3-2001

A Low Power Application-Specific Integrated Circuit (ASIC) Implementation of Wavelet Transform/Inverse Transform

Daniel N. Harvala

A LOW POWER APPLICATION-SPECIFIC INTEGRATED CIRCUIT (ASIC) IMPLEMENTATION OF WAVELET TRANSFORM / INVERSE TRANSFORM

THESIS

Daniel N. Harvala
Captain, USAF

AFIT/GE/ENG/01M-14

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY
Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U. S. Government.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
7-03-2001	Master's Thesis	June 2000 - March 2001

4. TITLE AND SUBTITLE
A LOW POWER APPLICATION-SPECIFIC INTEGRATED CIRCUIT (ASIC) IMPLEMENTATION OF WAVELET TRANSFORM / INVERSE TRANSFORM

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Daniel Nick Harvala, Captain, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2950 P. Street
WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GE/ENG/01M-14

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. Robert L. Ewing
AFRL/IFTA
Air Force Research Laboratory, Bldg 620, Area B
WPAFB OH 45433
DSN: 785-6653 ext. 3592

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

13. SUPPLEMENTARY NOTES
Major Charles P. Brothers, Jr., Ph.D.
DSN: 785-3636 ext.4618 Email: Charles.Brothers@afit.edu

14. ABSTRACT
A unique ASIC was designed implementing the Haar Wavelet transform for image compression/decompression. ASIC operations include performing the Haar wavelet transform on a 512 by 512 square pixel image, preparing the image for transmission by quantizing and thresholding the transformed data, and performing the inverse Haar wavelet transform, returning the original image with only minor degradation. The ASIC is based on an existing four-chip FPGA implementation. Implementing the design using a dedicated ASIC enhances the speed, decreases chip count to a single die, and uses significantly less power compared to the FPGA implementation. A reduction of RAM accesses was realized and a tradeoff between states and duplication of components for parallel operation were key to the performance gains. Almost half of the external RAM accesses were removed from the FPGA design by incorporating an internal register file. This reduction reduced the number of states needed to process an image increasing the image frame rate by 13% and decreased I/O traffic on the bus by 47%. Adding control lines to the ALU components, thus eliminating unnecessary switching of combination logic blocks, further reduced power requirements. The 22 mm2 ASIC consumes an estimated 430 mW of power when operating at the maximum frequency of 17 MHz.

15. SUBJECT TERMS
VLSI, Haar Wavelet Transform, Haar Wavelet Inverse Transform, FPGA, VHDL

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Charles P. Brothers, Jr., Major, USAF
U	U	U	UU	133	19b. TELEPHONE NUMBER *(Include area code)* 937-255-3636 ext.4618

AFIT/GE/ENG/01M-14

A LOW POWER APPLICATION-SPECIFIC INTEGRATED CIRCUIT (ASIC) IMPLEMENTATION OF WAVELET TRANSFORM / INVERSE TRANSFORM

THESIS

Presented to the Faculty

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Daniel N. Harvala, B.S.

Captain, USAF

March 2001

Acknowledgments

I would first like to thank my wife. She took care of our kids, our finances, and our every day problems so I could spend every waking minute in the VLSI lab. Without her support, I could not have put forth my best effort.

A special thanks goes out to my partner in crime, 1st Lt. Kirby Watson. Without his drive and determination, I would not have succeeded. Without his companionship, I would not have survived the infinite hours spent in the VLSI lab. Having someone to share in the pain made the whole experience somewhat bearable.

Finally, I would like to thank my advisor, Major Charles P. Brothers, for providing me with the insight necessary to make this thesis a success. Without his guidance the results of this work would have been trivial.

Daniel N. Harvala

Table of Contents

Page

List of Figures

Page

List of Tables

Page

Abstract

A unique ASIC was designed implementing the Haar Wavelet transform for image compression/decompression. ASIC operations include performing the Haar wavelet transform on a 512 by 512 square pixel image, preparing the image for transmission by quantizing and thresholding the transformed data, and performing the inverse Haar wavelet transform, returning the original image with only minor degradation. The ASIC is based on an existing four-chip FPGA implementation. Implementing the design using a dedicated ASIC enhances the speed, decreases chip count to a single die, and uses significantly less power compared to the FPGA implementation. A reduction of RAM accesses was realized and a tradeoff between states and duplication of components for parallel operation were key to the performance gains. Almost half of the external RAM accesses were removed from the FPGA design by incorporating an internal register file. This reduction reduced the number of states needed to process an image increasing the image frame rate by 13% and decreased I/O traffic on the bus by 47%. Adding control lines to the ALU components, thus eliminating unnecessary switching of combination logic blocks, further reduced power requirements. The 22 mm^2 ASIC consumes an estimated 430 mW of power when operating at the maximum frequency of 17 MHz.

A LOW POWER APPLICATION-SPECIFIC INTEGRATED CIRCUIT (ASIC) IMPLEMENTATION OF WAVELET TRANSFORM / INVERSE TRANSFORM

I. Introduction

1.1 Introduction

This document presents a piece in an overall research effort being conducted by the Dayton Area Graduation Studies Institutes (DAGSI). Currently, students at Air Force Institute of Technology (AFIT), University of Cincinnati (UC), University of Dayton (UD), and Ohio State University (OSU) are involved with the effort of advanced compression of video and audio communications and large image compression. Wavelet image compression using Field Programmable Gate Arrays (FPGA) is the focus of the UD research. This thesis effort expands upon image compression research by implementing the Haar transform/inverse transform on an Application-Specific Integrated Circuit (ASIC).

The main effort of this research was to create an ASIC with the same functionality as the existing Very High Speed Integrated Circuit Hardware Description Language (VHDL) behavioral description of the FPGA design. Efforts were not made to alter the

specific wavelet chosen for image compression/decompression. Savings in area, speed, and power are the primary goal. While a mathematical analysis of wavelets was not explored, a brief discussion of wavelets and their properties accompany this thesis to provide a theoretical basis.

1.2 Problem Statement

Image processing has always been a slow and difficult task since image resolution is directly tied to the number of sample points taken. To increase resolution, one has to increase the number of sample points. As the number of sample points increases, so does the time necessary to complete the computations (1). By using the 2-D Haar transform, a speedup is realized since the transform compresses the image information into a minimal number of coefficients. The quality of the reconstructed image obtained from the Haar transform is satisfactory for many applications including this research effort. By quantizing the wavelet coefficients, a greater compression ratio is attained allowing for a faster transmission rate. Quantizing the wavelet coefficients causes some image degradation, however, the tradeoff between the speed of transmitting the data and the loss of image integrity is necessary. Once the compressed image is received the reverse transform is applied leaving the original image with only a minimal loss of integrity. The integrity loss is relative to the level of quantization and thresholding performed. For many applications, the speedup obtained by transforming and quantizing the image greatly outweighs the minimal loss of the image integrity (2).

The current technology has been mapped on an Annapolis Microsystems Wildforce board (3). The board consists of 5 Xilinx FPGA 4062XL chips with up to 2Mbytes of SRAM per chip. A PCI interface exists on the board providing several I/O options. External FIFOs, DMA, and a reconfigurable Crossbar between FPGAs are among the I/O options.

The Wavelet ASIC research effort replaces the 5 FPGA board with a single ASIC.

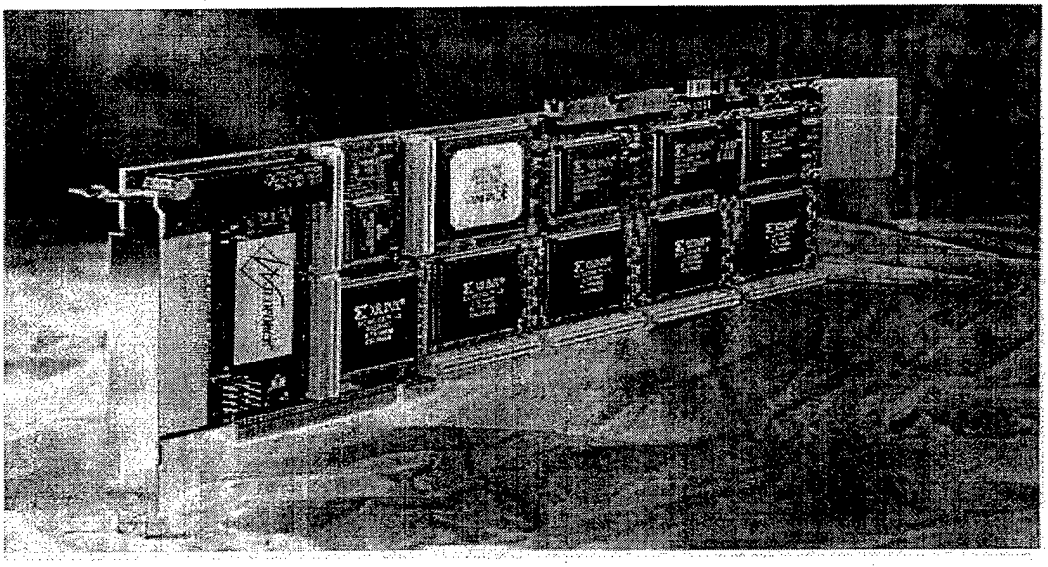


Figure 1. Wildforce Board (4)

1.3 Methodology

Converting from the FPGA design to an ASIC design required many iterative design steps. The first step is to compile and execute the existing VHDL behavioral code. Understanding how the current implementation operates is the key to translating and improving the code. Second, optimizations are performed to obtain a performance speedup, reduced area, and reduced power consumption. Next, a new behavioral VHDL

description is written to reflect the optimizations and tested. The new description is built with manageable blocks for easier implementation. A 9-bit adder is an example of a manageable block since it performs one function. Each block undergoes transitions from a behavioral description to a structural description and then to a physical layout. Each step is tested and revisited until it satisfies the required speed, area, power, and functionality criteria. Selected blocks are grouped together and tested for further verification. The final step is to test all blocks together.

1.4 Constraints and Assumptions

Validation of the FPGA behavioral code is assumed since it has been observed to produce expected results for sample images. Additionally, the Haar transform implementation of the code is assumed correct. Mathematical operations performed in the code were altered only to reduce the time to do the calculations. The final ASIC design process was verified against the behavioral VHDL code and produced identical results.

1.5 Materials and Equipment

All of the design tools used to create the layout operate in a UNIX environment. The first tool was the Synopsys System Simulator, (VHDLAN), (5). It tested the behavioral, structural, and functional aspects of the design. Another tool used was the Synopsys Design Analyzer (5), which produces a gate level layout of the behavioral description. A schematic based design tool, Synopsys Graphical Environment, (SGE),

(5), was used to produce a structural level description of the VHDL code. Netlists are also produced by SGE, which are used to obtain the final transistor layout. Octools (6) translated the netlist into a transistor layout. Magic (7), is used to view and edit the layout produced by Octools, as well as to produce a transistor layout design. The layout level of the design is tested with two other tools, IRSIM (8), which tested the functionality of the transistor layout by performing a logic level test and High Accuracy Simulation Program with Integrated Circuit Emphasis (HSPICE) (9,10,11), which tested the functionality of the transistor layout with emphasis on accurate timing of the circuit.

1.6 Thesis Overview

This document is organized into 6 chapters. The first chapter provides an introduction, overview of ASIC design, the steps used in the design process, and the tools needed to complete the ASIC design.

Chapter II summarizes current research in wavelet/transform technology. Research in ASIC design is also presented. Background research in FPGA design is also described.

Chapter III begins by stating the goals of the Wavelet ASIC. The original VHDL behavioral code is then analyzed. Next, the steps taken to execute the Haar transform and inverse Haar transform are discussed in detail. Optimizations of the original design are then presented. The next section discusses the design of the new synthesizeable VHDL code. The chapter finishes by describing the causes of image degradation.

Chapter IV presents the design at the component level of abstraction. First, the design steps used to create a component are listed. Each component, as well as each of the main logic blocks, is described. Next, the different state machines are discussed along with the design choices made to create them. The custom built internal register file is described in detail. Next, the top-level signals are listed along with their functionality. Finally, the system data buses are discussed.

Chapter V focuses on the verification and validation of the design. Tests made to the original VHDL behavioral code are discussed first. The design cycle is discussed along with the tests conducted at each step. The testing of the individual component is also described. Finally, the read/write logic is discussed.

In Chapter VI, research conclusions are presented. Research goals and accomplishments are discussed. The chapter concludes with suggestions for future work for both the VHDL level, as well as the component level of research.

II. Literature Review

2.1 Introduction

There is an enormous amount of research in transforms. For over 30 years, Fast Fourier Transforms have been the topic of many books (12). Wavelet transforms, in contrast, are relative newcomers but they have spawned many new signal processing algorithms over the past 10 to 15 years (13). A brief discussion of the 2D Wavelet Transform is presented along with an introduction to the Haar wavelet. Wavelets introduce various tradeoffs with respect to power, timing, and chip area. A small number of wavelets were analyzed for their specific impact in these areas with respect to ASIC design. Only a few theses were found that involved end-to-end chip design. Specific points from these theses are discussed along with their relevance to the Wavelet ASIC research. The chapter concludes by describing the FPGA design of the Haar Wavelet transform/inverse transform.

2.2 2D Wavelet Transform

The wavelet transform is popular for use in encoding a signal. After transformation, the input is separated into two sequences. The average values of the original input are represented in the first sequence while changes are represented in the second sequence. In other words, the first sequence describes the general trend of the input and the other sequence shows departures from that trend (14). There are an infinite

number of wavelet transforms and the particular criteria for choosing one over another is application specific (14). The 2D wavelet transform is an example of a wavelet transform that exhibits characteristics useful to image processing (2).

Prior to the transform, an image is digitized and represented as a two-dimensional matrix of pixels. Each pixel value represents an intensity and color value as sampled at that point. Normally, there is a high correlation between adjacent pixels. Correlation between neighboring pixels results in redundancy in image information which is exploited by the transform. The resulting data is compressed into a compact reversible transform of the original image. There are two schemes associated with the transformation of images for encoding. The schemes are either causal or noncausal. In a broad sense, causal transforms permit a sequential encoding process while noncausal transforms require solving large systems of simultaneous equations. Noncausal transforms provide a higher compression ratio but are harder to implement since they do not use a sequential encoding method (15). For the ASIC design, the causal transform is used.

The two most interesting characteristics of an image are its edges and texture. The characteristics are expressed as variations in the intensity and color of the adjacent pixels and these variations occur on several different scales. Edges of large objects are observable at low resolutions while edges of smaller objects are visible only at higher resolutions. At very high resolutions, even the texture of an image is observed as variations in intensity. While both edges and texture are distorted when applying transforms, edges are more perceptible to the human eye (15).

Each iteration of the 2D Wavelet transform produces four sub-images. First, a row decomposition is performed and results in a high pass sub-image and a low pass sub-image. The two sub-images are then decomposed by columns, which produce a total of four sub-images (low-low, low-high, high-low, high-high) as shown in Figure 2. The three high pass sub-images contain the edge information. For example, the 'High, Low' sub-image contains the horizontal high-pass information and the vertical low-pass information (2).

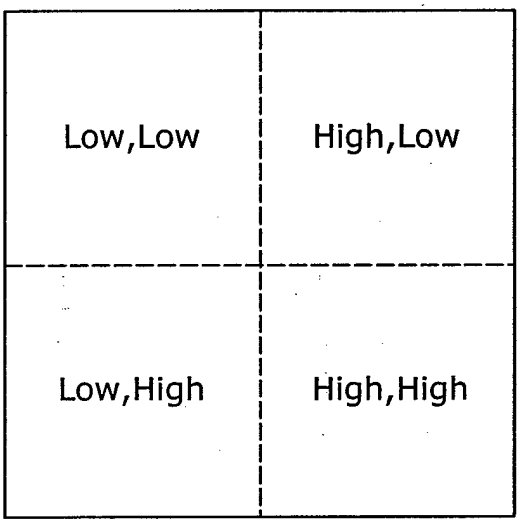


Figure 2. Sub-images

The fourth sub-image (low-pass image) is then transformed again producing four more sub-images with similar information but with lower resolution. The steps are repeated for a desired number of iterations. Usually the low-pass image after several iterations doesn't contain any more desirable information so the iteration is ceased (15).

Multiple passes are performed because image intensity changes may occur gradually. To localize the change in intensity, a low pass filter is applied to the image

which halves the intensity range. The divided intensity range is then examined for changes. By performing multiple passes, intensity variations are obtained at different scales. The multiple step transform allows both gradual changes as well as sharp transitions to be localized and saved for reconstruction of the image. The process of obtaining edge information at various scales is called multiscale edge detection and is very useful for image compression (15).

Wavelet compression is effective because the wavelet transform exploits the correlations in a signal. 1D transforms only exploit correlations in a small segment. 2D transforms find correlations within a region. The 2D transform, therefore increases the compression ratio (4). The increased compression ratio is an advantage of the 2D wavelet compared to other transforms (16).

Another significant advantage of the wavelet transform (4) is that it solves the synchronization problem between multimedia content streams such as adjacent video signals. Synchronization is also an issue in wireless LAN and Internet communications. In order to achieve synchronization, a time-control mechanism is needed. A 1D audio signal can be converted into a 2D signal to form an audio image. By attaching the audio image to the video image, synchronization is achieved (4).

In this section, the 2D wavelet transform and its application to image processing was discussed. For a more detailed analysis of wavelets, the interested reader is encouraged to consult (17).

2.3 Haar Wavelet

The wavelet chosen for the ASIC design, the Haar wavelet, is possibly the simplest of all the wavelets. Computation of the Haar wavelet is accomplished by simply averaging and differencing the data. These simple calculations are what make the Haar wavelet suitable for an ASIC implementation. Two types of coefficients are obtained from the transform. Scaling coefficients are obtained by averaging two adjacent pixels. These scaling coefficients represent a course approximation of the image. Wavelet coefficients are obtained from the differencing of two adjacent pixels. Wavelet coefficients contain the fine details of the image.

The Haar wavelet was chosen for its simplicity and speed of computation. Computation of the scaling coefficients requires adding two pixel values and dividing by two. Calculation of the wavelet coefficients requires subtracting two pixel values and dividing by two. The inverse transform simply requires subtraction and addition. Using logical shifts to perform division eliminates the need for a complex divide unit. Furthermore, implementing a logical shift in hardware requires much less power and space than an arithmetic logic unit (ALU). Given the computational requirements, the Haar wavelet is a simple and easy to implement transform. Computational simplicity makes the Haar transform a perfect choice for an initial design implementation. Further research is being conducted by UD to see if any advantages exist for using different transforms for their research effort.

2.4 History of Designs

Image compression is a required in many applications. One such application is in digital photography. Storing high-resolution images requires a significant amount of memory. Better compression results in more images being stored on a given storage media. One such effort involved designing a VLSI chip for Wavelet Image Compression (18).

The Joint Photographic Experts Group (JPEG) image compression algorithm is widely used for reduction of image data. JPEG is a real-time video/image processing application based on the Discrete Cosine Transform. However, JPEG has some drawbacks, such as artifacts being produced in the decompressed image. The artifacts are especially evident at the borders of the 8x8 sub-image and have resulted in exploration of other methods for image compression. Schwarzenberg's VLSI chip design for wavelet image compression is based upon wavelet transforms (18). A special Integrated Circuit (IC) was developed to perform image compression since software implementations of compressing even one still image requires a very long time. The speed of the Schwarzenberg's transform chip was obtained by performing certain operations in parallel (18).

Schwarzenberg uses a separable two-dimensional wavelet-transform. Performing a one-dimensional transform on the rows and then on the columns produces a separable 2-D transform. Schwarzenberg's design used internal RAM for the one-dimensional transforms allowing for increased speed since external RAM access was minimized. Since an ASIC wasn't actually built, valid operation of the design was based on the

synthesized code. The synthesized code performed identical to that of the software version of the wavelet compression (18). The FPGA design uses off-chip RAM to store intermediate values. The utilization of on chip memory is applied to the Wavelet ASIC transform design to gain speed.

Another research effort involved implementing a VLSI architecture for 2-D discrete wavelet transforms (DWT). The architecture was designed to process input signals in real-time. The VLSI DWT design used three programmable parallel filters, a storage unit, and a control unit, which minimized the hardware costs. The 2-D DWT design outperformed the direct approach, which uses the 1-D DWT. The direct approach only executes the transform in a row like fashion, which exploits correlations in small segments, not in regions. The direct approach has many shortcomings such as a long latency time and the requirement for a large memory space. Because of these shortcomings the 1-D DWT isn't widely used. The VLSI DWT approach had performance benefits over a direct approach making it suitable for many real-time video/image applications (19).

Singh, et al. (20), designed another application using a 2-D discrete wavelet transform. Parallel computation of the wavelet was proposed. The design is modular making it scalable to different levels of wavelet decomposition. A prototype architecture was implemented for an 8 x 8 image. The Singh architecture was synthesized and verified. Then a layout was designed in Cadence. The Singh design boasts fewer latches by utilizing control pipelining to generate the control signals. Control pipelining eliminated the need for latches for the horizontal dimension of the first stage processing

elements. The design used 3 stages of wavelet decomposition (20). The Haar wavelet implemented in the Wavelet ASIC also incorporates a 3-stage wavelet transform.

Zhang, et al. (21), proposed a 3D DWT. The 3-D transform is decomposed into three steps. Each step is a 1-D transform in the x, y, and z direction. Although the 3-D DWT outperforms the 2-D DWT by 40-90%, the 2-D and 1-D transforms still have their uses (21). The number of coefficients is proportional to the accuracy of the transform. Furthermore, as the number of coefficients increases, so does the time it takes to compute the transform. The 3-D DWT architecture was implemented with minimal area and predicts the consumption of less power. Low power was achieved in the 3-D DWT design by using low power building block cells, using central control design, which minimizes circuit complexity, eliminating redundant modules, and by constantly compromising tradeoffs of power, speed, and circuit complexity. The 3-D DWT design was verified with Synopsys software and is reported to use only 0.5W of power with a total delay of 91.65 ns while operating at a frequency of 272 MHz (21).

Another architecture was proposed by Lafruit, et al. (22), which greatly reduced power and memory usage. Lafruit's architecture reads the image data line by line, which results in a great area savings for on-chip storage. The method of reading line by line reduces complexity, which in turn reduces power consumption (22). Reading the data line by line was not used in the Wavelet ASIC design but an attempt to minimize the number of reads and writes was a goal. The Wavelet ASIC design uses internal registers to store image pixel values until the computation is completed and the results are written to memory yielding a speedup over the existing methodology. As with Lafruit, et al. reduction of power is obtained by minimizing memory access (22).

Hunt (23) explored some of the design issues associated with VLSI designs. Among the issues were the choices for synchronous versus asynchronous timing. Synchronous timing eliminates race conditions and other potential hazards by reducing or increasing the clock cycle time. The speed of combinational circuitry is not a concern if the clock cycle time is adjusted to account for the circuitry's operating speed. Power consumption of clock circuitry, however, is quite large since the clock is always switching. Switching is what uses power in CMOS designs. With an asynchronous design the power and area are reduced. However, with the absence of a clock, extra control circuitry is needed which sometimes offsets the area savings. With synchronous circuits, the speed is directly tied to the longest delay. An attempt to equally distribute the workload across all states should be made. Optimizing portions of a synchronous design, which are not in slow sections of the code doesn't increase performance. With asynchronous circuits the opposite is true. Since the next stage is waiting on the previous stage, the sooner it is completed the better (23). Aspects of both synchronous and asynchronous timing are used in the design of ASIC research effort.

Another design choice is deciding between performing operations in parallel by replicating components or operating in a serial fashion. For example, one can choose to use a single 32-bit adder and perform consecutive additions or replicate the adder and do additions in parallel. The former needs less die area but takes a longer time to compute which decreases throughput. The complexity of a single adder design is also increased. By replicating the adder, one can achieve faster operation and higher throughput. The cost is an increase in die area. The control circuitry is decreased in the parallel design but

not enough to offset the replication of components (23). Tradeoffs between adding extra states and parallelism are a major part of the Wavelet ASIC research.

Hauser looked into VLSI concerns, as well as Discrete Fourier Transform (DFT) implementations and their advantages of being placed on a chip (1). The DFT uses a finite set of sample points making it suitable for implementation on a digital computer Again, as the number of sample points grow, the time to compute the transform and the power needed to perform the computation also increases. Since the DFT uses multiplications to execute, time and power usage are issues.

Winograd demonstrated a reduction in the number of multiplications required by the DFT in 1978 (1). The class of algorithms known as the Winograd Fourier Transform Algorithms (WFTs) is able to compute a DFT with a minimal number of multiplications. The drawback to the WFT is the size of the algorithm. In other words, the size doesn't easily map to that of a VLSI chip. Hauser showed by using the Good-Thomas Prime Factor Algorithm (PFA) in conjunction with the WFT, the size of the algorithm is reduced and easily maps onto that of a VLSI chip (1). Since the goals of the Wavelet ASIC are low power and fast computation, the Haar wavelet transform is the best choice because it requires only addition, subtraction, and shifting to compute it's coefficients.

2.5 Current Research

Research at the University of Dayton implements the Haar wavelet transform using a Field Programmable Gate Array (FPGA) (Figure 3). An image is captured via a camera and then transformed, quantized, and encoded creating a compressed image. To

retrieve the original image, the process is reversed. The reconstructed image is the output from the inverse transform step.

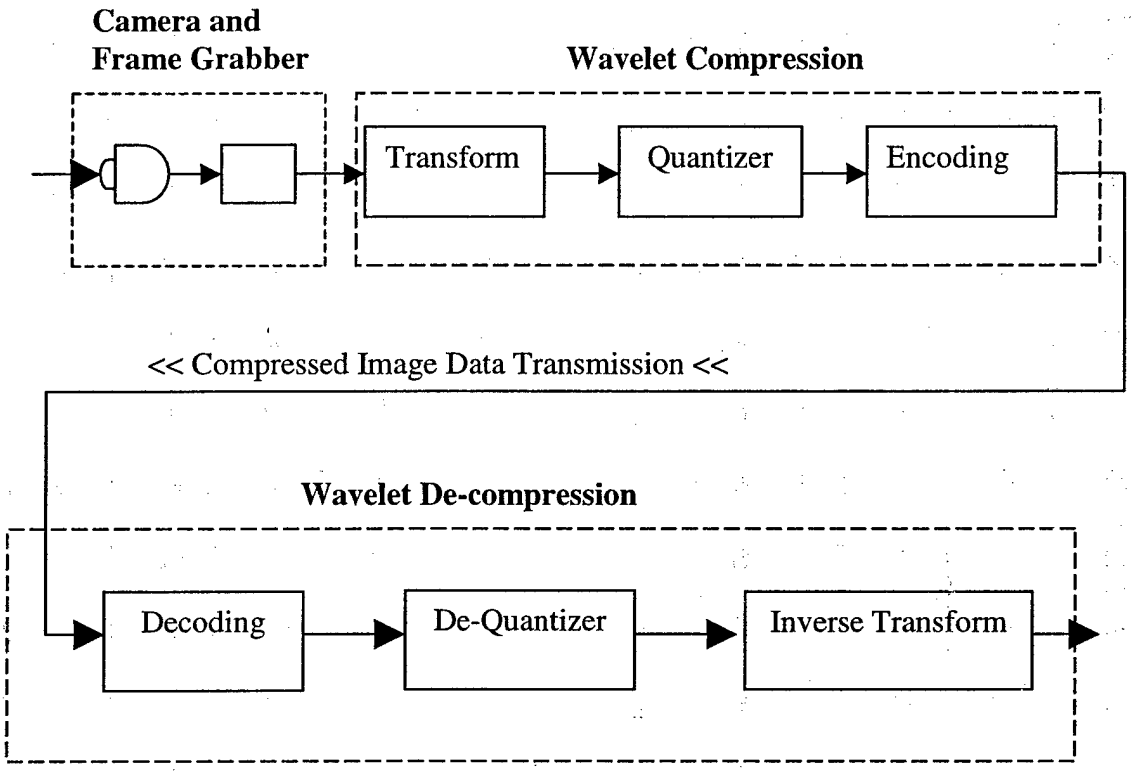


Figure 3. Image Compression/Decompression Flowchart

The FPGA design starts with a behavioral VHDL code level description. The VHDL code is used to program the FPGAs to perform the required tasks. The transform portion of the design is driven by the top level file, *Compression.vhd.* The *Compression.vhd* file uses 5 other VHDL files to perform the transform of the image. As Figure 4 shows, the image is processed first by rows and then by columns. First, one row is read in and packed. Next, the Haar transform is applied. After the row has been transformed it is unpacked. The three-step process is executed on all the rows. Next, the image is processed in column order. One column is read in and packed. Next, the

column is transformed. After the column has been transformed it is unpacked. The three-step process is executed on all the columns. Three iterations of the row/column process are executed by the transform portion of the FGPA design. The files used by *Compression.vhd* and their functionality are listed below.

Compression.vhd – Implements the 5 VHDL files listed in Figure 4.

