Embedded Vector Processor Architecture for Real-Time Wavelet Video Compression

Jeffrey A. Shafer

May 2004

Department of Electrical and Computer Engineering
University of Dayton
Dayton, OH 45469

Embedded Vector Processor Architecture for Real-Time Wavelet Video Compression

by

Jeffrey A. Shafer

Thesis

Submitted to the Faculty of

The School of Engineering of the

University of Dayton

in Partial Fulfillment

of the Requirements

for the Degree of

Masters of Science in Electrical Engineering

The University of Dayton

Dayton, Ohio

May 2004

Embedded Vector Processor Architecture for Real-Time Wavelet Video Compression

Approved by:

John G. Weber, Ph.D
Advisory Committee Chairman
Professor, Electrical and Computer Engineering Department

Frank A. Scarpino, Ph.D.
Committee Member
Professor, Electrical and Computer Engineering Department

Waleed W. Smari, Ph.D.
Committee Member
Associate Professor, Electrical and Computer Engineering Department

Donald L. Moon, Ph.D
Associate Dean
Graduate Engineering Programs & Research
School of Engineering

Joseph E. Saliba, Ph.D
Dean, School of Engineering

Abstract

Embedded Vector Processor Architecture for Real-Time Wavelet Video Compression

by

Jeffrey Shafer
University of Dayton

Dr. John G. Weber, Chair

In this thesis, a scalable vector processor is designed and implemented using VHDL on an Altera Stratix-series FPGA. The primary application of this processor is to compute forward and inverse wavelet transformations using matrix multiplications. This transformation is the fundamental component of a wavelet video compression system, and the ability to compute the transform in real-time with a low power device would be very advantageous for embedded civilian and military applications.

This method of computation combines both the flexibility of software with the high performance of custom logic by incorporating several unique features. First, it supports a sparse matrix multiplication approach to the wavelet transform. This method saves both computation time and memory, as only the wavelet coefficients need to be stored, and not a full transformation matrix.

Second, the machine has a Harvard architecture, which separates instructions and data. The fetch unit has an embedded dual-port memory containing the instructions, and it automatically preloads instructions in both possible directions of a branch, eliminating costly stalls if the branch is taken.

Third, a dedicated multiply and accumulate operation is provided to assist in computing the matrix multiplication. Through an innovative packed pixel scheme and the aid of three parallel ALUs, all three color planes of a standard image are transformed in parallel, greatly increasing compression performance.

Fourth, a special instruction was added to automatically stream data from input pins to data memory. This is used to incorporate the processor into a full compression system containing other modules such as quantizers or encoders.

When programmed in VHDL and fitted on an Altera Stratix FPGA, less than 2% of the reconfigurable resources available on the FPGA were used. Additional modules for quantization or encoding could easily be added on the same chip. The current performance of the processor at 75 MHz was sufficient to allow the full transformation of 19+ 256x256 color images per second. Although this is not quite real-time video, several options exist to improve device performance given that substantial FPGA fabric remains unused. Additional ALUs could be added to compute more results in parallel, accelerating each instruction. Or, multiple copies of this processor could be instantiated to run in parallel. Either of these design options would require adding additional ports to the data memory, or increasing its word width to provide higher bandwidth. Any or all of these design expansions along with some clock-speed related optimizations should boost performance to real-time levels.

Further, last minute tests on the just-released Altera Stratix-II FPGAs yielded a 33%+ improvement in clock rate without any design changes at all. This fabric was able to boost the processor to 106+ Mhz. Thus, it should be easy to foresee real-time performance of this processor architecture in the near future.

Acknowledgements

This thesis was made possible with the guidance and support of many people, including:

- John Weber, my principle advisor at Dayton, who provided the initial impetus to the vector processor project, and contributed countless bits of his real-world experience to enhance its design.
- Frank Scarpino, defense committee member and head of the configurable computing research effort, has served as a boss and mentor for the past 4 years. In that time, I still have not heard all of his stories!
- Waleed Smari, defense committee member, has been a long-time advisor from the time before my undergraduate thesis. His advice in choosing a grad school for my PhD and the wonderful letter of recommendation written in pursuit of that goal leaves me eternally in his debt.
- Rob Ewing, Kerry Hill, Al Scarpelli, and the rest of the Air Force Research Laboratory researchers and managers at Wright-Patterson Air Force Base, whose financial support and program leadership over the last 5+ years has built an outstanding environment for cutting-edge research into reconfigurable computing and video compression.
- Eric Balster, researcher at AFRL and a UD graduate, who started the wavelet compression project with Frank Scarpino several years ago, and whose intimate knowledge of wavelets made up for the "limited" gaps in my own.
- Joe Fieler and Bo Qiang, fellow grad students on the research team, who engaged in an uplifting trade of helpful technical tips, new research ideas, and most important of all, actual research results, that strengthened all three of our theses.
- Ken and Ruth Ann, my parents, for their unconditional love and support, and for the outstanding cooking and laundry services during the writing of this thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Video compression is currently a prominent topic for both military and commercial researchers, due to the rapid proliferation of digital media and the subsequent need to store and transmit it in a space and time-effective manner. Most successful compression methods have been based on mathematically transforming an image (or sequence of images) into a frequency domain representation, and then filtering that representation to obtain a form suitable for effective encoding and compression.

The mathematical process of wavelet image transformations can be characterized as a matrix problem. This has two key advantages. First, the transform and reconstruction matrices may be computed a` priori, reducing the amount of computation required for each image. Second, the matrix algorithms become identical and depend only upon the values in the transform matrix and the image matrix, improving the design's flexibility. Both of these advantages are highly significant for hardware implementations.

Much of the research in image transformation and encoding techniques is conducted using software prototypes on commodity PCs. Once effective algorithms are identified, some are transitioned to special purpose hardware (VLSI or FPGA) to provide real-time or near-real time image compression. For example, the Center for Collaborative Computing at the University of Dayton has been active in these endeavors. This research group, of which I am a member, performs both software algorithm research and develops prototype hardware cores for real-time video compression for embedded military systems [Balster, Shafer]. For maximum efficiency and performance, these hardware cores stream data from memory through a fixed combination of multipliers and adders to compute wavelet transform coefficients.

These past research activities by our team have highlighted some of the inherent tradeoffs between software and hardware implementations [Shafer]. While the software approach allows for quick implementations and superior design flexibility (important in a research environment), the dedicated hardware implementation yields much higher performance. Further, because the end goal of our research effort is an embedded application, such as a real-time video feed to a solider on the battlefield, custom hardware still offers critical advantages in smaller size, weight, and power consumption.

Vector processors, though not as general purpose as their scalar cousins, have potential for specific applications. Lee and Stoodley examined the use of a simple in-order long vector microprocessor for multimedia applications. They showed that a 2-way, in-order vector processor with a vector length of 64 and a vector width of 8 occupied no more die area than a 4-way, out-of-order superscalar processor with short vector extensions. More importantly, they showed that the long vector processor outperformed the superscalar design by a factor of 2.7 in multimedia applications, and by a factor of 1.7 against a superscalar design with short vector extensions. However, they did agree that it is necessary in many applications to still have an efficient general-purpose processor available to sustain effective performance levels across a wide variety of applications [Lee]. Although this analysis is perhaps dated (1998) and surpassed by the continuous growth of large-scale superscalar designs, its conclusions are certainly still valid in the field of small low-power processors for embedded applications (such as video compression, the task chosen for this thesis).

In this thesis, a special purpose vector processor is created with the goal of producing a low-cost design that combines the performance and flexibility of both hardware and software approaches within the framework of embedded video compression. This embedded processor will compute the wavelet transformation and inverse transformation stage which is fundamental to the overall compression process. High performance is delivered by optimizing the hardware architecture for greater parallelisms than are possible in a generic commodity processor. In addition, using a special purpose processor imparts significant flexibility to the programmer. Using the same software algorithm, different wavelets can be computed simply by changing the

transformation matrix in memory. Further, non-wavelet transform algorithms can also be executed. This is in stark contrast to fixed hardware implementations such as filter banks that, while quite efficient in computing the wavelet transform, lack the flexibility to be easily adapted to other purposes.

This processor was designed in VHDL using Altera's Quartus software. The design was targeted towards the Altera Stratix series of FPGAs due to their substantial size and large on-chip memories. Given an external SRAM memory, however, the design could be transitioned to other FPGAs, including those in the Xilinx Virtex and Virtex-II families.

1.2 Thesis Organization

This thesis document is separated into eight chapters. Chapter 1 provides the background to the compression challenge and a description of the resources used. Chapter 2 outlines the growing field of wavelet compression and how matrix math can be used to perform a wavelet transformation. Chapter 3 details the traditional vector processing architecture and some advanced techniques used to improve performance. Chapter 4 proposes a vector processor design that can perform an efficient wavelet transform via a matrix multiplication, while Chapter 5 provides details on its implementation in VHDL. Chapter 6 covers the simulations done to verify the implementation's effectiveness. Chapter 7 details the incorporation of the vector processor into a larger real-time video compression system. Chapter 8 provides a summary of the work done and proposes enhancements to the vector processor to improve its real-time performance.

1.3 Resources Used

1. Altera Quartus II Design Software version 3.0 SP2 - http://www.altera.com/
2. Altera Quartus II Design Software version 4.0
   (used after design completion to test performance on new Stratix-II FPGAs)
3. WinTim32 Table Driven Assembler
4. General-purpose PC running Windows XP with over 512MB RAM

Chapter 2

Wavelet Image Compression

As previously mentioned, the design application for this vector processor is real-time wavelet video compression; specifically, the transform and inverse transform stages. In this chapter, the motivations and mechanics of wavelet video compression is described. It is noted up-front that video compression in this chapter is limited solely to compressing independent images in sequence (i.e. "2-D compression"). The more advanced method of examining and exploiting redundancies within a group of successive video frames, in what is referred to as 3-D or temporal compression, is not examined for this targeted application.

2.1 Overview

Image (and by extension video) compression attempts to reduce the number of bits required to digitally represent an image while maintaining its perceived visual quality. The field is classified into two main categories: lossless and lossy compression. Lossless compression, also referred to as entropy coding, ensures that an exact reproduction of the original image can be obtained after decompression, with the drawback of only a minimal to moderate reduction in file size. Lossy compression, however, achieves a much smaller file size by only ensuring that a "close" reproduction will be available after decompression. In this thesis, only lossy compression is discussed.

Image compression exploits two kinds of redundancies in images: spatial and spectral [Subramanya]. Spatial redundancy is correlation between adjacent image pixels, while spectral redundancy is correlation between different color planes in the image. These color planes could either be the three primary colors (R,G,B), or the luminance and chrominance (Y,U,V) components. In addition to exploiting spatial and spectral redundancies, further data can often be discarded based on careful analysis of what parts

of an image (luminance, hue, saturation, etc.) the human eye is most and least sensitive to.

Of the many processes available for image compression, two of the most popular transformations are the Discrete Cosine Transform (DCT) used in the common JPEG format, and the Discrete Wavelet Transform (DWT) used in the newer JPEG 2000 format. The DWT differs from the traditional DCT in several fundamental ways. The DCT operates by splitting the image into 8x8 blocks that are transformed independently [Santa-Cruz]. Through this transformation process, the energy compaction property of the DCT ensures that the energy of the original data is concentrated in only a few of the transformed coefficients, which are used for further quantization and encoding [Subramanya]. It is the discarding of the lower-energy coefficients that results in image compression and the subsequent loss of image quality. Unfortunately, the rigid 8x8 block nature of the DCT makes it particularly susceptible to introducing compression artifacts (extraneous noise) around sharp edges in an image. This is the "halo effect" seen in over-compressed web images. Because the artifacts become more pronounced at higher compression ratios, JPEG's suitability for line drawings and cartoon-like images is significantly impaired.

In contrast to the DCT, the DWT operates over the entire image, eliminating artifacts like those caused by the 8x8 DCT blocking. Like the DCT, the fundamental wavelet transform is completely reversible, meaning that if the forward and reverse transforms are applied in sequence, the resulting data will be identical to the original. In addition, the DWT is based on subband coding where the image is analyzed and filtered to produce image components at different frequency subbands [Welstead]. This produces significant energy compaction that is later exploited in the compression process. The wavelet's two-dimensional nature results in the image visually being divided into quarters with each pass through the wavelet transformation. A key effect of this transformation is that all of the highpass quadrants in the image contain essentially equivalent data [Topiwala]. Because of this homogenization, quantization and encoding can be applied to each subband independently, allowing for greater optimization in later compression stages for different frequency information. It is because of these

optimizations in compression effectiveness and improved image quality that the new JPEG 2000 standard utilizes wavelet transforms instead of cosine transforms.

A wavelet video compression, transmission, and decompression process that represents the target application of this vector processor is shown in Figure 1.

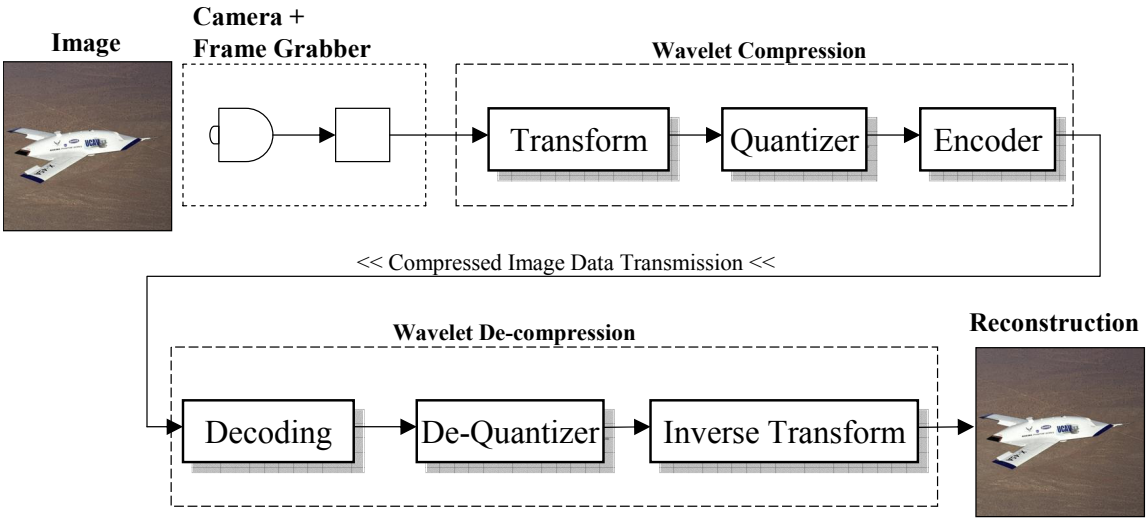

Figure 1: Wavelet Compression, Transmission, and Decompression Process

In this process, a single image or video frame is digitized by a camera and frame grabber. This image is then fed to a wavelet compression system. First, the compression system performs a wavelet transform of the image. This mathematical transform, which is perfectly reversible, converts the original image into a form suitable for encoding through a reversible spatial frequency separation. Next, the transformed image is quantized. This creates redundancy in the transformed image by reducing the number of allowable pixel values and thus the number of color or chrominance levels. Quantization is not reversible and is the principle cause of data loss in lossy compression. (The other cause of data loss can be attributed to floating-point rounding errors when calculating forward and inverse transformations with non-integer wavelets). The resulting bitstream from these two stages contain large blocks of redundant data that the encoder can easily locate and mathematically remove. The encoder is the only step of the process that actually achieves compression, which is does by removing redundancy present in the image. It is the

function of the transform and quantization stages to create this redundancy for the encoder to exploit.

At this point, a single image or video frame has been compressed into a bitstream that is some appreciable fraction of its original size, and is ready to be stored or transmitted to the destination system. Once at the destination, the process is reversed, whereby the image is first decoded, dequantized, and then inversed transformed to arrive at what is hopefully a convincing reproduction of the original image.

2.2 Wavelet Transform

In this section, the wavelet transform, which was briefly presented above, will be examined in further detail that includes the specific matrix implementation method used in the vector processor.

In the field of wavelets, the modified Haar Wavelet (referred to as Haar*) is traditionally used for rudimentary image compression because of its algorithmic simplicity and low computational complexity due to an integer-based design [Villasenor]. This transformation is given by

$$h_n = \begin{cases} \frac{1}{2} & \text{n} = 0,1 \\ 0 & \text{otherwise} \end{cases} \qquad (1)$$

$$g_n = \begin{cases} \frac{1}{2} & \text{n} = 0 \\ -\frac{1}{2} & \text{n} = 1 \\ 0 & \text{otherwise} \end{cases}$$

where h_n is the scaling function and g_n is the Haar* wavelet. The transformation is shown in Figure 2.

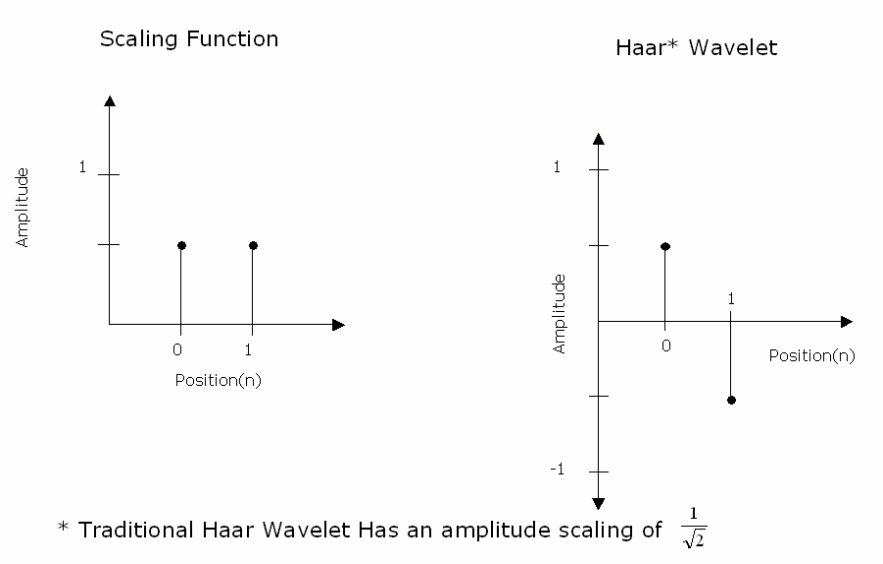

Figure 2: Discrete Haar* for Multiresolution Analysis

The application of the Haar* wavelet to a sample image is shown in Figure 3 to illustrate its algorithmic simplicity.

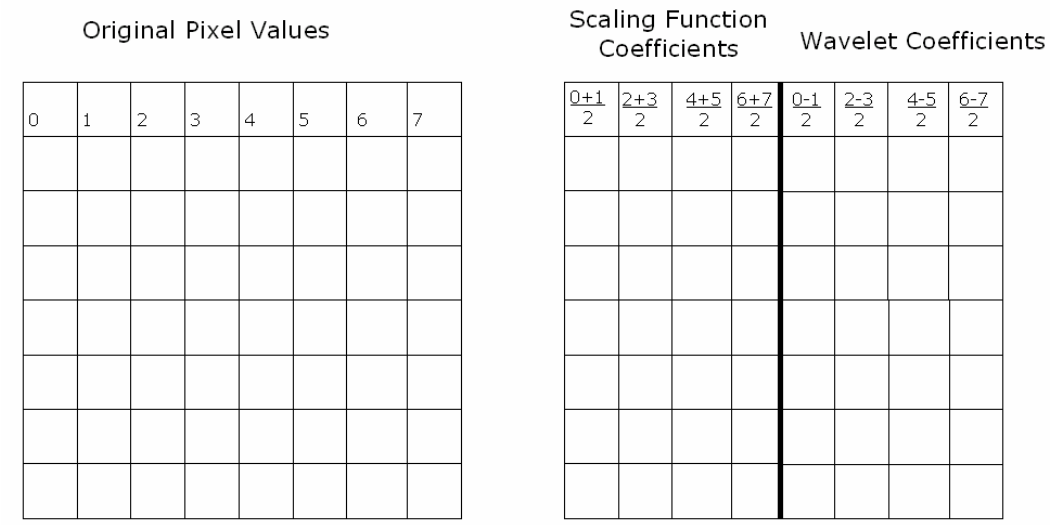

Figure 3: Application of Haar* Wavelet to Sample Data [Balster]

In Figure 3, the wavelet was only applied in a horizontal (row-oriented) fashion, and through its filtering process produced the scaling function coefficients (low-frequency) and wavelet coefficients (high-frequency) shown. From the application of the Haar* wavelet, it is evident that the scaling function coefficient is simply the average of two consecutive pixel values, while the corresponding wavelet coefficient is the difference

between the same two pixel values. The visual effect of this process is shown in Figure 4, which also goes one step further and applies the same transformation vertically (column-oriented).

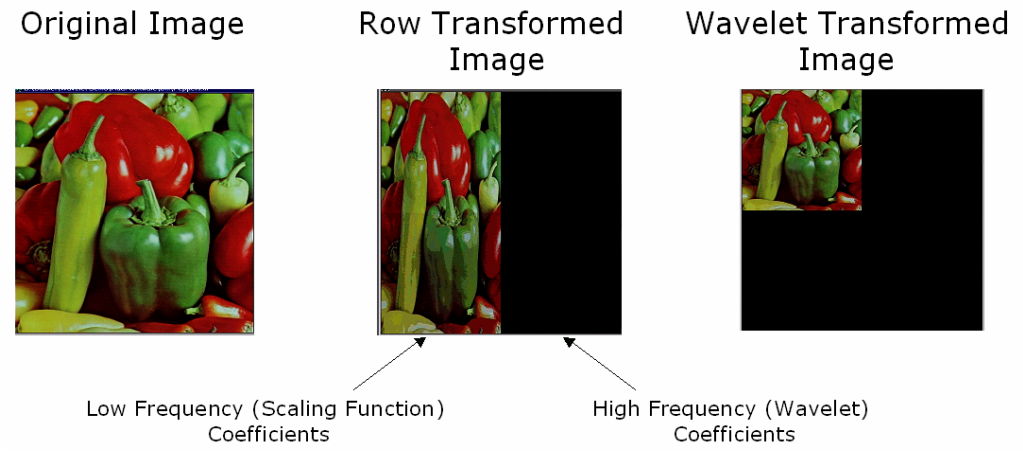

Figure 4: Successive Filtering of Imaging by Frequency [Balster]

As shown in Figure 4, the scaling function coefficients appear to contain all the image data, while the wavelet coefficients appear to be zero (black). If, however, the raw data was examined, it would be evident that the coefficients are only mostly zero, an effect shown later in Figure 6. In reality, the wavelet coefficients contain the difference between adjacent pixel values, which is the high-frequency edge information. Because the high-frequency coefficients approach zero, the encoder is more easily able to remove redundant information from the image.

Once the full wavelet transformation has been applied to two dimensions, as shown by the right-most image in Figure 4, the process can be repeated again by filtering only the low-frequencies quadrant of the image. This low frequency quadrant is the visible image in the rightmost frame of Figure 4. By repeating the filtering process several times over ever-shrinking low-frequency quadrants, the *multiresolution* aspect of the wavelet transform comes into effect. This is shown in Figure 5, which shows the frequency results of two passes of the wavelet transform.

