5-2004

Hyper-Spectral Image Processing Using High Performance Reconfigurable Computers

Yuan He
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Yuan He entitled "Hyper-Spectral Image Processing Using High Performance Reconfigurable Computers." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Gregory Peterson, Major Professor

We have read this thesis and recommend its acceptance:

Don Bouldin, Seong-Gon Kong

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Yuan He entitled "Hyper-Spectral Image Processing Using High Performance Reconfigurable Computers." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science, with a major in Electrical Engineering.

Gregory Peterson

Dr. Gregory Peterson, Major Professor

We have read this thesis
and recommend its acceptance:

Don Bouldin

Seong-Gon Kong

Accepted for the Council:

Anne Mayhew

Vice Chancellor and Dean of Graduate Studies

(Original signatures are on file with official student records)

Hyper-Spectral Image Processing

Using

High Performance Reconfigurable Computers

A

Thesis

Presented

For

The Master of Science Degree

The University of Tennessee,

Knoxville

Yuan He

May 2004

Acknowledgments

The work was performed at the Department of Electrical and Computer Engineering, University of Tennessee in Knoxville during the period from August 2002 to December 2003. I gratefully acknowledge the financial support of Dr. Gregory Peterson, the U.S. Air Force Research Laboratory and the Department of Electrical and Computer Engineering, UTK, for this work.

I want to thank my supervisor, Dr. Peterson, for introducing me to the interesting world of HPRC. He led my work, gave me support, and made valuable comments. Dr. Peterson is not only a very good teacher of digital systems but also an excellent supervisor.

I would like to express my special gratitude to Dr. Seong G. Kong for providing me with the original Matlab application, Dr. Hairong Qi for letting me use and modify a portion of her image processing header file, and Dr. Donald W. Bouldin for letting me use his laboratory equipment. Without them, this research study would not have been possible.

I want to thank all my colleagues and friends at ECE for their friendship, help and interesting discussions inside and beyond the field of electrical engineering. Special thanks to Choi Hing for his help with various practical problems.

Last, but not least, I wish to thank my family and my relatives for their love, understanding, patience, and continuous support over the years. I give my deepest gratitude to my father and grandmother for talking me into graduate school. I would also like to thank them, along with my mother and grandfather, for their advice and nurturing.

Remembrance to my grandmother. Still.

Yuan He

April 2004, Knoxville.

Abstract

The purpose of this thesis is to investigate the methods of implementing a section of a Matlab hyper-spectral image processing software application into a digital system that operates on a High Performance Reconfigurable Computer. The work presented is concerned with the architecture, the design techniques, and the models of digital systems that are necessary to achieve the best overall performance on HPRC platforms. The application is an image-processing tool that detects the tumors in a chicken using analysis of a hyper-spectral image. Analysis of the original Matlab code has shown that it gives low performance in achieving the result. The implementation is performed using a three-stage approach. In the first stage, the Matlab code is converted into C++ code in order to identify the bottlenecks that require the most resources. During the second stage, the digital system is designed to optimize the performance on a single reconfigurable computer. In the final stage of the implementation, this work explores the HPRC architecture by deploying and testing the digital design on multiple machines. The research shows that HPRC platforms grant a noticeable performance boost. Furthermore, the more hyper-spectral bands exist in the input image data, the better of the speedup can be expected from the HPRC design work.

Contents

List of Tables

List of Figures

Chapter 1. Introduction

This chapter gives a brief overview to the background of High Performance Reconfigurable Computers, the objectives, the contributions, and the composition of the thesis.

1.1 Background

Lately, many areas of research are exploring the use of reconfigurable computers (RC), such as field programmable gate array (FGPA), along with a conventional processor. In general, such a system is known as a Field Programmable Custom Computing Machine (FCCM). FCCMs offer the benefit of the speed from an application-specific coprocessor, combined with the capabilities and flexibilities of FPGAs. Conventional processors can compute general-purpose tasks, while leaving complex and processor-intensive work to the application-specific hardware units. Several research groups have demonstrated the performance improvements using RC architectures [19-21].

A related field of study extends the FCCMs to High Performance Reconfigurable Computers (HPRC). The idea of HPRC is to encompass parallel processors to work collectively on a common problem while each individual processor may or may not be a FCCM system.

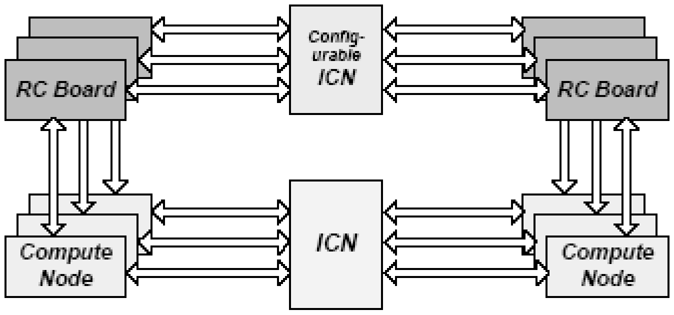


Figure 1-1: High Performance Reconfigurable Computer Architecture [7]

Figure 1-1 shows HPRC architecture, which consist a number of compute nodes that are linked by an interconnection network. A reconfigurable hardware may be attached to any of the compute nodes and there might be an interconnection network that exists between the RCs.

In this work, a reconfigurable superscalar processor model uses a reconfigurable system called Pilchard to simulate the HPRC environment [1]. At the Electrical and Computing Engineering Department of University of Tennessee Knoxville, there are eight Pilchard systems available for usage.

In this project, a hyper-spectral image processing application is considered. This particular application contains the functions that offer the complexity that calls for a dedicated reconfigurable device. In addition, the functions have to be performed on each of the hyper-spectral bands, thus its repetition can benefit from using a high performance computing system. The selection of this image processing application is appropriate for the study of the HPRC.

While much of the work focuses on transferring the complexity part of the original application into specialized hardware functions, other means also have to be considered for the communications between the software and the hardware and as well as at the superscalar level.

1.2 Objectives

In view of the background stated above, the first goal of the work is to, identify any performance bottlenecks that the original software application possesses, then accelerating the bottleneck code using the Pilchard platform. The RC system using the Pilchard platform is expected to give a speedup as compared to the software counterpart.

The second goal builds on the results of the first objective in order to explore HPRC platforms. The idea is to have each of the eight available Pilchard systems responsible for a subset of the hyper-spectral image bands. The eight systems can work concurrently with their corresponding FPGA components to produce even a better speedup than projected in the first goal.

The third and final goal is to consider the algorithm and design methodology used to help outline a standard approach to accelerate software applications by using HPRC.

1.3 Main Contributions

The work performed gives the following main contributions:

- Constructed an image processing hyper-spectral application on HPRC by converting it from a Matlab application.
- Constructed an implementation using HPRC with standard design flow.
- Designed, implemented, and/or verified the sub-modules of the digital system: Pilchard, Pcore, Parith, Fxmult, and Max.
- An analytical study of 2D Wavelet-Transform
- An analytical study of the data streaming process using the Pilchard platform.
- An analytical study of the Pilchard platform and its design package
- A comparative study of different methods of design for the chicken tumor application

1.4 Structure of Thesis

The thesis is divided into five chapters. Chapter 1 gives a general introduction to the work. Following the introduction, the principles and basic concepts of the Pilchard platform and its design packages are described in Chapter 2. Chapter 3 gives a detailed description of each algorithm and tasks performed in the original hyper-spectral image processing application. The work performed is treated in Chapter 4. It contains each step of the project flow. The results, the overall conclusions of the work and the suggestions of future work are given in the last Chapter.

Chapter 2. Pilchard Platform and Design Package

The hardware implementation of this project is to be developed on a reconfigurable computing environment named Pilchard [1]. Consequentially, this chapter presents an overview of the Pilchard platform and design issues related to it.

2.1 Pilchard Overview

The Pilchard is a high performance reconfigurable computing platform that was developed in the Computer Science and Engineering Department of the Chinese University of Hong Kong [3]. It exploits a field programmable gate array device that utilizes the dynamic RAM dual in-line memory module to interconnect with its host, which typically is a personal computer. The system is low-cost and with its efficient interface, it offers the flexibility for quick prototyping of various applications. The overhead, whether it is timing or hardware resources, is minimized to maximize the resources available for the developers. In addition, the learning curve for implementing a digital design with the Pilchard platform is not steep, as suggested by the Chinese University of Hong Kong [1,3]. These benefits give the developers more time to carry out their design work rather than spending excessive time on learning the interface protocols. For these reasons, the Pilchard is used.

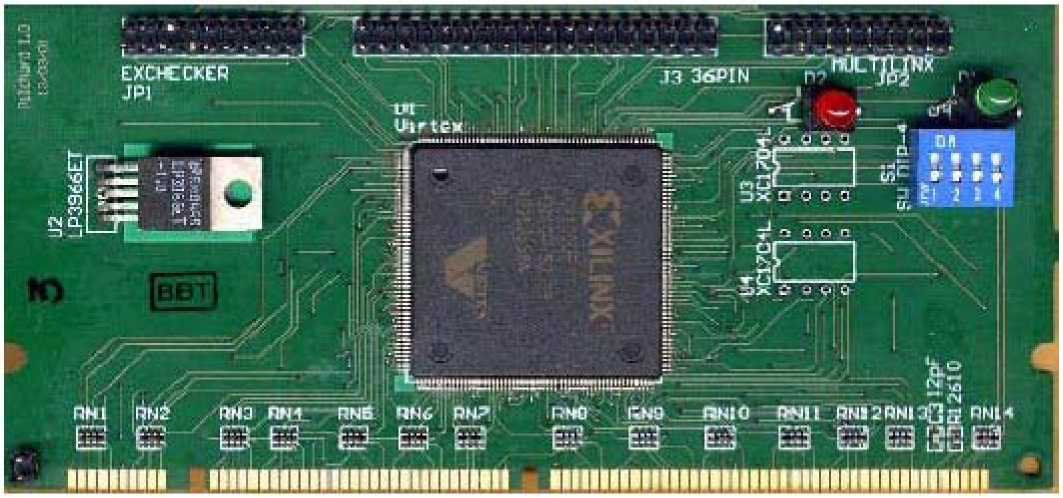


Figure 2-1: Photography of The Pilchard Board [1]

Besides the feasibility that the Pilchard system offers, it also contains some features and particulars that are worthy of mentioning. Figure 2-1 shows a picture of the Pilchard board.

The main FGPA component is a Xilinx Virtex-E, XCV1000EHQ240, chip, however, it is supported by any of the Xillinx Virtex and Virtex-E device family in the PQ240 or the HQ240 packages. The Pilchard board is designed to be compatible with the 168 pin 3.3 Volt, 133MHz, 72-bit, DIMMs. The printed board is a 6-layer impendence controlled FR4 board and roughly doubles the height of a standard DIMM card. Currently, the Pilchard is only supported by the ASUS CUSL2-C motherboard and tested and operated on Mandrake Linux 8.1 x86 version. The configuration bit-stream files are download onto the Pilchard platform using the Parallel Cable III with the Xchecker interface. A flowchart of the Pilchard board is shown on figure 2-2 and more in-depth specification is shown in table 2-1.

2.2 Xilinx Virtex-E Chip

The architecture of the Xilinx Virtex-E chip consists of three major configurable elements, an array of configurable logic blocks (CLBs), programmable input/output

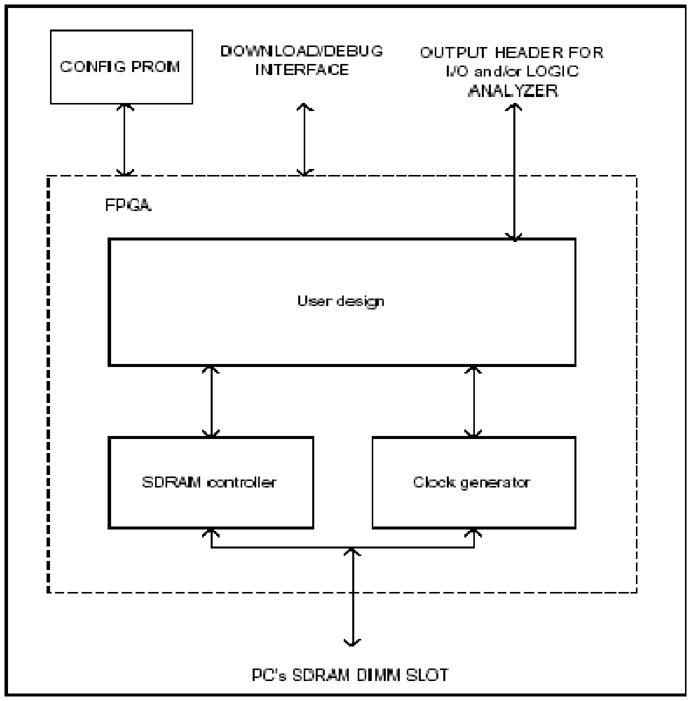


Figure 2-2: Block Diagram of The Pilchard Board [1]

Table 2-1: Pilchard Platform Specifications [3]

Features	Description
Host Interface	DIMM Interface 64-bit Data I/O 12-bit Address Bus
External (Debug) Interface	27 Bits I/O
Configuration Interface	X-Checker, Multil.ink and JTAG
Maximum System Clock Rate	133 MHz
Maximum External Clock Rate	240 MHz
FPGA Device	XCV 1000E-HQ240-6
Dimension	133mm * 65mm * 1mm
OS Supported	GNU / Linux

blocks (IOBs), and interconnects. The CLBs are the basic functional elements for mapping user-constructed logics. The IOBs connect the exteriors pins on the Pilchard board with the internal signal lines. The interconnect serves as the interface routing the connections between the CLBs and the IOBs. User-specific functions are configured onto the XCV1000E-HQ240 chip boarding the FPGA. Its specification is summarized in table 2-2.

The Xillinx Virtex-E FPGA has four digital Delay-Locked Loops (DLLs) and four Global Clock Buffers for global clock distribution. However, only three out of the four

Table 2-2: Features of XCV1000E-HQ240 [2]

Parameter	Features
System Gates	1,569,178
Logic Gates	331,776
CLB Arrays	64 * 96
Logic Cells	27,648
User I/Os	660
Differential I/Os	281
BlockRAM Bits	393,216
Distributed RAM Bits	393,216

Global Clock Buffers are used for Pilchard due to the Pilchard architecture. Two out of four outputs from the DLLs are labeled and available for use. They are labeled as CLK and CLKDIV in *pcore.vhd*, a Pilchard Harware design file, which will be covered later in the chapter. The remaining two DLLs are also available for use, but will require user modification of the Pilchard design files, namely the *pilchard.vhd* and the *pcore.vhd*. These two DLLs are not declared or labeled in the original Pilchard design files.

Another major feature of the Virtex-E FPGA chip is the on-board Block SelectRAM+, which had an impact on the overall design outcome. The Block SelectRAM+ uses a dual port BlockRAM, containing a total of 96 blocks of RAM, each holding 4096 bits data. A timing factor worth of underlining is that with the Dual-Port RAM, the read/write request can only be fulfilled in every two clock cycles but allows simultaneously access data on both ports at different memory address locations.

2.3 Pilchard Design Files

Another important part of the Pilchard platform, beside the physical device, is the included design file packages. The Pilchard design files contain both VHDL files and software files that are necessary for the user design implementations and FPGA-Host interfaces. Both of the resources need to be edited accordingly to ensure synchronized interface communications.

The VHDL files that needed by the developers are *Pilchard.vhd*, *pcore.vhd*, a Pilchard user constraint file (UCF), and a set of netlist files (EDIF). The *pilchard.vhd* is the top level VHDL codes that bring forth the interfaces with the host DIMM slot directly. It also configures the global clock signal, clock divider, I/O buffer, and startup reset of the FPGA device on the Pilchard. Unless new sources are added to the interface, such as a new clock signal, or special design constraints are to be met, this files does not need be modified. Instead, most of the design logic can acquire enough resources to communicate with the host from the “pcore.vhd” and should be placed in or under this module. Some of the default I/O ports in this file are predefined in association with its parent file, pilchard.vhd, for access the host interface, however, some others are for testing purposes can be left unused. The Pilchard's UCF is a hardware-dependent file that contains the information regarding pin locations and timing constraints of the Virtex-E chip. The Pilchard's EDIF is the pre-synthesized file that provides the netlist for I/O blocks used in "pilchard.vhd".

The included software packages are used for the host-side interface, and it contains a set of C library code, the "iflib.h" and "iflib.c", which are the library header file and the C source code, respectably. This set of library files defines four essential application-program-interface (API) functions that handle the data transfer between the host and the Pilchard board. The "write32" and "read32" are used for 32-bit data transfers, while the other two functions "write64" and "read64" are used for 64-bit data transfers. The Pilchard user reference recommends using the 64-bit interface, since the 32-bit interface is slow and inefficient [3]. Even when working with a 32-bit design application, the user may still use the 64-bit without decrease in speed performance. All of the files in the Pilchard design packages may be found in the Appendix.

2.4 Pilchard Host Interface

The data transfer is perhaps one of the most important parts of the Pilchard host interface. The "pcore.vhd" contains two 64-bit signals, "din" and "dout", which are connected directly to the system memory bus through the DIMM slot. Along with the memory bus signals, there is also an 8-bit memory address bus that allows Pilchard to address up to 2^8 memory locations. This limitation is constrained by the hardware resources and the software drivers. To access more memory locations, other means have to be implemented such as using data bus to store address locations, using counter schemes, or split the address bits into two or more address bus locations.

When the host issues a write request, the input data from “DIN” signal and the address bus “addr” are to be read simultaneously on the Pilchard side, to ensure the correct data is write to the corresponding location. Similarly, when the host releases a read command, both the read signal and address bus are triggered the same time. However, the memory address will only be ready at the data output port, “dout,” at the next clock cycle. Figure 2-3 and 2-4 show the Pilchard write and read cycles.

2.5 Chapter Summary

The high performance reconfigurable computer platform used for this project is called Pilchard. The hardware issues and specifications for the Pilchard platform were discussed in this chapter. The next chapter looks into the applications that are going to be implemented on this system.

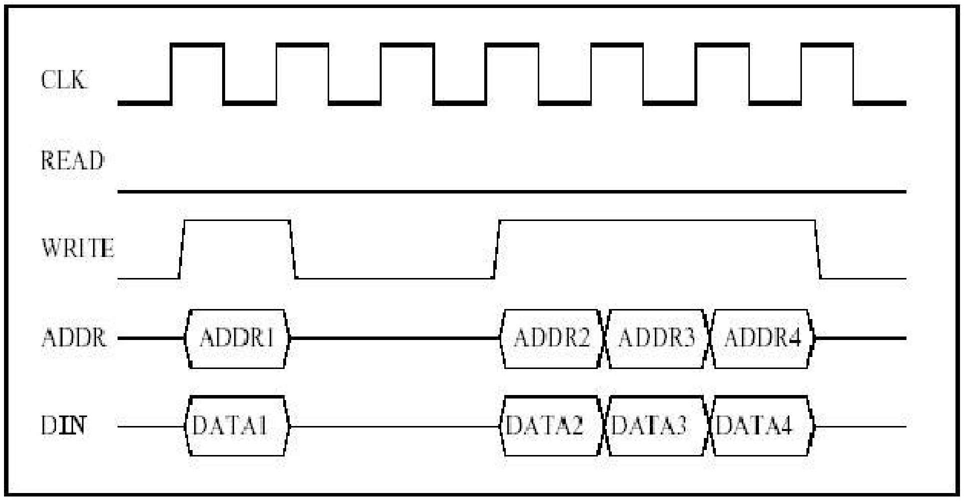


Figure 2-3: Pilchard Write Cycle [3]

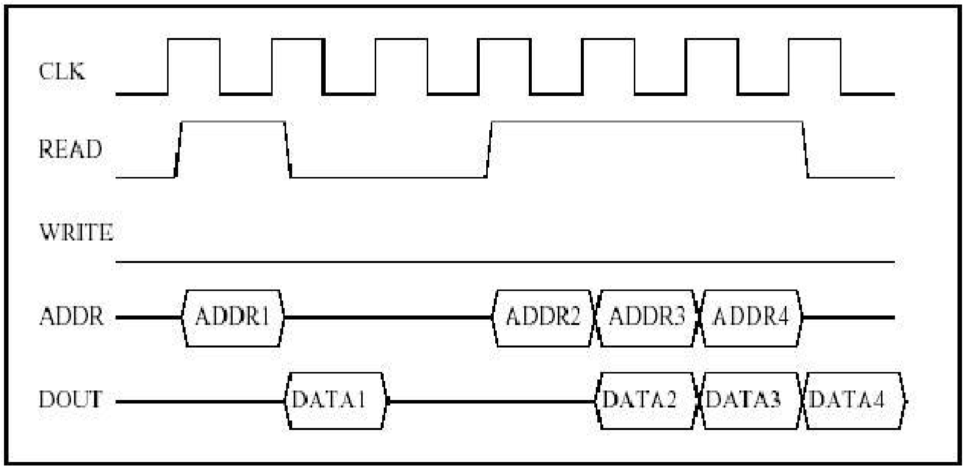


Figure 2-4: Pilchard Read Cycle [3]

Chapter 3. Hyper-Spectral Imaging Application

Chapter three provides a description to the original hyper-spectral software application, which was implemented using Matlab. Presented here are some of the key points within the software that will have an impact on the overall design flow.

