Design and Implementation of a Real-Time Pedestrian Detection System on SoC FPGA

Master Thesis

FILIPPOS CHALKIPOULOS
ELECTRONIC COMPUTER SYSTEMS ENGINEER

Patras, February 2019
Design and Implementation of a Real-Time Pedestrian Detection System on SoC FPGA

Master Thesis

FILIPPOS CHALKOPOULOS
ELECTRONIC COMPUTER SYSTEMS ENGINEER

Supervisor: Economou Georgios, Professor

Approved by the three-member examination committee on 26th of February, 2019

Economou Georgios
Professor

Anastassopoulos Vassilis
Professor

Bakalis Dimitris
Assistant Professor
Acknowledgement

I would like to express my deepest gratitude to my thesis advisor, Professor Mr. Evangelos Zigouris for his guidance and support during my postgraduate studies. His enthusiasm for transferring his knowledge on his students was a great inspiration for me and many other students as well. He was also always there to push me forward and guide me, which helped me to evolve personally and professionally.

I would also like to thank Professor Mr. Georgios Economou for the opportunity he gave me to work on this project. Furthermore, I would like to express my thanks to the other two members of the examination committee, Professor Anastasopoulos Vasilis and Assistant Professor Bakalis Dimitrios, who accepted to evaluate my work during this thesis.

Special thanks to Dr. Athanasios Kalantzopoulos for his advice and technical support during my studies.

I would like to thank all the professors and colleagues during my Master’s degree program who have helped me to develop skills, throughout our cooperation.

In conclusion, I would like to state my biggest appreciation for my family and my girlfriend for their patience and huge support and all my friends who were beside me, during this intensive period.
Dedication

To my beloved grandfather,
Dimitris…
Abstract

Object Detection Systems have been developing in recent years, in order to add a key characteristic in system automation, the ability of a machine to detect objects without human intervention. These systems are pushing forward and leading on technologies such as driverless cars, surveillance systems, security systems, and others.

In the past years, research in the field of object detection systems, produced algorithms that are capable to detect objects from an image, based on their feature extraction. An important algorithm that has been proposed for Pedestrian Detection, is Histogram of Oriented Gradients (HOG) by Dalal and Triggs. The algorithm extracts features of an image and feeds them in a machine learning algorithm, to determine if and where there is a person in an image with a big success rate.

This thesis adopts the HOG algorithm alongside a classification algorithm – SVM (implemented in software with OpenCV library), for developing a system able to detect pedestrians in real time, from a video feed of a camera. Real-time pedestrian detection requires fast computation of the HOG and the image processing procedures which means, that the system has to process a big amount of data, in a short period of time. In addition, there is a need for flexibility of the system. It has to be flexible enough to get adopted by other systems, (such as cars, surveillance systems, machines, robots etc) and that requires the system to be easily configurable, to occupy minimum space, and have minimum energy consumption.

Given the great demands of a real-time pedestrian detection system, which has to process big load of data while at the same time provide multiple interfaces for communication and adaption, this thesis proposes a design based on a Hardware-Software Co-design. This led to a system that uses either hardware or software for the implementation of the algorithms, designed to work together on a single chip (System on Chip – SoC). In this thesis the system is implemented with the use of Cyclone V SoC FPGA chip, which hosts an FPGA fabric, a dual-core ARM* Cortex*-A9 Hard Processor System (HPS), embedded peripherals, multiport memory controllers and serial transceivers.

Summing up, this thesis aimed at designing and implementing a functional system able to detect pedestrian from a video stream coming from an onboard camera. The aim is, for the system to be flexible enough to integrate into other systems, and operate fast enough as a result of the implementation of many of its processes, hardware accelerated. The whole system is implemented on a single board which hosts the Cyclone V SoC FPGA chip, memory chips supporting the FPGA fabric and the HPS, the D8M camera kit which is connected through a parallel port, as well as the video output of the system. The output is extracted from a video DAC and a VGA connector.

Keywords: Embedded System, Pedestrian Detection, Real-Time, SoC, FPGA, OpenCV, Hardware-Software co-design.
Τα συστήματα εντοπισμού αντικειμένων που αναπτύσσονται τα τελευταία χρόνια, έρχονται να προσθέσουν ένα σημαντικό χαρακτηριστικό στην αυτοματοποίηση ενός συστήματος, την ικανότητα να εντοπίζουν αντικείμενα χωρίς ανθρώπινη παρέμβαση. Τέτοια συστήματα ωθούν προς το εμπρός και οδηγούν προς την υλοποίηση τεχνολογιών όπως τα αυτοοδηγούμενα αυτοκίνητα, τα έξυπνα συστήματα επιτήρησης, ανάπτυξη πολυπλοκότερων συστημάτων ασφαλείας κ.α.

Τα τελευταία χρόνια, η έρευνα στον τομέα των συστημάτων εντοπισμού αντικειμένων, έχει παράξει αλγόριθμους που είναι ικανοί να ανιχνεύουν αντικείμενα από μια εικόνα με βάση την εξαγωγή των χαρακτηριστικών της. Ένας σημαντικός αλγόριθμος που έχει προταθεί για την ανίχνευση πεζών είναι το Histogram Oriented Gradients (HOG) από τους Dalal και Triggs. Ο αλγόριθμος εξάγει τα χαρακτηριστικά μιας εικόνας και τα τροφοδοτεί σε έναν αλγόριθμο μηχανικής μάθησης, για να προσδιορίσει πότε και πού υπάρχει ένα άτομο σε μια εικόνα με ένα μεγάλο ποσοστό επιτυχίας.

Αυτή η εργασία υιοθετεί τον αλγόριθμο HOG παράλληλα με έναν αλγόριθμο ταξινόμησης SVM, για την ανάπτυξη ενός συστήματος ικανού να ανιχνεύει τα πεζούς από μια εικόνα, για την υλοποίηση των τεχνολογιών όπως τα αυτοοδηγούμενα αυτοκίνητα, τα έξυπνα συστήματα ασφαλείας κ.α.

Δεδομένων των μεγάλων απαιτήσεων ενός συστήματος ανίχνευσης πεζών σε πραγματικό χρόνο, το οποίο πρέπει να επεξεργάζεται μεγάλο φορτίο δεδομένων, ενώ παράλληλα να παρέχει πολλαπλές διεπαφές επικοινωνίας και προσαρμογής, αυτή η εργασία προτείνει ένα σχέδιο συστήματος βασισμένο στην logica σχεδίασης Hardware-Software Co-design. Αυτό οδηγεί σε ένα σύστημα που χρησιμοποιεί συνολικά ετελικότερο λογισμικό για την υλοποίηση των αλγορίθμων, σχεδιασμένο να συνεργάζεται σε έναν ενιαίο chip (System on Chip - SoC).

Συνοψίζοντας, αυτή η εργασία στοχεύει στο σχεδιασμό και την υλοποίηση ενός συστήματος ικανού να ανιχνεύει τα πεζούς από μια ροή βίντεο που προέρχεται από μια ενσωματωμένη κάμερα. Στόχος, το σύστημα να είναι αρκετά ευέλικτο, καθώς και να εκτελεί ως αποτέλεσμα της υλοποίησης πολλών διεργασιών.
# Table of Contents

Acknowledgement.................................................................................................................. iii
Dedication.................................................................................................................................. v
Abstract..................................................................................................................................... viii
Περιλήψη .................................................................................................................................... ix
Table of Contents..................................................................................................................... xi
Table of Figures........................................................................................................................ xv
List of Tables ............................................................................................................................ xix

Chapter 1 Introduction .............................................................................................................. 1
  1.1 Applications ...................................................................................................................... 1
    1.1.1 Pedestrian Detection and HOG+SVM .................................................................... 2
    1.1.2 Real time performance ......................................................................................... 2
    1.1.3 Embedded pedestrian detection approach ......................................................... 3
  1.2 Thesis approach ............................................................................................................... 3
  1.3 Structure of thesis ......................................................................................................... 4

Chapter 2 Pedestrian Detection ............................................................................................... 7
  2.1 Detection challenges ....................................................................................................... 7
    2.1.1 Motion-based detection ...................................................................................... 7
    2.1.2 Part-based Detection ......................................................................................... 8
    2.1.3 Holistic Detection ............................................................................................. 9
  2.2 Histogram of Oriented Gradients .................................................................................. 9
    2.2.1 Computing Gradients ....................................................................................... 10
    2.2.2 Compute histogram of gradients (cells) .......................................................... 11
    2.2.3 Compute histogram of gradients (block normalization) .................................. 12
    2.2.4 Final feature vector and HOG visualization .................................................. 13
  2.3 Linear Support Vector Machine .................................................................................... 13
  2.4 Pedestrian Detection with OpenCV library ................................................................... 14
    2.4.1 HOG plus SVM detector initialization ......................................................... 14
    2.4.2 Perform pedestrian detection on image .......................................................... 15

Chapter 3 System Design ....................................................................................................... 17
  3.1 System Requirements .................................................................................................... 17
  3.2 System Specification ..................................................................................................... 18
  3.3 System Hardware .......................................................................................................... 18
    3.3.1 What is SoC FPGAs? ...................................................................................... 18
    3.3.2 Cyclone V SoC FPGA .................................................................................... 19
    3.3.3 De1-SoC Development Board ..................................................................... 22
    3.3.4 D8M Camera Kit ............................................................................................ 23
3.4 Hardware and Software Development Tools ........................................... 25
3.4.1 Hardware Programming with Quartus and IP Cores .......................... 25
3.4.2 Software Development of Intel® SoC FPGAs .................................. 26
3.5 Hardware-Software Co-design ............................................................. 26
3.5.1 Summing up Software Implementation ........................................... 27
3.5.2 Summing up Hardware Implementation .......................................... 27
3.5.3 System Flexibility ........................................................................... 28
3.6 System Architecture – Design Progress .............................................. 28
3.6.1 Validating Video Input Stream from Camera .................................. 29
3.6.2 Importing HPS as core processor of the system .............................. 30
3.6.3 Video Output of Embedded Linux OS ............................................. 31
3.6.4 Design the backbone of the System .............................................. 33
3.6.5 Enhancements of FPGA .................................................................. 35
Chapter 4 Cyclone V SoC FGPA - Computer System Interconnection and Memory Map 37
4.1 Hard Processor System Interconnection and Address Map .................. 37
4.2 FPGA Sub-System Interconnection and Memory Mapping .................. 39
4.3 HPS – FPGA – Peripherals Interconnection ....................................... 46
4.3.1 HPS Peripheral Region Address Map ........................................... 50
4.4 Computer System Memory Map ......................................................... 51
Chapter 5 System’s Hardware Implementation .......................................... 53
5.1 D8M Camera Kit Reference Design ..................................................... 53
5.1.1 Clock Source and PLL Settings ..................................................... 54
5.1.2 Nios II Processor System .............................................................. 56
5.1.3 D8M Camera Kit video capture and control interface ...................... 60
5.1.4 Video Sub-System ........................................................................... 63
5.1.5 Top-Level Routing and Pin Assignment ......................................... 69
5.1.6 Timing Constraint Configuration ................................................... 72
5.2 HPS as system’s application processor .............................................. 72
5.2.1 HPS Peripheral Pin configuration ................................................... 73
5.2.2 HPS PLL configuration ................................................................. 74
5.2.3 HPS External Memory configuration ............................................. 75
5.2.4 Replacing Nios II with HPS on Qsys system ................................. 75
5.3 Linux OS Video Output interface ......................................................... 76
5.3.1 Frame Reading from HPS .............................................................. 76
5.3.2 Picture in Picture display ............................................................... 79
5.4 Camera video stream reading by Linux OS ......................................... 83
5.4.1 Duplicate video stream ................................................................. 83
5.4.2 Scale and color transformation the video stream ............................ 85
Table of Figures

Figure 1.1 Hog overview ................................................................. 2
Figure 1.2 Cyclone V SoC FPGA ......................................................... 3
Figure 1.3 De1-SoC development board plus D8M camera kit .................. 4
Figure 2.1 Overall motion based detection architecture .......................... 8
Figure 2.2 Part based implementation ............................................... 9
Figure 2.3 Window patch over original image ..................................... 10
Figure 2.4 Filter Centered masks in x and y directions ......................... 10
Figure 2.5 Each cell consists of 64 pixel values .................................. 11
Figure 2.6 Hog histogram ............................................................... 12
Figure 2.7 Block and cell visualization .............................................. 12
Figure 2.8 HOG visualization ......................................................... 13
Figure 2.9 Sample separation of two classes ..................................... 13
Figure 2.10 Good margin between two classes ................................... 14
Figure 3.1 Heterogeneous embedded system before and after SoC FPGAs ... 19
Figure 3.2 Cyclone V SoC FPGA architecture ................................... 19
Figure 3.3 HPS overview ............................................................... 20
Figure 3.4 FPGA Fabric overview .................................................... 21
Figure 3.5 De1-SoC Development Board block diagram ....................... 23
Figure 3.6 Omnivision OV8865 MIPI Interface Auto Focus 8MP Camera Module ................................................................. 24
Figure 3.7 Data conversion from MIPI CSI-2 to Parallel and I2C interfaces ... 24
Figure 3.8 System design example in Design Platform ......................... 25
Figure 3.9 Quartus Design Flow ...................................................... 26
Figure 3.10 Simplified flow of pedestrian detection process (system) ......... 26
Figure 3.11 Integration of HOG plus SVM algorithm on system application .. 27
Figure 3.12 Data exchange through the components ........................... 28
Figure 3.13 D8M Camera Kit reference design from Terasic .................. 29
Figure 3.14 Video Streaming Flow .................................................... 30
Figure 3.15 Integration of Hard Processor System in System Design .......... 31
Figure 3.16 Importing Video Output of Linux OS ................................ 32
Figure 3.17 Integration of an On-Chip ROM Memory ............................ 33
Figure 3.18 Implementation of Frame Writing into On-Chip RAM ............ 34
Figure 3.19 Preprocessing of Captured Frames .................................. 35
Figure 3.20 Stream Blender Design Proposal ..................................... 36
Figure 4.1 HPS Interconnect Block Diagram ..................................... 38
Figure 4.2 HPS Address Space Relationships .................................... 38
Figure 4.3 Platform Designer tool - system creation ............................. 39
Figure 4.4 Address Map tab, in Platform Designer Tool ....................... 40
Figure 4.5 Importing Nios II Processor in system design ...................... 40
Figure 4.6 Address Map after importing Nios II Processor ..................... 40
Figure 4.7 Importing on-chip memory in system design ....................... 41
Figure 4.8 Address Map after importing on-chip memory ..................... 41
Figure 4.9 Master-Slave connection in Platform Designer Tool ................ 41
Figure 4.10 Manually change the Base Address of a Slave .................... 42
Figure 4.11 Assign Base Addresses option in Platform Designer tool ......... 42
Figure 4.12 New base address assignment and address locking ................ 42
Figure 4.13 Address Map after connecting data and instruction master to on-chip slave ..... 43
Figure 4.14 Importing Frame Buffer in system design ......................... 43
Figure 4.15 Address Map after importing Frame Buffer ........................ 43
Figure 4.16 Address map of the system ........................................... 44
Figure 4.17 Design example with Master-Slave relationship ........................................ 44
Figure 4.18 System design example implemented in Platform Designer .......................... 45
Figure 4.19 HPS Configuration Window Instance ....................................................... 46
Figure 4.20 FPGA Interface configuration for HPS component .................................... 47
Figure 4.21 Revised address map table ........................................................................ 47
Figure 4.22 Address Map after connecting Frame Buffer on FPGA to HPS SDRAM interfaces ........................................................................................................... 48
Figure 4.23 Address Map of the FPGA sub-system with the HPS as main processor ........ 48
Figure 4.24 System Design with the HPS as main Processor ........................................ 49
Figure 4.25 System Interconnection and Master – Slave relationship ............................. 51
Figure 4.26 System’s Address Map ............................................................................... 52
Figure 5.1 Clock Source parameter window ................................................................. 54
Figure 5.2 Clock Source’s pin export ............................................................................. 54
Figure 5.3 Block diagram of Qsys System Design with Clock Source IP Core ................. 55
Figure 5.4 Altera PLL parameter window ....................................................................... 55
Figure 5.5 PLL exported clock outputs ......................................................................... 56
Figure 5.6 Block diagram of Qsys System Design with integration of Altera PLL IP Core ... 56
Figure 5.7 Nios II instance selection .............................................................................. 57
Figure 5.8 On-Chip Memory parameter window ........................................................... 57
Figure 5.9 Block diagram of Qsys System Design with integration of Nios II and on-chip memory ........................................................................................................... 58
Figure 5.10 Interval Timer’s parameter window .............................................................. 58
Figure 5.11 JTAG UART parameter window .................................................................. 59
Figure 5.12 PIO Core parameter window instance for led control ................................. 59
Figure 5.13 Block diagram of Qsys System Design with integration of PIO Controller IP cores .................................................................................................................. 60
Figure 5.14 Terasic Camera IP core instance .................................................................. 60
Figure 5.15 OV8865 Sensor array region color filter layout ........................................... 61
Figure 5.16 Terasic Camera’s IP core parameter window ............................................... 62
Figure 5.17 I2C Master IP cores Instances on system implementation ............................ 62
Figure 5.18 D8M’s Reset and Power Down system integration ...................................... 62
Figure 5.19 Block diagram of Qsys System Design with integration of Terasic’s Camera Interface ............................................................................................................... 63
Figure 5.20 Frame Buffer Block Diagram ...................................................................... 64
Figure 5.21 Frame Buffer Parameter Window .................................................................. 64
Figure 5.22 SDRAM Controller block diagram ............................................................. 65
Figure 5.23 SDRAM Controller IP core parameter window .......................................... 65
Figure 5.24 Block diagram of Qsys System Design with integration of Frame Buffer .......... 66
Figure 5.25 Terasic Autofocus IP core ports ................................................................... 67
Figure 5.26 Block diagram of Qsys System Design with integration of Autofocus IP Core ... 67
Figure 5.27 Clocked Video Output parameter window instance ..................................... 68
Figure 5.28 Block diagram of Qsys System Design with integration of Video Output ......... 69
Figure 5.29 FPS Monitor Block Diagram ....................................................................... 69
Figure 5.30 Top Level Entity block diagram of reference design ..................................... 70
Figure 5.31 Pin Planer Tool instance ............................................................................... 71
Figure 5.32 HPS IP core instance and exported ports .................................................... 72
Figure 5.33 AXI Bridges configuration from HPS IP core .............................................. 73
Figure 5.34 HPS IP core Peripheral Pins configuration .................................................. 73
Figure 5.35 HPS Peripheral physical controller connections ......................................... 74
Figure 5.36 Clock configuration instance of the HPS IP core ........................................ 74
Figure 5.37 Lightweight HPS to FPGA AXI Master – Slave connection .......................... 75
Figure 5.38 Block diagram of Qsys System Design with integration of HPS IP core ........ 76
Figure 5.39 Frame Reader IP core instance ................................................................. 77
Figure 5.40 Color Plane Sequencer Streaming Input ................................................... 78
Figure 5.41 Color Plane Sequencer Streaming Output ............................................... 78
Figure 5.42 Block diagram of Qsys System Design with integration of Frame Reader and CPS .................................................................................................................. 79
Figure 5.43 Picture in Picture display format ............................................................... 80
Figure 5.44 Alpha Blending Mixer IP core configuration ............................................ 80
Figure 5.45 Clock Video Output IP core reconfiguration ........................................... 81
Figure 5.46 Block diagram of Qsys System Design with integration of Alpha Blending Mixer .................................................................................................................. 82
Figure 5.47 Top Level Entity block diagram of system ............................................... 83
Figure 5.48 Parameter window of the CPS IP core as Avalon Stream duplicator – Input configuration ........................................................................................................ 84
Figure 5.49 Parameter window of the CPS IP core as Avalon Stream duplicator – Output configuration ....................................................................................................... 84
Figure 5.50 Block diagram of Qsys System Design with Integration of CPS IP core as video stream duplicator ................................................................. 85
Figure 5.51 Scaller II IP core parameter window......................................................... 86
Figure 5.52 Color Space Converter UPIP core parameter window............................ 86
Figure 5.53 Avalon-ST IP core parameter window ....................................................... 87
Figure 5.54 DMA Controller IP core parameter window........................................... 88
Figure 5.55 On-chip RAM Memory parameter window .............................................. 89
Figure 5.56 Block diagram of Qsys System Design with integration of Scaller IP core, DMA controller IP core and on-chip memory .................................................. 90
Figure 5.57 Block diagram of integration of Qsys System, Top-Level Design and FPGA HPS interconnections on the Cyclone V-SoC FPGA chip ................................................. 91
Figure 6.1 Cmake GUI instance for choosing build and source folder ....................... 94
Figure 6.2 Cmake GUI compiler setup ...................................................................... 95
Figure 6.3 Cmake GUI compiler choice ................................................................... 95
Figure 6.4 Cmake GUI configuration of library install path ....................................... 96
Figure 6.5 Creating a new Linux application project on Arm DS-5 ............................ 97
Figure 6.6 Project Properties build settings window on Arm DS-5 ............................ 98
Figure 6.7 Initial screen of De1-SoC Linux system's video output ............................... 100
Figure 6.8 Commands executed from Putty terminal .............................................. 101
Figure 6.9 System's monitor display after imshow application executed on De1-SoC 101
Figure 6.10 Clock tree of D8M Camera Kit[10] .......................................................... 106
Figure 7.1 Perform Pedestrian detection on frames fetched from on-chip RAM ....... 112
Figure 7.2 Demo of the system with D8M camera capturing a video playing from a smartphone ................................................................................................. 112
Figure 7.3 Demo of the system with D8M camera capturing the lab room ............... 113
List of Tables

Table 4.1 Common Address Space Regions .................................................................................................................. 39
Table 4.2 HPS Peripherals Table ................................................................................................................................. 50
Table 4.3 HPS 2nd Peripheral Table .............................................................................................................................. 50
Table 5.1 Pin Assignment of Push-buttons .................................................................................................................... 71
Table 6.1 Alpha Blending Mixer Control Register Map .................................................................................................. 104
Chapter 1  Introduction

The development of object detection systems has been rapid in recent years, widely used in various fields. Part of this development came through the research and development of Machine Learning algorithms, coupled with the development of advanced image processing algorithms.

With the term Object Detection System, we describe a system that can detect objects, commonly through a visual representation of an area (image, video). Vision-based detection is a very important element in computer systems. It enables the development of systems capable of describing a site and finding points of interest automatically. Until a few years ago, detection procedures could not be performed automatically by a system as there was a requirement for human involvement in the procedures of each application.

In most systems that include computer vision processes, object detection is most likely to be the first task that is being performed, because it is easier to obtain and provide more information about the area and the objects in it. Once an object is detected, it is easier to obtain further information about it. In example, it can apply various tasks in the object such as tracking, recognition (who’s face is it), extraction of characteristics (skin color, man or woman) among other content information.

1.1  Applications

Many applications have been adapted object detection with most popular, human-computer interaction (HCI), robotics, consumer electronics (e.g. smart-phones), security (e.g. recognition, tracking), retrieval (e.g. search engines, photo management) and transportation (e.g. autonomous and assisted driving). Each of these applications has different requirements based on the class of the object to detect and processing time.
The most common class of detected object—among the above applications—is human. This is because extracting information about humans in a scene is critical to most of the applications. This type of detection is described as Pedestrian Detection (in Advanced Driver Assistance Systems), or Human Detection (in security and surveillance systems).

1.1.1 Pedestrian Detection and HOG+SVM

Detecting human from video images is a challenging task, as people may appear in different clothing and poses. Occlusions, unstructured background and uneven lighting condition especially for the outdoor environment may complicate the task. The basic requirement for an effective human detection system is to extract the perfect human shape from the background of the image. Locally normalized Histogram of Oriented Gradient algorithm proposed by Dalal & Triggs [1] presents excellent detection results. The input image is represented by 7 x 15 blocks. The detection window will slide over the grid of cells by overlapping each other. The histograms of gradient orientation are then computed over each block of the sliding window. Each cell consists of a 9-bin Histogram of Oriented Gradient. The concatenation of the histograms from each block forms the HOG feature vector.

The features vector will then be directed to a linear SVM classifier for classification purpose. Experimental results from the combination of HOG and linear SVM classifier show that the representation is effective for human classification. Figure 1.1 summaries the implementation proposed by Dalal & Triggs. Even that the operation of HOG is a quite heavy task, in this thesis we adopt the proposed algorithm from the OpenCV library.

![Hog overview](image)

**Figure 1.1 Hog overview.**

1.1.2 Real time performance

Another key characteristic of an object detection system is the processing time. There is discrimination of object detection systems based on the processing time as it can be described as off-line, on-line or real-time. Offline systems can store the images and process them after some time, as online systems push the image to a server that is capable to perform the detection process and pull the result.

But there are many applications, where the time for the system to perform the detection of an object is critical, so there is a need for performing real-time detection. Real-time systems have to process the video frames “on the fly”. The detection has to be done in a short period of time, desirable multiple times per second.

In the previous section was described one of the most critical points in object detection systems, the time the system needs to process and locate the object in an image. Real-time systems are mandatory in applications where the scene changes quickly, for example, in industrial environments where an object moves really fast or in vehicle applications where the vehicle goes fast.

In such a system a prerequisite is the operation of a camera capable of capturing multiple frames per second (30+ fps). Even if the system does not exploit all frames for detection, taking multiple frames per second enables the system to remove any frames that have been destroyed without losing information. This, combined with heavy computational algorithmic processing for each frame, increases dramatically the computational complexity and requires very fast processing of the algorithms.
Normally, general-purpose computer systems can perform real-time detection. This can be done not only with the use of a powerful CPU but also with the collaboration of the GPU of computer.

1.1.3 Embedded pedestrian detection approach
But what is like when things come to embedded system solutions? Like with general purpose computers, a similar design is followed in embedded systems.

In embedded systems, we meet FPGAs chips as a solution to high-volume applications such as industrial motor control drivers, protocol bridging, video converter, and video capture cards, and handheld devices. In this case, the implementation of a computer vision system is also based on an FPGA. As a customizable logic circuit, FPGAs can accelerate an image processing algorithm, by implementing procedures in a more parallel way as it can pipeline the data process of a frame of a video stream.

But for some complicated algorithms, the flexibility of FPGA is insufficient. Here comes the need of a microprocessor. In embedded systems, there are microcontrollers that are capable of implementing complicated algorithms most commonly programmed in C-based language. As a microcontroller with a kind of special structure, Digital Signal Processors (DSP) can make up for deficiency of FPGA by its flexible addressing mode and powerful calculation abilities.

