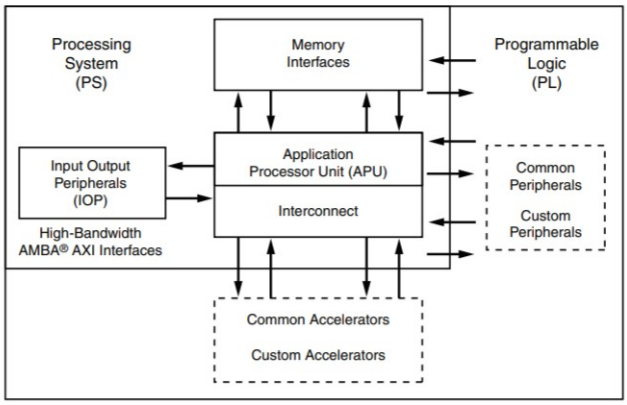


Figure 4.4: ZC706 Evaluation Board High-Level Block Diagram.

The Processing System block integrates two ARM-A9 cores, for increased multi-threading application implementation. The block also includes internal and external memory interfaces and common peripheral interfaces such as Universal Serial Bus(USB), Serial Peripheral Interface(SPI), Secure Digital Input Output(SDIO) and Inter-Integrated Circuit(I2C).

The Programmable Logic block can be used to emulate any intended hardware, which can easily be connected with the Processing System as well as the present peripherals.

The ZC706 Evaluation Board includes an advanced memory interface ideal for Image Processing application, suiting the HARVA's purpose.

4.1.2 Communication Interfaces

The HARVA Test Environment implements 2 AXI variants as communication interfaces, enabling the transmission of valid data and control configuration registers.

The AXI Lite interface variant is implemented to manipulate the control registers (see 3.2). This variant of the AXI is a lite bus protocol, ideal to manipulate registers. The AXI Lite interface is used to trigger new executions, in both the HOG and SVM modules, and provides access to the registers holding information about the execution state, i.e., if the components need new data from the main memory or if the execution is completed.

The AXI Full interface enables complete memory-mapped communications, including single and burst accesses to the connected devices. Although the AXI Full variant provides burst read

and write operations, for larger blocks of data transmitted the performance drops considerably, especially compared with a system where dedicated hardware handles the data exchange, such as the DMA.

The main advantage of the AXI Full is the easiness of integration in any system, since it does not require any device driver implementation and can be used simply through direct access to the memory address, similar as to handling a block of shared memory in a software application. The strategy to reduce the transmission latency is to decrease the amount of data sent in each burst access, since the AXI Full has better performance, compared with the DMA engine, for smaller blocks.

The AXI Full variant is also very advantageous for the implemented system, providing the software application the power to transfer the desired data into specific regions of the memory block. For the data cache concept, this is a major key point, enabling the partitioning of the memory and managing the read and write operation into different sections of the connected memory, decreasing the latency and the number of cycles the component is stalled waiting for new data.

4.1.3 Testbench

The testbenchs for the HOG and SVM components include all the required logic to emulate its integration in an Embedded System (see 2.11.1). The testbench injects data into the DuT, managing its execution through the Control Registers provided, and comparing the output data with pre-calculated values from an certified software implementation.

The testbench manipulates the components register banks, as well triggering data refresh in the respective memories when a cache miss is acknowledged, and also data fetching from local files, used to inject the HOG and SVM memories with valid data to enable the correct execution.

Block Random-Access Memory (BRAM) IPs emulate the connection between the components and the Data Cache. The BRAM is implemented as a dual port memory, where one port allocated to the data exchange with the component and the other is connected with the module responsible for refreshing data.

Between the values outputted by the DuT and the pre-calculated, there is an admissible error due to the implementation of the Fixed-Point representation in the HARVAHDL design.

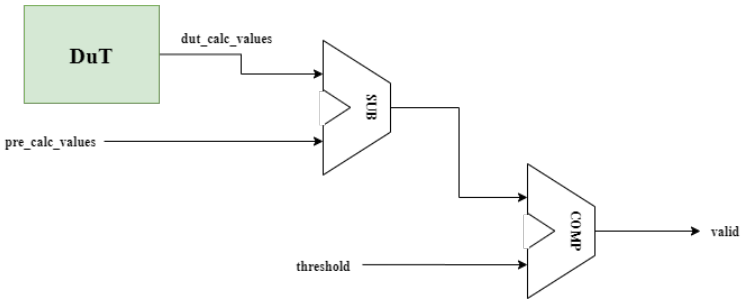


Figure 4.5: Fixed-Point error compensation hardware.

4.2 Test Cases

The HOG+SVM design requires validation of certain operations, as well as of the complete functional behavior. The testing was divided in multiple test cases, a specification of steps, inputs and execution conditions to test behavior of the target component. [59]

The Feature Extraction algorithm requires the validation of two tasks: the padding exerted in the Convolution Stage within the HOG pipeline, which occurs upon an HSYNC event, and the output of the HOG pipeline for a image subset of 16x16, the building blocks of the feature descriptor.

The Classification model does not require any direct testing, being tested in the cache miss and detecting window test cases, which target the design as a whole.

The test cases implemented include a test sequence, detailing the steps required to stimulate the design and the values to be set in the configuration registers for the intended case (see 3.2.1 and 3.2.2).

4.2.1 Block Diagram

The Block Diagram Test Case tests the HOG+SVM for subsets of 16x16 pixels from the input image, as illustrated in figure 4.6, which is the base image division within the HOG algorithm.

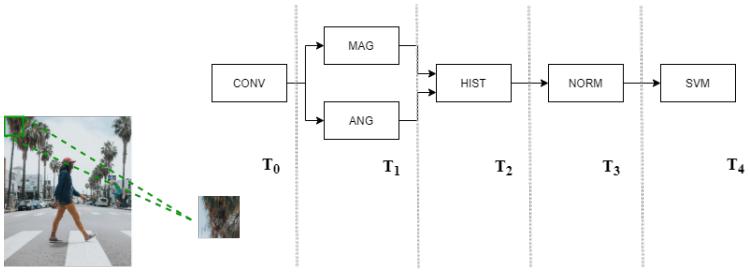


Figure 4.6: Block Diagram test case for the INRIA dataset.