Pack4row.vhd, Pack4Column.vhd – Packs four 8-bit pixel values into a single 32-bit integer. Packing the data speeds up the wavelet transform 4-fold. Pack4row.vhd packs an image pixel row to ¼ its original size; Pack4Column.vhd packs an image pixel column.

HaarVideo.vhd – Transforms one row/column of image data into wavelet coefficients (High Frequency coefficients) and scaling function coefficients (Low Frequency coefficients)

Unpack4row.vhd, Unpack4Column.vhd – Unpacks four 8-bit pixel values from a single 32-bit integer. Unpack4row.vhd unpacks pixels in an entire image pixel row; Unpack4Column.vhd unpacks pixels in an entire image pixel column.

The inverse transform half of the design is driven by the top level file, *Decompression.vhd.* The De*compression.vhd* file uses 5 other VHDL files to perform the transform of the image. As Figure 5 shows, the image is processed first by columns and then by rows. First, one column is read in and packed. Next, the Inverse Haar transform is applied. After the column has been inverse transformed it is unpacked. The three-step process is executed on all the columns. Next, the image is processed in row order. One row is read in and packed. Next, the row is inverse transformed. After the row has been

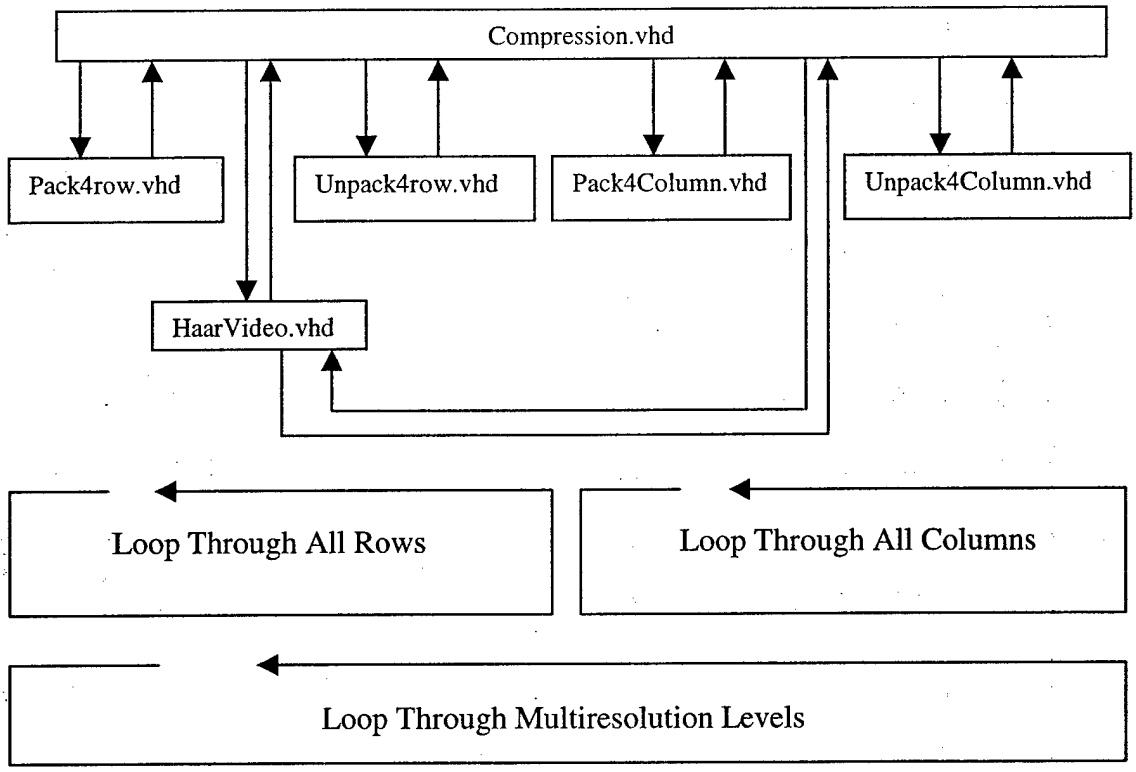


Figure 4. Compression.vhd File Flowchart (4)

inverse transformed it is unpacked. The three-step process is executed on all the rows. The inverse transform portion of the FPGA design executes three iterations of the column/row process. The files used by De*compression.vhd* and their functionality are listed below.

Decompression.vhd – Implements the 5 VHDL files listed in Figure 5.

Pack4row.vhd, Pack4Column.vhd – Same as in *Compression.vhd.*

InvhaarVideo.vhd – Inverse transforms wavelet coefficients (High Frequency coefficients) and scaling function coefficients (Low Frequency coefficients) into one row/column of image data.

Unpack4Column.vhd, Unpack4row.vhd – Same as in *Compression.vhd.*

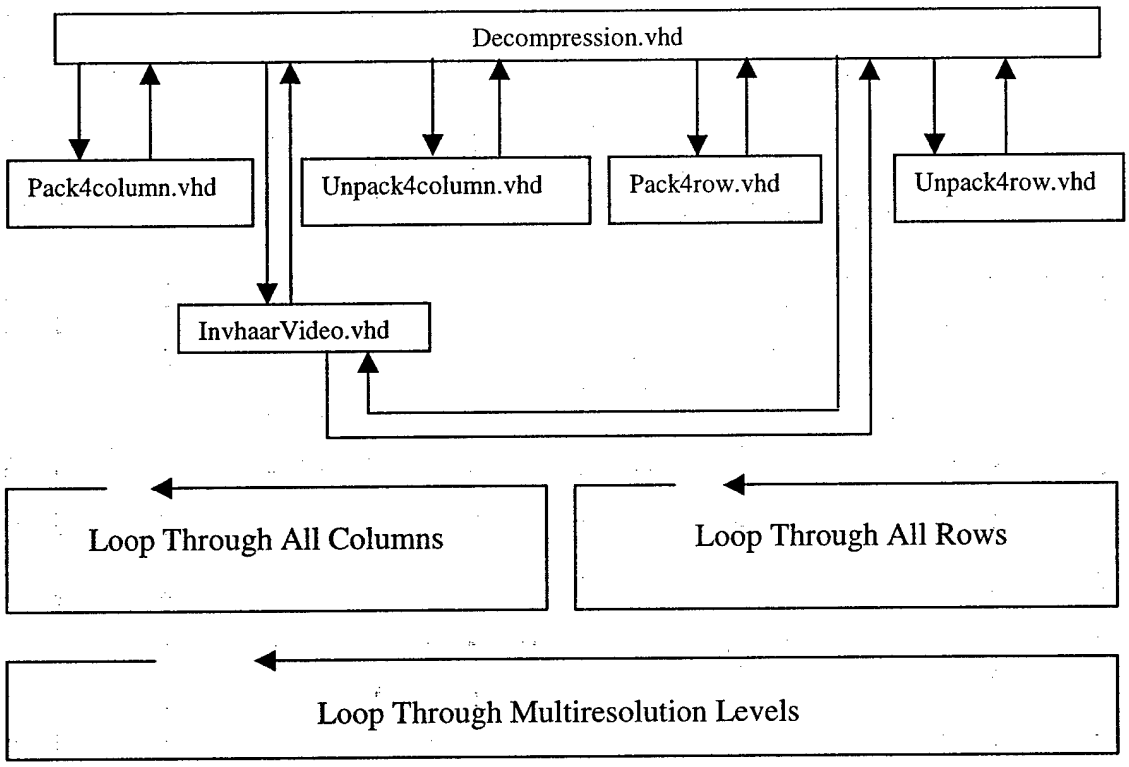


Figure 5. Decompression.vhd File Flowchart (4)

VHDL files for the main components are synthesized by the Wildforce system and the results are programmed onto the Wildforce board FPGAs (3). Software interfaces are used to control the programmed code on the FPGAs. First, the front-end code, written in C, is used to load an image into memory. The Wildforce FPGA implementation can handle image sizes of 16 x 16, 32 x 32, 64 x 64, 128 x 128, 256 x 256, 512 x 512, and 1024 x 1024 square pixels. After loading the image, the front-end code gives control to the *Compress.vhd* component by granting memory access. The image data is processed as shown in Figure 4. When the *Compress.vhd* component completes, the software takes control and simulates transmission of the compressed image. After simulated transmission, control is given to the *Decompress.vhd* (Figure 5)

component, which decodes the image data and executes the inverse transform. After the image has been reconstructed, another program, written in C, is used to display the image. The image has some degradation but is acceptable for many applications.

Static Random Access Memory (SRAM), which is used to store the image data, is located on a separate FPGA. The memory access time is two clock cycles from memory read to valid data on the data lines. A write operation takes one clock cycle. The bus controller and the memory controller are contained on other FPGAs.

Testing individual pieces as they were converted to FPGA compatible software was accomplished by running the other components not yet converted with the ones now on the FPGAs. Functionality of the VHDL code was demonstrated when a recognizable image appeared on the screen. The FPGA implementation has run at clock speeds up to 20 MHz. Advancements are currently in work to decrease the execution time of both the transform and inverse transform (4).

2.6 Summary

This chapter described many past and present research projects. First, the 2-D wavelet transform was analyzed for its applications relating to image processing. Next, the Haar wavelet was introduced. The Haar wavelet was chosen for the ASIC transform/inverse transform because of its simplicity. After introducing the Haar wavelet, research using other transforms was studied to gain an overall understanding about their applications relating to image processing. Finally, specific details from

different ASIC designs were studied. Many of the lessons learned from the different ASIC research efforts are applied to the Wavelet ASIC design.

III. Design Overview

3.1 Introduction

The behavioral VHDL code used to implement the Haar transform/inverse transform was obtained from the University of Dayton (UD) (4). Since the supplied VHDL code was written for FPGAs it wasn't readily synthesizeable and many changes were needed. The steps taken to translate the FPGA VHDL code to synthesizeable code are discussed in this chapter. Places to improve the code are described. The design flow of the synthesizeable VHDL code is explained and the differences between the FPGA code and the synthesizeable code are highlighted. Finally, degradation due to quantizing, thresholding, and shifting are discussed along with its impact on the usefulness of the image.

3.2 Goals

The goals of the ASIC research were directly tied with the current parameters of Wild Force Board application. Using the maximum operating speed of 20 MHz and the total number of states needed to transform one image, the FPGA design transforms one image every 196.609 ms. By increasing the operating speed and/or decreasing the number of states needed to process an image, the ASIC design will increase the frame rate. Adding control signals to the ALU components forcing them to switch only when necessary will minimize power usage. Chip area will be minimized by custom designing critical portions of the Wavelet ASIC. The FPGA implementation supports image sizes

from 16 x 16 to 1024 x 1024. For simplicity, the Wavelet ASIC uses a set image size of 512 x 512.

3.3 Analysis of Original Code

The analysis process began by compiling and testing the behavioral VHDL code received from UD. Initial tests were developed to determine the order pixels were accessed. Next, different operations performed on the pixels were analyzed. Operations performed on the pixels varied depending on the iteration of the transform and where in the image the current pixel information was obtained. The code was analyzed to determine the order of algorithm operations. Figure 6 shows the design flow of the FPGA code for the transform section. Only the specifics for the row operations are shown. Column operations occur in the same manner as that of the rows. The only difference is in the order the pixels are processed. Row operations read in the pixels from left to right. The column operations read in the pixels from the top to bottom.

The first step in the process is the packing of data. The FPGA implementation contained no internal storage, requiring intermediate values to be stored in off chip RAM. To minimize the RAM accesses the data was packed for later retrieval. Since pixel information only exists in the 8 least significant bits of a 32-bit memory word, the FPGA implementation reads in four locations and packs them into one 32-bit word. The 32-bits are then stored back to RAM for later retrieval. Subsequently all memory read accesses retrieve 4 pixels instead of only one.

The next step is the transform, which operates on the packed data. The packed data is read in, transformed, and written back to RAM. Once the packed data has been

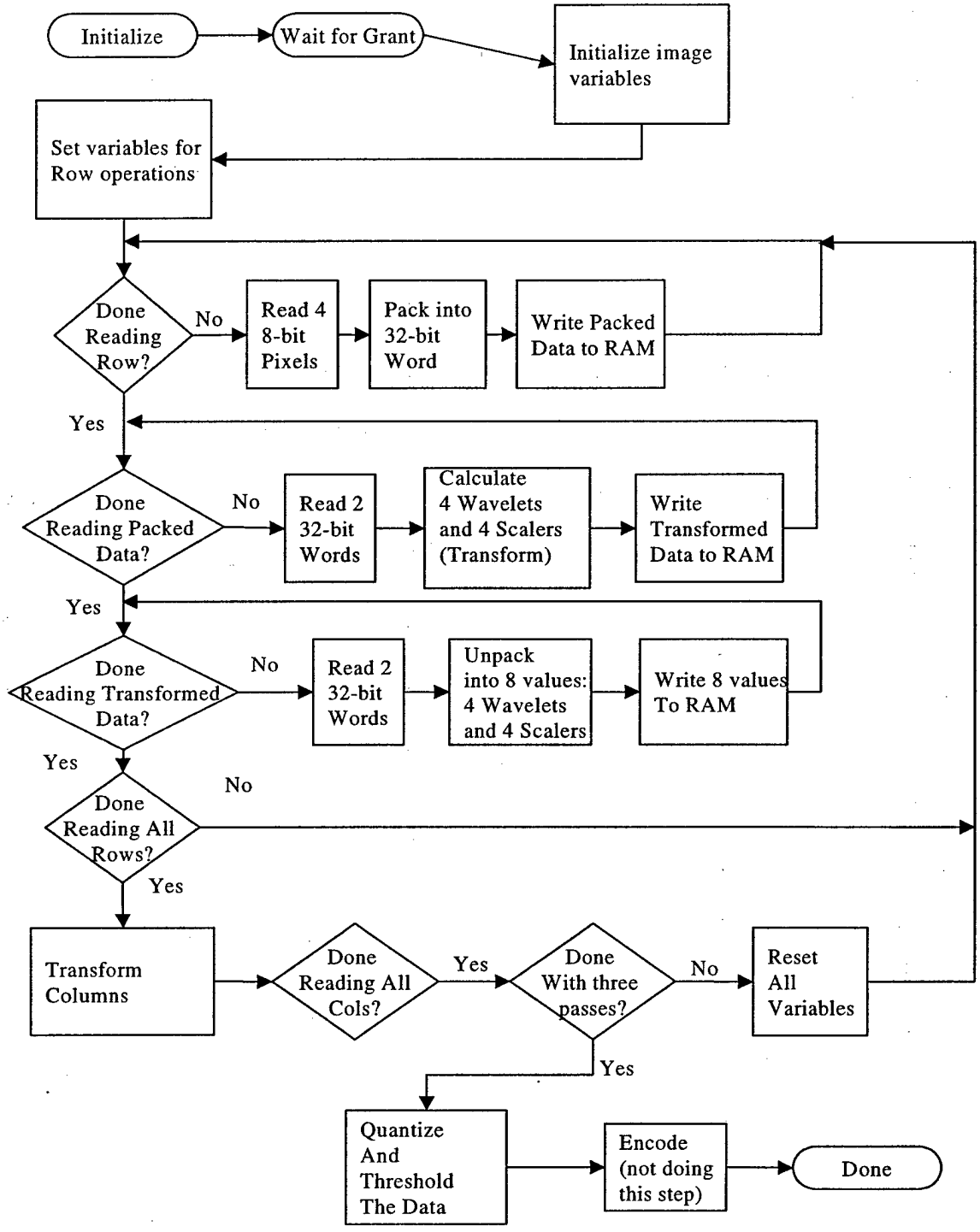


Figure 6. Flowchart Showing Transform Steps of FPGA Behavioral Code

transformed it is read in, unpacked, and stored as 8-bit pixel information within a 32-bit word. Three transform passes are performed on each image. During each subsequent pass the image size is decreased by a factor of four. The first pass processes an image size of 512 by 512 square pixels. The second pass processes an image size of 256 by 256 square pixels. The third and final pass processes an image size of 128 by 128 square pixels (Figure 7).

During the first pass, the transform processes the original image. Each subsequent pass operates on the scaling coefficients produced by the previous pass (Figure 8). Details of the subsequent passes of the wavelet transform are explained in Section 3.5. Each iteration only alters the original position of the current image i.e. the memory locations of the 256 by 256 square image are read in. After being calculated, the coefficients are written out to the same memory locations of the original 256 by 256 square image. After the three passes are performed, the image is quantized, thresholded and encoded. Details of the quantize and threshold steps are contained in section 3.4. The encode step is not part of the ASIC research and will not be discussed. After three passes, the transform is complete. Next, the inverse transform is applied.

The second half of the FPGA design recreates the original image from the wavelet transformed data file. The first step is to decode the encoded data file. Decoding details are not addressed, as encoding was not implemented in the ASIC development effort. The inverse transform process is simply the reverse of the transform process. The Haar transformed data is read in, packed, and written back out to memory. The packed data is then later read back in, inverse transformed, and written back out to memory. Finally, the

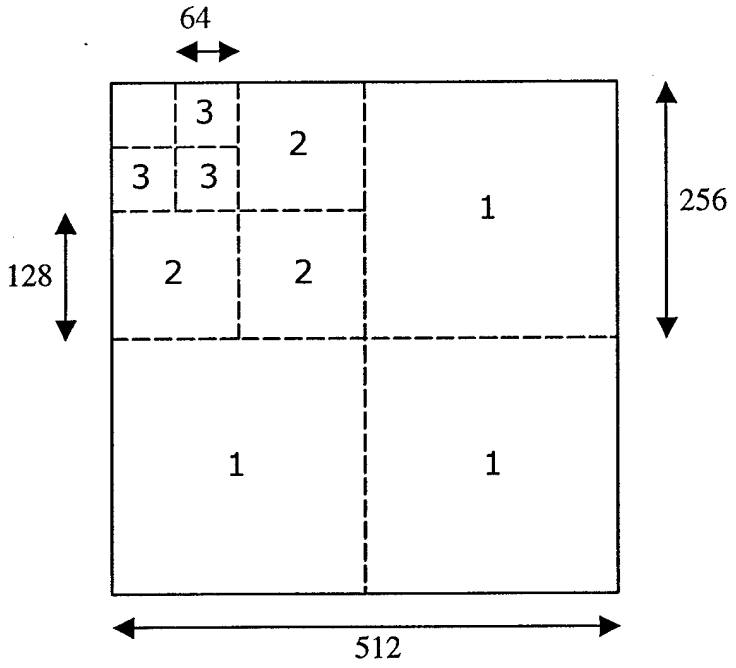


Figure 7. Transform Sections of Image

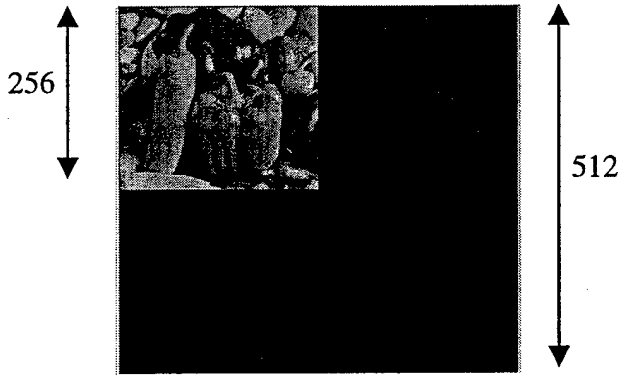


Figure 8. Result of One Transform Iteration

restored packed integer image is read back in, unpacked, and then written back to memory.

One difference between the transform and inverse transform is how the data is accessed. During the inverse transform, data is processed first by columns and then by

rows, which is opposite to that of the transform process. Another difference is the operations used to inverse transform the data. Details of the inverse transform operations are discussed in section 3.5. The order in which the memory locations are accessed is also different. The transform step operates on adjacent pixels creating coefficients. The coefficients are written to separate halves of the image. The inverse transform then operates on the coefficients. In other words, the memory accesses for the inverse transform are not sequential as in the transform step.

There are still three iterations. After each iteration the image is increased by four. That is, the first pass processes an image size of 128 by 128 square pixels. The second pass processes an image size of 256 by 256 square pixels. The third and final pass processes an image size of 512 by 512 square pixels. The first and second pass produce coefficients relative to their respective transform operations. The third pass produces the transform-degraded values of the original pixels. Details of the degradation are explained in the section 3.10.

The FPGA transform design takes 3,932,176 states to complete a three level transform and quantization of an image. The time to process one frame is 196.609 ms at a 20MHz operating speed. Appendix C contains a detailed breakdown of states for the FPGA design. The frame rate only measures the processing time of an image that already exists in memory. Associated operations like loading a new image into memory and transmitting the image would obviously affect the frame rate.

The FPGA code can process images of different sizes. The image width and height are located in memory locations one and two, respectively. When the FPGA code reads in the image size information, the internal counters are set to indicate the size of the

image being processed. A 512 by 512 square image is the only size image handled in the ASIC research, therefore, the details of dealing with the different sized images is not addressed. The details for quantizing and thresholding are discussed next.

3.4 Quantize and Threshold Rules

In the FPGA code, the quantize step is performed after both the row and the column transforms have been completed for all three iterations. The specific rules for quantizing and thresholding are presented next. See (24) for the specific, detailed information on the quantizing and thresholding process and theory. For explanation purposes, numbers are assigned to each quadrant. The numbering (Figure 9) is used to illustrate the different quantizing and thresholding rules for each quadrant.

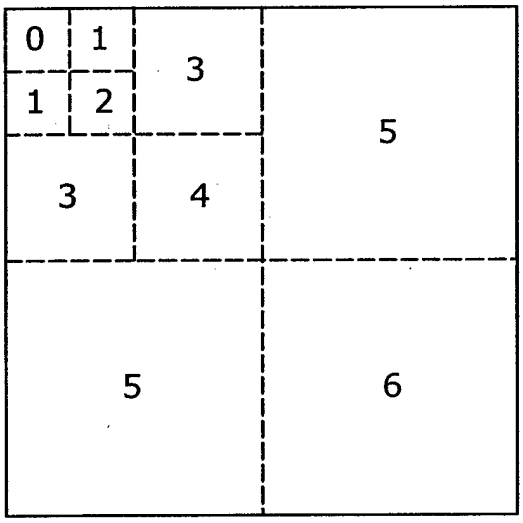


Figure 9. Quadrant Layout

In all passes the scaling coefficients are left alone. The reason some quadrants in Figure 9 have the same number is that the rules for processing those quadrants are the same.

The values are represented in hardware as 8 bits of data. When speaking of position of the bits the number is referenced from left to right as bit 7, bit 6, ..., bit 1, bit 0. Bit 0 is the least significant bit (LSB) and bit 7 is the most significant bit (MSB). The quantize and threshold steps alter the data which increases the compression ratio. The following quantization and threshold rules were taken straight from the FPGA code. The steps are executed in order for each value.

Quadrants 0:

Steps:

1. No altering of the data.

Quadrants 1:

Steps:

1. If number is negative and the LSB is equal to '1' then add 2 to the number.
2. Set the LSB equal to zero.

Quadrants 2:

Steps:

1. If number is negative and one or more of the lower two bits are equal to '1' then add 4 to the number.
2. Set low two bits equal to zero.

Quadrants 3:

Steps:

1. If number is negative and one or more of the lower two bits are equal to '1' then add 4 to the number.
2. Set low two bits equal to zero.
3. If value is less than –64 set equal to –64.
4. If value is greater than 64 set equal to 64.

Quadrants 4:

Steps:

1. If number is negative and one or more of the lower three bits are equal to '1' then add 8 to the number.
2. Set low three bits equal to zero.
3. If value is less than –64 set equal to –64.
4. If value is greater than 64 set equal to 64.

Quadrants 5:

Steps:

1. If number is negative and one or more of the lower three bits are equal to '1' then add 8 to the number.
2. Set low three bits equal to zero.
3. If value is less than –8 set equal to –8.
4. If value is greater than 8 set equal to 8.

Quadrants 6:

Steps:

1. If number is negative and one or more of the lower four bits are equal to '1' then add 16 to the number.
2. Set low four bits equal to zero.
3. If value is less than –8 set equal to –8.
4. If value is greater than 8 set equal to 8.

3.5 Wavelet Transform/Inverse Transform portions of the code

The Haar transform is very simple. The scaling coefficient is the sum of two pixels divided by two. The wavelet coefficient is the difference of two pixels divided by two. To retrieve the original pixel values the inverse Haar transform is executed. The sum of the scaling coefficient and the wavelet coefficient retrieves the first pixel. Subtracting the wavelet coefficient from the scaling coefficient retrieves the second pixel. The pixels are recovered with no loss in value. Table 1 depicts this process.

Pixel	Value	Scaler	Wavelet	inverse
Pixel 1	5	(5+6)/2 = 5.5	(5-6)/2 = -0.5	5.5 + (-0.5) = 5
Pixel 2	6			5.5 - (-0.5) = 6

Table 1. Example of Transform/Inverse Transform

The image is transformed first by rows and then by columns. When the rows are transformed the left side of the image space consists of scaling coefficients and the right side of the image consists of wavelet coefficients (Figure 10. Image B). Next, the image

is transformed by columns. When the column transform is complete the image consists of four quadrants (Figure 10. Image C). Upper left is the scaling coefficients. The upper right is wavelet coefficients showing the horizontal edges of the image. The lower left is wavelet coefficients showing the vertical edges of the image. The bottom right quadrant is wavelet coefficients showing the diagonal edges of the image (2).

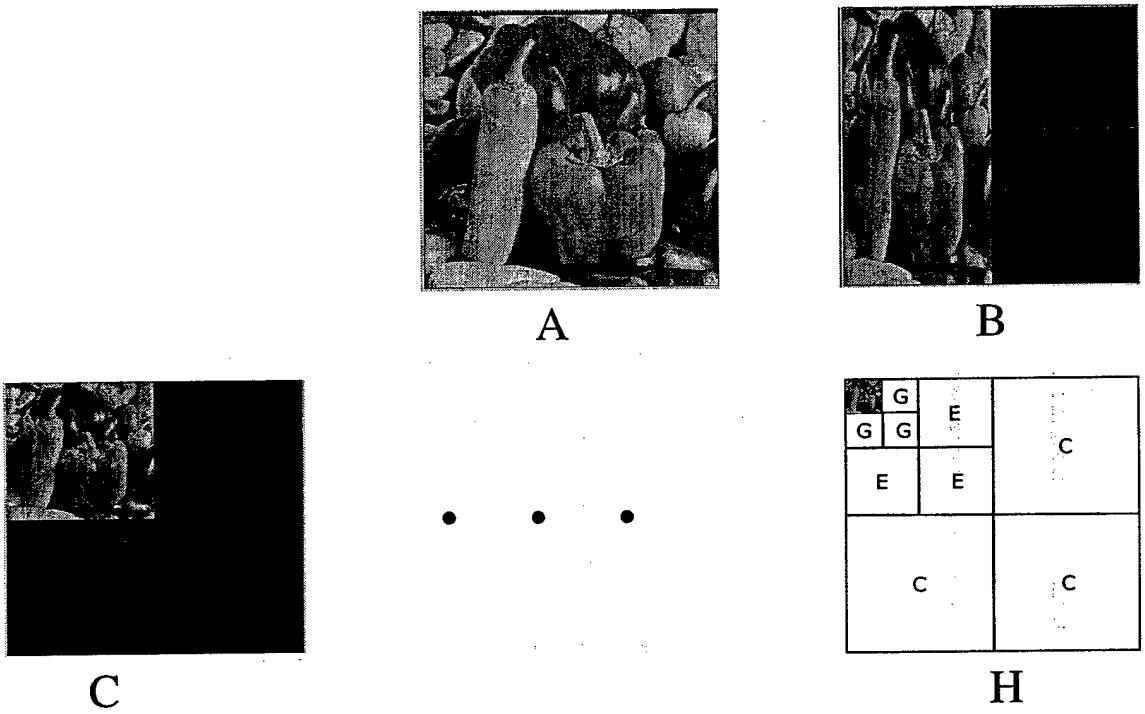


Figure 10. Transform of Image

Figure 10, Image C, shows the first iteration of the transform. The second iteration would only operate on the upper left quadrant (Figure 10. Image C). It would produce the same four quadrants as the first iteration but of different resolution levels (2). The third pass would operate on the upper left quadrant from the second iteration. The result of three iterations is shown in Figure 10, Image H.