3.1 Introduction

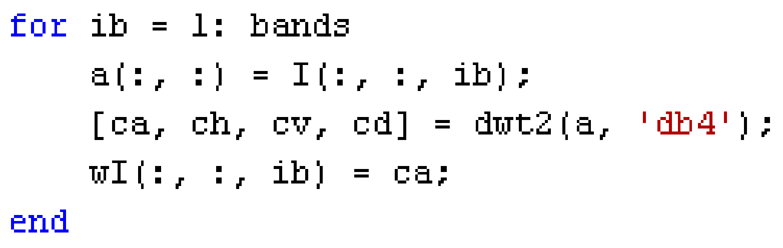
This research is based on an image processing application that use a hyper-spectral image taken from a chicken with numerous tumor spots. The original analysis code was written in Matlab by Dr. Seong G. Kong of the Department of Electrical and Computer Engineering at University of Tennessee Knoxville. It consisted of four main functions, discrete wavelet transform, normalization, signature plots and features extractions. The algorithm is scripted in the order specified above, together with few initialization or utility codes form the application. The application operates on a data image that has 65 hyper-spectral bands, each with a resolution of 460 by 400 digital pixels, making a 32M-byte data file. The source code itself is merely two-pages long and about 100 lines of code, however, due to the size of the image and the algorithms used, it takes Matlab an average of 3 to 4 minutes to perform these functions over one sample set of image data. These calculations were performed on a test bed using Pentium III, 1 GHz. processor with 512 Mb of RAM.

Consider remote sensing, a major technology field that uses hyper-spectral image processing, commonly produces images with up to 288 separate bands and covering regions from 0.4 to 2.5 micrometers [8]. This is about 300 times higher resolution than the chick data sample for each band and 1350 times larger in total resolution size. To put into perspective, assuming the Matlab calculation time operates linearly with input image size, then applications such as remote sensing would literally take Matlab 4725 minutes or more than 3 days to perform this application. Realistically, calculation times are not linearly proportionally to the input data size. The computation time actually increase more due to the reiteration of larger matrices or image resolutions. To overcome this deficiency, we will explore the idea of migrating the software bottleneck onto a hardware system using High Performance Reconfigurable Computers in this project.

The remaining of the chapter provides a description of all the functions used in the application according to their execution order. While the purpose and the functionalities of each function are important to understand, but moreover, the algorithms are the key to this project's success. The understanding of these algorithms will be used at later design stages.

3.2 Discrete Wavelet Transforms

The first step in this set of application is applying a 2-dimensional Daubechies 4 (Daub4) discrete wavelet transform [28, 29, 30]. Wavelet transform is an important spectral analysis tool. It is used in various applications such as signal processing and image processing, communications, and more. The extent of this information can be found in references [28, 29, 30]. For this section, we will only explore enough for the readers to understand the Matlab functions, used in this particular program, of the discrete wavelet transformation. The information presented in this section serves the fundamental stepping-stones to the two major designs stages, converting from Matlab code to C++ and the VHDL coding of the discrete wavelet transform. Figure 3-1 shows the section of codes used in the original Matlab program.



```
for ib = 1: bands
    a(:, :) = I(:, :, ib);
    [ca, ch, cv, cd] = dwt2(a, 'db4');
    wI(:, :, ib) = ca;
end
```

Figure 3-1: Matlab Coding of The Wavelet Transform

The discrete wavelet transform is defined by a square matrix of filter coefficient. Its fast linear operation operates on a data vector and transforming it into a numerically different vector whose length usually remains the same. When the wavelet transform is correctly constructed, the matrix is orthogonal, the transform and the inverse transform can be implemented [10]. In this project we will restrict ourselves to the Daubechies class wavelet filter due its mere presence in our application, denote by the “db4” in the dwt2 function. See figure 3-1. This class of filter includes members ranging from highly localized to highly smooth. The simplest or the most localized member is called DAUB4, which contain only four coefficient, h0, h1, h2, h3. Similarly less localized Daubechies could have more coefficients, the number of coefficient will corresponding to its name. Hence, DAUB6 will have 6 coefficients.

To understand the algorithm of the discrete wavelet transform, consider the transformation matrix, shown in figure 3-2, acting on a column vector of data to its right.

Note the structure of this matrix. The first row generates one component of the data convolved with the filter coefficient h0, … h3, likewise the third, fifth, and other odd rows. If the even rows follow the same pattern, offset by one, then the matrix would be a circulant, that is, an ordinary convolution that could be done by FFT methods. Instead of convolving with h0, h1, h2, h3, the even rows perform a different convolution, with different coefficient g0, g1, g2, g3, which correspond to the values of h3, -h2, h1, -h0 respectively. When compute the last set of data in a vector, the multipliers from the last

$$\begin{array}{cccccccccc}
h_0 & h_1 & h_2 & h_3 & 0 & 0 & 0 & 0 & & \\
g_0 & g_1 & g_2 & g_3 & 0 & 0 & 0 & 0 & & \\
0 & 0 & h_0 & h_1 & h_2 & h_3 & 0 & 0 & & \\
0 & 0 & g_0 & g_1 & g_2 & g_3 & 0 & 0 & & \\
0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 & & \\
0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3 & & \\
0 & 0 & 0 & 0 & 0 & 0 & h_0 & h_1 & h_2 & h_3 \\
0 & 0 & 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 & g_3
\end{array}
\bullet
\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \end{bmatrix}$$

Figure 3-2: DAUB4 Transformation Matrix

pair of the multiplications, denote as h2, h3, g2, and g3 in row seven and eight in figure 3-2, wraps around to the beginning of the vectors. The overall action of the matrix is, thus, to perform two related convolution, then to decimate each of them by half and interleave the remaining halves.

Sometimes, it is useful to think of the filter with f coefficients as a smoothing filter; it is like a moving average of four points. On the converse, the g coefficient filters is not a smoothing filter due to its minus signs. Together, both filters make up what image processing refers to as a quadrature mirror filter [28,29]. In fact, the coefficient in the g filter is chosen to make it yield a zero response to a sufficiently smooth data vector. This results in the output of h filter, decimated by half, accurately representing the data's

$$h_0 = \frac{1+\sqrt{3}}{4\sqrt{2}}$$
$$h_1 = \frac{3+\sqrt{3}}{4\sqrt{2}}$$
$$h_2 = \frac{3-\sqrt{3}}{4\sqrt{2}}$$
$$h_3 = \frac{1-\sqrt{3}}{4\sqrt{2}}$$

Figure 3-3: Daub4 Wavelet Filter Coefficient

"smooth" information. While the output of g filter is also decimated is referred to as the data's "detail" information. The coefficient is listed in figure 3-3.

The convolution with the h coefficient is sometime described as the low-pass filter effect and the convolution with the g coefficient is referred as the high-pass filter effect. Figure 3-4 shows the overall effect of the discrete discrete wavelet transform on a 2-demsional image. The LL represents a set of two low-pass filter used on the rows and the columns vectors of the data. This particular component is labeled as the variable "ca" in Matlab. See figure 3-1. The remaining functions presented in the Matlab program deals with only the result from the "ca" component of the discrete discrete wavelet transform. Thus, throughout the designing stages of the this project, only this component of the discrete discrete wavelet transform will be proposed, calculated, and compared.

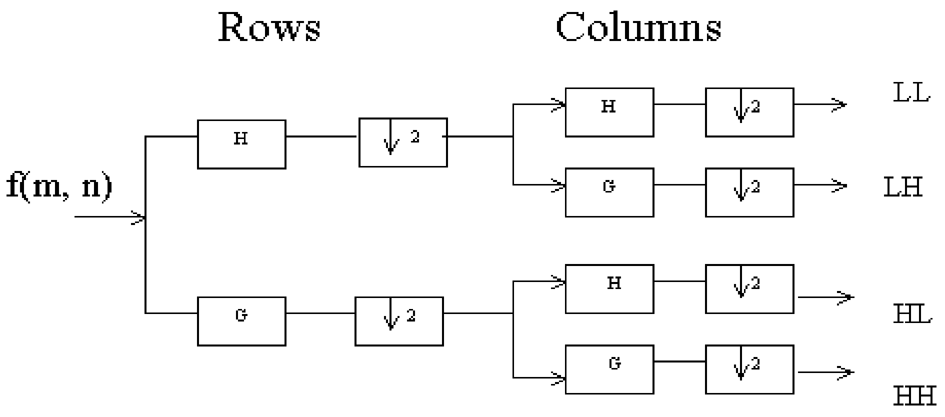


Figure 3-4: Wavelet Transform Filter Effect

3.3 Normalization

After the discrete wavelet transformation, normalization must be performed in order to show a spectral image of the data. Normalization is an easy concept to grasp. It is used to attain a normalization of the grey level by stretching the data to full dynamic range. This is achieved by dividing each pixel by the overall maximum value. The algorithm itself is fairly easy, however, most of computation time are consumed for finding the maximum value of each band then applied it to each pixel.

3.4 Hyper-spectral Plots

Hyper-spectral plots graph the data sets of normal tissues, tumor tissues, and background for each of the 65 spectral bands. The coordinates of each respective series are manually picked by examining the normalized spectral image. The algorithm is simply the intensity value at the given coordinates range divided by the maximum pixel value among the 65 bands.

3.5 Feature Extractions

The feature extraction process is the final operation in the image processing application. Generally, the feature extraction takes an image that has been processed and converts the areas of interest into well-defined regions that can be used for further investigation. Once areas of interest have been identified in the image, then convert the image into a bit map with areas of interest valued at one and remainder of the image set to zero.

This type of image can be processed in a variety of ways. Among the popular techniques use for this application are the area labeling, threshold comparison, and a more complex technique, Hough transform, just to name a few. Area labeling splits a segmented image into distinctly labeled areas. The image is scanned row-by-row and column-by-column to find the first filled pixel and then the output image is labeled as far to the right and left of that pixel as possible while the input image is zeroed. Then the labeled area is scanned from left to right checking for connected pixels above and below the line. When a connected pixel is found the procedure is repeated recursively starting from the connected pixel. This recursive procedure is continued until the whole area has been labeled and there are no more connected pixels. Then scanning recommences to find the next area in the input image to be labeled.

The Hough transform uses a technique to detect the basic shapes within the image. For example, at its simplest the Hough transform can be used to detect straight lines. If the pixels detected fall on a straight lines then they can be expressed by the equation $y=mx+c$. The basis of the Hough transform is to translate the points in (x,y) space into (mc,) space using the equation $c=(-x)m+y$. Thus each point in (x,y) space represents a line in (m,c) space. Where three or more of these lines intersect a value can be found for the gradient and intercept of the line that connects the (x,y) space points. The Hough transform can be expanded to consider circles by transforming the (x,y) space into a circle centre space, and even to arbitrary object providing that their shape and orientation are known before hand.

The particular method used by Dr. Kong is the second method mentioned above, the threshold comparison. While this method being one of the simpler methods in feature extraction, simpler than the two method mentioned above, however, it only works with image that has small range of intensive levels. The chicken tumor application only associates with three areas of intensity, which are the normal chicken tissue pixels, the tumor chicken tissue pixels, and the background image pixels. Each of the feature intensity level is assigned by a value calculated through the means and the Gaussian membership functions of their respected pixels. Then, the data is scanned pixel by pixel and comparing the data pixel against the three feature pixel values. The tumor spot pixels are valued at one if the pixels intensity is less than the normal tissue intensity and greater than the background intensive value. All other data values are assigned zero.

3.6 Chapter Summary

This chapter studied the original Matlab hyper-spectral imaging application, which contains four sub-sections, discrete wavelet transform, normalization, hyper-spectral plots and feature extractions. The algorithms for each sub-section are discussed. Next chapter will start by examining the profiles of each sub-section, thus determining the bottlenecks that will be implemented on the Pilchard platform.

Chapter 4. Design and Implementation

Chapter four discusses the design methodology and the design cycle that makes up this project. It is sectioned based on the design steps, from an overview to each of the design processes.

4.1 Overall Design Flow

The design stage of this project begins with examining the Matlab profile. Matlab is well established as an effective tool for performing numerical experiments and graphic simulations. Its simple, high-level programming language allows rapid development of new projects and facilitates debugging. However, a high-level interpreted language such as Matlab cannot compete in speed and memory efficiency with traditional compiled language such as FORTRAN and C/C++. Thus a good speed up can be gained from simply transforming the Matlab code to a simpler programming language, which leads to the next design stage. In the second stage, the original Matlab code is re-written to C++ from top to bottom. By analyzing the profile of the program in this new platform, the bottlenecks are pinpointed to a few operations. These bottlenecks are then re-designed, to match the benchmarks from the original Matlab output, with VHDL onto the FPGA using the Pilchard platform. The results from each of the programs are then compared and evaluated. This completes the design stages set for a single PC. When this is completed, the project further explores the potential performance boost from a parallel computing environment by utilizing all of the available Pilchard machines at the Department of Electrical and Computer Engineer at University of Tennessee. Figure 4-1 shows a flow chart of the overall design flow for a single Pilchard platform.

4.2 Matlab Profile

In order to obtain the best trade off between computation time versus hardware cost and design time, it is important to find the section/sections of the analysis that consume the most time, then transferring those sections onto the new platform. The analysis for the Matlab code is done by the built-in profile.

The program consists of seven sections, which are listed in table 4-1. The Matlab run-time is computed and shows the time used per its functions. The run time in each section is the sum of the total time taken for all functions under its corresponding section. It is worthy noting that even though a section consist a function with the longest run time, it is not necessarily the longest run-time section. Sections are divided in such a way that it contains a main function along with its corresponding function setups and/or declaration of variables. For example, dwt2 is one of the two functions used as part of the 2D discrete wavelet transform. It consumed the most run time as a single function calls, but

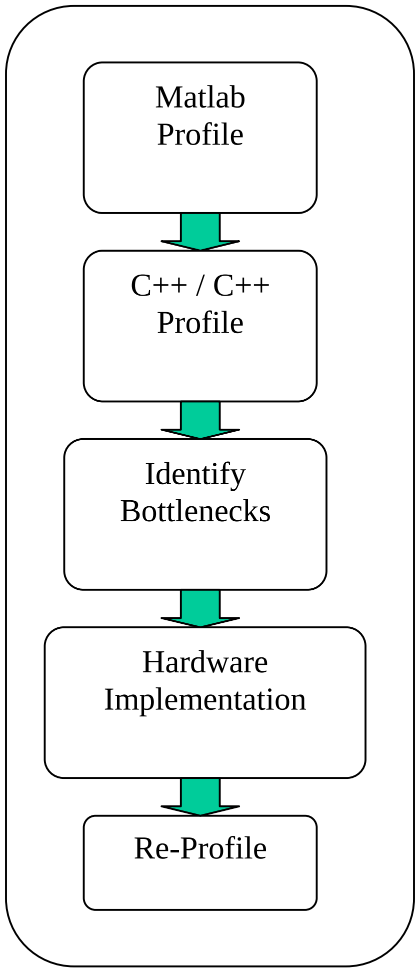


Figure 4-1: Overall Design Flow

Table 4-1: Matlab Profile

Matlab Profile		
	Seconds	% of Total Time
Total Run Time	201.15	-
Setup Time (read data, declare variable, etc.)	25.73	12.79%
2D Wavelet Transform	35.85	17.82%
Normalization	5.925	2.946%
Hyper-spectral Signatures Plots	n/a	n/a
Feature/Tumor Extraction	124.78	62.03%
Output 3D Image Result	n/a	n/a

the 2D discrete wavelet transform only ranked second as calculated by the sections. Table 4-1 reveals the result of the Matlab Profile. It is easily observed that a combined 92% of run time is spent on setup, 2D discrete wavelet transform, and feature tumor extractions. While the set up time ranks third, however, it is not categorized as a major application. It is merely a programming oriented protocol; the setup time will vary depending on the program software used. However, due to the size of the testing image, it is expected that little speed-up can be gained from this procedure. The focus lays on the remaining two functions, which are the 2-deminsional discrete wavelet transform and the feature/tumor extractions. These functions will be implemented.

4.3 C++ Designs

To reduce the overhead exhibited on Matlab, C++ was chosen to be the candidate of the lightweight platform for two reasons. First, an image-processing library is available from previous course work [31]. It contains a versatile image class that offers easy manipulation of rows and columns of the data. Second, C++ is the most familiar programming platform to the author, in comparison with other alternatives like, C, FORTAN, etc.

4.3.1 C++ Designs and Implementations

Much of the C++ coding follows the algorithm and process presented in the Matlab code with the exception of the few complicated Matlab functions. This included the input function and the display function of the image data and the discrete wavelet transform function. Since the source code of any Matlab function is undisclosed, the analogous implementation in each of these Matlab functions are only technically sound in their functionalities, the actual results may not be exact. Also, there are other factors that are unknown from the Matlab algorithms, such as round offs and precision bits, which could result in a minor discrepancy between the conversions. However, research was done to understand these functions, in order to keep the disparities at a minimum.

One of the biggest challenges and the one that was expected to show most of the disparities between the Matlab and the C++ program are the discrete wavelet transform functions. First, this is a three-dimensional image. Typical discrete wavelet transform algorithms use one-dimensional vectors. The three-dimensional hyper-spectral discrete wavelet transform algorithm behaves similarly as the one-dimensional transform. The algorithm for the two-dimensional discrete wavelet transform is to apply a transformation on the rows of the image and downsize the result by half, then once more over the column values, for each of the two-dimensional spectral bands. The setback with using the one-dimensional algorithm on a two-dimensional data set is that adjustments have to be made to correct the dimensional vectors to the proper size, so that the transformed one-dimensional vector matches their correct representation of rows and columns. Also, the algorithms used in C++ are a simplified version of the discrete wavelet transform; recall from chapter three the original function in Matlab only deals with the computation regarding the low-pass component of the transform. Many dissimilarities are expected between this and the Matlab algorithm. While Matlab uses an industrial-standard image processing system from specialized toolboxes to perform the transformations, the algorithm used in C++ is a rather simple straightforward.

Also worth mentioning is that in order to properly display the spectral images, the C++ outputs the image file into binary data files, then they are opened and displayed using the same Matlab function as in the original Matlab program.

4.3.2 C++ Profile

The C++ run time is computed by using the clock ( ) function, which is manually inserted at each appropriate corresponding section. Table 4-2 shows the result of the

Table 4-2: C++ Profile

	Matlab		C++	
	Seconds	% of Total Time	Seconds	% of Total Time
Total Run Time	201.15	-	11.67	-
Setup Time (read data, declare variable, etc.)	25.73	12.79%	0.6	5.14%
2D Wavelet Transform	35.85	17.82%	3.44	29.48%
Normalization	5.925	2.946%	4.09	35.05%
Hyper-spectral Signatures Plots	n/a	n/a	0.01	0.09%
Feature/Tumor Extraction	124.78	62.03%	0.13	1.11%
Output 3D Image Result	n/a	n/a	3.4	29.13%

C++ profile using a Pentium III, 1Ghz PC with 512 Mb of RAM, the same test bed that was used for the Matlab profile.

