IMPLEMENTATION OF IMAGE PROCESSING ALGORITHMS ON FPGA HARDWARE

By

Anthony Edward Nelson

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

May 2000

Nashville, TN

Approved:	Date:

ACKNOWLEDGEMENTS

I would like to thank the staff at ISIS for their invaluable input in regards to my education. My research position at ISIS has given me education as well as experience. In particular I would like to thank Dr. Ted Bapty, my research advisor, and Dr. Gabor Karsai, my academic advisor. Their advice has been most valuable in the past two semesters. I would also like to thank others involved in ISIS, including Jason Scott and Jim Nichols.

Dr. Bapty was responsible for bringing me to Vanderbilt University from my previous research position at LANL. I would like to thank John Szymanski, my mentor at LANL, as well as the rest of the DAPS project, Jeff Bloch, Phil Blain, Michael Caffrey, and Kevin McCabe.

None of this would have been possible without the constant support and advice of my family. Their insight throughout my college career has been invaluable. Lastly, I would like to thank my fiancée Johanna for her love and guidance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Recently, Field Programmable Gate Array (FPGA) technology has become a viable target for the implementation of algorithms suited to video image processing applications. The unique architecture of the FPGA has allowed the technology to be used in many such applications encompassing all aspects of video image processing [1,2]. The goal of this thesis is to develop FPGA realizations of three such algorithms on two FPGA architectures.

As image sizes and bit depths grow larger, software has become less useful in the video processing realm. Real-time systems such as those that are the target of this project are required for the high speeds needed in processing video. In addition, a common problem is dealing with the large amount of data captured using satellites and ground-based detection systems. DSP systems are being employed to selectively reduce the amount of data to process, ensuring that only relevant data is passed on to a human analyst. Eventually, it is expected that most video processing can and will take place in DSP systems, with little human interaction. This is obviously advantageous, since human data analysts are expensive and perhaps not entirely accurate.

Platforms Used for DSP Design

There are several different choices a designer has when implementing a DSP system of any sort. Hardware, of course, offers much greater speed than a software implementation, but one must consider the increase in development time inherent in creating a hardware design. Most software designers are familiar with C, but in order to develop a hardware system, one must either learn a hardware design language such as VHDL or Verilog, or use a software-to-hardware conversion scheme, such as Streams-C [3], which converts C code to VHDL, or MATCH [4], which converts MATLAB code to VHDL. While the goals of such conversion schemes are admirable, they are currently in development and surely not suited to high-speed applications such as video processing. Ptolemy [5] is a system that allows modeling, design, and simulation of embedded systems. Ptolemy provides software synthesis from models. While this type of system may be a dominant design platform in the future, it is still under much development, meaning that it may not be a viable design choice for some time. A discussion on the various viable options for DSP system design is found below.

PC Digital Signal Processing Programs

Signal processing programs used on a PC allow for rapid development of algorithms, as well as equally rapid debug and test capabilities. It is common for many hardware designers to use some sort of PC programming environment to implement a design to verify functionality prior to a lengthy hardware design.

MATLAB [6] is such an environment. Although it was created for manipulating matrices in general, it is well suited to some image processing applications. MATLAB treats an image as a matrix, allowing a designer to develop optimized matrix operations implementing an algorithm. However, if the eventual goal is a hardware device, the algorithms are instead often written to operate similarly to the proposed hardware system, which results in an even slower algorithm.

Systems such as IDL [7] and its graphical component ENVI [8] are more specifically geared to image processing applications, and include many pre-written algorithms commonly used to process images. However, even specialized image processing programs running on PCs cannot adequately process large amounts of high-resolution streaming data, since PC processors are made to be for general use. Further optimization must take place on a hardware device.

Application Specific Integrated Circuits

Application Specific Integrated Circuits (ASICs) represent a technology in which engineers create a fixed hardware design using a variety of tools. Once a design has been programmed onto an ASIC, it cannot be changed. Since these chips represent true, custom hardware, highly optimized, parallel algorithms are possible. However, except in high-volume commercial applications, ASICs are often considered too costly for many designs. In addition, if an error exists in the hardware design and is not discovered before product shipment, it cannot be corrected without a very costly product recall.

Dedicated Digital Signal Processors

Digital Signal Processors (DSPs) such as those available from Texas Instruments [9] are a class of hardware devices that fall somewhere between an ASIC and a PC in terms of performance and design complexity. They can be programmed with either assembly code or the C programming language, which is one of the platform's distinct advantages. Hardware design knowledge is still required, but the learning curve is significantly lower than some other design choices, since many engineers have knowledge of C prior to exposure to DSP systems. However, algorithms designed for a DSP cannot be highly parallel without using multiple DSPs. Algorithm performance is certainly higher than on a PC, but in some cases, ASIC or FPGA systems are the only choice for a design. Still, DSPs are a very common and efficient method of processing real-time data [10].

One area where DSPs are particularly useful is the design of floating point systems. On ASICs and FPGAs, floating-point operations are rather difficult to implement. For the scope of this project, this is not an issue because all images consist of only integer data.

Recent advances in DSP technology have resulted in very high-speed algorithm implementations [11]. While the advantages of ASICs and FPGAs are still applicable, this new generation of DSPs has made some engineers reconsider FPGA development. Still, as new DSPs arrive to the market, so do new FPGAs, and it is expected that the two architectures will have similarly increasing performance for each new generation of processors.

Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) represent reconfigurable computing technology [12], which is in some ways ideally suited for video processing. Reconfigurable computers are processors which can be programmed with a design, and then reprogrammed (or reconfigured) with virtually limitless designs as the designer's needs change. FPGAs generally consist of a system of logic blocks (usually look up tables and flip-flops) and some amount of Random Access Memory (RAM), all wired together using a vast array of interconnects. All of the logic in an FPGA can be rewired, or reconfigured, with a different design as often as the designer likes. This type of architecture allows a large variety of logic designs dependent on the processor's resources), which can be interchanged for a new design as soon as the device can be reprogrammed.

Today, FPGAs can be developed to implement parallel design methodology, which is not possible in dedicated DSP designs. ASIC design methods can be used for FPGA design, allowing the designer to implement designs at gate level. However, usually engineers use a hardware language such as VHDL or Verilog, which allows for a design methodology similar to software design. This software view of hardware design allows for a lower overall support cost and design abstraction.

The algorithms presented in this thesis were written for two FPGA architectures. The advantages of these devices have proven themselves for this type of design. In addition, the author has previous experience with FPGA development. The goal of this thesis is for real-time (30 frames per second) processing of grayscale image data, a goal in which an FPGA system using parallel algorithms should have little difficultly achieving.

FPGA Design Options

In order to create an FPGA design, a designer has several options for algorithm implementation. While gate-level design can result in optimized designs, the learning curve is considered prohibitory for most engineers, and the knowledge is not portable across FPGA architectures. The following text discusses several high-level hardware design languages (HDLs) in which FPGA algorithms may be designed.

Verilog HDL

Originally intended as a simulation language, Verilog HDL represents a formerly proprietary hardware design language. Currently Verilog can be used for synthesis of hardware designs and is supported in a wide variety of software tools. It is similar to the other HDLs, but its adoption rate is decreasing in favor of the more open standard of VHDL. Still, many designers favor Verilog over VHDL for hardware design, and some design departments use only Verilog. Therefore, as a hardware designer, it is important to at least be aware of Verilog.

Altera Hardware Design Language

Altera Hardware Design Language (AHDL) is proprietary, and is only supported in Altera-specific development tools. This may be seen as a drawback, but since AHDL is proprietary, its use can also result in more efficient hardware design, when code portability is not an issue. In typical design environments, different FPGA architectures are used for different designs, meaning that time spent learning AHDL may be wasted if a Xilinx FPGA is later chosen.

VHSIC Hardware Design Language

In recent years, VHSIC (Very High Speed Integrated Circuit) Hardware Design Language (VHDL) has become a sort of industry standard for high-level hardware design. Since it is an open IEEE standard, it is supported by a large variety of design tools and is quite interchangeable (when used generically) between different vendors' tools. It also supports inclusion of technology-specific modules for most efficient synthesis to FPGAs.

The first version of VHDL, IEEE 1076-87, appeared in 1987 and has since undergone an update in 1993, appropriately titled IEEE 1076-93. It is a high-level language similar to the computer programming language Ada, which is intended to support the design, verification, synthesis and testing of hardware designs.

Design Approach

Prior to any hardware design, the author chose to create software versions of the algorithms in MATLAB. Using MATLAB procedural routines to operate on images represented as matrix data, these software algorithms were designed to resemble the hardware algorithms as closely as possible. While a hardware system and a matrix-manipulating software program are fundamentally different, they can produce identical results, provided that care is taken in development. This approach was taken because it speeds understanding of the algorithm design. In addition, this approach facilitates comparison of the software and synthesized hardware algorithm outputs, allowing detailed error calculations.

This project was targeted for FPGA systems for two reasons. One, the author had some previous experience in FPGA implementations of video processing algorithms [13, 14]. Two, FPGAs represent a new direction for DSP systems, and there is much original work to be done in terms of optimized algorithms for this type of system.

One of the initial goals of this project was to implement designs for two different FPGA systems: the Altera FLEX 10K100 [15] and the Xilinx Virtex 300 [16]. The rationale behind this decision was that the Altera chip represents an older generation of FPGA technology, but it is also very commonly used. The Altera chips have been used often in many design environments, and are well understood. The Xilinx Virtex is a new technology, which has a larger gate count and higher possible clock speed than the Altera chip. On the other hand, the Xilinx chip is not as well understood and supported, since it was only recently introduced to the market. For example, more parameterized modules for high-speed mathematical operations are available for the Altera FLEX series than are available for the Xilinx Virtex series. This can certainly affect a design's success, so if specialized functions are needed, the designer must first determine whether or not they are available for the chosen device.

VHDL was chosen as a target design language because of familiarity and its wide-ranging support, both in terms of software development tools and vendor support. Today, more engineers are learning VHDL than Verilog, which is another compelling reason for its use in this project.

The design flow for this project is represented in Figure 1. This shows the interaction between the VHDL design environment and the FPGA-specific tools. In the first state, a design is created in VHDL. Next, the code's syntax is verified and the design is synthesized, or compiled, into a library. The design is next simulated to check its functionality. Stimulating the signals in the design and viewing the output waveforms in the VHDL simulator allows the designer to determine proper functionality of the design. Next, the design is processed with vendor-specific place-and-route tools and mapped onto a specific FPGA in software. This allows the engineer to view a floorplan and hierarchical view of the design, which can help verifying a proper mapping procedure. Next, the design is verified for proper functionality once again. This step is important because it assures that the design is correct in its translation from VHDL to gate-level. If this is found to be correct, the design can then be programmed onto the specified FPGA.

For this project, the author had access to two FPGAs, each from a different company and each with different design tools: the Altera FLEX 10K100 and the Xilinx Virtex XCV300.

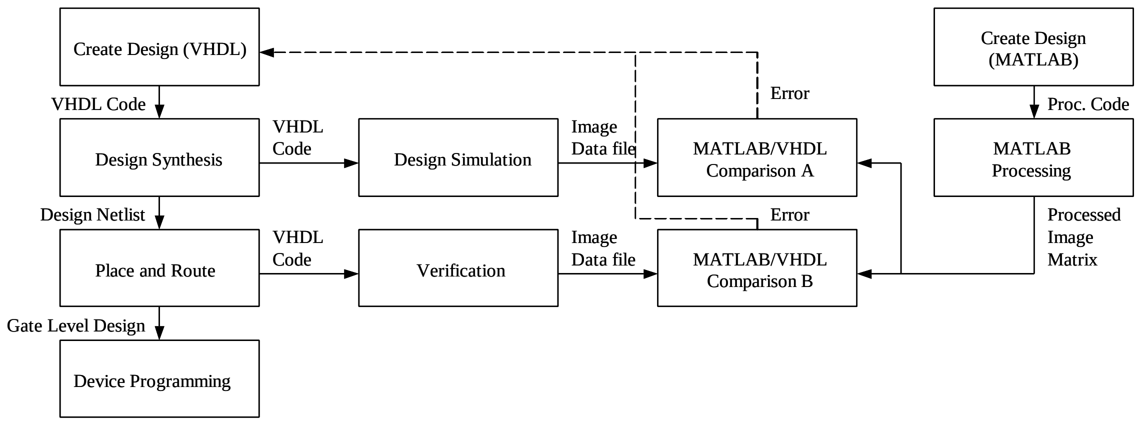


Figure 1: Hardware Design Flow

Altera FLEX 10K100

Due to architecture differences, the Altera FLEX 10K series is termed a Programmable Logic Device (PLD) and is not officially considered to be an FPGA. However for the purpose of simplicity it is commonly referred to as an FPGA, and will be so named in this document.

The FLEX 10K100 is a CMOS SRAM-based device, consisting of an embedded array for memory and certain logic functions and a logic array for general logic implementation. The embedded array is constructed of Embedded Array Blocks (EABs). The EABs can be used to implement limited memories such as First In First Out (FIFO) or RAM units. The FLEX 10K100 has 12 EABs, each with 2048 bits for use in a design.

The logic array in the FLEX 10K series is built from Logic Array Blocks (LABs). Each LAB consists of 8 Logic Elements (LEs), each of which is constructed of a 4-input Look Up Table (LUT) and a flip-flop. Each LAB can be considered to represent 96 logic gates. The FLEX 10K100 has 624 LABs, accounting for most of its 100,000 gates (the rest are accounted for in memory). Figure 2 shows the basic units in a FLEX 10K LE.

Input/Output functionality on the FLEX 10K series is handled in the Input/Output Blocks (IOBs). Each IOB has one flip-flop to register either input or output data. However for bi-directional signals, this is an inefficient design, since two flip-flops are needed and only one is available in the IOB. The second flip-flop must be implemented in the logic array, resulting in an overall slower design [15,17]. Figure 3 shows a floorplan view of the Altera FLEX 10K architecture, highlighting the elements discussed.

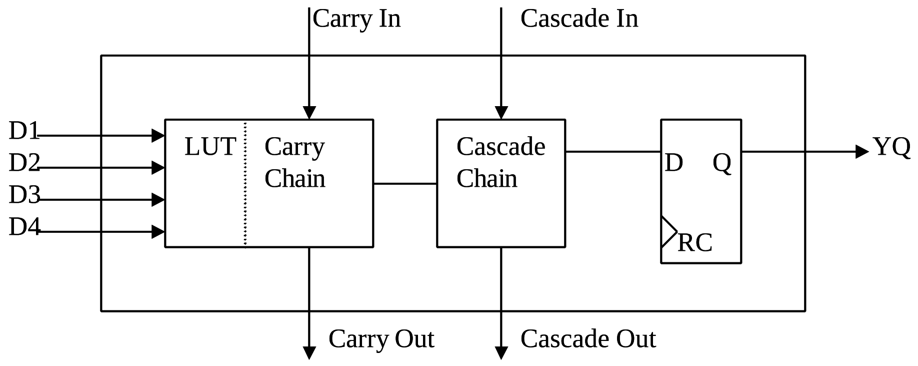


Figure 2: Altera FLEX 10K LE

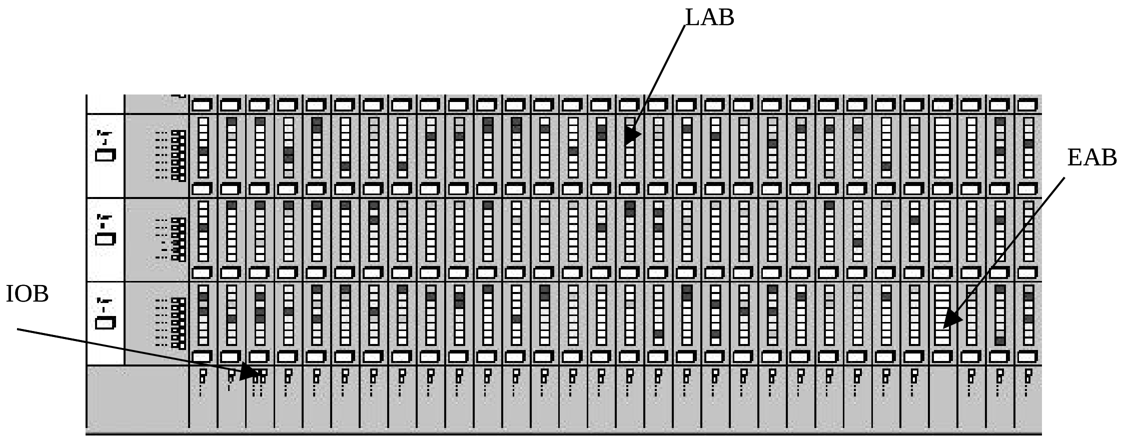


Figure 3: Altera FLEX 10K Floorplan Showing Elements Discussed

Xilinx Virtex XCV300

The Virtex is the most recent family of FPGAs from Xilinx. The previous generation, the Xilinx 4K series, was one of the most commonly used FPGA families, and can be conside red comparable to the Altera 10K series in many ways. The Virtex takes many of the features from the 4K series and combines them with several new features.

Technically, the Xilinx FPGAs are SRAM devices. This means that the chips must be configured after device power up. Configurable Logic Blocks (CLBs) are the primary logic elements in the Virtex FPGA. Each CLB is comprised of two slices, each of which contains two Look Up Tables (LUTs) and two D flip-flops. Each LUT can be used as one 32x1- or one 16x2-bit synchronous RAM. The Virtex XCV300 has a 32x48 array of CLBs, resulting in a total of 6912 logic cells and 322,970 gates. Figure 4, below, shows one slice of a Xilinx Virtex CLB.

The Virtex series has a system of Block RAM, which allows the use of the chip for limited RAM operations such as FIFO implementations or basic RAM usage. The XCV300 has 65,536 bits of Block RAM. Connecting the CLBs is a vast web of interconnects.

Input and output capabilities are handled by Input/Output Blocks (IOBs). The Virtex XCV300 has 316 IOBs. Figure 5 shows a typical Virtex floorplan and the elements common to all Virtex parts [16,17].

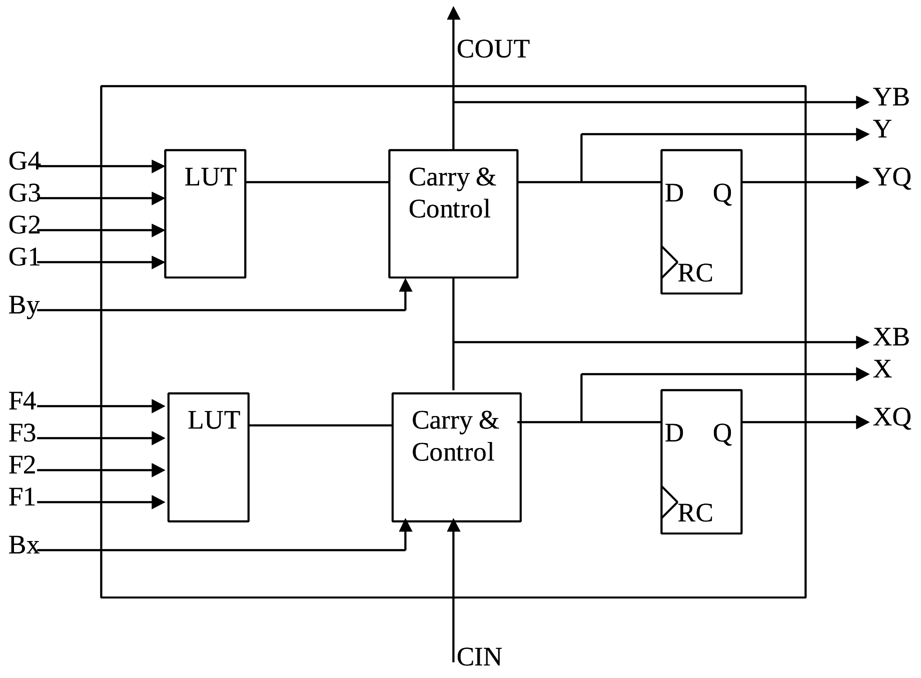


Figure 4: Slice of a Xilinx Virtex CLB

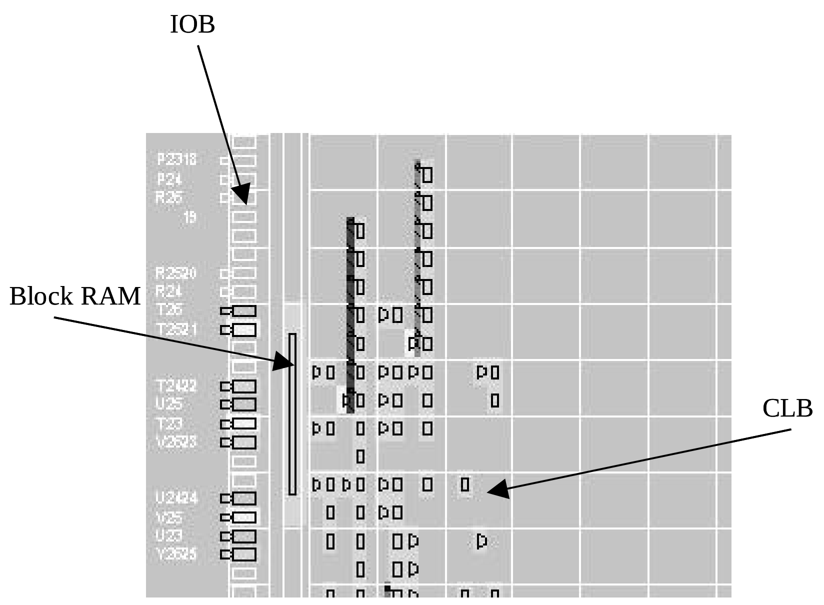


Figure 5: Xilinx Virtex Floorplan Showing Elements Discussed

Performance Comparison

The Xilinx Virtex FPGA is of a newer generation than the Altera FLEX 10K; therefore we expect higher performance. However, since the two architectures have different technology, some designs may perform better on one chip than the other, and vice versa. While the detailed analysis of this is not the focus of this paper, later chapters showing the project results compare the two FPGAs' performance for the same algorithms. This will yield a better understanding of the advantages of each FPGA, which will be quite useful in later projects using the same devices.

CHAPTER II

PROJECT ALGORITHMS

This project was focused on developing hardware implementations of three popular image processing algorithms for use in an FPGA -based video processing system. This chapter discusses these algorithms and their software implementations in MATLAB.