Such implementations mean that in computer vision systems there is a need to use both an FPGA chip for heavy video tasks alongside a DSP, connected to each other, share the resources and assembled on the same board. The result is a relatively large board, with the inherent reliability issues and a high cost [2].

1.2 Thesis approach
This kind of implementation adds complexity to the design of such demanding embedded systems. Fortunately, in recent years, they have launched SoC FPGA chips. SoC FPGA is a “System on Chip” FPGA which hosts both an FPGA fabric alongside a Hard Processor System (HPS), hard-wired in the same silicon. HPS consists of a dual-core Arm microprocessor with embedded peripherals, multiport memory controllers and plenty of I/Os. This eliminates the use of multiple processing chips on a board.

SoC FPGAs are ideal for building heterogeneous computing systems. This was a great element in the design and implementation of computer vision systems in embedded platforms. This thesis makes use of the Cyclone V SoC FPGA chip of Altera. The chip is hosted on a development board called De1-SoC of Terasic. For the video input of the system, there is a camera module called D8M-GPIO of Terasic. D8M-GPIO is a camera kit.
with a 2x20 pin GPIO connector interface. It carries an 8-megapixel MIPI camera sensor and a MIPI decoder. For video output, the system uses a VGA controller and connector. Both input and output are directly connected on the FPGA part of the SoC FPGA chip.

![Figure 1.3 De1-SoC development board plus D8M camera kit.](image)

### 1.3 Structure of thesis

This thesis is organized as follow. Chapter 2 presents the detection challenges and the theoretical approach of the Histogram of Oriented Gradients algorithm and Support Vector Machine algorithm. It also presents the implementation of pedestrian detection using OpenCV in a general a purpose computer.

Chapter 3 presents the hardware of the system (camera, SoC, memory chips, VGA controller, etc.) and the connections between them, as well as a detailed overview of the real-time pedestrian detection system architecture. Especially in Section 3.6, there is a detailed overview of the design flow of the system.

Chapter 4 describes the architecture behind the interconnections of Cyclone V SoC FPGA and the memory mapping of a SoC system. It goes through a reference design and presents the details of the connections between the HPS and the FPGA fabric.

Chapter 5 gives the implementation of the hardware sub-system. The implementation of hardware is mostly based on Qsys system design using IP Cores and Verilog HDL. Every step of the implementation is being followed by a presentation of the hardware sub-system in block diagrams.

Chapter 6 describes the implementation of the software application which runs on Linux OS on the HPS side of the Cyclone V SoC FPGA. This is done by presenting the C++ main
program function which initializing and operating the real-time pedestrian detection system. Chapter 6 also describes the procedure behind the OpenCV cross-compilation.

This thesis concludes with Chapter 7 which presents the results of the system, the operation, problems that were faced on the implementation and possible future work.
Chapter 2  Pedestrian Detection

In the introductory chapter was mentioned, that one of the most common operations in computer vision systems is pedestrian detection. This information of the presence of a person in an image, must be extracted from the image but requires complex algorithms in order to achieve that. This chapter provides an overview of the pedestrian detection algorithms presented in the past years and the procedures behind the pedestrian detection process when the most common pedestrian detection technique, HOG plus SVM is being used.

2.1 Detection challenges

The basic idea behind pedestrian detection is how a system can separate people from the environment and other objects that are likely to constitute the scene of an image. From time to time, algorithms have been proposed to identify individuals in an image, relying on the outline of the person or sometimes on the members of the body that makes up the human. There are also algorithms that depend on the human motion and can be found in applications with a static video camera. Following are some of the algorithms presented by category.

2.1.1 Motion-based detection

This kind of approach is mostly addressed in surveillance systems and that's because it requires a stationary video camera with minimum changes in lighting conditions. The algorithm proposed by O. Barnich, S. Jodogne, and M. Van Droogenbroeck\textsuperscript{[3]}, performs motion segmentation based on adaptive background subtraction of the images from the input video stream and produces binary silhouettes of the moving objects.

In the produced silhouettes it is necessary to extract their features based on morphological size distributions which are based on grouping rectangles that fit in the silhouette and
consists of five features: width, height, two relative coordinates of its center, and the percentage of uniquely covered pixels to its area.

Finally, it becomes possible to adopt a machine learning algorithm. They proposed to depend on Extremely Randomized Trees, for learning and classification procedures. The overall architecture of the algorithm is presented in the following figure.

![Overall motion based detection architecture](image)

**Figure 2.1** Overall motion based detection architecture.

### 2.1.2 Part-based Detection

Part-based pedestrian detection systems can describe a scene with segmentation of the image. This approach can provide a more complex model for pedestrian detection. A real-time pedestrian detection system of this kind was proposed by Hyunggi Cho, Paul E. Rybski, Aharon Bar-Hillel, and Wende Zhang\(^{(4)}\) and is intended for use in automotive applications. It describes the image in multiple scales, computing features at each one, performing classification at all possible locations and finally performing non-maximal suppression to generate the final set of bounding boxes. Figure 2.2 describes the procedure.
As intended mostly for automotive applications, this implementation also takes into account
the known position of the camera. It can perform safe geometric constraint analysis based
on known camera calibration information for efficient search. This information is very useful
because the algorithm can accelerate the process of detection as it excludes areas that are
impossible for a person to be present. This also helps on reducing significantly the number
of false positive detections.

2.1.3 Holistic Detection

Holistic pedestrian detectors are the one that is trained to detect pedestrians in an image by
scanning the whole image. This kind of approach can reliably detect people in a static image
without motion information. The first detector of this kind was proposed by C. Papageorgiou,
T. Evgeniou, and T. Poggio[5] and is based on a representation that encodes local intensity
differences in an image.

The system is based on a novel object representation that uses projections of the object
images onto a dense Haar wavelet basis which efficiently encodes structural features at
different scales over the entire image. Each wavelet coefficient corresponds to a single
feature and responds to vertical, horizontal and diagonally oriented intensity differences.
These features consist of 1326 components for a 128x64 pattern and are used to train a
Support Vector Machine (SVM) classifier.

This pedestrian detection system led to big research which produced many published papers
from modification or extension of this approach. The most notable is, Histogram of Oriented
Gradients by N Dalal and B. Triggs[1]. They have studied the issue of feature sets for human
detection, showing that locally normalized Histogram of Oriented Gradient descriptors
provides excellent performance.

The HOG feature vectors are extracted from the detector window -which scans the image-
and is tiled with a group of overlapping blocks. They’ve also used linear SVM as a baseline
classifier throughout the study, in order to feed it with the combined feature vectors.

Because of the detection performance the HOG plus SVM implementation offers to a
system, it was decided to adapt it, in this thesis system design. The system makes use of
the HOG plus SVM from OpenCV library. The following sections provide a more detailed
overview of the procedures of feature extraction as well as the learning and classifying
methods. There is also a presentation of the capabilities of HOG plus SVM with the use of
OpenCV library from examples of pedestrian detection running on general purpose
computer.

2.2 Histogram of Oriented Gradients

This section provides a more detailed overview of the HOG feature extraction. Firstly, there
is a short description of the procedure, which was also presented in figure 1.1. The original
paper suggests starting with a preprocessing of the image, with normalization on the gamma and color. Next step is the computation of the centered horizontal and vertical gradients with no smoothing and then the computation of gradient orientation and magnitude of the given image. For colored images, there is a selection of the color with the highest gradient magnitude for each pixel.

The HOG descriptor is a local descriptor and operates on a 64x128 pixel patch of the image. Other sizes can also be applied, with the only constraint to have a fixed aspect ratio of 1:2. For the 64x128, there is a division of the image into 16x16 pixels blocks with 50% overlap (105 blocks). Each block should consist of 2x2 cells with the size of 8x8 pixels. Then there is the quantization of the gradient orientation into 9 bins and weights depending on the gradient magnitude. Finally, the histogram is concatenated into a large one (Feature dimension: 105x4x9=3780).

**Figure 2.3** Window patch over original image.

### 2.2.1 Computing Gradients

Based on a 64x128 image patch the system starts by calculating HOG descriptor for the given patch. Firstly, there is a need for calculating the horizontal ($g_x$) and vertical ($g_y$) gradients. After several tests for computing gradients with Gaussian smoothing, Dalal and Triggs came to use simple 1-D $[-1, 0, 1]$ masks at $\sigma = 0$ (none smoothing). Using larger masks always seemed to decrease performance and smoothing damages it significantly.

![Filter Centered masks in x and y directions.](image)

**Figure 2.4** Filter Centered masks in x and y directions.
The next step is to find the magnitude and the direction of gradient by using the following equations.

\[ g = \sqrt{g_x^2 + g_y^2} \]

\[ \theta = \arctan \frac{g_y}{g_x} \]

### 2.2.2 Compute histogram of gradients (cells)

After computing the gradients of the pixels the system must divide the sliding window of 64x128 into 8x8 pixels cells, as it is illustrated in the following figure.

![Figure 2.5 Each cell consists of 64 pixel values.](image)

Now the calculation of histogram of gradients for every cell is performed. From every cell, there have been extracted 2 arrays with gradient magnitude and gradient direction, each for every pixel. These couple of 2D arrays with 8x8=64 values is representing each cell. The next step is to create a histogram of gradients in these 8x8 cells. The histogram consists of 9 bins, representing gradient magnitude values, shared between 0° and 180° gradient directions, with a step of 20°.

Here is one example of how histogram of gradients is computed, in a pixel with gradient magnitude 60, and gradient direction of 85°. The magnitude value must be split into the closest couple of bins based on their distance of bin center (for the bin 0-19°, 10° is the centered value). Same procedure for the gradient direction with value 85°, where the distance to the “bin 70°” and “bin 90°”, is 15° and 5° respectively.

Next step is the calculation of the ratios which are 5/20 = 1/4 and 15/20 = 3/4. For the magnitude value of 60, the contribution for the “bin 70°” is 60 * 1/4 = 15 and for the “bin 90°” is 60 * 3/4 = 45. This process is executed for every pixel and the values resulted are summed up to create the histogram of gradients. The following figure presents the histogram.
2.2.3 Compute histogram of gradients (block normalization)

After calculating the 9-bin histogram for every cell, block normalization must be performed. The next step is to divide the sliding window of 64x128 into 16x16 pixels blocks with 50% overlap. Each block consists of 2x2 cells. In the whole image patch, there are 16x16 pixels blocks with 50% overlap. That means \(7 \times 15 = 105\) blocks in total. The figure that follows presents the block selection.

Every block has now 4 histograms of gradients which must be concatenated into a 36x1 vector. After that, block normalization is performed. This normalization is done in order to remove the effect of local light differences. This step helps the process because the descriptor has to be independent of lighting variations. This could be done in every cell but tests of the algorithm proved that normalizing into a larger area can give better performance. The size of the block is not randomly selected, as it can describe areas in the image that contain a person (or part of it, such as his head, arms), more accurately.
The normalization of the 36-sized array is done, by dividing it by $\sqrt{v_1^2 + v_2^2 + \cdots + v_{36}^2 + 1}$ where $v$ is the array.

2.2.4 Final feature vector and HOG visualization
The final feature vector will concatenate all those 105 block histograms into an array. The 105x36=3780-sized feature vector describes the detecting window.

In the following figure is presented the visualization of each block histogram that consists the final feature vector which is extracted from images that contain a person. In the plot is presented the dominant direction according to the edges in each block.

![HOG visualization](image)

*Figure 2.8 HOG visualization.*

2.3 Linear Support Vector Machine
This section introduces the Linear Support Vector Machine (Linear SVM). This supervised machine learning algorithm in conjunction with the Histogram of Oriented Gradient can implement Pedestrian Detection over an image.

In a few words, SVM is a classifier able to separate data into classes, based on its training output. The training output consists of an optimal hyperplane which categorizes the training data based on their label. In this case –on pedestrian detection- training data are the feature vectors from HOG, extracted from images with human and non-human objects and the hyperplane is a line that divides the two parts where each class (human and non-human) lay in either side. An optimal hyperplane is the one that represents the largest margin between the two classes[^6].

![Sample separation of two classes](image)

*Figure 2.9 Sample separation of two classes.*

In the training process, a large number of positive (human) and negative (non-human objects) images are fed in the SVM. SVM classifier projects the feature vectors of these two classes in the feature space. Based on their extracted features, the two classes will normally form two different groups with their centers and distribution separated one from each other.
The algorithm will then try to find the best fitting line between the two classes in order to separate them. This line is the hyperplane of the SVM.

An example is given with training dataset of \( n \) points of the form \((x_1, y_1), \ldots, (x_n, y_n)\) where \( y_i \) is either 1 or \(-1\), which indicates the class of the point \( x_i \). The exported hyperplane is the one that divides the group of points \( x_i \) where \( y_i = 1 \) from the group of points \( x_i \) where \( y_i = -1 \), based on the maximum distance between the hyperplane and the nearest points \( x_i \) from either group.

![Figure 2.10 Good margin between two classes.](image)

Usually, the negative training data are at least twice the positive ones. Then SVM is tested on the negative image data. This will produce the hard-negative samples. Hard negative samples consist of these images that the classifier made a false positive detection after the initial training of the classifier. Next step is to re-train the algorithm with the same positive samples and the negative plus the hard-negative samples. After that, SVM is capable to classify an image and tell if a human exists (or not) with a big success rate.

In the original paper of Histogram of Oriented Gradients, Dalal and Triggs used Linear SVM with 1239 positive training examples and 12180 negative plus hard negative training examples. In this paper, the authors note, that the detector cues mainly on the contrast of the silhouette contours against the background.

### 2.4 Pedestrian Detection with OpenCV library

The current section presents the functions and structures that OpenCV library provides for the implementation of pedestrian detection with C++ code. OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and accelerate the use of machine perception in commercial products.

#### 2.4.1 HOG plus SVM detector initialization

OpenCV library provides a pre-trained HOG plus Linear SVM structure which is able to perform pedestrian detection on images. This structure is applied in C++ applications by calling the main constructor of the `HOGDescriptor` structure. The default constructor is presented in the following lines.

```cpp
CV_WRAP HOGDescriptor() :
    winSize(64,128), blockSize(16,16), blockStride(8,8),
    cellSize(8,8), nbins(9), derivAperture(1), winSigma(-1), histogramNormType(0),
    L2HysThreshold(0.2), gammaCorrection(true), free_coef(-1.f), nlevels(64),
    signedGradient(false)
```

The arguments of the constructor define the parameters for the computation of Histogram of Oriented Gradients. The `nbins` value represents the number of bins of the histogram, the
$\text{cellSize}$ value defines the cell window which by default is $8 \times 8$ pixels, $\text{blockSize}$ determines the pixels consisting each block which by default is $16 \times 16$ (4 cells), and $\text{winSize}$ is the size of the detecting window.

For the initialization of the structure, users must set the coefficients for the Linear SVM classifier, from a dataset with coefficients from a pre-trained detector. In order to set the coefficients of the Linear SVM classifier, $\text{setSVMDetector}$ function is being used. OpenCV 2.4 library provides two datasets with coefficients generated from pre-trained detectors for pedestrian detection, the first suitable for detecting windows of the size $64 \times 128$ pixels (Default People Detector) and the second suitable for detecting windows of the size $48 \times 96$ pixels (Daimler People Detector).

The second pre-trained pedestrian detector provides the option of performing pedestrian detection on images with a lower resolution. For this kind of images, it is recommended to set the Linear SVM detector with the coefficients from the Daimler People Detector because of its window size. This setting also demands to initialize the HOGDescriptor with a $\text{winSize}$ value of $\text{Size}(48 \times 96)$.

```cpp
Size win_size(48, 96);
Size block_size(16, 16);
Size block_stride(8, 8);
Size cell_size(8, 8);
int nbins = 9;

cv::HOGDescriptor hog(win_size, block_size, block_stride, cell_size, nbins);

hog.setSVMDetector(hog.getDaimlerPeopleDetector());
```

2.4.2 Perform pedestrian detection on image

At this point, the pedestrian detector is initialized and ready to accept images. The process of pedestrian detection is performed with the use of the $\text{detectMultiScale}$ function, of the HOGDescriptor structure. For the given HOG plus SVM structure, the $\text{detectMultiScale}$ function detects pedestrians of different sizes in the input image. The detected pedestrians are returned as a list of rectangles. The definition of the $\text{detectMultiScale}$ function is presented in the following lines.

```cpp
virtual void detectMultiScale(InputArray img, CV_OUT std::vector<Rect>& foundLocations, double hitThreshold = 0, Size winStride = Size(),
    Size padding = Size(), double scale = 1.05,
    double finalThreshold = 2.0, bool useMeanshiftGrouping = false) const;
```

The first two arguments of the function are the input image to perform pedestrian detection and the vector which describes the returned rectangles which define the position of the detected pedestrians, respectively.

Detect multiscale generates multiple resized copies of the given image, based on the scale parameter of the function. Then it performs the pedestrian detection process in the generated images. This process is conducted in order to detect pedestrians in different scales. For larger scale value, the function will evaluate less resized images which can make the detector run faster. However, having a large value on the scale parameter can lead to loss of detecting pedestrians in the image. Therefore, having small value on the scale parameters makes the detector occupying too much time for its execution and can lead to many false-positives detections.

The returned rectangles represent the bounding box of the scaled sliding detecting window if the result of the HOG calculation in the detecting window reveals the existence of a person. The detecting window size was discussed previously in the initialization of the hog
constructor. The step of the sliding detecting window is defined from the \textit{winStride} argument of the \texttt{detectMultiScale} function.

For the detecting window, the paper of Dalal and Triggs \cite{1} also suggests, adding \textit{padding} surrounding the detecting window, prior to HOG feature extraction and classification. This is done with the argument of the \texttt{detectMultiScale} function, \textit{padding}.

For tuning the \texttt{detectMultiScale} function on the false-positive detections the \textit{hitThreshold} and \textit{finalThreshold} arguments are being used. The \textit{hitThreshold} parameter stands for the distance between features and SVM classifying plane.

The following code presents the use of the \texttt{detectMultiScale} function. The \textit{frame Mat object} handles the image to perform pedestrian detection, and the vector \texttt{rects} defines the returned rectangles.

\begin{verbatim}
Mat frame;
std::vector<Rect> rects;

double hit_thr = 1;
Size win_stride(4,4);
Size padding(8,8);
double scale = 1.5;
const int final_thr = 2;

hog.detectMultiScale(frame, rects, hit_thr, win_stride, padding, scale, final_thr);
\end{verbatim}

The last step is to draw the returned rectangles in the image. The following lines of code present the function \texttt{rectangle}, which draws the given by its second argument rectangles in the \textit{frame} image. The color of the rectangles is defined in the third argument of the \texttt{rectangle} function.

\begin{verbatim}
for (size_t i = 0; i < rects.size(); i++){
    rectangle(frame, rects[i], CV_RGB(0, 255, 0), 1);
}
\end{verbatim}
Chapter 3  System Design

Embedded Pedestrian Detection Systems, bring great challenges on the design process of before-mentioned systems. Based on the requirements of such systems, the specifications of this system extracted, which in their turn led to decide the system's hardware. The following sections provide information about the System Hardware, the Development Tools and the chapter concludes with the design progress and the decisions leading to the final form of system architecture.

3.1  System Requirements

In the first chapter, it was described some fast-growing markets such as, surveillance systems, Advanced Driver Assistance Systems, and others, which started to adapt and integrate pedestrian detection techniques on their systems.

The basic requirement of these systems is to capture video (video frames) from a camera and output the result of pedestrian detection on the captured frames on a screen. They also require,

- real-time detection,
- high reliability,
- easy integration (limited size and weight),
- serial output interfaces for outputting the result,
- low power consumption,
- low cost.

Real-time processing is critical for such systems because they “act in the field”. Because of their necessity and their key role on these systems, reliability is also very important. Integration in cars, surveillance cameras or others systems that have limited space and power supply, size and power consumption are taken seriously into account. Last, but not
least, cost and “time to market”, play an important role in the design and implementation of such systems.

3.2 System Specification

Based on the above requirements, this section presents the specifications of the system. Keeping in mind the constraints, such as limited space, power consumption and low cost, embedded solution, is a one-way choice. Trying to build an Embedded Real-Time Pedestrian Detection System comes up with great challenges. As mentioned in Section 1.2, such kind of implementation brings the requirement for fast computational processes, in a big amount of data, like in our case, a video stream. Traditionally such systems are implemented with heterogeneous systems that integrate DSPs along with FPGAs that share the load of the detection process. In this thesis, we decided to build a system on a SoC FPGA. Based on System on Chip (SoC) Architecture and Hardware-Software co-design approach, every process of the system –except image capturing- will be executed on a single SoC FPGA chip. Seeking for the options we had for video capturing, a camera kit was the best solution, which is connected directly on the FPGA part of the SoC FPGA chip. The resolution of the input frames must be at least 640x480 RAW RGB @ 30fps.

Summing up system specification we have,

- Input resolution from a camera at 640x480 RAW RGB @ 30fps
- SoC System Architecture
- Plug and play (no user interaction)
- Video output with detection on captured frames and other information
- Low power consumption
- The small size of the application board

3.3 System Hardware

This section presents the hardware that composes the pedestrian detection system. The thesis system approach adopts De1-SoC development board of Terasic which brings a Cyclone V SoC FPGA 5CSEMA5F31C6 chip, alongside DDR3 memory, video and audio capabilities, Ethernet networking, and much more. This system also makes use of a camera kit, D8M by Terasic. D8M camera kit is connected through a parallel port on the FPGA and consists from a MIPI camera sensor and a MIPI decoder.

3.3.1 What is SoC FPGAs?

Beginning this presentation with the heart of the system -Cyclone V SoC FPGA chip- this section describes the architecture and the ideas unfolding behind it. SoC FPGAs chips development is based on System on Chip Architecture. A SoC chip integrates all components of an electronic computer system, such as a CPU, memories, I/Os and may contain DMAs, radio controllers, DAC, and advanced peripherals such as GPU, Wi-Fi module and others, everything built in a single chip [7].

The basic idea behind the SoC FPGA device family was to build a powerful SoC chip that could handle heavy tasks, with the most efficient way. Following the architecture on such demanding heterogeneous embedded systems, where processors work together with FPGAs, SoC FPGAs launch was a step further for such systems to provide higher integration, lower power consumption, smaller board size, and higher bandwidth communication between the processor and the FPGA.
SoC FPGAs high-level management functionality of Processors and the stringent, real-time operations, extreme data processing and interface functions of an FPGA, integrated into a single device, makes them suitable to build on it, a real-time pedestrian detection system.

3.3.2 Cyclone V SoC FPGA.

Cyclone V SoC FPGA device family is presented by Altera as a solution in the SoC FPGA area. The processors in these devices are fully dedicated “hardened” processor subsystems. Altera describes it as Hard Processor System. It employs a full-featured dual-core ARM® processor with a memory hierarchy and dedicated peripherals that largely boot, run, and act like any “normal” ARM processor [8].

More specific, HPS consists of:

- an ARM Cortex-A9 application processor that can run with a single or dual-core with up to 925 MHz,
- L1 Cache with 32KB data memory and 32KB instruction memory and L2 Cache with 512KB unified memory, with ECC,
- a Memory Management Unit,
a Floating Point Unit, an Acceleration Coherency Port, an Interrupt Controller, an 64KB on-chip RAM with ECC, a Direct Memory Access Controller, an External Memory Controller that supports LPDDR2, DDR2, DDR3L, DDR3 types of memory, with 16 or 32-bit ECC and 400MHz max frequency

Moreover, HPS brings a wide range of hardened embedded peripherals that eliminates the need to implement these functions in programmable logic, leaving more FPGA resources for application-specific custom logic while it manages to reduce power consumption. These peripherals are:

- one NAND controller
- two 10/100/1G Ethernet controller
- two USB 2.0 On the Go (OTG) controller
- one SD/MMC/SDIO controller
- two UART
- two CAN controller
- one Quad SPI controller
- two SPI master, 2x SPI slave controller
• four I2C controller
• four 32 bit general-purpose timers
• two 32 bit watchdog timers

On the other side of the chip, there is an FPGA fabric that consists of 85K Logic Elements (LE) and a shared, with the HPS, hardened multiport memory controller. FPGA logic fabric gives the opportunity to adapt quickly to varying or changing interfaces and protocols standards and to build a system based on hardware components in order to achieve acceleration of time-critical algorithms, implemented with custom or preconfigured IP cores. FPGA fabric also brings:

• 3,970 M10K and 480 MLAB Embedded Memory Blocks,
• 87 Variable Precision DSP Hard IP Blocks,
• 174 18x18 Multipliers
• 6 PLLs

Connections between the HPS and the FPGA fabric is established through a high-bandwidth interconnect. High-throughput datapaths between the HPS and FPGA fabric provide interconnect performance not possible in two-chip solutions. This tight integration provides over 100Gbps peak bandwidth and significant power savings by eliminating the external I/O paths between the processor and the FPGA.

The HPS–FPGA bridges, which support the Advanced Microcontroller Bus Architecture (AMBA®) Advanced eXtensible Interface (AXI™) specifications, consist of the following bridges:

• FPGA-to-HPS AXI bridge—a high-performance bus supporting 32, 64, and 128-bit data widths that allows the FPGA fabric to issue transactions to slaves in the HPS.
• HPS-to-FPGA AXI bridge—a high-performance bus supporting 32, 64, and 128-bit data widths that allows the HPS to issue transactions to slaves in the FPGA fabric.
• Lightweight HPS-to-FPGA AXI bridge—a lower latency 32-bit width bus that allows the HPS to issue transactions to slaves in the FPGA fabric. This bridge is primarily used for control and status register (CSR) accesses to peripherals in the FPGA fabric.

The HPS–FPGA AXI bridges allow masters in the FPGA fabric to communicate with slaves in the HPS logic and vice versa. For example, the HPS-to-FPGA AXI bridge allows data
sharing between the memories instantiated in the FPGA fabric with one or both cores of the microprocessor in the HPS, while the FPGA-to-HPS AXI bridge allows logic in the FPGA fabric to access the memory and peripherals in the HPS. Each HPS–FPGA bridge also provides asynchronous clock crossing for data transferred between the FPGA fabric and the HPS.

Last but not least, the device can support any boot sequence, such as Processor first, FPGA first or both simultaneously.

### 3.3.3 De1-SoC Development Board

The De1-SoC Development Board gives the opportunity to build a fully functional system using its features, which are integrated around the Cyclone V SoC chip. Based on the capabilities the SoC chip can offer, this development board integrates a wide range of peripherals and I/Os that are directly connected to FPGA Fabric or on the Hard Processor System of the Cyclone V SoC chip [9].