By comparing the results from the table below, the overall run time is reduced from 201.15 seconds to 11.67 second. The setup utilities, discrete wavelet transforms, and feature extraction functions were the three functions with the highest reduction. These sets of functions perform extensive matrix computations and iteration of loops. The matrix overhead that exists in Matlab is responsible for most of the performance hindrance observed here.

Within the C++ profile, three major time-consumers are the discrete wavelet transform, normalization, and the output of the 3D image result. Due to the size of the image data, the long output functions run time is unavoidable. Perhaps the greatest performance improvements are to be made from the remaining two functions. The designs and the implementation of these functions are covered more in detail in the next chapter.

4.4 VHDL and Hardware Designs

As suggested from the previous section, the hardware design is to better implement the two bottlenecks in C++, which are discrete wavelet transform and normalization. A successful implementation would show a good performance improvement. Although discrete wavelet transform has been widely researched and many IP cores exist, due to resource limitation on the Pilchard platform and the sizeable content of the input images, the implementation of the IPs for an entire wavelet transform is not feasible for this project. Thus, the digital design of the discrete wavelet transform applications has to be manually designed and implemented. However, the use of IPs for smaller scope of the digit design was explored.

In order to best balance the trade off between design time and performance, with consideration of the limited RAM resources on the Pilchard system, only the two bottlenecks, wavelet transform and normalization are addressed in the hardware design.

The lowpass-lowpass (LL) portion of the wavelet transformation is implemented on the Pilchard. Recall from chapter three, the LL portion of the wavelet transformation performs two tasks. First, it performs an operation of the sum of four products. Second, as it continues the numerical operation through its data image, it decimates the number of the output by half. With the current design, the function of the sum of the four products is implemented on the FPGA board. The second task is controlled by the data feed from the host side. As data are feed in using the streaming technique, only a portion of the data vectors is processed at a time. The process iterates until all of the vectors have been computed. The detail of this technique will be discussed later. During the process of the discrete wavelet transform, the maximum pixel value of each band is also collected and written to registers. These values will be used to improve the normalization performance by eliminating the unnecessary software iteration for calculating the maximum value of the normalization process.

The digital design is written with 32-bit fixed-point arithmetic, where the rightmost 10 bits represents fraction. This decision is based on the I/O bus of the Pilchard system and the data values found in the application arithmetic process by running the C++ version of the program. 32-bits covers almost all of the data values' range while providing a hundredth decimal fraction precision. The I/O port uses a 64-bit width bus, so it also fits two 32-bit data perfectly. However, smaller bit widths were also considered. Even with a 16-bit width, it only covers about 75 percent of the data values within the original software computation.

The blockRAM used in this project has a data bus width of 64-bit and depth of 256. The working address bus is 8-bit and takes 2 clock cycles to execute each read and write command. See figure 2-3 and 2-4. This in turn became the hardware limitation of the design. Consider there are 65 bands in the data, each with a resolution of 460*400, yielding 11,960,000 pixels. The first discrete wavelet transform operates on the data in groups of 4 pixels, which will yield 2,990,000 operations, and after being downsizing by 2 that leaves 5,980,000 pixels for the next set of transforms. Recall that in the 2-dimensional discrete wavelet transform, the operation is performed on both the rows and the columns of the data. All together, the digital design needs to take in 17,940,000 pixel values and performs 4,480,000 operations for each set of image data. Clearly, not all the data samples can be inputted onto the Pilchard at same time to perform even one complete discrete wavelet transform. Since 32-bit pixel values are used, it takes four clock cycles for read and four clock cycles for write for each discrete wavelet transform operations, plus a number of clock cycles to perform the operation. Since the read/write ports limit the throughput of the data flow, a pipeline has been designed to operate the read/write port at maximum frequency in order to optimize the overall performance.

4.4.1 Pilchard Design Flow

To create a functional system efficiently, several design cycles are required. The hardware design flow diagram shown in figure 4-2 illustrates all the steps in this project. These steps are iteratively implemented and verified until a stable functioning system is produced to the user specifications.

There are two design verification steps in this design cycle. The first verification is the functional simulation of the design logic, which is done before synthesizing the design. The second verification is in-circuit verification and is performed by downloading the bit-stream onto the Pilchard board and using interface software to verify the system behavior. The traditional post-layout simulation for the Pilchard entity was not used because it would require taking the back-annotation of a fully routed design and applying timing information to perform a functional simulation, however, the behaviors of many signals

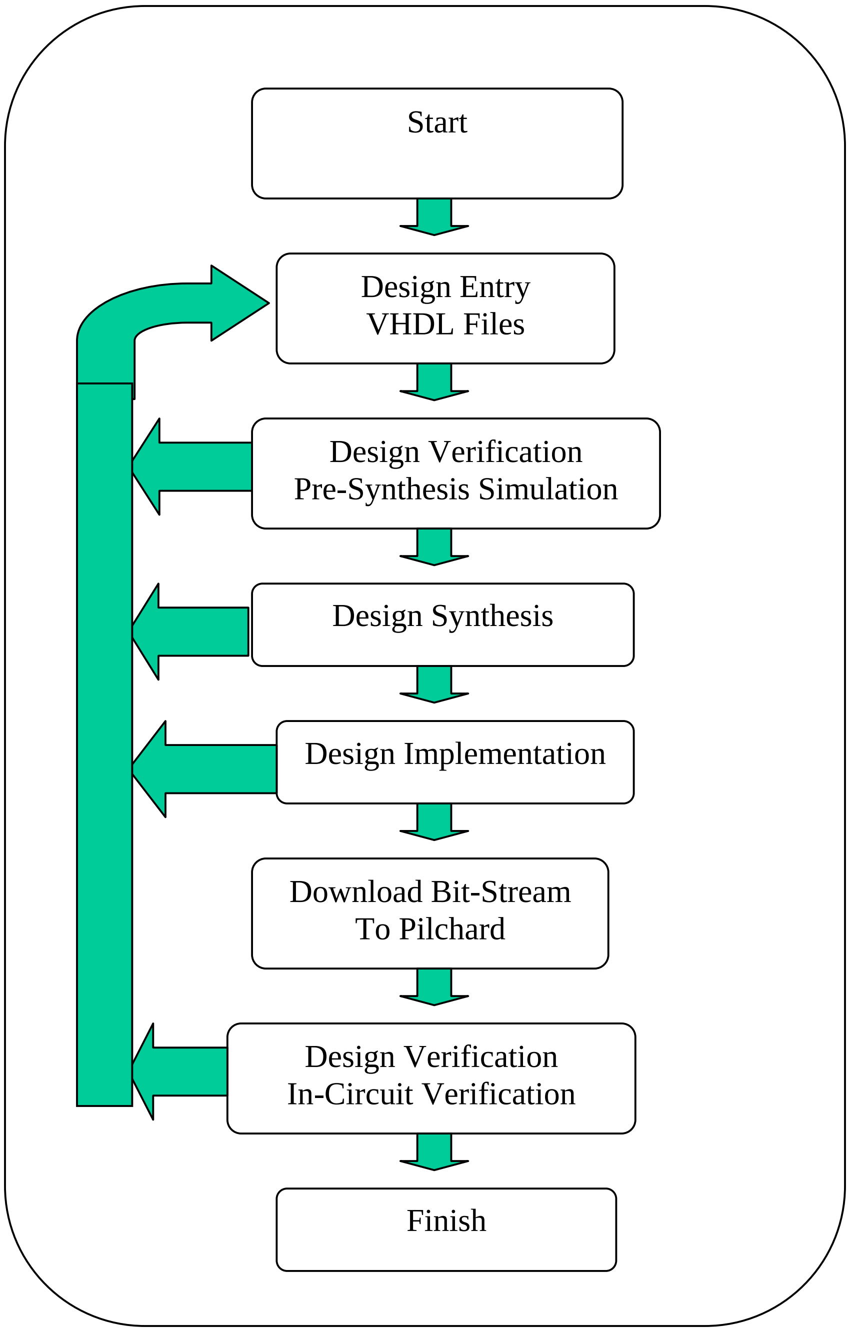


Figure 4-2: Digital Design Flow

in the top entity was unknown. Thus, the in-circuit verification became the only verification after the synthesis process.

4.4.2 Design Entry

VHDL was used in this project to develop a partial discrete wavelet transform function and a partial normalization process. This section discusses two main topics, the system components and structure, and functional behaviors of each implemented function. While the first sub-section focuses on the high-level hierarchy, the latter one is a more in-depth description of each function.

4.4.2.1 System Components and Structure

Because the Pilchard platform is used, the top-level hierarchy begins with the VHDL file “pilchard.vhd” that cames with the platform, which was developed by the Chinese University of Hong Kong. Within it is the VHDL file, “pcore.vhd,” which is used as wrapper file that allows the user to design, and an IP core to interface with the Pilchard board. The “parith.vhd” is the top level of the user design files, which consists of two other behavior components, the “max.vhd” and the “fxmult.vhd.” It is inside the “pcore.vhd” along with the source file, “dpram256_64.vhd,” of dual port Block RAM generated from the Xilinx Core Generator. In this project, port A from the dual port BlockRAM is used to interface with the “pcore.vhd” and port B is used to interface with the “parith.vhd.”

The abstract view of the architecture is shown in figure 4-3. The true representation of the block diagram, “pcore.vhd,” is shown in figure 4-4. It is generated by importing the actual VHDL codes using the FPGA Advance Pro from Mentor Tools. Figure 4-5 is a similar type of block diagram of the “parith.vhd.”

4.4.2.2 Functional Behaviors

The function behaviors of each design modules are described in the order of which they appear in the hierarchy, starting from the highest level that first contains the user design files.

In “pcore.vhd,” it performs several important task that overseer the overall operation of the digital designs. One important task is to start and reset the “parith.vhd” module. At the raising edge of the clock, the “pcore.vhd” set the start signal for “parith.vhd” to one when the “write” signal is high and the data reads a value of four. This signal triggers the start process of the design. The reset signal is also triggered by the write signal and the

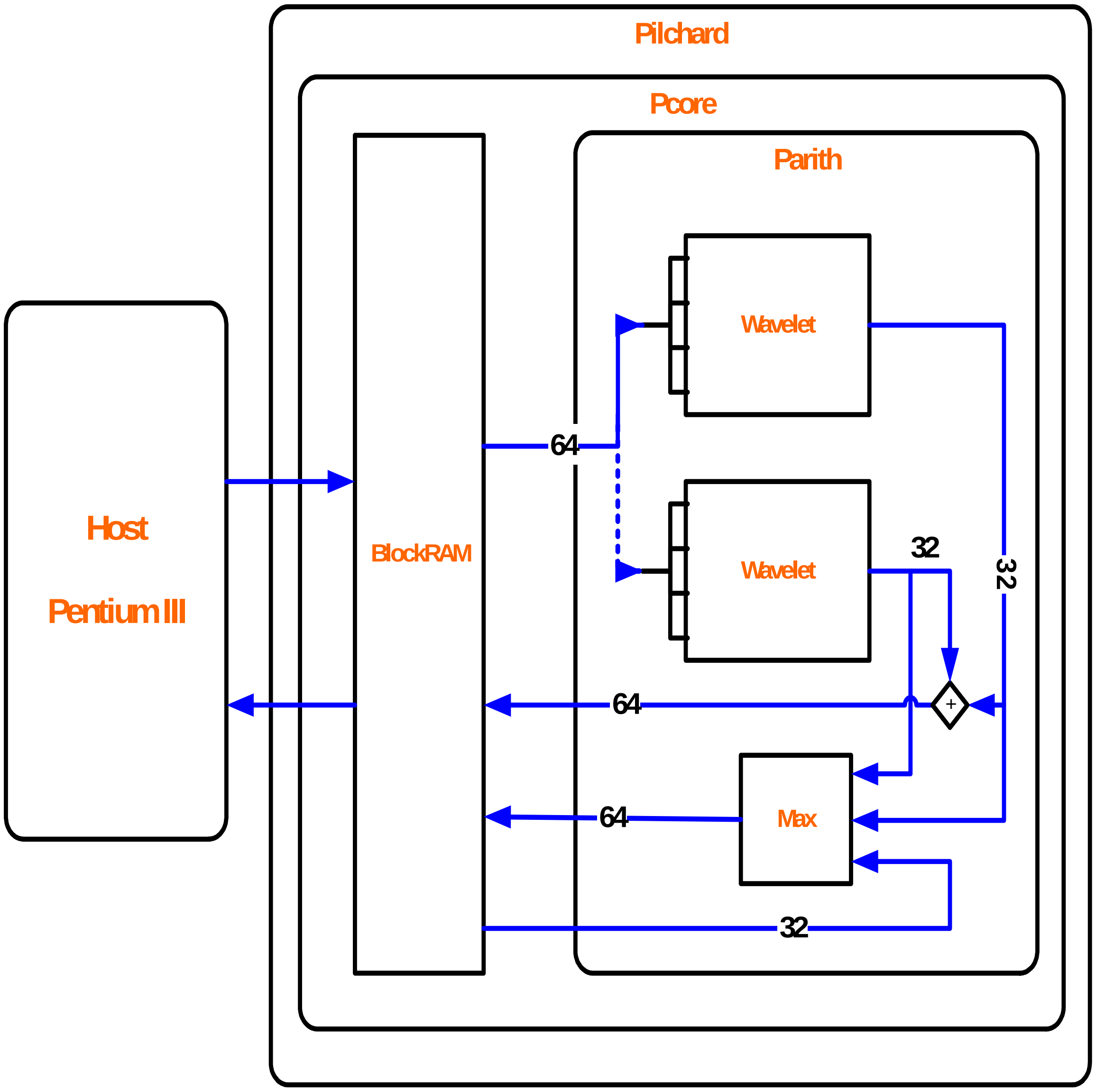


Figure 4-3: Abstract View of Overall Flow Block Diagram

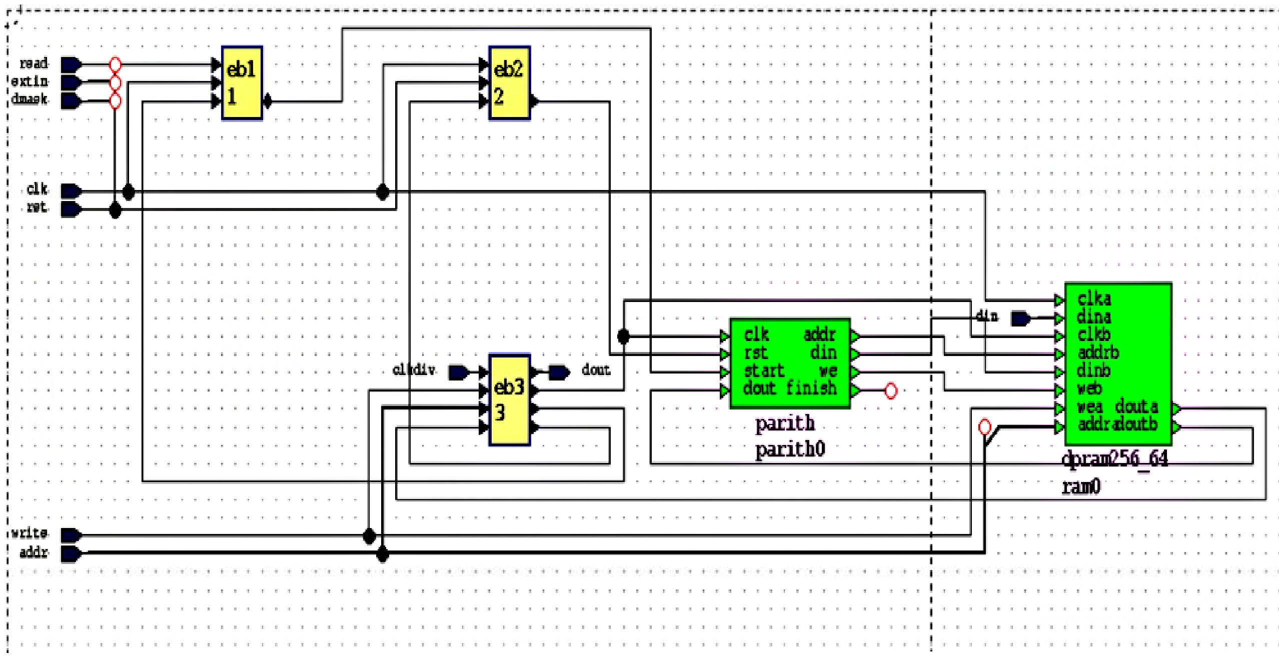


Figure 4-4: Generated Block Diagram of The "pcore.vhd"

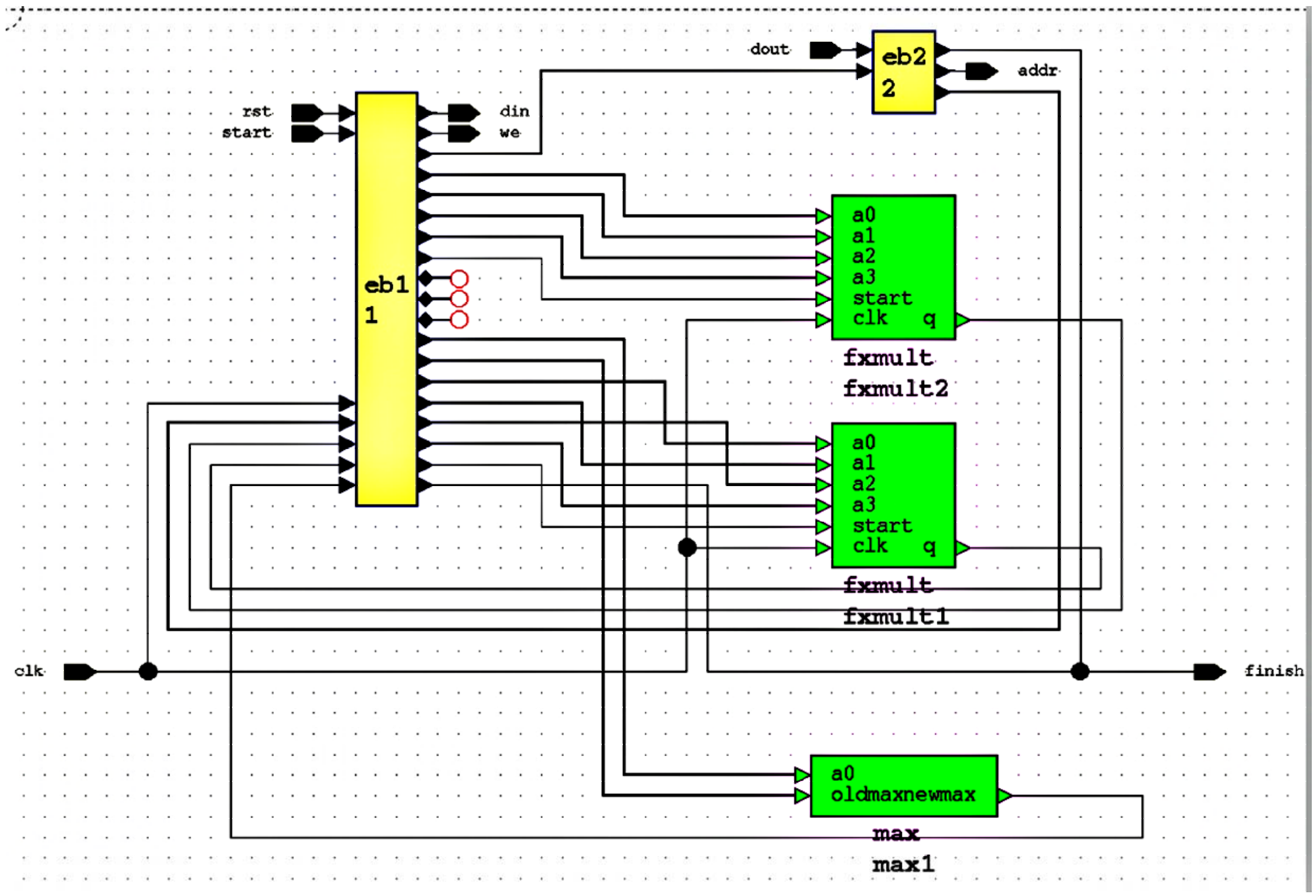


Figure 4-5: Generated Block Diagram of The "parith.vhd"

data signal. The module resets, when the “write” signal reads a value of five. This resets the state counter in “parith.vhd” and gives the ability to iteratively use the implemented design at run-time without re-download or reset the bit stream to the Pilchard board.

