FIELD-PROGRAMMABLE GATE ARRAY DESIGN OF A SCALABLE PARALLEL-ACCESSIBLE MEMORY SUBSYSTEM FOR IMAGE FEATURE EXTRACTION

by

Jhonattan Castillo

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Prof. Paul Israelsen Major Professor	Dr. Jacob Gunther Committee Member

Dr. Edmund A. Spencer
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2011

Abstract

Field-Programmable Gate Array Design of a Scalable Parallel-Accessible Memory Subsystem for Image Feature Extraction

by

Jhonattan Castillo, Master of Science

Utah State University, 2011

Major Professor: Prof. Paul Israelsen
Department: Electrical and Computer Engineering

The purpose of this project is the creation of an advanced and efficient memory subsystem architecture that will improve the Haar processing frame rate without compromising the use of Field-Programmable Gate Array resources. Only a few hardware implementations and embedded system implementations have been published. The high amount of computing power required to analyze a single image frame is yet to be provided by a small, low-power embedded system. The main idea is to build a new memory subsystem that provides all the feature points needed at any given time and in a single access cycle. When it is compared to current best implementation, this architecture would potentially increase the system's performance by four times in 80% - 85% of the test cases while maintainig a speedup of one in the remaining 15% - 20%. This new memory subsystem is going to be used in the Haar Window Buffer which is the one that supplies the data points to the Haar Stage Classifier. While this work does not implement some ideal modules, like the collision detection circuit, it proposes a possible design for their implementation in future systems.

(65 pages)

Realizing dreams... one step at a time.

Acknowledgments

I would like to thank Dr. Aravind Dasu for giving me an opportunity to work under his guidance. I would also like to thank my committee members, Prof. Paul Israelsen, Dr. Jacob Gunther, and Dr. Edmund Spencer for extending their support. I thank Josh Templin for his amazing ideas and for providing his help and knowledge in VHDL coding. I would like to thank my family who gave the love and necessary support during the making of this project. Special thanks go to my mother, Aidelys Veras, and my father, Hipolito Castillo. It would have never been possible without their encouragement.

I would like to thank Erica Mills and Michael Kimball who edited my writing. Last, but not least, I would like to thank my friends at Utah State University for their support and valuable suggestions. Special thanks go to my colleagues and friends Juan de la Cruz, Michael Calcano, Dulce Minaya, Abiezer Tejeda, Eduardo Monzon, Omar Rodriguez, and many others, for always being with me during the toughest times of my graduate studies.

Jhonattan Castillo

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter contains a brief introduction to the logic and design behind the new memory subsystem. It also includes supporting background of image processing algorithms, and a brief explanation of the Haar-like Feature Cascade image classifier. The new memory subsystem aims to reduce the amount of internal resources used in Field-Programmable Gate Array (FPGA) while maintaining, or even increasing the performance levels.

1.1 Background

One of the most popular object recognition algorithms that is currently used these days is the Haar-like Feature Cascade algorithm [1]. This algorithm is widely used in high performance computers like desktop or server-grade machines, but it has not found its way in the embedded computing world. This case is specially true for FPGA implementations which present the greatest lack of research documents.

When using this algorithm in a low-power, low-complexity embedded system, the frame rates achieved are usually low. Nevertheless, as presented later in this section, there are some hardware implementations of this algorithm that have achieved high frame rates with Video Graphics Array (VGA) images. These high performance figures were obtained by using a great amount of FPGA resources which makes them unrealizable when using current FPGA technology.

As seen in the implementations by Hiromoto et al. [2] and Cho et al. [3], one of the modules that utilizes most of the FPGA resources is the Haar Window Buffer (HWB). One of the key factors contributing to the utilization of a high amount of resources was the use of a naïve memory subsystem. In addition, the low performance achieved by this module, creates a bottleneck in the data flow that reduces the entire system's performance

significantly.

This project will focus on the creation of an advanced and efficient memory subsystem architecture that will improve the frame rate without compromising the use of FPGA resources.

The best published implementation of the Haar Feature Cascade algorithm in an FPGA system was published by Cho et al. [3]. Previous implementations, like the one by Chareonsak [4], could barely process 120x120 images. Other implementations, such as the one by Lai et al. [5], used a personalized, trimmed down version of the Haar cascade to achieve higher performance at the expense of result quality. Lai et al. used a single stage classifier containing 52 features in total. Tang et al. [6] also used a trimmed down version of the cascade in order to obtain higher frames per second (fps) in his cellular phone implementation. He only used 70% of the original Haar cascade. The design project presented in this work does not intend to obtain performance improvements by reducing the quality of the results.

Other implementations, like the one by Hefenbrock et al. [7], used powerful Graphic Processing Units (GPU) to run the Haar cascades. It was proven, that even when using the most powerful NVIDIA Tesla C2050 GPUs and NVIDIA CUDA Platform [8], the performance figures were lower than when using an FPGA-based system. In fact, a total of four GPUs were needed in order to produce comparable speeds to Cho et al. FPGA implementation. Hefenbrock et al. design presents two other major disadvantages to our application. The first being cost. A Tesla GPU-based implementation is more expensive than an FPGA-based. In fact, a Tesla-based implementation can be as much as four times more expensive than an FPGA-based. A Tesla C2050 development board can cost up to 2,500 dollars [9] compared to the Xilinx Virtex 6 LX240T board which costs 2,195 dollars (Note that in order to achieve comparable performance to the FPGA board, four GPU boards are needed).

The other disadvantage of the GPU implementation is power efficiency. Just a single Tesla C2050 consumes 238 Watts [9]. That is more than 50 times the power consumed by the FPGA counterpart. For the four boards, a total of 952 Watts would be consumed

by the GPUs only. It is virtually impossible to deploy a system in which a single sensor consumes that much power and costs that much money. For our application, the GPU-based architecture is unrealizable.

Cho et al. [3] used a Xilinx Virtex 5 FPGA to synthesize their architecture design which took almost the entire available area of the chip. Just for the HWB and related circuitry, this implementation used around 13,000 slice registers and more than 30,000 Lookup Tables (LUT) plus ten Block Random Access Memories (BRAMS). In terms of the Xilinx Virtex 5 LX110, which was used in their implementation, up to 75% of it's resources were consumed only by the HWB and related circuitry.

In addition, Cho et al. memory subsystem could only provide a single point for each image line per clock cycle because it used an individual BRAM per image line. Is important to note that several thousands of feature points are needed per every image frame. The fact that their window buffer could only provide one feature at a time, made the module become a bottleneck of the system flow. The importance of this module and the effects of such bottlenecks are explained in detail in Chapter 4.

It can be concluded that Cho et al. approach, while being an excellent research contribution, results inapplicable for real world cases. The low efficiency in the usage of FPGA resources makes the implementation of the system less appealing. In this work, this implementation is also referred to as the *baseline* implementation.

In our application, it is required to read specific image points marked by a set of special rectangles called *features*. A sample of a feature set is shown in Fig. 1.1. Note that the feature size and its location within the image buffer can vary from frame to frame. Therefore, finding a storage pattern that satisfies all feature point configurations at the same time is not an easy, or even possible, task.

There are other implementations of parallel memory subsystems, like the ones proposed by Chandrakar et al. [10], Kuzmanov et al. [11] and Vanne et al. [12], that could provide the throughput required for our system. However, these parallel architectures are specialized for providing patches of pixels at a time. Hence, if one of these implementations was used

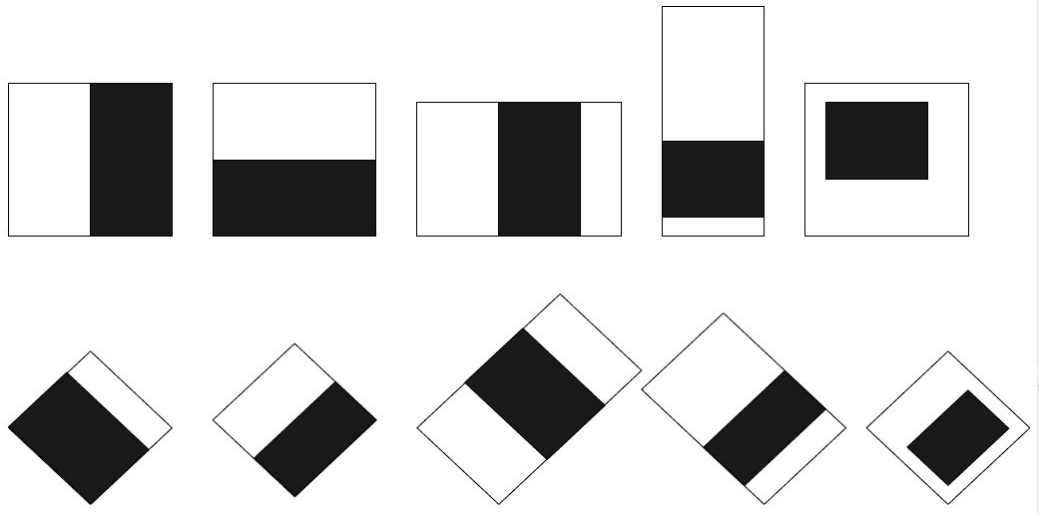

Fig. 1.1: Different feature sets.

as is, it would be required to burst an entire patch or section of pixels in order to select the corner pixels only. This would create a waste of power and bandwidth of the system.

Other architectures, like the one proposed by Peng et al. [13], utilized some skewed techniques to provide data in parallel. However, due to the physical limitations of the BRAMS, this skewed technique is not able to provide nine feature points at the same time. The reason being that the feature shown in Fig. 2.1 requires points from three different rows of the image at the same time. If Peng et al. approach was used, the BRAMS would have to be triple ported in order to satisfy the request in a single clock cycle.

Even though there are some other memory subsystem implementations that could possibly be used for our application, none of these offer the performance, simplicity, and adaptability that is desired.

1.2 Problem Definition

So far, most of the work that has been done for Haar-like Features has been done in high performance computer systems. Only ten years have passed since Viola and Jones [14] published their algorithm. Therefore, the majority of other researchers are still focused on improving the quality of the results [15], improving the performance [5], or improving the applicability of the algorithm [16, 17].

Unfortunately, not much of work has been done to make the algorithm more portable. Only a few hardware implementations and embedded system implementations have been published. One of the main reasons being that Haar-like Feature Cascade is a very resource-demanding algorithm. This high amount of computing power required to analyze a single image frame is yet to be provided by a small, low-power embedded system. The minimal system configuration to run this algorithm is shown in Fig. 1.2. As seen in the implementations by Hiromoto et al. [2] and Cho et al. [3], one of the modules that utilizes most of the FPGA resources is the HWB. Not only is it the one that uses most of the resources, it is also the one that creates the performance bottleneck in the system.