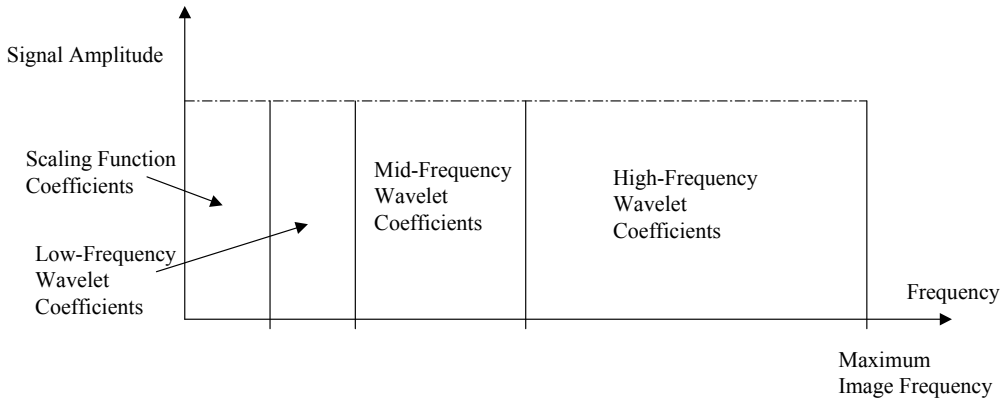

Figure 5: Frequency Partitioning with Several Frequency Sub-bands [Balster]

Visually, the image is partitioned into smaller and smaller blocks, each containing either low-frequency color information or high-frequency edge information, as shown in Figure 6.

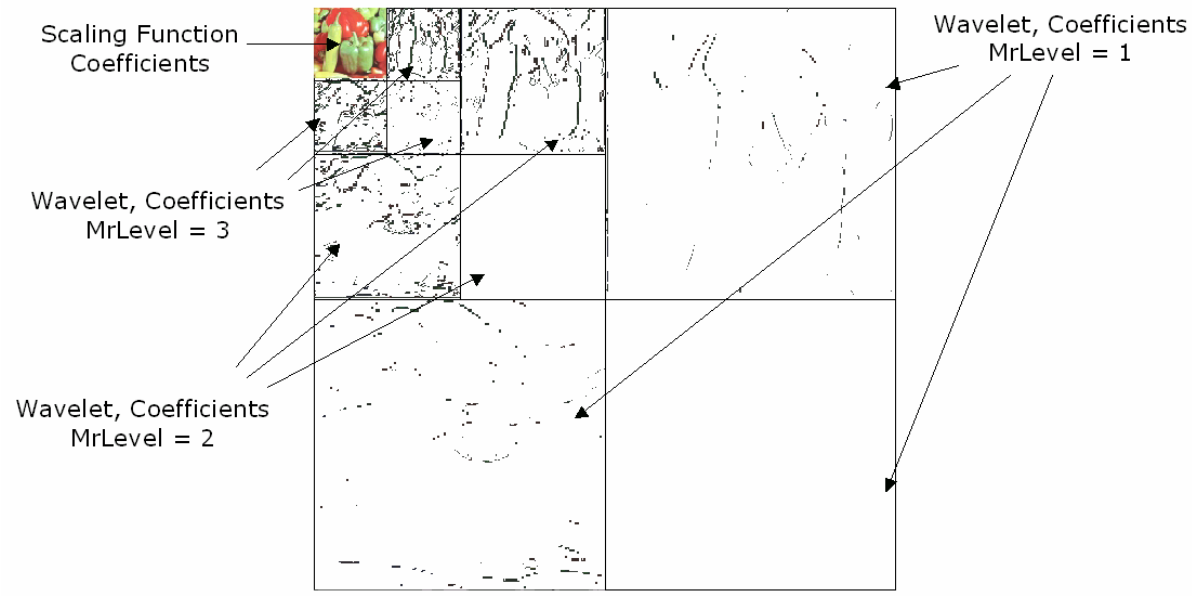

Figure 6: Multi-Resolution (MR) Levels in Wavelet Compression

Because the multiresolution wavelet transform cycles by repeatedly processing the low-frequency information, some of the image data is processed more than once. Because of this, the lower frequency wavelet coefficients are transformed by a wavelet of differing amplitude and duration than the higher frequency wavelet coefficients. Thus,

each sub-band shown in Figure 5 was generated by a different wavelet function. Figure 7 gives the different coefficient sub-bands with their corresponding wavelet function.

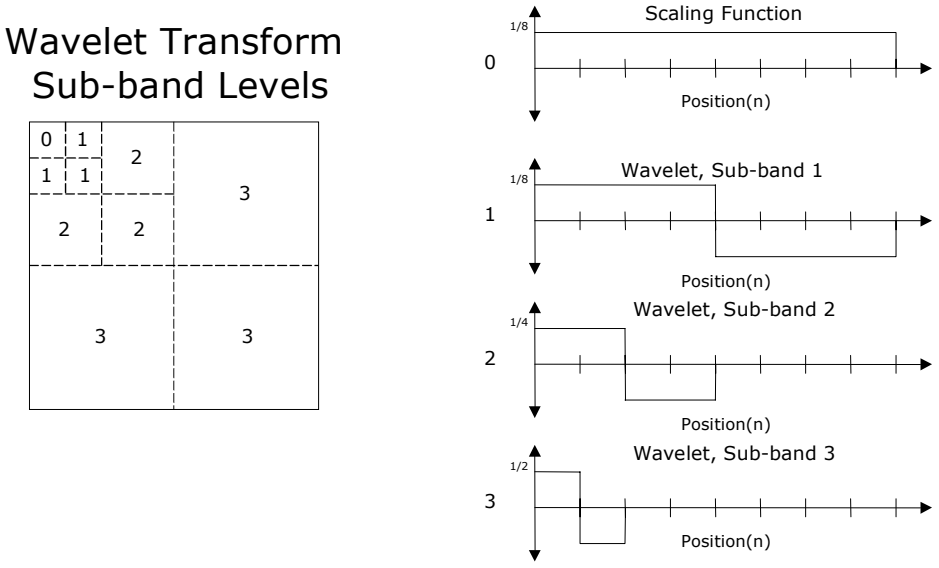

Figure 7: Wavelet Transforms Sub-Bands and their Corresponding Levels [Balster]

Thus, the wavelet transform becomes more selective at isolating low frequency components at each multiresolution stage of the transform.

In addition to the fundamental Haar wavelet transform and its derivatives, other more computationally complex transforms are available. They are designed to do a more efficient job at filtering low and high frequency image components, and suffer from much less pronounced blocking artifacts. For example, the Two-Six (TS) Wavelet adds additional filter taps for higher performance, as shown in Figure 8 [Villasenor].

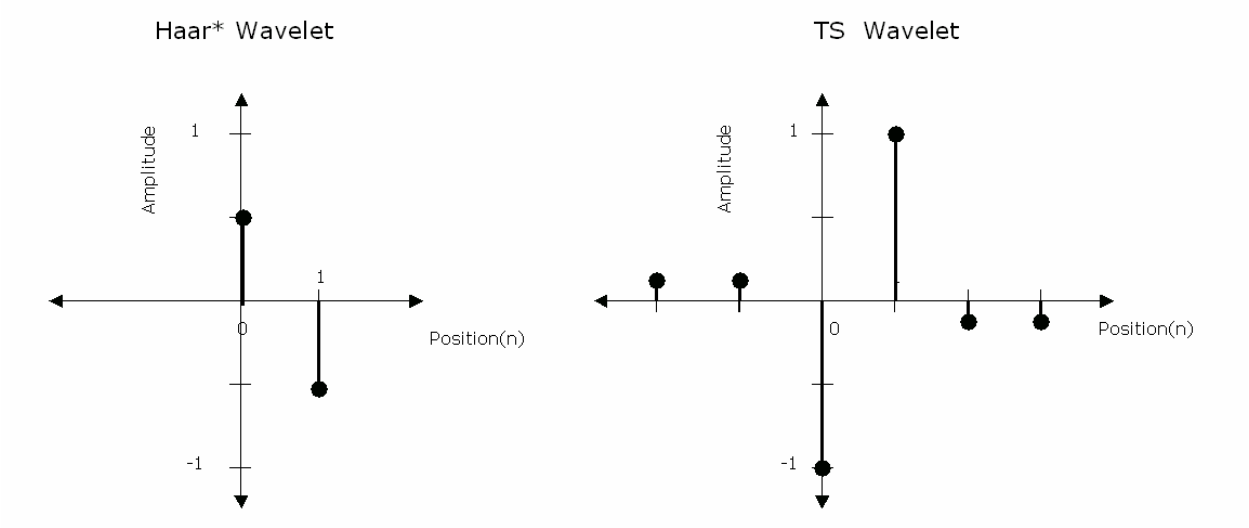

Figure 8: Haar* and TS ("Two-Six") Wavelet Filter

The left-most and right-most taps of the TS wavelet are at +1/8 and –1/8 respectively, and their power-of-two nature allows them to be replicated in a hardware environment via simple bit shifts. It is noted that the addition of two extra taps translates to a significant increase in the number of pixels that must be processed simultaneously to calculate each transformed coefficient, and thus system designers need to balance the width of the wavelet versus their own computational capabilities.

One well-known wavelet family was first identified by Ingrid Daubechies. It is this family, particular the D4 variant, which is integerized and used in the vector wavelet hardware system that will be subsequently discussed. Some examples of the Daubechies wavelet family are shown in Table 1.

Table 1: Daubechies Orthogonal Wavelets

Name	**Coefficients**
D4	Scaling: -0.1294 0.2241 0.8365 0.4830 Wavelet: -0.4830 0.8365 -0.2241 -0.1294
D6	Scaling: 0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327 Wavelet: -0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352
D8	Scaling: -0.0106 0.0329 0.0308 -0.1870 -0.0280 0.6309 0.7148 0.2304 Wavelet: -0.2304 0.7148 -0.6309 -0.0280 0.1870 0.0308 -0.0329 -0.0106

The floating-point Daubechies wavelets shown in Table 1 can be calculated using integer arithmetic if desired for ease of computation. It is simply a matter of picking a suitable multiplication factor, say, 2^6, multiplying that by each coefficient, and rounding the result to the nearest integer. The result of this process is shown in Table 2.

Table 2: Daubechies Wavelets – Integerized Versions

Name	Coefficients
D4	Scaling: -8 14 54 31 Wavelet: -31 54 -14 -8
D6	Scaling: 2 -5 -9 29 52 21 Wavelet: -21 52 -29 -9 5 2
D8	Scaling: -1 2 2 -12 -2 40 46 15 Wavelet: -15 46 -40 -2 12 2 -2 -1

Then, when computing the wavelet transform, each resulting intermediary and final pixel value should be divided by the same factor. This particular factor can be chosen for easy hardware implementation as a division by 2^6, for example, can be accomplished by shifting right by 6 binary positions. Thus, there is essentially zero overhead to integerizing the wavelets, as the original multiplication of the wavelet coefficients can be computed a` priori, and the subsequent divisions at each stage are handled by shifting the results while in transit to their final destination. While this method does entail small rounding errors, they tend to cancel each other out in the wavelet transform, and thus the final distortion is at least an order of magnitude less than the distortion deliberately introduced into lossy compress systems via the quantization stage.

One method of calculating wavelet transformations is through a sequence of matrix multiplications. Characterizing the transformation algorithms as matrix manipulations has two key advantages. First, the transform (and reconstruction) matrices may be computed a` priori, reducing the amount of computation required for each image. Second, the matrix algorithms become identical and depend only upon the values in the transform matrix and the image matrix. Both of these advantages are highly significant for hardware implementations. A gray-scale image can be represented by a matrix $\mathbf{I_m}$ where each element is a discrete pixel value. Color images simply consist of a pixel

matrix for each color plane. The transform of the image is accomplished by constructing a transform matrix, **W**, and forming the transformed image $\mathbf{TI_m}$ by the operation

$$\mathbf{TI_m} = \mathbf{WI_m W^T} \tag{2}$$

where $\mathbf{W^T}$ is the transpose of **W**. This operation performs the two-dimensional transformation of the image matrix [Goswami]. The inverse transformation, which maps from the transformed image back to the reconstructed image $\mathbf{RI_m}$, is simply

$$\mathbf{RI_m} = \mathbf{W^T TI_m W}. \tag{3}$$

There are restrictions in place on the above transformation equations. The transform matrix **W** must be orthogonal. That is, its transpose must be its inverse, or

$$\mathbf{WW^T = W^T W = I}. \tag{4}$$

Bi-orthogonal wavelets such as the Bio9/7 and the Interger 5/3 can be used if the analysis and synthesis transformation matrices are computed independently, and are not simply the transpose of each other. The application of this class of wavelets is covered in more depth in [Qiang].

Obtaining the transform matrix from the original wavelet coefficients is a straightforward process. For example, consider the Daubechies 4 wavelet which has 8 coefficients (4 scaling and 4 wavelet). Assume the scaling coefficients are numbered C_0 through C_3 and the wavelet coefficients C_4 through C_7. Then, the transform matrix for an n x n image is

$$W = \begin{bmatrix} C_0 & C_1 & C_2 & C_3 & \dots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & C_0 & C_1 & C_2 & C_3 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & C_0 & C_1 & C_2 & C_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ C_2 & C_3 & 0 & 0 & 0 & 0 & 0 & \dots & C_0 & C_1 \\ C_4 & C_5 & C_6 & C_7 & \dots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & C_4 & C_5 & C_6 & C_7 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & C_4 & C_5 & C_6 & C_7 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ C_6 & C_7 & 0 & 0 & 0 & 0 & 0 & \dots & C_4 & C_5 \end{bmatrix} \tag{5}$$

Multiresolution filtering can be performed by repeating the matrix multiplication with the upper-left quadrant of the image and an appropriately scaled version of the same transform matrix. Thus, each successive resolution level operates on 25% of the previous intermediate image and requires 25% of the computations.

While the computational requirements of performing 5 or 6 multiresolution levels are not prohibitive, and in fact asymptotically decreases as the levels increase, it would still be preferable if all levels could be computed in a single step. It is precisely this motivation that led to the introduction of the one-step wavelet transform by Bo Qiang. In this method, an alternate transformation matrix $\hat{\mathbf{W}}$ is computed by

$$\hat{\mathbf{W}} = \begin{bmatrix} \mathbf{W}_{N/2^{i-1}} & & & \\ & \mathbf{I}_{N/2^{i-1}} & & \\ & & \dots & \\ & & & \mathbf{I}_{N/2^{i-1}} \end{bmatrix} * \dots * \begin{bmatrix} \mathbf{W}_{N/4} & & \\ & \mathbf{I}_{N/4} & \\ & & \mathbf{I}_{N/4} \end{bmatrix} * \begin{bmatrix} \mathbf{W}_{N/2} & \\ & \mathbf{I}_{N/2} \end{bmatrix} * \mathbf{W}_N . \tag{6}$$

What the one-step method does, from left to right, is multiply all the transformation matrices at every level together, from the highest level N at the left to the lowest level N at the right. Because the transformation matrix size decreases (by 75%) at each resolution level, all the matrices except the first (on the right) are padded with the identity matrix to allow the multiplication to proceed.

This method does produce a slightly different arrangement of subbands, as shown in Figure 9.

Figure 9: Traditional Transform versus One-Step Method [Qiang]

Bo Qiang showed that this difference demonstrates that the one-step method further analyzes the higher frequency subbands of the image, resulting in lower entropy in the transformed image, and thus potential for higher compression ratios [Qiang].

2.3 Quantization / Thresholding

The quantizer stage of the image compression process works by performing a thresholding operation on the transformed image. Fundamentally, it performs a mapping from a continuously-parameterized set V to a discrete set X [Topiwala]. All values within a fixed range are set to a single pre-determined value. Values outside of a pre-determined maximum and minimum boundary are set to the respective boundary value. The effect of this operation is that the resulting bitstream has significantly longer runs of the equivalent coefficients for the encoder to exploit. The price of this gain in compression ratio is the loss of image richness as neighboring color values are merged into a single intensity. This generic thesholding process of quantization is shown in Figure 10.

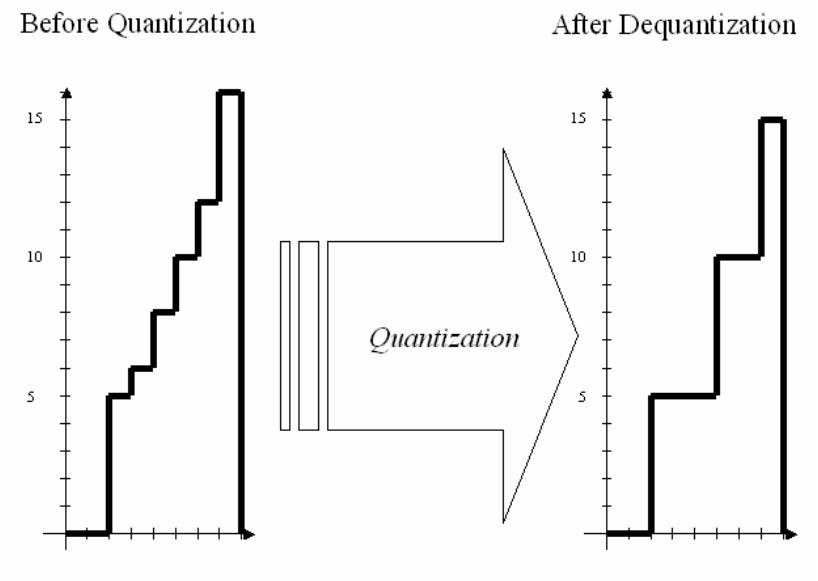

Figure 10: Generic Thresholding and Quantization

Note that the quantizer method in Figure 10 has features a constant step size, which insures that all color components "suffer" equally through the quantizer. Other methods use a non-constant step size, with smaller steps being reserved for "key" regions of the image (to preserve the highest quality), and larger steps for extraneous image regions or color components.

As mentioned previously, one of the key benefits of the wavelet transform is how its multiresolution property allows for greater optimization during compression. To this end, each of the three multiresolution (MR) levels in Figure 6 can be quantized with different step sizes and upper and lower boundaries. In practice, this results in the first MR level having the largest step size and smallest upper and lower boundaries because the wavelet transform has already reduced much of the high frequency information to zeros. In contrast, the third MR level has the smallest step size and widest upper and lower boundaries because much of the low-frequency information is still intact in those quadrants. Finally, the last remaining quadrant (in the upper left-hand corner) with all of the remaining scaling function coefficients may not quantized at all to preserve image quality. An example of this selective quantization approach is shown in Figure 11.

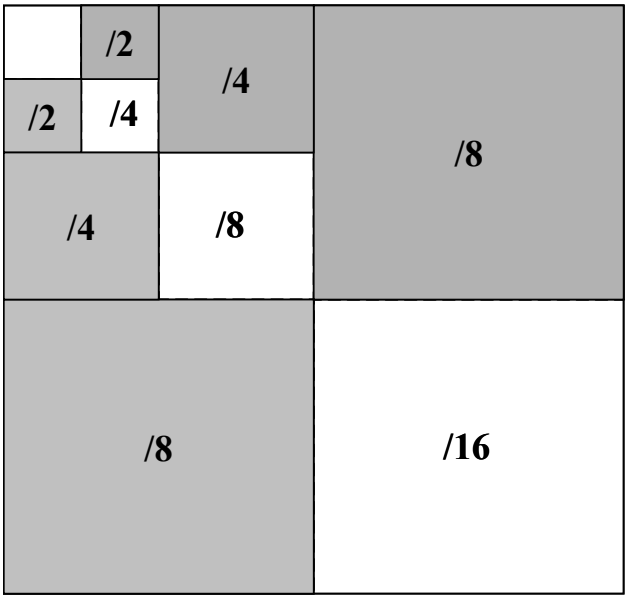

Figure 11: Variable Quantization by Sub-Bands

Here, subbands are either quantized by 2, 4, 8, or 16. What this means in practice is that an integer division operation (i.e. bit shift) is performed, and the least-significant bits of the image pixels are simply discarded. This provides the desired level of quantization.

2.4 Encoders

The encoder is the only stage of the process that actually achieves compression. It does this by removing redundancy present in the image. Two common algorithms that are used in conjunction with wavelet image compression are Stack-Run and Huffman encoders.

Stack-run encoders use raster (i.e. sequential) scanning within subbands to reduce the number of bits required to represent the image. They represent the uncompressed bitstream as a sequence of pairs (a[n],b[n]) where a[n] is the number of zero-valued coefficients before the non-zero-valued coefficient with value b[n]. These binary values are mapped into 4-ary arithmetic encoder which has an alphabet of 4 symbols specifically designed to eliminate both ambiguity and redundant information [Tsai]. This algorithm is effective in reducing long streams of zeros, which the wavelet transform and quantization stages commonly produce.

Following the application of Stack-Run Encoding, traditional image compression processes often employ Huffman coding, also known as entropy encoding. This process, through the systematic use of binary trees, attempts compress the bit stream by assigning shorter bit patterns for the most common data elements, and longer bit patterns for the less frequent data elements.

Encoders are not implemented as part of this thesis, because their algorithms do not contain significant vector components that a vector processor could exploit. Rather, the encoders would be more effectively implemented as a dedicated module that processes the transformed image calculated by the vector processor.

2.5 Wavelet Transformation in Matlab

The Matlab program shown in this section performs a 3 level multiresolution wavelet transform followed by an inverse transformation. It is the functionality shown in this program that the vector processor designed in this thesis attempts to emulate. Not included in this Matlab program are any quantization or encoding modules, which would not be executed by a vector processor in any case. Those modules are available, however, as part of a larger video compression application from which this wavelet transform was derived. This Rapid Prototyping Interface (RPI) was developed at the University of Dayton to test new wavelet compression technologies. It is available for download from http://quickplace.udayton.edu/crc.

