An FPGA-based Target Acquisition System

Alexander R. Marschner

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Dr. Peter M. Athanas, Chair
Dr. Mark T. Jones
Dr. Tom L. Martin

3 December 2007
Blacksburg, VA 24061

Keywords: OpenFire, multi-core, soft processor, object tracking, image clipping

An FPGA-based Target Acquisition System

Alexander R. Marschner

Abstract

This work describes the development of an image processing algorithm, the implementation of that algorithm as both a strictly hardware design and as a multi-core software design, and the side-by-side comparison of the two implementations. In the course of creating the multi-core software design, several improvements are made to the OpenFire soft core microprocessor that is used to create the multi-core network. The hardware and multi-core software implementations of the image processing algorithm are compared side-by-side in an FPGA-based test platform. Results show that although the strictly hardware implementation leads in terms of lower power consumption and very low area consumption, modification of and programming for the multi-core software implementation is simpler to perform.

Contents

List of Figures

List of Tables

Glossary

ALU Arithmetic Logic Unit

ANSI American National Standards Institute

ARM Advanced RISC Machine

ASIC Application Specific Integrated Circuit

ASIP Application Specific Instruction set Processor

BRAM Block RAM

CAuS Common Architecture for Microsensors

CLB Configurable Logic Block

CSP Communicating Sequential Processes

DAC Digital to Analog Converter

DDR Double Data Rate (Memory)

DOPB Data-side OPB

EDK Embedded Development Kit

EOF End of Frame

FIFO First In, First Out

FSL Fast Simplex Link

FPGA Field Programmable Gate Array

GPIO General Purpose Input / Output

GPP General Purpose Processor

HDL Hardware Design Language

IBM International Business Machines

ICAP Internal Configuration Access Port

IOPB Instruction-side OPB

IP Intellectual Property

LUT Lookup Table

MHz Megahertz

NTSC National Television System Committee

OPB On-chip Peripheral Bus

PLB Processor Local Bus

RAM Random Access Memory

RGB Red, Green, Blue

SCMP Single Chip Multiple Processor

SoC System-on-Chip

UART Universal Asynchronous Receiver Transmitter

VLSI Very Large System Integration

XUP Xilinx University Project

YUV A luma/chrominance based color space

Chapter 1

Introduction

When comparing hardware and software design implementation methodologies, conventional wisdom states that implementing a design in hardware has the advantage of speed while sacrificing design time, whereas implementing a design in software has a quicker design cycle whose final product may not be as fast. When a design is implemented in hardware the designer may harness the power of gate-level parallelism to allow many calculations to happen every clock cycle. A processor attempting to perform that same series of calculations will be forced, due to the traditionally serial nature of microprocessor computation, to execute each of those operations in sequence. Sequential computation significantly increases the amount of time required to complete an identical set of computations. On the other hand, if changes must be made to a hardware algorithm it frequently takes longer to produce a completed design, as the entire hardware set must be re-verified. Software can be modified much more easily than hardware, and with only one thread of execution the ramifications of each algorithmic change do not spread as far as in hardware.

1.1 Design Convergence

Moore's law[1] states that we can expect a doubling of the number of transistors on a chip roughly every two years. Careful planning in the semiconductor industry has allowed manufacturers to track this exponential growth curve with a fair amount of success[2]. However, doubling the computational performance of chips is not as simple as doubling the number of transistors on them. The smaller the feature size, the more signal quality issues are experienced when running at high clock rates[3][4], so running at an increased clock speed in order to improve performance is not always possible. In order to continue increasing performance as a function of number of available transistors, manufacturers of General Purpose Processor (GPP)s have turned to constructing multiple processing cores on each chip. Prime examples of this design methodology are Intel's Core2 Duo and Core2 Quad chips[5][6]. Although these processors can not take advantage of gate-level parallelism, they can run multiple processes concurrently, or multiple threads from a single process.

In the field of Configurable Computing, the FPGA takes the place held by GPPs in the field of general purpose computing. An FPGA provides the designer with a configurable environment in which to implement pure hardware algorithms. Designs implemented in an FPGA can take advantage of the gate-level parallelism that multi-core GPPs cannot; however, designs implemented in an FPGA have longer design cycles than designs implemented for parallel processing on a GPP. Software design for parallel processing units can take advantage of an existing hardware paradigm that supports multithreading, as well as using libraries and compilers that assist the designer to create a parallel design quickly. A hardware designer must create the entire system from the ground up, verifying that the base hardware as well as whatever application is running on top is working correctly. Any algorithmic change during the design process will necessitate the base hardware being verified again, a problem that does not exist for software designers. In addition, implementing an algorithm in an FPGA requires a special set of skills akin to assembly programming in that it requires deep knowledge of the hardware and varies from device to device. Many efforts have

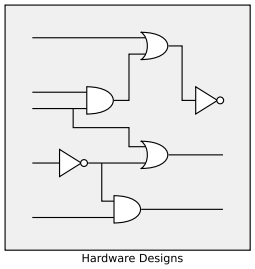
been made to allow designers to "program" an FPGA using a C-like language [7][8][9], however they have not been overly sucessful[10]. To make hardware design accessible to larger sections of the engineering design community, FPGA manufacturers have been embracing embedded processors. These processors can come in the form of hard macro processors[11] or in the form of Hardware Design Language (HDL) specified soft processors[12][13]. There is also development towards new types of FPGA fabrics that are abstracting away traditional micro-configurability and are moving towards a macro-configurability that focuses on configuring larger primitives, like simple processors or state machines[14].

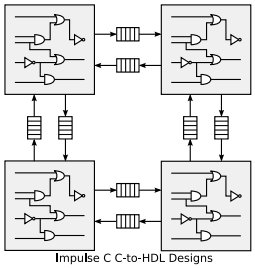
1.2 Solution Space

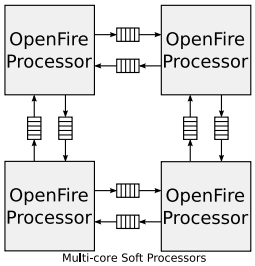
Designers of both GPPs and FPGAs are searching for ways to turn the ever-growing number of transistors on a chip into increasing performance. In the past GPP designers have had high clock speeds to work with, and FPGA designers have had gate-level parallelism as their main benefit. Sacrificing clock speed and a monolithic core, GPP designers have turned to multi-core designs for increased performance. FPGA designers have started to sacrifice gate-level configuration granularity in favor of many small processing units to increase performance and decrease design time. This convergence suggests that there exists an optimum trade-off between a gate-level design and a multi-core processor design.

The solution space for this convergence covers the area between pure hardware designs and monolithic processors. On the hardware end of the spectrum, performance is high, power consumption is low, but the effort required to design is very high. On the monolithic processor end of the spectrum the low-level hardware design has already been completed by the processor manufacturer, making further design for that system a simpler matter of writing software. This ease of design also has high performance, but consumes much more power and area to achieve it.

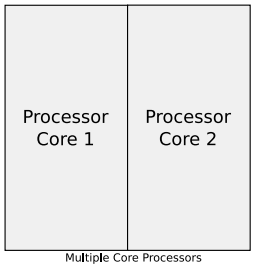
Designing hardware using a C language interface and then translating that software into hardware is perhaps the first step up from pure hardware designs. Impulse C[15] is a is one approach to writing hardware in C. The design paradigm for Impulse C is based upon the communicating sequential processes CSP model, where processing elements interact via well-defined channels. The implementation of the CSP model for Impulse C is realized via hardware blocks written in C that are connected by FIFOs. This method of design achieves the goal of allowing a designer to use a simpler tool than an HDL, while still maintaining power consumption and design area statistics close to a pure hardware design. Bisection bandwith far each hardware block is higher than for a hardware design as the bandwidth is only limited by the number of FIFOs attached, whereas in hardware the bisection bandwidth is limited by the number of input and output ports on the FPGA primitive blocks.


Hardware Designs

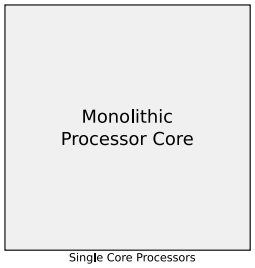

Impulse C C-to-HDL Designs



Multi-core Soft Processors



Multiple Core Processors



Single Core Processors

A further abstraction is to use a network of soft-core processors such as the OpenFire and connect the processor units with FSL busses. This is a very similar solution to the Impulse C solution, as it also presents the designer with the ability to write software to create blocks that are connected via what are essentially FIFOs. Including an entire processor core does have an area drawback; however, the processor design can be verified once and replicated, rather

than creating new hardware for each block in the system.

Multi-core processors and single core monolithic processors inhabit the software end of the solution space. At this stage the requirement for the designer to create hardware is gone. The hardware is fixed, and system design is performed solely in software. Although the speed of software designs can be quite fast depending on the host processor, large area and power consumption penalties are incurred.

1.3 Motivation

The motivation for this thesis is to explore a portion of the solution space and the design trade-offs experienced therein. Several requirements must be filled in order to explore this trade-off. It is necessary to have an algorithm that is implementable in both hardware and software, preferably a real-time algorithm that will indicate real-world performance. To make the multi-core comparison, and assuming that the platform for this comparison is an FPGA, a soft-core processor design is needed. The type of processor core is not necessarily important, however it needs to be fully featured for use in an embedded processor network. The algorithm must be implemented in hardware and in software, ideally running on the same platform at the same time so that a live comparison can be made. Once the system is implemented in both fashions, performance, power consumption, design area, ease of design, and other metrics can be used to perform a comparative analysis of the two implementations. Ideally the conclusion will indicate whether conventional wisdom is correct, or if this convergence between hardware systems and software systems indicates that a new design paradigm is reaching feasability.

1.4 Contributions

This thesis describes the development of the pieces required to explore the trade-off between gate-level parallelism and multi-core designs. An image processing algorithm is developed and implemented as both a hardware design and as a multi-core software design. In order to create a multi-core processor system to support a software version of the image processing algorithm, an open source soft-core processor called the OpenFire[13] is modified to increase its utility as an embedded network processor and then instanced several times in a processor network. The final contribution of this thesis is a comparison of the hardware and multi-core designs that attempts to show that multi-core designs do exhibit the performance required to compete with low-level hardware designs.

1.5 Organization

Chapter 2 discusses some related work relevant to this thesis, including FPGAs, image processing algorithms implemented on FPGAs, and single-chip multi-core devices implemented using soft processors. Chapter 3 describes the image processing algorithm designed for use in this thesis. Chapter 4 goes into detail about the OpenFire soft-core processor and the modifications that are made to make it more useful as an embedded network processor, namely the completion of its complement of FSL ports, and the addition of OPB support. Chapter 5 describes in detail the implementation of the harware and software multi-core implementations of the image processing algorithm, and Chapter 6 provides an analysis of the two implementations. Chapter 7 presents a summary of the conclusions that can be reached from this work.

Chapter 2

Related Work

The goal of this thesis is to demonstrate a real-time algorithm implemented as both a pure hardware design and as a multi-core software-based design, and to compare both of those algorithms in a side-by-side manner. An image processing algorithm, a soft processor core, and the hardware/software implementations of that algorithm are all required in order to perform the desired comparison. This section describes previous work that has been performed in related areas.

2.1 FPGAs

FPGAs are programmable logic devices that allow for prototyping of hardware designs. Their programmable fabric is chiefly composed of N-bit Lookup Table (LUT)s, which individually allow the emulation of functions with $logN$ inputs. In the Xilinx Virtex 2 Pro line of FPGAs, two LUTs, two flip-flops, and some routing logic are bundled together into a slice. Two slices are paired to make up a Configurable Logic Block (CLB)[11]. The CLB is the basic building block for implementing hardware designs on a Xilinx FPGA, and is tiled throughout much of the device. The area that is not used for CLBs is used for wire routing, hardware multiplyers,

Block RAM (BRAM)s, and two hard-macro PPC405 microprocessor cores.

FPGAs are usually configured using a design written in an HDL. A programming file or "bitstream" for an FPGA can be generated from the HDL design using a series of phases: synthesis, mapping, place-and-route, and final bitstream generation. The synthesis phase creates a netlist file with a gate-level implementation of the HDL design. The mapping phase modifies the netlist generated in the synthesis phase to use only those logic primitives that are available in the FPGA's fabric. For example, a two input AND gate might be mapped into a four bit, two input LUT, with the internal values of the lookup table (LUT) set to values that allow the LUT to replicate the AND function. Once the netlist has been mapped to FPGA logic primitives, the place-and-route phase attempts to assign each of those primitives to an actual instance of that logic block on the FPGA while still meeting timing requirements for the routing between logic blocks. If the place-and-route phase is successful a final bitstream can be created from the placed and routed design. The final bitstream contains the configuration information for the device to allow it to implement the design originally specified in HDL.

2.2 Soft Core Microprocessors

A "soft" processor core is an HDL implementation of a microprocessor that may then be implemented using the primitive blocks on an FPGA or similar device. A soft processor core could conceivably be placed in an Application Specific Integrated Circuit (ASIC) device during the design stages; however, there are VLSI cores available from vendors like ARM[16] that fill that design niche. The real flexibility of soft core processors lies in the designer's ability to tailor a configurable platform with the desired number and arrangement of processors to satisfy the requirements of the system.

Several soft processor cores exist for use in configurable designs. Xilinx and Altera, the

two main FPGA manufacturers, each have their own proprietary cores. The MicroBlaze[12] and PicoBlaze[17] processors are included with the Xilinx EDK toolset. Altera provides the Nios II processor[18] with their embedded design suite. Lattice Semiconductor Corporation has an open source microprocessor, the LatticeMicro32[19]. The Xilinx microprocessors, while not open source themselves, have attracted a following of open source clones such as the PicoBlaze clone the PacoBlaze[20], and the OpenFire[13], a binary-compatible MicroBlaze clone.

The primary reason to use an open-source microprocessor is that the source HDL is freely available and can be modified to fit the application at hand. In Application Specific Instruction set Processor (ASIP) research, this translates into the ability to create a special instruction set for a processor, removing useless instructions and adding in low level support for other operations that will speed up the algorithm the processor is being designed to run. Processors such as Tensilica's Xtensa 7[21] that use application-specific instruction sets are able to realize large performance boosts over general purpose instruction sets[22].

In Single Chip Multiple Processor (SCMP) research access to the processor's source allows the designer to support multiple instruction sets for each processor on the device. This extends to minimizing each instruction set as much as possible to conserve space and maximize the number of cores available, or to creating special cores for each task as in a System-on-Chip (SoC) design.

2.3 Single Chip, Multi-core

With modern chip fabrication processes reaching upwards of one billion transistors on a single die[23], low level hardware designers are able to place multiple processor cores on a single device to alleviate design problems faced by monolithic single-core designs. Single chip multi-core devices have a communications speed advantage over multi-core devices implemented on

several chips in that they can connect processors at their native bus speeds without wasting power or reducing speed in order to drive off-chip lines[24]. On chip connections between microprocessors can also take advantage of the tendency for configurable microprocessors to have many communication ports. Although each processor can only read or write from a single port on each clock edge, every processor in the multi-core network has a great deal of communication bandwidth[25].

Placing multiple cores on the same device also solves a problem referred to as the "design gap." The design gap is a divergence between the amount of transistors available on a processor die at the current feature size and the amount of work that it requires for a design team to create a single processor core that utilizes all of those gates[25]. By using multiple cores on the same die the design team can increase computational performance by adding processing parallelism while at the same time reducing the amount of design work needed by consuming more gates with copies of the same design. This strategy has the additional benefit of allowing a processor core to be formally verified a single time and then have that verified IP used multiple times.

The design of network topologies for single chip multi-core devices is a problem of hardware/software co-design. The network topology is dependent upon the task to be performed by that device, just as the tasks to be performed on each processor core are dependent upon the network topology. FPGA-based implementations have an advantage over ASIC-based implementations in that the connectivity of the soft core processors on the device may be reconfigured as the task set changes. Whether the processor network is configured as a linear daisy chain, a star network, or a mesh, the division of software tasks over the set of cores in the network can influence the efficiency of the network. To maximize efficiency the logical tasks should be divided between the cores such that each core will finish at the same moment[26].

2.4 Object Tracking using FPGAs

FPGAs are frequently used in image processing algorithm applications due to their ability to handle large amounts of incoming and outgoing data, and their ability to execute multiple operations in parallel on that stream of data. These two qualities allow FPGAs to handle data streams that GPPs cannot. Specially designed ASICs could be used to process the same image data; however, such ASICs have a fixed hardware, only able to perform a specific set of computations. FPGAs are reconfigurable and therefore have a much greater flexibility[27].

One popular method for implementing an object tracking image processing algorithm is the use of "particle filters." This filtering approach assumes that the general features of successive video frames can be approximated via a set of discrete samples taken over the entire area of each frame. From the samples "features" such as detected edges, gray level, contours, or color value may be used to track a target[28]. An FPGA-based implementation of this algorithm allows the filter values to be sampled and calculated as often as possible while not discarding incoming data during the update, an advantage over GPP-based implementations.

Color difference and thresholding represent a slightly simpler approach to an object tracking algorithm. A color representing a desired target can be detected in the incoming video stream, and if the detected color is close enough to the desired color, the coordinates where that color appeared can be fed to an ancillary algorithm that calculates the geometric center of the detected pixels[29]. Although the RGB color space can be used for the incoming video data, the YUV color space has the advantage of separating the color value and the brightness value, meaning that changes in the lighting do not affect the color value as much as in the RGB color space.

2.5 Summary

An FPGA provides an excellent hardware development platform with which to compare a hardware implementation of an object tracking image processing algorithm with a multi-core software-based implementation of the same algorithm. A strictly hardware design is able to leverage the gate level parallelism offered by an FPGA, and it remains to be seen whether the advantages of having multiple processor cores on the same device can make up the performance difference.

Chapter 3

Object Tracking

In a world where military technology is evolving towards automous systems that can interpret the vast visual environment around them[30][31], and as consumer electronics focus more heavily on mobile entertainment platforms with advanced streaming video features[32], real-time image processing algorithms that can analyze incoming streams of data and provide interpretations of the information that go beyond technical statistics are becoming necessary[33].

In this chapter, a real-time image processing application is presented, which will subsequently be used for a performance comparison between hardware-based and multicore-based image processing systems.

3.1 Image Chipping Algorithm

In 2001, Jon Scalera published his masters thesis wherein he described an image processing algorithm for an FPGA that would detect a region of movement in an incoming video stream to allow the host system to conserve power by focusing on or devoting resources to only the

active region of the frame. This technique was termed "image chipping," as it allowed the system to "chip" out the piece of the video stream that was of interest[34].

In essence, the image chipping algorithm functioned by calculating the difference between two sequential frames, forming a "difference image" where each pixel's value represented the difference between the value of a pixel in the current frame and the value of the pixel in the same location in a previous frame. The difference image was then passed through a thresholding filter, creating a binary image where pixels whose difference values were below the threshold appeared as white or "true" pixels, and those whose difference values were above the threshold appeared as black or "false" pixels. The final result then consisted of a binary image where the white regions denoted a region of interest and the black regions denoted the background. The input video signal to the Common Architecture for Microsensors (CAuS) system was assumed to be a grayscale image generated by a stationary infrared camera. A demonstrated ability of the completed system was identifying a group of people walking across a field some distance from the camera by giving a series of XY coordinates that denoted regions of motion within each frame.

3.2 Algorithm Modifications

The "image chipping" algorithm was modified to accomodate a color video stream coming from a mobile video source to make the chipping algorithm more useful on a variety of platforms. The transformation changed the details of how the algorithm worked; however, the end goal of the algorithm remained the ability to "chip" out sections of interest from the incoming video.

The key difference between a fixed video source and a moving video source is that there is no fixed frame of reference. The difference image formed in the chipping algorithm relies on the fact that the background does not appreciably change between frames, providing

an inherent frame of reference. However, if frames from a moving source are fed through the chipping algorithm, the difference stage of the algorithm will cause the entire image to appear as though it is moving. Instead of focusing on finding a difference between two sequential frames, the modified algorithm identifies a desired color within the frame. This identification is still performed via a difference operation and followed up with a threshold filter, so it is equivalent to the original algorithm.

Since the modified version of the algorithm utilses a user-specified desired color as a basis for the difference stage of the algorithm, the new algorithm does not require a full frame buffer to store previous frames. This reduces the algorithm's hardware footprint.

3.3 Algorithm Details

The modified image chipping algorithm, hereafter referred to as the "clipping" algorithm, can be broken into three logical blocks representing distinct stages of the algorithm. The first block is the difference engine, the second block performs noise reduction, and the third block keeps statistics on the incoming data in order to locate a bounding box for the color of interest (if it exists in the current frame).

3.3.1 Difference Engine

The difference engine receives a 24-bit color in the Red, Green, Blue (RGB) color space, a 24-bit desired color also in the RGB color space, and a threshold value. Equation 3.1 shows the generic form for the distance equation in three dimensional space. Treating the pixel color and the desired color as two different points in a three dimensional space whose X, Y, and Z coordinates are given by their respective R, G, and B values, the square of the distance between the two colors can be described by Equation 3.2.

$$D = \sqrt{(X_1 - X_0)^2 + (Y_1 - Y_0)^2 + (Z_1 - Z_0)^2} \tag{3.1}$$

$$D^2 = (R_{pixel} - R_{desired})^2 + (G_{pixel} - G_{desired})^2 + (B_{pixel} - B_{desired})^2 \tag{3.2}$$

The closer the pixel value is to the desired color, the smaller the value of D^2 will be. It should be noted that the square root of the distance is not taken. Instead the distance result is compared to the threshold as is, which assumes that the threshold value is given in terms of the distance squared. If the square of the distance is below the threshold then the two colors are close enough together and the difference engine outputs a 'true' value for that pixel's XY coordinate. If not, then the difference engine outputs a 'false' value. Figure 3.2 is a graphical representation of the distance of each pixel in Figure 3.1 from the desired color value of RGB = 0,255,0. Figure 3.3 shows the result of applying a threshold distance of 40,000 to Figure 3.2.

Figure 3.1: A katydid on the doors of Torgersen Hall.

