Enabling a Real-time Solution to Retinal Vascular Tracing Using FPGAs

A Thesis Presented

by

Shawn Miller

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Electrical Engineering

in the field of

Electrical Engineering

Northeastern University

Boston, Massachusetts

April 2004

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Enabling a Real-time Solution to Retinal Vascular Tracing Using FPGAs.

Author: Shawn Miller.

Department: Electrical and Computer Engineering.

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: Prof. Miriam Leeser	Date
Thesis Reader: Prof. Jennifer Dy	Date
Thesis Reader: Prof. Eric Miller	Date
Department Chair: Prof. Fabrizio Lombardi	Date

Graduate School Notified of Acceptance:

Dean: Prof. Yaman Yener	Date

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Enabling a Real-time Solution to Retinal Vascular Tracing Using FPGAs.

Author: Shawn Miller.

Department: Electrical and Computer Engineering.

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: Prof. Miriam Leeser	Date
Thesis Reader: Prof. Jennifer Dy	Date
Thesis Reader: Prof. Eric Miller	Date
Department Chair: Prof. Fabrizio Lombardi	Date

Graduate School Notified of Acceptance:

Dean: Prof. Yaman Yener	Date

Copy Deposited in Library:

Reference Librarian	Date

Abstract

There is a need in the biomedical field for a method of tracing blood vessels in retinal fundus images. The speed of motion of the human eye and the desire for these traces to be available to assist during laser surgery require these traces to be available immediately after the retinal image is acquired. A retinal vascular tracing (RVT) algorithm exists and is currently implemented in software, however it is not fast enough to return traced images in real-time. One computationally intensive part of the algorithm is the two-dimensional filtering of the frames with sixteen different templates. A general purpose microprocessor cannot take full advantage of the parallelism inherent to the filter calculations.

To speed up the RVT algorithm, and to enable a real-time solution, we mapped the filtering operation to the reconfigurable, customizable logic of a Field-programmable Gate Array (FPGA). We take the image data directly from the camera, compute the filter responses for every pixel in real-time, and send the results to memory on the host PC. Instead of the RVT algorithm computing filter responses in software, it simply looks up the results from memory, speeding up the overall solution. In order for the hardware to make results available in real-time with a minimized delay, a complex memory interface was designed.

Acknowledgements

First, I would like to thank my advisor, Professor Miriam Leeser, for giving me the opportunity to work in her research lab. Her support throughout this project has been invaluable.

Thanks to my colleagues at Rensselaer Polytechnic Institute, Professor Badrinath Roysam, Professor Charles Stewart, Kenneth Fritsche, and Alex Tyrrell. This project has been a joint effort, and the input they provided was always helpful.

I would also like to thank my fellow students working in the Rapid Prototyping Lab. They have helped to provide a great place to work by offering the knowledge from their experiences, and making the lab an enjoyable place to be.

Special thanks go out to my parents, Richard and Lynette Miller, and my friends and roommates for their support of my studies.

This research is supported in part by CenSSIS, the Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the National Science Foundation.

Contents

1 Introduction **1**

2 Background **3**

2.1 Retinal Vascular Tracing Algorithm 3

2.1.1 Tracing Algorithm 4

2.1.2 Finding Seed Points 7

2.2 Reconfigurable Hardware 9

2.2.1 FPGA Technology 11

2.2.2 Annapolis Microsystems Firebird Board 16

2.3 Hardware Design Methodology 17

2.3.1 Parallel Processing 18

2.3.2 Pipelining 19

2.3.3 Memory Considerations 22

2.4 Related Work 23

2.5 Summary 25

3 Hardware System Design **26**

3.1 Algorithm Analysis 26

3.2 Approach 30

3.3 Implementation 32

3.3.1 Filter Response Calculation 32

3.3.2 Incoming Data Packing 34

3.3.3 Neighborhood Module 36

3.3.4 Traversing the Image 37

3.3.5 Memory Interface 40
3.3.6 Framegrabber Interface 42
3.3.7 Host Interface 45
3.4 Summary 47

4 Results and Performance 48
4.1 Results From Implementation 48
4.2 Speed and Latency 50
4.3 Performance Speedup 53
4.4 Summary 54

5 Conclusion and Future Work 55
5.1 Conclusion 55
5.2 Future Work 56

Bibliography 59

List of Tables

4.1 Latency 52
4.2 FPGA-Host Communication Delay 53

List of Figures

2.1 Matched Filters 5

2.2 Left and Right Templates for a Vessel at 0° 5

2.3 Initial Seed Points From Line Search [1] 8

2.4 Speed and Flexibility Comparison for Software, FPGAs, and ASICs . 10

2.5 FPGA Architecture 12

2.6 Xilinx Virtex-E CLB Structure [2] 13

2.7 Xilinx Virtex-E Local Routing [2] 14

2.8 Annapolis Microsystems Firebird Board [3] 16

2.9 Firebird Block Diagram [3] 17

2.10 Sequential Processing Example 19

2.11 Pipeline Example 20

2.12 Pipelines for Unequal Task Completion Times: (a)Non-optimized (b)Optimized 21

3.1 Matched Filters 27

3.2 Overlapping of all 16 Templates 29

3.3 "Smart Camera" Hardware System 30

3.4 Response Module 33

3.5 Direction Module 35

3.6 Five Pixels Packed Into One Memory Word 35

3.7 Splitting an 11 x 15 Pixel Neighborhood Into Five 11 x 11 Pixel Windows 36

3.8 Inputting Windows to the Direction Pipeline 37

3.9 Shifting the Window 38

3.10 Creating New Neighborhoods by Shifting Data on the FPGA 39

3.11 Memory Timing Diagram 42

3.12 Timing of Writing and Transferring One Frame of Results 46

4.1 Visual Representation of the Three Hardware Outputs: a.) Original Pixel b.) Response c.) Direction 50

Chapter 1

Introduction

Real-time tracing of the vascular structure of retinal images is an important part of several medical applications, including live laser surgery. Some unique challenges are involved because of the need to attain reliable results from images that are acquired in non-ideal conditions. Poor illumination in conjunction with the inability to completely stop all body and eye movements in a live setting contribute to the challenges. These problems are further complicated by the fact that high data rates require a large amount of computation to be performed in very short periods of time.

With real-time vascular tracing results, it is possible for a surgeon to have an image of the retina that is being operated on with highlighted blood vessels. It is also possible to create a system where the surgical laser can be automatically shut off if it is detected that it is off-course. For these applications to be possible, not only do the results have to be computed at the same rate that the image data is output from the camera, but it is also important that there is very little delay in making the results available.

One part of the tracing algorithm requires the computation of several two-dimensional convolutions of the image data with 16 different filters. A hardware implementation of these convolutions is a good choice due to the large amount of data that must be processed in a short period of time. In order to provide the convolution results to the tracing algorithm, we have designed a “Smart Camera,” which includes an FPGA board attached directly to the framegrabber of the camera. The “Smart Camera” name comes from the fact that we are able to provide the host PC not only with the image data coming from the camera, but also with the results of the filter operations on every pixel in real-time. The results are sent to the host with a delay of less than one frame.

Chapter 2 of this thesis begins by describing the Retinal Vascular Tracing (RVT) algorithm, and then provides a background for reconfigurable hardware and a description of the FPGA board chosen for our hardware implementation. An introduction to some hardware design methods is provided, and related work is presented in this chapter. Chapter 3 describes the hardware design, including the filter implementation, the FPGA’s memory interface, and the interface with the host processor. Chapter 4 presents the results of the hardware implementation and discusses the performance of the design. Chapter 5 gives a conclusion to the work presented, and offers suggestions for future work.

Chapter 2

Background

In order to understand the concepts behind our hardware implementation, this chapter describes the basic tracing algorithm used in RVT, the reconfigurable hardware used in our implementation, and some parallel processing background. Related work is presented at the end of this chapter.

2.1 Retinal Vascular Tracing Algorithm

The RVT Algorithm was designed by colleagues at Rensselaer Polytechnic Institute (RPI) and implemented in software [1]. The algorithm is split into two parts. The first part of the algorithm detects and validates several seed points that are known to be on blood vessels. The tracing algorithm uses these seed points as initial points. The second part of the RVT algorithm is the basic tracing algorithm which starts at one point on a blood vessel, finds the next point on that same vessel, and continues to follow its path until it reaches the end. To simplify the explanation of the algorithm, the tracing step is described first, followed by the seed point detection algorithm. All of the background information on the RVT algorithm in this section is from [1].

2.1.1 Tracing Algorithm

In order to begin tracing, an initial starting point, $\vec{p}^k$, and orientation, s^k, are necessary. The starting point is in the center of a blood vessel, and the orientation gives the direction that the vessel is pointing. The initialization points are found in a pre-processing step described in Section 2.1.2. Vector $\vec{u}^k$ is the unit vector along the blood vessel at point $\vec{p}^k$, defined by Equation 2.1 [1].

$$\vec{u}^k = \begin{bmatrix} u_x^k \\ u_y^k \end{bmatrix} = \begin{bmatrix} \cos\left(\frac{2\pi s^k}{16}\right) \\ \sin\left(\frac{2\pi s^k}{16}\right) \end{bmatrix} \tag{2.1}$$

The next point on the vessel, $\vec{p}^{k+1}$, and its orientation, s^{k+1}, can be found by investigating the orientation of the vessel's boundaries. This is accomplished by calculating the results from a set of two-dimensional correlation kernels, designed by researchers at RPI [1], shown in Figure 2.1.

Each of the 16 templates represents a unique direction. The angles for these directions are discrete values separated by 22.5°. Given a target pixel and the correct neighborhood of pixels, each kernel will give a scalar result by multiplying the grayscale values of the pixels by 0, ±1, or ±2 as defined by the template. This result

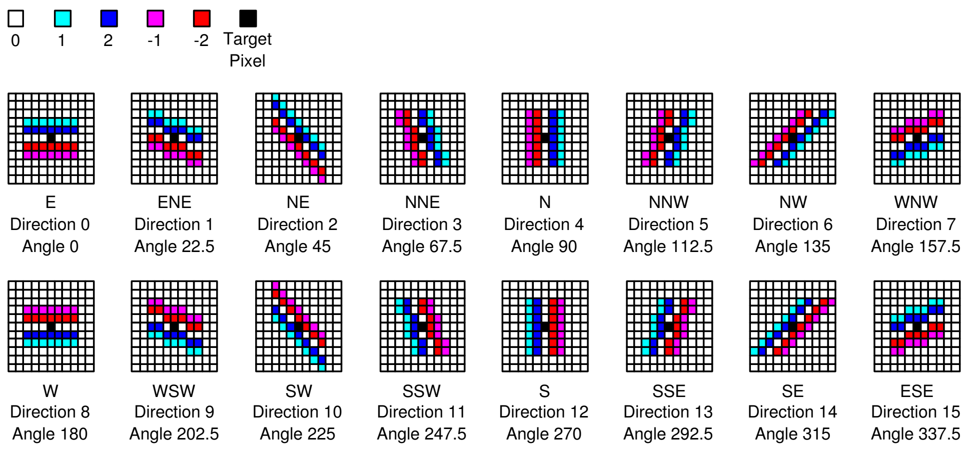

0
1
2
-1
-2
Target Pixel
E Direction 0 Angle 0
ENE Direction 1 Angle 22.5
NE Direction 2 Angle 45
NNE Direction 3 Angle 67.5
N Direction 4 Angle 90
NNW Direction 5 Angle 112.5
NW Direction 6 Angle 135
WNW Direction 7 Angle 157.5
W Direction 8 Angle 180
WSW Direction 9 Angle 202.5
SW Direction 10 Angle 225
SSW Direction 11 Angle 247.5
S Direction 12 Angle 270
SSE Direction 13 Angle 292.5
SE Direction 14 Angle 315
ESE Direction 15 Angle 337.5

Figure 2.1: Matched Filters

is referred to as the template response. The template with the greatest response for the same target pixel represents the orientation of the vessel at that pixel.