Matlab Wavelet Transform (Forward and Inverse)

```
close all; clear all; clc;

MrLevel = 3;
filename = '01_256x256.bmp';

original_image = double(imread(filename));

% Image must be true-color (3 planes).  Save each plane.
pic.a = original_image(:,:,1);                % Save color planes to separate
pic.b = original_image(:,:,2);                % arrays in structured variable
pic.c = original_image(:,:,3);                % so each can be different size.

% Show original image
image(uint8(original_image));
```

```
set(gca,'Position',[0 0 1 1])
axis off

% In this simple program, we assume all color planes are the same size.
% This may not be the case if fancy downsampling is used in YUV colorspace
% Set Max to the maximum dimension (length or width)
single_plane = pic.a;
Max = max(size(single_plane));

% The DAUB4 Transform
h = [-0.1294 0.2241 0.8365 0.4830;
       -0.4830 0.8365 -0.2241 -0.1294];

% Initializing the Transform Matrix
% Note: Wrap-Around Coefficients are added later in the program
% Initializing transform matrix
T = zeros(Max+size(h,2)-2,Max);

% Constructing Scalar half of Transform Matrix
scalar = [h(1,:) zeros(1,Max)];
scalar_matrix = repmat(scalar,1,Max/2);
T(:,1:Max/2) = reshape(scalar_matrix(1:(Max+size(h,2)-2)*Max/2),Max+size(h,2)-
2,Max/2);

% Constructing Wavelet Half of Transform Matrix
wavelet = [h(2,:) zeros(1,Max)];
wavelet_matrix = repmat(wavelet,1,Max/2);
T(:,Max/2+1:Max) = reshape(wavelet_matrix(1:(Max+size(h,2)-
2)*Max/2),Max+size(h,2)-2,Max/2);

% Constructing Transform Matrix at MrLevel = 1
SpecificMrLevel = 1;
x = 2^(SpecificMrLevel-1);
T_1 =
cat(2,T(1:Max/x,1:Max/2^SpecificMrLevel),T(1:Max/x,Max/2+1:Max/2+Max/2^Specific
MrLevel));
T_1(1:size(h,2)-2,Max/2^SpecificMrLevel+1-(size(h,2)-
2)/2:Max/2^SpecificMrLevel) = T(Max/x+1:Max/x+size(h,2)-
2,Max/2^SpecificMrLevel-(size(h,2)-2)/2+1:Max/2^SpecificMrLevel);
T_1(1:size(h,2)-2,Max/x+1-(size(h,2)-2)/2:Max/x) = T(Max/x+1:Max/x+size(h,2)-
2,Max/2+Max/2^SpecificMrLevel-(size(h,2)-2)/2+1:Max/2+Max/2^SpecificMrLevel);

% Constructing Transform Matrix at MrLevel = 2
SpecificMrLevel = 2;
x = 2^(SpecificMrLevel-1);
T_2 =
cat(2,T(1:Max/x,1:Max/2^SpecificMrLevel),T(1:Max/x,Max/2+1:Max/2+Max/2^Specific
MrLevel));
T_2(1:size(h,2)-2,Max/2^SpecificMrLevel+1-(size(h,2)-
2)/2:Max/2^SpecificMrLevel) = T(Max/x+1:Max/x+size(h,2)-
2,Max/2^SpecificMrLevel-(size(h,2)-2)/2+1:Max/2^SpecificMrLevel);
T_2(1:size(h,2)-2,Max/x+1-(size(h,2)-2)/2:Max/x) = T(Max/x+1:Max/x+size(h,2)-
2,Max/2+Max/2^SpecificMrLevel-(size(h,2)-2)/2+1:Max/2+Max/2^SpecificMrLevel);

% Constructing Transform Matrix at MrLevel = 3
SpecificMrLevel = 3;
x = 2^(SpecificMrLevel-1);
T_3 =
cat(2,T(1:Max/x,1:Max/2^SpecificMrLevel),T(1:Max/x,Max/2+1:Max/2+Max/2^Specific
MrLevel));
T_3(1:size(h,2)-2,Max/2^SpecificMrLevel+1-(size(h,2)-
2)/2:Max/2^SpecificMrLevel) = T(Max/x+1:Max/x+size(h,2)-
2,Max/2^SpecificMrLevel-(size(h,2)-2)/2+1:Max/2^SpecificMrLevel);
```

```
T_3(1:size(h,2)-2,Max/x+1-(size(h,2)-2)/2:Max/x) = T(Max/x+1:Max/x+size(h,2)-
2,Max/2+Max/2^SpecificMrLevel-(size(h,2)-2)/2+1:Max/2+Max/2^SpecificMrLevel);


% Transform image

% Pic is a structured array with multiple color planes labeled
% 'a', 'b', 'c', etc...  Each transform function below
% is only able to handle a single color plane at a time. Thus,
% call each function repeatedly for all planes.
for color_plane = 1:3

       % Convert these numbers to ASCII sequence a,b,c...
       % ('a' is ASCII 97)
       index = char(96 + color_plane);

       % Grab just that single color plane and isolate it
       single_plane = pic.(index);      % Iterate through pic.a, pic.b, ...
                                        % using dynamic field names
       % zero-pad image
       [row,col,plane]=size(single_plane);
       if(row>=col)
              transform = zeros(row,row,plane);
              Max = row;
       else
              transform = zeros(col,col,plane);
              Max = col;
       end

       % Implementing the Transform
       transform = single_plane;

       for i = 1:MrLevel
              x = 2^(i-1);

              % Choose transform matrix for current MrLevel
              switch(i)
                     case 1
                            T_x = T_1;
                     case 2
                            T_x = T_2;
                     case 3
                            T_x = T_3;
              end

           transform(1:Max/x,1:Max/x) = T_x' * transform(1:Max/x,1:Max/x) * T_x;
       end

       single_plane = transform;

       % Put that single color plane back into the structured array 'pic'
       pic.(index) = single_plane;      % Iterate through pic.a, pic.b, ...
                                        % using dynamic field names
end

% Show transformed image
figure;
transformed_image(:,:,1) = pic.a;
transformed_image(:,:,2) = pic.b;
transformed_image(:,:,3) = pic.c;
image(uint8(transformed_image));
set(gca,'Position',[0 0 1 1])
axis off
```

```
% Inverse transform image

% Pic is a structured array with multiple color planes labeled
% 'a', 'b', 'c', etc...  Each transform function below
% is only able to handle a single color plane at a time. Thus,
% call each function repeatedly for all planes.
for color_plane = 1:3

       % Convert these numbers to ASCII sequence a,b,c...
       % ('a' is ASCII 97)
       index = char(96 + color_plane);

       % Grab just that single color plane and isolate it
       single_plane = pic.(index);             % Iterate through pic.a, pic.b,
                                               % ... using dynamic field names
       transform = single_plane;

       for i = MrLevel:-1:1
              x = 2^(i-1);

              % Choose transform matrix for current MrLevel
              switch(i)
                     case 1
                            T_x = T_1;
                     case 2
                            T_x = T_2;
                     case 3
                            T_x = T_3;
              end

           transform(1:Max/x,1:Max/x) = T_x * transform(1:Max/x,1:Max/x) * T_x';
       end

       single_plane = transform;

       % Put that single color plane back into the structured array 'pic'
       pic.(index) = single_plane;      % Iterate through pic.a, pic.b, ...
                                        %  using dynamic field names

end


% Show inverse transformed image
figure;
itransformed_image(:,:,1) = pic.a;
itransformed_image(:,:,2) = pic.b;
itransformed_image(:,:,3) = pic.c;
image(uint8(itransformed_image));
set(gca,'Position',[0 0 1 1])
axis off
```

When executed, the above program produces three images. The first, shown in Figure 12, is simply the original image before any processing has been performed.

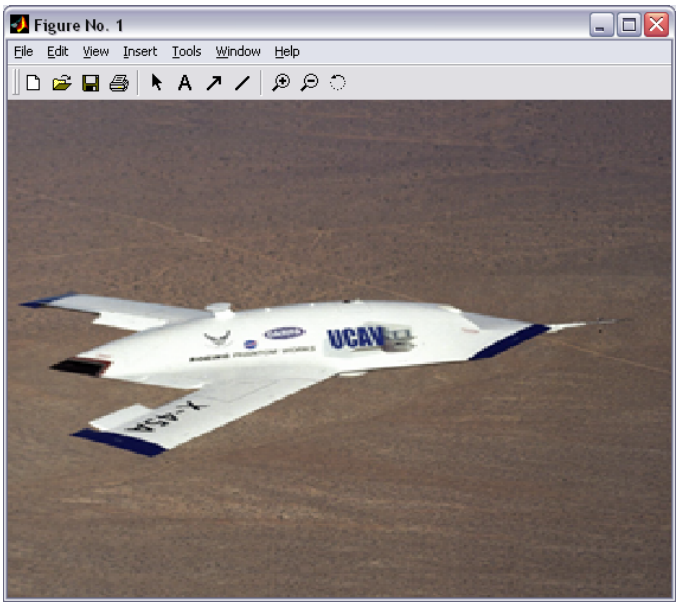

Figure 12: Original Image (256 x 256 pixels)

The second image, shown in Figure 13, is the result after the forward wavelet transform. Clearly visible are the three multiresolution levels that were performed in the transform stage.

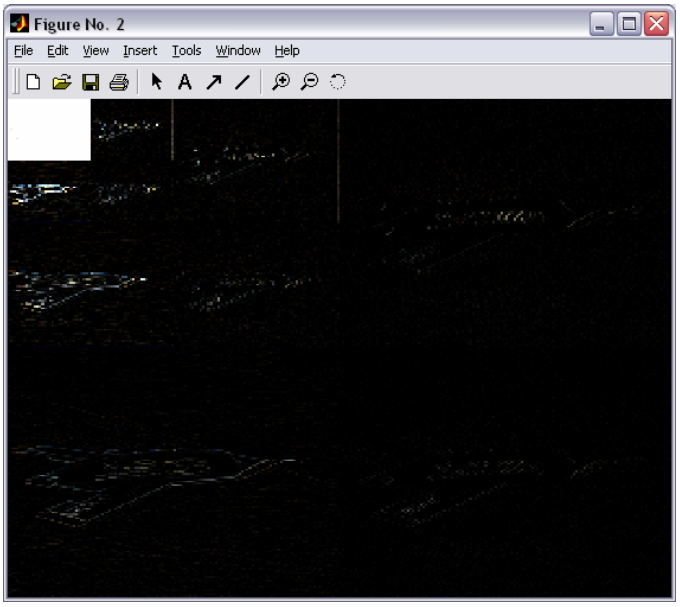

Figure 13: Transformed Image (3 Multiresolution Levels)

Finally, Figure 14 shows the result after the inverse wavelet transformation was computed. This figure should match the original image since the wavelet transform is reversible (except perhaps for some slight floating point arithmetic rounding errors). Remember that the lossy quantization stage was not performed in this sample application.

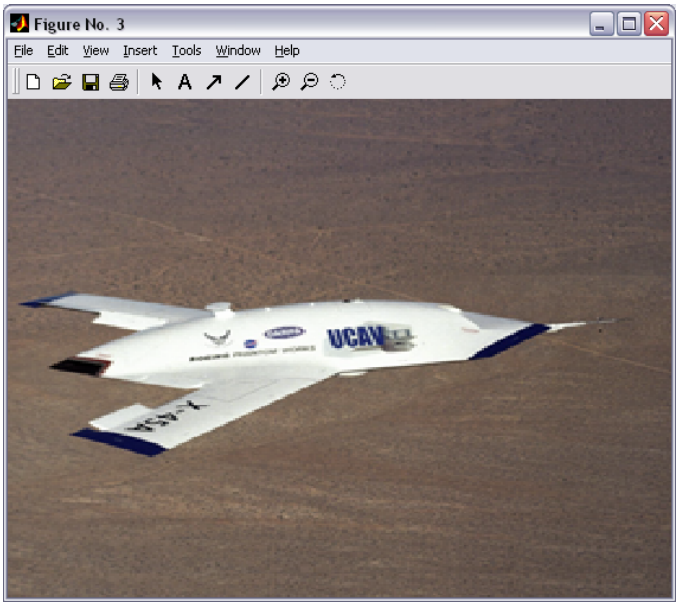

Figure 14: Final Image after Inverse Transformation

This Matlab program is analyzed in a future chapter to determine the appropriate functionality that must be provided by the vector processor to perform a complete wavelet transform.

Chapter 3

Generic Vector Processors

3.1 Overview

General purpose microprocessors were originally designed to perform scalar arithmetic operations on at most two values; e.g. A+B. However, it quickly became apparent that this processing paradigm could be enhanced to support vectors or linear arrays of data. A single vector instruction could accomplish the same function as multiple scalar instructions, increasing the code density. Mathematically, $\vec{A} + \vec{B} = \vec{C}$ is represented as $[VOP][V3][V2][V1]$, which is executed as $V3 := [V1][VOP][V2]$ [Flynn]. These vector processors are often referred to as SIMD, which stands for Single Instruction Stream, Multiple Data Stream.

This new design paradigm led to the development of a host of vector processing supercomputers such as the Cray series. These systems are commonly used for scientific applications with large datasets and complex data dependencies such as weather prediction, crash-test simulations, physical simulations, and weapons simulations [Kaxiras]. Because of these systems, common scientific languages such as Fortran 90 have built-in provisions for SIMD instructions [Jordan].

To program a vector processor, the program is either explicitly expressed as vectors of data, or implicitly contains loops whose data references in memory can be programmatically identified as vectors by the compiler. This requires the compiler or programmer to vectorize the code, which involves the transformation of loops of scalar operations into a sequence of vector instructions. Sophisticated vector processors also require compiler to identify independent calculations and generate appropriate code to utilize multiple independent hardware execution units [Flynn].

3.2 Vector Processing Advantages

There are several general advantages of using a vector processing paradigm instead of a scalar approach. First, vector instructions have significantly higher code density compared with a generic scalar system [Espasa]. Depending on the specific user program, this can significantly reduce the instruction count necessary to produce similar output [Flynn]. Although a single instruction fetch still yields a single (vector) instruction, this single instruction initiates a long vector operation. Thus, the bandwidth for instruction fetches is negligible in a vector processor compared with instruction fetch overheads in scalar designs, which are estimated to be 20-50% [Stone]. A single vector operation can represent tens or hundreds of arithmetic operations, and can keep multiple deeply-pipelined arithmetic units busy for lengthy time periods [Hennessy].

In a vector processing system, data is inherently organized in long continuous streams for highly efficient and convenient hardware processing. Further, a vector operation such as add or multiply inherently represents and removes a simple loop construct from the program, reducing "non-productive" overhead from the program execution. [Flynn]

When a programmer or compiler uses a vector instruction instead of a sequence of scalar operations, it signifies to the processor that the calculation of each vector element is independent from all other calculations in that vector. Thus, the processor only needs to check for data hazards between vector instructions, and not within a single instruction [Espasa]. While the complexity of this dependency checking logic is the similar to a scalar processor, the greater effective work produced by a single vector instruction results in a far lower control overhead for that architecture [Hennessy]. Similarly, because an entire loop can be replaced by a single vector operation, control hazards resulting from that loop are non-existent in a vector architecture. [Hennessy]

Further, because there are implicitly no data hazards within a vector instruction, the hardware is at liberty to use multiple parallel arithmetic units, a single deeply

pipelined unit, or some combination of the two. This imparts significant design flexibility and allows for easy creation of families of vector processors with a wide mix of low cost or high performance [Hennessy, Espasa].

There are other hardware design advantages to using a vector processor. For instance, they are typically able to efficiently utilize high memory bandwidth because of their efficient instruction format (one instruction per long stream of data) and because they can be designed with multiple deep arithmetic pipelines [Kaxiras]. Further, assuming a proper interleaving technique is used to distribute vector data elements across multiple memory modules, the high initial latency in accessing memory is effectively distributed across the entire length of the vector, in contrast to a single scalar operation [Hennessy].

Finally, vector processors can be designed to facilitate low power operation due to an inherent "localizing" property of vector computations [Espasa]. After a vector instruction has been initialized, only the applicable function units, registers, and data busses are required to sustain the operation. Other modules, such as the instruction fetch unit and reorder buffers are not needed until the next vector instruction, which may be dozens or hundreds of data calculations away. Thus, they can be shut down to save power.

3.3 Vector Processing Disadvantages

The advantages of vector processors do not come without several disadvantages as well. The most significant, in comparison to scalar machines, is that effective cache designs on vector processors are difficult to achieve for several reasons. First, a few vectors could completely fill a small-to-medium size cache, requiring caches on vector processors to be larger than their scalar cousins. Second, vector data often has poor temporal locality. This is because the data in large, long-running scientific applications is unlikely to be accessed again in the near-term, leading to a near-continuous cache misses and significant cache turnover [Flynn].

Perhaps the simplest method of dealing with the low cache effectiveness of vector processors is to decouple the processing unit from memory by using a vector register set. This register set bypasses the cache on loads/stores, leaving the cache to only store scalar data [Flynn].

Even assuming a large vector cache was provided in a vector processor, a second fundamental drawback remains. Effective vector processors operating on long vectors require high-bandwidth memory systems, which are expensive to design and implement [Flynn, Espasa]. Further, these systems are increasing difficult to achieve, as processor speed has increased far more rapidly than memory speeds, worsening the relative performance gap as time progresses [Gee].

3.4 Generic Vector Processor Architecture

In vector processors, as in scalar systems, an effective memory architecture is critical to the performance of the final system. For a SIMD system, two architecture design choices are the most important. First, how should the memory system be partitioned, so that data can be accessed in parallel? Second, once the data is partitioned, how will the bus structure route data to the multiple arithmetic units simultaneously? [Jordan] The second assumes, of course, that most vector processors will have multiple arithmetic units to accelerate performance.

There are two generalized architectures for vector processors, as in scalar machines. The first is a vector-memory architecture, where all operands are fetched directly from memory. The second is a vector-register architecture, which contains a high-speed local register set which is the source for all data with the exception of load and store operations.

The first generation of vector systems were designed with main memories with sufficient bandwidth to keep all computation units supplied with data. However, this became increasing difficult to do because of the processor-memory performance gap. Thus, designs in the past decade have consistently added a local high-speed memory bank

in the form of registers on-chip [Stone]. All commercials designs since the 1980's have been of this vector-register architecture, including the Cray Research systems, the Japanese supercomputers, and the Convex mini-supercomputers [Hennessy].

There are several major components of a typical vector processor system, including vector registers, scalar registers, arithmetic units, cache, and memory. These systems are linked with data paths as shown in Figure 15.

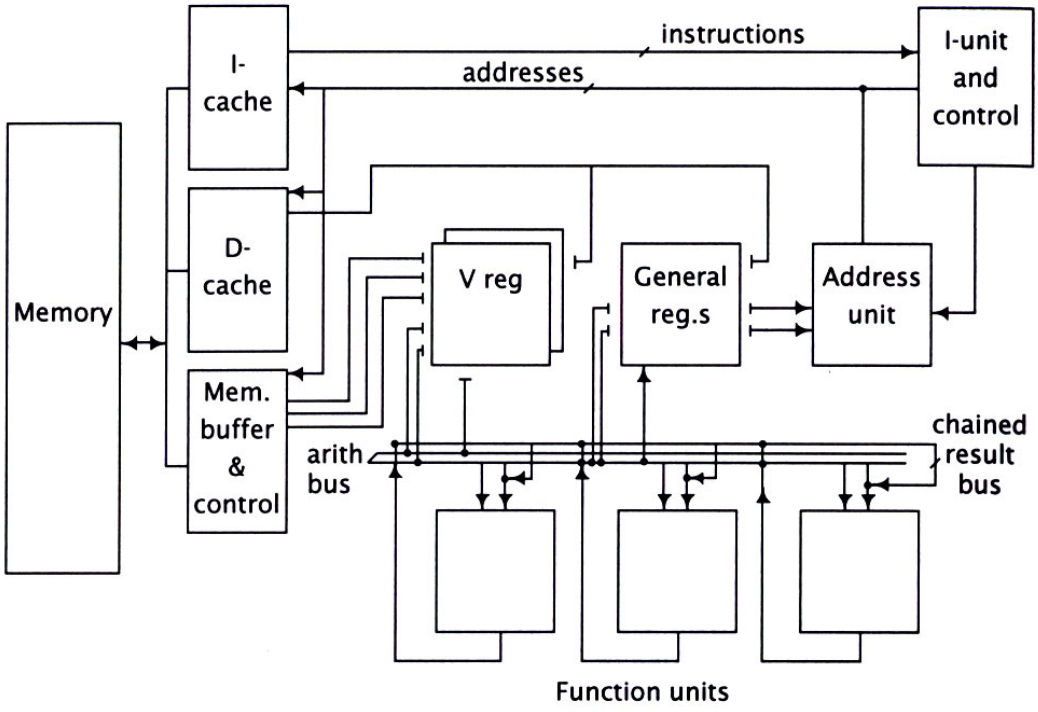

Figure 15: Major Data Paths in Generic Vector Processor [Flynn Figure 7.14]

Vector processors typically include a "standard" scalar unit, which could be an out-of-order or VLIW design [Hennessy]. A fast scalar unit is important to execute the control logic around the core vector instructions, which can often be quite substantial.

The vector register set of a generic processor usually stores at least 8 or more vectors, each containing 16-64 data elements. In high-end computational machines (which are the primary market for dedicated vector architectures), these data elements are typically a 64-bit floating point values [Flynn]. Like scalar registers, vector registers are manipulated using special load and store functions. In most designs, operations using a

register cannot proceed until prior loads (from memory) or stores (from the ALU) are complete [Flynn].

The system performance is necessarily limited by the number of register ports. Each concurrent load or store operation requires an input port, while ALU operations require at least one input and one output port. [Flynn]. An advanced vector processor may attempt to process several independent instructions concurrently, further increasing the number of register ports. To support these ports, designers often build vector registers from interleaved combinations of smaller registers.

The use of a vector register set in a vector-register architecture has one significant drawback compared to the vector-memory architecture. By using vector registers instead of directly accessing main memory, it becomes much more difficult to allow vectors to be of arbitrary length [Flynn]. Thus, there is typically some design tradeoff that allows for some resource inefficiencies (e.g. only fill up half of a vector in the register) in exchange for the faster access times of the register file.

The functional or arithmetic units of a vector processor are often similar or identical in design to those of their scalar cousin, with often the only difference being that several are used in parallel in the vector machine to allow for simultaneous calculations of multiple data elements. (Recall that the elimination of data hazards within a vector instruction is a key advantage of this architecture and thus should be taken advantage of to the maximum extent feasible.) These arithmetic units can calculate operations such as addition, subtraction, or multiplication across corresponding elements of two vectors. These functions are typically pipelined to produce an operation per cycle execution rate without being so deep as to lengthen overall execution time unnecessarily beyond perhaps 2-4 cycles [Flynn]. Division is typically an exception to this rule, and some architectures such as the Cray systems prefer to perform a reciprocal operation that can be more easily pipelined [Jordan]. This is because the reciprocal approach uses four shorter instructions in sequence to (1) approximate the reciprocal of an operand, (2) calculate a correction factor, (3) multiply reciprocal by the other operand, and (4) to multiply the result by the correction factor.

Logical operations such as comparisons, tests, shifts, negate, and the usual and/or/xor functions are typically included. These comparisons and tests are between a vector against either another vector or zero produce either a 1-bit logical (1 or 0) vector result, or alternatively a more compact and efficient scalar result where each logical bit corresponds to a vector element [Flynn].

Beyond the previously mentioned operations, which are common to both vector and scalar machines, vector processors add a new family of operations uniquely suited to their architecture. First are the Expand and Compress operations between a scalar and a vector. The expand operation takes a vector and a scalar bit map and zeros out entries in the vector that correspond to zeros in the scalar. This is shown below in Figure 16.

Register Values:	**Instruction:**	**Result:**
S1= 1010 V1= 1.1, 1.2, 1.3, 1.4.	VEXP V1, S1, V2	V2 = 1.1, 0, 1.2, 0

Figure 16: Expand Operation example

The compress operation also takes a vector and a scalar, but instead of zeroing out specific indices, it omits all entries in the vector that correspond to a zero in the scalar bit map. This is shown below in Figure 17.