Figure 3.2: Difference algorithm applied to photo in Figure 3.1

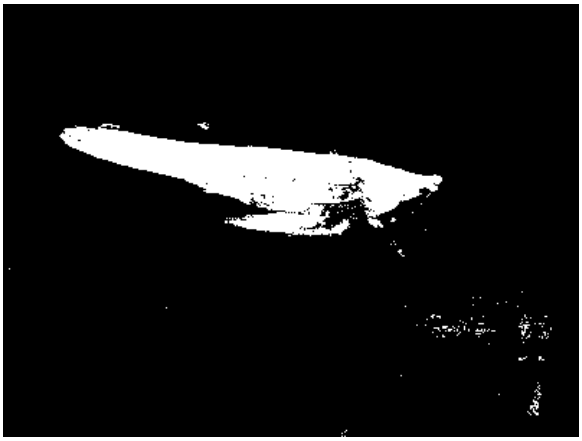


Figure 3.3: Threshold applied to Figure 3.2

3.3.2 Noise Reduction

The second block performs noise reduction on the true and false values generated by the difference engine by applying a morphological image processing operation called "opening." The "opening" of an image is performed by applying an "erosion" operation followed by a "dilation" operation[35]. Given a binary image where the pixels are represented as either ones or zeros the erosion algorithm inverts any 'true' pixels that border on 'false' pixels. Dilation does the opposite, causing 'false' pixels that neighbor 'true' pixels to invert. By applying the erosion operation several times to an image, features (defined as groupings of 'true' pixels) contained in the original image that are smaller than a certain size will be removed. Applying the dilation operation the same number of times then restores any regular features that were originally above that threshold size to their original dimensions. Non-regular features may be truncated, as edge patters may be completely removed during the erosion process. Comparing Figure 3.5 to Figure 3.3 shows this truncation of the edges of the dilated features.

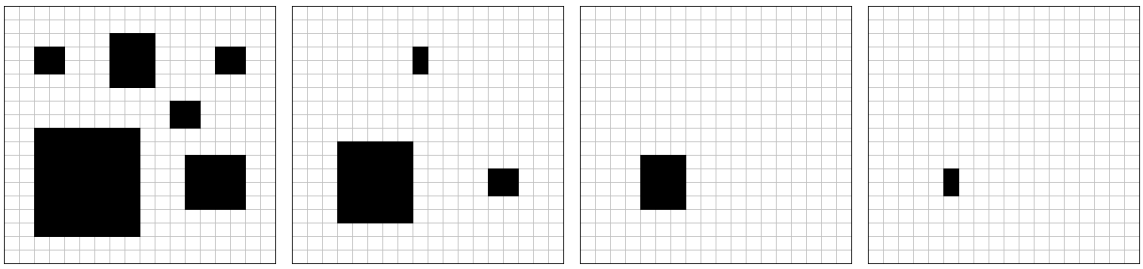


Figure 3.4: Erosion operation applied three times.

Figure 3.4 shows the erosion operation being applied multiple times to remove the smaller features from the original image. Figure 3.5 shows the dilation operation being applied, returning the feature in the lower left corner to its original dimensions.

Applying the "opening" operation to the results of the difference engine discussed in Section 3.3.1 results in the removal of small features that may be irrelevant to finding the desired color in the frame. Lighting conditions and surface reflectivity of objects in the field

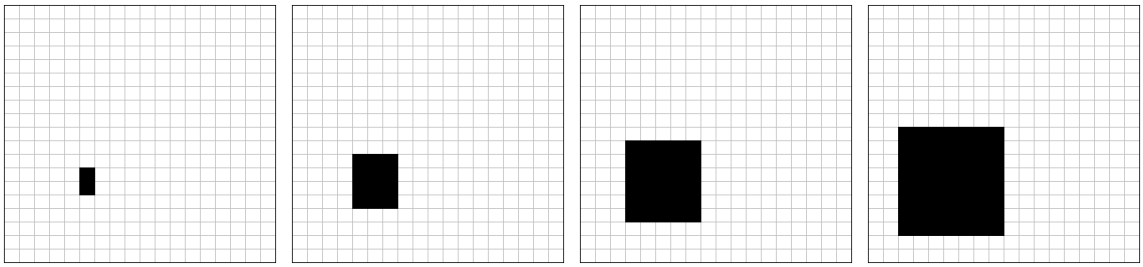


Figure 3.5: Dilation operation applied three times.

of view can contribute false positive pixels to the output of the difference engine. These erroneous pixels can be seen in Figure 3.3. Figure 3.6 shows the reduction in those pixels following an "opening" operation consisting of three erosions and three dilations.

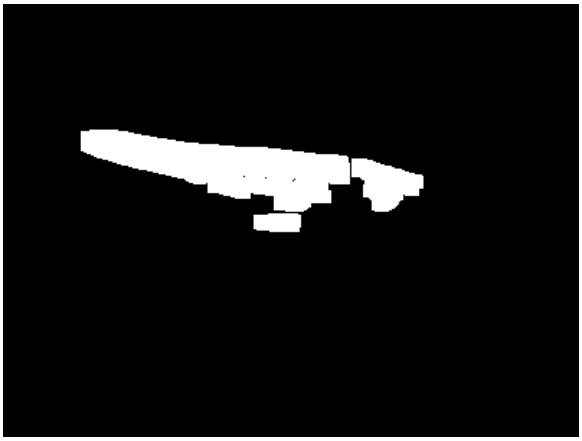


Figure 3.6: Noise reduction performed.

3.3.3 Bounding Box Statistics

The third and final block in the algorithm receives the "true" and false values for each pixel, as well as the X and Y coordinates for that pixel in the video frame. If the value is "true" for that coordinate, the coordinate is tested against the statistics registers being kept for the current frame. For each X and each Y value, if that value is greater than the current maximum value or less than the current minimum value then the value will be recorded as the current maximum or minimum, as appropriate. After applying these statistics to an

entire image a bounding box is formed, defined by the X and Y minimum and maximum coordinates that were below the distance threshold for this frame.

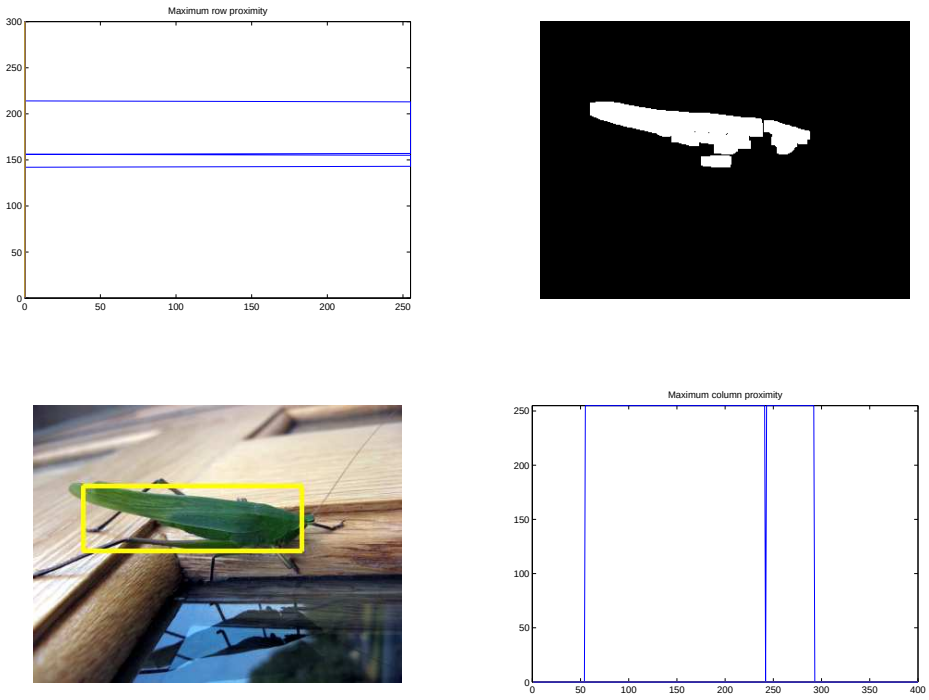


Figure 3.7: Color proximity of noise-reduced frame.

Figure 3.7 shows the statistics for X and Y minumum and maximum values being performed on the noise-reduced image from Figure 3.6. The image in the lower left of Figure 3.7 shows the bounding box drawn using the statistics taken. The bounding box neatly defines the greenest area in the image.

3.3.4 Summary

The successive application of a difference function, a thresholding algorithm, noise reduction, and analysis of the resulting data stream leads to the calculation of two XY coordinate pairs that describe a bounding box around the desired color in a given frame. Applied to several

frames in series, this bounding box tracks objects of the desired color by recalculating the bounding box for every frame. While differences in lighting may affect the ability of the algorithm to select the desired color, this algorithm is able to easily recover from such tracking errors, as the bounding box recalculation does not involve previous state information[29].

Chapter 4

Extending the OpenFire Processor

4.1 The OpenFire Processor

The OpenFire soft processor is an open source, binary compatible clone of the Xilinx MicroBlaze[13]. As it is binary compatible with the MicroBlaze, it can be used as a drop-in replacement for the MicroBlaze, and use the exact same toolchain. The OpenFire was chosen for this research because of its ability to leverage the MicroBlaze toolset, and because the OpenFire was originally designed for the express purpose of performing research with configurable processor arrays. Although the MicroBlaze processor has some configurable options, such as data and instruction caches, the number of FSL busses, and whether operations like multiplication and barrel shifting are expressed in hardware or software, the OpenFire exceeds its configurability by allowing sizing of the data path width, addition or subtraction of instructions to the instruction set, and a more efficient memory bus.

Version 0.3b of the OpenFire was released in 2001. That version of the processor was fully operational; however, it only offered one of eight possible FSL busses, and did not have the same standard bus connectivity offered by the OpenFire. For the OpenFire to be a truly

useful MicroBlaze replacement it needed full FSL support, as well as an OPB bus master.

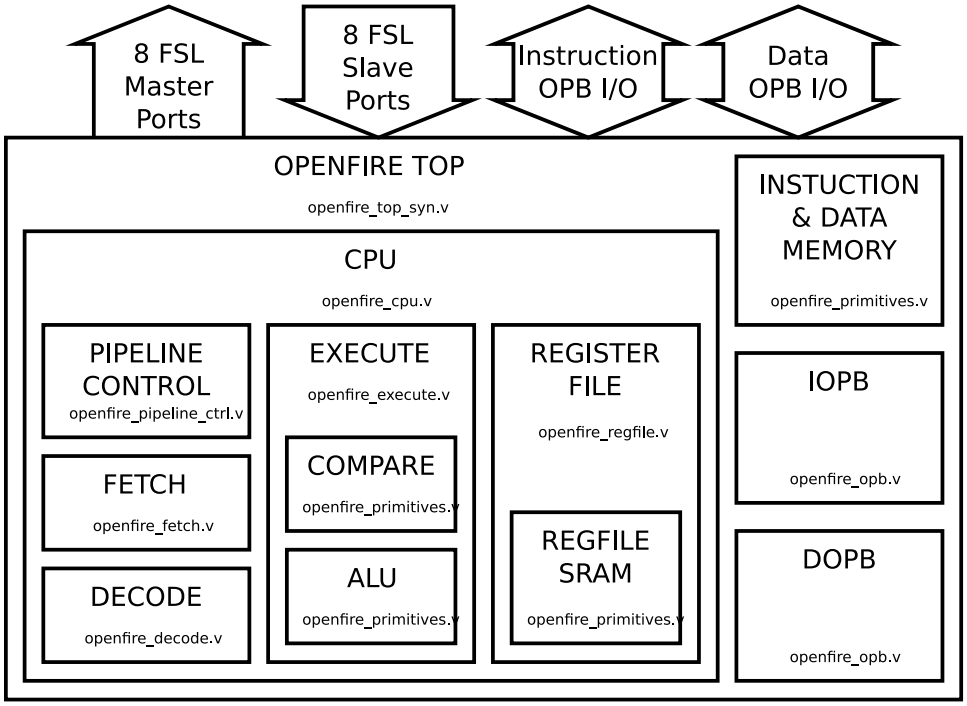


Figure 4.1: Top level diagram of the improved OpenFire processor.

Figure 4.1 shows a top-level block diagram of the OpenFire processor after the modifications in this chapter are applied. Each of the named blocks also includes the name of the file where that module is implemented in HDL. The top-level module has eight FSL master (output) ports, and eight FSL slave (input) ports, as well as two OPB bus connections. The instruction-side OPB (IOPB) port is a read-only port that interfaces with the OpenFire's instruction memory. The data-side On-chip Peripheral Bus (DOPB) port is a read/write port that interfaces with the OpenFire's data memory.

Inside the top-level OpenFire module are three main units, the CPU core, the local data and instruction memory, and the two OPB cores that control the IOPB and DOPB ports. The IOPB and DOPB modules are instances of the `openfire_opb` state machine, which is discussed in Section 4.5. The instruction and data memory are kept in the same BRAM-implemented memory, as the OpenFire's instruction and data address spaces overlap.

The OpenFire uses a three-stage pipeline, which is implemented inside the CPU core in the fetch, decode, and execute blocks. The pipeline control serves to stall that pipeline when executing multi-cycle instructions. The register file implements all the internal program registers, and interfaces with the other modules in the CPU core in order to route data to the appropriate places.

4.2 The Fast Simplex Link Bus

The Fast Simplex Link (FSL) bus[36] is a simple, monodirectional, point-to-point communications bus that has instruction set level support in both the MicroBlaze and OpenFire soft core processors[12][13]. The FSL bus is implemented as a hardware FIFO which makes it ideal for crossing clock boundaries and synchronizing source and sink modules.

Adding the full complement of eight FSL ports to the OpenFire is important because it will allow multiple OpenFire processors to be connected in various network configurations with a high bisection bandwidth.

4.2.1 FSL Signals

The signals associated with an FSL link are as follows:

FSL Global Signals

`FSL_CLK` is the clock used when both the master and the slave ports of the FSL link are operating synchronously.

`SYS_RST` is an active high system reset which clears the internal FIFO.

FSL Master Signals

`FSL_M_DATA` is the 32-bit bus carrying data from the master FSL port into the FIFO.

FSL_M_WRITE is a one bit signal that causes the data on FSL_M_DATA to be read into the FIFO on the rising edge of the clock.

FSL_M_FULL is a feedback signal from the FIFO indicating whether the FIFO is full. Writes can not happen when the FIFO is full.

FSL_M_CONTROL is a one bit line that can send an optional control signal along with the data, allowing both control words and data words to be sent on the same FSL link.

FSL_M_CLK is the clock used for the master port when the master and slave clocks for this FSL link are asynchronous, as when using an FSL to cross clock domains.

FSL Slave Signals

FSL_S_DATA is the 32-bit bus carrying data from the FIFO out to the slave FSL port.

FSL_S_EXISTS is a one bit signal that is high when data exists in the FIFO and remains high until the FIFO is empty.

FSL_S_READ is a one bit signal allowing the slave to remove data from the FIFO and have it expressed on FSL_S_DATA.

FSL_S_CONTROL is a one bit line that carries an optional control signal along with the data, allowing both control words and data words to be sent on the same FSL link.

FSL_S_CLK is the clock used for the slave port when the master and slave clocks for this FSL link are asynchronous.

4.2.2 FSL Usage

Figure 4.2 shows the connection of two devices via an FSL bus. For simplicity, the clock and reset lines have been ommited, assuming the FSL links to by synchronous. Device 1 is the master for FSL Bus 0, which is attached to Device 1's zeroth master port. Device 2 is the slave for FSL Bus 0, which is attached to Device 2's zeroth slave port. This connection allows Device 1 to send data accross the FSL link to Device 2, as shown by the arrow over FSL Bus 0. Two-way communication between the two devices is achieved by utilizing two FSL links, since each link is monodirectional.

Although the data width for an FSL bus is variable, it is assumed that the width is thirty-two bits.

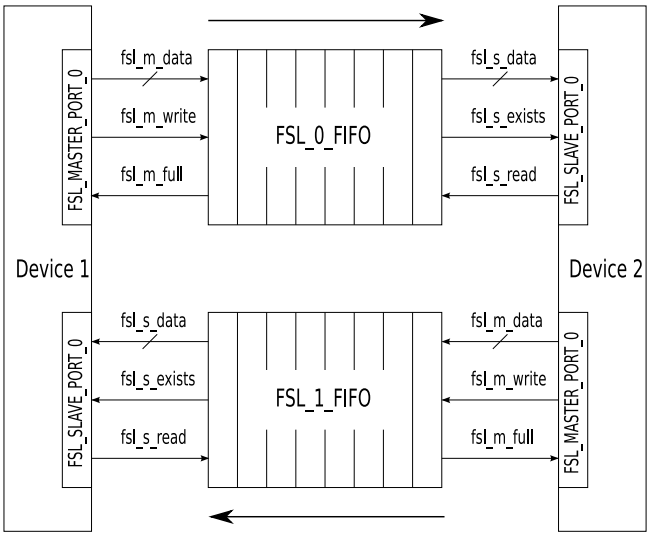


Figure 4.2: FSL bus connections.

The FSL busses on an OpenFire processor are particularly useful because the FSL read and write commands are supported by the MicroBlaze instruction set[12]. This means that all FSL instructions are single-cycle instructions with hardware support for blocking and non-blocking operation. If data is not available on an incoming FSL link, or if there is no room to send data on an outgoing FSL link then the processor stalls until the operation is able to complete. The folllowing functions are the C-language functions made available for the programmer to access the various FSL instructions. The functions are defined as MicroBlaze assembly macros in the `mb_interface.h` header file.

`getfsl(VAL,ID)` retrives the first value from FSL number ID and stores it in VAL.

`putfsl(VAL,ID)` sends the value of VAL via FSL number ID.

`ngetfsl(VAL,ID)` non-blocking version of `getfsl`.

`nputfsl(VAL,ID)` non-blocking version of `putfsl`.

`fsl_isinvalid(RES)` if the last non-blocking FSL action failed, RES is non-zero.

4.3 Completing FSL Support

The thirty-two bit assembly language instructions for the `getfsl` and `putfsl` operations reserve the three least significant bits of the machine instruction word to identify the destination bus, as shown in Figure 4.3 by the 'FSL_NO' field. The bit labeled 'N' determines whether the blocking or the non-blocking form of the instruction is used. The bit labeled 'C' determines whether the control bit is ignored. The 'FSL_GET' and 'FSL_PUT' fields are the opcodes corresponding to the particular instruction, and the 'SOURCE' and 'DESTINATION' fields are register addresses.

Since each FSL link is monodirectional and there are three bits to use for both send and receive commands, the OpenFire processor can support up to eight master and eight slave FSL ports. The master ports are accessed via the `getfsl` instruction, and the slave ports are accessed via the `putfsl` instruction.

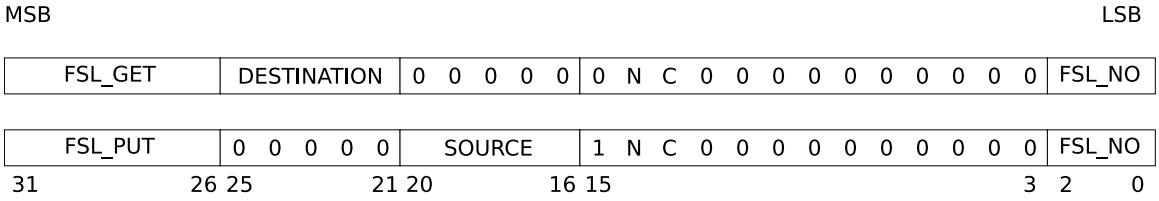


Figure 4.3: FSL instruction words.

Figure 4.4 shows the signal timing diagram for an FSL transaction between devices 1 and 2 using FSL Bus 0, as shown in Figure 4.2. The two devices are considered to be running the code shown in the appendix on Page 83. In the code, Device 1 attempts to write to the bus five times. Device 2 attempts to read five times. Since the bus FIFOs shown in Figure 4.2 are greater than five spaces deep, Device 1 is able to write all five times without being interrupted, and Device 2 is able to continually read as there is always data available. Figure 4.5 shows what happens when the FSL links are implemented with single register FIFOs. In that case the bus is continually full, alternately stalling Device 1 on writes while Device 2 waits on more data.

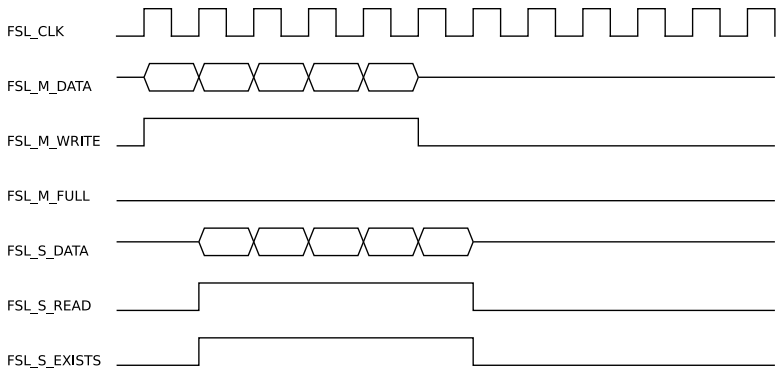


Figure 4.4: FSL transactions with 8 register deep FIFOs.

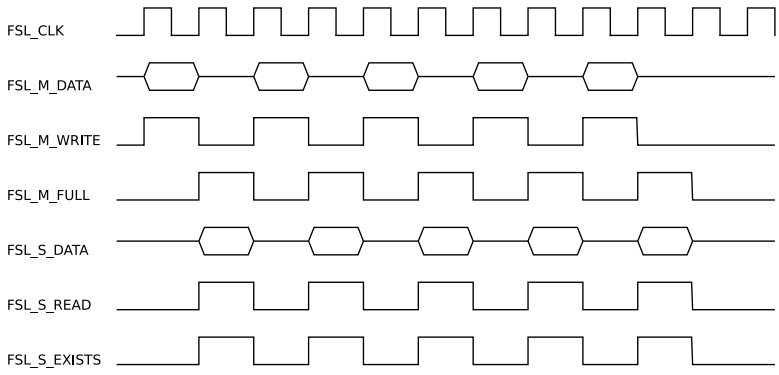


Figure 4.5: FSL transactions with 1 register deep FIFOs.

To enable all eight FSL busses, the FSL instruction must be decoded and the FSL number must be passed out of the decode module of the OpenFire. In Version 0.3b of the OpenFire processor this decode was not done, which could have resulted in an error condition where assembly code referring to any of the other seven FSL busses would result in the zeroth FSL bus being used.

The second necessary change was to multiplex the incoming and outgoing control and data lines, based on the number of the FSL bus being accessed. Figure 4.6 shows the hardware required to do the decoding and multiplexing operations.