Blood vessels are defined by two boundaries, which are parallel lines separated by some distance. The boundaries are labelled as left and right with respect to the vector $\vec{u}^k$. In order to predict $\vec{p}^{k+1}$ and s^{k+1}, the orientation of the vessel boundaries must be found. Figure 2.2 shows the orientation of the left and right templates on the boundaries of a vessel at 0°.

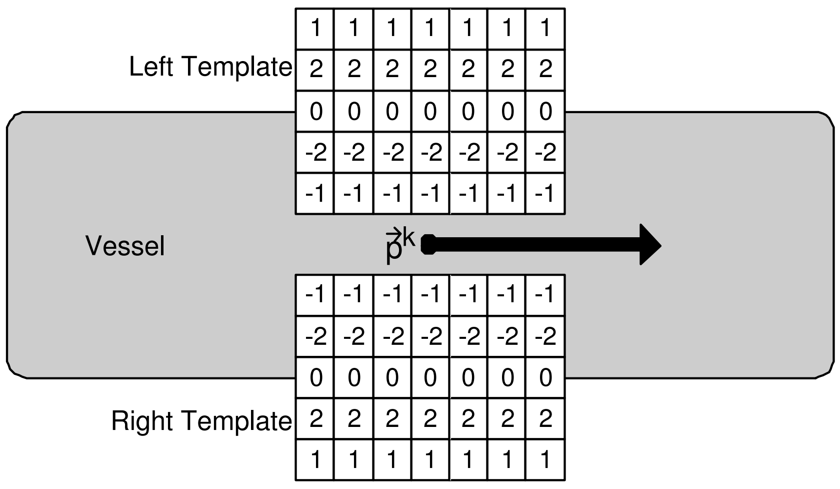


Figure 2.2: Left and Right Templates for a Vessel at 0°

The width of the vessel being investigated is unknown ahead of time, so to find the left boundary, the templates are applied first to the point $\vec{p}^k$, then to the point one pixel to the left of $\vec{p}^k$ in the direction perpendicular to $\vec{u}^k$. The maximum expected vessel width is predetermined, and given as M, so the templates are shifted to the left by one pixel $M/2$ times. At first, the left template responses are calculated for the direction s^k. $M/2$ responses are calculated, and two numbers are found: the maximum response value, $L_{max}(s^k)$, and the distance where the greatest template response is found, $d_L(s^k)$. $R_{max}(s^k)$ and $d_R(s^k)$ are found in a similar manner using the template responses shifted to the right of $\vec{p}^k$ in the direction perpendicular to $\vec{u}^k$.

Once the right and left template results are found for direction s^k, the other directions must be tested. Since the direction of the vessels changes gradually over short distances, only the template responses for the angles neighboring s^k need to be calculated. Equation 2.3 defines the initial orientation for the next iteration of the tracing algorithm [1].

$$s^{k+1} = \arg\max\{\max\{R_{max}(s), L_{max}(s)\}\} \tag{2.2}$$

$$for\ s \in \{(s^k - 1) \bmod 16, s^k, (s^k + 1) \bmod 16\}$$

Given s^{k+1}, $\vec{u}^{k+1}$ can be found using Equation 2.1. The new position vector is estimated as

$$\vec{p}^{k+1} = \vec{p}^k + \alpha\vec{u}^{k+1} \tag{2.3}$$

where α is a step size [1]. With $\vec{p}^{k+1}$ and s^{k+1}, the tracing algorithm can begin the next iteration to find $\vec{p}^{k+2}$ and s^{k+2}.

The tracing is terminated if one of the following conditions are satisfied [1]:

1) The new position vector is outside of the image field.

2) The current vessel intersects with a previously detected vessel.

3) The sum of the right and left template responses, $R_{\max}(s^{k+1}) + L_{\max}(s^{k+1})$ is less than a predefined threshold, T.

More information on the tracing algorithm, including the handling of jumps and branches, preventing repetitious tracing, and adaptive parameter estimation, can be found in Can, et. al. [1].

2.1.2 Finding Seed Points

Line Searches

In order to find the initial points to begin the tracing algorithm, a coarse grid of 1-pixel wide lines is placed over the grayscale retinal image. The pixel values along each line are filtered with a one-dimensional discrete approximation of a Gaussian kernel with the three coefficient values 1/4, 1/2, and 1/4 [1]. This filtering step returns the local minima along the line, which corresponds to pixels which are darker than their neighbors; a characteristic of blood vessels. Figure 2.3 shows a coarse grid and the seed points found by the line searches on a retinal image. Some seed points are found that are not actually blood vessels due to noise in the image. The seed points must be validated before the tracing algorithm can begin.

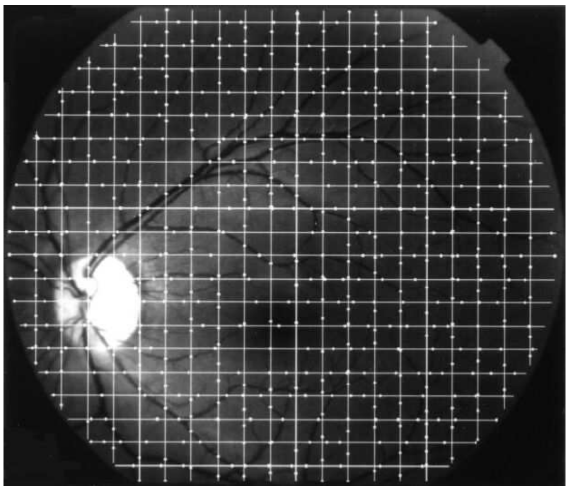


Figure 2.3: Initial Seed Points From Line Search [1]

Seed Point Validation

A valid seed point must be on a blood vessel. To verify this, it must be shown that vessel boundaries exist, i.e. two lines that are nearly parallel and separated by a short distance. In order to find out if the edges are parallel, the orientation of each edge must be found by using the direction kernels described in Section 2.1.1.

Since there is no knowledge of the orientation of the seed point, responses for all 16 directions must be calculated for both the left and right templates. In order to be considered a valid seed point, the following conditions must be met [1]:

1) The responses of the right templates in all 16 directions must have two local maxima.

2) The directions between the local maxima computed in 1) must differ by $180° \pm 22.5°$.

3) The responses of the left templates in all 16 directions must have two local maxima.

4) The directions between the local maxima computed in 3) must differ by $180° \pm 22.5°$.

5) The directions between the local maxima computed in 1) and 3) should differ by $\pm 22.5°$.

6) The sum of local maxima computed in 1) and 2) should exceed the sensitivity threshold T.

These rules are checked in the order that they are listed above, and if any rule fails, then the seed point is not validated, and the next seed point is evaluated. Validated points are saved along with the pair of opposite directions that returned the maximum template responses. The tracing algorithm starts at the seed point and traces twice; once for each complementary direction. More information on the seed point selection and validation rules is available from Can, et. al. [1].

2.2 Reconfigurable Hardware

The goal of this research is to accelerate the RVT algorithm with reconfigurable hardware. There are also other methods available for implementing custom logic.

One method is to write software for a general purpose microprocessor. This solution is very flexible because the functionality can be changed by rewriting and compiling code in software. The disadvantage to programming with a microprocessor is the performance. The amount of parallel processing that can be achieved in a processor is limited. There is also additional overhead in software processing because each instruction must first be fetched, and then decoded before being executed [4].

Another method is to use Application-Specific Integrated Circuits (ASICs). Since the hardware in an ASIC is customizable, each chip is designed to maximize the performance of a specific application. The hardware in a microprocessor is not customizable, which is why there is additional overhead required to run any command. Although ASICs provide a fast solution, they are inflexible. The design on an ASIC cannot be changed after the chip has been produced. If alterations must be made, a new chip must be redesigned and fabricated [4]. The design time for an ASIC is long compared to software, and an ASIC redesign requires the fabrication of several expensive masks.

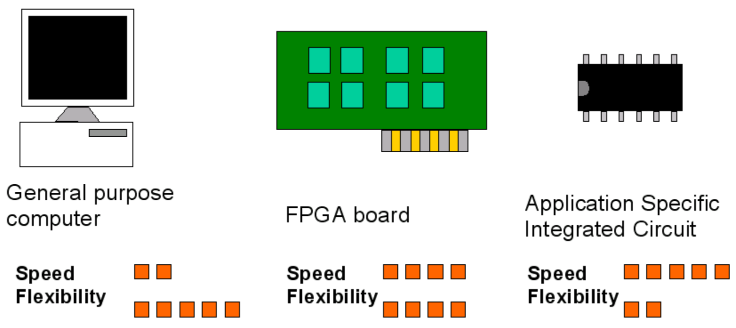


Figure 2.4: Speed and Flexibility Comparison for Software, FPGAs, and ASICs

FPGAs offer more flexibility than an ASIC, and greater speed than a general purpose microprocessor, as illustrated in Figure 2.4. An FPGA is made up of many functional blocks which contain customizable logic, and an interconnection network which connects the blocks. The logic and routing are determined by several configuration bits, programmed by sending a bitstream to the FPGA at initialization time. Implementing a design in an FPGA requires neither the cost nor the turnaround time of fabricating an ASIC. FPGA designs can also be verified in-circuit, by programming an actual chip and analyzing the results. ASIC design verification relies on complicated simulations that can take days to run [5].

One disadvantage to FPGAs is that they are slower than ASICs. The programmability of the interconnection network requires switches that an ASIC does not have. The additional circuitry adds resistance and capacitance to the internal paths [5]. FPGAs require an additional area overhead for programming circuitry that ASICs do not. As a result, FPGAs are larger in area. The greater area requires longer interconnection wires, which also accounts for additional delay.

2.2.1 FPGA Technology

Figure 2.5 shows the architecture of a Xilinx FPGA. There are I/O Blocks (IOBs) around the border, Configurable Logic Blocks (CLBs) in an array, BlockRAM in column stripes throughout the chip, and an interconnection network of wires and switching matrices.

The main logic to be programmed in an FPGA is contained in the CLBs. The CLBs in the Xilinx Virtex-E FPGA contain two slices, as shown in Figure 2.6. A slice

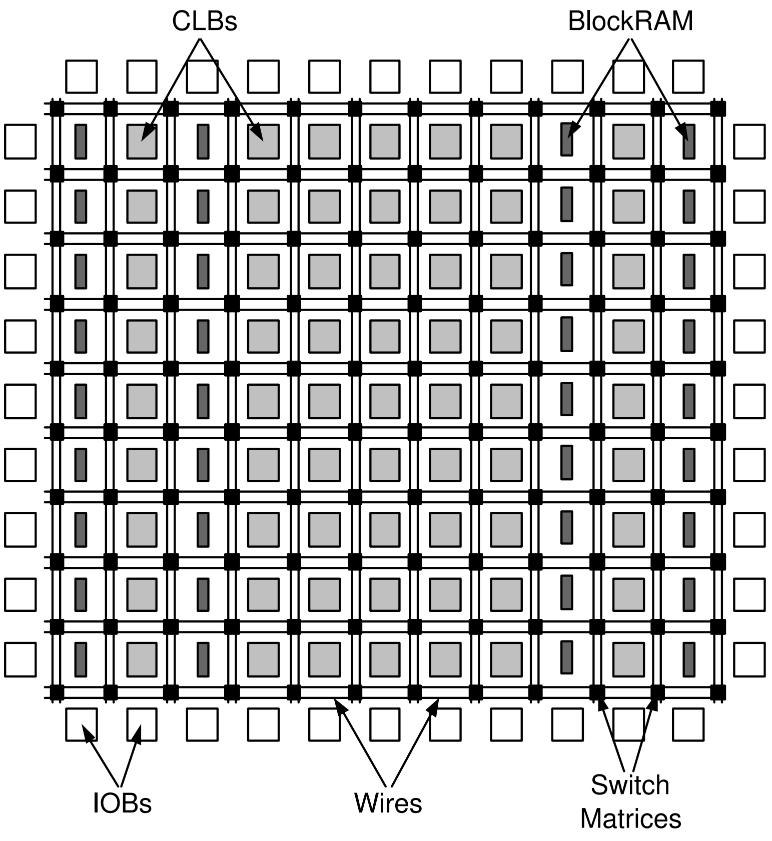


Figure 2.5: FPGA Architecture

consists of Look-up Tables (LUTs) and flip-flops. The LUTs have four inputs and one output. A LUT is programmed with 2^4 possible results, one for every combination of inputs. The contents of the LUT are customized so that any function with four inputs and one output can be realized. By connecting several LUTs, more complicated functions can be mapped to hardware. The flip-flops can be used to store the data from the LUTs. CLBs can implement combinational logic, registers, a combination of the two, or even routing.