This bottleneck is created because current designs for this window buffer can only provide one point per image line at the same time. Therefore, in order to read the pixel values of some features, up to four or more clock cycles are needed. Given the fact that there are approximately 20,000 features per cascade, this simple bottleneck can create severe performance problems for the final system implementation. In addition, because of the high amount of logic used, this module makes the system highly inefficient. Unlike the system presented on this work, previous hardware implementations were not focused to mass produce a reliable, low-cost object detection system. The improvement of the memory subsystem's performance and the efficient utilization of resources are the primary goals of

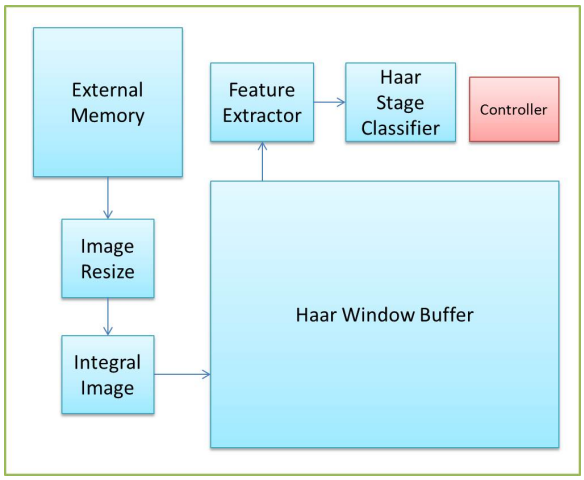

Fig. 1.2: Block diagram of the basic modules needed by the Haar algorithm.

the architecture design presented in this work.

1.3 Proposed Solution and Objectives

In this work, a newly designed memory subsystem is proposed. This new memory subsystem will utilize a special pattern to store image data in the FPGA BRAMS. The pattern will be repeated as many times as the designer wants, or until the space available in the BRAMS is depleted.

As presented in section 4.3.3, the selection of the right pattern is crucial for the successful implementation of this new memory subsystem. This access pattern represents the soul of the whole memory subsystem. The pattern has to be simple enough so it does not require a complicated address generation circuits, but also robust and efficient. A simple representation of a pattern patch is presented in Fig. 1.3. Note that each cell of the image represents a pixel, and each number represents the memory module where it should be stored.

Using this special pattern, it should be possible to obtain as many as twice the data points as the BRAMS used in the FPGA system. All of these points should be provided in a single clock cycle. By using this memory subsystem, and based on the feature complexity, it should be able to achieve a $4X$ speedup over the baseline approach. This is possible because of the use of true dual port BRAMS, as detailed in section 4.5.4.

The memory subsystem has to be widely configurable in order to accept different types

1	1	2	2	3	3
1	1	2	2	3	3
4	4	5	5	6	6
4	4	5	5	6	6
7	7	8	8	9	9
7	7	8	8	9	9

Fig. 1.3: Sample of the pattern patch to be used.

of features, each one having different access patterns. The system's configuration should not interfere with the latency of the system, so the throughput must remain constant regardless of the feature type.

The final goal of the project is to increase system's performance while maintaining or optimizing the power consumed and area used by the Haar cascade algorithm. Therefore, the final architecture for this new memory subsystem has to be as area and power efficient as possible in order to be implemented in a low-end, low-cost FPGA.

Since the feature information is provided in a coordinate system fashion, the memory subsystem will also include a *(X,Y)* address converter. This will alleviate the processing and complexity for the Haar controller. The system also becomes more generic because the controller's program becomes independent of the pattern or memory subsystem used. It will only provide the *(X,Y)* location of the pixel to be requested to the memory subsystem without having to deal with the addressing scheme or pattern itself.

1.4 Scope of the Project

This project will focus on the improvement and design of the HWB module. This module will manage the pixels sent and requested by an external controller. This project does not intend to design the interface with other modules such as integral image generator or external memory controllers. In addition, the design of the Haar controller is also out of the scope of this project. Due to the fact that this module constitutes a single part of the whole Haar object detection algorithm, and that no other module will be designed in this work, the tests will only be run and measured for this particular module.

This project will include the design and implementation of the new addressing scheme to be used by the memory subsystem. The design of the connection network of the BRAMS and all other internal submodules of the memory subsystem are also included. In Chapter 3, it is presented a person detection software written using the Opencv library [1]. This software will be used as a source of valuable statistical data about the performance and quality of the algorithm. Performance results and resource utilization are included in the last chapter of this work.

Chapter 2

Haar-like Features Cascade Algorithm

This chapter will introduce the Haar-like Feature Cascade algorithm. The Haar-like Features Cascade is one of the most used and most accurate object detection algorithms to date [1]. The name comes from the similarity of the features used to the Haar wavelets. Modern implementations of the Haar-like Feature algorithm can be used to detect any rigid object. Because of its popularity, the algorithm was embedded in the open source library Opencv [18].

2.1 The Algorithm: Concepts and Parts

Originally used only for face detection, the Haar-like Feature Cascade algorithm has evolved to the point that it can be trained to detect any kind of rigid object. Some examples of modern applications of this algorithm are: pedestrian detection [17], face feature detection [16], and it is even used as a development platform for object detection using SURF [19]. The algorithm was originally proposed by Viola and Jones [14] several years ago. It was then improved by Lienhart and Maydt [15] when they included the concept of rotated features. Lienhart and Maydt's version is the one used in popular image processing libraries like openCV [1].

The algorithm follows a feature based concept. That is, instead of analyzing the whole image frame in a pixel by pixel fashion, patches of pixels are extracted at the same time and analyzed. Simple operations are applied to these patches/image sections and, based on the results obtained, a decision is made. The decision, in this case, consists of whether the region being analyzed contains the object we are looking for or not. For our application, the object to be found in the image is a human upper body.

In order to improve the performance of the Haar object detection algorithm, Viola and

Jones modified their original approach and added support for the Adaboost algorithm [20]. The use of Adaboost allowed Viola and Jones to create a very robust object algorithm by using simple classifiers. In addition, the cascade is organized in such a way that it disqualifies negative matches in the early stages of the classification process. This effectively reduces the computation time needed to make a decision.

Another improvement made to the algorithm was the inclusion of integral images. Thanks to the rectangular nature of the Haar features, the use of integral images increased the performance considerably. When using integral images, only the corner points of the features are needed rather than reading the full feature patch. Each one of these components is explained in the rest of this chapter.

2.2 Haar-like Features

The feature information was originally obtained from the Haar basis functions. This basis functions were used by Papageorgiou et al. [21] for object detection applications. Each feature consists in a group of rectangles arranged one next to the other and which weight can vary between -1 and +1. A sample of a feature is given in Fig. 2.1. Note that the black rectangles have weight equal to -1 and the white rectangles have weight equal to +1.

The idea is to use this set of organized rectangles to get image properties. Let us assume we select a region A of the image I that its been processed. In order to obtain the properties of the region R, we place a feature within region R in location (X,Y). Then the pixels located inside of the feature's rectangles are read out of the image.

Each feature value is computed by taking the sums of all the pixels contained in each of the feature's rectangles and multiplying them by each rectangle's weight. The total

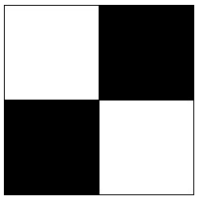

Fig. 2.1: Four rectangle - Nine point feature.

feature value is the sum of these weighted results. Note that the complexity of the feature extraction operation is subject to the size of the feature. For bigger features, more pixels are needed from memory than for smaller features.

2.3 Integral Image

In the original algorithm proposed by Viola and Jones [14], it was introduced the concept of integral images. In this new type of image, each pixel location *(X, Y)* contains the sum of the pixels above and to the left of it. The pixels of the integral image are computed using the following equation:

$$ii(x, y) = \sum_{x'<x, y'<y} i(x'y'), \tag{2.1}$$

where *ii(x,y)* is the integral image and *i(x,y)* is the original image.

Using this new type of image, the total number of pixels required to do a feature extraction is reduced considerably. In fact, thanks to the used of integral images, the only pixels required for the feature extraction are those located in the corners of the features. That is, to calculate the area of each rectangles we only use their four corner points in this fashion:

$$A_R =: L4 - L3 - L2 + L1, \tag{2.2}$$

where A_R is the area of the rectangle R shown in Fig. 2.2. The process to obtain the area of the rectangle using just the corner points is also explained in the same figure.

As explained in section 2.2, the complexity of the feature extraction varies based on the feature size. When using integral images, only four data points are required to compute each of the rectangle's areas regardless of their size. Therefore, for a 4-rectangle feature, like the one presented in Fig. 2.1, only nine points are needed. Thanks to this, the feature extraction is done in a lot less time and with simpler calculations.

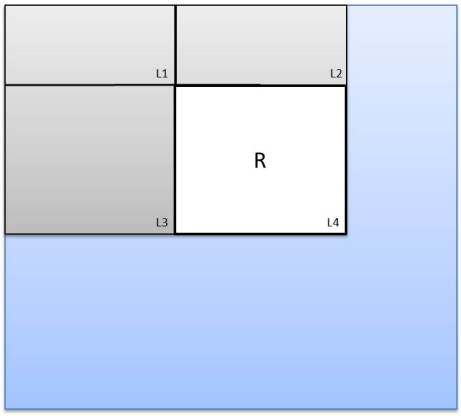

Fig. 2.2: Calculating the area of a rectangle R is done using the corner of the rectangle: L4-L3-L2+L1.

2.4 Haar Cascade Stages

The Haar-like Feature Cascade algorithm is a tree-based algorithm where every level is called a *stage*. The group of all the levels of the tree is called a *cascade*. Each one of the stages is formed by several simple *classifiers*. As introduced with the Adaboost algorithm, Viola and Jones used these simple classifiers to produce the higher complexity and more robust ones found in the upper levels of the cascade. This means, the cascade's complexity is augmented from one level to the next. Each classifier extracts several feature values of an image section and then performs a basic comparison between the feature's values obtained and the threshold assigned to the stage. If the values exceed a known threshold, the region in question is said to have passed the stage; otherwise it is said that it did not.

As it was observed, every stage of the cascade becomes more and more complex. An image section is needed to pass all levels (stages) of the cascade in order to be classified as a positive match. As soon as any classifier's result does not meet the stage threshold, the region of the image is considered to be a negative match. In this case, no further operation is evaluated for that section and it is proceeded to evaluate the next one. Note that the classifiers are organized in such a way that areas not likely to contain an object/face are discarded in early stages of the cascade which happen to be the simplest ones.

The flow chart of the whole algorithm is presented in Fig. 2.3.

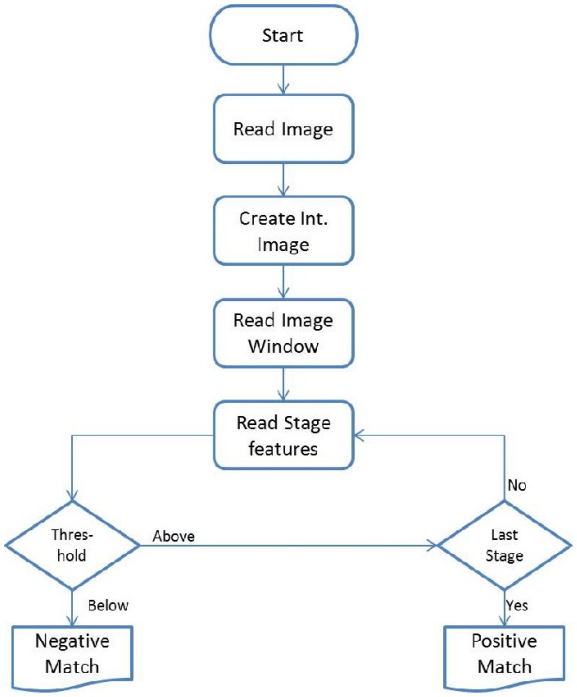

Fig. 2.3: Flow chart of the Haar-like Features Cascade algorithm.