Figure 3.5 presents the De1-SoC board block diagram which provides the following hardware accessed by the FPGA Fabric and the Hard Processor System.

- **FPGA**
  - Altera Cyclone® V SE 5CSEMA5F31C6N device
  - Altera serial configuration device – EPCS128
  - USB-Blaster II onboard for programming; JTAG Mode
  - 64MB SDRAM (16-bit data bus)
  - four push-buttons
  - ten slide switches
  - ten red user LEDs
  - six 7-segment displays
  - four 50MHz clock sources from the clock generator
  - 24-bit CD-quality audio CODEC with line-in, line-out, and mic-in jacks
  - VGA DAC (8-bit high-speed triple DACs) with VGA-out connector
  - TV decoder (NTSC/PAL/SECAM) and TV-in connector
  - PS/2 mouse/keyboard connector
  - IR receiver and IR emitter
  - two 40-pin expansion header with diode protection
  - A/D converter, 4-pin SPI interface with FPGA

- **HPS**
  - Dual-core ARM Cortex-A9 MPCore processor
  - 1GB DDR3 SDRAM (32-bit data bus)
  - one Gigabit Ethernet PHY with RJ45 connector
  - two-port USB Host, normal Type-A USB connector
  - micro SD card socket
  - accelerometer (I2C interface + interrupt)
  - UART to USB, USB Mini-B connector
  - warm reset button and cold reset button
  - one user button and one user LED
  - LTC 2x7 expansion header
Although there is a wide range of peripherals, this system focuses on the multimedia and debugging peripherals. From the FPGA part, the system makes use of Video DAC and VGA Connector, the 40-pin GPIO (for camera input), the 64MB SDRAM and push buttons. From the HPS part of the chip, the system adapts the 1GB DD3 SDRAM, micro SD card input, the Ethernet and UART. The board is powered from a 12V DC input.

3.3.4 D8M Camera Kit

D8M 8-Mega Pixel Camera kit by Terasic offers the ability to connect a camera onto an FPGA development board and provide vision capabilities. It is suitable for applications that require image capturing. It can be connected to the development board, through a 2x20 Pin GPIO connector [10].

This camera kit consists of a MIPI Camera Module and a MIPI Decoder. The 8MP Autofocus with MIPI CSI-2 interface camera module (JAL-OV8865-A898B V2.0) is based on Omnivision’s OV8865 image sensor. It also integrates a Voice Coil Motor driver chip which allows the VCM to move its lens to the desired position for getting a sharp image and offer the Auto Focus feature.
Omnivision’s camera sensor, outputs its pixel data to a 10-bit RAW RGB format (Bayer Pattern). It supports image resolution up to 3264x2448 pixels at a maximum image transfer rate of 15fps. It provides options for multiple resolutions while maintaining full field of view. Users can program image resolution, frame rate, and image quality parameters.

Due to the lack of MIPI CSI-2 interface on development boards of Terasic, D8M camera kit integrates a MIPI Decoder to convert MIPI interface, to a parallel port interface. MIPI Decoder supports for up to 14-bit RAW data. Based on the MIPI Decoder, D8M also supports alternative camera modules with more output bits. For this reason, there are 4 pins that are reserved for alternative camera modules.

Figure 3.7 presents all three components of D8M camera kit, which can be configured through the I²C serial bus. Users can program the D8M’s components through two different I²C serial interfaces on the following addresses. With the first I²C bus users can access the camera module components (address is 0x6C for OV8865 camera sensor and VCM driver’s address is 0x18) and the second I²C bus is used for the MIPI Decoder access. The address of the I²C serial interface of MIPI Decoder is 0x1C.
3.4 Hardware and Software Development Tools

This section provides the basic information about the development tools for configuring the hardware for implementing the real-time pedestrian detection system.

3.4.1 Hardware Programming with Quartus and IP Cores

The Intel® Quartus® Prime software (formerly Quartus II) offers flexible design methodologies, advanced synthesis, and supports the latest Intel® FPGA architectures and hierarchical design flows. The compiler provides powerful and customizable design processing to achieve the best possible design implementation in silicon.

In this thesis the system there was a choice of designing the system’s hardware using Platform Designer Tool (formerly Qsys tool). Platform Designer is a system integration tool included as part of the Intel® Quartus® Prime software. Platform Designer simplifies the task of defining and integrating custom Intellectual Property Components (IP Cores) into FPGA designs. The Platform Designer IP Catalog offers configurable IP cores optimized for Intel devices [11].

In Chapter 4 and Chapter 5 it is presented the implementation of the hardware of the system, which is based mostly on IP cores that were instantiated in system design. Designer Platform offers a variety of hardware IP components that instantiate configurable functions and components, like Color Space Conversion, or a component to access the Hard Processor System. Designer Platform also simplifies the process of customizing and integrating IP components into systems, which design is based on a modular approach. It also provides visualization of the system that makes it easier for interconnecting the components, auto-adapting of different data widths and inter-operating between standard protocols.

![System design example in Design Platform.](image)

After generating the Qsys system design, the system has to constraint based on the design specifications. The following figure presents how the Intel® Quartus® Prime software compiles the design close timings based on our constraints and configure the design for the target board [12].
3.4.2 Software Development of Intel® SoC FPGAs

Developed in partnership with ARM, the ARM Development Studio 5 (DS-5) Intel® SoC FPGA Edition is an end-to-end suite of tools for embedded C/C++ software development on any Intel® SoC FPGA. The ARM® DS-5 Intel® SoC FPGA Edition tool is installed as part of the Intel® SoC FPGA Embedded Development Suite (SoC EDS) and it combines almost all features of the ARM DS-5 Professional Edition with powerful FPGA-adaptive debugging capabilities, providing unmatched visibility and control of your SoC FPGA [13].

It gives the opportunity to write software with an Eclipse-based IDE which provides a great workbench with ARM’s assembly editor and project management tools. It also provides compilation and building code options for high performance or small footprint with included ARM compiler 5 or 6. Last but not least by using ARM DS-5 SoC EDS we can build, debug and optimize Linux or bare-metal applications.

3.5 Hardware-Software Co-design

Based on the hardware-software co-design system design has to use the resources of both the HPS and FPGA for the pedestrian detection process. A simplified flow of the video stream on the pedestrian detection system is presented in figure 3.10.
3.5.1 Summing up Software Implementation

Section 2.3, provided an example of a software only implementation of pedestrian detection application with OpenCV library running on a general purpose computer. This implementation enabled us to decide to adapt OpenCV functions in this system. The idea was to integrate the algorithms of HOG plus SVM in a C++ application, which controls the system and runs on the HPS side of the chip (on the application processor). The goal was to adopt only the necessary functions from OpenCV which implements complicated processes of pedestrian detection flow, and make them run, in the same way like on general purpose computer.

Given the opportunity that the development board offers, this system integrates an Operating System (OS). With the ARM Cortex A9 hosting a Linux OS with a build in frame buffer, the system design focus on implementing the application of the system and especially the above-described functions, in an appropriate way to run in the OS. Input and output frames will be generated on the FPGA part of the chip along with other procedures.

![Diagram](image)

**Figure 3.11 Integration of HOG plus SVM algorithm on system application.**

3.5.2 Summing up Hardware Implementation

Hardware implementation of pedestrian detection system processes can accelerate the system’s processes. The hardware components of the system are implemented with the use of IP cores of Intel and Tersasic. The implementation of the hardware sub-system can be very tricky because the components require precise settings. Every mistake can lead to big-time loses in system implementation, because of the hardware compilation, which consumes a lot of time.

Starting from the image input, the first module of the streaming flow which is implemented on hardware is the camera input. Like it was mentioned in the previous section, D8M camera kit is connected directly on the FPGA part of the chip, and streams the pixel of the captured frames in a parallel way. The first in the video stream path component must read the input pixels, group them as frames and transform them from Bayern Pattern to RGB color space. Additionally, it must organize the video frames in frame packets, in order to meet the specification of Avalon Streaming Interface.

After the initial frame packeting process, the design integrates a frame buffering process, in order to ensure a correct video streaming flow. Next step was to duplicate the video stream. The first video stream is directed on a memory -after being converted to grayscale- in order to be accessible from the HPS, as the second one feeds the video mixing output component.
The video mixing output consists of the mix from two video streams, one coming directly from the camera, and the other coming from the frame buffer from Linux OS. At last, the system integrates the video output component for controlling the video DAC and the VGA connector of the board.

3.5.3 System Flexibility

Hardware-Software co-design not only provides us with multiple options in system design but also enhances system flexibility compared to hardware only implementations. Hardware only implementation requires re-compilation of the system in order to change even the slightest parameters, in contrast with this thesis implementation which allows to adjust most of the parameters of the system through the software application.

Section 2.4 describes the functions of OpenCV library, is presenting the opportunity to change parameters necessary for the pedestrian detection algorithms. These parameters are,

- the detecting window size
- the cell size
- the window scaling

These parameters can easily adjust the detecting accuracy which is directly connected to computational time.

Not only software functions can provide such flexibility, but also many hardware components that are implemented with IP cores, provide the opportunity to change their parameters through the software application.

3.6 System Architecture -- Design Progress

Keeping in mind the pedestrian detection system processing flow that was presented in figure 3.10, system design and implementation must be build step by step. The initial design was a minimum system that would ensure the input and output of the video stream. This design changed over time based on the tests that took part in the components that made up the system. Several ideas were tested -some with success, some others with failures- each and everyone, leading to design decisions in order to achieve the final form of the system.

This section also provides some figures with the block diagram of the system design progress. Data exchange through the components is shown with arrows which are described on the following figure.
3.6.1 Validating Video Input Stream from Camera
Described earlier in this section, video input and the correct video streaming flow are the base of the system which guides us to decide, that implementing the video streaming flow was a good starting point for designing the system. For this purpose, system design follows a reference design that Terasic offers, with the use of De1-SoC Development kit and D8M Camera Kit. The reference design block diagram is presented in the following figure which was fetched from D8M’s reference manual [10].

![Diagram](image1)

Figure 3.13 D8M Camera Kit reference design from Terasic.
This system is implemented only in the FPGA part of the SoC FPGA chip. The design presents the procedure to implement a video stream from D8M camera kit, on DE1-SoC board. Intel VIP-Video Image Processing suite is used to display images on the VGA monitor and the Nios II processor is used to configure the I2C devices. There is a Camera IP component from Terasic in Designer Platform, which translates the Bayer pattern from the camera to an RGB Avalon MM steam format and feeds it to the Altera VIP. At last the IP core developed by Terasic for auto-focus is used to find the optimized focus settings of the user-defined image area.

Frame Buffer IP Core is there to ensure a correct video flow without flickering. Frame Buffer supports double and triple buffering and stores frames in an external SDRAM in order to achieve simple frame rate conversion. Frame buffering helps solve throughput issues in the video stream data path and support frame dropping and frame repeating.

Based on the above reference design the first design attempt was to start seeking the options for increasing the resolution of the input video stream.

Figure 3.14 shows a simplified version of the above-described reference design. In this system, in order to achieve higher resolution changes had to be made in every single component of the system.

Starting from the camera sensor’s datasheet there was an attempt to locate the registers that are tied with the control of the camera sensor’s resolution. This would allow changing the camera sensor registers with the desired values, in order to make the camera sensor to feed the FPGA-based system with a higher image resolution. These values can be changed in a C based program which runs in the Nios II processor.
Next step was to change the parameters of IP Cores inside the Designer Platform, first by changing the resolution for all video components in the system. These components were Terasic Camera IP, Terasic Auto Focus IP, Frame Buffer and Clocked Video Output. At last, frequency calculation and change of the PLL settings have to perform in order to match the Pixel Clock rate. Unfortunately, the minimum information provided from the camera sensor’s datasheet allowed us to reach up to 800x600 resolution at 60 fps with some flickering occurring during the video stream. That led us to decide to keep the resolution of the video stream to the stable 640x480 pixels resolution @ 60fps.

3.6.2 Importing HPS as core processor of the system

Based on the previous design, the efforts focused on changing the processor that runs the system, from Nios II to Hard Processor System. Nios II wasn’t a good choice because of the limited capabilities and FPGA cost. Nios II is a soft-processor which is implemented entirely in the programmable logic and memory blocks of FPGA. On the other hand, HPS is a computer system with MPU, memories and several hardwired peripherals. The HPS component was added in the Designer Platform system and configured with the necessary parameters.

In the HPS IP Core, there are multiple parameters to configure, giving the opportunity to enable specific peripherals and decide the external clocks and memories of the system. Furthermore, the HPS IP core offers the interconnection between FPGA and HPS part.

Discussed earlier in this chapter, this system will run a Linux OS with a build in frame buffer, on the ARM Cortex-A9 dual-core processor which is part of the Cyclone V SoC FPGA device of DE1-SoC board.

![Figure 3.15 Integration of Hard Processor System in System Design.](image)
After building the hardware sub-system the development of the software application took place. This application will run on the ARM processor under Linux and will make use of the hardware resources of the DE1-SoC Computer.

Based on the initial reference design we had to build an identical program that will run and function on the HPS side under Linux OS the same way like it was on Nios II. The application controls the system and makes use of the I2C controllers which connect the camera sensor and MIPI controller with the SoC FPGA chip. The app will provide a window to initialize and configure the video capture easier.

3.6.3 Video Output of Embedded Linux OS

Bearing in mind the built-in frame buffer of the Embedded Linux OS system enhanced with the ability to provide information which generated in the Linux OS through the display of the system. The image output of the Linux OS will be display through the operating system’s frame buffer in the output monitor.

Till now, the system was able to display information on the screen through the Clocked Video Output only one stream, the camera feed. In this case, we care on displaying both the camera feed and the Linux’s OS image output. For this reason, the system makes use of the Alpha Blending Mixer IP core of Intel. This core can output a picture-in-picture video stream, based on two up to twelve layers. Each layer is a video stream, driven from a frame buffer or a frame reader so that data are feed with the correct timing. Each layer must fit within the dimensions of the background layer.
The Video Mixing IP core is necessary to get driven from a frame buffer or a frame reader. In this case, a frame reader performs for this job, as it connects the Linux OS frame buffer through the high-speed FPGA to HPS AXI Slave bus to the FPGA part of the SoC chip. The output frames from the Linux OS are at 1024x768 pixel resolution in a four-color plane sequence, RGB and an Alpha color value. That means that the alpha value has to get discarded from the video stream, and that achieved by importing a Color Plane Sequencer after the Frame Reader. The block diagram presents the feed of the CPS with a 1024x768 RGBA video stream, which outputs a 1024x768 RGB video stream.

Comparing the two video streams, the camera stream, and the Linux OS output stream, there is a difference in the resolution. Based on the highest by its resolution layer, the video mixing core will stream out a picture-in-picture video stream. The background will consist of the Linux OS output stream and the foreground of the smaller camera stream.

### 3.6.4 Design the backbone of the System

The goal of this thesis was to build a system able to perform pedestrian detection on the frames obtained from the D8M camera kit. These frames are captured and procedure on the FPGA part of the chip. The main processes of pedestrian detection algorithms perform on the application which runs on Linux OS on the HPS side of the SoC FPGA chip.

*Figure 3.16 Importing Video Output of Linux OS.*
As we can figure out, the captured frames must get carried in the HPS part in order to get processed from the pedestrian detection algorithms. Every single frame captured from the camera must be available for reading from the HPS. Therefore, there is a need for an intermediate memory that will buffer the captured frames. The connection between the HPS and the FPGA will establish with the AXI Bridge.

The first step for implementing the idea of frame buffering from FPGA to HPS was to verify the functionality of AXI Bus. By importing an On-Chip ROM Memory in the FPGA and connect it with the HPS to FPGA AXI Master Port of the HPS component an application running on Linux OS, is able to read data from on-chip memory. For this purpose, an application able to read data from the FPGA's On-Chip ROM was implemented. The On-Chip ROM Memory was initialized with known data during the compilation of the FPGA design, in order to perform a cross-check of the data, during the reading process. The following figure presents the block diagram of the system with On-Chip ROM integration.

![Figure 3.17 Integration of an On-Chip ROM Memory.](image)

The reading process and the evaluation of the AXI Bus from the program that runs on Linux OS are illustrated by the following pseudocode. For this process the on-chip ROM Memory was initialized with known data. More accurately on the addresses [0-100], the values were integers from “0” up to “100”.

```c
onchipBaseAddress = 0x004000;
for (i=0; i<100; i++) {
  k = read_data(onchipBaseAddress+i);
  if (k!=i) {
    break;
  }
}
```
Next step was to replace the On-Chip ROM with an On-Chip RAM Memory. The On-Chip RAM gives the opportunity to write the captured frames from the stream. Then the frames could be accessed from the application that runs on Linux OS. Writing the captured frames on the On-Chip RAM was not an easy procedure. Based on the design, the captured frames had to appear in the output screen at the same time. In order to achieve both the procedures the video stream had to get duplicated, which was implemented with the use of a Color Plane Sequencer IP core. The constraint of memory size was also very important. In order to achieve the writing of a whole frame in the memory, the image had to scale in a lower resolution. The last step was to transform the streaming packet into plain data and write them into memory.

![Figure 3.18 Implementation of Frame Writing into On-Chip RAM.](image)

The above figure also presents the integration of Scaller IP core and a Video DMA Controller (from stream to memory) IP Core, for those previously-discussed processes of image scaling and frame writing into memory. Ensuring access to the captured frames from both the HPS and the FPGA part of chip one can say that figure 3.18 presents the backbone of the system.
3.6.5 Enhancements of FPGA

After studying in a more detailed way the pedestrian detection algorithms from OpenCV library, we figured out that feeding the functions of detection with a gray-scaled image instead of a colored one, the process of detection was sped up without losing accuracy and the same time the system managed to reduce the size of the on-chip ram. For this purpose, it was decided to convert the captured frames from RGB color space in grayscale before the writing into the On-Chip RAM. The following figure presents the block diagram of the revised system.

![Block Diagram of Revised System]

**Figure 3.19 Preprocessing of Captured Frames.**

At last, there was an idea of implementing a Stream Blender. The idea behind implementing the Stream Blender IP core was to design a custom IP core capable of blending two streams. The Stream Blender was designed to replace the Video Mixing core. Based on the two same input streams of Video Mixing core, but writing only the captured rectangles on Linux’s OS video output, the idea was to project the green rectangles on the captured frames. Unfortunately, this idea wasn’t achieved and the system which presented in this
thesis is the one of described in the block diagram of Figure 3.19. The following figure presents the block diagram of the system design for the proposed Stream Blender core.

![Block Diagram of System Design](image)

**Figure 3.20** Stream Blender Design Proposal.
Chapter 4  Cyclone V SoC FGPA - Computer System Interconnection and Memory Map

Building a system on SOC FPGA chip which brings a Hard Processor System and an FPGA fabric, bears the need for clarification of the bridging between these two distinct portions of the chip. The designer has to clear up the communication between the individual parts which consisting the system. Every single component in the chip, communicate through memory-mapped interfaces. One can say that the HPS, the FPGA components and the interconnections between them consisting the SoC Computer System's hardware.

4.1  Hard Processor System Interconnection and Address Map

In this system's design the Hard Processor System acts as the main processor which runs the system. From this point of view, it is very important to know how the components communicate, based on their memory address. The following figure presents the block diagram of the HPS interconnections. Looking up closely the components that make up the HPS and their connections, it is clear the master-slave relationship between them.
The three main parts of the HPS are the MPU, the L3 interconnect and the FPGA to SDRAM controller which acting as masters. The HPS address map specifies the addresses of slaves, such as memory and peripherals, as seen by the MPU and the other masters. The HPS has three address spaces which assigned in the three master parts of the HPS. Address spaces are divided into one or more nonoverlapping regions. For example, the MPU address space has the peripheral, FPGA slaves, SDRAM window, and boot regions. The following figure presents the relationships between the HPS address spaces.\textsuperscript{[14]}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4_1.png}
\caption{HPS Interconnect Block Diagram.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4_2.png}
\caption{HPS Address Space Relationships.}
\end{figure}
L3 and MPU address spaces are 4GB and the FPGA to SDRAM address space is up to 4GB. L3 and MPU have access to different regions of the HPS because of the system interconnection, but there are also some common regions in their address maps. The following table presents the base address and size of each region that is common to the L3 and MPU address spaces.

<table>
<thead>
<tr>
<th>Region Name</th>
<th>Base Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA Slaves</td>
<td>0xC0000000</td>
<td>906 MB</td>
</tr>
<tr>
<td>Peripherals</td>
<td>0xFC000000</td>
<td>64 MB</td>
</tr>
<tr>
<td>Lightweight FPGA Slaves</td>
<td>0xFF200000</td>
<td>2 MB</td>
</tr>
</tbody>
</table>

*Table 4.1 Common Address Space Regions.*

These three regions are the one we are dealing most, in this system design from the HPS side. The key factor in this design is that the MPU, which runs the system, has direct access to these three regions. Implementing any FPGA Component that requires control from or carries a memory data space, the application which runs on the MPU has access to them. This is done by connecting their slave port in the HPS2FPGA AXI Bridge or in the Lightweight HPS2FPGA AXI Bridge. The selection, of which port should handle the communication, is usually being decided based on the amount of data the application want to pass through the AXI bus.

4.2 FPGA Sub-System Interconnection and Memory Mapping

Not only the HPS can handle the FPGA’s components through their slaves. FPGA components have access to other FPGA component’s registers and memory spaces through a master-slave connection. This procedure creates new memory spaces. In a more abstract way of presenting the memory spaces of the system, we can say that every Master creates a unique memory space which contains at least one Slave Address Region. Of course multiple Masters can have access to the same Slave’s address region.

To better understand the assignment of base addresses and size in an FPGA IP Component this section presents the implementation of the first design (Figure 3.14) described in the previous chapter which controls the streaming flow from the camera kit. The system implementation is performed with Platform Designer tool and does not include the HPS.

When starting a new system design in Platform Designer, clock source is imported by default (Figure 4.3).
Before importing any other component, a quick check on the Address Map tab reveals that there are no address spaces created (Figure 4.4).

![Figure 4.4 Address Map tab, in Platform Designer Tool.](image)

In order to start the system design, the first step was made by importing Nios II Processor. Nios II Processor has two different Master ports, the Data Master and the Instruction Master port (Figure 4.5). A quick look on the Address Map tab now exposes a table, in which there are two columns with the Nios Master’s ports names (Figure 4.6).

![Figure 4.5 Importing Nios II Processor in system design.](image)

![Figure 4.6 Address Map after importing Nios II Processor.](image)

By default, Nios II connects the *debug_mem_slave* Slave port into data and instruction master ports and assigns a memory region. Every Master port which is inserted on the system from a component will occupy a new column on the table. The rows of this table are filling with every component’s Slave port. The difference was revealed by importing the on-chip memory component (Figure 4.7, Figure 4.8).
The previous figure presents that there are no memory regions assigned in the two masters space addresses for the on-chip memory slave, and that’s because there are no connections between them on Qsys system design. The assignment of a slave memory regions is done by connecting a master port to a slave port. If more than one master connects to a slave, then the masters share the same memory region for accessing the slave. Addresses are assigned automatically by the Platform Designer tool. (Figure 4.9)
The previous figure presents that this operation may lead to conflict in memory regions of a master’s address space because the tool does not check in first place the assigned memory regions. By manually changing the base address of a component with a correct base address will override the problem. This is done by double-clicking the base address (Figure 4.10). A correction on the assignment of the memory region on a component can also get configured automatically with the option in the menu System → Assign Base Addresses (Figure 4.11). This option calculates again the memory regions and makes the new assignments which may lead to different base and end addresses in the previously imported components. In order to forbid the tool to change the assigned addresses, an option for locking the base address is available by clicking the lock icon on the left side of a base address.

**Figure 4.10** Manually change the Base Address of a Slave.

**Figure 4.11** Assign Base Addresses option in Platform Designer tool.

The difference in the base addresses of the components is presented in figure 4.12. By checking again, the Address Map tab there are differences in the table (Figure 4.13). The End Address in each component was assigned based on component’s size. In the example the on-chip memory component, its memory size is 100kbyte – 1 byte per word. So the memory holds a memory region from 0x0 to 0x1869f (100000 → 0x186A0).

**Figure 4.12** New base address assignment and address locking.
Figure 4.13 Address Map after connecting data and instruction master to on-chip slave.

But what happens by importing a second or more components that have only Master ports? With the use of a “Master Component” the system implements a sub-system which runs in a parallel way with the previous designed sub-system, which master is Nios II Processor. Further on, on this implementation next step is to import the Frame Buffer component which contains two Master ports, the frame writer and frame reader ports (Figure 4.14). This component captures the frames from the video stream and stores them in the external SDRAM memory with the use of an SDRAM controller.

Figure 4.14 Importing Frame Buffer in system design.

Checking again the address space tab, two new columns have added in Address map. These are the Frame Buffer’s Master ports (Figure 4.15).

Figure 4.15 Address Map after importing Frame Buffer.

The same procedure previously-presented is followed for all the necessary components which make up the system. The whole system is presented in figure 4.18. This design offers a multiple master-slave relationship resulting from the component connections. The final address map table is presented in the following figure.
Introduced earlier in this section, this design was implemented entirely in the FPGA fabric. Figure 4.17 presents a projection of the memory regions and memory addresses which assigned on the system design introduced in Chapter 3.

*Figure 4.16 Address map of the system.*

*Figure 4.17 Design example with Master-Slave relationship.*
Figure 4.18 System design example implemented in Platform Designer.
This system executes an application program on Nios II processor which configures the MIPI’s decoder’s and camera’s module registers in order to initialize the D8M camera kit and start the video stream. Their registers are accessed through I2C Interface (i2c_opencores_camera and i2c_opencores_mipi components). The system also makes use of two I/O pins for the MIPI control (PIO components). These hardware components are configured from the application program as they are seen as memory regions because Nios II processor has access to them from its data master port.

In a few words, the application processor (in this case the Nios II processor) can only have access to the components that are connected to its data master, or instruction master ports.

In this system design, Nios II can’t have access to the SDRAM because his master port is not connected on the slave port of the SDRAM controller. The reason the SDRAM controller’s address region might have a common address with other components is that only Frame Buffer is connected on the SDRAM controller. Discussed earlier in this section, Nios II and Frame Buffer act as masters in two distinct sub-systems.

4.3 HPS – FPGA – Peripherals Interconnection

In this thesis, the system relies the pedestrian detection processing and control of the system, on the Hard Processor System. The HPS and its peripherals are hardened in silicon but they require configuration in order to establish the connection to and from the FPGA Fabric. This configuration is performed in the FPGA sub-system design in Platform Designer tool.