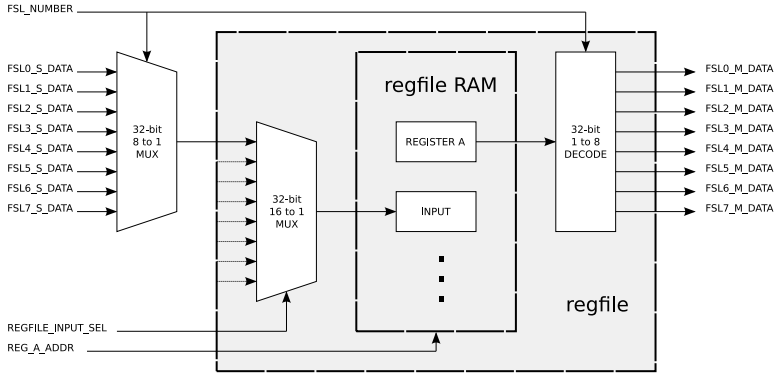


Figure 4.6: Modifications to the OpenFire register file for FSL data I/O.

Once the seven additional FSL links were added, an OpenFire was implemented in hardware as shown in Figure 4.7 and the C-language code shown in the appendix on Page 84 was run, producing the correct result and proving the correct operation of all eight FSL links.

4.4 The On-chip Peripheral Bus

The On-chip Peripheral Bus (OPB) was developed by IBM as a fully synchronous, multiple master, multiple slave, arbitrated bus for the connection of lower performance peripherals[37]. It was designed with a simple, well specified interface to make it easy to connect IP to the bus. Due to its ease of implementation, and its adoption by Xilinx as a standard bus for

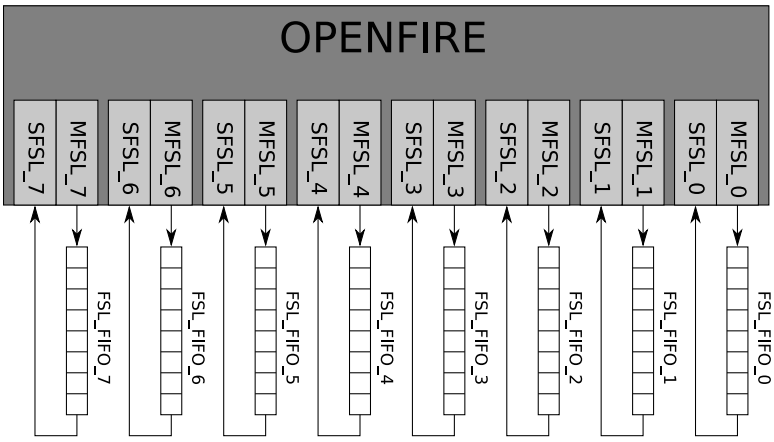


Figure 4.7: OpenFire FSL connections in a test configuration.

their MicroBlaze soft processor core[12], there are many IP cores available for use with the bus. Some examples of OPB-based IP included with Xilinx's Embedded Development Kit (EDK) are a DDR memory controller, an interface to the Internal Configuration Access Port (ICAP), and General Purpose Input / Output (GPIO) cores that can be connected to arbitrary outputs of the host device.

The OPB standard supports up to 16 masters and 16 slaves, and requires an arbiter to do bus access arbitration between the master devices. The bus data width can be specified as either 32 or 64 bits, although the Xilinx implementation is fixed at 32 bits[38]. OPB performance cannot be considered deterministic, as bus arbitration and slave response time can vary depending on the state of the bus and the type of peripherals. In the single master scenario the worst-case response time for a peripheral is usually on the order of 16 cycles. If a slave does not respond in that time window the arbiter will raise the timeout signal, informing the master that initiated the connection that the slave did not respond. However, if a slave device is designed to take more than 16 cycles to respond, a timeout suppression signal may be used to suppress the arbiter's timeout signal.

4.4.1 OPB Specifications

Allowing the OpenFire to access the OPB required the addition of an OPB master port and the associated control logic. The following signals are required for an OPB master port.

OPB Master signals coming from the bus

`OPB_MGrant` is a 1-bit signal originating at the arbiter that indicates when a master has been given permission to access the OPB.

`OPB_DBus` is the 32-bit data bus on which OPB slaves return data from queries by a master.

`OPB_xferAck` is a 1-bit signal that is used to end an OPB transaction. During the single cycle that this signal remains high the data on the `OPB_DBus` signal is valid.

`OPB_errAck` is a 1-bit signal that is used to terminate an OPB transaction when an error condition prevents it from completing normally.

`OPB_retry` is a 1-bit signal that is used to signal a retry of the previous (failed) transaction.

`OPB_timeout` is a 1-bit signal raised by the arbiter that terminates an OPB transaction when the slave peripheral takes more than 16 cycles to respond without raising the timeout suppression signal.

OPB Master signals sent out to the bus

`M_BE` is a 4-bit signal that enables byte and half-word reads and writes along with full-word access.

`M_ABus` is a 32-bit address bus allowing the OPB master to address the various OPB peripherals.

`M_busLock` is a 1-bit signal used to indicate multiple transactions by the master device that asserts it. When the bus is locked the arbiter will not grant access to other masters.

`M_DBus` is a 32-bit data bus carrying data from the master device when performing a write operation.

`M_request` is a 1-bit signal that is raised by the master device when OPB access is desired, and lowered when the arbiter responds by raising `OPB_MGrant`.

`M_RNW` is a 1-bit signal that when high indicates a read operation and when low indicates a write operation.

`M_select` is a 1-bit signal raised by the master immediately after the arbiter grants that master access via `OPB_MGrant`.

`M_seqAddr` is a 1-bit signal indicating to the peripheral that sequential accesses will be using sequential addresses. This signal is used when bursting data accross the OPB.

4.4.2 OPB Operation

An example system using OPB for connectivity between modules can be seen in Figure 4.8. The two masters will negotiate with the arbiter for bus access, after which they can address any of the four slaves shown. Each slave has a 32-bit address, shown below it, by which it may be addressed.

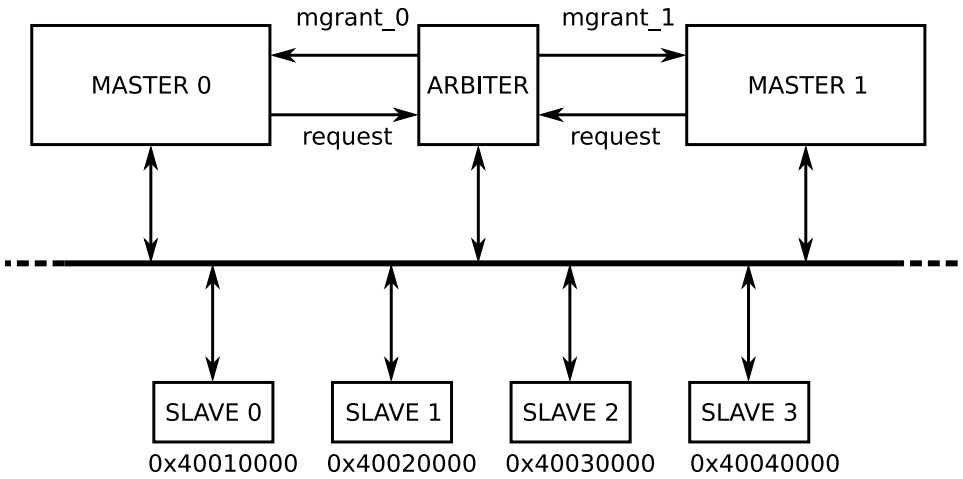


Figure 4.8: OPB example.

There is an important difference between an OPB access and a regular memory access. Before a master may access the bus it must request access from the arbiter. Following that, an OPB slave has up to 16 cycles to respond to the master. It is essential to ensure that the master device, in this case the OpenFire, waits for the slave's response. Figure 4.9 shows the signal diagram for a memory access directly from the data memory. The requested data appears on the bus the very next cycle after the address is applied. Figure 4.10 shows an OPB read with a latency of three cycles. The `Stall` signal in Figure 4.10 is used to

prevent the master device from continuing with execution until the incoming data is valid. Before the transfer occurs, arbitration is performed using the `M_request` and `OPB_MGrant` signals. The master's request using `M_request` is granted by the arbiter on the following cycle, via `OPB_MGrant`. Immediately upon being granted access to the bus, the master raises the `M_select` signal. When the `M_select` signal is high the address bus, `M_ABus`, must be valid. This would also apply to the `M_DBus` and `M_RNW` wires in the case of a read. In the example shown in Figure 4.10, the slave signals its reply with the `OPB_xferAck` signal after only one cycle of delay. The returned data on `OPB_DBus` is valid for the one cycle that `OPB_xferAck` is held high by the slave. As soon as the `OPB_xferAck` signal is asserted, the `Stall` signal is dropped, allowing the master to receive the results of the OPB read. In both Figure 4.9 and Figure 4.10 the vertical dotted lines bracket the active portion of the read, indicating that three more cycles are required to read from the OPB than to read from local memory.

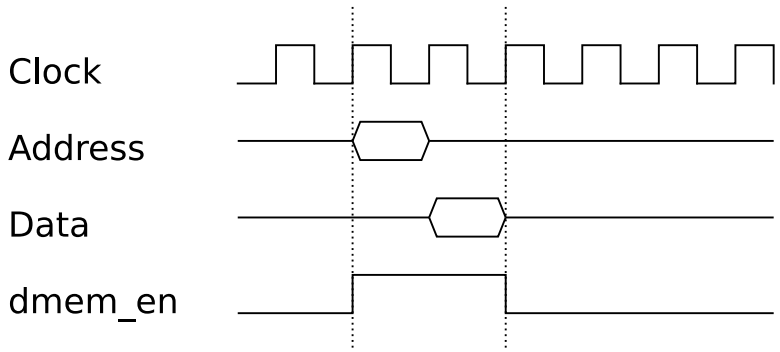


Figure 4.9: Local memory read, no delay.

Latency during an OPB access can be many more cycles, and not only because of increased latency from the slave. When multiple masters attempt to use the bus at the same time, the master with a lower priority will be forced to wait while the master with higher priority completes its transaction. Figure 4.11 shows the two masters from Figure 4.8 attempting to access the bus at the same time. Master 1 obviously has a higher priority than Master 2, as Master 2 is denied access and must wait until Master 1 is done its transaction.

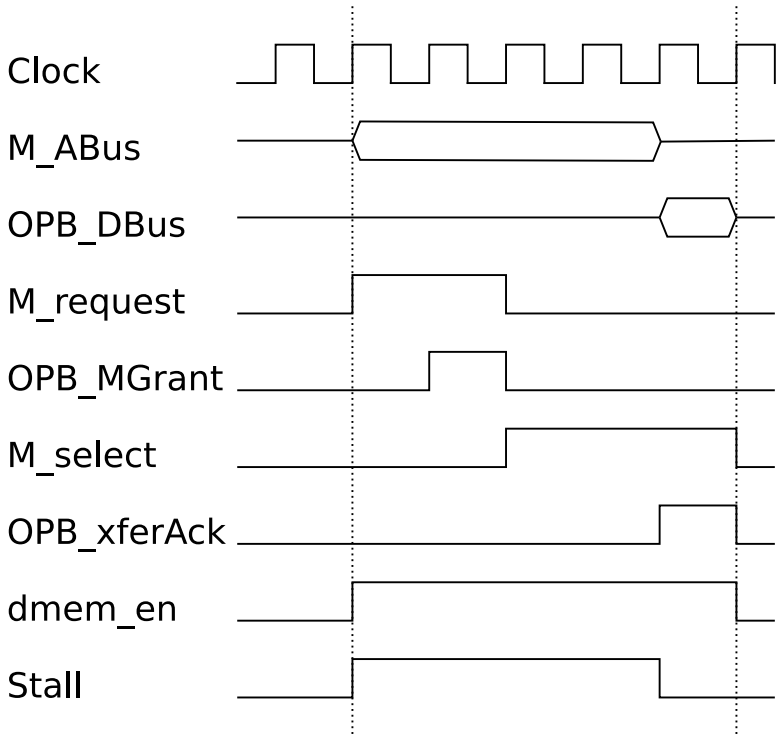


Figure 4.10: OPB read, two cycles of latency.

4.5 Adding OPB Support

The OPB master core was implemented as a state machine mapped into the OpenFire's data memory space in order to keep changes to the OpenFire core as minimal as possible, and to make OPB access as transparent as possible to the software designer. This implementation structure allows the software designer to directly address peripherals, although it has the disadvantage of hiding a performance penalty. Each time the processor accesses the OPB, the processor is stalled until the operation completes.

4.5.1 OpenFire OPB Implementation

A piece of hardware is needed to allow the OpenFire to access the OPB bus as a master and perform protocol translation between the data memory bus of the OpenFire and the

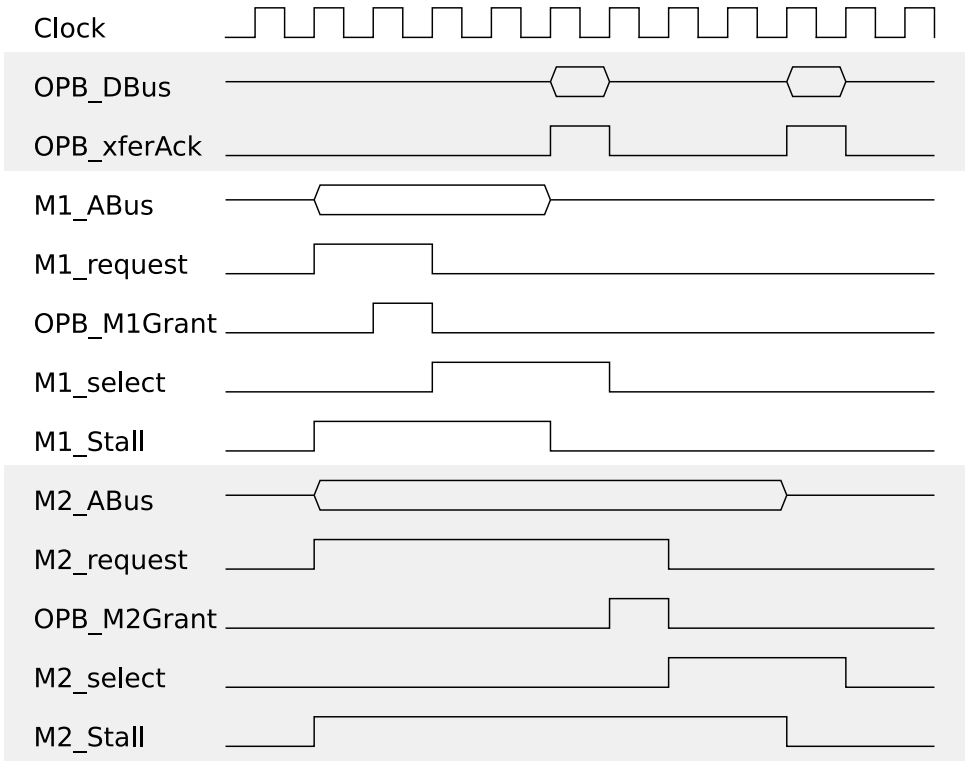


Figure 4.11: OPB arbitration between masters 1 and 2.

On-chip Peripheral Bus. In addition to the OPB signals listed in Section 4.4.1, the OPB state machine also has the following signals to interface with the OpenFire:

OPB state machine signals

`PROC_enable` is a 1-bit input that allows the OpenFire to signal that an OPB transaction is requested.

`PROC_we` is a 1-bit write enable from the OpenFire.

`PROC_ABus` is the 32-bit data memory address bus coming from the OpenFire.

`PROC_DBus` is the 32-bit data memory data bus coming from the OpenFire.

`OPB_wait_on_others` is a 1-bit input signal that prevents the OPB master core from starting another transaction until another core in the same design has finished. This signal is intended to be tied to the `OPB_busy` output of another OPB core and will be discussed further in Section 4.5.2

`OPB_busy` is a 1-bit output signal that is high when the OPB master core is performing a transaction.

`PROC_stall` is a 1-bit output signal that is passed to the OpenFire's stall input, forcing the OpenFire to wait until the most recent OPB access is complete.

`PROC_ack` is a 1-bit output signal that is high for one cycle following the completion of an OPB access.

`OPB_data2PROC` is a 32-bit output bus that sends read data back to the OpenFire's data memory data bus.

The OPB master core was implemented as a state machine, as shown in Figure 4.12. `STATE_WAIT_EN` is the initial state for the OPB master state machine. The state machine waits in this state until it receives an enable signal from the memory mapping hardware, signalling that the OpenFire has attempted to access an address within the OPB address range. In this state, all output signals to the OPB bus are held low because the bus has a wired-OR structure. If any of those signals were allowed to vary, it could disrupt ongoing transactions from other bus peripherals. During the waiting period, the state machine continually registers the contents of the `PROC_ABus` and `PROC_DBus` coming from the OpenFire, in order to be ready when the processor requests OPB access.

STATE_REQ_ACS becomes the active state on the rising edge of the clock after the PROC_enable signal is asserted. The state machine remains in this state while the bus arbitration takes place. During this state the PROC_stall output signal is asserted to halt the OpenFire until the OPB access has completed. The M_request signal output to the OPB is also high, requesting access to the bus. The OPB_busy output signal is also asserted, which can be used to indicate to other OPB master cores that one of the cores attached to this processor is attempting to access the bus, helping prevent bus contention.

STATE_TXRX becomes the active state on the rising edge of the clock after the OPB_MGrant signal is asserted. This indicates that all prior transactions on the OPB have completed and this master now has permission to transmit. As soon as the OPB_MGrant signal is asserted by the arbiter the M_request signal is negated by the OPB master core. The M_select signal is asserted in this state to indicate that this master has control of the bus. While in this state, all of the data lines out to the bus are allowed to take on the values coming from the OpenFire. M_ABus and M_DBus are set to the registered values of PROC_ABus and PROC_DBus that were captured during STATE_WAIT_EN. The M_RNW output signal is set to the inverse of the PROC_we, and the M_BE signal is set to 0xF. In order to allow byte or halfword access to the OPB, the OpenFire's execute pipeline stage will need to be modified to output the proper control bits depending on whether a whole word, half word, or byte operation is begin performed. In Version 0.3b of the OpenFire, a read-modify-write tactic is used to perform partial-word writes, which does not provide enough information to set the M_BE signal. At the moment only word access of the OPB is supported.

STATE_RETRY becomes the active state on the rising edge of the clock after the OPB_retry signal is asserted. In order to retry an OPB access, all outputs to the bus are again brought to zero. The PROC_stall and OPB_busy outputs to the processor remain high, and at the next clock cycle the state machine transitions back to the STATE_REQ_ACS state. This interruption in service is invisible to the processor, unless the running code is keeping track of the amount of time it takes for this OPB transaction. The retry condition is slightly dangerous, as it

can hang the processor indefinitely if the condition that caused the retry persists.

STATE_DONE becomes the active state on the rising edge of the clock after the OPB_xferAck or OPB_timeout signals go high. In the case of the OPB_xferAck signal, the peripheral has completed the requested action. If the previous transaction was a read from the bus, the requested data is returned on OPB_DBus and is registered into the OPB_data2PROC output to the OpenFire. In the case of the OPB_timeout signal, the addressed peripheral took more than sixteen cycles to respond to the master's request and failed to notify the arbiter. The timeout signal can also be returned if no peripheral exists at the address given by the processor. The current implementation does not allow software running on the OpenFire to query the OPB master core to determine whether the last transaction successfully completed or timed out. Read operations that time out will return zero.

Figure 4.13 shows how the OPENFIRE_OPB module that implements the OPB master core is mapped into the OpenFire's data memory. Essentially the incoming data from the OpenFire's local memory, represented in the diagram by the OPENFIRE_SRAM block, with the incoming data from the OPB. This is achieved by using the DOPB_access signal as a selector between DOPB_data2PROC, which comes from the OPB, and dmem_data_o, which comes from the data memory. The DOPB_access signal is formed by the product of the most significant four bits of the dmem_addr signal and the dmem_en signal. This effectively maps the OPB bus into the range 0x10000000 to 0xFFFFFFFF, reserving three and three quarters gigabytes from the four gigabyte memory OpenFire memory space for OPB peripherals.

Due to the internal construction of the OpenFire, the dmem_addr output does not always represent a data memory address. If the current instruction is something other than a load or a store, the dmem_addr signal may appear to reference an address in the OPB range without intending to, as the dmem_addr signal is tied to the output of the OpenFire's internal ALU. The dmem_en output from the OpenFire is a modification from Version 0.3b which indicates when the dmem_addr output should be treated as an address by raising dmem_en. For this reason, it is necessary to include the dmem_en signal in the AND logic which creates the

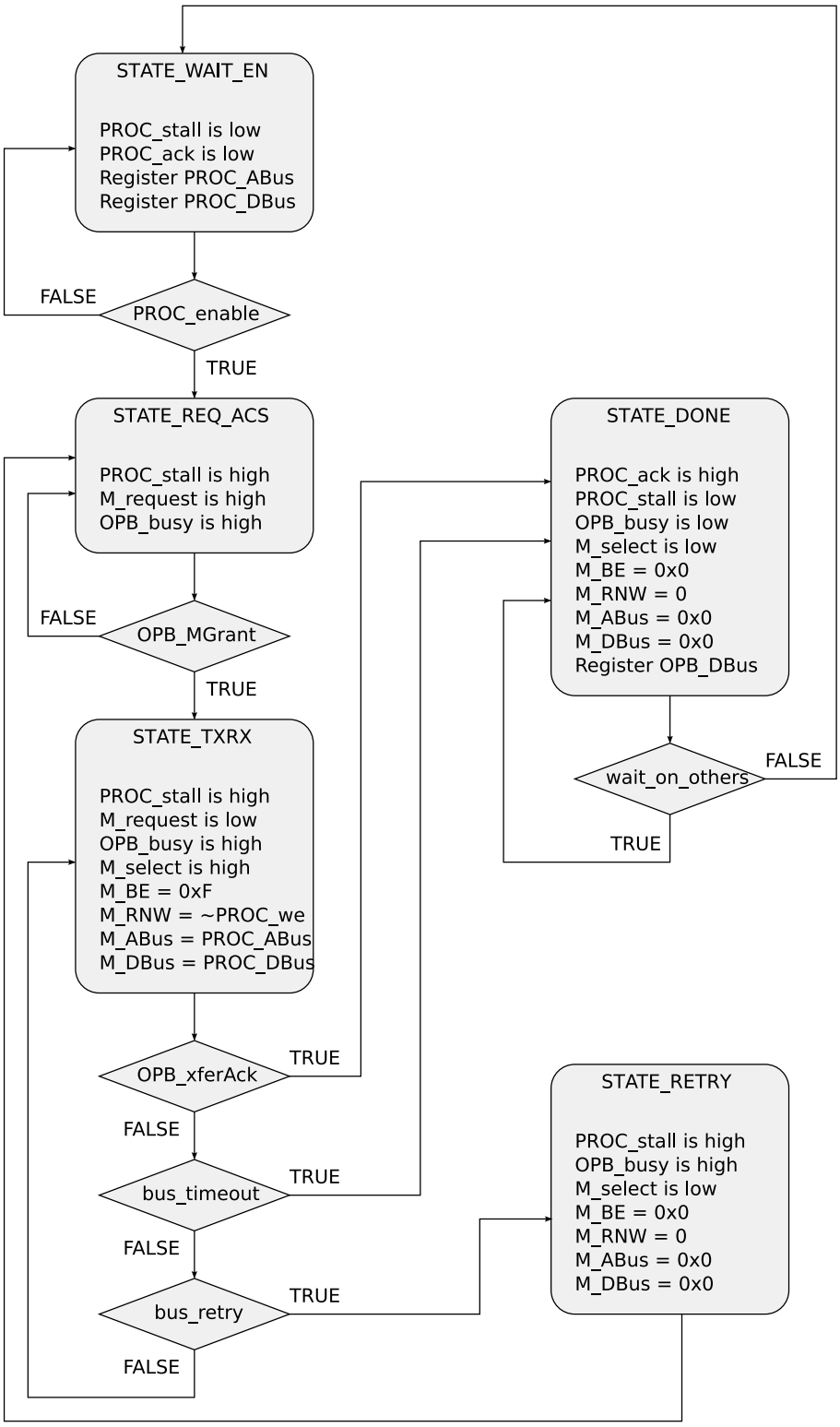


Figure 4.12: OpenFire OPB master state machine.

