NOTE TO USERS

This reproduction is the best copy available.

UMI®

TOWARD THE PARALLEL DISTRIBUTED CAMERA ARRAY – DESIGN OF A RECONFIGURABLE FRAMEWORK

Pierre-Olivier Laprise

Department of Electrical and Computer Engineering

McGill University, Montréal

February 2004

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfilment of the requirements of the degree of

Master of Engineering

© PIERRE-OLIVIER LAPRISE, 2004

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-98540-7
Our file Notre référence
ISBN: 0-612-98540-7

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

The remaining work to finish in order to reach your goal increases as the deadline approaches.

–Bove's Theorem

—-

The first 90% of the task takes 90% of the time, and the last 10% takes the other 90%

–Ninety-Ninety rule of task scheduling

—-

It always takes longer than you expect – even when you take into account Hofstadter's law.

–Hofstadter's Law

Abstract

Any attempt at extracting information about a three dimensional scene from an image or image sequence is inherently ill-posed. This is due to the large amount of information that is lost in projecting a three-dimensional world onto a two-dimensional sensor. One way to retrieve some of this lost information is to have many cameras covering overlapping regions of the scene, in the hopes of retrieving complementary information, which can be combined to form a more accurate whole. However, as the number of cameras grows, it becomes infeasible to have all the raw video feeds treated by a central processor. Our solution to this problem is to have each of the cameras do some local processing, sharing its results rather than its raw data.

This thesis describes the design of an FPGA-based reconfigurable computing framework for a single node in the camera array. Notions of the network architecture insofar as it relates to each node are developed, along with the general framework upon which applications will be built as inter-changeable modules. The framework is tested by adapting and implementing a person detection algorithm [**16**]. Extrapolation from simulation results suggests that the design, when running on a 100MHz clock, should be able to run at 3 frames per second when a person is in the scene, and 40 frames per second otherwise.

Résumé

Extraire de l'information sur une scène tri-dimensionnelle à partir d'une image ou d'une séquence d'images mène nécessairement à une multitude de solutions possibles. Ceci est dû à la grande quantité d'information perdu dans la projection d'un monde tri-dimensionnel sur un senseur bi-dimensionnel. Une méthode pour pallier ce problème est d'avoir plusieurs caméras couvrant des régions chevauchées de la scène, dans l'espoir d'en extraire des morceaux d'information complémentaires pouvant mener à une compréhension accrue de la scène. Toutefois, à mesure que le nombre de caméras augmente, il devient impossible de traiter l'ensemble de ces sources en un point central. Notre solution est de déléguer à chaque caméra le traitement de ses propres images, ne partageant que ses résultats plutôt que ses données brutes.

Ce mémoir décrit la conception d'une platforme pour le traitement d'images à chaque noeud du réseau de caméras. Les notions de réseautage ayant un impact direct sur la conception de la platforme sont explorées, de même que la marche à suivre pour construire des applications pouvant être intégrées en modules interchangeables. La platforme est vérifiée par l'adaptation et l'implémentation d'un algorithme de détection de personnes [**16**]. Une extrapolation des résultats en simulation suggère que l'application roulant à partir d'une horloge de 100MHz devrait pouvoir traiter 3 images par seconde lorsqu'il y a une personne dans la scène, et 40 images par seconde autrement.

Acknowledgments

I would like to start by thanking my supervisor, James J. Clark, without whom none of this would have been possible. He provided my funding, but more importantly he provided me with the freedom I needed to find my own solutions, and with the guidance I needed to avoid going too far astray. I would also like to thank the other members of the Motor Vision group, particularly Vinod Nair, for his help in developing some of the algorithms contained in this thesis. Still at McGill, I would like to thank the staff and crew of CIM, especially the people of 436 for making my time here enjoyable, and specifically Stephen Spackman, resident guru, for teaching me so much computer miscellany that didn't make it into this thesis.

I would also like to thank John McCluskey, our Xilinx FAE, for never letting any of my questions go unanswered, and for always making sure I had access to the latest and greatest from Xilinx. Thanks go out as well to the people at Celoxica, who graciously provided us with both hardware and software, and to IRIS for funding me both directly and indirectly, and more generally for encouraging research into intelligent systems.

Last, but certainly far from least, I would like to thank my entire family for their support, with a special thanks to my parents for instilling in me a thirst for knowledge of all kinds, and to my sister Emmanuelle for listening to me describe my circuits, and for offering the occasional bit of helpful advice.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Despite the seeming effortlessness with which our brain tackles the problem, vision is not a simple thing. Vision can be loosely defined as extracting information about a scene from optical cues. The difficulties, however, are numerous. To start with, the scene in question is occurring in a three dimensional world, but the available cues, in the form of an image, are intrinsically two-dimensional. Worse still, before it is interpreted, an image is only a mass of seemingly uncorrelated points of light of varying intensity and wavelength. It is up to the vision system to decide which of these points of light "belong" together, and what they might be doing. And therein lies the second problem. Whether or not these points belong together not only depends on their intensities and positions relative to each other and all the other pixels in the image, which is already a non-trivial issue, but, worse still, it often depends on the very information that needs to be extracted from the image, causing something of a chicken and egg problem.

The problem is so complex, in fact, that it has been split into multiple subtasks, such as depth recovery, object segmentation, and object recognition, that have been solved repeatedly with varying degrees of success. One thing that can be learned from this body of work is that computer vision involves a three-way tug-of-war between how much is learned about a scene, the amount of detail contained in the world model (sometimes referred to as hard-coded knowledge), and the amount of information taken as input. For example the simplest motion detector looks at two consecutive frames in time and determines that if they are not identical within certain noise tolerances, something in the scene has changed. By augmenting the world-model to know something about luminance, the same system can be made illumination independent so that simple changes in lighting do not register as changes in the scene. Alternately, by taking a longer sequence in time, the detector can be made less sensitive to high frequency noise in the sensors. Unfortunately, every addition to the world-model or the amount of input increases the system's complexity.

The Parallel Distributed Camera Array (PDCA) is an attempt to increase the amount of input while keeping the increase in complexity manageable. As the name indicates, it is composed of an array of cameras distributed over a certain area in the scene. The standard method of processing this information would be to send each camera's video data to one or more general purpose computers that digitize and process the video feeds. If each computer only processes a few feeds, this is an adequate setup, but the true advantage of having an array of cameras is the additional information that can be extracted from what they share. While processing multiple feeds simultaneously in real-time is possible with a very high-end setup, one quickly reaches the limits of a single processor's computing power and input bandwidth as the number of feeds grows. To reduce this problem, a small processing unit is attached to each camera. Instead of communicating raw video data, each camera can now communicate higher level information about what it sees. What's more, by networking the cameras together, they can cooperate in understanding the scene by sharing their point of view.

1.1. Project Overview

However, just as every journey begins with a single step, every network begins with a single node. The goal of this project was to design a prototype framework for the processing unit attached to the camera at each node. A proper balance had to be found between flexibility, performance and cost, properties that tend to be antagonistic.

A Field Programmable Gate Array (FPGA)-based solution offers very high flexibility in terms of design possibilities, as well as a reasonably low cost-per-unit, and very attractive performance capabilities. However, the full potential of such a system can only be achieved at significantly higher costs, in terms of application design man-hours, than an equivalent processor-based solution. This is largely due to the lack of sufficiently powerful high-level hardware design tools. Although these are coming along nicely, initial evaluation of the tools suggests that they are still far less effective than even a moderately experienced firmware designer (see section 4.10.1.2).

Pursuant to common practice in reconfigurable computing applications, the system was organized as a microprocessor tightly coupled with a reconfigurable processor. However, the increasing densities available in commercial FPGAs have made it possible to implement a "soft" processor side by side with the reconfigurable co-processor on the same FPGA chip. This increases system flexibility, since the processor itself can also be reconfigured according to system requirements, at the cost of a less powerful processor. However, since most of the processing is to be done by the application-specific reconfigurable co-processor, this is less of a disadvantage than it might seem.

Although the main goal was to design a single node, the fact that it would fit inside a network could not be ignored. To this end, the larger guidelines of the communication protocols used in the network needed to be established in order to ensure that the node could support these. Consequently, it was determined that the network's flexibility and scalability constraints could be met using Ethernet links, while higher bandwidth communications between near neighbors could be achieved using a faster, point-to-point protocol such as LVDS.

Once the basic framework had been established, an application needed to be developed for it in order to verify that it in fact fulfilled the initial requirements, as well as to allow all the details to be ironed out in a real-world application. A person detection algorithm based on [**16**] was therefore developed and implemented using the framework.

1.2. Thesis Overview

This thesis describes the various steps from framework design to application implementation. Chapter 2 reviews some papers from several fields relating to the present work. Chapter 3 describes the design of the framework, along with justifications of the various design decisions. Chapter 4 describes in detail the application that was chosen to test the framework. A description of the workings of each module is given, as well as how they are combined to give the final application. The chapter rounds off with an analysis of the resulting design. Finally, it wraps up with the conclusion in chapter 5.

CHAPTER 2

Literature Review

Although this thesis is only concerned with the design of a single node, the fact that it is but a node in a network cannot be ignored. The node's design will be constrained in large part by the fact that it is targeted to be but one processing element in a large network forming a large, distributed parallel processor. Consequently, it is important to be aware of previous work in this field. This will be covered in section 2.1. On the other hand, each node is an FPGA-based processing unit. Understanding the challenges and opportunities that this implies requires that one be familiar with what is being done in the field of reconfigurable computing. This is covered in section 2.2.

2.1. Parallel Image Processing

2.1.1. History of Parallel Image Processing. In developing a system, it is important to be aware of what has come before, since it allows the designer to avoid reinventing the wheel, and to avoid the pitfalls discovered by those that came before him. With this in mind, Michael Duff's paper on the history of parallel image processing [7] is a good place to start. He begins in the early 60's, when there was much optimism about imminent successes in duplicating the main functionalities of the human visual system. In anticipation of the algorithms that would soon be developed, it became obvious that more processing power would be required. After glossing over the problem of parallel processing taxonomy, the paper then proceeds to cover some of the more basic architectures.

First to be examined are SIMD processor arrays. These are composed of multiple processing elements (PE) running the same instructions on a small portion of an image. The typical setup has the processors receiving their instructions from a master controller, and have limited communication abilities with their nearest neighbors. Such architectures work well for low-level processing requiring access to only a small portion of the image, but the limited inter-processor communication lowers its effectiveness for intermediate and higher level processing.

This is followed by a quick overview of pipeline processors, which can be useful for processing sequences of images. In such a setup, a processor treats an image and passes the result down the line to the next processor in the chain, freeing itself up for another image. While such an architecture can have an interesting throughput, limited by the longest step in the chain, the latency is usually much too long to make this an interesting option for real-time image processing.

Next to be covered are MIMD arrays, where each PE is executing distinct operations on distinct portions of the image. These PEs are usually rather sophisticated, with significant amounts of processing power and memory. However, it would seem that the overhead quickly overtakes any gains from splitting the work. This is in part due to the difficulty of programming such machines, and to the lack of compilers capable of adequately balancing the load between the processors.

The remaining sections of Duff's paper describe the history of the attempts to produce retina-inspired processors, with large arrays of extremely simple processing elements implementing local processing on their assigned pixel and its neighbors. This research culminated with development of the CLIP4 processor in 1980.

2.2. Reconfigurable Image Processing

A reconfigurable computing system is, by the most general of definitions, any system that uses reconfigurable devices as computational elements. This approach has many advantages compared to other techniques, but also some additional pitfalls that a reconfigurable computing system designer must be aware of. Compton and Hauck [**6**] have published a very comprehensive and complete survey of the work done in reconfigurable computing. The authors begin by explaining how there is a gap between the rigid but powerful Application Specific Integrated Circuit (ASIC), and flexible but inefficient software, and that this gap can be filled by reconfigurable computing platforms. Reconfigurable devices are made up of a large number of functional units whose exact function can be configured, interconnected by programmable routing resources. By programming the various functional blocks and fixing how they are interconnected, the designer can implement custom digital circuits. Of course, this flexibility comes at a cost in performance with respect to Aspics. By extension, since microprocessors are ASICs, some functions will be more efficient running in microprocessors than in reconfigurable devices. Consequently, reconfigurable computing platforms are usually implemented using a combination of general-purpose microprocessors and reconfigurable devices, with the parts that cannot be executed efficiently with the reconfigurable logic mapped to the microprocessor, while the computational cores are mapped to the reconfigurable logic.

A reconfigurable system's run-time can be separated into two distinct phases, configuration and execution. The configuration phase consists of the host processor loading a configuration into the reconfigurable device or devices. The processor can then use the implemented circuits to speed its calculations. This process can either be done a single time at start-up, or can be done repeatedly throughout a task's lifetime depending on the functions that need to be executed. The details of this sequence vary from system to system, but usually follow this general pattern.

When integrating a microprocessor and reconfigurable hardware, many methods can be used to couple the two, as shown in figure 2.1[**6**].

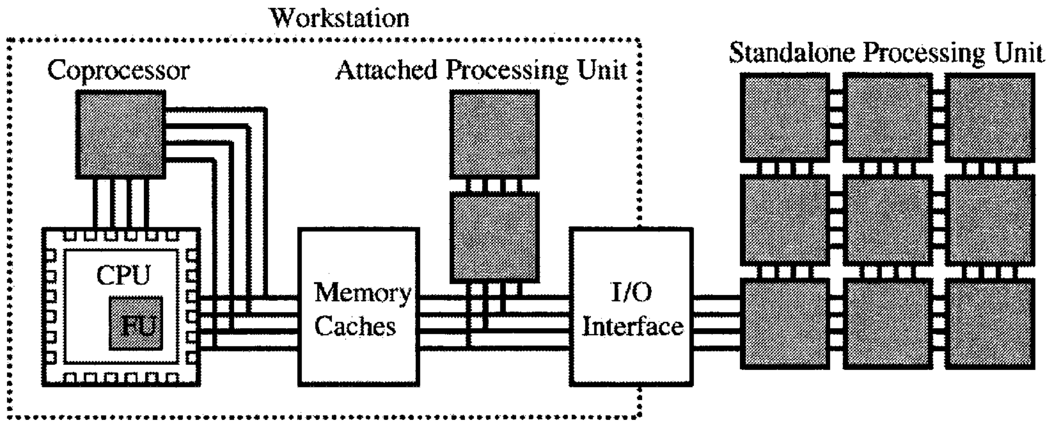


FIGURE 2.1. "Different levels of coupling in a reconfigurable system. reconfigurable logic is shaded."[**6**]

The gradation from functional unit, through coprocessor and attached processing unit, to standalone processing unit, corresponds to an increasing amount of communication overhead and cost, but also to an increase in the unit's complexity and independence from the processor. At one extreme, the functional unit extends the processor's instruction set, making its use extremely well integrated with the processor, but limiting its operational complexity to something on par with the other processor instructions. At the other extreme, the standalone processing unit is fully independent of the processor, executing entire algorithms and returning only the results, but the communication between it and the processor tend to be more involved, since such systems are more loosely integrated with the host processor, being full computational entities in their own right.