BlockRAM is memory on the FPGA chip. Having data stored on the chip prevents having to make slower data transactions with the off-chip memory. The BlockRAM is

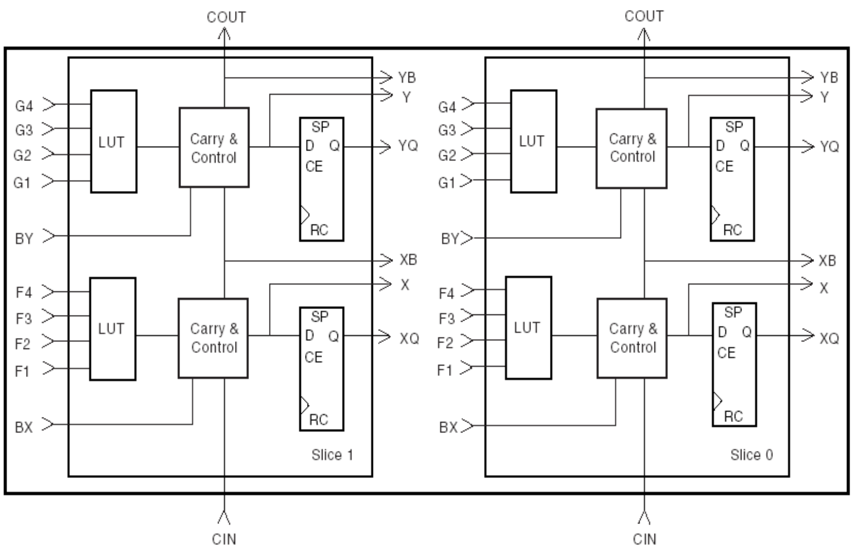


Figure 2.6: Xilinx Virtex-E CLB Structure [2]

implemented in columns, and the amount of BlockRAM and its location on the chip are predetermined by the FPGA manufacturer. If a design does not use the BlockRAM, it is wasted space on the chip. On the other hand, if all of the BlockRAM is used in a small design, the logic gets spread over the entire chip. Since the BlockRAM is distributed over the entire chip, using all of it in a smaller design requires long connections and more routing than may otherwise be necessary. Normally, it is desirable to keep all of the logic in a small area to avoid long wires and delays.

The IOBs are around the border of the FPGA. They enable communication between the FPGA and off-chip hardware. IOBs can be configured as input pads, output pads, or both.

The interconnection network consists of wires running through the FPGA and General Routing Matrices (GRMs) that control which components are connected. In addition, there is a local routing mechanism (shown in Figure 2.7) to speed up communications between CLBs that are in close proximity. Other connections can be made by passing through one or more routing matrices; however, every routing matrix adds more delay to the communication path. The routing matrices are configured to make the correct connections by programming bits in the configuration bitstream in the same manner as the CLBs are programmed.

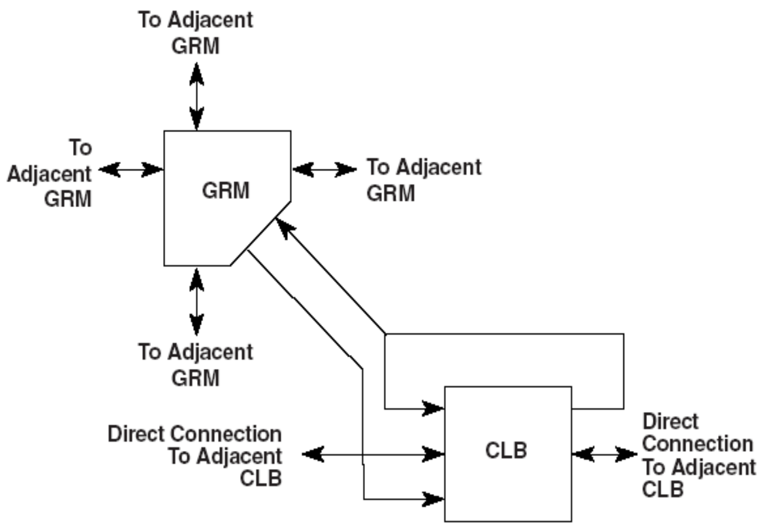

To Adjacent GRM
To Adjacent GRM
GRM
To Adjacent GRM
To Adjacent GRM
Direct Connection To Adjacent CLB
CLB
Direct Connection To Adjacent CLB

Figure 2.7: Xilinx Virtex-E Local Routing [2]

The FPGA is programmed with a bitstream which sets all of the configuration bits in the chip to a one or zero. Rather than having an FPGA designer manually set the thousands of configuration bits, higher level design tools are used. A Hardware Descriptive Language (HDL) is a software programming language used to model the intended operation of a piece of hardware [6]. The behavioral description of the hardware's functionality is written in HDL, and compiled with commercial tools to create the configuration bitstream.

The design flow for programming with HDL begins by writing the code and simulating the behavior using a program such as Modelsim, from Model Technology. Once the behavior of the design is verified, the behavioral description is converted to the Register-Transfer Level (RTL). The RTL is a description of the design using specific hardware, such as memories, adders, register files, and controllers. One software tool for compiling HDL code and mapping the logic to the RTL level is Synplify Pro, from Synplicity. The next step is to realize the circuit using the basic blocks of the FPGA architecture. Modern FPGAs can implement designs with millions of logic gate equivalents and hundreds of thousands of bits of BlockRAM memory. There are four main considerations to take into account when synthesizing circuits onto the architecture of an FPGA [7]:

1) The number of logic blocks on a chip

2) What functions can be put on a block

3) Wiring resources

4) Interconnection delay incurred in the switching elements.

Xilinx has its own tools for the logic placement and routing for their FPGAs. The place and route tool outputs the bitstream that is used to program the functionality of the FPGA.

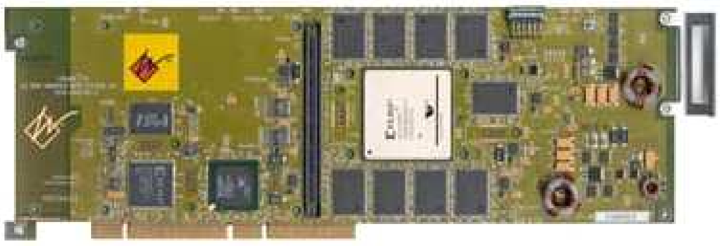


Figure 2.8: Annapolis Microsystems Firebird Board [3]

2.2.2 Annapolis Microsystems Firebird Board

FPGAs usually reside on a board with supporting hardware to interface with the host computer and supply additional memory. Annapolis Microsystems, Inc. sells commercial off-the-shelf FPGA boards. Figure 2.8 shows their Firebird board [3], which we use for our design. The Firebird contains a Xilinx Virtex 2000E FPGA. There are five banks of synchronous ZBT SRAM on the board, four of which are 64-bits wide, one which is 32-bits wide. The Firebird interfaces to the host processor via the PCI bus. The PCI bus is 64-bits wide and runs at 66MHz. There is a PCI controller on the board to manage host-board communication. The long connector in the middle of the Firebird board is a 228 pin port used for additional I/O. Figure 2.9 shows a block diagram of the Firebird board.

The FPGA is physically connected to all of the peripheral components on the Firebird, but the FPGA must be programmed to use these components. Annapolis Microsystems provides the HDL code for interfacing the FPGA to all of these components to simplify their use. For the host to communicate with the Firebird, there is an Application Program Interface (API) library provided by Annapolis Microsystems.

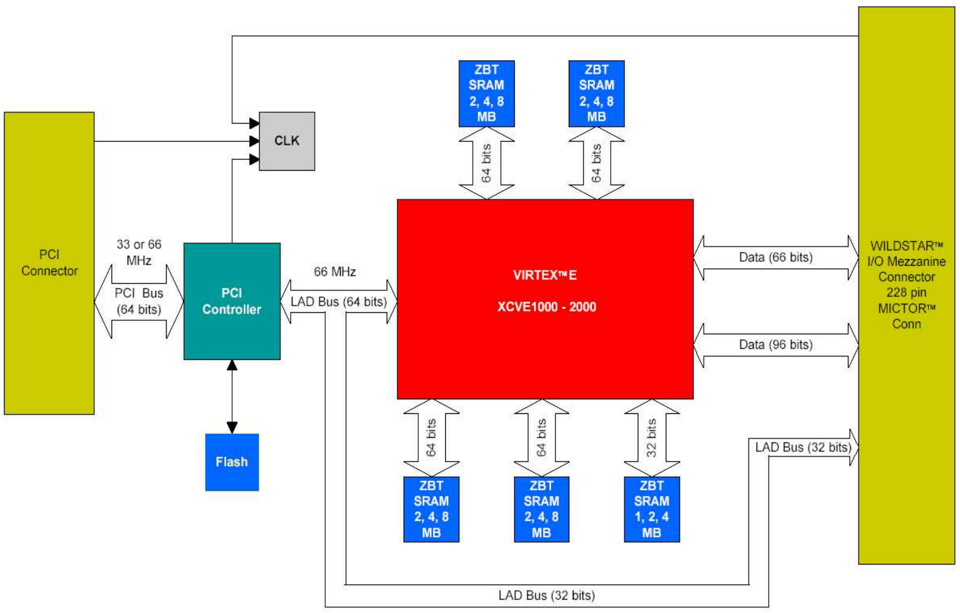


Figure 2.9: Firebird Block Diagram [3]

The API library contains functions which perform operations such as programming the FPGA, reading and writing data to the on-board memory, and reading and writing from registers on the FPGA. A C program containing calls to the functions in the API library is written by the user and run on the host.

2.3 Hardware Design Methodology

Writing code to be synthesized in hardware requires a different approach than writing software code. When programs are written in software, they are compiled and the instructions are executed in the microprocessor. The hardware in the microprocessor is designed with fixed pipelines and limited parallelism. The improved speed from a customized hardware solution is a direct result from being able to manipulate the hardware to maximize parallel instructions and create optimized pipelines. Hardware programming also requires careful consideration of memory accesses, which are often the slowest part of a design.

2.3.1 Parallel Processing

The main idea behind parallel processing is to run multiple operations at the same time. If an operation takes one clock cycle and needs to be performed 100 times, then it would take 100 total cycles to run sequentially, with one operating unit. Assuming that there are no data dependencies in any of the operations, if a second operating unit were available, then 50 of the operations could be run on each unit concurrently, and the total time for completion would be halved. To maximize parallelism and minimize processing time, 100 operating units could be used, and all of the computations could be completed in a single clock cycle. In order for parallel processing to work, the operations must be independent of each other; the results from any of the computations cannot rely on the results from previous computations.

The operations do not need to be the same for parallelization to be useful. For example, if two operands, A and B, need to be subtracted and compared, then these operations could be parallelized. Instead of first subtracting, then comparing, both operations could be run at the same time since their results are independent of each other.

FPGAs are efficient at processing data in parallel because the hardware functions are customizable. Logic can be easily duplicated to create more operating units for increased parallelism. Also, since the timing of the computation is controlled by the designer, different functions can be run simultaneously. The limit on parallelism is based on the number of independent functions inherent to the algorithm, and on the amount of space available for duplicated logic on the FPGA.

2.3.2 Pipelining

If a process is defined as a series of tasks, then the basic idea of pipelining is to speed up the computation of several processes by starting a new task before an old one has been completed [8]. This idea is best described with an example. If a process contains five tasks, A, B, C, D, and E, each taking equal time to complete, then it would take five cycles to complete the process once, and 10 cycles to complete two processes, as shown in Figure 2.10. If each task had a unique hardware unit, then in the first cycle, the hardware for A would be working, and the hardware for B, C, D, and E would be idle. At any given time, there is only one piece of hardware working in the sequential design.