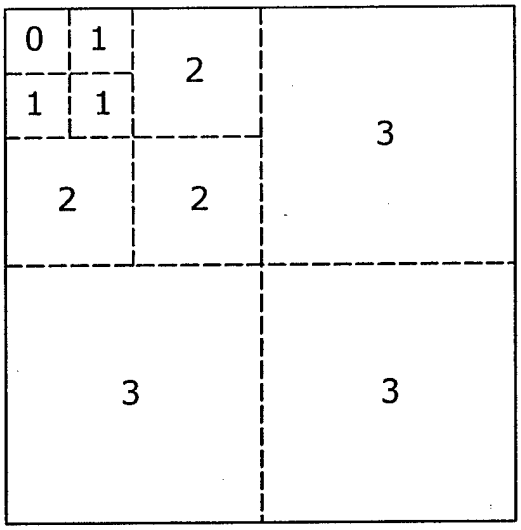


Figure 11. Quadrants of the Three Transform Passes

The inverse transform would operate on the image in reverse order to the transform process by first processing the columns and then the rows. The first pass would operate on the squares labeled '0' and '1' in Figure 11. The second pass would operate on the result of the first and the squares labeled with a '2'. The final pass would operate on the whole image recovering the original image. Since the calculations are being performed in hardware, fractional values are lost during integer division, and the original image isn't perfectly recovered. Other factors such as quantizing and thresholding the data are the major reason for distortion in the final retrieved image. For a more detailed description of these distortions, reference Section 3.10. Next, the steps taken to convert from the FPGA VHDL code to synthesizeable code are discussed.

3.6 Converting the Original FPGA Code to Synthesizeable Code

Once the behavioral level VHDL code was working it was necessary to convert it to a synthesizeable form. Converting the FPGA compatible code to synthesizeable code was a complex task. The following statement is an example of the coding style used in the original VHDL behavioral code.

indx1(18 downto 0) <= indx(18 downto 0) + 10;

Similar types of statements occurred simultaneously in the behavioral VHDL code. The first problem with above segment of code is the utilization of bit vectors. AFIT synthesis tools are not compatible with the bit vector construct of the VHDL language. All bit vectors had to be converted to individual bits, making the code longer and harder to follow. The '+' signs were also inappropriate. The following analysis assumes that 4 additions occur simultaneously. At least three methods are available to implement additions. The first method implements four adders that operate simultaneously. The second method places each addition in a different state, thus solving the problem serially. The second method takes four times as long to compute but requires only one adder. Other combinations such as two adders and two states were explored. An analysis of extra states and replication of components was done to decide what combinations of states and adders were best. The analysis is explained in section 4.6.

The tradeoffs for multiple additions are as follows:

1. Multiple adders working in parallel increase power usage and increase overall area.
2. A single adder increases the execution time because extra states are needed.

3. A combination of multiple adders and more states provided the best solution for the ASIC design, allowing for some speedup without the area becoming too large.

Many of the defined signals were not used in the original behavioral VHDL code. If two values are added they must be the same size since the '+' operator was used. For example:

newaddress(18 downto 0) <= address(18 downto 0) + count(18 downto 0)

Even if *count* only used four bits, *count* must be 19 bits long in order for the addition operation to compile in VHDL. As part of generating synthesizeable VHDL, all unnecessary bits were removed from the code.

Another non-synthesizeable portion of the behavioral VHDL was the code for the data latches. Changes were required so latches would properly synthesize. Figure 12 shows an example of some code that was changed to synthesize properly. When the reset line was placed before the clock edge detection line, as it was in the FPGA code, the component wouldn't synthesize. Figure 13 shows the correct way to program a latch for synthesis.

```
if (RESET = '1') then
        PAKPXV5 <= '0';
      elsif ( ( CLOCK = '1' ) and ( CLOCK'event ) ) then
        if (ReadPixel3 = '1') then
            PAKPXV5 <= '1';
        else –no change
            PAKPXV5 <= PAKPXV5;
        end if;
      end if;
```

Figure 12. Incorrect Way to Code a Latch

```
if ( ( CLOCK = '1' ) and ( CLOCK'event ) ) then
        if (RESET = '1') then
            PAKPXV5 <= '0';
        elsif (ReadPixel3 = '1') then
            PAKPXV5 <= '1';
        else –no change
            PAKPXV5 <= PAKPXV5;
        end if;
    end if;
```

Figure 13. Correct Way to Code a Latch

The FPGA code used many RAM accesses to process an image. With some additional logic the RAM accesses were reduced. The details for reducing the Ram accesses and the savings from the reduction are explained in the following section.

3.7 Optimizations on the Original Code

Analysis of the initial behavioral VHDL code showed that the three stages of Wavelet transform operations (Pack, Transform, Unpack) could be combined reducing RAM accesses by 47%. As with most designs, off-chip memory accesses are a performance bottleneck. Minimizing the number of memory accesses greatly reduces the overall execution time. The FPGA implementation included no internal data storage provisions requiring RAM reads/writes of intermediate values to be stored in off-chip memory. The inclusion of an internal register file eliminated the storing and retrieving of intermediate values. Pixels were simply read in once, transformed, and written back out to RAM. Appendix B analyzes the exact savings from combining the three memory access steps. The additional steps to quantize and threshold the data were also

incorporated with the column transform step. In the FPGA code, the quantize step is performed after both the row transform and the column transform have been completed. By incorporating the quantize step into the column transform step, one additional memory access is eliminated. The details to the reductions in RAM accesses are presented in Appendix B. Combining all the above operations eliminated all the reads and writes the original code needed to execute the quantize step. For a 512 by 512 square image the savings were 524,288 memory accesses. Of course, the logic was more complex and the need for a 256 by 8-bit internal register file utilized more chip area. The total chip area used by the internal register file and its associated address decode logic is 8,969,114 lambda^2. See Appendix B. Savings of Ram Accesses, for detailed Read/Write access numbers. Table 2 summarizes the total savings for a 512 by 512 image.

Total Ram Accesses by Old System	2,588,672
Total Ram Accesses by New System	1,376,256
Savings over FPGA implementation	47%

Table 2. Total Savings in Ram Accesses

The next section explains the steps taken to develop the new synthesizeable VHDL code.

3.8 Development of the Synthesizeable VHDL Code

To constrain the scope of the research, noncritical portions of the FPGA design were not implemented. The noncritical portions are explained in this section. The specifics of the new VHDL behavioral code are also discussed. Detailed differences between the ASIC and the FPGA design are addressed. Some of the logic used

specifically for the FPGA design is incorporated in the synthesizeable VHDL code. The extra logic, 'artifacts' are listed.

The Encoder/Decoder portions of the original algorithm were not implemented as part of the Wavelet ASIC design. The encoder simply reduced the transformed data file for transmission while the decoder expanded the compressed file back to its original size. Another functionality not implemented was the capability to process a variable size image. Rather, a constant 512 x 512 image size was used for the ASIC research. However, only a few minor design changes would be required to process any size image smaller than 512 by 512. The memory used by the FPGA started the image data at location 10. Since no internal storage existed, data relevant to the operation was kept in locations 0 through 9 and loaded every time it was needed. The same data memory mapping was retained for the ASIC development. As the design progressed it was found the data stored in locations 0 through 9 was not needed for the ASIC research. The impact of the FPGA memory address offset added one state to each of the four state machines use in the ASIC.

The original VHDL code allowed for a variable input of the number of transform levels to perform. Based on tests using the FPGA implementation, three transform levels was determined to be the optimum number of levels to perform (4). Therefore, the ASIC design used three transform levels for every image. Due to the hardwiring of three levels, the transform counter was reduced to 2-bits. Another signal, *icolumn,* was reduced from 19-bits to 10 bits. It was originally 19-bits long to accommodate for the '+' operation restrictions. Eliminating unnecessary signals made the code more compact and reduced unnecessary steps later in the design process.

Another change to the behavioral VHDL code was to break it into smaller more manageable files. The original VHDL code for each main section (Transform, Quantize, etc...) was contained in one file. To properly synthesize the design, smaller files, each containing less logic, were needed. First, the state machine and the state control signals were separated out. Next, all of the Arithmetic Logic Unit (ALU) type operations (Additions, Subtractions, Incrementers, etc...) were removed. Each of the operations was moved to their own separate file. The signals necessary for the operations were passed as input and output parameters to each file. Any component that could stand-alone was more efficiently implemented if synthesized by itself. Typically, design tools are more efficient when small modules are used. Testing the modules is also much easier and faster when it contains only a single operation.

Another artifact from the FPGA implementation code is bus arbitration. Bus arbitration along with the other FPGA's artifacts is implemented in the event the ASIC design is ever interfaced with the continuing FPGA effort. The ASIC design requires the assertion of the bus grant signal to a logic zero for the operations to begin. The original code did not allow for the bus grant signal to be deasserted once it was granted. The bus grant logic is the same for the ASIC as that of the FPGA design.

Since the FPGA artifacts are not needed by the ASIC design, further work on the ASIC design may require the removal of all the extra logic, clock cycles, chip area, and power needed to execute the additional steps.

3.9 Basic Operation of the New VHDL Code

The operation of the ASIC design is discussed in this section. The process begins by resetting the ASIC's states and ends when the image is retrieved. Both the transform and inverse transform processing steps are explained.

As shown in Figure 14, the process begins by resetting the states in the ASIC. Once the reset signal is asserted the top-level state machine (Appendix A.1) is initiated. The first state asserts the bus request. The circuit remains in a bus request state until the bus grant input is asserted indicating the bus has been given to the Haar ASIC for use. Once the bus grant is received, the image variables are initialized and processing of the image begins. First, the image pixel values are read in row by row. The first four pixels (in locations 0,1,2,3) are read from memory. The image's pixels are numbered 0 to 262,143 starting in the upper left corner as you view the image and proceeding left to right as shown in Figure 15.

The first row is numbered 0 to 511, the second row is numbered 512 to 1023, and so on. The image is assumed to reside in memory locations 0 to 262143. The actual algorithm operates on an image that begins at location 10. The starting address offset is an artifact from the FPGA implementation. Once the pixel values are read in from memory the transform is executed producing transform coefficients.

The scaling and wavelet coefficients are calculated from the pixel values. The two scaling coefficients are written back out to memory as shown in Figure 16

The wavelet coefficients are temporarily written to the internal register file in the same manner as the scaling coefficients were written out (wavelet coefficient1 is written

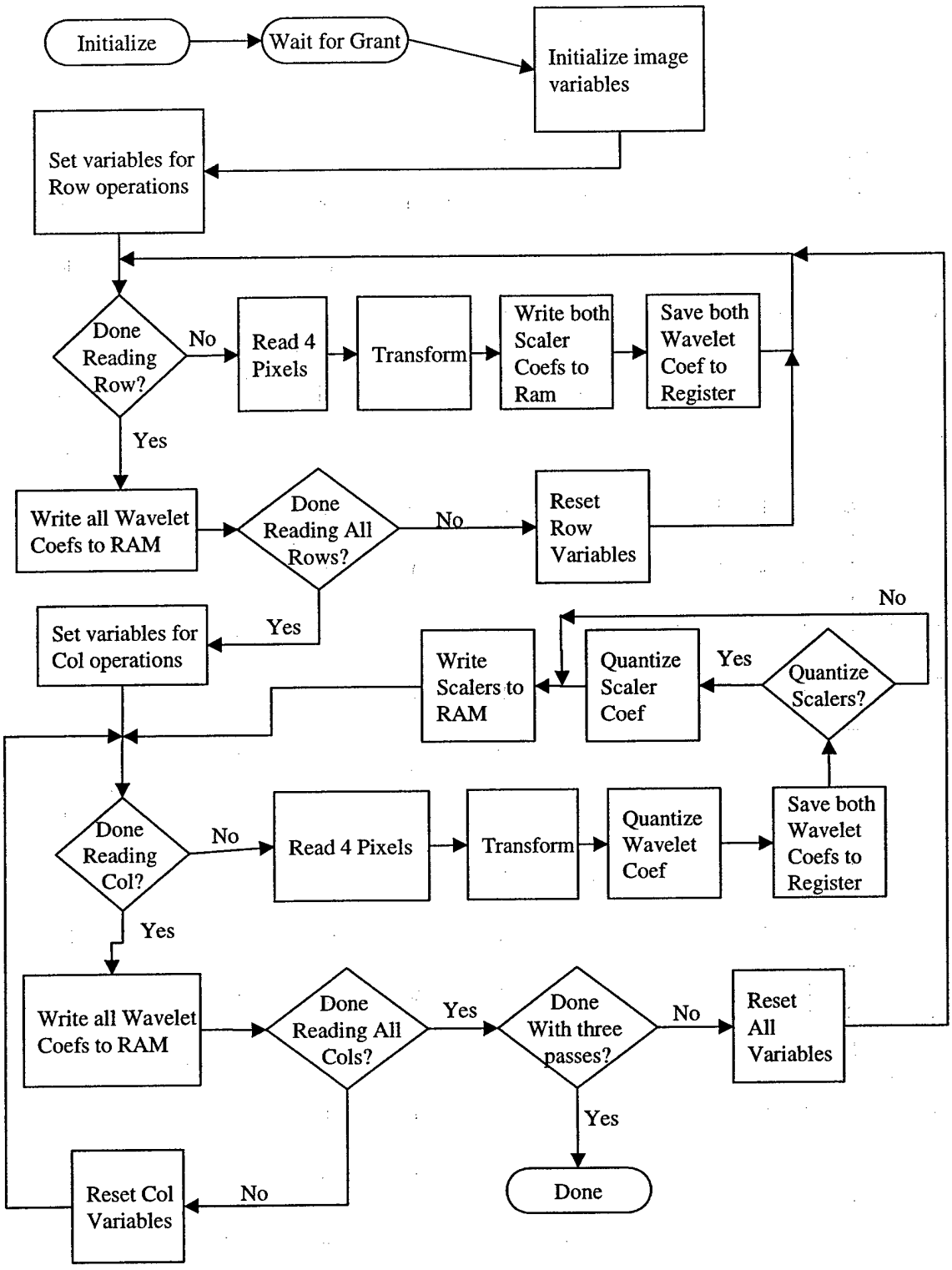


Figure 14. Flowchart Showing Steps to Complete the Haar Transform

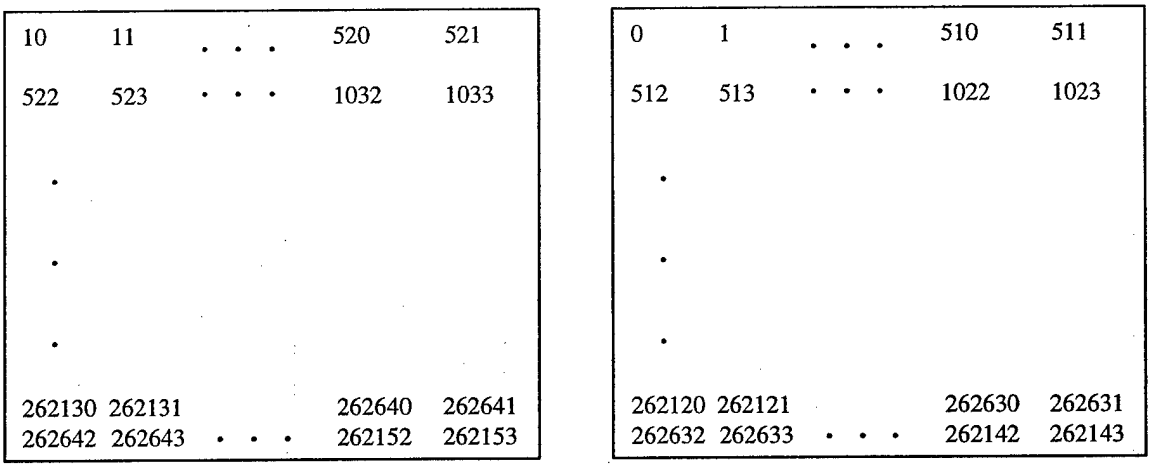


Figure 15. Address Mapping

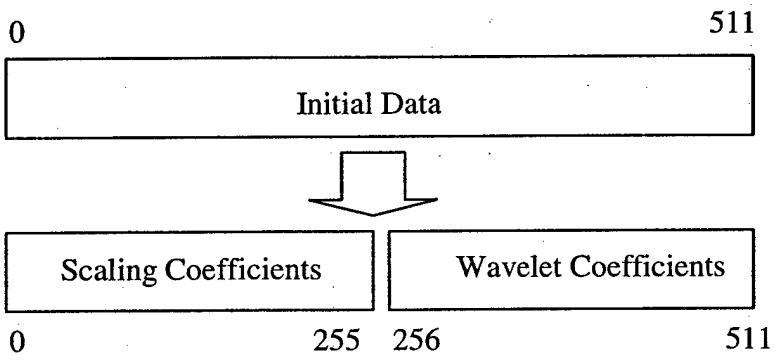


Figure 16. Placement of Coefficients Relative to Original Pixel Data

to register location 0 and the wavelet coefficient2 is written to register location 1). Once an entire row has been processed the wavelet coefficients are read from the internal register file and written out to main memory. After the register values are written to RAM, the scaling coefficients are located on the left half of the current image and the

wavelet coefficients are located on the right half of the current image (Figure 10, image B).

Next, the entire image is processed again but in column order. After initialization, four pixels are read in from memory starting with the left most column. During the column operations the scaling and wavelet coefficients are created in the same manner as in the row operations. After the coefficients are calculated, they are quantized and thresholded. The rules change for quantizing and thresholding depending on which iteration of the transform is being executed and on which quadrant the current pixels are being written. The rules for quantizing and thresholding are discussed in Section 3.4. Referencing Figure 9, the first pass quantizes and thresholds quadrants '5' and '6'. The second pass quantizes and thresholds quadrants '3' and '4'. The third pass quantizes and thresholds quadrants '1' and '2'. Once the coefficients have been quantized and thresholded they are written to RAM or the register file. The wavelet coefficients are written to the internal register file in the same order as the scaling coefficients were written out. Once an entire column has been processed the wavelet coefficients are read from the internal register file and written out to main memory. The scaling coefficients reside in the top half of the current image and the wavelet coefficients in the bottom half of the current image. The algorithm continues until all columns have been processed, resulting in a fully transformed, quantized, and thresholded image (Figure 10, image C). The row/column operations continue for three passes.

The next step, encoding the image, reduces the size of the data file for transmission yielding a quicker transmission time. The encoding step was eliminated as it was out of scope for the ASIC development.

The second half of ASIC design process involves the recreation of the image. Figure 17 summarizes the inverse transform process. As explained in Section 3.10, the recreated image is not a perfect replica of the original image.

The process of performing the inverse transform is much the same as the transform process. However, one difference is that the transformed data file is processed first by columns and then by rows. Bus arbitration is the same as in the transform case. Three iterations are required, however, the first iteration of the inverse transform is performed on the 128 square image, the second on the 256 square image and the third processes the 512 square image. The order in which the memory locations are accessed is also reversed.

The inverse transform continues until all columns have been read in and processed. Again each iteration works with a different area of the image. Next, the entire file is processed again, but in row order. Four pixels are read in from memory. The order the values are read in and the operations performed are exactly the same as that of the inverse column operations except the values are read in by rows. All of the rows are read in and processed.

The column and row inverse transform operations continue for three passes. Each time the image size is increased by a factor of four. Explicitly stated, the first pass processes an image size of 128 by 128 square pixels. The second pass processes an image size of 256 by 256 square pixels. The third and final pass processes an image size of 512 by 512 square pixels. The first and second pass produce coefficients relative to their respective transform operations. The third pass produces the full size image with some degradation. Image degradation is explained in the next section

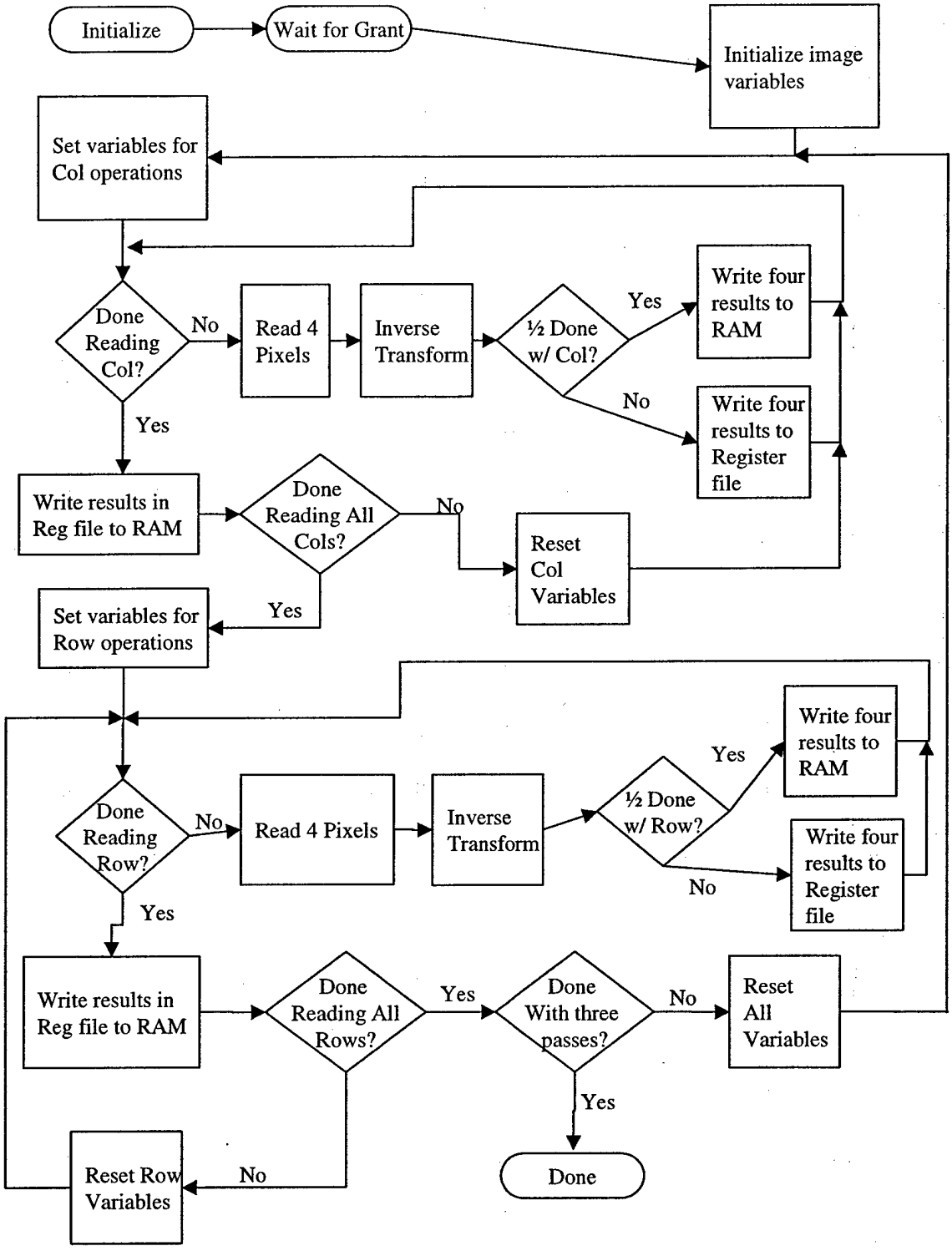


Figure 17. Flowchart Showing the Steps to Complete the Haar Inverse Transform

3.10 Degradation Due to Shifting, Quantizing, and Thresholding

The Haar wavelet transform introduces no degradation. However, in the VLSI implementation some degradation is introduced. The ASIC implementation uses integers to represent pixel values. Thus, when a divide is performed the exact quotient is not obtained and fractional remainders are truncated. Doing a shift on the register holding the values performs the divide. A single shift to the right is equivalent to a divide by two. The problem with a shift is the LSB is discarded. When an even number is shifted a zero gets shifted out resulting in no loss of data. However, when the number is odd there is a loss associated with the shift.

	15_{10}	=	1111_2		
Divide by 2	7.5_{10}	$\neq$	111_2	=	7_{10}
Multiply by 2	15_{10}	$\neq$	1110_2	=	14_{10}

Table 3. Example of Loss of Data Due to a Right Shift

Table 3 illustrates one place that degradation occurs in the ASIC design. Obviously, when the inverse transform is executed some of the original pixel values may be altered. Reference Table 4 for example of actual loss to individual pixels. Pixel 2 is recovered but Pixel 1 is recovered as a four not a five.

Pixel	Value	Scaler	Wavelet	inverse
Pixel 1	5	(5+6)/2 = 5	(5-6)/2 = -1	5 + (-1) = 4
Pixel 2	6			5 - (-1) = 6

Table 4. Example of Loss Due to Integer Shift

Degradation increases when negative numbers are involved. To help mitigate the impact of the loss an additional step was introduced. Depending on the iteration of the transform and quadrant being transformed, an offset value is added to negative numbers allowing for a more accurate recovery of the complete image. Degradation still exists but is lessened by the offset (4), (24).

Some loss of integrity is a tradeoff for increasing execution speed and reducing power consumption. Executing a bit-wise shift in hardware requires a simple routing of the signal lines. Implementing a Divide unit is much more complicated, requires a significant portion of area, and is much slower to execute than a right shift. The power used for an integer division operation is much greater than for a right shift operation. Degradation from the quantization and thresholding of the pixel values limits the overall accuracy of the reconstructed image. However, to obtain a greater compression ratio over the original image, required for a faster transmission time, degradation was deemed necessary.

3.11 Summary

The behavioral level VHDL code simulated operation of the wavelet ASIC properly when run with a 512 by 512 square image size. Reduction of RAM accesses reduced the power usage and decreased the time needed to transform an image. Degradation due to the transform, quantize, and threshold steps was a tradeoff for improvements to execution speed and compression ratio. For some applications the

degradation would be a hindrance. However, for the ASIC research, the degradation is an acceptable tradeoff.

IV. Design Implementation

4.1 Introduction

Detailed design of the Wavelet ASIC is covered in this chapter. Explanation is from a component level of abstraction. Some components are discussed in great detail. Others, such as multiplexers, speak for themselves. How the components fit together is also discussed. Most of the components were generated automatically from a behavioral description using the Synopsys Design Analyzer Tool (5). Certain blocks of logic, like the register file and its associated address decode logic, were custom built. Custom built components are more compact, consume less power, and run faster, but time constraints don't always allow for a full custom design. Off chip memory, bus control, and memory control were not part of the ASIC design. Only the provided code was translated and implemented.

4.2 Steps Used to Create a Magic Layout of a Component

The steps used to create a component starting with a behavioral description and finishing with a metal level layout in Magic (7) are as follows.

Step 1. Describe the logic in behavioral VHDL.
Step 2. Write a test bench in VHDL to test the code.
Step 3. Compile and test the code.
Step 4. Input the behavioral VHDL to Design Analyzer for synthesis.
Step 5. Optimize until satisfied with the timing and area usage.
Step 6. Convert output of Design Analyzer to input for Synopsys Graphical Environment (SGE) tool.
Step 7. Using SGE, hand place and route any D Flip Flops.

Step 8. Create structural VHDL of the logic using SGE.
Step 9. Compile and test the structural level code.
Step 10. Create a netlist of the logic using SGE.
Step 11. Convert netlist to Schematic Driven Layout (SDL) file format used by Octools.
Step 12. Verify the SDL file by comparing to netlist.
Step 13. Add commands to SDL forcing the ordering of the input and output signals.
Step 14. Use Octools to produce a Magic Layout of the logic.
Step 15. Use the Magic command 'drc check' to verify the correctness of the Magic layout.
Step 16. Extract a '.EXT' file from the layout.
Step 17. Convert '.EXT' file to IRSIM file format.
Step 18. Write an IRSIM test bench and test the logic.
Step 19. Convert '.EXT' file to HSPICE file format.
Step 20. Write an HSPICE test bench and test the logic.
Step 21. Connect with other components then test as a larger block.

The above steps are listed to help clarify when certain steps occur relative to each other.

4.3 Using the Synopsys Design Analyzer

The "increment by 10" component is used to illustrate how the design analyzer was used to design the ASIC Wavelet chip. Synopsys Design Analyzer takes behavioral VHDL code as input and creates a gate level layout of the logic. Design Analyzer can optimize the design for minimal area or minimal execution time. In the incrementer example, the first step is to describe the incrementer in behavioral VHDL. After the VHDL file is compiled and tested, the VHDL is used as input to the Design Analyzer. During the ASIC development effort, the initial iteration of the design analyzer is set to optimize the logic for a minimal amount of area. After the component is laid out it usually is necessary to optimize it based on the critical path. Subsequent optimization usually reduces the critical path time considerably as compared with the initial synthesis.

It has been proven with the Wavelet ASIC research and with past projects from the AFIT VLSI EENG 695 course, that iterating more that two times isn't necessary. The area of the component continues to expand yielding only a minimal increase in speed. The rule of only optimizing twice was proven with adders, subtractors, incrementers, and multiplexers. For the ASIC effort, components were, at most, optimized twice. The optimization steps for the incrementer are as follows:

1) Optimize on minimal area : Worst case timing = 13.71ns
2) Optimize on critical path : Worst case timing = 2.81ns

The critical path timing was reduced by 79%. As stated above with an additional iteration the timing is only reduced by small increments and the area continues to expand. The area does expand for the second iteration but trading area for a 79% speed up is a valid exchange.

4.4 Components

A complete list of components and their general description is listed in Appendix D. Many of the components are used more than once in the design. During some states, simultaneous additions and increments are executing. Adding additional states could have eliminated multiple operations occurring in a single state. A tradeoff was made with the number of states and the number of times to replicate components. Since incrementers use less area and power than adders, they were used whenever possible. Another tradeoff was in the control of each of the ALU type components. Each component had a control line associated with it. The control line caused the component to only switch when it was supposed to. Without the control line, the component would

switch whenever any of the input lines fluctuated. Incorporating the component level control added extra logic and area to the design but offered a savings in power as the components wouldn't unnecessarily switch. The HSPICE timing of all the components is contained in Appendix D.