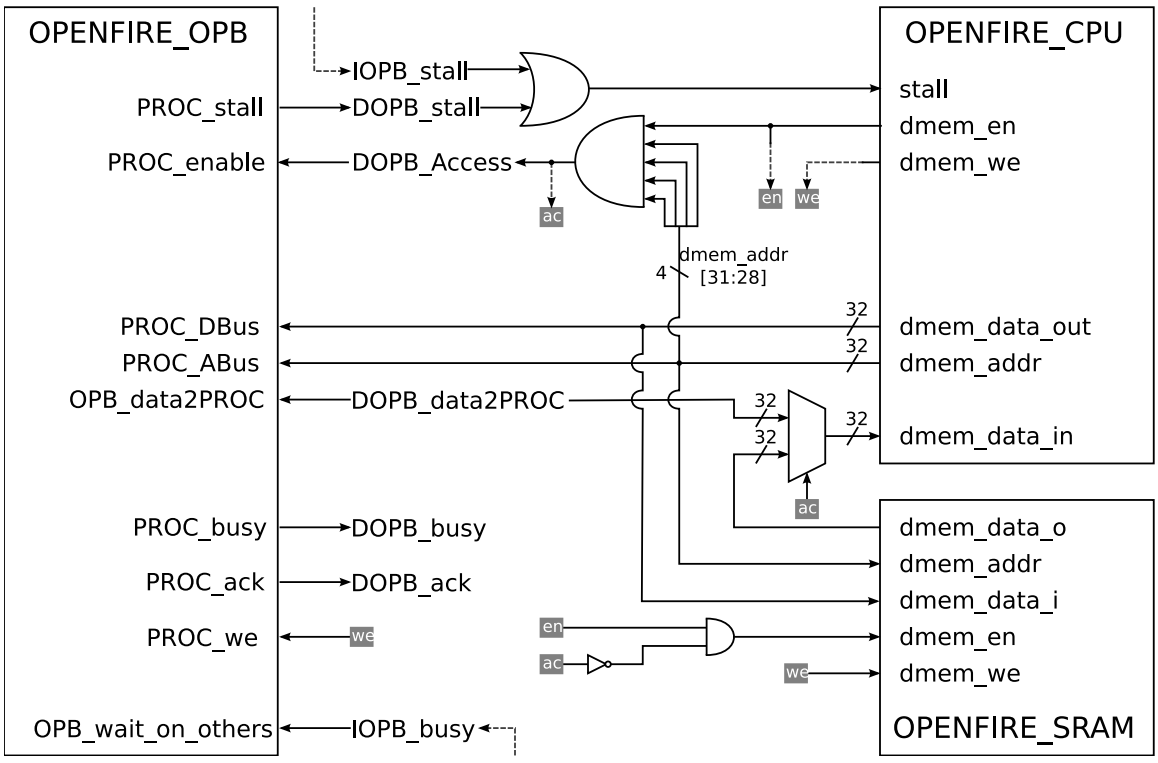


Figure 4.13: OpenFire OPB memory mapping.

`DOPB_Access` signal.

It is also important to note the connections to the OpenFire's `stall` input, and the data memory's `dmem_we` input. The `stall` input is produced by a logic OR of both the Instruction OPB and the Data OPB stall signals. The Instruction OPB is discussed in Section 4.5.2. The `dmem_we` logic is important because it prevents the OpenFire from overwriting portions of the data memory (which also holds the instruction memory) during writes to the OPB. If the OPB is being accessed then the data memory's write enable signal, `dmem_we` is held low.

4.5.2 OpenFire Instruction OPB

Although allowing data memory access to the OPB is a large step towards making the OpenFire more functional as an embedded processor, the MicroBlaze also has instruction memory access to the OPB, a feature that should also be duplicated for the OpenFire. Instruction access to the OPB will allow the OpenFire to use memory on the OPB as instruction memory.

To allow instruction access, the same OPB master core is used. As shown in Figure 4.14, the hardware used for memory mapping the IOPB core into the OpenFire's instruction memory is almost identical to the DOPB hardware shown in Figure 4.13. In the case of the instruction memory, it is not important to protect the local memory block from writing, because the instruction memory bus is never used to write.

Implementing both data and instruction OPB access presents the problem of bus contention. If a program running on the OpenFire is located in memory accessed through the OPB and that program attempts to access data memory located on the OPB then both the IOPB and DOPB modules will be attempting to access the bus at the same time. One of the two will be given priority by the arbiter, but that does not solve the problem, as shown in Figure 4.15. In the figure, the currently running program attempts to read from the DOPB

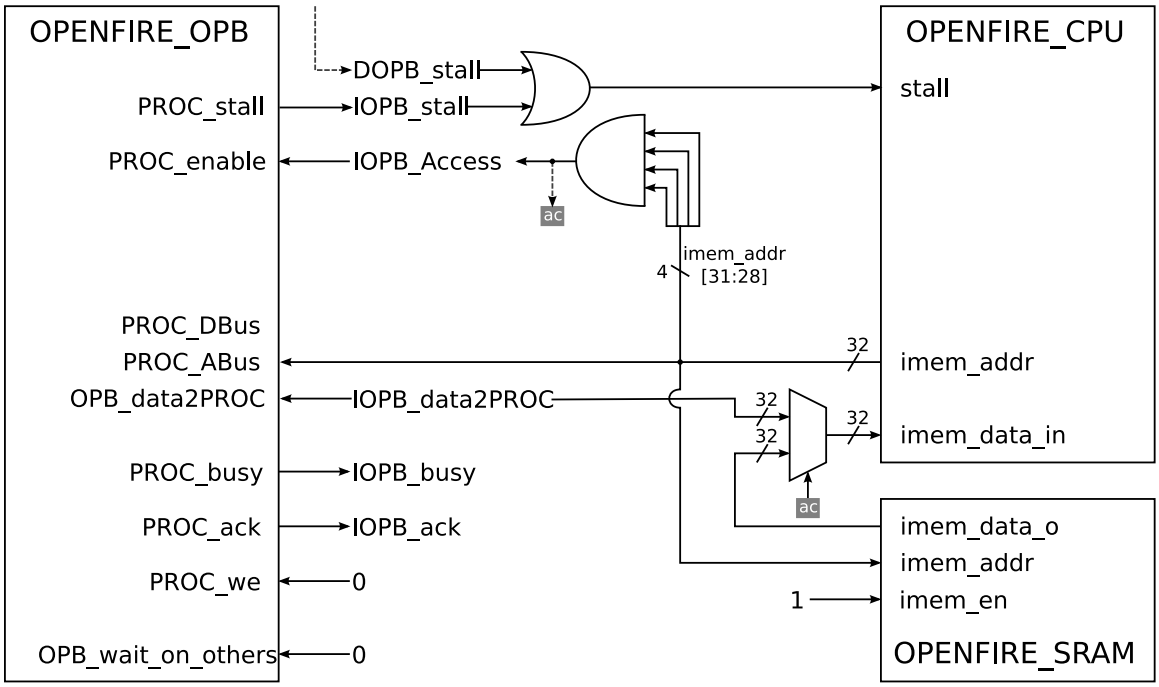


Figure 4.14: OpenFire IOPB memory mapping.

while at the same the IOPB starts fetching the next instruction from memory located on the OPB. The arbiter allows the DOPB core first access, forcing the IOPB core to wait. Once the DOPB transaction is complete the IOPB core is granted access to the bus and begins its transaction. Normally when the DOPB core returns to the waiting state, the stall signal is deasserted and the the host processor is ready to register the incoming data and move on, changing the address expressed on the data memory's address bus. However in this situation the data memory address has not changed, because the processor is still stalled by the IOPB stall signal. This means that the requirements for starting an OPB read still exist on the DOPB input lines, and the very next cycle after the DOPB core enters the waiting, it will re-enter the requesting bus access state. This of course is a circular problem, because when the IOPB core returns to the wait state the processor is being stalled by the repeated DOPB access. Therefore, the IOPB core also re-enters the requesting bus access state, locking the OpenFire into a perpetually stalled state where the two OPB cores continually repeat the last requested action, jointly participating in a cycle that freezes the processor.

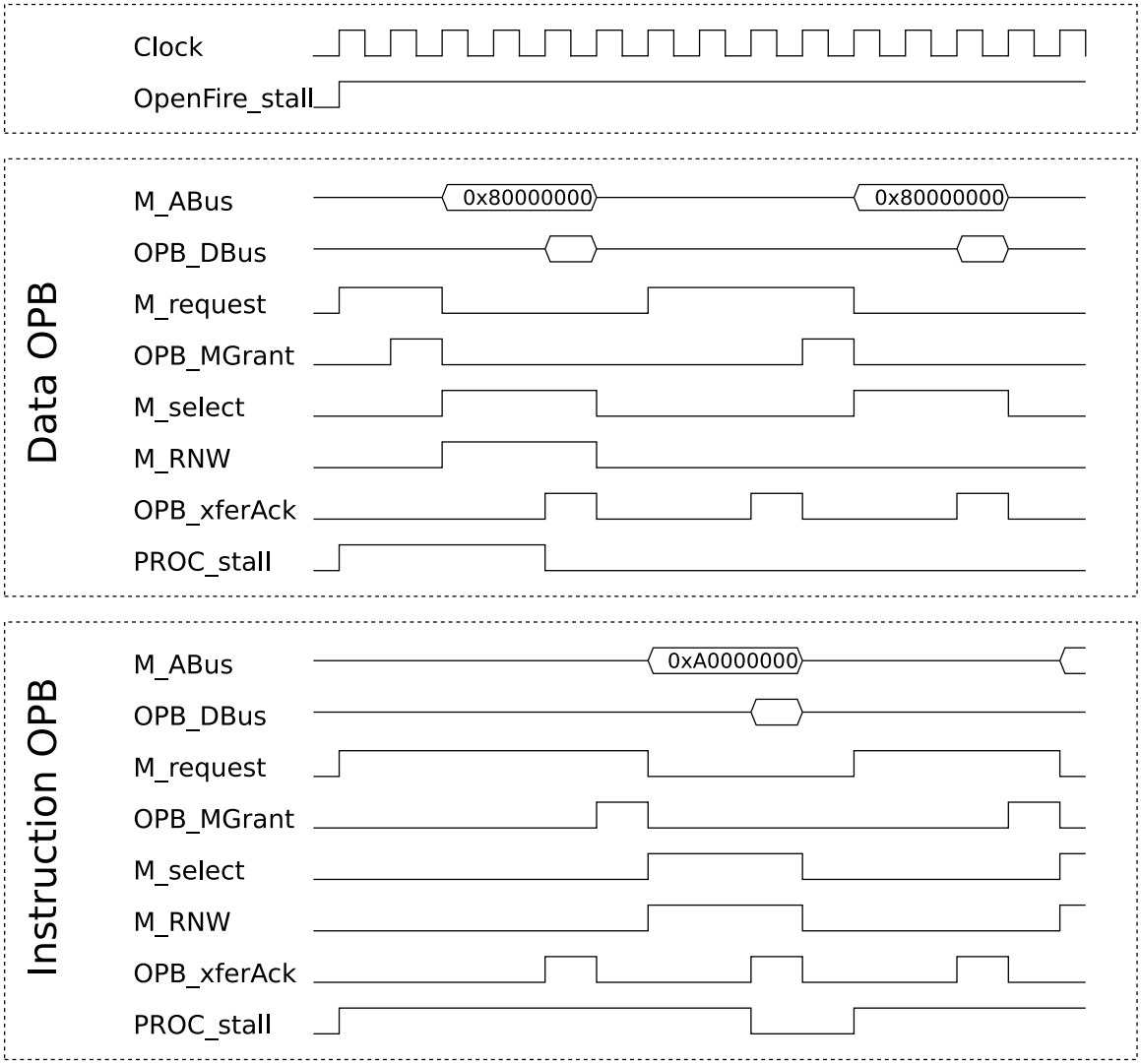


Figure 4.15: Contention between the IOPB and DOPB cores.

The solution to this bus contention is to prevent one of the cores from attempting another transaction if any other OPB master cores are currently waiting for bus access. This modification does not prevent a core from starting a transaction if another OPB core is waiting for access, but it keeps the finished core waiting in the done state until the busy core is finished. This solution is accomplished by adding the `OPB_busy` output and the `OPB_wait_on_others` input to the OPB core. In this implementation, the `OPB_busy` output of the IOPB core is attached to the `OPB_wait_on_others` input of the DOPB core. This causes the DOPB instantiation of the OPB master core to wait until the IOPB core has completed its current transaction, as shown in Figure 4.16.

The essential difference between figures 4.15 and 4.16 is that in Figure 4.16 the DOPB core waits on the `OPB_wait_on_others` signal, which is tied to the output of the IOPB `OPB_busy` signal, before starting another OPB access. Due to this cooperation, the stall input to the OpenFire is allowed to go low for once cycle, which is enough for the processor to advance past the instruction which caused the DOPB core to activate.

Several HDL bugs in the internal stall logic of the OpenFire were corrected to allow the stall input to work as advertised. Version 0.3b of the OpenFire did not store the Arithmetic Logic Unit (ALU) values properly during a stall state, which caused instruction stalls to fail, setting the processor into an unstable state. This issue was corrected during the debugging of the IOPB core.

4.5.3 OpenFire Instruction OPB Performance Comparison

To test the new OpenFire configuration, a program was designed that ensured simultaneous utilization of the DOPB and IOPB cores. The program's secondary function was to time a series of transactions and measure the average performance degradation caused by OPB delay. The test hardware setup is shown in Figure 4.17, and was implemented on a Xilinx XUP board. The system clock was set at 50MHz. The code running on the OpenFire is

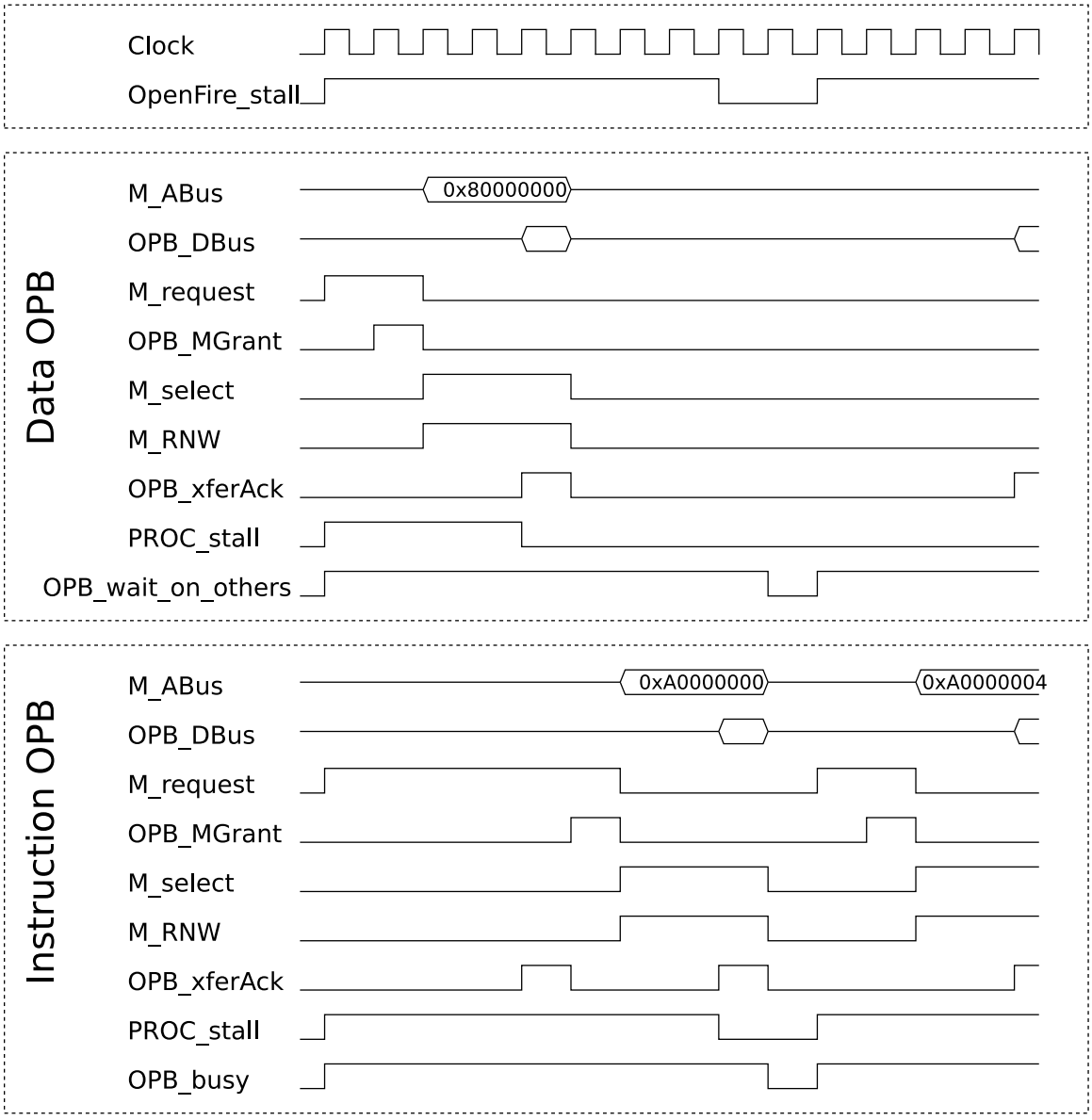


Figure 4.16: Cooperation between the IOPB and DOPB cores.

shown in the appendix on Page 85. The code starts by initializing the DDR memory with an array of `array_sz` numbers. The summing function is then run `num_tests` times on that array and the time it takes is recorded in `timetaken`. The average time is found by dividing `timetaken` by `num_tests` and is then printed out via the UART. Using a function pointer that points into the DDR, the summing function code is then copied into the DDR. Once the code is copied, the summing test is again run `num_tests` times on the data stored elsewhere in the DDR. After the test completes the average time is again computed and printed out via the UART. Test results show that the average time to completion varied by a factor of twelve, meaning that code run from the DDR over the OPB runs twelve times slower than code that is local to the OpenFire's internal program memory. This is an expected value, as testing showed that the OPB DDR peripheral provided by Xilinx takes twelve cycles to respond to read requests. Higher performance could be obtained by utilizing burst transactions accross the OPB; however, that would require addition of instruction and data caches to the OpenFire, as well as the addition of burst support to the OPB master core.

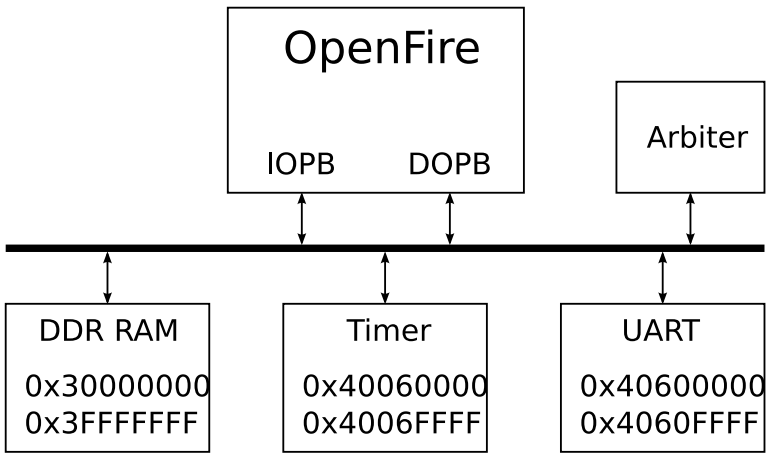


Figure 4.17: IOPB / DOPB testing hardware.

Chapter 5

Multi-core Object Tracking

The object tracking algorithm discussed in Chapter 3 must be implemented both in hardware and on a network of embedded OpenFire processor(s) to compare the two implementations.

The general strategy is to implement the algorithm in hardware first and gain a deeper understanding of the implementation steps that are required and then transfer the hardware implementation into software. The actual transfer of the algorithm to software will require partitioning the hardware algorithm into a series of operation blocks that can be performed by processors, and the connection of a set of processors into an appropriately shaped network. Typically this process would be done in reverse, moving from software to hardware in order to realize a speedup over software, however in this case it is not necessary to show speedup, but merely to attempt to show parity between the two implementations.

Although color has been chosen in this case to act as a discriminating feature, the algorithm discussed in Chapter 3 is flexible enough to support a variety of discriminators. The side-by-side comparison of these two implementations will hopefully show that the color-based object tracking algorithm works equally well on both the hardware and the multi-core software platforms. The calculations required for using color as a discriminator are relatively simple, so a side-by-side implementation of a different type of discriminator might produce

different results.

5.1 Hardware Object Tracking

The hardware for the object tracking algorithm is best implemented as a series of discrete modules whose operation may be tested individually and then strung together to form the entire set of calculations.