The Block Diagram Test Case respects the following sequence:

1. Inject the image data into the HOG data cache data cache;
2. Write 16 to the least and the most 16 significant bits of the IMG_SIZE register to set the height and width of the input image to match the block size;
3. Write '1' to bit 0 of HOG_CTRL register;
4. Poll the bit 2 of HOG_CTRL register for the completion of the HOG processing.

The Block Diagram test validates the basic HOG functions, ensuring the block histograms are built correctly and with the expected behavior, and therefore the SVM component can be disabled to reduce the simulation time.

The output from the HOG is compared with pre-calculated values for the same block, accounting for a admissible error associated with Fixed-Point data representation.

4.2.2 HSYNC

The HSYNC Test Case aims to test the behavior of the components when the event End of Horizontal Pixel Line occurs. Upon this event, the input data must be padded to account for the convolution, as illustrated in figure 4.7, and therefore this process must be validated.

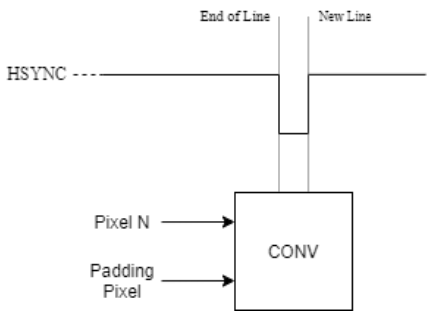


Figure 4.7: HSYNC event for the HARVA system.

This test case is an expansion of the Block Diagram, running the same testbench but for a bigger time window and for multiple image subsets, until an HSYNC event is triggered, as shown in figure 4.8.

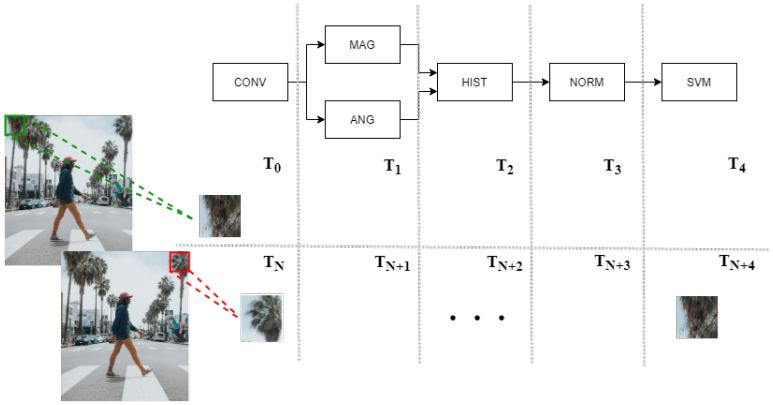


Figure 4.8: HSYNC test case for the INRIA dataset.

The HSYNC Test Case follows the next sequence:

1. Inject the image data into the HOG data cache;
2. Write intended image dimensions to the IMG_SIZE register;
3. Write '1' to bit 0 of HOG_CTRL register;
4. Poll the bit 2 of HOG_CTRL of the register for the completion of the HOG processing.

Similar to the Block Diagram test case, the HSYNC test targets a specific operation within the HOG component and, therefore, does not require the SVM module to be enabled.

4.2.3 Cache Miss

The Cache Miss Test Case is designed to validate data refresh requests for the HOG and SVM data caches. The test injects the required data into the memories and polls for any active data requests, upon which it injects new data to enable the components to resume the normal execution. The Cache Miss test follow the next sequence:

1. Inject the image data into the HOG data cache and the respective Support Vectors into the SVM data cache;

2. Write intended image dimensions to the IMG_SIZE register;

3. Write '1' to bit 0 of HOG_CTRL and SVM_CTRL registers;

4. Poll the bit 1 of HOG_CTRL and bit 2 of the SVM_CTRL registers for the signaling of a cache miss.

5. Inject the image data into the HOG data cache and the respective Support Vectors into the SVM data cache;

In order to optimize the simulation time of this test, the data cache memories can be set to smaller sizes, increasing the frequency of Cache Miss events within a smaller time window.

4.2.4 Detecting Window

The Detecting Window Test Case is the complete testing of the HOG+SVM design, validating the execution of the components in its target environment. Multiple subsets from the input image are sampled and processed, until an VSYNC event occurs, dictating the end of valid image data, as illustrated in figure 4.9.

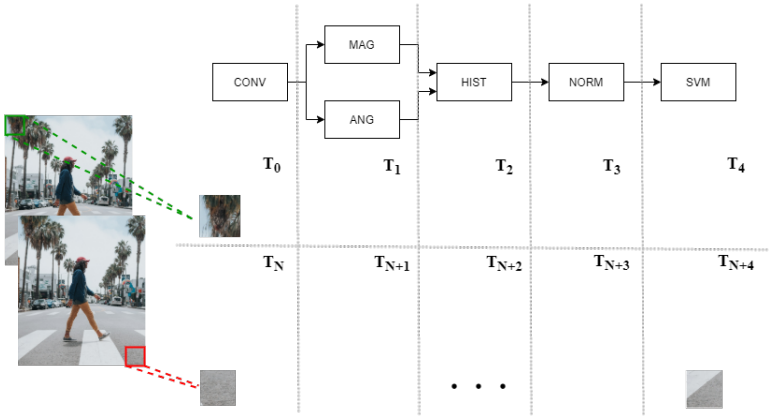


Figure 4.9: Detecting Window test case for the INRIA dataset.

The test requires the configuration of the control registers, similar to the other mentioned test cases, and the polling for Cache Miss events. The test sequence follows as:

1. Inject the image data into the HOG data cache and the respective Support Vectors into the SVM data cache;

2. Write intended image dimensions to the IMG_SIZE register;

3. Write '1' to bit 0 of HOG_CTRL and SVM_CTRL registers;

4. Poll bit 1 of HOG_CTRL and bit 2 of SVM_CTRL registers for a data refresh requests;

   (a) Inject the image data into the HOG data cache if a HOG data refresh requests is active;

   (b) Inject the image data into the SVM data cache if a SVM data refresh requests is active;

5. Poll the bit 2 of HOG_CTRL and bit 3 SVM_CTRL register for the completion of the HOG+SVM processing.

This test case is design for two purposes, validation of the functional behavior, as mentioned, and profiling of the implementation results, providing the latency of the design for the image dimensions chosen at the conofiguration of the test.