Introduction to Windowing Operators

In image processing, several algorithms belong to a category called windowing operators. Windowing operators use a window, or neighb orhood of pixels, to calculate their output. For example, windowing operator may perform an operation like finding the average of all pixels in the neighborhood of a pixel. The pixel around which the window is found is called the *origin*. Figure 6, below, shows a 3 by 3 pixel window and the corresponding origin.

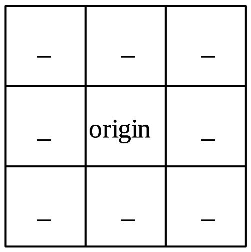


Figure 6: Pixel Window and Origin

The work for this project is based on the usage of image processing algorithms using these pixel windows to calculate their output. Although a pixel window may be of any size and shape, a square 3x3 size was chosen for this application because it is large enough to work properly and small enough to implement efficiently on hardware.

Rank Order Filter

The rank order filter is a particularly common algori thm in image processing systems. It is a nonlinear filter, so while it is easy to develop, it is difficult to understand its properties. It offers several useful effects, such as smoothing and noise removal. The median filter, which is a rank order filt er, is especially useful in noise removal [18].

Algorithm

This filter works by analyzing a neighborhood of pixels around an origin pixel, for every valid pixel in an image. Often, a 3x3 area, or window, of pixels is used to calculate its output. For every pixel in an image, the window of neighboring pixels is found. Then the pixel values are sorted in ascending, or rank, order. Next, the pixel in the output image corresponding to the origin pixel in the input image is replaced with the value specifi ed by the filter order. The rank order filter can be represented by the following lines of pseudo-code:

```
order = 5 (this can be any number from 1 -> # pixels in the window)
for loop x –> number of rows
  for loop y –> number of columns
      window_vector = vector consisting of current window pixels
      sorted_list = sort(window_vector)
      output_image(x,y) = sorted_list(order)
  end
end.
```

Figure 7 shows an example of this algorithm for a median filter (order 5), a filter that is quite useful in salt-and-pepper noise filtering [19]. Since the rank order filter uses no arithmetic, a mathematical description is difficult to represent efficiently.

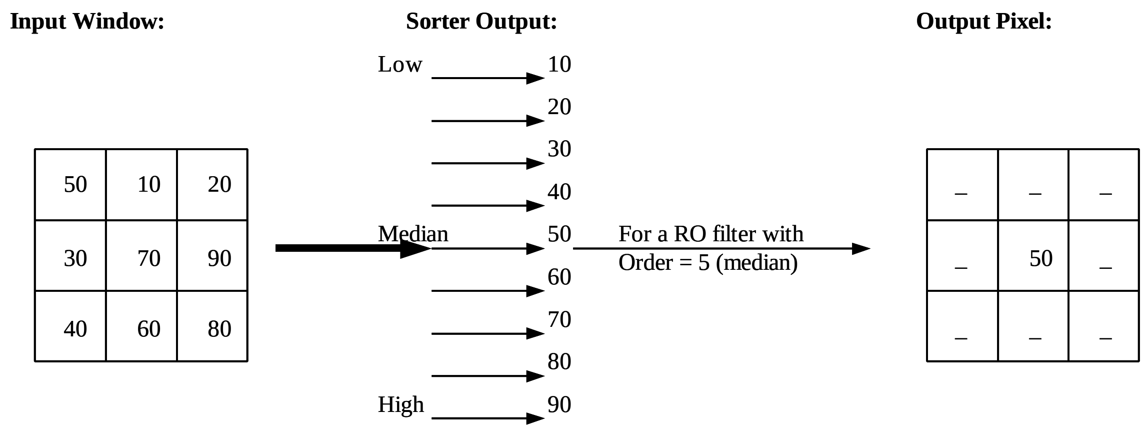


Figure 7: Graphic Depiction of Rank Order Filter Operation

As is evident in the above figure, it is possible to use any or der up to the number of pixels in the window. Therefore a rank order filter using a 3x3 window has 9 possible orders and a rank order filter using a 5x5 window has 25 possible orders. No matter what the window size used in a particular rank order filter, using the middle value in the sorted list will always result in a median filter. Similarly, using the maximum and minimum values in the sorted list always results in the flat dilation and erosion of the image, respectively. These two operations are considered part of the morphological operations, and are discussed in the next sub -chapter.

MATLAB Implementation

The PC software program MATLAB was used to develop an initial version of the rank order filter, so that its operation could be verified and its results could be compared to the hardware version. While MATLAB offers features that speed up operations on matrices like images, custom operations were used so that the software would closely mimic the functionality of the proposed hardware implementatio n.

The MATLAB implementation of the rank order filter is called ro_filt.m and is found in Appendix A. It works by using *for* loops to simulate a moving window of pixel neighborhoods. For every movement of the window, the algorithm creates a list of the p ixel values in ascending order. From this list, the algorithm picks a specific pixel. The pixel that is chosen from the list is specified in the order input. The output of the program is an image consisting of the output pixels of the algorithm. Since a full 3x3 neighborhood is used in this implementation, the window must have gotten to the second line of the input image in order to create an output. The result of this is that some ‘edge effects’ occur in the output image, meaning that there is always an invalid strip along the borders of the output image. This is true for all algorithms using the windowing approach to image processing. Figure 8 shows some example output images for a given input image using ro_filt.m. From this figure it is easy to o bserve the effect that the rank order filter has on an image, given the various algorithm orders used.

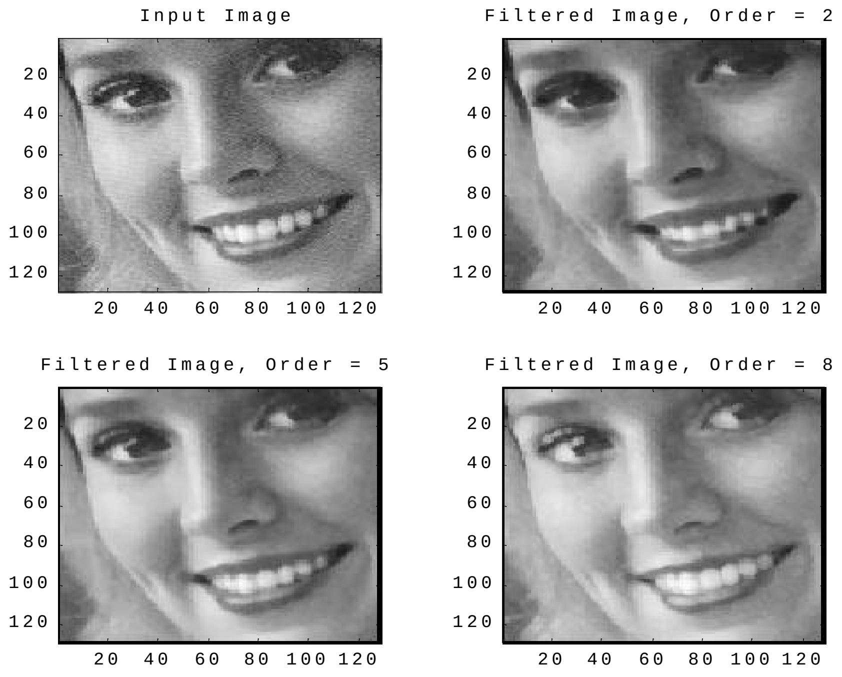

Input Image
Filtered Image, Order = 2
Filtered Image, Order = 5
Filtered Image, Order = 8
20
40
60
80
100
120

Figure 8: Example Images Obtained Using ro_filt.m

Morphological Operators

The term morphological image processing refers to a class of algorithm that is interested in the geometric structure of an image. Morphology can be used on binary and grayscale images, and is useful in many areas of image processing, such as skeletonization, edge detection, restoration, and texture analysis.

A morphological operator uses a structuring element to process an image. We usually think of a structuring element as a window passing over an image, which is similar to the pixel window used in the rank order filter. Similarly, the structuring element can be of any size, bu t 3x3 and 5x5 sizes are common. When the structuring element passes over an element in the image, either the structuring element fits or does not fit. At the places where the structuring element fits, we achieve a resultant image that represents the structure of the image [20]. Figure 9 demonstrates the concept of a structuring element fitting and not fitting inside an image object.

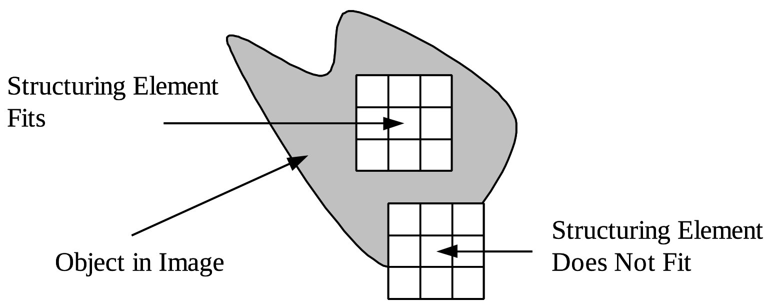


Figure 9: Concept of Structuring Element Fitting and Not Fitting

Algorithm

There are two fundamental operations in mor phology: erosion and dilation [20]. It is common to think of erosion as shrinking (eroding) an object in an image. Dilation does the opposite; it grows the image object. Both of these concepts depend on the structuring element and how it fits within the object. For example, if a binary image is eroded, the resultant image is one where there is a foreground pixel for every origin pixel where its surrounding structuring element -sized fit within the object. The output of a dilation operation is a foregrou nd pixel for every point in the structuring element at a point where the origin fits within an image object [20]. Figure 10 shows a simple binary image and its erosion and dilation, using a 3x3 sized structuring element consisting of all ones. From [20], Erosion and dilation can be represented mathematically by the following formulas:

Erosion: $A\ q\ B = \{x: B + x < A\}$ and

Dilation: $A\ Å\ B = È\{A + b: b\ Î\ B\}$,

where A is the input image and B is the structuring element.

Grayscale morphology is more powerful and more difficult to understand. The concepts are the same, but instead of the structuring element fitting inside a two -dimensional object, it is thought to either fit or not fit within a three -dimensional object. Grayscale morphology also allows the us e of grayscale structuring elements. Binary structuring elements are termed *flat* structuring elements in grayscale morphology. The combination of grayscale images and grayscale structuring elements can be quite powerful [20].

One of the strongest features of morphological image processing extends from the fact that the basic operators, performed in different orders, can yield many different, useful results. For example, if the output of an erosion operation is dilated, the resulting operation is called an opening. The dual of opening, called closing, is a dilation followed by an erosion. These two secondary morphological operations can be useful in image restoration, and their iterative use can yield further interesting results, such as skeletonization and granulometries of an input image. Figure 11 shows an example of binary opening and closing on the same input image as was used in the erosion/dilation example, again using a structuring element of size 3x3 consisting of all ones.

Grayscale erosion and dilation can be achieved by using a rank order filter as well. Erosion corresponds to a rank order filter of minimum order, and dilation corresponds to a rank order filter of maximum order. The reason for this is that the result of a minimum order ran k order filter is the minimum value in the pixel neighborhood, which is exactly what an erosion operation is doing. This also holds true for a maximum order rank order filter and a dilation operation. However, the rank order filter only works as a morphological operation with a flat structuring element. This is because the rank order filter window works as a sort of structuring element consisting of all ones. Still, this is a powerful feature, since grayscale morphology using flat structuring elements a ccounts for the most common usage of morphology.

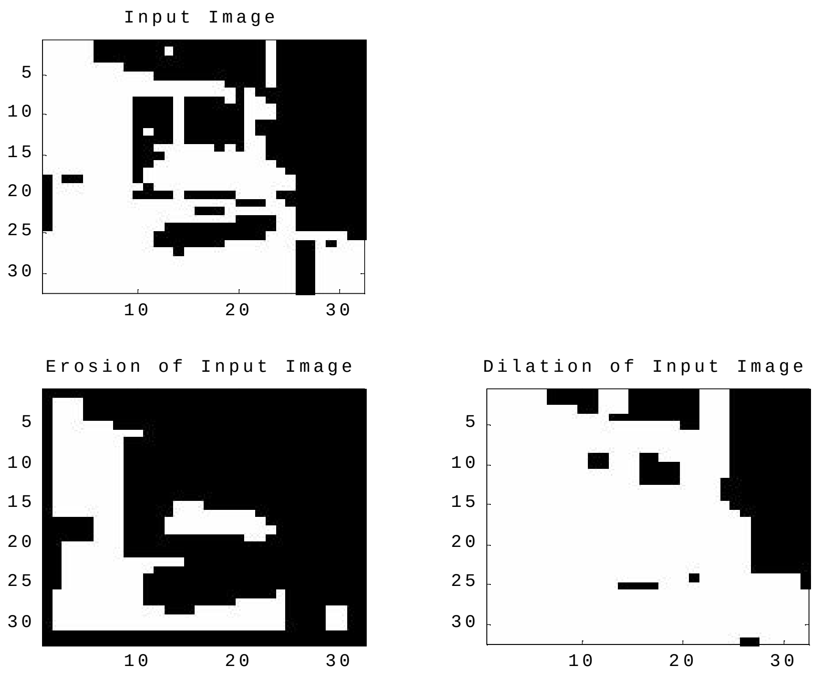


Figure 10: Binary Erosion and Dilation on a Simple Binary Image

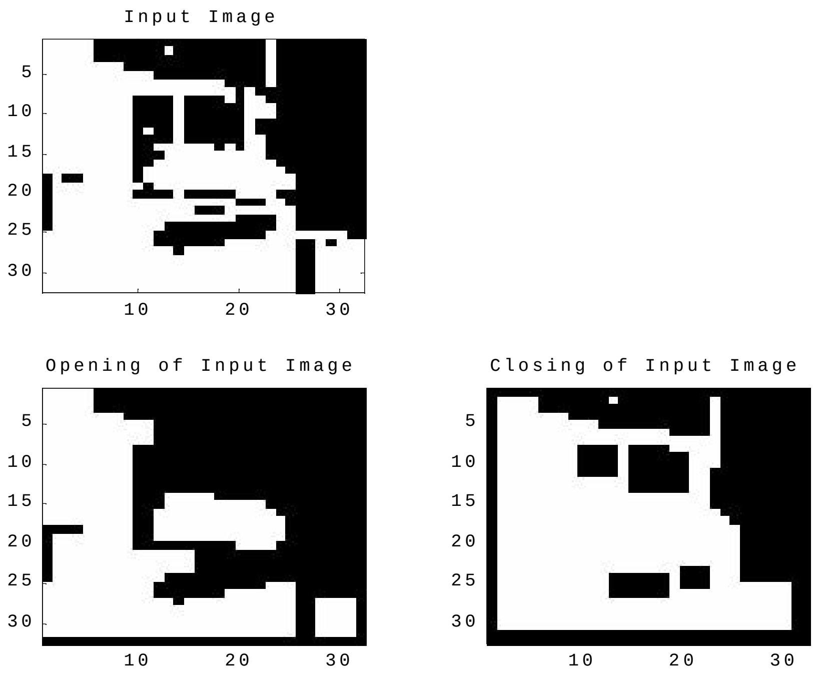


Figure 11: Binary Opening and Closing on a Simple Binary Image

MATLAB Implementation

In order to garner a full understanding of the morph ological operation, the algorithms were written using MATLAB prior to any hardware development. Initially, only binary versions of the algorithms were written, because it was easier to understand the effect of morphology in the binary output images than i n grayscale images. However, since a grayscale implementation on hardware was desired, the algorithms had to be re-written to facilitate grayscale morphology. The MATLAB implementations of erosion and dilation are called aip_erode_gs.m and aip_dilate_gs. m, respectively. The source code for these algorithms is shown in Appendix A. Figures 12 and 13 show the output of an erosion, a dilation, an opening, and a closing as applied on a grayscale input image using a flat 3x3 structuring element.

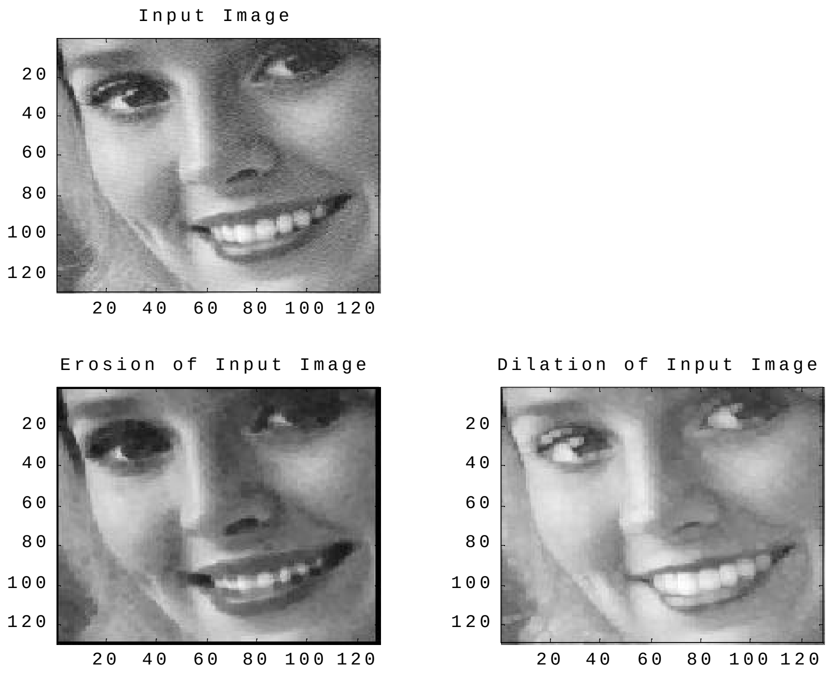


Figure 12: Grayscale Erosion and Dilation on an Input Image

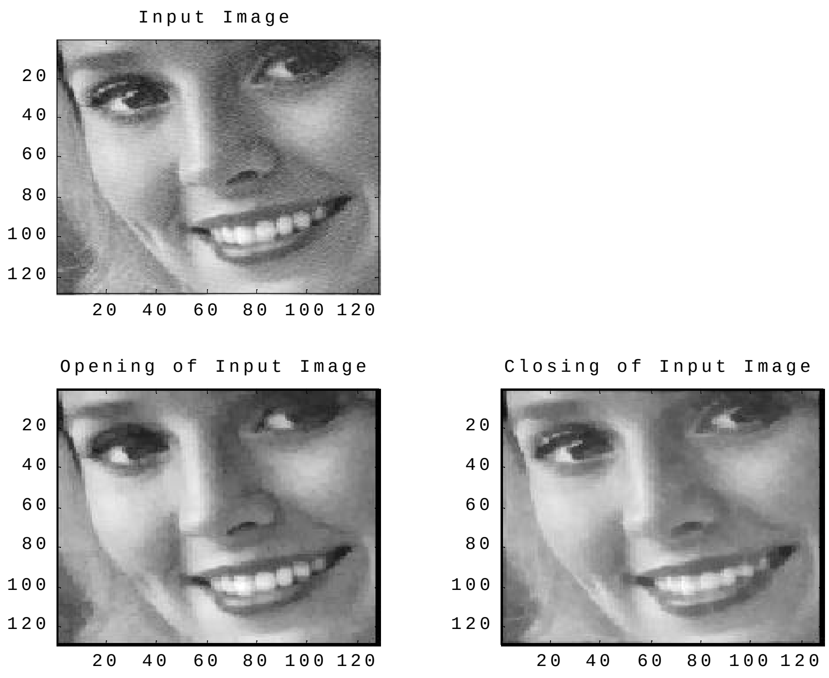


Figure 13: Grayscale Opening and Closing on an Input Image

From the figures above, the effects of morphological operations are apparent. In a grayscale image, erosion tends to grow darker areas, and dilation tends to grow lighter areas. Opening and closing each tend to emphasize certain features in the image, while de-emphasizing others. Iteratively, the morphological operations can be used to pick out specific features in an image, such as horizontal or vertical lines [20].

Convolution

Convolution is another commonly used algorithm in DSP systems. It is from a class of algorithms called spatial filters. Spatial filters use a wide variety of *masks*, also known as *kernels*, to calculate different results, depending on the function desired. For example, certain masks yield smoothing, while others yield low pass filtering or edge detection.

Algorithm

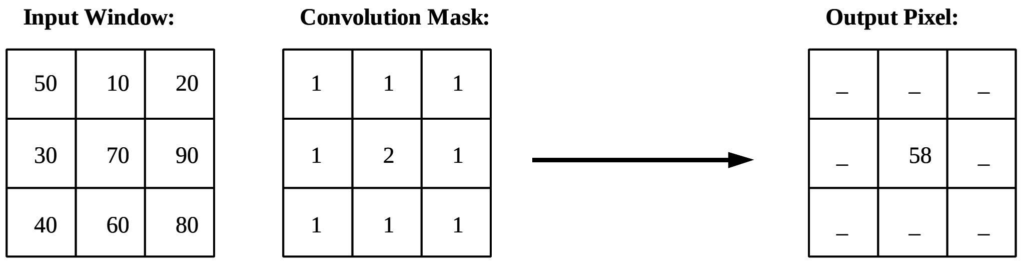
The convolution algorithm can be calculated in the following manner. For each input pixel window, the values in that window are multiplied by the convolution mask. Next, those results are added together and divided by the number of pixels in the window. This value is the output for the origin pixel of the output image for that position. Mathematically, this is represented using the following equation [21]:

$$y(n_1,n_2) = \sum_{k1=-\infty}^{\infty} \sum_{k2=-\infty}^{\infty} A(k_1,k_2)k(n_1-k_1,n_2-k_2),$$

where A is the input image and k is the convolution kernel.

The input pixel window is always the same size as the convolution mask. The output pixel is rounded to the nearest integer. As an example, Figure 14 shows an input pixel window, the convolution mask, and the resulting output. This convolution mask in this example is often used as a noise -cleaning filter [21].

The results for this algorithm carried over an entire input image will result in an output image with reduced salt-and-pepper noise. An important aspect of the convolution algorithm is that it supports a virtually infinite variety of masks, each with its own feature. This flexibility allows for many powerful uses.