In order to study the details of the Haar-like Feature Cascade algorithm, a C++ application was developed based on the OpenCV library. The application provided performance information, resource utilization, and useful statistical data of the algorithm in question. The details of the software implementation are presented in Chapter 3.

Chapter 3

Software Implementation of the Haar-Like Feature Algorithm

The first step taken for the system implementation consisted in the development of a computer application that served to study the details of the Haar-like Feature Cascade algorithm. It was developed in C++ based on the OpenCV library. The application provided performance information, resource utilization, and useful statistical data. The details of the application and the results obtained are presented in the rest of this chapter.

3.1 OpenCV Library: Haar Person Detection

OpenCV is an open source computer vision library written in C and C++ that runs under Linux, Windows, and Mac OS. Most of OpenCV's functions have been optimized for real-time applications and for high performance processing cores using the Intel Performance Primitives (IPP) Application Programming Interface (API) [1].

The primary goal of the library is to offer a solid API platform for computer vision developers. This platform helps to build complex, yet reliable, machine vision applications in little time. It contains more than five hundred functions that span across object detection applications, machine interfacing, security, and even hardware calibration.

One of the most popular functions included in the library is the one used for face detection. This function takes the name of Haar classifier because it is based on the boosted version of the Haar-like Feature Cascade algorithm originally proposed by Viola and Jones [14]. The cascades used to detect different parts of the human body, like frontal face, eyes, legs, upper body, among others, are already provided in the library. The supervised learning method, which allows designers to train the program to detect virtually any kind of rigid object, is also included in the library.

The OpenCV implementation of the Haar classifier includes several performance improvements such as the use of integral images, statistical boosting, and intelligent organization of the rejection cascade. Thanks to the extensive training data used to create each one of the cascades, the quality of the Haar classifier results is very high. In the case of face detection cascades, the true positive face detection rate is near 98% with a false positive detection rate of 0.0001% [1].

These prominent results can be obtained whenever the object we are trying to detect contains a high amount of internal blocky features. That is, the algorithm works well when there are distinguishable features inside the object's perimeter. For example, frontal faces are easily detectable thanks to inner objects like eyes, nose, and mouth. However, faces in profile view do not offer the same levels of detection since only the ear can be considered as a unique object feature. For profile views, most of the distinguishable features are located on the contour itself of the head, not within it. This can potentially reduce the quality of the results.

It is very important to know the type of features present in the object that it is wanted to be detected. Even when using a very robust algorithm, such as Haar feature detection, low-quality results will be obtained if not enough information is present in the image.

3.2 Haar-like Features Classifier Program

In order to test the Haar classifier algorithm, a C++ program was written based on the OpenCV computer vision library. The idea was to collect statistical data of the performance and reliability of the algorithm. In addition, the program served to analyze the memory access patterns used, and to brainstorm on a possible set of solutions for the bottlenecks created in Cho et al. implementation. The program was built for a 64-bit Linux machine.

In order to obtain the video frame information, a simple USB camera connected to the host computer was used. Thanks to OpenCV's well defined hardware interface, it was very easy to set up a test system. The program was modified later to accept video files as input as well.

The real-world tests gave an idea about the difficulty of the task being faced. Due to

changes in lighting, position of the subjects, object occlusion, and other scene transformations, it was not possible to obtain the same levels of accuracy that were advertised in the OpenCV book [1]. Therefore, a high amount of effort was put into the software implementation in order to improve its reliability. These improvements are explained in detail in section 3.3.

3.3 Improvements to the Haar-like Feature Algorithm

Even though the Haar cascade algorithm is classified as a real-time object detection algorithm, it requires a lot of processing power. In addition, the quality of the results obtained in the software implementation was not as good for real-life tests as it was for static test images. Therefore, a lot of work was put into the optimization of the algorithm. Some of the measures taken to improve the standard Haar algorithm are presented in the remainder of this section.

3.3.1 Motion History Information

Motion history was originally introduced by Jain and Nagel [22] in 1979. They introduced the idea as a combination of two concepts: first order difference picture (FODP) and second order difference picture (SODP). The concept proposed by Jain and Nagel, has been used in several other works and for many different purposes like pose recognition [23], recognition of human movement [24], and even facial movement detection [25], among several others.

The concept behind the difference picture is very simple: every pixel element of the resultant motion history image is incremented by one whenever the corresponding pixel from the second or subsequent frames is incompatible with the corresponding pixel from the reference frame. That is, if a pixel from the reference frame does not match with the pixel value of the following frame, the resulting motion history pixel value will be incremented by one.

Jain and Nagel [22] extended the concept of FODP when they introduced the concept of SODP. In this case, the motion frames obtained by FODP are then compared between

themselves. Today, this is what we know by *motion history*. It is a history image of how the motion has changed across several past image frames.

This very simple but essential function was implemented as an optimization for the naïve Haar algorithm that comes embedded in OpenCV. The calculation of the motion history frame only requires simple ADD operations. Therefore, the calculation of the motion history frame can be done easily even in small/low end embedded processors.

The use of motion history allowed to focus the program's processing power only in the areas that contained motion. This technique is known as Region of Interest (ROI) selective processing.

3.3.2 Region of Interest Selection

It is very important to note that, for the scope of this project, it was assumed that the motion present in the scene was produced solely by humans. Since the program goal was to use the Haar algorithm to detect humans only, the data produced was used to mark the areas of the frames where a human was supposed to be. These areas were then marked as ROI sections.

The areas marked as ROI are the only ones that are passed through the Haar algorithm processing pipeline. This helps to reduce computation time considerably (although linearly). Another benefit of the use of ROI is the improvement of the result's quality . The quality is improved because only the areas that are likely to contain a person are processed reducing the probability of a false result. Thanks to the reduction of the searchable area, the possibility of finding a false positive inside this area is reduced. Note that the human itself occupies an important section of the movement region.

3.3.3 Size and Position Variance Constraints

Other of the improvements implemented for the new version of the algorithm was the creation of a set of constraints that controlled the size and position change of the detected positive regions or *(*Haar boxes). From the software implementation, it was noted that most of the false negative detections appeared randomly in the image and only for one or two

frames. A constraint was set in place to ignore these random and instantaneous detections. Unlike the aforementioned functions, this function was implemented only to improve the quality of the results and not the performance.

In addition, the assumption that people cannot change their size or position instantly was made. By intuition, we know that if a subject walks towards the camera, it appears that he is becoming bigger and the opposite occurs if he is walking away from the camera. However, several frames of the video capture are required to record this action. That is, human movement is relatively slow when compared to the frame acquisition speed. When a person is detected in the image, the location and size information is locked for ten frames. That is, for the following ten frames this position and size can not change dramatically. In fact, the new position can not be more than 20 pixels away from original and its size can not be more than 20 squared pixels larger/smaller. Note that the time difference from a frame to another can be as low as 3.33 ms (at 30 fps).

In addition, women and men generally fall within a certain height range. According to the National Health Statistics Reports (NHSR) [26], the average height of adult individuals is about 5'5.4". Using the software, several tests with different individuals of different heights were performed. It was found that the height data provided by the NHSR could be translated into our field as the average number of pixels occupied by a person. The average number of pixels was found to be in the range from 4,900 (70x70) to 10,000 (100x100) in a 640*480 image. These numbers where obtained by measuring different individuals at different distances from the camera in different scenarios.

Having this range, from 70x70 to 100x100, allowed the addition another constraint to the person detection software. Any detection that was outside this range, was considered non-human and automatically discarded.

3.3.4 Aged Haar Boxes

This was the last improvement written for the software. The concept of aged Haar boxes is pretty simple: If there is motion in the scene and we also detect a person in that region using the Haar algorithm, then the detection gets the highest confidence value. The

highest value possible is 100.

The confidence value is maintained if the person is detected by Haar in future frames. However, if the Haar algorithm cannot find any person in this location even while there is movement, the confidence value is reduced. In addition, if the motion stops completely in that area of the image, the confidence value is reduced with an extra penalty.

If at any moment the confidence value reaches zero, then it is said that the person is no longer in the scene. If the person is detected by Haar again, the confidence level is reset to the maximum value. This ensures that any object having the shape of an upper body does not trigger the positive detection flag (or at least not for long). Examples of these objects are chairs, hanged coats, among others.

3.3.5 Improvements Summary

Thanks to these simple improvements, the quality of the results reached the advertised levels of 98%-99% of accuracy, and the false positives were reduced to near 1%. The summary of all the parameters that were used to get this improvement are shown in Table 3.1.

3.4 Summary of the Software Implementation

In order to obtain statistical data of the Haar-like Feature algorithm's performance and results quality, a C++ application based on the OpenCV library was written. The use of this software allowed the analysis of data produced in real-world tests. In addition, the basic Haar detection algorithm that comes embedded in OpenCV, was improved with the use of

Table 3.1: Software parameters.

Variable	Value
Min Person Size	70 pixels
Max Person Size	100 pixels
Max Distance Change	20 pixels
Max Size change	20% piexels
# Of History frames	13

other image processing algorithms and techniques. Some of the most relevant improvements were the use of region of interest, aged Haar boxes and motion history information.

Thanks to these optimizations, the quality of the results obtained was increased up to 98%-99% of accuracy. This software served as a guideline for the hardware implementation presented in Chapter 4.

Chapter 4

Hardware Implementation: Redesigning the Haar Window Buffer Module

Even though the Haar-like Feature Cascade algorithm was created to perform at real-time speeds, it was not intended to be run in a real-time system. The high amount of resources used and the high number of computations required to process a frame make this algorithm unappealing to embedded systems designers. As seen in the implementations by Hiromoto et al. [2] and Cho et al. [3], one of the components that represents the highest utilization of resources is the Haar Window Buffer (HWB). This chapter presents the design characteristics of the HWB as well as the design details of the new data access pattern created for it.

4.1 Haar-like Feature Cascade Hardware Implementation

The Haar-like Feature Cascade algorithm is well known and accepted among developers for its relative speed and accuracy [1]. However, the algorithm is not very popular in the embedded computing field. The lack of documented research in the area is evident. The main reason, up to this date, has been the hardware limitations encountered in low-power, embedded processors.

The most advanced and precise Haar cascades contain around 23 stages, each one having hundreds of classifiers. Each classifier is formed by a feature or set of features that has to be processed independently. Based on the classifier results, it is then decided if the image region in question passed the stage or not. The classification process is explained in detail in Chapter 2.

In order to process each feature, between four to nine points are read from an specially-constructed data structure called integral image. For the designer's convenience, this inte-

gral image can be located in any form of memory, from internal to external Double Data Rate (DDR) memory. However, in order to increase the access performance and to reduce the number of external memory requests, the integral image information is usually stored in the HWB, inside the FPGA.