4.4.1 *Adders/Subtractors*

There are three adders and one subtractor used in the Wavelet ASIC. The 8-bit adder, 9-bit adder and the 9-bit subtractor are all implemented with ripple carry logic. The timing from using ripple carry logic was sufficient for the small adders. The other adder was a 19-bit adder; it was implemented as a carry-select adder. The carry-select is larger in area but produces a much faster adder than the ripple carry logic (25). The reason for the custom sizing of the adders/subtractors was because certain states had multiple additions and subtractions being executed simultaneously. In one state the 8-bit and 9-bit adders and 9-bit subtractor are all being used. In some cases component reuse was selected. In one state there was a need for two 8-bit adders. The 9-bit adder was used for the second adder with the 9^{th} bit not being used. Using the 9-bit adder saved the building of a second 8-bit adder and the associated area with the component.

4.4.4.2 *Incrementers*

Several incrementers are required for the Wavelet ASIC. An increment by 10 was needed to account for the offset of the starting memory location of the input image. The 19-bit adder could have been used but it was already being used in the same state that the

increment by 10 is required. Other incrementers were built to accommodate both a 10-bit input and a 19-bit input. Again, the reason for multiple incrementers was that some states use more than one incrementer simultaneously. Rather than replicate the 19-bit incrementer and use it for smaller inputs, the 10-bit incrementer was built saving area and power.

The incrementers were implemented as ripple-carry adders. Ripple-carry adders are sufficient since only the LSB is '1' and all other bits are '0'. The incby10 was also designed using the ripple-carry adder logic. Hard coding '10' as the second input simplified its design. As explained earlier with further refinements and design choices the incby10 component could be completely eliminated since the offset by 10 of the image is an artifact of the FPGA design.

4.4.3 *Comparator*

A comparator was needed for the code since loops with end conditions needed to be tested. A comparator module was designed and tested. The following snippet of code was taken from the original design:

Original line of code:

```
(jshift(18 downto 0) >= Column(18 downto 0))
```

The following lines of code show the implementation of compare logic as written in behavioral VHDL for synthesis. The code segment only shows two of the 19 bits. RESUL is the output of the logic. If RESUL equals one then JSHIFT is greater than or equal to COLUMN. If RESUL equals zero then COLUMN is greater than JSHIFT.

```
JRES2 <= ((JSHIFT2 xor COLUMN2) and JSHIFT2);
CRES2 <= ((JSHIFT2 xor COLUMN2) and COLUMN2);
JRES3 <= ((JSHIFT3 xor COLUMN3) and JSHIFT3);
CRES3 <= ((JSHIFT3 xor COLUMN3) and COLUMN3);
if (JRES3 = '1' or CRES3 = '1') then
    RESUL <= JSHIFT3;
elsif(JRES2 = '1' or CRES2 = '1') then
    RESUL <= JSHIFT2;
else
    RESUL <= '1';
end if;
```

4.4.4 *Multiplexers*

Several multiplexers were used in the Wavelet ASIC. All the multiplexers are listed in Appendix D. Only the larger multiplexers were optimized during the design analyzer phase. Explicitly, the 6 x 19 input and the 7 x 19 input multiplexers were optimized once to reduce the critical path time. As stated in Appendix D, the 6 x 19 input multiplexer is the slower of the two. However, the propagation time is still minimal at 3.04 ns.

4.5 Four Parts of Code

The behavioral VHDL code was sectioned into four parts. Row transform (Figure 18), Col transform (Figure 19), Col inverse transform (Figure 20), and Row inverse transform (Figure 21). Quantizing and thresholding was incorporated into the Column transform section. Each part was tested separately. The transform pieces were tied together with a higher-level state machine called transform. The transform section was then tested (Figure 22 and Figure 23). The inverse transform pieces were tied together

with a higher-level state machine called inverse transform. The inverse transform section was then tested (Figure 24 and Figure 25). Since the two top-level pieces, transform and inverse transform, operate independently it wasn't necessary to test them together.

A high level multiplexer that is controlled by an input signal separates the two halves. The input signal chooses which half is executed. The other half remains in the reset state. Two additional signals control which data is routed to the output. The additional control allows isolation of smaller sections of the chip enabling the verification of these sections in the event the entire chip does not function correctly.

Each of the four main parts shares base level components: adders, subtractors, incrementers, and a comparator. Components were duplicated only when necessary to support simultaneous operation.

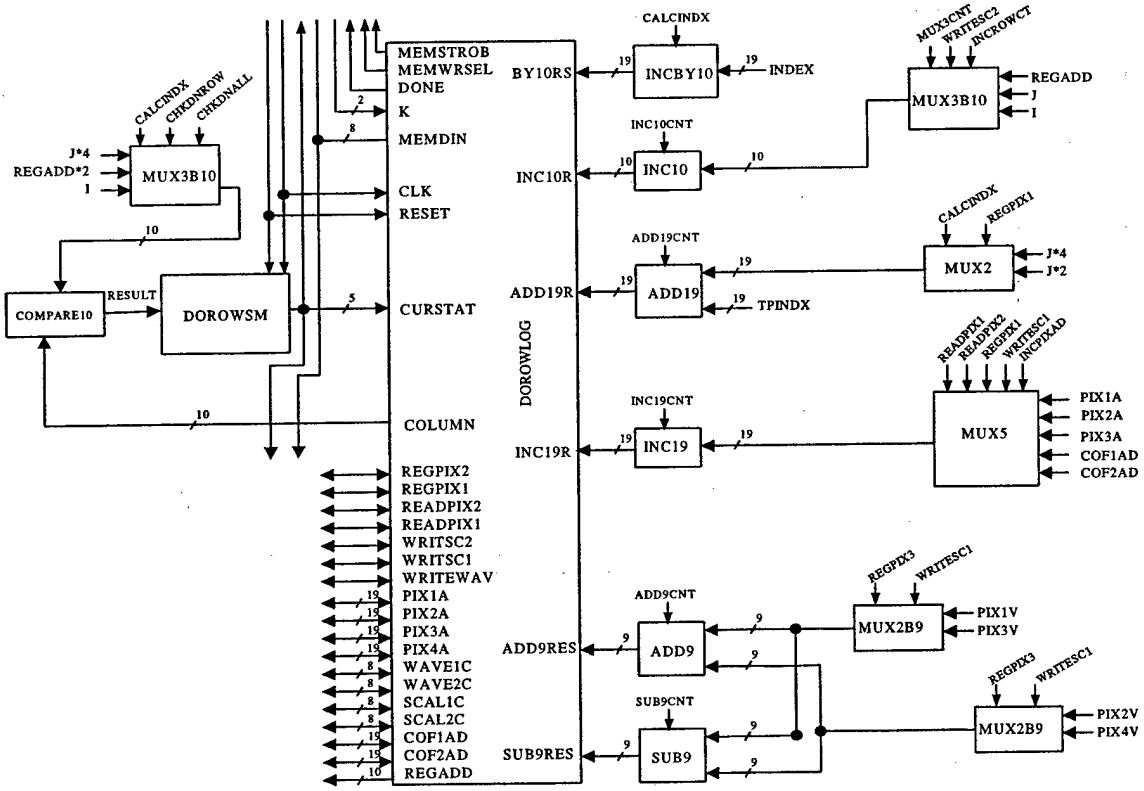


Figure 18. Row Logic for Transform

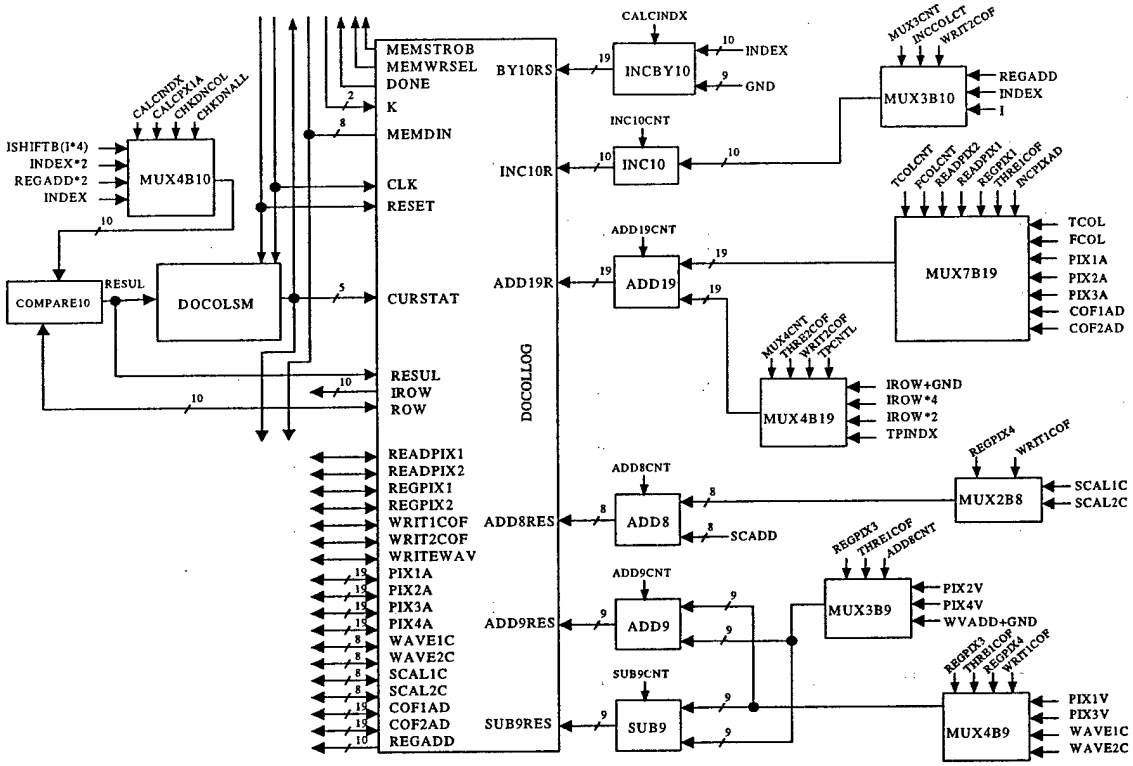


Figure 19. Column Logic for Transform

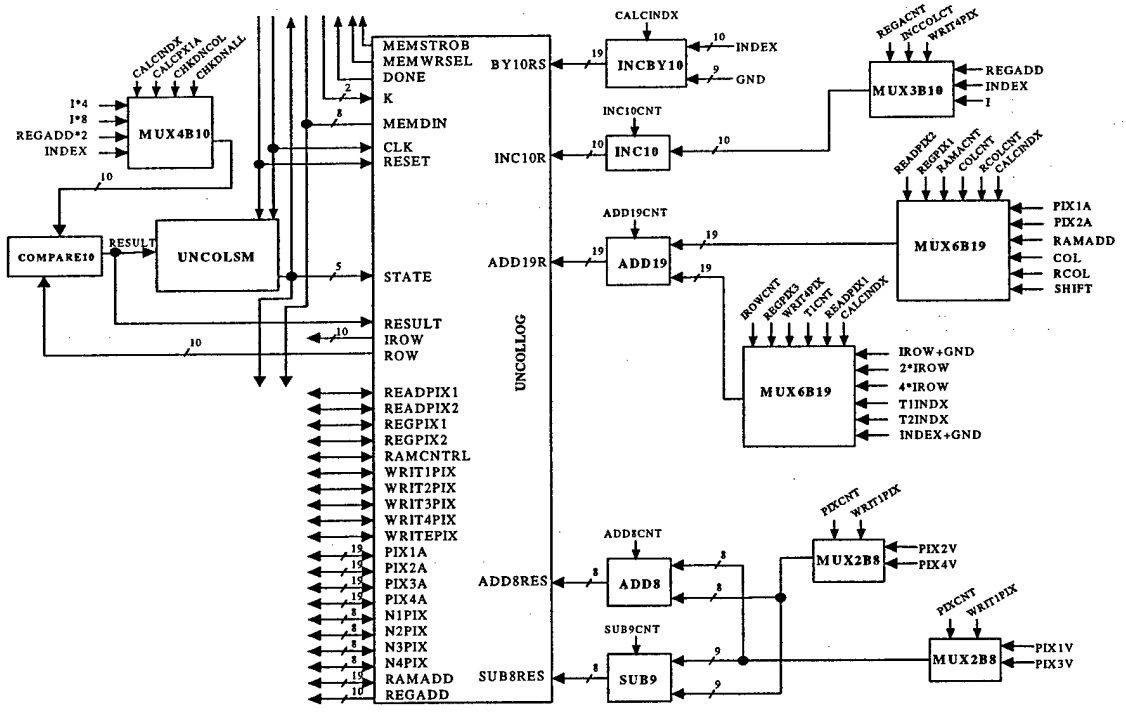


Figure 20. Column Logic for Inverse Transform

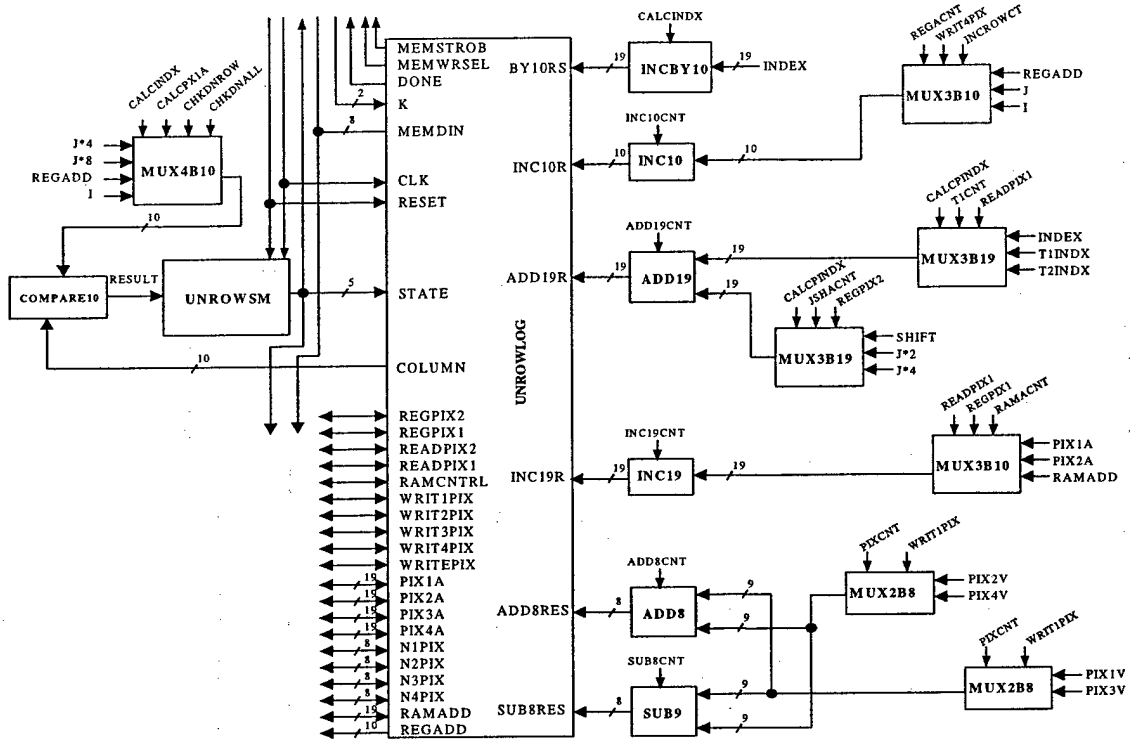


Figure 21. Row Logic for Inverse Transform

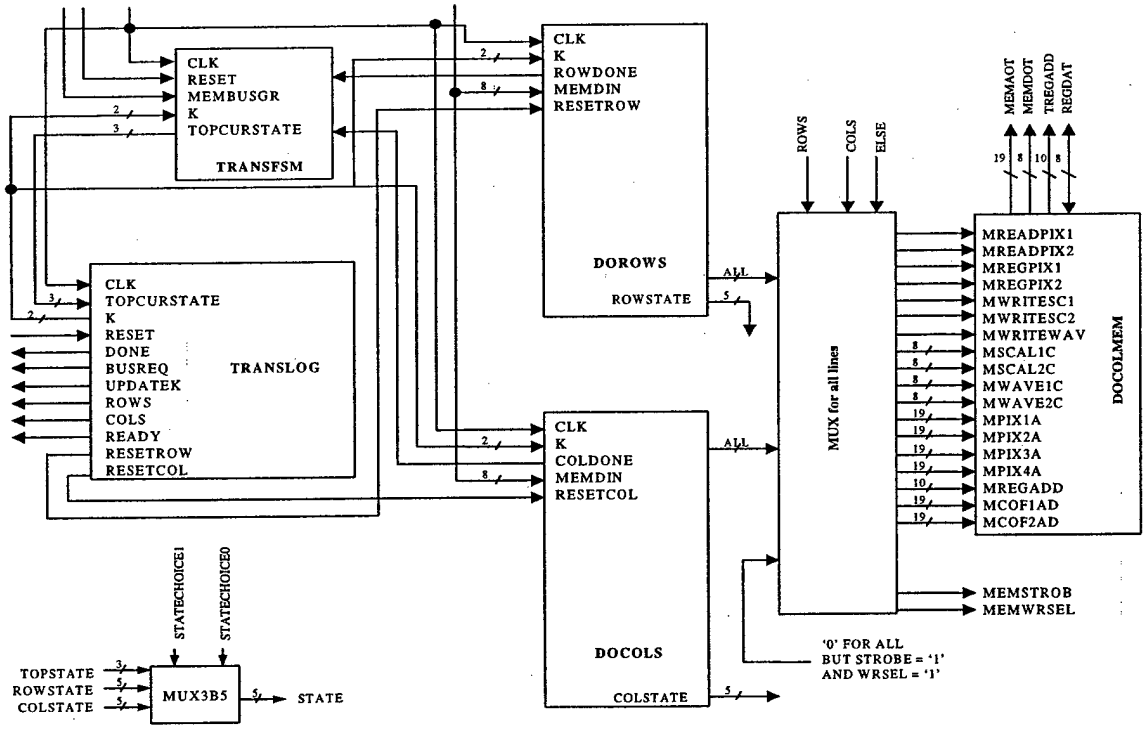


Figure 22. Top Level For Transform Logic

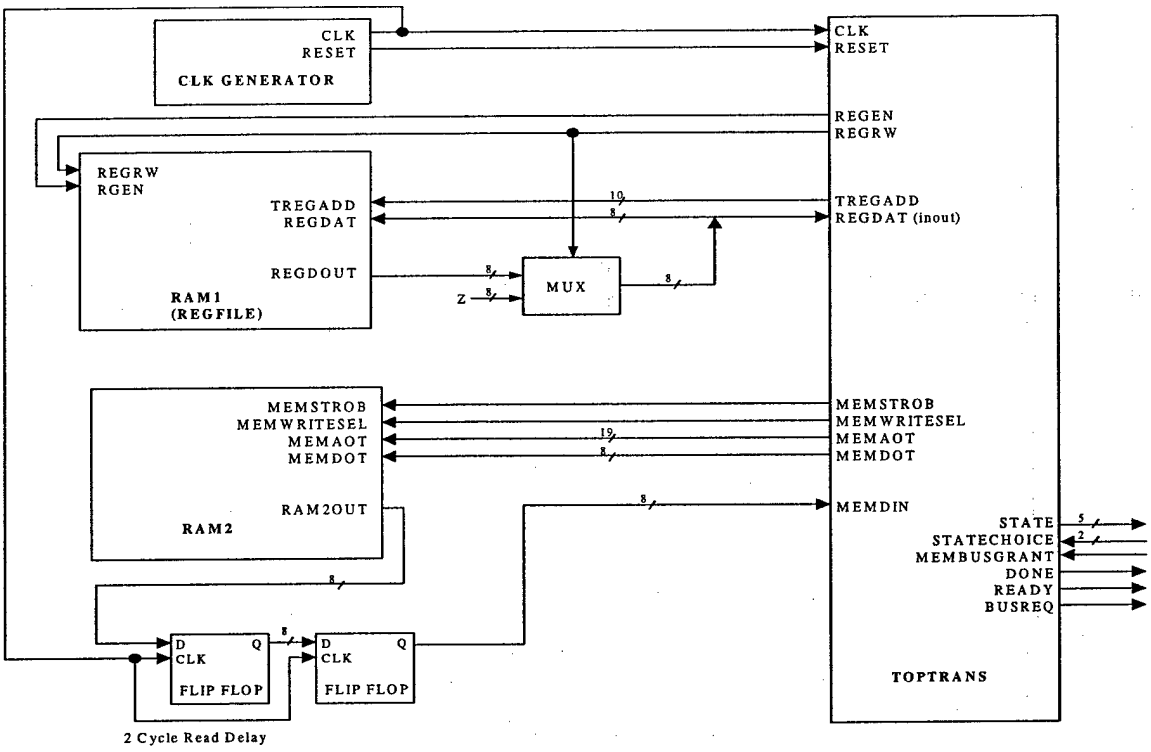


Figure 23. Test Bench For Transform Logic

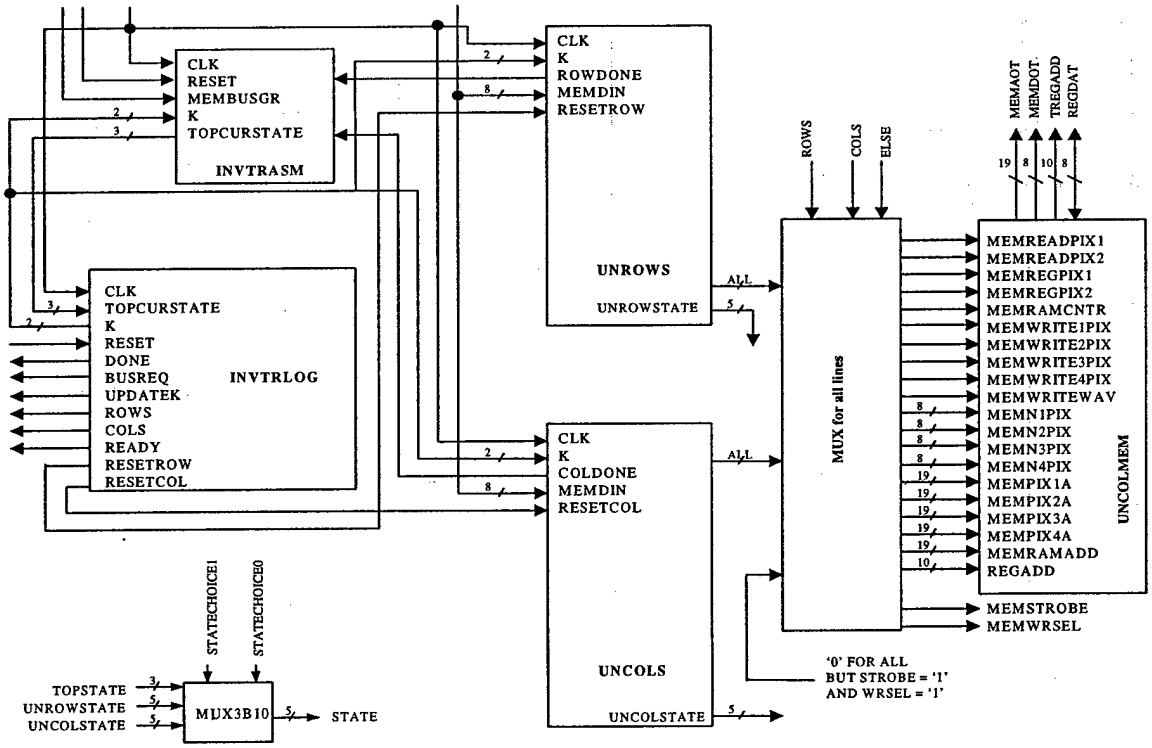


Figure 24. Top Level for Inverse Transform

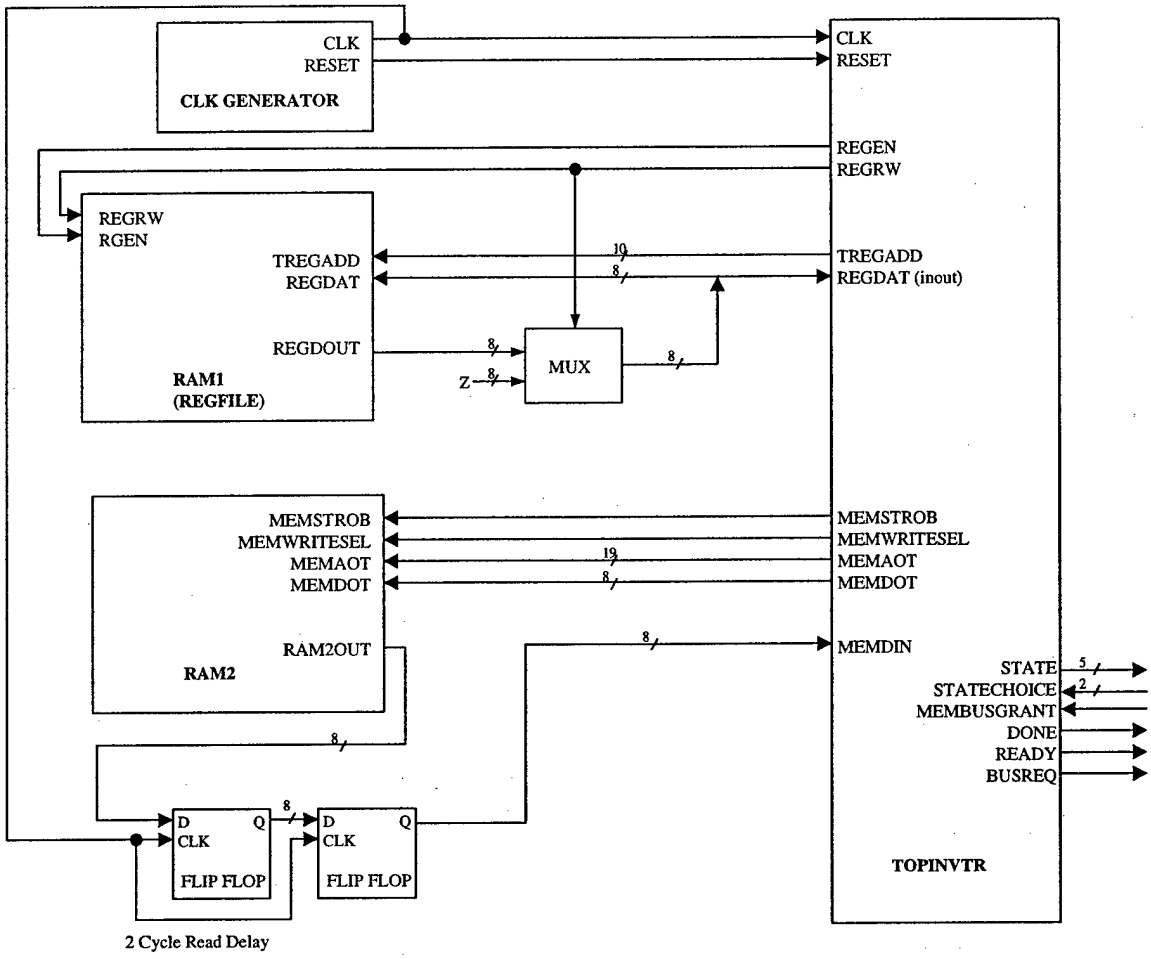


Figure 25. Test Bench for Inverse Transform Logic

4.6 State Machines

A total of six state machines control the operation of the Wavelet ASIC (Appendix A). The workload is distributed across all the states. Effort was made to have as few states as possible. A tradeoff was made in the replication of components and the number of states. In some instances a component was duplicated so two things could happen in one state. Duplication of a component saves one state in the state machine and ultimately saves thousands of clock cycles.

For example, an extra state in the portion of the state machine that read in the pixels for the transform part of the design would be executed 128 times for each row, since each iteration of the state machine reads in 4 pixels. Take 128 and multiple it by the number of rows, 512. Next, double the result to account for the columns. Finally, do the same calculations for two more iterations. Total savings for one 512 by 512 square image is 172,032 clock cycles. Assume a clock cycle of 50 ns or 20 MHz. The total time saved is 8.6 ms.

(512/4)*512*2=	131,072	--first iteration
(256/4)*256*2=	32,768	--second iteration
(128/4)*128*2=	8,192	--third iteration
	172,032	--total for all three iterations

It is obvious that adding an additional state becomes costly very fast. The tradeoff of adding an extra state is extra area consumed by replicating components to operate in parallel. Similar design choices are always being made during the ASIC design process.