4.3 Software Model Architecture

The HARVA system was developed as a C/C++ software application, implementing the HOG and SVM components as concurrent threads, to simulate the existing parallelism of hardware implementation.

The implementation of the concurrent threads was handled with the Portable Operating System Interface (POSIX) Threads, commonly denominated pthreads within the software development community. Pthreads enable the developer to control, through calls to POSIX Threads APIs, several threads of execution that overlap in time.

The HARVA software application was implemented based in the Pthread Pipeline Model [60], graphically described in figure 4.10, where the concurrent threads represent the pipe stages that process sequentially the input stream of data while processing several blocks in parallel.

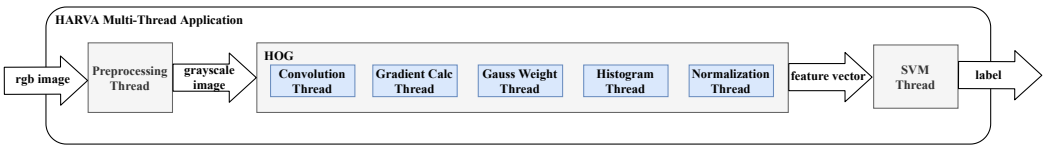


Figure 4.10: Object detection application as Pipeline multi-threaded Model.

The different threads of the HARVA pipeline present different latencies, therefore synchronization mechanisms were implemented to enable buffering between the HOG and SVM algorithms, avoiding stalling either component when data is not available, similar to the hardware implementation.

4.3.1 Preprocessing Thread

The preprocessing thread, described in the flowchart in figure 4.11, handles the image acquisition, decoding, and color space conversion in a one-shot operation. The thread also formats the image into the required format to enable the HOG and SVM threads to executed properly.

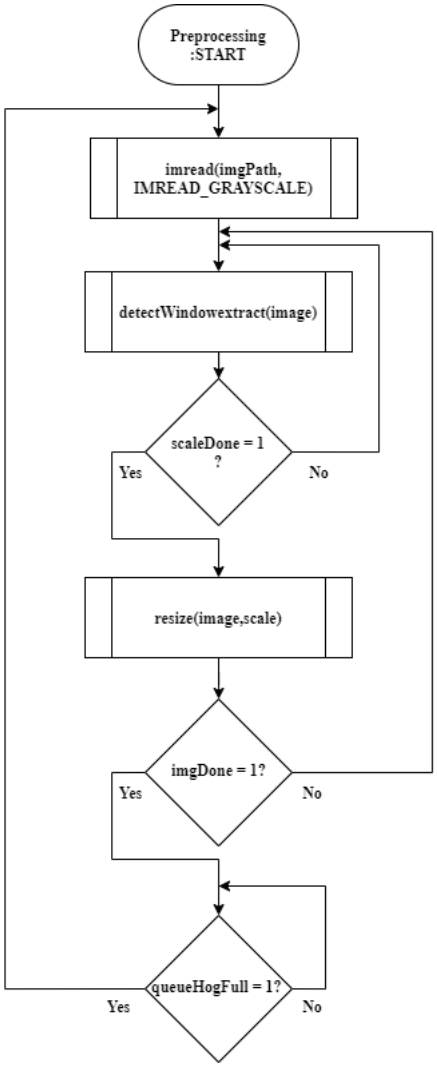


Figure 4.11: Preprocessing component flowchart for the HARVA system.

The usage of the Linux OS in the test environment enables the utilization of open source libraries built for Image Processing purposes. The OpenCV (Open Source Computer Vision Library) is a library, constantly updated by several developers around the world, aiming to solve a great variety of problems, mainly from real-time computer vision.

The *imread* API provides all the required functionalities to read image data, as well as for handling color space conversions in one single function call (see 2.5.2). The *imread* API, supports the popular image file storage formats, such as jpeg and png, and provides the grayscale image generation by setting *flag* argument to *IMREAD_GRAYSCALE*, represented in figure 4.12.

Figure 4.12: *imread* API's documentation.

The image resizing algorithm is also based on the OpenCV library, through the *resize* API, represented in figure 4.13 (see 2.5.3). This API provides several popular interpolation algorithms, such as Pixel Repetition, Bicubic, and Bilinear Interpolation addressed in the State of Art 2.5.3.

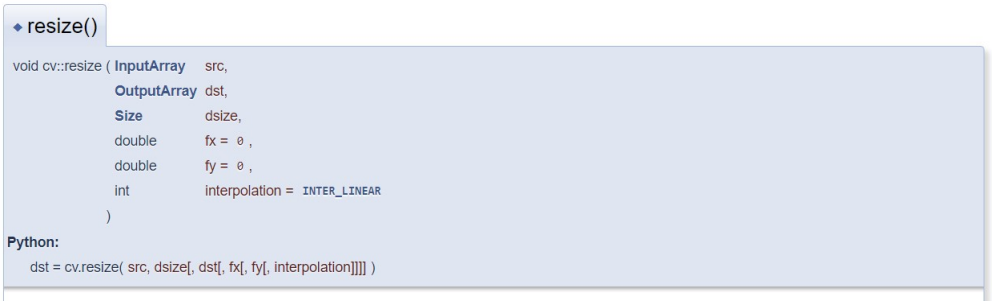


Figure 4.13: OpenCV *resize*'s documentation.

The preprocessed images are placed in a queue element, connected with the HOG thread.

4.3.2 HOG Thread

The HOG thread spawns 5 child threads, which process the input image as a 5-stage pipeline similar to the hardware counterpart implementation.

The Convolution stage, implemented based in the flowchart of the figure 4.14, queries the $pixelDataAvail$ semaphore for new data available in the input buffer, which is padded, when

required, and then used to compute, concurrently, the vertical and horizontal derivatives through a call to the $compDeriv$ function. This process is repeated for a second input, necessary for producing 4 parallel outputs for each derivative, and the new valid output signal is transmitted to the next pipeline stage.