Register Values:	**Instruction:**	**Result:**
S1 = 1010 V1 = 1.1, 1.2, 1.3, 1.4	VCPRS V1, S1, V2	V2 = 1.1, 1.3

Figure 17: Compress Operation example

Vector arithmetic units typically contain an accumulator. This is quite useful in the Vector Dot Product (Inner Product) operation, which is also referred to as Multiply & Accumulate. This "compound instruction" calculates $\sum_{i=1}^{n} V1.i * V2.i \rightarrow S$, which is recognized as the key component of a full-scale matrix multiplication [Flynn].

Vector operations on vectors of unequal length can be handled in one of several manners. The vector length could be specified in each instruction. Or, unused elements in fixed-length vectors could be required to be filled with zeros. Or, unused elements could be filled with NaN, and all calculations with a NaN input set to produce a NaN result, as is commonly done in the IEEE floating-point specification [Flynn].

A common term in vector processor arithmetic units is vector "strip mining." What this refers to is simply processing a lengthy vector in multiple shorter iterations. Thus, for example, if a processor had 4 parallel arithmetic units, a long vector would be processed 4 elements at a time [Jordan].

3.5 Advanced Vector Techniques

There are several advanced vector processing techniques used in modern microprocessors to improve performance. The first, pipelining of the arithmetic unit, should always be done for maximum efficiency [Flynn]. Pipelining allows arithmetic units to produce results at a much faster rate than their inherent latency allows them to produce a single result. Further, it allows slower memories to be coupled to a faster arithmetic unit [Stone]. The choice to pipeline a design assumes that instructions have a large number of operands, which is a reasonable choice since that is the design methodology of a vector architecture. Thus, once an operation is initialized it can operate at the cycle rate of the machine [Flynn]. See Figure 18 for the approximate timing of a 4-stage pipelined arithmetic unit.

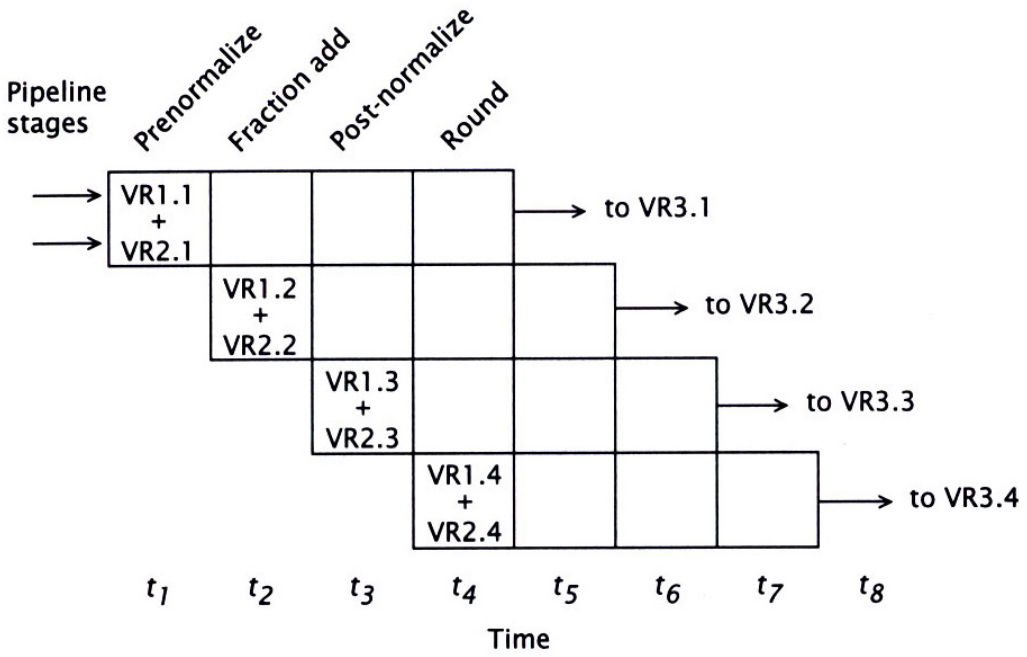

Figure 18: Generic Timing for 4-Stage Arithmetic Pipeline [Flynn Figure 7.4]

The second advanced vector technique is referred to as "vector chaining." This technique allows execution of more than one vector arithmetic operation per clock cycle. This is made possible when the results of the first operation can be directly fed in as an operand to the second operation, without first buffering it in a vector register or memory [Flynn]. This can be seen in Figure 19 and Figure 20 below.

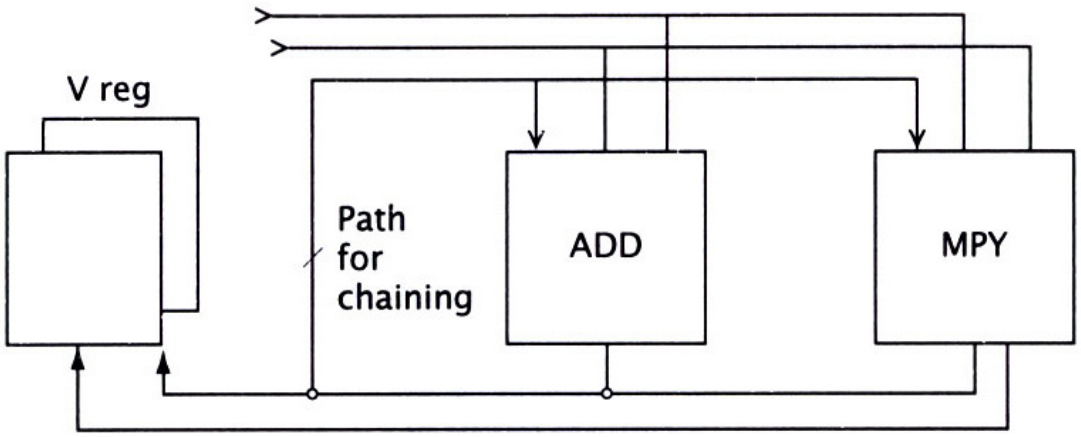

Figure 19: Vector Chaining Data Paths [Flynn Figure 7.13]

Two instructions: VADD C, A, B

VMPY E, C, D

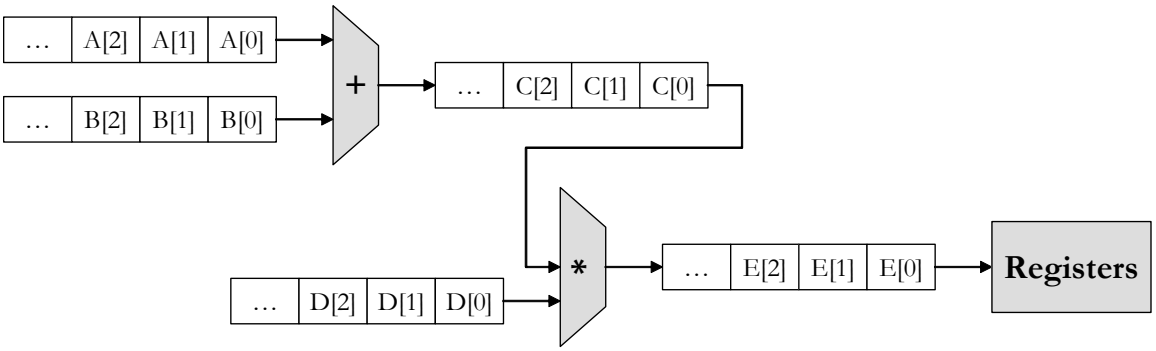

Figure 20: Data Flow Diagram for Vector Chaining [Based on Flynn Figure 7.12]

As an example of chaining, consider an addition operation followed by a multiply. Unchained, it would take 4 cycles (startup) + 64 cycles (elements/vector) = 68 cycles for each instruction, or a total of 136 cycles over both operations. With chaining, the system performance is enhanced because execution time is now 4 cycles (addition startup) + 4 cycles (multiply startup) + 64 cycles (elements/vector) for a grand total of 72 cycles to complete both operations.

Another common technique to improve vector processor performance is to use multiple parallel pipelined execution units. This exploits the fact that there are implicitly no data hazards within a vector instruction. A comparison between a single pipeline architecture and a 4-unit parallel pipeline system is shown in Figure 21.

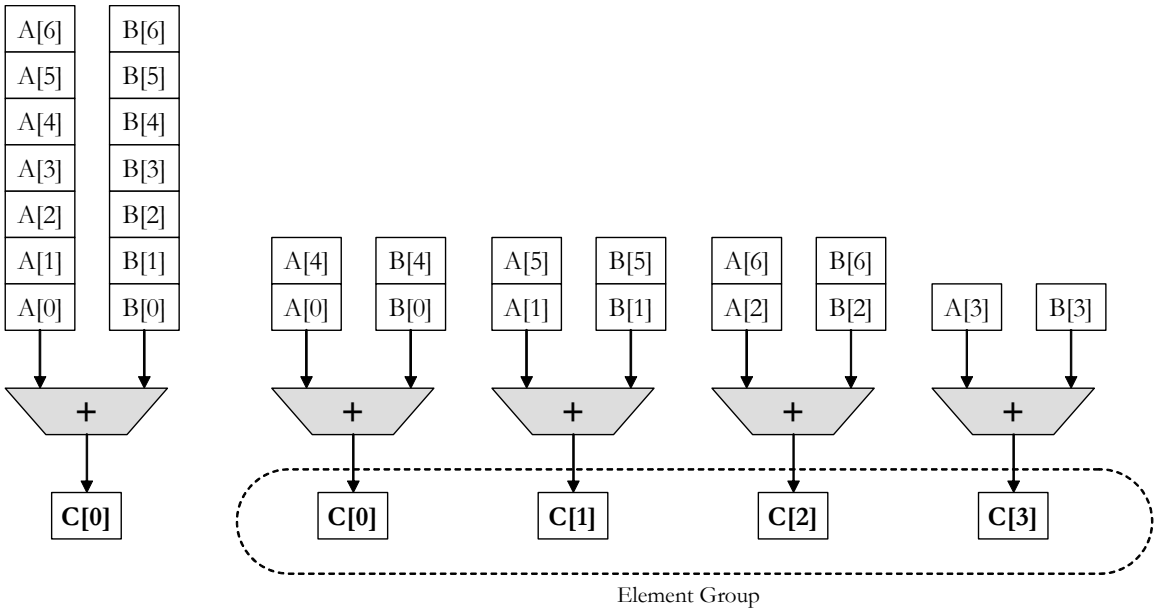

Figure 21: Multiple Functional Units for Improved Performance of C = A+B [Based on Asanovic Figure 2.4]

Here, while architecture (a) can only compute one result per unit time, architecture (b) can compute four results of the "element group" in parallel in the same length of time. This is the previously mentioned technique of "strip mining." The advantage of this approach is that the processor throughput increases without a significant change in control unit complexity, and no change in the machine instruction code. Further, it allows for a wide range of processors in a single design family by simply varying the number of parallel arithmetic units [Hennessy].

Finally, scatter and gather functions are typically part of the load/store module in vector supercomputers. These allow convenient support of sparse matrix operations and transitioning between a normal data representation (with zeros included) and a dense representation (where zeros are omitted). To enable this capability, the gather operation takes a vector containing either addresses in memory, or offsets from a base address specified in a scalar register. Then, it collects ("gathers") data from memory and saves it in a compacted vector register where all data elements are adjacent. To reverse this operation, the scatter operation saves the dense vector data elements to memory at the addresses (or offsets) specified in a vector address register [Hennessy, Mathew].

Implementation of scatter / gather capabilities can substantially improve performance on sparse matrices with significant numbers of zero entries.

An advanced technique not related to hardware architecture, but used in vector systems, is a method to accelerate matrix multiplications. The product of an *l* x *m* matrix **A** and a *m* x *n* matrix **B** is a *l* x *n* matrix **C** calculated as $\mathbf{C_{i,j}} = \sum_{k=0}^{m-1} \mathbf{A_{i,k}} \mathbf{B_{k,j}}$. This calculation requires *l* x *m* x *n* additions and the same number of multiplications, and has a calculation time of $\Theta(n^3)$ [Quinn]. The Strassen Algorithm has a lower computational complexity of $\Theta(4.7n^{2.807})$ which can improve performance on large problem sets. In this algorithm, only 7 half-size matrix multiplies are required to complete the full matrix multiply, compared with 8 using the traditional approach. The remaining multiplication is replaced with a shorter sequence of additions, which accelerates the calculation by approximately 14%. Further, this algorithm is typically applied recursively to calculate the half-sized blocks, and so on, until reaching a certain minimum scale at which point the multiplications can be calculated directly with either scalar arithmetic or via the traditional cross-product method (if recursion is terminated a few levels earlier). This yields an additional 14% theoretical improvement at each recursion level. However, this improvement has a practical limit, since, as discussed previously, vector processors are typically more efficient in longer vector lengths, not short segments. Further, there are several general tradeoffs in the overall Strassen algorithm at any level to obtain this improved theoretical performance. These include greater programming complexity, additional storage requirements for temporary values, and greater number of memory accesses [Bailey, Huss-Lederman].

3.6 Vector Processor Memory Systems

Just like their scalar cousins, high-speed memory systems are crucial for vector processors, since it is desirable for even the simplest of architectures to compute on average one arithmetic operation per clock cycle. Thus, the memory system should be able to support at minimum one read and one write per clock cycle. This allows for new

data to be loaded into the vector registers and for completed data to be written back to memory. An improvement would be a memory system that allowed two reads and one write, allowing for two vector execution units to operate concurrently.

Without sufficient bandwidth the processor will often be forced into a wait state. The greater demands of vector processing cores because of their higher efficiencies on large data sets means that designers should not simply append a vector processor to an existing scalar design without examining and likely upgrading the memory system [Flynn].

When designing a memory system, it is well known that memory accesses often need to be made to different physical modules in order to realize the maximum theoretical bandwidth [Jordan]. Modern designs typically interleave memory modules (each a full word wide) in order to provide sufficient memory bandwidth to the processor. This is quite effective for scalar processors, which often access data sequentially, spreading out access among different interleaved memory modules. A problem develops, however, when using a vector processor to perform matrix calculations from an interleaved memory system. If the calculations are row-oriented, memory accesses will be sequential, and the system will operate efficiently. However, if the calculations are column-oriented, and particularly if the 'stride' of the column in memory is equal to the interleaving factor of the hardware, the memory accesses can become unbalanced and focused heavily on a single memory module, much to the detriment of overall system performance. The term 'stride' refers to the distance in physical memory between adjacent words in the virtual vector, which could be column-oriented, row-oriented, diagonal, or perhaps even square sub-matrices within the overall virtual memory structure [Stone].

This tradeoff in storage methods of an 8x8 matrix is shown below in Figure 22.

(a) Row-Oriented Interleaving

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

(b) Column-Oriented Interleaving

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(7,4)
(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)	(7,5)
(0,6)	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)	(7,6)
(0,7)	(1,7)	(2,7)	(3,7)	(4,7)	(5,7)	(6,7)	(7,7)

Figure 22: Memory Interleaving Techniques [Based on Stone Figure 5.12]

In section A, the interleaving between distinct memory modules is suitable for access by row vectors, but bad for column vectors due to the concentration of access in a single module. Similarly, the arrangement in section B is good for access by column vectors, but is bad for access by row vectors.

Further, either a long stride or the nature of the special-purpose application can negate the positive benefits of a data cache, if one even exists. Performance can even be degraded when using a poorly tuned active cache that is designed to anticipate sequential memory accesses and actively fetch future data. Active caches could cause further bus congestion and memory traffic without any positive benefits. This is because the long vector strides ensures that no sequential data may ever be needed [Flynn].

Too much contention for the same memory band is called a *hot spot*. This term has some correlation to a physical condition, where the memory temperature increases

dramatically from continuous use. Although physical temperature can be reduced by advanced cooling hardware, the performance bottleneck remains. Thus, a goal of a designer is to reduce this bottleneck by either software or hardware schemes [Quinn].

This memory interleaving problem can be alleviated by using a memory architecture that attempts to balance data access across a system with N interleaved physical memory modules. Access should be balanced for any stride of memory access in the vector, whether the vector is row-oriented or column-oriented. Note that column orientation, while a very useful concept for doing matrix manipulations, is a virtual term that only has meaning at the programming level, as all memory systems are one dimensional. This system is shown in Figure 23.

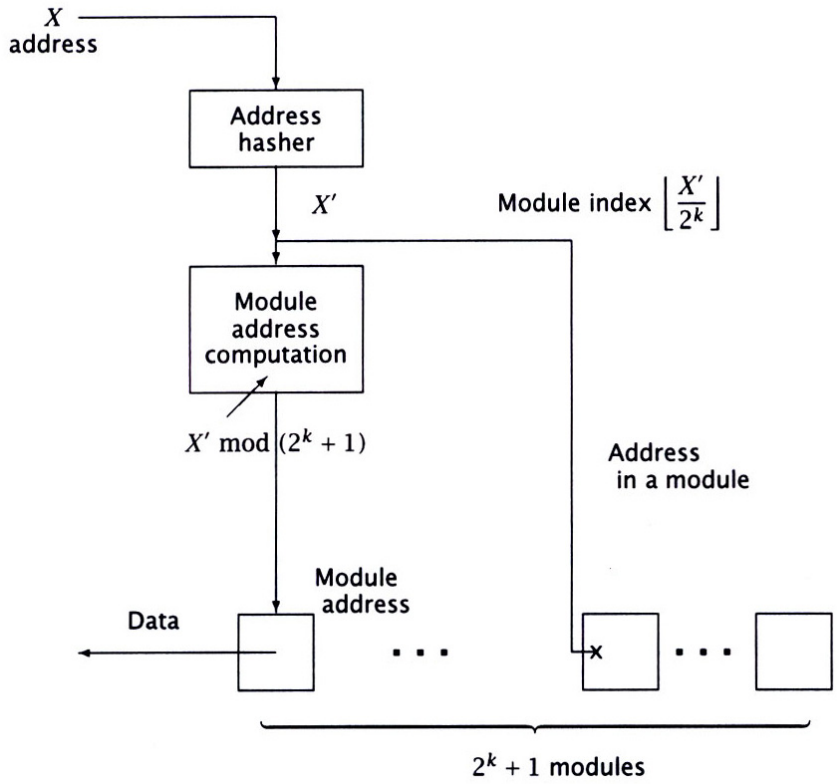

Figure 23: Vector Memory N-Interleaving System [Flynn Figure 7.17]

Flynn describes a hardware approach to balance data access. In step 1, the addresses are hashed. This disperses them throughout the memory system. A common hash algorithm used in hardware takes several bits, perhaps 3, that would normally be used to select the specific memory module from a group. Then, either an XOR of specific

combinations, negation of values, or rearrangement of values is performed. This is shown in Figure 24.

X_i	X_j	X_k	Conventional Address	$X_i \forall X_j$	$X_j \forall X_k$	$\overline{X_k}$	Mapped Address
0	0	0	0	0	0	1	1
0	0	1	1	0	1	0	2
0	1	0	2	1	1	1	7
0	1	1	3	1	0	0	4
1	0	0	4	1	0	1	5
1	0	1	5	1	1	0	6
1	1	0	6	0	1	1	3
1	1	1	7	0	0	0	0

Figure 24: Simple Address Mapping (Hashing) [Flynn Table 7.2]

Hashed memory addresses can reduce memory contention in mixed-application environments where the vector access stride is varied. But, in large matrix manipulations, contention is likely to reoccur because the same memory stride will be used for a lengthy period of time. Thus, further techniques are needed after hashing the addresses.

In step 2, a module mapping is performed with 2^k+1 memory modules to distribute the hashed address across an odd number of modules. The specific calculation performed is (Address mod 2^n) mod (2^k + 1). This has the effect of spreading N (even) "buckets" of data across N+1 (odd) memory modules, so that neither column-wise or row-wise array access will produce unbalanced memory reads such that all requests are to the same physical memory module. This wastes 1/(N+1) of the total memory capacity, but the performance improvement may be well worth it. This approach is shown in Figure 25.

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
(1,7)		(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(2,6)	(2,7)		(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(3,5)	(3,6)	(3,7)		(4,0)	(4,1)	(4,2)	(4,3)
(4,4)	(4,5)	(4,6)	(4,7)		(5,0)	(5,1)	(5,2)
(5,3)	(5,4)	(5,5)	(5,6)	(5,7)		(6,0)	(6,1)
(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)		(7,0)
(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	

Figure 25: Alternate Arrangement with 8 Columns [Based on Stone Figure 5.13]

As shown, both row and column access is equally well supported. Row access has a stride of 1, while column access has a stride of 9. The empty elements form the unused 9^{th} column of the 8 x 8 matrix.

One reason this specific approach was proposed by Flynn and Stone is that the implementation overhead is fairly low, especially considering the extensive pipelining and overlapping inherent in modern vector processors. The hashing module should have 1-2 gate delays, while the module address mapping module performs 2 serial bit "additions" that add no more than a few clock cycles to the overall system [Flynn].

3.7 Vector Processor Performance Analysis

There are several factors that influence vector processor performance. These include:

1. The amount of program code that can be expressed in vector form
2. The average length of the vectors processed
3. The startup overhead, which corresponds directly to the length of the pipeline
4. The number of parallel execution units, and whether those units allow for the chaining of operands
5. The number of operands in memory that can be loaded or stored in parallel
6. The number of vector registers

When analyzing the performance of a vector processor, there are several fundamental equations. The first calculates the time to perform a vector operation of length L.

$$T(L) = T_{start} + Lt_p \qquad \text{[Jordan]} \qquad (7)$$

where

$$T_{start} = T_i + (K-1)t_p \qquad (8)$$

In this equation, T_{start} is the startup time, T_i is the instruction issue time, t_p is the cycle time, and (K-1) is the number of cycles required to fill the pipeline before results appear [Jordan].

The second fundamental equation calculates the efficiency of pipeline usage for a vector of length L.

$$E(L) = \frac{L}{(T_i / t_p) + (K-1) + L} \qquad \text{[Jordan]} \qquad (9)$$

From this equation, it is seen that the processor efficiency approaches 100% as the vector size increases.

The third equation calculates the overall performance speedup of a vector processor over a scalar design.

$$S_P = \frac{T_1}{T_P} = \frac{T_1}{\%\ \text{vector} \cdot \frac{T_i}{S_M} \cdot \%\ \text{nonvector} \cdot T_1} \qquad \text{[Flynn]} \qquad (10)$$

Here, T_1 is the execution time of a generic pipelined processor, S_M is the maximum speedup of the proposed vector processor, and the percentage of scalar code that can be expressed in vector form is known. This S_M is generally limited to a factor of 4, or at most 6 if the system supported operand chaining and the memory system was expanded

to allow three reads and one write in parallel [Flynn]. This speedup is shown in Figure 26, which assumes ideal conditions of long vector lengths and no memory contention.