Compton and Hauck then proceed to describe various FPGA architecture types. However, this description is at a much lower level than is needed for this thesis, and is therefore left to the interested reader to look into (see [**19, 22**]).

Multi-FPGA reconfigurable systems face the added complexity of interconnecting the FPGAs and partitioning the system between them. The two basic methods of interconnection are the mesh, in which each FPGA is connected to its neighbors, and the crossbar, where each FPGA has access to every other FPGA. The advantage of the mesh architecture is that communications between proximate chips is very efficient. However, intermediate chips need to serve as intermediaries for more distant communications. The crossbar architecture, on the other hand, provides more uniform chip-to-chip communication delays, but each delay is longer than for the mesh. Another disadvantage of the crossbar is that it doesn't scale well.

On the flip side of reconfigurable hardware is the software that allows one to use it. Part of the reconfigurable computing effort includes developing the methods and tools that allow the hardware to implement the desired algorithm. In order for an algorithm to run on a specific system, it first has to be elaborated in functional terms, for example using a programming language. Since most reconfigurable systems have both a processor and reconfigurable hardware, a decision has to be taken about which parts of the algorithm should go where. Once each of these parts have been described in their respective languages, the software is ready to be compiled. A structural description of the hardware must be extracted, after which these structural components can be mapped to the specific technology of choice, finishing with the placement of the components and the routing of the connection signals. Of course, any or all of these steps can be done automatically, but the general trend is that while automating increases the ease with which new functionalities are developed, it comes at a cost in system performance and resource-use efficiency.

It can happen that the amount of hardware required to accelerate a program surpasses the amount of hardware that is available in the reconfigurable device. It is in such cases that run-time reconfiguration becomes useful. Run-time reconfiguration implies that the reconfigurable device will be reconfigured at least once during a task's execution so as to maximize the device's use. The concept is based on that of virtual hardware, which is analogous to virtual memory in traditional systems.

There are three basic models for reconfigurable computing: single context, multi-context, and partial reconfiguration. A context is a specific arrangement of functional units on the reconfigurable device. Single context devices need to be completely externally reloaded with a new configuration every time a change needs to be made. Consequently, it is important to limit the number of changes, and to optimize each configuration to have the combination of functional units that will delay the next configuration for the longest time. Multi-context devices can switch very rapidly between two or more contexts, and can swap inactive contexts out while the active context is running. This allows the user to "prefetch" configurations, but the user is still limited to swapping entire contexts. If this is required too frequently, new configurations will not have time to be loaded and the system will stall. Partial reconfiguration devices allow reconfiguration of only part of the device. This allows a drastic reduction of configuration file sizes, and thus load times, in addition to the fact that very often portions of the chip can continue running while other portions are being reconfigured. However, real-time reconfiguration adds another dimension of complexity to system design, since the functional units will now need to be placed and routed not only in space, but in time as well.

CHAPTER 3

System Framework

Before the applications of distributed and reconfigurable computing to computer vision can be explored, it is necessary to establish a basic system framework to support higher level algorithms. The core idea for the framework is to have a network of cooperating camera-processors. Each camera-processor should be composed of a camera unit tightly coupled to a processing unit. Each node in the network should have enough processing power to implement moderately complex image processing algorithms in real-time, with the true power coming from the networked interaction between modules.

The system can in fact be divided into two subsystems, each relatively independent of one another, but both equally important to the overall performance of the system. In order to take full advantage of the network processing, a proper network architecture must be chosen. The basic idea in having a processor at each node is for nodes to pre-process their information, transmitting higher level information instead of raw data, thereby reducing bandwidth requirements. However, many computer vision algorithms utilizing multiple images require large amounts of raw data to be shared, and this must also be taken into account. More important than the bandwidth consideration, though, is the requirement of scalability. The proposed network should be able to accommodate from only a few nodes to as many as several hundred, possibly spread out over large distances. This will be covered in detail in 3.1.1

The second subsystem is the camera-processor node itself. It is important to have the camera tightly coupled to the processing unit in order to permit high-bandwidth communication between the two. These nodes need to be as flexible as possible in order to provide for a large variety of algorithms. It is also important to get as much "bang for the buck" as possible, since the algorithms will need to be functioning in real-time, yet a network composed of a large number of nodes should still be economically feasible. The methods used to fulfill these requirements are covered in section 3.1.2

3.1. Framework Description

3.1.1. Inter-Camera Communications. The network architecture is subject to two seemingly exclusive constraints: one for scalable communication over long distances, and one for high-bandwidth communication between nodes. However, high bandwidth communications are mostly needed for transferring large amounts of raw image data, and the further apart in space or time two images are, the harder it is to find a direct correlation between their pixels. In addition to this, high-speed communication becomes harder as the distance between the nodes grows. Therefore, it becomes more advantageous as the cameras get farther apart to use higher level constructs such as edges, shapes, or even object types instead of raw pixel data. Such higher level constructs tend to require fewer bits to describe than the raw pixels they are derived from.

This naturally leads to developing a two-level architecture with a limited number of high-bandwidth, near-neighbor, dedicated links to transmit raw pixel data, supplemented by a standard, scalable network architecture for longer distance communication of higher level constructs.

3.1.1.1. *Scalable Network Architecture.* The most important factor in deciding what architecture to use for the global network are those of flexibility and scalability. Another important factor is that of ease of use and implementation. Although the networking aspect is an important aspect of this project, it is not the main subject of research and should be as transparent as possible, with integration as seamless as possible.

Such restrictions led to choosing Ethernet as the base global communication protocol. The Ethernet protocol provides for a sufficiently large number of nodes, theoretically bounded by the size of the MAC address, a number which is unlikely to be the true limiting factor in network size. Ethernet also provides for communication between any two arbitrary nodes, and permits simple addition and removal of nodes. It also simplifies connection to existing networks, allowing the use of existing infrastructures. Ethernet thereby meets both the flexibility and scalability requirements. It also satisfies ease of use and implementation due to the widespread support of the protocol by the community at large. Innumerable commercial and free implementations of the Ethernet protocol exist in many different formats, be it software or hardware. It is also easy to find board layout examples of the implementation of the Ethernet physical layer, simplifying the future physical implementation.

3.1.1.2. *Near-Neighbor Network Architecture.* High bandwidth communications on the network will most often be required between cameras located physically close to one another. Few of these links should be required for each node, however. This suggests that a simple point-to-point high speed bus protocol could be used to reduce overhead.

Many low-level I/O standards lend themselves to such high-speed transfers, chief amongst these being the many differential-signaling standards, such as the various flavors of Low Voltage Differential Signaling (LVDS), or Lightning Data Transport (LDT). These specific standards have the added advantage of being directly supported by the family of FPGAs that was chosen for this project (Xilinx's Virtex-II, [**19, 22**]), and it appears likely that they will continue to be supported in upcoming Xilinx FPGAs as well.

Since the exact disposition of the modules is not known in advance, it is important to allow for multiple possible configurations using the same physical module. Using the LVDS standard with Virtex-II FPGAs allows a large range of connection distances using the same PCB by using output buffers of different drive strengths and voltage swings [**19, 23**], for example using LVDS or LVDSEXT. It is conceivable that using this standard will allow each line to be driven by Double Data Rate (DDR) registers at up to 100MHz. This amounts to a raw data transfer rate of up to 25MBps for each data line.

The LVDS standard can be used for high speed point-to-point inter-board communication, but a data layer protocol still needs to be chosen. Such a layer would need to be very simple and have low overhead. One promising avenue of research is to use the phase-locking or phase-shifting capabilities of the Virtex-II's Digital Clock Manager (DCM) in an attempt to synchronize the sender and receiver, and dealing with the internal clock domain crossover using traditional methods.

Although LVDS is a point-to-point protocol, it would also be possible to organize a small group of neighbors into simple sub-networks, for example using a ring organization. Using a reconfigurable matrix such as the FPGA to control these communications also allows a lot of flexibility in the trade-off between the number of neighbors that each node has against the width of the bus. For example, if 64 pins were dedicated to LVDS communications, various applications could use these for bi-directional communication on 32-bit buses, or split into smaller buses to form hypercube sub-networks.

3.1.2. On-Camera Processing Unit. The desire to have as many nodes as possible, combined with the requirement that each node be capable of executing moderately complex algorithms in real-time, were the guiding constraints in node design. This required the solution to be as cost-effective as possible, yet still be capable of high-performance image processing. Reconfigurable computing architectures have shown time and again an ability toward accelerating image processing tasks [**1, 4, 5, 8, 14**], and are therefore ideally suited to this application.

Current advances in FPGA technologies are making it possible to envisage the design of a System On a Programmable Chip (SOPC), in which all the components which previously required separate components on a Printed Circuit Board (PCB), can be fit onto a single FPGA chip. This allows embedding of a microprocessor onto the FPGA itself instead of placing it alongside on a PCB. Xilinx provides two avenues to this effect, the MicroBlaze$^{\text{TM}}$soft processor for the Virtex-II$^{\text{TM}}$family of chips, and the IBM PowerPC$^{\text{TM}}$processor embedded into the Virtex-II Pro$^{\text{TM}}$family of FPGA's.

Using such an approach, the microprocessor can be used to control the general behavior of the system, as well as the complex network communication protocols. In parallel, an Application Specific FirmWare (ASFW) module can be configured to do the bulk of the processing, taking full advantage of the parallelism offered by custom hardware designs. This permits a tight, optimized, coupling between the data processing elements and the data source, such as the camera or memory.

The more conventional approach to a reconfigurable computing architecture is to have a dedicated micro-processor ASIC using an FPGA as a reconfigurable co-processor. Such a configuration has many advantages, namely performance, but it sacrifices some of the flexibility and cost-effectiveness of a single-chip solution. In the early stages of system design, flexibility is more important, but the exact physical layout of the various components has minimal impact on application design and can be decided on a case-by-case – or even node-by-node – basis as long as the interfaces between the nodes are respected.

An example node system architecture setup can be found in figure 3.1. If a two-chip system were chosen, the MicroBlaze could be replaced by a separate processor, and the Fast Simplex Link (FSL) [**21**] bus by whichever interface is required by the processor.

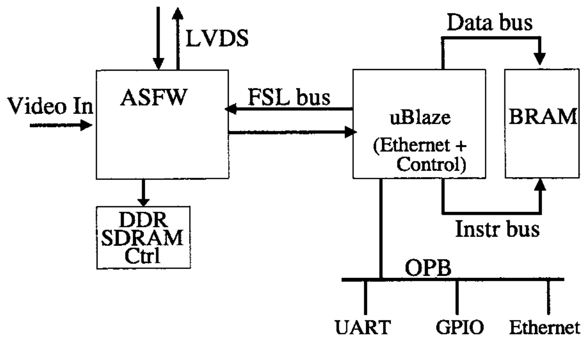


FIGURE 3.1. Organization of IP Cores on the FPGA for a single chip solution. The ASFW is directly connected to the video input and other performance-critical I/O, while the MicroBlaze communicates with slower peripherals.

However, several aspects of this layout should be conserved to allow for maximum efficiency. Performance for most applications will possibly be limited by the speed at which the data can be brought into and sent out of the FPGA chip, rather than by internal processing power, as illustrated by the test application described in chapter 4. Connecting the high-throughput data sources directly to the ASFW, instead of routing the data through the processor, allows an optimization of these interfaces on an application specific basis, and helps in minimizing communication overhead, increasing overall system performance. This also increases system flexibility since, should these signals be required by the processor instead, they can still be routed through the FPGA with minimal delays, with the added opportunity of transforming them into a more convenient form, for example by doing color space conversions, illumination adjustments, noise reduction, etc, as they stream by.

The framework's implementation details, along with an analysis of its inherent strengths and weaknesses, will be explored in the next chapter in the context of an algorithm that was implemented to test the limits of the design.

CHAPTER 4

Proof of Concept Application

Although many problems can be predicted ahead of time and planned for in the framework's initial design, many issues can only become apparent after it has been used in a real-world problem. To this end, an application is required to test as many of the system constraints as possible. The application in question needs to show that the framework is capable of running useful image processing algorithms at high speeds. Given the larger context in which the system will operate, it is also important that the implemented algorithm's output be useful for other nodes implementing higher level algorithms.

Although reconfigurable computing systems have often proved themselves for use in image processing, the systems usually can only implement simple, low-level, algorithms [**1, 15**], or have access to a large amount of processing power [**3, 13**]. Until recently [**8**], many of the more complex algorithms were usually deemed unsuitable for FPGA implementation due to the necessity for floating point operations or because of the limited amount of rapid, on-chip, memory compared to the volume of data that needs to be processed. However, this greatly limits the choice of algorithms, so it was deemed more profitable in the long term to try to meet these problems head on and find methods to mitigate the impact of these problems on the overall performance of the system.

An overview of the chosen algorithm is given in section 4.1, with a review of the guiding constraints of the design in section 4.2, followed by an exploration of the algorithm's implementation in sections 4.3 to 4.8. The results of this implementation are examined in section 4.9.