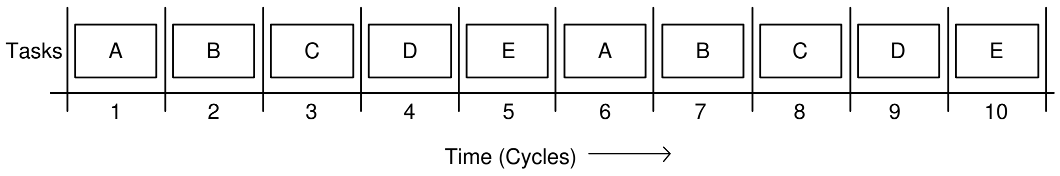


Figure 2.10: Sequential Processing Example

A pipelined version of this example works to maximize the amount of time each piece of hardware is working. Figure 2.11 shows that if a new process is started as soon as the first task is completed, then the hardware for each task is being used on every cycle. The effect of pipelining is that a result from a process is output from the system on every clock cycle instead of one result every five cycles. Each process still takes five cycles, but the overlapping of computation enables greater throughput. In this example, after the pipeline is full, there is a five times speedup compared to the sequential solution. In general, the speedup factor is equal to the number of pipeline stages if the time to fill up the pipeline is ignored.

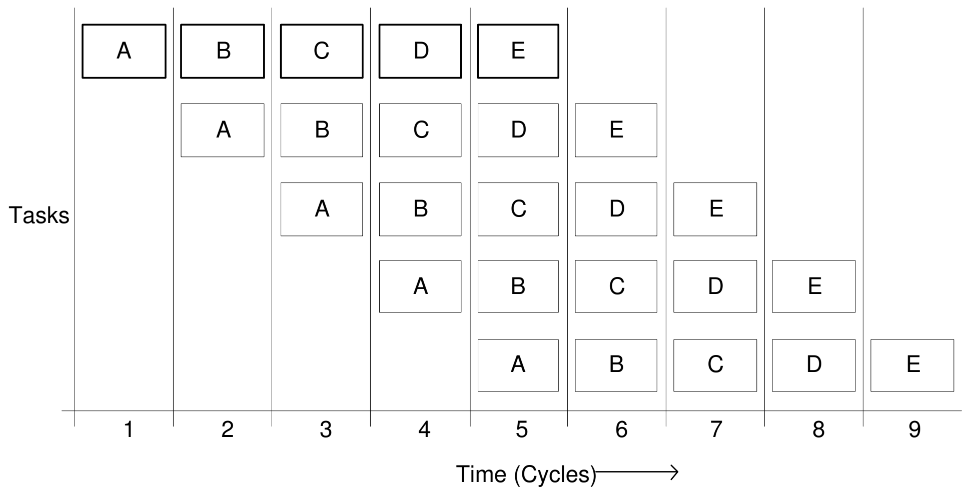
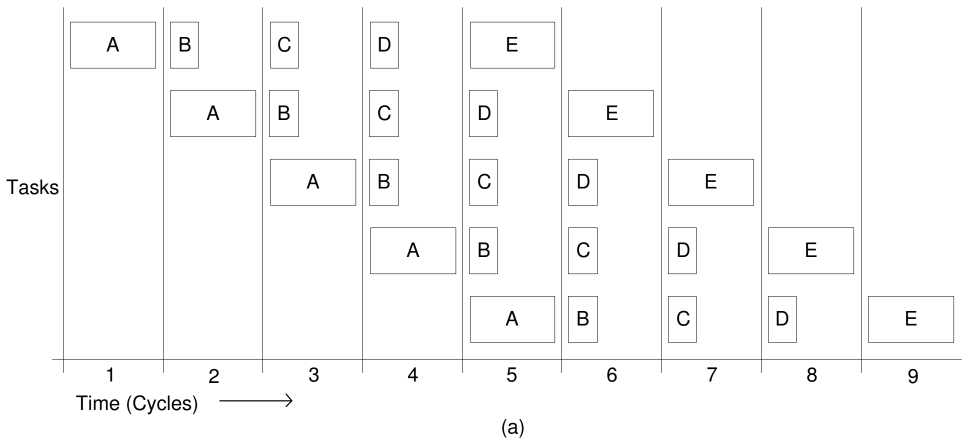


Figure 2.11: Pipeline Example

The completion rate for pipelined processes is not a function of the total processing time, but of how soon a new process can be introduced [8]. One thing to take into consideration is that each of the tasks in the previous example took the same amount of time. If this is not the case, the clock period is set to the amount of time it takes to process the longest task. The ideal situation is to balance the pipeline stages to take the same amount of time as the longest stage. If, for example, tasks A and E take three seconds, and tasks B, C,and D only take one second, then tasks B, C,and D could be combined to a single task taking the same amount of time as A and E.

Figure 2.12 shows the difference between an optimized and non-optimized pipeline for this example. The balanced pipeline stages minimize the length of the overall pipeline, and therefore minimize the latency of the design.



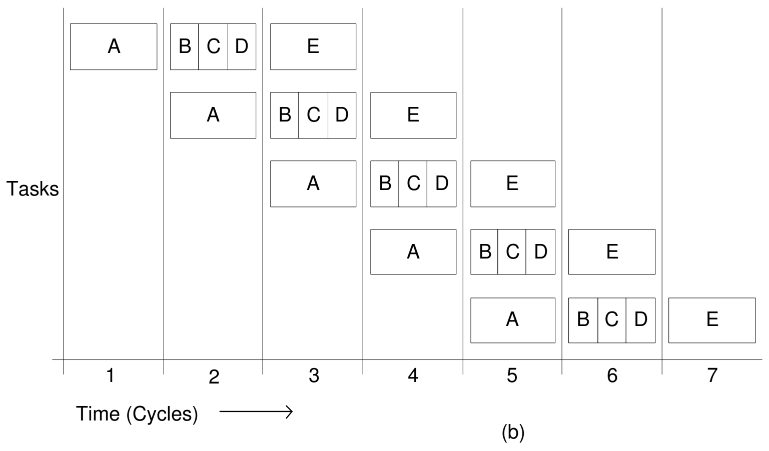


Figure 2.12: Pipelines for Unequal Task Completion Times: (a)Non-optimized (b)Optimized

Since the input data is changing every clock cycle, only the first task in the pipeline can use the incoming data for its input. To remedy this problem, pipeline registers are added to the hardware. The pipeline registers are in between every stage, and store all of the necessary intermediate data required for any future tasks. This adds additional overhead to the design, but the increase in speed justifies the cost.

2.3.3 Memory Considerations

There are three different types of memory used in our hardware architecture: on-chip, on-board, and host. The on-chip memory is also known as BlockRAM. This is the fastest of the three memory types since it uses short wires to connect to the FPGA logic. Since the amount of area on the FPGA chip is limited, so is the amount of BlockRAM available to use. The Virtex 2000E that we are using contains 160 blocks, each able to store four kilobits, for a total of 655,360 bits of BlockRAM. On-board memory refers to the five banks of memory on the Firebird board. This memory takes longer to access than on-chip memory, but there is a considerably larger amount of memory. There is a total of 18MB of on-board memory on the Firebird. The host memory is located off of the FPGA board. This is the slowest of the three memory types since there is a large physical distance between it and the FPGA, and the data must go over the PCI bus and through control logic. The amount of host memory available depends on the machine, but generally there is far more available memory on the host than on the board.

Since communications with the host PC are the slow compared to on-board communications, to maximize the performance of an FPGA design, it is necessary to minimize the amount of accesses to the host memory. Any large blocks of data should be stored in the on-board memory. Since BlockRAM takes the least amount of time to access, it is a good design strategy to preload data from the on-board memory to the BlockRAM, so that the FPGA only needs to process data from the fastest available memory. The additional delay of using host or on-board memory can be hidden by loading data into BlockRAM at the same time that data is being processed.

2.4 Related Work

There have been several projects working on research related to ours. It is important to investigate related work to avoid redesigning something that already exists and to get an understanding of how the field is progressing.

In 1989 Chaudhuri, et. al. [9] was able to detect retinal vascular structure by filtering a grayscale image with a set of two-dimensional directional filters. Their algorithm was implemented in software. Twelve 32 x 32 pixel filters, separated by 15°, were run over the image, and the filter with the greatest response for a window was returned. This is the same idea that we are using in our design, only implemented over a decade ago. The problem with the idea was that it took a very long time to process a single image. It was noted that implementation was designed for a hardware solution that could run the filters in parallel to increase the performance.

Sartori [10] designed a “Smart Camera” in 1991 that was able to take digital data directly from an image sensor and input it into an FPGA that performed edge detection on the images and returned the results to the host. Although the input images only had a resolution of 32 x 32 pixels with 1-bit monochrome pixel data, and the FPGA was running between 3 and 4 MHz, the idea of moving image processing to hardware and closer to the sensor enabled a real-time solution for his application.

Scalera, et. al. [11] designed the CAμS board which included an FPGA and a DSP that interfaced directly to microsensors and preprocessed the data in order to reduce the amount of information necessary to transmit to the host processor. The goals of that project were to minimize power consumption and size. The CAμS project was not as concerned with the speed and latency of their design as we are, however, the idea of processing data before it gets to a host PC is the same. Lienhart, et. al. [12] implemented an image processing design in real-time using FPGA technology. The application was video compression, and they were able process three image sequences in parallel at 30 frames per second. The performance of their design suffered because their input data came from the host memory via the PCI bus. It was noted that the solution could be improved by connecting the camera directly to the FPGA board. Cerro-Prada and James-Roxby [13] describe methods for implementing 3 x 3 convolutions with arbitrary weights using distributed arithmetic in an FPGA. While our design uses 11 x 11 filters, our computation is significantly simpler since we require no multiplications with our coefficients set at ±1 and ±2. Their experiments were run by inputting data from the host memory over the PCI bus. The limited I/O of their system reduced the overall performance to an order of magnitude short of being able to run at 25 frames per second.

A software solution for a real-time retinal tracking system was proposed by Solouma, et. al. [14]. The algorithm is designed to be run in software, so it is different from our design. They find the vascular structure in one image, and using image registration techniques, track the motion of the patient's retina from frame to frame in real-time. The performance of this algorithm is data dependent; interframe shifts and rotations of different amounts require different processing times. They reported that the time to process one frame was on the order of hundreds of milliseconds. Our design enables real-time retinal vascular tracing by processing frames on the order of tens of milliseconds, as well as having a minimized delay between the time frames are captured by the camera and results are available to the user.

2.5 Summary

This chapter gives the background information for this thesis. The Retinal Vascular Tracing algorithm is described, and FPGA technology is discussed. Related work by other researchers is included. In the next chapter, the hardware algorithm and implementation are presented.

Chapter 3

Hardware System Design

This chapter explains how parallelism is extracted from the filter response calculations, and how the calculations are implemented in hardware. Also described in this chapter are the methods used to insure that new data is continuously being processed in order to keep up with the rate at which new data is being streamed into the design. The memory, framegrabber, and host interfaces are described in detail.

3.1 Algorithm Analysis

The calculation of the 16 direction templates is the core component of RVT. Many template responses must be calculated since they are required for both the tracing and seed point validation steps. Also, due to the nature of the computations, the large number of responses to be calculated, and the limited parallelism available in a microprocessor, this is also the most computationally complex part of the algorithm.