The “parith.vhd” is the main component of the design, where it handles the data retrieval, calls for the other two components to perform the transforms and maximum calculations, writes the result to the blockRAM and register, and positioning them within a pipeline. This module is operating under the clock-divider clock that is generated from the delay lock loop on the Xilinx Virtex1000E chip. The behavior of the “parith.vhd” is determined by a state counter. When the start signal from the “pcore.vhd” reads high, it triggers the state machine. From its dormant state, s_0, it moves up state by state until it completes its eleventh cycle and goes back to s_0.

The state machine is designed to maximize the frequency of the read/write ports of the blockRAM. Consider the two models of pipelines described in table 4-3 and 4-4. Model-A represents the minimum clock cycles and instructions sets needed to perform a complete iteration of the digital implementation. Here, each “iss” represents a single read request for an address location; “read” represents reading the 64-bit from that address location and splitting it to two 32-bit data; “wt” and “max” represents a set number of clock cycles of each corresponding computation; and the “write” in this model represents writing both the current maximum value and the discrete wavelet transform result into a single 64-bit address location. The process is same for instruction set i + 1. However for i

Table 4-3: Parith - Pipeline Model A

	Clock number										
Instructions	1	2	3	4	5	6	7	8	9	10	11
i	iss read	iss read	read	read	wt	max	iss write*		write*		
i + 1			iss read	iss read	read	read	wt	max	iss write*		write*

Table 4-4: Parith - Pipeline Model B

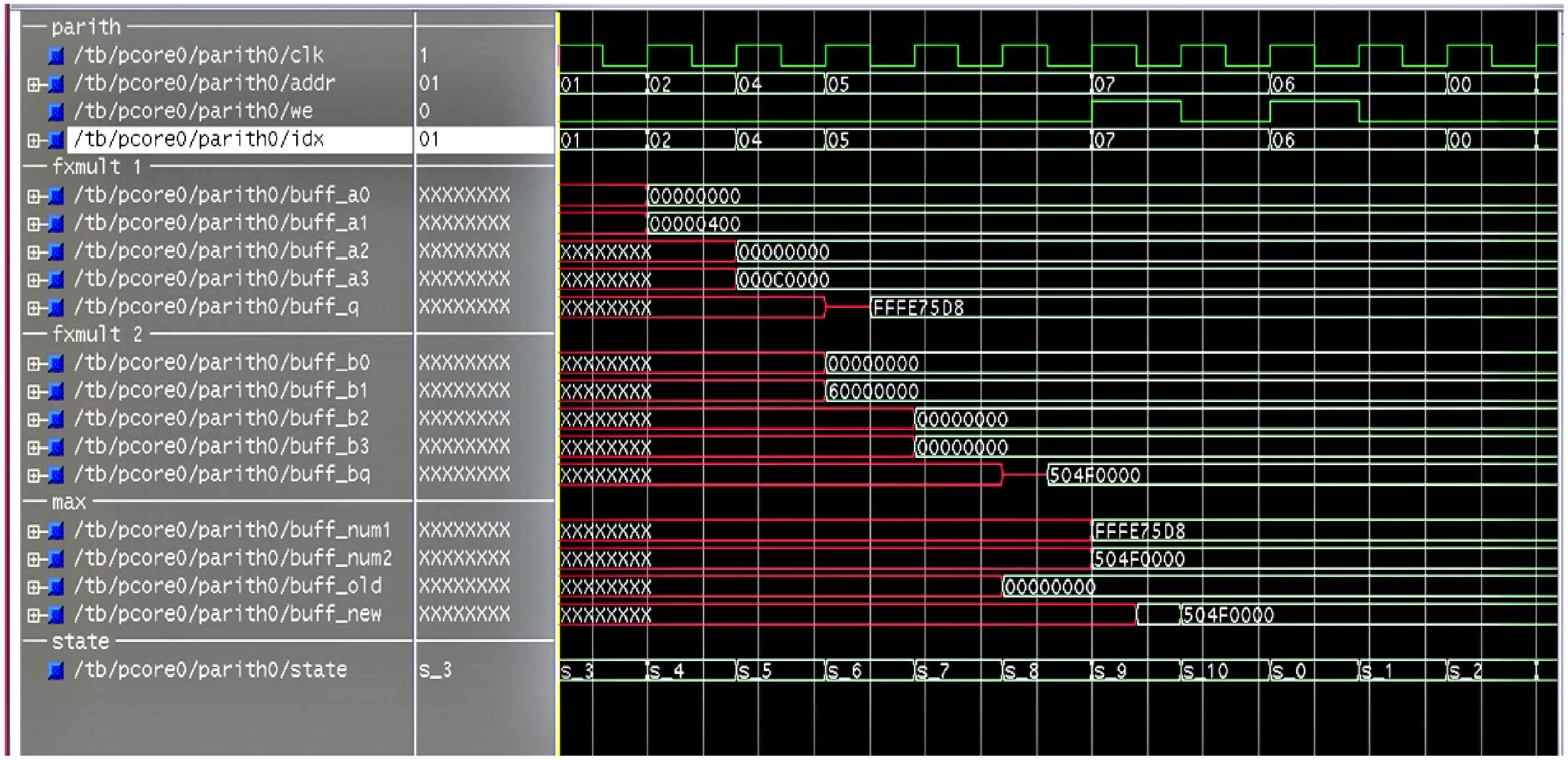
	Clock number										
Instructions	1	2	3	4	5	6	7	8	9	10	11
i	iss read *1*	iss read *1*	read *1*	read *1*	wt *1*	max *1*					
			iss read *2*	iss read *2*	read *2*	read *2*	wt *2*	iss write * /max	iss write max	write *	write max

+ 2 instruction, the first set of instruction must start at the 12th clock cycle due to the availability of the read/write head. With this model, a complete iteration of operating 4 data uses seven states and 2 iteration of operating 8 data uses 11 states. The difference with model B is that it operates on two sets of data at a time and group the results from each of the two wavelet computations together into an address location and also write the maximum value on a separate stage to another address location. However, here only half of the 64 bit data is used in maximum value address location, the other half of address location is vacant. With the same clock cycles used per every two computations, model B provides an additional data storage, thus this model was chosen. The vacant data memory was later used to store counters, which also signal to the host PC when a transformation is completed.

In model B, the instruction set can be viewed as an eleven-stage state machine. In the initial two states, a read command is requested for the first two location of the memory address, zero and one. During the third and forth state, the data are read from first two locations. Since four 32-bit numbers are read, the first wavelet transformation is ready to begin in the sub-module, fxmult.vhd. Before moving onto the next state, another read command is issued for the second and third address location. In the fifth and sixth state, the second sets of data are feed into another fxmult.vhd sub-module. Also, in the sixth states, the computation for the discrete wavelet transform is finished for the first set of data. They are then ready to be feed into the module, max.vhd, and to be compared against the current maximum value, which is read from the address location six in the seventh state. During the eighth state, the results from the second discrete wavelet transform is ready, and together with the result from the first set of computation, the two 32-bit answer are write out together. Also in this state, the maximum function starts to compare the values among the first, the second answers of the transform modules and the current maximum value. The new maximum is the write to the memory location six, during the ninth and final state. Figure 4-6 shows first cycle of the pipeline. The red lines denote the undefined signals. In a way, it helps to provide a more noticeable illustration of how each signal is progressed at each state denote by “s_0” “s_1” “s_2”… When the maximum value is ready to be written into the blockRAM via “din”, it is concatenated with a signal “count.”

The remaining two components, “fxmult.vhd” and “max.vhd,” both are the lowest level modules. The “fxmult.vhd” computes the low-pass filter component of the discrete wavelet transforms on the four input values. The functionality of the fxmult.vhd is actually performing a sum of four products. This function was implemented and verified using the sum of product design ware from Synopsis and a self implemented function. The implemented function, in the end, yield a better timing constraint after the place and route process, so it was used.

While the coding for the implemented function may appears to be simple, it actually represent a sound solution to what otherwise might have been a complex algorithm. Recall the algorithm for discrete wavelet transform, each of the data is multiplied by a



parith
/tb/pcore0/parith0/clk 1
/tb/pcore0/parith0/addr 01
/tb/pcore0/parith0/we 0
/tb/pcore0/parith0/idx 01
fxmult 1
/tb/pcore0/parith0/buff_a0 XXXXXXXX
/tb/pcore0/parith0/buff_a1 XXXXXXXX
/tb/pcore0/parith0/buff_a2 XXXXXXXX
/tb/pcore0/parith0/buff_a3 XXXXXXXX
/tb/pcore0/parith0/buff_q XXXXXXXX
fxmult 2
/tb/pcore0/parith0/buff_b0 XXXXXXXX
/tb/pcore0/parith0/buff_b1 XXXXXXXX
/tb/pcore0/parith0/buff_b2 XXXXXXXX
/tb/pcore0/parith0/buff_b3 XXXXXXXX
/tb/pcore0/parith0/buff_bq XXXXXXXX
max
/tb/pcore0/parith0/buff_num1 XXXXXXXX
/tb/pcore0/parith0/buff_num2 XXXXXXXX
/tb/pcore0/parith0/buff_old XXXXXXXX
/tb/pcore0/parith0/buff_new XXXXXXXX
state
/tb/pcore0/parith0/state s_3
01 02 04 05 07 06 00
01 02 04 05 07 06 00
00000000
00000400
XXXXXXXX 00000000
XXXXXXXX 000C0000
XXXXXXXX FFFE75D8
XXXXXXXX 00000000
XXXXXXXX 60000000
XXXXXXXX 00000000
XXXXXXXX 00000000
XXXXXXXX 504F0000
XXXXXXXX FFFE75D8
XXXXXXXX 504F0000
XXXXXXXX 00000000
XXXXXXXX 504F0000
s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_10 s_0 s_1 s_2

Figure 4-6: Parith Waveforms

fraction coefficient then accumulated together to produce the answer. The obvious solution is to multiple the coefficients, however, binary representation of decimal number and arithmetic with another decimal number of different precisions can lead to loss of precision when using VHDL fixed points. Instead multiplying the fractional coefficient, h1, h2, h3, and h4, these constants are shift to the left and treated as integers. When the sum of products computation finishes, the result is then adjusted by shifting the decimal place to the right.

The detail of this implemented can be explained through figure 4-7. In the system's fixed-point representation, 10 bits of the binary value of the data is set for the decimal precisions where as 13 bits of the coefficient is used for the fraction value. The result of the arithmetic is a 45 bit binary number with 23-bit in decimals and offset by 13 bit precision places. In order to balance the offset, the result is then shifted by 13 places to the right and discarded. The module uses signed arithmetic where the first bit represents the sign bit. If it is negative, the algorithm uses the 2's compliment to convert the format. This functionality is included in the standard library of ieee.std_logic_signed.all.

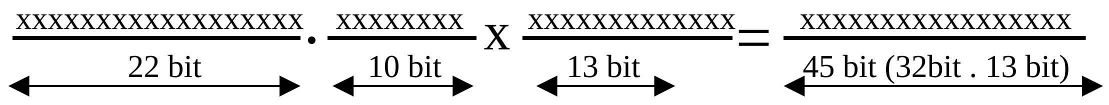

xxxxxxxxxxxxxxxxxxx . xxxxxxxxx x xxxxxxxxxxxxxx = xxxxxxxxxxxxxxxxxx
22 bit
10 bit
13 bit
45 bit (32bit . 13 bit)

Figure 4-7: Fixed Point Arithmetic in Parith.vhd

The “max.vhd” also uses one process statement and a set of if else nested function to compare the max value for two input signals. The arithmetic in this module is also signed and using two’s compliment. Observe from figure 4-6, the four signals within the last set of dividers represents the maximum comparison. Notice when FFFE75D8 is compared with 504F0000, the max value yields 504F0000, because FFFE75D8 is a negative number.

4.4.3 Hardware Simulation

Similar to the compile process, the pre-synthesis simulation is also performed using the Mentor Graphics Modelsim SE VHDL5.6a; and all of the VHDL files are compiled in a hierarchical order. This is the first of the two design verifications in the digital design process. In this stage, the simulation of the design is being tested to verify that the logic in the functions behave correctly. Since it is a pre-synthesis simulation, the timing information is unavailable and is not needed at this time. In order to simulate the design, a test bench is applied to obtain the simulation waveform for signals in the design. Since the blockRAM is an IP core that was generated using Xilinx Core Generator, the XilinxCoreLib is required to run the simulation. The simulation can be done using either the GUI interface or the by running a script file. Two additional files that were used in this simulation are the wave.do and stim.do files. These file contain the signals, formats, and run-time information need for the simulation. A copy of such script is shown in figure 4-8.

The top-level of the hierarchy for this testbench is shown in figure 4-9. The testbench is only used for the simulator and is not to be synthesized. The testbench in this project has a loop that contains three sets of testing data to emulate the behavior of the software interface. The data is often modified to check the functional behavior of the fixed-point arithmetic. The simulation results are found in the previous chapter. By inspecting these waveforms, the functions are verified to be correctly simulated.

4.4.4 Design Synthesis

The Synopsys' FPGA Compiler II was used to synthesis the design work. Since the Pilchard uses the Xilinx VirtexE chip, XCV1000E-HQ240, several options needed to be selected in order to assurance functional operation. When creating the new project under the synthesis tool, no VHDL or EDIF files are used for the IP cores. Those are generated using the Xilinx Core Generation from the Pilchard package itself.

```
#!/usr/bin/csh -f

source ~cad/.cshrc
mentor_tools

# The following commands are associated with compiling the RAM which
has been generated using Xilinx's Coregen

vcom -work XilinxCoreLib
/sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/ul_utils.vhd
vcom -work XilinxCoreLib
/sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/mem_init_file_pack_v3_1.vhd
vcom -work XilinxCoreLib
/sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/blkmemdp_pkg_v3_1.vhd
vcom -work XilinxCoreLib
/sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/blkmemdp_v3_1_comp.vhd
vcom -work XilinxCoreLib
/sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/blkmemdp_v3_1.vhd
vcom -work XilinxCoreLib
/sw/Xilinx4.1i/vhdl/src/XilinxCoreLib/blkmemdp_v3_1.vhd


vcom -work work fxmult.vhd
vcom -work work max.vhd
vcom -work work dpram256_64.vhd  # RAM module
vcom -work work parith.vhd       # Parithmetic Module
vcom -work work pcore.vhd      # Top Module
vcom -work work tb.vhd         # Test Bench

# Simulating using ModelSim


vsim  -coverage tb  -do  wave.do stim.do

# vsim - Command to open Modelsim
# wave.do - File that opens Parith and Pcore signals in Modelsim
# stim.do - File to run the Simulation
```

Figure 4-8: Scripts for Compile and Simulate VHDL files

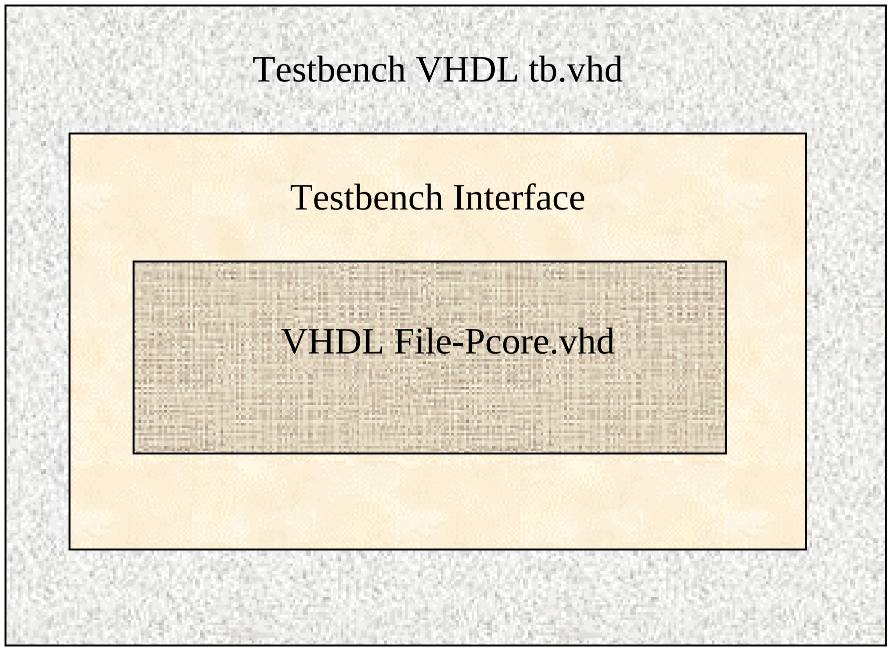
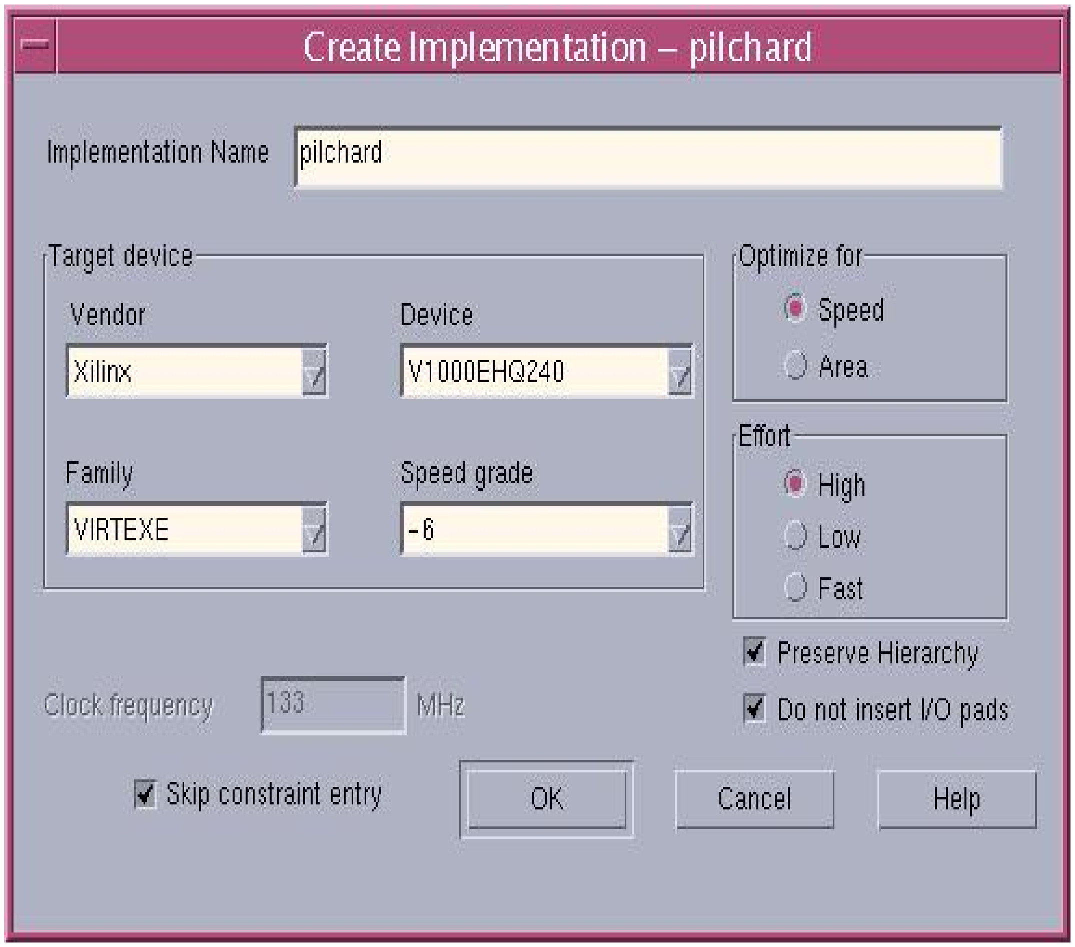


Figure 4-9: Testbench Hierarchy

Figure 4-10 illustrates the correct options that should be selected when creating the implementation of the chip from the GUI version of the FPGA compiler. Also, when creating the implementation for the Pilchard design, the existing I/O pads should be used instead of the default option, which automatically insert necessary I/O pads. Instead using the GUI, scripts can also be used for synthesis. A copy of such script is shown on next page in figure 4-11.