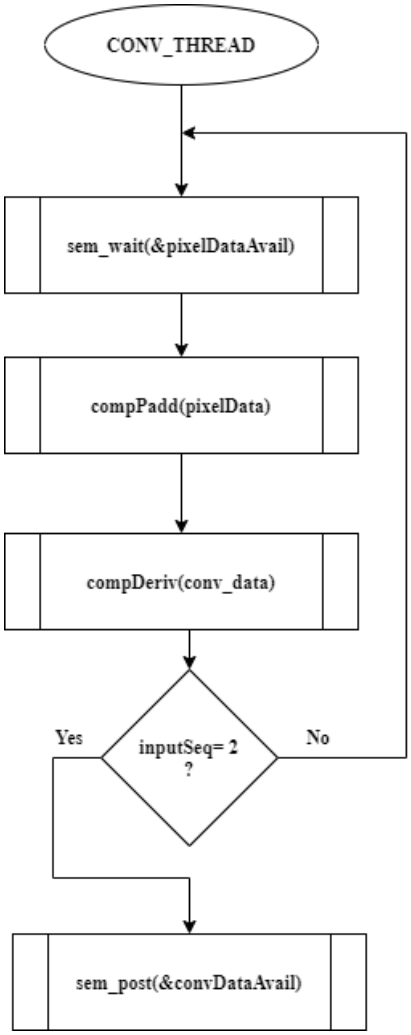


Figure 4.14: Convolution Stage software implementation for the HARVA's profiling.

The Gradient Calculation stage, represented by the flowchart in figure 4.15, polls the $convDataAvail$ semaphore, which signals the availability of new derivative data. Upon new data available, the thread computes the image gradients by calling the $compGrad$ function. With the computation completed, the tread signals to the next stage, through the $gradDataAvail$, the availability of the newly produced gradients.

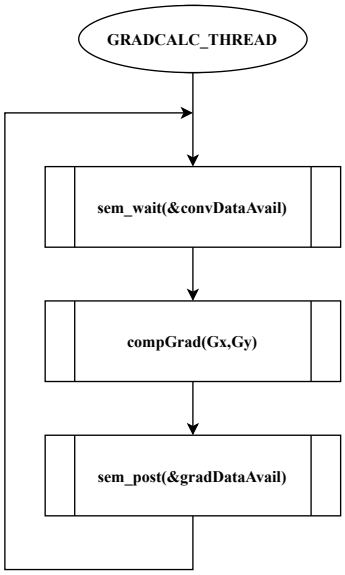


Figure 4.15: Gradient Calculation Stage software implementation for the HARVA profiling.

The Histogram stage, based in the flowchart in figure 4.16, queries the for new filtered Magnitude data, by checking the $magFiltDataAvail$ semaphore. The Histogram thread processes the input gradients through the $compHist$ function, repeating the sequence until the block histogram is completed. Upon completion of the histogram building process, the tread signals the availability of new data to the final stage of the pipeline.

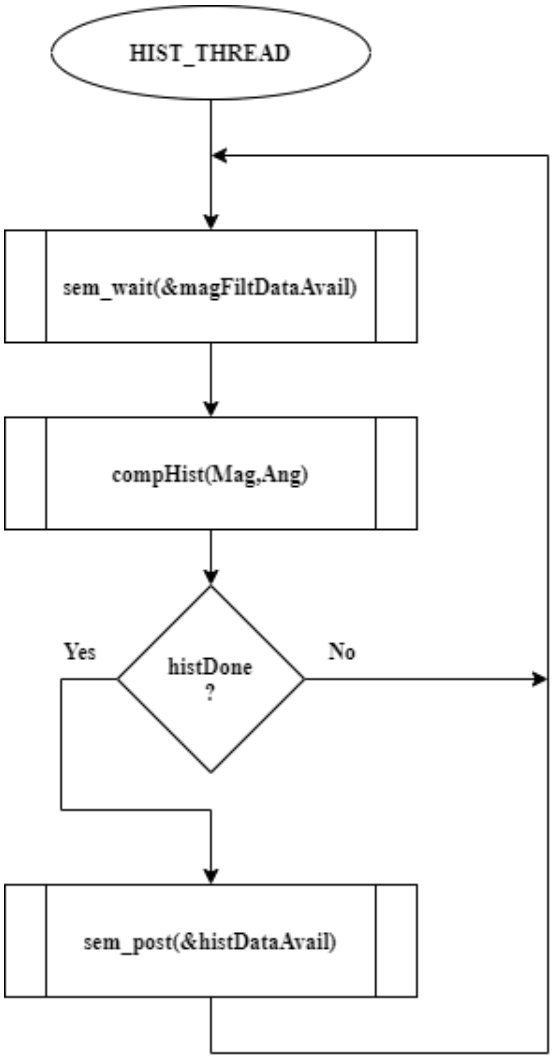


Figure 4.16: Histogram Stage software implementation for the HARVA profiling.

The Normalization stage, based in the flowchart in figure 4.17, polls for new histogram data, checking the $histDataAvail$ semaphore. The histogram normalization is computed bu the $compNorm$ function, normalizing sequentially the histogram values. With the normalization completed, the thread signalizes and inject data in the output buffer, which is connected with the classification algorithm.

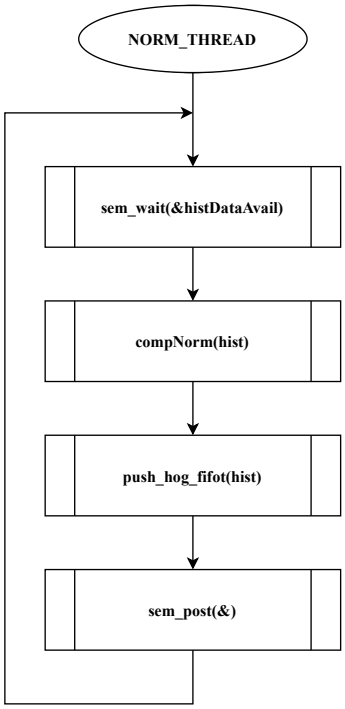


Figure 4.17: Normalization Stage software implementation for the HARVA profiling.

4.3.3 SVM Thread

The SVM thread implements the classification algorithm, taking as input the feature vector computed in the HOG thread. The classification, similarly to the hardware counterpart, implements the Linear Combination between the feature vectors and the support vectors values. The result from the Linear Combination is then added to the bias value, or more commonly known as the threshold, and the label value is computed.

If the label value is 1, it means there is a person within the input image, otherwise, it means the opposite.

The execution flow is described in the flowchart of figure 4.18.