To better understand the template response calculations, it is helpful to look at an example. Figure 3.1, repeated from Section 2.1.1, shows the 16 different filters, and the coefficients corresponding to each pixel for each template. If we are looking for

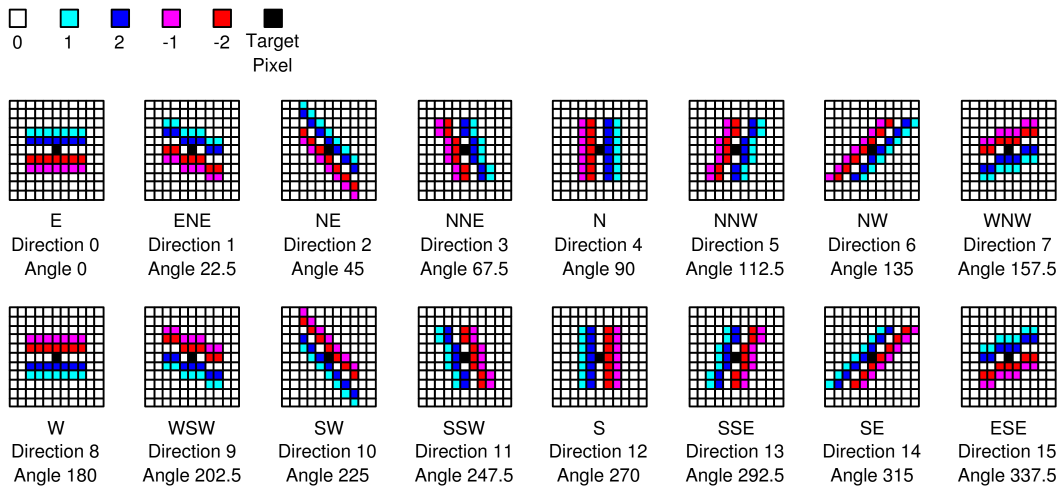

0
1
2
-1
-2
Target Pixel
E Direction 0 Angle 0
ENE Direction 1 Angle 22.5
NE Direction 2 Angle 45
NNE Direction 3 Angle 67.5
N Direction 4 Angle 90
NNW Direction 5 Angle 112.5
NW Direction 6 Angle 135
WNW Direction 7 Angle 157.5
W Direction 8 Angle 180
WSW Direction 9 Angle 202.5
SW Direction 10 Angle 225
SSW Direction 11 Angle 247.5
S Direction 12 Angle 270
SSE Direction 13 Angle 292.5
SE Direction 14 Angle 315
ESE Direction 15 Angle 337.5

Figure 3.1: Matched Filters

the response for the template corresponding to Direction 0 for a specific target pixel, first the pixels with non-zero coefficients for Angle 0 are loaded. Note from Figure 3.1 that there are 28 pixels with non-zero coefficients for every template. The seven pixels with the “1” color code have their values multiplied by one and added together. The seven pixels with the “2” color code are multiplied by two and added together. The sum of these two intermediate results are the positive template response. The same steps are taken for pixels with the “-1” and “-2” color codes to get the negative template response, which is less than zero. The positive and negative responses are summed to compute the total response of the template for Direction 0. The responses are also calculated for the other filters, and then compared to find which template returns the highest response.

Analyzing the filter response calculations reveals some optimizations to reduce the computational complexity. First, all of the filter coefficients are ±1 and ±2. A multiplication by two is equivalent to a single binary shift to the left. All of the multiplications in this application can be replaced with simpler binary shifts and sign changes. It can also be noted from Figure 3.1 that the filters for Direction 0 through Direction 7 are the same as the filters for Direction 8 through Direction 15, except for a change in sign. Only eight of the responses need to be calculated; the other eight can be found by simply copying the results and changing the sign. All of the template responses are also independent of each other. This implies that given all of the pixel data required to compute every template response, all of the responses can be calculated in parallel, and the results returned at the same time. To be sure that all of the data is loaded so that every template can be calculated in parallel, we overlap all 16 of the templates from Figure 3.1, and look only at the pixels with non-zero coefficients. In Figure 3.2, the black squares show which pixels need to be input for the parallel filter calculations to be possible. To simplify our memory reading scheme, we input the entire 11 x 11 pixel window into the algorithm. For every template calculation, only the 28 pixels required are used from this window of data.

The hardware implementation is not a straight-forward transformation of the RVT algorithm from software to an FPGA. Due to the complexity of calculating many template responses with a microprocessor, the software algorithm uses techniques such as seed point detection to minimize the number of filter calculations. The

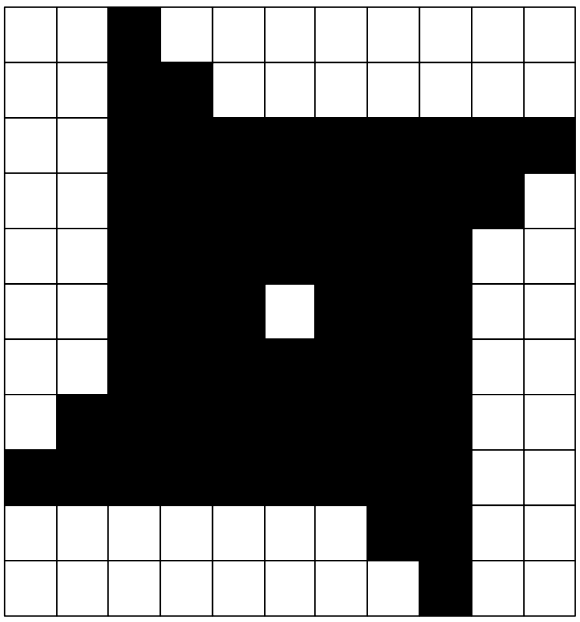


Figure 3.2: Overlapping of all 16 Templates

hardware algorithm calculates the responses for all directions for every pixel. This is because FPGAs are very good at performing the same computation repeatedly on large data sets. Since every pixel of every frame sent to the FPGA before being sent to the host processor, it is simpler to get results for every pixel than it is to search and find which pixels to compute responses for. Another difference between the hardware and software implementations is that the software implementation performs the complete retinal vascular tracing algorithm. The hardware only calculates the template responses, then sends the results to the host processor. In this case, the software still performs the tracing algorithm, but instead of calculating the template responses, it looks them up in the results returned by hardware.

3.2 Approach

From the algorithm analysis, it is seen that the filter response calculations can be computed as a series of parallel shifts and adds. A hardware solution can take advantage of the parallelism, and offer a great deal of speedup compared to a serial solution implemented in software.

An FPGA can be programmed to perform these filter response calculations in parallel. In order for these results to create a worthwhile speedup in the overall RVT algorithm, these results need to be available to the host processor very quickly. Since the communication between the host and the hardware board is over a relatively slow interface, the best way to speed up the overall design is to minimize host-board communications. To achieve this goal, we designed the "Smart Camera" shown in Figure 3.3.

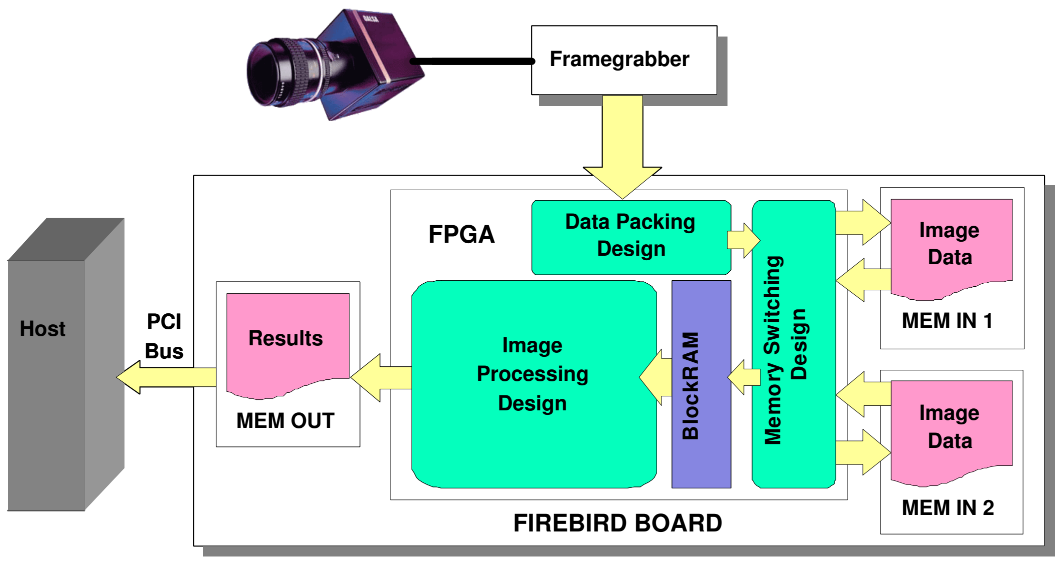


Figure 3.3: "Smart Camera" Hardware System

The FPGA board receives data directly from the camera, before the host receives it. There is a framegrabber, designed by Dillon Engineering, which captures the data from the camera and interfaces directly to the FPGA over the I/O port on the Firebird board. The FPGA stores the data, received in raster scan order, in the on-board memory. The data is retrieved from memory, and the filter response calculations are performed. The results are stored in a different bank of on-board memory, and the host retrieves them. Without the hardware board, the host would receive image data from the camera at frame rate. The reason that the hardware solution is referred to as a "Smart Camera" is because from the host's perspective, unaltered image data as well as results from the filter calculations are received at the same rate that the image data would be received from the camera if no additional hardware processing was being done.

In addition to designing a hardware solution that can output results at the same rate that new image data is coming from the camera, latency is a large concern in this application. Since this design is intended for use in a live surgery setting, and the human eye can move very rapidly, the tracing results must be available with a minimal delay to insure that they are still accurate when they are received. The goal of our research is to enable a real-time solution while minimizing the overall latency introduced by the hardware.

3.3 Implementation

The function of the FPGA solution is to collect streaming image data from a camera at frame rate, perform the filter response calculations for each pixel in the fastest and most efficient way possible, and send the results over the PCI bus to the host PC to be stored in the host memory and made available to the software RVT algorithm. For each pixel, three results are returned:

1) The original grayscale value of the pixel,

2) The direction (0-15) which has the greatest response, and

3) The value of the response for the direction chosen.

3.3.1 Filter Response Calculation

Figure 3.4 shows the `Response` module, where the filter responses are calculated on the FPGA. There are eight identical copies of the `Response` module that each correspond to one of the eight unique direction templates. The pixels that are input are split into four groups, based on the four coefficients the pixels will be multiplied by, as shown in Figure 3.1. There is a three-deep adder tree for each group of pixels, which finds the total response for the corresponding coefficient. Next there is a binary shift operator for the coefficients that require a multiplication by two. The two positive responses are added, as are the two negative responses. This gives two intermediate results: the total positive response, POS, and the total negative response, NEG.

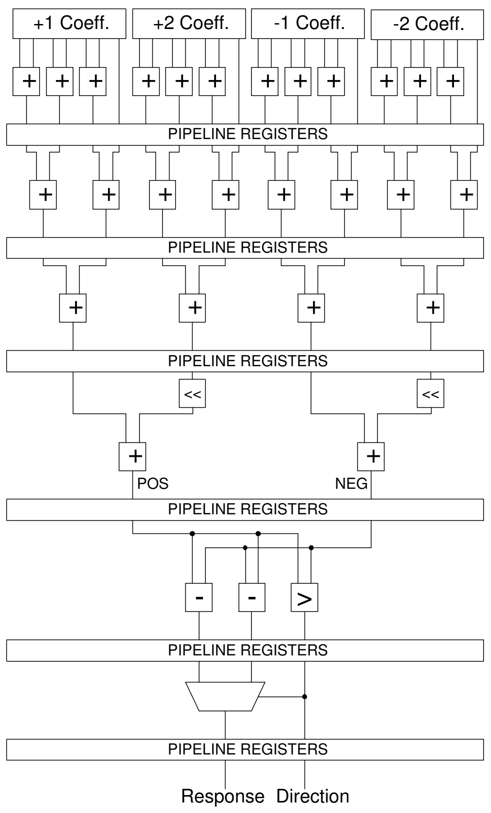


Figure 3.4: Response Module

Recall that each filter's complement response, found by changing the sign of the original response, must also be considered. All 16 filters, the eight calculated and their complements, must be included in the search for the greatest response. However, since only the greatest response is required, we can immediately eliminate the filters with a negative response. Each `Response` module returns the absolute value of the response, along with a flag that is set high if the filter's complement was chosen. To find the absolute value of the response, there are two parallel subtractors. One finds the result of $POS - NEG$ and the other finds $NEG - POS$. In parallel with these subtractors is a comparator that finds whether the POS or NEG response is greater in value. The output of this comparator is used to select the subtractor output that was positive in the next stage. The comparator output is also stored as the DIRECTION output, which is the flag telling whether the RESPONSE output is for the specified template, or for its complement.