Create Implementation – pilchard

Implementation Name: pilchard

Target device
Vendor: Xilinx
Device: V1000EHQ240
Family: VIRTEXE
Speed grade: -6

Optimize for: [✓ Speed] Area

Effort: [✓ High] Low Fast

[✓ Preserve Hierarchy]

[✓ Do not insert I/O pads]

Clock frequency 133 MHz

[✓ Skip constraint entry]

Figure 4-10: Create Implementation Options Using Synopsys FGPA Compiler

```
set proj syn
set top pilchard
set target VIRTEXE
set chip pilchard
set export_dir export_dir
set device V1000EHQ240
set speed -6

exec rm -rf $proj
create_project -dir . $proj

open_project $proj

proj_export_timing_constraint = "yes"

default_clock_frequency = 100

add_file -library WORK -format VHDL pcore.vhd
add_file -library WORK -format VHDL pilchard.vhd
add_file -library WORK -format VHDL parith.vhd
add_file -library WORK -format EDIF dpram256_64.edn
add_file -library WORK -format VHDL fxmult.vhd
add_file -library WORK -format VHDL max.vhd

analyze_file -progress

create_chip -target $target -device $device -speed $speed -frequency
100 -module -name $chip $top

current_chip $chip

set opt_chip [format "%s-Optimized" $chip]
optimize_chip -name $opt_chip

list_message

report_timing

exec rm -rf $export_dir
exec mkdir -p $export_dir
export_chip -progress -dir $export_dir -no_timing_constraint

close_project

quit
```

Figure 4-11: Synthesis Script

```
#!/bin/csh -f

source /sw/Xilinx4.2i/settings.csh
ngdbuild -p V1000EHQ240-6  $1.edf
map $1.ngd
par $1.ncd -w $1_r.ncd
trce -s 6 $1_r.ncd
ngdanno $1.ncd
ngd2vhdl -w $1.nga time_sim.vhd
```

Figure 4-12: Place and Route Script

4.4.5 Place Route and Bit Streams

The place and route process is managed using Xilinx Design Manager. Several files are needed for the place and route process, the "iob_fdc.edif", which is the net-list file for the I/O blocks used in "Pilchard.vhd"; the net-list file for the dual port BlockRAM; and finally but not least, the user constraint files that contains the information regarding the physical pin connections and the timing specification are required. The listing in figure 4-12 is the script used for place and route.

4.4.6 In-Circuit Design Verification

After the bit-stream is downloaded onto the Pilchard board, a C program is used to perform the in circuit verification. The C program feeds in the data for the discrete wavelet transform and stores the result in a data file. The library file “iflib.c” has a set of APIs to handle the data transfer from the software to the FPGA. This library file was compiled together with the C design files.

To interface with the Pilchard board, a memory map to the hardware was created at the beginning. The data that feed to the Pilchard board is located in 65 different data files; each contains its respective band image. The C software opens one file at a time, writes eight 32-bit data values to the address location 0, 1, 2, and 3, follows by a write command at address 4, which triggers the start signal for the digital logics. The two results from the wavelet transform are written to location 7 and the counter and maximum values are written to location 6. Normally, a “hand-shake” method is needed to verify when the correct values are received, however, with the streaming technique implemented with this project, the digital system actually computes faster than read64 function in the iflib.c library files. So when the start signal is issued in the host program, the next instruction reads the answer back.

Another 16-bit version of the system was also created for result comparison purposes. It uses three address locations for the entire computation cycle. It writes four data values in each of the first two address locations. The two wavelet-transform results, along with the counter and the maximum value are written back in one address location. The differences with 32-bit version is that it uses six address locations, four for two sets of data inputs, one for the two results, and another one for the maximum and counter. The address location used is proportional with the bit width used in these two cases.

4.5 Chapter Summary

This chapter focuses on the design methodology that was used in this thesis. From this chapter, the design processes are revealed and each is discussed in detail. Next chapter looks at how the results are compared based on these design works.

Chapter 5. Result and Discussion

This chapter is divided into three sections. In the first section, the results from this work is studied and compared. The second section investigates the difficulties during the process. The conclusion and future thought are suggested in the last pages.

5.1 Results Comparisons

The configuration file for the Pilchard's on-board Xillinx E chip had been successfully implemented. The original design system of the 32-bit worked correctly and met the design goals. A 16-bit system was also implemented for testing purposes. In this section, the comparison between the two versions will be examined, in the areas of resource consumptions and performance.

5.1.1 Resource Comparison

The resource data are acquired after the Xilinx's place and route process and are tabulated into table 5-1. The number of the used slices denotes the logic resource consumption. This is almost a direct portion to the number of bit that is used. As the bit

Table 5-1: Resources Used During Place and Route

Resource	Number Used		Total Amount	Percent Usage	
	16-bit	32-bit		16-bit	32-bit
Slices	677	1280	12288	5.5%	10.4%
Input LUTs	1026	2015	24576	4.2%	8.2%
Bonded IOBs	104	104	158	65%	65%
Block RAMS	4	4	96	4%	4%
GCLKs	2	2	4	50%	50%

width doubles from 16 to 32, the number of slices increased by almost two-fold. This occurrence is expected since each added bit width corresponding to adding a logic block of the same operation. The design logics that are independent to the bit width of the input data, such as the state signals or flags, stay the same for both design editions and it is the main reason why the number of slices is not completely doubled. This argument is also true for the like up tables or LUTs. The BlockRAM usage represents the resource memory used for the data process. Since the design system uses a streaming technique, only a minimum number of test data are loaded to the system at a time. The increase of the bit width on two sets of data values does not make any significant impact on the BlockRAM. The remaining of the resources, such as Bonded IOBs, GCLKs, and among the others that are omitted from the list, such as DLLs or Startups, are hardware dependent to the Pilchard hierarch logic module. Since the user-design modules operate below this level, all of the resources in this category should remain the same for all design implementations.

5.1.2 Performance Comparison

In generally, when evaluating digital system performance, two aspects of the system are inspected upon, the system throughput and the system latency. However, in both versions of the hardware design, the throughput of both systems is processing the same two sets of data at a time. Although the bit-width varies, the number of sampled data still remains the same. Thus, the performance evaluation in this project will be based on the latency aspect of the systems only. Starting this section, the performance of the hardware system is revealed, then an overall comparison between the software and hardware implementation will be discussed, and followed by the assessments between the two implementations

From the Place and Route process, the maximum frequency was determined to be 21.62 Mhz and 14.314 Mhz for 16-bit version and the 32-bit version respectably. To ensure the Pilchard system works correctly, the clock divide is set to label five and eight while the actual clock runs at 100Mhz. This yields 12.5Mhz for the 32-bit version and 20Mhz for the 16-bit version.

The Pilchard runtime is calculated based on the average of ten runs of 4,485,000 iterations that are needed to cover the entire 65 bands of the hyper-spectral imaging data. The average run-time with loading the data is about 4.09 seconds for the 16-bit version and 6.54 seconds for 32-bit version; and 2.14 second for the 16-bit version and 2.15 for the 32-bit version for without counting the load time for writing in the data to Pilchard. The run time for both of the versions without load time is very adjacent to each other. This was expected result from using the streaming method. The calculations on-board the Pilchard system is actually performing faster than the time it take the host program to write the starting signal and reading the answers. In comparison with the load, the 32-bit version runs slower than the 16-bit version, because there is more read/write instruction set on the host side. The run-time comparisons between the 16-bit and 32-bit version is shown in table 5-2 and figure 5-1. One thing worth of mentioning is the precision of how the run-time is calculated. The calculation is preformed on the host side using the gettimeofday function in the standard C library, sys/time.h. This function, however, only ceiling to nearest microseconds. So when timing one instruction set such as write64, the difference of the gettimeofday functions before and after the write64 function is 1 microsecond. With the same method, two consecutive write64 functions are also found to be 1 microsecond. Clearly, there is a threshold point that shows when the gettimeofday function can correctly shows the run-time calculation. Figure 5-2 investigates this scenario. It shows the Run-Time per iterations for both the 16-bit version and the 32-bit version when the load times are included in the performance evaluation. From the data, this figure shows that the threshold for the precision of the gettimeofday function can be reached around 1000 iterations.

Table 5-3 shows the overall comparison among the three implementations. The normalization process is broken down to two steps in order to better illustrate the correct comparison; since only the first step of the normalization was implemented on the Pilchard system. Without counting the load time, the overall speedup achieved by using the Pilchard design is about a factor of 2.56 and 2.55 for the 16-bit and 32-bit respectable. When taking into the consideration of load in the data onto the Pilchard, the speedup for the 16-bit is by a factor of 1.34. The 32-bit version is actually slower than the C++ version by 0.84 seconds.

Table 5-2: Run-Time Comparison Between 16-bit and 32-bit Version

	16-bit (seconds)		32-bit (seconds)	
Trials	**w/o load**	**w/ load**	**w/o load**	**w/ load**
1	2.13	4.1	2.16	6.51
2	2.15	4.04	2.14	6.55
3	2.14	4.1	2.16	6.57
4	2.12	4.11	2.12	6.53
5	2.14	4.09	2.16	6.5
6	2.15	4.11	2.13	6.58
7	2.12	4.07	2.16	6.51
8	2.14	4.1	2.14	6.49
9	2.16	4.1	2.13	6.59
10	2.17	4.08	2.15	6.53
average	2.142	4.09	2.145	6.536

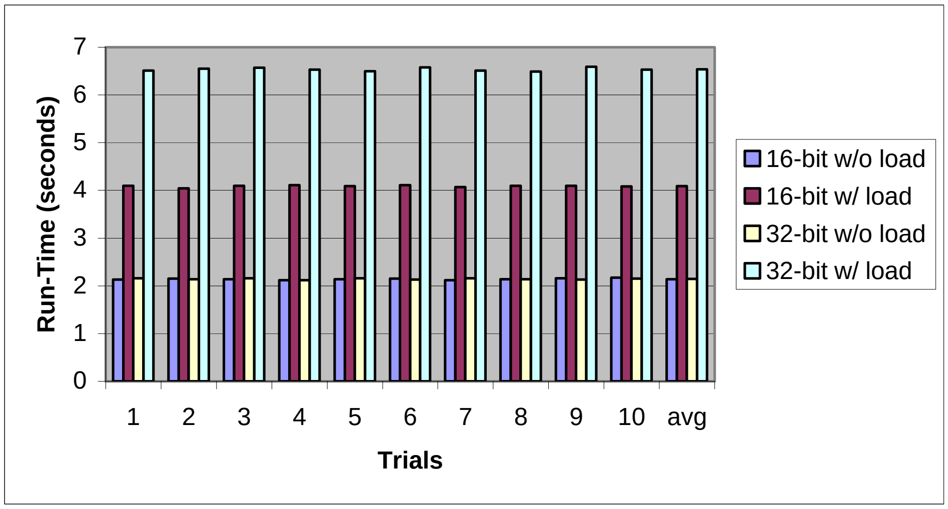


Figure 5-1: Run-Time Comparison Between 16-bit and 32-bit Version

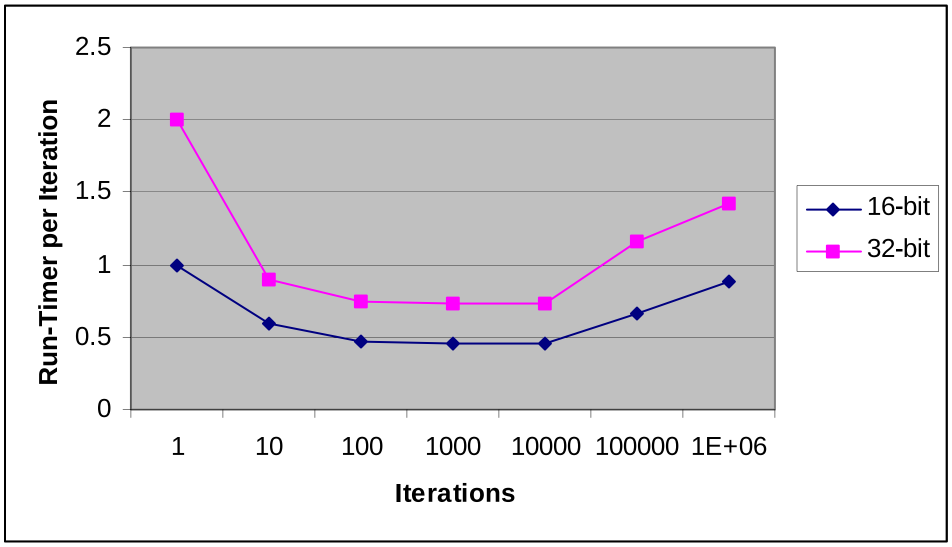


Figure 5-2: Run-Time Threshold (w/ load)

Table 5-3: Overall Run-Time Comparison

	Matlab* (Second)	C++ (Second)	Pilchard w/ read write (without load / with load)	
			16-bit	32-bit
2D Wavelet Transform	35.85	3.44	2.14 / 4.09	2.15 / 6.54
Normalization step1 : find max	2.44	2.04		
Normalization step2 : divide by max	3.485	2.05	-	-

5.1.3 Parallel Computing Results

After the digital design is successfully tested using a single pilchard system, the parallel aspect of the HPRC architecture was explored. The program responsible for distributing and controlling the clients and tasks is called BioGrid [33]. In BioGrid, there are three types of workstations: server, workers, and clients. The server hosts the Java application and distributes the tasks sent in by the client to any number of workers.

In this work, the chicken tumor problem is tested with BioGrid using one, three, and five Pilchard systems. We used the one case to measure overheads. The Pilchard.bit files are preloaded onto each of the reconfigurable units before running BioGrid, thus the run-time results from this test exclude the load time. When running the parallel test using three Pilchards, the workloads on the three systems are different. Since 65 bands cannot equally divide into three equal sections, the first two Pilchard machines, each process 22 bands of total bands and third machine processes the remaining 21 bands. The result was 1.84 seconds, a speedup of 1.17 comparing to the result of a single Pilchard platform, which is 2.15 seconds. When the load is equally divided among five Pilchard machines, the result was 1.37 seconds, a speedup of 1.58.

At first glance, these results may not be the expected performance from a high performance parallel architecture. The reason is the extra overheads exerted by the BioGrid. When we run 13 bands through a single system, the runtime is 0.43 seconds, however, when BioGrid is used, the run-time is 1.01 second. It required an additional 0.58 seconds. Because of this overhead, a similar difference of 0.57 seconds is induced with 22 bands. If this overhead is zero, the speedup would be 3.69 (1.01 * 5 / 1.37) when using five parallel systems. Similarly, the speedup without this overhead would be 2.28 for using three machines. These results are listed in tables 5-4 and 5-5. For larger problem sizes, we anticipate that the overheads would have less impact on speedup.

Table 5-4: BioGrid Results with Single Pilchard Machine

Bands	**Runtime Using Pilchard (sec)**	**Runtime Using Pilchard with BioBrid (sec)**
65	2.15	2.92
22	0.72	1.4
13	0.43	1.01

Table 5-5: BioGrid Results with Multiple Pilchard Machines

Bands	Runtime with Numbers of Pilchard Machines (sec)		
	1	2	3
65	2.92	-	-
22	1.40	1.84	-
13	1.01	-	1.37

5.2 Difficulties Encountered

The development of this project is not without any difficulties. Problems, challenges and pre-mature considerations were struggled at numerous prospects of the design process, including hardware limitation, programming techniques and styles, and overall design perceptions. This section addresses each of these problems and followed by the methods that were used to overcome them. These are the valuable lessons learned and the awareness of these points will bring more success and faster prototyping in the future implementations of familiar kinds.

A major problem with design hardware implementation is to meet the timing constraints. A number of times, the digital systems functioned correctly under the simulation; but after the place and route process, it generated a clock cycle twice or three times slower than the minimal clock cycles required. An important lesson learned here is to used a good programming style and maximize the usages of concurrent processes and pipelines. The following guidelines, “HDL Coding Guidelines,” by Damjan Lampret and Jamil Khatib, were used in this project at appropriate situation [32].

For clocks:

- Use as few clock domains as possible in any design.
- Do not use clocks or resets as data or enable and vice versa.
- Clock signals must be connected to global dedicated resets or clock pin on an FGA or CPLD.

For timing optimization:

- Use synchronous design to avoid problems in synthesis, in timing verification and in simulations.
- Avoid using latches.
- Include all signals that are read inside a combinational process in its sensitivity list.
- Ensure variables are assigned in every branch of a combinational logic process to prevent inferring of unwanted latches.

For general rules

- In RTL, never initialized register in their declaration use reset logic instead. The initialization statements cannot be synthesized.
- Write finite state machines in two always blocks – one for sequential assignments and other for combinational logic.
- Compare buses with the same width.
- Avoid using long if-then-else statements and use case statements instead. This helps to prevent inferring of large priority decoders and makes the code easier to be read.

By simply following these good programming techniques, an early work of a design system had an increase of clock cycle around 30Mhz. This speedup was achieved without altering the functional logic behavior of the design modules. Another way to improve the timing constraints is to use higher place and route effort levels. When multiple place and route process are performed only the implementation with the best optimization score should be kept and tested.

Besides from the programming techniques and style, hardware also contributed to some difficulties. The baselines for generating the profiles are from different computers, more importantly, with different processor speed. The Pilchard systems are on a 1Ghz hosts and is solely dedicated to Pilchard-related researches. Thus, the Matlab is not available on these machines and the profiles had to be run on different PCs. The difference in the baselines contributes a margin of error when comparing the results.

5.3 Conclusions

The goal of this project was to implement a digital system that performs the functionalities of the bottlenecks presented in C++ profile combined with the enhanced performance. The pre-synthesis simulation and the in-circuit simulation prove the correct functionalities of the design and the interface software show that it had the best speedup comparing with the software counterparts. The success of the Pilchard implementation demonstrated the potential of performance improvement from using of such platforms.

The study also showed that when computing data on a digital system with host side interface, the streaming technique increases the run-time as a direct proportion of the read write functions used. On a positive note, the streaming process can be easily configured to adapt multiple levels of parallel computing. The task can be broken down at band levels, where each Pilchard system can be responsible for a certain number of bands. If more system are available, each band process can be divided multiple data sections that each section can run on its only hardware system. The streaming technique used provide the scalabilities at multi-levels but at the cost of lower speedup, penalized from the read write execution during run-time.

From a parallel processing perspective, this design did not achieve the desired speedup compared to a single machine. This setback is mostly due to the overhead with BioGrid. With more optimization to BioGrid or the use of other parallel programming environment, this overhead may be reduced. Our experiments had short runtimes, with an execution time less than two seconds. Of this time, 0.6 to 0.7 seconds was overhead from BioGrid. With larger input image data, the application will require more time to perform the wavelet transform. We do not expect the BioGrid overhead to significantly worsen, which will minimize the effect of this overhead on the total performance.

5.4 Future Work

The design work in this project provides the foundation for the high performance parallel computing. At least two tasks can be explored to further increase the speedup of the application.