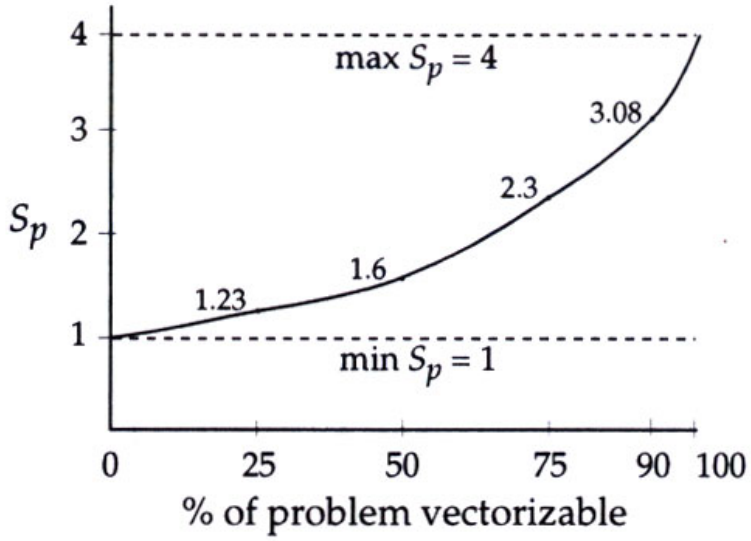

Figure 26: Vector Processor Speedup versus Percentage of Vectorizable Code
[Flynn Figure 7.22]

Finally, we come to the widely known Hockney and Jesshope vector efficiency calculation which produces results in the range of 0 to 1.

$$R / R_\infty = \frac{1}{1 + 1/(n / n_{1/2})} \qquad \text{[Hockney]} \qquad (11)$$

In this equation, $R_\infty = 1/\Delta t$ =1/ cycle time of the generic vector pipeline. This measures the maximum vector arithmetic execution rate that is sustainable in an un-chained processor design. If the system is chained, the maximum execution rate is simply $c * R_\infty$

The variable n is the number of elements in a vector register, i.e. the max vector length. $n_{1/2}$ is the length of a vector that achieves exactly ½ of the maximum performance. It is approximately equal to either the number of startup cycles for vector arithmetic (the depth of the pipeline plus any memory bus overhead), or the number of overhead cycles when loading a vector from memory [Hockney]. A plot of relative vector efficiency versus relative length is shown in Figure 27.

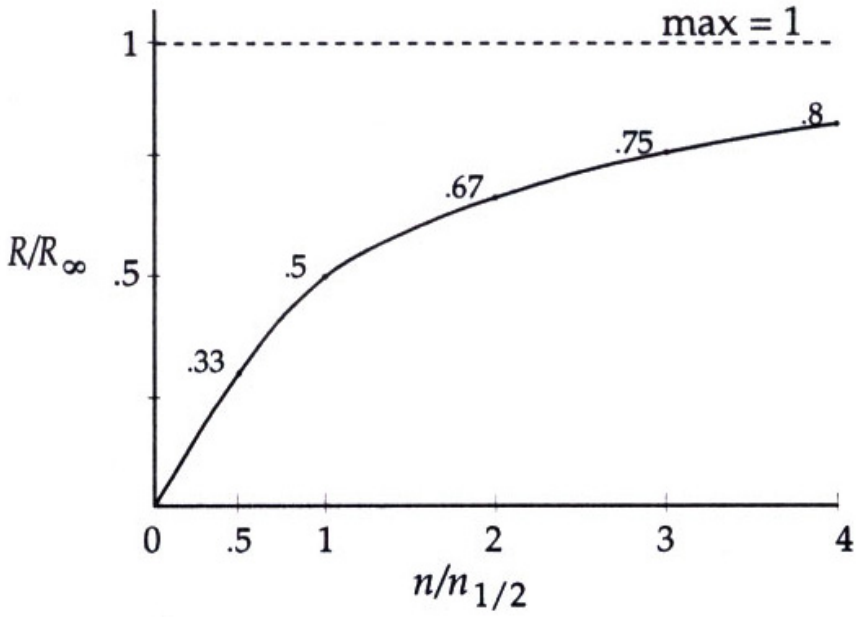

Figure 27: Relative Vector Performance vs. Relative Vector Length [Flynn Figure 7.25]

As expected, the relative vector performance (efficiency) increases as the vector length increases.

3.8 Real-World Systems

Vector processing capabilities are accessed at many levels in real-world systems. At the lowest level, they can be implemented using an assembler that supports the vector instructions. This might be useful for low-level device drivers for video cards, but is rarely used by application-level programmers except perhaps in small frequently-executed modules where small program optimizations can generate significant overall improvements in system performance.

For applications, vectorizing compilers are typically available for most commercial hardware architectures to convert Fortran, C, or Pascal code to vector assembly language. This automatic vectorization is the easiest method. It seeks to find code loops that can be expressed in a single or combination of vector operations. Its effectiveness is limited, however, as it is difficult or impossible to automatically vectorize loops containing recursive calls or complex branching operations [Marksteiner]. Other

compilers extend existing languages such as C to allow direct vector manipulation from a higher level than assembly programming.

Besides assemblers and compilers, vector processing capabilities are often used by vectorized program libraries. These libraries, often written by the hardware manufacturer, contain dedicated routines for common math functions that have been specifically optimized for high vector performance. All the programmer needs to do is call the correct function in the abstracted library [Marksteiner].

While a proper history of commercial vector processors is beyond the scope and objective of this thesis, a quick history culminating in a discussion of future trends will help put this new custom vector processor in proper perspective.

The first commercial vector processors were the TI-ASC by Texas Instruments and the STAR-100 by Control Data Corporation; both introduced in 1972 [Espasa]. They utilized a memory-to-memory architecture that featured a long pipeline from memory, through the processor, and back to memory. The pipeline took a long time to fill, but was very efficient for longer vectors. To take full advantage of memory, it supported advanced scatter/gather operations. However, the designers used the same arithmetic units for scalar instructions that were used by vector operations. This made that pipeline relatively deep, with substantial start-up penalties for each scalar computation [Hennessy].

It was the mistake of these early designers to provide such a slow scalar unit that Seymour Cray famously corrected with his ground-breaking Cray-1 system in 1976. His new design had a vector-register architecture to reduce memory bandwidth requirements, and was the first to implement the concept of chaining vector operations together in the ALU. Perhaps most significantly, the Cray-1 was the fastest scalar processor in the world when it was introduced. This helped make it a commercial success to customers interested in scalar performance, vector performance, or both. The Cray-1 design served as the foundation for many future vector processor architectures [Hennessy].

Of course, computer technology didn't stop advancing after 1976. Significant new machines from Cray, CDC, and other new companies continued to push performance to new heights through the use of deeper pipelines, advanced memory architectures for higher bandwidth, more parallel execution units, sparse vector support, and other novel enhancements. [Hennessy] provides a brief summary of many of these vector architectures, along with additional references.

In recent years, many commercial scalar processors have added vector processing instructions to assist the primary scalar unit in intensive computation tasks such as multimedia. Several good examples of this are Sun's VIS for UltraSPARC and Intel's MMX and SSE for Pentium processors.

The Sun VIS for UltraSPARC was the first comprehensive SIMD instruction set extension to a general purpose microprocessor. It attempts to solve the problem that while SPARC registers, ALUs, and datapaths are 64 bits, many common integer values are only 8, 16, or 32 bits. Thus, why not put that space to good use by packing multiple integer values into a single register, and process them in parallel? Thus, the VIS extension performs SIMD arithmetic within a single register. The packing approach is shown in Figure 28.

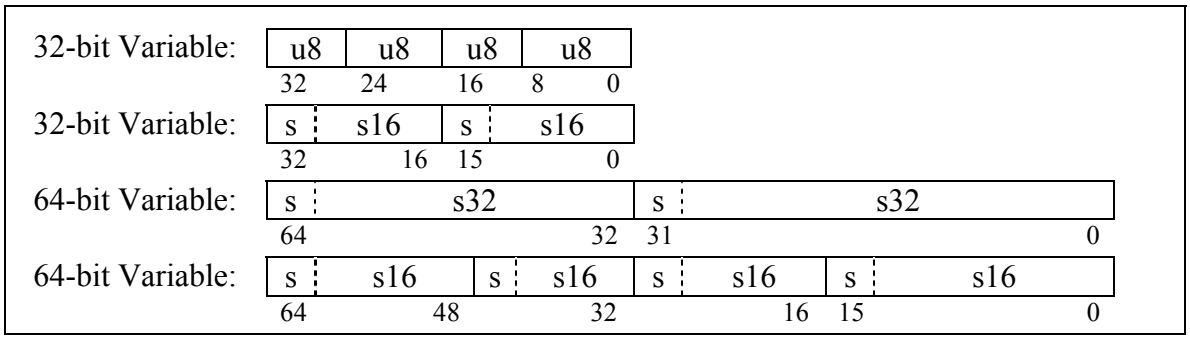

Figure 28: Sun VIS 1.0 Data Types [Sun]

The first version of VIS was introduced with the UltraSPARC 1 in 1995. It operated only on integer and fixed point data. Version 2.0 introduced with the UltraSPARC 3. It contains additional instructions for "data shuffling", which allow VIS

to handle incompatible data formats by reordering the packed words during ALU operations [Sun].

Following (and perhaps inspired by) the introduction of VIS by Sun, Intel developed what is the most widely distributed family of vector instructions: MMX, SSE, and SSE2. Each instruction set enhancement is a superset of all previous versions.

The original MMX architecture ("MultiMedia eXtensions") was first added to the Intel Pentium processor. Similar to VIS, MMX SIMD instructions are used to operate on packed integers in the registers, making it a useful addition to the processor when repetitive operations on consistently-formatted data are needed [Intel]. These applications are most commonly high performance graphics programs (i.e. video games), although scientific and commercial applications can certainly benefit from the capabilities as well.

The MMX extension to the Intel x86 ISA uses 64-bit packed integer data types of byte, word, and double word as shown in Figure 29.

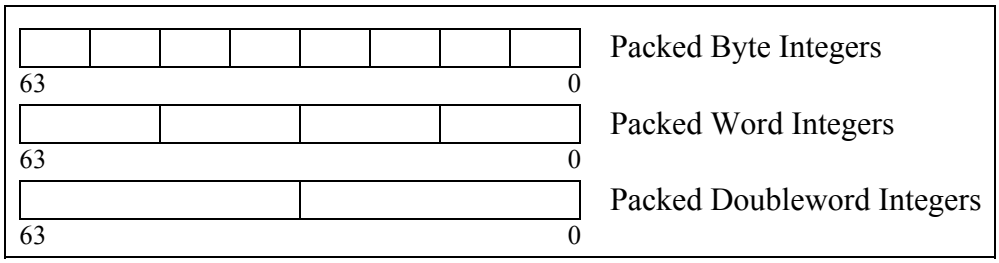

Figure 29: Intel MMX Data Types [Intel]

Eight virtual 64-bit MMX registers were added to the processor to store frequently accessed data. These MMX registers are in fact aliased onto the existing floating point registers to avoid forcing operating systems to modify their context switch routines. Because of this design choice, however, programmers must operate exclusively in SIMD or floating point mode for as long as possible to avoid expensive register context switches between the modes [Intel].

A few years after the introduction of MMX, Intel extended the SIMD capabilities of its Pentium III product line by developing SSE ("Streaming SIMD Extensions"). SSE

is a superset of MMX, and was enhanced to support packed single precision IEEE floating-point arithmetic in a SIMD fashion. An SSE-enabled processor can operate on 4 packed 32-bit (single-precision) values in a single instruction. To provide for full use of these capabilities, 8 128-bit floating-point registers were added that can be directly addressed by the new SIMD instructions. SSE instructions use the same functional units as floating-point instructions, which creates a structural hazard in that the CPU pipeline cannot issue both instructions at the same time [Intel].

Continuing this logical progression of upgrades that take advantage of increasing transistor counts, Intel added the SSE2 standard to its Pentium IV processor. This standard extends SSE to support IEEE double-precision floating point arithmetic. An SSE2-enabled processor can operate on 2 64-bit (double-precision) values in a single instruction. Further, in those same 128 bits, integer operations can also be performed in groups of 8, 16, 32, and 64 bits. An SSE2-enabled processor can thus alternate between calculating low and high precision values depending on application needs, although not in the same instruction word. Further, by performing integer operations from the 128-bit registers first added in SSE, SSE2 finally removed the bottleneck that MMX first created by mapping integer MMX registers on top of the physical floating-point registers [Intel].

Other vector-inspired co-processing systems include a series by AMD including 3DNOW, 3DNOW Enhanced, and 3DNOW Professional. AMD followed a year or two behind the Intel extensions and provides similar capability, although without full IEEE floating point compliance [AMD]. Besides AMD, Motorola added vector processing capabilities with AltiVec for their PowerPC architecture, as did MIPS with MDMX. Finally, there are several embedded vector architectures for video processing in the Sony Playstation 2 and Nintendo-64 console video game systems [Hennessy].

3.9 Future Directions

The vector processor landscape has changed dramatically over its 30+ year lifespan. In its heyday in the 1980's, vector designs demonstrated marked superiority over scalar machines. As of 2003, however, this performance gap between vector supercomputers and pipelined superscalar processors has rapidly decreased. In fact, "The peak floating-point performance of the low-cost microprocessors is within a factor of 4 of the leading vector supercomputer CPUs" [Hennessy].

The key remaining difference today between commodity designs and supercomputer is in memory bandwidth. In 2002, the fastest commodity processor could sustain transfers of approximately 1 GB/sec from main memory, while the fastest vector supercomputers have memory architectures that can sustain memory bandwidths approaching 50 GB/sec per CPU [Hennessy].

One key reason for the narrowing of the performance gap is the substantially higher clock rates of commercial processors. This is achieved through their more advanced design and fabrication methods, made affordable because their development expenses can be amortized over huge production runs. (In contract, a supercomputer does well if it sells over a hundred individual machines). Recently, supercomputers have been moving away from expensive proprietary bipolar ECL or gallium arsenide fabrics to standard CMOS technology to take advantage of recent advances in performance, power consumption, and heat dissipation. The technology exchange does go both ways, however. As previously mentioned, commodity processors made by Intel and AMD, among others, are adding short vector instructions to their architecture [Hennessy].

Because of the narrowing performance gap, most recent supercomputers, particularly in the United States, have been large clusters of relatively cheap commodity processors, not special vector processors. Perhaps the final stronghold of vector supercomputers is in specialized simulation applications that have large data sets and frequent scatter-gather operations from memory. Some of these applications and underlying algorithms need to be parallelized to run effectively on clusters of commodity machines. It is perhaps inevitable, as Hennessy and others predict, that as this set of

vector applications shrinks, vector machines will become unviable as commercial products and disappear from the marketplace. But, some of the advantages of vector processing architectures will likely persist in the form of SIMD extensions to superscalar machines, as has been already done by Intel, AMD, and others [Hennessy]. Further, some of these same architectures first pioneered for general purpose vector machines can be applied to special purpose processors for dedicated applications. One such application, wavelet video compression, is described in the next chapter.

Chapter 4

Vector Processor Design for Wavelet Compression

4.1 Overview

Vector processing is a proven technology that has been around for several decades, most often in high performance, expensive, supercomputers. Recently, though, these machines have often been outperformed by lower-cost clusters of commodity workstations running superscalar CISC or RISC architectures. However, with the advent of the Field-Programmable Gate Array (FPGA) chip and high-level hardware description languages, we now have the opportunity to apply this proven technology to specific implementations where it would never have been cost effective to do so before. In this thesis, a vector processor is applied to the problem of real-time embedded wavelet video compression.

There are some specific concerns when designing a processing system for image or video processing applications. Like matrix arithmetic with non-unity strides in memory, image processing often has problems with cache locality. Many image algorithms have a 2-D *spatial* locality, instead of the typical 1-D locality found in regular vector arithmetic and in ordinary scalar computations. This is, they are just as likely to request the next pixel up or down from the current location as they are to request the adjacent pixel left or right. This vertical dimension, while adjacent in the virtual data structure, has no physical adjacency in the hardware memory system. This causes significant problems for cache structures that anticipate physical adjacency in memory when prefetching and preserving blocks of data. The loss of performance in this cache misses can be quite significant [Cucchiara]. In this vector processing system, cache memory is not used at all because of the predicted high miss rates mentioned above. Rather, on-chip memory is used directly in the FPGA fabric to provide the necessary high memory bandwidth for effective computation.

The choice of matrix multiplications to calculate the wavelet transform seems problematic at first. Certainly, it is a convenient and straightforward method. But, a brute-force implementation would be both computationally inefficient and a waste of memory to store the full transformation matrix. After all, a single wavelet transform using this method has a complexity of $O(n^2)$, where n is the number of elements to transform. A closer examination of a typical transformation matrix, however (as shown in Equation 5, Chapter 2, page 15) reveals that it is highly sparse with many zero elements. Thus, a vector dot product operation which can operate on sparse data can retain much of the algorithmic elegance of matrix multiplies while significantly reducing their performance and storage overhead. It is just that approach that is applied in this vector processor.

The performance benefit of the sparse matrix approach is amplified even further by noting a characteristic of the matrix-based wavelet transform: The transformation matrix becomes increasingly sparse as the image size increases. Thus, the sparse matrix multiply approach becomes increasingly efficient! This is shown in several figures below, when the memory requirements of a generic transformation are calculated at the image sizes shown in Table 3.

Table 3: Common Image Sizes

Pixel Count	**Dimensions**
256	(16x16)
1,024	(32x32)
4,096	*(64x64)*
16,384	*(128x128)*
65,536	*(256x256)*
2,621,44	*(512x512)*
1,048,576	*(1024x1024)*
4,194,304	*(2048x2048)*

In a non-sparse method, the number of storage elements to hold the transformation matrix must equal the number of image pixels. But, in the sparse method, only the wavelet coefficients themselves must be stored (or, perhaps, twice that number to accommodate edge wrapping). This benefit is shown in Table 4 and Figure 30.

Table 4: Transformation Matrix Storage Elements

Non-Sparse	Sparse
256	16
1,024	16
4,096	16
16,384	16
65,536	16
262,144	16
1,048,576	16
4,194,304	16

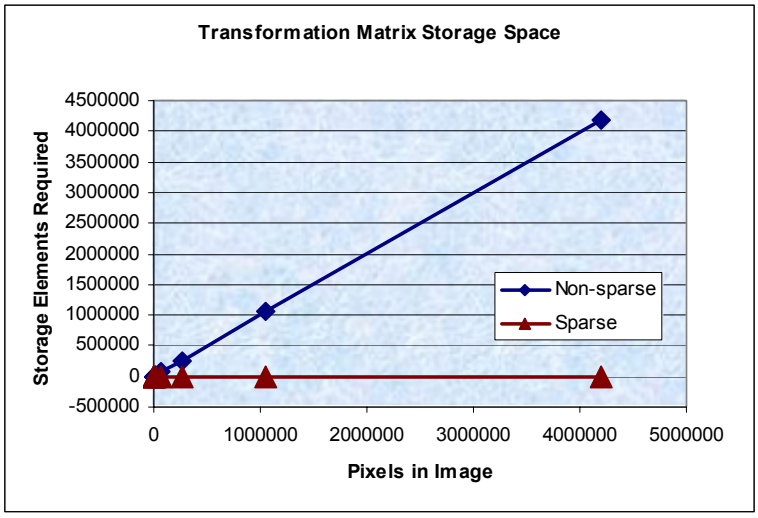

Figure 30: Improvement in Storage Requirements

The sparse method is barely even visible at the bottom of the figure! A similar improvement is shown when calculating the number of memory accesses to compute a transformation. While the non-sparse method is the square of the pixel count, the sparse method is the pixel count multiplied by the wavelet length. This is a substantial improvement that scales favorably as size increases, as shown in Table 5 and Figure 31.

Table 5: Transformation Memory Accesses

Non-Sparse	Sparse
4,096	2,048
32,768	8,192
262,144	32,768
2,097,152	131,072
16,777,216	524,288
134,217,728	2,097,152
1,073,741,824	8,388,608
8,589,934,592	33,554,432

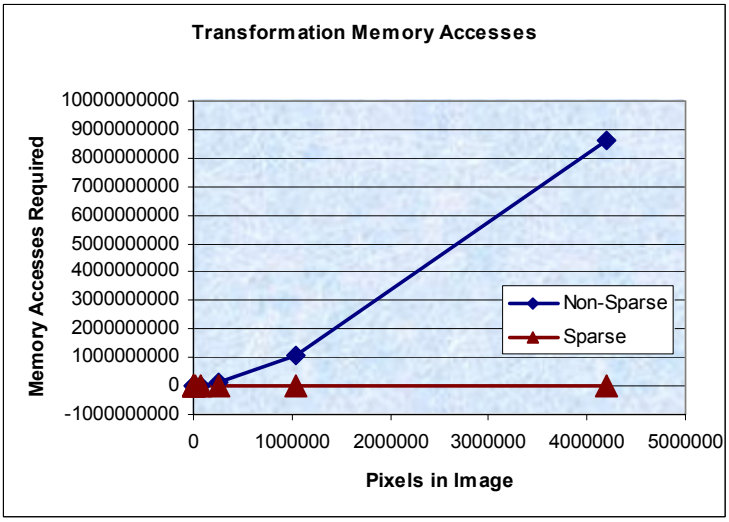

Figure 31: Improvement in Memory Accesses Required (i.e. Performance)

4.2 Instruction Set

The wavelet transformation Matlab program in the preceding chapter was analyzed to determine the appropriate set of low-level instructions necessary to provide similar functionality. It was determined that a small set of register-based arithmetic operations, both vector and scalar, could accomplish the desired tasks. Load and store routines with both immediate, direct, and indexed addressing modes were provided to manipulate the registers. To further enhance performance, however, a vector multiply and accumulate instruction was added to this baseline set that fetches one operand directly from memory and saves the result directly to memory. While a noticeable departure from a strict RISC organization, this "dot product" operation forms the foundation of a matrix multiply, and was the most frequent operation in the final wavelet program. Having its operands come from memory reduces the instruction count compared with a pure RISC design (no need for adjacent loads and stores), and thus accelerates the final program execution speed.

The instruction formats used in the vector processor are shown in Figure 32. In this format, both scalar and vector register labels are 4 bits long. Because only 3 bits are actually needed to represent the 8 vector registers, a leading 0 is padded to the label to fill the instruction field.

Load / Store Instructions: *VR = Vector Register, SR = Scalar Register*

Op-Code (5)	Address / Immediate Data (23)	VR / SR (4)

Store Indirect Instruction:

Op-Code (5)	*Unused* (9)	SR (4)	*Unused* (10)	SR (4)

Multiply and Accumulate Instruction:

Op-Code (5)	ALU Mode (5)	VR (4)	VR (4)	SR (4)	*Unused* (6)	SR (4)

ALU Instruction:

Op-Code (5)	ALU Mode (5)	VR / SR (4)	VR / SR (4)	*Unused* (10)	VR / SR (4)

Branch / Jump Instruction:

Op-Code (5)	ALU Mode (5)	SR (4)	SR (4)	Address (14)

Figure 32: Instruction Formats

Table 6 lists all the instructions of the constructed vector processor, using the 5 instruction formats listed in Figure 32.