A minimal comparison was made in effort to reduce the number of states needed in the state machines. Two pixels are needed in order to execute the Haar transform. Many methods of reading in the pixels could have been studied. A state machine that read in two pixels and performed the required operations was created. A state machine that read in four pixels was also created. The number of cycles necessary to read and transform four pixels using each method was 35% less when using the state machine that read in four pixels. Two different sections of the code were analyzed using the two types of state machines. The number of states listed in Table 5 and Table 6 isn't the total number of states needed to process the data. Only that portion of the two state machines that is different was counted. Table 5 shows the savings between the two types of state

machines for the row transform stage of the design. Table 6 shows the savings between the two types of state machines for the inverse row transform stage of the design. In both cases the state machine that read in 4 pixels outperformed the other state machine. A state machine that read in 4 pixels was also used in the column and row processing portions of the design.

Type of State Machine	2 Pixels Per Iteration	4 Pixels Per Iteration
Number of States to Read and Process 4 Pixels	17	11
Number of States for 1 Row	(512/4)* 17 = 2176	(512/4)*11 = 1408
Number for All Rows first pass	2176*512 = 1114112	1408*512 = 720896
Number for All Rows second pass	(256/4)* 17 * 256 = 278528	(256/4)* 11 * 256 = 180224
Number for All Rows third pass	(128/4)* 17 * 128 = 69632	(128/4)* 11 * 128 = 45056
Total Number for Three Iterations	1,462,272	946,176
% savings of 4 Pixel Read		0.352941176
Time savings assuming 20 MHZ clock		0.0258048

Table 5. Savings By Reading 4 Pixels For the Transform of Rows Stage

Type of State Machine	2 Pixels Per Iteration	4 Pixels Per Iteration
Number of States to Read and Process 4 Pixels	19	13
Number of States for 1 Row	(512/4)* 19 = 2432	(512/4)*13 = 1664
Number for All Rows first pass	2432*512 = 1245184	1664*512 = 851968
Number for All Rows second pass	(256/4)* 19 * 256 = 311296	(256/4)* 13 * 256 = 212992
Number for All Rows third pass	(128/4)* 19 * 128 = 77824	(128/4)* 13 * 128 = 53248
Total Number for Three Iterations	1,634,30	1,118,20
% savings of 4 Pixel Read		0.315789474
Time savings assuming 20 MHZ clock		0.0258048

Table 6. Savings Reading 4 Pixels for Inverse Transform of Rows Stage

Using an operating speed of 20 MHz and the total number of states needed to transform one image, the ASIC design transforms one image every 146.5 ms. See Appendix C for breakdown of states for the ASIC design. The frame rate only accounts for the processing time of an image that exists in memory. Associated operations like loading a new image into memory and transmitting the image would obviously affect the frame rate.

As stated, 6 state machines were used in the Wavelet ASIC. It might be possible to have fewer state machines by having the different parts of the chip use the same state machine. Reusing state machines is possible by not using any of the control signals produced by the state machine that are not needed. In other words if a state machine has 10 control lines but you only need 4 of them just use 4 and not connect the others. It is also possible to only use a subset of the states calculated. For example, if a state machine produces five state bits and you only need four, just utilize the lower 4 bits.

4.7 Internal Register File

Savings of RAM accesses came by creating a 256 by 8 bit internal register file. The internal registers required fewer memory accesses as intermediate values were saved to the register file rather than writing them back out to RAM. The obvious tradeoff is chip area for speed. The basic register file design, Figure 26, was taken from Weste (26). The basic cell design did not have inverter number 1 in the design. Inverter '1' was added to correct problems discussed next.

The basic operation of the cell is as follows. WriteEnableColumn and WriteEnableRow are asserted to a logic '1', turning on the n-transistors causing the value on the WriteData line to feed in to the inverter loop (inverters 2 and 4). The write enable lines are then brought low and the cell retains the level stored. The feedback inverter (inverter 4) is sized correctly so it can drive inverter 2 retaining the stored value, but be overpowered by an n-transistor when a new value is written to the cell. The single n-transistor would not operate properly in the 0.35 micron technology files no matter what

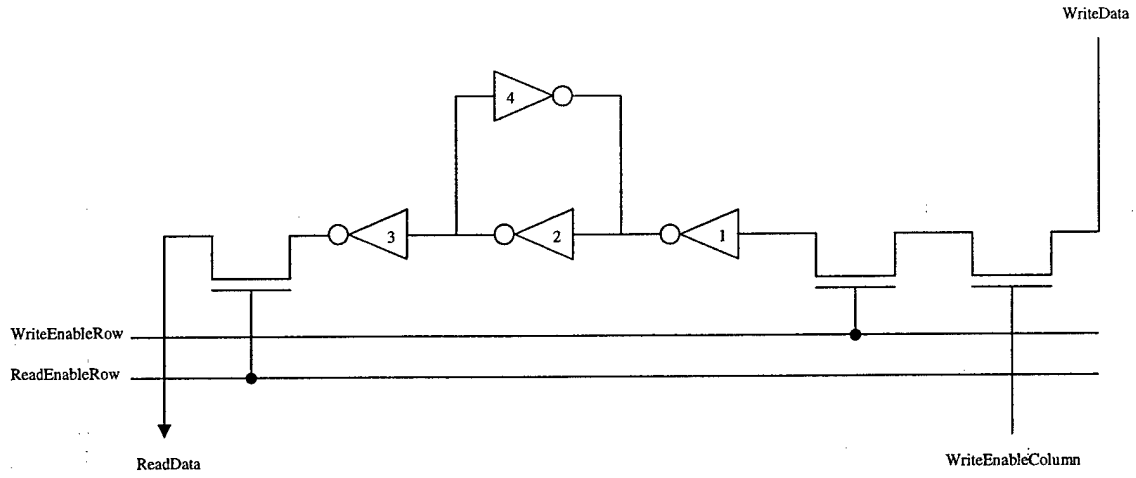


Figure 26. Single Register Cell Location

the sizing of the feedback inverter was. Replacing the n-transistors with T-gates had no positive effect. Basically, the n-transistors were not large enough to out-drive the feedback inverter; therefore, the stored value could not be changed. The problem was fixed by isolating the cell with an inverter. Inverter 1 was added to the design to isolate the inner loop (inverters 2 and 4) from the n-transistors. Adding inverter 1 eliminated the problem of sizing of the feedback inverter, i.e. the drive of the feedback inverter wasn't a factor to the n-transistors. As long as the drive of the inverter feeding the loop was bigger than the drive of the feedback inverter, the stored value could be changed. With the addition of inverter 1, it was necessary to add an additional inverter to the output since the data stored would be inverted upon storage. Therefore, the data had to be inverted when accessed by a read operation.

A single register cell location was created and tested. The next concern was how wide to make the ground and Vdd rails of the register cell. A good rule of thumb for rail thickness is the following. For every milliamp of current there should be 1 micrometer of

metal (26). In our design tools 5 lambda = 1 micrometer. Therefore, a metal of width 5 would hold 1 mA of current (26). The single register file was tested.

a. Writing a one required 150.8616 μA of current.
b. Reading a one required 25.0854 μA of current.
c. Writing a zero required 249.4874 μA of current.
d. Reading a zero required 10.7905 μA of current.

The largest current needed for a single cell was 150.8616uA. 8 bits can be written at one time so the total estimated current is 8 x 150.8616 μA = 1.2069 mA. Therefore, making the rails 10 lambda would allow for 2 mA of current, providing a significant safety margin. Actual Vdd rails in the register section were increased to 13 lambda because the minimum spacing of the p-diffusion and the polysilicon needed a minimum spacing of 13 lambda. Since the n-diffusion for the register cell wasn't as wide as the p-diffusion, the ground rails were able to be made 10 lambda wide and still satisfy the 13 lambda spacing requirement. A second register location was then created and butted up against the first location (Figure 27). The two-cell register was capable of being arrayed in MAGIC.

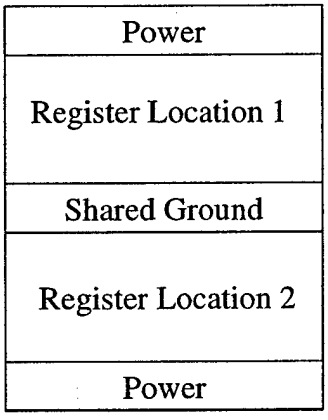


Figure 27. Two Cell Register Layout

A small register file of 2 rows by 4 columns was arrayed and tested for current usage. Results were as follows:

a. Maximum current when writing a zero to all 4 locations: 3.1014 μA
b. Maximum current when writing a one to all 4 locations: 606.3709 μA.

Using the single cell current measurements, the storing of 4 ones should have used 603.4464 μA. It was concluded to keep the rail lines at the current designed widths. The 256x8 bit array with the rail widths as specified above was constructed. The actual array dimensions are 32 bits wide by 64 bits high register file. Next, the Column and Row decode circuitry was designed.

4.8 Column and Row Decode For Register File

The column and row decode circuitry was built in a similar manner to the register file construction. The logic for one row select bit was built and then arrayed. Three decoding parts were needed. One part was needed for the ReadEnableRow control line. Another part was needed for the WriteEnableRow control line. The third part was for the WriteEnableColumn control line (See Figure 26).

There are 10 address lines that lead to the register file. The bits are numbered left to right: 9,8,7,6,5,4,3,2,1,0. Since the register file is 32 bits wide by 64 bits high, the decode logic went as follows. The first row of the register file is locations 0,1,2, and 3. The second row is 4,5,6, and 7, etc. Only the top 8 bits of the address line are needed to decode the rows. The first row is accessed when bits 9-2 are zero. The second row is accessed when bits 9-3 are zero and bit 2 is a logic '1'. The decode circuitry was designed to handle the first rows access. Each other row could utilize the same circuitry

by just selectively inverting the top 8 bits. Two designs for the decode circuitry were analyzed. One design consisted of only NAND gates and inverters (Figure 28). The other design consisted of mostly NOR gates, 1 NAND gate and an inverter (Figure 29). Each design was tested for speed and current usage.

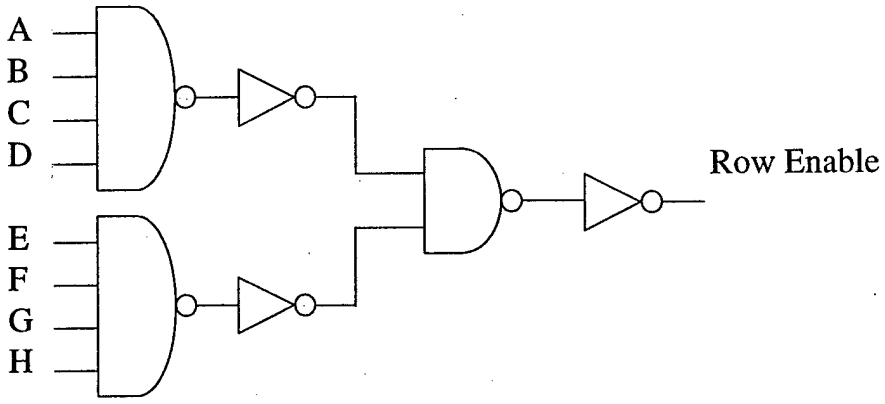


Figure 28. Row Enable Using NAND Gates

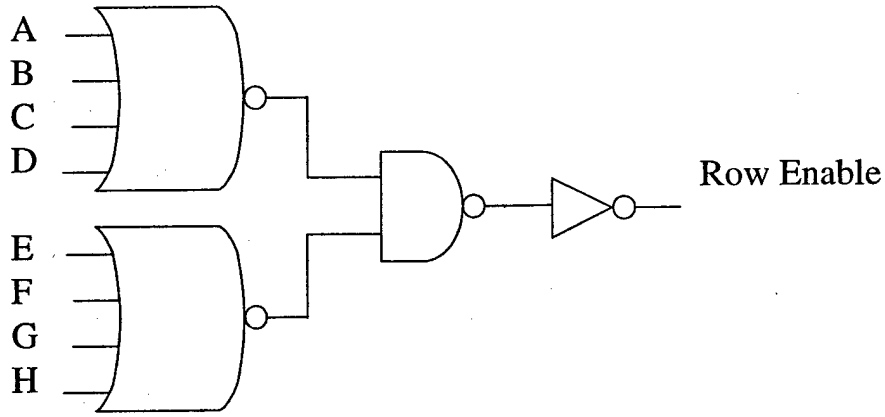


Figure 29. Row Enable Using NOR Gates

The analysis was done by testing each of the individual gates and then using those findings for the analysis of the two designs. The results of the test are shown in Table 7.

Component	Worst Timing	Max Current
Inverter	0.370 ns	132.6154 uA
4 Input NAND	0.515 ns	207.5331 uA
4 Input NOR	0.890 ns	126.6100 uA
2 Input NAND	0.400 ns	171.7821 uA

Table 7. Current and Timing of Some Simple Gates

Analysis for the worst-case path of the NAND configuration:

Timing:
1-4NAND + 2 inverters + 1-2NAND = 0.515 + 0.37 + 0.4 + 0.37 = 1.655ns
Current:
1-4NAND + 2 inverters + 1-2NAND = 2*207.5331uA + 1*171.7821uA + 3*132.6145uA =984.6918uA
Approximate Area Used: 16428 lambda^2

Analysis for the worst case path for the NOR configuration:

Timing
1-4NOR + 1 inverter + 1-2NAND = 0.89 + 0.4 + 0.37 = 1.66ns
Current
1-4NOR + 1 inverter + 1-2NAND = 2*126.6100uA + 1*171.7821uA + 1*132.6145uA = 557.6157uA
Approximate Area Used: 12136 lamda^2

For the current calculation, the current of all the gates in the configuration was added together. The area was simply the total area of all gates used for each configuration. The bottom line is the NOR configuration uses 56% less current and saves 4k lambda^2 of area, while taking approximately the same time to switch. Thus, the NOR configuration was selected for implementation in the Wavelet ASIC.

The actual decode logic uses a few more gates than the above diagrams since the R/W and register enable lines are also required (Figure 30). The additional logic is identical whether one uses the NAND gate or the NOR gate configuration, so it was left out of the current and area analysis. The decode logic is the same for the WriteEnableRow line except the R/W line logic which is driven low to signify writing.

Thus the same logic block used for the ReadEnableRow signal is modified with the addition of two gates for use as the WriteEnableRow control line.

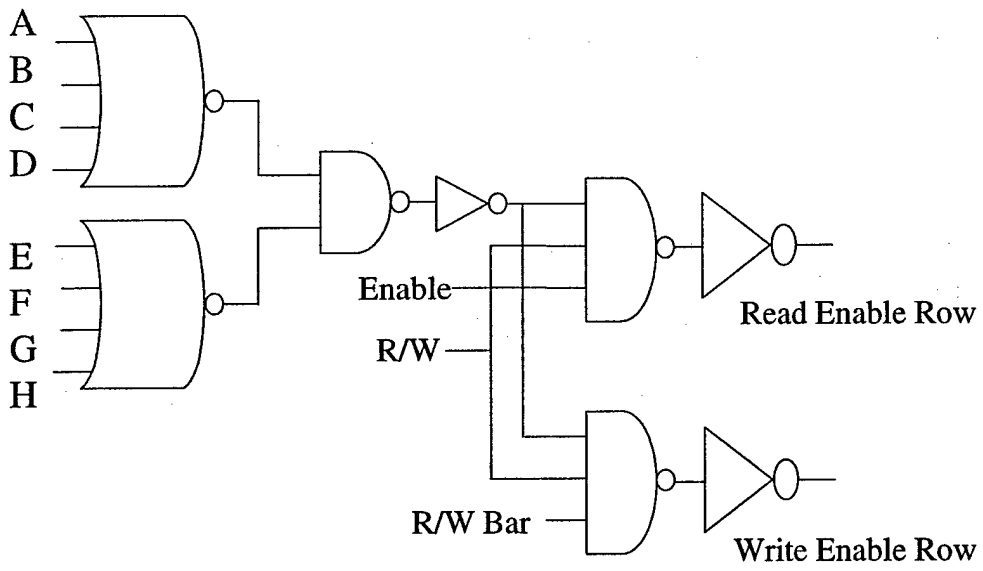


Figure 30. Row Enable Circuitry

The WriteEnableColumn control line uses NAND logic. A two input NAND gate with an inverter decodes the columns. Like the row decode the inputs are inverted as necessary. Since each column enable activates 8 columns the decode circuitry is much simpler than the row decode circuitry. Each address location of the register is 8 bits long. So writing to a register location stores 8 bits at a time. A write to the first column activates register locations 0,4,8,16,20,24, etc. Since so many locations are activated, only the low two bits are needed to decode the column. Table 8 illustrates the decode logic for all columns.

Once the row and column decode circuitry was complete it was connected to the register file, after which the entire module was tested. Timing for the entire register file is covered in Chapter 6

Bit 1	Bit 0	Enable
0	0	Column1
0	1	Column2
1	0	Column3
1	1	Column4

Table 8. Column Decode of Bits 1 and 0

4.9 Input, Output, Input/Output Pads

Micro Optical Silicon Systems (MOSIS) supplied the input and output pads. Timing of the pads was not calculated since the pads contained polysilicon 2 for high voltage transistors, which did not have any SPICE parameters available. A nominal delay of 1 ns for the input pads and 0.5 ns for the output pads were used in the timing analysis. Bi-directional pads use the same timing since they aren't enable controlled.

4.10 Top Level Input, Output, and Bi-directional Pins

This section covers the input/output signal pins. The general functionality is described in Table 9.

4.11 Data Buses

There are four data buses used in the ASIC design.

Memory Address - 19 bits
Memory Data - 8 bits
Register Address - 10 bits
Register Data - 8 bits

The address bus carries the RAM address off chip. The data bus is bi-directional moving data to and from the off chip memory. The register bus carries the internal 256-byte register address. The register data bus is bi-directional moving data to and from the

Signal	Number of Bits	Direction	Description
Address	19	Output	Address for RAM
Data	8	Bidirectional	Data lines for RAM.
Ready (3)	1	Output	Artifact from the FPGA logic. Signal to host telling hardware is ready (3).
Done	1	Output	When done goes high the transform or inverse transform is complete.
Busreq	1	Output	Artifact from the FPGA logic. When this signal is low the bus is being requested for use.
Busgrant	1	Input	Artifact from the FPGA logic. When this signal is low the bus has been granted and processing can begin.
StateChoice	2	Input	These two bits choose which state machine will be seen on the five state output pins. Either the top level, row logic, or column logic of whichever half of the design specified by the Trans/Inverse input pin.
Trans/Inverse	1	Input	This signal chooses which half of the design is executed. Either the Transform half or the Inverse Transform half.
Clock	1	Input	Clock input that drives the design logic.
Reset	1	Input	When this signal is high all state machines are in reset state.
Memstrobe	1	Output	Signal to RAM that memory is enabled.
Memwrsel	1	Output	Signal to RAM whether want to read('1') or write('0').
StateChoice	5	Output	Shows the states of whichever state machine was chosen by the Statechoice and Trans/Inverse pins.

Table 9. List of Pins and Their Functionality

internal register file. Since more than one component drives the two address buses the output drivers must be connected through tristate buffers allowing only one source to drive the bus at a given time. The data buses are also driven by more than one source and

with data flow in different directions. The RAM read/write signal line controls the direction of the bi-directional pads, which are connected to the data buses. Tristate buffers control what is placed on the internal register data bus. All output sources of both data buses are connected to the bus through tristate buffers.

4.12 Conclusions

The edited and synthesizeable behavioral-level VHDL code executed correctly when applied to a 512 by 512 image size. The structural version of the ASIC VHDL was tested and the results were equivalent to the results of the behavioral-level VHDL tests. However, many changes were made to the individual components as they progressed through the design process.

One significant difference from the synthesizeable VHDL and the physical components is the absence of any signal connected directly to ground. The CAD tools did not synthesize the code when signals were connected to ground. Grounded signals were removed from the VHDL code and hand connected in the layout.

Some sections of code, after being optimized by the design tools, contained duplicate signals. That is, two different signals were set to the same value in all states. Since the design tools did not allow for a line to have two names, it was necessary to delete one of the names. A manual trace of the VHDL code verified the optimizations. Annotation of the deletions was enough to allow for manual wiring of the signals later in the design layout.

Many of the components that were built with the AFIT tools were less than optimal. The channel routing technique of Octools provides a poor use of area. In some cases more than 50 percent of the total area is due to the channel routing. Further development of the Wavelet ASIC should involve more custom layout or channel-less routing of the individual sections. A rough estimate of removing channels from the automated layout could result in an area savings of 40 percent.

V. Testing and Timing

5.1 *Introduction*

Testing of the Wavelet ASIC started with the original FPGA VHDL code. The FPGA VHDL code was run and the results were studied. After the test run results were noted and understood, incremental design changes and testing began. Small parts of the FPGA code were transformed into synthesizeable code and retested for correctness. The synthesizeable pieces were converted into layout level components and tested individually. After confirming the correct operation of each separate component, the components were connected together and tested. Components that were custom built were also tested individually before being connected to the rest of the design. Since the Wavelet ASIC is composed of four separate engines (Row Transform, Column Transform, Row Inverse Transform, and Column Inverse Transform), each engine could be built and tested before moving on to another engine. Once an entire engine was connected and working correctly, timing for memory accesses was calculated.

5.2 Testing of VHDL Files

The ASIC consists of four main parts: row transform, column transform, column inverse transform, and row inverse transform. Each part was built and tested separately before combining and testing with its respective half. For ease of testability and a reduced execution time, a smaller image, 32 by 32 square pixels, was used until the ASIC was stable. Once the design was stable, the full 512 by 512 square image was input and

tested. The steps taken to translate the FPGA VHDL code into synthesizeable code are discussed next.

Each of the main parts originated as a large separate file, making it necessary to break up each file into small scalable components in preparation for the ASIC design. The small scalable components, once modeled and tested, were combined creating a structural VHDL version of the ASIC. The steps taken in parsing of each part was identical. First, the state machine was extracted and all decision paths were executed. Second, the logic assigning the control signals was extracted. The control signal logic was first tested alone making sure the appropriate states drove the correct control lines. The state logic was then integrated with the state machine and the two pieces were tested together. All paths of the state machine were simulated. During each state, the appropriate control lines were checked for accuracy.

Next, it was necessary to extract all of the ALU operations. After analysis it was observed that three adders and two incrementers were needed along with one subtractor and one comparator. Each component was modeled and tested for correct functionality.

Once the ALU components were individually verified they were integrated with the rest of the structural VHDL code. Each of the four main parts of the ASIC was again tested separately using the separate files integrated by VHDL port mapping (27). A list of each of the four main parts of the structural VHDL code and associated components is found in Appendix E.

The steps used to test each of the main parts are discussed next. The same steps were used to test each of the main sections. Each section begins by reading in four pixels from memory. Wavelet operations are performed on the pixels and the results are either

written back to memory or stored in the register file and later written to memory. Differences between each of the four main parts are which locations are accessed and in which order. Obviously, different wavelet operations are performed on the data in each piece. The tests used to verify correct functionality of the structural VHDL version of the ASIC are discussed next.

The first test was to see if the correct memory locations were being accessed. As stated earlier, three iterations of the wavelet transform are performed, each on a different section of the image. Testing verified that the memory addresses were accessed, both for reads and writes, in the correct order for each of the transform iterations. Once the correct RAM locations were verified, the addresses used for the internal register file were tested. Again, the correct locations for reads and writes were accessed for all three transform iterations. To aid in the testing of memory reads and writes, VHDL components for modeling the RAM and the internal register file were created and tested. VHDL components were used by the ASIC design for storing and retrieving of data, allowing for an accurate simulation of the design at a VHDL behavioral and structural level.

After verification of the correct data locations access was complete, testing of the operations performed on the pixel data was verified. Pixel operations are different depending on which part of the ASIC design is being run. Each transform section was tested for accurate manipulation of the data. For the row transform, row inverse transform, and column inverse transform, the operations were simple. As explained earlier, only additions, subtractions, and shifts are used. Different pixel values were input

to the ASIC design. Since pixel values can range from –127 to +127, correct calculations were verified using both positive and negative pixel values.

The column transform was much more difficult to test because it contained the steps for quantizing and thresholding of the data. Since the quantize and threshold rules are different depending upon which iteration and which quadrant of the image you are currently processing, different tests were run to check each of the situations. For example, in the lower left quadrant of the first iteration, the output pixel is in the range –8 ,..., +8. Some of the quadrants allow only the values –8,0, and +8. To illustrate, one test produced the ending values of: -20, -9, -3, 5, 17. These numbers were the result before the rules of threshold and quantize were applied. Once the rules were applied the five values were: -8, -8, 0, 0, 8. The five values are the correct result for the quadrant being tested. Appendix F contains the input data used to test each quadrant of each iteration. The 'k' term references the loop variable, which is used to specify which section of the image is being processed. The pixel values from start to finish are shown and in the appropriate memory locations. The steps displayed in Appendix F are for the transform half of the design. The results in parenthesis are the final results output to the RAM. As shown by the test cases, the range of values for each quadrant was tested. The test data, Appendix F, was used to test each of the four pieces and again used to test the two halves, Transform and Inverse Transform, of the design. Since the two halves are independent, no further combining was appropriate.

When structural VHDL code was complete and verified it was time to start building and testing the components at the physical layout level.

5.3 Testing Components

This section explains the different tests performed on each of the physical components. Reasons for optimizing certain components are explained. All of the components were built and tested individually. Timing was more critical for some components than others.

Testing the components at a layout level involved two types of tests. The first test checks the component for correct functionality. There is a one-to-one mapping from the inputs to the outputs of the structural VHDL to the physical layout level of each component. The mapping allowed reuse of the test vectors, used to test the VHDL files, to verify functional accuracy. The second type of test checked each component for timing of the critical path. The critical path is the longest delay through the component and, therefore, controls the worst-case timing delay of the component.

The critical path of the ASIC research effort occurs in the states that access RAM. A RAM access time of 35 ns for both reads and writes was used for the design of the ASIC. The implications of the 35 ns access time are simple. As long as other operations occurring in non-RAM access states are faster than 35 ns, the RAM access states would drive the speed of the Wavelet ASIC design.

The slowest component in the ASIC design is the 19-bit adder. A carry-select methodology was used to create the 19-bit adder and produced a simulated execution time of 5.62 ns, a breakpoint value for all other components. Since operations were evenly distributed among the different states of the design and none were done in series, as long as they were faster than 5.62 ns, no other performance optimizations were

necessary. If multiple additions, for example, are required during a single state, they are performed in parallel by replicating the adders.

As shown in Table 10, the execution times of some of the smaller components are longer than larger components. As stated earlier, once the speed of a component was analyzed and found to be faster than the 19-bit adder, it was no longer optimized for speed. The only way further optimizations for speed would enhance the design is if the speed up came from a reduction in area. However, faster execution time comes with the tradeoff of having larger die area. A complete listing of all the components and their execution times is contained in Appendix D. Unless otherwise stated, the timing, referred to in this chapter and Appendix D, was the result of running HSPICE.

Component	Number of Bits	Critical Execution Time (ns)
Adder	19	5.62
Adder	9	4.11
Adder	8	2.80
Subtractor	9	2.95
Incrementor	19	2.12
Incrementor	10	3.16
comparator	10	1.85

Table 10. List of ALU Components

5.4 Register File

Most of the components were built using design tools. However, two components were manually designed and laid out: the register file and the address decode logic for the register file. First, a one-bit register was designed, laid out, and tested for reading and writing. Next, two one-bit locations were integrated and again tested for reading and writing. The two-bit location was replicated into a 4 by 8 bit array, which was tested for

reading and writing. Then the entire 2048 locations were designed and tested. The layout of the 2048-bit register array is shown in Figure 31. The bits for each location are numbered left to right: 7,6,5,4,3,2,1,0. Various locations were tested and timed for both reads and writes. Table 11 shows the access times for the locations tested. The maximum time of 2.57 ns is trivial compared to the access time of the off-chip RAM. The minimal retrieval time allowed both the reading of the internal register file and the writing of the value to RAM to occur in the same state. Combining the register read with the RAM write saved time by eliminating one state. Savings from using fewer states was explained in Section 4.6.