It is assumed that the input data to the algorithm is in the form of 24-bit RGB color data, accompanied by X and Y coordinates for each pixel and an End of Frame (EOF) signal. To do a difference function, the user or the system must also supply a desired 24-bit color value, as well as a threshold. The image data is then streamed through the calculation pipeline, allowing real time operation by processing each pixel as it is received. The outputs of the algorithm are bounding box coordinates that are updated each time an EOF signal is received. Figure 5.1 shows a high level block diagram of the modules involved.

The difference engine module is discussed in Section 5.1.1. The erode and dilate modules are discussed in Section 5.1.2. The buffers that carry the X, Y, and EOF data are discussed in Section 5.1.3. The X and Y minimum and maximum modules are discussed in Section 5.1.4. The update module is discussed in Section 5.1.5. The last stage of the algorithm, the coordinate output module, is discussed in Section 5.1.6. All modules are considered to have clock, reset, and enable lines that are not mentioned in the module descriptions. All modules represent a pipelined design that accepts data every cycle while performing a set of operations on the data already inside the module.

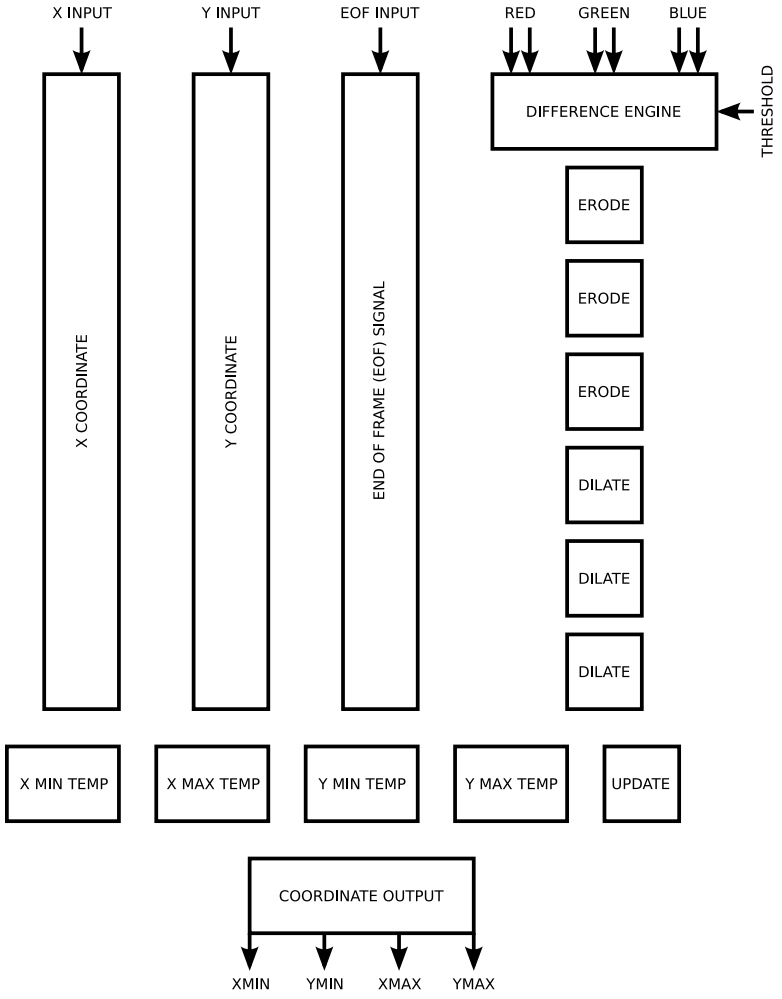


Figure 5.1: A top level view of the hardware algorithm.

5.1.1 Difference Engine

The difference engine, shown in Figure 5.2, is a four stage module that computs the difference between each pixel and the desired color, as discussed in Section 3.3. The inputs to this module are the red (`R_IN`), green (`G_IN`), and blue (`B_IN`) pixel values, the red (`R_DES`), green (`G_DES`), and blue (`B_DES`) desired color values, and the threshold. Each of the incoming pixel values is eight bits wide, while the threshold is sixteen bits wide.

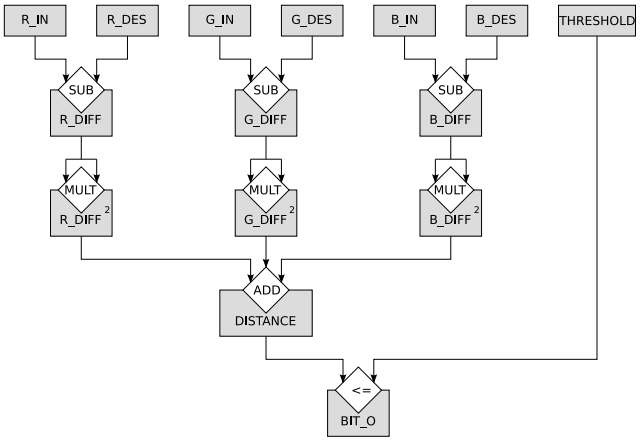


Figure 5.2: The difference engine module.

In the first stage of the algorithm, the incoming desired color component value is subtracted from the corresponding incoming pixel color component value. This forms the difference values `R_DIFF`, `G_DIFF`, and `B_DIFF`, which are shown in Equations 5.1 through 5.3.

$$R\_DIFF = R\_IN - R\_DES \tag{5.1}$$

$$G\_DIFF = G\_IN - G\_DES \tag{5.2}$$

$$B\_DIFF = B\_IN - B\_DES \tag{5.3}$$

$$DISTANCE = R\_DIFF^2 + G\_DIFF^2 + B\_DIFF^2 \tag{5.4}$$

The next step of the module is to form the distance squared term shown in Equation 3.1.

In terms of the variables in this module, the distance squared can be stated as shown in Equation 5.4. The final step of the difference engine module compares the threshold distance to the distance found by Equation 5.4. If the distance is less than the threshold, which is to say that the pixel color is at least that close to the desired color, then the binary output value `BIT_O` is set to one, otherwise it is set to zero. The stream of binary data that is output from the difference engine can be used to construct the difference image shown in Figure 3.2.

5.1.2 Erosion/Dilation

The erosion and dilation modules perform the opening operation described in Section 3.3.2 in order to reduce the noise in the difference image received from the difference engine. The `DATA_I` input reveives bits from the difference engine, and buffers them using the next three stages. The erosion operation, shown in Figure 5.3, is implemented by using a logic AND operation on the three stored pixels, effectively setting a pixel to zero if the pixels on either side of it are zero.

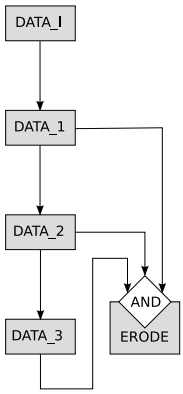


Figure 5.3: The erosion module.

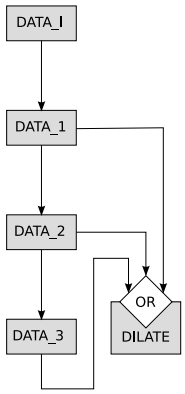


Figure 5.4: The dilation module.

This implementation of the erosion algorithm does not consider the pixels on the previous or next lines, making the implementation one dimensional; however, implementing the erosion and dilation operations like this requires much less storage space. The dilation algorithm is almost identical to the erosion algorithm, as shown in Figure 5.4. The logical AND

operation is substituted for the logical OR operation, effectively turning a pixel on if either of the pixels next to it are on. Both the erosion and the dilation modules have two cycles of delay from input to output. The output pixel corresponds to the pixel in the second of the three delay registers. To achieve more than one level of erosion or dilation, the erosion and dilation modules may be stacked, as demonstrated in the algorithm overview shown in Figure 5.1.

5.1.3 Coordinate Buffers

The coordinate buffers are designed to store the X and Y values, as well as the EOF signal that correspond to pixels traversing the difference engine and erosion/dilation stages of the algorithm. The X, Y, and EOF signals must be delayed the appropriate number of cycles so that when the noise-reduced difference engine image reaches the modules keeping track of bounding box statistics the X, Y, and EOF signals are properly aligned with the pixel data. In this implementation there is one difference image, and six erosion/dilation modules, for a total of sixteen cycles of delay. The coordinate buffer module is simply an N-bit wide shift register with M cycles of delay. In this case the X and Y coordinates are stored in separate coordinate buffer modules with N equal to ten and M equal to sixteen. The EOF signal, being only one bit wide, is stored in a coordinate buffer module with N equal to one and M equal to sixteen.

5.1.4 Coordinate Minimums and Maximums

The bounding box statistics are kept by the coordinate minimum and maximum modules. This module is instantiated four times to keep track of the x and y minimum and maximum values for the current frame. Parameterization allows the same module definition to handle both the minimum and the maximum calculation. As an example the x minimum module is shown in Figure 5.5.

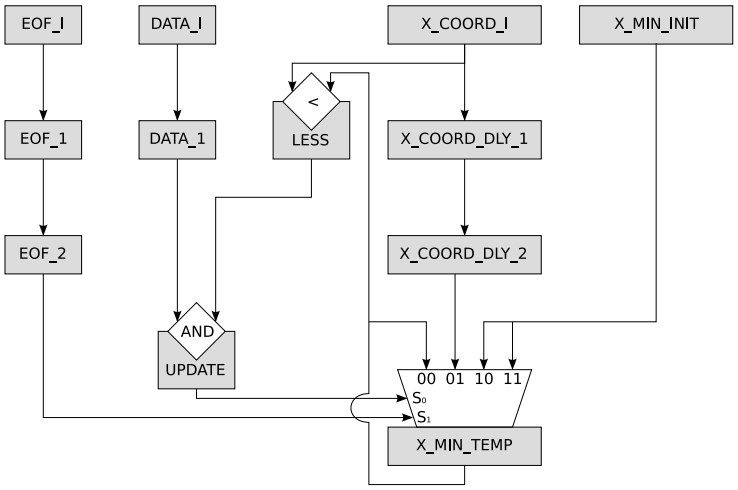


Figure 5.5: The X coordinate minimum module.

Each of the min/max modules requires four inputs: `EOF_I` to indicate when the current frame has ended, `DATA_I` for the pixel value from the thresholded difference image, `N_COORD_I` for either the X or Y coordinate value, and `N_(MIN/MAX)_INIT` for the initial value to which the incoming coordinates are compared. In the first stage of the module, the EOF, data, and coordinate inputs are all registered. The incoming coordinate is also compared to the current output value (in the case of the x minimum module that is `X_MIN_TEMP`) and the result is stored. In the case of a module looking for X or Y minimums, the result (show in Figure 5.5 as the `LESS` register) is one if the incoming coordinate is less than the current output and zero otherwise. The reverse would be true if the module is being used to track maximum coordinate values. In the second stage of the module, the EOF signal is again delayed, along with the incoming coordinate value. The calculation in the second stage is whether the current min or max should be replaced by the incoming coordinate. If the incoming coordinate is less (or greater, as appropriate) than the currently stored value and if the incoming pixel (`DATA_I`) is not zero then the `UPDATE` register stores a one, otherwise it stores a zero. The output of the `UPDATE` register and the twice delayed EOF signal are used as inputs to a multiplexer that chooses the new value of the `N_(MIN/MAX)_TEMP` register. If the EOF signal has been received then the output register is reset to the initial value input to the module. Otherwise, if the currently stored value should be replaced by the incoming

coordinate value from two cycles ago, the delayed form of that coordinate is registered. If not, then the output register's value stays the same.

5.1.5 Output Update Decision

The update decision module is important because it is possible that no pixels close enough to the desired color may be found in the current frame. In that case, it is better to not change the coordinate output from the system, as those coordinates would directly result from the inital values set for the minimum and maximum modules. The update decision module, whose implementation is shown in Figure 5.6, detects whether any good pixels exist in this frame's thresholded difference image.

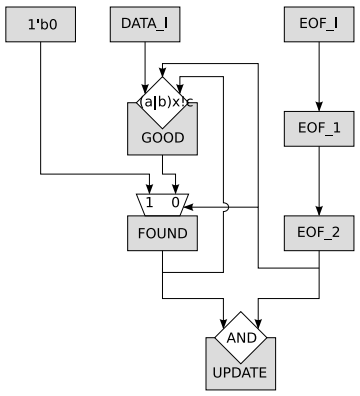


Figure 5.6: The update decision module.

In the first stage of this module, the `GOOD` register is set to one if the incoming data is good, or if the `FOUND` register already indicates that a pixel has been found in this frame. The results of the `GOOD` register are stored in the `FOUND` register in the second stage, providing a feedback loop until the received EOF signal clears the `GOOD` and `FOUND` registers. It is necessary to clear both registers at once to avoid a condition that would keep the EOF from being reset. The one bit update signal that is the output of this module goes high if and only if the `FOUND` register is one when the EOF signal is received. Therefore if there were no good pixels detected during the current frame the EOF signal merely resets the update

decision module without producing a high `UPDATE` output.

5.1.6 Coordinate Output

The coordinate output module, shown in Figure 5.7 is a very simple module that buffers the output X and Y bounding box coordinates. The X and Y minimum and maximum statistics from the modules discussed in Section 5.1.4 are provided as inputs, as well as the update signal from Section 5.1.5.

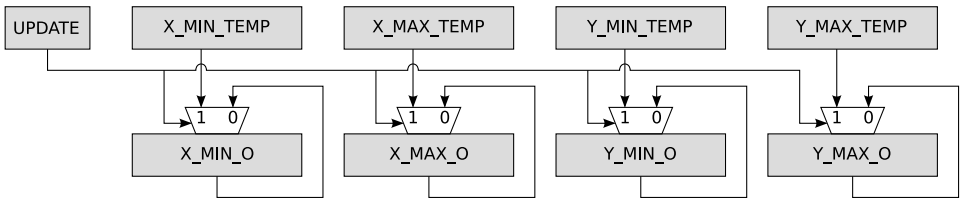


Figure 5.7: The coordinate output module.

When the update signal is high, the `X_MIN_O`, `X_MAX_O`, `Y_MIN_O`, and `Y_MAX_O` output registers store the corresponding "temporary" input value. This module provides a steady output from the entire algorithm that represents the bounding box found in the last frame that contained pixels close enough to the desired color.

5.1.7 Throughput

Since each module discussed in this section is pipelined, they each support a continuous stream of pixel data. The difference engine for example performs the eight operations necessary to execute the difference and threshold algorithms on a single pixel in four cycles. During that time it has also performed seven of the operations for the following pixel, six of the operations for the pixel after that, and three operations for the pixel three clock cycles removed from the output pixel. This is in addition to the calculations that were performed for pixels that were already in the pipeline. In all, the difference engine can perform eight

operations per cycle. This utilization of gate level parallelism will make it difficult for a processor to compare in terms of performance.

5.2 Software Object Tracking

The entire object tracking algorithm described in Chapter 3 and implemented in hardware in Section 5.1 could conceivably be implemented in a single OpenFire, the processor would be at a severe performance disadvantage. As mentioned in Section 5.1.7, just one module of the hardware algorithm can perform eight operations per cycle on a group of pixels. The processor is limited to one operation per cycle and can not work on more than one pixel at a time due to a 32-bit wide data path. To achieve acceptable performance rates, it is necessary to distribute the object tracking algorithm tasks amongst a network of OpenFire processors.

5.2.1 Processor Network Structure

To implement the algorithm realized in hardware as a multi-core software algorithm, the calculations must be compartmentalized in such a way that they can be executed in parallel by the network of OpenFire processors. How the algorithm is compartmentalized is a function of how the network of processors is structured.

The two general network architecture types that could be applied to this problem are parallel networks and sequential networks. In a parallel network, each processor in the network would receive a pixel and apply the entire algorithm to that piece of data. With enough processors in the network a high enough throughput rate might be achieved. However, this approach is not practical when considering the data flow for the algorithm in question. It is necessary during the erosion and dilation stages to have the information for not just the current pixel but the three pixels on either side, as three iterations of erosion and three iterations of dilation are performed. Using a parallel network would then require redundant

distribution of data, something that would reduce the efficiency of the implementation. Also, the more processors working in parallel, the greater a problem it is to synchronize their outputs.

A sequential network is an arrangement of processors that is similar to the pipelined module arrangement used in the hardware implementation of the object tracking algorithm. Given four OpenFire processors, the first processor could perform the difference engine operation, the second processor could perform all three erosion stages, the third all the dilation stages, and the fourth OpenFire could keep the statistics. This network arrangement provides a speedup over a uniprocessor implementation by reducing the amount of work any one processor has to perform before reading in another pixel. The multi-core software implementation for this design takes the general form shown in Figure 5.8

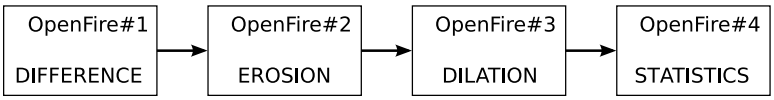


Figure 5.8: Algorithm partitioning over a sequential OpenFire network.

5.2.2 Processor Network Interconnection

Different types of processor interconnect are available to a sequential network. Perhaps one of the most conceptually simple styles of interconnect is shared memory. Figure 5.9 shows a candidate network arrangement that has each OpenFire sharing the same memory via the OPB. Some communication would be required between processors to ensure that data was not corrupted by simultaneous accesses; however, a shared memory model for the network would allow the passing of large blocks of data between processors by simply sending a pointer to the shared memory.

Although the shared memory arrangement has some advantages, the key problem is latency. With all four of the OpenFire processors accessing the bus the arbitration latency caused when one or more OpenFires are forced to wait to access the bus until the current

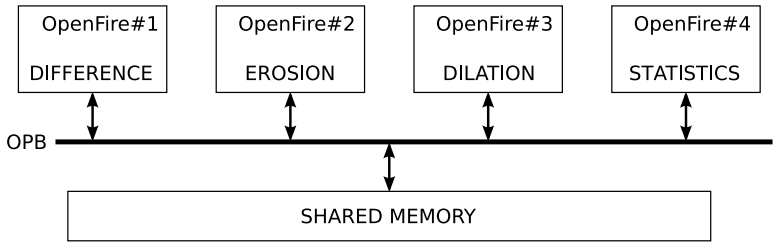


Figure 5.9: Shared memory network.

operation is complete would be quite high. This time, coupled with delay while waiting for the peripheral to respond, would be difficult to accept and still remain competetive with the hardware implementation.

Compared to a shared memory system, a point-to-point network can be a very efficient way to arrange a network of processors. There are some disadvantages, as a point-to-point network does not allow the sharing of large blocks of data unless the source processor sends the entire block to the destination. Also, messages in a point-to-point network may need to go through multiple nodes in order to reach their destination. However in the algorithm partitioning given in Section 5.2.1 only processors that are directly "next to" each other need to communicate, a situation that lends itself well to point-to-point networking.

The FSL bus interfaces added to the OpenFire in Section 4.2 are perfect for implementing point-to-point networks. They contain their own buffers and the OpenFire hardware instructions that access the FSL busses support blocking, which solves the problem of synchronizing the processors in the network. Also, the FSL busses allow for asynchronous input and output clocks, which means that the processors can be run at a higher clock speed than the pixel clock of the incoming image data, using the FSL bus to buffer those pixels that come in while the the previous pixel is being processed.

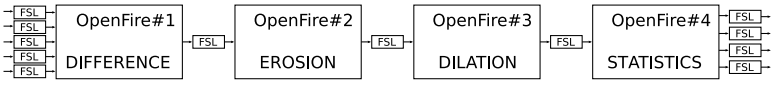


Figure 5.10: FSL-based point-to-point network.

Figure 5.10 shows a network structure that would use the OpenFires' FSL ports to implement a four processor point-to-point network for performing the object tracking algorithm. Five input FSL busses would be enough for OpenFire#1 to input the RGB color data, the desired color data, the threshold, and the pixel coordinates. Four output FSL busses would allow OpenFire#4 to output the X and Y minimum and maximum coordinates for the bounding box. A single FSL connection between the rest of the processors should be sufficient to pass the binary pixel data from the difference image. The more FSL ports a processor has, the more time it takes to receive a set of data, however due to the low level integration of the FSL ports into the OpenFire's instruction set, each FSL access is a single cycle, unlike the memory accesses in a shared memory system.

5.2.3 Modified Data Flow

Some changes can be made to the input FSL structure in order to use fewer cycles to read in the pixel data and make the OpenFire network more efficient. The incoming RGB values for the pixel color and the desired color are each 8 bits wide. The three 8-bit color values and the 8-bit red component of the desired color can be packed together into one 32-bit FSL bus. The 16-bit threshold value can then be packed into the upper 16-bit FSL bus shared with the 8-bit green and blue components of the desired color. The 10-bit X and Y coordinates can also be packed into the lower and upper halves of a 32-bit FSL, respectively. This decreases the number of FSL busses to three, freeing up two clock cycles per pixel in the OpenFire algorithm.

One signal that is missing from the processor FSL list that is supplied to the hardware algorithm is the EOF signal. Although it might be possible for OpenFire#4 to detect the EOF condition on its own by testing for a condition where both the X and Y coordinates are zero, it is also possible that due to buffer overflows the X and Y equal to zero condition may be missed. In order to prevent this error condition the EOF signal is sent directly to

OpenFire#4 which can test for the EOF condition using a non-blocking read to that FSL. These two modifications yield the modified OpenFire network shown in Figure 5.11.

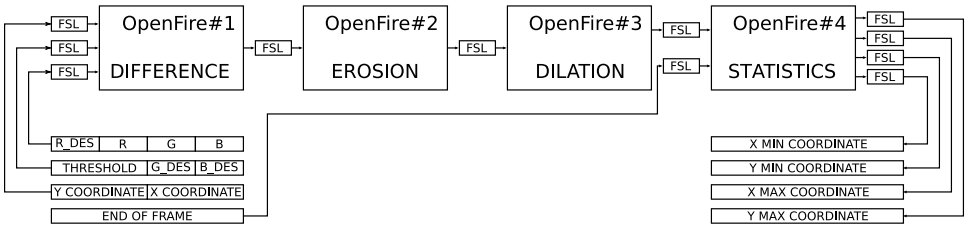


Figure 5.11: Modified implementation of the point-to-point network.

5.2.4 Algorithm Software

With the algorithm partitioned between processors, the task remains to translate the algorithm into optimized code for each of the four OpenFires. Several techniques that involve processor specific instructions can be used to minimize the number of cycles each program spends on its pixel value, maximizing the throughput of the software. These optimizations do not necessarily mean that the programs need to be coded in assembly, but rather that smart C-style coding that has an eye towards the assembly implementation can save cycles.