Input Window:

50	10	20
30	70	90
40	60	80

Convolution Mask:

1	1	1
1	2	1
1	1	1

Output Pixel:

_	_	_
_	58	_
_	_	_

Convolution Output = (50*1 + 10*1 + 20*1 + 30*1 + 70*2 + 90*1 + 40*1 + 60*1 + 80*1)/9 = 57.7778 => 58

Figure 14: Convolution Algorithm Example

MATLAB Implementation

MATLAB was again used to produce a software version of the algorithm. It is called conv_3x3.m and is found in Appendix A. The MATLAB version of this algorithm performs con volution on an input image using a 3x3 -sized kernel, which can be modified as the user wishes. Figure 16 shows some examples of this algorithm on an input image, with the kernels K1, K2, and K3, as shown below in Figure 15.

1	2	1
2	4	2
1	2	1

K1

1	1	1
1	-7	1
1	1	1

K2

-1	-1	-1
-1	9	-1
-1	-1	-1

K3

Figure 15: Kernels Used to Compute the Images in Figure 16

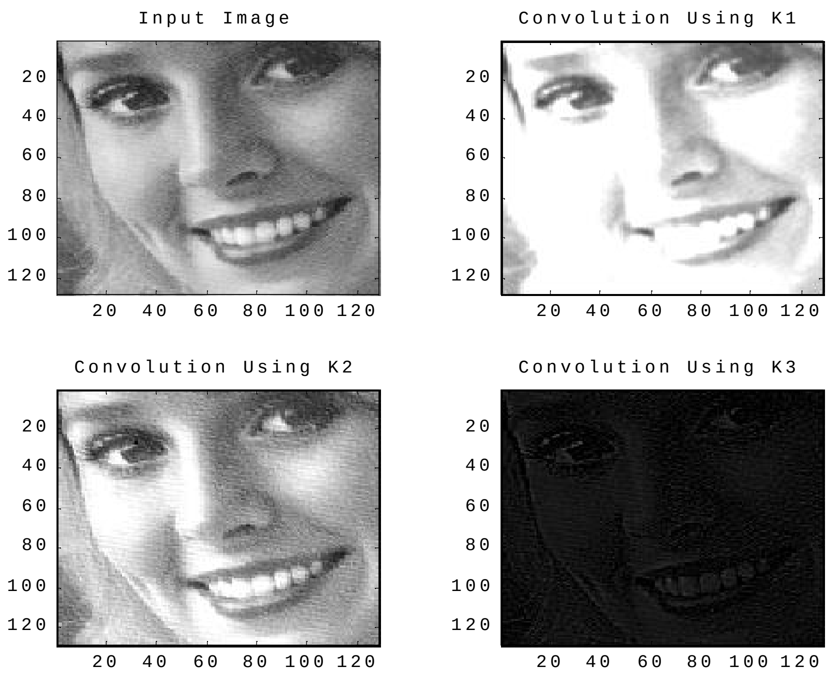


Figure 16: Grayscale Convolution

The figures above demonstrate the wide variety of feature enhancement possible with the convolution operation. It is important to note that these images are not scaled. Often an image t hat has been convolved will have a smaller pixel value variance than the input image. For example, in Figure 16 it is obvious that the convolution operation using the K3 kernel results in a mostly black image. Scaling would bring the brightest points in the image (around 30,30) up to a value of 256, and scale the rest of the image likewise. This results in an image that may have more discernable features.

CHAPTER III

PERTINENT NON-ALGORITHM WORK

Prior to a full hardware realization of the algorithms discussed in Chapter II, some initial non-algorithm work was necessary. This work included an overall VHDL hierarchy concept, the development of a moving window unit for real-time image data, a VHDL test bench for testing purposes, MATLAB interfaces for file input/output, and an analysis of the project data.

VHDL Hierarchy

Most hardware designers find it convenient to develop a hierarchy prior to any VHDL development. This is done to facilitate code reusability and to develop a common hierarchy. One of the main concepts of this project has been the development of VHDL code that is largely device independent, meaning that most of the code can be compiled for any FPGA architecture with little difficultly. The use of hardware-specific arithmetic and memory units has been limited to achieve nearly seamless code interchangeability. Essentially, a convolution algorithm written in VHDL with this approach should be easy to use on both Altera and Xilinx architectures.

In fact, for these designs, the only hardware-specific VHDL code lies within the FIFO memory units found within the 3x3 window generator, which is discussed in the second part of this chapter. Use of *port maps* in VHDL allow the designer to connect VHDL signals from the current level of hierarchy to another, separate VHDL architecture. This means that an algorithm can reference separate VHDL algorithms, thereby allowing code reuse. Figure 17 shows an example of how this is done in VHDL.

```
entity AlgorithmA is
   port (...);
end AlgorithmA;

architecture AlgorithmA_arch of AlgorithmA is

   component SubAlgorithmA
   port (...);
   end component SubAlgorithmA;

   component WindowGenerator
   port (...);
   end component WindowGenerator;

   component RowColumnCounter
   port (...);
   end component RowColumnCounter;

begin

   SubAlgorithmAMap: SubAlgorithmA
   port map (...);

   WindowGeneratorMap: WindowGenerator
   port map (...);

   RowColumnCounterMap: RowColumnCounter
   port map (...);

   -- algorithm process

end AlgorithmA_arch;
```

Figure 17: VHDL Component Mapping

3x3 Moving Window Archi tecture

In order to implement a moving window system in VHDL, a design was devised that took advantage of certain features of FPGAs. FPGAs generally handle flip -flops quite easily, but instantiation of memory on chip is more difficult. Still, compared wi th the other option, off-chip memory, the choice using on-chip memory was clear.

It was determined that the output of the architecture should be vectors for pixels in the window, along with a data-valid signal, which is used to inform an algorithm using the window generation unit as to when the data is ready for processing.

Since it was deemed necessary to achieve maximum performance in a relatively small space, FIFO units specific to the target FPGA were used. Importantly though, to the algorithms usi ng the window generation architecture, the output of the Altera and Xilinx window generation units is exactly the same. For example, for a given clock rate, the Altera unit's data -valid signal will change to a logic value of 1 at exactly the same time as the comparable Xilinx unit's data-valid signal. This useful feature allows algorithm interchangeability between the two architectures, which helped significantly cut down algorithm development time.

A 3x3 window size was chosen because it was small enough to be easily fit onto the target FPGAs, and is considered large enough to be effective for most commonly used image sizes. With larger window sizes, more FIFOs and flip-flops must be used, which increases the FPGA resources used significantly. Figure 18 shows a graphic representation of the FIFO and flip-flop architecture used for this design for a given output pixel window.

Appendix B shows the VHDL source code for the window generation unit. The Altera version is called window_3x3.vhd and the Xilinx version is called window_3x3_x.vhd. Not included are the codes for each FIFO entity, which were generated by the vendor tools.

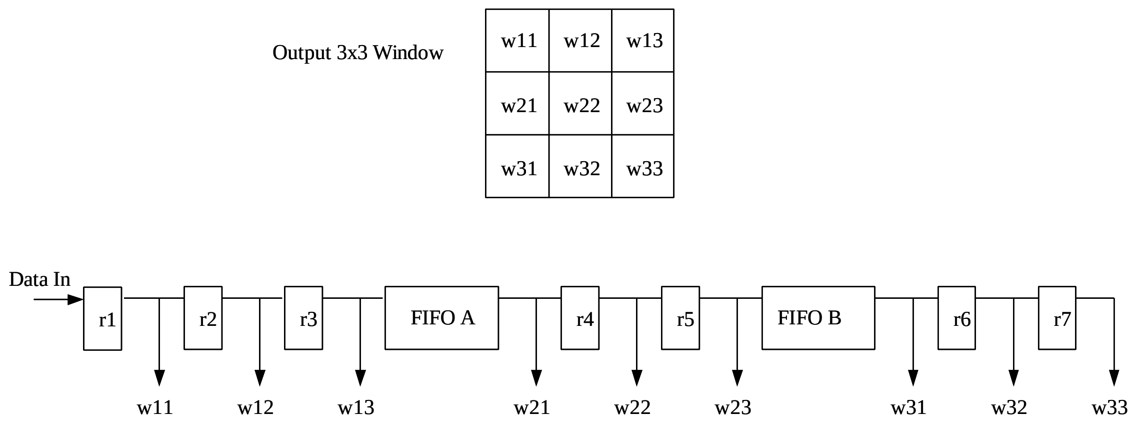


Figure 18: Architecture of the Window Generator

VHDL Test Bench Processes

In order to examine the VHDL code for correct functionality, VHDL tools provide a feature called simulation. Simulation takes the VHDL code and simulates how it would work in hardware. In order to do this, the designer must provide to the simulator valid inputs to produce expected outputs.

An efficient and common method of simulating VHDL code is through the use of a special type of VHDL code called a *test bench*. Test benches effectively surround the VHDL code the designer wishes to simulate and also provide stimulus to the tested entity.

A test bench for an algorithm is responsible in stimulating essential input signals to that algorithm. Since the designs used in this approach are all synchronous, a clock signal must be stimulated. In addition, all of the algorithms designed provide a reset functi onality, which allows the algorithms to be cleared at any point. In addition, since all the algorithms in this project take input images and produce some kind of output image, some method of data input and output must be provided for functional simulation .

When one wishes to process images with a VHDL algorithm, they must first create a test bench that can read in the file containing this data. If one wishes to view the processed image, another feature must be included into the test bench to allow file wr iting. These features are key to the usability of the test benches used in the project, and are quite useful when paired with a program such as MATLAB, which provides efficient image representation and viewing capabilities. This functionality is discusse d in the next section of this chapter. An example test bench for the ro_filt_3x3.vhd file, appropriately named ro_filt_3x3_TB.vhd, is found in Appendix B.

MATLAB – To – VHDL File IO Routines

In order to process real image data in VHDL simulations, it is necessary to create a method of transferring images in a standard format, a bitmap for example, into a file that the VHDL file read routine can understand. Since VHDL read/write routines operate easily with files consisting of a new word of data on each line, this method was chosen. MATLAB was used to implement this functionality because it is quite efficient in manipulating matrix data, such as images.

A MATLAB m-file called m2vhdl.m was created to take an input file in the bitmap format and convert it to a file with a new word of data on every line. Data in this format could then be read into the VHDL test bench by using standard VHDL text input/output functions. After this data has been run through the simulator (effectively, processed by the algorith m), the output data of the algorithm is written by the test bench into another file formatted in the same way.

Next, another MATLAB m-file was composed to read that data and convert it back into a matrix in MATLAB. This routine, called vhdl2m.m, allows analysis and comparison of VHDL -processed images with MATLAB-processed images. This analysis is crucial in determining proper functionality of the algorithms and is also useful in determining whether hardware design compromises produce invalid results. This aspect of the project is discussed in greater detail in the following chapter.

Project Dataset

The data used for this project consisted of 8 bit grayscale image data of size 128 by 128. For any real-time image processing system, the data size used gr eatly affects the project's performance on hardware, as well as the number of resources used on that hardware. Fortunately, the VHDL code for this project is written in such a way that facilitates different data bit widths and resolution quite easily. Th is can be done by replacing the FIFO elements in the window_3x3.vhd design and changing the *vwidth* generic in all of the VHDL designs. Some code needs to be hand coded, so inclusion of different data sizes will require some thought and a small amount of r edesign.

CHAPTER IV

VHDL ALGORITHMS

The focus of this project is the actual implementation of the proposed algorithms on target FPGA hardware. As discussed in previous chapters, this was accomplished by composing the algorithms in the VHDL language and synthesizing the algorithms for the FPGAs. This chapter discusses the hardware design specifics for each algorithm.

Rank Order Filter

The rank order filter was the first algorithm to use the window_3x3 pixel window generator. Since its operation is fairly simple, it was an ideal choice. As discussed above, the rank order filter must first sort the pixel values in a window in ascending (or rank) order. The most efficient method accomplishing this is with a system of hardware compare/sort units, which allow for sorting a window of nine pixels into an ordered list for use in the rank order filter.

The author implemented the structure found in Figure 19. This system results in a sorted list after a latency of 14 clock cycles. Since the design is pipelined, after the initial latency the system produces a valid sorted list on every clock cycle. The VHDL algorithm which implements this design, sort_3x3.vhd, is really just a series of registers and compares, as is shown if Figure 19. Not all levels of the sorting algorithm are shown to conserve space. Sort_3x3.vhd is found in Appendix B.

Every rxx box is a register and every cxx box is a compare unit, consisting of a simple decision. This design is accomplished quite simply in VHDL by using the following if/else statement:

```
if wx1 < wx2 then
      cx1_L <= wx1;
      cx1_H <= wx2;
else
      cx1_L <= wx2;
      cx1_H <= wx1;
end if;
```

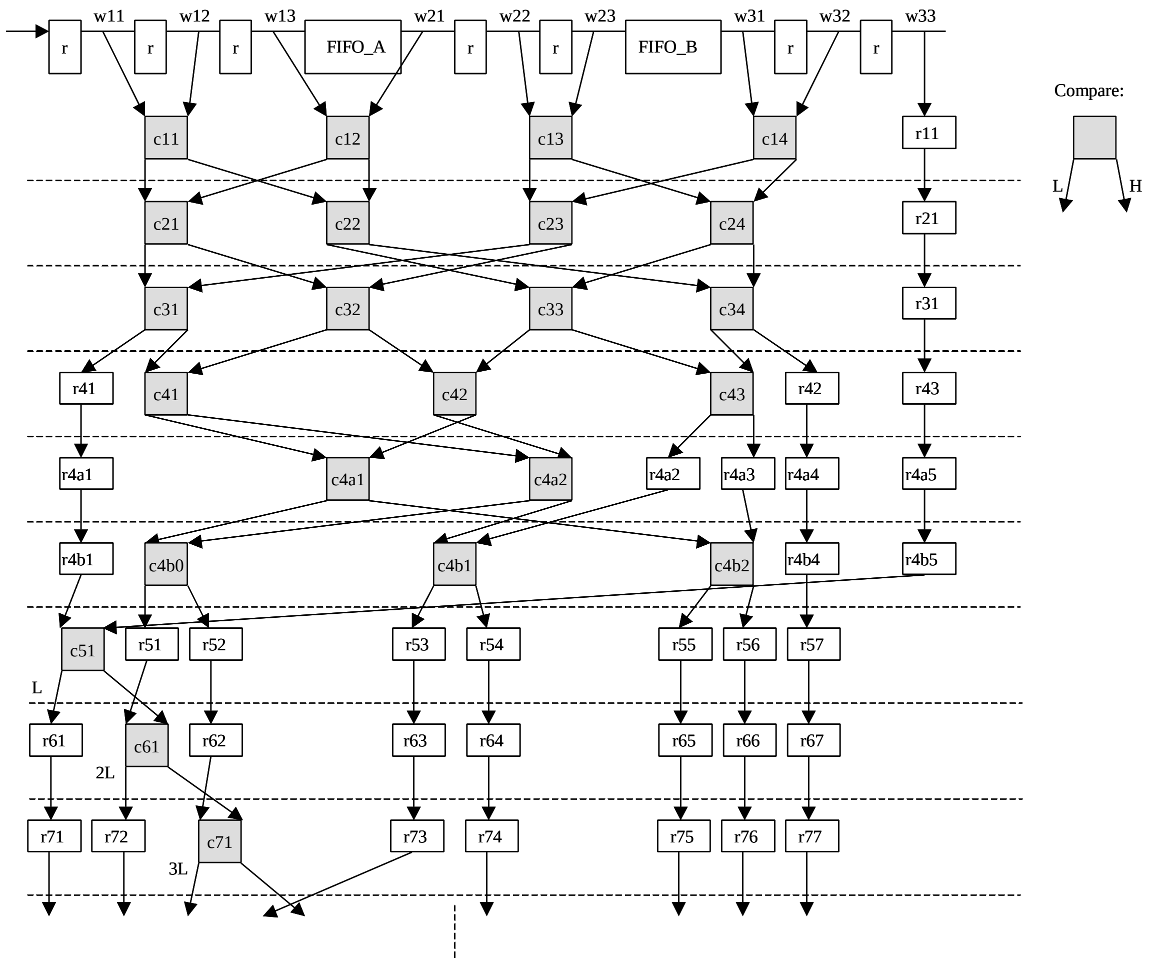


Figure 19: Hardware Design for Sorting Algorithm

After the sorted list is generated with the VHDL entity sort_3x3, the algorithm describing the rank order filter functionality, ro_filt_3x3.vhd, can operate on the list to produce its output. As is discussed above, the rank order filter outputs a pixel value in the origin location as specified by the rank of the filter.

In order to do this properly, a counter must be used to tell the output data-valid signal when to change to its 'on' state. Since it is desired that the output image be the same size as the input image, and use of the window generator effectively reduces the amount of valid output data, borders with zero value pixels must be place around the image. In order to do this properly, the counters are used to tell the algorithm when the borders start. A VHDL counter was written to count pixel movement as the data streams into the entity. Since images are two-dimensional data, two counters were needed: one to count rows and one to count columns in the image. The VHDL entity that implements this functionality is called rc_counter.vhd and is found in Appendix B. Since it is a separate VHDL entity, this counter was usable to later algorithms, where this functionality was also needed.

In order for the rank order filter to work properly, all three of these VHDL entities must be instantiated within the algorithm itself. This is done with standard VHDL component statements and port maps. Figure 20 shows the VHDL design structure used for this algorithm.

```
entity ro_filt_3x3 is
   port (...);
end ro_filt_3x3;

architecture ro_filt_3x3_arch of ro_filt_3x3 is

   component sort_3x3
   port (...);
   end component sort_3x3;

   component window_3x3
   port (...);
   end component window_3x3;

   component rc_counter
   port (...);
   end component rc_counter;

begin

   sort_3x3x : sort_3x3
   port map (...);

   window_3x3x : window_3x3
   port map (...);

   rc_counterx : rc_counter
   port map (...);

   -- algorithm process

end ro_filt_3x3_arch;
```

Figure 20: VHDL Algorithm Structure

Ro_filt_3x3 interprets and controls signals from all three entities to achieve a cohesive design, the result of which is a valid rank order filter. Order is specified with a VHDL generic, and is presently only modifiable pre-synthesis.

Comparison of VHDL and MATLAB Algorithms

Usage of the vhdl2m.m file converter allows for analysis of the results of the VHDL simulation of algorithms. This is particularly useful because it allows for comparison between hardware (e.g. VHDL) algorithms and software (e.g. MATLAB) algorithms. This is exciting because it allows the designer to a) verify a hardware alg orithm's accuracy and b) decide whether or not to implement design tradeoffs based on output validity.

Figures 21 and 22 show comparisons of the VHDL and MATLAB algorithms for two orders. Also shown are error plots and mesh plots, which show a three -dimensional view of the error. In these cases it is obvious that the two algorithms are identical.

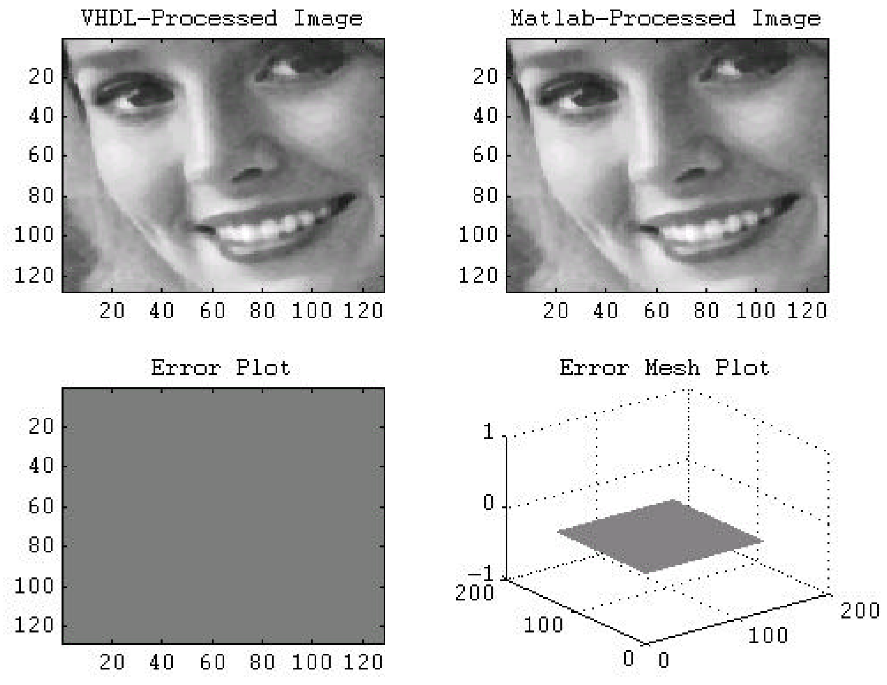


Figure 21: VHDL and MATLAB Comparison Plots for ro_filt_3x3 with Order = 4

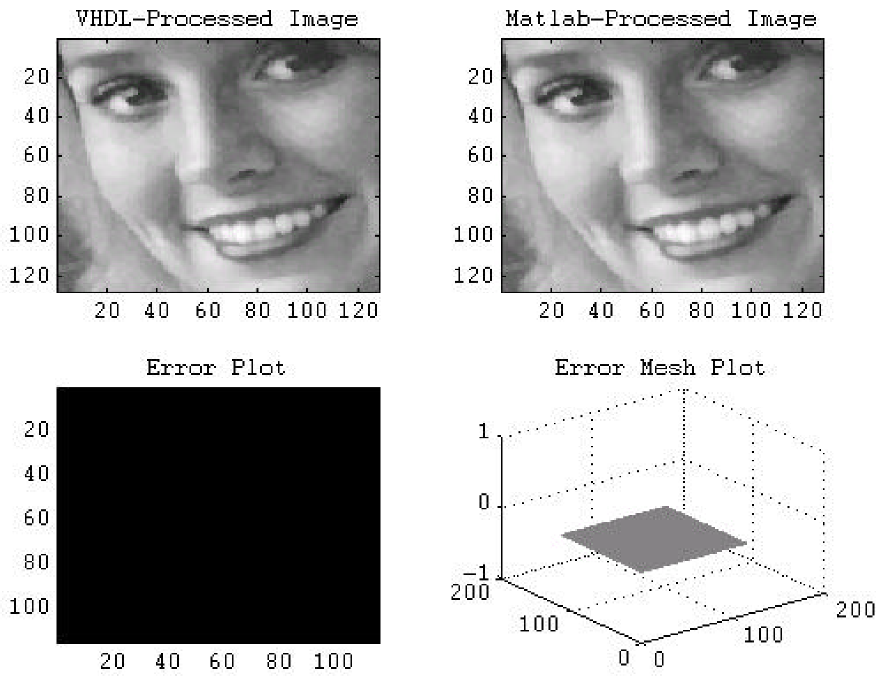


Figure 22: VHDL and MATLAB Comparison Plots for ro_filt_3x3 with Order = 8

Algorithm Synthesis

The VHDL rank order filter design has been synthesized for both the Altera and Xilinx architectures. Since the Xilinx Virtex is a newer generation FPGA, it was expected that it would provide superior performance over the Altera FLEX 10K FPGA. This surmise was true, and was a constant throughout the design. Table 1 shows the synthesis results for the two architectures.