This project focuses in the design of a new memory subsystem to be used in the HWB. The data supplied by this memory subsystem is then fed to the cascade stage classifier for further processing. As presented in section 1.1, other implementations like the one proposed by Cho et al. [3] were not very efficient. It was concluded that one of the most resource demanding modules of their design was the window buffer. In fact, the HWB and related circuits utilized around 75% of the available resources of a Xilinx Virtex 5 LX110 FPGA.

Due to the fact that their HWB implementation uses a BRAM per image line, it can only provide a single pixel per each image line at every clock cycle. Therefore, their memory subsystem is very inefficient. It uses massive amounts of resources but still the throughput is kept very low. Since several thousands of feature points are needed per every image, the window buffer becomes a major bottleneck in the system. It is clear that it is necessary to improve this module in order to synthesize the whole Haar algorithm in a lower end FPGA and get acceptable performance.

4.2 The New Memory Subsystem

This section will explain the details and the internal characteristics of the newly designed memory subsystem. The section will also include the definition of new concepts and the details of the module's implementation.

4.2.1 Conceived Idea

The main idea is to build a new memory subsystem that provides all the feature points needed at any given time and in a single access cycle. The use of this parallel architecture would potentially increase the entire system's performance. This new memory subsystem is going to be used in the HWB which is the one that supplies the data points to the

Haar Stage Classifier. The primary motivation to develop this new memory subsystem is to reduce the amount of resources used by the HWB.

The primary constraint that was set during the design of this module was to achieve higher performance than the standard method proposed by Cho et al. [3]. The second constraint was set to the efficient use of resources. While it is usually very difficult to achieve considerable higher speedups without sacrificing resources, the author believes that during the design of this new memory subsystem, it has achieved a very good balance. As it will be presented in the remainder of the chapter, all the modules where individually optimized to get the best performance per FPGA resource used.

4.2.2 Achieved Characteristics

In order to increase the system's throughput, the system was designed to be fully pipelined. Therefore, the memory subsystem can accept whether a load or store instruction at every clock cycle. In addition, the number of access ports is completely configurable. The number of ports can be configured from a single port to *2*N* ports, where N is the number of BRAMS in the system. Each access port can supply a feature point at every clock cycle. Thanks to this approach, the throughput of the system can be easily configured by the designer.

The complexity of the system has been maintained considerably low. In fact, the entire system only uses five ADD operations per access port. Considering the complexity of the addressing scheme, the author believes this is an excellent achievement. Other operations required, like multiplication and division, were all changed to their power-of-2 representations. Hence, the final implemented system was simplified considerably.

A new data access pattern was designed. When used in conjunction with the configurable multiplexer/de-multiplexer network presented in this work, it can provide the required feature points at any output port in a parallel fashion. The details of the new access pattern are presented in section 4.3.

4.3 QuadraPattern: A Simple But Intelligent Data Access Pattern

In order to access several data points from the same BRAM bank at the same time, a new memory access pattern was created. This access pattern constitutes the soul of the entire system. The pattern was named *QuadraPattern* after the fact that it is formed by placing a set of squares (blocks) next to each other. The number and the size of each one of the squares is completely configurable and it depends only on the available resources and the designer's goals.

Each one of these squares or blocks represent each independent BRAM. A simple graphical example of a QuadraPattern patch is shown in Fig. 4.1. Note that in this image, each cell represents a data point or pixel and each number represents the BRAM ID were such a pixel is stored.

4.3.1 Pattern Components: Definition and Description

In order to ease the understanding of the pattern properties and the addressing scheme shown in the following sections of this work, the author has also introduced a set of concepts that will be used to define each of the pattern parts/properties.

- *Block:* This is the simplest structure in the pattern. It directly relates to the number of BRAMS used. This relation will be explained in section 4.4. In the patch shown in Fig. 4.1, a block is formed by the set of cells that contain the same numbers, *i.e.* a

1	1	2	2	3	3
1	1	2	2	3	3
4	4	5	5	6	6
4	4	5	5	6	6
7	7	8	8	9	9
7	7	8	8	9	9

Fig. 4.1: Simple example of a QuadraPattern patch.

sample of a block would be the set of four cells containing a 1 that are located in the top left corner of the patch.

- *Set:* A set is defined as a group of *blocks*. The number of blocks used to conform a set is selected at convenience of the designer. In Fig. 4.1, a set is represented by the blocks numbered from one to nine. In this work, a set is also referred as a *patch*.

- *Bank:* Is the biggest component of the pattern. It is formed by a group of *sets* that are placed next to each other. If we formed a matrix of sets, a bank would be the group of sets that fall in the same horizontal line, *i.e.* a bank would be represented by each row of the matrix.

4.3.2 Why a Squared Pattern?

As stated in previous chapters of this work, the primary goal of the memory subsystem design was to increase the performance of previous implementations. One way to achieve such performance increase was to design the system as simple as possible. The regular shape of the square figure allows to place infinite number of blocks next to each other while still maintaining the addressing alignment properties. In addition, if the designer chooses a power-of-2 size for the blocks, the addressing scheme and the address translation operations are simplified considerably.

It is true that other patterns could allow the same parallel access capability. However, these other patterns require complex addressing schemes which are not affordable for this implementation. The use of a power-of-2 squared pattern reduces the resource use of the address generator significantly. A sample of a possible addressing pattern is presented in Fig. 4.2. Due to its irregularity, this pattern would require a complex addressing scheme. Therefore, in order to maintain low complexity levels for the address generation circuit, the QuadraPattern was selected on top of other possible patterns.

Another true benefit of a square pattern is its uniformity. In the case the designer wants to cover more image area (to build a larger buffer), the size of each square can be easily changed without losing any performance. This can be done thanks of the uniformity

1	3	5	7	1	3	5	7	1	3	5	7	1	3	5	7
2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
1	3	5	7	1	3	5	7	1	3	5	7	1	3	5	7
2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
1	3	5	7	1	3	5	7	1	3	5	7	1	3	5	7
2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
1	3	5	7	1	3	5	7	1	3	5	7	1	3	5	7
2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
1	3	5	7	1	3	5	7	1	3	5	7	1	3	5	7
2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
1	3	5	7	1	3	5	7	1	3	5	7	1	3	5	7
2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
1	3	5	7	1	3	5	7	1	3	5	7	1	3	5	7
2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8
1	3	5	7	1	3	5	7	1	3	5	7	1	3	5	7
2	4	6	8	2	4	6	8	2	4	6	8	2	4	6	8

Fig. 4.2: Simple example of another possible data access pattern.

of the shape of the square itself. The addressing scheme is completely independent of the size of the blocks. In the case of the pattern shown in Fig. 4.2, a simple way to increase increase the size of the buffer does not exist. A change in the patch size would suggest a change in the pattern itself, so a brand new addressing scheme would have to be written each time the size of the buffer is modified. This does not happen for the QuadraPattern, where additional blocks can be put in place without any hassle.

4.3.3 Importance of the Selection of the Correct Pattern Size

In order to utilize each BRAM efficiently, the size of the blocks must be carefully chosen. Based on the application the designer is working towards to, and based on the original image size, the selection of the right block size is indispensable. A bigger sized block would represent a higher utilization of each BRAM. Smaller blocks, however, offer inferior utilization with the benefit of higher feature resolution.

When using smaller block sizes, each BRAM stores less data, that is, each block covers less area of the image. Therefore, when requesting data points to the memory subsystem, the probability that two points are located in the same block (BRAM) is reduced, *i.e.* by default, the blocks size is set to be 16 pixels each. This size allows the ability to read

features sized from 4 pixels wide to 32 pixels wide in the same clock cycle. If any feature falls outside of this range, the points cannot be read in parallel and additional clock cycles will be required to read all the points. If 4-pixel blocks were used instead, features with resolution from 2 to 16 pixels could be read in the same clock cycle. Due to the fact that the main application of our system is going to be person detection, and thanks to the data obtained in Chapter 3, it was concluded that it is not necessary to have a feature resolution higher than 4 pixels.

For a fixed 50*50 Haar window, the average size of feature cascades that come embedded in the Opencv library is from 4 to 32 pixels pixels wide. This makes the default setting even more convenient. Just for illustration purposes, in Fig. 4.3, it is shown a standard passport photo. Note that the smaller rectangle, which corresponds to a 4*4 block, is big enough to detect smaller facial features like the eyes. The bigger rectangle represents a 32*32 block. This bigger rectangle is big enough to contain the entire subject's face. Note that in the case it is wanted to detect an object smaller or larger than 4 and 32 pixels, respectively, the entire image can be scaled down or up accordingly. For example, in the case when an object does not fit in the 32 pixel bound, the entire image can be scaled down until the object fits. The opposite can be done for objects smaller than 4 pixels.

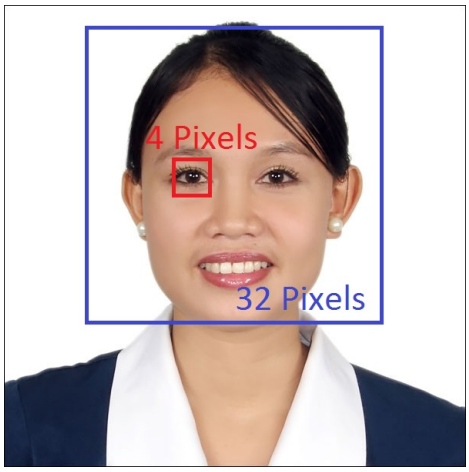

Fig. 4.3: Passport photo that demonstrates the 4-32 pixels feature coverage.

4.4 The Addressing Scheme

In section 4.3.2, the importance of the QuadraPattern for the successful development of this memory subsystem was presented. In addition to the pattern, it is necessary to design the addressing scheme that is going to be used. This section will explain how the requested points and their coordinates are processed and converted into useful information for the memory subsystem.

4.4.1 The Basics: Things that Should be Remembered

It is important to note that location *(0,0)* corresponds to the top left corner of the image. The coordinate system is numbered from left to right and from top to bottom. *i.e.* The last location fitted in the default buffer configuration is defined by coordinate *(319,63)*. This coordinate corresponds to the bottom right corner of the image window.

As mentioned in section 4.3.1, the blocks directly relate to the number of BRAMS. Once determined in which block the data point is at, the BRAM is automatically inferred. By storing each block in different BRAMs it is ensured that feature points located in different blocks will also be stored in different BRAMS, allowing them to be accessed in parallel. This is the basic principle why the pattern works for our application and also the same principle that creates the feature size constraint explained in section 4.5.5.

In order to provide higher throughput levels, the system was designed to be fully pipelined. The pipeline's architectural details are also explained in detail in section 4.5.1.

In Opencv libraries, the feature point location is stored as a *(X, Y)* point. This location coordinate is referenced to the window being processed at the moment. The system was designed to take this *(X, Y)* location information to extract the BRAM ID and generate the absolute BRAM address. Therefore no conversion is required by the Haar controller.

4.4.2 The X Coordinate Input Vector

By default, the window buffer width is limited to 320 pixels. For this reason, it is needed a 9-bit wide input vector to address all the pixels in each row. Note that by using a vector this wide, a total of 512 memory locations can be addressed. The system does not

implement any validation or special checks to the data coming from the controller. It is the responsibility of the controller designer to supply circuitry for those checks (if ever needed). More information on this can be found in Chapter 7.