One such optimization is the use of pointers to enforce halfword or byte reads rather than masking and shifting to load a particular byte or halfword. This becomes important when manipulating the packed input data to the processors.

```
unsigned thresh_desired;
unsigned short * thresh = &(((unsigned short *)&thresh_desired)[0]);
```

Figure 5.12: Example C code for forcing halfword reads.

In the code shown in Figure 5.12, when the `thresh` pointer is dereferenced, it will result in a halfword read of the most significant half of the `thresh_des` variable. The `thresh_des` value is assumed to have been retreived inside of OpenFire#1 as depicted in Figure 5.11 and therefore contains the threshold as well as the blue and green components of the desired

color. It is important to note that the byte numbering for the OpenFire is Big Endian, with the most significant byte or halfword being the zeroth byte or halfword inside a 32-bit word.

Another optimization technique is the usage of branch commands that test whether a register is zero[12]. This can be used to treat the packed X and Y coordinate data for pixels that make it through the difference engine as the "binary" pixels that make up the thresholded difference image. That way, instead of sending both binary pixel and a coordinate pair through the erosion and dilation stages, just the coordinate pair is sent. If a coordinate pair is zero, then the pixel will cause other pixels around it to be set to zero during an erode. If the pixel is not zero, as tested by a "branch not equal" command, then it does not cause those around it to be eroded.

Sending the coordinate pair instead of a binary pixel also saves time as the pixel works through the processor network because it means that the data for eroded pixels does not need to be passed on to the dilation stages. No information is lost because the next non-eroded pixel that makes it through the erosion stages will contain its own coordinate data.

The time saving technique of substituting coordinate pairs for binary pixels does create a problem with the dilation stages. Once pixels have been eroded away and subsequently dropped instead of being passed on, it is necessary to insert the pixels created during a dilation operation into the output stream of a dilate module. However since only the maximum and minimum pixels created by a dilation operation matter, due to the fact that the statistics engine is recording only minimum and maximum values, it becomes possible to simplify the dilation operation into two additions - one that creates the new pixel three pixels to the left of the input pixel, and the other that creates the new pixel three pixels to the right of the input pixel. These two new pixels can then be passed statistics engine.

It was also discovered, mostly by accident, that the `mb-gcc` compiler used to cross compile code for the MicroBlaze and OpenFire processors does not handle the ternary operator well. An example of the assembly difference between a C implementation using the ternary

operator verses a C implementation using an if-else statement can be found on Page 101 in the appendix.

The C code for OpenFire#1, performing the difference engine algorithm, can be found on Page 88 in the appendix. The C code for OpenFires 2, 3, and 4 can be found on pages 91, 94, and 96, respectively. The resulting assembly code from the programs for OpenFires 1-4 can be found on pages 89, 92, 95, and 98.

Chapter 6

Hardware vs. Multi-core Comparison

6.1 Implementation Platform

Both implementations of the image processing algorithm described in Chapter 3 were implemented side-by-side in an FPGA based video platform. A framework was built to display the video stream output overlayed with a visualization of the bounding box coordinates generated by the hardware and software implementations.

6.1.1 Implementation Hardware Platform

The hardware platform consisted of a Xilinx University Project (XUP) board[39] connected via a high speed data port to a VDEC1 video decoder board produced by Digilent, Inc[40]. Figure 6.1 shows a labeled photograph of the XUP board attached to the video decoder. A Canon Optura500 digital video camera is used as a video source, attached to the video decoder via an SVideo cable. An onboard 24-bit video Digital to Analog Converter (DAC) on the XUP board is used to produce output to be displayed on a monitor.

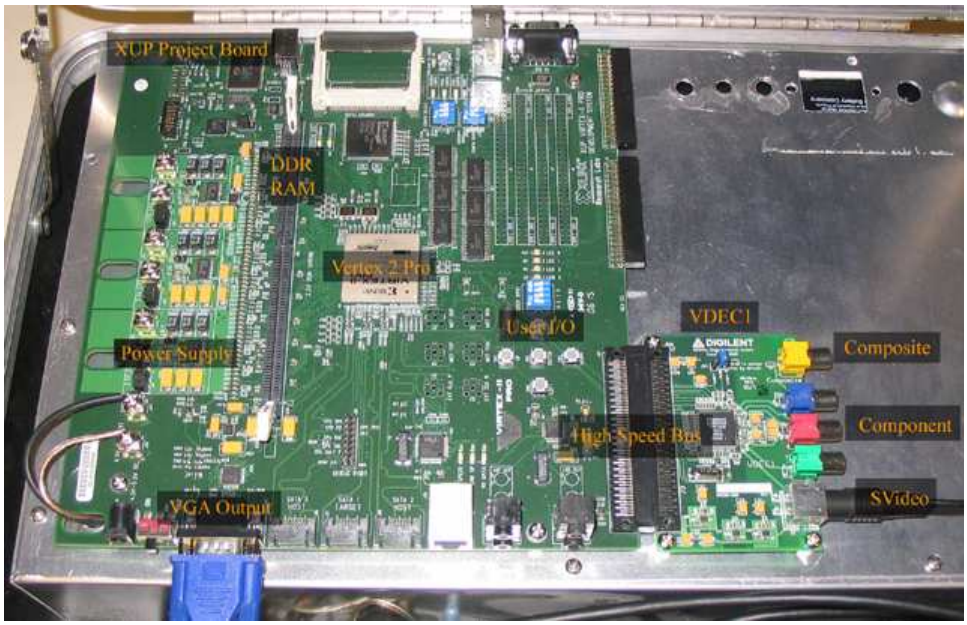


Figure 6.1: The XUP platform with attached VDEC1 video decoder.

A Xilinx reference design is used to access the VDEC1 board and configure it for use with SVideo inputs. The reference design provides the color and pixel coordinate outputs used by the image processing algorithm, after converting the YUV data received from the camera into the RGB color space. The video data is provided in a raw format that does require some processing on the part of the FPGA hardware. The incoming pixel clock from the VDEC1 board runs at 27MHz, producing 60 frames per second at 858 pixels wide and 525 lines tall. However, not all of those pixels are valid color pixels, as shown in Figure 6.2. Around 22% of each frame received from the VDEC1 is composed of non-valid filler pixels that occur during the rescan periods of the National Television System Committee (NTSC) video standard.

Both the hardware and the multi-core implementations can ignore pixels that occur during the front and back porches and the synch period. All three of those conditions cause the video enable line to become negated.

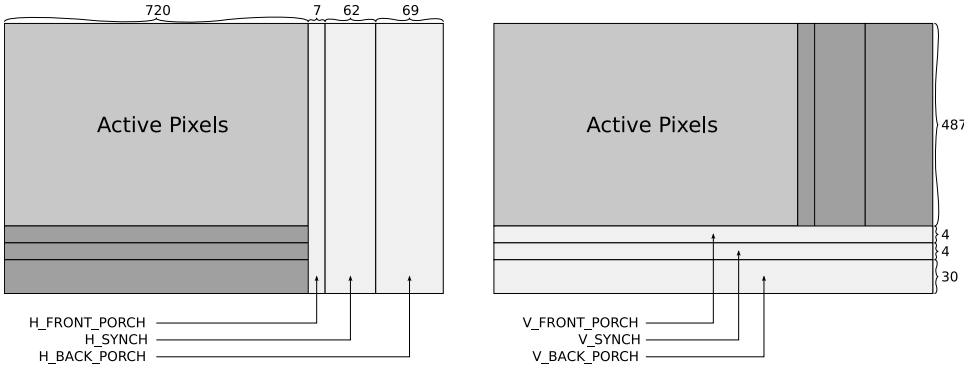


Figure 6.2: Active and inactive video pixel fields.

6.1.2 Implementation Software Platform

The XUP board is host to a Xilinx Virtex II Pro FPGA. This FPGA has 13,696 slices available for implementing logic, 136 18Kbit BRAMs, and two embedded PowerPC 405 processors[11]. One of the two PowerPC processors is in use in this project to create a text overlay for the video output stream to display useful debugging information during the development process, and to display the visualized bounding box in the final product. Figure 6.3 shows the interaction between the incoming video stream from the camera, the overlayed frame buffer, and the PowerPC that controls the text overlay.

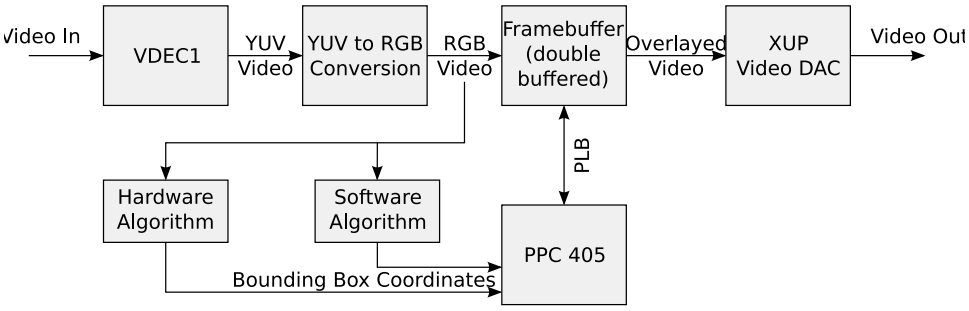


Figure 6.3: Implementation system framework.

In the figure, the PowerPC has a memory mapped bus encompassing the registered outputs of the hardware and software implementations of the image processing algorithm, allowing code running on the PowerPC to read the bounding box coordinates from the two implementations. The coordinates are then used to draw the bounding boxes in the frame

buffer, which is connected to the PowerPC via the Processor Local Bus (PLB). The frame buffer is meanwhile accepting a stream of incoming pixel data from the VDEC1 and the Xilinx reference design.

The frame buffer uses a double buffering scheme that allows the PowerPC 405 processor to write to an inactive buffer while the other buffer is being used to create the output. The frame buffer accepts the incoming video stream and writes the video stream and the active buffer out to the video DAC together, replacing pixels in the video stream with pixels from the frame buffer. The PowerPC draws bounding boxes into each frame frame processed by the frame buffer, which has the effect of overlaying a bounding box based on coordinates calculated from the previous frame's pixel data. At 30 frames per second, this one frame lag between the calculation and display of the bounding box is not visible.

The PowerPC was chosen instead of the OpenFire to perform this task because the OpenFire, as a soft-core processor implemented in an HDL, is not able to run as fast as the PowerPC core, a hard-macro processor embedded in the Virtex II Pro's silicon. The OpenFire's advantage is in numbers, as the designer is limited to the number of Virtex II Pro devices fabricated into the device, whereas the OpenFire may be replicated as long as there is still configurable area available.

6.2 Hardware Performance Details

The implemented hardware algorithm takes up less than ten percent of the total configurable resources consumed on the Virtex II Pro for this design. Although the image algorithm does not discriminate well in cluttered visual environments, the hardware implementation is able to follow quickly moving objects of a specific color in its field of view with minimally visible lag.

6.2.1 Device Utilisation

Table 6.1 shows the hardware used to implement the image processing algorithm along with the total resources available on the device and a percent usage.

Resource	Used	Device Total	Percent Usage
4-bit LUT	252	27392	1%
Flip Flop	676	27392	2.5%
Multiplier	3	136	2%

Table 6.1: Hardware algorithm device utilization.

As shown in the table, a single instance of the hardware image processing algorithm takes very few configurable resources to implement. The timing report generated by the implementation tools states that the algorithm module was placed and routed such that it will meet timing at up to 100 MHz, more than enough to easily handle the 27 MHz video stream.

6.2.2 Visual Performance

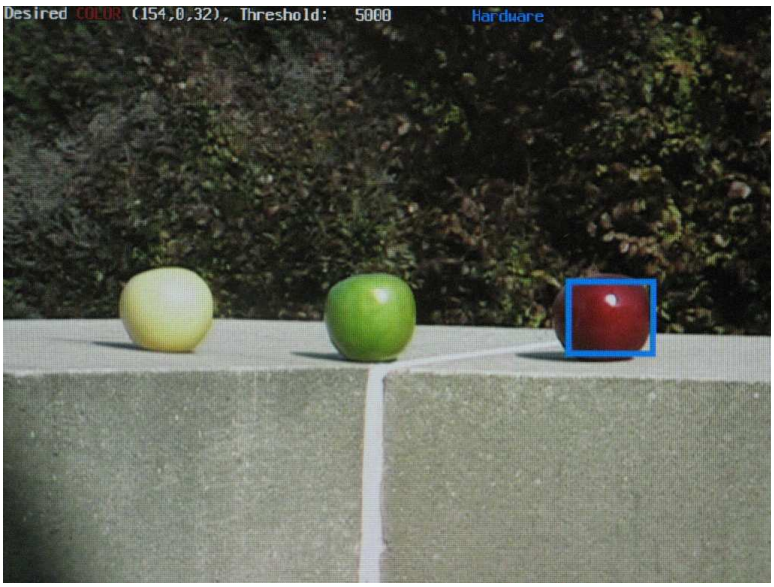


Figure 6.4: The hardware algorithm, attuned to red.

The hardware algorithm's visual performance is very good, as long as the desired color and threshold values given to the algorithm are appropriate to the lighting conditions being experienced by the target. This is a weakness of using the RGB color space to detect color, as variations in lighting drastically change the color values expressed in the video stream.

Figures 6.4, 6.5, and 6.6 show images captured from the hardware algorithm being applied to a video stream depicting three different colored apples. Figure 6.4 shows a bounding box drawn around the red apple when attuned to the RGB color value of {154,0,32} and a threshold value of 5,000.

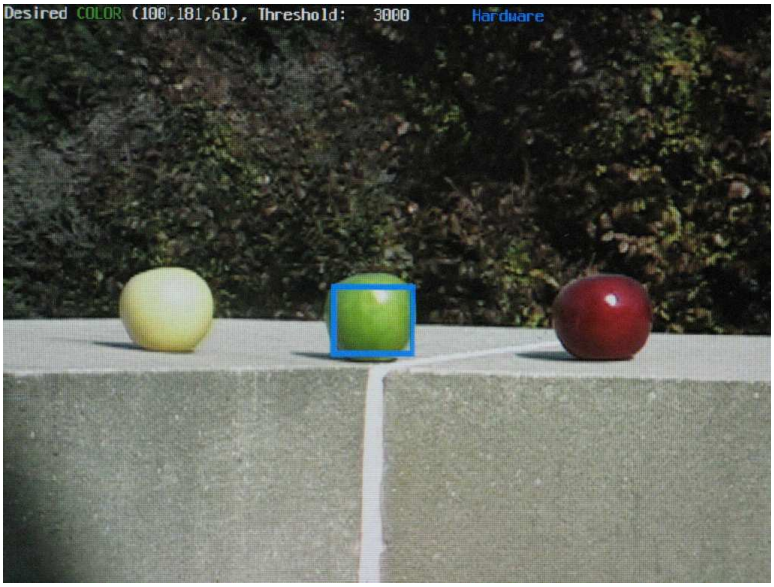


Figure 6.5: The hardware algorithm, attuned to green.

Figure 6.5 shows a bounding box drawn around the green apple when attuned to the RGB color value of {100,181,61} and a threshold value of 3,000.

Figure 6.6 shows a bounding box drawn around the yellow apple when attuned to the RGB color value of {255,180,80} and a threshold value of 5,500. The color values for the yellow and green apples are very close, making it likely that changes in the lighting will cause the algorithm to select both the yellow and the green apples.

Figure 6.6: The hardware algorithm, attuned to yellow.

6.3 Software Performance Details

The multi-core software implementation of the image processing algorithm was implemented with the OpenFire processors running at 50 Megahertz (MHz). More speed was desired, but at present the OpenFire does not reliably support a faster clock. All FSL First In, First Out (FIFO)s were implemented as 128 position Random Access Memory (RAM)s. Although the design was implemented with no timing errors, it initially failed to operate. Even at twice the clock speed the OpenFires failed to keep up with the incoming pixel rate from the video stream, causing the input FSL busses bridging the 27/50 MHz clock domains to overflow.

6.3.1 Pixel Downsampling

The incoming data is sampled at a reduced rate to lower the amount of data that the processors are required to retrieve. Horizontal downsampling reduces the number of pixels per line, and verticle downsampling reduces the number of lines fed to the OpenFires.

Although the incoming pixel rate is 27 MHz, not all of those pixels are valid. Only the pixels that occur in the active range are valid. While the video stream is in any of the other regions the FSL busses do not input data, allowing the processors to deplete the buffer. As shown in Figure 6.2, only 350,640 of the total 450,450 pixels per frame, or 78%, are in the active region. This indicates that the incoming data rate that must be achieved by the OpenFire processors is not 324 MB/s, but only 253 MB/s, as shown in Equation 6.1.

$$12bytes/cycle * 27Megacycles/sec * 78\% = 253Megabytes/sec \tag{6.1}$$

According to the assembly code shown on page 89, the program for OpenFire#1 from Figure 5.11 has a loop period of 42 instructions. All of the load and store instructions are two cycle instructions and the branch instructions take three cycles as their delay bits are not set. The slowest instruction in the program loop is the multiply instruction. Multiplication takes five cycles on the OpenFire, instead of the three cycles it would take on a MicroBlaze. All other instructions require one cycle[12]. Analysis of the assembly code for OpenFire#1 shows that it takes an average of 81 cycles to execute the 42 instruction loop, assuming that the processor does not block while waiting to send or receive data via the FSL ports. During those 81 cycles, 12 bytes of data are read. So OpenFire#1, running at 50 MHz and reading in 12 bytes every 81 cycles has a top input data rate of 7.4 MB/s, as shown in Equation 6.2.

$$12bytes/81cycles * 50Megacycles/sec = 7.4Megabytes/sec \tag{6.2}$$

Five of the bytes that contribute to the incoming data rate of 253 MB/s, namely the three bytes of desired color information and the two bytes of threshold information, could be stored inside OpenFire#1 to reduce that incoming rate. However, as seen in Equation 6.2, reducing the number of input bytes would also reduce the OpenFire's effective input rate because the computation time would not be greatly reduced with the removal of a single

cycle `getfsl` instruction.

	1	2	4	8	16	32
1	253	126.5	63.25	31.6	15.8	7.9
2	126.5	63.25	31.6	15.8	7.9	2.5
4	63.25	31.6	15.8	7.9	2.5	0.01
8	31.6	15.8	7.9	2.5	0.01	0
16	15.8	7.9	2.5	0.01	0	0
32	7.9	2.5	0.01	0	0	0

Table 6.2: Downsampled data rate table.

Table 6.2 shows the resulting data rates from various combinations of horizontal and vertical downsampling amounts. To achieve the 7.4 MB/s or lower data rate for the incoming data and at the same time maintain a similar aspect ratio for the downsampled image, both the horizontal and vertical downsampling factors are chosen to be eight. This results in a data rate divisor of 64, bringing the incoming data rate down to 2.5 MB/s. This data rate is well within the abilities of the multi-core network to process without falling behind and causing the FSL buffer to overflow.

6.3.2 Erosion Removal

Originally the erosion and dilation stages, discussed in Section 3.3.2, were instantiated to reduce the noise generated by the difference and threshold stages of the algorithm. However, downsampling an image has a similar effect. With a horizontal downsampling factor of 8 and a vertical downsampling factor of 4, the active pixel dimensions for each frame reduce from the 720x487 dimensions shown in Figure 6.2 to 90x121, a reduction of more than 90%. Three stages of erosion could completely remove any features that remain, whereas any noise that existed was likely to have been in the 90% of the frame that was removed by downsampling. Figure 6.7 shows a modified processor network that performs the erosion-free algorithm in parallel with the original multicore algorithm.

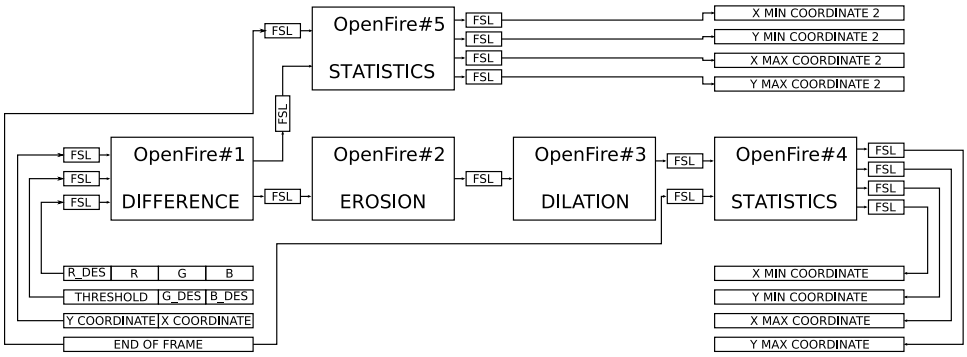


Figure 6.7: A parallel computation that avoids the erosion and dilation stages.

6.3.3 Device Utilisation

With five OpenFire processors and seventeen FSL busses, the multi-core software implementation of the image processing algorithm accounts for almost 50% of the LUT resources used in this design. Table 6.3 details the resources used by a single OpenFire. The eight BRAMs are used by the program and data memory internal to the OpenFire.

Resource	Used	Device Total	Percent Usage
4-bit LUT	1250	27392	4.5%
Flip Flop	415	27392	1.5%
Multiplier	3	136	2%
BRAM	8	136	6%

Table 6.3: OpenFire device utilization.

Table 6.4 shows the resources consumed by a single FSL bus. The FSL busses were implemented using BRAMs in order to use asynchronous input and output clocks.

Resource	Used	Device Total	Percent Usage
4-bit LUT	44	27392	0.2%
Flip Flop	52	27392	0.2%
BRAM	1	136	0.7%

Table 6.4: FSL device utilization.

Table 6.5 shows the resources consumed by the multi-core design, with five OpenFire processors and seventeen FSL busses.

Resource	Used	Device Total	Percent Usage
4-bit LUT	6998	27392	25%
Flip Flop	2959	27392	11%
Multiplier	15	136	11%
BRAM	57	136	42%

Table 6.5: Software algorithm device utilization.