Note that there is a pipeline register in between every set of parallel computations. While these registers may require additional space on the FPGA and add several clock cycles of latency, they are necessary so that we can achieve clock speeds that can keep up with the frame rate of the camera.

The `Direction` module is shown in Figure 3.5. The `Direction` module contains eight of the `Response` modules previously described. This module includes the interconnection network that insures that the correct pixels from the 11 x 11 window are input to the correct `Response` modules. Once the results from the eight `Response` modules are valid, there is a comparator tree that finds the response which is the greatest, and stores the corresponding direction as a four bit label. The maximum response and the four bit label that identifies that response are output from the `Direction` module to be written into the on-board memory. Note that there are pipeline registers between each comparator.

3.3.2 Incoming Data Packing

Image data from the framegrabber is input to the FPGA at a rate of one pixel per clock cycle in raster scan order. The order of pixels begins in the upper left corner of the image and moves along the top row from left to right. Next the left-most pixel in the second row is input, followed by the rest of the row. This order continues until

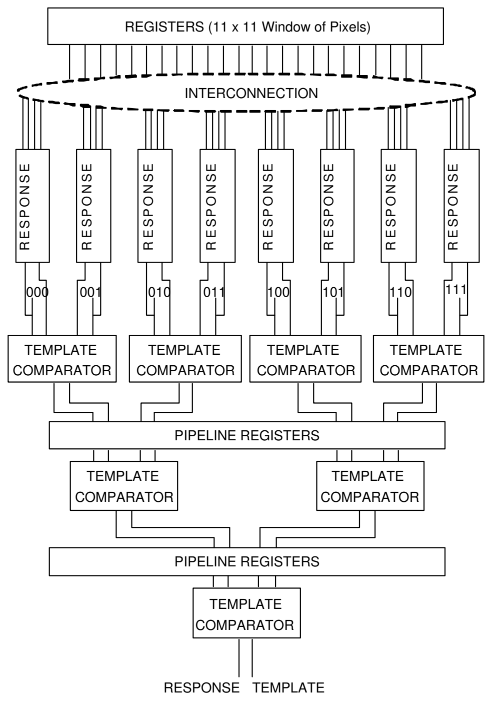


Figure 3.5: Direction Module

the pixel in the lower right corner is read. Then the next frame is input in the same manner.

The camera we are using has 12-bit pixels, and they need to be stored in the off-chip memory on the Firebird board, which is 64-bits wide. To minimize the number of memory accesses, we need to pack as much data into each word of memory as possible. Five consecutive pixels are stored in each 64-bit memory word with four wasted bits, as shown in Figure 3.6. A similar method of data packing is used in [15].

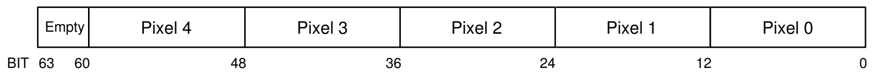


Figure 3.6: Five Pixels Packed Into One Memory Word

This method of storing incoming data requires that data be read into the design in groups of five pixels. The frame size for our application is 512 x 512. To simplify our data processing, we prevent pixels on different rows from being stored in the same word by padding each row with three additional columns of blank pixels. This makes the width of the image we process 515 pixels, which is divisible by five.

3.3.3 Neighborhood Module

The filter response calculations performed in the `Direction` module, described in Section 3.3.1, take an 11 x 11 pixel window as an input. Since pixels are read from memory row-wise in groups of five, we cannot efficiently input a window that is 11 pixels wide. Instead, we read in a neighborhood of pixels that is 11 rows by 15 columns. The `Neighborhood` module takes the 11 x 15 window and splits it up into the five separate 11 x 11 pixel windows for the `Direction` module, as shown in Figure 3.7.

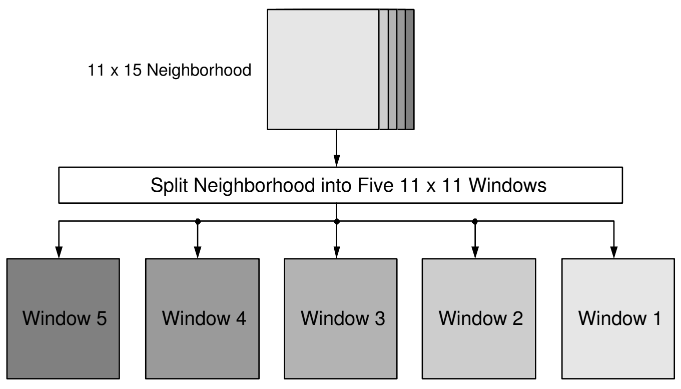


Figure 3.7: Splitting an 11 x 15 Pixel Neighborhood Into Five 11 x 11 Pixel Windows

Since the `Direction` logic is pipelined, we can input a new window of pixels to it every clock cycle. While window one is in the second cycle of being processed, window two can be in the first processing cycle, and the third, fourth, and fifth windows can each be input on consecutive clock cycles. The `Neighborhood` module puts the correct pixel data on the `Direction` module's input registers, along with a signal that tells the `Direction` module that valid data is ready to be processed. When the results are output from the `Direction` module, the results from each window are output on consecutive clock cycles, and the `Neighborhood` module passes these results to the output memory interface. Figure 3.8 shows how the processing of the separate windows overlaps.

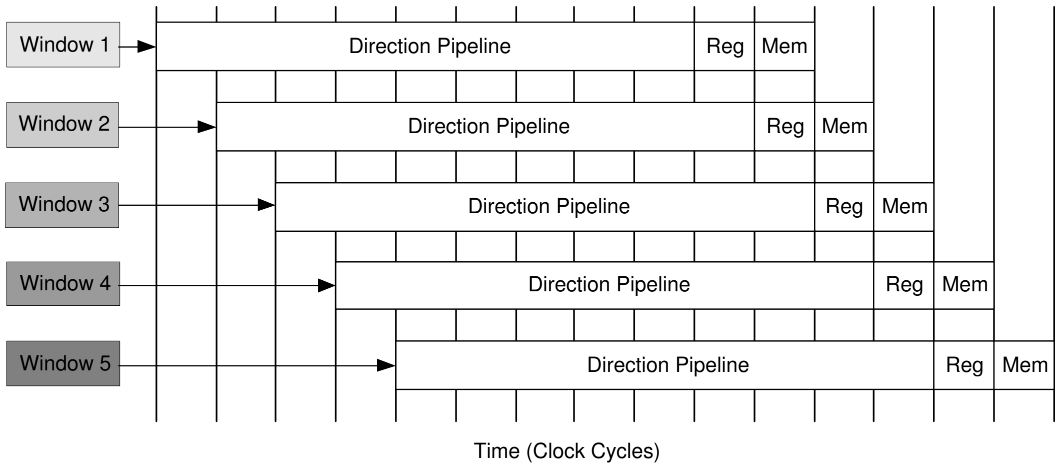


Figure 3.8: Inputting Windows to the Direction Pipeline

3.3.4 Traversing the Image

Processing a single neighborhood is only a small part of this design. The challenge is providing data is such a way that a stream of neighborhoods can be processed consecutively. New image data must be made available at a rate fast enough to keep up with the frame rate of the camera.

Every neighborhood that is input returns results for five pixels. To get results for the next five pixels, the neighborhood is shifted to the right by five, as shown in Figure 3.9. Adjacent neighborhoods contain ten columns of common data. Since reading data from the off-chip memory is the slowest part of the design, we only read in the five new columns of pixels when shifting neighborhoods.

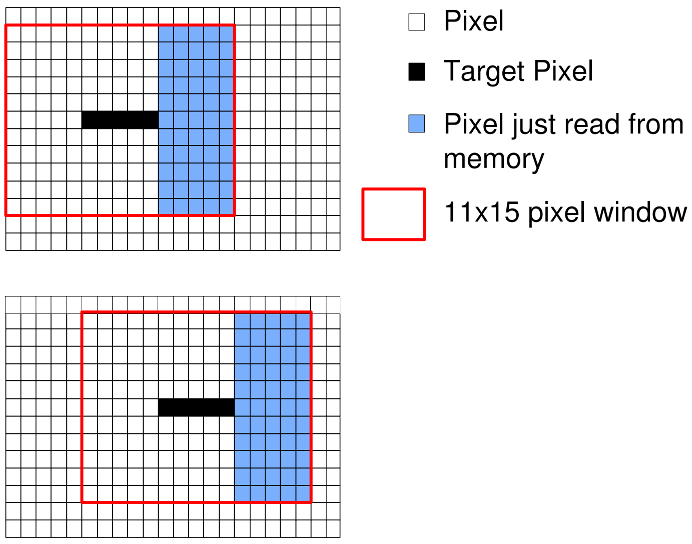


Figure 3.9: Shifting the Window

A mix of BlockRAM and registers are used to store the 11 x 15 pixel neighborhoods, so the data can be quickly accessed by the FPGA logic. A neighborhood is stored in three 11 x 5 pixel sections. The idea is that to shift the neighborhood, we can throw away one section, move the data from the other two sections, which is common to the adjacent neighborhood, and read in 11 new words from memory to fill the third section. Figure 3.10 shows how the shifting of data works.

To run the design as fast as possible, new data must be coming in from the off-chip memory on every clock cycle. One word of memory can be read every clock cycle, so it

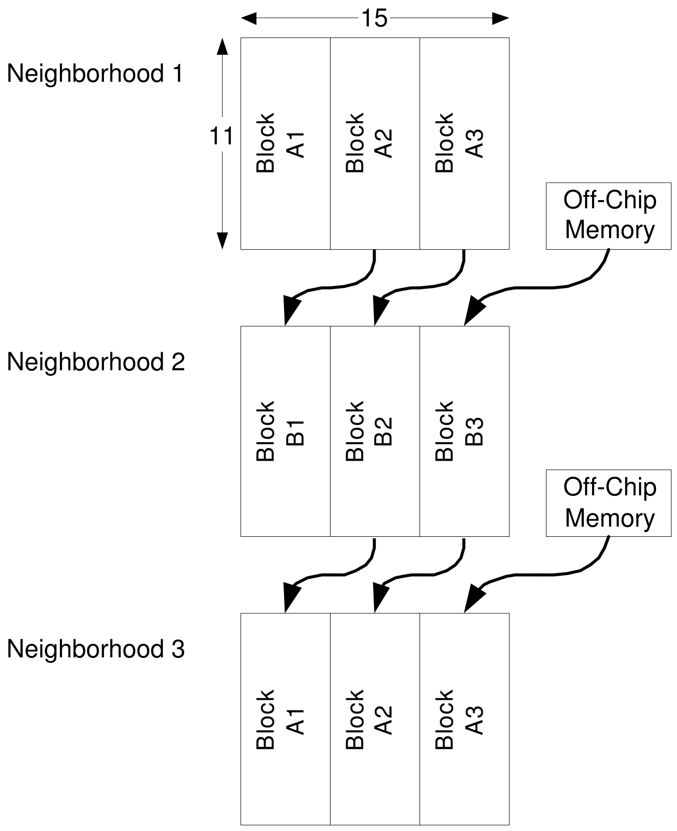


Figure 3.10: Creating New Neighborhoods by Shifting Data on the FPGA

takes 11 clock cycles to fill up the third 11 x 5 section of the neighborhood. This data cannot be processed, however, until it is all transferred from the off-chip memory. To avoid having to process the data, then wait 11 cycles before processing it again, two banks of on-chip memory are used. While one bank is being processed, the other is being filled with new data. Copying the 11 x 10 section of recurring data can be done in one clock cycle, in parallel with the memory reads. As soon as the 11 clock cycles are up, and the new neighborhood is valid, the two banks reverse roles, and new data starts to fill the bank of on-chip memory that was previously being processed. The amount of processing done is maximized, and five results are computed every 11 clock cycles. Different methods for buffering image data are described in [16].

The neighborhood shifts from left to right until it gets to the end of the row, then it needs to shift down one pixel and start at the beginning of the next row. Since the data is stored in the memory in raster scan order, if we continue to increment the memory addresses in the same manner, and continue reading data from the off-chip memory, the window will automatically shift down to the next line as desired. At the end of every row, however, there are two filter calculations that are invalid because the window spans two different rows. Calculating invalid results is a price that we are willing to pay, because hardware algorithms are more efficient when there are fewer special cases, and can run uninterrupted over large sets of data.