In the address calculation process, there are three main parts obtained from the X vector: the *set, column*, and *Bram Part **. Due to the fact that the addressing information comes from two independent input vectors (X and Y), the BRAM selection requires a part from each vector. These parts are *Bram Part** and *Bram Part***, respectively. The use of Bram Part* is explained in detail in section 4.4.4.

A graphical representation of the X input vector is shown in Fig. 4.4. Note that the number of bits assigned to each address part is correspondent to the component's size presented in the default configuration of the implementation, *i.e.* blocks are 16 pixels each, there are 64 blocks in a set, and 10 sets per bank. Therefore, four bits are used for the set, three for BRAM Part*, and two for the column.

4.4.3 The Y Coordinate Input Vector

By default the window buffer height is limited to 64 pixels. Therefore, a 6-bit wide input vector is selected in order to access all the pixels in a column.

Just as with the X input vector explained in section 4.4.2, the second part of the data required for the address calculation is extracted from the Y input vector. For this case the

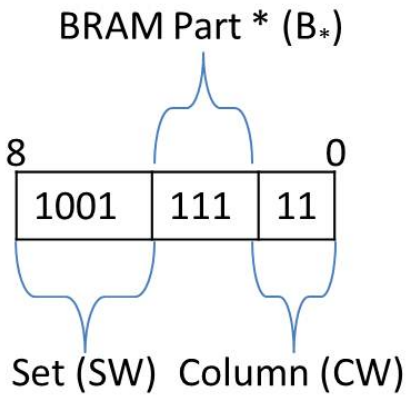

Fig. 4.4: X input vector representation.

parts obtained are: *bank, row*, and *Bram Part***. The use of Bram Part** is explained in detail in section 4.4.4.

A graphical representation of the Y input vector is shown in Fig. 4.5. Note that the number of bits assigned to each address part is correspondent to the component's sizes presented in the default configuration, *i.e.* 64 blocks sized 16 pixels each with 2 banks. Therefore, 3 bits are used for BRAM Part**, 2 bits are used for the row, and 1 for the bank.

4.4.4 BRAM Selection and Absolute Address Calculation

In this section, it will be explained how to use all the address parts obtained in sections 4.4.2 and 4.4.3 to create useful information for the BRAM addressing.

In order to facilitate the explanation to the reader, the author has defined the following constants:

- B: Number of banks;
- S: Number of sets per bank;
- b: Number of blocks per set;
- S_{size}: Height of the sets (in blocks);

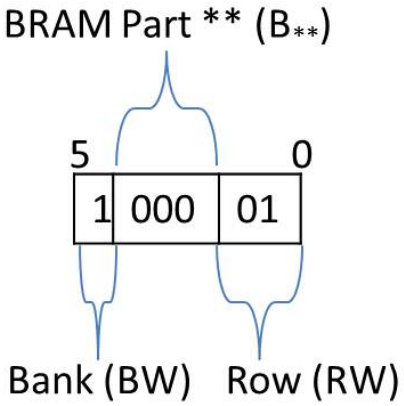

Fig. 4.5: Y input vector representation.

- w: Width of each block;
- h: Height of each block;
- P: Number of pixels per block, this is equal to $h*w$;
- A: Calculated address;
- B_* and B_{**}: BRAM Parts * and **, respectively;
- B_{id}: Selected BRAM ID.

For the default configuration, the system is set up to use 64 BRAMS distributed as follows: B equals 2 banks, S equals 10 sets per bank, b equals 64 blocks per set (8x8 blocks), and P equals 16 pixels per block (4x4 pixels).

The absolute address is computed as the sum of the weighted values of each address part. Each of these parts is extracted using the methods explained in sections 4.4.2 and 4.4.3. Given the aforementioned constants, the weight of each part is computed as follows:

- *Bank weight (BW)*: For every unit of this part, a total of $P*S$ units are added to A;
- *Row weight (RW)*: For every unit of this part, a total of $h*S$ units are added to A;
- *Set weight (SW)*: For every unit of this part, a total of w units are added to A;
- *Column weight (CW)*: For every unit of this part, a total of one unit is added to A;

where A represents the absolute address of the pixel in question.

Therefore, the final equation results as follows:

$$A = (BW * P * S) + (RW * h * S) + (SW * w) + CW. \tag{4.1}$$

In addition, the following equation is used in order to calculate the value of B_{id}:

$$B_{id} = B_* + (B_{**} \ * \ S_{size}). \tag{4.2}$$

As an example, let us take point *(319,33)*. The binary representations of the input vectors correspondent to X and Y are shown in Figs. 4.4 and 4.5, respectively. For this case we obtain *BW=1, SW=9, RW=1*, and *CW=3*.

$$A = (1 * 16 * 10) + (1 * 4 * 10) + (9 * 4) + 3 = 239 \tag{4.3}$$

$$B_{id} = 7 + (0 \ * \ 8) = 7 \tag{4.4}$$

Using equation (4.3) and equation (4.4), it is computed that in order to access point location *(319,33)* the BRAM with B_{id} equals to 7 has to be accessed with address 239.

4.5 Architectural Details

In this section, the architectural design of the memory subsystem will be explained in detail. Also, the breakdown of all the components, as well as the connection networks that links them together, will be presented. The entire system was built using *generic* VHSIC Hardware Description Language (VHDL) statements [27, 28]. All the principal parameters of all the modules are completely configurable. This will allow future designers to configure the memory subsystem to their own needs.

4.5.1 Memory Subsystem Design Overview

Due to physical limitations of the BRAMS, the memory subsystem was designed to respond only to the rising edge of the clock; unlike DDR memories that respond to both edges. All the module interfaces latch and produce valid outputs at this clock edge. The module is fully pipelined and it is able to provide twice the pixels as BRAMS are included in the system. That is, for the default configuration, the system can be configured to provide or store up to 128 pixels per clock cycle. The system designed is formed by the following 5 basic components:

1. The address generation circuit,

2. The multiplexer network,

3. The BRAM bank,

4. The de-multiplexer network,

5. The interconnection network.

Most of the design efforts were focused in the creation of the data paths that serve as a link between all the components. These data paths provide each input port the required independence to request any data at any time in a parallel fashion.

The basic data flow is as follows:

1. The request reaches the input port just before the clock edge;

2. The request is analyzed and processed by the address generation circuit;

3. The data request is channelized to the correct BRAM using the network of multiplexers;

4. The BRAM receives the request and supplies the point;

5. The information coming from the BRAM is channelized to the correct output port by the network of de-multiplexers.

One of the main characteristics achieved in the design of the system was its flexibility. The system was designed to be completely generic. This allows designers to accommodate the module to their exact needs. The module configuration is done during compile time with the use of VHDL *generic* statements [27, 28]. An additional configuration method is also available during runtime through module's control port.

The control port, *en_port*, controls how many ports of the system are enabled at any given time. Therefore, the designer can choose the maximum number of ports to be created at compile-time and choose which of these ports are going to be active during runtime. The runtime port selection and top module configuration are detailed in section 4.6.1.

The block diagram of the top module is shown in Fig. 4.6.

4.5.2 Address Generation Module

This is the first module in the data path. The data requests coming from the controller reach the input ports of this module just a couple of nanoseconds before the rising edge of the master clock. At the clock's edge, this module latches the input values and begins to decode the request.

The address generation module has 5 inputs and 5 outputs, as shown in Fig. 4.7. In order to facilitate the interface with other modules the 5 inputs have been arranged as a single vector. The binary representation of the vector is shown in Fig. 4.8. Exact copies of this module are used by each port.

The module is based on the addressing scheme presented in section 4.4. Therefore, as explained in the aforementioned section, the BRAM and ADDRESS outputs correspond to the specific BRAM identification and the address within that BRAM, respectively. This information is read by the multiplexer network and it is then when the address is passed to

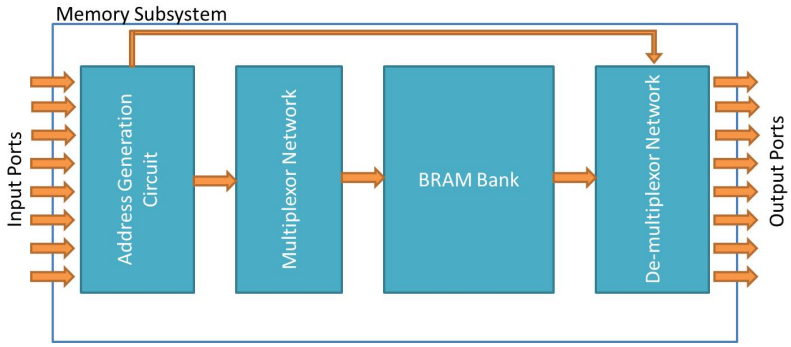

Fig. 4.6: Memory subsystem's top module block diagram.

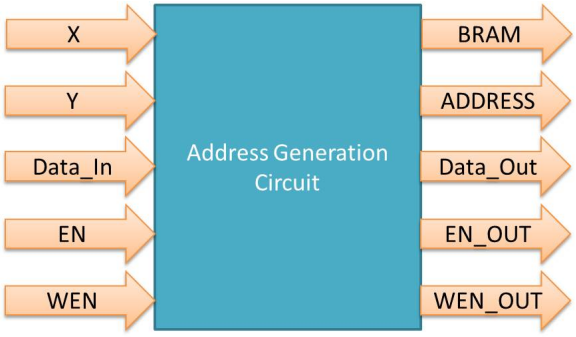

Fig. 4.7: Address generation block representation.

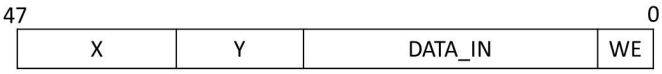

Fig. 4.8: Address generator's input vector representation.

the correct BRAM.

In order to produce a fully pipelined circuit and due to the fact that this module has the biggest latency of the 5, the outputs of the address generation circuit are also latched at the rising edge of the master clock. This allows placing a request for a data point at every clock cycle.

4.5.3 Multiplexer/De-multiplexer Network

In order to connect every input port to every single BRAM of the system, a battery of multiplexers and de-multiplexers was designed.

The multiplexers used have 64 possible outputs (one per BRAM). Each one of these outputs is conformed as a vector of 50 bits. The binary representation of this vector is shown in Fig. 4.9. For this network there is the same number of multiplexers as number of inputs.

4.5.4 Block RAM Module

For this module, a battery of BRAMS was generated. In the default configuration, the memory subsystem uses a total of 64 BRAMS to store the HWB information. The system was configured to use the 36k Xilinx BRAMS [29]. The BRAMS were configured as true dual ported with their respective enable and write-enable bits.

For the default configuration, each of the BRAMS is configured to store a total of 320, 32-bit words. Even though the necessary word size for our application was 26 bits, 32

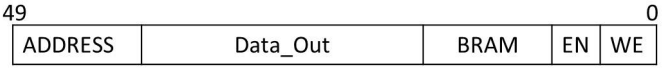

Fig. 4.9: Multiplexer's input vector representation.

bits was still chosen to increase the system's compatibility with other data types. Also, no additional registering of outputs was used because it is assumed that the top design will control the pipeline stages and the location of the pipe registers.