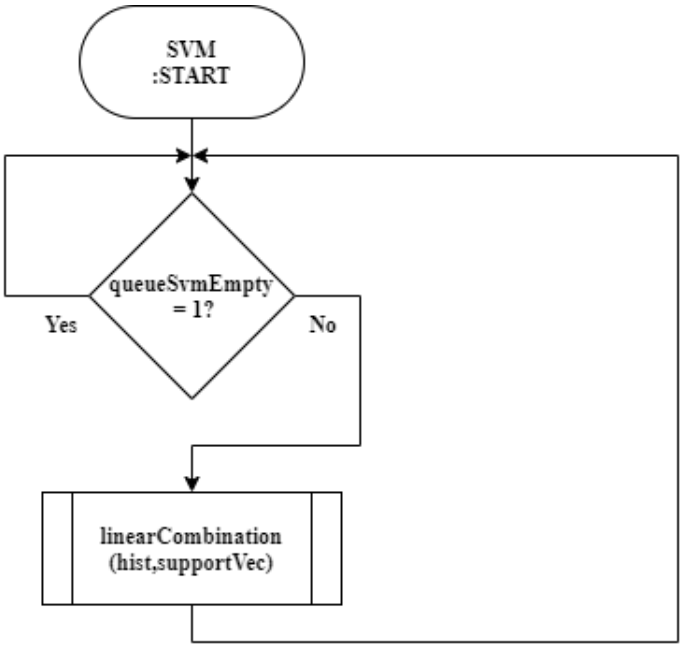


Figure 4.18: SVM thread's diagram.

4.4 Results

The Oprofile profiling tool was used for sampling the developed software application, designed to exploit the HOG + SVM Object Detection algorithm's performance and the existing bottlenecks.

The HOG and SVM components were tested resorting to the Xilinx ISim, which provides a full-featured HDL simulator supporting simulation of customer-designed AXI-based modules with up to 50% acceleration of the process. The ISim enables the analysis of the component's logic behavior with high accuracy. The HOG and SVM components were simulated with several levels of parallelism to extract the most information about the impact of parallelism in the system.

4.4.1 Software Model Profiling

The software model is implemented with the INRIA Pedestrian dataset, composed by pre-processed 64x128 images. The HOG algorithm divides each image into 128 blocks, building for each one the respective histogram representing the pixel intensity distribution.

With a block histogram being comprised of 36 values, the Support Vector, as well as the Feature Vector, have a total size of 4608 values.

The large amount of floating-point operations implemented by the Object Detection model to produce the Feature vector and then to classify it, alongside with the great memory footprint, are common bottlenecks in software application in constrained systems, where the number of FPUs and memory capacity are very limited.

Oprofile is an open-source statistical profiler capable running with several configurations for a wide range of computers;

The Oprofile accesses the Hardware Performance Counter of the CPU, gathering a wide variety of information related to the program execution;

The overhead introduced by the profiling is reduced, depending on the interrupt load configured for the Oprofile;

The Oprofile is run with the default configuration, which samples the CPU_CLK_UNHALTED Hardware Performance Counter. The generated report provides information about the sampling frequency, as well as the number of samples for each symbol, as shown in figure 4.19 for the Oprofile results of the software application.

```
CPU: Core 2, speed 2000 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (Unhalted core cycles) count 100000
samples  %        symbol name
200220   46.6817  gradCalc()
82119    19.1462  norm()
69564    16.2190  histogram()
66501    15.5048  conv2d()
10278     2.3963  svm()
223       0.0520  main
```

Figure 4.19: Oprofile results for the profiling of the HARVA software implementation.

The Gradient Calculation Stage has the most impact in the runtime of the software application, having an percentage of allocation near 50%. This stage includes the most demanding computation, within the HOG pipeline. Combined with the higher computational level required, this stage implements trigonometric operations, which have much higher latency than the arithmetic present in other stages.

The second highest task, rounding a percentage of 19% for the runtime consumption, is the normalization. This happens because the stage has to process the 36-value block histogram twice: first for calculating the sum of all coefficients and second to normalize the histogram.

The convolution and the histogram creation tasks have a similar runtime impact, rounding the 15%, since the computation are composed, mostly, of additions and subtractions.

The classification have considerably lower impact, near 2% of the runtime, due to the reduced and simplicity of computations implemented.

4.4.2 HOG Component

The HOG is profiled, for both latency and the resource usage to provide information on the impact of the module in the system.

The main reason for the decrease in latency is the fact that the HOG HDL module has an higher level of parallelism, i.e., several values are processed in parallel. Another key factor in the performance boost is the implementation of Fixed-Point data representation instead of Floating-Point, since Computer Systems have a reduced number of FPU, which often creates bottlenecks in software applications. Fixed-Point representation is implemented through integer logic, resulting in a decrease in hardware complexity and latency. Another key factor is that the tackling of HOG bottleneck, the Gradient Calculation stage, is tackled and reduced from a latency of 133 clock cycles, for each gradient, in an software implementation to 18 clock cycles in the HDL design. The convolution of 4 parallel pixel values also has a major impact in the performance boost, since it increases, even more, the level of parallelism of the pipeline.

The Hardware Description Languages (HDL) implementation of the HOG algorithm for an INRIA image takes around 1.972 ms, for the default configuration of 1 core, while its best performance has a Latency of 0.439 ms, as indicated in the graph of figure 4.20.

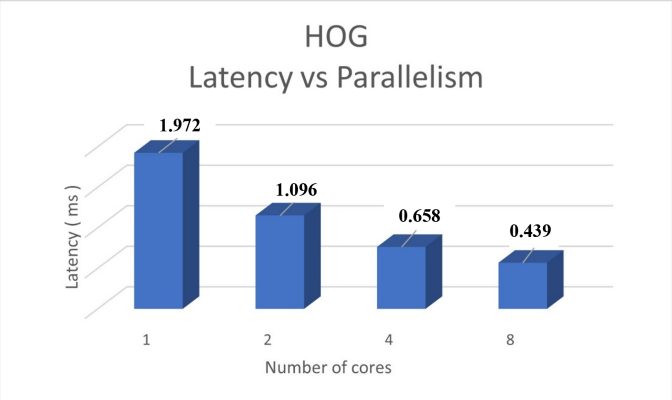


Figure 4.20: Hardware profiling chart for the HOG HDL design. The chart provides information about the latency variation for different number of cores configurations of the HOG.

This performance, translates in an acceleration of, at least, 72% in the runtime execution time.