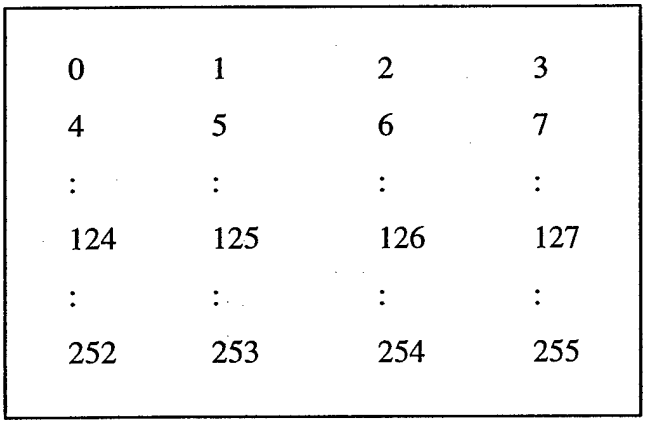


Figure 31. Register Locations

Next, the decode logic was built, tested, and timed. The timing for the decode circuitry is shown in Table 12. Signal line references (A, B, C, D, E, F, G, H) refer back to Figure 30. The input signals, labeled A-H, are switched to test the longest delay of the decode circuitry, both for a low-to-high transition on the output and a high-to-low

Timing of 256x8 bit register file			
Location	Bit	Operation	Time (ns)
0	7	Write '0'	1.22
0	7	Read '0'	2.52
0	7	Write '1'	1.52
0	7	Read '1'	2.02
3	0	Write '0'	1.22
3	0	Read '0'	2.16
3	0	Write '1'	1.52
3	0	Read '1'	1.48
124	7	Write '0'	1.22
124	7	Read '0'	2.15
124	7	Write '1'	1.51
124	7	Read '1'	1.47
127	0	Write '0'	1.22
127	0	Read '0'	2.57
127	0	Write '1'	1.51
127	0	Read '1'	2.00
252	7	Write '0'	1.22
252	7	Read '0'	2.15
252	7	Write '1'	1.51
252	7	Read '1'	1.47
255	0	Write '0'	1.22
255	0	Read '0'	2.57
255	0	Write '1'	1.51
255	0	Read '1'	2.00
Max time			2.57

Table 11. Access Times for Register File

Timing of Read Decode Circuitry		
	Output	
	0 to 1	1 to 0
All H to L	1.71 ns	
All L to H		0.457 ns
ABCGFE are L		
DH go L	1.53 ns	
All L to H		0.457 ns
DCBHGF are L		
AE go H	1.34 ns	
Max time	1.71 ns	0.457 ns

Timing of Write Decode Circuitry		
	Output	
	0 to 1	1 to 0
All H to L	1.70 ns	
All L to H		0.453 ns
ABCGFE are L		
DH go L	1.52 ns	
All L to H		0.454 ns
DCBHGF are L		
AE go H	1.33 ns	
Max time	1.70 ns	0.454 ns

Table 12. Timing for Read/Write Decode Logic

transition on the output. As shown, a maximum switching time of 1.71 ns was calculated.

5.5 Testing of the Inverse Transform by Rows Section

Due to time constraints, only one of the four sections, the inverse row transform, was completely integrated and tested as a unit. Expected timing for the remaining three sections should be similar, since the inverse row transform section uses all but one of the ALU components used in the other sections.

The inverse row transform section was integrated keeping in mind where the other three sections would be located in the physical layout. See Figure 32 for how the inverse row transform section is physically integrated and where the other three sections are placed relative to the inverse row transform section. Once integrated, the inverse row transform section was tested using IRSIM.

First, data values were read in from RAM. It was checked that the RAM locations being accessed for the reads were correct and data was accessed in the right order. Results of the inverse transform performed on the first half of the image are written to the internal register file. It was checked that the correct register file locations were being written to and in the correct order. Next, eight different values were input to the inverse row transform section. It was checked that the correct results from the inverse transform were obtained. Results were then checked for accuracy and placed on the data bus for storing into the register file.

Once simulations were run successfully with IRSIM, HSPICE was used to get more accurate timing information. The timing data obtained by using HSPICE was then used to facilitate the design of the memory strobe for RAM and the register enable for the internal register file. The timing diagrams for the inverse row transform section are described next.

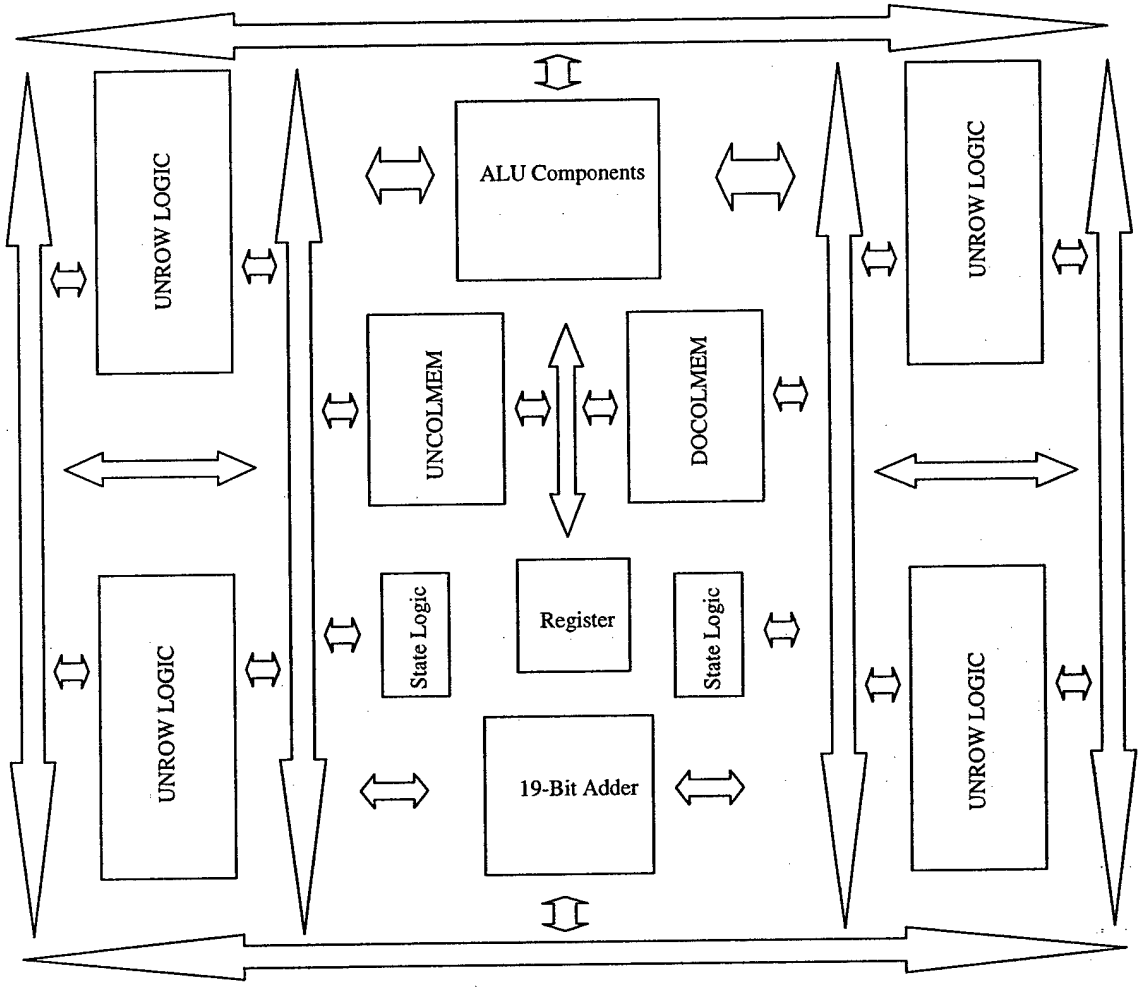


Figure 32. Layout of ASIC Design

5.6 Read/Write Logic

Timing of the control lines used to access RAM is critical and has to abide by certain rules. As mentioned, a RAM access time of 35 ns is used for the ASIC design. Once all necessary signals are stable the RAM strobe is asserted and remains asserted for 35 ns, the time needed for the RAM to perform either a read or a write operation.

The inverse row transform section contains four different memory read/write scenarios. A design margin is incorporated into the timing of the read and write accesses for both the RAM and the register file. The design margin is in case the other three sections, not yet connected, require extra time. Other factors, such as variances in fabrication, pad frame delays and routing to and from the pad frame, add delay time. When the complete ASIC design is integrated the timing delays will have to be refined.

The first scenario is a read from RAM. Four reads occur sequentially. Figure 33 shows the timing diagram for two sequential reads. Data from the read isn't latched until two cycles after the request. External circuitry is required to account for the two-cycle delay and is consistent with the Wildforce board implementation. The memory address is valid 12 ns after the clock edge. The memory strobe is asserted 2.5 ns later and stays asserted for 35.4 ns. Data is latched in on the rising edge of a clock. The 7.1 ns specified in Figure 33 is the time it takes to latch the data through the logic. The actual time the data needs to be valid is approximately 2 ns before the clock edge until 1 ns after the clock edge providing a 4.1 ns design margin in the ASIC implementation.

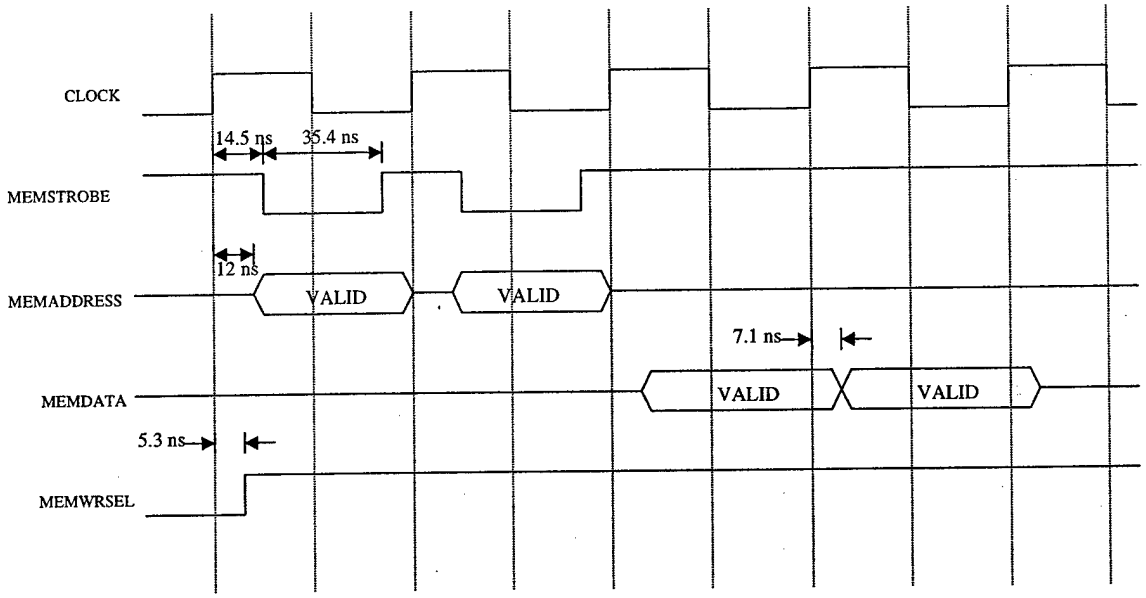


Figure 33. Read from RAM Timing Diagram

The second scenario is a write to the register file. Four writes occur sequentially in the operation of the Wavelet ASIC. Figure 34 shows the timing diagram for two sequential register writes. The data is valid 10.9 ns after the clock edge, while the register address is asserted 11.1 ns after a clock edge. Then the register enable is asserted 2.4 ns later and stays asserted for 37.0 ns. The actual time it takes for the register file to store the data is approximately 4.28 ns. The logic that asserts the register enable also asserts the memory strobe in a different read/write scenario and, therefore needs to hold the assertion for at least 35 ns. The write to register file state isn't on the critical path state of the ASIC design so the extra time used by the write to register file state doesn't slow down the ASIC operation.

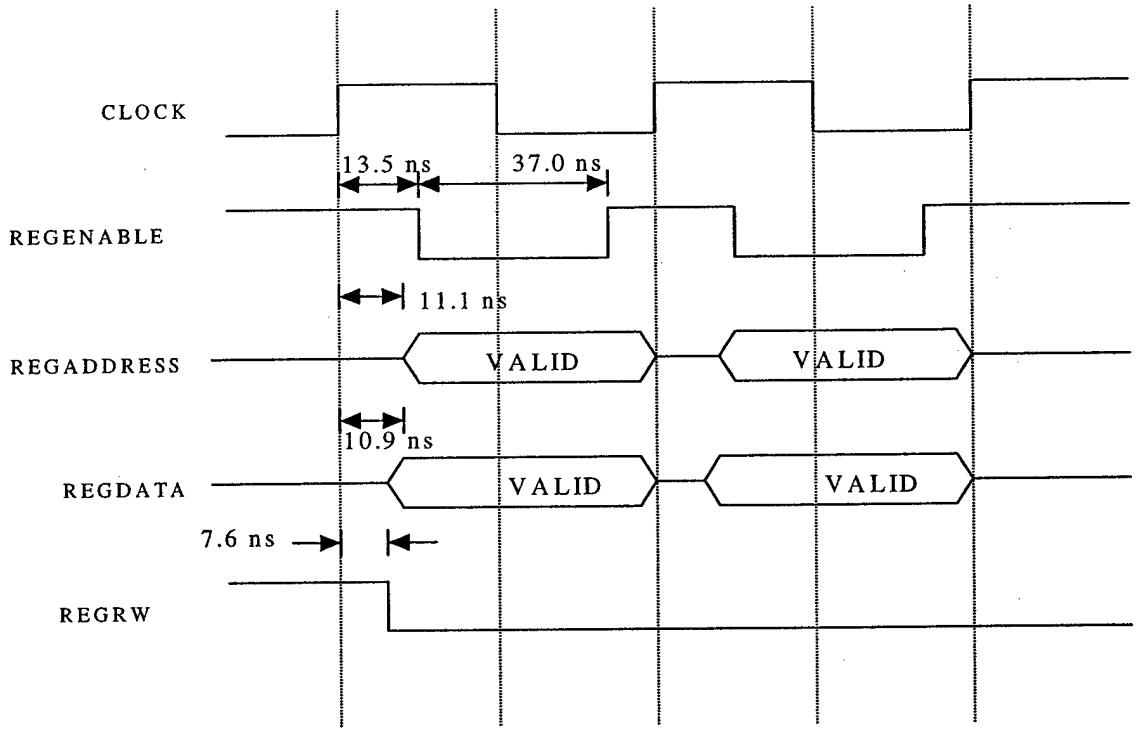


Figure 34. Write to Register File Timing Diagram

The third scenario is a data write to the RAM. Four writes occur sequentially during normal operation. Figure 35 shows the timing diagram for two sequential writes. The data is valid 10.9 ns after the clock edge followed by the memory address, which is valid 12.0 ns after a clock edge. Then the memory strobe is asserted 1.5 ns later and stays asserted for 37.0 ns.

The last scenario is a data read from the register file combined with a write to the RAM. Figure 35 shows one cycle of the read/write scenario. The register address is asserted 11.1 ns after the clock edge. Then the register enable is asserted 1.93 ns later and stays asserted for 35.73 ns. The register takes approximately 7 ns to output data valid. The memory address is asserted and valid 12.0 ns after the clock edge. The memory strobe is then asserted 20.35 ns after the clock edge and is valid for 35.74 ns.

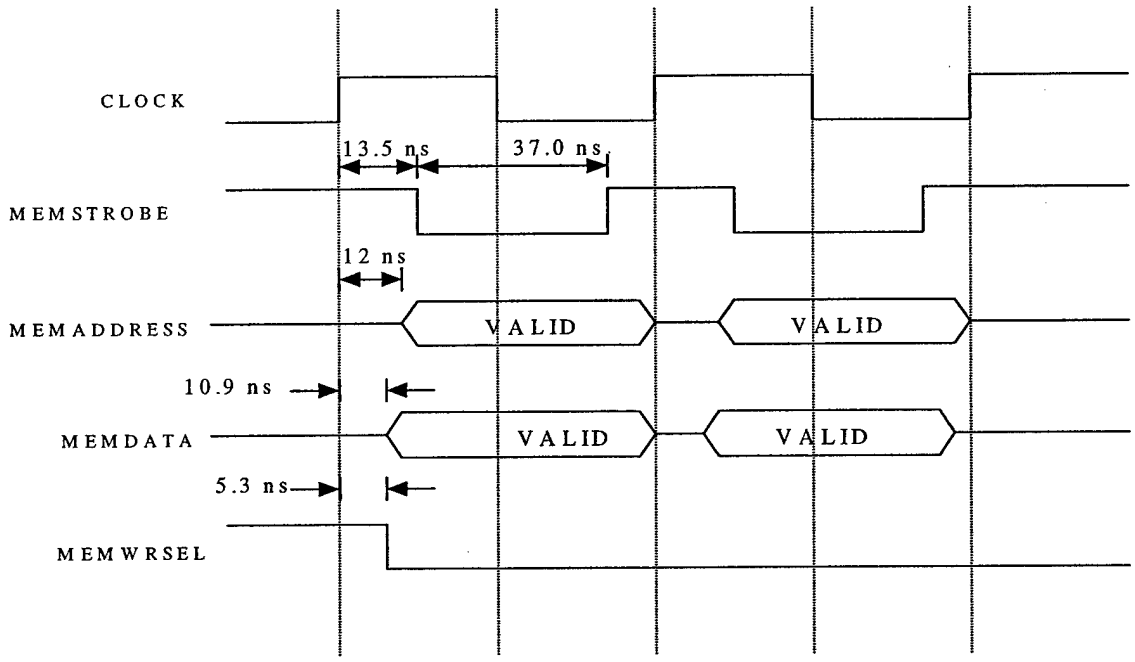


Figure 35. Write to RAM Timing Diagram

For the inverse row transform section, the most complicated and longest state for reading and writing occurs when the internal register file is accessed for a data read and the data is then written to RAM. Control lines necessary for the read and write states are stable long before the memory addresses are stable and therefore are have no impact on the timing calculations. Worst-case timing of data accesses was used from tests of the register file to simulate the operation of the register file. The register address is first decoded in 1.71 ns. Once the address is decoded the location in the register file is accessed and the results are available on the data lines after 2.57 ns. The data passes through two multiplexers, 0.42 ns and 0.39 ns, and then through a tristate buffer, 0.396 ns, before being asserted on the RAM data lines. The total time from register address stable to data stable on RAM data lines is 5.486 ns plus some estimated time for

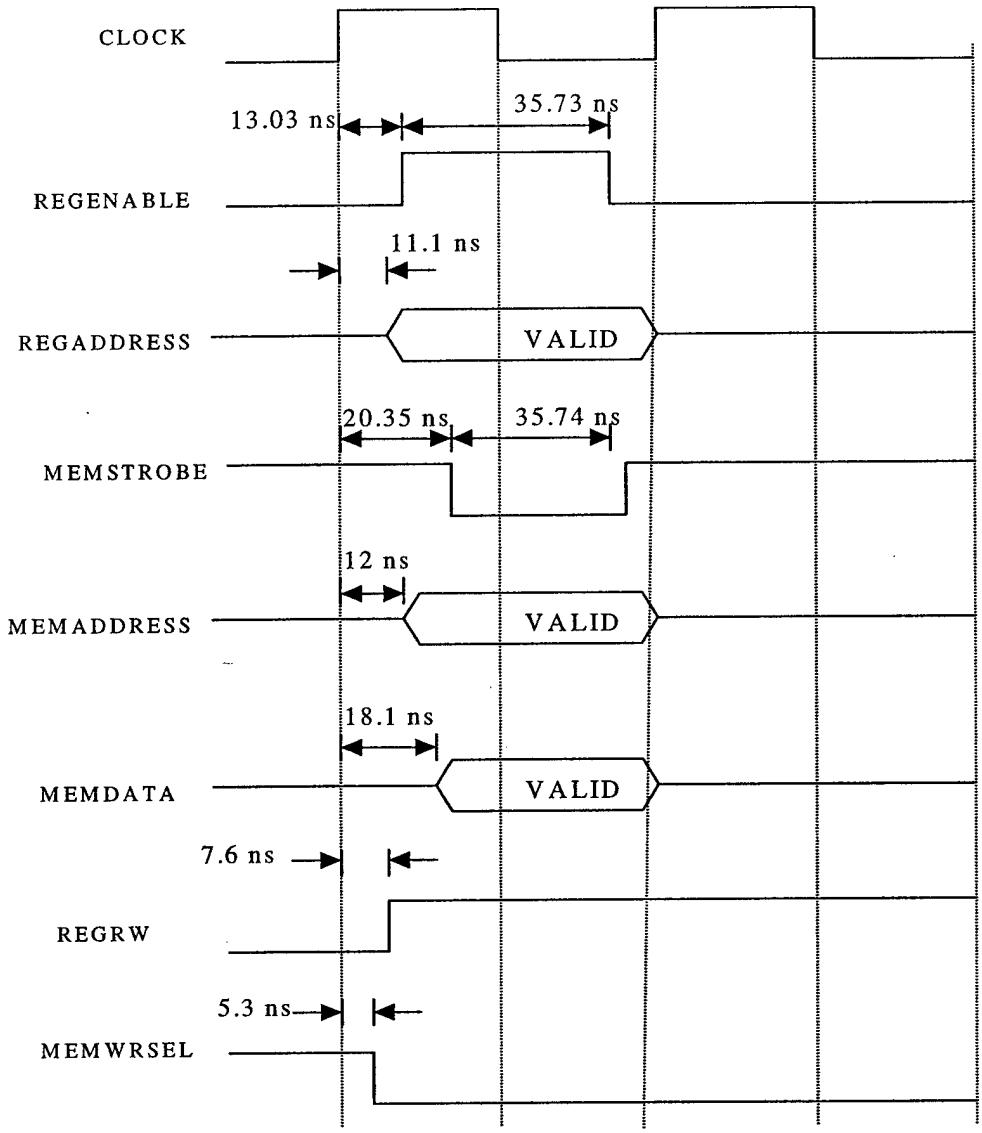


Figure 36. Read from Register and Write to RAM Timing Diagram

routing. Incorporating a design margin of 1.5 ns, a total delay time of 7 ns is used to simulate the rest of the timing delays.

The state control lines are used to generate the internal register file enable and the RAM strobe. The state control line is asserted after the clock edge and is used to indicate to the circuitry which state is active. To create the enable and strobe signals, the control

line is first sent through an inverter yielding the inverse of the control signal (control bar). Both signals are delayed to create a pulse of approximately 35 ns in width (Figure 37). The width has to be greater than 35ns for the RAM strobe and greater than 4.28 ns for the register enable. The control circuitry schematic used to generate the pulses is shown in Figure 38. The control circuitry asserts the enable and strobe signals when both control

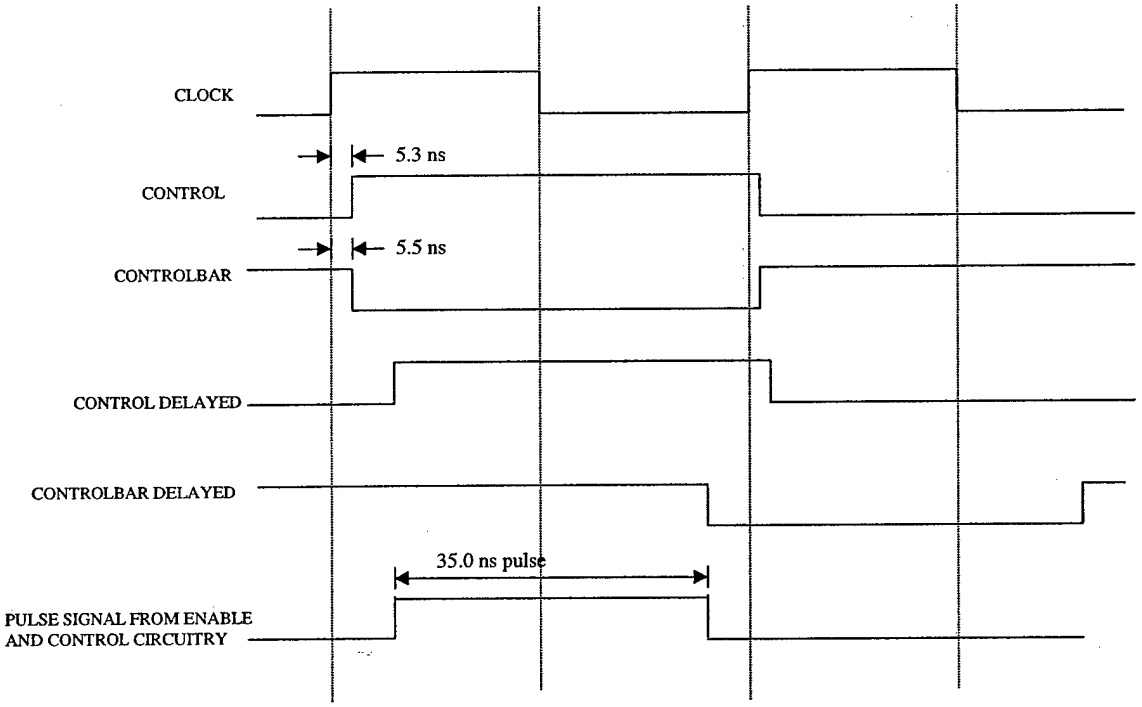


Figure 37. Timing Diagram Showing Pulse Created From Control Signals

signals, state control and state control bar, are logic '1'. The delay for each of the control lines and control bar lines is not shown in Figure 38. By delaying the control signals, they can be used to assert the enable and strobe lines. By delaying the control bar signals even longer, they can be used to clear the enable and strobe lines. The delay blocks in Figure 38 make the strobe signal line assert and deassert approximately 7 ns after the

register enable signal asserts and clears. The 7 ns delay is necessary for the register data to become valid.

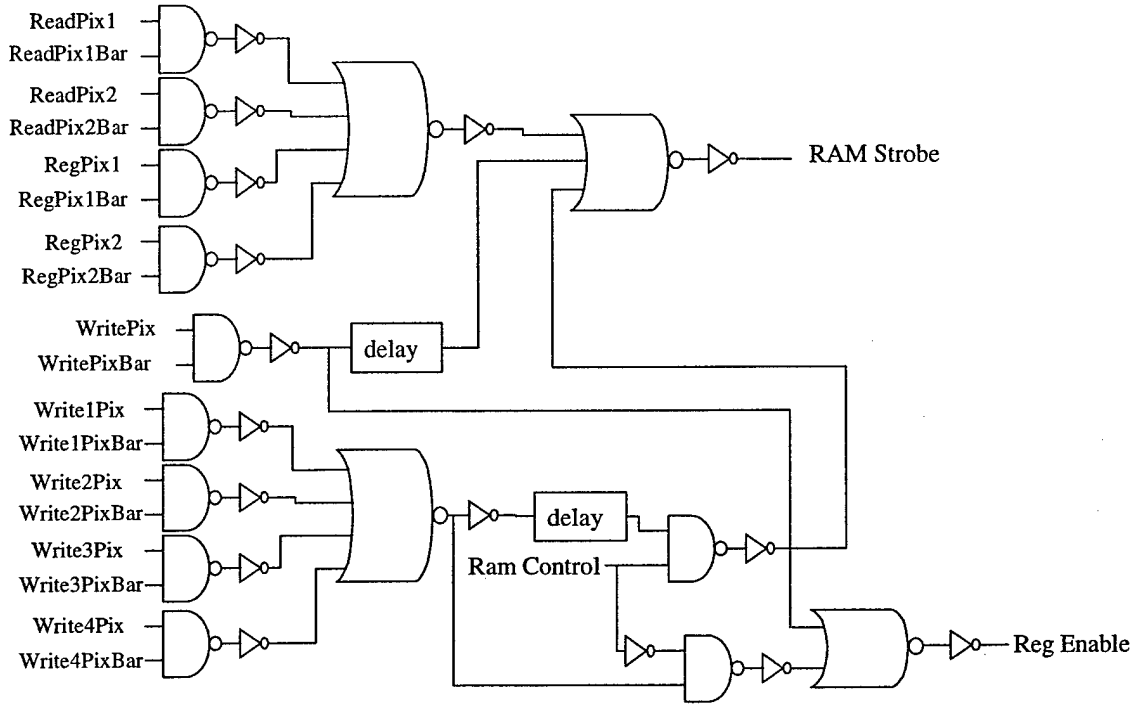


Figure 38. Enable and Strobe Control Circuitry

The critical path, read/write state, limits the speed of the ASIC design. The memory strobe is cleared 56.09 ns after the clock edge. Adding approximately 2 ns from the clearing of the memory strobe signal makes the worst-case state timing equal to 58.09 ns. The 2 ns before the next clock edge is a design margin for the ASIC. Further design optimizations could reduce the buffer time. The 58.09 ns worst-case state time translates into a maximum operating speed of 17.21 MHz.

5.7 Conclusion

The inverse row transform was the only section that was fully integrated and simulated for timing data. A maximum operating speed of approximately 17.21 MHz is expected. The other three main sections could affect the speed once they are integrated into the layout. Other factors that affect execution speed are input/output pad delays and routing to and from the pads. Design margins incorporated into the timing of the ASIC should account for extra delays. Using an operating speed of 17.21 MHz and the total number of states needed to transform one image, the Wavelet ASIC design outputs one transformed image every 170.2 ms or just under six images per second. See Appendix C for the complete breakdown of states used for the Wavelet ASIC design. The frame rate only accounts for the processing time of an image that exists in memory. Associated operations like loading a new image into memory and transmitting the image could affect the frame rate if a single port RAM is used.