4.5.5 System Limitations

As previously mentioned in previous chapters, in order for the memory subsystem to supply the requested points in a single clock cycle, these points have to be separated by an euclidean distance equal to the size of the block (w or h). The reason for this limitation is that points that are located closer than h or w pixels will physically be at the same BRAM. That is, they will be located in the same BRAM so they will not be provided at the same clock edge.

The standard separation of the pixels for both axes (X and Y) is four. However, when using true dual ported BRAMS, the system can read two points from the same block. In other words, the system can provide any number of feature points in the same clock cycle if no more than two of these feature points are located in the same B_{id}. In the case that three or more points, located in the same B_{id}, are requested, the requester will have to submit a new request for the third and any other additional points. The dual port capability, allows reading higher complexity features while still having single cycle latency.

Currently, the system assumes that the Haar controller will never access parallel data that violates this condition. Therefore, as of now, the controller is responsible for the correct utilization of the memory subsystem. A possible workaround to this limitation is presented in Chapter 7.

The memory subsystem was designed in such a way that lets the HWB be configured to work as a ring buffer. Hence, any port of the memory subsystem can be configured to read the data out, while other ports write new data into the system. That is, the Haar controller can write and read data from any access port at the same time. The Haar controller can perform these actions provided it does not violate the condition aforementioned.

4.5.6 Architectural Design Summary

The memory subsystem was designed with performance increase as a primary constraint. The use of the new QuadraPattern allowed accessing the requested data without compromising the circuit's complexity. In addition the usage of resources was considerable reduced in circuits like the address generation and the multiplexer network. This makes the system not only faster, but more effective.

The entire system was built using generic VHDL statements. All the main parameters for all the modules are completely configurable. The default configurations allows the ability to store a window buffer of 320*64 pixels using only 64 BRAMS. The detailed block diagram can be observed in Fig. 4.10.

Even though the system is limited to read two points from the same B_{id}, the average size of the features that are required for our application allows single cycle accesses in most cases. In fact, the system provides single cycle access in up to 85% of the test cases. In addition, the possible use of future FPGA architectures with higher number of BRAM access ports could potentially eliminate this constraint.

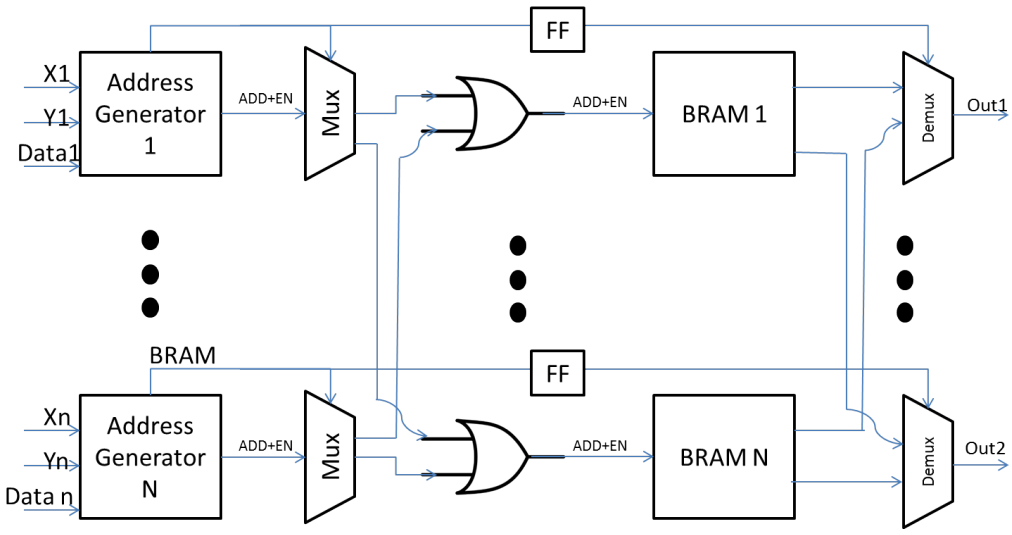

Fig. 4.10: Memory subsystem's detailed block diagram.

4.6 VHDL Code Highlights

In this section, some of the highlights of the VHDL implementation of the memory subsystem will be presented . Some of the most important control and configuration variables are explained in this section.

4.6.1 Top Module's Generic Constants

Besides the clock and reset ports, the top module has three other ports for data in/out and one for port control. In addition, it is in this module that all the generic variables are defined. In order to expand the system and use more BRAMS, create wider/narrower data vectors or simply to change the QuadraPattern parameters, the following generic variables must be updated:

- *NUM_BRAMS*: Used to set up the number of BRAMS to be used by the system;
- *X_BITS*: Used to specify the width of the X input vector;
- *Y_BITS*: Used to specify the width of the Y input vector;
- *DATA_WIDTH*: Used to specify the width of the data ports;
- *BRAM_ADDR_WIDTH*: Due to the fact that the system used Xilinx Coregen [30] to generate the BRAM blocks, the user has to specify the data width of the BRAM modules manually;
- *BRAM_PER_ROW*: This specifies how many BRAMS are going to be used per row for each pattern patch, in other words, it tells how many blocks are in a QuadraPattern row;
- *ROW_BITS*: Controls how many bits are used from the Y vector to define the block's row;
- *COLUMN_BITS*: Controls how many bits are used from the X vector to define the block's column;

- *SET_BITS*: Controls how many bits are used from the X vector to define the pattern SET;
- *BRAM_X_PART*: Controls how many bits are used from the X vector to define BRAM*;
- *BRAM_Y_PART*: Controls how many bits are used from the Y vector to define BRAM**;
- *N_INPUTS*: Controls the physical number of ports that will be synthesized for the system.

4.6.2 Port Control

As it has been stated in previous sections, the number of ports of the system can be changed at runtime to save power. Even though all the physical ports are mapped into the FPGA, a disabled port will not consume dynamic power.

In order to enable/disable a port at runtime, the Haar controller has to specify which ports are going to be used through the configuration port. Each port can be easily turned on or off by setting the port's enable bit to 1 or 0, respectively. Note that the input and output ports are enabled or disabled with the same control signal. Therefore, if a port is disabled, both the input and correspondent output port will be disabled.

As all the data inputs of the memory subsystem, the port control is also latched during the rising edge of the master clock (*clk100*). It is important to note that the configuration selected in the vector is effective as soon as it is latched by the input registers.

The pipeline of the QuadraPattern memory subsystem was designed to have four stages. If the controller submits a pixel request to the memory subsystem, it has to wait until the pipeline is completely drained before changing the value of the configuration port. Any change in the control port flushes the affected port's pipeline. If this is not carefully handled by the designer, it could potentially produce problems for the controller.

4.6.3 Address Generation

As it was shown in equation (4.1), the address generation requires several multiplications and additions to convert the X and Y vectors into useful information. The author suggests to the reader to keep QuadraPattern's components sizes in power of two boundaries. This allows the synthesize tool to convert such multiplications in simple zero padding operations.

However, in order to cover the entire 320 pixels of the buffer, ten sets (32 pixels wide each) are required to be placed side by side. For the default configuration, this requirement makes the value of $h * S$ become forty (40). Therefore, the row weight (RW) calculation would infer a non-power-of-2 integer multiplication. The solution that was implemented in the code was to manually divide the $h*S$ value into its power-of-2 factors. In the case of 40, 8 and 32 were chosen. That is, for the VHDL code implementation, equation (4.1) becomes

$$A = BW * P * S + RW * 8 + RW * 32 + SW * w + CW. \tag{4.5}$$

Using equation (4.5), the synthesize tools managed to infer zero padding operations inside the Address Generation module. Therefore, no real multiplication was implemented in the physical design. This was proved to improve the maximum frequency of the module by 4X.

4.6.4 Multiplexer and De-multiplexer Network

The multiplexer and de-multiplexer modules were written using VHDL arrays. This facilitated the management of the data and simplified the VHDL code. In the VHDL code implementation 50-bit words where used to create the arrays. While it was possible to build 450-to-50 multiplexers, it was found that considerable amount of resources were saved when building a 9-to-1 (50-bit words) multiplexers. In fact, a 9-to-1 multiplexer consumes around 25 times less resources than its equivalent 450-to-50 multiplexer.

4.6.5 BRAM Configuration

In order to create the BRAMS modules, Xilinx Core Generator 12.3 [30] was used. The BRAMS were configured as true dual port with no byte write function. The data size for both ports was chosen to be 32 bits for read and write operations. Only 320 data entries were configured for the BRAMS since this was enough to hold the 320*64 buffer window. Note that if the reader wants to configure a bigger buffer, he must change this value to the desired one and change the generic variable *BRAM_ADDR_WIDTH* to the new value. Also, enable signals were configured for both ports.

Some of the configurations values where chosen in order to reduce the power consumption of the BRAMS. One of these values is the operation mode of the BRAM module. As explained in the block memory generator manual [29], there are three operating modes which are detailed below.

- *WRITE_FIRST*: In this mode, the input data is simultaneously written into memory and driven on the data output. This mode serves as a confirmation that the data was stored in the address location specified in the *ADDRESS* port of the BRAM.

- *READ_FIRST*: In this mode, data previously stored at the write address appears on the data output while the input data is being stored in memory. This mode allows the ability to validate the data overwrites.

- *NO_CHANGE*: Unlike the other two modes, this operating mode does not offers any confirmation from the BRAM. The output port does not change its value while a write operation is being performed. That is, during a write operation the data in the output port will be the data obtained from the last read operation.

For the lowest power consumption configuration, Xilinx recommends the use of the *NO_CHANGE* configuration. This mode offers the lowest power consumption of the three methods. Therefore, the *NO_CHANGE* operating mode was chosen for the QuadraPattern implementation.

Chapter 5

Simulation Setup and Results

In this section the simulation's results for the constructed model are presented. In addition, it will also be presented all the data collected using Xilinx and Modelsim tools like resources utilization, power consumption, and timing reports. All the components used to build the simulation and their configurations are also explained in this chapter.

5.1 Simulation Configuration

For system's simulation, the memory subsystem was configured to hold a 320*64 image window. In order to do so, the configuration parameters of the system defined in section 4.4.4 were set as follows:

- B: =2 banks,
- S: =10 sets per bank,
- b: =64 blocks per set,
- w: =4 pixels,
- h: =4 pixels,
- P: =16 pixels.

Also, the generics constant defined in section 4.6.1 were set as follows:

- NUM_BRAMS: =64,
- X_BITS: =9,
- Y_BITS: =6,

- *DATA_WIDTH*: =32,
- *BRAM_ADDR_WIDTH*: =9,
- *BRAM_PER_ROW*: =8,
- *ROW_BITS*: =2,
- *COLUMN_BITS*: =2,
- *SET_BITS*: =16,
- *BRAM_X_PART*: =3,
- *BRAM_Y_PART*: =3,
- *N_INPUTS*: =9.

The simulation was synthesized for a Xilinx Virtex6 [31]. Due to the large amount of I/O ports inferred in the component's synthesis, the Virtex6 XC6VLX760-2FF1760 [32] was used. The I/O ports were only used for testing purposes. However, in real-world implementations, the module would not have any physical I/O interfaces. The device was configured in its *-2* speed step to ensure maximum performance.