Based on the previous system design the system makes use of the HPS as the main processor of the system. In order to demonstrate its configuration and the access of the HPS to the FPGA components, it was decided to replace the Nios II processor with the HPS component on Platform Designer. The HPS component, actually implements the configuration of AXI bridges between the HPS and the FPGA fabric, among other configurations that will present in the next chapter.

The enhanced design stars by importing the Cyclone V HPS component in Qsys system design. In the configuration window of the HPS component, tuning of the HPS is mandatory in order to meet the system’s specifications. The configuration of the HPS consists of the HPS peripherals set up, the choice of the parameters of the DRAM and the connection between the HPS and the FPGA. These options are grouped in four different sections (Figure 4.19) of the configuration window of the HPS component.

![Figure 4.19 HPS Configuration Window Instance.](image)

This chapter discusses only the options which provided for the configuration of the interconnection of the SoC FPGA. The following figure presents the configuration tab of the HPS component for the FPGA Interfaces.
In this figure and in conjunction with figure 4.1, it is clear the existence of the four AXI Bridge interfaces (FPGA to HPS, HPS to FPGA, Lightweight HPS to FPGA and FPGA to HPS SDRAM Interfaces). The HPS to FPGA interfaces acts as Master Ports and the other two acts as Slave ports. The HPS to FPGA interfaces provides to the HPS access on the FPGA's components. FPGA to HPS provides to the FPGA's implemented components access to the whole address space of L3 Main Switch of the HPS and the FPGA to HPS SDRAM interface provides access to the SDRAM address region. These two address regions were presented in figure 4.2.

Following the previously-presented system design, Nios II component and the SDRAM controller core were removed and the HPS component came as their replacement. The following figure presents the address map table where the two new Master and Slave ports have been added. Now it’s time to make the connection and assign the base addresses.

![Revised address map table](image-url)
Starting from the Frame Buffer, a connection between its Master reader and writer in the FPGA to HPS SDRAM slave was performed. This provides access to the HPS’s SDRAM from the Frame Buffer component. The memory region accessed from this port is 4GB wide as it was discussed earlier, which consists of the whole SDRAM address space. This means that every Master which connects on this port can’t have access to any other component, because it occupies his whole address bus (from 0x0000_0000 to 0xffff_ffff). The same way of addressing provides the FPGA to HPS AXI Bridge. Connecting a master port in the slave port of the HPS, a component can have access to the space region of the L3 Main Switch.

![Figure 4.22 Address Map after connecting Frame Buffer on FPGA to HPS SDRAM interfaces.](image)

In order to configure the system design to act like the previous-presented system design - in this condition with the HPS as the main processor- the newly imported application processor (HPS MPU) must gain access to the FPGA components. The HPS to FPGA AXI Bridge and the Lightweight HPS to FPGA AXI Bridge acts in a similar way, as they both provide access to the FPGA fabric. The main difference is that the HPS to FPGA can drive up to 960MB address space in total and is configured with 32, 64 or 128-bit wide data bus, in contrast with the Lightweight Bridge which can only drive up to 2MB address space with a 32-bit wide data bus. The lightweight bridge is mostly used for accessing components that provide only their registers for configuration, while the HPS to FPGA AXI Bridge is mainly used for passing a big amount of data to and from the FPGA. For demonstrating purposes of the HPS to FPGA AXI Bridge, the system uses the 100kB on-chip RAM of the current design. The following figures (Figure 4.23, Figure 4.24) present the new address map of the system and the system content and connections in Platform Designer.

![Figure 4.23 Address Map of the FPGA sub-system with the HPS as main processor.](image)
Figure 4.24  System Design with the HPS as main Processor.
4.3.1 HPS Peripheral Region Address Map

In the HPS address space, there are memory regions which are dedicated for accessing and controlling the peripherals of the HPS. Each peripheral slave interface has a dedicated address range in the peripheral region. The table below lists the base address, end address and memory size for each slave.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Name</th>
<th>Start Address</th>
<th>End Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>stm</td>
<td>Space Trace Macrocell Module</td>
<td>0xFC000000</td>
<td>0xFEFFFFFF</td>
<td>48 MB</td>
</tr>
<tr>
<td>dap</td>
<td>Debug Access Port Module</td>
<td>0xFF000000</td>
<td>0xFF1FFF00</td>
<td>2 MB</td>
</tr>
<tr>
<td>peripherals</td>
<td>Peripherals and L3 GPV</td>
<td>0xFF400000</td>
<td>0xFFF0201F</td>
<td>11 MB</td>
</tr>
<tr>
<td>rom</td>
<td>Boot ROM Module</td>
<td>0xFFF00000</td>
<td>0xFFFF0000</td>
<td>64 MB</td>
</tr>
<tr>
<td>mpu</td>
<td>MPU Module Address Space</td>
<td>0xFFEC0000</td>
<td>0xFFFF0000</td>
<td>8 KB</td>
</tr>
<tr>
<td>mpuL2</td>
<td>MPU L2 cache controller Module Address Space</td>
<td>0xFFF00000</td>
<td>0xFFF01000</td>
<td>4 KB</td>
</tr>
<tr>
<td>ocram</td>
<td>On-chip RAM Module</td>
<td>0xFFF00000</td>
<td>0xFFFF0000</td>
<td>64 MB</td>
</tr>
</tbody>
</table>

Table 4.2 HPS Peripherals Table.

In the peripheral region in the 3rd row of table 4.2 there is the mapping of all other peripherals of the HPS which are not present in this table, such as the AXI Bridge global programmer’s view (GPV) registers, the Ethernet, SD controller, and others. In the following table, these peripherals are presented with their base address and their size [15].

<table>
<thead>
<tr>
<th>Interface</th>
<th>Base Address</th>
<th>Size</th>
<th>Interface</th>
<th>Base Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW HPS2FPGA GPV</td>
<td>0xFF400000</td>
<td>1 MB</td>
<td>UART 1</td>
<td>0xFFC03000</td>
<td>4 KB</td>
</tr>
<tr>
<td>HPS-to-FPGA GPV</td>
<td>0xFF500000</td>
<td>1 MB</td>
<td>I2C controller 0</td>
<td>0xFFF04000</td>
<td>4 KB</td>
</tr>
<tr>
<td>FPGA-to-HPS GPV</td>
<td>0xFF600000</td>
<td>1 MB</td>
<td>I2C controller 1</td>
<td>0xFFF05000</td>
<td>4 KB</td>
</tr>
<tr>
<td>Ethernet MAC 0</td>
<td>0xFFF00000</td>
<td>8 KB</td>
<td>I2C controller 2</td>
<td>0xFFF06000</td>
<td>4 KB</td>
</tr>
<tr>
<td>Ethernet MAC 1</td>
<td>0xFFF02000</td>
<td>8 KB</td>
<td>I2C controller 3</td>
<td>0xFFF07000</td>
<td>4 KB</td>
</tr>
<tr>
<td>SD/MMC</td>
<td>0xFFF04000</td>
<td>4 KB</td>
<td>SP Timer 0</td>
<td>0xFFF08000</td>
<td>4 KB</td>
</tr>
<tr>
<td>QSPI flash controller</td>
<td>0xFFF05000</td>
<td>4 KB</td>
<td>SP Timer 1</td>
<td>0xFFF09000</td>
<td>4 KB</td>
</tr>
<tr>
<td>FPGA manager regs</td>
<td>0xFFF06000</td>
<td>4 KB</td>
<td>SDRAM controller</td>
<td>0xFFF02000</td>
<td>128 KB</td>
</tr>
<tr>
<td>ACP ID mapper regs</td>
<td>0xFFF07000</td>
<td>4 KB</td>
<td>OSC1 Timer 0</td>
<td>0xFFF00000</td>
<td>4 KB</td>
</tr>
<tr>
<td>GPIO 0</td>
<td>0xFFF08000</td>
<td>4 KB</td>
<td>OSC1 Timer 1</td>
<td>0xFFF01000</td>
<td>4 KB</td>
</tr>
<tr>
<td>GPIO 1</td>
<td>0xFFF09000</td>
<td>4 KB</td>
<td>Watchdog Timer 0</td>
<td>0xFFF02000</td>
<td>4 KB</td>
</tr>
<tr>
<td>GPIO 2</td>
<td>0xFFF0A000</td>
<td>4 KB</td>
<td>Watchdog Timer 1</td>
<td>0xFFF03000</td>
<td>4 KB</td>
</tr>
<tr>
<td>L3 interconnect GPV</td>
<td>0xFFF00000</td>
<td>1 MB</td>
<td>Clock manager</td>
<td>0xFFF04000</td>
<td>4 KB</td>
</tr>
<tr>
<td>NAND flash controller</td>
<td>0xFFF09000</td>
<td>64 KB</td>
<td>Reset manager</td>
<td>0xFFF05000</td>
<td>4 KB</td>
</tr>
<tr>
<td>QSPI flash data</td>
<td>0xFFF0A000</td>
<td>1 MB</td>
<td>System manager</td>
<td>0xFFF08000</td>
<td>16 KB</td>
</tr>
<tr>
<td>USB OTG 0 control</td>
<td>0xFFF0B0000</td>
<td>256 KB</td>
<td>DMA nonsecure regs</td>
<td>0xFFF00000</td>
<td>4 KB</td>
</tr>
<tr>
<td>USB OTG 1 control</td>
<td>0xFFF0C0000</td>
<td>256 KB</td>
<td>DMA secure regs</td>
<td>0xFFF01000</td>
<td>4 KB</td>
</tr>
<tr>
<td>NAND flash controller</td>
<td>0xFFF0B0000</td>
<td>64 KB</td>
<td>SPI slave 0</td>
<td>0xFFF02000</td>
<td>4 KB</td>
</tr>
<tr>
<td>FPGA manager config</td>
<td>0xFFFBB0000</td>
<td>4 KB</td>
<td>SPI slave 1</td>
<td>0xFFF03000</td>
<td>4 KB</td>
</tr>
<tr>
<td>CAN 0 controller regs</td>
<td>0xFFF0C0000</td>
<td>4 KB</td>
<td>SPI master 0</td>
<td>0xFFF00000</td>
<td>4 KB</td>
</tr>
<tr>
<td>CAN 1 controller regs</td>
<td>0xFFF0C0100</td>
<td>4 KB</td>
<td>SPI master 1</td>
<td>0xFFF01000</td>
<td>4 KB</td>
</tr>
<tr>
<td>UART 0</td>
<td>0xFFF0C2000</td>
<td>4 KB</td>
<td>Scan manager regs</td>
<td>0xFFF02000</td>
<td>4 KB</td>
</tr>
</tbody>
</table>

Table 4.3 HPS 2nd Peripheral Table.
4.4 Computer System Memory Map

The topics described in the previous sections consist the memory spaces of the system. The address spaces for the MPU and the Frame Buffer are the ones this demo system design is dealing with. In order to get a better understanding of the memory map of this system, this section provides a visualized representation of the system interconnections and their memory space assignments. This will be guided through the implemented design example that was presented in section 4.3. The application processor of the system is the A9 MPU dual-core microprocessor which is part of the HPS and that's where the application program for controlling the system is executed.

The following figure presents the connections between the parts of the system.

The FPGA components presented in figure 4.25 have been assigned to a memory region. Their implementation and memory space assignments have been configured in Platform Designer and presented in Section 4.3. This region was mapped on a specific address space of the MPU address map, based on their connection. In this example the I2C Camera component which occupies the address between 0x0000_0040 – 0x0000_005f and is connected on the MPU through the Lightweight HPS to FPGA Bridge was mapped on the address 0xff20_0040 – 0xff20_005f, because, the Lightweight HPS to FPGA Bridge address

---

**Figure 4.25 System Interconnection and Master – Slave relationship.**
space starts from \textit{0xff20_0000} address. The following figure presents the final system’s address map.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{system_address_map.png}
\caption{System’s Address Map.}
\end{figure}
Chapter 5  System’s Hardware Implementation

This chapter follows the structure of the system’s design progress introduced in section 3.6 and presents the procedures for the implementation of the system’s hardware. The tests of the hardware components that took part during the design process, led to decisions on which way to guide this system design in order to meet the specifications that were described earlier.

The hardware FPGA sub-system was implemented mostly from IP cores within Platform Designer tool. This kind of implementation with IP cores offers us faster implementation and easier integration but adds constraints in system design. Summing up the design progress, system design was decided to focus on implementing an FPGA sub-system that is capable of providing a correct video flow from a camera along with preprocessing of the video frames.

This chapter is organized as follows. It presents the implementation of D8M’s reference design and its details. It follows up, with the modifications that took part in this design which led to the building of the backbone of the system using the IP cores provided by Intel and Terasic.

5.1  D8M Camera Kit Reference Design

In many cases of system design, the designer is recommended to follow the reference designs of a component in order to get a better understanding of their operation. In this case reference design of the Camera Kit implemented on De1-SoC was a good starting point for design and implementation of the pedestrian detection system. This system design was implemented mostly within Qsys (now Platform Designer). The needed hardware parts for the specific system design are the D8M Camera Kit, the De1-SoC Development Board and a monitor capable of connecting with the VGA video output of De1-SoC dev board. The design’s block diagram was presented in figure 3.13 in sub-section 3.6.1 \(^{[10]}\).
Design and Implementation of a Real-Time Pedestrian Detection System on SoC FPGA

An important note for D8M’s reference system design is that the reference manual notes, that the reference design was implemented in Quartus II v15.0 and suggests compiling it with the same Quartus version. Trying to compile the system with any other Quartus version might lead to malfunctions of the components of the current system design. In case there is a need to switch to another Quartus version to design a system based on a reference design, an operation test on each component is mandatory. For the implementation of the system, Quartus v.15.1 was used and from the tests that took place, no issues have arisen with the IP cores from Terasic. From time to time Terasic might offer revised versions of the current design implemented in newer Quartus versions. A quick look on the Terasic’s website is highly suggested.

This section provides a step by step representation of the components of D8M’s reference design. D8M’s reference design is mostly implemented with the use of Platform Designer tool. In order to present the system's details, this section will go through the details of Qsys system's components with a detailed overview of the IP cores that consist the Qsys system design. Through the steps, the components of the system are presented along with a detailed block diagram which simulates the build of the system. The section sums up with a reference of hardware's Top-Level design and constraints.

5.1.1 Clock Source and PLL Settings

In system contents tab of Qsys system design it is the presentation of the components which consist the system, their connections, and their base addresses. Every single component requires a clock source and a reset source to operate. The Clock Source component provides the system with the clock source from an output crystal oscillator. De1-SoC brings four 50 MHz clock sources from the clock generator which is connected as clock source of the system. The 50 MHz is the value to provide in the parameter window of Clock Source IP core and it’s presented in the following figure.

![Clock Source parameter window](image)

**Figure 5.1 Clock Source parameter window.**

In the system contents tab, the clock input and the reset input ports are exported in order to connect with the 50 MHz clock from clock generator, and with a reset source respectively. The following Figure 5.2 presents the exported connections.

![Clock Source's pin export](image)

**Figure 5.2 Clock Source’s pin export.**

The reset output of the Clock Source is connected in every single component’s reset input of the system. The clock output feeds the clock source of the necessary components. The clock output operates at 50 MHz and in this design, it is connected to components that are
capable to perform with a 50 MHz input. The following Figure 5.3 presents the block diagram of the Clock Source IP core implemented in Qsys system design.

![Block diagram of Qsys System Design with Clock Source IP Core](image)

**Figure 5.3** Block diagram of Qsys System Design with Clock Source IP Core.

With the IP cores that require different clock frequencies to operate, a PLL is used to generate clock sources with different frequencies. In Qsys system design, the 50MHz clock output is connected to a PLL input. In Qsys system design, the PLL is implemented with an IP core called *Altera PLL*. This PLL can generate up to 9 clocks. The following figure presents the parameter window of the *Altera PLL IP core*, which in this design generates 5 clocks (the first it is not used), with the first three clocks producing a clock pulse at 100MHz and the following two at 25MHz and 20MHz respectively.

![Altera PLL parameter window](image)

**Figure 5.4** Altera PLL parameter window.
These clocks are generated based on the needs of the system. The outclk1 feeds the external SDRAM chip, the outclk3 feeds the VGA video output port and acts as the pixel output clock, the outclk4 feeds the camera kit (D8M requires a clock source at 20MHz) and the outclk2 feeds the component of the system which handles the video stream. The following figure presents the outclk1, outclk3, and outclk4 which are exported in order connect on the previously-specified components.

![PLL exported clock outputs.](image)

The following figure presents the block diagram of Qsys system design with the addition of PLL.

![Block diagram of Qsys System Design with integration of Altera PLL IP Core.](image)

5.1.2 Nios II Processor System

The reference design presenting in this section demonstrates the capabilities of D8M camera kit. This demo software application written in a C-based program runs on the system’s application processor. This system’s application processor is Nios II. Nios II is a general-purpose RISC core processor. A system that brings Nios II as the core processor, a combination of peripherals and a memory is equivalent to a microcontroller. The Nios II processor system consists of a Nios II processor core, a set of on-chip peripherals, on-chip memory, and multiple interfaces.\(^{[16]}\)

Nios II is a configurable soft IP core, as opposed to a fixed, off-the-shelf microcontroller. We can add or remove features on a system-by-system basis to meet performance or price goals. Soft means the processor core is not fixed in silicon and can be targeted to any Altera FPGA family because it is implemented entirely from the FPGA’s logic elements.

This section provides the steps for building a system in Qsys around the Nios II processor. For the Nios II IP core we can choose from two different instances of the Nios II processor, from the one which brings more features and is designed to perform faster than the other which design based on using the minimum FPGA resources. The following figure presents the selection between the two instances of Nios II processor.
The other tabs of the parameter window configure the features of Nios II processor. In Vectors tab, users have to select the reset and exception vector memory. This feature is configured after the connection of a memory on the data and instruction master of Nios II.

Next step was to import a memory. In this system, on-chip memory is used as both program and data memory. This is why both data and instruction masters of Nios II are connected on the on-chip RAM. The configuration of the on-chip RAM is presented in the following figure.

By importing the memory module and connect its slave port with the data and instruction master of Nios II, it is necessary to go again in the parameter window of Nios II in Vectors...
tab and select the memory as the Reset and Exception vector. Figure 5.9 presents the block diagram of the Qsys system design with the integration of Nios II and on-chip memory.

**Figure 5.9** Block diagram of Qsys System Design with integration of Nios II and on-chip memory.

One key element in embedded systems like the presented one is time. Because time is critical, all microcontrollers have one or more built-in hardware timers, designed to run independently from the processor. In this design, a timer is also being used, which is implemented with an Interval Timer IP core. Interval Timer is configured to perform as a 32-bit counter which can also provide interrupts on the Nios II processor every single millisecond. The following figure presents the parameter window of the Interval Timer.

**Figure 5.10** Interval Timer’s parameter window.

This system also implements a method to communicate serial character streams between a host PC and the Nios II, giving the ability on embedded software programmers for printing messages and debugging their software programs. This is done with the use of JTAG UART IP core that provides an Avalon interface that hides the complexities of a JTAG interface. JTAG UART IP core is added to the system design. Figure 5.11 shows the parameterization of the component. Nios II processor users can access the JTAG UART via the Nios II IDE or the nios2-terminal command line.
JTAG UART

![Image of JTAG UART parameter window]

**Figure 5.11** JTAG UART parameter window.

Last but not least, General Purpose Input and Output interface of a system is mandatory. GPIOs provide ease of access to a system’s internal properties. The number of available I/Os and their functionalities are also very important. In many applications, the GPIOs are configured as interrupt lines for a CPU, to signal immediate processing of input lines. In the current design GPIOs of the system are implemented with PIO IP cores.

PIO core is a component that provides a memory-map interface between an Avalon MM slave port and general purpose I/O ports. The PIO core can be configured as input, output, both input, and output or bidirectional port and provides up to 32 I/O ports per core. The PIO core is used to access I/O to user logic or it can be routed to access the pins of the FPGA fabric. If a port is configured as input, PIO core gives the option for edge capturing of the state of the port or it can be configured as a level sensitive port for detecting the high level state of the port. [17]

In De1-SoC, many of the Cyclone V SoC FPGA’s pins are directly connected to LEDs, switches, keys and parallel inputs of the board. In order to access these peripherals and map them in the memory space of the system, the system makes use of the PIO core as a connection interface. The following figure presents the parameter window of a PIO core for controlling the 10 LEDs of the De1-SoC board.

![Image of PIO Core parameter window]

**Figure 5.12** PIO Core parameter window instance for led control.

The PIO core is configured as a ten, output-only port. Writing to the first register of the component and setting the bits to one or zero value, PIO core translates the register’s bits as...
the high or low state of the corresponding port, respectively. The following figure presents the block diagram of Qsys system design with the integration of PIOs Controllers.

![Block diagram of Qsys System Design with integration of PIO Controller IP cores.](image)

**Figure 5.13** Block diagram of Qsys System Design with integration of PIO Controller IP cores.

Of course, the Nios II system can be configured to provide many solutions to the embedded software programmers, by integrating additional peripherals. In this reference design system, Nios II system acts like the control sub-system that is responsible for camera and video initialization and video stream control. The above-described IP cores have their clock input connected to the clock output of the Clock Source core, operating at 50 Mhz.

5.1.3 D8M Camera Kit video capture and control interface

In order to provide video acquisition from D8M Camera Kit, the system integrates IP cores from Terasic that are used for pixel capturing and color conversion of the video stream and IP cores from Opencores.org for initialization and control of the camera kit.

The IP core which is connected to the output of D8M camera kit is *Terasic Camera IP Core*. This particular IP core was built from Terasic in order to provide Qsys system integration of D8M Camera Kit's video and image output. The input of the IP core is exported and the output makes use of the *Avalon Streaming Protocol* which is configured as *Avalon Streaming Source*. The figure below presents the IP core.

![Terasic Camera IP core instance.](image)

**Figure 5.14** Terasic Camera IP core instance.

The exported pins are grouped in four different ports, the *Camera Output Data port* which is 12-bit wide, the *Frame Value port*, the *Line Value port*, and the *Camera Pixel Clock port*
which are 1-bit wide each. These are the control and data signals which are routed from system output to Camera Kit pins, through the GPIO1 port of the De1-Soc board.

The Camera's Kit's sensor has an image array of 3296 columns and 2528 rows with 3264x2448 active pixels of the array. Figure 5.15 presents the color filters for the active pixels which are arranged in Bayern Pattern. The other pixels are used for black level calibration and interpolation. The Camera Kit with the use of MIPI Decoder outputs the active pixel data from the sensor to a 10-bit parallel pixel data format in Bayer Pattern. The Terasic Camera IP captures the pixels and translates the parallel Bayer Pattern data into RGB data and makes packets from the video frames in order to meet the specification of the Avalon Streaming Protocol.

**Figure 5.15 OV8865 Sensor array region color filter layout.**

The only parameter the Terasic Camera IP core provides to the system designer is the resolution of the input image. Figure 5.16 presents the parameter window of Terasic Camera IP core. This reference design system uses an image resolution of 640x480 pixels.
The output image resolution, frame rate, focus level and other parameters of the D8M Camera Kit, which are necessary for the initialization and the proper operation of the D8M Camera Kit, are configured through the registers of MIPI controller and MIPI's camera module components. In order to access their registers, the MIPI Decoder and the OV8865 camera module provide I2C serial interface for communication. The system uses an IP core designed from Opencores which implements an I2C serial Master interface.

The I2C Master core provides an Avalon MM Slave interface which is connected to Nios II Avalon MM Master and exports its communication clock and data line (SCL, SDA). For the D8M’s reference design, the system makes use of two different cores that provide the communication interface between the system and the D8M Camera Kit, one for the MIPI Decoder and one for the OV8865 camera module. The MIPI camera module consists from an OV8865 Camera Sensor and a Voice Coil Motor driver chip. The programmer must use the same I2C bus for both the camera sensor and the VCM. Connections of the camera kit were presented in figure 3.7. The two instances of the I2C Master IP core are implemented in the system design and presented in figure 5.17.

The D8M Camera Kit also provides two separate pins, one for reset of the OV8865 camera module and MIPI Decoder and a power-down pin for the OV8865 camera module. These pins are operated from Nios II processor with the use of two PIO IP cores. The PIO cores are configured as 1-bit outputs. The following figure shows the integration of the PIO cores in the system.
Figure 5.19 presents the revised Qsys system design block diagram with the integration of Terasic Camera IP core, I²C Controllers IP cores and the extra PIO cores for controlling the MIPI reset and power down pins.

### Video Sub-System

Capturing the video data frames from the camera kit doesn’t ensure a correct video flow to the VGA controller, which acts as the video output of the system. The input and output video data may differ to the frame rate they produce. This is why the reference design adopts a frame buffering method that can help solve throughput issues in the data path.

In this reference design, the Frame Buffer IP core from the Video and Image Processing (VIP) IP library of Intel is being used for the frame buffering process. This IP core accepts frames from its data input (Avalon Streaming Sink) and uses an external memory to write and read the buffered frames. The reading and writing of the frames are performed with two basic blocks which are presented in figure 5.20. The readied frames are then transferred to the data output of the core (Avalon Streaming Source) \[18\].
The following figure presents the parameter window of the Frame Buffer IP core and its functions a designer can import to the system.

**Figure 5.20 Frame Buffer Block Diagram.**

In the parameter window in the Image Data Format area, the designer has to complete the image resolution, the number of pixel of each color and the way the color planes are being transferred of the buffered frames. On the behavior area, the IP core gives the opportunity to select from multiple operations about the behavior of the IP core. In this design frame dropping, discard invalid frames, and frame repetition was chosen. When frame dropping and/or frame repeating are allowed, the IP core provides a triple-buffering process that is used to perform simple frame rate conversion. These options led to the need of 3.5 MB of memory space.

The Frame Buffer uses the Avalon Memory Map Interface in order to read and write frames in memory. This interface is configured in the Avalon Memory-Mapped Interface area of the parameter window. The use of on-chip memory that provides Avalon MM interface is quite
insufficient for the size of the buffered frames. For this purpose, the external 64 MB SDRAM memory of the De1-SoC is being used.

The system must have access to the SDRAM memory chip. In order to achieve that, an *SDRAM controller IP core* is being used. The SDRAM controller actually maps the memory of the external chip that is connected to and uses its Avalon MM Slave port to present a flat, contiguous memory space as wide as the SDRAM chip. The Avalon-MM interface behaves as a simple memory interface. There are no memory-mapped configuration registers. The physical connections between the Avalon Interface and the SDRAM memory are presented in the following figure.