4.1. Algorithm Overview

4.1.1. Algorithm Description. The original version of the algorithm that is being adapted, developed by Viola and Jones, can be found in [**16**]. It is an object detection algorithm allowing for efficient detection of objects of variable size, but constant shape.

Objects are detected by examining rectangular regions of fixed aspect ratio, but of variable scale and position in the image. The base aspect ratio is chosen to fit the desired object. In the case of a person standing upright or walking down a corridor, this is on the order of 1:3.

Each region is examined using a series of cascaded strong classifiers of increasing complexity. Each strong classifier is composed of a number of weak classifiers, referred to throughout this work as "features". Each feature is a scaled version of one of the 10 feature types from Figure 4.1. A feature's distinguishing characteristics are its position in the window, its size, a feature threshold, and its weight. The weight of a feature determines how much importance a given feature has in the final decision for a stage. If the sum of the weights for all the positive features in a stage is greater than a certain threshold, then the stage is positive.

The value of a feature is determined by the difference between the sum of the pixels in the black regions and the sum of the pixels in the white regions. A feature

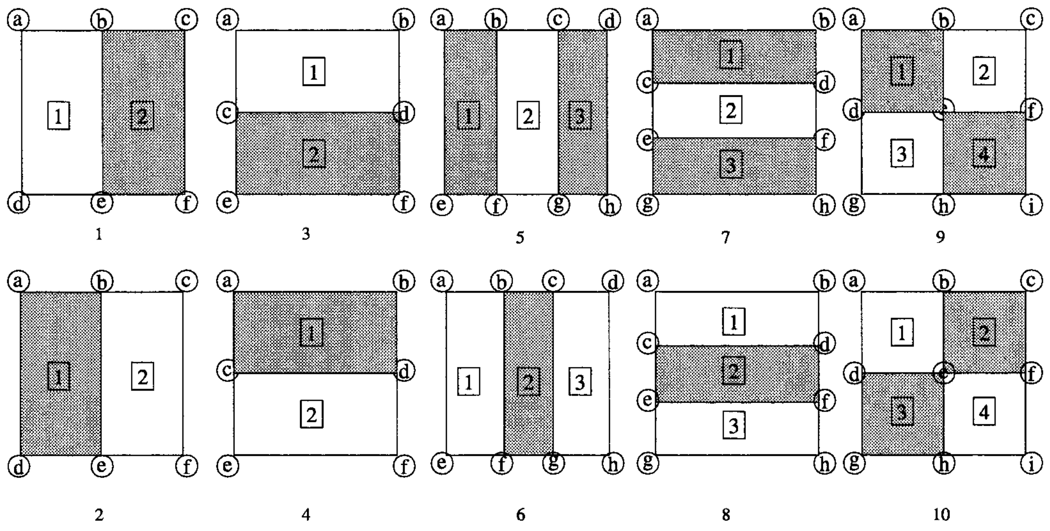


FIGURE 4.1. Set of all possible feature types used in detection algorithm.

is deemed positive if its value is beyond a certain threshold, and negative otherwise. Once the sum of the positive features' weights surpasses a certain threshold, established in training, the window is declared positive for the current classifier stage, and passed on to the next stage, where the process is repeated with a greater number of features. If the window fails in any stage, it is declared negative, and the subsequent classifier stages do not need to be evaluated.

The efficient evaluation of features is achieved using an "integral image" encoding format. In an integral image, the point in the r_{th} row and c_{th} column is $\sum_{i=1}^{r}\sum_{j=1}^{c} p_{i,j}$, where $p_{i,j}$ is the the value of the pixel in the i_{th} row and j_{th} column of the original image. This has the the advantage that once the integral image is calculated, the sum of pixels inside any rectangular region in the image requires just 3 additions and a subtraction, independently of the size of the region, which in turn allows an efficient evaluation of the basic features at all scales required by the algorithm

As illustrated by figure 4.2, the first classifiers in the cascade only have a few features, allowing them to be evaluated very rapidly. However, the features are chosen so that the most "obvious" negative images are discarded in this stage. Any regions that pass this first stage are evaluated using the second stage, which has more features, and therefore takes longer to compute, but will have fewer false positives. This continues through a fixed number of stages, chosen before training, and any region that passes through all stages successfully is deemed a positive match.

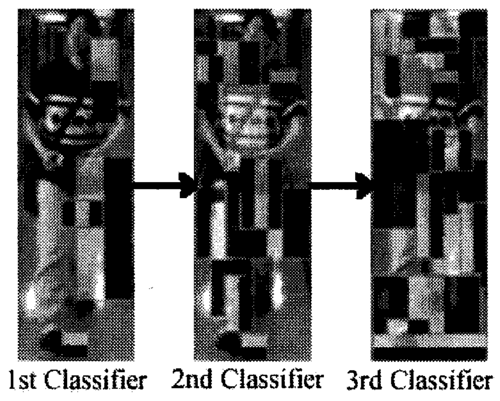


FIGURE 4.2. The detector is composed of multiple classifier stages, each of increasing complexity. As soon as a stage is failed, the window is declared negative.

Of course, there are innumerable possible features when all the possible combinations of scale, position, type, and threshold are taken into account. Selecting the most suitable features for each classifier stage is done using the AdaBoost training algorithm [**16**]. This algorithm is fed a large number of labeled training images in two sets, one with cropped positive images of different sizes, and one with cropped negative images of various sizes taken from the scene. More ample details of the decisions taken in training and in the implementation of the software version of this algorithm can be found in [**12**].

4.1.2. Suitability of the Algorithm to the Problem. Even from such a succinct description of the algorithm, it is possible to see how implementing this algorithm using the framework will test the various system constraints.

4.1.2.1. *Usefulness.* Object detection is a much studied and often used image processing goal. More importantly, it can serve as a first step in many more complex algorithms. For example, if power consumption is an issue, the arrival of an object in a scene could be used as a trigger to wake other nodes in the network. Additionally, the algorithm outputs the location of the object in its frame of reference, which can often be translated to its position in the scene with a minimum of effort and calibration. Such information can be used by neighboring nodes to focus exclusively on this region, limiting the amount of processing necessary to accomplish their task.

4.1.2.2. *Algorithm Complexity.* Viola and Jones' algorithm requires multiple passes over an image. Combined with the size of the source image, this forces integration of an efficient method for communicating with off-chip memory. As will be seen in following sections, the use of the integral image, although it forces us to use 32 bits per point, saves memory bandwidth by reducing the number of points that need to be fetched per operation, and thereby the total number of bits.

The original algorithm uses floating point numbers for many of its operations. It is possible to replace these by scaled integer operations without losing any generality. Although these examples are specific to this algorithm, there are deeper principles that can be extracted from these examples for use in future projects.

4.2. Design Constraints

Implementation of a design forces one to come face to face with reality, and all the added constraints and problems that this implies. The added constraints can be separated into two groups: those that are intrinsic to the system as a whole, which should therefore recur from one problem to the next, and those more specific to the current application.

4.2.1. Intrinsic System Constraints Identification. The main constraint faced by most FPGA-based image processing applications will be the small amount of local memory. Although memory densities are constantly increasing, permitting more and more local memory, even the largest member of the Virtex-II family of chips, which at the time of this writing is one of the high-capacity families, only contains 3.4Mbits of memory distributed throughout the chip into. Even treating this memory as a single block, it would only be enough to hold 432 8-bit pixels, barely enough for a 20x20 grayscale image. Furthermore, the memory is actually divided into blocks of memory are distinct entities distributed roughly evenly over the FPGA's lattice. Feeding the data from a block of memory to a position on the chip which is far away introduces significant routing delays that degrade the system's overall performance. For all these reasons, the chip memory should not be used as a central cache available to all modules on the chip, but should rather be used in a local manner by the various functional units. The memory blocks in the Virtex families of chip are fully dual-port, with each port having an independently variable word width [**23**], which permits the use of a single block for storing two different types of data of varying width. However, only certain widths are permitted, which means that much care should be taken in choosing the width of data types to avoid wasting memory.

Unfortunately, most image processing algorithms require a lot more memory than what is strictly available on an FPGA, which implies that some sort of off-chip storage must also be available. Communication with such a memory will come at a significant performance overhead, and the number of external memory accesses should therefore be reduced to a minimum. Although not always practical, data compression techniques should be considered, as compression and decompression hardware can often be implemented with minimal impact on the actual performance of the design. Any preprocessing that can be done to the data to minimize the number of memory fetches (for example storing the image in integral format as described in section 4.4) can also result in overall gains in the final design, and should thus be seriously considered.

4.2.2. Application Specific System Constraints Identification. The application specific constraints mostly stem from the prototyping platform that was chosen for implementation of the initial phases. Although the video input will be tightly coupled to the FPGA in future versions, no readily available commercial board was found to satisfy this requirement along with all the others. The chosen prototyping board was Insight Memec's V2MB1000 board, which provides support for a large variety of I/O, such as an Ethernet physical layer and LVDS compliant traces, which is useful for high speed board-to-board communication. It was decided that the actual source of the video feed had limited impact on the functionality of the system, and that reasonable performance estimates could still be garnered from the implementation.

For the example node application described in this paper, the source images were gathered from a network camera with on-board JPEG compression and HTTP server over Ethernet, the Axis Communication NetCam 200+. Unfortunately, the live stream from this camera is limited to one 352x288 pixel frame per second. Having a JPEG video source allowed exploration into the possibilities of integrating real-time image compression and decompression into a standard computer vision algorithm. Integrating compression and decompression of a data stream into the application (whether using JPEG or other means), is an important step toward reducing the bandwidth required in transferring data between network nodes or even between the FPGA and memory. In fact, hardware implementations of computer vision and image processing algorithms are often limited by the speed of their I/O instead of the speed at which they are able to process data, which is why most of the algorithms implemented in hardware are of the streaming filter variety, where memory demands are limited. Compressing the data that is to be transferred therefore has the potential of improving system performance by increasing the effective bandwidth as long as compression and decompression of the data have a limited effect on the speed of the overall system.

4.3. Implementation Overview

The global system organization can be seen in figure 4.3. Execution starts when the MicroBlaze processor retrieves the JPEG image from the network camera using HTTP over Ethernet. The JPEG data is separated from the header and passed to the people detection module through the JPEG decoder, described in more detail in this section. The integral image is stored in DDR SDRAM for the use of the application specific firmware. Since only a single node was implemented, the LVDS bus was not required. Each of the hardware modules are often referred to as Intellectual Property (IP) Cores. They are designed to be as independent as possible of one an other so as to encourage module re-use.

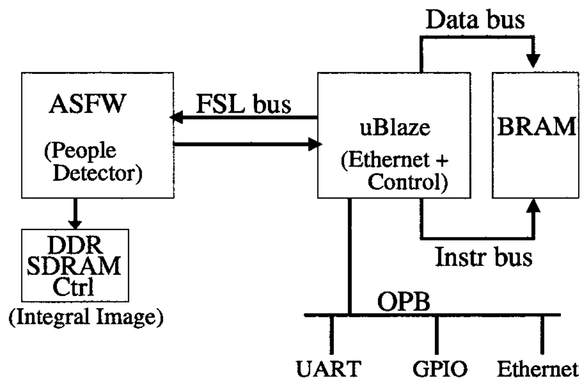


FIGURE 4.3. The ASFW contains the people detector firmware, while the MicroBlaze is only concerned with interactions with the outside world.

4.4. People Detection in Firmware

At first glance, the implementation of Viola and Jones' detection algorithm is rather straightforward. The module implementation's flow-chart is shown in figure 4.4. Having stored the image in integral image format, the sum of pixels in a rectangular region of any size only requires the addition or subtraction of the region's four corners. Therefore, all that is required to calculate a feature's value is a simple accumulator circuit. Each accumulated point is either multiplied by ±1 or ±2, and the features that were used can have either 6 or 8 points of interest. The only difficulty stems from the limited amount of on-chip memory available. Since the integral image is much too large to fit on-chip, it must be stored in an external memory, which is necessarily much slower to access. In the current implementation, the off-chip memory is a DDR SDRAM with a 16 bit data bus working at 100MHz. This allows for a transfer rate which can never surpass 32 bits per 10ns. With the targeted FPGA system clock speed also at 100MHz, the system would be receiving at most one integral point per cycle, even without taking into account addressing overhead and memory refresh times. Various methods were considered in order to compress the integral image and allow greater throughput, but the necessity to access widely separate points in a pseudo-random order made this a difficult task. Given that we have a priori knowledge of the patterns with which data points can be fetched, it should be possible to optimize the memory controller for this application, for example by inserting memory refreshes in natural pauses in the flow, but this would be far from trivial and is therefore left as future work.