The software tools used to create the module and to obtain the simulation results are detailed below:

- *Notepad++ 5.7* [33]: Used for VHDL code editing,
- *Xilinx ISE 12.3* [34]: Used as the project manager and synthesis tool,
- *Xilinx Core Generator 12.3* [30]: Used to generate Xilinx's BRAMS,
- *Xilinx Power Analyzer 12.3* [35]: Used to obtain the estimate power consumption,
- *Xilinx ISim 12.3* [36]: Used for simulation,
- *Mentor Graphics Modelsim XE III 6.5c* [37]: Used for simulation.

The test bench used for simulation and the results obtained are presented in the rest of this chapter.

In order to test the memory subsystem module, a simple but efficient VHDL test bench was written. The test bench simulated a memory self-test in which all the memory locations are tested at least nine times (in average). The test system instantiated an image buffer that was based in the QuadraPattern memory subsystem. A simple C++ program was written in order to generate the data to be stored in the image buffer. The program also validated the output of the memory test produced by Modelsim or ISim simulators.

First, the image information is generated using the C++ program. Then, the test bench takes this image information and stores it in the 20,480 memory locations (320*64 locations). After all the data has been stored, the memory is exercised using the most complex feature available for this system. That is the nine point feature presented in Fig. 2.1. The test bench slides the feature one pixel at a time across the whole image buffer. Since this is a nine point feature, by the time the test is complete, each pixel would have been read at least nine times on average.

The feature points extracted are then stored in a file which is used for further data validation. When the data fetching is complete, the data stored in the output file is validated by the C++ program. The memory test is said to be unsuccessful if a single feature point's value does not match with the validation file.

Using this simple memory test method it was the correctness of the MemSubsystem module and all its dependencies was tested. As expected for the default configuration, the maximum throughput reached by the system was one nine-point feature per clock cycle.

5.2 Physical Resources Used

After running the XST synthesis tool found in Xilinx ISE, the results presented in Table 5.1 were obtained.

Note that the resource utilization has a linear behavior with respect to the ports chosen in *N_INPUTS*. These results are extremely important in complex system configurations that

Table 5.1: Synthesis reports: Components used.

Resource	9 Ports	7 Ports	5 Ports	4 Ports	3Ports	2 Ports
Adders/Subtracters	45	35	25	20	15	10
Flip Flops	25,317	19,705	14,075	11,260	8,439	5,626
BRAMS	64	64	64	64	64	64
Slice Registers	25,317	19,705	14,075	11,260	8,439	5,626
Slice LUTs	37,301	30,249	23,179	19,644	13,322	7,132

use a big number of ports. This information can be used to quickly estimate the resources that are going to be used and also to select the correct FPGA device to target.

As mentioned in section 4.2.2, only five adders are needed per port. This can be confirmed by looking at the first row of the Table 5.1.

5.3 Timing Results

After running the Xilinx Synthesis Tool (XST) found in Xilinx ISE, the results presented in Table 5.2 were obtained. MIAT is the minimum input arrival time, MORT is the maximum output required time and MCD represents the maximum combinational path delay.

The memory design had a frequency constraint of 400MHz. Note that all configurations met the timing constraints for the system. The timing figures obtained in the timing report were pretty much constant. In fact, due to the parallel fashion or the memory subsystem presented in this work, it is expected that the timing properties are not affected at all by the port configuration. However, there are some changes in the timing values. These small changes are produced by the synthesis algorithm itself. That is, once the algorithm reaches the global solution that meets all the constraints, it stops the optimization process.

Table 5.2: Synthesis reports: Timing report.

Timing component	9 Ports	7 Ports	5 Ports	4 Ports	3Ports	2 Ports
Minimum period (ns)	2.480	2.247	2.247	2.247	2.479	2.479
Max frequency(MHz)	403.2	445	445	445	403.4	403.4
MIAT(ns)	3.157	2.950	2.742	2.638	2.44	2.341
MORT(ns)	4.43	4.593	4.586	4.586	4.01	4.01
MCD (ns)	2.057	1.694	1.486	1.382	2.233	2.130

5.4 Power Consumption

After running the Xilinx Power Analyzer (XPA) tool found in Xilinx ISE design suite, the results shown in Table 5.3 were obtained. TDP represents the total dynamic power, TLP represents the total leaked power and TP represents the total power dissipated by the device.

Note that the power dissipated through leakage is kept constant regardless of the port configuration. The leakage represents the major source of power consumption in this device. Therefore, in order to build a power efficient design, the author recommends using the highest number of ports available. The efficiency is computed as the throughput of the system over the power consumed. In Fig. 5.1 it is presented the comparison of the system's efficiency for various ports configurations.

As explained in section 4.6.5, the operating mode of the BRAMS was set to be *NO_CHANGE*. Thanks to this configuration, the power consumed by the BRAM bank was kept really low across all tests. As it can be observed in Table 5.3, the entire bank of 64 BRAMS only consumed 9.3mW regardless of the port configuration. This is one of the key factors that allowed the system to become so power efficient.

Is important to note that this graph represents the best achievable efficiency that can be obtained per port configuration. Side effects like collision, which is more common for higher number of ports, are not considered in this study and it is presented as future work

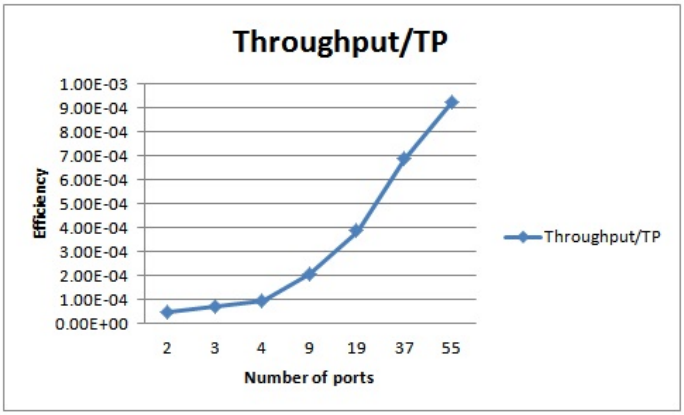

Fig. 5.1: Graphical representation of the efficiency depending the port configuration.

Table 5.3: XPA reports: Power dissipated.

Power component	9 Ports	7 Ports	5 Ports	4 Ports	3Ports	2 Ports
Clocks (mW)	224.03	182.55	139.27	113.78	105.6	70.85
Logic (mW)	19.9	15.66	11.21	8.99	6.17	4.08
Signals (mW)	115.23	84.4	63.17	50.94	37.75	28.83
BRAMS (mW)	9.3	9.3	9.3	9.3	9.3	9.3
TDP (mW)	368.46	291.91	222.95	183.01	158.82	113.06
TLP (mW)	4477.41	4472.05	4467.13	4464.36	4462.5	4459.41
TP (mW)	4845.87	4763.96	4690.08	4647.37	4621.32	4572.47

in Chapter 7.

5.5 Comparison with Baseline Memory Subsystem

As it was introduced in section 1.1, the best published implementation of the Haar person/object detection algorithm in an FPGA system was done by Cho et al. [3]. Only the HWB and related circuitry, consumed up to 75% of the available resources of a Xilinx Virtex-5 FPGA. In addition, their HWB implementation had really low throughput given that it could only provide a single point per image line at every clock cycle. Therefore, after considering the resources used and the throughput achieved, it can be said that their memory subsystem is very inefficient.

In this section, it is going to be compared the resources used by Cho et al. implementation and the memory subsystem presented in this work. In order to make a fair comparison between the two systems, the 3-port configuration of the QuadraPattern memory subsystem and Cho et al. best implementation are compared. Both of these implementations can read a 9-point feature in three clock cycles. The list of resources used by both implementations is presented in Table 5.4.

Note that it is clear that the QuadraPattern approach uses less flip flop registers and LUTs. However, Cho et al. implementation takes the leading edge in terms of the BRAM use which are kept to the minimum. In fact, their entire Haar cascade algorithm implementation only uses forty two BRAMS.

Unfortunately, Cho et al. did not publish the power consumption figures of their design. As it was presented in section 5.4, most of the FPGA power dissipation is due to transistor

Table 5.4: Comparison of the resources used.

Resource	QuadraPattern	Cho
BRAMS	64	10
Slice Registers	8,439	17,767
Slice LUTs	13,322	30,360

leakage. This occurs due to a physical characteristic of the FPGA and its independent of the design ported to it. Therefore, it is logical to assume that Cho et al. implementation will also have a high amount of power dissipated because of transistor leakage. If this is the case, the QuadraPattern is not only more resource efficient, but also more power efficient when it is configured with more than three ports.

Even though that Cho et al. implementation only uses a mere 15% of the BRAMS used by the QuadraPattern approach, the memory subsystem presented in this work is truly superior. For all other resources, the implementation presented uses less than 50% of the resources used their architecture. In addition, other advantages like the performance increase and the relative power efficiency gained with this design, make the QuadraPattern approach more appropriate for a real-world FPGA implementation.

5.6 Summary of Results

The system achieved a feature read throughput of one feature per clock cycle when configured with nine access ports. This represents a speedup of 4X when compared to the best FPGA implementation of the Haar cascade algorithm. All the performance constraints were met and with the inclusion of more than nine ports, they can even be surpassed (reading more than one feature per clock cycle). In addition, the efficient utilization of FPGA resources led to a relatively compact and power efficient design.

After analyzing all the results obtained during the simulation, it can be concluded that the system design, implementation, and testing were successful.

Chapter 6

Conclusions

Due to the relatively high complexity and computing power required by the Haar Features Cascade algorithm, it has not found its way in the embedded world yet. Not a lot of research efforts have been put into the implementation of this algorithm in embedded systems, especially in FPGA implementations. Therefore, previous attempts of embedded implementations have obtained pretty performance and/or low quality results.

The current best FPGA implementation was done by Cho et al. [3] using a Xilinx Virtex5 FPGA to synthesize their architecture design. However, after analyzing their design, it was found that it was very inefficient and that it achieved very low performance. Their HWB created a bottleneck when trying to read the feature values for the classification calculations; this resulted because their design could only provide one point per image line at the same time. In addition, their implementation used a very high amount of resources to conceive the HWB.

In this project report was presented the design, implementation, and results of a parallel-accessible memory subsystem. This memory subsystem is to be used in a HWB module of the Haar-like Feature Cascade FPGA implementation. Unlike Cho et al. implementation, this module resulted to be very resource and power efficient.

In addition, a new memory access pattern called QuadraPattern was introduced. The pattern allows designers to obtain maximum system performance without compromising scalability or implementation complexity. As it was presented in section 4.2.2, only five ADD operations were required to translate the X and Y vectors into the required addressing format. Also, the system was designed to be fully pipelined. As a result, the system performance was very high with frequency speeds reaching up to 450MHz. The high performance achieved resulted in a 4X gain in the throughput when compared to the baseline

implementation.

The use of all FPGA resources was reduced to their minimum. In fact, if compared to Cho et al. implementation, the QuadraPattern approach presented a 50% reduction in the use of FPGA resources. In addition, other advantages like performance increase and relative power efficiency gained with this design made the QuadraPattern architecture more appropriate for a real-world FPGA implementation.