![Figure 5.22 SDRAM Controller block diagram.](image)

The SDRAM Controller IP core has to get configured in order to meet the specifications of the external SDRAM memory chip. In the parameter window that follows figure 5.23 presents the *memory profiling* and the *timings* for the 64 MB IS42/45R16320D memory chip which is hosted on the De1-SoC HWRevF board. The parameters of the memory chip can be found in the device datasheet [19].

![Figure 5.23 SDRAM Controller IP core parameter window.](image)
The block diagram of Qsys system with the addition of Frame Buffer and SDRAM Controller is configured as follows.

The IP cores that consist the video sub-system along with the Terasic Camera IP core that capture and transform the pixel data from the Camera Kit, must operate under the same clock reference. In this case, the video sub-system operates at 100MHZ generated from outclk2 of Altera PLL that was presented earlier in this section.

With the above-described components, the system guarantees a correct video flow to the clocked video output. Right before that in this reference design, Terasic makes use of an Autofocus IP core. This IP core offers autofocus capabilities by user selection. The operation of autofocus is being done in two steps. First by writing the D8M Voice Coil Motor (VCM) driver IC registers, which controls the camera lens movements to perform image focusing and second by calculating the current image high-frequency component statistic. When the VCM drives the camera lens’ movement, a real-time statistics of image high-frequency sum is performed in every step of the lens movement. Finally, the lens moves to the position which has the largest number of high frequency to complete the automatic focus operation. In order to write to VCM IC registers the IP core uses the I2C bus of the camera module. The Autofocus IP core writes the I2C bus through its own I2C master controller. Users must make sure there is only one I2C master used at a time. Figure 5.25 presents the provided ports of the Terasic Autofocus IP core.
The din and dout ports are the Avalon ST sink and source ports respectively, and the exported port consists of the I2C SCL and SDA and a clock input port for connecting the 50 MHz clock.

The only parameter which can change in the parameter window of the Autofocus IP core is the video frame resolution. From figure 5.25, it is clear that the Autofocus IP core provides an Avalon MM Slave port which provides runtime control of the IP core. In this system, Nios II is connected to the Autofocus IP core through the Avalon MM Slave port, in order to provide users the ability to run or stop the autofocus function and to define the image resolution and the autofocus area through its registers. During the autofocus procedure, the streaming data remain unattached. Figure 5.26 presents the Qsys system design block diagram with the integration of Autofocus IP core.

The last component of the video sub-system is the Clocked Video Output IP core. The CVO core connects through an exported port to the VGA controller which in its turn displays
images on the VGA monitor. The CVO’s input is an Avalon ST Sink which captures the streaming frames. The Autofocus’ Avalon Streaming Source is connected to the data_in port. From the parameter window of the component, the designer must choose the right values in order to provide the correct video output. Figure 5.27 that follows presents the CVO IP core parameter window.

![Clocked Video Output](image-url)

**Figure 5.27** Clocked Video Output parameter window instance.

The necessary values that the user needs to configure are the image resolution, color space, and the synchronization. In the sync configuration field of parameter window, the user must provide the synchronization parameters for Horizontal Sync, Horizontal Front Porch, Horizontal Back Porch, Vertical Sync, and Vertical Front Porch. These parameters control the timing configuration and they based on the VGA timing format. For example, in 640x480@60fps VGA timing format the parameters are the one presented in figure 5.27.

Figure 5.28 presents the final form of Qsys system’s design block diagram with the addition of VCO IP core. The Qsys system design includes all the necessary information for the system’s components and their connections. In order to create the Qsys system for synthesis user must generate the Qsys system design. The creation of Verilog design files implements the instances and the interconnection of the system. Then HDL files are grouped under the “.qip” file which is found in the IP folder of the project.
5.1.5 Top-Level Routing and Pin Assignment

The generation of Qsys system for synthesis creates in Verilog language a Top-level module. This module describes the connections between the components of Qsys system, as well as its exported ports. The top-level Verilog file of the Qsys system module shares the same name as the ".qsys" file. In this reference design "Qsys.v" describes the Qsys system. This file along with the sub-module files of Qsys system are added to the project by adding the ".qip" file.

This reference design besides the Qsys System that was described earlier in this section, also makes use of a module which calculates the frame rate of the captured video from D8M camera kit. This module counts D8M’s MIPI_PIXEL_VS signal pulses in one second (it represents the fps) and converts the number to a decimal number to display on two seven-segments of the development board. Figure 5.29 presents the block diagram of the FpsMonitor module.

![Figure 5.28 Block diagram of Qsys System Design with integration of Video Output.](image)

![Figure 5.29 FPS Monitor Block Diagram.](image)
The system in its final form makes use of the Qsys system and the FPS Monitor modules. Their ports must be routed to the pins of the chip. In order to compile the system and make the routing to the physical pinout of the SoC FPGA chip, there is a need for creating a top-level design file. The top-level file describes the inputs and outputs of the system, the integration of the components and their interconnections and the connections to the inputs and outputs of the system.

De1-SoC hardware which brings the Cyclone V SoC FPGA chip has its components and connectors, connected directly to the chip pins. This is why there is a need for defining the De1-SoC peripherals that the system makes use, before the implementation of Top-Level design.

D8M’s reference design has its Top-Level module (DE1_SOC_D8M_SDRAM) which describes its inputs and outputs as the system’s I/Os. The Top-Level design file must contain I/Os signals that are necessary for describing the pins of the chip which are connected directly to the I/O of the components of the De1-SoC. The components the system needs to operate are the Clocks, the SDRAM, the Seven-Segments, the Keys, the Switches, the LEDs, the VGA controller and the GPIO_1 Connector for connecting the Camera Kit.

Summing up, Top-Level I/O signals are the connecting link between the physical pins of the chip and the signals of the modules of the system. Figure 5.30 presents the integration of Qsys system and FPS Monitor modules based on their description by the Top-Level design file. The Top-Level design file can be found in Appendix A.

Figure 5.30 Top Level Entity block diagram of reference design.

CLOCK_50, KEY[3..0], SW[9..0], LED[9..0], GPIO1, SEG7, SDRAM and VGA are the I/O signals of Top-Level module.
In order to assign the Top-Level module’s I/O signals to the correct physical pins of the chip, Pin Planer tool is used. The user must consult the De1-SoC user manual in order to map the custom logic to the peripherals I/O through the FPGA pins. The following table presents the pin assignments of the Push Buttons of the board.

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>FPGA Pin Number</th>
<th>Description</th>
<th>I/O Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY[0]</td>
<td>PIN_AA14</td>
<td>Push-button[0]</td>
<td>3.3V</td>
</tr>
</tbody>
</table>

Table 5.1 Pin Assignment of Push-buttons.

This operation must take place for every component or connector of the De1-SoC the user need to use. An easy way to implement pin assignments for De1-SoC board is by using a utility from Terasic which is called DE1-SoC System Builder. In the GUI of System Builder, users can enable or disable one or more components at will. If a component is enabled, the DE1-SoC System Builder will automatically generate its associated pin assignment, including the pin name, pin location, pin direction, and I/O standard all included in a .qsf file. This file must be imported in Quartus project by the menu Assignments → Import Assignments.

Figure 5.31 presents the pin assignments and pins physical position on the SoC FPGA chip which was fetched from Pin Planer tool.

Figure 5.31 Pin Planer Tool instance.
5.1.6 Timing Constraint Configuration

During the synthesis of the FPGA design by Quartus II, TimeQuest tool will be called. This tool will read in timing constraints files. The timing constraints files describe the timing for the FPGA, for example, the target frequency of the FPGA and the timing to external peripherals. This constraint file uses the Synopsys timing constraints description language.

In this reference design, the .sdc file describes the timing constraints for the signals to the SDRAM and the VGA controller of the FPGA system design. Based on the timing specifications, the Fitter makes the integration of the components on the FPGA fabric. After the fitting process, TimeQuest will then calculate the timing of the internal FPGA signals and compare these timings to the required timing given by the timing constraints files. The requirements must meet the calculated ones. If not, in the compilation status of the project, there will be critical warnings which describe the problems for the timing constraints of the design. These warnings are advised to get fixed in order for the system to operate normally.

5.2 HPS as system’s application processor

For the implementation of the pedestrian detection system, the reference design presented in the previous section was adopted and changes within the Designer Platform tool performed. The system must make use of the Hard Processor System of the SoC FPGA chip as the application processor. Consequently, it is necessary to establish the connections between the HPS logic and the FPGA fabric. In order to achieve that, Qsys system with its previously-presented form was enhanced with the integration of HPS as the application processor. The steps for modifying the Qsys system with the integration of the Cyclone V Hard Processor System IP core are presented in the current section. The core instance presented by figure 5.32 gives the first picture of the integration of HPS in Qsys system design.

![HPS IP core instance and exported ports](image)

The IP core provides the connecting interface between the HPS and the FPGA fabric, the pin configuration for HPS peripherals, HPS clock configuration and HPS memory configurations. The connecting interfaces to configure for the system are the Lightweight HPS to FPGA, the HPS to FPGA and the FPGA to HPS AXI Bridges and presented by the following figure.
5.2.1 HPS Peripheral Pin configuration

From the peripheral pin configurations tab of the IP core, the HPS peripherals ports can be routed on the SoC FPGA chip pins. For the De1-SoC board and in order to operate its peripherals, the configurations on the HPS IP core presented in figure 5.34 must take place.

The peripherals enabled by the HPS configuration IP core are, one Ethernet Media Access Controller (EMAC), the QSPI Flash Controller, the SD/MMC Controller, one USB Controller, one SPI Controller, one UART Controller and two I2C Controllers. These peripherals are used to control the Ethernet PHY for Gigabit Ethernet connection, the EPCQ flash memory, the microSD Card connected to microSD card slot, the USB PHY for the two USB ports, the LTC connector, UART to USB controller and the Accelerometer respectively. The second I2C Controller is used for the LTC connector as well. Figure 5.35 presents the physical position of the peripheral controllers of the De1-SoC board.
5.2.2 HPS PLL configuration

In the *HPS Clocks configuration* tab, the IP core holds information necessary for input and output clocks of the HPS. In the *input clocks* tab, the external clock sources of the HPS are configured at 25 MHz. On the *output clocks* tab, there are options for the clock sources for the Peripherals and the main core of the HPS. For peripherals, the user must set the desired frequencies generated by the *Peripherals PLL*. There are also options for the configuration of the desired frequencies of the MPU, L3, L4 and Debug interface generated from the Main PLL of the HPS. In this system design the Peripherals PLL clock source is the *EOSC1* and the frequency of the MPU set to default. These settings are presented in figure 5.36.

![Clock configuration instance of the HPS IP core.](image)

---

*Figure 5.35* HPS Peripheral physical controller connections.

*Figure 5.36* Clock configuration instance of the HPS IP core.
5.2.3 HPS External Memory configuration

On the SDRAM tab of the HPS IP core, there are options for configuring the *SDRAM Controller of the HPS*. The SDRAM tab is divided into four tabs, the *PHY Settings*, the *Memory Parameters*, the *Memory Timings*, and *Board Settings*.

All these parameters were set based on the SDRAM of the De1-SoC Board. The specific 1 GB DDR3 Memory is operating at 400 MHz. The clock source of the memory is generated by the main PLL of the HPS. The Memory parameters and Timings parameters are applied based on the memory’s manufacturer datasheet.

At last, on the Board Settings tab users must set the necessary parameters for accessing the external memory, by specifying these parameters which are unique for the De1-SoC Board. A guide for setting these parameters correctly is to follow the *Golden Hardware Reference Design* of the De1-SoC Board.

5.2.4 Replacing Nios II with HPS on Qsys system

Replacing Nios II with the Hard Processor System as the application processor of the system, from the hardware perspective, is not a tough task. In Qsys system design, the *slave components* of the system that were mastered by Nios II processor must be connected to the *Lightweight HPS to FPGA AXI Bridge*. This is because these components don’t occupy much address space and can be mapped only from the Lightweight HPS to FPGA AXI Bridge. The HPS now has access to these peripherals from the Lightweight Bridge. Figure 5.37 presents part of the connections. Figure 5.38 presents the block diagram of the enhanced Qsys system design.

![Figure 5.37](image)  
*Figure 5.37* Lightweight HPS to FPGA AXI Master – Slave connection.
5.3 Linux OS Video Output interface

The system presented in this thesis adopts the ability of De1-SoC Board to boot Linux from an inserted microSD card. Through a wide variety of Linux images that Intel and Terasic offer, the system makes use of a command line "Linux distribution with a build in frame buffer" driver, provided by Terasic. Linux OS, as well as an application program that runs on Linux, can access the frame buffer and write the pixel data.

5.3.1 Frame Reading from HPS

In order to display the frames of Linux's frame buffer on the monitor through the VGA connector, the pixel data must be read by the FPGA sub-system, because the VGA controller is accessed only from the FPGA fabric. This is achieved by implementing a Frame Reader IP core. This core reads the video frames stored in external memory and outputs them as a video stream.

From the IP core perspective, the parameters must follow the reference design of De1-SoC – DE1_SOC_Linux_FB which comes along with this Linux distribution, because Linux OS has a specific configuration based on memory addresses from the reference design. Linux's frame buffer produces frames organized at 1024x768 pixel resolution in four color planes with 8-bit consist the value per color plane. The color planes are the Blue, Green, Red, and
Alpha (BGRA) values of every pixel. These are the first five parameters of the Frame Reader IP core. Figure 5.39 presents the instance of the Frame Reader IP core parameters.

The parameters of the IP core are defined as follows. The first five parameters reveal the properties of the video Avalon-ST output from the IP core. Like it was described earlier, the video frames consist of 1024x768 pixels with the 1024 value defining the image width and 768 value the image height. The pixels are formed from four colors each represented from an 8-bit value. Then the IP core transmits the color planes of each pixel in parallel.

The Master port width represents the width of the port in bits, from where the IP core captures the pixel values. In this design, the Master port drives the FPGA to HPS AXI Bridge which was configured as a 128-bit wide data bus (see Figure 5.33). From Qsys system perspective, Frame Reader masters the whole address range of HPS and has access to every single component in the HPS address space. In order to guide the Frame Reader to capture the pixel data, run-time configuration from Linux OS during the boot sequence takes place. That's why users that use the specific distribution of Linux OS must follow the reference design that comes with it. In this system design, Avalon-MM Slave port of the IP core must be configured with a base address 0x100.

Frame Reader IP core also demands a different clock rate for the different interfaces. The first clock and reset inputs are for the Avalon MM Slave and Avalon Streaming Source and the clock master and reset master inputs are for the Avalon MM Master port. Frame Reader’s input master clock is connected to outclk2 of PLL because this is the synchronization clock for the video streaming interface of the system. The master's clock is connected to the 50 MHz output from Clock Source.

The streaming output of the Frame Reader IP core is in BGRA format. Since the system doesn't require the Alpha value, it adopts a Color Plane Sequencer IP core which changes the way the color plane samples are transmitted across the Avalon-ST interface. In this instance, the CPS IP core is used to discard the Alpha color plane and also reverse the color planes from BGR to RGB. The following figures present the input and output configuration of the CPS IP core.
In order to meet the format of input and output stream of the IP core, the bits per pixel per color plane is set to 8 and the color planes are transmitted in parallel. Now the video stream output of the Linux OS is in RGB format, with each pixel expressed by 8 bits per color plane. The block diagram of Qsys system with the addition of Frame Reader and Color Plane Sequencer IP cores is configured as follows.
5.3.2 Picture in Picture display

At this point, the system has two parallel video streams, the Linux OS video output, and the camera video stream, each one needs to find the way out to the VGA controller and connector. In sub-section 5.1.4 the Clocked Video Output IP core was presented. This core provides the connecting interface between the Avalon-ST interface and the VGA controller. The problem is that the CVO IP core provides only one Avalon-ST sink and doesn’t provide picture-in-picture ability. That’s why the system integrates a Video Mixing IP core.

In this system design, the Alpha Blending Mixer IP core is being used to provide picture-in-picture display of the two different video streams. Based on their video resolution, the system adopts the bigger video stream for the background and the smaller video stream for the foreground. Linux OS video stream with its 1024x768 pixel resolution acts as the background layer of the video output stream and camera video stream with its 640x480 resolution acts as the foreground layer. Figure 5.43 simulates the picture-in-picture display.
Design and Implementation of a Real-Time Pedestrian Detection System on SoC FPGA

Figure 5.43 Picture in Picture display format.

Alpha Blending Mixer IP core accepts run-time control from its Avalon-MM slave port with access to the registers for the location, and control of each foreground layer. This means that the foreground layer can be manually set to a position within the background layer, with background’s dimensions acting as boundaries. The user needs to be accurate in order to meet the criteria of the foreground layer to fit within the dimensions of the background layer or else the mixing IP core stalls. From the previous figure and for the given image resolution of the background and the foreground layer, one can figure out that the maximum values for the offsets of foreground layer are $1024-640=384$ for the $x$-offset and maximum value for the $y$-offset is $768-480=288$.

This IP core must have its two streaming inputs in the same color format with an equal bit representation. In the parameter window of the Alpha Blending Mixer IP core the information given for the operation of the core, are the background width and height, the bits per pixel per color plane and the way these planes are streamed. The Alpha Blending Mixer IP core can also provide more than two layers for mixing. Figure 5.44 presents the parameters of the IP core.

![Figure 5.44 Alpha Blending Mixer IP core configuration.](image)

The connections of the IP core are configured as follows. The $din_0$ sink (background layer) connects with the Color Plane Sequencer’s streaming source which provides the Linux OS video output stream. At this point the Autofocus IP core presented in the previous section is discarded from system design. The mixing core’s $din_1$ sink (foreground layer 1) is
connected on the *streaming source of Frame’s Buffer* which provides the camera video stream. The *streaming output* of the mixing core is connected to the *CVO’s Avalon-ST interface*. The Avalon-MM Slave *control port* is connected to the Lightweight H2F AXI Bridge.

Connecting the Alpha Blending Mixer streaming source to the Clock Video Output streaming sink Clock Video Output IP core must be re-configured in order to provide the correct video output for the input of 1024x768 pixel resolution. Figure 5.45 presents the parameters of the CVO IP core that was modified in order to meet the specification of the 1024x768@60Hz VGA format.

![Clock Video Output](image)

*Figure 5.45 Clock Video Output IP core reconfiguration.*

The 1024x768@60Hz VGA format also requires a pixel clock operating at 65 MHz. In order to obtain that, reconfiguration of PLL’s VGA clock output (outclk3) must be applied and set to 65 MHz. Clock source of the IP cores that handle the stream also needs to be reconfigured at double the rate of VGA clock output in order to operate normally. For this reason, the SDRAM clock source (outclk1) and the system clock (outclk2) is set to 130 MHz. The block diagram of Qsys system, with the addition of Alpha Blending Mixer, the removal of Autofocus, the configuration of CVO and PLL IP cores, is configured as follows. Figure 5.47 presents the revised Top-Level Entity of the system with the integration of the revised Qsys system (Figure 5.46).
Figure 5.46 Block diagram of Qsys System Design with integration of Alpha Blending Mixer.
5.4 Camera video stream reading by Linux OS

Implementing the pedestrian detection process on Linux OS requires the transmission of camera frames to the HPS part of the chip. In sub-section 3.6.4 the architecture of the hardware sub-system was presented, along with the process behind the application program that runs on Linux OS, which reads a camera frame and performs the detection process with the use of OpenCV library. This frame reading process requires the establishment of a data path between the camera video streaming process on the FPGA and Linux OS. The idea behind this task was to duplicate the video stream after the frame buffering process and capture one of the two streams by saving its pixel data in on-chip RAM memory. From that time, the application program is able to capture the camera frames by reading the on-chip memory data.

5.4.1 Duplicate video stream

The system has to perform frame capturing from the video stream and at the same time transmit the video stream to the VGA output, in a way previously-presented with the picture-in-picture process. For this purpose, the system duplicates the camera video stream right after the frame buffer. For the process of frame duplicating Color Plane Sequencer IP core is being used.

The configuration of the core is quite simple. The input of the CPS IP core must meet the profile of the frame buffer’s video output stream. In this design, the input is configured as BGR with the color planes transmitted in parallel. The key for transforming the CPS IP core in order to perform duplication of the video stream is to enable the second output port dout1 and then set the two output ports dout0 and dout1 with the same configuration as the input.
port *din0*. Figure 5.48 and figure 5.49 present the parameter window of the Color Plane Sequencer IP core.

![Port Configuration](image)

**Figure 5.48** Parameter window of the CPS IP core as Avalon Stream duplicator – Input configuration.

![Parameter window](image)

**Figure 5.49** Parameter window of the CPS IP core as Avalon Stream duplicator – Output configuration.

Now the system is able to provide two separate streaming paths of the input camera video stream. The first stream is directed on the mixing IP core for the PIP process and the second stream stands-by for implementing the frame reading process from Linux OS (HPS side). Figure 5.50 presents the block diagram of the Qsys system with the integration of the CPS IP core configured as video stream duplicator.
5.4.2 Scale and color transformation the video stream

The second video stream which is the result from the duplication of the video camera stream is transformed in size and color space before the writing process to on-chip RAM. This is because, of the limitation of the on-chip memory size which can handle only a small amount of data to be stored compared to the ~1MB on the original frame of 640x480 RGB-8bit of the video camera stream.

In order to save a whole frame on the on-chip memory the video camera stream is scaled on half the size of its resolution, resulting in a video camera stream of 320x240 pixel resolution RGB 8-bit per frame. In order to implement the scaling process, the Scaler II IP core is used by the system. The input resolution is configured with 640x480 pixels, and the output resolution is configured with 320x240 pixels as seen in figure 5.51. For the scaling process no run-time control is required.
The second step for reducing the size of the frames of the video camera stream is to convert the stream from BGR color space to gray-scale color space. This is implemented by importing the Color Space Converter IP core from the UP library into the streaming flow. The only configuration of this IP core is the selection of the color conversion which presented in figure 5.52.

Scaller II IP core from VIP library core and the Color Space Converter IP core from the UP IP library use different control packet format. In order to connect their source and sink ports respectively, an Avalon-ST Adapter IP core was used as a packet format interchange. This adapter transforms the control packets of the video stream in order to meet the control requirements for both ends. For this system design, the Avalon-ST adapter was configured as the following figure 5.53. Upstream configuration was set for the source output and the downstream was set for the sink input.
5.4.3 Storing the video frames on on-chip memory

The video camera stream on the second duplicated video stream has now consisted from frames of 320x240 pixels in grayscale. Next step is to store the frames of the video stream on on-chip RAM memory. In order to store the frames a DMA Controller IP core from the UP library is being used. This core was configured to perform a DMA transaction of the pixels of the frames packets which are shaped as Avalon-ST packets and head them to on-chip memory.

The DMA controller IP core is configured to store the pixel data in a memory consecutive, starting from the base address of the memory. Figure 5.54 presents the parameters necessary for the streaming port that describes the stream, which this case is a video stream with frames of 320x240 pixels resolution with an 8-bit value representation per pixel (grayscale format).
The DMA controller’s Avalon-MM master port was designed to connect to a memory. This system integrates an on-chip RAM memory capable of storing a frame of 320x240 pixels with an 8-bit value representation per pixel. These frames require 76800 bytes, thus an on-chip memory larger than that is enough for the job.

The on-chip memory of the system is configured as follows. The system makes use of the On-Chip RAM/ROM IP core which is configured as an on-chip RAM. The IP core offers two slave ports which provide simultaneous access to the memory region from two different masters, in this case, the DMA Controller Master port, and the H2F AXI Master port. Figure 5.55 presents the configuration of the On-Chip RAM/ROM IP core for the current system design.
**On-Chip Memory (RAM or ROM)**

<table>
<thead>
<tr>
<th>Memory type</th>
<th>RAM (Writable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual-port access</td>
<td></td>
</tr>
<tr>
<td>Single clock operation</td>
<td></td>
</tr>
<tr>
<td>Read During Write Mode</td>
<td>DON'T CARE</td>
</tr>
<tr>
<td>Block type</td>
<td>AUTO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data width</td>
<td>32</td>
</tr>
<tr>
<td>Total memory size</td>
<td>100000 bytes</td>
</tr>
<tr>
<td>Minimize memory block usage</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Read latency</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Slave s1 Latency</td>
<td>1</td>
</tr>
<tr>
<td>Slave s2 Latency</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ROM/RAM Memory Protection</th>
<th>Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset Request</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 5.55  On-chip RAM Memory parameter window*

Figure 5.56 presents the integration of Scaller, DMA Controller and On-Chip RAM in system design. Due to lack of space in the block diagram, the figure doesn't present Color Plane Converter IP core which transforms the video stream from RGB to Grayscale.
5.5 Memory Map of Qsys System and Top Level Integration

Figure 5.56 presents the block diagram of Qsys System in its final form. From this block diagram, one can easily figure out the multiple Master-Slave relationships between the components of the system. These address maps are the access point of the hardware components which consisting the system and presented in the Designer Platform editor. Each Master port has at least one Slave port access which is described by a memory region (described in Chapter 4).

Appendix A presents the memory map of Qsys system and the Master-Slave relationships between the components. Qsys system integration is performed on the Top-Level design. Keeping in mind the changes that took part, the designer must configure the Top-Level I/Os, the Qsys components I/O and the pin assignment to meet the desired hardware configuration. The Top-Level design file of the FPGA sub-system of this thesis is presented in Appendix B. Figure 5.57 presents the integration of Qsys system on Top-Level Design,
the interconnection between FPGA and HPS and the integration of Top-Level Design on the Cyclone V SoC FPGA chip.

**Figure 5.57** Block diagram of integration of Qsys System, Top-Level Design and FPGA HPS interconnections on the Cyclone V-SoC FPGA chip.
Chapter 6  System’s Software Implementation

Creating an embedded system also requires the development of embedded software, which is formed to control the particular embedded system’s hardware. In this thesis, the real-time pedestrian detection system requirements guided the design and implementation of the application. The application, which runs on Linux OS is responsible for the control of the system’s hardware. The control of the hardware requires the initialization and configuration of the hardware components. Also, it has to implement the pedestrian detection process with the use of OpenCV library. This application and its libraries are written in C and C++, cross-compiled in a host computer which runs Linux (Ubuntu 16.04 OS).

This chapter presents the implementation of the software application, the adaption of OpenCV functions and techniques on the software application on how to access the memory mapped components of the FPGA sub-system. The chapter begins by presenting the procedure on setting up the main tools (SoC EDS, ARM DS-5) for building Linux applications for Cyclone V SoC FPGA and how to build OpenCV library to make it operate under the Linux OS that runs on ARM A9 cores.