As shown, the 35 ns RAM access limits the overall speed of ASIC design. A faster RAM chip could be used but the speed of the ASIC design cannot be increased without altering the layout, since a 35 ns pulse width is hard-wired into the ASIC design. Since a RAM read incorporating a two-cycle delay is used as implemented on the Wildforce board and the FPGA logic, a delay for the read data would need to be buffered by off chip logic when using a standard RAM chip with a one cycle read access time. Power needed to operate the Wavelet ASIC is discussed next.

After initialization of the inverse row transform section of the ASIC, the maximum power used is 220 mW. The maximum power is only for the inverse row

transform section. Since the more complicated piece of the design is the column transform section, it is probable that the maximum power used will increase. Measuring the individual pieces of the inverse row transform section, it was found that the maximum power used is 118 mW. Measuring the individual pieces of the column transform section, it was found that the maximum power used is 212 mW. Using these power measurements, the estimate of the total maximum power used by the transform row section is 395 mW. Since the register file was not used in the tests for the inverse row transform section, the measured power of the register file, 32.1 mW, needs to be added to the power calculation. Adding the power used by the register file to the power used by the transform row section yields a maximum estimated power of 427.1 mW.

VI. Conclusions and Recommendations

6.1 Conclusions

The steps necessary to take a design from an FPGA implementation to an ASIC implementation design were discussed. The objective of this research was to first translate the FPGA VHDL behavioral code to synthesizeable VHDL behavioral code. Performing the translation uncovered many unnecessary RAM accesses in the FPGA design. Combining the quantize and threshold steps with the column transform step saved 47 percent of the RAM accesses.

Another objective was to minimize the power needed by the design. Extra control circuitry was added to decrease the amount of switching by the transistors. The extra control circuitry makes the ALU components active only when they are being used. The decreased amount of switching has a positive impact on the overall power used by the design. The estimated maximum power used by the ASIC is 427.1 mW. The power rating is for the first iteration of the Wavelet ASIC and can be used as a benchmark for future design work. Comparing the estimated power of the ASIC to that of the FPGA, 11.6 W, a 96 % reduction in power usage is achieved.

Minimizing the area of the Wavelet ASIC was another goal. The total die area used by the ASIC is 22.138 mm^2. The core of the ASIC is 16.146 mm^2 without the pad frame. The size of the ASIC is less than 25 % of the 4 FPGA chips utilized in the original FPGA design.

The final goal was speed. The FPGA design runs at 20 MHz. The 20 MHz operation translates into a 196.609 ms per frame transform rate or 5 frames per second. Using an operating speed of 17.21 MHz and the total number of states needed to transform one image, the ASIC design transforms one image every 170.2 ms or 5.8 frames per second, a 13 % improvement over the FPGA design. The image rate calculation is an estimation based on the inverse row transform timing. Using a faster RAM chip, 10 ns access time, would significantly improve on the frame rate by approximately 43%, since the frame transform rate would decrease to 96.95 ms. The decreased frame rate equates to 10.3 frames per second.

6.2 Recommendations

There are several research possibilities regarding the ASIC design. One obvious extension is to connect the remaining components and present the design for fabrication. Improvements on area, speed, and power are still possible.

A significant improvement would be in the area used by the ASIC design. The use of automatic layout tools added a significant amount of area to each of the components, since a channel routing algorithm was used. The algorithm also limits its routing to only two layers of metal. Using a non-channel routing technique, as well as using metal 3 and metal 4 could realize a savings of 40 percent. This estimate is based on a visual analysis of the individual blocks. An additional way to decrease area is by adding data buses to handle all traffic to the ALU components. Currently each of the four components has separate signal lines that trace to the ALU components. Using

buses to combine many of the signals would reduce the number of multiplexers needed for each of the ALU components. The area used by signal lines running to the ALU components would be reduced by approximately a factor of two. By removing approximately three-fourths of the multiplexers and half of the signal lines, 20,630,143 lambda^2 and 83,940,000 lambda^2 would be saved. For the Wavelet ASIC, 25 lambda^2 equates to 1 μm^2 in die area. The total savings equates to 0.825 mm^2 and 3.357 mm^2, which totals 4.182 mm^2 in die area. The cost of using buses would be the extra area needed to add a tristate buffer on all signals connected to the buses. Approximately 1200 signal lines require 1200 tristate buffers at 4736 lambda^2 for each buffer, totaling 5,683,200 lambda^2. Subtracting the total tristate buffer area from total area saved from multiplexers and signal lines gives an estimated savings of 3.955 mm^2 in die area.

Increasing the operating speed of the ASIC is also possible. The first way to increase speed is by reducing the area. If the area is reduced so is the propagation time of the signals. Another way to increase the operating speed is to choose a faster RAM. Choosing a faster RAM would reduce the critical path of the design. For example if a RAM with a 10 ns read/write access time was used, 25 ns could be shaved off the longest state causing the design speed to increase from 17.21 MHz to 30.22 MHz. The speed increase would improve the frame rate by 73 ns, yielding 10.3 frames per second.

Another way to increase the output is by redesigning the state machines. By simply removing the one state used to calculate the FPGA memory offset and using two states instead of three to empty the internal register file, 7.1 frames per second would be

achieved. Using a 10 ns RAM in conjunction with the reduced number of states yields an output of 12.5 frames per second.

Finally, power consumption would decrease as a result of the reduced number of signal lines and the reduced number of multiplexers. The design would be smaller and faster but extra control circuitry needed for the added tristate buffers would require extra power. A study would have to be done to see whether or not the above changes would have a positive or negative affect on the power consumption. Estimating the power saved by subtracting the power from the added tristate buffers from the power of the removed multiplexers is possible. However, the estimate isn't very accurate since neither the multiplexers nor the tristate buffers are switching at the same time.

Bibliography

1. Robert S. Hauser, "Design and Implementation of a VLSI Prime Factor Algorithm Processor" Master's Thesis, AFIT/GCE/ENG/87D-5, School of Engineering, Air Force Institute of Technology Air University, Wright-Patterson AFB OH, December 1987.

2. M. Antonini, T. Gaidon, P. Mathieu, and M. Barlaud, "Wavelet Transform and Image Coding", *Advances in Image Communication Wavelets in Image Communication*, Volume 5, M. Barlaud, editor, Elsevier Science B.B., 1994.

3. WildForce Reference Manual, Revision 3.0. Annapolis Micro Systems, Inc. 1997.

4. DAGSI Project SN-AFIT-99-04, "Global Information Compression Methodology & Implementation for Enhanced Command, Control, Computing, Communication, Intelligence, Surveillance, Reliability (C4ISR) System Integration".

5. SYNOPSYS, Version 2000.05, Synopsys, Inc. Mountain View, CA, 2000.

6. OCTOOLS, Distribution 3.0, University of California, Berkeley, 1989.

7. MAGIC, Version 6.5.1, EECS/ERL industrial Liaison Program, University of California at Berkeley, Berkeley, CA.

8. IRSIM, Version 9.5, Stanford University, CA. 1988-1990.

9. HSPICE User's Manual, Volume 1, Getting Started, Meta-Software, Inc. 1992.

10. HSPICE User's Manual, Volume 2, Elements and Models, Meta-Software, Inc. 1992.

11. HSPICE User's Manual, Volume 3, Analysis and Methods, Meta-Software, Inc. 1992.

12. C. S. Burrur, Notes on the FFT, Department of Electrical and Computer Engineering, Rice University, Houston, TX, September 29, 1997.

13. Mladen Victor Wickerhauser, *Adapted Wavelet Analysis form Theory to Software,* Wellesley, MA: A. K. Peters, Ltd., 1994.

14. Don Morgan, "Haar Wavelets?" Online. Internet. 5 Feb 2001. Available: http://www.embedded.com/97/sp9712.htm.

15. L. Prasad and S. S. Iyengar, *Wavelet Analysis With Applications To Image Processing*, Boca Raton, New York: CRC Press, 1997.

16. Larry L. Schumaker and Glenn Webb, *Recent Advances in Wavelet Analysis.* San Diego, CA: Academic Press, Inc, 1994.

17. I. Daubechies, *Ten Lectures on Wavelets.* Society for Industrial and Applied Mathematics CBMS-NSF Regional Conf.\ Series in Appl.\ Math., Vol.\ 61 Philadelphia, PA: 1992.

18. M. Schwarzenberg, M. Traber, M. Scholles, and R. Schuffny. "A VLSI chip for wavelet image compression," ISCAS'99, Proceedings of the 1999 IEEE International Symposium on Circuits and Systems VLSI, pp. 271-4 vol. 4.

19. Chu Yu and Sao-Jie Chen. "Design of an efficient VLSI Architecture for 2-D Discrete Wavelet Transforms," IEEE Transactions of Consumer Electronics, vol. 45, no. 1 p. 135-40.

20. J. Singh, A. Antonio, and D. J. Shpak. "A Distributed Memory and Control Architecture for 2D Discrete Wavelet Transform," ISCAS'99, Proceedings of the 1999 IEEE International Symposium on Circuits and Systems VLSI, pp. 587-90 vol. 3.

21. Guoqing Zhang, M. Talley, W. Badawy, M. Weeks, and M. Bayoumi. "A Low Power Prototype for a 3d discrete Wavelet Transform Processor," ISCAS'99, Proceedings of the 1999 IEEE International Symposium on Circuits and Systems VLSI, pp. 145-8 vol. 1.

22. G. Lafruit, F. Catthoor, J. P. H. Cornelis, and H. J. De Man. "An Efficient VLSI Architecture for 2-D Wavelet Image Coding with Novel Image Scan," IEEE Transactions on Very Large Scale Integration Systems, vol. 7, no. 1, pp. 56-68

23. Bruce W. Hunt, "A Single Chip Low Power Implementation of an Asynchronous FFT Algorithm for Space Applications" Master's Thesis, AFIT/GCS/ENG/97D-08, School of Engineering, Air Force Institute of Technology Air University, Wright-Patterson AFB OH, December 1997.

24. Chengjiang Lin, "Time and Space Efficient Wavelet Transform For Real-Time Applications" Dissertation, The Ohio State University, Oh, 1999.

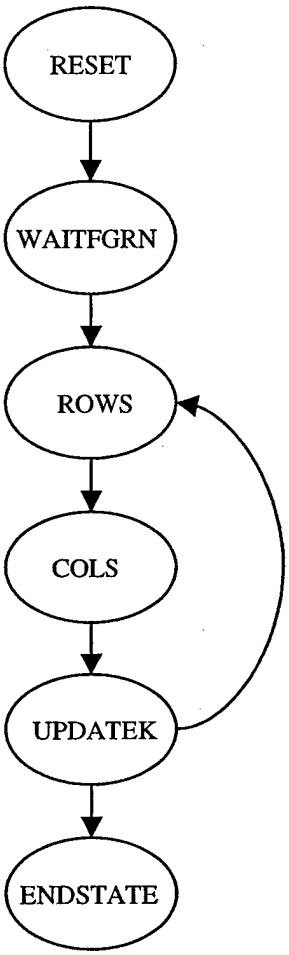
25. Michael John Sebastian Smith, *Application-Specific Integrated Circuits.* Addison Wesley Longman, Inc. 1997

26. Neil H. E. Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design, A system Perspective*, Second Edition. Addison-Wesley Publishing Company, 1993 by AT&T.

27. Peter J. Ashenden, The Designer's Guide to VHDL, Morgan Kaufmann Publishers, Inc. 1996.

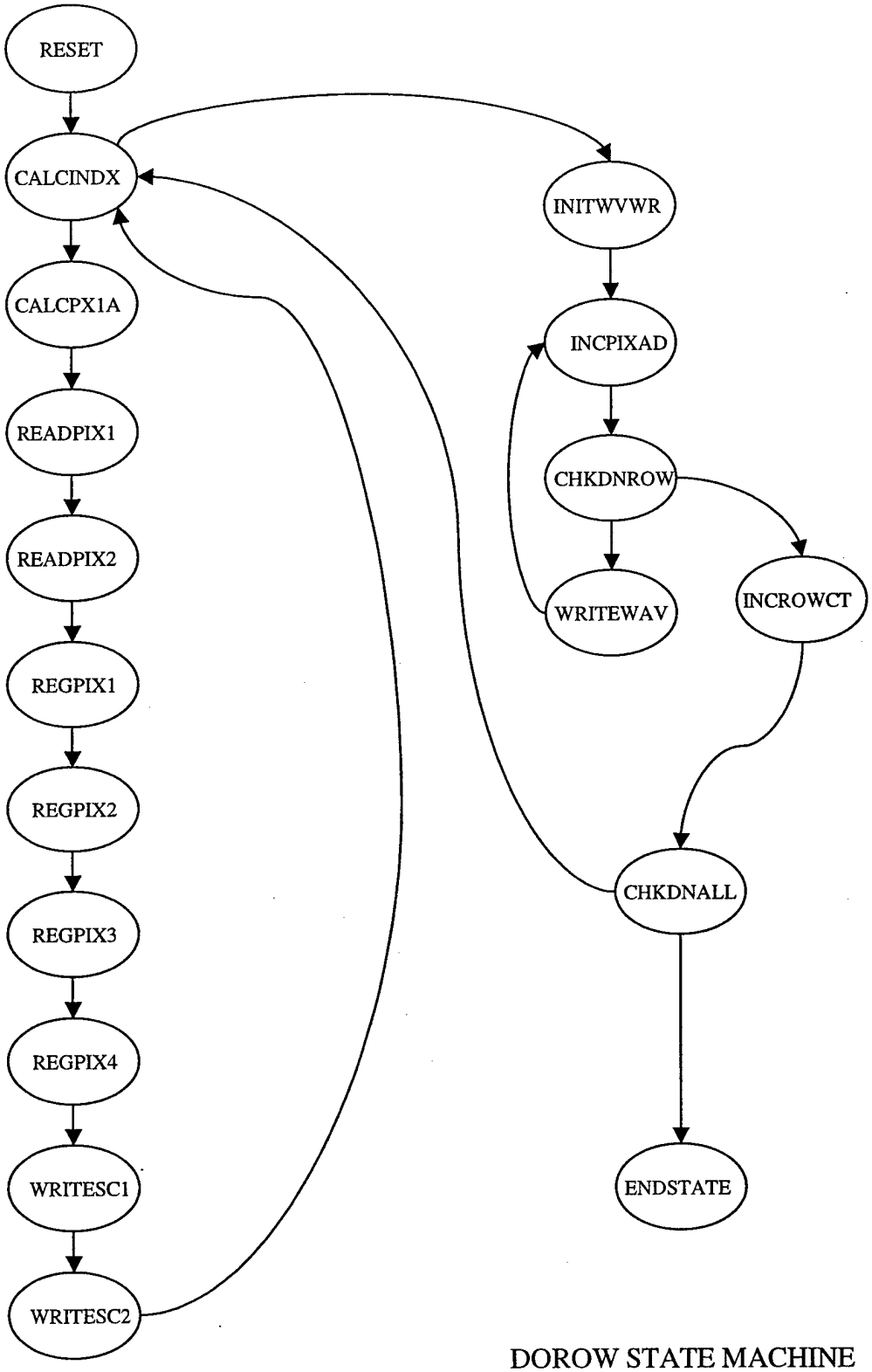
Appendix A. State Diagrams

A.1 Transform State Diagram



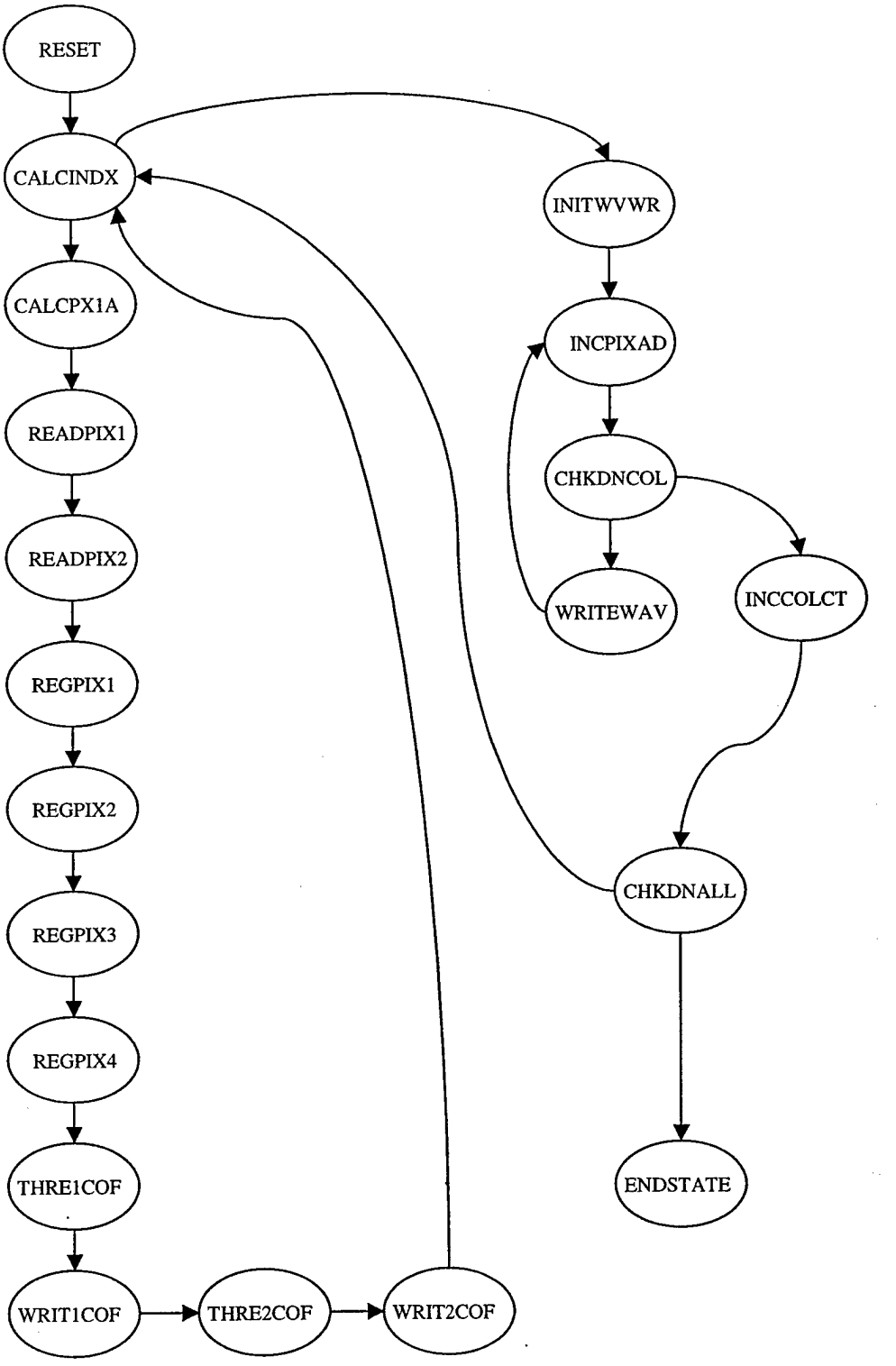
TRANSFORM STATE MACHINE

A.2 Row Transform State Diagram



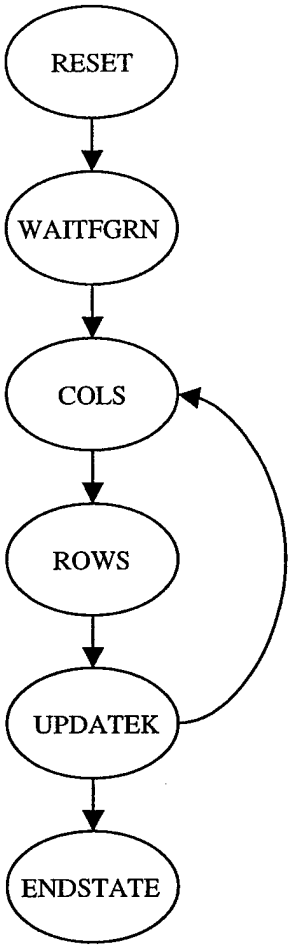
DOROW STATE MACHINE

A.3 Column Transform State Diagram



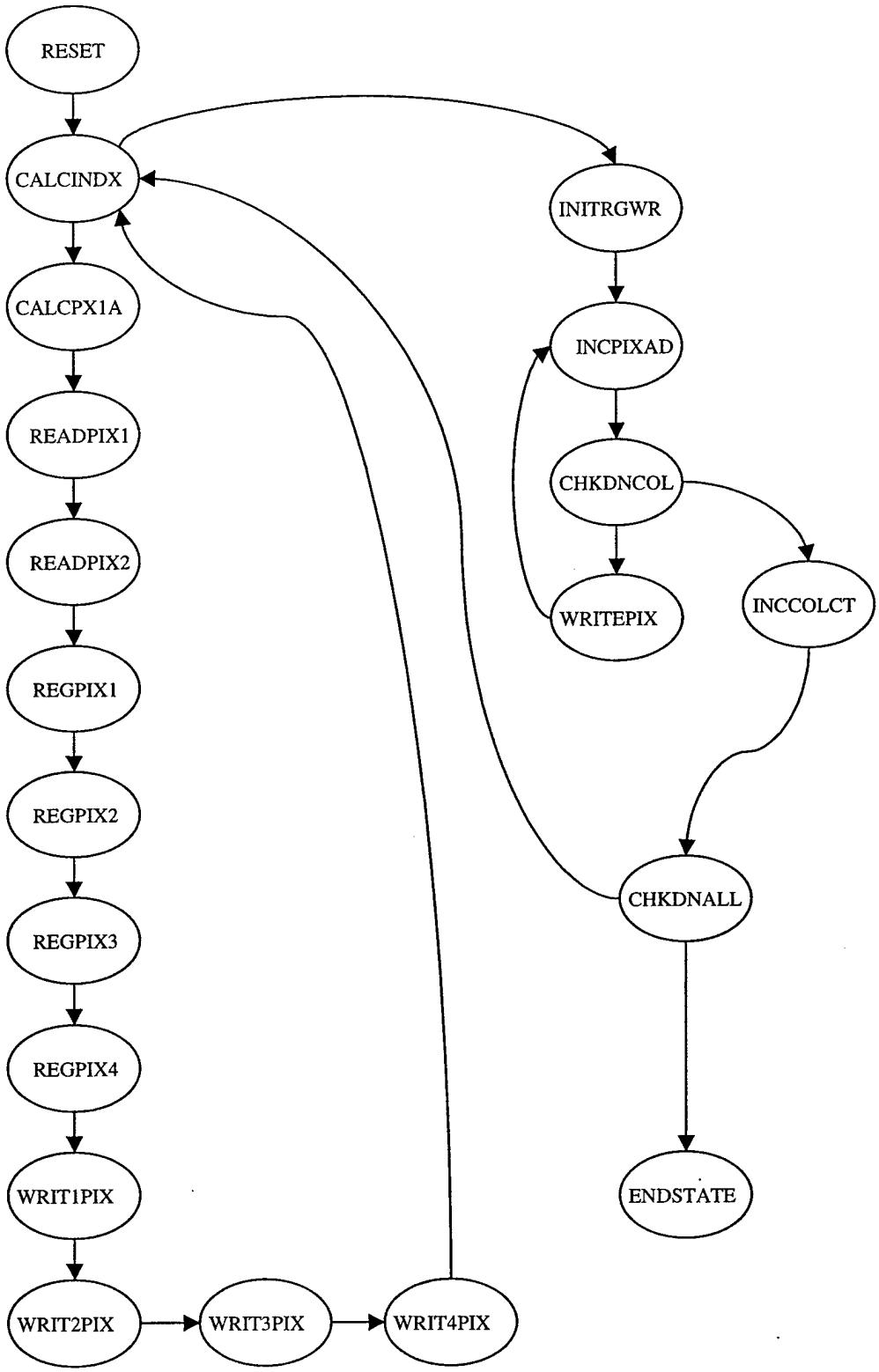
DOCOL STATE MACHINE

A.4 Inverse Transform State Diagram



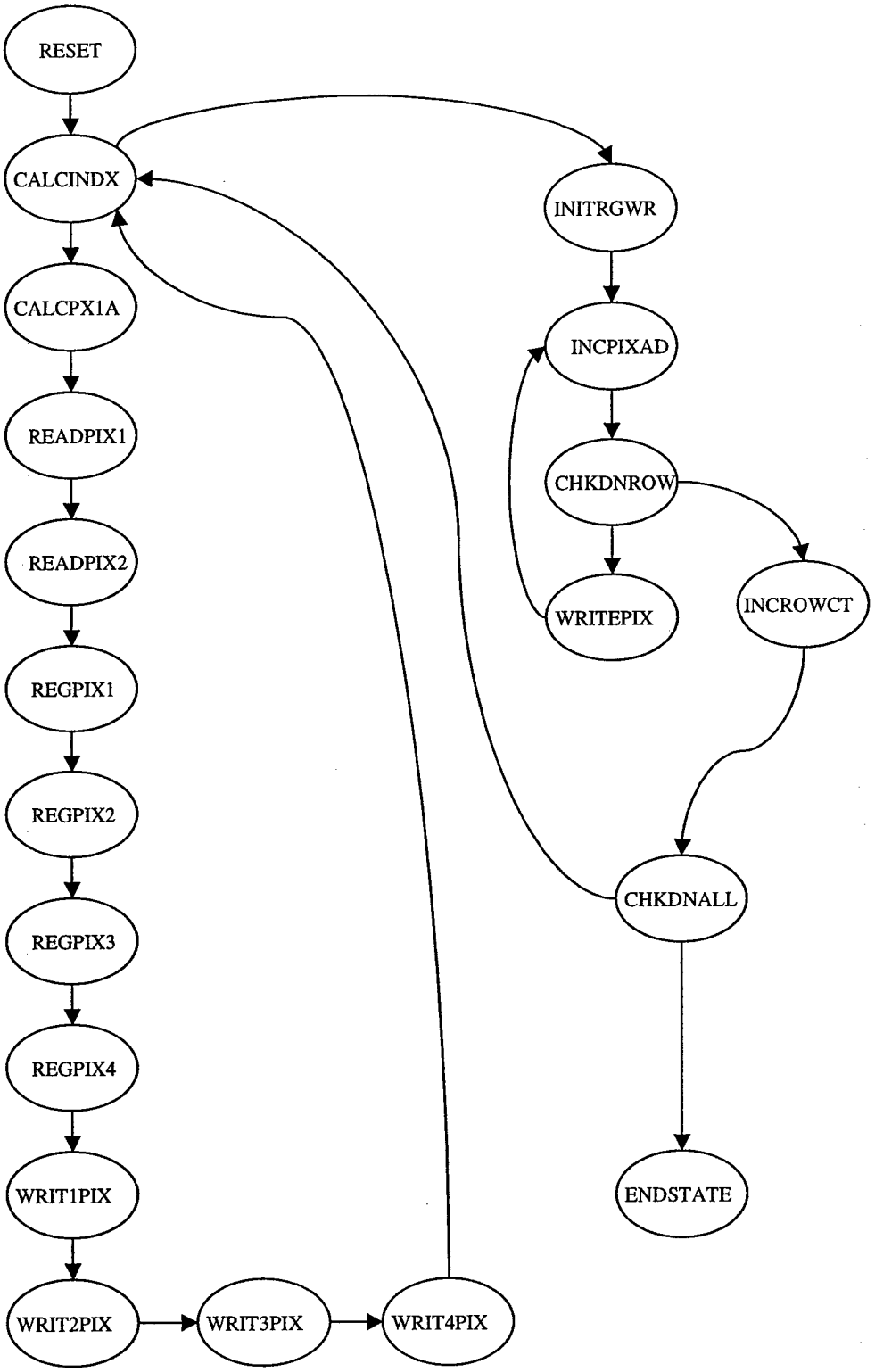
INVERSE TRANSFORM STATE MACHINE

A.5 Inverse Column Transform State Diagram



UNCOL STATE MACHINE

A.6 Inverse Row Transform State Diagram



UNROW STATE MACHINE

Appendix B. Savings of Ram Accesses

B.1 Original Code Ram Accesses

Operation (Original Code)	**Reads**	**Writes**
Read in 4 words, **Pack**, write out 1 word	512	128
Read in 2 words, **Transform**, write 2 words	128	128
Read in 2 words, **Unpack**, write 8 words	128	512
Do these 512 times (512 rows)		
Total Ram accesses	**393216**	**393216**
Read in 4 words, **Pack**, write out 1 word	512	128
Read in 2 words, **Transform**, write 2 words	128	128
Read in 2 words, **Unpack**, write 8 words	128	512
Do these 512 times (512 cols)		
Total Ram accesses	**393216**	**393216**
Read in 4 words, **Pack**, write out 1 word	256	64
Read in 2 words, **Transform**, write 2 words	64	64
Read in 2 words, **Unpack**, write 8 words	64	256
Do these 256 times (256 rows)		
Total Ram accesses	**98304**	**98304**
Read in 4 words, **Pack**, write out 1 word	256	64
Read in 2 words, **Transform**, write 2 words	64	64
Read in 2 words, **Unpack**, write 8 words	64	256
Do these 256 times (256 cols)		
Total Ram accesses	**98304**	**98304**
Read in 4 words, **Pack**, write out 1 word	128	32
Read in 2 words, **Transform**, write 2 words	32	32
Read in 2 words, **Unpack**, write 8 words	32	128
Do these 128 times (128 rows)		
Total Ram accesses	**24576**	**24576**
Read in 4 words, **Pack**, write out 1 word	128	32
Read in 2 words, **Transform**, write 2 words	32	32
Read in 2 words, **Unpack**, write 8 words	32	128
Do these 128 times (128 rows)		
Total Ram accesses	**24576**	**24576**
Total Ram access for transform	**1032192**	**1032192**
Quantize/Threshold image	512	512
512 rows		
Total Ram accesses	**262144**	**262144**
Grand Total Ram access	**1294336**	**1294336**

B.2 New Code Ram Accesses

Operation (New Code)	**Reads**	**Writes**
Read in 4 words, **Transform**, write out 4 words	512	512
Do these 512 times (512 rows)		
Total Ram accesses	**262144**	**262144**
Read in 4 words, **Transform**, write out 4 words	512	512
Do these 512 times (512 cols)		
Total Ram accesses	**262144**	**262144**
Read in 4 words, **Transform**, write out 4 words	256	256
Do these 256 times (256 rows)		
Total Ram accesses	**65536**	**65536**
Read in 4 words, **Transform**, write out 4 words	256	256
Do these 256 times (256 cols)		
Total Ram accesses	**65536**	**65536**
Read in 4 words, **Transform**, write out 4 words	128	128
Do these 128 times (128 rows)		
Total Ram accesses	**16384**	**16384**
Read in 4 words, **Transform**, write out 4 words	128	128
Do these 128 times (128 rows)		
Total Ram accesses	**16384**	**16384**
Total Ram accesses for transform	**688128**	**688128**
Quantize/Threshold: done during column transform	0	0
Total Ram accesses	**0**	**0**
Grand Total Ram access	**688128**	**688128**

Appendix C. Total States Required for Transform Half of Both The ASIC Design and The FPGA Design.