The frames are 512 x 512 pixels, so we pad the right hand side of the image with three columns of zeros to create a 512 x 515 image; i.e. an image with a width that is divisible by five. This allows us to continue shifting the window to the right by five pixels without creating a problem at the end of a row. Due to the nature of the application, boundary conditions are not an issue. There is no important information on the edges of the image, so any invalid results that occur when an 11 x 11 window spans multiple rows can simply be ignored.

3.3.5 Memory Interface

Memory management is an important consideration in real-time hardware solutions. Data streams in from the framegrabber continuously, so the incoming data must be stored and overwritten carefully. Another, larger problem is that a single on-board memory chip cannot be read from and written to on the same clock cycle. Like many other hardware applications, reading data from off-chip memory into the FPGA is the bottleneck in speeding up the design. The only way to prevent the entire design from having to wait for new data is to read a new word from memory on every clock cycle. However, if every clock cycle is dedicated to reading data, then there would be no free clock cycles to write new data to the same memory chip. To solve this problem, two memory chips must be used to store the incoming data.

One solution to the problem is to store the first frame that comes from the camera into Memory 0. The next frame is stored into Memory 1, and while new data is being written into Memory 1, the data from Memory 0 could be read into the FPGA with no conflicts. This "ping-pong" method would continue, so the memory chip that is being read from and written to alternates every frame, avoiding concurrent reads and writes on the same memory chip. While this is a common solution to streaming data applications, it introduces unacceptable delay in our application. No processing can begin until the first frame is completely written, and this latency carries through the entire design. It is essential in our application to minimize the latency.

In order to calculate all of the filter responses shown in Figure 3.1, an 11 x 11 pixel neighborhood is required. Since the data coming from the camera is in raster scan order, we can theoretically start processing data after the first 11 rows of data are available. In order to make this happen, we designed a more complex memory swapping method. After the `Data Packing Design` packs five incoming pixels into a 64-bit word, every consecutive word is stored in a different memory. The first word into Memory 0, the second word into Memory 1, etc. After the first 11 rows of data are written, the FPGA can read the data in the correct order by alternating reads between the two memories. Using this method, data is still being read on every clock cycle so that the processing doesn't slow down. In addition, both memory chips are only being accessed on every other clock cycle for a read, and new data can be written on the off cycles. Figure 3.11 shows a basic timing diagram of how the reads and writes alternate. Note that there are more reads than writes in this diagram. For the calculation of results for the first row of the frame, all of the pixels in the first 11 rows are read once. When the neighborhoods move down by one pixel to the next row, however, 10 out of the 11 rows of pixels must be re-read. Due to the nature of reading two-dimensional blocks of data, we must re-read a great deal of data, so there is not a one-to-one relationship between reads and writes.

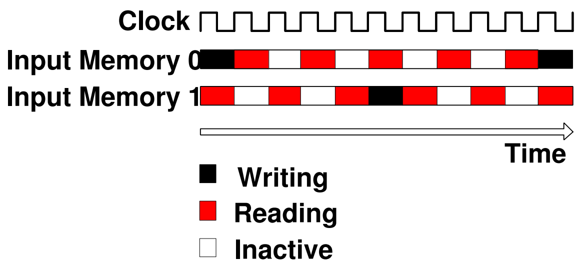


Figure 3.11: Memory Timing Diagram

3.3.6 Framegrabber Interface

The framegrabber takes the streaming data coming from the camera, buffers it, and creates some signals that are useful for the FPGA. The *LineSync* signal signifies the beginning of a new line of the image. The *FrameSync* has the same function for the beginning of a new frame. The datasheet provided by Dillon Engineering explains the functionality of these signals [17]. The FPGA sends a clock to the framegrabber, and data streams into the chip at a rate of one pixel per clock cycle. The interface between the FPGA and the framegrabber is the 228-pin I/O Mezzanine connector on the Firebird board.

Section 3.3.5 described that the FPGA must read data from the on-board memory more than once. There is not a one-to-one relationship between the number of pixels written into the input memory, and the number of pixels read out of the input memory. This means that the rate at which data is read from on-board memory into the FPGA needs to be faster than the rate at which data is input to the FPGA from the framegrabber. Different clock speeds are required to insure that the data processing can keep up with the rate of the incoming data. We send a slower clock to the framegrabber than we use for the rest of the design. The framegrabber sends data directly to the FPGA at the slower clock rate, and the data is stored in registers on the chip temporarily until it is written to the on-board memory. The data will later be read from this on-board memory at the faster clock rate to be input into the filter response section of the design. A memory chip can only run at one clock frequency, so since we must read data at the faster speed, we also have to write the data to the on-board memory at the faster speed. This means that there is a change in clock domains between the time that the data is introduced to the FPGA and the time that is written to memory.

The `Data Packing Design` runs on the slower clock. Every cycle, a new pixel is introduced and the data packing design stores five pixels into a 64-bit register. When the register is full of all five pixels, the `Data Packing Design` generates a READY signal that tells the memory interface that a word of data is ready to be written. The new data remains in the register until the memory interface, running at the faster clock rate, is ready to write the data into memory, as described in Section 3.3.5. Since new data is only changed once every five cycles of the slower clock, this gives the memory interface several clock cycles to write the data into memory before it is updated.

Processing live data is challenging, because if the data is processed too slowly, then incoming data will fill up the memory and start overwriting itself before results are calculated. On the other hand, if the design runs faster than the incoming data, then memory will be accessed before the new data is written to it, and invalid results will be calculated. If the design were running at the same rate as the incoming data, then this would not be a problem, but since we use two different clock speeds, this problem must be solved. To accomplish this, we run the design faster than the incoming data comes in. To avoid reading data that is not yet available, we don't start processing any data until 11 full lines are written to memory. While the 12th line is being written, the first 11 lines are being processed. Results for the first row of pixels will be finished before the 12th line is available, so the design goes into a wait state until the 12th line is written, signified by the *LineSync* signal. This guarantees that the design will always be processing valid data, even if the framegrabber clock is run slower or the memory clock is run faster in the future.

3.3.7 Host Interface

The results from the FPGA data processing are stored in the 32-bit wide off-chip memory on the Firebird board. Once the results are in the Firebird memory, they must be transferred to the host memory to be available to the software tracing algorithm. Any latency incurred when transferring the data from the hardware board to the host is included in the total latency of the hardware solution. We must make sure that the results are being transferred at least as fast as the frame rate of the camera, and that the delay between the time that results are stored in the Firebird memory and the time they are stored in the host memory is minimized.

The Firebird board supports Direct Memory Access (DMA) for data transfer. The advantage of using DMA is that once the host processor initiates the transfer, it is free to perform other duties while the data is being sent into the host memory. The output memory bank on the Firebird can have data read from it via DMA at the same time that new results are being written. This allows us to store results non-stop while we are transferring previous results to the host.

The interface between the Firebird and the host is the PCI bus. The bus on our machine is 64-bits wide with a clock of 66MHz. This means that two 32-bit words, equivalent to two results, can be sent to the host memory every 66 MHz clock cycle. The results are being written into the Firebird memory at a rate of five results every eleven cycles, at 65 MHz. Note that the DMA transfer rate is faster than the rate which results become available. We are able to send the results to the host in blocks the size of one frame without incurring great latency costs. We wait until several results are already stored in memory, and then start transferring the frame of results to the host memory. The starting point is timed so that the transfer ends shortly after the last result of the frame is stored in the Firebird memory, as shown in Figure 3.12.

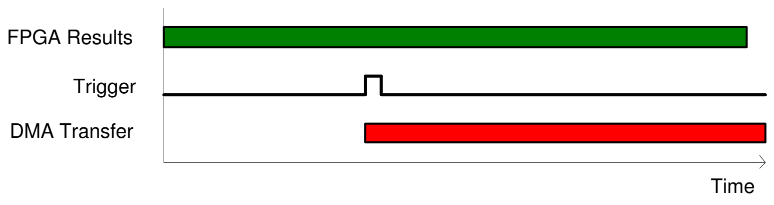


Figure 3.12: Timing of Writing and Transferring One Frame of Results

In order to coordinate the timing of the DMA transfer, some handshaking signals between the host and FPGA are required. First, the FPGA sets a trigger signal high when it is time for the transfer to begin, and stores that signal in a register. The host is continuously reading this register, waiting for the trigger signal to go high. Once the host receives the signal for the DMA transfer to begin, the host sends a signal acknowledging that the trigger was received back to the FPGA, and the trigger signal is set back to zero until the next frame is ready to be transferred. After sending the acknowledgement back to the FPGA, the host begins the DMA transfer, and then starts waiting for the next trigger signal. When the full frame of data is available in the DMA buffer, it can be accessed by the tracing algorithm.

3.4 Summary

This chapter described the hardware algorithm and how it is implemented on the FPGA. The interfaces for the memory, framegrabber, and host memory are described. The results and performance analysis of this implementation are presented in the next chapter.

Chapter 4

Results and Performance

This chapter presents the experimental results from the hardware processing described in the previous chapter. The performance of the hardware design is discussed, and a comparison with a software implementation is provided. It is shown that this implementation can be run in real-time with a minimum amount of latency, and that it far outperforms an equivalent software solution.

4.1 Results From Implementation

The hardware outputs three results for each pixel of every frame. First, the 12-bit grayscale value of the pixel is sent, unaltered, through the design. Second, the maximum response out of all 16 templates is output. The maximum response is a 17-bit value, since it comes from performing several additions and a binary shift on the 12-bit pixel data. The results from each addition and the shift require an additional bit, since the answers have a larger magnitude. The third output is the direction, which is the label of the template which returned the greatest response. The direction can be any value from 0 to 15, which requires a 4-bit result.

All three results can be packed into 33-bits. The output memory on the board can only store either 32 or 64-bit words, so in order to store the entire 33-bit result, we would need to use the 64-bit wide memory and waste 31 bits. Since transferring the data from the board to the host memory quickly is a large concern, we do not want to send a lot of wasted bits. To avoid this problem, we chose to use only 16-bits to represent the response result by removing the least significant bit. The actual value of the response can be recovered by the host, if necessary, by multiplying the 16-bit response by two, and introducing a possible error of one. By introducing this error, we are effectively doubling the rate at which valid results can be transferred to the host.

In order to visualize the results to verify that the outputs are correct, we create one grayscale image for each of the three outputs. Figure 4.1 shows the results from one frame of retinal image data. Note that these images are used only to visually verify the hardware output, and that they are not the output from the RVT algorithm. Results are valid only if the 11 x 11 pixel template window does not overlap rows or frames, therefore the results on all four edges of the frame are invalid and thrown away, leaving 507 x 507 pixels of valid results per frame. Currently, the host receives the full 512 x 515 pixel frames, including the three columns of zeros described at the end of Section 3.3.4. Currently, the host ignores the invalid data, but if it were desired, the hardware design could be modified to send only the valid results.