6.1    Setting up the tools for cross compilation

Since it is not an easy task to write and compile an application program consisting from multiple files and libraries directly in the Embedded Console Linux OS environment, writing and compiling of the software must take place in a host computer. The tools which help for cross-compilation are the SoC EDC, and the ARM DS-5 based on Eclipse IDE. SoC EDC provides the hardware libraries for accessing the peripherals (including the FPGA fabric) of the ARM and ARM DS-5 provides the compilers which are necessary for cross-compiling a program dedicated for ARM, on the host machine.

The application makes use of the OpenCV library and its functions, thus, a build of the OpenCV source files with the use of ARM compilers that ARM DS-5 provides is prerequisite.
This section provides the necessary information to build OpenCV libraries with CMake tool and ARM compilers, and the steps to set up the Eclipse IDE for the current thesis software application. The host computer runs Ubuntu 16.04.

6.1.1 OpenCV build for cross compilation

In order for OpenCV to operate under the environment of ARM-based Linux OS which runs on the embedded ARM microprocessor of Cyclone V SoC FPGA chip, it is necessary to build OpenCV library with ARM compilers. This section provides the information for building OpenCV library with CMake GUI on the host machine. In the host machine, users must have installed, Git, Cmake 2.6 or higher, pkgconfig and Python 2.6 or higher, along with the Quartus, SoC EDS, and ARM DS-5. The commands that follow were executed in a terminal of the host machine.

The first thing needs to be done, is to download OpenCV source files. From terminal create a working directory in order to save OpenCV source files by cloning them with Git:

```
mkdir MasterThesisCV
cd MasterThesisCV
git clone https://github.com/opencv/opencv.git
```

Next step is to build the OpenCV library. Create a build directory and run CMake GUI.

```
mkdir opencv_build
cmake-gui
```

In the CMake GUI first step is choosing the source code directory and build directory as figure 6.1 presents. The first one targets the OpenCV source code downloaded from GitHub and the second one is the build directory which created for OpenCV built operation. Next step is CMake build configuration.

![Figure 6.1 Cmake GUI instance for choosing build and source folder.](image)
In Figure 6.2 and in the dialog box that emerges, users need to choose the options for cross-compilation of OpenCV library. This option allows the selection of the desirable compilers for compiling the OpenCV library. Figure 6.3 presents the next dialog box where users can select the location of the compilers which come with ARM DS-5. These compilers are located under the Quartus root directory. In this presentation, the QUARTUS_ROOTDIR is /home/ellabuser/altera/15.1/. Based on the Quartus root directory the C compiler is located in QUARTUS_ROOTDIR/embedded/ds-5/sw/gcc/bin/arm-linux-gnueabihf-gcc, the C++ compiler is located in QUARTUS_ROOTDIR/embedded/ds-5/sw/gcc/bin/arm-linux-gnueabihf-g++, and in the target root field users must choose the bin directory where the compilers are located.
By pressing Finish CMake GUI generates the building variables. These variables configure the build of OpenCV library, and by adding or removing these options, Cmake configures the build. An option to consider is located under CMAKE option, in CMAKE_INSTALL_PREFIX which presents the installation directory of the library. This option is very important because this is the location of the compiled libraries of OpenCV that the application is going to use. After configuring all the desirable options press Configure and Generate.

![CMake GUI configuration of library install path.](image)

The last step, was to run make command from the build directory. When make is completed without errors (if errors exist re-run CMake and configure the building options), execute install.

```
    cd opencv_build
    make
    sudo make install
```

Now the cross-compiled OpenCV library is located under the directory "arm_opencv" in the root directory.

### 6.1.2 Arm DS-5 configuration for cross compilation

A great tool for developing an application which runs on ARM is ARM DS-5. ARM DS-5 is based on Eclipse IDE and comes with the necessary compilers for compiling bare-metal and Linux-based software application for ARM A9 of Cyclone V SoC FPGA. Since SoC EDC provides the necessary tools and libraries for building an application for the SoC system, it is recommended to run *Altera Embedded Command Shell* which is located in the embedded
folder under Quartus root directory. From the host’s machine terminal change directory, (in this case `/home/ellabuser/altera/15.1/embedded`) and execute the command to open Embedded Command Shell. From this point, users can access the tools of SoC EDC by executing commands on terminal. In order to open ARM DS-5 application, the user must execute on terminal the command `eclipse`.

```
cd /home/ellabuser/altera/15.1/embedded
./embedded_command_shell.sh
eclipse
```

The thesis application is written in C and C++ language and makes use of the OpenCV library. For this purpose, this section presents the configuration for writing a C++ program on ARM DS-5. The first step is to start a new C++ project. Figure 6.5 presents the initialization of the project, where the user must place the desirable name of the project and in the toolchain field, select the necessary compilers. In order to build an application to run on Linux, arm-linux-gnueabihf compilers are being used. In the next tab, user can select the debug and release option of the application.

![C++ Project](image)

**Figure 6.5 Creating a new Linux application project on Arm DS-5.**

Before creating a new C++ file, it is mandatory to set up the required libraries for Cyclone V SoC FPGA and OpenCV for the project. This configuration must be done for every project. On the properties of the project, navigate on the Tools Settings under C/C++ Build → Settings tab. Figure 6.6 shows the instance of the properties window.
In this field, the following settings need to get set. Under the GCC C++ Compiler 4 sub-menus:

- **Dialect:** *Language Standard: ISO C++11*
- **Preprocessor:** *Defined Symbols: add soc_cv_av*
- **Include:** *Include Paths (libraries path): add /arm_opencv/include/opencv, /arm_opencv/include, QUARTS_ROOTDIR/embedded/ip/altera/hps/altera_hps/hwlib/include/soc_cv_av, QUARTS_ROOTDIR/embedded/ip/altera/hps/altera_hps/hwlib/include*

under the GCC C Compiler 4 sub-menus:

- **Symbols:** *Defined Symbols: add soc_cv_av*
- **Includes:** *Include Paths (libraries path):*

```
add QUARTS_ROOTDIR/embedded/ip/altera/hps/altera_hps/hwlib/include/soc_cv_av, QUARTS_ROOTDIR/embedded/ip/altera/hps/altera_hps/hwlib/include
```

and under the GCC C++ Linker 4 sub-menus

- **Libraries:** *Libraries: add opencv_objdetect, opencv_highgui, opencv_core, opencv_imgproc*
- **Libraries:** *Library search path: add /arm_opencv/lib.*
In the previously-presented settings, the libraries paths must follow the SoC EDC and OpenCV directories paths that were presented earlier. In the Libraries sub-menu of the C++ linker, the libraries values refer to the compiled options that were presented in 6.1.1 during OpenCV cross-compilation.

6.1.3 Demonstrating OpenCV application on De1-SoC

Now that the software project was configured to include OpenCV library, it is ready to create a C++ file to write the demonstration application (Code 6.1). This application will read an image from the file system by using `imread` function included in OpenCV library, and it will be shown by writing its pixels on the monitor, with the use of frame buffer. More details about the writing of pixels on the monitor are presented in the following section.

```cpp
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include "opencv2/imgcodecs.hpp"
#include <opencv2/core.hpp>
#include "fbdraw.h"
#include <stdio.h>
#include <iostream>
using namespace std;
using namespace cv;

int main(){
    Mat image;
    dev_fb fb;
    fb_init(&fb);

    image = imread("upatras.bmp", CV_LOAD_IMAGE_COLOR); // Read the file

    if(! image.data ) // Check for invalid input
    {
        cout << "Could not open or find the image" << std::endl;
        return -1;
    }

    int channels = image.channels();
    int nRows = image.rows;
    int nCols = image.cols * channels;
    int i,j,x,y;
    uchar r,g,b;
    for( i = 0; i < nRows; ++i )
    {
        for ( j = 0; j < nCols; j=j+3 )
        {
            y = i ;
            x = j/3 ;

            r = image.at<uchar>(i,j);
            g = image.at<uchar>(i,j+1);
            b = image.at<uchar>(i,j+2);
            //if we want offset of the image x+ y+
            fb_drawPixel(&fb,x+106,y+400,b,g,r);
        }
    }

    return 0;
}
```

**Code 6.1** Demonstration code for OpenCV on De1-SoC.
After the successful compilation of `imshow` project, Arm DS-5 creates the execution file in the release folder. The execution file is created with the same name as the project, in this case, `imshow`. In order to run the application, user must transfer the execution file on Embedded Linux OS that runs on De1-SoC System, along with the compiled library of OpenCV, discussed on sub-section 6.1.1.

For this demonstration, the system makes use of a Linux distribution with a build in frame buffer and an FPGA configuration file created from Terasic. The system files are available from Terasic on a single image file, called `DE1_SoC_FB.img` which is stored on a microSD card. The FPGA fabric is configured while booting Linux through microSD card which starts on power up of De1-SoC board. For the video output of the system, VGA output and a monitor connected on the VGA D-SUB were used.

Terasic suggests, running the Linux Terminal of De1-SoC Linux system from a host machine, through the USB-to-UART interface of De1-SoC board at 115200 baud rate. After the successful boot of the system (Linux OS and FPGA configuration), the following figure presents the result on the monitor connected to De1-SoC board.

![Initial screen of De1-SoC Linux system's video output.](image)

For this demo, the cross-compiled library of OpenCV was copied under the root directory of Linux OS. The demonstration application and the image to display, have been copied on the home directory of Linux OS. Figure 6.8 presents the copied files in the Linux system. Before executing the demonstration program, user must set the library path for OpenCV, with the following command:

```bash
export LD_LIBRARY_PATH=/arm_opencv
```
After following all the steps, the system is ready for the execution of `imshow` application file. Figure 6.9 presents the result of the demonstration, which is the displayed image (University of Patras logo) on the monitor.

![Figure 6.8 Commands executed from Putty terminal.](image)

![Figure 6.9 System’s monitor display after imshow application executed on De1-SoC.](image)

### 6.2 Build of system’s application

This section provides all the information about the build of the application that handles the operation of the real-time pedestrian detection system. Summing up the requirements, this application must set the parameters of hardware components through their registers, initialize, and handle the camera kit, read and write image data, and of course, perform the pedestrian detection process with the use of OpenCV library.

#### 6.2.1 Generating address map header file

Making use of the hardware components of Qsys sub-system, from an application that runs on the Linux OS which runs on HPS side of the chip, can be very tricky. The memory map of
Qsys system changes every time a component with a master-slave connection is being added. In order to keep up with the progress of Qsys system design, it is easier to link the slave addresses of Qsys components connected to masters of the HPS with the name of the components. For this purpose Intel provides a tool integrated into SoC EDC, which can help users generate a header file (.h), containing the address map and other details of the Qsys system. In order to generate this header file, user must navigate to the directory of the Quartus project, where Qsys.sopcinfo file is located and execute the following script.

```
./generate_hps_qsys_header.sh
```

This script generates a header file with the name of HPS IP core of Qsys system. In the thesis system implementation Qsys system’s HPS IP core name is hps_0, which results to the header file hps_0.h. This header file contains macros for the module of HPS and the components connected to its Lightweight HPS to FPGA AXI Master and HPS to FPGA AXI Master. Code 6.2 presents the definition of \textit{I2C Opencores Camera Component}.

\begin{verbatim}
/* Macros for device 'i2c_opencores_camera', class 'i2c_opencores'
* The macros are prefixed with 'I2C_OPENCORES_CAMERA_'.
* The prefix is the slave descriptor.
*/
#define I2C_OPENCORES_CAMERA_COMPONENT_TYPE i2c_opencores
#define I2C_OPENCORES_CAMERA_COMPONENT_NAME i2c_opencores_camera
#define I2C_OPENCORES_CAMERA_BASE 0x0
#define I2C_OPENCORES_CAMERA_SPAN 128
#define I2C_OPENCORES_CAMERA_END 0x7f
\end{verbatim}

\textbf{Code 6.2} Part of the generated hps_0.h header file.

The most critical definitions of every component of this header file are the ones that define the base address of the component. These are the one used to calculate the address of every component, which is connected on the HPS. The base address of every component shows the offset from the base address of the master connected to, in this case, the Lightweight H2F AXI Bridge Master. The calculation of the base address of the components, needed from the application to have access to, will be discussed in the following lines. The whole hps_0.h header file is presented in Appendix C.

\subsection*{6.2.2 Accessing FPGA and Qsys peripherals}
An application running on Linux on the ARM processor on Cyclone V SoC FPGA, communicates with the hardware Qsys components through the H2F and LW H2F AXI Bridges which are connected to the FPGA fabric. Figure 5.57 which presents the hardware design of the system and its interconnection, one can easily figure out, the H2F and LW H2F AXI Bridges which connect the Qsys components with the HPS and therefore with ARM microprocessor and Linux OS. Chapter 4 presented how these bridges are mapped into regions of the MPU memory space. It also presented how the hardware Qsys components are mapped into the region of the two bridges in respect to their connection on Qsys. These components and their registers are seen as peripherals from the MPU, and mapped in its address space (Figure 4.26).

A peculiarity of Linux Operating System is that an application which runs on Linux cannot access the hardware address space directly, because it is in protected mode by Linux OS. The physical system memory is represented and can be accessed through a system memory \texttt{device} file, \texttt{/dev/mem}. Access into this file at some offset is equivalent of accessing physical memory at the offset address. Linux also provides the function called \texttt{mmap}, which is a function that maps a file into virtual memory. By using \texttt{mmap} to map the \texttt{/dev/mem} file into memory, physical addresses could be mapped to virtual addresses, providing programs access to physical addresses. This procedure was used in this thesis application, to access
the physical memory region of the two HPS to FPGA Bridge’s memory span to communicate with the Qsys components and presented in the following lines.

First thing needs to be done, is to open the memory device file /dev/mem and assign it to a handler. This handler will be used from the mmap function.

```c
if( ( fd = open( "\./dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 ) {
    printf( "ERROR: could not open \./dev/mem\...\n" );
    return( 1 );
}
```

The mmap function returns a pointer which offers access to the virtual memory space. The offset and span of the virtual memory space are determined by the arguments of the mmap function. In this implementation, was chosen to implement two different address spaces, one for the Lightweight AXI Bridge address span and one for the H2F AXI Bridge address span. With the lw_axi_master and h2f_axi_master pointers, the application can access the FPGA Qsys components.

```c
lw_axi_master = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, HW_REGS_BASE );

if( lw_axi_master == MAP_FAILED ) {
    printf( "ERROR: Lightweight H2F AXI Bridge mmap() failed...\n" );
    close( fd );
    return( 1 );
}

h2f_axi_master = mmap( NULL, H2F_AXI_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, ALT_FPGASLVS_OFST );

if( h2f_axi_master == MAP_FAILED ) {
    printf( "ERROR: H2F AXI Bridge mmap() failed...\n" );
    close( fd );
    return( 1 );
}
```

Qsys component registers are accessed by the application starting from the components base address plus their master virtual address. This procedure was performed for every Qsys component of the system which is connected to the HPS. Components offset addresses are defined by the macros of hps_0.h discussed earlier in this section. The following code represents access to Qsys components, connected with the Lightweight H2F AXI Bridge on the HPS.

```c
lw_fpga_led_addr = (int*)(lw_axi_master + LED_BASE);
lw_fpga_mipi_pdn_n = (int*)(lw_axi_master + MIPI_PWDN_N_BASE);
lw_fpga_mipi_reset_n = (int*)(lw_axi_master + MIPI_RESET_N_BASE);

h2p_lw_mipi_camera = (int*)(lw_axi_master + I2C_OPENCORES_CAMERA_BASE);

h2p_lw_mipi_ctrlr = (int*)(lw_axi_master + I2C_OPENCORES_MIPI_BASE);

lw_fpga_mip_addr = (int*)(lw_axi_master + ALT_VIP_MIX_0_BASE);

lw_fpga_dma_addr = (int*)(lw_axi_master + ALT_VIP_CL_CLP_1_BASE);
lw_fpga_dma_control = (int*)(lw_axi_master + VIDEO_DMA_CONTROLLER_0_BASE);
```

For the FPGA on-chip RAM which is connected to HPS through the HPS to FPGA AXI Bridge, the application has access to, by adding its base address with the H2F AXI Bridge virtual base. The pointer that targets the base of the address space of on-chip RAM was implemented with the following line of code.

```c
onchip_ram = (int*)(h2f_axi_master + ONCHIP_MEMORY_BASE);
```
Writing or reading data to the virtual memory space, starting from the virtual address specified by these pointers, provides direct access to the registers of Qsys components. Their registers are 32-bit wide, which results in a four-byte ascending for defining the next register of the specified component. Reading and writing to the registers is implemented with the use of a reading function, \textit{IORD}, and a write function \textit{IOWR}. These two functions require as arguments the base address of the component to access, and the offset of the register to access. \textit{IOWR} also requires an argument that specifies the data to write in the defined register. These two functions are defined in the build of the program.

\begin{verbatim}
#define IORD(base, index) (*( ((uint32_t *)base)+index))
#define IOWR(base, index, data) (*((uint32_t *)base)+index) = data)
\end{verbatim}

In this thesis, most of Qsys components in system design require run-time control in order to operate. This requires writing to their registers in order to achieve specific control and operation.

For debugging purposes, this system adapts the ten red LEDs of De1-SoC. In order to access these LEDs, implemented in Qsys system design, \textit{PIO Controller IP core} was used. Looking back on Chapter 5, this \textit{PIO core} configured as a 10-bit output port. These I/Os are connected directly with a negative logic (configured on Top-Level Design) to the inputs of the ten red LEDs. The negative logic is required because the LED’s anode is hardwired on VCC. Writing the first register (offset 0) of the \textit{PIO core} is translated to set/reset of the I/O ports. In this thesis, the application starts by setting all ten bits of the first register at ‘1’ logic value which is translated as the \textit{HIGH} state on the output of \textit{PIO port} which in turn enables all ten LEDs.

\begin{verbatim}
IOWR(lw_fpga_led_addr, int(0), int(0x3ff));
\end{verbatim}

One basic component for the display of the application is the \textit{Alpha Blending Mixer IP core}. This component is configured only on runtime by the application. In this system, \textit{Alpha Blending Mixer} has its background layer connected to the 1024x768 pixel video output of the Linux OS and Layer 1 is fed with the streaming from the D8M camera kit. Runtime control requires the correct set up for the component’s control registers. In the following table, the \textit{Alpha Blending Mixer}’s control registers map is presented.

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Control Bit</td>
<td>Bit 0 of this register is the Go bit, all other bits are unused.</td>
</tr>
<tr>
<td>1</td>
<td>Status</td>
<td>Bit 0 of this register is the Status bit, all other bits are unused</td>
</tr>
<tr>
<td>2</td>
<td>Layer 1 X</td>
<td>Offset in pixels from the left edge of the background layer to the left edge of layer 1.</td>
</tr>
<tr>
<td>3</td>
<td>Layer 1 Y</td>
<td>Offset in pixels from the top edge of the background layer to the top edge of layer 1.</td>
</tr>
</tbody>
</table>
| 4       | Layer 1 Active | • If set to 0—data from the input stream is not pulled out.  
|         |            | • If set to 1—layer 1 is displayed.  
|         |            | • If set to 2—data in the input stream is consumed but not displayed.         |
| 5       | Layer 2 X  | The rows in the table are repeated in ascending order for each layer from 1 to the foreground layer... |

\textbf{Table 6.1} Alpha Blending Mixer Control Register Map.

Alpha Blending Mixer has a register for controlling its operation. By setting its control register (offset 0) bit 0 with logic value ‘1’, makes Alpha Blending Mixer enabling its operation and starts by streaming the background layer.

\begin{verbatim}
//Enable the Alpha Blending Mixer Component
IOWR(lw_fpga_mix_addr,0,0x01);
\end{verbatim}
For control of the foreground layers of the mixer component, there are three registers (offset 2,3,4 for layer 1, offset 5,6,7 for layer 2, etc...) which control the offset x, offset y and the display operation for every layer, respectively. In this application, Layer 1 top left corner was set at 555 pixels offset from the left edge of the background layer, and 200 pixels from the top edge of the background layer. After setting the desired offset of the layer, the application sets Layer 1 Active register with value ‘1’, which enables the display of the layer.

```c
void display_mixer_layer(int layer, int offset_x, int offset_y){
    if (layer < 1 || layer >12) layer = 1;
    if (offset_x < 0 || offset_x >1024-321) offset_x = 0;
    if (offset_y < 0 || offset_y >768-241) offset_y = 0;
    int x_register = (layer-1)*3 + 2;
    int y_register = (layer-1)*3 + 3;
    int display_register = (layer-1)*3 + 4;
    //Set x,y for the displayed Layer 1 of Mixer
    IOWR(lw_fpga_mix_addr,x_register,offset_x); //555 0x022B
    usleep(20);
    IOWR(lw_fpga_mix_addr,y_register,offset_y); //200 0x96
    usleep(20);
    //Display Layer 1 of Mixer
    IOWR(lw_fpga_mix_addr,display_register,0x01);
    usleep(20);
}
```

6.2.3 Initializing D8M camera kit and video capture

The first device needs to get initialized is the camera kit. This requires initialization of the camera sensor module and the MIPI Decoder. These modules are accessed from the two I2C interfaces which implemented in FPGA fabric, by adapting `I2C Oopencores IP core` in Qsys system design.

This thesis application has adapted the C library of the reference design of D8M Camera Kit. This library provides the basic functions to initialize, read and write to I2C IP cores and in turn to I2C buses connected to camera sensor and MIPI Decoder. These functions can be found in `I2C core (.h and .c files)`. These functions are adapted from the other three pair of files which consists the library, the `mipi_bridge_config (.h and .c files), mipi_camera_config (.h and .c files) and camera_func (.h and .c files).

In `Camera_func.h` file is the main function of the library, which initialize the camera kit. Right before calling D8M’s initialization function users must set the MIPI decoder Power Down pin and Reset pin at a high state. These two pins are configured as two separate 1-bit output PIO IP cores. In this application these two bits were set, by setting their PIOs control registers, first with zeros, and then with ones, causing D8M to restart. The registers are being set the same way like in the LEDs case.

```c
IOWR(lw_fpga_mipi_pwdn_n, 0x00, 0x00);
IOWR(lw_fpga_mipi_rest_n, 0x00, 0x00);
usleep(200);
IOWR(lw_fpga_mipi_pwdn_n, 0x00, 0xFF);
usleep(200);
IOWR(lw_fpga_mipi_rest_n, 0x00, 0xFF);
usleep(200);
```

At this point, the application calls the `MIPI_Init` function. This function is configured to return the success or failure of the initialization of the I2C bus of the MIPI Decoder, which is described with ‘1’ if successful, otherwise returns ‘0’.
In `MIPI_Init()` function the first step is to initialize the I²C bus of MIPI Decoder. As arguments of I²C bus initialization function `oc_i2c_init_ex`, were set, the virtual base address of I²C Opencores IP core for the MIPI Decoder, the frequency of reference clock of the IP core (50 MHz), and the frequency of the clock of I²C bus (SCL: 400 KHz).

The second step is the initialization of the MIPI Decoder, with the use of `MipiBridgeInit` function. This function first read the MIPI Decoder chip ID and compares it with the expected one (0x4401).

```c
int MIPI_Init()
{
  int bSuccess;
  bSuccess = oc_i2c_init_ex(h2p_lw_mipi_contrlr, 50*1000*1000, 400*1000);
  if (!bSuccess) printf("failed to init MIPI- Bridge i2c\r\n");
  usleep(50*1000);
  MipiBridgeInit();
  usleep(500*1000);
  MipiCameraInit();
  usleep(1000);
  oc_i2c_uninit(h2p_lw_mipi_contrlr);
  return bSuccess;
}
```

`MipiBridgeInit` function continues by setting the MIPI Decoder's registers. This is done by reading the register address and the desired value from an array of structures and then writing the value on the defined address on the I²C bus. The structure that holds the address and the desired value is configured as follow. The registers and the defined values were set in the `MipiBridgeReg` array. The settings that this array of structure holds are the one to configure the MIPI Decoder's PLL to output the reference clock for the camera sensor module, and the settings for MIPI Decoder to output the data in parallel. The most critical registers are the one that set the PLL of MIPI Decoder, which generates the reference clock for the camera sensor module (MCLK: 25MHz) and the pixel clock which is fed to FPGA to deal with the parallel data (PCLK: 25/50MHz). Figure 6.10 presents the clock tree of the D8M Camera kit.

![Figure 6.10 Clock tree of D8M Camera Kit](image)

```c
ttypedef struct{
  alt_u16 Addr;
  alt_u16 Data;
} SZ_MIPI_REG_T;
```
System's Software Implementation

```c
static SZ_MIPI_REG_T MipiBridgeReg[] = {
    {0x0002,0x0001}, // System Control Register
    {0x0002,0x0000}, // System Control Register
    ...
};
```

In order to set these registers, the `MipiBridgeRegWrite` function is used. This function provides an easy way to write the registers of MIPI Decoder, by setting only the register address and the value to be written. The I²C address and the base address of the I²C Opencores were set into the function.

```c
void MipiBridgeRegWrite(alt_u16 Addr, alt_u16 Value){
    const alt_u8 device_address = MIPI_BRIDGE_I2C_ADDR;
    OC_I2CL_Write(h2p_lw_mipi_contrlr, device_address, Addr, (alt_u8 *)&Value, sizeof(Value));
}
```

The third and fourth step of the initialization of D8M Camera kit, are the initialization of the I²C bus of the camera sensor module and then, the initialization of camera sensor module OV8865. Both the operations are held inside the `MipiCameraInit` function. This function starts by enabling the I²C bus of the camera sensor module the same way as I²C bus of the MIPI Decoder. Then it writes to the registers of camera sensor module OV8865 to initialize the frame capturing. This is done by reading an array of structure. This structure is slightly different from the previous one, as it holds an additional field, in order to implement operation like, delay or setting the end of the script.

```c
typedef struct{
    alt_u8 Type;
    alt_u16 Addr;
    alt_u8 Data;
}SZ_CONFIG_T;
```

```c
static SZ_CONFIG_T MipiCameraReg[] = {
    {0x6c,0x0103, 0x01}, // software reset
    {TIME_DELAY, 0, 10},
    {0x6c,0x3638, 0xff}, // analog control
    ...
    {END_OF_SCRIPT, 0, 0}
};
```

These registers were being set by using the `OV8865_write_cmos_sensor_8` function, which implements data transfers to the I²C bus of the camera sensor module, but only in the address of the camera sensor (I²C address: 0x6C). This is because the camera sensor and the voice coil motor (I²C address: 0x18) share the same I²C bus. The function to write a register of the voice coil motor is the `OV8865_write_AF` function. These two functions set the desired I²C address and write on the I²C bus with the given register and value.

```c
void OV8865_write_cmos_sensor_8(alt_u16 Addr, alt_u8 Value){
    const alt_u8 device_address = MIPI_I2C_ADDR;
    OC_I2CL_Write(h2p_lw_mipi_camera, device_address, Addr, (alt_u8 *)&Value, sizeof(Value));
}
```

```c
void OV8865_write_AF(alt_u8 msb, alt_u8 lsb){ // VCM149C
    const alt_u8 device_address = MIPI_AF_I2C_ADDR;
    OC_I2C_Write(h2p_lw_mipi_camera, device_address, msb, (alt_u8 *)&lsb, sizeof(lsb));
}
```
The last step of D8M Camera Kit initialization process is to release the two separate I2C buses. This is done, by calling the `oc_i2c_uninit` function. This function takes as argument the base address of I2C Opencores IP Core.

```c
oc_i2c_uninit(h2p_lw_mipi_contrlr);
oc_i2c_uninit(h2p_lw_mipi_camera);
```

### 6.2.4 Reading a video frame

At this point, the hardware initialization performed, the video camera streaming has been initialized, captured and displayed on the monitor through the FPGA fabric, while at the same time the video frames of the camera stream are duplicated and saved on an on-chip RAM in grayscale format. The HPS and consequently the application running on Linux OS, have access to the video frames by reading the on-chip RAM, through the H2F AXI Bridge.