Table 1: Performance and Resources Used for ro_filt_3x3 Synthesis

FPGA	% Memory Used	% Logic Used	Maximum Synthesized Pe rformance[1]
Altera FLEX 10k100	8	32	33 MHz / 2014 Frames Per Second
Xilinx Virtex XCV300BG352	12	19	47.134 MHz / 2876 Frames Per Second

1: for project data size (128x128 8 bit grayscale)

Morphological Operators

Since the most commonly used morpholog ical operators are those with flat structuring elements of square shape, it was decided that the development time necessary to implement full grayscale structuring element morphological operators was unfounded. The most common functionality of the morphol ogical operators is implemented in the rank order filter discussed above, so a separate version of the morphological operators was deemed unnecessary.

Comparison of VHDL and MATLAB Algorithms

The ro_filt_3x3 simulation was again used to verify the algo rithm's accuracy, this time against the MATLAB algorithm ro_filt.m using orders of 1 and 9, which yields the same results as aip_erode_gs and aip_dilate_gs using flat 3x3 structuring elements, respectively. Figures 23 and 24 show the comparison plots for the two basic morphological operators. Again, the figures show that there is no error in the VHDL algorithm.

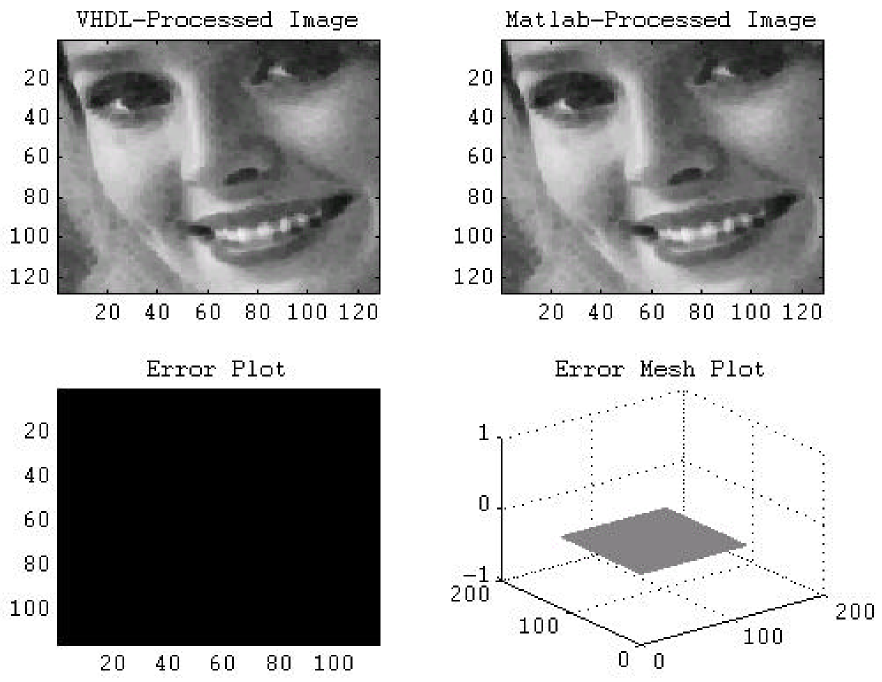


Figure 23: VHDL and MATLAB Comparison Plots for ro_filt_3x3 with Order = 1 (erosion)

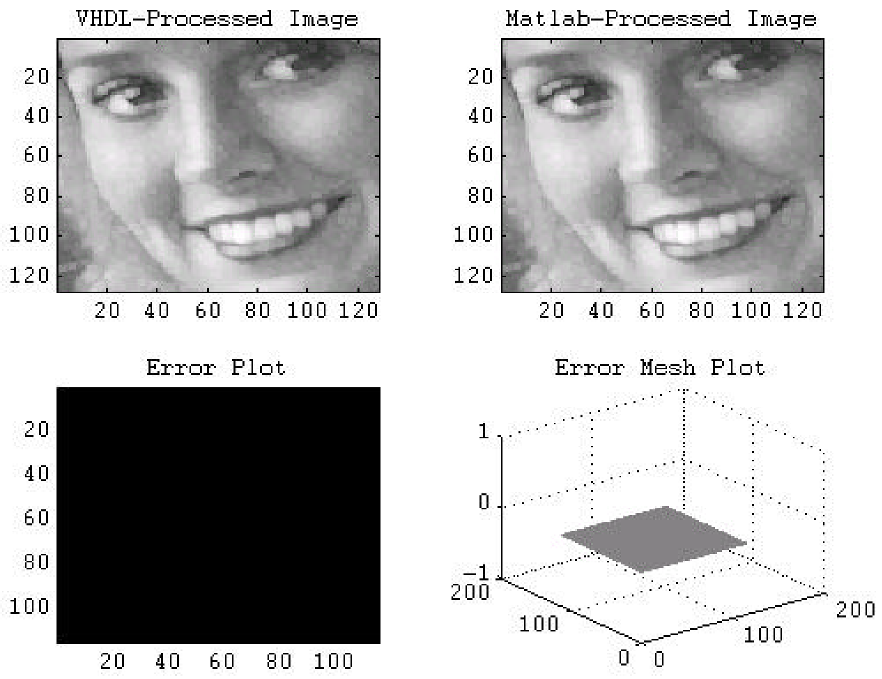


Figure 24: VHDL and MATLAB Comparison Plots for ro_filt_ 3x3 with Order = 9 (dilation)

Convolution

The design of the convolution algorithm in VHDL was a much more difficult problem than the rank order filter design. This was due to its use of more complex mathematics. For example, the rank order filter really just sorts the pixels in a window and outputs one of them, while the convolution algorithm uses adders, multipliers, and dividers to calculate its output. On FPGAs, use of mathematics tends to slow down performance. Many designers favor techniques that reduce the algorithm's dependency on complex mathematics. Still, since the mathematics used in convolution are simple, implementation of a convolution algorithm was an achievable goal.

Yet another obstacle in this algorithm's design was implementing the capability to handle negative numbers. In a proper convolution algorithm, the mask can (and often does) consist of negative numbers. Effectively, the VHDL had to be written to handle these numbers by using *signed* data types. Signed data simply means that a negative number is interpreted into the 2's complement of its non-negative dual. This means that all vectors within the design must use an extra bit as compared to unsigned numbers. The extra bit always carries the sign of the number – 0 for a positive number, 1 for a negative number.

Because of this, the output of the convolution algorithm is a number in 2's complement. In order for another unit to interface data from this algorithm, the unit must be able to understand or convert 2's complement data. Fortunately, this is a simple matter in the ACS system, which is discussed in the following chapter.

Addition and multiplication were instantiated using simple + and * signs in the VHDL code. The VHDL synthesis tool provides mapping to efficient hardware mathematics designs for each of these, so device-specific parameterized modules were not necessary.

Since a proper convolution involves a division by the number of pixels in the window, some thought had to be put into this part of the algorithm's hardware implementation. Hardware dividers on FPGAs are quite large and slow. In addition they must be tied directly to the FPGA's architecture, meaning that one divider would not work for both architectures pursued. It was deemed necessary to instead use the bit shifting method of division. Since this is only possible with powers of two, a divide by 8 was implemented instead of a divide by 9, as was planned in the algorithm's design. The effect of this is discussed in the Algorithm Synthesis section of this sub-chapter.

Figure 25 shows a graphic representation of the mathematics of the hardware convolution. Note that a valid output for the convolution algorithm occurs six clock cycles after the first window is valid. Since this design is pipelined and will run in the megahertz range, this kind of startup latency has very little effect on overall design speed.

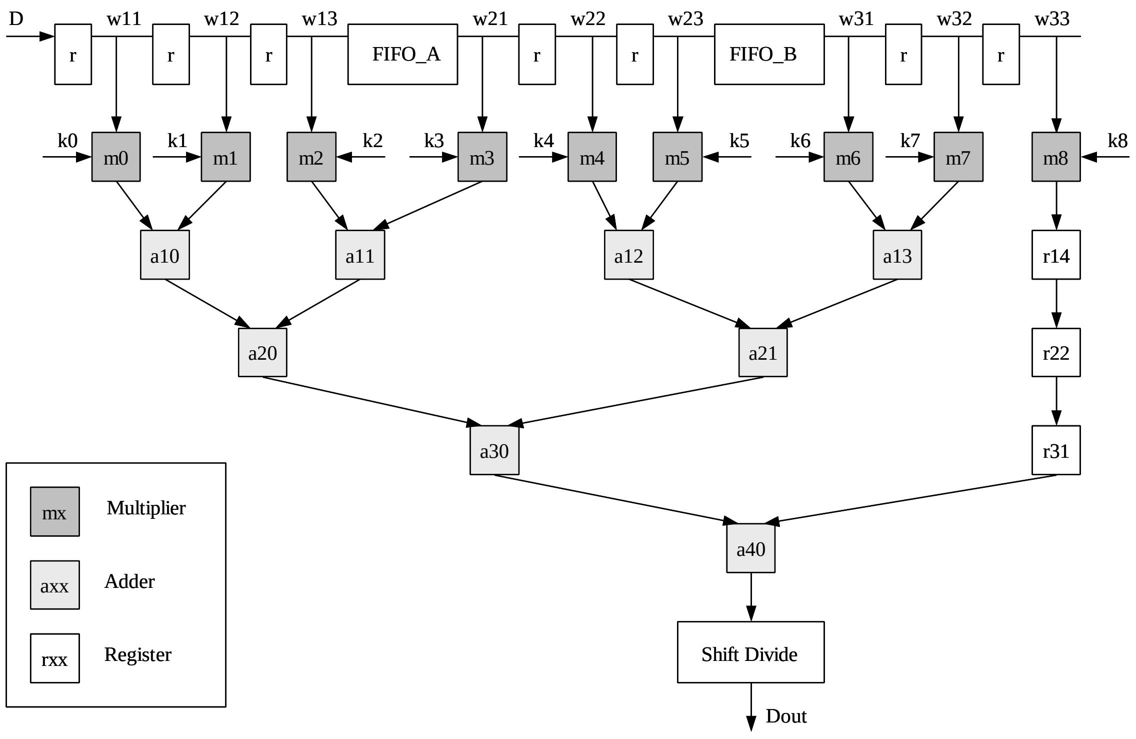


Figure 25: Hardware Design of Convolution

The VHDL implementation of the convolution, which is called conv_3x3.vhd, has a hierarchy that is similar to the ro_filt_3x3 hierarchy. It contains an instantiation of window_3x3 to provide access to the moving pixel window functionality as well as rc_counter for counting capabilities. The VHDL algorithm structure is shown in Figure 26. The VHDL sou rce code is found in Appendix B.

Optimization of the convolution algorithm can be easily achieved if one has limited kernel specifications. For example, if all coefficients in the kernel are powers of two, the VHDL synthesizer is able to result in a desig n that uses fewer resources. This is due, of course, to the way numbers are represented in digital systems, where a number that is a power of two is represented with only one bit. Further optimization is possible by reducing the bit widths of the kernel constants. This is result in a smaller coefficient data range, but this compromise may be acceptable in certain cases.

```
entity conv_3x3 is
   port (...);
end conv_3x3;

architecture conv3x3_arch of conv_3x3 is

   component window_3x3
   port (...);
   end component window_3x3;

   component rc_counter
   port (...);
   end component rc_counter;

begin

   window_3x3x : window_3x3
   port map (...);

   rc_counterx : rc_counter
   port map (...);

   -- algorithm process

end conv_3x3_arch;
```

Figure 26: VHDL Algorithm Structure

Comparison of VHDL and MATLAB Algorithms

MATLAB again played an important part in the analysis o f the VHDL file outputs for an algorithm. In the conv_3x3 design it was especially important because the divide by eight compromise discussed above changed the nature of the algorithm's output. Analysis of this was important to determine whether or not this compromise results in a reasonably valid output.

Figures 27 and 28 show comparisons of the VHDL- and MATLAB-convolved images using the K1 kernel described in Chapter 2. Figure 29 shows the mesh error plot of the VHDL -processed image versus the MATLAB-processed image using a divide by 9. From this plot it is evident that the compromise of using a shift divide does result in a different output, but this is fairly consistent over the entire image. Therefore, it is reasonable to assume that a divide by 8 convolution using a 3x3 window is an adequate approximation of a real divide by 9 convolution. Figure 29 shows that the minimum pixel value difference is approximately 10 pixels and the maximum pixel value difference is approximately 53 pixels.

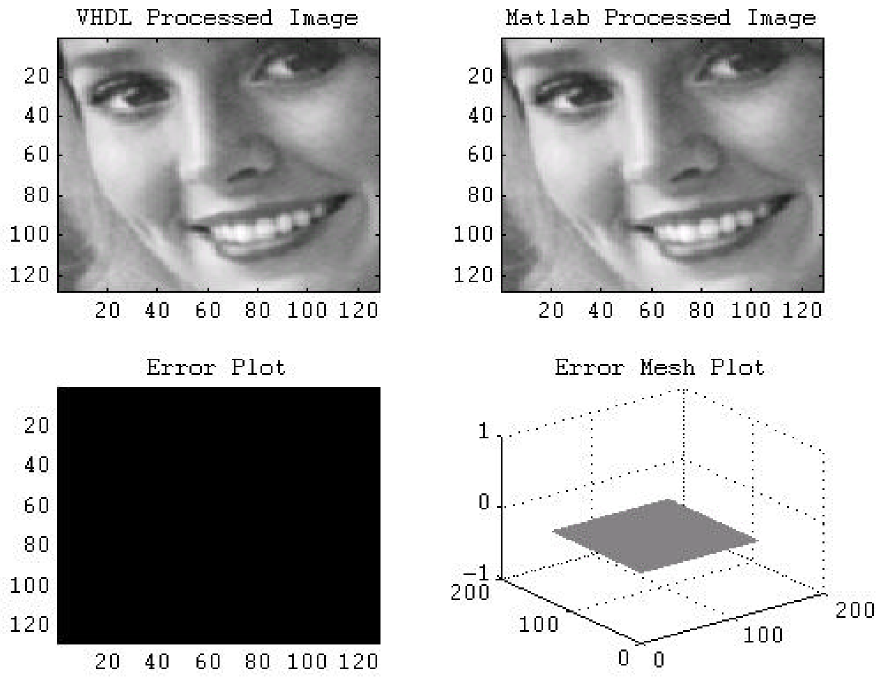


Figure 27: VHDL and MATLAB Comparison Plots for conv_3x3 with K1 Kernel

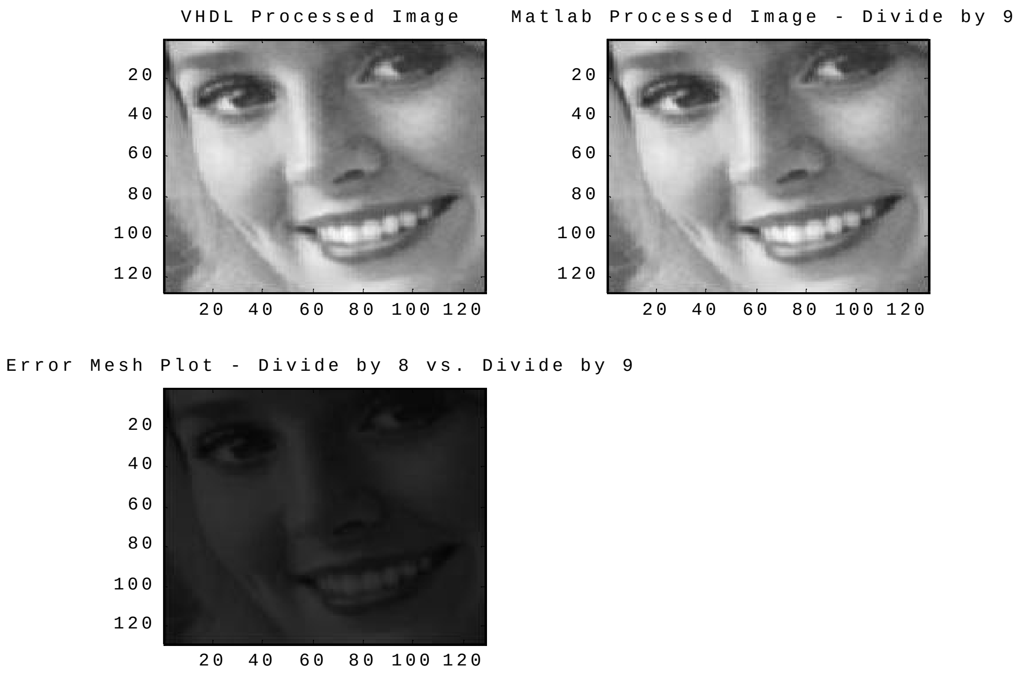


Figure 28: Comparison Plots for VHDL (Divide by 8) and MATLAB (Divide by 9), Showing Error

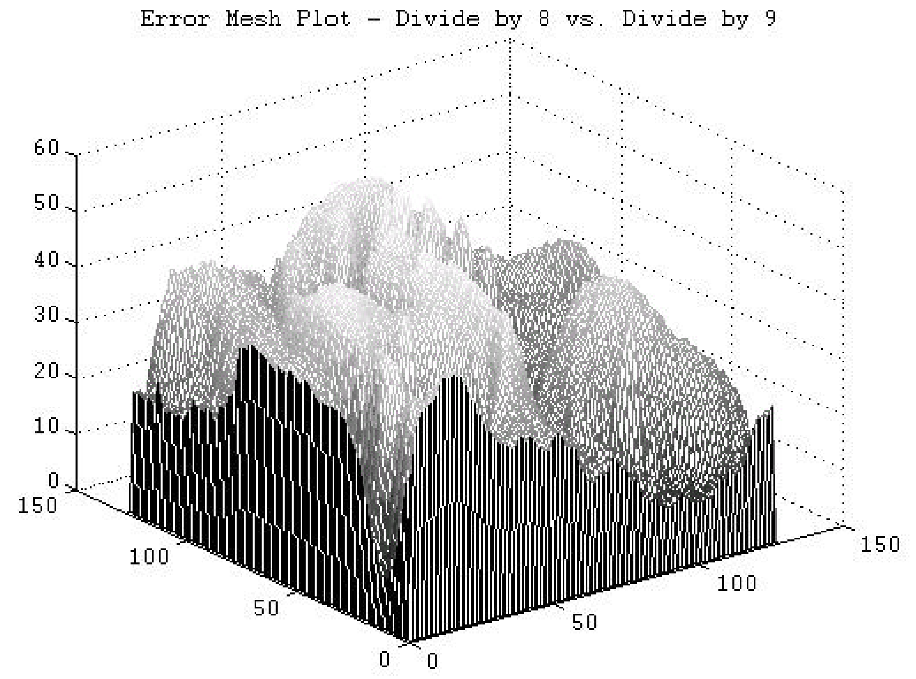


Figure 29: Mesh Plot of Error for VHDL (Divide by 8) and MATLAB (Divide by 9)

It is believed that reasonable normalization of the algorithm to compensate for the divide by 8 errors should be possible by adding an average of the error shown above to the output of conv_3x3.

Algorithm Synthesis

The hardware design for the 3x3 convolution algorith m was also synthesized for both Altera and Xilinx FPGA architectures. Once again, the Xilinx FPGA provided a faster implementation, just as expected. Table 2 shows the results for the synthesis of the conv_3x3 design.

Table 2: Performance and Resources Used for conv_3x3 Synthesis

FPGA	% Memory Used	% Logic Used	Maximum Synthesized Performance [1]
Altera FLEX	8[2]	24[2]	28.49 MHz / 1738 FPS [2]
10k100	8[3]	26[3]	32.67 MHz / 1994 FPS [3]
Xilinx Virtex	12[2]	19[2]	51.39 MHz / 3136 FPS [2]
XCV300BG352	12[3]	19[3]	48.952 MHz / 2987 FPS [3]

1: for project data size (128x128 8 bit grayscale)
2: for kernel consisting of powers of 2
3: for kernel consisting of all powers of 2 except for one element

CHAPTER V

INTEGRATION OF ALGORITHMS INTO ISIS ACS TOOLS

Integration of this system in to a real FPGA system is key to the algorithms' success. At the Institute for Software Integrated Systems (ISIS), a reconfigurable system consisting of Altera FPGAs is in use. The Xilinx Virtex FPGAs are not currently a part of this system, but will be a t some point, which is the reason this target was pursued. This system requires that FPGA algorithms must be integrated into a modeling environment, called ACS [22]. This modeling environment is useable by implementing the design in a modeling tool calle d GME (Graphical Model Editor). This tool allows for VHDL files (or DSP files, among others) to be represented as a model or a set of models. The ACS modeling environment *interprets* these models by synthesizing a hardware system that is represented in GM E.

The ACS system has a library of algorithms for various applications. The algorithms presented in this thesis will be integrated into that library for later use. As mentioned previously, algorithms in ACS can be mapped for any number of platforms, in cluding DSPs and FPGAs. Ideally, each algorithm has more than one implementation. For example, to allow maximum flexibility in system synthesis, DSP implementations of the image processing algorithms presented in this thesis should be written. This will allow the system designer to have a choice on which algorithm to use based on the system's requirements. For example, if a high -speed system is desired, the fastest combination of FPGA and DSP algorithms can be synthesized. If a low power system is preferred, a different combination of devices can be synthesized with this characteristic. This flexibility is one of the key advantages of the ACS system, and is represented in Figure 30, which shows a system containing both FPGA and DSP versions of the same algorithm. It is important to note that the system in Figure 30 is not a parallel system. Rather, it shows two options for the same algorithm.

When a VHDL algorithm is written for an FPGA in the ACS system, it must be characterized in terms of its maxim um performance and resource usage. This allows the system synthesis to be based on real algorithm properties. An algorithm's model contains attribute information where this data is represented. Figure 31 shows the attributes for an ACS model.