The synthesis operation for the HOG component generates the tables in figures 4.21, 4.22 and 4.23. The HOG module uses LuT slices purely for component logic, requiring a fairly reduced amount of slices from the available on the target FPGA fabric, shown in figure 4.21.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	4082	0	218600	1.87
LUT as Logic	4082	0	218600	1.87
LUT as Memory	0	0	70400	0.00
Slice Registers	4369	0	437200	0.99
Register as Flip Flop	4369	0	437200	0.99
Register as Latch	0	0	437200	0.00
F7 Muxes	329	0	109300	0.30
F8 Muxes	126	0	54650	0.23

Figure 4.21: HOG HDL design's synthesis results.

A key point from figure 4.21 is the fact that no Latches are generate from the HDL code developed, which when inferred non-intentionally might cause setup and hold timing violations in the design.

The table from figure 4.22 shows that the HOG uses 8 DSP slices, allocated for the Gradient Calculation and Normalization stages. The Gradient Calculation stage uses DSP slices for the CORDIC core, to ensure a more efficient implementation of the algorithm, while the Normalization stage implements a division operation, alongside several multiplications, which are normally mapped to DSP slices.

Site Type	Used	Fixed	Available	Util%
DSPs	8	0	900	0.88

Figure 4.22: HOG HDL design's synthesis results for DSP slices.

The HOG module implements 1 BRAM, which translates to the use of 1 RAMB18 slice, as shown in figure 4.23. The Block Memories implemented can be configured as an 36 Kb RAM(RAMB36) or two independent 18 Kb RAMs(RAMB18), which is why in the Block RAM Tile the value is set to 0.5 in the synthesis table.

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	0.5	0	60	0.83
RAMB36/FIFO*		0	60	0.00
RAMB18	1	0	120	0.83

Figure 4.23: HOG HDL design's synthesis results for BRAM blocks.

4.4.3 SVM Component

The SVM is profiled, for both latency and the resource usage to provide information on the impact of the Classification in the overall Object Detection system.

The implementation of the SVM translates to an acceleration of, at least, 87% in the runtime execution. The increase of the component's parallelism shows, naturally, an improvement in the component's performance, as shown in the graph of figure 4.24

The SVM component presents an increase in memory efficiency, since the SVM's Support Vectors can be stored as 8-bit quantized values, decreasing the module's memory footprint in 88%, relatively to the software counterpart.

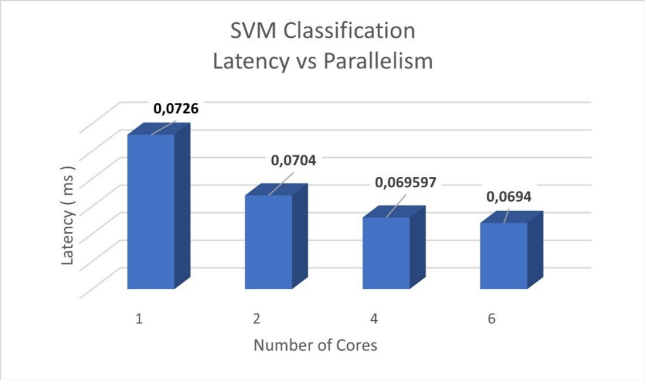


Figure 4.24: Hardware profiling chart for the SVM HDL design. The chart provides information about the latency variation for different number of cores configurations of the SVM.

The synthesis reports, which produce the tables from figures 4.25 and 4.26, show a low resource usage, compared with the slices available in the target FPGA fabric.

Similarly to the HOG reports, the SVM HDL code does not generate latches, avoiding possible timing violations in the design.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	411	0	218600	0.19
LUT as Logic	411	0	218600	0.19
LUT as Memory	0	0	70400	0.00
Slice Registers	566	0	437200	0.13
Register as Flip Flop	566	0	437200	0.13
Register as Latch	0	0	437200	0.00
F7 Muxes	64	0	109300	0.06
F8 Muxes	0	0	54650	0.00

Figure 4.25: HOG HDL design's synthesis results.

The SVM module implements a BRAM and a FIFO for Support Vectors and Feature Vector data. The design allocates 1.5 BRAM slices, as shown in figure 4.26, where 1 slice of RAMB36 is

implemented for the Feature Vector FIFO and 1 slice of RAMB18 is implemented for the Support Vectors Data Cache.

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	1.5	0	60	2,50
RAMB36/FIFO	1	0	60	1,67
RAMB18	1	0	120	0.83

Figure 4.26: HOG HDL design's synthesis results for BRAM blocks.

4.5 Detection Performance

The Detection Performance evaluates the HOG+SVM model for two different metrics: False Detection and Detection rate. A test set, a combination of negative and positive samples from the INRIA dataset, combined with the respective classification for each item to provide valid results for the performance of the algorithm with real data.

4.5.1 False Detection

False Detection is an error in the Classification operation within a Detection System, which wrongly classify a image, or image subset, related to the present of the system's target object. This error is forked into two types: False Positive and False Negative. [61]

A False Positive error consists in the Classification operation to incorrectly classify the input image with the presence of the target object, as exemplified in figure 4.27 where the image is classified as containing a pedestrian.

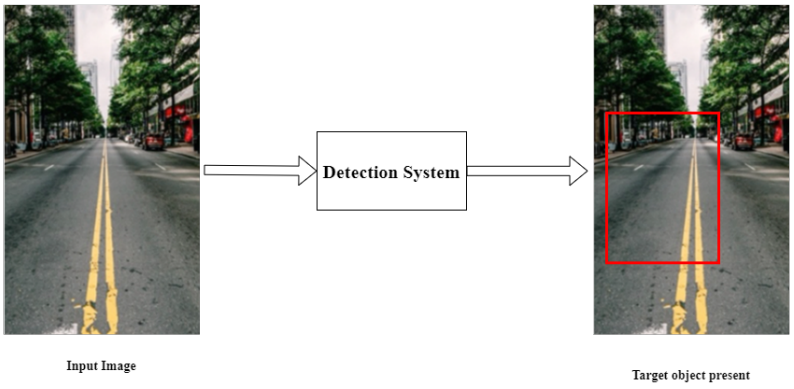


Figure 4.27: False Positives within a Object Detection System.

A False Negative occurs when the Classification algorithm wrongly indicates that the input image does not contain the target object when in fact it does, illustrated in figure 4.28.