First is to overcome the hardware constraints. The blockRAM and the read/write port were the most important constraints affecting the result of the performance in this project. With better I/O interface and bigger RAM, a more effective algorithm method can be use. For example, the data vectors for each band image can be input onto the blockRAM before starting the computation. This way eliminates the need for the second cycle of input from the host side for the wavelet transform. The values can be kept in the memory until the final set of 200 by 240 answer is reached. This not only eliminates an entire cycle of I/Os but it also eliminates the host-side data management at each set of calculation. The use of better hardware is expected to improve the speedup dramatically.

Another important future work is to explore the implementation on high performance reconfigurable computers more extensively. The shared task program, such as the BioGrid, can be detailed to reduce the overhead that was shown in section 5.1.3. or explore the usage of other programs such as MPI or PVM. Also, more HPC tests should be performed over a larger network of workers, so a threshold can be determined for an optimized ratio of workers per application.

With exploring new design scheme, there will be many other issues that need to be dealt with, like scheduling, optimum resource utilization, modeling and performance analysis. Eventually, moving towards building a development system to efficiently utilize the processing power of such system is the goal.

Bibliography

[1] P.H.W. Leong, M.P. Leong, O.Y.H. Cheung, T. Tung, C.M. Kwok, and K.H. Lee, “Pilchard – A Reconfigurable Computing Platform with Memory Slot Interface”, *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM),* Rohnert Park, CA, April 2001.

[2] Xilinx, “Virtex-E 1.8 V Field Programmable Gate Arrays”, *Datasheet* (*DS022)*, March 2003. http://www.xilinx.com/partinfo/ds022.htm

[3] K.H. Tsoi, *Pilchard User Reference (V0.1),* Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, NT Hong Kong, January 2002.

[4] Randall Hyde, "The Art of Assembly Language Programming", January 200. http://webster.cs.ucr.edu/Page_AoALinux/HTML/AoATOC.html

[5] K.C. Chang, “Digital Systems Design with VHDL and Synthesis, An Integrated Approach”, Matt Loeb, May 1999.

[6] JICS, “Introduction to Parallel Processing”, Lecture Notes. http://www.jics.utk.edu/documentation.html

[7] Melissa C. Smith and Gregory D. Peterson, “Analytical Modeling for High Performance Reconfigurable Computers,” Proceedings of the SCS International Symposium on Performance Evaluation of Computer and Telecommunications Systems, 2002.

[8] Clyde H. Spencer, “Using Hyperspectral Imagery to Create GIS Layers,” 1995. http://www.biogeorecon.com/usinggis.htm

[9] Shu-Jen Steven Tsai, “Power Transformer Partial Discharge (PD) Acoustic Signal Detection Using Fiber Sensors and Wavelet Analysis, Modeling, and Simulation.” December, 2002, Blacksburg, Virginia.

[10] Rudolf K. Bock, “Data Analysis BriefBook,” Version 16, April 1998. http://rkb.home.cern.ch/rkb/AN16pp/AN16pp.html

[11] Z. Ye, P. Banerjee, S. Hauck, and A. Moshovos, "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Unit," presented at International Symposium on Computer Architecture, Toronto, CANADA, 2000.

[12] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," presented at IEEE Symposium on FPGAs for Custom Computing Machines, 1997.

[13] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "PipeRench: a Coprocessor for Streaming Multimedia Acceleration," presented at ISCA, 1999.

[14] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and T. R.R., "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer*, vol. 33 No. 4, 2000.

[15] G. D. Peterson and M. C. Smith, "Programming High Performance Reconfigurable Computers," *SSGRR 2001*, 2001.

[16] M. C. Smith and G. D. Peterson, "Programming High Performance Reconfigurable Computers (HPRC)," *SPIE International Symposium ITCom*, 2001.

[17] Matlab.Documentation, "MATLAB-The Language of Technical Computing, Using Matlab version 6.0," August 2002 ed: COPYRIGHT 1984 - 2002 by The MathWorks, Inc., 2002.

[18] C. Moler, "Why there isn't a parallel MATLAB," *Matlab News and Notes*, 1995.

[19] G. Cappuccino, G. Cocorullo, P. Corsonello, S. Perri, and G. Staino, "Custom Reconfigurable Computing Machine for High Performance Cellular Automata Processing," University of Calabria, Italy. http://www.techonline.com/community/ed_resource/feature_article/14547

[20] Osman Devrim Fidanci1, Dan Poznanovic2, Kris Gaj3, Tarek El-Ghazawi1, Nikitas Alexandridis1, "Performance and Overhead in a Hybrid Reconfigurable Computer." George Washington University

[21] N. W. Bergmann, G. Brebner, and J.P. Gray, "Reconfigurable Computing and Reactive Systems" *Proceedings of the Australasian Workshop on Parallel and Real-Time Systems: PART '00*, Newcastle, November, 2000

[22] P. Waldeck, N.W. Bergmann, ""Dynamic Hardware-Software Partitioning on Reconfigurable System-on-Chip", International Workshop on System-on-Chip for Real-Time Applications, Calgary Canada, June 2003.

[23] D. Thomas, J. Adams, and H. Schmit, "A Model and Methodology for Hardware-Software Codesign", *IEEE Design & Test of Computers*, Vol.

10, No. 3, September 1993, pp. 6-15.

[24] J. Villarreal et al., “Improving Software Performance with Configurable Logic,” *J. Design Automation of Embedded Systems,* Nov. 2002, pp. 325-339.

[25] Yi Pan, Jie Li, Ranga Vemuri, “Continuous Wavelet Transform On Reconfigurable Meshes,” 15th International Parallel and Distributed Processing Symposium (IPDPS'01) Workshops April 23 - 27, 2001

[26] M. Vishwanath, R. M. Owens, M. J. Irwin, “VLSI Architectures for the Discrete Wavelet Transform,” IEEE Transactions on Circuits and Systems, Part II, pp 305-316, May 1995.

[27] ``Garp: A MIPS Processor with a Reconfigurable Coprocessor," by John R. Hauser and John Wawrzynek, published in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM '97, April 16-18, 1997).

[28] R.C. Gonzalez, R.E. Woods “Digital Image Processing (2nd Edition),” Addison-Wesley Publishing Company, 1993.

[29] C. Sidney Burrus, “Introduction to Wavelet and Wavelets Transform.”

[30] Charles K. Chui, “Wavelets: A Tutorial in Theory and Applications (Wavelet Analysis and Its Applications, Vol 2).”

[31] Imaging Processing and Matrix libraries from pervious course work in Image Processing and Pattern Reorganization classes. See appendix.

[32] Damjan Lampret, Jamil Khatib, “HDL Coding Guidelines.” http://www.doe.carleton.ca/~gallan/478/478_coding.pdf

[33] James M. McCollum, Chris D. Cox, Michael L. Simpson, and Gregory D. Peterson, “Accelerating Gene Regulatory Network Modeling Using Grid-Based Simulations.” SIMULATION: Transactions of The Society of Modeling and Simulation Interventional. In review.

Appendix

Original Software Application in Matlab

```
% HYPERSPECTRAL IMAGE PROCESSING
% by S. G. KONG

close all
% Reading data ...
input_file = 'fchicktum04.img';
samples = 460;
lines = 400;
%Bands = 65;
bands = 65;
counter=0;
[fid, msg] = fopen(input_file, 'r');
[ImageFile, count] = fread(fid, [samples, lines*bands], 'int16');
status = fclose(fid);

% 3-d and 2-d variables to store the image
I = zeros(samples, lines, bands);
BandImage = zeros(samples, lines);

% find BandImage from ImageFile
for ib = 1: bands
    for il = 1: lines
        BandImage(:, il) = ImageFile(:, (il - 1)*bands + ib);
    end
    I(:, :, ib) = BandImage;
end
clear ImageFile;

% 2-D Wavelet transform of band images
row = 233;
col = 203;
bands = 65;
wI = zeros(row, col, bands);
a = zeros(samples, lines);
for ib = 1: bands
    a(:, :) = I(:, :, ib);
    [ca, ch, cv, cd] = dwt2(a, 'db4');
    wI(:, :, ib) = ca;
end

% Find normalized nI with respect to max of each band I(:, :, k)
bands = 65;
```

```
for ib=1: bands
   MAX = max(max(I(:, :, ib)));
   nI(:, :, ib) = I(:, :, ib)/MAX;
   MAX = max(max(wI(:, :, ib)));
   nwI(:, :, ib) = wI(:, :, ib)/MAX;
end
FMAX = max(max(max(wI)));

% Display a spectral image
figure, imshow(nwI(:, :, 5))

% Plot hyperspectral signatures ...
% Normal tissue
normal = zeros(20, bands);
normal(1:5, :) = wI(56:60, 140, :)/FMAX;
normal(6:10, :) = wI(91:95, 120, :)/FMAX;
normal(11:15, :) = wI(86:90, 130, :)/FMAX;
normal(16:20, :) = wI(51:55, 51, :)/FMAX;

% Tumor
tumor = zeros(20, bands);
tumor(1:5, :) = wI(161:165, 120, :)/FMAX;
tumor(6:10, :) = wI(206:210, 125, :)/FMAX;
tumor(11:15, :) = wI(121:125, 130, :)/FMAX;
tumor(16:20, :) = wI(196:200, 100, :)/FMAX;

% Background
bg = zeros(20, bands);
bg(1:10, :) = wI(131:140, 25, :)/FMAX;
bg(11:20, :) = wI(116:125, 180, :)/FMAX;

figure, plot(normal', ':b')
xlabel('Bands (Channels)')
ylabel('Relative Fluorescence Intensity (RFI)')
hold on
plot(tumor', 'r')
plot(bg', 'g')
hold off
axis([0 66 0 0.8])

% Gaussian membership functions
gaussf = inline('exp(-(x-m).^2/(2*s^2))');
t = linspace(0,1,500);
```

```
% Features
d = zeros(row, col);
feature = zeros(row, col, 2);
for ir = 1: row
    for ic = 1: col
        af = mean(wI(ir, ic, 15:25))/FMAX;
        rb = (mean(wI(ir, ic, 20:25)) - mean(wI(ir, ic, 40:45)))/mean(wI(ir, ic, 20:25));
        feature(ir, ic, 1) = af;
        feature(ir, ic, 2) = rb;

        mb = gaussf(0, 0.05, af);
        mt = min(gaussf(0.2, 0.12, af), gaussf(0, 0.12, rb));
        mn = min(gaussf(1, 0.4, af), gaussf(1, 0.3, rb));
        if ((mt > mn) & (mt > mb))
            d(ir, ic) = 1;
        end
    end
 end

figure,
imshow(d)
```

Modified C++ Version of the Application

```
/*
==========================================================
Hyperspectral Imaging
Chicken Tumor

-- Yuan He

he@student.ece.utk.edu
Department of Electrical & Computer Engineer
University of Tennessee, Knoxville.

using chickimgv20.h & chickmtrix library

*** Profiler for chickv20.cpp ***
*** calculate run times       ***

This program takes a Matlab binary hyperspectral
image of a chicken(460*400*65) convert to bandimages
then performs 2D wavelet transform to obtain its
low frequency transformed image. Then we
normalized the transformed images to obtain the
hyperspectral signatures to determine the tumor
locations.

note:
1) From an image file, Matlab assumes the data are
in columns whereas C++ reads as rows
2) Wavelet Transform: to obtain LL or CA coefficient
we do a lowpass filter to the rows then to the
columns.

Parts of header library code are from Dr. Qi of
Department of Electrical & Computer Engineer
University of Tennessee, Knoxville.

input file : FCHIKCKTUM04.IMG  - tumor image 460*400*50

output files: normdata.txt     -20*65
              tumordata.txt    -20*65
              bgddata.txt      -20*65 background
              tumorimage.txt   -230*200 tumor image
===========================================================
*/

#include "chickmatrix.h"
#include "chickimgv20.h"
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <math.h>
#include <string>
```

```
#include <stdio.h>
#include <ctime>

using namespace std;

#define bands 65             //k channel
#define samples 460          //i row
#define lines 400            //j col
#define inputFile "FCHICKTUM04.IMG"
#define outputImg1 "OUTPUTIMG1.IMG"
#define outputImg2 "OUTPUTIMG2.IMG"

float gaussf (float m, float s, float x){
 float g;
 g=exp(-((x-m)*(x-m)) / ( 2 * s * s ) );
 return (g);
}

float minny (float x, float y){
 float minny0;
 if (x<=y) minny0=x;
  else minny0=y;
 return (minny0);
}

int main(int argc, char **argv)
{
 Image img1,imgW1,imgNW1,imgNorm(20,65),
imgTumor(20,65),imgBGD(20,65),imgD(230,200);
 int i,j,k,j2,m,x,ii,jj,iii,counter=0;
 float max, maxW, Fmax, Fmin, af, rb, rb1, rb2,mt,mn,mb;
 Matrix bandimage, A,dRow,dRowdCol,dRowCF;

 float M[samples*lines];

 unsigned long int s1=0, s2=0,  t1=0, t2=0;
 unsigned long int w1=0, w2=0,  n1=0, n2=0;
 unsigned long int h1=0, h2=0,  f1=0, f2=0, write1=0, write2=0;

 t1=clock();

 s1=clock();

 for (i=0;i<samples*lines;i++) M[i]=0;

 A.createMatrix(samples,2);
 bandimage.createMatrix(samples,lines);
 dRow.createMatrix(samples,200);
 dRowCF.createMatrix(samples,200);
 dRowdCol.createMatrix(230,200);

 img1.readImage(inputFile);
 imgW1.createImage();
 imgNW1.createImage();
```

```
  s2=clock();

 //================================================= wavelet transfer

  w1=clock();

for (k=0; k<bands;k++){

  for(i=0; i<samples;i++)
   for(j=0;j<lines;j++)
    bandimage(i,j)=img1(i,j,k);

//lowpass filter rows

  bandimage.lowpass(460*400,0);  //bandimage becomes 230*400

//changes 230*400 to 460*200 format: fill row first

  ii=0; jj=0;
  for (i=0;i<460;i++)
   for (j=0;j<200;j++){
    if (jj==400) { ii++; jj=0; }
    dRow(i,j)=bandimage(ii,jj++);
   }

//lowpass filters col

// change format from fill row first to fill column first

  iii=0;
  for (j=0; j<200; j++)
   for (i=0;i<460;i++)
    M[iii++]=dRow(i,j);

  iii=0;
  for (i=0; i<460; i++)
   for (j=0;j<200;j++)
    dRowCF(i,j)=M[iii++];

  dRowCF.lowpass(200*460,0);

// change back from column first to row first and then onto
imgW1(i,j,k)

  for (i=0;i<samples*lines;i++) M[i]=0;

  iii=0;
  for (i=0;i<230;i++)
   for (j=0; j<200; j++)
    M[iii++]=dRowCF(i,j);

  iii=0;
  for (j=0;j<200;j++)
```

```
   for (i=0; i<230; i++)
    imgW1(i,j,k)= M[iii++];

}                                            // end of K loop ! for
each band

  w2=clock();

//===================================find normalized with respect to
max of each band

  n1=clock();

Fmax=0;
Fmin=0;


 for (k=0;k<65;k++){

  max=0;
  maxW=0;

  for (i=0;i<samples;i++)
   for (j=0; j<lines; j++){

    if (max < img1(i,j,k)) max =img1(i,j,k);
    if (maxW < imgW1(i,j,k)) maxW =imgW1(i,j,k);
    if (Fmax < imgW1(i,j,k)) Fmax =imgW1(i,j,k);
    if (Fmin > imgW1(i,j,k)) Fmin =imgW1(i,j,k);
   }

  for (i=0;i<samples;i++)
   for (j=0; j<lines; j++){
    img1(i,j,k)=img1(i,j,k)/max;
    imgNW1(i,j,k)=imgW1(i,j,k)/maxW;
   }
 }

  n2=clock();

 write1=clock();

 imgNW1.writeImage(outputImg1);            // normallized Spectral
Image

 write2=clock();


 //===============================================Plot Hyperspectral
Signatures

 h1=clock();

 for(k=0;k<65;k++){
```

```
 for(i=0;i<20;i++){
  if(i>=0 && i<5){
  imgNorm(i,k)=imgW1(55+i,140,k)/Fmax;
  imgTumor(i,k)=imgW1(160+i,120,k)/Fmax;
  imgBGD(i,k)=imgW1(130+i,25,k)/Fmax;
 }
  if(i>=5 && i<10){
  imgNorm(i,k)=imgW1(90-5+i,120,k)/Fmax;
  imgTumor(i,k)=imgW1(205-5+i,125,k)/Fmax;
  imgBGD(i,k)=imgW1(130-5+i,25,k)/Fmax;
 }
  if(i>=10 && i<15){
  imgNorm(i,k)=imgW1(85-10+i,130,k)/Fmax;
  imgTumor(i,k)=imgW1(120-10+i,130,k)/Fmax;
  imgBGD(i,k)=imgW1(115-10+i,180,k)/Fmax;
 }
  if(i>=15 && i<20){
  imgNorm(i,k)=imgW1(50-15+i,51,k)/Fmax;
  imgTumor(i,k)=imgW1(195-15+i,100,k)/Fmax;
  imgBGD(i,k)=imgW1(115-15+i,180,k)/Fmax;
  }
 }
}

ofstream Norm ("normdata.txt");
ofstream Tumor ("tumordata.txt");
ofstream BGD ("bgddata.txt");

for(i=0;i<20;i++){
 for (j=0;j<65;j++){
  Norm<<imgNorm(i,j)<<" ";
  Tumor<<imgTumor(i,j)<<" ";
  BGD<<imgBGD(i,j)<<" ";
 }
 Norm<<endl;Tumor<<endl;BGD<<endl;
}
Norm.close(); Tumor.close(); BGD.close();

h2=clock();


//==========================================================Features

f1=clock();


for (i=0;i<230;i++)
 for (j=0;j<200;j++){

  af=0; rb=0; rb1=0; rb2=0;

  for(k=14;k<25;k++)
   af+=imgW1(i,j,k);
```

```
    af = ( af / 11 ) / Fmax;

    for(k=19;k<25;k++)
     rb1+=imgW1(i,j,k);
    for(k=39;k<45;k++)
     rb2+=imgW1(i,j,k);

    rb = ( (rb1 / 6) - (rb2 / 6) ) / (rb1 / 6);

    mb = gaussf ( 0, 0.05, af);
    mt = minny ( gaussf(0.2, 0.12, af), gaussf(0,0.12,rb) );
    mn = minny ( gaussf(1, 0.4, af), gaussf(1,0.3,rb) );
    if ((mt>mn) && (mt > mb))  imgD(i,j)=1;
   }

  ofstream Img2 ("tumorimage.txt");
   for(i=0;i<230;i++){
   for (j=0;j<200;j++){
    Img2<<imgD(i,j)<<" ";
   }
   Img2<<endl;
  }
  Img2.close();


  f2=clock();

  t2=clock();

  cout<<"**************************************************"<<endl;
  cout<<"****  Profiler for chickv20.cpp is as following..."<<endl;
  cout<<"**************************************************"<<endl;

  cout<<"  Total run time  = "<<(t2-t1)*1e-6<<" second"<<endl;
  cout<<"  Setup time      = "<<(s2-s1)*1e-6<<" second"<<endl;
  cout<<"  Wavelet time    = "<<(w2-w1)*1e-6<<" second"<<endl;
  cout<<"  Normalized time = "<<(n2-n1)*1e-6<<" second"<<endl;
  cout<<"  Signatures time = "<<(h2-h1)*1e-6<<" second"<<endl;
  cout<<"  Features time   = "<<(f2-f1)*1e-6<<" second"<<endl;
  cout<<"  Write 3D Image  = "<<(write2-write1)*1e-6<<" second"<<endl;

  cout<<"**************************************************"<<endl;
  cout<<"Fmax= "<<Fmax<<endl;
  cout<<"Fmin= "<<Fmin<<endl;
  cout<<"max= " <<max<<endl;
  cout<<"maxW= "<<maxW<<endl;



   return 0;
 }
```