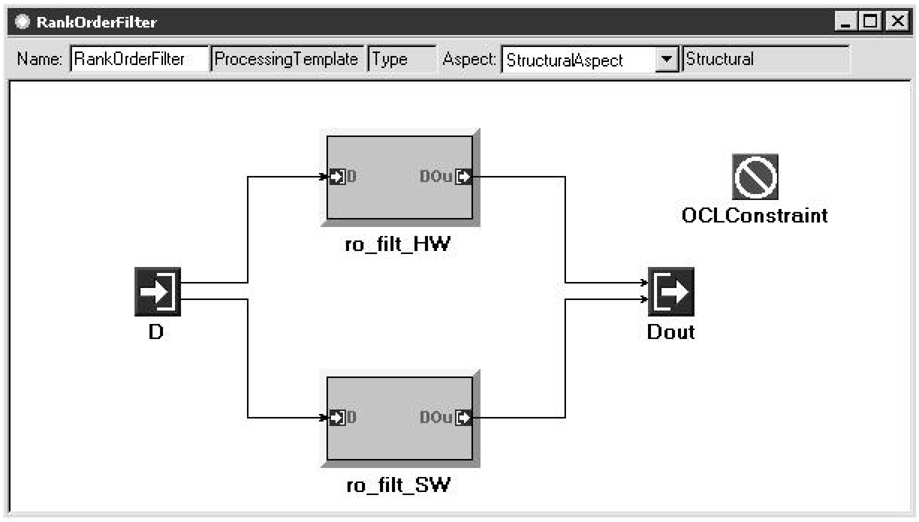
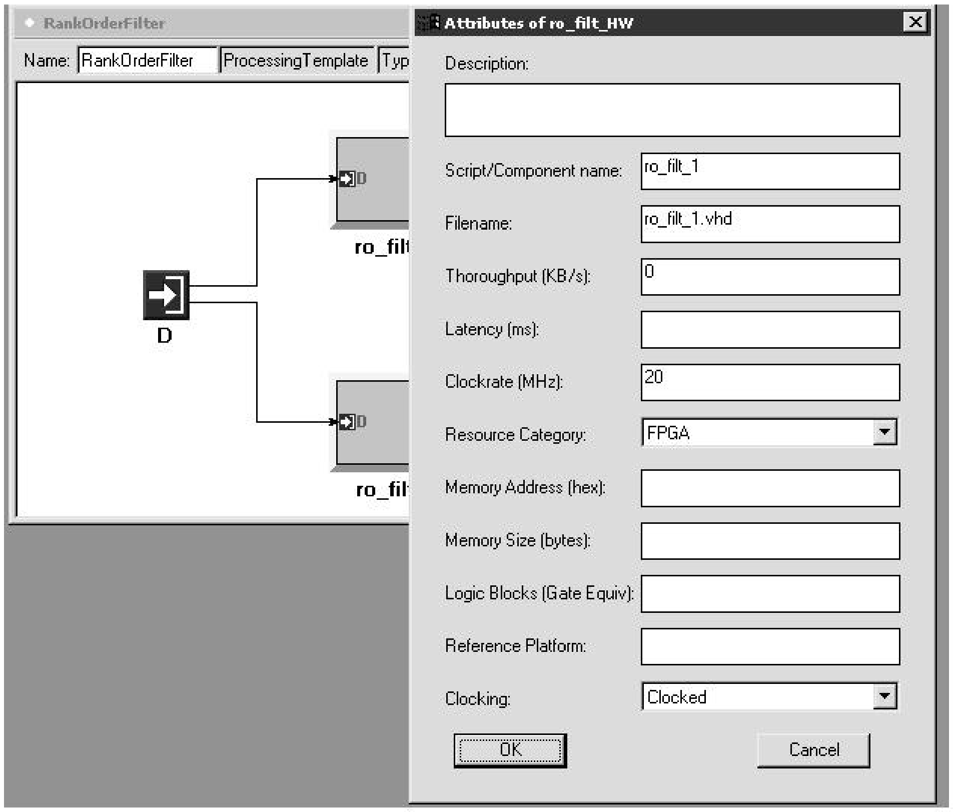


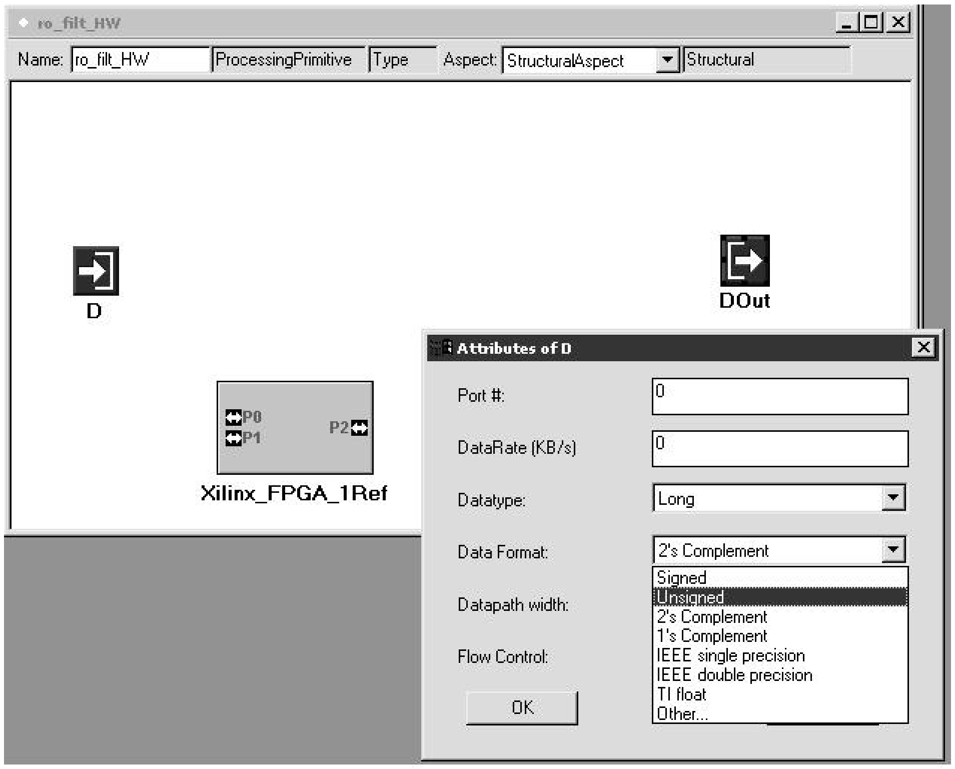
Figure 30: Representation of an Algorithm in an ACS Model



Description:	
Script/Component name:	ro_filt_1
Filename:	ro_filt_1.vhd
Thoroughput (KB/s):	0
Latency (ms):	
Clockrate (MHz):	20
Resource Category:	FPGA
Memory Address (hex):	
Memory Size (bytes):	
Logic Blocks (Gate Equiv):	
Reference Platform:	
Clocking:	Clocked

Figure 31: Attributes of an ACS Model

A wide variety of data types are supported in ACS, and are selectable in I/O port attributes. The rank order filter design detailed in this thesis uses the unsign ed data type while the convolution filter design uses the signed data type. While several other data types are supported in the ACS modeling software, these two designs only work with their specified data types as of this writing. This limitation can be overcome by implementing data type converters in a top -level design containing the algorithms. Figure 32 shows a screen capture of I/O port data type selection in GME.



Name: ro_filt_HW | ProcessingPrimitive | Type | Aspect: StructuralAspect | Structural

Port #: 0
DataRate (KB/s): 0
Datatype: Long
Data Format: 2's Complement
Datapath width:
Flow Control:

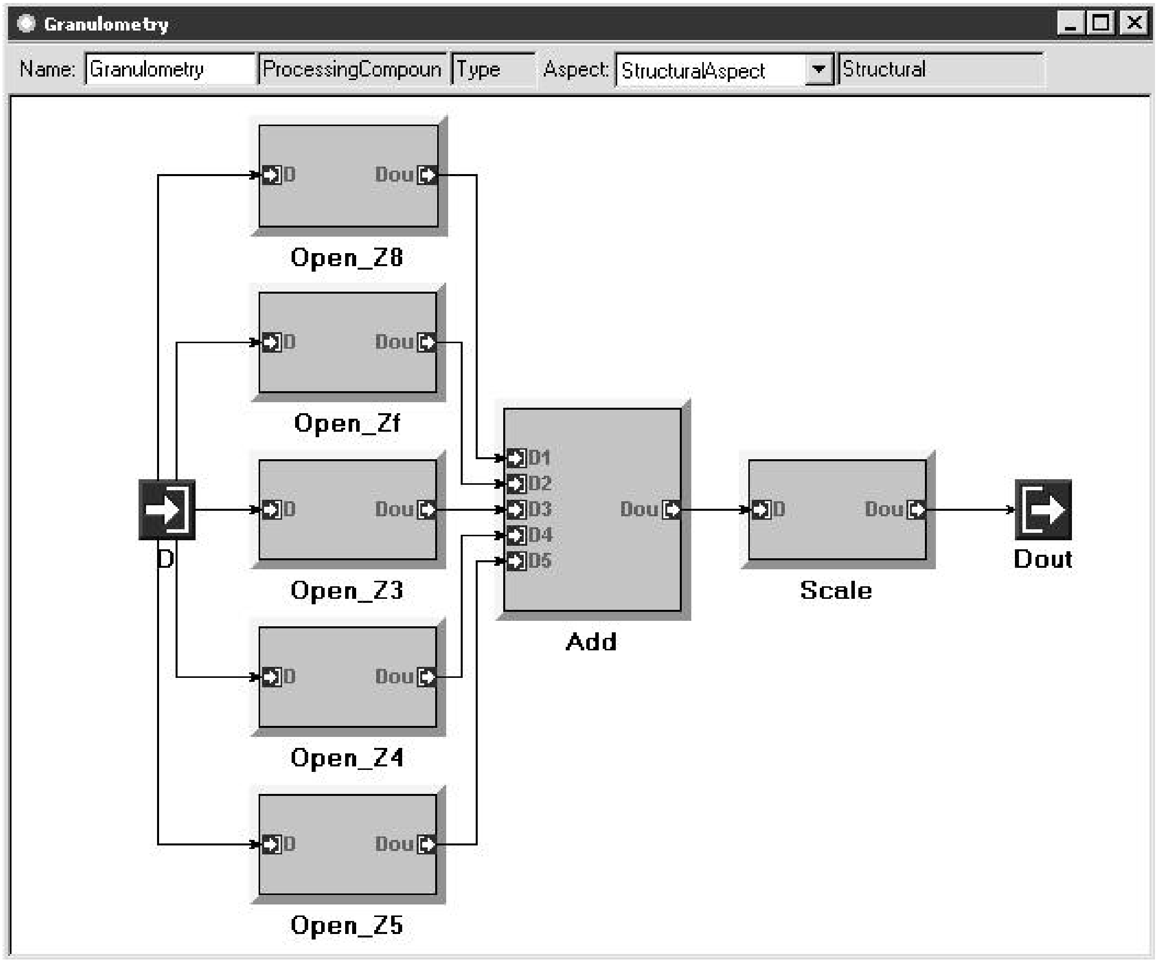
Signed
Unsigned
2's Complement
1's Complement
IEEE single precision
IEEE double precision
TI float
Other...

Figure 32: I/O Port Data Type Selection in an ACS Model in GME

The VHDL files that are specified in the models must adhere to a specific format in order to work properly in the system. Designers are given a choice between two formats: a valid/clear system or a standby/ready system. In addition, the paradigm supports designer -defined formats. Data width can be any number and the data format can be any one of a number of formats.

The algorithms composed in this thesis were written to use the same type of data, which is a set of 8-bit unsigned integers representing the pixels in an image. However, the algorithms were written to work in a streaming -data fashion, where as soon as data first arrives into the entities, it is assumed that a new pixel of image data arrives on each clock pulse. In order for these algorithms to work with the ACS system, it was imperative to modify them slightly so that they would be able to accept data that does not necessarily arrive on each clock. This involved adding another layer to the VHDL design, which provides the data valid/clear signals mentioned above . These designs will be called ro_filt_3x3_top and conv_3x3_top and will be detailed in a later paper.

An example of a morphological granulometry [20] in an ACS compound model is shown in Figure 33. This particular granulometry operation consists five m orphological openings (each of which consist of an erosion followed by a dilation) followed by addition and scaling operations. Figure 34 shows how the erosion and dilation algorithms combine to form an opening operation in an ACE compound model. Figure 35 shows the erosion algorithm in an ACE primitive model, and how it is mapped to a particular FPGA. This example shows the power of the ACS environment for system synthesis.


Granulometry

Name: Granulometry | ProcessingCompoun | Type | Aspect: StructuralAspect | Structural

Figure 33: Morphological Granulometry Example

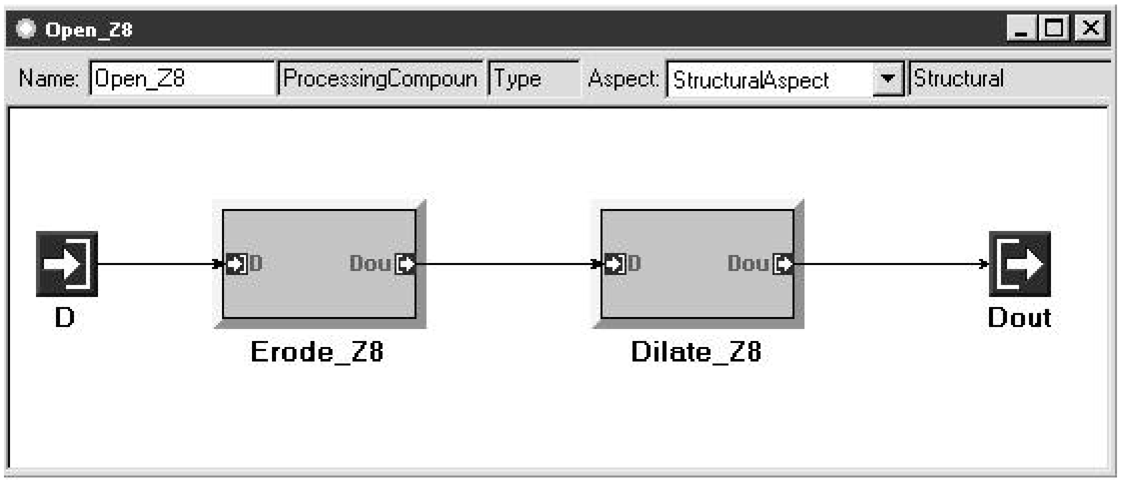


Figure 34: Morphological Open ing from Example

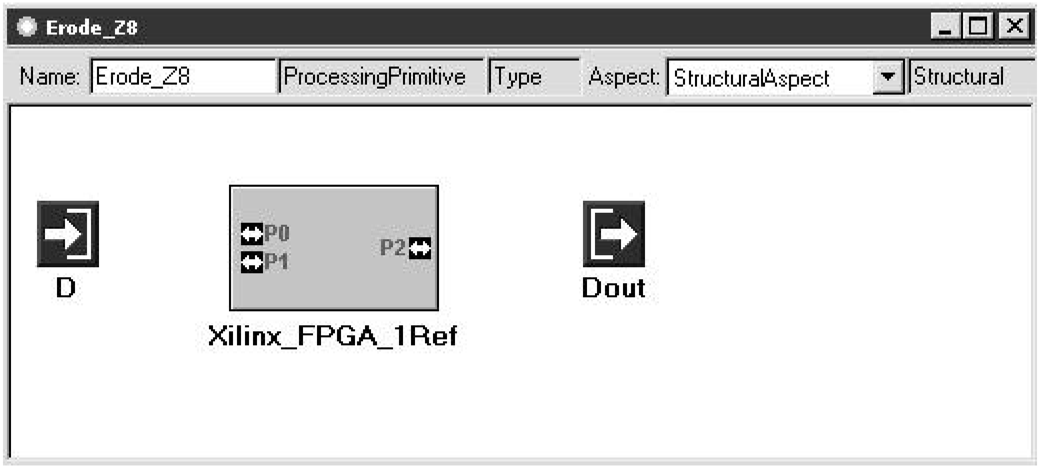


Figure 35: Morphological Erosion from Example

The modification for integration into the ACS system will result in a lower throughput. This is due in part to additional synthesized logic and in part to the lower efficiency of the ACS d ata valid/clear system as compared to a traditional streaming data system. Since data valid signals must be sent and acknowledged for incoming data, the algorithms cannot process the data on every clock pulse. However, since the dataset is relatively sma ll and the algorithms are capable of rather high speeds, a resultant speed of around 20 MHz is expected by using this method. While this is a performance hit, it still falls within the requirements imposed by the dataset and the design specifications. Th is is a compromise, but with the algorithms in the ACS modeling environment, assembly of systems can be much faster than in traditional DSP systems.

CHAPTER VI

CONCLUSIONS

The development of FPGA image processing algorithms can at times be quite tediou s, but the results speak for themselves. If high -speed, windowing algorithms are desired, this paper shows that FPGA technology is ideally suited to the task. In fact, with the aid of the window generator, a whole series of image processing techniques is available to the designer, many of which can be synthesized for high -speed applications.

One of the drawbacks of the techniques presented in the paper is the large size of the algorithms, as shown in the Algorithm Synthesis section of Chapter IV. This i s largely due to the FIFO units being used in the design. If off-chip RAM is used for FIFO operations, the designs' synthesized size can be greatly reduced.

Also, the stack filter [23] method of image processing can greatly reduce the size of algorithms using a window generator. Still, this method achieves a more serial method of processing, which is not entirely efficient with FPGA systems. The design presented here is quite capable, and it tries to take advantage of the parallelism possible with FPGA devices.

A great deal of knowledge was gained from the completion of this project. While FPGAs are excellent for some uses, such as a large number of image processing applications, difficulties in using more complex mathematics speak volumes towards the argument of using dedicated DSP chips for some applications. Indeed, it is expected that a designer who desires the best combination of speed and flexibility should look toward a system consisting of both FPGAs and DSPs. Such a system can take advantage of the positive aspects of each architecture, and can allow the designer to create an algorithm on a system that is best suited for it. That said, it should also be noted that this project's algorithms were excellent choices for FPGA implementation. This is because they don't use floating-point mathematics and they include no complex mathematics.

VHDL simulation and FPGA synthesis tools are getting consistently better. Simulation of large and complex VHDL is now simple and fast, and generic VHDL can eas ily be synthesized into efficient hardware-specific designs. It is expected that as the FPGA hardware continues to improve, so will the tools. In the future, the longer development time that is inherent in FPGA design may disappear, and FPGA design will be more comparable to DSP design.

Future Work

The interchangeable nature of the VHDL components of this design allow for its components to be used in different designs quite easily. For example, the window_3x3 architecture allows it to be used in any algorithm that uses a pixel window to compute its output. Since VHDL components can easily be instantiated in any design, using the pixel window generator is as simple as dropping component and port map statements into another VHDL design.

Because of this, the applications for the code created for this project can be used in many different image processing algorithms. With the window generator and row/column counter code complete, about fifty percent of the work is done and the designer simply has to use t heir outputs to generate a desired result. It could be said that the real result of this project is not simply a few algorithms, but instead a system of VHDL code which allows for efficient implementations of many algorithms. Still, these VHDL designs sh ould be made to operate more generically, so that modification of hard -coded values is not necessary.

A large part of the improvement possible in this design lies in the algorithms themselves. For the rank order filter, changing the order to be an input v ector would allow on-the-fly switching of algorithm properties. While this does increase the synthesized size of the design, it also maximizes its on -chip capability. Similarly, if the kernel for the convolution design were to be changed to inputs instea d of constants in a package, the convolution algorithm would also have increased functionality, this time with no added logic to synthesize.

Another extension to this work could be creation of larger -sized window generators. With larger image sizes, small window sizes such as 3x3 are not as useful. Windows of size 5x5 or 7x7 are rather easily attainable. Still, memory limitations will relegate such designs to larger FPGAs such as the Xilinx Virtex XCV300. In addition, the sorting algorithm sort_3x3 cann ot be used with larger window sizes.

Indeed, a sorting algorithm for larger window sizes is an incredibly daunting task. Instead, a different method of calculating rank order would have to be considered.

Despite these possible improvements, this thesis i s considered to be a success. The knowledge and experience gained from completing this project will certainly be helpful in future designs.