Table 6: Instruction Set for Vector Processor

Op	Opc	Operands	Description
NOOP	00000	N/A	No operation
LDS	00001	Addr (23), SR# (4)	Scalar Load Direct
LDSI	00010	Imm (23), SR# (4)	Scalar Load Immed.
STS	00011	Addr (23), SR# (4)	Scalar Store Direct
STSI	01110	0 (9), SR#(4), 0 (10), SR# (4)	Scalar Store Indirect
LDV	00100	Addr (23) 0, VR# (3)	Vector Load
STV	00101	Addr (23), 0, VR# (3)	Vector Store
MLACS	00111	ALU (5), 0, VR# (3), 0, VR#(3), SR# (4), 00 0000, SR#(4)	Vector Mult And Accum (Sparse)
ALUVS__	01000	ALU (5), 0, VR# (3), SR# (4), 00 0000 0000 0 VR# (3)	Vector ALU Scalar
		Allowed variants: *ADD, SUB, MUL, AND, OR, XOR, NOT, NEG, SLL, SRL, SLA, SRA* (e.g. ALUVSADD)	
ALUVV__	01001	ALU (5), 0, VR# (3), 0, VR# (3), 000 0000 0000, VR# (3)	Vector ALU Vector
		Allowed variants: A*DD, SUB, MUL, AND, OR, XOR, NOT, NEG* (e.g. ALUVVADD)	
ALUSS__	01010	ALU (5), SR# (4), SR# (4), 00 0000 0000, SR# (4)	Scalar ALU Scalar
		Allowed variants: *ADD, SUB, MUL, AND, OR, XOR, NOT, NEG, SLL, SRL, SLA, SRA* (e.g. ALUSSADD)	
BRT__	01011	ALU (5), SR# (4), SR# (4), Addr(14)	Scalar ALU Comparison, Jump if true
		Allowed variants: *GT, GTE, LT, LTE, EQ, NEQ* (e.g. BRTGT)	
JMP	01100	0 0000 0000 0000, Addr (14)	Jump (absolute)
STRM	01111	0 (27)	Stream image (in new, out old)

An overview of each instruction is described in the following section. It should be noted that the Vector Stride Register (VSR) implicitly maps to scalar register 14, and the Vector Length Register (VLR) implicitly maps to scalar register 15. These parameters are specified implicitly as registers and not immediate data in the instructions because it frees up valuable instruction space and because their values may not always be known at compile time [Hennessy].

LDS - Scalar Load
Operands: Memory Address, Scalar Register

The data in memory at the specified memory address is loaded into the specified scalar register.

LDSI – Scalar Load Immediate
Operands: Immediate data

The data in the instruction is loaded into the specified scalar register.

STS - Scalar Store
Operands: Memory Address, Scalar Register

The data in the specified scalar register is stored in memory at the specified memory address.

STSI - Scalar Store Indirect
Operands: Scalar Register #1 (holds address), Scalar Register #2 (holds data)

The data in the second scalar register is stored in memory at the address located in the first scalar register.

LDV - Vector Load
Operands: Address, Vector Register

The virtual vector in memory at the specified starting address is loaded into the specified vector register. The virtual vector in memory has a length specified implicitly by the Vector Length Register (VLR). The virtual vector in memory has a distance between elements ("stride") specified implicitly by the Vector Stride Register (VSR). The VSR register only affects the stride when accessing data from main memory. The stride when accessing the vector register is always 1.

> Note: The VSR has values of 1-16383. Otherwise, results are undefined.
> Note: The VLR has values of 2-32. Otherwise, results are undefined.

STV - Vector Store
Operands: Address, Vector Register

The vector in the specified vector register is saved to a virtual vector in memory. This virtual vector starts at the specified address. It has a length specified implicitly by the Vector Length Register (VLR) and a stride specified implicitly by the Vector Stride Register (VSR). The VSR register only affects the stride accessing of data from main memory. The stride when accessing the vector register is always 1.

> Note: The VSR has values of 1-16383. Otherwise, results are undefined.
> Note: The VLR has values of 2-32. Otherwise, results are undefined.

MLACS - Wavelet / Vector Sparse Multiply and Accumulate
Operands: VRa# (sparse memory addresses), VRb#(multiplicand), SR# (shift amount), SR#(dest)

This instruction is custom designed specifically to optimize the key operation of a matrix multiplication when used in a wavelet transform. It performs a multiply and accumulate operation which functions as a vector dot product. Unlike the other register-based ALU operations, one operand in this instruction comes directly from memory through a sparse addressing scheme. The result of the dot product is shifted to the right by a specified amount automatically to compensate for the multiplication factor that was applied to integerize the previously fractional wavelet coefficients. Finally, the computed result is indirectly saved to memory.

Operation performed: $\left(\sum_{x=0}^{VLR}\left(Mem[VRa(x)]*VRb(x)\right)\right) >> ShiftAmount$

The first vector operand (VRa) contains absolute addresses in memory of data to multiply against second vector operand (VRb). This is the only instruction that allow the accessing of sparse (i.e. non-uniform stride) data to be accessed. Thus, in this sparse instruction, the VSR (stride register) is irrelevant and not used. The Vector Length Register (VLR) implicitly contains the length of the vector to be processed. The shift amount is stored in the scalar register, although the effective shifting range is limited to 0 to 15. The final result is stored in memory at the address found in the destination scalar register.

The sparse data stored in memory needs to be formatted in a specific manner to function correctly. This is because this operation processes 3 color planes simultaneously. Each color plane must be packed in 16-bit lengths into the full 48-bit wide data memory. Each packed plane is then multiplied against the same single word stored in the vector register. All three parallel results are then concatenated (repacked) before being written back to memory.

ALUVS – ALU Operation between Vector and Scalar
Operands: Vector Register (source), Scalar Register (source2), Vector Register (dest)

An ALU operation is performed between each element in the source vector register and the contents of the scalar register. For shift operations, the scalar register must contain the shift amount. Each result is stored sequentially in the destination vector register. The Vector Length Register implicitly stores the length of the operation.

Valid arithmetic operations are: ADD / SUB / MUL / AND / OR / XOR / NOT / NEG / SLL, SRL, SLA, SRA. The full instruction would be, for example, ALUVSADD or ALUVSNEG.

Note: Using NOT / NEG in this vector/scalar operation has the same effect as using it in a vector/vector operation.

ALUVV – ALU Operation between Vector and Vector
Operands: Vector Register (source), Vector Register (source2), Vector Register (dest)

An ALU operation is performed element by element between two vector registers, with the results stored sequentially in the third vector register. The Vector Length Register implicitly stores the length of the vector operation.

Valid arithmetic operations are: ADD / SUB / MUL / AND / OR / XOR / NOT / NEG. The full instruction would be, for example, ALUVVADD or ALUVVXOR.

For shift operations, the ALUVS command between a vector and a scalar should be used instead.

ALUSS – ALU Operation between Scalar and Scalar
Operands: Scalar Register (source), Scalar Register (source2), Scalar Register (dest)

An ALU operation is performed between two scalar registers, with the result stored in the third scalar register. For shift operations, the second source register must contain the shift amount. For single-operand operations (e.g. NOT, NEG), the second source register is not used.

Valid arithmetic operations are ADD / SUB / MUL / AND / OR / XOR / NOT / NEG / SLL / SRL / SLA / SRA. The full instruction would be, for example, ALUSSADD or ALUSSXOR.

BRT – Comparison and Branch if True
Operands: Scalar Register (source), Scalar Register (source2), Destination Address

An ALU comparison is performed between two scalar registers. If the comparison condition is true, then the program counter is set to the destination address, and the program branches.

Valid comparison operations are:

- Greater than
- Less than
- Equal
- Greater than or equal to
- Less than or equal to
- Not equal

JMP – Jump
Operands: Destination Address

The program counter is set to the destination address, and the program branches.

STRM – Stream

An I/O operation is performed. Incoming image data on the 48-bit wide external input port is copied to the data memory module, while previously transformed data is copied to the 48-bit wide external output port. The address to read and write at any given cycle is specified on the 19 bit wide external address port. The stream operation starts when the instruction is issued (indicating processor is ready to receive data) and the external write enable port is asserted (indicating the external system is ready to send data). The operation both sends and receives 1 48-bit word per clock cycle until the external finished port is asserted, indicating the external system is finished sending/receiving data.

A summary of the functionality provided by the main ALU is shown in Table 7. Entries denoted with a (*) reflect arithmetic operating modes that were initially implemented but later removed to conserve device resources because they were not needed in the wavelet transformation program. They are simply commented out of the VHDL code, and remain in the specification for possible later use.

Table 7: Operating Modes of Primary ALU

ALU Mode	Function
0	No Operation
1	Addition
2	Subtraction
3	Multiplication
4	Logical: And
5	Logical: Or
6	Logical: Xor
7	Logical: Not
8	Logical: Neg
9	(*) *Shift Left Logical*
10	(*) *Shift Right Logical*
11	(*) *Shift Left Arithmetic*
12	(*) *Shift Right Arithmetic*
13	(*) *Shift Left Circular*
14	(*) *Comparison: Greater than*
15	(*) *Comparison: Greater than or equals*
16	(*) *Comparison: Less than*
17	(*) *Comparison: Less than or equals*
18	Comparison: Equals
19	Comparison: Not Equals

4.3 Processor Architecture

When designing the processor architecture, one key decision to make was whether the system should operate memory-to-memory or register-to-register. Certainly, a memory-to-memory architecture would yield the most flexible design, and would reduce the processor's hardware footprint. However, it would require a high-speed memory system to be viable (ideally supporting multiple simultaneous reads and writes from separate memory banks). Further, if not carefully partitioned, this approach could tightly couple the processor design to a specific memory architecture; limiting future use.

In contrast, a register-to-register design would allow for the efficient reuse of common data, such as wavelet transformation matrices. But, it would have overhead in loading/storing registers from memory, and require a greater hardware footprint due to

the vector register set, which is a complicated module requiring multiple simultaneous reads & write ports.

Based on these factors, a compromise architecture was designed. The primary ALU operations are all register based as in a standard RISC machine. This allows for efficient reuse of data such as the wavelet coefficients and memory address pointers. But, a separate multiply and accumulate instruction was added that can access sparse data directly in memory for one of its operands. Thus, some of the benefits of the memory-memory approach have been incorporated as well. The final vector processor architecture as implemented is shown in Figure 33.

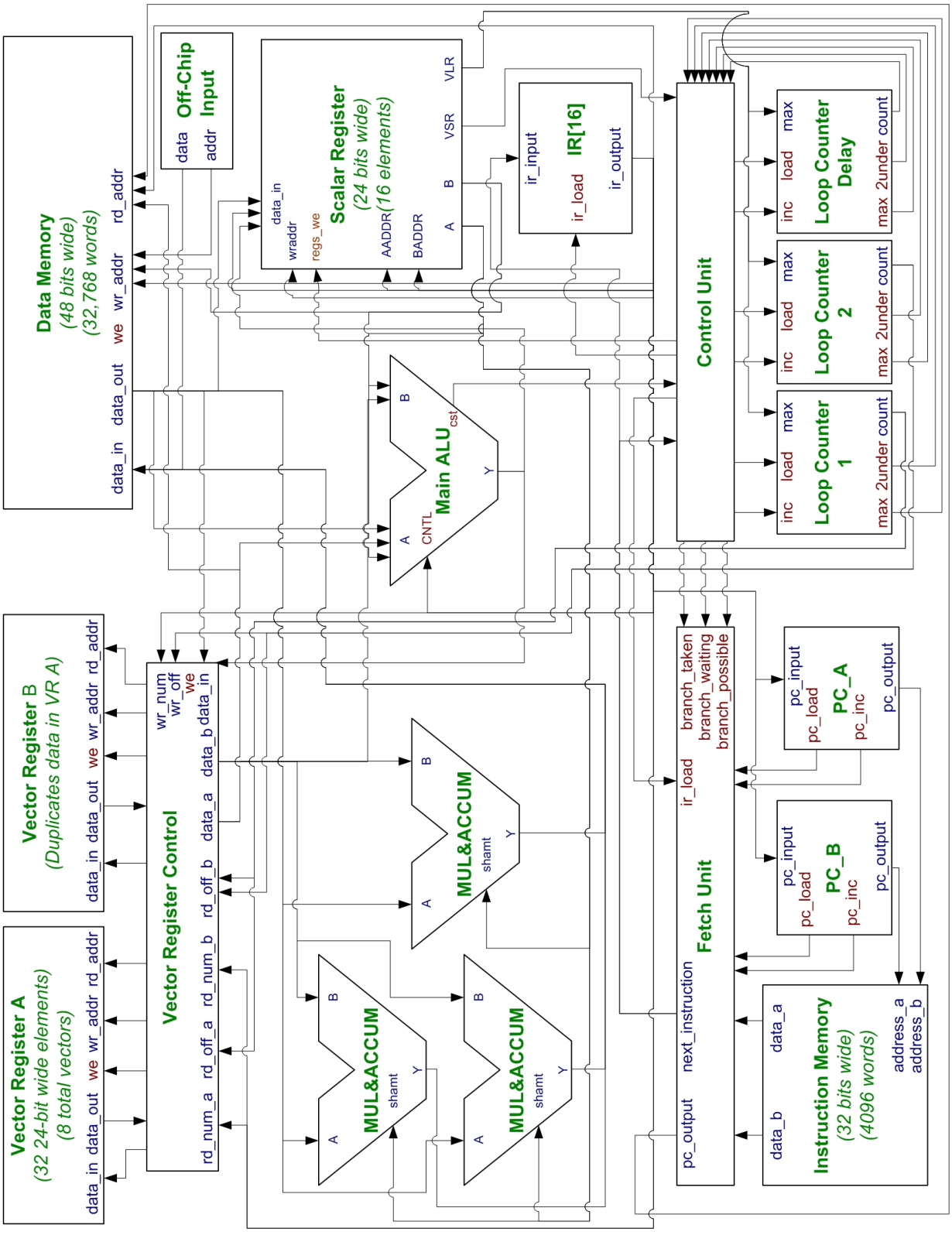

Figure 33: Vector Processor Architecture

As shown in the architecture figure, this processor utilizes a Harvard design with separate instruction and data storage elements. The instruction memory is embedded inside the Fetch unit. It stores 4,096 32-bit wide instructions, although it could easily be expanded for longer programs. Further, this instruction memory makes use of the inherent dual port memory capabilities of the Altera Stratix devices to have two read ports, so the fetch unit can prefetch down both directions of branches. Thus, the branch instruction timing is identical regardless of whether the branch is taken. All instructions are available immediately following the end of completion of the previous instruction. No separate decode state is necessary in the processor control unit.

To facilitate the dual-direction pre-fetching on branches, there are two program counters, A and B. Both are interchangeable and increment automatically with each instruction. The A counter is initially dominant, meaning that the instructions stored at these addresses are routed to the single instruction register. During a branch instruction, the non-dominant counter is reset to the branch destination. While the ALU decides if a branch should be taken, both counters continue incrementing for several cycles. Thus, after 2 cycles, when the ALU determination is ready, two full memory pipelines are available, one in the original direction, and the other in the branch direction. If the branch is taken, the dominant counter toggles between A and B. This results in a different instruction stream being routed to the instruction register. Otherwise, the dominant counter and instruction routing stays the same. In either case, both counters resume incrementing in subsequent instructions.

In the actual system implementation, instruction memory is initialized on the FPGA when the design is compiled by use of a .MIF (Memory Initialization File). There is no provision made in the architecture to change the program at any later stage. Of course, memory could be easily enlarged to allow several programs to reside in it at a single time (e.g. forward and inverse transforms), and the program control could branch between them as necessary.

In addition to the instruction memory, a separate data memory is provided to store the wavelet coefficients, the incoming image, and temporary values during computation.

It stores 32,768 48-bit wide elements, which is enough for 2 128x128 pixel images to be stored with one 16-bit pixel from each of the three color planes packed into a 48-bit word. This memory currently utilizes the on-chip M-RAM blocks on the Stratix device, which limits its maximum size to approximately 131,072 48-bit words in current devices. When loading data into the 24-bit wide scalar and vector registers, only the least-significant 24 bits are saved. The full 48-bit memory width is only utilized by the wavelet multiply and accumulate instruction, which is the most common instruction in the prototype wavelet transform program.

The M-RAM blocks used to construct the data memory cannot be initialized at startup via a .MIF file as was used in the much smaller instruction memory. This is not a difficulty, however, because in any real-world system the incoming image stream would not be known at compile time. In this system, the streaming data instruction is used after the vector processor has been activated. It connects the on-chip memory to external data pins to both load and unload raw and processed image pixels. In the system simulation, these external pins are connected to a small memory which contains image data used to verify processor operation.

The vector registers are created using on-chip memory elements. The vector register stores 8 vectors that are each 32 24-bit wide words. Three ports were desired to allow for 2 concurrent reads and 1 write. Because the built-in memory is only dual-ported, two separate identical memories were used with connected write ports. Thus, each vector is actually stored twice, once in each memory. Each virtual vector read port can be connected to a separate port on the physical memory, facilitating the desired number of ports.

Because the vector register is stored as a one-dimensional array of data in memory, an embedded multiplier is used in the register control unit to calculate the effective address in memory given a vector number and an offset within the vector. Because the multiplication factor is constant (the maximum length of the vector, 32), one port on the multiplier can be hard-wired, allowing for acceptable performance. It has one clock cycle to complete the address computation before the calculated address takes

effect at the memory read address port. At that point, one empty fetch cycle is required before the first vector element is finally returned on the third cycle. The vector register set is pipelined, however, so that this latency only affects the first vector element access.

The scalar register set has 3 ports allowing for 2 concurrent reads and 1 write. Each of the 16 register elements is 24 bits wide. Scalar register 14 is implicitly used as the Vector Stride Register (VSR), and scalar register 15 is implicitly used as the Vector Length Register (VLR). Both the VSR and the VLR have an additional read port from the vector unit so that they are always read. By mapping them onto standard registers, the usual complement of load and store instructions can be used to modify them. Unlike the vector register, the scalar register set uses on-chip registers instead of memory, and thus has a minimal latency of less than 20% of a clock cycle. This performance comes at the expense of significant amounts of reconfigurable fabric, as discussed later.

Four arithmetic units are provided in the processor. The primary general-purpose ALU computes the full range of arithmetic, logical, and comparison functions are described in Table 7. It is 24 bits wide, which is the width of the vector and scalar registers. Three secondary ALUs are provided for the exclusive use of the wavelet multiply and accumulate instruction, and allow it to compute all three color planes in parallel via a packed pixel approach. Each of these secondary ALUs are 16 bits wide, and only perform the multiply and accumulate function. The A input ports are all the same least-significant 16 bits from the same vector register (i.e. the wavelet coefficient), while the B input ports are fed from distinct packed pixels (i.e. three pixels from the three color planes), as shown in Figure 34.

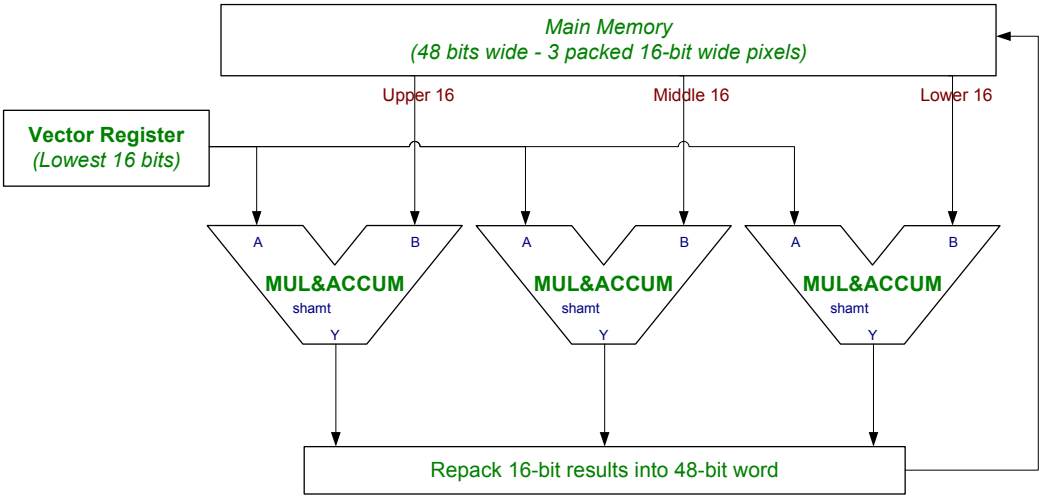

Figure 34: Dedicated Multipliers/Accumulators for Wavelet Instruction

The output from the three dedicated multipliers/accumulators is truncated to the least significant 16 bits, which should provide sufficient dynamic range during normal computation of the wavelet transform. The multiply and accumulate cycle is repeated for the next wavelet coefficient and image pixel for the length specified in the Vector Length Register. Upon computation of the last accumulate operation, the three 16-bit transformed pixels are then repacked into the 48-bit original format and saved directly to memory.

Once the system architecture was fully defined, it was implemented in VHDL, as described in the next chapter.

Chapter 5

Vector Processor Implementation

The vector processor architecture designed in the previous chapter was implemented in VHDL using the Altera Quartus II development environment. This chapter provides both a high level view of the processor data paths and a low level view of the control signals and multiplexers necessary to manage those data paths. A summary report of the FPGA device utilization after fitting is provided, along with performance metrics on each instruction.

5.1 Data Paths and Control Signals

A description of the processor operation at a high level is shown in Table 8. This table was built from the processor architecture figure and an analysis of the instruction set.

Table 8: Register Transfer Language Processor Description – High Level

#	Instr	RTL	Description
0	LDS	SR[IR] ← **M**[IR]	Load scalar register from memory Go to next instruction
1	LDSI	SR[IR] ← [IR]	Load scalar register from immediate data
2	STS	**M**[IR] ← SR[IR]	Store scalar register to memory (direct)
3	STSI	**M**[SR[IR]] ← SR[IR]	Store scalar register to memory (register indirect)
4	LDV	VR[IR][Count] ← **M**[IR+Count]	Initialize counter to zero Load vector register from memory Repeat until counter equals vector length
5	STV	**M**[IR+Count] ← VR[IR][Count]	Initialize counter to zero Store vector register to memory Repeat until counter equals vector length
6	MLACS	Accum ← **M**[VR[IR][Count]] * VR[IR][Count] + Accum **M**[IR] ← Accum	Initialize counter to zero Multiply and accumulate Repeat until counter equals vector length Store result in memory
7	ALUVS	SR[IR] ← VR[IR][Count] *(ALUOP)* SR[IR]	Initialize counter to zero Perform ALU operation between vector and scalar Repeat until counter equals vector length Store result in scalar register
8	ALUVV	VR[IR][Count] ← VR[IR][Count] *(ALUOP)* VR[IR][Count]	Initialize counter to zero Perform ALU operation between vector and vector Repeat until counter equals vector length Store result in vector register
9	ALUSS	SR ← SR[IR] *(ALUOP)* SR[IR]	ALU operation between scalar and scalar
10	BRT	CST ← SR[IR] *(ALUOP)* SR[IR] →(∨/[CST],!∨/[CST]) /(PC ← IR, No change)	ALU comparison between two scalar registers Decode CST register If CST = 1, branch directly
12	JMP	PC ← IR	Unconditional branch
13	STRM	**M**[Count] ← Incoming Bus	Initialize counter to zero Copy pixel on incoming bus to memory Repeat until counter equals memory size

A low-level description of the processor from the perspective of the control unit is provided in Table 9.