Classifier training determines the position, size, and type of features that are required to detect a person. These values are all given with respect to a 16x48 template window, which is then shifted and scaled to detect people of different sizes in varying places in the image. The address of a point in external memory therefore

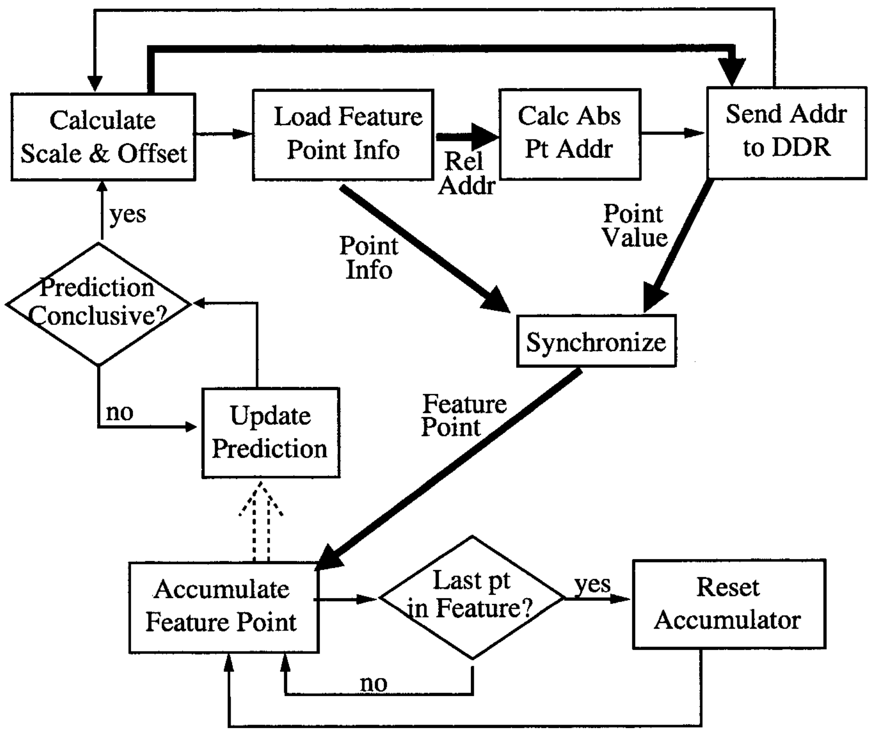


FIGURE 4.4. Flowchart for the firmware implementation of the people detection algorithm.

depends both on its position in the template window, and on the template's position and scale in the image at any given point in the scan. Training also determines the feature's threshold, its weight in the stage, and the global threshold for each stage. In order to minimize delays, this information should all be stored on-chip, but this can require a large amount of memory if one is not careful. Consequently, it is necessary to organize the information so as to most tightly pack it into the available memory formats. The first step toward accomplishing this is to realize that many pieces of data are always accessed together. For example, all the information identifying a specific feature will never be fetched separately. This means that they can also be stored together in memory. Concatenating feature information elements into a single memory word allows the designer to optimize memory use, and can have the added advantage of allowing a more compact design by simplifying the routing of closely related datapaths. However, there is too much feature information to store into a single memory word, which forces the feature information to be split into a total of four memory blocks. In an effort to hide the inner complexity of the memory map, the memory blocks are encapsulated so as to show the functional data separations instead of the physical ones. This has the added advantage of increasing inter-device portability, since the same code could be kept by only changing the architecture of the memory modules.

Once the width of the memory blocks is determined, a word addressing scheme needs to be developed. The total number of classifier stages, as well as the number of features in each stage, are determined in training, and vary widely depending on the targeted detection and false positive rates, and can be used to tweak overall detector performance. The need for flexibility, added to the need for a compact storage scheme to maximize memory utilization, and given that features are evaluated in a fixed order, naturally leads to storing feature words sequentially in memory in the order that they are evaluated. However, this requires that the memory address of the first word in each stage be accessible for when the predictor forces the preemptive end of a stage.

In order to avoid using external memory to store feature information, it is necessary to optimize internal memory use by selecting the data width and formats for the different feature data. While there is no restriction on the data width as there is in a general processor, the FPGA can only allow certain addressing modes for its high-density memory, which restricts the width of the fetched data word. This leads to storing disparate pieces of information into the same data word, splitting it according to use in the hardware. The critical choices at this juncture are the size of data for the threshold and weight of a feature.

The feature threshold and weights are good examples of the two types of data that can benefit from proper sizing, and a good opportunity to show the possible approaches to choosing a data width for a particular signal. The threshold is an absolute value determined by the size of the template and the maximum value of a pixel. Its maximum value can therefore be determined to fit inside 18 bits. However, classifier training shows that no threshold ever requires more than 15 bits. Given that the FPGA can be reconfigured if retraining ever changes this, the datapath can be optimized to use this fact. Judicious use of generics (in VHDL) or defparams (in Verilog), and timing and placement constraints, can make changing the data path width a simple matter of resynthesizing after a parameter change, and can lead to improved performance and reduced area usage.

Unlike the thresholds, the weights only have importance in terms of their sizes relative to one another. An analysis of the distribution of weights in the trained classifiers shows that using 18 bits to represent the weight values uniformly distributed from zero to the maximum allows all but 0.13% of the weights to have a unique value compared to the full floating point representation.

Despite all this, the main bottleneck for the people detection module is still the external memory interface to the integral image values. This can be alleviated somewhat by taking advantage of consecutive reads from a same row, but it does not eliminate the problem. The best way to further alleviate the bottleneck is to reduce the number of memory accesses. The main method chosen to this effect is to detect whether or not additional features are necessary to determine the success or failure of a window. The maximum value that can be achieved by a window is equal to the sum of all its features' weights, and since the features in a window are all known ahead of time, this maximum can be precalculated for each classifier stage. As features are calculated, they get resolved either into increasing the currently accumulated window value, or into decreasing the maximum attainable value. If the window value surpasses the target threshold, it is judged positive. Similarly, if the maximum attainable value falls below threshold, success becomes impossible and the window can immediately be judged negative. Either way, the success or failure of the window is known without needing to calculate (and thereby fetch from memory) more features. However, it is imperative that this prediction algorithm be outside the critical path to avoid slowing down the algorithm. This can be achieved by implementing it in a separate module running in parallel, which can be checked in a non-blocking manner to determine whether or not the next feature point needs to be fetched.

4.5. Calculating the Integral Image from JPEG

Computing the integral image of a grayscale frame is simple (see [**16**] for details) if the frame is not compressed. In our case, however, it is compressed, in JPEG format. The JPEG decompression algorithm involves computing the inverse Discrete Cosine Transform (DCT) [**11**], which requires nontrivial hardware resources and computational effort. Therefore we seek to avoid computing the inverse DCT. It is possible to obtain the integral image directly from the DCT coefficients because both the forward and inverse discrete cosine transforms are linear transformations, which means that the coefficients are linear combinations of pixel values and vice versa. So the pixel sums required in the integral image computation can be obtained through linear combinations of the DCT coefficients.

Figure 4.5 shows how such a direct method would work, and for comparison the gray box shows the indirect method of computing the integral image. But computing the inverse DCT and computing the integral image from the DCT coefficients are roughly equivalent since both the grayscale frame and its integral image contain the same amount of information, and the conversion between the integral and grayscale forms is trivial compared to the inverse DCT. Therefore calculating the integral image directly from the DCT coefficients requires about as much effort as the inverse DCT itself. However, it may be possible to directly compute an *approximate* integral image with fewer computations.

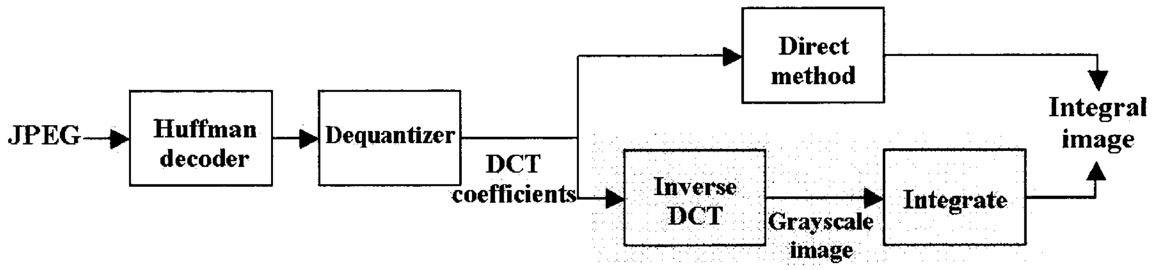


FIGURE 4.5. Once the JPEG image's DCT coefficient are decoded, the standard method would be to perform an inverse DCT and integrate the resulting grayscale image. Instead, the integral image can be extracted directly from the coefficients.

4.5.1. Extraction of DCT coefficients.

The extraction of DCT coefficients from a JPEG stream requires first that a Huffman encoded value be decoded, which is then used to decode the bits in the stream which encode the coefficient's actual value. This necessitates the use of a Huffman table which is transmitted with the image. However, JPEG encoders (including the one used for this experiment) generally use the same Huffman table for all the images that they generate. Having verified whether this is the case for a particular encoder, and with knowledge that all future images in the series will come from the same encoder, it is possible to only extract the table from the first image received or, in a prototyping environment, to hardcode the tables into the FPGA's configuration bitstream. The quantization tables used in section 4.7 may be treated in the same manner.

It was clear from the start that the speed at which the JPEG decoding module processes data would be limited at the input by the fact that the data is being sent over a 10/100 Ethernet line, which has a maximum transfer rate of $10ns/bit$, and at the output by the integral image module, which needs to write its results to external memory. Therefore, a simple serial lookup table approach to Huffman decoding, such as the one described in [**11**], should be sufficient to meet data rate limitations at both ends. Once the Huffman decoding is complete, decoding the coefficient's value and index is relatively simple to do in parallel at a small additional cost in complexity. Simplified block diagrams of these modules' implementations can be seen in figures 4.6, 4.7 and 4.8.

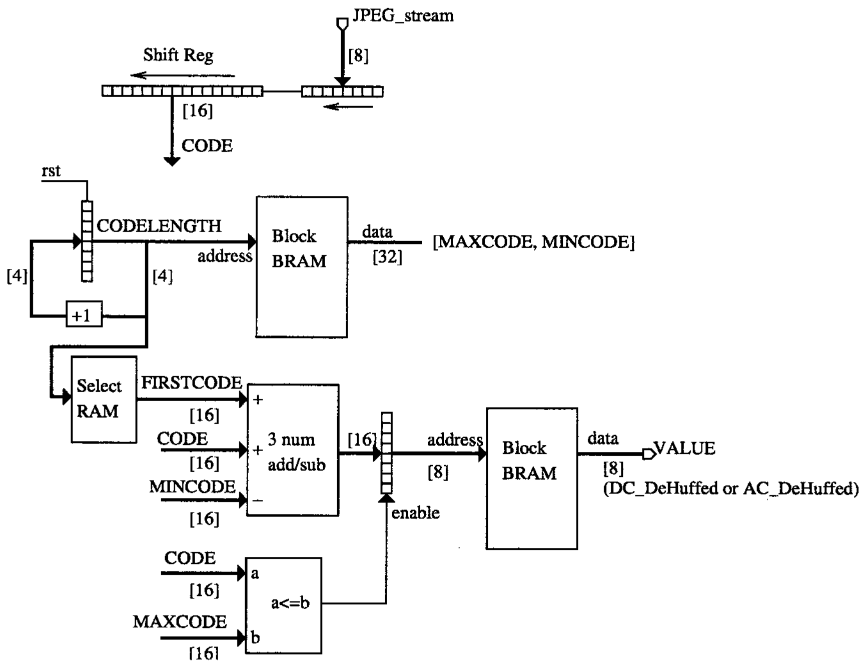


FIGURE 4.6. Simplified block diagram for Huffman decoding module.

Tests on a source image suggest that if the decoding hardware were only slightly limited by input speed (overhead of 2 cycles per 16 bits of data), the hardware should take approximately $75kcycles$ to treat a typical image, which translates to $1.5ms$ for a worst case $20ns$ minimum period. However, as will be seen in section 4.7, most of this time can be absorbed by the calculation of the integral image, with a simple FIFO buffer to synchronize the modules.

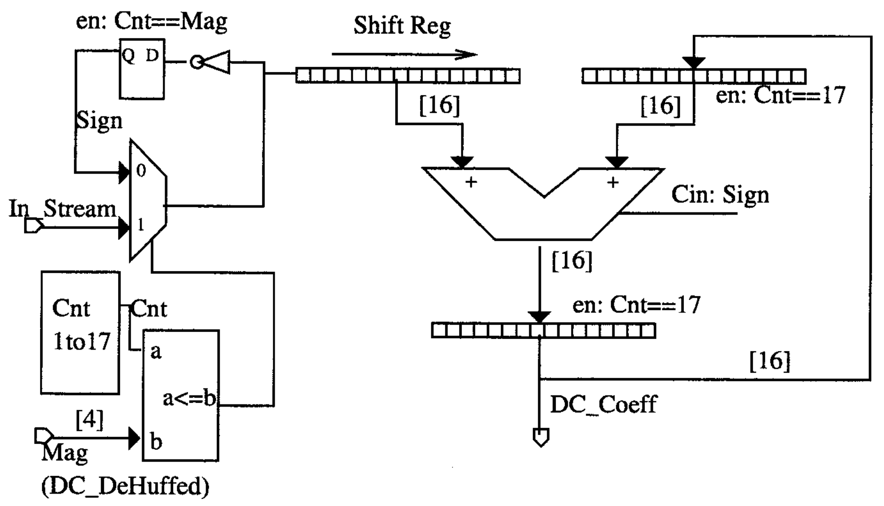


FIGURE 4.7. Simplified block diagram for decoding JPEG DC coefficients.

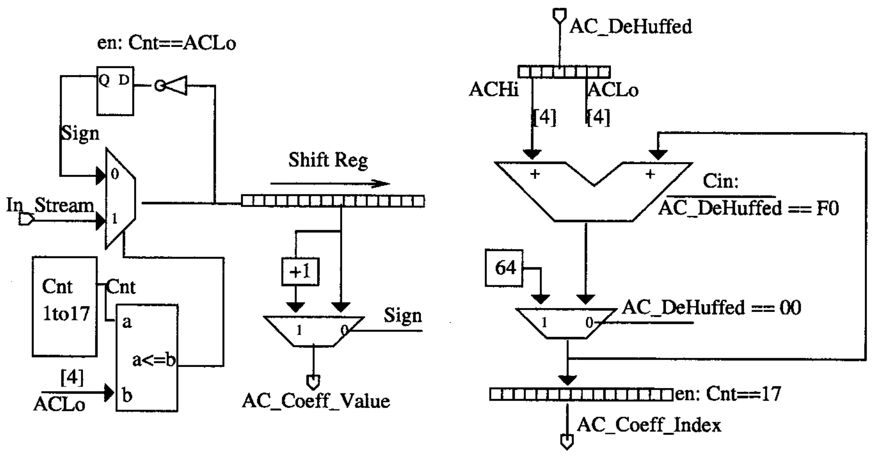


FIGURE 4.8. Simplified block diagram for decoding JPEG AC coefficients.