APPENDIX A

MATLAB M-FILES

ro_filt.m

```
function output_image = ro_filt(image_file,order);
%
%   filename:  ro_filt.m
%   author:    Tony Nelson
%   date:      1/11/00
%   detail:    performs basic 3x3 rank order filtering
%

input_image = LoadImage(image_file);         % loads image into input_image
[ylength,xlength] = size(input_image);       % determines size of input image
output_image(1:ylength,1:xlength) = zeros;   %inits output_image

% loops to simulate SE window passing over image
for y=1:ylength-2
    for x=1:xlength-2
        window = [input_image(y:(y+2),x:(x+2))];
        window_v = [[window(1,1:3)] [window(2,1:3)] [window(3,1:3)]];
        sorted_list = sort(window_v);
       output_image(y+1,x+1) = sorted_list(order);
        sorted_list(order);
    end
end

%plots ro filtered image
figure;
image(output_image)
colormap(gray(256));
title('Rank Order Filter Output');
```

aip_erode_gs.m

```
function output_image = aip_erode_gs(image_file,se_file);
%
%   filename:  aip_erode.m
%   author:    Tony Nelson
%   date:      12/7/99
%   detail:    performs grayscale erosion on image_file using specified se_file
%

[Bx,By,Ox,Oy,SE_data] = LoadSE_gs(se_file);  % loads SE parameters and data
input_image = LoadImage(image_file);      % loads image into input_image
[ylength,xlength] = size(input_image);       % determines size of input image
output_image(1:ylength,1:xlength) = zeros;   %inits output_image

% loops to simulate SE window passing over image
for y=1:ylength-By
   for x=1:xlength-Bx
      im_se = input_image(y:(y+By-1),x:(x+Bx-1)) - SE_data;
      output_image(y+Oy,x+Ox) = min(min(im_se));
   end
end

%plots eroded image
figure;
imagesc(output_image)
colormap(gray);
```

```
title([image_file, ' eroded by ', se_file]);
```

aip_dilate_gs.m

```
function output_image = aip_dilate_gs(image_file,se_file);
%
%   filename:  aip_dilate_gs.m
%   author:    Tony Nelson
%   date:      12/7/99
%   detail:    performs grayscale dilation on image_file using specified se_file
%

[Bx,By,Ox,Oy,SE_data] = LoadSE_gs(se_file);  % loads SE parameters and data
input_image = LoadImage(image_file);      % loads image into input_image
[ylength,xlength] = size(input_image);       % determines size of input image
output_image = input_image;   %inits output_image

SE_data = -(SE_data);  % finds negative of SE_data for dilation

% loops to simulate SE window passing over image
for y=1:ylength-By
   for x=1:xlength-Bx
      % dilation is the dual of erosion....
      im_se = input_image(y:(y+By-1),x:(x+Bx-1)) - SE_data;
      output_image(y+Oy,x+Ox) = max(max(im_se));
   end
end

%plots dilated image
figure;
imagesc(output_image)
colormap(gray);
title([image_file, ' dilated by ', se_file]);
imwrite(output_image,gray(256),'dilated_image.bmp','bmp');
```

conv_3x3.m

```
function [output_image,output_image_8] = conv_3x3(image_file);
%
%   filename:  conv_3x3.m
%   author:    Tony Nelson
%   date:      1/20/00
%   detail:    performs 3x3 convolution with specified kernel
%

K = [1 2 1;...
     2 4 2;...
     1 2 1];

input_image = LoadImage(image_file);         % loads image into input_image
[ylength,xlength] = size(input_image);       % determines size of input image
output_image(1:ylength,1:xlength) = zeros;   %inits output_image
output_image_8(1:ylength,1:xlength) = zeros; %inits output_image_8

% loops to simulate SE window passing over image
for y=1:ylength-2
    for x=1:xlength-2
      window = [input_image(y:(y+2),x:(x+2))];
      mult = window.*K;
      mult_v = [[mult(1,1:3)] [mult(2,1:3)] [mult(3,1:3)]];
      add = sum(mult_v);
      output_image(y+1,x+1) = add/9;
      output_image_8(y+1,x+1) = add/8;
    end
end

%plots convolved image
```

```
figure;
imagesc(output_image)
colormap(gray(256));
title(['Convolution Operation Output']);

%plots convolved image
figure;
imagesc(output_image_8)
colormap(gray(256));
title(['Convolution Operation Output with shift divide']);
```

m2vhdl.m

```
function m2vhdl(input_bmp,output_bin);
%   filename:   m2vhdl.m
%   author:     Tony Nelson
%   date:       1/21/00
%   detail:     a program to output a specified image to a stream of
%               integers for VHDL file input
%
%   parameters: input_bmp - file to convert to bin format
%               output_bin - file ready for vhdl file input

I = LoadImage(input_bmp);
J = int16(I);
K = double(J);
K = K';
M = reshape(K,128*128,1);

fid = fopen(output_bin,'wb');
fprintf(fid,'%d\n',M);
fclose(fid);
```

vhdl2m.m

```
function I = vhdl2m(input_bin);
%   filename:   vhdl2m.m
%   author:     Tony Nelson
%   date:       1/21/00
%   detail:     a program to read in the VHDL output file
%
%   paramter:   input_bin - vhdl output bin file
%

close all;
fid = fopen(input_bin);
[I,cnt] = fscanf(fid,'%d',inf);
fclose(fid);
I = reshape(I,128,128);
I = I';

originalI = LoadImage('d:/usr/nelson/courses/aip/elaine_128x128.bmp');
J = int16(originalI);
originalI = double(J);

figure;
imagesc(I);
title(input_bin);
Cmap = gray(256);
Colormap(Cmap);
```

APPENDIX B

VHDL SOURCE FILES

window_3x3.vhd

```
---------------------------------------------------------------------------
--      filename:       window_3x3.vhd
--      author:         Tony Nelson
--      date:           12/13/99
--
--      detail:         3x3 window generator
--
--      limits:         none
---------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity window_3x3 is
        generic (
                vwidth: integer:=8
                );
        port (
                Clk     : in std_logic;
                RSTn    : in std_logic;
                D       : in std_logic_vector(vwidth-1 downto 0);
                w11     : out std_logic_vector(vwidth-1 downto 0);
                w12     : out std_logic_vector(vwidth-1 downto 0);
                w13     : out std_logic_vector(vwidth-1 downto 0);
                w21     : out std_logic_vector(vwidth-1 downto 0);
                w22     : out std_logic_vector(vwidth-1 downto 0);
                w23     : out std_logic_vector(vwidth-1 downto 0);
                w31     : out std_logic_vector(vwidth-1 downto 0);
                w32     : out std_logic_vector(vwidth-1 downto 0);
                w33     : out std_logic_vector(vwidth-1 downto 0);
                DV      : out std_logic:='0'
        );
end window_3x3;

architecture window_3x3 of window_3x3 is

        component fifo_128x8u
        PORT
        (
                data            : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                wrreq           : IN STD_LOGIC ;
                rdreq           : IN STD_LOGIC ;
                clock           : IN STD_LOGIC ;
                aclr            : IN STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
                full            : OUT STD_LOGIC ;
                empty           : OUT STD_LOGIC ;
                usedw           : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)
        );
        END component fifo_128x8u;

        signal a11      : std_logic_vector(vwidth-1 downto 0);
        signal a12      : std_logic_vector(vwidth-1 downto 0);
        signal a13      : std_logic_vector(vwidth-1 downto 0);
        signal a21      : std_logic_vector(vwidth-1 downto 0);
        signal a22      : std_logic_vector(vwidth-1 downto 0);
        signal a23      : std_logic_vector(vwidth-1 downto 0);
        signal a31      : std_logic_vector(vwidth-1 downto 0);
        signal a32     : std_logic_vector(vwidth-1 downto 0);
        signal a33     : std_logic_vector(vwidth-1 downto 0);

        --fifoa signals
        signal clear   : std_logic;
        signal wrreqa  : std_logic:='1';
        signal rdreqa  : std_logic:='0';
        signal ofulla  : std_logic;
        signal oemptya : std_logic;
        signal ofifoa  : std_logic_vector(vwidth-1 downto 0);
        signal ousedwa : std_logic_vector(vwidth-2 downto 0);
        --fifob signals
        signal wrreqb  : std_logic:='0';
        signal rdreqb  : std_logic:='0';
        signal ofullb  : std_logic;
        signal oemptyb : std_logic;
        signal ofifob  : std_logic_vector(vwidth-1 downto 0);
        signal ousedwb : std_logic_vector(vwidth-2 downto 0);

        signal dwrreqb: std_logic:='0';

        -- signals for DV coordination
        signal dddddddddDV: std_logic:='0';
        signal ddddddddDV: std_logic;
        signal dddddddDV: std_logic;
        signal ddddddDV: std_logic;
        signal dddddDV: std_logic;
        signal ddddDV: std_logic;
        signal dddDV: std_logic;
        signal ddDV: std_logic;
        signal dDV: std_logic;

begin

        fifoa: fifo_128x8u
                port map (
                        data    => a13,
                        wrreq   => wrreqa,
                        rdreq   => rdreqa,
                        clock   => Clk,
                        aclr    => clear,
                        q       => ofifoa,
                        full    => ofulla,
                        empty   => oemptya,
                        usedw   => ousedwa
                );

        fifob: fifo_128x8u
                port map (
                        data    => a23,
                        wrreq   => wrreqb,
                        rdreq   => rdreqb,
                        clock   => Clk,
                        aclr    => clear,
                        q       => ofifob,
                        full    => ofullb,
                        empty   => oemptyb,
                        usedw   => ousedwb
                );

        clear <= not(RSTn);

        clock: process(Clk,RSTn)
        begin
                if RSTn = '0' then
                        a11 <= (others=>'0');
                        a12 <= (others=>'0');
                        a13 <= (others=>'0');
                        a21 <= (others=>'0');
                        a22 <= (others=>'0');
                        a23 <= (others=>'0');
```

```
			a31 <= (others=>'0');
			a32 <= (others=>'0');
			a33 <= (others=>'0');

			w11 <= (others=>'0');
			w12 <= (others=>'0');
			w13 <= (others=>'0');
			w21 <= (others=>'0');
			w22 <= (others=>'0');
			w23 <= (others=>'0');
			w31 <= (others=>'0');
			w32 <= (others=>'0');
			w33 <= (others=>'0');

			wrreqa <= '0';
			wrreqb <= '0';

			ddddddddDV <= '0';
			dddddddDV <= '0';
			ddddddDV <= '0';
			dddddDV <= '0';
			ddddDV <= '0';
			dddDV <= '0';
			ddDV <= '0';
			dDV <= '0';
			DV <= '0';
		elsif rising_edge(Clk) then
			a11 <= D;
			a12 <= a11;
			a13 <= a12;
			a21 <= ofifoa;
			a22 <= a21;
			a23 <= a22;
			a31 <= ofifob;
			a32 <= a31;
			a33 <= a32;

			w11 <= a11;
			w12 <= a12;
			w13 <= a13;
			w21 <= a21;
			w22 <= a22;
			w23 <= a23;
			w31 <= a31;
			w32 <= a32;
			w33 <= a33;

			wrreqa <= '1';
			wrreqb <= dwrreqb;

			ddddddddDV <= dddddddddDV;
			dddddddDV <= ddddddddDV;
			ddddddDV <= dddddddDV;
			dddddDV <= ddddddDV;
			ddddDV <= dddddDV;
			dddDV <= ddddDV;
			ddDV <= dddDV;
			dDV <= ddDV;
			DV <= dDV;
		end if;
	end process;

	req: process(Clk)
	begin
	if rising_edge(Clk) then
		if ousedwa = "1111011" then
			rdreqa <= '1';
			dwrreqb <= '1';
		end if;
		if ousedwb = "1111011" then
			rdreqb <= '1';
```

```
                elsif ousedwb = "1111100" then
                        dddddddddDV <= '1';
                end if;
        end if;
        end process;

end window_3x3;
```

window_3x3_x.vhd

```
---------------------------------------------------------------------------
--      filename:       window_3x3_x.vhd
--      author:         Tony Nelson
--      date:           1/13/99
--
--      detail:         3x3 window generator for Xilinx
--
--      limits:         none
---------------------------------------------------------------------------

Library XilinxCoreLib;
use xilinxcorelib.ul_utils.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity window_3x3 is
        generic (
                vwidth: integer:=8
                );
        port (
                Clk     : in std_logic;
                RSTn    : in std_logic;
                D       : in std_logic_vector(vwidth-1 downto 0);
                w11     : out std_logic_vector(vwidth-1 downto 0);
                w12     : out std_logic_vector(vwidth-1 downto 0);
                w13     : out std_logic_vector(vwidth-1 downto 0);
                w21     : out std_logic_vector(vwidth-1 downto 0);
                w22     : out std_logic_vector(vwidth-1 downto 0);
                w23     : out std_logic_vector(vwidth-1 downto 0);
                w31     : out std_logic_vector(vwidth-1 downto 0);
                w32     : out std_logic_vector(vwidth-1 downto 0);
                w33     : out std_logic_vector(vwidth-1 downto 0);
                DV      : out std_logic:='0'
        );
end window_3x3;

architecture window_3x3 of window_3x3 is

component fifo_128x8x
        port (
        din     : IN std_logic_VECTOR(7 downto 0);
        wr_en   : IN std_logic;
        wr_clk  : IN std_logic;
        rd_en   : IN std_logic;
        rd_clk  : IN std_logic;
        ainit   : IN std_logic;
        dout    : OUT std_logic_VECTOR(7 downto 0);
        full    : OUT std_logic;
        empty   : OUT std_logic;
        wr_count: OUT std_logic_VECTOR(6 downto 0));
end component;

        for all : fifo_128x8x use entity XilinxCoreLib.async_fifo_v1_0(behavioral)
                generic map(
                        c_wr_err_low => 0,
                        c_has_rd_count => 0,
                        c_has_rd_ack => 0,
                        c_wr_ack_low => 0,
                        c_has_wr_count => 1,
                        c_has_wr_ack => 0,
                        c_has_almost_full => 0,
                        c_has_almost_empty => 0,
                        c_wr_count_width => 7,
                        c_rd_count_width => 2,
                        c_has_rd_err => 0,
                        c_data_width => 8,
                        c_has_wr_err => 0,
                        c_rd_ack_low => 0,
                        c_rd_err_low => 0,
                        c_fifo_depth => 127,
                        c_enable_rlocs => 0,
                        c_use_blockmem => 1);

        signal a11     : std_logic_vector(vwidth-1 downto 0);
        signal a12     : std_logic_vector(vwidth-1 downto 0);
        signal a13     : std_logic_vector(vwidth-1 downto 0);
        signal a21     : std_logic_vector(vwidth-1 downto 0);
        signal a22     : std_logic_vector(vwidth-1 downto 0);
        signal a23     : std_logic_vector(vwidth-1 downto 0);
        signal a31     : std_logic_vector(vwidth-1 downto 0);
        signal a32     : std_logic_vector(vwidth-1 downto 0);
        signal a33     : std_logic_vector(vwidth-1 downto 0);

        --fifoa signals
        signal clear   : std_logic;
        signal wrreqa  : std_logic:='1';
        signal rdreqa  : std_logic:='0';
        signal ofulla  : std_logic;
        signal oemptya : std_logic;
        signal ofifoa  : std_logic_vector(vwidth-1 downto 0);
        signal ousedwa : std_logic_vector(6 downto 0);
        --fifob signals
        signal wrreqb  : std_logic:='0';
        signal rdreqb  : std_logic:='0';
        signal ofullb  : std_logic;
        signal oemptyb : std_logic;
        signal ofifob  : std_logic_vector(vwidth-1 downto 0);
        signal ousedwb : std_logic_vector(6 downto 0);

        signal dwrreqb: std_logic:='0';

        -- signals for DV coordination
        signal ddddddddDV: std_logic:='0';
        signal dddddddDV: std_logic;
        signal ddddddDV: std_logic;
        signal dddddDV: std_logic;
        signal ddddDV: std_logic;
        signal dddDV: std_logic;
        signal ddDV: std_logic;
        signal dDV: std_logic;

        signal ousedwa_temp: integer:=0;
        signal ousedwb_temp: integer:=0;

begin

        fifoa: fifo_128x8x
                port map (
                        din     => a13,
                        wr_en   => wrreqa,
                        wr_clk  => Clk,
                        rd_en   => rdreqa,
                        rd_clk  => Clk,
                        ainit   => clear,
                        dout    => ofifoa,
                        full    => ofulla,
                        empty   => oemptya,
                        wr_count => ousedwa
                );
```

```
fifob: fifo_128x8x
        port map (
                din     => a23,
                wr_en   => wrreqb,
                wr_clk  => Clk,
                rd_en   => rdreqb,
                rd_clk  => Clk,
                ainit   => clear,
                dout    => ofifob,
                full    => ofullb,
                empty   => oemptyb,
                wr_count => ousedwb
        );

clear <= not(RSTn);

clock: process(Clk,RSTn)
begin
        if RSTn = '0' then
                a11 <= (others=>'0');
                a12 <= (others=>'0');
                a13 <= (others=>'0');
                a21 <= (others=>'0');
                a22 <= (others=>'0');
                a23 <= (others=>'0');
                a31 <= (others=>'0');
                a32 <= (others=>'0');
                a33 <= (others=>'0');

                w11 <= (others=>'0');
                w12 <= (others=>'0');
                w13 <= (others=>'0');
                w21 <= (others=>'0');
                w22 <= (others=>'0');
                w23 <= (others=>'0');
                w31 <= (others=>'0');
                w32 <= (others=>'0');
                w33 <= (others=>'0');

                wrreqa <= '0';
                wrreqb <= '0';

                dddddddDV <= '0';
                ddddddDV <= '0';
                dddddDV <= '0';
                ddddDV <= '0';
                dddDV <= '0';
                ddDV <= '0';
                dDV <= '0';
                DV <= '0';
        elsif rising_edge(Clk) then
                a11 <= D;
                a12 <= a11;
                a13 <= a12;
                a21 <= ofifoa;
                a22 <= a21;
                a23 <= a22;
                a31 <= ofifob;
                a32 <= a31;
                a33 <= a32;

                w11 <= a11;
                w12 <= a12;
                w13 <= a13;
                w21 <= a21;
                w22 <= a22;
                w23 <= a23;
                w31 <= a31;
                w32 <= a32;
                w33 <= a33;
```

```
                        wrreqa <= '1';
                        wrreqb <= dwrreqb;

                        ddddddddDV <= dddddddddDV;
                        dddddddDV <= ddddddddDV;
                        ddddddDV <= dddddddDV;
                        dddddDV <= ddddddDV;
                        ddddDV <= dddddDV;
                        dddDV <= ddddDV;
                        ddDV <= dddDV;
                        dDV <= ddDV;
                        DV <= dDV;
                end if;
        end process;

        req: process(Clk)
        begin
        if rising_edge(Clk) then
                if ousedwa = "1111011" then
                        rdreqa <= '1';
                        dwrreqb <= '1';
                end if;
                if ousedwb = "1111011" then
                        rdreqb <= '1';
                        dddddddddDV <= '1';
                end if;
        end if;
        end process;
end window_3x3;
```

ro_filt_3x3_TB.vhd

```
---------------------------------------------------------------------------
--      filename:       ro_filt_3x3_TB.vhd
--      author:         Tony Nelson
--      date:           1/24/00
--
--      detail:         TestBench for ro_filt_3x3
--                      reads image data from specified file and writes processed
--                      data to vhdl_output.bin
--                      To use this functionality, use the following method for
--                      determining simulation length:
--
--                      t_valid = time when output data first becomes valid
--                      t_delay = t_valid - 5 ns
--                      t_sim_stop = 163835 ns + t_delay + 10 ns
--                      this is 165305ns for this entity
--
--      limits:         none
---------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use std.textio.all;

entity ro_filt_3x3_tb is
        generic(
                vwidth  : INTEGER := 8;
                order   : INTEGER := 4;
                num_cols : INTEGER := 128;
                num_rows : INTEGER := 128 );
end ro_filt_3x3_tb;

architecture TB_ARCHITECTURE of ro_filt_3x3_tb is
        component ro_filt_3x3
        generic(
                vwidth  : INTEGER := 8;
                order   : INTEGER := 4;
                num_cols : INTEGER := 128;
```

```
		num_rows : INTEGER := 128 );
	port(
		Clk	: in std_logic;
		RSTn	: in std_logic;
		D	: in std_logic_vector((vwidth-1) downto 0);
		Dout	: out std_logic_vector((vwidth-1) downto 0);
		DV	: out std_logic );
	end component;

	signal Clk	: std_logic;
	signal RSTn	: std_logic;
	signal D	: std_logic_vector((vwidth-1) downto 0);

	signal Dout	: std_logic_vector((vwidth-1) downto 0);
	signal DV	: std_logic;

begin

	UUT : ro_filt_3x3
		port map
			(Clk	=> Clk,
			RSTn	=> RSTn,
			D	=> D,
			Dout	=> Dout,
			DV	=> DV );

	read_from_file: process(Clk)
		variable indata_line: line;
		variable indata: integer;
		file input_data_file: text open read_mode is "elaine_128x128.bin";
	begin
		if rising_edge(Clk) then
			readline(input_data_file,indata_line);
			read(indata_line,indata);
			D <= conv_std_logic_vector(indata,8);
			if endfile(input_data_file) then
				report "end of file -- looping back to start of file";
				file_close(input_data_file);
				file_open(input_data_file,"elaine_128x128.bin");
			end if;
		end if;

	end process;

	write_to_file: process(Clk)
		variable outdata_line: line;
		variable outdata: integer:=0;
		file output_data_file: text open write_mode is "vhdl_output.bin";
	begin
		if rising_edge(Clk) then
			outdata := CONV_INTEGER(unsigned(Dout));
			if DV = '1' then
				write(outdata_line,outdata);
				writeline(output_data_file,outdata_line);
			end if;
		end if;
	end process;

	clock_gen: process
	begin
		Clk <= '0';
		wait for 5 ns;
		Clk <= '1';
		wait for 5 ns;
	end process;

	reset_gen: process
	begin
		RSTn <= '0';
		wait for 10 ns;
		RSTn <= '1';
```

```
                wait;
        end process;

end TB_ARCHITECTURE;

configuration TESTBENCH_FOR_ro_filt_3x3 of ro_filt_3x3_tb is
        for TB_ARCHITECTURE
                for UUT : ro_filt_3x3
                        use entity work.ro_filt_3x3(ro_filt_3x3);
                end for;
        end for;
end TESTBENCH_FOR_ro_filt_3x3;
```