6.3.4 Visual Performance

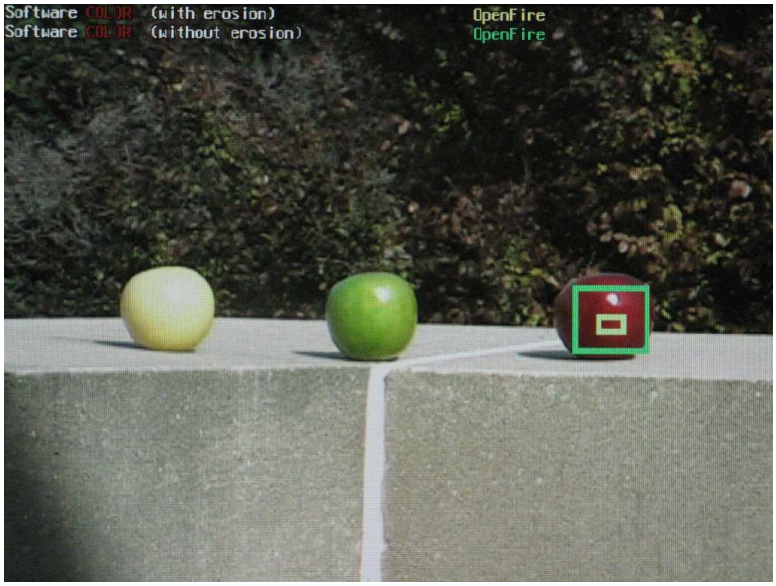


Figure 6.8: The software algorithm, attuned to red.

The software algorithm's visual performance is not as good as the hardware algorithm's visual performance. The downsampling causes the software algorithm to erode away most of the pixels in any bounding box discovered. The non-eroded bounding box occasionally includes pixels that do not belong, although most of the noise that would be removed by the erosion is removed instead by the downsampling. Figures 6.8, 6.9, and 6.10 show both the eroded and non-eroded versions of the software algorithm being run on the video feed simultaneously. Figure 6.8 shows the red apple being identified by both the eroded and non-eroded versions of the algorithm. The green bounding box is being drawn by the non-eroded version of the algorithm, while the yellow box is being drawn by the eroded version of the

algorithm. The yellow box is smaller than the green box, an effect of the erosion process.

Figure 6.9: The software algorithm, attuned to green.

Figure 6.9 shows the green apple being identified by only the non-eroded version of the software algorithm. The number of pixels left of the green apple after the downsampling and the erosion is too small for the eroded version of the software algorithm to detect.

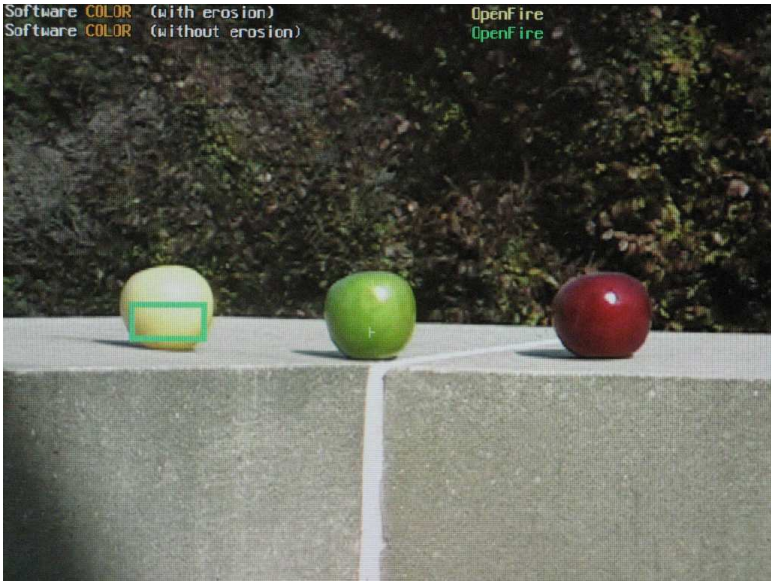


Figure 6.10: The software algorithm, attuned to yellow.

Figure 6.10 shows the yellow apple being identified by only the non-eroded version of the software algorithm. Just as with the hardware algorithm, the yellow color and the green color are very close, allowing changes in lighting to cause the software algorithm to select both the yellow and the green apple.

6.4 Side-by-Side Comparisons

Of the two implementations the hardware implementation consumes far fewer device resources than the multi-core software implementation. Table 6.6 shows a direct comparison of the amount of resources used in each implementation.

Resource	Hardware	Multicore	Percent
4-bit LUT	252	6998	3.6%
Flip Flop	676	2959	20.8%
Multiplier	3	15	20%
BRAM	0	57	NA

Table 6.6: Comparison of device utilization.

The hardware implementation used none of the BRAMs used by the multi-core implementation, and only 3.6% of the 4-bit LUTs. Even though the multi-core implementation had access to all of those extra resources, it was still not able to input video data at the full rate.

6.4.1 Visual Performance

With both processing algorithms implemented side-by-side in the hardware the visual performance is easy to compare. The hardware algorithm is able to keep track of objects of the desired color in a non-cluttered visual space. Small items such as the light from a laser pointer are detected and followed with little visible lag. The multi-core software implementation, using a downsampled form of the video, has a much more difficult time in tracking

small items. The erosion algorithm coupled with the downsampling means that unless an item is taking up a significant portion of the camera's field of view, it will not be detected. However, when the erosion algorithm is removed from the multi-core implementation, the performance is similar to that of the hardware implementation. Without the erosion algorithm the multi-core implementation is more susceptible to changes in lighting that cause the algorithm to select the entire screen instead of a colored object on the screen, however it is able to follow motion just as quickly as the hardware implementation.

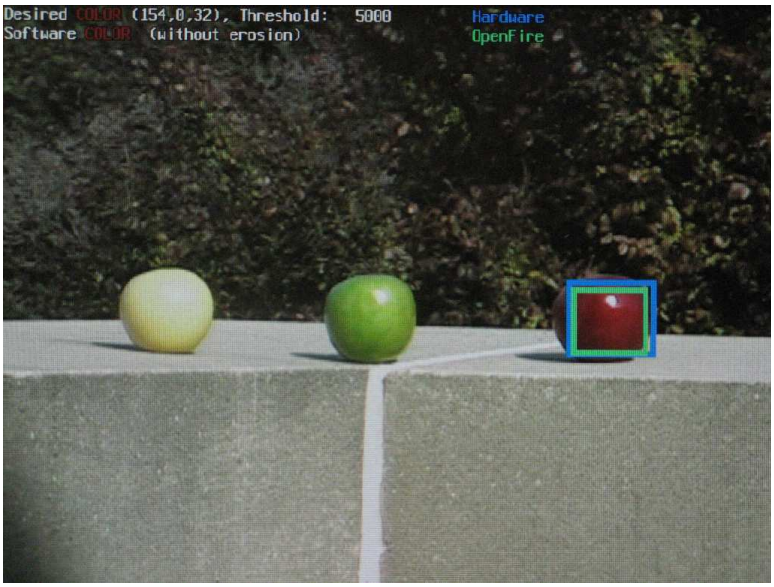


Figure 6.11: A side by side comparison attuned to red.

Figure 6.11 shows both the hardware and the software algorithms running in parallel, both detecting the red apple. The blue bounding box is being drawn by the hardware, and the green bounding box is being drawn by the software. The bounding box being drawn by the software is smaller than the box drawn by the hardware due to the downsampling of the incoming video stream, however the boxes are very close in dimension and placement.

Figure 6.12 shows both the hardware and the software algorithms detecting the green apple. Figure 6.13 shows the yellow apple being detected.

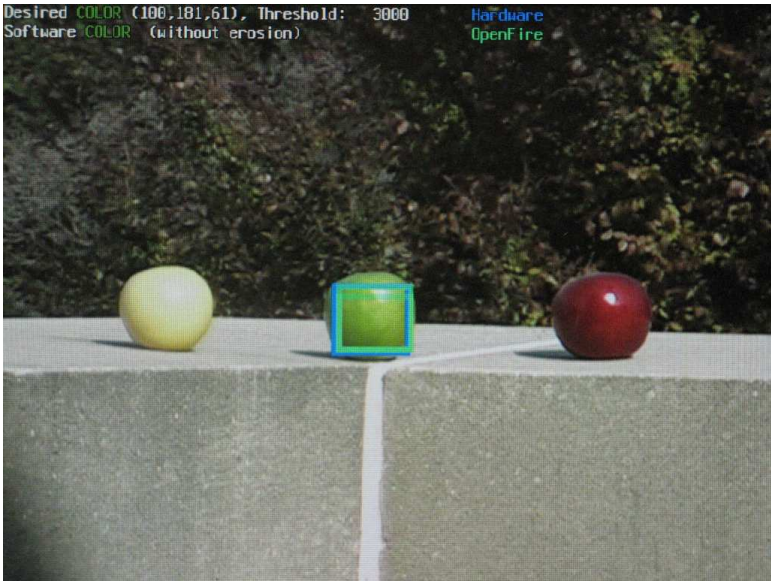


Figure 6.12: A side by side comparison attuned to green.

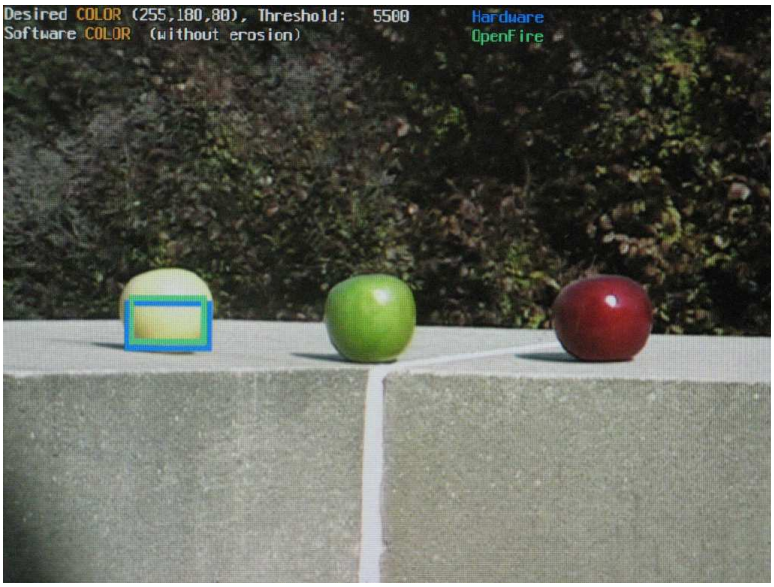


Figure 6.13: A side by side comparison attuned to yellow.

6.4.2 Power Consumption

A further metric for comparing the hardware and the software implementations is to compare power utilization. Table 6.7 displays a comparison of the power consumed during varions system states. This power was measured via an ammeter placed in series with the XUP board, and all power measurements except for the programming power measurement include the power consumed by the base system design shown in Figure 6.3.

System State	Current at 5V	Power
Programming	0.75 A	3.75 W
Hardware Running	1.26 A	6.3 W
Software Running	1.36 A	6.8 W
Both Running	1.36 A	6.8 W

Table 6.7: Comparison of design power utilization.

The programming state requires a little more than half the power of any of the running states. When the five OpenFire processor cores are installed in the design the power requirements rise by half a watt as compared to the strictly hardware design. However when both the hardware and the software implementations are instanced in parallel the power requirements are the same as when only the software implementation is active. This indicates that the power requirements for the hardware implementation are negligible when compared to the requirements for running all five OpenFires in the software implementation.

Chapter 7

Conclusions

A side-by-side comparison of the hardware and multi-core software implementations of the object tracking image processing algorithm shows that the convergence point between direct hardware designs and multi-core processor designs cannot be fully realized using soft processors implemented on a Virtex-II Pro FPGA. Although the system functionality is quite similar there are a number of metrics in which the hardware implementation pulls far ahead of the multi-core software implementation.

7.1 Comparison Summary

Device utilization is a large concern when building embedded devices, and in this metric the pure hardware design has a clear advantage. As shown in Section 6.2.1, the hardware implementation does not use more than three percent of any resource found on the host FPGA. The multi-core implementation on the other hand uses 25% of the available LUTs and 42% of the available BRAMs. Adding processors and operations to the multi-core design is more simple than adding stages to the hardware design, as adding an FSL connection and another processor is trivial; however, this ease of modification comes with a large area

penalty.

As far as visual performance goes, the subjective impression gained from figures 6.11, 6.12, and 6.13 is that the hardware and multi-core implementations work equally well. This impression does not take into account the greater quality of the hardware algorithm. Section 6.3.1 describes how the incoming video signal must be degraded in order to accomodate the maximum input bandwidth of the multi-core OpenFire network. The quality of the multi-core results would be greatly improved if the OpenFire were able to run at 100 MHz, however the current version is unable.

The multi-core implementation also falls behind in terms of power consumption. Table 6.7 shows that the hardware implementation requires around half a watt less power than the multi-core implementation. This is largely influenced by the disparity in device utilization between the two devices, making it an unavoidable consequence of a design using processors instead of custom hardware.

The one saving grace of the multi-core implementation is the ease of development. Using the OpenFires allows the designer to repeat the use of already-verified hardware. It also exposes a familiar C or C++ programming environment with a unified memory model and well-supported tool chains. When writing software for the processors the designer does not need to be concerned with timing information or other hardware level issues, as these are abstracted away by the processor core.

7.2 Future Work

Several avenues for future work exist well within the scope of this thesis. One weakness of both the hardware and multi-core software implementations of the algorithm is the reliance on the RGB color space, which considers changes in lighting to be changes in the observed colors. Other color spaces, such as the YUV color space, exist and might make an appreciable

difference in the performance of the object tracking algorithm. If the OpenFire processor can be modified to operate at a higher frequency, this would provide better performance of the multi-core implementation by removing the need to downsample the incoming video stream. A larger FPGA with more than four or five OpenFire processors could divide the incoming frames into regions and handle each region with a subset of processors, further increasing the quality of the results.

Comparing the multi-core software implementation of the image processing algorithm to a GPP-based implementation of the algorithm, something that was not done in this thesis, might help to vindicate the use of multi-core systems, especially considering power. Although the hardware implementation uses very little power compared to the multi-core implementation, the power density of modern GPPs makes it likely that the multi-core implementation would require far less power.

Moving further away from work already performed, the next step in exploring the use of multi-core designs is to implement other algorithms with different OpenFire network topologies. In this thesis a linear daisy-chain network was used because it most closely resembled the original hardware algorithm that was being transferred into software. Other real time applications, such as ethernet packet routing, might use a quite different type of multi-core network.

One final comparison that could be of high interest in the near future is to compare this paradigm of substituting processors for LUTs to the C-style hardware description language Impulse C. Impulse C uses ANSI C libraries to allow hardware implementation using standard C without requiring language extensions. The design model for Impulse C revolves around computational blocks connected via FIFOs. This is a very similar structure compared to a multi-core OpenFire system connected via FSL busses. The desirability of persuing a processor-centric model of hardware design may depend on an in-depth performance comparison between these two methods of allowing designers to program in C rather write in an HDL.

Appendix A

A.1 FSL demo code

```
// This code is used by Device 1 to write five times to Device 2 using FSL bus 0.
#include "fsl.h"

int main(void) {

     putfsl(1, 0);
     putfsl(2, 0);
     putfsl(3, 0);
     putfsl(4, 0);
     putfsl(5, 0);

     while(1);
 return(0);
}


// This code is used by Device 2 to read five times from Device 1 using FSL bus 0.
#include "fsl.h"

int main(void) {
     int data;

     getfsl(data, 0);
     getfsl(data, 0);
     getfsl(data, 0);
     getfsl(data, 0);
     getfsl(data, 0);

     while(1);
 return(0);
}
```

A.2 FSL loopback test

```
// This code is used by an OpenFire with all eight of its FSL ports connected
// in loopback to ensure that all ports are functional.
#include "fsl.h"

bool test_fsls(void);

int main(void) {

     test_fsls();
     while(1);
}

bool test_fsls(void) {
     int data = 0;
     putfsl(data+1,0);
     getfsl(data,0);
     putfsl(data+1,1);
     getfsl(data,1);
     putfsl(data+1,2);
     getfsl(data,2);
     putfsl(data+1,3);
     getfsl(data,3);
     putfsl(data+1,4);
     getfsl(data,4);
     putfsl(data+1,5);
     getfsl(data,5);
     putfsl(data+1,6);
     getfsl(data,6);
     putfsl(data+1,7);
     getfsl(data,7);

     return(data == 8);
}
```

A.3 IOPB / DOPB test code

```
#include "microblaze_0/include/xparameters.h"
#include "microblaze_0/include/xuartlite_l.h"
#include "microblaze_0/include/fsl.h"

extern void xil_printf(const char*, ...);

void outbyte(char c)
{
XUartLite_SendByte(STDOUT_BASEADDRESS, c);
}

char inbyte(void)
{
return XUartLite_RecvByte(STDIN_BASEADDRESS);
}

volatile unsigned int * timer =   (unsigned int *)0x40060008;
volatile unsigned int * ddr   =   (unsigned int *)0x30000000;

unsigned int (*moved_summer)(const unsigned int * values, const int numvals);
unsigned int summer(const unsigned int * values, const int numvals)
{
int ctr;
unsigned int sum = 0;

for(ctr=0;ctr<numvals;ctr++)
sum += values[ctr];

return sum;
}

int main(void)
{
unsigned int starttime;
unsigned int endtime;
unsigned int timetaken;
int test_ctr;
int num_tests = 16;
int num_shifts = 4; // num_shifts = log2(num_tests)
```

```
int array_sz = 16;
int array_ctr;
int num_instructions = 100;
int instr_ctr;
unsigned int retval;

for(array_ctr = 0; array_ctr < array_sz; array_ctr++)
ddr[array_ctr] = array_ctr;

xil_printf("-- Starting Test (BRAM) --\r\n");
retval = 0;
timetaken = 0;

for(test_ctr=0;test_ctr<num_tests;test_ctr++)
{
starttime = timer[0];

retval = summer(ddr,array_sz);

endtime = timer[0];
timetaken += endtime - starttime;
}

xil_printf("-- Test Finished --\r\n");
xil_printf("RtVal: %d\r\n", retval);
xil_printf("Taken: %d\r\n", timetaken);
xil_printf("#Runs: %d\r\n", num_tests);
for(test_ctr=0;test_ctr<num_shifts;test_ctr++) timetaken >>= 1;
xil_printf("Avg:   %d\r\n", timetaken);

((unsigned int *)moved_summer) = ddr + 0x1000;
for(instr_ctr=0; instr_ctr<num_instructions; instr_ctr++)
((unsigned int *)moved_summer)[instr_ctr] = ((unsigned int *)summer)[instr_ctr];

xil_printf("-- Starting Test (DDR) --\r\n");
retval = 0;
timetaken = 0;

for(test_ctr=0;test_ctr<num_tests;test_ctr++)
{
starttime = timer[0];
```

```
retval = moved_summer(ddr,array_sz);

endtime = timer[0];
timetaken += endtime - starttime;
}

xil_printf("-- Test Finished --\r\n");
xil_printf("RtVal: %d\r\n", retval);
xil_printf("Taken: %d\r\n", timetaken);
xil_printf("#Runs: %d\r\n", num_tests);
for(test_ctr=0;test_ctr<num_shifts;test_ctr++) timetaken >>= 1;
xil_printf("Avg:   %d\r\n", timetaken);

while(1);

return(0);
}
```

A.4 OpenFire#1 C Code

```
unsigned mostly_colors;
unsigned thresh_desired;
unsigned char  * rdes   = &((unsigned char *)&mostly_colors)[0]);
unsigned char  * r      = &((unsigned char *)&mostly_colors)[1]);
unsigned char  * g      = &((unsigned char *)&mostly_colors)[2]);
unsigned char  * b      = &((unsigned char *)&mostly_colors)[3]);
unsigned short * thresh = &((unsigned short *)&thresh_desired)[0]);
unsigned char  * gdes   = &((unsigned char *)&thresh_desired)[2]);
unsigned char  * bdes   = &((unsigned char *)&thresh_desired)[3]);
unsigned yx_data, sum;

while(1)
{
   getfsl(mostly_colors,0);
   getfsl(thresh_desired,1);
   getfsl(yx_data,2);

   sum =  ((*r) - (*rdes)) * ((*r) - (*rdes)) +
          ((*g) - (*gdes)) * ((*g) - (*gdes)) +
          ((*b) - (*bdes)) * ((*b) - (*bdes));

   if(sum < (*thresh))
   {
       putfsl(yx_data,0);
       putfsl(yx_data,1);
   }
   else
   {
       putfsl(0,0);
       putfsl(0,1);
   }
}
```

A.5 OpenFire#1 Assembly Code

```
00000168 <main>:
 168: 3021ffcc  addik r1, r1, -52
 16c: fa610030  swi r19, r1, 48
 170: 12610000  addk r19, r1, r0
 174: 30730004  addik r3, r19, 4
 178: f873000c  swi r3, r19, 12
 17c: 30730005  addik r3, r19, 5
 180: f8730010  swi r3, r19, 16
 184: 30730006  addik r3, r19, 6
 188: f8730014  swi r3, r19, 20
 18c: 30730007  addik r3, r19, 7
 190: f8730018  swi r3, r19, 24
 194: 30730008  addik r3, r19, 8
 198: f873001c  swi r3, r19, 28
 19c: 30730008  addik r3, r19, 8
 1a0: 30630002  addik r3, r3, 2
 1a4: f8730020  swi r3, r19, 32
 1a8: 30730008  addik r3, r19, 8
 1ac: 30630003  addik r3, r3, 3
 1b0: f8730024  swi r3, r19, 36
 1b4: 6c600000  get r3, rfsl0
 1b8: f8730004  swi r3, r19, 4
 1bc: 6c600001  get r3, rfsl1
 1c0: f8730008  swi r3, r19, 8
 1c4: 6c600002  get r3, rfsl2
 1c8: f8730028  swi r3, r19, 40
 1cc: e8730010  lwi r3, r19, 16
 1d0: e0830000  lbui r4, r3, 0
 1d4: e873000c  lwi r3, r19, 12
 1d8: e0630000  lbui r3, r3, 0
 1dc: 14632000  rsubk r3, r3, r4
 1e0: 40a31800  mul r5, r3, r3
 1e4: e8730014  lwi r3, r19, 20
 1e8: e0830000  lbui r4, r3, 0
 1ec: e8730020  lwi r3, r19, 32
 1f0: e0630000  lbui r3, r3, 0
 1f4: 14632000  rsubk r3, r3, r4
 1f8: 40631800  mul r3, r3, r3
 1fc: 10a51800  addk r5, r5, r3
```

```
200: e8730018  lwi r3, r19, 24
204: e0830000  lbui r4, r3, 0
208: e8730024  lwi r3, r19, 36
20c: e0630000  lbui r3, r3, 0
210: 14632000  rsubk r3, r3, r4
214: 40631800  mul r3, r3, r3
218: 10651800  addk r3, r5, r3
21c: f873002c  swi r3, r19, 44
220: e873001c  lwi r3, r19, 28
224: e4830000  lhui r4, r3, 0
228: e873002c  lwi r3, r19, 44
22c: 16441803  cmpu r18, r4, r3
230: bcb20018  bgei r18, 24 // 248
234: e8730028  lwi r3, r19, 40
238: 6c03c000  nput r3, rfsl0
23c: e8730028  lwi r3, r19, 40
240: 6c03c001  nput r3, rfsl1
244: b800ff70  bri -144 // 1b4
248: 10600000  addk r3, r0, r0
24c: 6c03c000  nput r3, rfsl0
250: 10600000  addk r3, r0, r0
254: 6c03c001  nput r3, rfsl1
258: b800ff5c  bri -164 // 1b4
```