4.6. An Efficient Algorithm for Approximating the Integral Image

We have developed an algorithm for calculating an approximate integral image that needs significantly fewer computations and hardware resources than the inverse DCT. The basic idea is to compute the integral image exactly at some points in the image and then approximate it everywhere else by interpolation. The JPEG compression algorithm partitions a grayscale image into non-overlapping 8x8 pixel blocks and computes the 64 DCT coefficients for each block. These coefficients can be obtained from the JPEG data by Huffman decoding and dequantization [**11**]. Since the DC coefficient of a block encodes the average pixel value of that block [**17**], the sum of all pixels in an 8x8 block can be calculated from its DC coefficient alone. Using all such local 8x8 block sums of an image, it is possible to compute the exact value of the integral image at the bottom-right corner of every 8x8 block in the image, as shown by the example in figure 4.9(a). Suppose that S_1, S_2, S_3 and S_4 are the 8x8 block sums for the four blocks shown in figure 4.9(a). The exact value of the integral image at point A is S_1, at point B is $S_1 + S_2$, at point C is $S_1 + S_3$, and at point D is $S_1 + S_2 + S_3 + S_4$. The rest of the integral image can then be filled in by interpolating these exact values, but the resulting approximate integral image may have a large error compared to the true integral image.

The approximation error can be reduced, at a greater computational expense, if we divide up each 8x8 block into four 4x4 blocks and calculate all the 4x4 block sums from the DCT coefficients. Then the exact integral image value can be obtained in a similar manner as above at four times more points than before, as shown in figure 4.9(b). Reducing the error further by computing the exact integral image values even more densely further diminishes the benefits of avoiding the inverse DCT. (Taken to the extreme, reducing the error to zero by computing the exact integral image values everywhere becomes roughly equivalent to the inverse DCT.) The amount of approximation error which can be tolerated in the integral image should be determined by how the error affects the detection accuracy of the people detector. As the results in [**12**] show, the approximate integral image computed from 4x4 block sums provides a reasonable balance of high people detection accuracy and low computational effort, and is the approximation level that was used.

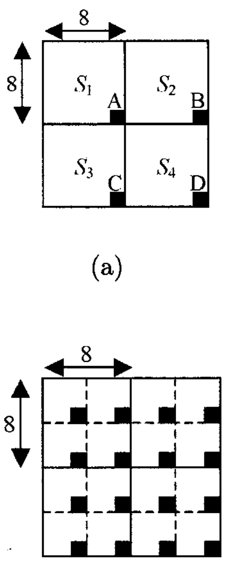


FIGURE 4.9. A, B, C and D in (a) can be calculated directly from the DCT DC coefficients, but more coefficients are needed if the desired resolution is increased (b).

Given an 8x8 block of DCT coefficients, how can the 8x8 and 4x4 block sums be calculated for that block? If we consider the DCT coefficients in the block to be a 64-dimensional vector **d**, and the corresponding 8x8 block of pixels to be a 64-dimensional vector **p**, then we can write $\mathbf{p} = A\mathbf{d}$, where A is the constant 64x64 matrix representing the inverse DCT. Computing the sum of an arbitrary set of pixels within an 8x8 block is equivalent to taking the dot product of **p** with a 64-D vector **i** whose components corresponding to the pixels included in the sum are 1 and all other components are 0. So the sum S of the pixels can be written as $S = \mathbf{i}^T\mathbf{p} = \mathbf{i}^T A\mathbf{d} = \mathbf{r}^T\mathbf{d}$. For a given **i** (i.e. a given set of pixels to add up in an 8x8 block), **r** is a constant vector that can be pre-computed independently of **d**.

For example, to find the sum of all pixels in any 8x8 block, we set all components of **i** to 1 and then compute $\mathbf{r}^T = \mathbf{i}^T A$. The components of the resulting **r** turn out to be all zeros except for the one that multiplies the DC coefficient, which has a value of 8. This means that the sum of an 8x8 block can be computed by multiplying the block's DC coefficient by 8. Once **d** is computed for a particular 8x8 DCT block, the pixel sum is given by the dot product of **r** and **d**. Note that the number of additions and multiplications needed to compute the dot product of **r** with any **d** is equal to the number of non-zero components of **r**.

To find the four exact integral image values in an 8x8 block, the sums of the shaded pixels denoted by R_1, R_2, R_3 and R_4 in figure 4.10 are needed. The **r** vectors for these sums contain respectively 25, 5, 5, and 1 nonzero components.

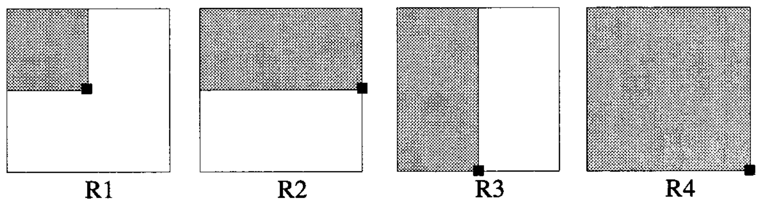


FIGURE 4.10. Calculating the exact half-block integrals (4x4 resolution) involves computing the pixel sums for the shaded regions in an 8x8 block.

4.6.1. Interpolating From Exact Integral Image Values. Once the exact integral image values are obtained, the rest of the image is filled in by interpolation. There are many different types of interpolation methods that can be used here, but to keep the computation hardware-friendly, we assume that the integral image values are approximately linear within a 4x4 neighborhood and use simple local linear interpolation. This is equivalent to assuming that the pixel values in a 4x4 neighborhood of the grayscale image are equal, since the derivation of a straight line gives its slope, which is constant.

The interpolation can be done in two steps: initially the integral image consists of 5x5 neighborhoods of the kind shown in figure 4.11(a). The black squares are the points where the integral image values have already been computed and the white squares are the missing points. In the first step, the gray squares in figure 4.11(a) are obtained using the equations shown there. The four gray squares along the border of the 5x5 neighborhood are computed by averaging the two nearest black squares, and the middle gray square is computed by averaging all four black squares.

After the first interpolation step, the integral image consists of 3x3 neighborhoods of the kind shown in figure 4.11(b). The procedure for filling in the remaining missing points in the 3x3 neighborhood is analogous to that of the 5x5 neighborhood, as shown by the equations in figure 4.11(b). A valid integral image must be non-decreasing (since grayscale pixels are never negative), and it can be easily shown that the interpolated values computed using the equations in figure 4.11 satisfy the non-decreasing requirement, provided that the exact integral image values satisfy them.

The interpolation scheme is suitable for hardware implementation because it only requires additions and divisions by 2 and 4, which can be done with shifts. The total work needed to fill in an 8x8 block is 27 multiplications, 64 additions and 20 shifts. On the other hand, the inverse DCT is an $O(nlog_2n)$ algorithm, so it requires $64 * log_2 64 = 384$ multiplications and additions for an 8x8 block. Basically, the savings in computational effort and hardware resources come from replacing the multiplications in the inverse DCT algorithm with shift operations.

The algorithm for computing the approximate integral image is related to the idea of decompressing a JPEG image by "scaled decoding". Scaled decoding is a feature of the JPEG format that allows efficient decompression of an image at either 1/2, 1/4 or 1/8 of its original resolution. The algorithm is equivalent to first computing a grayscale image by scaled decoding at a lower resolution, but still maintaining the same dimensions as the original image by filling in the missing pixels with replicas.

This lower resolution grayscale image is then integrated to obtain an approximate integral image.

Figure 4.12 shows the result of "differentiating" integral images computed at different approximation levels. It illustrates the effect of increasing the approximation error of an integral image on its corresponding grayscale image. Figure 4.12(a) shows the grayscale image corresponding to an integral image computed without any approximation. As the resolution decreases, the approximation error increases, thus making the grayscale image increasingly blurry. This is because the effect of linear integral image interpolation is to replace pixel neighborhoods in the grayscale image by the neighborhood average. As the resolution decreases, the size of this neighborhood increases, which is why the grayscale image becomes more blurry with decreasing resolution.

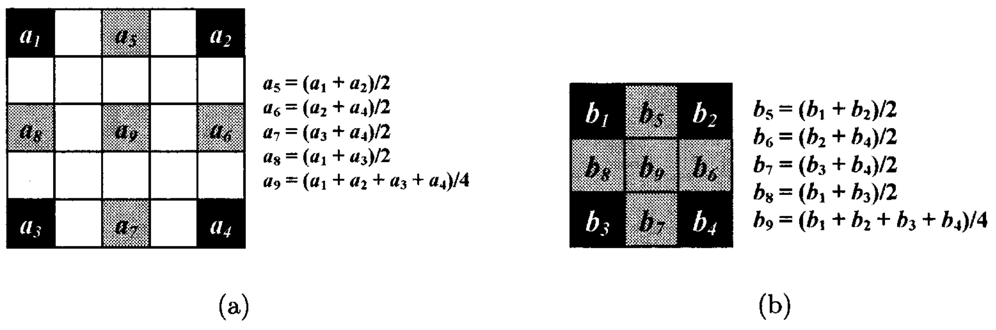
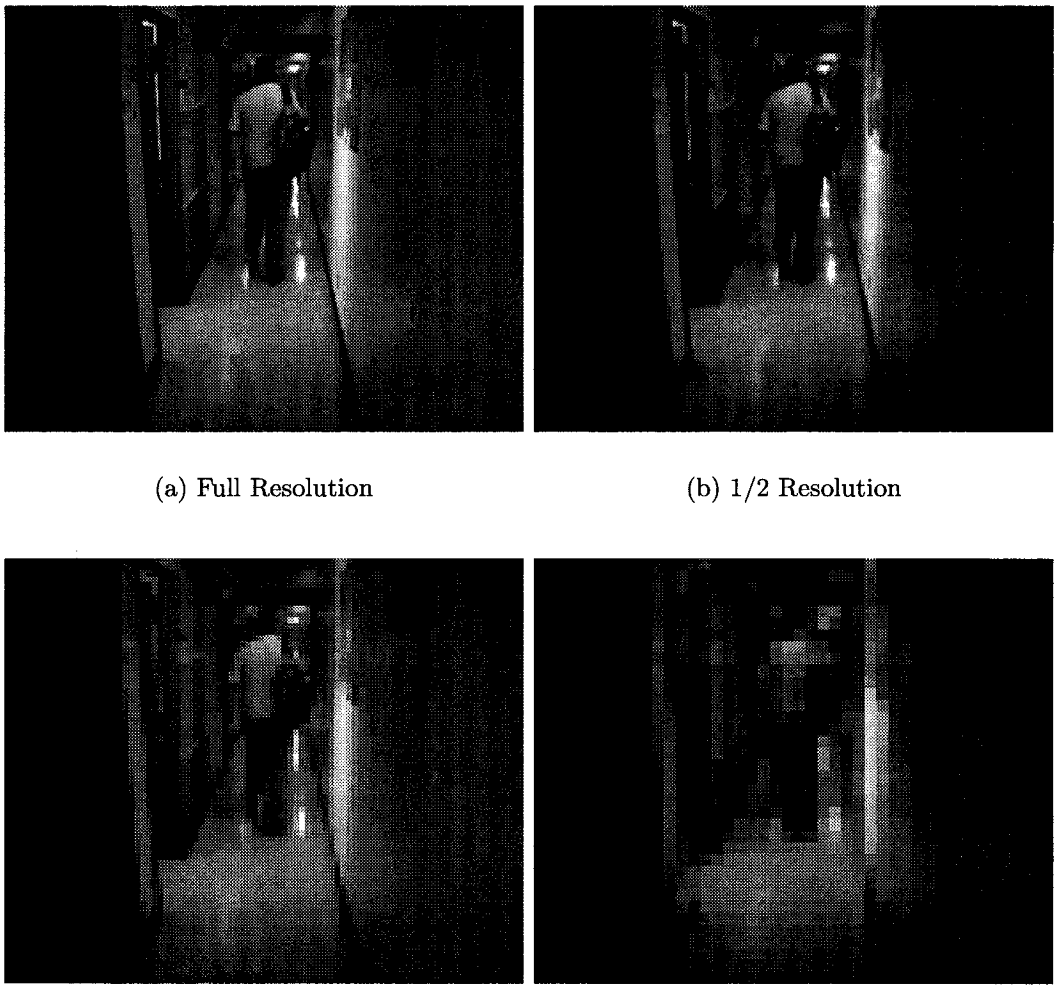


FIGURE 4.11. Integral image interpolation scheme for (a) 5x5 and (b) 3x3 neighborhoods.

4.7. Implementation of the Approximate Integral Image Computation

The algorithm described in section 4.5 uses sums of DCT coefficients multiplied by a constant to exactly calculate the half-block integral points. Since the coefficients are fed sequentially to the module by the JPEG coefficient extractor, this can



(a) Full Resolution

(b) 1/2 Resolution

(c) 1/4 Resolution

(d) 1/8th Resolution

FIGURE 4.12. Result of differentiating integral images calculated with varying resolutions. The lower resolutions require fewer DCT coefficients.

be implemented using a multiply and accumulate (MAC) circuit for each point, as illustrated in the simplified block diagram in figure 4.13. Careful inspection of the coefficient multipliers shows that their values are dependent on the index of the multiplied DCT coefficient rather than the position in the position of the point that it is being accumulated for. This suggests that a single multiplier can be shared by all the points. The MAC circuit also intrinsically takes advantage of zero runs in the

JPEG stream, since not accumulating these coefficients is the same as accumulating 0. This means that the time needed to calculate a point is dependent on the number of non-zero coefficients in the image.