C.1 States for Transform Half of FPGA Design.

The transform occurs in three iterations. The three iterations are listed with subtotals for each iteration. A total number of states is listed for the transform step. Following the three iterations is the quantize step. The total number of states for the quantize step are listed followed by a total number of states for the transform and quantization. The totals are not exact. States that occur only a few times are not used in the totals. Not using the minimal occurring states simplified the counting process. The minimal number of extra states would not affect the overall number significantly.

TRANSFORM Steps			
First iteration			
Pack4Row			
7 states to process 4 pixels	7*128 for one row	896*512 for all rows	**458752**
Transform			
6 States to process 8 pixels	6*64 for one row	384*512 for all rows	**196608**
Unpack4row			
12 States to process 8 pixels	12*64 for one row	768*512 for all rows	**393216**
Total			**1048576**
Pack4Col			
7 states to read in and pack 4 pixels	7*128 for one row	896*512 for all rows	**458752**
Transform			
6 States to process 8 pixels	6*64 for one row	384*512 for all rows	**196608**
Unpack4Col			
12 States to unpack and write out 8 pixels	12*64 for one row	768*512 for all rows	**393216**
Total			**1048576**
Subtotal			**2097152**

Second Iteration			
Pack4Row			
7 states to process 4 pixels	7*64 for one row	448*256 for all rows	**114688**
Transform			
6 States to process 8 pixels	6*32 for one row	192*256 for all rows	**49152**
Unpack4row			
12 States to process 8 pixels	12*32 for one row	384*256 for all rows	**98304**
Total			**262144**
Pack4Col			
7 states to read in and pack 4 pixels	7*64 for one row	448*256 for all rows	**114688**
Transform			
6 States to process 8 pixels	6*32 for one row	192*256 for all rows	**49152**
Unpack4Col			
12 States to unpack and write out 8 pixels	12*32 for one row	384*256 for all rows	**98304**
Total			**262144**
Subtotal			**524288**

Third iteration			
Pack4Row			
7 states to process 4 pixels	7*32 for one row	224*128 for all rows	**28672**
Transform			
6 States to process 8 pixels	6*16 for one row	96*128 for all rows	**12288**
Unpack4row			
12 States to process 8 pixels	12*16 for one row	192*128 for all rows	**24576**
Total			**65536**
Pack4Col			
7 states to process 4 pixels	7*32 for one row	224*128 for all rows	**28672**
Transform			
6 States to process 8 pixels	6*16 for one row	96*128 for all rows	**12288**
Unpack4Col			
12 States to process 8 pixels	12*16 for one row	192*128 for all rows	**24576**
Total			**65536**
Subtotal			**131072**
TOTAL			**2752512**
QUANTIZE Steps			
9 states to process 2 coefficients	9*256 for one row	2304*512 for all rows	**1179648**
GRAND TOTAL			**3932160**

C.2 States for Transform Half of ASIC Design

The transform occurs in three iterations. The three iterations are listed with subtotals for each iteration. A total number of states is listed for the transform step. The total is not exact. States that occur only a few times are not used in the totals. Not using the minimal occurring states simplified the counting process. The minimal number of extra states would not affect the overall number significantly.

First iteration			
Row state			
10 states to read in 4 pixels	10*128 for one row	1280*512 for all rows	**655360**
1+3*256 +1+1 states to write out register	771*512 for all rows		**394752**
Total			**1050112**
Col state			
12 states to read in 4 pixels	12*128 for one row	1536*512 for all rows	**786432**
1+3*256 +1+1 to write out register		771*512 for all rows	**394752**
Total			**1181184**
Subtotal			**2231296**
Second interation			
Row state			
10 states to read in 4	10*64 for one row	640*256 for all rows	**163840**
1+3*128 +1+1 to write out register		387*256 for all rows	**99072**
Total			**262912**
Column State			
12 states to read in 4	12*64 for one row	768*256 for all rows	**196608**
1+3*128 +1+1 to write out register		387*256 for all rows	**99072**
Total			**295680**
Subtotal			**558592**

Third interation			
Row state			
10 states to read in 4 pixels	10*32 for one row	320*128 for all rows	**40960**
1+3*64 +1+1 to write out register		195*128 for all rows	**24960**
Total			**65920**
Column State			
12 states to read in 4 pixels	12*32 for one row	384*128 for all rows	**49152**
1+3*64 +1+1 to write out register		195*128 for all rows	**24960**
Total			**74112**
Subtotal			**140032**
GRAND TOTAL			**2929920**

Appendix D. Component Listing and Timing

Each component built for the ASIC design is listed in this appendix. Each component went through many changes. Some of the steps were not executed for some of the components. The steps are as follows:

1. Write Behavioral VHDL (Beh VHDL).
2. Test Behavioral VHDL (Test Beh).
3. Executed Design Analyzer (DA).
4. Number of optimizations performed in Design Analyzer (# Opt).
5. Timing from Design Analyzer (Design Analyzer Timing Parameters).
6. DB2SGE conversion (DP2SGE).
7. Creation of Structural VHDL (Str VHDL).
8. Test Structural VHDL (Test Struct).
9. Timing from Synopsys VHDL analyzer (Timing (ns)).
10. Edit SDL file produced by SGE (Edit SDL).
11. Open component in MAGIC (Mag).
12. Size of component in MAGIC (Magic Size).
13. IRSIM timing result of the component (IRSIM (ns)).
14. HSPICE execution (HSPICE).
15. Timing results from HSPICE (HSPICE Timing (ns)).

The 'X' indicates the component was run through the associated step. A '-----' indicates the component did not run through the associated step. Information following the component listing indicates several changes made to the components while they were being realized into a layout level component. Several signals were removed because they were either not necessary or they were realized by another signals. Verification of the unused signals was verified in the VHDL files but not changed. In some cases, buffers were added to some of the components built using the automated tools.

Components	Beh vhdl	Test Beh	D A	# Opt	Design Analyzer Timing Parameters				DB2 SGE	Str vhdl	Test Struct		Timing (ns)		Edit SDL	Mag	Magic Size	IRSIM (ns)		H SPICE	HSPICE Timing (ns)	
											alone	with code	L->H	H->L				L->H	H->L		L->H	H->L
add19_behav	X	X	X	2	10.26	9.88	1.82	1.11	X	X	X	X	3.10	2.78	X	X	1536x1080	7.25	7.33	X	5.62	5.46
add8_behav	X	X	X	2	6.29	5.91	1.14	0.77	X	X	X	X	1.48	1.44	X	X	768x728	3.38	3.46	X	2.80	2.57
add9_behav	X	X	X	0	12.39	12.01	1.82	1.18	X	X	X	X	2.30	2.30	X	X	595x640	5.28	5.17	X	4.11	3.94
compare10_behav	X	X	X	1	5.80	5.82	2.31	2.26	X	X	X	X	1.23	0.78	X	X	635x512	2.29	1.05	X	1.85	0.95
docollog_behav	Break into the following 5 files																					
docolla_behav	X	-----	X	1	3.18	3.15	1.85	1.44	X	X	-----	-----	-----	-----	X	X	744x536	-----	-----	X	0.83	0.38
docollb_bahav	X	-----	X	1	5.34	5.14	0.80	0.60	X	X	-----	-----	-----	-----	X	X	2192x1296	-----	-----	X	2.02	2.73
docollc_bahav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	1964x1408	-----	-----	X	1.85	1.56
docolld_bahav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	780x816	-----	-----	X	1.36	1.19
docolle_bahav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	5496x2712	-----	-----	X	2.01	2.44
docolsm_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	680x608	1.59	1.54	X	1.18	0.90
dorowlog_behav	Break into the following 5 files																					
dorowla_behav	X	-----	X	1	4.06	4.03	1.14	1.17	X	X	-----	-----	-----	-----	X	X	880x632	-----	-----	X	1.35	0.56
dorowlb_behav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	927x888	-----	-----	X	0.97	0.84
dorowlc_behav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	1436x1176	-----	-----	X	1.05	0.86
dorowld_behav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	2296x1720	-----	-----	X	1.81	1.54
dorowle_behav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	2619x1648	-----	-----	X	1.94	1.65
dorowsm_behav	X	X	X	1	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	532x632	1.39	1.30	X	1.00	0.82
inc10_behav	X	X	X	0	8.17	7.78	1.14	0.77	X	X	X	X	1.38	1.38	X	X	400x472	4.72	4.72	X	3.16	2.94
inc19_behav	X	X	X	1	3.58	3.20	1.27	0.89	X	X	X	-----	-----	-----	X	X	736x568	4.93	4.93	X	2.12	1.78
incby10_behav	X	X	X	1	3.58	3.20	1.27	0.89	X	X	X	-----	-----	-----	X	X	783x656	4.31	3.52	X	2.04	1.09
invtrasm_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	508x440	-----	-----	X	0.95	0.91
invtrlog_behav	X	X	X	1	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	368x392	-----	-----	X	0.12	0.49
mux2_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	575x568	-----	-----	X	0.32	0.43
mux2b8_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	307x352	-----	-----	X	0.28	0.39
mux2b9_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	323x424	-----	-----	X	0.29	0.39
mux3b10_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	567x520	-----	-----	X	0.34	0.43
mux3b19_behav	X	X	X	0	5.52	6.12	1.35	1.18	X	X	X	-----	-----	-----	X	X	931x744	-----	-----	X	0.35	0.45
mux3b5_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	323x400	-----	-----	X	0.31	0.39
mux3b9_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	535x488	-----	-----	X	0.33	0.42
mux4b10_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	787x560	-----	-----	X	2.56	1.31
mux4b19_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	1255x848	-----	-----	X	3.14	1.70
mux4b9_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	747x520	-----	-----	X	2.33	1.20
mux5_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	1595x1008	-----	-----	X	2.47	1.67
mux6b19_behav	X	X	X	1	6.27	6.22	2.14	2.20	X	X	X	-----	-----	-----	X	X	2635x1440	-----	-----	X	3.04	2.30
mux7b19_behav	X	X	X	1	5.43	5.23	0.80	0.60	X	X	X	-----	-----	-----	X	X	2259x1312	-----	-----	X	1.86	1.89
sub9_behav	X	X	X	2	9.76	9.37	1.96	1.18	X	X	X	-----	-----	-----	X	X	1080x816	2.28	3.79	X	1.05	2.95
topdocols	Logic implemented as topdclog_behav																					
topdclog_behav	X	X	X	3	1.92	2.26	1.33	1.31	X	X	X	-----	-----	-----	X	X	392x312	-----	-----	X	0.49	0.69
topdorows	Logic implemented as topdrlog_behav																					
topdrlog_behav	X	X	X	0	1.34	1.45	1.22	1.12	X	X	X	-----	-----	-----	X	X	251x232	-----	-----	X	0.29	0.72

Components	Beh vhdl	Test Beh	D A	# Opt	Design Analyzer Timing Parameters				DB2 SGE	Str vhdl	Test Struct alone	Test Struct with code	Timing (ns) L->H	Timing (ns) H->L	Edit SDL	Mag	Magic Size	IRSIM (ns) L->H	IRSIM (ns) H->L	H SPICE	HSPICE Timing (ns) L->H	HSPICE Timing (ns) H->L
topinvtr_behav	Logic implemented as topinla_bahave and topinlb_behav. Statechoice machine implemented as tinsmchc_behav.																					
topinla_behav	Same as toptrla_behav. Rpix = rowpix, mpix = mempix, cpix = colpix																					
topinlb_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	-----	X	1547x856	-----	-----	X	0.63	0.60
topinlc_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	-----	X	479x416	-----	-----	X	0.29	0.48
tinsmchc	Same as ttrsmchc_behav. Row=unrow,col=uncol																					
toptrans_behav	Logic implemented as toptrla_bahave and toptrlb_behav and toptrlc_behav. Statechoice machine implemented as ttrsmchc_behav.																					
toptrla_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	-----	X	2031x960	-----	-----	X	0.67	0.63
toptrlb_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	-----	X	2055x1008	-----	-----	X	0.69	0.64
toptrlc_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	-----	X	415x336	-----	-----	X	0.53	0.49
ttrsmchc	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	316x384	-----	-----	X	1.09	0.90
topuncols	Logic implemented as topuclog_behav																					
topuclog_behav	X	X	X	3	1.63	1.61	1.22	1.45	X	X	X	-----	-----	-----	X	X	432x352	-----	-----	X	0.31	0.77
topunrows	Logic implemented as topurlog_behav																					
topurlog_behav	X	X	X	1	1.22	1.45	1.22	1.06	X	X	X	-----	-----	-----	X	X	400x344	-----	-----	X	0.29	0.55
transfsm_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	491x440	-----	-----	X	0.92	0.88
translog_behav	X	X	X	1	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	368x424	-----	-----	X	0.24	0.12
uncollog_behav	Break into the following 4 files																					
uncolla_behav	X	-----	X	1	3.68	3.62	1.74	1.71	X	X	-----	-----	-----	-----	X	X	1036x696	-----	-----	X	1.75	1.19
uncollb_behav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	1852x1400	-----	-----	X	1.72	1.49
uncollc_behav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	1664x1256	-----	-----	X	1.32	1.52
uncolld_behav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	4436x2288	-----	-----	X	2.37	2.01
uncolsm_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	728x608	-----	-----	X	1.05	1.13
unrowlog_behav	Break into the following 4 files																					
unrowla_behav	X	-----	X	1	3.59	3.58	1.86	1.92	X	X	-----	-----	-----	-----	X	X	988x680	-----	-----	X	1.51	0.70
unrowlb_behav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	1884x1368	-----	-----	X	1.63	1.41
unrowlc_behav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	2148x1488	-----	-----	X	1.81	1.56
unrowld_behav	X	-----	X	0	-----	-----	-----	-----	X	X	-----	-----	-----	-----	X	X	3344x1920	-----	-----	X	2.17	1.87
unrowsm_behav	X	X	X	0	-----	-----	-----	-----	X	X	X	-----	-----	-----	X	X	728x608	-----	-----	X	1.08	1.20
zbuftwo	-----	-----	---	---	-----	-----	-----	-----	----	-----	-----	-----	-----	-----	-----	-----		-----	-----	X	0.39	0.30
docolmem_behav	Implemented with the following 4 blocks																					
mux6b19																						
mux3b9																						
mux2b8																						
45 tristate buffers																						
uncolmem_behav	Implemented with the following 4 blocks																					
mux5																						
mux3b9																						
mux2b9																						
mux3b9																						
mux4b9																						
45 tristate buffers																						

Component	Changes											
unrowla:	column	1	=	shift	0		shift	1	=	shift	3	
	column	3	=	shift	2							
	column	5	=	shift	4							
	column	6	=	shift	5							
	column	8	=	shift	7							
	column	7	=	shift	6							
	column	9	=	shift	8							
uncolld:	11 buffers use 13 dff											
	1 buffer uses 14 dff											
uncollb:	4 buffers use 16 dff											
uncollc	4 buffers use 11 dff											
unrowlb	5 buffers use 13 dff											
unrowlc	3 buffers use 13 dff											
	1 buffers use 15 dff											
uncolla	shift	0	=	0			shift1=shift3					
	shift	13	=	row	9							
	shift	8	=	row	4							
	shift	9	=	row	5							
	shift	5	=	row	1							
	shift	6	=	row	2							
	shift	7	=	row	3							
	shift	4	=	row	0							
	shift	10	=	row	6							
	shift	11	=	row	7							
	shift	12	=	row	8							
docollb_behav	scal1c7 = tscal1c7											
	scal2c7 = tscal2c7											
	wave1c7 = twavl1c7											
	wave1c7 = twavl1c7											
	scadd3 = wvadd4											
docollc:	4 buffers use 14 dff, 1 buffer use 10 dff											
docolld:	1 buffer use 10 dff, 1 buffer use 9 dff											
docolle:	Removed add9res0 from SGE but it is still in VHDL											

Appendix E. Sections of Code with the Utilized Components.

The list of components contains all the VHDL files used for each of the four main pieces of the design. The files used to tie each halve together are also listed. It is noted if the file was implemented or just needed for testing purposes. The list doesn't specify how many of each component was used. Referencing the block diagrams will show the number of each component utilized.

Row Transform	**Description**	**Purpose**
dorowla	Control logic	Implemented
dorowlb	Control logic	Implemented
dorowlc	Control logic	Implemented
dorowld	Control logic	Implemented
dorowle	Control logic	Implemented
dorowsm	State Machine	Implemented
compare10	10 bit comparator	Implemented
mux3b10	3 by 10 Bit Multiplexer	Implemented
incby10	19 Bit Increment by 10	Implemented
inc10	10 bit Increment by 1	Implemented
add10	10 Bit Adder	Implemented
inc19	19 Bit Adder	Implemented
add9	9 Bit Adder	Implemented
sub9	9 Bit Subtractor	Implemented
mux2	2 by 19 Bit Multiplexer	Implemented
mux5	5 by 19 Bit Multiplexer	Implemented
mux2b9	2 by 9 Bit Multiplexer	Implemented
topdrlog	Misc Control Logic	Implemented
Column Transform		
docolla	Control logic	Implemented
docollb	Control logic	Implemented
docollc	Control logic	Implemented
docolld	Control logic	Implemented
docolle	Control logic	Implemented
docolsm	State Machine	Implemented
mux4b10	4 by 10 Bit Multiplexer	Implemented
compare10	10 bit comparator	Implemented
incby10	19 Bit Increment by 10	Implemented
inc10	10 bit Increment by 1	Implemented
add19	19 Bit Adder	Implemented
add8	8 Bit Adder	Implemented
add9	9 Bit Adder	Implemented
sub9	9 Bit Subtractor	Implemented

mux3b10	3 by 10 Bit Multiplexer	Implemented
mux7b19	7 by 19 Bit Multiplexer	Implemented
mux4b19	4 by 19 Bit Multiplexer	Implemented
mux2b8	3 by 10 Bit Multiplexer	Implemented
mux3b9	3 by 9 Bit Multiplexer	Implemented
mux4b9	4 by 9 Bit Multiplexer	Implemented
topdclog	Misc logic	Implemented
Column Inverse Transform		
uncolla	Control Logic	Implemented
uncollb	Control Logic	Implemented
uncollc	Control Logic	Implemented
uncolld	Control Logic	Implemented
uncolsm	State Machine	Implemented
mux4b10	4 by 10 Bit Multiplexer	Implemented
compare10	10 bit comparator	Implemented
incby10	19 Bit Increment by 10	Implemented
inc10	10 bit Increment by 1	Implemented
add19	19 Bit Adder	Implemented
add8	8 Bit Adder	Implemented
sub9	9 Bit Subtractor	Implemented
mux3b10	3 by 10 Bit Multiplexer	Implemented
mux6b19	6 by 19 Bit Multiplexer	Implemented
mux2b8	2 by 8 Bit Multiplexer	Implemented
topuclog	Misc logic	Implemented
Row Inverse Transform		
unrowla	Control Logic	Implemented
unrowlb	Control Logic	Implemented
unrowlc	Control Logic	Implemented
unrowld	Control Logic	Implemented
unrowsm	State Machine	Implemented
mux4b10	4 by 10 Bit Multiplexer	Implemented
compare10	10 bit comparator	Implemented
incby10	19 Bit Increment by 10	Implemented
inc10	10 bit Increment by 1	Implemented
add19	19 Bit Adder	Implemented
inc19	19 Bit Increment by 10	Implemented
add8	8 Bit Adder	Implemented
sub9	9 Bit Subtractor	Implemented
mux3b10	3 by 10 Bit Multiplexer	Implemented
mux3b19	3 by 19 Bit Multiplexer	Implemented
mux2b8	2 by 8 Bit Multiplexer	Implemented
topurlog	Misc logic	Implemented
Transform Half of Code		
transfsm	State Machine	Implemented
translog	Control Logic	Implemented
toptrla	Control Logic	Implemented
toptrlb	Control Logic	Implemented
toptrlc	Control Logic	Implemented
ttrsmchc	State Bit Multiplexer	Implemented

Inverse Transform Half of Code		
invtrasm	State Machine	Implemented
invtrlog	Control Logic	Implemented
topinla	Control Logic	Implemented
topinlb	Control Logic	Implemented
topinlc	Control Logic	Implemented
ttrsmchc	State Bit Multiplexer	Implemented
Extra Logic for ASIC		
mux3b5	Multiplexer	Implemented
clock_gen	Clock and Reset signals	Testing only
std_logic_vector_to_integer	Used for simulated Ram	Testing only
memory1	Used for simulated Ram	Testing only
ram1_behav	Used for simulated Ram	Testing only
std_logic_vector19_to_integer	Used for simulated Register File	Testing only
memory2	Used for simulated Register File	Testing only
ram2_behav	Used for simulated Register File	Testing only
flipflop_behav	Simulates 2 cycle read delay	Testing only

Appendix F. Data Used for Testing.

The following data shows the beginning value placed in RAM. It also shows the intermediate values in the memory locations that are used to compute the ending value that is subject to the current test. Each location was verified while executing the code. The final result was verified as well. The result in parenthesis is the actual result written to RAM. Each quadrant of each iteration was verified. The table shows the steps used for the Transform half of the code. It was the most complicated as it had many design rules. The inverse transform half was tested but it wasn't necessary to use the same range of inputs, as the operations were always the same in each iteration for each quadrant.

Iteration k=0: Lower Left								Iteration k=1: Lower Left								Iteration k=2: Lower Left						
Loc	Val	Loc	Val	Loc	Val	EndVal		Loc	Val	Loc	Val	Loc	Val	EndVal		Loc	Val	Loc	Val	Loc	Val	EndVal
10	0	10	0	522	-20	(-8)		10	-82	10	-82	266	-82	(-64)		10	-82	10	-82	138	-82	(-82)
11	0							11	-82							11	-82					
42	0	42	40					42	82	42	82					42	82	42	82			
43	80							43	82							43	82					
74	0	74	0	554	-9	(-8)		74	-65	74	-65	298	-65	(-64)		74	-65	74	-65	170	-65	(-64)
75	0							75	-65							75	-65					
106	0	106	18					106	65	106	65					106	65	106	65			
107	36							107	65							107	65					
138	0	138	0	586	-3	(0)		138	-21	138	-21	330	-21	(-20)		138	-21	138	-21	202	-21	(-20)
139	0							139	-21							139	-21					
170	0	170	6					170	21	170	21					170	21	170	21			
171	12							171	21							171	21					
202	20	202	10	618	5	(0)		202	20	202	10	362	5	(4)		202	20	202	10	234	5	(4)
203	0							203	0							203	0					
234	0	234	0					234	0	234	0					234	0	234	0			
235	0							235	0							235	0					
266	68	266	34	650	17	(8)		266	101	266	101	394	101	(64)								
267	0							267	101													
298	0	298	0					298	-101	298	-101											
299	0							299	-101													

Iteration k=0: upper right (scaler)						
Loc	Val	Loc	Val	Loc	Val	EndVal
10	-20	26	-20	26	-20	(-8)
11	20					
42	-20	58	-20			
43	20					
74	-9	90	-9	58	-9	(-8)
75	9					
106	-9	122	-9			
107	9					
138	-3	154	-3	90	-3	(0)
139	3					
170	-3	186	-3			
171	3					
202	20	218	10	122	5	(0)
203	0					
234	0	250	0			
235	0					
266	17	282	17	154	17	(8)
267	-17					
298	17	314	17			
299	-17					

Iteration k=1: upper right (scaler)						
Loc	Val	Loc	Val	Loc	Val	EndVal
10	-82	18	-82	18	-82	(-64)
11	82					
42	-82	50	-82			
43	82					
74	-65	82	-65	50	-65	(-64)
75	65					
106	-65	114	-65			
107	65					
138	-21	146	-21	82	-21	(-20)
139	21					
170	-21	178	-21			
171	21					
202	20	210	10	114	5	(4)
203	0					
234	0	242	0			
235	0					
266	101	274	101	146	101	(64)
267	-101					
298	101	306	101			
299	-101					

Iteration k=2: upper right (scaler)						
Loc	Val	Loc	Val	Loc	Val	EndVal
10	-82	14	-82	14	-82	(-82)
11	82					
42	-82	46	-82			
43	82					
74	-65	78	-65	46	-65	(-64)
75	65					
106	-65	110	-65			
107	65					
138	-21	142	-21	78	-21	(-20)
139	21					
170	-21	174	-21			
171	21					
202	20	206	10	110	5	(4)
203	0					
234	0	238	0			
235	0					

Iteration k=0: Lower Right						
Loc	Val	Loc	Val	Loc	Val	EndVal
10	0	26	0	538	-24	(-8)
11	0					
42	0	58	48			
43	-96					
74	0	90	0	570	-8	(0)
75	0					
106	0	122	16			
107	-32					
138	0	154	0	602	-1	(0)
139	0					
170	0	186	2			
171	-4					
202	28	218	14	634	7	(0)
203	0					
234	0	250	0			
235	0					
266	80	282	40	666	20	(8)
267	0					
298	0	314	0			
299	0					

Iteration k=1: Lower Right						
Loc	Val	Loc	Val	Loc	Val	EndVal
10	-65	18	-65	274	-65	(-64)
11	65					
42	65	50	65			
43	-65					
74	-63	82	-63	306	-63	(-56)
75	63					
106	63	114	63			
107	-63					
138	-7	146	-7	338	-7	(0)
139	7					
170	7	178	7			
171	-7					
202	48	210	24	370	12	(8)
203	0					
234	0	242	0			
235	0					
266	101	274	101	402	101	(64)
267	-101					
298	-101	306	-101			
299	101					

Iteration k=2: Lower Right						
Loc	Val	Loc	Val	Loc	Val	EndVal
10	-65	14	-65	142	-65	(-64)
11	65					
42	65	46	65			
43	-65					
74	-63	78	-63	174	-63	(-60)
75	63					
106	63	110	63			
107	-63					
138	-7	142	-7	206	-7	(-4)
139	7					
170	7	174	7			
171	-7					
202	48	206	24	238	12	(12)
203	0					
234	0	238	0			
235	0					

Appendix G. Power Calculation of the FPGA Design.

Using the WildForce board documentation (3), an estimate of the power usage was calculated. According to the documentation:

Total Power = Base Power + Memory Power + External I/O Power + Total PE Power

Base Power = 3.75 W
Memory Power = 5 W
External I/O Power = 0W
Total PE Power = Number of Pe's * PE Power
Number of Pe's = 5 (CPE0, PE1, PE2, PE3, PE4)
PE Power = ((.02 * Frequency) + 0.09) * Activity * Size Factor * 5V
Frequency = 20 MHz
Activity is percent of registers that are switching at same time:
(% utilization of flip-flops) * (% active at any given time)
Activity = .47 * .40
Size Factor for FPGA type 4062XL = 1.23
PE Power = ((.02 * 20) + 0.09) * 0.188 * 1.23 * 5V = 0.567 W
Total PE Power = 5 * 0.567 W = 2.8 W
Total Power = 3.75 + 5 + 0 + 2.8 = 11.6 W

Activity was estimated based on the Behavioral Code and the WildForce Documentation. A better estimate could have been calculated by analyzing the files produced by the WildForce loading program (3). However, since the majority of the total power is based on the *memory* and *base* power, the estimate is sufficient for comparison purposes.