Based on the performance gain, the quantity of resources used and the relative scalability of the QuadraPattern system, it was concluded that the memory subsystem design and implementation were successful.

Chapter 7

Future Work

7.1 Reliability Improvements

We have seen the successful design and implementation of a parallel-accessible memory subsystem for feature extraction. Even though the systems works as expected and that it achieved all the expected performance marks, there is still a lot of room for improvements that would increase the memory subsystem security and reliability.

7.1.1 Address Bound Check

Up to now, it was assumed that the controller would never request data that violate the addressing convention. In the actual system, there is no protection circuit of any kind. The entire control is given to the designer of the Haar controller module. While this design decision gives the designer more flexibility, it can create several problems if some extra care is not taken.

In the default system configuration, the HWB was configured to hold a window of 320*64 pixels. If the controller accidentally requests or submits data outside the buffer bounds, it can create unexpected writes or reads to/from random locations. Note that the address 320 is represented using nine bits which allow to address a total of 512 possible locations. There is not validation performed to the request coming from the controller. One future task would be to design a bound checker for the addresses coming from the processor.

7.1.2 Collision Detection Circuit

Another possible source of problem is improper the understanding of the feature points separation constraint. For the default system this number is set to four. Thanks to the

use of a true dual port BRAM, the system can provide two points that fall closer than four pixels, but not any more than that. That is, in the case the controller accidentally requests data from three points located in positions *(0,0),(0,1)* and *(0,2)* the system will respond unpredictably. Even thought it seems pretty simple to detect this problem, the detection can become complicated when the points seem completely unrelated (like when accessing points in different Banks. (see section 4.3.1).

A solution for this problem would be to implement a collision detection circuit that detects when more than two input ports are trying to access the same BRAM. To maintain the collision detection circuit's complexity low, the circuit can be located between the multiplexer network and the BRAM bank. The circuit can work as a priority advisor to the memory subsystem. That is, if the collision is created between ports 3, 4 and 9, the system can potentially continue with requests 3 and 4 and discard the one coming from port number 9. The selection of the best priority scheme is left for future research.

The circuit has to be intelligent enough to activate the *collision* flag in the correspondent port. This will announce the controller that it needs to resubmit that particular request to the memory in the next clock cycle. Using our previous example, the memory subsystem should announce the controller that the request placed in port nine will not come through and that it needs to be resubmitted in the next clock cycle. The implementation of this circuit will allow the controller to submit any points without validating the data.

7.1.3 Dynamic Queuing of Collided Requests

A possible way to alleviate the controller's load is to create an intelligent queuing system. Whenever a collision is detected by the collision detector, such colliding request is placed in a dynamic queue that will be serviced at every new clock edge. This capability would allow the controller and designer to submit any request to the memory subsystem without having to worry about a possible resubmission.

This queue has to be intelligent enough to avoid submitting requests that will create further request collisions. It is important to say that the implementation of this queuing system is not trivial. Several modifications are required to the memory subsystem. Mod-

ifications like the inclusion of a *valid* flag system which will inform the controller that a past request (queued due to a collision) is now ready at an specific port are required. The basic controller design has to be heavily modified, too. It should be able to keep a stack of unserved requests, so when they become valid, the proper action is performed.

The capability of dynamically queuing memory requests will improve the reliability of the system considerably. In fact, if this circuit is successfully implemented, the resultant Haar architecture would have the base to act as an out-of-order processor. Any architectural changes required by the queuing system are left for future work.

7.2 External Memory Interface

As it was explained in numerous previous chapters, the system presented in this work can provide or store up to 128 pixels in a single cycle. However, in order to take advantage of this capability, the points need to be separated by a distance inferred by the block size. The memory subsystem was designed to be used as a temporary buffer for pixels coming from an external memory.

External memories do not provide data in the same standard that was implemented in this work. Standard memory subsystems, like DDR memory, provide data as continuous blocks or bursts. Therefore, the data coming from the external memory cannot be used as is. In order to interface the memory subsystem with an external memory module, it is required an additional circuit that was not covered in the scope of this work.

A possible way to implement this circuit is to design a small buffer that stores as many pixels as the external memory can burst in a single cycle times the width of the block of the QuadraPattern used (four by default). The data coming from the external memory gets stored in this small buffer while the memory subsystem reads the data. That way, the system will have enough time to store all the information just before the small buffer gets full. The interface circuit should not present a major difficulty.

References

[1] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision with the OpenCV Library*, 1st ed. Sebastopol: O'Reilly, 2008.

[2] M. Hiromoto, H. Sugano, and R. Miyamoto, "Partially Parallel Architecture for AdaBoost-Based Detection With Haar-Like Features," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 1, pp. 41–52, Jan. 2009 [Online]. Available: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4703222`.

[3] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner, *Fpga-based face detection system using Haar classifiers*. New York: ACM Press, 2009 [Online]. Available: `http://portal.acm.org/citation.cfm?doid=1508128.1508144`.

[4] C. Chareonsak, "FPGA implementation of adaboost algorithm for detection of face biometrics," *IEEE International Workshop on Biomedical Circuits and Systems, 2004.*, pp. 317–320, 2004 [Online]. Available: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1454161`.

[5] H.-C. Lai, M. Savvides, and T. Chen, "Proposed FPGA Hardware Architecture for High Frame Rate (100 fps) Face Detection Using Feature Cascade Classifiers," *2007 First IEEE International Conference on Biometrics: Theory, Applications, and Systems*, pp. 1–6, Sept. 2007 [Online]. Available: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4401930`.

[6] X. Tang, Z. Ou, T. Su, and P. Zhao, "Cascade AdaBoost classifiers with stage features optimization for cellular phone embedded face detection system," *Advances in Natural Computation*, 2005 [Online]. Available: `http://www.springerlink.com/index/erq8a5qebtpucleh.pdf`.

[7] D. Hefenbrock, J. Oberg, N. T. N. Thanh, R. Kastner, and S. B. Baden, "Accelerating Viola-Jones Face Detection to FPGA-Level Using GPUs," *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 11–18, May 2010 [Online]. Available: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5474075`.

[8] "NVIDIA CUDA Website" [Online]. Available: `http://www.nvidia.com/object/cuda\_home\_new.html`.

[9] N. corp., "NVIDIA Produt page" [Online]. Available: `www.nvidia.com`.

[10] S. Chandrakar, A. Clements, A. Sudarsanam, and A. Dasu, "Memory architecture template for Fast Block Matching algorithms on FPGAs," *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1–8, Apr. 2010 [Online]. Available: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5470751`.

[11] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "Multimedia rectangularly addressable memory," *IEEE Transactions on Multimedia*, vol. 8, no. 2, pp. 315–322, Apr. 2006 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1608112.

[12] J. Vanne, E. Aho, T. D. Hamalainen, and K. Kuusilinna, "A Parallel Memory System for Variable Block-Size Motion Estimation Algorithms," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 18, no. 4, pp. 538–543, 2008 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4449079.

[13] J.-y. Peng, X.-l. Yan, D.-x. Li, and L.-z. Chen, "A parallel memory architecture for video coding," *Journal of Zhejiang University SCIENCE A*, vol. 9, no. 12, pp. 1644–1655, Dec. 2008 [Online]. Available: http://www.springerlink.com/index/10.1631/jzus.A0820052.

[14] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, pp. I–511–I–518, 2001 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=990517.

[15] R. Lienhart and J. Maydt, "An extended set of Haar-like features for rapid object detection," *Proceedings. International Conference on Image Processing*, pp. I–900–I–903, 2002 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1038171.

[16] P. Wilson and J. Fernandez, "Facial feature detection using Haar classifiers," *Journal of Computing Sciences in Colleges*, vol. 21, no. 4, pp. 127–133, 2006 [Online]. Available: http://portal.acm.org/citation.cfm?id=1127416.

[17] S. Munder and D. M. Gavrila, "An experimental study on pedestrian classification." *IEEE transactions on pattern analysis and machine intelligence*, vol. 28, no. 11, pp. 1863–8, Nov. 2006 [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/17063690.

[18] "OpenCV Library" [Online]. Available: http://opencv.willowgarage.com/.

[19] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," *Lecture notes in computer science*, vol. 3951, p. 14, 2006 [Online]. Available: http://www.springerlink.com/index/e580h2k58434p02k.pdf.

[20] Y. Freund, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting,," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, Aug. 1997 [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S002200009791504X.

[21] C. Papageorgiou, M. Oren, and T. Poggio, *A general framework for object detection.* Narosa Publishing House, 1998 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=710772.

[22] R. Jain and H.-H. Nagel, "On the Analysis of Accumulative Difference Pictures from Image Sequences of Real World Scenes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-1, no. 2, pp. 206–214, Apr. 1979 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4766907.

[23] G. R. Bradski and J. W. Davis, "Motion segmentation and pose recognition with motion history gradients," *Machine Vision and Applications*, vol. 13, no. 3, pp. 174–184, July 2002 [Online]. Available: http://www.springerlink.com/openurl.asp?genre=article\&id=doi:10.1007/s001380100064.

[24] J. Davis and A. Bobick, "The representation and recognition of human movement using temporal templates," *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 928–934, 2002 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=609439.

[25] M. Valstar, M. Pantic, and I. Patras, "Motion history for facial action detection in video," *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*, pp. 635–640, 2004 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1398371.

[26] M. McDowell, C. Fryar, C. Ogden, and K. Flegal, "Anthropometric reference data for children and adults: United States, 2003-2006," *National health statistics reports*, vol. 10, no. 10, 2008 [Online]. Available: http://ftp.cdc.gov/pub/health_statistics/nchs/printer/disc_2/data/nhsr/nhsr010.pdf.

[27] V. Z. Brown S., *Fundamentals of Digital Logic with VHDL Design*, 3rd ed. New York: McGraw-Hill Science, 2008.

[28] P. J. Ashenden, *The Designer's Guide to VHDL*, 3rd ed. Burlington: Morgan Kaufmann, 2008.

[29] Xilinx, *Block Memory Generator v3.3 Manual*, 11st ed. San Jose: Xilinx Corporation, 2009.

[30] Xilinx, "Xilinx Core Generator," 2010 [Online]. Available: http://www.xilinx.com/tools/designtools.htm.

[31] Xilinx, *Virtex-6 Family Overview*, 2nd ed. Xilinx Corporation, 2010, vol. 150 [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.

[32] Xilinx, *Virtex-6 FPGA Data Sheet: DC and Switching Characteristics*, 2nd ed. Xilinx Corporation, 2010, vol. 152 [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds152.pdf.

[33] D. Ho, "Notepad++," 2010 [Online]. Available: http://notepad-plus-plus.org/.

[34] Xilinx, "ISE Design Suite," 2010 [Online]. Available: http://www.xilinx.com/tools/designtools.htm.

[35] Xilinx, "Xilinx Power Analyzer," 2010 [Online]. Available: http://www.xilinx.com/tools/designtools.htm.

[36] Xilinx, "Xilinx ISim," 2010 [Online]. Available: http://www.xilinx.com/tools/designtools.htm.

[37] Mentor Graphics, "Modelsim XE III," 2010 [Online]. Available: http://model.com/.