Table 9: Low-Level Control Unit Description

State	Outputs																										Next State
	ir_load	branch_possible	branch_waiting	branch_taken	stream	alu_cst_load	loop1_load	delay_load	loop2_load	mux_delay_max	loop1_inc	delay_inc	loop2_inc	memory_we	regs_we	regv_we	alu_mlacs_clr	mux_regv_data_in	mux_regv_rd_b	mux_regs_rd_addr_a	mux_regs_rd_addr_b	mux_alu_a	mux_alu_b	mux_mem_data_in	mux_mem_rd_addr	mux_mem_wr_addr	mux_regs_data_in	
noop	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	(*)
lds_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	lds_2
lds_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	lds_3
lds_3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	(*)
ldsi_1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	2	(*)
sts_1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	(*)
stsi_1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	3	0	(*)
ldv_1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	ldv_2
ldv_2	1 (c)	0	0	0	0	0	0	0	0	0	1 (b)	1 (d)	1 (c)	0	0	1 (c) (d)	0	1 (c)	0	0	0	0	0	0	2 (f)	0	0	ldv_2 / (*)
stv_1	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	stv_2
stv_2	1 (c)	0	0	0	0	0	0	0	0	1	1 (b)	1 (d)	1 (c)	1 (c) (d)	0	0	0	0	0	0	0	0	0	2	0	2	0	stv_2 / (*)
mlacs_1	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	mlacs_2
mlacs_2	0	0	0	0	0	0	0	0	0	1	1 (b)	1 (d)	1 (c)	0	0	0	1	0	1	2	0	3	2	0	3 (c)	0	0	mlacs_2 / mlacs_3
mlacs_3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	2	0	3	2	0	0	0	0	mlacs_4
mlacs_4	1 (b)	0	0	0	0	0	0	0	0	0	1 (b)	0	0	1 (b)	0	0	0	0	0	2	1	0	0	1 (b)	0	3 (b)	0	mlacs_4 / (*)
aluvs_1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	2	1	0	0	0	0	aluvs_2
aluvs_2	1 (c)	0	0	0	0	0	0	0	0	0	1 (b)	1 (d)	1 (c)	0	0	1 (c) (d)	0	1 (c)	0	0	0	2	1	0	0	0	0	aluvs_2 / (*)

(Continued on next page…)

(a) Conditional upon ALU Condition Status = 1
(b) Conditional upon Loop 1 Counter Finished = 1
(c) Conditional upon Loop 2 Counter Finished = 1
(d) Conditional upon Delay Counter Finished = 1
(e) Conditional upon Stream Finished = 1
(f) Conditional upon Loop 2 Counter 2Under = 1
(*) Jump directly to next instruction

(Continued…)

State	Outputs																										Next State
	ir_load	branch_possible	branch_waiting	branch_taken	stream	alu_cst_load	loop1_load	delay_load	loop2_load	mux_delay_max	loop1_inc	delay_inc	loop2_inc	memory_we	regs_we	regv_we	alu_mlacs_clr	mux_regv_data_in	mux_regv_rd_b	mux_regs_rd_addr_a	mux_regs_rd_addr_b	mux_alu_a	mux_alu_b	mux_mem_data_in	mux_mem_rd_addr	mux_mem_wr_addr	mux_regs_data_in	
aluvv_1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	2	0	0	2	2	0	0	0	0	aluvv_2
aluvv_2	1 (c)	0	0	0	0	0	0	0	0	0	1 (b)	1 (d)	1 (c)	0	0	1 (c) (d)	0	1 (c)	2	0	0	2	2	0	0	0	0	aluvv_2 / (*)
aluss_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	1	1	0	0	0	0	aluss_2
aluss_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	aluss_3
aluss_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	3	(*)
jmp_1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	jmp_2
jmp_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	jmp_3
jmp_3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	jmp_4
jmp_4	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	(*)
brt_1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	1	1	0	0	0	0	brt_2
brt_2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	brt_3
brt_3	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	brt_4
brt_4	1	0	0	1 (a)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	(*)
strm_1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	strm_2
strm_2	1 (e)	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	strm_2 / (*)

(a) Conditional upon ALU Condition Status = 1
(b) Conditional upon Loop 1 Counter Finished = 1
(c) Conditional upon Loop 2 Counter Finished = 1
(d) Conditional upon Delay Counter Finished = 1
(e) Conditional upon Stream Finished = 1
(f) Conditional upon Loop 2 Counter 2Under = 1
(*) Jump directly to next instruction

These signals from the control unit connect with a number of multiplexers in the top level VHDL file to switch data paths in the system. The multiplexers and their associated data paths are shown in Table 10.

Table 10: Top Level Multiplexers

Signal	Function	Data Paths
mux_mem_data_in	Memory input	0: memory_data_in = input_data; 1: memory_data_in(23 - 0) = regs_data_out_a 2: memory_data_in(23 - 0) = regv_data_out_b 3: memory_data_in = alu_mlacs_y (From wavelet mul/accum)
mux_mem_rd_addr	Memory read address	0: memory_rd_addr = pc_output 1: memory_rd_addr = ir_output 2: memory_rd_addr = control_mul_add_result (Mul/add unit handling vector addresses) 3: memory_rd_addr = regv_data_out_a (Sparse addresses stored in vector)
mux_mem_wr_addr	Memory write address	0: memory_wr_addr = input_addr; 1: memory_wr_addr = ir_output 2: memory_wr_addr = control_mul_add_result 3: memory_wr_addr = regs_data_out_b
mux_regs_data_in	Scalar register input	1: regs_data_in = memory_data_out 2: regs_data_in = ir_output 3: regs_data_in = alu_y
mux_regs_rd_addr_a	Scalar register read address A	0: regs_rd_addr_a = ir_output(3 - 0) 1: regs_rd_addr_a = ir_output(26 - 23) 2: regs_rd_addr_a = ir_output(13 – 10) 3: regs_rd_addr_a = ir_output(21 - 18)
mux_regs_rd_addr_b	Scalar register read address B	0: regs_rd_addr_b = ir_output(17 -14) 1: regs_rd_addr_b = ir_output(3 - 0)
mux_regv_data_in	Vector register input	0: regv_data_in = memory_data_out 1: regv_data_in = alu_y
mux_delay_max	Delay counter input	0: delay_max = 4 (Delay when reading from memory) 1: delay_max = 2 (Delay when writing to memory or in the wavelet mul/accum operation)
mux_alu_a	Primary ALU A input	1: alu_a = regs_data_out_a 2: alu_a = regv_data_out_a 3: alu_a = memory_data_out
mux_alu_b	Primary ALU B input	1: alu_b = regs_data_out_b 2: alu_b = regv_data_out_b
mux_regv_rd_b	Vector Register Read Address B	0: regv_rd_num_b = ir_output(2 - 0) regv_rd_off_b<= loop1_count 1: regv_rd_num_b = ir_output(16 - 14) regv_rd_off_b = loop2_count 2: regv_rd_num_b = ir_output(16 - 14) regv_rd_off_b = loop1_count

5.2 VHDL Design Hierarchy and Device Utilization

The VHDL programming language allows systems to be built in a hierarchy from high-level to low-level. In this implementation, 'cpu' was the top-level module, and it instantiated a number of lower level modules such as the control unit and ALU. Shown below in Figure 35 is the actual hierarchy of the implemented system. Also listed on this hierarchy is the device utilization by module, which shows, for example, how the multi-port minimum-latency scalar register set at 536 logic cells occupies more reconfigurable fabric than the primary ALU (which, to its credit, uses hardwired DSP blocks on chip to save logic cells for other uses).

CPU

- Control Unit – 106 logic cells, 32 registers
- Fetch Unit – 34 logic cells, 2 registers
    - Instruction memory – 4096 32-bit words
- Main Memory – 32768 48-bit words
- Vector Register Set – 24 logic cells
    - Memory (1/2) – 256 24-bit words
    - Memory (2/2) – 256 24-bit words
- Scalar Register Set – 536 logic cells, 384 registers
- Instruction Register – 32 logic cells, 32 registers
- Program Counter (1/2) – 14 logic cells
- Program Counter (2/2) – 14 logic cells
- Primary ALU – 168 logic cells, 142 registers, 8 DSP elements
- Secondary ALU – 310 logic cells, 144 registers, 6 DSP elements
    - Multiply / Accumulate (1/3)
    - Multiply / Accumulate (2/3)
    - Multiply / Accumulate (3/3)
- Loop Counter (1/3) – 30 logic cells, 16 registers
- Loop Counter (2/3) – 35 logic cells, 16 registers
- Loop Counter (3/3) – 18 logic cells, 7 registers
- Input / Output Simulator – 68 logic cells, 60 registers
    - Memory – 1024 32-bit words *(for simulation only)*

Figure 35: Design Hierarchy and Device Utilization by Module

The Quartus-produced chip placement summary for the Statix EP1S80F1020C6 device that was used in the design is shown in Figure 36.

Total Logic Elements: 1,849 / 79,040 (2%)

Logic cells	1,849 / 79,040 (2 %)
Registers	903 / 83,614 (1 %)
I/O pins	397 / 781 (50 %)
Clock pins	2 / 20 (10 %)
Global clocks	3 / 16 (18 %)
Global signals	3

Memory:

M512s	0 / 767 (0 %)
M4Ks	44 / 364 (12 %)
M-RAMs	3 / 9 (33 %)
Total memory bits	1,748,992 / 7,427,520 (23 %)
Total RAM block bits	1,972,224 / 7,427,520 (26 %)

On-chip DSP blocks:

(Number Used - Available per Block - Max Available)			
Simple Multipliers (9-bit)	3	8	176
Simple Multipliers (18-bit)	4	4	88
Simple Multipliers (36-bit)	1	1	22
Multiply Accumulators (18-bit)	0	2	44
Two-Multipliers Adders (9-bit)	0	4	88
Two-Multipliers Adders (18-bit)	0	2	44
Four-Multipliers Adders (9-bit)	0	2	44
Four-Multipliers Adders (18-bit)	0	1	22
DSP Blocks	5	--	22
DSP Block 9-bit Elements	19	8	176

Figure 36: Overall Device Utilization

Clearly, substantial space remains on the FPGA. Although this is the largest Stratix device (chosen primarily for on-chip memory), only 2 percent of the reconfigurable fabric is being used! Thus, substantial opportunities are available to increase the number of parallel pipelines in this processor in future design revisions.

Also of interest in the above figure is the utilization of on-chip memory resources, particularly the M-RAM blocks. These blocks are the largest available on-chip, and are 33% filled with the storage for 2 128x128 images. Unfortunately, then, to process a larger

(and perhaps more realistic) image of 512x512, external memories will be needed to store the pixel data and intermediate compression values.

5.3 Instruction Performance

Upon completing the processor implementation, the number of cycles necessary to execute each instruction was measured, as shown in Table 11.

Table 11: Instruction Cycle Counts

Instruction:	**Cycles:**	**Description:**	**Notes:**
LDS	3	Scalar Load Direct	
LDSI	1	Scalar Load Immediate	
STS	1	Scalar Store Direct	
STSI	1	Scalar Store Indirect	
LDV	6	Vector Load	Latency only (add vector length)
STV	4	Vector Store	Latency only (add vector length)
MLACS	10	Vector Sparse Mul & Accumulate	Latency only (add vector length)
ALUVS__	6	Vector ALU Scalar	Latency only (add vector length)
ALUVV__	10	Vector ALU Vector	Latency only (add vector length)
ALUSS__	3	Scalar ALU Scalar	
JMP	4	Jump	
BRT__	4	Jump if Comparison true	

These performance numbers will be used in the next chapter to benchmark the processor against its real-world wavelet image transformation application.

Chapter 6

Vector Processor Simulation

After completing the processor implementation and verifying each instruction independently, additional testing was performed to verify its suitability to perform the wavelet transform in a real-time system.

6.1 Target Application: Wavelet Transform

First, a wrapper program was written in Matlab to accelerate the design testing. This wrapper program, listed below, opens a color image and produces a stream of hex values that simulates the input stream of a new image frame to the processor. These pixels are 48 bits long and consist of 3 packed 16-bit color planes in sequence. When executed, the program produces a hex entry in the format of "00F700FA00F7", which is 48 bits long. One word is output per line. The output from Matlab is then copied into the Memory Initialization window in the Quartus software to initialize the processor simulation with data.

Matlab Wrapper Program

```
% Wrappper file for HW vector processor

% Part 1:
% Open a color image and produce a stream of hex values that simulate
%   the input stream to the processor of a new image frame. These
%   pixels are 48-bits long and consist of packed 16-bit color planes:
%   ------------------------------------------------
%   | 16-bits Plane1 | 16-bits Plane2 | 16-bits Plane3 |
%   ------------------------------------------------
% In this file, these planes are RGB, but they could be YUV or anything else
% In this file, the source image pixels values are positive integers
%   in the range of 0-255, but the HW supports signed integers up to 16-bits.
%   (useful for the wavelet transform which expands the dynamic range)

% Part 2:
% Open a file containing packed hex digits (output from the vector processor)
% Unpack and display them on screen.
```

```
close all; clear all; clc;

image_filename = '01_16x16.bmp';
hex_filename = 'hex_input.txt';

% ** Part 1 **

original_image = double(imread(image_filename));
size_rows = size(original_image,1);
size_columns = size(original_image,2);

for i=1:size_rows
    for j=1:size_columns

        % Grab a single pixel from each of the three color planes
        pix_1 = original_image(i,j,1);
        pix_2 = original_image(i,j,2);
        pix_3 = original_image(i,j,3);

        % Convert integer to hex representation, 16 bits wide
        pix_1_hex = dec2hex(pix_1,4);
        pix_2_hex = dec2hex(pix_2,4);
        pix_3_hex = dec2hex(pix_3,4);

        % Pack the color planes into a single 48-bit word
        packed_pixel = [pix_1_hex pix_2_hex pix_3_hex];

        % Print the packed pixel
        % (copy and paste from output into Quartus MIF editor)
        disp(packed_pixel)

    end
end

% ** Part 2 **

% Assume that the size of this image data is the same as the
% image we just outputed above.
hex_image = zeros(size_rows,size_columns,3);   % Make empty color image
file_array = textread(hex_filename,'%c');      % Read in entire file to array

k = 1;
for i=1:size_rows
    for j=1:size_columns

             % Convert 4 hex digits (packed pixel) to integer
             pix_1 = hex2dec(file_array(k:k+3)');
             pix_2 = hex2dec(file_array(k+4:k+7)');
             pix_3 = hex2dec(file_array(k+8:k+11)');

        % Save each color plane to the matrix
        hex_image(i,j,1) = pix_1;
        hex_image(i,j,2) = pix_2;
        hex_image(i,j,3) = pix_3;
             k = k+12;

    end
end

% Display the imported image
image(uint8(hex_image));
```

Further, once the processor simulation using the new image has completed, the memory contents in the simulation can be copied into a text file, and then re-imported using the second half of the same wrapper program shown above. The wrapper program re-imports the packed hexadecimal values and displays the resulting image on screen. Thus, it is easy to compare the results of the simulation with the results of the original Matlab test program for verification.

Once the wrapper program was written, the actual wavelet transform assembly program was written and assembled using the WinTim table driven assembler. The custom definition file written for this assembler is in Appendix A-1. The wavelet assembly program is in Appendix A-2.

6.2 Simulation Results

The wavelet compression program was loaded into the memory initialization file of the processor, and the simulation was started. After the simulation was completed, the current contents of the data memory was copied to a text file and fed back through the Matlab wrapper program to analyze the results.

Before the processor simulation was started, the original Matlab wavelet program was run on the sample 16 x 16 test image shown in Figure 37.

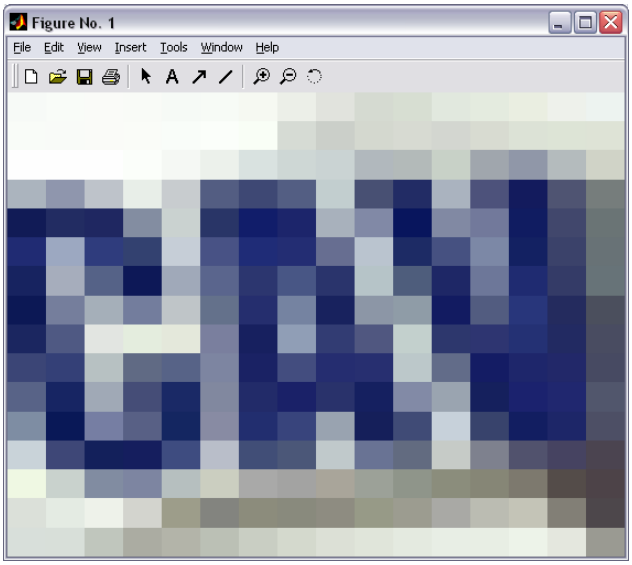

Figure 37: Original Image (16 x 16) (Part of UCAV logo on airframe)

The small image was chosen to keep simulation times to a minimum. The Matlab program produced the 1-step transformation image shown in Figure 38.

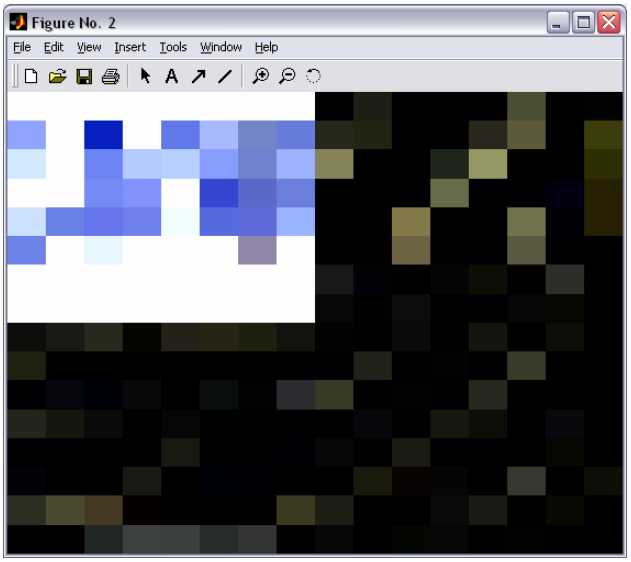

Figure 38: Transformed Image (via Matlab program)

This image is nearly identical to that produced by the vector processor in the Quartus simulation, as shown in Figure 39. When comparing the actual coefficient values, there were only slight differences (less than 0.1%) that were attributable to the intergerization method discussed in the wavelet chapter.

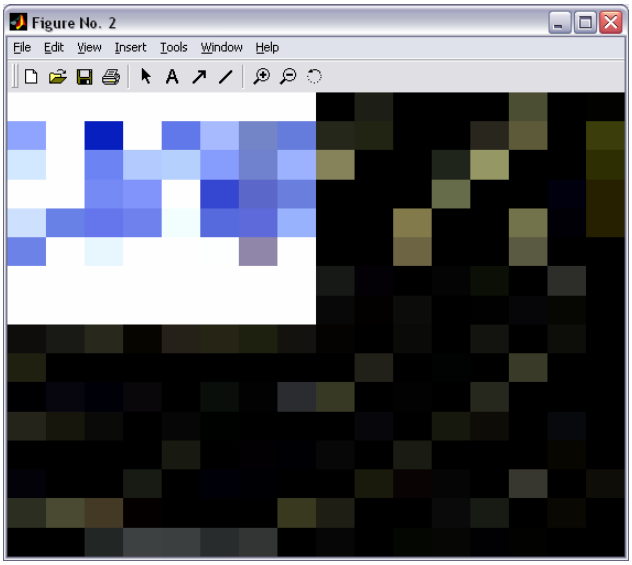

Figure 39: Transformed Image (via Vector Processor)

Clearly, the processor has achieved its goal of functional equivalence with the Matlab prototype algorithm.

6.3 Performance Analysis

In addition to the determination of the instruction cycle counts (Table 11), other key performance parameters are summarized in Table 12

Table 12: Key Performance Parameters

Clock Rate (MHz):	75
Cycle Time (sec):	1.33333E-08
Wavelet Length:	4
Image Size:	256
Color Planes:	1
Frames per Second:	30

These parameters that affect the system performance include the clock rate, wavelet length (longer is slower), the image size, and number of desired frames per second. Note that the "color planes" entry is set to 1, as the processor handles all color planes simultaneously via the packed pixel method previously discussed.

Using these parameters and a knowledge of the wavelet transformation program, it is possible to accurately account for every cycle executed during the transform. This is shown in Figure 40.

Per Image Initialization Costs (single stage of 2 stages)		
	Instr. Count	Total Cycles
LDSI	4	20
LDV	2	20
Import/Export	N/A	65,536
	= Cycles/Image	65,576
Single Transform Row (single stage of 2 stages, 1 MR Level):		
	Instr. Count	Total Cycles
MLACS	256	3,584
ALUSSxx	258	774
ALUVSxx	258	2,580
BRT	129	516
	=Total Cycles/Row:	7,454
	x Image Rows	256
	x second stage	2
	= Cycles/Image	3,816,448
	= Total Cycles/Image	3,882,024
	x Clock Period	1.3333E-08
	x Color Planes	1
	x Frames per sec	30
	= Total Execution Time (sec)	1.5528096
	= Frames per Second:	19.3198187

Figure 40: Transformation Program Performance

Thus, as shown above, the processor does not achieve real-time performance on a 256x256 test image (19.3 frames per second versus the desired 30).

The vector processor performance was also compared to another technique used in computing the wavelet transform: a streaming filter bank. This approach was implemented by Bo Qiang for an older Altera Apex FPGA. To more easily compare results, his design was recompiled for the same Stratix-series FPGA used in the vector

processor design. The streaming approach occupied 2,275 of the 79,040 logic elements on the FPGA, and had a maximum clock rate of 98.77 MHz. This allowed the computation of nearly 30 frames/sec for a 512x512 pixel grayscale image, or 10 frames/sec for an equivalently sized color frame. This compares favorably to the vector processor, which achieved approximately 5 frames/sec on the same large test image with a device utilization of 1,849 logic elements (less than 2%).

There are two reasons why the streaming approach yielded higher performance. First, the design had a significantly higher clock rate, which is likely due to its lower routing and switching complexity. Second, there is less overhead during execution using the filter bank method. Unlike the vector processor, the streaming approach is hard-wired to continually execute a predefined sequence of operations. Thus, it does not have to increment pointers to data in memory, compare data values, and execute program branches. Further, it can utilize a continuous stream of data from memory, instead of starting and stopping as intermediate control instructions interrupt the data flow.

Despite some of these drawbacks, the vector processor approach has significantly greater flexibility to both perform a wide variety of wavelet transforms, in addition to a whole suite of other algorithms that can use vector data. Thus, it is worth pursuing to maintain application design flexibility. In the final chapter, a variety of architectural improvements are discussed to improve the performance of the processor.