C++ Header Files and Libraries

Chickimgv20.cpp

```
#include "chickimgv20.h"
#include <fstream>
#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <cmath>
using namespace std;

#define PI 3.1415926

// default constructor
Image::Image()
{
  row = 460;
  col = 400;
  channel = 65;
//  type = PGMRAW;
  image = (float *) new float [row * col * channel];
  if (!image)
    outofMemory();
}

// constructor for 2D images
Image::Image(int r, int c)
{
  row = r;
  col = c;
  channel = 1;
  image = (float *) new float [row * col * channel];
  if (!image)
    outofMemory();
}


// constructor for grayscale/color images
Image::Image(int r, int c, int t)
{
  row = r;
  col = c;

  image = (float *) new float [row * col * t];
  if (!image)
    outofMemory();
}

// destructor
Image::~Image()
{
  if (image)
```

```
    delete [] image;       // free the image buffer
}

// allocate memory for the image
void Image::createImage()
{
  image = (float *) new float [row * col * channel];
  if (!image)
    outofMemory();
}

void Image::create2D(int r, int c)
{
  image = (float *) new float [r * c  * 1];
  if (!image)
    outofMemory();
}

// read image from a file
void Image::readImage(char *fname)
{
  ifstream ifp;
  short *img;
  int i, j, k;

  ifp.open(fname, ios::in);

  if (!ifp) {
    cout << "Can't read image: " << fname << endl;
    exit(1);
  }

  // read the image data
  img = (short *) new short [row * col * channel];
  if (!img)
    outofMemory();

  ifp.read(img, (row * col * channel * sizeof(short)));

  // convert the data type from unsigned char to float
  image = (float *) new float [row * col * channel];
  if (!image)
    outofMemory();

 for (j=0; j<col; j++)
    for (k=0; k<channel; k++)
      for (i=0; i<row; i++)
       image[i+k*row+j*row*channel] =
(float)img[i+k*row+j*row*channel];

  ifp.close();
  delete img;
}
```

```
// write image buffer to a file
void Image::writeImage(char *fname)
{
  ofstream ofp;
  int i, j, k;
  float *img;

  ofp.open(fname, ios::out);

  if (!ofp) {
    cout << "Can't write image: " << fname << endl;
    exit(1);
  }

  // convert the image data type back to unsigned char
  img = (float *) new float [row * col * channel];
  if (!img)
    outofMemory();

  for (i=0; i<row; i++)
    for (j=0; j<col; j++)
      for (k=0; k<channel; k++)
        img[i+k*row+j*row*channel] =
(float)image[i+k*row+j*row*channel];

      //  img[(i*col+j)*channel+k] = (float)image[(i*col+j)*channel+k];

  ofp.write(img, (row * col * channel * sizeof(float)));

  ofp.close();
  delete img;
}


// overloading () operator
float & Image::operator()(int i, int j, int k)
{

  return image[i + k * row + j * row * channel];
}

// overloading () operator
float & Image::operator()(int i, int j)
{
  return image[i*col + j];
}


// output out of memory error
void Image::outofMemory()
{
  cout << "Out of memory!\n";
  exit(1);}
```

Chickimgv20.h

```
#define BINARY   11                    // binary image

class Image {
 public:

  // constructors and destructor
  Image();                             // default constructor
  Image(int, int);                     // constructor for grayscale
images
  Image(int, int, int);                // constructor for
grayscale/color images
  Image(Image &);                      // copy constructor
  ~Image();                            // destructor

  float getMaximum() const;              // get the maximum pixel value

  void create2D(int, int);                   // create image, allocate
memory for 2D
  void createImage();                  // create image, allocate memory
for the image
  void readImage(char *);              // read image from a file
  void writeImage(char *);             // write image to a file

  float & operator()(int, int, int); // operator overloading, default
k=1
  float & operator()(int, int); // operator overloading, default k=1

  Image operator+(Image);              // overloading + operator
  Image operator-(Image);              // overloading - operator
  Image operator*(Image);              // overloading * (element-wised
multiplication)
  Image operator/(Image);              // overloading pixelwise
division
  Image operator->*(Image);            // overloading ->* operator
(matrix multiplication)

 private:
  int row;                  // number of rows / height
  int col;                  // number of columns / width
  int channel;              // nr of channels (1 for gray-level, 3 for
color image)
  int type;                 // image type (PGM, PPM, etc.)
  int maximum;              // the maximum pixel value
  int setmax;               // indicates if users want to set their own
maximum
  float *image;             // image buffer
  void outofMemory();       // output out of memory message

};
```

Chickmatrix.cpp

```
#include "chickmatrix.h"
#include <fstream>
#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <cmath>
using namespace std;

#define C0 0.4829629131445341
#define C1 0.8365163037378079
#define C2 0.2241438680420134
#define C3 -0.1294095225512604
// default constructor
Matrix::Matrix()
{
  createMatrix(0, 0);
}

// constructor when knowing row and column
Matrix::Matrix(int nr, int nc)
{
  createMatrix(nr, nc);
}

// copy constructor
Matrix::Matrix(Matrix &m)
{
  int i, j;

  createMatrix(m.getRow(), m.getCol());             // allocate memory

  for (i=0; i<row; i++)
    for (j=0; j<col; j++)
      matrix[i * col + j] = m(i, j);
}

int Matrix::getRow() const
{
  return row;
}

// get number of columns
int Matrix::getCol() const
{
  return col;
}


// allocate memory for the matrix
void Matrix::createMatrix(int nr, int nc)
{
  int i;

  row = nr;
  col = nc;
  matrix = (double *) new double [row * col];
  if (!matrix)
    outofMemory();

  for (i=0; i<row*col; i++)
    matrix[i] = 0.0;
}

// output out of memory error
void Matrix::outofMemory()
{
  cerr << "Out of memory!\n";
  exit(1);
}

// destructor
Matrix::~Matrix()
{
  if (matrix)
    delete [] matrix;       // free the matrix buffer
}

// overloading () operator
double & Matrix::operator()(int i, int j)
{
  return matrix[i * col + j];
}
// wavelet transform

void Matrix::lowpass(int n,int isign){
  Matrix wksp;
  int nh,nh1,i,j;

  if(n<4) exit(1);
  wksp.createMatrix(1,n);
  nh1 = (nh=n>>1)+1;
  if(isign>=0){
    for(i=0,j=0;j<n-3;j+=2,i++){                        //downsizing
j+=2
      wksp(0,i) =
C0*matrix[j]+C1*matrix[j+1]+C2*matrix[j+2]+C3*matrix[j+3];
     // wksp(0,i+nh) = C3*matrix[j]-C2*matrix[j+1]+C1*matrix[j+2]-
C0*matrix[j+3];
    }
    wksp(0,i) = C0*matrix[n-2]+C1*matrix[n-
1]+C2*matrix[0]+C3*matrix[1];
   // wksp(0,i+nh) = C3*matrix[n-2]-C2*matrix[n-1]+C1*matrix[0]-
C0*matrix[1];
  }
  for(i=0;i<n;i++) matrix[i] = wksp(0,i);}
```

Chickmatrix.h

```
class Matrix {
 public:
  // constructors and destructor
  Matrix();                          // default constructor
  Matrix(int,                        // constructor with row
       int);                       // column
  Matrix(Matrix &);                  // copy constructor
  ~Matrix();                         // destructor

// create a matrix
  void createMatrix(int,             // row
                int);            // column

  int getRow() const;                // get row number / the number of
sample
  int getCol() const;                // get column number / the number
of feature


// operator overloading functions
  double & operator()(int,           // row index
                int);            // column index

  void lowpass(int,int);

// other functions

 protected:
  int row;                           // number of rows / sample
  int col;                           // number of columns / feature
  double *matrix;                     // matrix buffer

  void outofMemory();

  // the following four functions are used by inverse()
  int findPivot(Matrix &, int);      // find the row with maximum
absolute value in that volumn
  void switchRow(Matrix &, int, int);// switch two rows
  void dividePivot(Matrix &, int);   // divide that row with the
element in that column
  void eliminate(Matrix &, int);     // eliminate the following columns
};


/**
 * Data includes t
 * (1) m x n matrix where
 *     m is the number of samples and n is the number of features
 *     t is the number of categories
 * (2) m x 1 matrix that stores to which category a sample belongs
 **/
```

```
class Data : public Matrix {
 public:
  // constructors and destructor
  Data();                                // default constructor
  Data(int,                              // constructor with number of
feature
       int,                              // number of sample
       int);                             // number of type

  // get and set functions
  Matrix getType(int);                   // get the subMatrix for a certain
category
  void setType(int);                     // set the number of categories
  void readData(char *,                  // file name
            int,                     // number of feature
            int);                    // number of type
  void readData(char *,                  // file name
                int);                    // number of feature
  void readData(char *);

 private:
  int type;
 };
```

VHDL Source Code

#pilchard.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity pilchard is
port (
      PADS_exchecker_reset: in std_logic;
      PADS_dimm_ck: in std_logic;
      PADS_dimm_cke: in std_logic_vector(1 downto 0);
      PADS_dimm_ras: in std_logic;
      PADS_dimm_cas: in std_logic;
      PADS_dimm_we: in std_logic;
      PADS_dimm_s: std_logic_vector(3 downto 0);
      PADS_dimm_a: in std_logic_vector(13 downto 0);
      PADS_dimm_ba: in std_logic_vector(1 downto 0);
      PADS_dimm_rege: in std_logic;
      PADS_dimm_d: inout std_logic_vector(63 downto 0);
      PADS_dimm_cb: inout std_logic_vector(7 downto 0);
      PADS_dimm_dqmb: in std_logic_vector(7 downto 0);
      PADS_dimm_scl: in std_logic;
      PADS_dimm_sda: inout std_logic;
      PADS_dimm_sa: in std_logic_vector(2 downto 0);
      PADS_dimm_wp: in std_logic;
      PADS_io_conn: inout std_logic_vector(27 downto 0) );
end pilchard;

architecture syn of pilchard is

      component INV
      port (
            O: out std_logic;
            I: in std_logic );
      end component;

      component BUF
      port (
            I: in std_logic;
            O: out std_logic );
      end component;

      component BUFG
      port (
            I: in std_logic;
            O: out std_logic );
      end component;

      component CLKDLLHF is
      port (
            CLKIN: in std_logic;
            CLKFB: in std_logic;
            RST: in std_logic;
            CLK0: out std_logic;
            CLK180: out std_logic;
            CLKDV: out std_logic;
            LOCKED: out std_logic );
      end component;

      component FDC is
      port (
            C: in std_logic;
            CLR: in std_logic;
            D: in std_logic;
            Q: out std_logic );
      end component;

      component IBUF
      port (
            I: in std_logic;
            O: out std_logic );
      end component;

      component IBUFG
      port (
            I: in std_logic;
            O: out std_logic );
      end component;

      component IOB_FDC is
      port (
            C: in std_logic;
            CLR: in std_logic;
            D: in std_logic;
            Q: out std_logic );
      end component;

      component IOBUF
      port (
            I: in std_logic;
            O: out std_logic;
            T: in std_logic;
            IO: inout std_logic );
      end component;

      component OBUF
      port (
            I: in std_logic;
            O: out std_logic );
      end component;

      component STARTUP_VIRTEX
      port (
            GSR: in std_logic;
            GTS: in std_logic;
            CLK: in std_logic );
      end component;
```

```
component pcore
port (
      clk: in std_logic;
      clkdiv: in std_logic;
      rst: in std_logic;
      read: in std_logic;
      write: in std_logic;
      addr: in std_logic_vector(13 downto 0);
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in std_logic_vector(25 downto 0);
      extout: out std_logic_vector(25 downto 0);
      extctrl: out std_logic_vector(25 downto 0) );
end component;

signal clkdllhf_clk0: std_logic;
signal clkdllhf_clkdiv: std_logic;
signal dimm_ck_bufg: std_logic;
signal dimm_s_ibuf: std_logic;
signal dimm_ras_ibuf: std_logic;
signal dimm_cas_ibuf: std_logic;
signal dimm_we_ibuf: std_logic;
signal dimm_s_ibuf_d: std_logic;
signal dimm_ras_ibuf_d: std_logic;
signal dimm_cas_ibuf_d: std_logic;
signal dimm_we_ibuf_d: std_logic;
signal dimm_d_iobuf_i: std_logic_vector(63 downto 0);
signal dimm_d_iobuf_o: std_logic_vector(63 downto 0);
signal dimm_d_iobuf_t: std_logic_vector(63 downto 0);
signal dimm_a_ibuf: std_logic_vector(14 downto 0);
signal dimm_dqmb_ibuf: std_logic_vector(7 downto 0);
signal io_conn_iobuf_i: std_logic_vector(27 downto 0);
signal io_conn_iobuf_o: std_logic_vector(27 downto 0);
signal io_conn_iobuf_t: std_logic_vector(27 downto 0);

signal s,ras,cas,we : std_logic;

signal VDD: std_logic;
signal GND: std_logic;

signal CLK: std_logic;
signal CLKDIV: std_logic;
signal RESET: std_logic;
signal READ: std_logic;
signal WRITE: std_logic;
signal READ_p: std_logic;
signal WRITE_p: std_logic;
signal READ_n: std_logic;
signal READ_buf: std_logic;
signal WRITE_buf: std_logic;
signal READ_d: std_logic;
signal WRITE_d: std_logic;
```

```
      signal READ_d_n: std_logic;
      signal READ_d_n_buf: std_logic;

      signal pcore_addr_raw: std_logic_vector(13 downto 0);
      signal pcore_addr: std_logic_vector(13 downto 0);
      signal pcore_din: std_logic_vector(63 downto 0);
      signal pcore_dout: std_logic_vector(63 downto 0);
      signal pcore_dmask: std_logic_vector(63 downto 0);
      signal pcore_extin: std_logic_vector(25 downto 0);
      signal pcore_extout: std_logic_vector(25 downto 0);
      signal pcore_extctrl: std_logic_vector(25 downto 0);
      signal pcore_dqmb: std_logic_vector(7 downto 0);

--    CLKDIV frequency control, default is 2
--  uncomment the following lines so as to redefined the clock rate
--  given by clkdiv
      attribute CLKDV_DIVIDE: string;
      attribute CLKDV_DIVIDE of U_clkdllhf: label is "8";


begin

      VDD <= '1';
      GND <= '0';

      U_ck_bufg: IBUFG port map (
            I => PADS_dimm_ck,
            O => dimm_ck_bufg );

      U_reset_ibuf: IBUF port map (
            I => PADS_exchecker_reset,
            O => RESET );

      U_clkdllhf: CLKDLLHF port map (
            CLKIN => dimm_ck_bufg,
            CLKFB => CLK,
            RST => RESET,
            CLK0 => clkdllhf_clk0,
            CLK180 => open,
            CLKDV => clkdllhf_clkdiv,
            LOCKED => open );

      U_clkdllhf_clk0_bufg: BUFG port map (
            I => clkdllhf_clk0,
            O => CLK );

      U_clkdllhf_clkdiv_bufg: BUFG port map (
            I => clkdllhf_clkdiv,
            O => CLKDIV );

      U_startup: STARTUP_VIRTEX port map (
            GSR => RESET,
            GTS => GND,
            CLK => CLK );
```

```
U_dimm_s_ibuf: IBUF port map (
      I => PADS_dimm_s(0),
      O => dimm_s_ibuf );

U_dimm_ras_ibuf: IBUF port map (
      I => PADS_dimm_ras,
      O => dimm_ras_ibuf );

U_dimm_cas_ibuf: IBUF port map (
      I => PADS_dimm_cas,
      O => dimm_cas_ibuf );

U_dimm_we_ibuf: IBUF port map (
      I => PADS_dimm_we,
      O => dimm_we_ibuf );

G_dimm_d: for i in integer range 0 to 63 generate

      U_dimm_d_iobuf: IOBUF port map (
            I => dimm_d_iobuf_i(i),
            O => dimm_d_iobuf_o(i),
            T => dimm_d_iobuf_t(i),
            IO => PADS_dimm_d(i) );

      U_dimm_d_iobuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => dimm_d_iobuf_o(i),
            Q => pcore_din(i) );

      U_dimm_d_iobuf_i: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => pcore_dout(i),
            Q => dimm_d_iobuf_i(i) );

      U_dimm_d_iobuf_t: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => READ_d_n_buf,
            Q => dimm_d_iobuf_t(i) );

end generate;

G_dimm_a: for i in integer range 0 to 13 generate

      U_dimm_a_ibuf: IBUF port map (
            I => PADS_dimm_a(i),
            O => dimm_a_ibuf(i) );

      U_dimm_a_ibuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => dimm_a_ibuf(i),
            Q => pcore_addr_raw(i) );

end generate;

pcore_addr(3 downto 0) <= pcore_addr_raw(3 downto 0);
addr_correct: for i in integer range 4 to 7 generate
      ADDR_INV: INV port map (
            O => pcore_addr(i),
            I => pcore_addr_raw(i) );
end generate;
pcore_addr(13 downto 8) <= pcore_addr_raw(13 downto 8);

G_dimm_dqmb: for i in integer range 0 to 7 generate

      U_dimm_dqmb_ibuf: IBUF port map (
            I => PADS_dimm_dqmb(i),
            O => dimm_dqmb_ibuf(i) );

      U_dimm_dqmb_ibuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => dimm_dqmb_ibuf(i),
            Q => pcore_dqmb(i) );

end generate;

pcore_dmask(7 downto 0) <= (others => (not pcore_dqmb(0)));
pcore_dmask(15 downto 8) <= (others => (not pcore_dqmb(1)));
pcore_dmask(23 downto 16) <= (others => (not pcore_dqmb(2)));
pcore_dmask(31 downto 24) <= (others => (not pcore_dqmb(3)));
pcore_dmask(39 downto 32) <= (others => (not pcore_dqmb(4)));
pcore_dmask(47 downto 40) <= (others => (not pcore_dqmb(5)));
pcore_dmask(55 downto 48) <= (others => (not pcore_dqmb(6)));
pcore_dmask(63 downto 56) <= (others => (not pcore_dqmb(7)));

G_io_conn: for i in integer range 2 to 27 generate

      U_io_conn_iobuf: IOBUF port map (
            I => io_conn_iobuf_i(i),
            O => io_conn_iobuf_o(i),
            T => io_conn_iobuf_t(i),
            IO => PADS_io_conn(i) );

      U_io_conn_iobuf_o: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => io_conn_iobuf_o(i),
            Q => pcore_extin(i - 2) );

      U_io_conn_iobuf_i: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => pcore_extout(i - 2),
            Q => io_conn_iobuf_i(i) );

      U_io_conn_iobuf_t: IOB_FDC port map (
            C => CLK,
            CLR => RESET,
            D => pcore_extctrl(i - 2),
            Q => io_conn_iobuf_t(i) );

end generate;

U_io_conn_0_iobuf: IOBUF port map (
      I => dimm_ck_bufg,
      O => open,
      T => GND,
      IO => PADS_io_conn(0) );

U_io_conn_1_iobuf: IOBUF port map (
      I => GND,
      O => open,
      T => VDD,
      IO => PADS_io_conn(1) );

READ_p <=
      (not dimm_s_ibuf) and
      (dimm_ras_ibuf) and
      (not dimm_cas_ibuf) and
      (dimm_we_ibuf);

U_read: FDC port map (
      C => CLK,
      CLR => RESET,
      D => READ_p,
      Q => READ );

U_buf_read: BUF port map (
      I => READ,
      O => READ_buf );

U_read_d: FDC port map (
      C => CLK,
      CLR => RESET,
      D => READ,
      Q => READ_d );

WRITE_p <=
      (not dimm_s_ibuf) and
      (dimm_ras_ibuf) and
      (not dimm_cas_ibuf) and
      (not dimm_we_ibuf);

U_write: FDC port map (
      C => CLK,
      CLR => RESET,
      D => WRITE_p,
            Q => WRITE );

      U_buf_write: BUF port map (
            I => WRITE,
            O => WRITE_buf );