sort_3x3.vhd

```
---------------------------------------------------------------------------
--      filename:       sort_3x3.vhd
--      author:         Tony Nelson
--      date:           12/15/99
--
--      detail:         3x3 sorting algorithm.  sorts input 3x3 window to output
--                      vectors from lowest to highest.  s1 <= L, s5 <= M, S <= H.
--
--      limits:         none
---------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity sort_3x3 is
        generic (
                vwidth: integer:=8
                );
        port (
                Clk     : in std_logic;
                RSTn    : in std_logic;
                w11     : in std_logic_vector((vwidth-1) downto 0);
                w12     : in std_logic_vector((vwidth-1) downto 0);
                w13     : in std_logic_vector((vwidth-1) downto 0);
                w21     : in std_logic_vector((vwidth-1) downto 0);
                w22     : in std_logic_vector((vwidth-1) downto 0);
                w23     : in std_logic_vector((vwidth-1) downto 0);
                w31     : in std_logic_vector((vwidth-1) downto 0);
                w32     : in std_logic_vector((vwidth-1) downto 0);
                w33     : in std_logic_vector((vwidth-1) downto 0);
                DVw     : in std_logic;
                DVs     : out std_logic;
                s1      : out std_logic_vector(vwidth-1 downto 0);
                s2      : out std_logic_vector(vwidth-1 downto 0);
                s3      : out std_logic_vector(vwidth-1 downto 0);
                s4      : out std_logic_vector(vwidth-1 downto 0);
                s5      : out std_logic_vector(vwidth-1 downto 0);
                s6      : out std_logic_vector(vwidth-1 downto 0);
                s7      : out std_logic_vector(vwidth-1 downto 0);
                s8      : out std_logic_vector(vwidth-1 downto 0);
                s9      : out std_logic_vector(vwidth-1 downto 0)
        );
end sort_3x3;

architecture sort_3x3 of sort_3x3 is

        -- compare signals
        signal c11_L: std_logic_vector((vwidth-1) downto 0);
        signal c11_H: std_logic_vector((vwidth-1) downto 0);
        signal c12_L: std_logic_vector((vwidth-1) downto 0);
        signal c12_H: std_logic_vector((vwidth-1) downto 0);
        signal c13_L: std_logic_vector((vwidth-1) downto 0);
        signal c13_H: std_logic_vector((vwidth-1) downto 0);
        signal c14_L: std_logic_vector((vwidth-1) downto 0);
signal c14_H: std_logic_vector((vwidth-1) downto 0);
signal c21_L: std_logic_vector((vwidth-1) downto 0);
signal c21_H: std_logic_vector((vwidth-1) downto 0);
signal c22_L: std_logic_vector((vwidth-1) downto 0);
signal c22_H: std_logic_vector((vwidth-1) downto 0);
signal c23_L: std_logic_vector((vwidth-1) downto 0);
signal c23_H: std_logic_vector((vwidth-1) downto 0);
signal c24_L: std_logic_vector((vwidth-1) downto 0);
signal c24_H: std_logic_vector((vwidth-1) downto 0);
signal c31_L: std_logic_vector((vwidth-1) downto 0);
signal c31_H: std_logic_vector((vwidth-1) downto 0);
signal c32_L: std_logic_vector((vwidth-1) downto 0);
signal c32_H: std_logic_vector((vwidth-1) downto 0);
signal c33_L: std_logic_vector((vwidth-1) downto 0);
signal c33_H: std_logic_vector((vwidth-1) downto 0);
signal c34_L: std_logic_vector((vwidth-1) downto 0);
signal c34_H: std_logic_vector((vwidth-1) downto 0);
signal c41_L: std_logic_vector((vwidth-1) downto 0);
signal c41_H: std_logic_vector((vwidth-1) downto 0);
signal c42_L: std_logic_vector((vwidth-1) downto 0);
signal c42_H: std_logic_vector((vwidth-1) downto 0);
signal c43_L: std_logic_vector((vwidth-1) downto 0);
signal c43_H: std_logic_vector((vwidth-1) downto 0);
signal c4a1_L: std_logic_vector((vwidth-1) downto 0);
signal c4a1_H: std_logic_vector((vwidth-1) downto 0);
signal c4a2_L: std_logic_vector((vwidth-1) downto 0);
signal c4a2_H: std_logic_vector((vwidth-1) downto 0);
signal c4b0_L: std_logic_vector((vwidth-1) downto 0);
signal c4b0_H: std_logic_vector((vwidth-1) downto 0);
signal c4b1_L: std_logic_vector((vwidth-1) downto 0);
signal c4b1_H: std_logic_vector((vwidth-1) downto 0);
signal c4b2_L: std_logic_vector((vwidth-1) downto 0);
signal c4b2_H: std_logic_vector((vwidth-1) downto 0);
signal c51_L: std_logic_vector((vwidth-1) downto 0);
signal c51_H: std_logic_vector((vwidth-1) downto 0);
signal c61_L: std_logic_vector((vwidth-1) downto 0);
signal c61_H: std_logic_vector((vwidth-1) downto 0);
signal c71_L: std_logic_vector((vwidth-1) downto 0);
signal c71_H: std_logic_vector((vwidth-1) downto 0);
signal c81_L: std_logic_vector((vwidth-1) downto 0);
signal c81_H: std_logic_vector((vwidth-1) downto 0);
signal c91_L: std_logic_vector((vwidth-1) downto 0);
signal c91_H: std_logic_vector((vwidth-1) downto 0);
signal c101_L: std_logic_vector((vwidth-1) downto 0);
signal c101_H: std_logic_vector((vwidth-1) downto 0);
signal c111_L: std_logic_vector((vwidth-1) downto 0);
signal c111_H: std_logic_vector((vwidth-1) downto 0);

-- register signals
signal r11: std_logic_vector((vwidth-1) downto 0);
signal r21: std_logic_vector((vwidth-1) downto 0);
signal r31: std_logic_vector((vwidth-1) downto 0);
signal r41: std_logic_vector((vwidth-1) downto 0);
signal r42: std_logic_vector((vwidth-1) downto 0);
signal r43: std_logic_vector((vwidth-1) downto 0);
signal r4a1: std_logic_vector((vwidth-1) downto 0);
signal r4a2: std_logic_vector((vwidth-1) downto 0);
signal r4a3: std_logic_vector((vwidth-1) downto 0);
signal r4a4: std_logic_vector((vwidth-1) downto 0);
signal r4a5: std_logic_vector((vwidth-1) downto 0);
signal r4b1: std_logic_vector((vwidth-1) downto 0);
signal r4b4: std_logic_vector((vwidth-1) downto 0);
signal r4b5: std_logic_vector((vwidth-1) downto 0);
signal r51: std_logic_vector((vwidth-1) downto 0);
signal r52: std_logic_vector((vwidth-1) downto 0);
signal r53: std_logic_vector((vwidth-1) downto 0);
signal r54: std_logic_vector((vwidth-1) downto 0);
signal r55: std_logic_vector((vwidth-1) downto 0);
signal r56: std_logic_vector((vwidth-1) downto 0);
signal r57: std_logic_vector((vwidth-1) downto 0);
signal r61: std_logic_vector((vwidth-1) downto 0);
```

```
        signal r62: std_logic_vector((vwidth-1) downto 0);
        signal r63: std_logic_vector((vwidth-1) downto 0);
        signal r64: std_logic_vector((vwidth-1) downto 0);
        signal r65: std_logic_vector((vwidth-1) downto 0);
        signal r66: std_logic_vector((vwidth-1) downto 0);
        signal r67: std_logic_vector((vwidth-1) downto 0);
        signal r71: std_logic_vector((vwidth-1) downto 0);
        signal r72: std_logic_vector((vwidth-1) downto 0);
        signal r73: std_logic_vector((vwidth-1) downto 0);
        signal r74: std_logic_vector((vwidth-1) downto 0);
        signal r75: std_logic_vector((vwidth-1) downto 0);
        signal r76: std_logic_vector((vwidth-1) downto 0);
        signal r77: std_logic_vector((vwidth-1) downto 0);
        signal r81: std_logic_vector((vwidth-1) downto 0);
        signal r82: std_logic_vector((vwidth-1) downto 0);
        signal r83: std_logic_vector((vwidth-1) downto 0);
        signal r84: std_logic_vector((vwidth-1) downto 0);
        signal r85: std_logic_vector((vwidth-1) downto 0);
        signal r86: std_logic_vector((vwidth-1) downto 0);
        signal r87: std_logic_vector((vwidth-1) downto 0);
        signal r91: std_logic_vector((vwidth-1) downto 0);
        signal r92: std_logic_vector((vwidth-1) downto 0);
        signal r93: std_logic_vector((vwidth-1) downto 0);
        signal r94: std_logic_vector((vwidth-1) downto 0);
        signal r95: std_logic_vector((vwidth-1) downto 0);
        signal r96: std_logic_vector((vwidth-1) downto 0);
        signal r97: std_logic_vector((vwidth-1) downto 0);
        signal r101: std_logic_vector((vwidth-1) downto 0);
        signal r102: std_logic_vector((vwidth-1) downto 0);
        signal r103: std_logic_vector((vwidth-1) downto 0);
        signal r104: std_logic_vector((vwidth-1) downto 0);
        signal r105: std_logic_vector((vwidth-1) downto 0);
        signal r106: std_logic_vector((vwidth-1) downto 0);
        signal r107: std_logic_vector((vwidth-1) downto 0);
        signal r111: std_logic_vector((vwidth-1) downto 0);
        signal r112: std_logic_vector((vwidth-1) downto 0);
        signal r113: std_logic_vector((vwidth-1) downto 0);
        signal r114: std_logic_vector((vwidth-1) downto 0);
        signal r115: std_logic_vector((vwidth-1) downto 0);
        signal r116: std_logic_vector((vwidth-1) downto 0);
        signal r117: std_logic_vector((vwidth-1) downto 0);

        -- signals for DV coordination
        signal dddddddddddddDV: std_logic:='0';
        signal ddddddddddddDV: std_logic;
        signal dddddddddddDV: std_logic;
        signal ddddddddddDV: std_logic;
        signal dddddddddDV: std_logic;
        signal ddddddddDV: std_logic;
        signal dddddddDV: std_logic;
        signal ddddddDV: std_logic;
        signal dddddDV: std_logic;
        signal ddddDV: std_logic;
        signal dddDV: std_logic;
        signal ddDV: std_logic;
        signal dDV: std_logic;

begin

        process(Clk,RSTn)
        begin
                if RSTn = '0' then
                        c11_L <= (others=>'0');
                        c11_H <= (others=>'0');
                        c12_L <= (others=>'0');
                        c12_H <= (others=>'0');
                        c13_L <= (others=>'0');
                        c13_H <= (others=>'0');
                        c14_L <= (others=>'0');
                        c14_H <= (others=>'0');
                        c21_L <= (others=>'0');
```

```
c21_H <= (others=>'0');
c22_L <= (others=>'0');
c22_H <= (others=>'0');
c23_L <= (others=>'0');
c23_H <= (others=>'0');
c24_L <= (others=>'0');
c24_H <= (others=>'0');
c31_L <= (others=>'0');
c31_H <= (others=>'0');
c32_L <= (others=>'0');
c32_H <= (others=>'0');
c33_L <= (others=>'0');
c33_H <= (others=>'0');
c34_L <= (others=>'0');
c34_H <= (others=>'0');
c41_L <= (others=>'0');
c41_H <= (others=>'0');
c42_L <= (others=>'0');
c42_H <= (others=>'0');
c43_L <= (others=>'0');
c43_H <= (others=>'0');
c4a1_L <= (others=>'0');
c4a1_H <= (others=>'0');
c4a2_L <= (others=>'0');
c4a2_H <= (others=>'0');
c4b0_L <= (others=>'0');
c4b0_H <= (others=>'0');
c4b1_L <= (others=>'0');
c4b1_H <= (others=>'0');
c4b2_L <= (others=>'0');
c4b2_H <= (others=>'0');
c51_L <= (others=>'0');
c51_H <= (others=>'0');
c61_L <= (others=>'0');
c61_H <= (others=>'0');
c71_L <= (others=>'0');
c71_H <= (others=>'0');
c81_L <= (others=>'0');
c81_H <= (others=>'0');
c91_L <= (others=>'0');
c91_H <= (others=>'0');
c101_L <= (others=>'0');
c101_H <= (others=>'0');
c111_L <= (others=>'0');
c111_H <= (others=>'0');
r11 <= (others=>'0');
r21 <= (others=>'0');
r31 <= (others=>'0');
r41 <= (others=>'0');
r42 <= (others=>'0');
r43 <= (others=>'0');
r4a1 <= (others=>'0');
r4a2 <= (others=>'0');
r4a3 <= (others=>'0');
r4a4 <= (others=>'0');
r4a5 <= (others=>'0');
r4b1 <= (others=>'0');
r4b4 <= (others=>'0');
r4b5 <= (others=>'0');
r51 <= (others=>'0');
r52 <= (others=>'0');
r53 <= (others=>'0');
r54 <= (others=>'0');
r55 <= (others=>'0');
r56 <= (others=>'0');
r57 <= (others=>'0');
r61 <= (others=>'0');
r62 <= (others=>'0');
r63 <= (others=>'0');
r64 <= (others=>'0');
r65 <= (others=>'0');
```

```
                r66 <= (others=>'0');
                r67 <= (others=>'0');
                r71 <= (others=>'0');
                r72 <= (others=>'0');
                r73 <= (others=>'0');
                r74 <= (others=>'0');
                r75 <= (others=>'0');
                r76 <= (others=>'0');
                r77 <= (others=>'0');
                r81 <= (others=>'0');
                r82 <= (others=>'0');
                r83 <= (others=>'0');
                r84 <= (others=>'0');
                r85 <= (others=>'0');
                r86 <= (others=>'0');
                r87 <= (others=>'0');
                r91 <= (others=>'0');
                r92 <= (others=>'0');
                r93 <= (others=>'0');
                r94 <= (others=>'0');
                r95 <= (others=>'0');
                r96 <= (others=>'0');
                r97 <= (others=>'0');
                r101 <= (others=>'0');
                r102 <= (others=>'0');
                r103 <= (others=>'0');
                r104 <= (others=>'0');
                r105 <= (others=>'0');
                r106 <= (others=>'0');
                r107 <= (others=>'0');
                r111 <= (others=>'0');
                r112 <= (others=>'0');
                r113 <= (others=>'0');
                r114 <= (others=>'0');
                r115 <= (others=>'0');
                r116 <= (others=>'0');
                r117 <= (others=>'0');
                s1 <= (others=>'0');
                s2 <= (others=>'0');
                s3 <= (others=>'0');
                s4 <= (others=>'0');
                s5 <= (others=>'0');
                s6 <= (others=>'0');
                s7 <= (others=>'0');
                s8 <= (others=>'0');
                s9 <= (others=>'0');
                ddddddddddddDV <= '0';
                dddddddddddDV <= '0';
                ddddddddddDV <= '0';
                dddddddddDV <= '0';
                ddddddddDV <= '0';
                dddddddDV <= '0';
                ddddddDV <= '0';
                dddddDV <= '0';
                ddddDV <= '0';
                dddDV <= '0';
                ddDV <= '0';
                dDV <= '0';
                DVs <= '0';
        elsif rising_edge(Clk) then
                if DVw = '1' then
                        -- level 1
                        if w11 < w12 then
                                c11_L <= w11;
                                c11_H <= w12;
                        else
                                c11_L <= w12;
                                c11_H <= w11;
                        end if;
                        if w13 < w21 then
                                c12_L <= w13;
        c12_H <= w21;
else
        c12_L <= w21;
        c12_H <= w13;
end if;
if w22 < w23 then
        c13_L <= w22;
        c13_H <= w23;
else
        c13_L <= w23;
        c13_H <= w22;
end if;
if w31 < w32 then
        c14_L <= w31;
        c14_H <= w32;
else
        c14_L <= w32;
        c14_H <= w31;
end if;
r11 <= w33;
-- level 2
if c11_L < c12_L then
        c21_L <= c11_L;
        c21_H <= c12_L;
else
        c21_L <= c12_L;
        c21_H <= c11_L;
end if;
if c11_H < c12_H then
        c22_L <= c11_H;
        c22_H <= c12_H;
else
        c22_L <= c12_H;
        c22_H <= c11_H;
end if;
if c13_L < c14_L then
        c23_L <= c13_L;
        c23_H <= c14_L;
else
        c23_L <= c14_L;
        c23_H <= c13_L;
end if;
if c13_H < c14_H then
        c24_L <= c13_H;
        c24_H <= c14_H;
else
        c24_L <= c14_H;
        c24_H <= c13_H;
end if;
r21 <= r11;
-- level 3
if c21_L < c23_L then
        c31_L <= c21_L;
        c31_H <= c23_L;
else
        c31_L <= c23_L;
        c31_H <= c21_L;
end if;
if c21_H < c23_H then
        c32_L <= c21_H;
        c32_H <= c23_H;
else
        c32_L <= c23_H;
        c32_H <= c21_H;
end if;
if c22_L < c24_L then
        c33_L <= c22_L;
        c33_H <= c24_L;
else
        c33_L <= c24_L;
        c33_H <= c22_L;
```

```
end if;
if c22_H < c24_H then
        c34_L <= c22_H;
        c34_H <= c24_H;
else
        c34_L <= c24_H;
        c34_H <= c22_H;
end if;
r31 <= r21;
-- level 4
r41 <= c31_L;
if c31_H < c32_L then
        c41_L <= c31_H;
        c41_H <= c32_L;
else
        c41_L <= c32_L;
        c41_H <= c31_H;
end if;
if c32_H < c33_L then
        c42_L <=       c32_H;
        c42_H <= c33_L;
else
        c42_L <=       c33_L;
        c42_H <= c32_H;
end if;
if c33_H < c34_L then
        c43_L <= c33_H;
        c43_H <= c34_L;
else
        c43_L <= c34_L;
        c43_H <= c33_H;
end if;
r42 <= c34_H;
r43 <= r31;
-- level 4a
r4a1 <= r41;
if c41_L < c42_H then
        c4a1_L <= c41_L;
        c4a1_H <= c42_H;
else
        c4a1_L <= c42_H;
        c4a1_H <= c41_L;
end if;
if c41_H < c42_L then
        c4a2_L <= c41_H;
        c4a2_H <= c42_L;
else
        c4a2_L <= c42_L;
        c4a2_H <= c41_H;
end if;
r4a2 <= c43_L;
r4a3 <= c43_H;
r4a4 <= r42;
r4a5 <= r43;
-- level 4b
r4b1 <= r4a1;
if c4a1_L < c4a2_L then
        c4b0_L <= c4a1_L;
        c4b0_H <= c4a2_L;
else
        c4b0_L <= c4a2_L;
        c4b0_H <= c4a1_L;
end if;
if c4a2_H < r4a2 then
        c4b1_L <= c4a2_H;
        c4b1_H <= r4a2;
else
        c4b1_L <= r4a2;
        c4b1_H <= c4a2_H;
end if;
if c4a1_H < r4a3 then
```

```
        c4b2_L <= c4a1_H;
        c4b2_H <= r4a3;
else
        c4b2_L <= r4a3;
        c4b2_H <= c4a1_H;
end if;
r4b4 <= r4a4;
r4b5 <= r4a5;
-- level 5
if r4b1 < r4b5 then
        c51_L <= r4b1;
        c51_H <= r4b5;
else
        c51_L <= r4b5;
        c51_H <= r4b1;
end if;
r51 <= c4b0_L;
r52 <= c4b0_H;
r53 <= c4b1_L;
r54 <= c4b1_H;
r55 <= c4b2_L;
r56 <= c4b2_H;
r57 <= r4b4;
-- level 6
if r51 < c51_H then
        c61_L <= r51;
        c61_H <= c51_H;
else
        c61_L <= c51_H;
        c61_H <= r51;
end if;
r61 <= c51_L;           -- L
r62 <= r52;
r63 <= r53;
r64 <= r54;
r65 <= r55;
r66 <= r56;
r67 <= r57;
-- level 7
if r62 < c61_H then
        c71_L <= r62;
        c71_H <= c61_H;
else
        c71_L <= c61_H;
        c71_H <= r62;
end if;
r71 <= r61;             -- L
r72 <= c61_L;           -- 2L
r73 <= r63;
r74 <= r64;
r75 <= r65;
r76 <= r66;
r77 <= r67;
-- level 8
if r73 < c71_H then
        c81_L <= r73;
        c81_H <= c71_H;
else
        c81_L <= c71_H;
        c81_H <= r73;
end if;
r81 <= r71;             -- L
r82 <= r72;             -- 2L
r83 <= c71_L;           -- 3L
r84 <= r74;
r85 <= r75;
r86 <= r76;
r87 <= r77;
-- level 9
if r84 < c81_H then
        c91_L <= r84;
        c91_H <= c81_H;
else
        c91_L <= c81_H;
        c91_H <= r84;
end if;
r91 <= r81;            -- L
r92 <= r82;            -- 2L
r93 <= r83;            -- 3L
r94 <= c81_L;          -- 4L
r95 <= r85;
r96 <= r86;
r97 <= r87;
-- level 10
if r95 < c91_H then
        c101_L <= r95;
        c101_H <= c91_H;
else
        c101_L <= c91_H;
        c101_H <= r95;
end if;
r101 <= r91;           -- L
r102 <= r92;           -- 2L
r103 <= r93;           -- 3L
r104 <= r94;           -- 4L
r105 <= c91_L;         -- M
r106 <= r96;
r107 <= r97;
-- level 11
if r106 < c101_H then
        c111_L <= r106;
        c111_H <= c101_H;
else
        c111_L <= c101_H;
        c111_H <= r106;
end if;
r111 <= r101;          -- L
r112 <= r102;          -- 2L
r113 <= r103;          -- 3L
r114 <= r104;          -- 4L
r115 <= r105;          -- M
r116 <= c101_L;        -- 4L
r117 <= r107;
-- level 12
if r117 < c111_H then
        s8 <= r117;    -- 2H
        s9 <= c111_H;  -- H
else
        s8 <= c111_H;  -- 2H
        s9 <= r117;    -- H
end if;
s1 <= r111;            -- L
s2 <= r112;            -- 2L
s3 <= r113;            -- 3L
s4 <= r114;            -- 4L
s5 <= r115;            -- M
s6 <= r116;            -- 4H
s7 <= c111_L;          -- 3H

ddddddddddddDV <= dddddddddddddDV;
dddddddddddDV <= ddddddddddddDV;
ddddddddddDV <= dddddddddddDV;
dddddddddDV <= ddddddddddDV;
ddddddddDV <= dddddddddDV;
dddddddDV <= ddddddddDV;
ddddddDV <= dddddddDV;
dddddDV <= ddddddDV;
ddddDV <= dddddDV;
dddDV <= ddddDV;
ddDV <= dddDV;
dDV <= ddDV;
DVs <= dDV;
```

```
                        end if;
                        if DVw = '1' then
                                dddddddddddddDV <= '1';
                        end if;
                end if;
        end process;

end sort_3x3;
```

rc_counter.vhd

```
---------------------------------------------------------------------------
--      filename:       rc_counter.vhd
--      author:         Tony Nelson
--      date:           12/22/99
--
--      detail:         row/column counter
--
--      limits:         none
----------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity rc_counter is
        generic (
                num_cols: integer:=128;
                num_rows: integer:=128
                );
        port (
                Clk     : in std_logic;
                RSTn    : in std_logic;
                En      : in std_logic;
                ColPos  : out integer;
                RowPos  : out integer
        );
end rc_counter;

architecture rc_counter of rc_counter is

begin

        process(RSTn,Clk,En)
                variable ColPos_var: integer:=0;
                variable RowPos_var: integer:=0;
        begin
                if RSTn = '0' then
                        ColPos_var := -1;
                        ColPos <= 0;
                        RowPos_var := 0;
                        RowPos <= 0;
                elsif rising_edge(Clk) then
                        if En = '1' then
                                ColPos_var := ColPos_var +1;
                                if ColPos_var = num_cols then
                                        RowPos_var := RowPos_var +1;
                                        ColPos_var := 0;
                                        if RowPos_var = num_rows then
                                                RowPos_var := 0;
                                        end if;
                                end if;
                                ColPos <= ColPos_var;
                                RowPos <= RowPos_var;
                        end if;
                end if;
        end process;

end rc_counter;
```