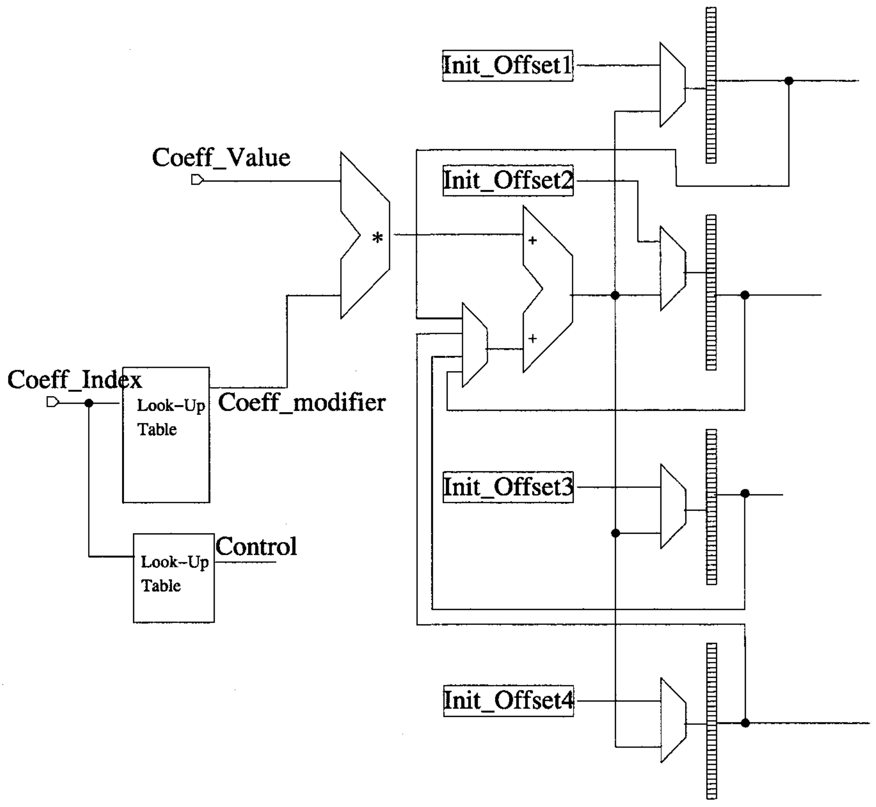


FIGURE 4.13. Simplified block diagram of calculation of half-block integral points.

However, JPEG DCT coefficients only contain information about the 8x8 block that they are in, and are totally independent of their position in the image. An integral image point, on the contrary, is dependent on all the points above and to its left in the image. It is therefore necessary to offset each block-integral point extracted from the DCT coefficients by the integral image as it has been accumulated so far. Referring to figure 4.14, where the grayed-out portions are the blocks that have already been received, and the the black squares are the half-block integral points that are extracted directly from the DCT coefficients, it is evident that going from the block-integral points that are extracted from the coefficients to the final image-integral points requires points from the blocks immediately above and to the left of the current block. For any given 8x8 block decoded from JPEG, the desired image-integral points, $(r+3, c+3)_i$, $(r+7, c+3)_i$, $(r+3, c+7)_i$, $(r+7, c+7)_i$, where $(r, c)_i$ is the position of the upper-left pixel of the block in the image, can be calculated from the block-integral points $(3,3)_b$, $(7,3)_b$, $(3,7)_b$, $(7,7)_b$, with the origin at $(0,0)_b$ in the upper-left corner of the block, according to equation 4.1.

$$(r+i, c+i)_i = (i,j)_b + (r-1, c+j)_i + (r+i, c-1)_i - (r-1, c-1)_i \qquad (4.1)$$

This dependence on previous points requires the use of some form of memory. Although it would be possible to refer to the image stored off-chip, this would incur significant delays, making on-chip caching preferable. Since JPEG blocks are read from left to right and top to bottom, it is only necessary to keep the integral points from a single row of the image, in addition to the final column of the previous block.

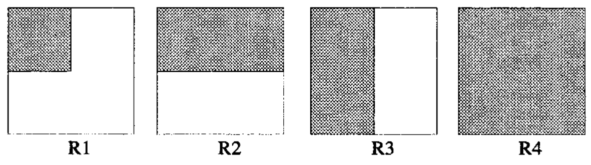


FIGURE 4.14. Illustration of integral image being constructed from 8x8 blocks. The grayed-out portions are the blocks for which the integral image has already been calculated, and the black squares are the points which are extracted exactly from the DCT coefficients.

The astute reader will notice that no mention has yet been made of the quantization factor required by JPEG decompression. When a JPEG image is encoded, each DCT coefficient is divided by a quantization factor chosen according to its index in the block. The reason that this is not dealt with by the coefficient extractor is that, similarly to the coefficient modifiers, the quantization factor is fixed for a particular index value, and is known in advance, which means that the two multiplicative factors can be combined offline, so that only one multiplication is required online.

In an effort to minimize the memory taken up by the various tables, it is necessary to optimize the number of bits that need to be stored. A close study of the JPEG standard reveals that 14 bits are needed to store all possible coefficient values. Consequently, the largest useful quantization factor also requires 14 bits. Since the coefficient modifiers' absolute values are all less than 3 (excluding the DC modifiers, which are powers of 2, and can be taken care of with shifts), 3 bits are required to store the modifiers' whole parts in 2's complement notation. The number of bits reserved for the fractional part will increase precision, but will not otherwise limit the range, and is therefore temporarily left undefined. The combined "quantized" modifier consequently requires 17 bits to represent its whole part. Since multiplication of the quantized coefficient by the quantization factor simply restores the original 14 bit coefficient, the final result should also fit in 17 bits for a properly encoded image. These values are then accumulated to give a 32 bit integer, the value of the integral image at that point. Since the result is expected to be an integer, the fractional part is only useful in intermediate results, and rounding off to the closest integer according to the MSb of the fractional part should be enough to correct for any lack of precision in intermediate calculations, as long as the accumulated error in the block is under 0.5.

Calculating the integral image directly from the JPEG coefficients has the obvious advantage of eliminating the need for an explicit integrator. In fact, calculating the integral image directly is equivalent to decompressing the image. One might wonder why linear interpolation is used instead of simply storing a smaller image, since the images are essentially equivalent. Although a high resolution image is not required by this algorithm to detect people, the features will be misaligned at large scales unless they are placed at what is essentially sub-pixel resolution at small scales. The method that was chosen to achieve this was to duplicate pixels to allow more precise placement of features. Although this could have been achieved by fully decompressing a smaller image, it was evaluated that the bottleneck was more likely to be in storing the image to memory rather than in receiving the compressed data. A trade-off can be achieved between the size of the input stream and the complexity of the on-chip decompresser. This is due to the observation that JPEG decompression does not scale linearly with the resolution. While a full resolution decompression would require 64 accumulators, one for each pixel in the block, a 1/4 resolution scan only requires 4 accumulators, or $1/16^{th}$ of that needed for the full resolution. To give a feel for the amount of resources saved by this method, the module calculating the 4 exact points takes up 400 slices in a Virtex-II FPGA (each slice contains 2 flip-flops and 2 four-input lookup tables). The modules approximating the remaining 60 points take up collectively less than 100 slices. Even limiting estimates to the storage space required for the DCT coefficients' accumulators, calculating the exact values of the 60 remaining integral image points would require more than take 960 slices. This would have severe impacts on both placement and routing efforts for the entire module, possibly resulting in a reduced minimum period.

4.8. Putting It All Together

Once the various modules have been designed, a method still needs to be chosen to allow them to communicate. In an attempt to maximize flexibility and code re-use, a single, common, interface protocol was required for all the modules so that they could be swapped in and out easily without affecting adjoining modules. And, of course, this must be achieved while minimizing the impact on system performance.

The transport layer chosen to satisfy these various constraints was the Fast Simplex Link (FSL) unidirectional point-to-point bus protocol already used by Xilinx for communication with its MicroBlaze processor. This protocol essentially boils down to a 33-bit wide (32 bits of data and 1 of control) First In First Out (FIFO) buffer with all of the traditionally associated synchronization flags. Using this protocol has many advantages, namely that any module using it can be plugged directly into the processor, is very low overhead, and most of the more complex protocol already use FIFOs for synchronization of the different modules, making it a simple matter to adapt them to simulate the FSL bus on one end. Figure 4.15 shows the layout of the different modules in the data path, as well as the flow of data through them. The JPEG stream is received by the MicroBlaze and sent to the JPEG decoding module after the header has been stripped off. This stream is decoded, and the non-zero JPEG coefficients are sent to the integral image module. The integral image points are then sent to the memory controller through the person detection module. Once the integral image is received, the person detection algorithm is started, and the size and coordinates of the bounding boxes for any positive matches are sent back to the MicroBlaze. Once the image has been scanned at all scales, the MicroBlaze is informed so that it can start the process over.

As mentioned previously, and given in more detail in [**21**], the FSL bus is 33 bits wide. This is split into 32 bits of data and 1 bit indicating whether the word is "data" or "command". Following is a description of the data and command formats expected by each of the major modules. All numbers are in bit-wise little-endian format.

4.8.1. JPEG/Integral Image Decoding Module.

4.8.1.1. *Incoming Command Format.* The JPEG/Integral Image module does not have any commands to receive, so any command word received on its input is simply forwarded as-is to its output.

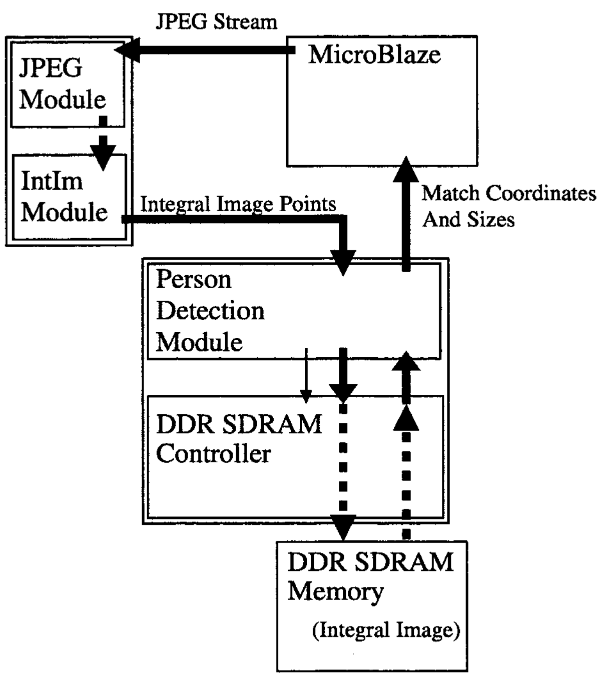


FIGURE 4.15. Flow of data through functional modules – bold arrows are FSL bus connections, thin arrows are single control lines, and bold dashed arrows are module-specific interconnections.

4.8.1.2. *Incoming Data Format.* Each incoming data word is a 32-bit slice of the JPEG data stream. To correctly order the data, one can imagine the JPEG data being fed into a 32-bit shift register, with the next bit being fed into bit 0 of the register. Every 32 bits, the register would then be loaded out in parallel into the FSL bus. If the JPEG stream is not a multiple of 32 bits, then the final word should be padded with zeros in the least significant bits.

4.8.1.3. *Outgoing Command and Data Format.* The command words sent by the JPEG/Integral Image module give the target memory address of the integral image point sent in the subsequent data word in accordance with the format expected by the DDR SDRAM controller module, as described in section 4.8.3.2, with the exception that the memory flushing extension is not implemented.

The address in the command word should be immediately followed by the data word to be written to that address. Each data word output by the JPEG/Integral

Image module is a 32 bit point in the integral image, output in the order they are generated. This order is selected to minimize the latency with which points are sent to memory. The upper-left point in the image is stored in memory address 0. The mapping for integral image coordinates to memory coordinates are as follows: $(x, y)_i = (2x, y)_m = (c, r)_m$, where $(x, y)_i$ are the image coordinates starting at $(0, 0)$ in the upper left, and $(c, r)_m$ are the memory coordinates. c maps directly into the memory as the column address, while r is the concatenation of row and bank memory addresses.

4.8.2. Person Detection Module.

4.8.2.1. *Incoming Data and Command Formats.* In general, the person detection module forwards all commands and data that it receives to the memory through its FSL output. The only exception to this is that it watches for the end of the image transfer by examining if bit 28 of a command word is '1'. Apart from this, the commands and data should be formatted according to section 4.8.3.2.

4.8.2.2. *Outgoing Command Format.* Outgoing commands from the person detection module are used to send status information. For the moment, the only bits used are bits 31 to indicate a problem with the internal digital clock managers (DCM), and bit 30 to indicate that the image has been scanned at all scales.

4.8.2.3. *Outgoing Data Format.* Each data word output by the Person Detection module represents a single match, giving the row and column of its upper left point, along with the scale, which is the number by which the 16x48 window was multiplied, thereby giving the match's size. The data word is formatted as shown in figure 4.16. The scale is in scaled integer format with an implied 2^{-3} multiplier. This means that the the width of the match window is equal to $scale * 16 * 2^{-3}$, and the height is equal to $scale * 48 * 2^{-3}$.

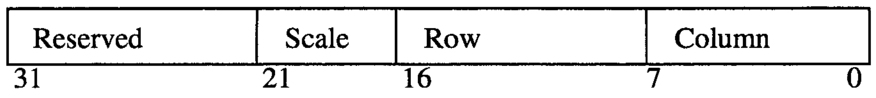


FIGURE 4.16. Format of outgoing data words from the person detection module.

4.8.3. DDR SDRAM Controller Module.

4.8.3.1. *Protocol Extension.* The design of the DDR SDRAM controller uses FIFOs for synchronization purposes. This inserts a non-negligible latency between the time that a read or write command is sent to the controller and the time that the command is sent to memory. Given the long delays incurred by going to memory, it is desirable that as few unnecessary memory accesses as possible be executed. However, in order to minimize overall latencies, the person detection module sends out multiple requests for point values that may turn out to be superfluous if the prediction module finds an early match. A method for flushing the memory of undesired requests was therefore added to the memory controller. This option is not compliant with the FSL protocol, however, and breaks across timing domains, requiring careful consideration before being used.

A flush request is issued by asserting the 'flush_req' input. This input bypasses the FSL bus. A command word (see 4.8.3.2) must then be written to the FSL bus to execute the terminate flush extended command by asserting bits 31 to 29. This is to separate the words that need to be flushed from the valid words that follow. Since there might be some data words pending in the outgoing FSL, an outgoing command with bit 30 set to 1 (see 4.8.3.4) is pushed into the queue following all the invalid data to be flushed.