      U_write_d: FDC port map (
            C => CLK,
            CLR => RESET,
            D => WRITE,
            Q => WRITE_d );

      READ_n <= not READ;

      U_read_d_n: FDC port map (
            C => CLK,
            CLR => RESET,
            D => READ_n,
            Q => READ_d_n );

      U_buf_read_d_n: BUF port map (
            I => READ_d_n,
            O => READ_d_n_buf );

      -- User logic should be placed inside pcore
      U_pcore: pcore port map (
            clk => CLK,
            clkdiv => CLKDIV,
            rst => RESET,
            read => READ,
            write => WRITE,
            addr => pcore_addr,
            din => pcore_din,
            dout => pcore_dout,
            dmask => pcore_dmask,
            extin => pcore_extin,
            extout => pcore_extout,
            extctrl => pcore_extctrl );

end syn;
```

#pcore.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity pcore is
port (
      clk: in std_logic;
      clkdiv: in std_logic;
      rst: in std_logic;
      read: in std_logic;
      write: in std_logic;
      addr: in std_logic_vector(13 downto 0);
      din: in std_logic_vector(63 downto 0);
      dout: out std_logic_vector(63 downto 0);
      dmask: in std_logic_vector(63 downto 0);
      extin: in    std_logic_vector(25 downto 0);
      extout:  out std_logic_vector(25 downto 0);
      extctrl: out std_logic_vector(25 downto 0) );
end pcore;


architecture syn of pcore is
component dpram256_64
      port (
      addra: IN std_logic_VECTOR(7 downto 0);
      clka: IN std_logic;
      dina: IN std_logic_VECTOR(63 downto 0);
      douta: OUT std_logic_VECTOR(63 downto 0);
      wea: IN std_logic;

      addrb: IN std_logic_VECTOR(7 downto 0);
      clkb: IN std_logic;
      dinb: IN std_logic_VECTOR(63 downto 0);
      doutb: OUT std_logic_VECTOR(63 downto 0);
      web: IN std_logic);
end component;

component parith
port (
      clk: in std_logic;
      rst: in std_logic;
      addr: out std_logic_vector(7 downto 0);
      din: out std_logic_vector(63 downto 0);
      dout: in std_logic_vector(63 downto 0);
      we: out std_logic;
      start: in std_logic;
      finish: out std_logic
);
end component;

signal addrb:std_logic_VECTOR(7 downto 0);
signal clkb: std_logic;
signal dinb: std_logic_VECTOR(63 downto 0);
signal doutb: std_logic_VECTOR(63 downto 0) :=
"0000000000000000000000000000000000000000000000000000000000000000";
signal web: std_logic;

--signal read_latch: std_logic;
--signal addr_latch: std_logic_vector(7 downto 0);
signal finish: std_logic;
signal start : std_logic;

signal bram_dout : std_logic_VECTOR(63 downto 0);

--debug signal
signal  start_parith:std_logic;

--register interface
--signal reg0: std_logic_VECTOR(31 downto 0);

begin

ram0:dpram256_64 port map  (
      addra => addr(7 downto 0),
      clka  => clk,
      dina  => din,
      douta => bram_dout,
      wea   => write,

      addrb => addrb,
      clkb  => clkb,
      dinb  => dinb,
      doutb => doutb,
      web   => web
);

parith0: parith port map (
      clk   => clkb,
      rst   => rst,
      dout  => doutb,
      start => start_parith,

      addr  => addrb,
      din   => dinb,
      we    => web,
      finish      => finish
);


process(clk)
begin

if ( rst = '1') then
      start_parith <= '0';
elsif (clk'event and clk ='1') then
```

```
      start_parith <= start_parith or start;
end if;


end process;

dout(31 downto 0) <= bram_dout(31 downto 0);
dout(63 downto 32) <= bram_dout(63 downto 32);

start <= '1' when (write = '1' and addr(7 downto 0) = "00000100")
      else '0';

-- define the core clock
clkb <= clkdiv;

end syn;
```

#Parith 32bit version

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity parith is
generic (n : integer := 32);

port (
      clk: in std_logic;
      rst: in std_logic;
      addr: out std_logic_vector(7 downto 0);
      din: out std_logic_vector(63 downto 0);   -- write to block ram
      dout: in std_logic_vector(63 downto 0);   -- read  to block ram
      we: out std_logic;                                    -- write
enable
      start: in std_logic;
      --dc_in: in std_logic_vector( 7 downto 0);
      finish: out std_logic
);
end parith;

architecture rtl of parith is

component fxmult
port(

      clk,start: in std_logic;
      a0, a1, a2, a3 : IN std_logic_VECTOR(n-1 downto 0);
      --finish : out std_logic;
      q : OUT std_logic_VECTOR(n-1 downto 0)
      );
end component;

component max
port(

      clk,start: in std_logic;
      num1, num2, oldmax : IN std_logic_VECTOR(n-1 downto 0);
      --finish : out std_logic;
      newmax : OUT std_logic_VECTOR(n-1 downto 0)
      );
end component;
type states is ( s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9,
s_10);


signal idx: std_logic_vector(7 downto 0);
signal buff_a0, buff_a1, buff_a2, buff_a3, buff_q: std_logic_vector (n-
1 downto 0);
signal buff_b0, buff_b1, buff_b2, buff_b3, buff_bq: std_logic_vector
(n-1 downto 0);
```

```
signal buff_num1, buff_num2, buff_old, buff_new: std_logic_vector (n-1
downto 0);
signal start_a, start_b, start_max : std_logic;
signal count : std_logic_vector(n-1 downto 0);
signal state : states;
signal buff_dout : std_logic_vector ( 63 downto 0);

begin


fxmult1: fxmult port map (
      clk => clk,
      start => start_a,
      a0 => buff_a0,
      a1 => buff_a1,
      a2 => buff_a2,
      a3 => buff_a3,
      --finish => finish_a,
      q => buff_q
);

fxmult2: fxmult port map (
      clk => clk,
      start => start_b,
      a0 => buff_b0,
      a1 => buff_b1,
      a2 => buff_b2,
      a3 => buff_b3,
      --finish => finish_a,
      q => buff_bq
);


max1: max port map (
      clk => clk,
      start => start_max,
      num1 => buff_num1,
      num2 => buff_num2,
      oldmax => buff_old,
      newmax => buff_new
);

process (clk, rst)
begin

 if (rst = '1' ) then
      state <= s_0;
      finish <= '0';
 elsif ( clk = '1' and clk'event ) then

 -- state machine

      if (start = '1') then
            if (state = s_0) then
```

```
                state <= s_1;
                finish <= '0';
        end if;

end if;

case state is

when s_1 =>
        idx <= "00000000";
        state <= s_2;

 when s_2 =>
        idx <= "00000001";
        state <= s_3;

 when s_3 =>
        idx <= "00000010";
        buff_a0 <= buff_dout(63 downto 32);
        buff_a1 <= buff_dout(n-1 downto 0);
        state <= s_4;

when s_4 =>
        idx <= "00000100";
        buff_a2 <= buff_dout (63 downto 32);
        buff_a3 <= buff_dout (n-1 downto 0);
        state <= s_5;
        start_a <= '1';

 when s_5 =>
        idx <= "00000101";
        buff_b0 <= buff_dout(63 downto 32);
        buff_b1 <= buff_dout(n-1 downto 0);
        state <= s_6;

 when s_6 =>
        buff_b2 <= buff_dout(63 downto 32);
        buff_b3 <= buff_dout(n-1 downto 0);
        state <= s_7;
        start_b <= '1';

 when s_7 =>
        count <= buff_dout(63 downto 32);
        buff_old <= buff_dout(n-1 downto 0);
        state <= s_8;

 when s_8 =>
        buff_num1 <= buff_q;
        buff_num2 <= buff_bq;
        start_max <='1';
        state <= s_9;
        idx <= "00000111";
        we <= '1';
```

```
      when s_9 =>
            we <= '0';
            state<=s_10;

      when s_10 =>
            idx <= "00000110";
            we <= '1';
            state <= s_0;
            finish <= '1';

      when others =>
            we <= '0';
       end case;
 end if;
end process;




 addr <= idx;
 din <= (buff_q & buff_bq) when (state = s_8 or state=s_9) else (count
& buff_new);
 buff_dout <= dout;

end rtl;
```

#parith.vhd 16bit version

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity parith is
generic (n : integer := 16);

port (
      clk: in std_logic;
      rst: in std_logic;
      addr: out std_logic_vector(7 downto 0);
      din: out std_logic_vector(63 downto 0);   -- write to block ram
      dout: in std_logic_vector(63 downto 0);   -- read  to block ram
      we: out std_logic;                                    -- write
enable
      start: in std_logic;
      --dc_in: in std_logic_vector( 7 downto 0);
      finish: out std_logic
);
end parith;

architecture rtl of parith is

component fxmult
port(

      clk,start: in std_logic;
      a0, a1, a2, a3 : IN std_logic_VECTOR(n-1 downto 0);
      --finish : out std_logic;
      q : OUT std_logic_VECTOR(n-1 downto 0)
      );
end component;

component max
port(

      clk,start: in std_logic;
      num1, num2, oldmax : IN std_logic_VECTOR(n-1 downto 0);
      --finish : out std_logic;
      newmax : OUT std_logic_VECTOR(n-1 downto 0)
      );
end component;
type states is ( s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8);


signal idx: std_logic_vector(7 downto 0);
signal buff_a0, buff_a1, buff_a2, buff_a3, buff_q: std_logic_vector (n-
1 downto 0);
signal buff_b0, buff_b1, buff_b2, buff_b3, buff_bq: std_logic_vector
(n-1 downto 0);
```

```
signal buff_num1, buff_num2, buff_old, buff_new: std_logic_vector (n-1
downto 0);
signal start_a, start_b, start_max : std_logic;
signal count : std_logic_vector(n-1 downto 0);
signal state : states;
signal buff_dout : std_logic_vector ( 63 downto 0);

begin


fxmult1: fxmult port map (
      clk => clk,
      start => start_a,
      a0 => buff_a0,
      a1 => buff_a1,
      a2 => buff_a2,
      a3 => buff_a3,
      --finish => finish_a,
      q => buff_q
);

fxmult2: fxmult port map (
      clk => clk,
      start => start_b,
      a0 => buff_b0,
      a1 => buff_b1,
      a2 => buff_b2,
      a3 => buff_b3,
      --finish => finish_a,
      q => buff_bq
);


max1: max port map (
      clk => clk,
      start => start_max,
      num1 => buff_num1,
      num2 => buff_num2,
      oldmax => buff_old,
      newmax => buff_new
);

process (clk, rst)
begin

 if (rst = '1' ) then
      state <= s_0;
      finish <= '0';
 elsif ( clk = '1' and clk'event ) then

 -- state machine

      if (start = '1') then
            if (state = s_0) then
```

```
              state <= s_1;
              finish <= '0';
       end if;

end if;

case state is

when s_1 =>
       idx <= "00000000";
       state <= s_2;

 when s_2 =>
       idx <= "00000001";
       state <= s_3;

 when s_3 =>
       idx <= "00000110";
       buff_a0 <= buff_dout(63 downto 48);
       buff_a1 <= buff_dout(47 downto 32);
       buff_a2 <= buff_dout (31 downto 16);
       buff_a3 <= buff_dout (15 downto 0);
       start_a <= '1';
       state <= s_4;

when s_4 =>
       buff_b0 <= buff_dout(63 downto 48);
       buff_b1 <= buff_dout(47 downto 32);
       buff_b2 <= buff_dout (31 downto 16);
       buff_b3 <= buff_dout (15 downto 0);
       start_b <= '1';
       state <= s_5;

 when s_5 =>
       count <= buff_dout(63 downto 48);
       buff_old <= buff_dout(47 downto 32);
       state <= s_6;

 when s_6 =>
       buff_num1 <= buff_q;
       buff_num2 <= buff_bq;
       start_max <='1';
       state <= s_7;

 when s_7 =>
       state <= s_8;

 when s_8 =>
       idx <= "00000111";
       we <= '1';
       state <= s_0;
       finish <= '1';

when others =>
```

```
            we <= '0';
       end case;
 end if;
end process;




 addr <= idx;
 din <= (buff_q & buff_bq & count & buff_new);
 buff_dout <= dout;

end rtl;
```

#fxmult with DesignWare from CoreGen.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;


ENTITY fxmult IS

generic (n: integer :=32);

      port (
      clk: IN std_logic;
      a0, a1, a2, a3: IN std_logic_VECTOR (n-1 downto 0);
      q: OUT std_logic_VECTOR(n-1 downto 0);
      start : IN std_logic;
      finish : OUT std_logic
      );
END fxmult;

ARCHITECTURE syn OF fxmult IS

component DW02_prod_sum_inst
port(
      inst_A : in std_logic_vector(n*4-1 downto 0);
        inst_B : in std_logic_vector(15*4-1 downto 0);
        inst_TC : in std_logic;
        SUM_inst : out std_logic_vector(47-1 downto 0));
end component;


constant c0 : std_logic_vector (14 downto 0) := "000111101110000";
-- 0.48242
constant c1 : std_logic_vector (14 downto 0) := "001101011000101";
-- 0.83654
constant c2 : std_logic_vector (14 downto 0) := "000011100101100";
-- 0.22412
constant c3 : std_logic_vector (14 downto 0) := "111101111011100";
-- -.12939

signal buff_A : std_logic_vector(n*4-1 downto 0);
signal buff_B : std_logic_vector(15*4-1 downto 0);
signal buff_SUM : std_logic_vector(47-1 downto 0);
signal step1: std_logic;
signal buff_TC : std_logic := '1';
signal temp : std_logic_vector(n-1 downto 0);

begin


mult1: DW02_prod_sum_inst port map (
      inst_A      => buff_A,
      inst_B      => buff_B,
      inst_TC     => buff_TC,
      SUM_inst => buff_SUM

);

------------------------------

process
begin

wait until rising_edge (clk);

buff_A <= a0 & a1 & a2 & a3;
buff_B <= c0 & c1 & c2 & c3;

step1 <= start;

end process;

--------------------------

process
begin
wait until rising_edge (clk);

if (step = '1') then
      temp <= shr ( buff_SUM, "1101");
end if;

finish <= step1;

end process;

q <= temp;

end syn;
```

#DW02_prod_sum_inst

```
library IEEE,DWARE,DW02;
use IEEE.std_logic_1164.all;
use DWARE.DWpackages.all;
use DW02.DW02_components.all;

entity DW02_prod_sum_inst is
      generic (
          inst_A_width : NATURAL := 32;
          inst_B_width : NATURAL := 15;
          inst_num_inputs : POSITIVE := 4;
          inst_SUM_width : NATURAL := 47
          );
      port (
          inst_A : in std_logic_vector(inst_num_inputs*inst_A_width-1
downto 0);
          inst_B : in std_logic_vector(inst_num_inputs*inst_B_width-1
downto 0);
          inst_TC : in std_logic;
          SUM_inst : out std_logic_vector(inst_SUM_width-1 downto 0)
          );
    end DW02_prod_sum_inst;

architecture inst of DW02_prod_sum_inst is

begin

    -- Instance of DW02_prod_sum
    U1 : DW02_prod_sum
      generic map ( A_width => inst_A_width, B_width => inst_B_width,
num_inputs => inst_num_inputs, SUM_width => inst_SUM_width )
      port map ( A => inst_A, B => inst_B, TC => inst_TC, SUM =>
SUM_inst );


end inst;


-- pragma translate_off
library DW02;
configuration DW02_prod_sum_inst_cfg_inst of DW02_prod_sum_inst is
for inst
    for U1 : DW02_prod_sum use configuration
DW02.DW02_prod_sum_cfg_sim; end for;
end for; -- inst
end DW02_prod_sum_inst_cfg_inst;
-- pragma translate_on
```

#fxmult using ieee.arith.signed

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;


ENTITY fxmult IS
generic (n: integer := 32);

      port (
      clk, start: IN std_logic;
      a0, a1, a2, a3: IN std_logic_VECTOR (n-1 downto 0);
      q: OUT std_logic_VECTOR(n-1 downto 0)
      );
END fxmult;

ARCHITECTURE syn OF fxmult IS

signal z0, z1, z2, z3, tmp, tmp1: std_logic_VECTOR (n-1+15 downto 0);

begin

process (clk)
begin

if (start= '1') then
      z0 <= a0 * "000111101110000";     -- 0.48242   2bit bec 13bits
fract
      z1 <= a1 * "001101011000101";     -- 0.83654
      z2 <= a2 * "000011100101100";     -- 0.22412
      z3 <= a3 * "111101111011100";     -- -.12939

      tmp <= z0 + z1 + z2 + z3;

      tmp1 <= shr ( tmp, "1101");       --shift 13
end if;
end process;

 q <= tmp1(n-1 downto 0);

end syn;
```

#max.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;


ENTITY max IS
generic (n: integer :=32);

      port (
      clk, start : IN std_logic;
      num1, num2, oldmax : IN std_logic_VECTOR (n-1 downto 0);
      newmax: OUT std_logic_VECTOR(n-1 downto 0)
      );
END max;

ARCHITECTURE syn OF max IS

signal buff_newmax,temp: std_logic_VECTOR(n-1 downto 0);
begin
process (clk)
begin
if ( start = '1') then

      if (num1 > num2) then
            temp <= num1;
      else
            temp <= num2;
      end if;
end if;
end process;

process (clk)
begin
if ( start ='1' ) then

      if (temp > oldmax) then
            buff_newmax <= temp;
      else
            buff_newmax <= oldmax;
      end if;
end if;
end process;
newmax <= buff_newmax;
end syn;
```

#float2fix.m

```
close all
clear all
% Reading data ...
input_file = './b4Wavelet/band5.txt';
output_file = './b4wfix/band5.txt';
samples = 460;
lines = 400;
data=load(input_file);

fid = fopen (output_file, 'w');
% 3-d and 2-d variables to store the image
for ib = 1: samples
    for il = 1: lines
        fprintf (fid,sdec2bin(data(ib,il),32,10));
        fprintf (fid, ' ');
    end
        fprintf (fid, '\n');
end

status = fclose(fid);
```

#readoutput.m

```
% ====================================================
% Yuan He
% ECE Department
% Univ. of Tennessee
%
% he@student.ece.utk.edu
%
% This file is use to view results from chickv20.cpp
%====================================================

close all

samples = 460;
lines = 400;
bands = 65;

[fid, msg] = fopen('OUTPUTIMG1.IMG', 'r');
[ImageFile, count] = fread(fid, [460, 400*65], 'float');
status = fclose(fid);


% 3-d and 2-d variables to store the image
I = zeros(samples, lines, bands);
BandImage = zeros(samples, lines);

% find BandImage from ImageFile
for ib = 1: bands
    for il = 1: lines
        BandImage(:, il) = ImageFile(:, (il - 1)*bands +
ib);
    end
    I(:, :, ib) = BandImage;
end
clear ImageFile;


% Display a spectral image
figure, imshow(I(1:230, 1:200, 4))

%===========================================================
==========================
img2 = load('tumorimage.txt');
figure, imshow(img2(:,:))

%===========================================================
=======================

Normal = load('normdata.txt');
Tumor = load('tumordata.txt');
BGD = load('bgddata.txt');

figure, plot(Normal', ':b')
xlabel('Bands (Channels)')
ylabel('Relative Fluorescence Intensity (RFI)')
hold on
plot(Tumor', 'r')
plot(BGD', 'g')
hold off
axis([0 66 0 0.8])
```

Vita

Yuan He was born on March 22, 1979 in Beijing, China. For the first twelve years, he lived there and completed to his early education at Beijing Di Yi Shi Ying Xiao Xiu. In 1992, he came to the USA with his parents. He enrolled in the University of Tennessee, Knoxville in fall 1997 and received a Bachelor’s degree in Electrical and Computer Engineering in 2001. He continued working toward his master’s degree in the same department. He also works at the department’s Information Technology Administrator office and serves as the vice president of the Chinese Student and Scholars Association.