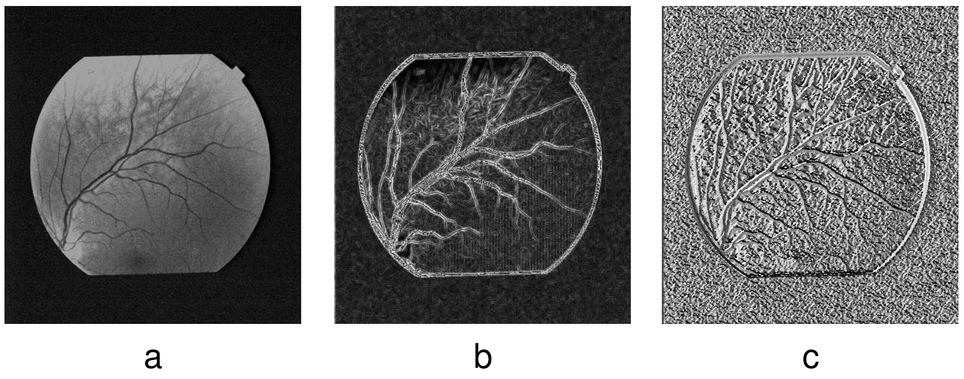


Figure 4.1: Visual Representation of the Three Hardware Outputs: a.) Original Pixel b.) Response c.) Direction

4.2 Speed and Latency

The goal of the design is for the processing to keep up with the frame rate of the camera. There are two clocks that are programmable. The framegrabber clock determines the rate at which data is sent from the framegrabber to the FPGA. The second clock is the memory clock, which is the rate at which the data is processed in the FPGA and stored to and read from the on-board memory. The camera has an internal pixel clock, which is running at 25MHz. The framegrabber buffers one line of the image, and sends it to the FPGA at the rate of the framegrabber clock. In order to prevent this buffer from overflowing, the framegrabber clock must be running at least as fast as the camera's pixel clock. The memory clock must be running fast enough so that the time it takes to process one line of pixels is less than the time it takes to transfer one line of data from the framegrabber to the FPGA. If the data is processed faster than it is input, the processing goes into a wait state, and lets the incoming data catch up. If the data is processed slower than it is input, then the processing could never catch up, because we cannot tell the camera to wait and stop sending data. For our design, the framegrabber clock is set to 25MHz, and the memory clock is set to 65MHz. These clocks could be run faster, however, since the camera's pixel clock is the limiting factor, the results would still be output at the same rate.

Minimizing the latency is a very important part of this project. The first source of latency is in the framegrabber. One line is buffered in the framegrabber, and this is an unavoidable source of delay. Next, the FPGA must store 11 rows of data before any processing can begin. The time required for data to be sent to the FPGA includes not only one clock cycle per pixel sent, but also several clock cycles in between lines and frames. The faster the framegrabber clock is set, the more time there is between lines and frames. The timing information for the Dalsa camera used in this design is provided in the datasheet [18]. The next source of latency is the time it takes to fill up the pipeline. Since data must be passed through every stage of the pipeline before any results are available, this adds delay to our design. Once results become available, they are written to the output memory. The time it takes to write these results must also be included in the overall latency. Table 4.1 shows how much delay each source of latency in the FPGA adds.

The total latency, from the time the framegrabber sends the first pixel to the

Operation	*Clock Cycles*	*Clock Speed*	*Latency*
Write 11 rows	7001	25 MHz	280 μsec
Fill Pipeline	20	65 MHz	308 nsec
Write Results	5	65 MHz	77 nsec

Table 4.1: Latency

time that the first filter response is available in the on-board memory, is approximately 280μsec. Note that the latency from filling the pipeline and writing results is negligible due to the length of time required for reading the initial 11 rows of data. Minimizing the amount of initial data to read is the biggest concern, which is why the complicated memory design described in Section 3.3.5 is required.

There is additional delay introduced when the data is sent from the hardware board to the host. The biggest source of this latency is from the time when the FPGA sets a flag telling the host it can begin the DMA transfer, to the time the host reads this value from the FPGA register and begins the data transfer. In order to measure this time, a counter was set in the FPGA to determine how long it took from the time the FPGA set the flag to start the read, to the time that the FPGA received the acknowledgement from the host that the DMA transfer had begun. This total time, divided by two, gives an idea of the latency added by the host-FPGA communication. The host was set to perform 1000 consecutive DMA transfers, and the value of this counter was recorded each time. Table 4.2 shows the time required for the round-trip handshaking, including the average delay, the maximum and minimum delay, and the standard deviation in number of clock cycles and in seconds.

It can be seen from these numbers that the amount of delay is highly variable

	Average	*Maximum*	*Minimum*	*Std.Dev.*
Clock Cycles	1008	2041	8	591
Time	15.5 μsec	31.4 μsec	123 nsec	9.1 μsec

Table 4.2: FPGA-Host Communication Delay

from one DMA transfer to the next. It is impossible to provide a reliable latency number from this data. One reason that these numbers fluctuate so much is because this experiment was run on a PC running Windows 2000. The microprocessor is running several operations aside from the code which communicates with the FPGA board. The latency is a function of the host processor and operating system, and not of the hardware board. In order to minimize the host to FPGA board communication delay and get more reliable latency numbers, the software portion of the RVT implementation should be run on an operating system that is designed for real-time operations, and that can guarantee a response in a given time frame. Moll and Shand investigated the delays involved with DMA over the PCI bus, and reported a high variance in latencies for hosts with different processors, and also for consecutive data transfers on the same processor [19].

4.3 Performance Speedup

Calculating the speedup from the software solution to the hardware solution is not a straight-forward task. The software algorithm includes several shortcuts to minimize the number of filter calculations necessary. The best way to calculate speedup is to integrate the hardware filter calculations into the software tracing algorithm; however, this is beyond the scope of this thesis. Ideally, we want our hardware to provide real-time filter responses while adding zero latency. When considering that it takes 33msec for the camera to capture a single frame of data, however, 280μsec is a small price to pay for the amount of computation occurring.

We took the filter response algorithm that was implemented in hardware, and wrote it in C to compare the amount of time required for the hardware and software to perform the same operation. Random image data was written to the host memory, and we found the time that it took to read an 11 x 11 window, run the eight unique filters over the window, and decide which filter returned the greatest response. The program was run 50 million times on an Intel Xeon 2.6GHz PC, and we found that it took, on average, 1.96μsec to find the result for one window. When the pipeline is full, the FPGA can return five results every 11 clock cycles. Running at 65MHz, one result, on average, is returned every 33.85nsec. The hardware has a speedup factor greater than 50 compared to the software implementation of filter response calculations.

4.4 Summary

This chapter provided the results from the hardware implementation of the filter response calculations. The performance was analyzed, and a comparison was made to a software implementation performing the same calculations. The next chapter will offer some conclusions and suggestions for future work.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Retinal vascular tracing must be run in real-time and with very low latency in order to be useful in the medical applications for which it is intended. A software solution has been designed, however the underlying architecture for software processing is not suitable for the computationally intensive image filtering operations required by the algorithm. We have moved the filter response calculations to hardware in order to speed up the tracing algorithm. An FPGA implementation was chosen to enable a real-time solution.

The filter response calculations were parallelized and optimized for our custom hardware solution. The memory interface was set up to keep data flowing through the design uninterrupted. In order to be able to continue storing new data in the on-board memory at the same time we are reading data into the FPGA, a method for swapping between two input memories was designed. This avoids the delay of waiting for an entire frame to be available before beginning our processing. After the data is processed, there are three outputs: the original unaltered pixel data, the maximum response from the 16 filters, and the filter which corresponds to the maximum response. These results are stored in on-board memory and passed to the host using DMA over the PCI bus. Using DMA minimizes additional delays that can occur when the host is receiving data.

We were able to provide the filter response results in real-time by running the framegrabber clock at 25MHz and the memory/processing clock at 65MHz. The hardware processing adds approximately 280μsec of latency to the overall tracing algorithm, not including the delay from transferring results to the host, which is introduced by the host. Implementing the same filter calculations on a 2.6GHz PC takes over 50 times longer than the hardware implementation.

5.2 Future Work

In order for the RVT algorithm to be run in real-time, the hardware solution must be integrated with the existing software solution. The original software algorithm was written to calculate the filter responses on the fly, as needed, and several optimizations were put in place to minimize the number of filter responses that need to be computed. This algorithm must be altered to take advantage of the results provided by the FPGA. Once the hardware/software co-design is completed, then the performance will be reanalyzed to determine if the hardware implementation improves the overall performance of the entire RVT algorithm.

Other future work relates to upgrading the camera. A camera with more bits per pixel will improve the quality of the image. This would require drastic changes in the hardware. Although the algorithm would be the same, all of the components and signals in the FPGA have a custom bitwidth. Increasing the number of bits per pixel on the incoming data would require major changes in the way that the data is stored in memory, which creates a ripple of changes throughout the rest of the design.

It is desirable to have a camera that can output only smaller windows of an image, and output these smaller frames at rates significantly faster than the current frame rate. This change would require minor changes to the hardware design, depending on how the framegrabber dealt with the windowed frames. An increased frame rate may require running the clocks on the FPGA at faster rates, which would mean further optimizing the HDL code. Note that as camera technology progresses, so does FPGA technology. Larger chips translate to more area for routing, and faster designs.

This design implements filter response calculations. The framework from this design could be reused for another image processing algorithm that included a two-dimensional windowing operation. The memory interfaces could be reused, and our filter response design could effectively be unplugged, and replaced with another image processing design.

All of the future work described in this chapter would require modifications to the hardware design. The advantage of using reconfigurable hardware, such as the

FPGA in our design, is that the system can be upgraded by changing the HDL code and reloading a new configuration onto the FPGA. There is no need for the costs of having a new chip manufactured like there would be for an ASIC design.

Bibliography

[1] Ali Can, Hong Shen, James N. Turner, Howard L. Tanenbaum, and Badrinath Roysam, "Rapid automated tracing and feature extraction from retinal fundus images using direct exploratory algorithms," *IEEE Transactions on Information Technology in Biomedicine*, vol. 3, no. 1, March 1999.

[2] Xilinx, Inc., *Virtex-E 1.8V Field Programmable Gate Arrays*, November 2002, http://direct.xilinx.com/bvdocs/publications/ds022-2.pdf.

[3] Annapolis Microsystems, Inc., *Firebird Hardware Reference Manual*, 2000, Revision 3.0.

[4] Katherine Compton and Scott Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, June 2002.

[5] Stephen M. Trimberger, Ed., *Field-Programmable Gate Array Technology*, Kluwer Academic Publishers, 1994.

[6] Douglas J. Smith, *HDL Chip Design*, Doone Publications, 1996.

[7] Rajeev Murgai, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli, *Logic Synthesis for Field-Programmable Gate Arrays*, Kluwer Academic Publishers, 1995.

[8] Harold S. Stone, *High-Performance Computer Architecture*, Addison-Wesley Publishing Company, 1987.

[9] Subhasis Chaudhuri, Shankar Chatterjee, Norman Katz, Mark Nelson, and Michael Goldbaum, "Detection of blood vessels in retinal images using two-dimensional matched filters," *IEEE Transactions on Medical Imaging*, vol. 8, no. 3, September 1989.

[10] Alvise Sartori, "A smart camera," in *FPGAs*, Wayne Luk Will Moore, Ed., chapter 6.6, pp. 353–362. Abingdon EE and CS Books, Abingdon, England, 1991.

[11] Stephen Scalera, Mark Falco, and Brent Nelson, "A reconfigurable computing architecture for microsensors," in *FCCM '00 Preliminary Proceedings*, Napa, CA, April 2000, Field-Programmable Custom Computing Machines.

[12] G. Lienhart, R. Manner, R. Lay, and K. H. Noffz, "An FPGA-based video compressor for H.263 compatible bit streams," in *FPGA '01: ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2001.

[13] Elena Cerro-Prada and Philip B. James-Roxby, "High speed low level image processing on FPGAs using distributed arithmetic," in *Field-Programmable Logic: From FPGAs to Computing Paradigm*, Reiner W. Hartenstein and Andres Keevallik, Eds. / 1998, pp. 436–440, Springer-Verlag, Berlin.

[14] Nahed H. Solouma, Abou-Bakr M. Youssef, Yehia A. Badr, and Yasser M. Kadah, "A new real-time retinal tracking system for image-guided laser treatment," *IEEE Transactions on Biomedical Engineering*, vol. 49, no. 9, September 2002.

[15] Jack Jean, Xinzhong Guo, Fei Wang, Lei Song, and Ying Zhang, "A study of mapping generalized sliding window operations on reconfigurable computers," in *ERSA '03: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, NV, June 2003.

[16] Xuejun Liang and Jack Jean, "Mapping of generalized template matching onto reconfigurable computers," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 11, no. 3, June 2003.

[17] Dillon Engineering, Inc., *Camera Interface*, 2002.

[18] Dalsa, *Dalstar 1M30P User's Manual and Reference*, 2001.

[19] Laurent Moll and Mark Shand, "Systems performance measurement on pci pamette," in *FCCM '97: 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, Napa Valley, CA, April 1997.