The goal for the application is to capture those frames, one frame at a time, perform the pedestrian detection process and display the frame with the detected pedestrians. This process is explained by the following figure.

Performing the pedestrian detection process with the use of OpenCV library requires the captured frames to be handled by the OpenCV library `Mat` object. This object handles the image data along with the information of the image resolution and color depth. In this system, `Mat` object is initialized with a 320x240 pixel image resolution and with the 8-bit grayscale color format. For the data of the captured frame, the `Mat` object points on an array initialized to have the size of the frame to be captured (320*240 = 76800).

```c
def read_video(frame, image_array):
    # Initialize Mat object
    img = Mat(RES_Y, RES_X, CV_8UC1, &image_array)

    # Save the pixels of the frame on the on-chip memory space
    for (j=0; j<RES_Y; j++)
        on_chip_row_step=j*(RES_X/step);
        row = j*RES_X;
        for(i=0; i<RES_X; i+=step){
            int read_step = i/step;
            pixel_data = *((uint64_t *)onchip_ram)+read_step+on_chip_row_step;
            image_array[i+row]=uchar(pixel_data & 0xff);
            image_array[i+row+1]=uchar((pixel_data>>8) & 0xff);
            image_array[i+row+2]=uchar((pixel_data>>16) & 0xff);
            image_array[i+row+3]=uchar((pixel_data>>24) & 0xff);
            image_array[i+row+4]=uchar((pixel_data>>32) & 0xff);
            image_array[i+row+5]=uchar((pixel_data>>40) & 0xff);
            image_array[i+row+6]=uchar((pixel_data>>48) & 0xff);
            image_array[i+row+7]=uchar((pixel_data>>56) & 0xff);
        }
```

**Code 6.3 Reading process of a frame from on-chip memory.**

Reading from on-chip memory is performed with 64-bit data at a time and stored in `pixel_data` variable which consists of 8 pixels of the frame. Then the `pixel_data` variable is
shifted and stored in image_array. Now the img Mat object holds the frame that read from on-chip memory.

6.2.5 Perform Pedestrian Detection

Performing pedestrian detection on an image frame, with the use of OpenCV library, was presented in section 2.4 (Chapter 2). At this point, making use of the detectMultiscale function with the parameter for the input image the captured frame from on-chip memory will perform and return the detection results for the frames that the D8M camera captures. These results are fed into the rectangle function which draws the detection rectangles in an RGB image, clone of the captured frame image. This is done to draw the rectangles with a green color, for a better display of the results. Code that follows presents the detectAndDisplay function that was created for this purpose. As arguments take the readied frame as Mat object and frame buffer's device object.

```c
void detectAndDisplay( Mat frame_gray, dev_fb fb )
{
    std::vector<Rect> rects;
    Mat frame;
    cvtColor(frame_gray,frame, COLOR_GRAY2BGR);
    hog.detectMultiscale(frame_gray, rects, hit_threshold, win_stride, padding, scale, gr_threshold);
    for (size_t i = 0; i < rects.size(); i++){
        rectangle(frame, rects[i], CV_RGB(0, 255, 0), 1);
    }
    displayImage(fb, frame, 150, 150);
}
```

**Code 6.4 Function for performing the pedestrian detection process and display of the result.**

After drawing the rectangles on the frame Mat object, the application handles the display of the frame. The display of the frame is performed on the frame buffer of the Linux OS. For this reason, detectAndDisplay function has an input argument the instance of the frame buffer device. This, along with the image to display, and the coordinates of the top-left corner of the image feed as arguments on the displayImage function.

6.2.6 Displaying an Image on Linux OS frame buffer device

Displaying an image from the system file or displaying the resulted frame from pedestrian detection process on the Linux OS, the frame buffer device is required to show the results of the system in a user-friendly environment. For this purpose displayImage function created for displaying Mat objects in the Linux frame buffer device. This function accepts as arguments the frame buffer’s device object, the Mat object to display and the offset x and offset y of the position of the top left corner of the frame, against the Linux OS’s 1024x768 display output.

This function handles the Mat object pixel RGB data and draws them with the use of fb_drawPixel function. This function along with the constructor of the frame buffer device can be found on fbdraw (.h and .c files) library. This abstraction-level library for manipulating the framebuffer via /dev/fb0, created from Keith Lee for its DEVBOX project [20]. The fb_drawPixel function accepts as arguments the frame buffer’s device instance, the position of the pixel (x and y offset from the top left corner of Linux OS’s display output) and the color values of the pixel in BGR format. The following code presents the displayImage function.

```c
void displayImage(dev_fb fb, Mat image, int offset_x, int offset_y){
    int channels = image.channels();
    int nRows = image.rows;
    int nCols = image.cols * channels;
    int i,j,x,y;
```
uchar r,g,b;
for( i = 0; i < nRows; i++)
{
    for ( j = 0; j < nCols; j+= channels)
    {
        y = i;
        x = j/channels ;
        r = image.at<char>(i,j);
        g = image.at<char>(i,j+1);
        b = image.at<char>(i,j+2);
        //if we want offset of the image x+ y+
        fb_drawPixel(&fb, x + offset_x, y + offset_y, b, g, r);
    }
}

Code 6.5 displayImage function used to display images on Linux OS display output.

6.2.7 Close devices
The application must end its operation by closing the previously opened Linux devices and unmapping any virtual memory maps created earlier. In this application, the end of the main function is performing by unmapping the virtual memory map created for the Lightweight H2F AXI Bridge memory map and for the HPS to FPGA AXI Bridge memory map. This is done by using the memory unmap function munmap, which accepts as arguments the pointer of the virtual memory map and its span.

Last but not least, users must consider closing all the devices opened during the application runtime. By closing the devices, the application releases the device file for another application or system process may want to have access. For the memory device file this is done with system function close and for the frame buffer device the application makes use of the fb_close function included in fbdraw library. The above processes are presented in the following lines of code.

if (munmap (lw_axi_master, HW_REGS_SPAN) != 0)
{
    printf (“ERROR: munmap() failed...
”);
    return (-1);
}
if (munmap (h2f_axi_master, H2F_AXI_SPAN) != 0)
{
    printf (“ERROR: munmap() failed...
”);
    return (-1);
}
close(fd);
fb_close(&fb);

Code 6.6 Closing opened devices and unmapping virtual addresses.
Chapter 7  Results & Conclusion

The main goal of this thesis was to design and implement a functional System on Chip, based on hardware-software co-design, able to perform real-time pedestrian detection. This system implemented with the use of De1-SoC board carrying Cyclone V SoC FPGA chip, along with a D8M Camera kit for video capture. The target was to discover the capabilities of SoC FPGA chip in designing a fully functional embedded system for object detection, reconfigurable from one end to the other (from the camera sensor to video output).

The biggest challenge was to drive the design of the system and test its distinct parts, like the video streaming path and its interconnections. Also, a lot of effort was placed to accelerate the distinct processes with the use of the FPGA fabric while at the same time take the best out of the ARM A9 microprocessor, which accelerated the design process.

7.1 Results

The above-described requirements resulted in a functional system which is capable of performing pedestrian detection, on the video captured frames from D8M Camera kit. Hardware acceleration performed only on video streaming process, and detection process operates on the Linux OS running on ARM A9. This design resulted to a system which captures frames from the D8M camera kit at 640x480 pixel resolution @ 60fps, but the approach of software-only implementation of HOG plus SVM resulted to pedestrian detection performance on nearly 4.5 resized frames per second. These frames have a resolution of 320x240 pixels and fetched from the 60 frames captured from D8M. The timings measured on the application with OpenCV’s `getTickCount` and `getTickFrequency` functions.

Pedestrian Detection performed in: 242.532 ms
Reading frame from on-chip RAM performed in: 5.21513 ms
Detect Multiscale performed in: 224.543 ms
Show image process performed in: 11.8129 ms
Figure 7.1 presents the way the application fetches the intermediate captured frames from on-chip RAM.

![Diagram showing frames fetched from on-chip RAM](image)

**Figure 7.1** Perform Pedestrian detection on frames fetched from on-chip RAM.

The following figures present the video output of the system. On the center of the VGA monitor, are presenting two frames. On the right side the camera video streaming resized at 320x240 pixel resolution @60fps, which is coming from the camera sensor, and on the left side the result of the detection process performed by the system application in a frame of 320x240 pixels resolution in nearly @5fps. The system tested in several conditions, two of them are demonstrating in the following figures. In figure 7.2, the system performs pedestrian detection in an internal area, in this case, the lab.

![Image showing video output](image)

**Figure 7.2** Demo of the system with D8M camera capturing a video playing from a smartphone.
In the second demonstration which presented in figure 7.3 the system is performing pedestrian detection, but in that case, in front of the camera, it has been placed a smartphone which displays a demo video with pedestrians passing by.

![Demo of the system with D8M camera capturing the lab room.](image)

### 7.2 Problems

A lot of work that didn’t pay off took place on configuring the camera sensor of the D8M camera kit. The OV8865 camera sensor has plenty of registers needed to configure, in order to calibrate the sensor to take the best out of it. That was a big setback for the system’s implementation because it took many hours to figure out the details of the OV8865 registers through its datasheet, which sadly are not well documented. Also, the 64MB SDRAM turned out to be a bottleneck when the system designed to achieve higher resolution from the camera video stream. Frame buffer uses the SDRAM for storing the video frames of the video streaming process. That requires the clock of the memory to operate at double the rate of the pixel clock, but in De1-SoC board case, the SDRAM maximum clock maximum frequency is 100MHz.

A problem that was successfully overcome was issues which faced with the integration of Terasic Camera IP core component and some of the Intel's VIP IP core components. At first, there was a big issue with the Terasic Camera IP core on different versions of Quartus, as it some cases the IP core didn’t perform at all. That narrowed us to design the system on Quartus v15.1, which guided to change many of the VIP IP core components which have differences in other Quartus versions.
7.3 Future work

This system can be an important basis for the development of vision-based embedded systems. The configurability that provides from the camera sensor till the video output gives the opportunity to enhance the system with other hardware and software computer vision algorithms.

On this particular design, a future work that will certainly improve the system is the hardware acceleration of pedestrian detection algorithm. HOG hardware implementation which is the most time-consuming process of the system along with parallel SVM software or hardware implementation can speed up the system many times.

Also, there is plenty of room for building the system to operate at a higher video resolution. One can experiment with the calibration of OV8865 camera sensor which can capture images as high as 8MP on lower frame rate. There is also the option for providing a different video input on the system instead of the D8M camera kit, with slight changes on system design.
References


v/cv_5v4.pdf [Accessed February 2019]


Design and Implementation of a Real-Time Pedestrian Detection System on SoC FPGA
The Quartus project files can be found on my Github page: https://github.com/PhilipChalkiopoulos/DE1_SOC_PedestrianDetection
Appendix B

Hardware sub-system, Top-Level Design File

`define ENABLE_HPS

module DE1_SOC_Linux_FB(
    inout   ADC_CS_N,
    output  ADC_DIN,
    input   ADC_DOUT,
    output  ADC_SCLK,

    ///////// AUD /////////
    input     AUD_ADCDAT,
    inout     AUD_ADCLRCK,
    inout     AUD_BCLK,
    output    AUD_DACDAT,
    inout     AUD_DAclrCK,
    output    AUD_XCK,

    ///////// CLOCK2 /////////
    input     CLOCK2_50,

    ///////// CLOCK3 /////////
    input     CLOCK3_50,

    ///////// CLOCK4 /////////
    input     CLOCK4_50,

    ///////// CLOCK /////////
    input     CLOCK_50,

    ///////// DRAM /////////
    output    [12:0] DRAM_ADDR,
    output    [1:0]  DRAM_BA,
    output    DRAM_CAS_N,
    output    DRAM_CKE,
    output    DRAM_CLK,
    output    DRAM_CS_N,
    inout     [15:0] DRAM_DQ,
    output    DRAM_LDQM,
    output    DRAM_RAS_N,
    output    DRAM_UDQM,
    output    DRAM_WE_N,

    ///////// FAN /////////
    output    FAN_CTRL,

    ///////// FPGA /////////
    output    FPGA_I2C_SCLK,
    inout     FPGA_I2C_SDAT,

    ///////// GPIO /////////
    inout     [35:0] GPIO_0,

    ///////// HEX0 /////////
    output    [6:0]  HEX0,

    ///////// HEX1 /////////
)
output [6:0] HEX1,

///////// HEX2 //////////
output [6:0] HEX2,

///////// HEX3 //////////
output [6:0] HEX3,

///////// HEX4 //////////
output [6:0] HEX4,

///////// HEX5 //////////
output [6:0] HEX5,

`ifdef ENABLE_HPS
/////////// HPS /////////////
inout HPS_CONV_USB_N,
output [14:0] HPS_DDR3_ADDR,
output [2:0] HPS_DDR3_BA,
output HPS_DDR3_CAS_N,
output HPS_DDR3_CKE,
output HPS_DDR3_CK_N,
output HPS_DDR3_CK_P,
output HPS_DDR3_CS_N,
output [3:0] HPS_DDR3_DM,
inout [31:0] HPS_DDR3_DQ,
inout [3:0] HPS_DDR3_DQS_N,
inout [3:0] HPS_DDR3_DQS_P,
output HPS_DDR3_ODT,
output HPS_DDR3_RAS_N,
output HPS_DDR3_RESET_N,
input HPS_DDR3_RZQ,
output HPS_DDR3_WE_N,
output HPS_ENET_GTX_CLK,
inout HPS_ENET_Int_N,
output HPS_ENET_MDC,
inout HPS_ENET_MDI_O,
inout HPS_ENET_RX_CLK,
inout [3:0] HPS_ENET_RX_DATA,
inout HPS_ENET_RX_DV,
output [3:0] HPS_ENET_TX_DATA,
output HPS_ENET_TX_EN,
inout [3:0] HPS_FLASH_DATA,
output HPS_FLASH_DCLK,
output HPS_FLASH_NCSO,
inout HPS_GSENSOR_INT,
inout HPS_I2C1_SCLK,
inout HPS_I2C1_SDAT,
inout HPS_I2C2_SCLK,
inout HPS_I2C2_SDAT,
inout HPS_I2C_CONTROL,
inout HPS_KEY,
inout HPS_LED,
inout HPS_LTC_GPIO,
output HPS_SD_CLK,
inout HPS_SD_CMD,
inout [3:0] HPS_SD_DATA,
output HPS_SPI1_CLK,
inout HPS_SPI1_MISO,
output HPS_SPI1_MOSI,
inout HPS_SPI1_SS,
inout HPS_UART_RX,
output HPS_UART_TX,
input      HPS_USB_CLKOUT, 
input      [7:0] HPS_USB_DATA, 
input      HPS_USB_DIR, 
input      HPS_USB_NXT, 
output     HPS_USB_STP, 
`endif /*ENABLE_HPS*/

///////// IRDA /////////
input      IRDA_RXD, 
output     IRDA_TXD, 

///////// KEY /////////
input      [3:0] KEY, 

///////// LEDR /////////
output     [9:0] LEDR, 

///////// PS2 /////////
inout      PS2_CLK, 
inout      PS2_CLK2, 
inout      PS2_DAT, 
inout      PS2_DAT2, 

///////// SW /////////
inout      [9:0] SW, 

///////// TD /////////
inout      TD_CLK27, 
inout      [7:0] TD_DATA, 
inout      TD_HS, 
output     TD_RESET_N, 
inout      TD_VS, 

///////// VGA /////////
output     [7:0] VGA_B, 
output     VGA_BLANK_N, 
output     VGA_CLK, 
output     [7:0] VGA_G, 
output     VGA_HS, 
output     [7:0] VGA_R, 
output     VGA_SYNC_N, 
output     VGA_VS, 

///////// GPIO_1, GPIO_1 connect to D8M-GPIO //////////
inout      CAMERA_I2C_SCL, 
inout      CAMERA_I2C_SDA, 
output     CAMERA_PWDN_n, 
output     MIPI_CS_n, 
inout      MIPI_I2C_SCL, 
inout      MIPI_I2C_SDA, 
output     MIPI_MCLK, 
input      MIPI_PIXEL_CLK, 
input      [9:0] MIPI_PIXEL_P, 
input      MIPI_PIXEL_HS, 
input      MIPI_PIXEL_VS, 
output     MIPI_REFCLK, 
output     MIPI_RESET_n

);
// internal wires and registers declaration
wire [1:0] fpga_debounced_buttons;
wire [3:0] fpga_led_internal;
wire hps_fpga_reset_n;
wire clk_65;
wire [7:0] vid_r, vid_g, vid_b;
wire vid_v_sync;
wire vid_h_sync;
wire vid_datavalid;

//=======================================================
// Structural coding
//=======================================================
assign VGA_BLANK_N = 1'b1;
assign VGA_SYNC_N = 1'b0;
assign MIPI_CS_n = 0;

///////////////////////////////////
wire MIPI_PIXEL_CLK_d;
reg MIPI_PIXEL_VS_d;
reg MIPI_PIXEL_HS_d;
reg [9:0] MIPI_PIXEL_D_d;
assign MIPI_PIXEL_CLK_d = ~MIPI_PIXEL_CLK;
always @(posedge MIPI_PIXEL_CLK_d) begin
MIPI_PIXEL_VS_d <= MIPI_PIXEL_VS;
MIPI_PIXEL_HS_d <= MIPI_PIXEL_HS;
MIPI_PIXEL_D_d <= MIPI_PIXEL_D;
end

// Debounce logic to clean out glitches within 1ms
debounce debounce_inst ( .clk (CLOCK3_50),
.reset_n (hps_fpga_reset_n),
.data_in (KEY),
.data_out (fpga_debounced_buttons) );
defparam debounce_inst.WIDTH = 2;
defparam debounce_inst.POLARITY = "LOW";
defparam debounce_inst.TIMEOUT = 50000; // at 50Mhz this is a
debounce time of 1ms
defparam debounce_inst.TIMEOUT_WIDTH = 16; // ceil(log2(TIMEOUT))
soc_system u0 ( .clk_clk (CLOCK_50),
.reset_reset_n (hps_fpga_reset_n),
.memory_mem_a (HPS_DDR3_ADDR),
.memory_mem_ba (HPS_DDR3_BA),
.memory_mem_ck (HPS_DDR3_CK_P),
.memory_mem_ck_n (HPS_DDR3_CK_N),
.memory_mem_cke (HPS_DDR3_CKE),
.memory_mem_cs_n (HPS_DDR3_CS_N),
.memory_mem_ras_n (HPS_DDR3_RAS_N),
.memory_mem_cas_n (HPS_DDR3_CAS_N),
.memory_mem_we_n (HPS_DDR3_WE_N),
.memory_mem_reset_n (HPS_DDR3_RESET_N),
.memory_mem_dq (HPS_DDR3_DQ),
.memory_mem_dqs (HPS_DDR3_DQS_P),
.memory_mem_dqs_n (HPS_DDR3_DQS_N),
);
.memory_mem_odt (HPS_DDR3_ODT),
.memory_mem_dm (HPS_DDR3_DM),
.memory_oct_rzqin (HPS_DDR3_RZQ),

.hps_0_hps_io_hps_io_emac1_inst_TX_CLK (HPS_ENET_GTX_CLK),
.hps_0_hps_io_hps_io_emac1_inst_TXD0 (HPS_ENET_TX_DATA[0]),
.hps_0_hps_io_hps_io_emac1_inst_TXD1 (HPS_ENET_TX_DATA[1]),
.hps_0_hps_io_hps_io_emac1_inst_TXD2 (HPS_ENET_TX_DATA[2]),
.hps_0_hps_io_hps_io_emac1_inst_TXD3 (HPS_ENET_TX_DATA[3]),
.hps_0_hps_io_hps_io_emac1_inst_RXD0 (HPS_ENET_RX_DATA[0]),
.hps_0_hps_io_hps_io_emac1_inst_RXD1 (HPS_ENET_RX_DATA[1]),
.hps_0_hps_io_hps_io_emac1_inst_RXD2 (HPS_ENET_RX_DATA[2]),
.hps_0_hps_io_hps_io_emac1_inst_RXD3 (HPS_ENET_RX_DATA[3]),

.hps_0_hps_io_hps_io_qspi_inst_I00 (HPS_FLASH_DATA[0]),
.hps_0_hps_io_hps_io_qspi_inst_I01 (HPS_FLASH_DATA[1]),
.hps_0_hps_io_hps_io_qspi_inst_I02 (HPS_FLASH_DATA[2]),
.hps_0_hps_io_hps_io_qspi_inst_I03 (HPS_FLASH_DATA[3]),
.hps_0_hps_io_hps_io_qspi_inst_SS0 (HPS_FLASH_NCSO),
.hps_0_hps_io_hps_io_qspi_inst_CLK (HPS_FLASH_DCLK),

.hps_0_hps_io_hps_io_sdio_inst_CMD (HPS_SD_CMD),
.hps_0_hps_io_hps_io_sdio_inst_D0 (HPS_SD_DATA[0]),
.hps_0_hps_io_hps_io_sdio_inst_D1 (HPS_SD_DATA[1]),
.hps_0_hps_io_hps_io_sdio_inst_CLK (HPS_SD_CLK),
.hps_0_hps_io_hps_io_sdio_inst_D2 (HPS_SD_DATA[2]),
.hps_0_hps_io_hps_io_sdio_inst_D3 (HPS_SD_DATA[3]),

.hps_0_hps_io_hps_io_usb1_inst_D0 (HPS_USB_DATA[0]),
.hps_0_hps_io_hps_io_usb1_inst_D1 (HPS_USB_DATA[1]),
.hps_0_hps_io_hps_io_usb1_inst_D2 (HPS_USB_DATA[2]),
.hps_0_hps_io_hps_io_usb1_inst_D3 (HPS_USB_DATA[3]),
.hps_0_hps_io_hps_io_usb1_inst_D4 (HPS_USB_DATA[4]),
.hps_0_hps_io_hps_io_usb1_inst_D5 (HPS_USB_DATA[5]),
.hps_0_hps_io_hps_io_usb1_inst_D6 (HPS_USB_DATA[6]),
.hps_0_hps_io_hps_io_usb1_inst_D7 (HPS_USB_DATA[7]),
.hps_0_hps_io_hps_io_usb1_inst_CLK (HPS_USB_CLKOUT),
.hps_0_hps_io_hps_io_usb1_inst_STP (HPS_USB_STP),
.hps_0_hps_io_hps_io_usb1_inst_DIR (HPS_USB_DIR),
.hps_0_hps_io_hps_io_usb1_inst_NXT (HPS_USB_NXT),

.hps_0_hps_io_hps_io_spim1_inst_CLK (HPS_SPIM_CLK),
.hps_0_hps_io_hps_io_spim1_inst_MOSI (HPS_SPIM_MOSI),
.hps_0_hps_io_hps_io_spim1_inst_MISO (HPS_SPIM_MISO),
.hps_0_hps_io_hps_io_spim1_inst_SS0 (HPS_SPIM_SS),

.hps_0_hps_io_hps_io_uart0_inst_RX (HPS_UART_RX),
.hps_0_hps_io_hps_io_uart0_inst_TX (HPS_UART_TX),

.hps_0_hps_io_hps_io_i2c0_inst_SDA (HPS_I2C1_SDAT),
.hps_0_hps_io_hps_io_i2c0_inst_SCL (HPS_I2C1_SCLK),

.hps_0_hps_io_hps_io_i2c1_inst_SDA (HPS_I2C2_SDAT),
.hps_0_hps_io_hps_io_i2c1_inst_SCL (HPS_I2C2_SCLK),