A.6 OpenFire#2 C Code

```
unsigned erode_0_left = 0, erode_0_center = 0, erode_0_right = 0;
unsigned erode_1_left = 0, erode_1_center = 0, erode_1_right = 0;
unsigned erode_2_left = 0, erode_2_center = 0, erode_2_right = 0;

while(1)
{
   if(erode_2_left && erode_2_center && erode_2_right) putfsl(erode_2_center,0);

   erode_2_right  = erode_2_center;
   erode_2_center = erode_2_left;

   if(erode_1_left && erode_1_center && erode_1_right) erode_2_left = erode_1_center;
   else                                                erode_2_left = 0;

   erode_1_right  = erode_1_center;
   erode_1_center = erode_1_left;

   if(erode_0_left && erode_0_center && erode_0_right) erode_1_left = erode_0_center;
   else                                                erode_1_left = 0;

   erode_0_right  = erode_0_center;
   erode_0_center = erode_0_left;

   getfsl(erode_0_left,0);
}
```

A.7 OpenFire#2 Assembly Code

```
00000168 <main>:
 168: 3021ffd0  addik r1, r1, -48
 16c: fa61002c  swi r19, r1, 44
 170: 12610000  addk r19, r1, r0
 174: f8130004  swi r0, r19, 4
 178: f8130008  swi r0, r19, 8
 17c: f813000c  swi r0, r19, 12
 180: f8130010  swi r0, r19, 16
 184: f8130014  swi r0, r19, 20
 188: f8130018  swi r0, r19, 24
 18c: f813001c  swi r0, r19, 28
 190: f8130020  swi r0, r19, 32
 194: f8130024  swi r0, r19, 36
 198: e873001c  lwi r3, r19, 28
 19c: bc03001c  beqi r3, 28 // 1b8
 1a0: e8730020  lwi r3, r19, 32
 1a4: bc030014  beqi r3, 20 // 1b8
 1a8: e8730024  lwi r3, r19, 36
 1ac: bc03000c  beqi r3, 12 // 1b8
 1b0: e8730020  lwi r3, r19, 32
 1b4: 6c038000  put r3, rfsl0
 1b8: e8730020  lwi r3, r19, 32
 1bc: f8730024  swi r3, r19, 36
 1c0: e873001c  lwi r3, r19, 28
 1c4: f8730020  swi r3, r19, 32
 1c8: e8730010  lwi r3, r19, 16
 1cc: bc030020  beqi r3, 32 // 1ec
 1d0: e8730014  lwi r3, r19, 20
 1d4: bc030018  beqi r3, 24 // 1ec
 1d8: e8730018  lwi r3, r19, 24
 1dc: bc030010  beqi r3, 16 // 1ec
 1e0: e8730014  lwi r3, r19, 20
 1e4: f873001c  swi r3, r19, 28
 1e8: b8000008  bri 8 // 1f0
 1ec: f813001c  swi r0, r19, 28
 1f0: e8730014  lwi r3, r19, 20
 1f4: f8730018  swi r3, r19, 24
 1f8: e8730010  lwi r3, r19, 16
 1fc: f8730014  swi r3, r19, 20
```

```
200: e8730004  lwi r3, r19, 4
204: bc030020  beqi r3, 32 // 224
208: e8730008  lwi r3, r19, 8
20c: bc030018  beqi r3, 24 // 224
210: e873000c  lwi r3, r19, 12
214: bc030010  beqi r3, 16 // 224
218: e8730008  lwi r3, r19, 8
21c: f8730010  swi r3, r19, 16
220: b8000008  bri 8 // 228
224: f8130010  swi r0, r19, 16
228: e8730008  lwi r3, r19, 8
22c: f873000c  swi r3, r19, 12
230: e8730004  lwi r3, r19, 4
234: f8730008  swi r3, r19, 8
238: 6c600000  get r3, rfsl0
23c: f8730004  swi r3, r19, 4
240: b800ff58  bri -168 // 198
```

A.8 OpenFire#3 C Code

```
unsigned data_in = 0;
unsigned short * xptr = &(((unsigned short *)&data_in)[1]);

while(1)
{
   getfsl(data_in,0);

   if((*xptr) >= 3)
   {
       putfsl((data_in-3),0);
   }
   if((*xptr) < (XMAX-3))
   {
       putfsl((data_in+3),0);
   }
}
```

A.9 OpenFire#3 Assembly Code

```
00000168 <main>:
 168: 3021fff0  addik r1, r1, -16
 16c: fa61000c  swi r19, r1, 12
 170: 12610000  addk r19, r1, r0
 174: f8130004  swi r0, r19, 4
 178: 30730006  addik r3, r19, 6
 17c: f8730008  swi r3, r19, 8
 180: 6c600000  get r3, rfsl0
 184: f8730004  swi r3, r19, 4
 188: e8730008  lwi r3, r19, 8
 18c: e4630000  lhui r3, r3, 0
 190: 22400002  addi r18, r0, 2
 194: 16439003  cmpu r18, r3, r18
 198: bcb20010  bgei r18, 16 // 1a8
 19c: e8730004  lwi r3, r19, 4
 1a0: 3063fffd  addik r3, r3, -3
 1a4: 6c038000  put r3, rfsl0
 1a8: e8730008  lwi r3, r19, 8
 1ac: e4630000  lhui r3, r3, 0
 1b0: 224002b8  addi r18, r0, 696
 1b4: 16439003  cmpu r18, r3, r18
 1b8: bc52ffc8  blti r18, -56 // 180
 1bc: e8730004  lwi r3, r19, 4
 1c0: 30630003  addik r3, r3, 3
 1c4: 6c038000  put r3, rfsl0
 1c8: b800ffb8  bri -72 // 180
```

A.10 OpenFire#4 C Code

```
unsigned x = 0,y = 0,y_temp = 0,y_prev = 0;
unsigned short * yptr = &(((unsigned short *)&x)[y]);
unsigned xmin_temp = 0, xmax_temp = 0, ymin_temp = 0, ymax_temp = 0;
unsigned eof, error;
unsigned found;

while(1)
{
   getfsl(x,0);
   ngetfsl(eof,1);
   fsl_isinvalid(error);

   if(!error) eof = 1;
   else eof = 0;

   y_temp = y;
   y = (*yptr);
   x &= 0x3FF;

   if(eof)
   {
       if(found)
       {
           putfsl(xmin_temp,0);
           putfsl(xmax_temp,1);
           putfsl(ymin_temp,2);
           putfsl(ymax_temp,3);
           xmin_temp = XMAX;
           xmax_temp = 0;
           ymin_temp = YMAX;
           ymax_temp = 0;
       }

       y_prev = 0;
       y_temp = 0;
       found = 0;
   }

   if(x < xmin_temp) {xmin_temp = x; y_prev = y_temp; found = 1;}
```

```
   if(x > xmax_temp) {xmax_temp = x; y_prev = y_temp; found = 1;}
   if(y < ymin_temp) {ymin_temp = y; y_prev = y_temp; found = 1;}
   if(y > ymax_temp) {ymax_temp = y; y_prev = y_temp; found = 1;}
}
```

A.11 OpenFire#4 Assembly Code

```
00000168 <main>:
 168: 3021ffc4  addik r1, r1, -60
 16c: fa610038  swi r19, r1, 56
 170: 12610000  addk r19, r1, r0
 174: f8130004  swi r0, r19, 4
 178: f8130008  swi r0, r19, 8
 17c: f813000c  swi r0, r19, 12
 180: f8130010  swi r0, r19, 16
 184: e8730008  lwi r3, r19, 8
 188: 10831800  addk r4, r3, r3
 18c: 30730004  addik r3, r19, 4
 190: 10632000  addk r3, r3, r4
 194: f8730014  swi r3, r19, 20
 198: f8130018  swi r0, r19, 24
 19c: f813001c  swi r0, r19, 28
 1a0: f8130020  swi r0, r19, 32
 1a4: f8130024  swi r0, r19, 36
 1a8: f8130028  swi r0, r19, 40
 1ac: 6c600000  get r3, rfsl0
 1b0: f8730004  swi r3, r19, 4
 1b4: 6c604001  nget r3, rfsl1
 1b8: f873002c  swi r3, r19, 44
 1bc: 28600000  addic r3, r0, 0
 1c0: f8730030  swi r3, r19, 48
 1c4: e8730030  lwi r3, r19, 48
 1c8: bc230010  bnei r3, 16 // 1d8
 1cc: 30600001  addik r3, r0, 1
 1d0: f873002c  swi r3, r19, 44
 1d4: b8000008  bri 8 // 1dc
 1d8: f813002c  swi r0, r19, 44
 1dc: e8730008  lwi r3, r19, 8
 1e0: f873000c  swi r3, r19, 12
 1e4: e8730014  lwi r3, r19, 20
 1e8: e4630000  lhui r3, r3, 0
 1ec: f8730008  swi r3, r19, 8
 1f0: e8730004  lwi r3, r19, 4
 1f4: a46303ff  andi r3, r3, 1023
 1f8: f8730004  swi r3, r19, 4
 1fc: e873002c  lwi r3, r19, 44
```

```
200: bc030068  beqi r3, 104 // 268
204: e8730028  lwi r3, r19, 40
208: 30630001  addik r3, r3, 1
20c: f8730028  swi r3, r19, 40
210: e8730034  lwi r3, r19, 52
214: bc030040  beqi r3, 64 // 254
218: e8730018  lwi r3, r19, 24
21c: 6c038000  put r3, rfsl0
220: e873001c  lwi r3, r19, 28
224: 6c038001  put r3, rfsl1
228: e8730020  lwi r3, r19, 32
22c: 6c038002  put r3, rfsl2
230: e8730024  lwi r3, r19, 36
234: 6c038003  put r3, rfsl3
238: 306002bc  addik r3, r0, 700
23c: f8730018  swi r3, r19, 24
240: f813001c  swi r0, r19, 28
244: 306001e0  addik r3, r0, 480
248: f8730020  swi r3, r19, 32
24c: f8130024  swi r0, r19, 36
250: b800000c  bri 12 // 25c
254: 3060004d  addik r3, r0, 77
258: 6c038004  put r3, rfsl4
25c: f8130010  swi r0, r19, 16
260: f813000c  swi r0, r19, 12
264: f8130034  swi r0, r19, 52
268: e8930004  lwi r4, r19, 4
26c: e8730018  lwi r3, r19, 24
270: 16432003  cmpu r18, r3, r4
274: bcb2001c  bgei r18, 28 // 290
278: e8730004  lwi r3, r19, 4
27c: f8730018  swi r3, r19, 24
280: e873000c  lwi r3, r19, 12
284: f8730010  swi r3, r19, 16
288: 30600001  addik r3, r0, 1
28c: f8730034  swi r3, r19, 52
290: e8930004  lwi r4, r19, 4
294: e873001c  lwi r3, r19, 28
298: 16441803  cmpu r18, r4, r3
29c: bcb2001c  bgei r18, 28 // 2b8
2a0: e8730004  lwi r3, r19, 4
2a4: f873001c  swi r3, r19, 28
```

```
2a8: e873000c  lwi r3, r19, 12
2ac: f8730010  swi r3, r19, 16
2b0: 30600001  addik r3, r0, 1
2b4: f8730034  swi r3, r19, 52
2b8: e8930008  lwi r4, r19, 8
2bc: e8730020  lwi r3, r19, 32
2c0: 16432003  cmpu r18, r3, r4
2c4: bcb2001c  bgei r18, 28 // 2e0
2c8: e8730008  lwi r3, r19, 8
2cc: f8730020  swi r3, r19, 32
2d0: e873000c  lwi r3, r19, 12
2d4: f8730010  swi r3, r19, 16
2d8: 30600001  addik r3, r0, 1
2dc: f8730034  swi r3, r19, 52
2e0: e8930008  lwi r4, r19, 8
2e4: e8730024  lwi r3, r19, 36
2e8: 16441803  cmpu r18, r4, r3
2ec: bcb2fec0  bgei r18, -320 // 1ac
2f0: e8730008  lwi r3, r19, 8
2f4: f8730024  swi r3, r19, 36
2f8: e873000c  lwi r3, r19, 12
2fc: f8730010  swi r3, r19, 16
300: 30600001  addik r3, r0, 1
304: f8730034  swi r3, r19, 52
308: b800fea4  bri -348 // 1ac
```

A.12 mb-gcc Implementation of the Ternary Operator

```
       fsl_isinvalid(error);
1bc:   28600000    addic   r3, r0, 0
1c0:   f8730030    swi   r3, r19, 48

     eof = (!error) ? 1 : 0;
1c4:   e8730030    lwi   r3, r19, 48
1c8:   14830000    rsubk   r4, r3, r0
1cc:   e8730030    lwi   r3, r19, 48
1d0:   80641800    or   r3, r4, r3
1d4:   a863ffff    xori   r3, r3, -1
1d8:   90630041    srl   r3, r3
1dc:   90630041    srl   r3, r3
1e0:   90630041    srl   r3, r3
1e4:   90630041    srl   r3, r3
1e8:   90630041    srl   r3, r3
1ec:   90630041    srl   r3, r3
1f0:   90630041    srl   r3, r3
1f4:   90630041    srl   r3, r3
1f8:   90630041    srl   r3, r3
1fc:   90630041    srl   r3, r3
200:   90630041    srl   r3, r3
204:   90630041    srl   r3, r3
208:   90630041    srl   r3, r3
20c:   90630041    srl   r3, r3
210:   90630041    srl   r3, r3
214:   90630041    srl   r3, r3
218:   90630041    srl   r3, r3
21c:   90630041    srl   r3, r3
220:   90630041    srl   r3, r3
224:   90630041    srl   r3, r3
228:   90630041    srl   r3, r3
22c:   90630041    srl   r3, r3
230:   90630041    srl   r3, r3
234:   90630041    srl   r3, r3
238:   90630041    srl   r3, r3
23c:   90630041    srl   r3, r3
240:   90630041    srl   r3, r3
244:   90630041    srl   r3, r3
248:   90630041    srl   r3, r3
```

```
24c:   90630041    srl   r3, r3
250:   90630041    srl   r3, r3
254:   f873002c    swi   r3, r19, 44

     y_temp = y;
258:   e8730008    lwi   r3, r19, 8
25c:   f873000c    swi   r3, r19, 12
```

Assembly code using the ternary operator.

```
       fsl_isinvalid(error);
1bc:   28600000    addic   r3, r0, 0
1c0:   f8730030    swi   r3, r19, 48

       if(!error) eof = 1;
1c4:   e8730030    lwi   r3, r19, 48
1c8:   bc230010    bnei   r3, 16      // 1d8
1cc:   30600001    addik   r3, r0, 1
1d0:   f873002c    swi   r3, r19, 44
1d4:   b8000008    bri   8      // 1dc
       else eof = 0;
1d8:   f813002c    swi   r0, r19, 44

       y_temp = y;
1dc:   e8730008    lwi   r3, r19, 8
1e0:   f873000c    swi   r3, r19, 12
```

Assembly code using the if-else C construct.

Bibliography

[1] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, January 1998.

[2] L. Gea-Banacloche, J.; Kish, "Future directions in electronic computing and information processing," *Proceedings of the IEEE*, vol. 93, no. 10, pp. 1858–1863, October 2005.

[3] R. T. D. Frank, D.J.; Puri, "Design and cad challenges in 45nm cmos and beyond," *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pp. 329–333, November 2006.

[4] S. Xiaoliang Bai; Dey, "High-level crosstalk defect simulation methodology for system-on-chip interconnects," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 23, no. 9, pp. 1355–1361, September 2004.

[5] "Intel core2 duo processors and intel core2 extreme processors for platforms based on mobile intel 965 express chipset family," Online Datasheet, Intel, August 2007, http://www.intel.com/design/mobile/datashts/316745.htm.

[6] "Intel core2 extreme quad-core processor qx6800," Online Datasheet, Intel, April 2007, http://www.intel.com/design/processor/datashts/316852.htm.

[7] "C-based design and behavioral synthesis: Handel-c," Online, December 2007, http://www.celoxica.com/technology/c_design/handel-c.asp.

[8] D. Ku and G. DeMicheli, "Hardwarec – a language for hardware design (version 2.0)," Stanford, CA, USA, Tech. Rep., 1990.

[9] D. Soderman and Y. Panchul, "Implementing c algorithms in reconfigurable hardware using c2verilog," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, 15-17 April 1998, pp. 339–342.

[10] S. Edwards, "The challeges of synthesizing hardware from c-like languages," *Design & Test of Computers, IEEE*, vol. 23, no. 5, pp. 375–386, May 2006.

[11] *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, Xilinx, October 2005, http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf.

[12] *MicroBlaze Processor Reference Guide*, Xilinx, June 2006, http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf.

[13] S. D. Craven, *OpenFire Documentation Version 0.3 Beta*, Virginia Tech CCM Lab, January 2006, http://www.ccm.ece.vt.edu/ amarschn/openfire/.

[14] S. Kelem, B. Box, S. Wasson, B. Plunkett, J. Hassoun, and C. Phillips, "An elemental computing architecture for sd radio," Milpitas, CA, USA, Tech. Rep., 2007, http://www.elementcxi.com/Element%20CXI%20sdr_conference-2007%20whitepaper.pdf.

[15] "Impulse c," Online, December 2007, http://www.impulsec.com.

[16] "Arm processor families," Online, December 2007, http://www.arm.com/products/CPUs/families.html.

[17] *PicoBlaze 8-bit Embedded Microcontroller User Guide*, Xilinx, November 2005, http://www.xilinx.com/support/documentation/user_guides/ug129.pdf.

[18] *Nios II Processor Reference Handbook*, Altera, October 2007, http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.

[19] *LatticeMicro32 Processor Reference Manual*, Lattice Semiconductor Corporation, August 2007, http://www.latticesemi.com/documents/doc20890x45.pdf.

[20] "Pacoblaze - a synthesizable behavioral verilog picoblaze clone," Online, December 2007, http://bleyer.org/pacoblaze/.

[21] "Xtensa 7 product brief," Online, December 2007, http://www.tensilica.com/pdf/xtensa_7.pdf.

[22] "Configurable processors: What, why, how?" Online, December 2007, http://www.tensilica.com/products/WP_config.htm.

[23] C. f. X. Ivo Bolsens, "Programming modern fpgas," MPSOC 2006 Presentation Slides, August 2006, http://www.xilinx.com/univ/mpsoc2006keynote.pdf.

[24] S. Craven, C. Patterson, and P. Athanas, "Configurable soft processor arrays using the openfire processor," in *Proceedings of the 8th Annual Conference on Military and Aerospace Programmable Logic Devices*. Washington, DC: MAPLD, September 2005.

[25] S. A. Guccione, "Microprocessors: The new lut," in *Proceedings of the 2005 International Conference on Engineering of Reconfigurable Systems & Algorithms*. Las Vegas, NV, USA: ERSA, June 2005, pp. 26–25.

[26] T. Bataineh, S.; Robertazzi, "Performance limits for processor networks with divisible jobs," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 33, no. 4, pp. 1189–1198, October 1997.

[27] A. L. Abbott, P. M. Athanas, and A. D. Tarmaster, "Accelerating image filters using a custom computing machine," J. Schewel, Ed., vol. 2607, no. 1. SPIE, 1995, pp. 62–70.

[28] J. U. Cho, S. H. Jin, X. D. Pham, and J. W. Jeon, "Multiple objects tracking circuit using particle filters with multiple features," in *Robotics and Automation, 2007 IEEE International Conference on*, 10-14 April 2007, pp. 4639–4644.

[29] W. Piekarski, R. Smith, G. Wigley, B. Thomas, and D. Kearney, "Mobile hand tracking using fpgas for low powered augmented reality," in *Wearable Computers, 2004. ISWC 2004. Eighth International Symposium on*, vol. 1, 31 Oct.-3 Nov. 2004, pp. 190–191.

[30] B. P. C. V. P. Kaaniche, K.; Champion, "A vision algorithm for dynamic detection of moving vehicles with a uav," *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pp. 1878–1883, April 2005.

[31] P. Zhihai He; Iyer, R.V.; Chandler, "Vision-based uav flight control and obstacle avoidance," *American Control Conference, 2006*, pp. 5 pp.–, June 2006.

[32] K. P. M. T. P. Repo, P.; Hyvonen, "Users inventing ways to enjoy new mobile services - the case of watching mobile videos," *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pp. 8 pp.–, January 2004.

[33] G. Engelsberg, A.; Schmidt, "A comparative review of digital image stabilising algorithms for mobile video communications," *Consumer Electronics, IEEE Transactions on*, vol. 45, no. 3, pp. 591–597, August 1999.

[34] J. E. Scalera, "Image chipping with a common architecture for microsensors (caus)," Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, 2001.

[35] J. Serra, *Image Analysis and Mathematical Morphology*. Academic Press, 1982, pp. 50–51.

[36] *Fast Simplex Link (FSL) Bus (v2.00a)*, Xilinx, December 2005, http://www.xilinx.com/products/ipcenter/FSL.htm.

[37] *On-Chip Peripheral Bus Architecture Specifications Version 2.1*, IBM, April 2001, http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/9A7AFA74DAD200D087256AB30005F0C8.

[38] *OPB Arbiter (v1.02e)*, Xilinx, September 2005, http://www.xilinx.com/bvdocs/ipcenter/data_sheet/opb_arbiter.pdf.

[39] *Xilinx University Program Virtex-II Pro Development System Hardware Reference Manual*, Xilinx, March 2005, http://www.xilinx.com/univ/xupv2p.html.

[40] *Digilent Video Decoder Board (VDEC1) Reference Manual*, Digilent Inc., April 2005, http://www.digilentinc.com/Data/Products/VDEC1/VDEC1-rm.pdf.