4.8.3.2. *Incoming Command Format.* Incoming commands must be formatted as described in figure 4.17. The fields have the following meanings:

- Cmd: Main command bits, which can take the following values:
  - "00": no-op

  - "01": read word
  - "10": write word
  - "11": extended command
- Ext Cmd: Extended Command space. Only valid if Cmd is "11". Formatted in the following manner:
  - bit 29: terminate flush
  - bit 28: transfer complete
  - bits 27 to 24: reserved
- Row: Row address of target word in memory
- Bank: Bank address of target word in memory
- Column: Column address of target word in memory

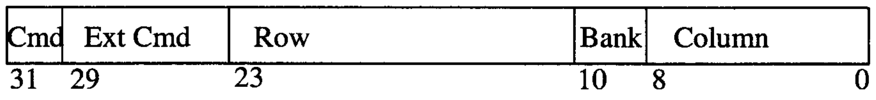


FIGURE 4.17. Format of incoming command words to the DDR SDRAM memory controller module.

4.8.3.3. *Incoming Data Format.* Any incoming data word needs to be immediately preceded by a write command to provide the destination address in memory. Although the memory uses 16 bit words, the controller uses burst lengths of 2 to always retrieve 32 bits at a time. Because of this, the memory should be read in a word-aligned manner, exclusively using either odd or even addresses.

4.8.3.4. *Outgoing Command Format.* Outgoing commands from the DDR SDRAM controller module are used to send status information. For the moment, the only bits used are bits 31 to indicate a problem with the internal digital clock managers (DCM), and bit 30 to indicate that the flush command has been completed and that following output data words come from read requests issued after the flush.

4.8.3.5. *Outgoing Data Format.* Outgoing data from the DDR SDRAM memory controller module is simply a sequence of 32-bit data words output in the same order as the read requests were received.

4.9. Result Analysis

Given the primitive interfaces available for board-computer communications in the prototyping setup, extensive tests could not be performed on the physical implementation of this algorithm. However, a detailed analysis of the algorithms accuracy was extracted from the full software implementation, and can be found in [**12**]. Given the functional equivalence between these two approaches, the analysis found therein should hold true for the hardware implementation as well. Exact timing numbers are equally difficult to extract from the physical chip. However, synthesis tools are very proficient at estimating the internal delays with greater precision than might even be achieved through direct measurement. Combining these estimates with functional models and extrapolating using the statistical distributions observed in the software implementation, it is possible to get an accurate measure of what the system's average performance should be under various circumstances.

Since the design was made in a fully synchronous manner, it should be sufficient to know whether or not all timing requirements have been met to know whether the functional simulation is an accurate reflection of the actual implementation. This allows us to use a functional model of the memory provided with the memory controller to get a cycle-accurate functional simulation. This is necessary since the exact time taken in retrieving data from a given address depends on many memory dependent factors. Using an accurate functional module for the memory allows these delays to be taken into account in functional simulations, and thus allows accurate timing estimates to be extracted from these simulations. The average memory access time was calculated by observing a large number of memory accesses during the person detection module's normal operation. The mean access time per point was then calculated by dividing the total number of points fetched from memory by the total time that it took to fetch these points. This spreads the memory overhead over the entire operation, giving a more accurate estimate than using local measures. It is important to note that memory throughput could be further optimized by customizing the memory controller to take advantage of natural pauses in memory accesses to refresh or activate the memory banks in view of future accesses. This would have the effect of lowering the average memory access time, thereby increasing overall performance.

The creation and storage of the integral image to memory for a 352x288 image, cropped to 216x288 by the hardware, is approximately 30 ms. This is governed by the number of points written to memory, which is fixed from one image to the next, and therefore should be relatively constant. Evaluation of the average frame rate of the detector is complicated by the fact that the number of points that need to be calculated varies according to the number of near-people windows in the image. Using a sample space of 981 frames containing people, it is found that on average 40.43 features are required in each of the 3079 windows of an image. Assuming an average of 7 points per feature, this means that there are approximately 870000 points evaluated in an average frame. Given a memory access time hovering around 350ns per point, it can be estimated that frames containing people can be treated at the rate of approximately 1 frame every 0.3 seconds. In comparison, treating an image that has very few false positives can take as little as 25ms once the integral image has been written to memory.

Of course, this is assuming that the memory controller is able to run at 100 MHz. Although this should be possible, recent versions of the synthesis tools have unexpectedly been unable to meet the timing constraints at such speeds. This forces the use of the on-chip Digital Clock Managers (DCM) to synthesize a slower clock. Although the DCMs allow synthesis of most rational number multiples of the input clock, 75MHz is sufficiently slow and reduces complications in crossing clock domains. With the design running at this speed, the frame rate with a subject in the image should drop from around 3 fps to slightly under 2.5 fps. This constraint makes running the design at full speed less of an issue. This turns out to be quite useful, as the synthesis tools were not quite able to meet the 10 ns period requirement, even using the highest effort level for placement and routing, due to some paths of excessive length in the MicroBlaze processor. While careful floorplanning should make it possible to run the design at 100MHz, the memory controller speed limitations make this not worth the effort, especially considering that it would be significantly more trouble.

Synthesizing the design without the MicroBlaze processor (leaving only the framework's ASFW) reveals that the 100MHz constraint could be satisfied if the MicroBlaze were replaced by an off-chip processor. Given that the place and route tools abandon their search for greater performance once the requirements are met, it is possible that this design could run slightly faster still, but the trouble that the tools had in achieving even this level of performance, hint that this would probably not be a significant gain. The fact that system performance is limited by memory bandwidth makes any possible speed optimizations unnecessary.

The place and route reports also show that the system has some space remaining for extra logic, with the design only taking up 3438, or 67%, of all slices in the Virtex-II 2V1000 chip. In fact, excluding the MicroBlaze, the full system only takes 2233, or 43%, of the available space. A more significant difference is in the usage of Block SelectRAMs, and hardware multipliers. Although the ASFW only uses 7 blocks of memory and 9 multipliers, the MicroBlaze requires an extra 18 blocks of memory, and an extra 3 multipliers. This means that while the ASFW accounts for 65% of the logic used by the entire system, the MicroBlaze accounts for 72% of the memory used.

4.10. Recommended Design Flows

Although the most concrete product of this project was the node's design, much of the work that was done was in evaluating the various tools, design flows, and methods available for node design. Following are the impressions and recommendations that came from working with the various tools.

Embedded designs are complex systems requiring that attention be given to a large number of different abstraction levels, from low level hardware design all the way up to high level software programming. Consequently, designing an embedded system of any but the lowest complexity cannot be accomplished inside a reasonable time frame without the support of appropriate tools.

While most of the necessary tools are available commercially, some are not quite sufficient to the task, and the creation of some custom tools was required. This section therefore reviews both the commercial tools that were used, and presents the custom tools, explaining what need they fill and giving an overview of their structure to facilitate in their maintenance.

Finally, a recommended design flow is presented in an effort to provide a starting point for future projects.

4.10.1. Commercial Tools.

4.10.1.1. *Embedded Design Kit.* The Embedded Design Kit (EDK) is a toolkit provided by Xilinx to help in the design of embedded systems incorporating MicroBlaze or PowerPC processors. It allows the system designer to combine module integration, system synthesis, and software integration into a single tool flow. In keeping with the Xilinx tool flow philosophy, each of the steps in the flow can be executed from the command-line, allowing the designer to use his own scripting tools instead of the Graphic User Interface (GUI) if so desired. Although the most recent version of the toolkit (EDK 6.1) is approaching maturity, it still requires some manual intervention from the user.

The EDK flow relies on a series of configuration files to determine how the system should be built. These files can either be automatically or manually generated and edited, and a full description of their functions and formats can be found in [**20**] along with a description of the various utilities available for the generation of these files. All of these tools can be centrally launched from the Xilinx Platform Studio (XPS) program, which is the GUI interface for the EDK.

Generally, a designer will start out by designing the custom hardware to implement the system's main functionality. The hardware should then be packaged as a peripheral to be included in an EDK project. The packaging of "custom IP cores", as Xilinx calls them, can be done manually or through the use of the "Import Peripheral Wizard", accessible from XPS, or by manually generating the files described in the Platform Specification Format (PSF) [**20**].

The EDK facilitates the incorporation of these user peripherals with the MicroBlaze (Virtex-II) or embedded PowerPC (Virtex-II Pro) microprocessors. The software for these processors is written in standard C, and uses the GNU software flow for compilation. Xilinx provides MicroBlaze and PowerPC adaptations of the gcc compiler, linker and debugger programs. Program compilation can be done through the use of Makefiles or using the XPS user interface. The output executable can then be integrated using the 'data2bram' tool. The XPS interface provides a very user friendly option for system generation, allowing the user to go from system building, through software integration and debugging, to downloading to the FPGA in a seamless flow.

More importantly though, it generates the configuration files that can be used for batch processing if the user wishes to switch over.

One important clarification must be made with respect to peripherals. The "Load Path" section of the "Platform Specification Format" chapter of [**20**] describes the directory structure that is required for XPS to automatically recognize custom peripherals and allow the user to integrate them into a design. Setting the "library path" option is done through the "Project Options" dialog. However, it is unclear how one is supposed to change the "Library Name" option, which seems to be set by default to "my_periphs". Given that the words "my_periphs" seem to not appear anywhere in Xilinx's documentation, it was felt that this should be explicitly noted.

Apart from this fact, the EDK has been found to be a very well designed tool, which suffered from multiple format changes in its early versions, but seems to have stabilized. It greatly simplifies system design by allowing integration of the various steps into a single design flow, and by automating the more tedious aspects of system design.

4.10.1.2. *Handel-C.* Handel-C is a high level hardware description that uses the 'C' programming language as its base, and adds certain constructs to allow the description of parallel processes. The two main additions are the "par" construct and the "chan" data type. A "par" construct groups together statements – such as basic computations, function or macro calls, or even other "par" statements – that should execute in parallel one with respect to the others. A "chan" variable is used for communication and synchronization between two parallel pieces of code. When a segment writes to a channel, execution of this segment is halted until the message is read at the other end. Although there are more extensions to the language, these are the two principle ones for hardware support.

The advantages of such an approach are evident. As with all higher level languages (HLL), Handel-C allows a higher level of abstraction, which leads to the possibility of designing more complex systems in a shorter time frame. In fact, in situations where resource use and circuit performance are not an issue, Handel-C and similar HLLs win hands down over lower level hardware description languages (HDL).

Some comparative designs were done in both Handel-C and VHDL to gauge the suitability of Handel-C to the system's design. While these tests were admittedly quite incomplete, and severely biased in favor of VHDL by the experimenter's greater experience in this language, the results showed consistently better performance and resource use by VHDL designs. It can be expected that an experienced Handel-C programmer would know how to tailor his statements to produce the desired hardware and narrow the gap, but this could be expected to result in what HDL coders call "pushing the rope". This expression describes the phenomenon whereby a hardware designer has a clear idea of the construct that is wanted, but is unable to transmit this behaviorally to the hardware synthesis program. The only ways around this problem is to find the exact wording which will cause the synthesis tools to implement the correct construct, or to use direct structural instantiation. The first solution has a tendency to create solutions which are not only tool-dependent, but often version-dependent, since modifications to the tools' inference engines can change how a piece of code is interpreted. The structural instantiation of components, however, tends to be device-dependent. Neither of these solutions is ideal, but one can expect that the extra level of abstraction in HLLs can only exacerbate the problem.

This is not to say that HLLs should be discounted. As HLL compilers improve, the gap with HDL coded designs will inevitably narrow. For large, complex designs, the possibly small cost in area and performance from an HLL design can be more than worth the reduced development cost. This is especially given that a wide variety of algorithms will be required for the PDCA to achieve its full potential. What will remain true of hardware designs for some time yet, is that independently of the language used, it is essential for the designer to have a clear idea of how the final design should look.

4.10.2. Custom Tools.

4.10.2.1. *Testbench Automation Tools.* Testing is always one of the most time-consuming portions of any design. In hardware design, testing is generally a multi-step process, with a synthesized design being verified as functionally correct in simulation before it is implemented on-chip. Simulation testing of hardware is done through the use of testbenches, which generate the type of inputs the device is expected to see and compare the outputs to what is expected. The low level nature of hardware means that testbenches are by their nature rather primitive, which makes verification of complex functions such as JPEG decompression an arduous task.

Consequently, Perl scripts were written to partially automate testbench creation. These scripts read in the VHDL source file for which the testbench should be generated and output a testbench skeleton which can read in a vector file in a standard format, and output a record of the results.

Of course, this simply transfers the task's complexity to the generation of the test vector and expected results files. For this reason, Perl modules were written to help in the creation of these files from readily available sources. For example, the JPEG module was verified by inputing JPEG data in vector format and comparing the module's output to the decompressed image corresponding to that data.

While the overhead of designing these tools would not be justified for limited use, it can be expected that most computer vision tasks will require complex test data, and that having these tools will allow more complete pre-implementation testing by reducing the time required in organizing such data.

4.10.2.2. *Memory Management Tools.* One of the most precious resources when implementing image processing algorithms on an FPGA is the device's on-chip memory. Images require large amounts of storage, and many transformations can be most efficiently implemented using large look-up tables. And of course, as is the case for any computer system, the farther the data is, the longer it takes to fetch, which means that data should whenever possible be stored in the FPGA's on-chip dedicated memory. However, the distance–delay relation has an additional meaning on FPGAs than in more conventional computer architectures. Even on-chip, routing delays from memory to logic can become non-negligible, reducing overall system performance, unless care is taken to have the memory positioned close to where its data is used. The organization of on-chip memory into distinct blocks of restricted widths and depths cannot be ignored. Optimal system performance can only be achieved by carefully considering exactly how various data structures are stored into memory.

For example, Xilinx's Virtex-II$^{\text{TM}}$family of chips, which was used in the prototyping environment, has 18kb dual-port memories that can be configured for access in words of 1, 2, 4, 9, 18, or 36 bits. If a look-up table consists of three 3-bit data elements, one has the choice between concatenating these into a single 9-bit word, which can be separated at no cost in hardware, or of storing them in three separate blocks, causing additional routing delays and sub-optimal memory use. The better choice in terms of system performance is obviously in combining them into a single word.

However, the commercially available tools do not provide an easy way to convert from the logical mapping (three 3-bit words) to the physical mapping (one 9-bit word). It was therefore necessary to code my own tools for doing this. This was once again done using Perl modules to improve code reusability.