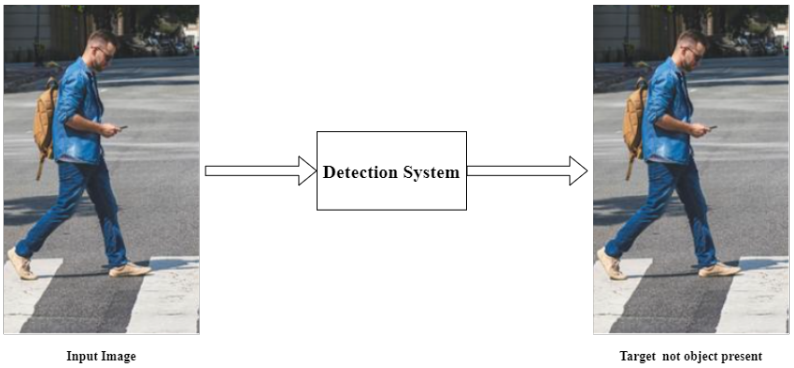


Figure 4.28: False Negatives within a Object Detection System.

The HARVA system, upon processing a test set composed of 900 images from the INRIA, outputted a total of 212 false detections, translating to a rate of 18%.

4.5.2 Detection Rate

The Detection Rate is a metric for the quantization of the performance of the HARVA system, and is defined by the percentage of correct guesses made by the model for the test set feeded.

The Detection Rate metric takes into account the False Detection rate, as well as the correct guesses, in order to determinate how well the model behaves

The testing set was set to include a total of 900 images from the INRIA dataset, which were not used in the training routines. Within the test images, there were a total of 1178 target objects, i.e., pedestrians.

Chapter 5

Conclusion

This chapter concludes the dissertation, providing an overview of the developed work, shown in previous chapters, as well as addressing possible future works with the HARVA system and its components.

The HARVA project required a varied set of skills, encompassing several areas, from Computer Architecture knowledge, for better exploit the possible data parallelism within the system to Machine Learning algorithms, to Electrical and Electronic Engineering, for the development of the HDL of the HARVA system and its multiple components.

The objectives established for a configurable implementation of a Pedestrian Detection and Bluring System were accomplished.

5.1 Developed Work

In this dissertation scope, an Object Detection algorithm was developed both as a software application, for comparison and profiling purposes, and as a combination of two hardware modules, HOG and SVM.

The software application required research on multi-thread models to find the most suitable model for the HARVA system. The development was boosted with the use of the OpenCV open-source library, which provided the APIs to handle the acquisition of the image and the operations required to transform the input image into an RGB format.

The implementation of the software application as a multi-threaded program was handled by the POSIX threads library, that enabled the emulation of parallel tasks within the program.

The HARVA system was also implemented in HDL, in order to test the behavior and performance of HOG and SVM components with the INRIA dataset.

The HOG component includes generic components for Image Processing, such as Convolution, the most popular process in signal processing systems, responsible for computing the pixel intensity derivations, Histogram representation and normalization, which are processes commonly used in Machine Learning applications for processing the input images.

The SVM component implements the feature classification, including a quantization and a linear combination module. Both modules are used across a wide range of applications within the Machine Learning area.

The modules developed for the implementation of the HARVA system were designed to enable several configurations, implementing different levels of parallelism in data process within the module. The user is, then, provided with the choice to privilege the system's performance or resource consumption, by processing 1 value, similar to the software sequential execution, or multiples values in parallel. The buffering elements, present in the HOG components are configurable, supporting different frequencies within stages.

The testing of the hardware implementation was executed through software, which handled the image acquisition, similar to the software model, and communicated with the HDL modules through the AXI bus. The program also managed the Memory Blocks data, ensuring there was new data ready for processing as soon as possible.

The HARVA implementation required some interconnecting cores, that handled the connection between the software and hardware components, without any interaction necessary. The AXI SmartConnect core, provided by Xilinx, was analyzed and the possible supported configurations studied upon its integration in the architecture.

The HARVA system also provides the possibility to accelerate only the Feature Extraction or the Classification process, by implementing either the HOG or the SVM component, resulting in lower manufacturing cost.

5.2 Future Work

The HARVA system was developed in a modular manner, where each module included was designed independently to enable integration in different systems if required.

Several modules of the HARVA are generic to image processing processes, such as the convolution, the normalization and the Gaussian filter, and can be reused in different use cases with little, or even none, rework.

In order to increase the system's performance the DMA engine, and the respective driver can be implemented, requiring some minor adjustments.

The Preprocessing routine can also be implemented in hardware, avoiding the overhead of the transmission from the system's main memory and the hardware memory blocks, with the drawback of increasing, considerably, the manufacturing cost.

References

[1] A. A. Khan, N. Rink, F. Hameed, and J. Castrillón, "Optimizing tensor contractions for embedded devices with racetrack memory scratch-pads," 06 2019.

[2] Machine learning. [Online]. Available: https://www.ibm.com/cloud/learn/machine-learning

[3] Applications of machine learning. [Online]. Available: https://www.javatpoint.com/applications-of-machine-learning

[4] Everything you need to know about the ecu. [Online]. Available: https://philcarnews.com/car-maintenance/everything-you-need-to-know-about-the-ecu-ta255

[5] Hdd vs flash: The differences between flash and hdd. [Online]. Available: https://nexstor.com/hdd-vs-flash

[6] C. Y. Qing Li, *Real-Time Concepts for Embedded Systems*, 2003.

[7] K. Shin and P. Ramanathan, "Real-time computing: a new discipline of computer science and engineering," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 6–24, 1994.

[8] K. Sankaralingam, S. Keckler, W. Mark, and D. Burger, "Universal mechanisms for data-parallel architectures," vol. 2003, 01 2004, pp. 303– 314.

[9] K. Ebcioglu, J. Dehnert, M. Schlansker, T. M. Conte, J. Z. Fang, and C. L. Thompson, "Compilers for instruction-level parallelism," *Computer*, vol. 30, no. 12, pp. 63–69, dec 1997.

[10] What is speculative execution? [Online]. Available: https://www.extremetech.com/computing/261792-what-is-speculative-execution

[11] J. L. H. D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2011.

[12] “Ieee standard for floating-point arithmetic,” *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[13] T. Fryza, “Introduction to fixed-point multiplication and signal processing application,” 04 2009.