ro_filt_3x3.vhd

```
---------------------------------------------------------------------------
--      filename:       ro_filt_3x3.vhd
--      author:         Tony Nelson
--      date:           12/21/99
--
--      detail:         3x3 Rank Order Filter.  Generic order sets filter order.
--                      order: integer:= 5 is a Median Filter.
--
--      limits:         none
---------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity ro_filt_3x3 is
        generic (
                vwidth: integer:=8;
                order: integer:=4;
                num_cols: integer:=128;
                num_rows: integer:=128
                );
        port (
                Clk     : in std_logic;
                RSTn    : in std_logic;
                D       : in std_logic_vector(vwidth-1 downto 0);
                Dout    : out std_logic_vector(vwidth-1 downto 0);
                DV      : out std_logic
        );
end ro_filt_3x3;

architecture ro_filt_3x3 of ro_filt_3x3 is

        component sort_3x3
        generic (
                vwidth: integer:=8
                );
        port (
                Clk     : in std_logic;
                RSTn    : in std_logic;
                w11     : in std_logic_vector((vwidth-1) downto 0);
                w12     : in std_logic_vector((vwidth-1) downto 0);
                w13     : in std_logic_vector((vwidth-1) downto 0);
                w21     : in std_logic_vector((vwidth-1) downto 0);
                w22     : in std_logic_vector((vwidth-1) downto 0);
                w23     : in std_logic_vector((vwidth-1) downto 0);
                w31     : in std_logic_vector((vwidth-1) downto 0);
                w32     : in std_logic_vector((vwidth-1) downto 0);
                w33     : in std_logic_vector((vwidth-1) downto 0);
                DVw     : in std_logic;
                DVs     : out std_logic;
                s1      : out std_logic_vector(vwidth-1 downto 0);
                s2      : out std_logic_vector(vwidth-1 downto 0);
                s3      : out std_logic_vector(vwidth-1 downto 0);
                s4      : out std_logic_vector(vwidth-1 downto 0);
                s5      : out std_logic_vector(vwidth-1 downto 0);
                s6      : out std_logic_vector(vwidth-1 downto 0);
                s7      : out std_logic_vector(vwidth-1 downto 0);
                s8      : out std_logic_vector(vwidth-1 downto 0);
                s9      : out std_logic_vector(vwidth-1 downto 0)
        );
        end component sort_3x3;

        signal w11: std_logic_vector((vwidth-1) downto 0);
        signal w12: std_logic_vector((vwidth-1) downto 0);
        signal w13: std_logic_vector((vwidth-1) downto 0);
        signal w21: std_logic_vector((vwidth-1) downto 0);
        signal w22: std_logic_vector((vwidth-1) downto 0);
        signal w23: std_logic_vector((vwidth-1) downto 0);
        signal w31: std_logic_vector((vwidth-1) downto 0);
        signal w32: std_logic_vector((vwidth-1) downto 0);
        signal w33: std_logic_vector((vwidth-1) downto 0);
        signal DVw: std_logic;
        signal DVs: std_logic;
        signal s1: std_logic_vector(vwidth-1 downto 0);
        signal s2: std_logic_vector(vwidth-1 downto 0);
        signal s3: std_logic_vector(vwidth-1 downto 0);
        signal s4: std_logic_vector(vwidth-1 downto 0);
        signal s5: std_logic_vector(vwidth-1 downto 0);
        signal s6: std_logic_vector(vwidth-1 downto 0);
        signal s7: std_logic_vector(vwidth-1 downto 0);
        signal s8: std_logic_vector(vwidth-1 downto 0);
        signal s9: std_logic_vector(vwidth-1 downto 0);

        component window_3x3
        generic (
                vwidth: integer:=8
                );
        port (
                Clk     : in std_logic;
                RSTn    : in std_logic;
                D       : in std_logic_vector(vwidth-1 downto 0);
                w11     : out std_logic_vector(vwidth-1 downto 0);
                w12     : out std_logic_vector(vwidth-1 downto 0);
                w13     : out std_logic_vector(vwidth-1 downto 0);
                w21     : out std_logic_vector(vwidth-1 downto 0);
                w22     : out std_logic_vector(vwidth-1 downto 0);
                w23     : out std_logic_vector(vwidth-1 downto 0);
                w31     : out std_logic_vector(vwidth-1 downto 0);
                w32     : out std_logic_vector(vwidth-1 downto 0);
                w33     : out std_logic_vector(vwidth-1 downto 0);
                DV      : out std_logic:='0'
        );
        end component window_3x3;

        component rc_counter
        generic (
                num_cols: integer:=128;
                num_rows: integer:=128
                );
        port (
                Clk     : in std_logic;
                RSTn    : in std_logic;
                En      : in std_logic;
                ColPos  : out integer;
                RowPos  : out integer
        );
        end component rc_counter;

        signal ColPos: integer:=0;
        signal RowPos: integer:=0;
        signal ColPos_c: integer:=0;  -- corrected positions
        signal RowPos_c: integer:=0;
        signal rt1: integer:=0;
        signal rt2: integer:=0;
        signal rt3: integer:=0;
        signal rt4: integer:=0;
        signal rt5: integer:=0;
        signal rt6: integer:=0;
        signal rt7: integer:=0;
        signal rt8: integer:=0;
        signal rt9: integer:=0;
        signal rt10: integer:=0;
        signal rt11: integer:=0;
        signal rt12: integer:=0;
        signal rt13: integer:=0;
        signal rt14: integer:=0;
        signal rt15: integer:=0;
        signal rt16: integer:=0;
```

```
        signal flag: std_logic:='0';


begin

        sort_3x3x: sort_3x3
                generic map (
                        vwidth => 8
                )
                port map (
                        Clk     => Clk,
                        RSTn    => RSTn,
                        w11     => w11,
                        w12     => w12,
                        w13     => w13,
                        w21     => w21,
                        w22     => w22,
                        w23     => w23,
                        w31     => w31,
                        w32     => w32,
                        w33     => w33,
                        DVw     => DVw,
                        DVs     => DVs,
                        s1      => s1,
                        s2      => s2,
                        s3      => s3,
                        s4      => s4,
                        s5      => s5,
                        s6      => s6,
                        s7      => s7,
                        s8      => s8,
                        s9      => s9
                );

        window_3x3x: window_3x3
                generic map (
                        vwidth => 8
                )
                port map (
                        Clk     =>      Clk,
                        RSTn    =>      RSTn,
                        D       =>      D,
                        w11     =>      w11,
                        w12     =>      w12,
                        w13     =>      w13,
                        w21     =>      w21,
                        w22     =>      w22,
                        w23     =>      w23,
                        w31     =>      w31,
                        w32     =>      w32,
                        w33     =>      w33,
                        DV      =>      DVw
                );

        rc_counterx: rc_counter
                generic map (
                num_cols        => 128,
                num_rows        => 128
                )
        port map (
                Clk             => Clk,
                RSTn            => RSTn,
                En              => RSTn,
                ColPos          => ColPos,
                RowPos          => RowPos
        );

        ro_filt_proc: process(RSTn,Clk)
        begin
                if RSTn = '0' then
                        ColPos_c <= 0;
```

```
			rt1 <= 0;
			rt2 <= 0;
			rt3 <= 0;
			rt4 <= 0;
			rt5 <= 0;
			rt6 <= 0;
			rt7 <= 0;
			rt8 <= 0;
			rt9 <= 0;
			rt10 <= 0;
			rt11 <= 0;
			rt12 <= 0;
			rt13 <= 0;
			rt14 <= 0;
			rt15 <= 0;
			rt16 <= 0;
			RowPos_c <= 0;
			Dout <= (others=>'0');
			DV <= '0';
			flag <= '0';
		elsif rising_edge(Clk) then
			-- counter correction
			ColPos_c <= ((ColPos-16) mod 128);
			rt1 <= ((RowPos-1) mod 128);
			rt2 <= rt1;
			rt3 <= rt2;
			rt4 <= rt3;
			rt5 <= rt4;
			rt6 <= rt5;
			rt7 <= rt6;
			rt8 <= rt7;
			rt9 <= rt8;
			rt10 <= rt9;
			rt11 <= rt10;
			rt12 <= rt11;
			rt13 <= rt12;
			rt14 <= rt13;
			rt15 <= rt14;
			rt16 <= rt15;
			RowPos_c <= rt16;
			-- screen edge detection
			if (ColPos_c = num_cols-1) or (RowPos_c = num_rows-1) or (ColPos_c
			= num_cols-2) or (RowPos_c = 0) then
				Dout <= (others=>'0');
			else
				if order = 1 then
					Dout <= s1;
				elsif order = 2 then
					Dout <= s2;
				elsif order = 3 then
					Dout <= s3;
				elsif order = 4 then
					Dout <= s4;
				elsif order = 5 then
					Dout <= s5;
				elsif order = 6 then
					Dout <= s6;
				elsif order = 7 then
					Dout <= s7;
				elsif order = 8 then
					Dout <= s8;
				elsif order = 9 then
					Dout <= s9;
				end if;

			end if;
			if ColPos >= 16 and RowPos >= 1 then
				DV <= '1';

				flag <= '1';
			elsif flag = '1' then
```

```
                        DV <= '1';
                else
                        DV <= '0';
                end if;

        end if;
    end process;

end ro_filt_3x3;
```

conv_3x3.vhd

```
---------------------------------------------------------------------------
--      filename:       conv3x3.vhd
--      author:         Tony Nelson
--      date:           12/25/99
--
--      detail:         2D convolution operator with 3x3 size kernel, selectable in
--                      conv_3x3_pkg in the K constant.
--
--      limits:         none
---------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

package conv_3x3_pkg is
        -- the constants kx defines the kernel to be used in the convolution operation
        -- the kx value may be in the range -128<kx<128
        constant k0 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(1,8));
        constant k1 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(2,8));
        constant k2 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(1,8));
        constant k3 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(2,8));
        constant k4 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(9,8));
        constant k5 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(2,8));
        constant k6 : std_logic_vector(7 downto 0):=std_logic_ vector(to_signed(1,8));
        constant k7 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(2,8));
        constant k8 : std_logic_vector(7 downto 0):=std_logic_vector(to_signed(1,8));

        constant vwidth         : integer := 8;
        constant order          : integer := 1;
        constant num_cols       : integer := 128;
        constant num_rows       : integer := 128;
end conv_3x3_pkg;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.conv_3x3_pkg.all;

entity conv_3x3 is
        port (
                Clk     : in std_logic;
                RSTn    : in std_logic;
                D       : in std_logic_vector(vwidth-1 downto 0);
                Dout    : out std_logic_vector((vwidth*2)+1 downto 0);
                DV      : out std_logic
        );
end conv_3x3;

architecture conv_3x3 of conv_3x3 is

        signal w11: std_logic_vector((vwidth-1) downto 0);
        signal w12: std_logic_vector((vwidth-1) downto 0);
        signal w13: std_logic_vector((vwidth-1) downto 0);
        signal w21: std_logic_vector((vwidth-1) downto 0);
        signal w22: std_logic_vector((vwidth-1) downto 0);
        signal w23: std_logic_vector((vwidth-1) downto 0);
        signal w31: std_logic_vector((vwidth-1) downto 0);
signal w32: std_logic_vector((vwidth-1) downto 0);
signal w33: std_logic_vector((vwidth-1) downto 0);
signal DVw: std_logic;

component window_3x3
generic (
        vwidth: integer:=8
        );
port (
        Clk     : in std_logic;
        RSTn    : in std_logic;
        D       : in std_logic_vector(vwidth-1 downto 0);
        w11     : out std_logic_vector(vwidth-1 downto 0);
        w12     : out std_logic_vector(vwidth-1 downto 0);
        w13     : out std_logic_vector(vwidth-1 downto 0);
        w21     : out std_logic_vector(vwidth-1 downto 0);
        w22     : out std_logic_vector(vwidth-1 downto 0);
        w23     : out std_logic_vector(vwidth-1 downto 0);
        w31     : out std_logic_vector(vwidth-1 downto 0);
        w32     : out std_logic_vector(vwidth-1 downto 0);
        w33     : out std_logic_vector(vwidth-1 downto 0);
        DV      : out std_logic:='0'
);
end component window_3x3;

-- 16 bits for 8x8 plus 1 bit for sign
signal m0: signed((vwidth*2) downto 0):=(others=>'0');
signal m1: signed((vwidth*2) downto 0):=(others=>'0');
signal m2: signed((vwidth*2) downto 0):=(others=>'0');
signal m3: signed((vwidth*2) downto 0):=(others=>'0');
signal m4: signed((vwidth*2) downto 0):=(others=>'0');
signal m5: signed((vwidth*2) downto 0):=(others=>'0');
signal m6: signed((vwidth*2) downto 0):=(others=>'0');
signal m7: signed((vwidth*2) downto 0):=(others=>'0');
signal m8: signed((vwidth*2) downto 0):=(others=>'0');
signal a10: signed((vwidth*2)+1 downto 0):=(others=>'0');
signal a11: signed((vwidth*2)+1 downto 0):=(others=>'0');
signal a12: signed((vwidth*2)+1 downto 0):=(others=>'0');
signal a13: signed((vwidth*2)+1 downto 0):=(others=>'0');
signal a14: signed((vwidth*2)+1 downto 0):=(others=>'0');
signal a20: signed((vwidth*2)+2 downto 0):=(others=>'0');
signal a21: signed((vwidth*2)+2 downto 0):=(others=>'0');
signal a22: signed((vwidth*2)+2 downto 0):=(others=>'0');
signal a30: signed((vwidth*2)+3 downto 0):=(others=>'0');
signal a31: signed((vwidth*2)+3 downto 0):=(others=>'0');
signal a40: signed((vwidth*2)+4 downto 0):=(others=>'0');
signal d0: signed((vwidth*2)+1 downto 0):=(others=>'0');

component rc_counter
generic (
        num_cols: integer:=128;
        num_rows: integer:=128
        );
port (
        Clk     : in std_logic;
        RSTn    : in std_logic;
        En      : in std_logic;
        ColPos  : out integer;
        RowPos  : out integer
);
end component rc_counter;

signal ColPos: integer:=0;
signal RowPos: integer:=0;
signal ColPos_c: integer:=0;  -- corrected positions
signal RowPos_c: integer:=0;
signal rt1: integer:=0;
signal rt2: integer:=0;
signal rt3: integer:=0;
signal rt4: integer:=0;
signal rt5: integer:=0;
```

```
        signal rt6: integer:=0;
        signal rt7: integer:=0;
        signal rt8: integer:=0;
        signal flag: std_logic:='0';


begin

        window_3x3x: window_3x3
                generic map (
                        vwidth => 8
                )
                port map (
                        Clk     =>      Clk,
                        RSTn    =>      RSTn,
                        D       =>      D,
                        w11     =>      w11,
                        w12     =>      w12,
                        w13     =>      w13,
                        w21     =>      w21,
                        w22     =>      w22,
                        w23     =>      w23,
                        w31     =>      w31,
                        w32     =>      w32,
                        w33     =>      w33,
                        DV      =>      DVw
                );

        rc_counterx: rc_counter
                generic map (
                num_cols        => 128,
                num_rows        => 128
                )
        port map (
                Clk             => Clk,
                RSTn            => RSTn,
                En              => RSTn,
                ColPos          => ColPos,
                RowPos          => RowPos
        );

        convproc: process(Clk,RSTn)
        begin
                if RSTn = '0' then
                        m0 <= (others=>'0');
                        m1 <= (others=>'0');
                        m2 <= (others=>'0');
                        m3 <= (others=>'0');
                        m4 <= (others=>'0');
                        m5 <= (others=>'0');
                        m6 <= (others=>'0');
                        m7 <= (others=>'0');
                        m8 <= (others=>'0');
                        a10 <= (others=>'0');
                        a11 <= (others=>'0');
                        a12 <= (others=>'0');
                        a13 <= (others=>'0');
                        a14 <= (others=>'0');
                        a20 <= (others=>'0');
                        a21 <= (others=>'0');
                        a22 <= (others=>'0');
                        a30 <= (others=>'0');
                        a31 <= (others=>'0');
                        a40 <= (others=>'0');
                        d0 <=  (others=>'0');
                        Dout <= (others=>'0');
                        DV <= '0';
                        ColPos_c <= 0;
                        rt1 <= 0;
                        rt2 <= 0;
                        rt3 <= 0;
```

```
				rt4 <= 0;
				rt5 <= 0;
				rt6 <= 0;
				rt7 <= 0;
				rt8 <= 0;
				RowPos_c <= 0;
				flag <= '0';
			elsif rising_edge(Clk) then
				-- counter correction
				ColPos_c <= ((ColPos-8) mod 128);
				rt1 <= ((RowPos-1) mod 128);
				rt2 <= rt1;
				rt3 <= rt2;
				rt4 <= rt3;
				rt5 <= rt4;
				rt6 <= rt5;
				rt7 <= rt6;
				rt8 <= rt7;
				RowPos_c <= rt8;
				-- screen edge detection
				if (ColPos_c = num_cols-1) or (RowPos_c = num_rows-1) or (ColPos_c
				= num_cols-2) or (RowPos_c = 0) then
					Dout <= (others=>'0');
				end if;
				if DVw = '1' then
				-- window*kernel multipliers
				-- this could be optimized by using hardware-specified multipliers
					m0 <= signed('0'&w11)*signed(k0);
					m1 <= signed('0'&w12)*signed(k1);
					m2 <= signed('0'&w13)*signed(k2);
					m3 <= signed('0'&w21)*signed(k3);
					m4 <= signed('0'&w22)*signed(k4);
					m5 <= signed('0'&w23)*signed(k5);
					m6 <= signed('0'&w31)*signed(k6);
					m7 <= signed('0'&w32)*signed(k7);
					m8 <= signed('0'&w33)*signed(k8);
					a10 <= (m0(16)&m0)+m1;
					a11 <= (m2(16)&m2)+m3;
					a12 <= (m4(16)&m4)+m5;
					a13 <= (m6(16)&m6)+m7;
					a14 <= m8(16)&m8;
					a20 <= (a10(17)&a10)+a11;
					a21 <= (a12(17)&a12)+a13;
					a22 <= a14(17)&a14;
					a30 <= (a20(18)&a20)+a21;
					a31 <= a22(18)&a22;
					a40 <= (a30(19)&a30)+a31;
					d0 <=  a40(20 downto 3);
					if (ColPos_c = num_cols-1) or (RowPos_c = num_rows-1) or
					(ColPos_c = num_cols-2) or (RowPos_c = 0) then
						Dout <= (others=>'0');
					else
						Dout <= std_logic_vector(d0);
					end if;
				end if;
				if ColPos >= 8 and RowPos >= 1 then
					DV <= '1';

					flag <= '1';
				elsif flag = '1' then
					DV <= '1';
				else
					DV <= '0';
				end if;

			end if;
		end process;

	end conv_3x3;
```

REFERENCES

[1] Chou, C., Mohanakrishnan, S., Evans, J.: "FPGA Implementation of Digital Filters," Proc. ICSPAT, 1993.

[2] Benedetti, A., Perona, P.: "Real-time 2-D Feature Detection on a Reconfigurable Computer," Proceedings of the 1998 IEEE Conference on Computer Vision and Pattern Recognition, 1998.

[3] Gokhale, M., et. al.: "Stream-Oriented FPGA Computing in the Streams -C High Level Language," unpublished paper, 2000.

[4] Banerjee, N., et. al.: "MATCH: A MATLAB Compiler for Configurable Computing Systems," Technical Report, Center for Parallel and Distributed Computing, Northwestern University, August 1999.

[5] Lee, E., et. al.: "Overview of the Ptolemy Project," Department of Electrical Engineering and Computer Science, University of California, Berkeley, July 1999.

[6] Mathworks, Inc.: "MATLAB 5.3 Fact Sheet," Natick, MA, 1999.

[7] Research Systems, Inc.: "Getting Started with IDL," Boulder, CO, September 1999.

[8] Research Systems, Inc.: "ENVI User's Guide," Boulder, CO, July 1999.

[9] Texas Instruments, Inc.: "TMS320C4X User's Guide," Houston, TX, May 1999.

[10] Moore, M.: "A DSP-Based Real-Time Image Processing System," Proceeding of the 6th International Conference on Signal Processing Applications and Technology, Boston, MA, August 1995.

[11] Texas Instruments, Inc.: "C67x Floating -Point Benchmarks," Houston, TX, 2000.

[12] Virtual Computer Corporation: "What is Reconfigurable Computing," Reseda, CA, 2000.

[13] Nelson, A.: "An Implementation of t he Optical Flow Algorithm on FPGA Hardware," Independent Study Paper, December 1998.

[14] Nelson, A.: "Further Study of Image Processing Techniques on FPGA Hardware," Independent Study Paper, May 1999.

[15] Altera, Inc.: "Altera FLEX 10K Embedded Program mable Logic Family Data Sheet," San Jose, CA, 1999.

[16] Xilinx, Inc.: " Xilinx Virtex 2.5V Field Programmable Gate Array Specification," San Jose, CA, 2000.

[17] Andraka Consulting Group, Inc,: "Digital Signal Processing for FPGAs," Seminar Notes, 1999.

[18] Russ, J.: "The Image Processing Handbook," CRC Press, Boca Raton, FL, 1992.

[19] Hussain, Z.: "Digital Image Processing – Practical Applications of Parallel Processing Techniques," Ellis Horwood, West Sussex, UK, 1991.

[20] Dougherty, E.: "An Introduction to Morphological Image Processing," SPIE, Bellingham, WA, 1992.

[21] Pratt, W.: "Digital Image Processing," Wiley, New York, NY, 1978.

[22] Scott J., Bapty T., Neema S., Sztipanovits J.: "Model-Integrated Environment for Adaptive Computing," Proceedings of the Military and Aerospace Applications of Programmable Devices and Technologies Conference, Greenbelt, MA, September, 1998.

[23] Chen, K.: "Bit-Serial Realizations of a Class of Nonlinear Filters Based on Positive Boolean Functions," IEEE Trans. On Circuits and Systems, Vol. 36, No. 6, June 1989.