4.10.3. Design Flow Recommendations. The recommendations that follow stem from my experiences with the various tools in the course of this project.

It goes without saying that the first step to a successful design is a thorough conceptual design stage, in which the particulars of the system are established before any tools are involved. However, once this has been accomplished, there are multiple paths that could be taken in pursuit of a fully functional implementation of the design.

Part of the conceptual design phase is in determining what the individual functional units are, and their interfaces. This will allow the proper partitioning between software and hardware, as well as allow one to determine how partial reconfiguration might be implemented.

Although no HLLs were used in the design of the current system, they should not be discounted. HLLs can be a powerful tool in the initial conceptualization and design of the system. A "first draft" in an HLL not only allows the design to crystallize in the designer's mind, but also provides a rapid first attempt at the design. If performance and area targets are achieved, then the design is done. Otherwise, it provides a comparison point for the final design. Although it is possible, it is unlikely that both systems will suffer from the same defects. Since the HLL-built system should be easier to build and debug, it should be able to provide a functional reference for the final design.

Irrespectively of whether the design is done with HDLs or HLLs, designing complex systems in a modular manner always simplifies design and verification. By thoroughly testing individual modules before combining them to form the system, one limits the depth that needs to be covered in chasing bugs, and limits the repercussions of fixing them. This implies that interfaces between modules should be defined as well as possible before a module is coded. Of course, some issues may not have been considered ahead of time, causing a change in the interface, but this should be kept to a minimum. Using a modular approach to design also encourages code reuse between projects.

Using a modular approach and having partitioned hardware from software, the hardware modules should be fully implemented and thoroughly tested using simulation tools and complete testbenches to verify that the code is functionally sound. The hardware modules should then be synthesized using standard FPGA synthesis tool flows (for example Xilinx's ISE toolflow). To ensure that the design will function correctly, it is essential to set the proper timing constraints (of particular use for this is the PERIOD constraint [**18**], which specifies the target clock period). If a properly synchronous design meets timing constraints, it is almost guaranteed to correspond to the functional simulations. One possible exception to this is in designs with multiple clock domains. In such cases, the simulator might need timing information to know the relationship between the clocks and the signals they generate. In such cases, it is possible to generate post-map and post-place-and-route simulation models to ensure that the design works. Another possible exception to this is with designs that communicate with separate devices. Given that the synthesis engine cannot know the characteristics of these signals, it cannot take them into account in its design. To help in this effect, one can use the OFFSET constraints [**18**] on input and output nets.

The phase in which software implementation begins depends in large part on the role of the software in the overall program flow. Minimizing software and hardware interaction allows the two design phases to be decoupled, and simplifies debugging of both parts. Independently of this, software design in the EDK flow is done using the C programming language. However, it is important to always be aware of the final executable's memory footprint. Although the amount of on-chip memory can be increased to accommodate larger programs, the amount of available total on-chip memory is limited, and using more blocks can decrease overall system performance. It is also possible to store the program in off-chip memory, but this comes at a cost, since it is much slower. Additionally, most rapid memories lose their contents when power is cycled, forcing the program to be stored in a slower, persistent memory, and transfered to a faster memory on power-up. This introduces an added complexity to the design, requiring the use of linker scripts. However, the current design only required a very simple program which fit in on-chip memory, so that such options were not required and, consequently, not explored in detail.

Once the hardware has been proved to function according to the established specifications, it can be integrated with the software components using the appropriate tools (for example Xilinx's EDK). In the case of the EDK, the hardware modules should be made into processor peripherals using the techniques described in section 4.10.1.1. Once the all the ports have been connected and the proper parameters set to customize the peripherals, the software can be integrated into the processor, and the system tested. If the system works, nothing more needs to be done. However, debugging the internal hardware can be tricky without the right tools. Debugging the peripherals is probably easiest by generating simulation models of the entire system. It is possible to generate behavioral, structural or timing models. The models from behavioral to timing take more and more time to simulate, but are more and more accurate in their reflection of the actual system. If the problem is instead with the software, the system can be debugged as described in chapter 13 of [**20**]. This allows the software to be stepped through while it is running in the actual hardware instead of in a simulation. Comments from Xilinx employees on newsgroups suggest that the preferred method is to use the "opb_mdm" module instead of the "xmd_stub" approach to debugging, although it was not made clear why. It is assumed that this will lead to fewer problems in the debugging process.

CHAPTER 5

Conclusion

This thesis described the design and test of a multiple camera image processing framework. The networking requirements were explored rapidly along with some possible avenues of research, and a node framework designed. This framework's suitability was then tested by implementing a person detection algorithm on it. This algorithm is not only appropriate for testing the framework, but can also be used as a first level of processing for more complex algorithms.

The chosen framework is in keeping with traditional reconfigurable computing architectures, and could be classified as a micro-processor with a reconfigurable co-processor. However, unlike most reconfigurable computing designs, the processor was implemented within the same reconfigurable matrix as the application specific firmware. While this comes at significant costs in terms of FPGA real-estate and processor performance, it allows for significantly more flexibility in configuring the processor, and a much tighter interface between processor and firmware.

The framework design was tested by implementing a person detection algorithm on it. This algorithm took a JPEG encoded data source, from which an integral image was extracted. This integral image, stored in off-chip memory, was then used in a feature-based, cascaded classifier approach to identify the portions of the image containing a person. Under ideal conditions, which could not be achieved given the available resources, it is believed that the detector could run at 2.5 to 3 frames per second when there is someone in the image, and at above 30 frames per second when there are no close matches.

5.1. Possible Extensions

5.1.1. Training in Firmware. The current iteration of this project does not have a hardware training module for the person-detector. However, given that the FPGA-based detector will be operating on the same images as the software one, the training can be done in software and the results loaded into hardware. Since the training data is currently hardwired into the HDL code, it is necessary to resynthesize the code whenever a new training set needs to be loaded. However, it is a relatively simple matter to isolate the parts of the bitstream that correspond to this data and modify them. This allows the creation of a partial reconfiguration bitstream which only modifies the memory locations containing training data and leaves the rest of the FPGA untouched. In fact, it should be possible to use the MicroBlaze itself to reconfigure these sections using the Virtex-II's Internal Configuration Access Port (ICAP). For memory segments that only ever need to be changed in their entirety, or that are seldom modified, partial reconfiguration of the memory segments permits the read/write capabilities of RAM without the added complexity of providing a datapath and control for writing to that memory.

Partial reconfiguration also offers interesting possibilities in view of on-line training of the detector. A camera module could be configured to gather training data to establish or refine the features that are needed by the detector. These feature points could then be stored in the same format and physical location as will be used by the detector itself. When training is complete, the FPGA can be partially reconfigured to replace the trainer with the detector, but leaving intact the data written by the trainer. The detector will then automatically be using the new training data, without the need to explicitly load it. However, this requires a judicious use of hard macros in both modules, and may lead to sub-optimal place and route in one or both modules. Since real-time training is not required, sub-optimal performance in the trainer can be accepted, which suggests that the detector be optimized first, and the trainer implemented according to the restrictions this imposes.

5.1.2. Using People Detection in a Camera Network.

5.1.2.1. *Homogeneous Detection Networks.* The most obvious way of integrating people detection into a cooperative networking is to have each of the cameras detecting in their own region of a scene. After some inter-camera calibration, the cameras could communicate the position of their objects in the scene. In this manner, it would be possible to seamlessly track targets through a large scene, with cameras passing detection from one to the next as the target moves around. This could also be used to overcome occlusion problems by having overlapping camera regions, with the camera that has the best view giving the object's location. If power consumption is an issue, cameras could operate in a low power consumption mode until they are awakened by a neighboring camera warning them of an object entering their field of view.

5.1.2.2. *Use of Detection Nodes in Heterogeneous Networks.* The term "heterogeneous network" is used here to describe a network of cameras that are running distinct applications that are nonetheless cooperating to achieve the final product. It can be viewed as the equivalent of a Multiple Instruction Multiple Data (MIMD) architecture compared to the Single Instruction Multiple Data (SIMD) organization of the homogeneous network.

The principal use of a person detector in a heterogeneous network is to provide "regions of interest" to the other nodes in the network. Using overlapping fields of view and some inter-camera calibration, this would allow the other nodes to focus in on those regions of their scenes that contain a person (or other object which the node has been trained to detect), ignoring the rest of the scene. This allows the use of algorithms that are more computationally intensive, since they are being applied to a more restricted region.

5.1.3. Connecting the Network.

Although simple in principle, the idea of connecting multiple processing elements presents many problems in practice. The field of wireless sensor networks provides some interesting insights into how this might be done, however. Of particular interest are the works of Kulik et al [**10**], and Intanagonwiwat et al [**9**].

In [**10**] Kulik et al. introduce the SPIN communication protocol, which has both broadcast and point-to-point variants. This protocol is based on a simple three-phase handshake, where data is advertised by the sender, and only forwarded to the nodes that specifically request it, minimizing superflous communications, and optimizing bandwidth usage. Although the broadcasting protocol was meant for wireless broadcast media, it can also readily be applied to wired ethernet. The point-to-point protocol can be used for local links.

The SPIN protocol was intended for full dissemination of data in a network. However, this work can be combined with that of Intanagonwiwat et al [**9**], who introduce directed diffusion, which describes how to optimize transmission of data from a source to a sink through multiple hops in a network. This is done by disseminating a request from the sink for a specific type of data, which is routed from source to sink through intermediate nodes in the network.

Of particular interest in both [**9**] and [**10**] is the tagging of data with "meta-data". The meta-data contains information describing the type of data that is available, and allowing other nodes to decide whether or not they are interested in receiving the data. This is very close to the idea of processing the image into higher levels of abstraction in order to reduce consumed bandwidth.

REFERENCES

[1] A.B. Abdelali, L. Boussaid, A. Mtibaa, and M. Abid, *Run-time reconfiguration for real-time low-level image processing: architecture and algorithm architecture adequation (AAA)*, 2002 IEEE International Conference on Systems, Man and Cybernetics, vol. 2, Oct. 2002, pp. 69–73.

[2] J. Arnold and K.L. Pocek (eds.), *10th annual IEEE symposium on field-programmable custom computing machines*, IEEE, 2002.

[3] A. Benedetti and P. Perona, *Real-time 2-d feature detection on a reconfigurable computer*, Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, June 1998, pp. 586–593.

[4] M.R. Boschetti, A.M.S. Adario, I.S. Silva, and S. Bampi, *Techniques and mechanisms for dynamic reconfiguration in an image processor*, 15th Symposium on Integrated Circuits and Systems Design, Sept. 2002, pp. 177–182.

[5] E. Cerro-Prada, S.M. Charlwood, and P.B. James-Roxby, *Designing image processing applications using reconfigurable computing*, Seventh International Conference on Image Processing And Its Applications, vol. 1, July 1999, pp. 450–454.

[6] K. Compton and S. Hauck, *Reconfigurable computing: A survey of systems and software*, ACM Computing Surveys **34** (2002), no. 2, 171–210.

[7] M.J.B. Duff, *Thirty years of parallel image processing*, VECPAR (J.M.L.M. Palma, Jack Dongarra, and V. Hernández, eds.), Lecture Notes in Computer Science, vol. 1981, Springer, 2001, pp. 419–438.

[8] T.W. Fry and S. Hauck, *Hyperspectral image compression on reconfigurable platforms*, in Arnold and Pocek [**2**], pp. 251–260.

[9] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, *Directed diffusion for wireless sensor networking*, ACM/IEEE Transactions on Networking **11** (2002), no. 1, 2–16.

[10] J. Kulik, W.R. Heinzelman, and H. Balakrishnan, *Negotiation-based protocols for disseminating information in wireless sensor networks*, Wireless Networks **8** (2002), no. 2-3, 169–185.

[11] J. Miano, *Compressed image file formats: JPEG, PNG, GIF, XBM, BMP*, SIGGRAPH, Addison Wesley Longman, Inc., 1999.

[12] V. Nair, *Learning-based detection of people for automated video surveillance*, Master's thesis, Electrical and Computer Engineering Department, McGill University, 2004.

[13] J.E. Scalera, III Jones C.F., M. Soni, M.B. Bucciero, P.M. Athanas, A.L. Abbott, and A. Mishra, *Reconfigurable object detection in FLIR image sequences*, in Arnold and Pocek [**2**], pp. 284–285.

[14] N. Srivastava, J.L. Trahan, R. Vaidyanathan, and S. Rai, *Adaptive image filtering using run-time reconfiguration*, International Parallel and Distributed Processing Symposium, April 2003, pp. 180–186.

[15] M.A. Vega-Rodriguez, J.M. Sanchez-Perez, and J.A. Gomez-Pulido, *Real time image processing with reconfigurable hardware*, The 8th IEEE International

Conference on Electronics, Circuits and Systems, ICECS 2001, vol. 1, Sept. 2001, pp. 213–216.

[16] P. Viola and M. Jones, *Robust real-time object detection*, Second International Workshop on Statistical and Computational Theories of Vision – Modelling, Learning, Computing, and Sampling (2001).

[17] G. K. Wallace, *The JPEG still picture compression standard*, IEEE Transactions on Consumer Electronics **38** (1992), no. 1, 18–34.

[18] Xilinx, Inc, *Constraints guide*, ISE 6.1.

[19] Xilinx, Inc, *Virtex-II platform FPGA user guide*, December 2002.

[20] Xilinx, Inc, *Embedded systems tools guide*, October 2003.

[21] Xilinx, Inc, *Microblaze processor reference guide*, September 2003.

[22] Xilinx, Inc, *Virtex-II platform FPGAs: Complete data sheet*, October 2003, DS031.

[23] Xilinx, Inc, *Virtex-II platform FPGAs: Detailed description*, October 2003, DS031-2.

Document Log:

Manuscript Version 0

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-LaTeX — 12 February 2004

Pierre-Olivier Laprise

Centre for Intelligent Machines, McGill University, 3480 University St., Montréal (Québec) H3A 2A7, Canada, *Tel.* : (514) 398-8200

E-mail address: `plapri@cim.mcgill.ca`

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-LaTeX