[14] B. Widrow and I. Kollár, *Quantization noise in Digital Computation, Signal Processing, Control and Communications*, 2007.

[15] What is linux. [Online]. Available: https://opensource.com/resources/linux

[16] Gnu operative system. [Online]. Available: https://www.gnu.org/

[17] Difference between bare metal vs. embedded linux. [Online]. Available: https://miscircuitos.com/difference-between-bare-metal-vs-embedded-linux

[18] An introduction to using linux in embedded systems. [Online]. Available: https://thenewstack.io/an-introduction-to-using-linux-in-embedded-systems

[19] Iot – internet of things: um caminho para a sustentabilidade. [Online]. Available: https://autossustentavel.com/2019/08/iot-internet-of-things-um-caminho-para-a-sustentabilidade.html

[20] In-vehicle infotainment (ivi). [Online]. Available: https://www.st.com/en/applications/in-vehicle-infotainment-ivi.html#overview

[21] Explanation of “everything is a file” and types of files in linux. [Online]. Available: https://www.tecmint.com/explanation-of-everything-is-a-file-and-types-of-files-in-linux

[22] M. Jones. Anatomy of the linux virtual file system switch. [Online]. Available: https://developer.ibm.com/tutorials/l-virtual-filesystem-switch

[23] R. Luna and S. A. Islam, “Security and reliability of safety-critical rtos,” *SN Computer Science*, vol. 2, 09 2021.

[24] Realtime kernel patchset. [Online]. Available: https://wiki.archlinux.org/index.php/Realtime_kernel_patchset

[25] Go board - vga introduction. [Online]. Available: https://www.nandland.com/goboard/vga-introduction-test-patterns.html

[26] G. Hoffmann. Cielab color space. [Online]. Available: http://docs-hoffmann.de/cielab03022003.pdf

[27] R. A. Peters, "Eece 4353 image processing: Resizing images," 2018.

[28] Differences between the l1-norm and the l2-norm. [Online]. Available: http://www.chioka.in/differences-between-the-l1-norm-and-the-l2-norm-least-absolute-deviations-and-least-squares

[29] The difference between l1 and l2 regularization. [Online]. Available: https://explained.ai/regularization/L1vsL2.html

[30] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*, 2012.

[31] Supervised learning. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/supervised-learning

[32] Machine learning classifiers. [Online]. Available: https://towardsdatascience.com/machine-learning-classifiers-a5cc4e1b0623

[33] Introduction to machine learning algorithms: Linear regression. [Online]. Available: https://towardsdatascience.com/introduction-to-machine-learning-algorithms-linear-regression-14c4e325882a

[34] Unsupervised learning and data clustering. [Online]. Available: https://towardsdatascience.com/unsupervised-learning-and-data-clustering-eeecb78b422a

[35] Exploring clustering algorithms: Explanation and use cases. [Online]. Available: https://neptune.ai/blog/clustering-algorithms

[36] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2005)*, vol. 2, 06 2005.

[37] Hog (histogram of oriented gradients): An overview. [Online]. Available: https://towardsdatascience.com/hog-histogram-of-oriented-gradients-67ecd887675f

[38] B. Boser, I. Guyon, and V. Vapnik, “A training algorithm for optimal margin classifier,” *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, vol. 5, 08 1996.

[39] Support vector machine — introduction to machine learning algorithms. [Online]. Available: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

[40] G.-X. Yuan, C.-H. Ho, and C.-J. Lin, “Recent advances of large-scale linear classification,” *Proceedings of the IEEE*, vol. 100, pp. 2584–2603, 09 2012.

[41] Chipverify uvm testbench. [Online]. Available: https://www.chipverify.com/uvm/macros-and-defines

[42] Socs with hardware and software programmability. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[43] Specifying performance counter events. [Online]. Available: https://oprofile.sourceforge.io/doc/eventspec.html

[44] E. A. Lee, “Embedded software,” ser. Advances in Computers, M. V. Zelkowitz, Ed. Elsevier, 2002, vol. 56, pp. 55–95. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245802800043

[45] Co-simulation. [Online]. Available: https://opensimulationplatform.com/co-simulation/

[46] Chipscope pro and the serial i/o toolkit. [Online]. Available: https://www.xilinx.com/products/design-tools/chipscopepro.html

[47] Qemu. [Online]. Available: https://www.qemu.org/

[48] W. Zabolotny, *Development of embedded PC and FPGA based systems with virtual hardware*, 10 2012, vol. 8454, pp. 84 540S–1.

[49] How do self-driving cars see? [Online]. Available: https://medium.com/@albertlai631/how-do-self-driving-cars-see-13054aee2503

[50] Why deep learning over traditional machine learning? [Online]. Available: https://towardsdatascience.com/why-deep-learning-is-needed-over-traditional-machine-learning-1b6a99177063

[51] H. Bristow and S. Lucey, "Why do linear svms trained on hog features perform so well?" 06 2014.

[52] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, "Fpga-based real-time pedestrian detection on high-resolution images," 06 2013.

[53] R. Hamdini, N. Diffellah, and A. Namane, "Robust local descriptor for color object recognition," *Traitement du Signal*, vol. 36, pp. 471–482, 12 2019.

[54] Ordic. [Online]. Available: https://www.xilinx.com/products/intellectual-property/cordic.html

[55] Trigonometric functions and the unit circle. [Online]. Available: https://courses.lumenlearning.com/boundless-algebra/chapter/trigonometric-functions-and-the-unit-circle/

[56] Test environments 101: Definition, types, and best practices. [Online]. Available: https://launchdarkly.com/blog/test-environments-101-definition-types-and-best/

[57] N. Dalal, "Finding people in images and videos," 07 2006.

[58] Xilinx zynq-7000 soc zc706 evaluation kit. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html#hardware

[59] D. Almog and T. Heart, "What is a test case? revisiting the software test case concept," in *Software Process Improvement*, R. V. O'Connor, N. Baddoo, J. Cuadrago Gallego, R. Rejas Muslera, K. Smolander, and R. Messnarz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 13–31.

[60] B. Nichols, D. Buttlar, and J. Farrell, *Pthreads Programming: A Posix Standard for Better Multiprocessing*, 01 1996.

[61] Evaluating the performance of machine learning models. [Online]. Available: https://towardsdatascience.com/classifying-model-outcomes-true-false-positives-negatives-177c1e702810