.hps_0_hps_io_hps_io_gpio_inst_GPIO09 (HPS_CONV_USB_N),
.hps_0_hps_io_hps_io_gpio_inst_GPIO35 (HPS_ENET_INT_N),
Design and Implementation of a Real-Time Pedestrian Detection System on SoC FPGA

```vhdl
// SDRAM
.clk_sdram_clk (DRAM_CLK),
.sdram_wire_addr (DRAM_ADDR),
.sdram_wire_ba (DRAM_BA),
.sdram_wire_cas_n (DRAM_CAS_N),
.sdram_wire_ce (DRAM_CKE),
.sdram_wire_cs_n (DRAM_CS_N),
.sdram_wire_dq (DRAM_DQ),
.sdram_wire_dqm (DRAM_UDQM, DRAM_LDQM),
.sdram_wire_ras_n (DRAM_RAS_N),
.sdram_wire_we_n (DRAM_WE),

// I2C of camera and mi pi
.i2c_opencores_camera_export_scl_pad_io (CAMERA_I2C_SCL),
.i2c_opencores_camera_export_sda_pad_io (CAMERA_I2C_SDA),
.i2c_opencores_mipi_export_scl_pad_io (MIPI_I2C_SCL),
.i2c_opencores_mipi_export_sda_pad_io (MIPI_I2C_SDA),

// KEYS of mi pi
.mipi_pwdn_n_external_connection_export (CAMERA_PWDN_N),
.mipi_reset_n_external_connection_export (MIPI_RESET_N),

// d8m Camera module
.d8m_xclk1in_clk (MIPI_REFCLK),
.terasic_camera_0_conduit_end_D (MIPI_PIXEL_D_d[9:0],2'b00),
.terasic_camera_0_conduit_end_FVAL (MIPI_PIXEL_VS_d),
.terasic_camera_0_conduit_end_LVAL (MIPI_PIXEL_HS_d),
.terasic_camera_0_conduit_end_PIXCLK (MIPI_PIXEL_CLK_d),
.clk_vga_clk (VGA_CLK),

// i tc
.alt_vip_itc_0_clocked_video_vid_clk (VGA_CLK),
.alt_vip_itc_0_clocked_video_vid_data ([VGA_R,VGA_G,VGA_B]),
.alt_vip_itc_0_clocked_video_underflow (),
.alt_vip_itc_0_clocked_video_vid_datavalid (vid_datavalid),
.alt_vip_itc_0_clocked_video_vid_v_sync (VGA_VS),
.alt_vip_itc_0_clocked_video_vid_h_sync (VGA_HS),
.alt_vip_itc_0_clocked_video_vid_f (),
.alt_vip_itc_0_clocked_video_vid_h (),
.alt_vip_itc_0_clocked_video_vid_v (),
);

FpsMonitor uFps(
   /*input*/ .clk50(CLOCK2_50),
   /*input*/ .vs(MIPI_PIXEL_VS),
   /*output reg [7:0]*/ .fps(),
   /*output reg [6:0]*/ .hex_fps_h(HEX1),
   /*output reg [6:0]*/ .hex_fps_l(HEX0)
);
```
The Quartus project files can be found on my Github page: https://github.com/PhilipChalkiopoulos/DE1_SOC_PedestrianDetection
Appendix C

main.c file of pedestrian detection application

```c
#include <iostream>
#include <iomanip>
#include <stdexcept>
#include <opencv2/objdetect/objdetect.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/imgcodecs.hpp>
#include "t_includes/terasic_includes.h"
#include "fbdraw.h"
#include "hps_0.h"
#include <stdio.h>
using namespace std;
using namespace cv;

int fd;
dev_fb fb;
void *lw_axi_master = NULL;
void *h2f_axi_master = NULL;

volatile int* lw_fpga_led_addr;
volatile int* lw_fpga_key_addr;
volatile int* lw_fpga_mipi_pwdn_n;
volatile int* lw_fpga_mipi_rest_n;
volatile int* lw_fpga_mix_addr;
volatile int* lw_fpga_clp_addr;
volatile int* lw_fpga_frame_reader;
volatile int* lw_fpga_dma_control;
//volatile int* h2p_lw_dma_alpha_control;
volatile int* onchip_ram;
uint64_t pixel_data;

int a,i,j;

long on_chip_row_step,row, read_step;
int step = 8; //64/8

int loop_count;
int led_direction;
int black_level_0=0, black_level_1=0;
int led_mask;

//Image resolution
#define RES_X 320
#define RES_Y 240

int64_t t0, ttdetect, ttread, ttconvert0, ttconvert1;
int64 ttdetect0, ttdetect1, ttshow0, ttshow1;
int64 ttdetect00, ttdetect11;

uint8_t start_mix = 0x01;
uint8_t stop_mix = 0x00;
```
Design and Implementation of a Real-Time Pedestrian Detection System on SoC FPGA

```c
#define HW_REGS_SPAN (ALT_LWFPGASLVS_UB_ADDR - ALT_LWFPGASLVS_LB_ADDR + 1)
#define H2F_AXI_SPAN (ALT_FPGASLVS_UB_ADDR - ALT_FPGASLVS_LB_ADDR + 1)
#define IORD(base, index) (*((uint32_t *)base)+index))
#define IOWR(base, index, data) (*((uint32_t *)base)+index) = data)

/** Function Headers */
void detectAndDisplay(Mat frame, dev_fb fb);
void displayImage(dev_fb fb, Mat image, int offset_x, int offset_y);
void displayLogo(String img, dev_fb fb, int offset_x, int offset_y);
void displayMixerLayer(int layer, int offset_x, int offset_y);

Size win_size(48, 96);
Size block_size(16, 16);
Size block_stride(8, 8);
Size cell_size(8, 8);
int nbins = 9;

double hit_threshold = 1;
Size win_stride(4, 4);
Size padding(8, 8);
double scale = 1.5;
const int gr_threshold = 2;
cv::HOGDescriptor hog(win_size, block_size, block_stride, cell_size, nbins);
uint32_t led_data = 0x0;

/** @function main */
int main( int argc, const char** argv)
{
    fb_init(&fb);
    fb_fillScr(&fb,0,0,0);
    displayLogo("title.bmp", fb, 0, 0);
    displayLogo("upatras.bmp", fb, 106, 450);

    hog.setSVMDetector(hog.getDaimlerPeopleDetector());

    if( ( fd = open("/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 ) {
        printf( "ERROR: could not open "/dev/mem"...
"
        )
        return( 1 );
    }

    // map the address space for the Hardware Components registers
    // into user space so
    // we can interact with them.
    lw_axi_master = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ),
    MAP_SHARED, fd, ALT_LWFPGASLVS_OFST );

    if( lw_axi_master == MAP_FAILED ) {
        printf( "ERROR: Lightweight H2F AXI Bridge mmap() failed...
"
        );
        close( fd );
        return( 1 );
    }

    h2f_axi_master = mmap( NULL, H2F_AXI_SPAN, ( PROT_READ | PROT_WRITE ),
    MAP_SHARED, fd, ALT_FPGASLVS_OFST );
```
if( h2f_axi_master == MAP_FAILED ) {
    printf("ERROR: H2F AXI Bridge mmap() failed...\n");
    close( fd );
    return( 1 );
}

lw_fpga_led_addr = (int*)(lw_axi_master + LED_BASE);
lw_fpga_mipi_pwdn_n = (int*)(lw_axi_master + MIPI_PWDN_N_BASE);
lw_fpga_mipi_rest_n = (int*)(lw_axi_master + MIPI_RESET_N_BASE);
h2p_lw_mipi_camera = (int*)(lw_axi_master + I2C_OPENCORES_CAMERA_BASE);
h2p_lw_mipi_contrlr = (int*)(lw_axi_master + I2C_OPENCORES_MIPI_BASE);
lw_fpga_mix_addr = (int*)(lw_axi_master + ALT_VIP_MIX_0_BASE);
lw_fpga_clp_addr = (int*)(lw_axi_master + ALT_VIP_CL_CLP_1_BASE);
lw_fpga_dma_control = (int*)(lw_axi_master + VIDEO_DMA_CONTROLLER_0_BASE);
lw_fpga_key_addr = (int*)(lw_axi_master + KEY_BASE);
onchip_ram = (int*)(h2f_axi_master + ONCHIP_MEMORY_BASE);

IOWR(lw_fpga_led_addr, int(0), int(0x3ff));

//Disable Clipper Component. Actually stops the streaming
IOWR(lw_fpga_clp_addr, int(0), int(0x0));
usleep(20);

//Enable the Alpha Blending Mixer Component
IOWR(lw_fpga_mix_addr, 0, 0x01);
usleep(20);

//Din1 x,y, enable
displayMixerLayer(1, 555, 200);

IOWR(lw_fpga_mipi_pwdn_n, 0x00, 0x00);
IOWR(lw_fpga_mipi_rest_n, 0x00, 0x00);
usleep(200);
IOWR(lw_fpga_mipi_pwdn_n, 0x00, 0xFF);
usleep(200);
IOWR(lw_fpga_mipi_rest_n, 0x00, 0xFF);
usleep(200);

// MIPI Init
if (!MIPI_Init()){
    printf("MIPI_Init Init failed!\r\n");
} else{
    printf("MIPI_Init successfully!\r\n");
    MIPI_BIN_LEVEL(3);
}

//Enable Clipper Component.
IOWR(lw_fpga_clp_addr, int(0), int(0x01));
usleep(10);

alt_u16 focus_value=0; //0-1023

while(focus_value!=1050){
    cout << "\n* for camera focus value [0~1023] [1050 for skipping]: ";
    cin >> focus_value;
    if((focus_value>0) && (focus_value<1024))
        OV8865_FOCUS_Move_to(focus_value);
    }
usleep(20);
//---------Set the Mat Array for the frame to detect--------/
uchar image_array[RES_X*RES_Y];
Mat img = Mat(RES_Y, RES_X, CV_8UC1, &image_array);

while(IORD(lw_fpga_key_addr,0) == 3){
    //for (a=0; a<iterations; a++){
        if (led_data == 0){
            led_data = 0x200;
        } else{
            led_data = (led_data >> 1);
        }
        IOWR(lw_fpga_led_addr, int(0), int(led_data));
    // calculate timings
    t0 = cv::getTickCount();
    for (j=0; j<RES_Y; j++){
        on_chip_row_step=j*(RES_X/step);
        row = j*RES_X;
        for (i=0; i<RES_X; i+=step){
            read_step = i/step;
            pixel_data = (*((((uint64_t *)onchip_ram)+read_step+on_chip_row_step));
            image_array[i+row]=uchar(pixel_data & 0xff);
            image_array[i+row+1]=uchar((pixel_data>>8) & 0xff);
            image_array[i+row+2]=uchar((pixel_data>>16) & 0xff);
            image_array[i+row+3]=uchar((pixel_data>>24) & 0xff);
            image_array[i+row+4]=uchar((pixel_data>>32) & 0xff);
            image_array[i+row+5]=uchar((pixel_data>>40) & 0xff);
            image_array[i+row+6]=uchar((pixel_data>>48) & 0xff);
            image_array[i+row+7]=uchar((pixel_data>>56) & 0xff);
        }
    }
    ttread = cv::getTickCount();
    detectAndDisplay(img, fb);
    ttdetect = cv::getTickCount();
}

double det_secs = (ttdetect-t0)/cv::getTickFrequency();
double read_secs = (ttread - t0)/cv::getTickFrequency();
double conv_secs = (ttconvert1 - ttconvert0)/cv::getTickFrequency();
double multiscale_sec_d = (ttdetect11 - ttdetect00)/cv::getTickFrequency();
double show_sec = (ttshow1 - ttshow0)/cv::getTickFrequency();

cout << endl;"Pedestrian Detection performed in: " << det_secs*1000 << " Milliseconds";
cout << endl;"Reading frame from on-chip RAM performed in: " << read_secs*1000 << " Milliseconds";
cout << endl;"Convert Gray2RGB performed in: " << conv_secs*1000 << " Milliseconds";
cout << endl;"Detect Multiscale performed in: " << multiscale_sec_d*1000 << " Milliseconds";
cout << endl;"Show image process performed in: " << show_sec*1000 << " Milliseconds";
if (munmap (lw_axi_master, HW_REGSSPAN) != 0) {
    printf ("ERROR: munmap() failed...\n");
    return (-1);
}
if (munmap (h2f_axi_master, H2F_AXISPAN) != 0) {
    printf ("ERROR: munmap() failed...\n");
    return (-1);
}
close(fd);
fb_close(&fb);
return 0;
}

void displayImage(dev_fb fb, Mat image, int offset_x, int offset_y){
    int channels = image.channels();
    int nRows = image.rows;
    int nCols = image.cols * channels;
    int i,j,x,y;
    uchar r,g,b;
    for( i = 0; i < nRows; i++) {
        for ( j = 0; j < nCols; j+= channels) {
            y = i;
            x = j/channels;
            r = image.at<char>(i,j);
            g = image.at<char>(i,j+1);
            b = image.at<char>(i,j+2);
            //if we want offset of the image x + y+
            fb_drawPixel(&fb, x + offset_x, y + offset_y, b, g, r);
        }
    }
}

void detectAndDisplay( Mat frame_gray, dev_fb fb )
{
    std::vector<Rect> rects;
    Mat frame;
    ttconvert0 = cv::getTickCount();
cvtColor(frame_gray,frame,COLOR_GRAY2BGR);
ttconvert1 = cv::getTickCount();
    ttdetect00 = cv::getTickCount();
hog.detectMultiScale(frame_gray, rects, hit_threshold, win_stride, padding,
scale, gr_threshold);
    for (size_t i = 0; i < rects.size(); i++){
        rectangle(frame, rects[i], CV_RGB(0, 255, 0), 1);
    }
    ttdetect11 = cv::getTickCount();
    tttshow0 = cv::getTickCount();
    displayImage(fb, frame, 150, 200);
    tttshow1 = cv::getTickCount();
}
void displayLogo(String img, dev_fb fb, int offset_x, int offset_y ){
  
  Mat image;
  image = imread(img, CV_LOAD_IMAGE_COLOR);  // Read the file
  
  if(! image.data )  // Check for invalid input
  {
    cout << "Could not open or find the image" << std::endl;
  }
  displayImage(fb, image, offset_x, offset_y);
}

void displayMixerLayer(int layer, int offset_x, int offset_y){
  if (layer < 1 || layer >12) layer = 1;
  if (offset_x < 0 || offset_x >1024-321) offset_x = 0;
  if (offset_y < 0 || offset_y >768-241) offset_y = 0;
  int x_register = (layer-1)*3 + 2;
  int y_register = (layer-1)*3 + 3;
  int display_register = (layer-1)*3 + 4;
  //Set x,y for the displayed Layer 1 of Mixer
  IOWR(lw_fpga_mix_addr,x_register,offset_x);  //555 0x022B
  usleep(20);
  IOWR(lw_fpga_mix_addr,y_register,offset_y);  //200 0x96
  usleep(20);
  //Display Layer 1 of Mixer
  IOWR(lw_fpga_mix_addr,display_register,0x01);
  usleep(20);
}

hps_0.h file of pedestrian detection application

#ifndef _ALTERA_HPS_0_H_
#define _ALTERA_HPS_0_H_

/*
 * This file contains macros for module 'hps_0' and devices
 * connected to the following masters:
 *  * h2f_axi_master
 *  * h2f_lw_axi_master
 *  *
 * Do not include this header file and another header file created for a
 * different module or master group at the same time.
 * Doing so may result in duplicate macro names.
 * Instead, use the system header file which has macros with unique names.
 */

/*
 * Macros for device 'onchip_memory', class 'altera_avalon_onchip_memory2'
 * The macros are prefixed with 'ONCHIP_MEMORY_'.
 * The prefix is the slave descriptor.
 */
#define ONCHIP_MEMORY_COMPONENT_TYPE altera_avalon_onchip_memory2
#define ONCHIP_MEMORY_COMPONENT_NAME onchip_memory
#define ONCHIP_MEMORY_BASE 0x0
#define ONCHIP_MEMORY_SPAN 262144
#define ONCHIP_MEMORY_END 0x3ffff
/* Macros for device 'i2c_opencores_camera', class 'i2c_opencores' */
#define I2C_OPENCORES_CAMERA_COMPONENT_TYPE i2c_opencores
#define I2C_OPENCORES_CAMERA_COMPONENT_NAME i2c_opencores_camera
#define I2C_OPENCORES_CAMERA_BASE 0x0
#define I2C_OPENCORES_CAMERA_SPAN 128
#define I2C_OPENCORES_CAMERA_END 0x7f

/* Macros for device 'i2c_opencores_mipi', class 'i2c_opencores' */
#define I2C_OPENCORES_MIPI_COMPONENT_TYPE i2c_opencores
#define I2C_OPENCORES_MIPI_COMPONENT_NAME i2c_opencores_mipi
#define I2C_OPENCORES_MIPI_BASE 0x20
#define I2C_OPENCORES_MIPI_SPAN 128
#define I2C_OPENCORES_MIPI_END 0x9f

/* Macros for device 'mipi_pwdn_n', class 'altera_avalon_pio' */
#define MIPI_PWDN_N_COMPONENT_TYPE altera_avalon_pio
#define MIPI_PWDN_N_COMPONENT_NAME mipi_pwdn_n
#define MIPI_PWDN_N_BASE 0x40
#define MIPI_PWDN_N_SPAN 64
#define MIPI_PWDN_N_END 0x7f
```c
#define MIPI_PWDN_N_EDGE_TYPE NONE
#define MIPI_PWDN_N_FREQ 50000000
#define MIPI_PWDN_N_HAS_IN 0
#define MIPI_PWDN_N_HAS_OUT 1
#define MIPI_PWDN_N_HAS_TRI 0
#define MIPI_PWDN_N_IRQ_TYPE NONE
#define MIPI_PWDN_N_RESET_VALUE 0

/*
 * Macros for device 'mipi_reset_n', class 'altera_avalon_pio'
 * The macros are prefixed with 'MIPI_RESET_N_'.
 * The prefix is the slave descriptor.
 */
#define MIPI_RESET_N_COMPONENT_TYPE altera_avalon_pio
#define MIPI_RESET_N_COMPONENT_NAME mipi_reset_n
#define MIPI_RESET_N_BASE 0x50
#define MIPI_RESET_N_SPAN 64
#define MIPI_RESET_N_END 0x8f
#define MIPI_RESET_N_BIT_CLEARING_EDGE_REGISTER 0
#define MIPI_RESET_N_BIT_MODIFYING_OUTPUT_REGISTER 0
#define MIPI_RESET_N_CAPTURE 0
#define MIPI_RESET_N_DATA_WIDTH 1
#define MIPI_RESET_N_DO_TEST_BENCH_WIRING 0
#define MIPI_RESET_N_DRIVEN_SIM_VALUE 0
#define MIPI_RESET_N_EDGE_TYPE NONE
#define MIPI_RESET_N_FREQ 50000000
#define MIPI_RESET_N_HAS_IN 0
#define MIPI_RESET_N_HAS_OUT 1
#define MIPI_RESET_N_HAS_TRI 0
#define MIPI_RESET_N_IRQ_TYPE NONE
#define MIPI_RESET_N_RESET_VALUE 0

/*
 * Macros for device 'alt_vip_cl_clp_1', class 'alt_vip_cl_clp'
 * The macros are prefixed with 'ALT_VIP_CL_CLP_1_'.
 * The prefix is the slave descriptor.
 */
#define ALT_VIP_CL_CLP_1_COMPONENT_TYPE alt_vip_cl_clp
#define ALT_VIP_CL_CLP_1_COMPONENT_NAME alt_vip_cl_clp_1
#define ALT_VIP_CL_CLP_1_BASE 0x60
#define ALT_VIP_CL_CLP_1_SPAN 32
#define ALT_VIP_CL_CLP_1_END 0x7f

/*
 * Macros for device 'video_dma_controller_0', class
 * 'altera_up_avalon_video_dma_controller'
 * The macros are prefixed with 'VIDEO_DMA_CONTROLLER_0_'.
 * The prefix is the slave descriptor.
 */
#define VIDEO_DMA_CONTROLLER_0_COMPONENT_TYPE altera_up_avalon_video_dma_controller
#define VIDEO_DMA_CONTROLLER_0_COMPONENT_NAME video_dma_controller_0
#define VIDEO_DMA_CONTROLLER_0_BASE 0x80
#define VIDEO_DMA_CONTROLLER_0_SPAN 16
#define VIDEO_DMA_CONTROLLER_0_END 0x8f

/*
 * Macros for device 'video_dma_controller_1', class
 * 'altera_up_avalon_video_dma_controller'
 * The macros are prefixed with 'VIDEO_DMA_CONTROLLER_1_'.
 * The prefix is the slave descriptor.
 */
```
#define VIDEO_DMA_CONTROLLER_1_COMPONENT_TYPE
altera_up_avalon_video_dma_controller
#define VIDEO_DMA_CONTROLLER_1_COMPONENT_NAME video_dma_controller_1
#define VIDEO_DMA_CONTROLLER_1_BASE 0x90
#define VIDEO_DMA_CONTROLLER_1_SPAN 16
#define VIDEO_DMA_CONTROLLER_1_END 0x9f

/*
* Macros for device 'alt_vip_vfr_vga', class 'alt_vip_vfr'
* The macros are prefixed with 'ALT_VIP_VFR_VGA_'.
* The prefix is the slave descriptor.
*/
#define ALT_VIP_VFR_VGA_COMPONENT_TYPE alt_vip_vfr
#define ALT_VIP_VFR_VGA_COMPONENT_NAME alt_vip_vfr_vga
#define ALT_VIP_VFR_VGA_BASE 0x100
#define ALT_VIP_VFR_VGA_SPAN 512
#define ALT_VIP_VFR_VGA_END 0x2ff

/*
* Macros for device 'alt_vip_mix_0', class 'alt_vip_mix'
* The macros are prefixed with 'ALT_VIP_MIX_0_'.
* The prefix is the slave descriptor.
*/
#define ALT_VIP_MIX_0_COMPONENT_TYPE alt_vip_mix
#define ALT_VIP_MIX_0_COMPONENT_NAME alt_vip_mix_0
#define ALT_VIP_MIX_0_BASE 0x200
#define ALT_VIP_MIX_0_SPAN 1024
#define ALT_VIP_MIX_0_END 0x5ff

/*
* Macros for device 'sysid_qsys', class 'altera_avalon_sysid_qsys'
* The macros are prefixed with 'SYSID_QSYS_'.
* The prefix is the slave descriptor.
*/
#define SYSID_QSYS_COMPONENT_TYPE altera_avalon_sysid_qsys
#define SYSID_QSYS_COMPONENT_NAME sysid_qsys
#define SYSID_QSYS_BASE 0x10000
#define SYSID_QSYS_SPAN 8
#define SYSID_QSYS_END 0x10007
#define SYSID_QSYS_ID 2899645186
#define SYSID_QSYS_TIMESTAMP 1528723323

/*
* Macros for device 'led', class 'altera_avalon_pio'
* The macros are prefixed with 'LED_'.
* The prefix is the slave descriptor.
*/
#define LED_COMPONENT_TYPE altera_avalon_pio
#define LED_COMPONENT_NAME led
#define LED_BASE 0x10040
#define LED_SPAN 64
#define LED_END 0x1007f
#define LED_BIT_CLEARING_EDGE_REGISTER 0
#define LED_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LED_CAPTURE 0
#define LED_DATA_WIDTH 10
#define LED_DO_TEST_BENCH_WIRING 0
#define LED_DRIVEN_SIM_VALUE 0
#define LED_EDGE_TYPE NONE
#define LED_FREQ 50000000
#define LED_HAS_IN 0
#define LED_HAS_OUT 1
#define LED_HAS_TRI 0
#define LED_IRQ_TYPE NONE
#define LED_RESET_VALUE 0

/*
 * Macros for device 'sw', class 'altera_avalon_pio'
 * The macros are prefixed with 'SW_'.
 * The prefix is the slave descriptor.
 */
#define SW_COMPONENT_TYPE altera_avalon_pio
#define SW_COMPONENT_NAME sw
#define SW_BASE 0x10080
#define SW_SPAN 64
#define SW_END 0x100bf
#define SW_IRQ 2
#define SW_BIT_CLEARING_EDGE_REGISTER 1
#define SW_BIT_MODIFYING_OUTPUT_REGISTER 0
#define SW_CAPTURE 1
#define SW_DATA_WIDTH 10
#define SW_DO_TEST_BENCH_WIRING 0
#define SW_DRIVEN_SIM_VALUE 0
#define SW_EDGE_TYPE ANY
#define SW_FREQ 50000000
#define SW_HAS_IN 1
#define SW_HAS_OUT 0
#define SW_HAS_TRI 0
#define SW_IRQ_TYPE EDGE
#define SW_RESET_VALUE 0

/*
 * Macros for device 'key', class 'altera_avalon_pio'
 * The macros are prefixed with 'KEY_'.
 * The prefix is the slave descriptor.
 */
#define KEY_COMPONENT_TYPE altera_avalon_pio
#define KEY_COMPONENT_NAME key
#define KEY_BASE 0x100c0
#define KEY_SPAN 64
#define KEY_END 0x100ff
#define KEY_IRQ 1
#define KEY_BIT_CLEARING_EDGE_REGISTER 1
#define KEY_BIT_MODIFYING_OUTPUT_REGISTER 0
#define KEY_CAPTURE 1
#define KEY_DATA_WIDTH 4
#define KEY_DO_TEST_BENCH_WIRING 0
#define KEY_DRIVEN_SIM_VALUE 0
#define KEY_EDGE_TYPE FALLING
#define KEY_FREQ 50000000
#define KEY_HAS_IN 1
#define KEY_HAS_OUT 0
#define KEY_HAS_TRI 0
#define KEY_IRQ_TYPE EDGE
#define KEY_RESET_VALUE 0

/*
 * Macros for device 'jtag_uart', class 'altera_avalon_jtag_uart'
 * The macros are prefixed with 'JTAG_UART_'.
 * The prefix is the slave descriptor.
 */
```c
#include 

/* Macros for device 'sdram', class 'altera_avalon_new_sdram_controller'
 * The macros are prefixed with 'SDRAM_'.
 * The prefix is the slave descriptor.
 */
#define SDRAM_COMPONENT_TYPE altera_avalon_new_sdram_controller
#define SDRAM_COMPONENT_NAME sdram
#define SDRAM_BASE 0x4000000
#define SDRAM_SPAN 67108864
#define SDRAM_END 0x7ffffff
#define SDRAM_CAS_LATENCY 3
#define SDRAM_CONTENTS_INFO
#define SDRAM_INIT_NOP_DELAY 0.0
#define SDRAM_INIT_REFRESH_COMMANDS 2
#define SDRAM_IS_INITIALIZED 1
#define SDRAM_POWERUP_DELAY 100.0
#define SDRAM_REFRESH_PERIOD 15.625
#define SDRAM_REGISTER_DATA_IN 1
#define SDRAM_SDADV_ADDR_WIDTH 25
#define SDRAM_SDADV_BANK_WIDTH 2
#define SDRAM_SDADV_COL_WIDTH 10
#define SDRAM_SDADV_DATA_WIDTH 16
#define SDRAM_SDADV_NUM_BANKS 4
#define SDRAM_SDADV_NUM_CHIPSELETS 1
#define SDRAM_SDADV_ROW_WIDTH 13
#define SDRAM_SHARED_DATA 0
#define SDRAM_SIM_MODEL_BASE 0
#define SDRAM_STARVATION_INDICATOR 0
#define SDRAM_TRISTATE_BRIDGE_SLAVE ""
#define SDRAM_T_AC 5.5
#define SDRAM_T_MRD 3
#define SDRAM_T_RCD 20.0
#define SDRAM_T_RFC 70.0
#define SDRAM_T_RP 20.0
#define SDRAM_T_WR 14.0
#define SDRAM_MEMORY_INFO_DAT_SYM_INSTALL_DIR SIM_DIR
#define SDRAM_MEMORY_INFO_GENERATE_DAT_SYM 1
#define SDRAM_MEMORY_INFO_MEM_INIT_DATA_WIDTH 16
#endif /* _ALTERA_HPS_0_H_ */
```

Include libraries file of pedestrian detection application

```c
#ifndef TERASIC_INCLUDES_H_
#define TERASIC_INCLUDES_H_

void *h2p_lw_mipi_camera;
void *h2p_lw_mipi_ctrlr;
```

The ARM DS-5 application project can be found on my Github page:  
https://github.com/PhilipChalkiopoulos/PedestrianDetection