Chapter 7

Processor Transition

In this thesis, a vector processor was designed and implemented. This processor can compute the wavelet transform, the fundamental component of wavelet video compression. It is not, however, the only component. As described in Chapter 2, other modules including a quantizer and several encoders are needed to actually compress a video stream.

Because the vector processor occupied only 2% of the largest Altera Stratix FPGA, it is proposed to instantiate these additional modules onto the same FPGA fabric, as shown in Figure 41.

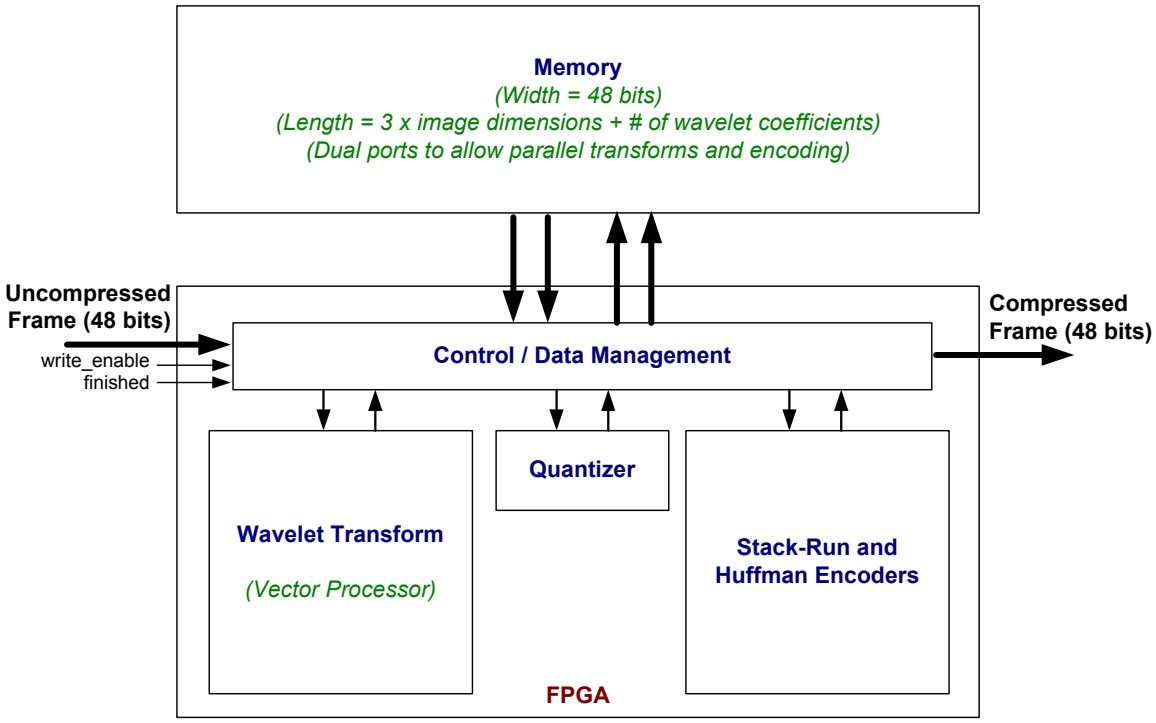

Figure 41: Proposed System Implementation

Here, the incoming video stream would enter the FPGA and be saved by the control unit to an external memory module. (Recall that the on-chip memory resources are too limited to support image sizes in excess of 128 x 128.) This memory module would likely be interleaved to support 4 parallel memory ports (2 read and 2 write), which would allow for simultaneous operation of both the vector processor and encoder modules. The memory needs to be at least 3 times the desired image dimensions to allow sufficient space for temporary copies of the working data set. As in the initial design, the stream instruction would be used to synchronize data (both raw and transformed) with the external system. Through this design, a single FPGA should be able to perform the entire wavelet compression operation on the incoming data stream.

Chapter 8

Conclusions & Future Work

In this thesis, a scalable vector processor was designed and implemented using VHDL on an Altera Stratix-series FPGA. The primary application of this processor is to compute the wavelet transform, which is the fundamental component of a wavelet video compression system. Other key compression modules, such as quantization and encoding, were not implemented in this thesis.

Two categories of future work are proposed for this processor: (1) Technology transition efforts and (2) Performance enhancement efforts. The first, technology transition, involves upgrading this design to a complete compression system. That effort will involve adding quantization and encoding modules, as well as designing a memory system big enough to handle the desired image size. Specific details on this effort are currently limited and vary greatly depending on the specific target application.

The other future work category, performance enhancement, however, could be done immediately without waiting on further information. When the processor was programmed in VHDL and fitted on an Altera Stratix FPGA, less than 2% of the reconfigurable resources were used. The current performance of the processor at 75 MHz was sufficient to allow the full transformation of 19+ 256x256 color images per second. Although this is not quite real-time video, several options exist to improve device performance given that substantial FPGA fabric remains unused.

First, the number of dedicated dot-product arithmetic units could be increased from the current three to six, nine, twelve, or even more. These are used for the wavelet multiply and accumulate operation, which performs the useful work in the wavelet transform program. Doubling the number of these arithmetic units would halve the execution time of the program, since twice the effective (useful) work could be

accomplished with the same amount of non-productive overhead as before. However, the memory and vector register subsystems would need to be upgraded to provide the necessary bandwidth to operate all of these execution units in parallel. Specifically, the vector register bank and memory would need to be widened to allow more data to be packed into a single word. Alternatively, the number of read and write ports would need to be increased through banking techniques to support increased numbers of simultaneous accesses.

If the number of data access ports was increased, additional parallelisms in the vector architecture could be exploited. In addition to adding dedicated dot-product ALUs, additional full-featured arithmetic units could also be added to compute other vector operations in parallel. These general-purpose ALUs are used for the remaining vector mathematical or logical operations. Adding 3 more general-purpose arithmetic units to the single primary ALU already in place would allow 4 adjacent vector elements to be computed in parallel in a single clock cycle. Since the wavelet used in the test program (the Daubechies 4) is only 4 elements long, this would enable it to be manipulated in a single cycle plus latency instead of 4 cycles plus latency. Longer wavelets (or vectors in general) could still be handled through "strip mining" batches in groups of 4. To achieve this performance improvement, the vector register set would need to be interleaved in 4 separate banks to support sufficient read ports to keep all arithmetic units filled with data.

Another possible performance enhancement would be to instantiate multiple copies of this processor to run in parallel. They could either be linked via a single instruction memory and thus operate in lockstep, or be completely independent and operate on different parts of the image entirely. To fully support independent operation, the number of memory read and write ports would need to be doubled. Or, to support lock-step operation, the memory width could be increased to 96 bits, and two pixels from all color planes packed into a word instead of just one.

The clock rate of the processor could likely be enhanced by creating specialized modules for the computation of effective memory addresses in vector instructions. These

are currently handled by general purpose adders with moderate latency, and have not been fully optimized to exploit the predictability of the memory addresses.

One area that could be studied is the feasibility of processing the wavelet transformation in tiles as is done with the DCT in the JPEG standard. These tiles could be large (128 x 128), but still a fraction of the proposed image size (512 pixels or above). If this method could be shown to retain acceptable quality compared with applying the wavelet over the entire image, it would have several key advantages. First, it would lower the on-chip memory requirements. Second, the finer grain of the tiles more closely approximates a "streaming" filtering technique that is very desirable when doing real-time compression, as data will be available much sooner to send to subsequent modules such as the quantizer or encoder.

Any or all of these design expansions along with other clock-speed related optimizations should boost performance to real-time levels. Further, last minute tests on the just-released Altera Stratix-II FPGAs yielded a 33%+ improvement in clock rate without any design changes at all. This fabric was able to boost the processor to 106+ Mhz. Thus, it should be easy to foresee real-time performance of this processor architecture in the near future.

References

1. AMD (Advanced Micro Devices), "3DNow!™ Technology Manual", March 2000, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/21928.pdf
2. Asanovic, K. (Advisors: Wawrzynek, Patterson, Wessel), "Vector Microprocessors," Ph.D. Thesis, Computer Science Division, University of California at Berkeley, May 1998
3. Bailey, D.H., "Extra-High Speed Matrix Multiplication on the Cray-2", *SIAM Journal on Scientific and Statistical Computing*, Vol. 9, No. 3, May 1988
4. Balster, E.J., Scarpino, F.A., and W.W. Smari, "Wavelet Transform for Real-Time Image Compression Using FPGAs," *12th IASTED International Conference on Parallel and Distributed Computing and Systems*, Las Vegas, Nevada, Nov. 6 – 9, 2000, pp. 232-238.
5. Cucchiara, R., Piccardi, M, and A. Prati, "Exploiting Cache in Multimedia", *IEEE International Conference on Multimedia Computing and Systems*, , pp. 345 – 350, June 7, 1999
6. Daubechies, I., *Ten Lectures on Wavelets*, Society for Industrial and Applied Mathematics, 1992
7. Espasa, R. and M. Valero, "Vector Architectures: Past, Present and Future", *Proceedings of the 2ne International Conference on Supercomputing*, pp. 425-432, Melbourne, Australia, July 1998
8. Flynn, M.J., *Computer Architecture: Pipelined and Parallel Processor Design*, Jones and Bartlett Publishers, Boston, 1995
9. Gee, J.D. and A.J. Smith, "The Performance Impact of Vector Processor Caches", *Proceedings of the Annual Hawaii International Conference on System Sciences*, pp. 437 – 448, January 7, 1992
10. Goswami, J.C. and Chan, A.K, *Fundamentals of Wavelets: Theory, Algorithms and Applications*, John Wiley & Sons, New York, 1999

11. Hennessy, J.L. and Patterson, D.A., *Computer Architecture: A Quantitative Approach, 3rd Edition*, Morgan Kaufman Publishers, San Francisco, 2003
12. Hockney, R.W. and C.R. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms, 2nd Edition*, IOP Publishing Limited, Bristol, England, 1998
13. Huss-Lederman, S, et al., "Strassen's Algorithm for Matrix Multiplication: Modeling, Analysis, and Implementation", Technical Report CCS-TR-96-147, *Center for Computing Sciences*, November 1996.
14. Intel Corporation, "IA-32 Intel® Architecture Software Developers Manual", http://www.intel.com/design/Pentium4/manuals/24547012.pdf
15. Jordan, H.F. and Alaghband, G., *Fundamentals of Parallel Processing*, Prentice Hall, Pearson Education Ltd., Upper Saddle River, New Jersey, 2003
16. Kaxiras, S., "Distributed Vector Architectures", *Journal of Systems Architecture* Volume: 46, Issue: 11, pp. 973-990, September, 2000
17. Lee, C.G. and M.G. Stoodley, "Simple Vector Microprocessors for Multimedia Applications", *Proceeding of the 31st IEEE Annual International Symposium on Microarchitecture*, pp. 25 – 36, December 1998
18. Marksteiner, P., "High-Performance Computing", *Computer Physics Communications*, Volume: 97, Issue: 1-2, pp. 16-35, August 2, 1996
19. Mathew, B.K, McKee, S.A., Carter, J.B., and A. Davis, "Design of a Parallel Vector Access Unit for SDRAM Memory Systems", Proceedings of the 6th IEEE International Symposium on High-Performance Computer Architecture, Toulouse, France, p39, January 8-12, 2000
20. Qiang, Bo, et al., "Image Coding Based on One-Step Wavelet Transform", Proceedings of the ISCA 19th International Conference on Computers and their Applications (CATA-2004), Seattle, Washington, March 18-20, 2004
21. Quinn, M.J, *Parallel Computing: Theory and Practice, 2nd Edition*, McGraw-Hill, Inc., New York, 1994
22. Santa-Cruz, D, T. Ebrahimi, J. Askelöf, M. Larsson and C. A. Christopoulos, "JPEG 2000 Still Image Coding Versus Other Standards," *Proceedings of SPIE*

45th Annual Meeting, Applications of Digital Image Processing XXIII, Vol. 4115, San Diego, CA, July 30-August 4, 2000, pp. 446-454

23. Shafer, J. and F. Scarpino, "A Case for a Collaborative Computing Tool for Image Processing", The 2004 International Symposium on Collaborative Technologies and Systems (CTS 2004), San Diego, CA, January 18-23, 2004
24. Shanley, Tom, *Pentium Pro and Pentium II System Architecture, 2nd Edition*, Addison-Wesley, Reading, Massachusetts, 1998
25. Stone, H.S., *High-Performance Computer Architecture, 3rd Edition*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1993
26. Subramanya, S.R., "Image Compression Techniques," *IEEE Potentials*, Vol. 20, No. 1, February/March 2001
27. Sun Microsystems, "VIS Instruction Set Whitepaper", June 2002, http://www.sun.com/processors/whitepapers/vis_wp_external.pdf
28. Topiwala, P.N., *Wavelet Image and Video Compression*, Kluwer Academic Publishers, 1998
29. Tsai, M, Villasenor, J. and F. Chen, "Stack-run image coding," *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 6, pp. 519-521, October 1996
30. Villasenor, J., Belzer, B. and J. Liao, "Wavelet Filter Evaluation for Image Compression", *IEEE Transactions on Image Processing*, Vol. 4, No. 8, pp. 1053-1060, August 1995
31. Welstead, S, *Fractal and Wavelet Image Compression Techniques*, SPIE Optical Engineering Press, 1999

Appendices

A-1 Table Driven Assemble Definition File

```
; mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
; ECE 599 - Thesis
;
; Instruction Set Definition Table - Vector Processor
; mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm

TITLE Assembly Language Definition File - Vector Processor Design
WORD  32                ; Word length is 32 bits...
WIDTH       100         ; Listing file character width is 100...
LINES       50          ; Listing file has 50 lines per page...


; mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
; Data Pseudo-Ops
; mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm

DW:       DEF     32VH#0000  ;32-BIT DATA DIRECTIVE

; mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
; Scalar Register Equates
; mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
R0:   EQU B#0000
R1:   EQU B#0001
R2:   EQU B#0010
R3:   EQU B#0011
R4:   EQU B#0100
R5:   EQU B#0101
R6:   EQU B#0110
R7:   EQU B#0111
R8:   EQU B#1000
R9:   EQU B#1001
R10:  EQU B#1010
R11:  EQU B#1011
R12:  EQU B#1100
R13:  EQU B#1101
R14:  EQU B#1110
R15:  EQU B#1111

; mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
; Vector Register Equates
; mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
VR0:  EQU B#000
VR1:  EQU B#001
VR2:  EQU B#010
VR3:  EQU B#011
VR4:  EQU B#100
```

```
VR5:  EQU B#101
VR6:  EQU B#110
VR7:  EQU B#111

; mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
; Instruction Elements
; (Common pieces of instruction formats)
; mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm

; Instruction Op Codes

ONOOP:            EQU         B#00000
OLDS:             EQU         B#00001
OLDSI:            EQU         B#00010
OSTS:             EQU         B#00011
OSTSI:            EQU         B#01110
OLDV:             EQU         B#00100
OSTV:             EQU         B#00101
OMLAC:            EQU         B#00110
OMLACS:           EQU         B#00111
OALUVS:           EQU         B#01000
OALUVV:           EQU         B#01001
OALUSS:           EQU         B#01010
OBRT:             EQU         B#01011
OJMP:             EQU         B#01100
OSTRM:            EQU         B#01111


; ALU Control Signals

ALUNOOP:          EQU         B#00000
ALUADD:           EQU         B#00001
ALUSUB:           EQU         B#00010
ALUMUL:           EQU         B#00011
ALUAND:           EQU         B#00100
ALUOR:            EQU         B#00101
ALUXOR:           EQU         B#00110
ALUNOT:           EQU         B#00111
ALUNEG:           EQU         B#01000
ALUSLL:           EQU         B#01001
ALUSRL:           EQU         B#01010
ALUSLA:           EQU         B#01011
ALUSRA:           EQU         B#01100
ALUSLC:           EQU         B#01101
ALUGT:            EQU         B#01110
ALUGTE:           EQU         B#01111
ALULT:            EQU         B#10000
ALULTE:           EQU         B#10001
ALUEQ:            EQU         B#10010
ALUNE:            EQU         B#10011


; Undefined section of instruction
EMPTY1:           EQU         B#0
EMPTY10:    EQU         B#0000000000

; mmmmmmmmmmmmmmmmmmmm
```

```
; Instruction Formats
; mmmmmmmmmmmmmmmmmmmmmm

NOOP:       DEF  ONOOP,  ALUNOOP, 5H#00000, 2B#00
LDS:        DEF  OLDS,   23VB#00000000000000000000000, 1VH#0
LDSI:       DEF  OLDSI,  23VB#00000000000000000000000, 1VH#0
STS:        DEF  OSTS,   23VB#00000000000000000000000, 1VH#0
STSI:       DEF  OSTSI,  B#000000000, 4VB#0000, B#0000000000, 4VB#0000
LDV:        DEF  OLDV,   23VB#00000000000000000000000, EMPTY1, 3VB#000
STV:        DEF  OSTV,   23VB#00000000000000000000000, EMPTY1, 3VB#000
MLACS:      DEF  OMLACS, ALUMUL, B#0, 3VB#000, B#0, 3VB#000, 4VB#0000,
B#000000, 4VB#0000
ALUVSADD:   DEF  OALUVS, ALUADD, B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVSSUB:   DEF  OALUVS, ALUSUB, B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVSMUL:   DEF  OALUVS, ALUMUL, B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVSAND:   DEF  OALUVS, ALUAND, B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVSOR:    DEF  OALUVS, ALUOR,  B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVSXOR:   DEF  OALUVS, ALUXOR, B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVSNOT:   DEF  OALUVS, ALUNOT, B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVSNEG:   DEF  OALUVS, ALUNEG, B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVSSLL:   DEF  OALUVS, ALUSLL, B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVSSRL:   DEF  OALUVS, ALUSRL, B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVSSLA:   DEF  OALUVS, ALUSLA, B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVSSRA:   DEF  OALUVS, ALUSRA, B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVSSLC:   DEF  OALUVS, ALUSLC, B#0, 3VB#000, 4VB#0000, B#00000000000,
3VB#000
ALUVVADD:   DEF  OALUVV, ALUADD, B#0, 3VB#000, B#0, 3VB#000,
B#00000000000, 3VB#000
ALUVVSUB:   DEF  OALUVV, ALUSUB, B#0, 3VB#000, B#0, 3VB#000,
B#00000000000, 3VB#000
ALUVVMUL:   DEF  OALUVV, ALUMUL, B#0, 3VB#000, B#0, 3VB#000,
B#00000000000, 3VB#000
ALUVVAND:   DEF  OALUVV, ALUAND, B#0, 3VB#000, B#0, 3VB#000,
B#00000000000, 3VB#000
ALUVVOR:    DEF  OALUVV, ALUOR,  B#0, 3VB#000, B#0, 3VB#000,
B#00000000000, 3VB#000
ALUVVXOR:   DEF  OALUVV, ALUXOR, B#0, 3VB#000, B#0, 3VB#000,
B#00000000000, 3VB#000
ALUVVNOT:   DEF  OALUVV, ALUNOT, B#0, 3VB#000, B#0, 3VB#000,
B#00000000000, 3VB#000
ALUVVNEG:   DEF  OALUVV, ALUNEG, B#0, 3VB#000, B#0, 3VB#000,
B#00000000000, 3VB#000
ALUSSADD:   DEF  OALUSS, ALUADD, 4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
```

```
ALUSSSUB:   DEF  OALUSS, ALUSUB, 4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
ALUSSMUL:   DEF  OALUSS, ALUMUL, 4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
ALUSSAND:   DEF  OALUSS, ALUAND, 4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
ALUSSOR:    DEF  OALUSS, ALUOR,  4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
ALUSSXOR:   DEF  OALUSS, ALUXOR, 4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
ALUSSNOT:   DEF  OALUSS, ALUNOT, 4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
ALUSSNEG:   DEF  OALUSS, ALUNEG, 4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
ALUSSSLL:   DEF  OALUSS, ALUSLL, 4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
ALUSSSRL:   DEF  OALUSS, ALUSRL, 4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
ALUSSSLA:   DEF  OALUSS, ALUSLA, 4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
ALUSSSRA:   DEF  OALUSS, ALUSRA, 4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
ALUSSSLC:   DEF  OALUSS, ALUSLC, 4VB#0000, 4VB#0000, B#0000000000,
4VB#0000
BRTGT:      DEF  OBRT, ALUGT,  4VB#0000, 4VB#0000, 14VB#00000000000000
BRTGTE:     DEF  OBRT, ALUGTE, 4VB#0000, 4VB#0000, 14VB#00000000000000
BRTLT:      DEF  OBRT, ALULT,  4VB#0000, 4VB#0000, 14VB#00000000000000
BRTLTE:     DEF  OBRT, ALULTE, 4VB#0000, 4VB#0000, 14VB#00000000000000
BRTEQ:      DEF  OBRT, ALUEQ,  4VB#0000, 4VB#0000, 14VB#00000000000000
BRTNEQ:     DEF  OBRT, ALUNE,  4VB#0000, 4VB#0000, 14VB#00000000000000
JMP:        DEF  OJMP, B#0000000000000, 14VB#00000000000000
STRM:       DEF  OSTRM, B#00000000000000000000000000000
```

Appendices

A-2 Wavelet Transform Program

```
; mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
; Wavelet Transform, 1 MR Level, 16x16 image
; mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm


LIST A       ; List addresses on every line rather than just on lines
             ; where object code is listed
LIST B       ; List formatted object code in block following end of source code
LIST F       ; List formatted object code to left of source code if S is on.


; ** PROGRAM **
; Memory module 1
; No data allowed here, only instructions
; All memory reads/writes under program control
; occur in memory module 2
ORG H#00            ; Orign = 0

   ; ** INITIALIZATION **

   NOOP       ; No instruction (sometimes the simulator "cheats" and makes this
              ; first memory spot available early. We restore honesty to
              ; the simulation by ignoring the location entirely)

   ; Define some compiler constants
   cols:      EQU    16     ; Width of image
   rows:      EQU    7      ; Row counter (equal to 1/2 the height - 1)
                            ; The last row is handled separately
   shift:     EQU    6      ; Bit shifting used to "integerize" wavelet transform

   ; Note: Tag "%:" right-justifies and truncates address
   LDSI 1%:,R14             ; Load VSR with stride=1
   LDSI 4%:,R15             ; Load VLR with length=4

   ; Stream in new image
   STRM

   ; Load in all 4 wavelet coef vectors into separate vector registers
   LDV xform1%:,VR1
   LDV xform2%:,VR2
   LDV xform3%:,VR3
   LDV xform4%:,VR4

   LDSI shift%:, R1  ; Load in the shift amount to shift right in
              ; the mul&accumulate operation.
              ; We need this to offset the amount we shifted our transform coefs
              ; to the left to "integerize" them.

   LDSI 1%:, R10     ; Load value 1 into register for future use
                     ; (incrementing/decrementing counters)
   LDSI 0%:, R11     ; Load value 0 into register for future use
                     ; (comparison value for loops)
```

```
   LDSI cols%:,R12  ; Load value 16 (image width) into register for future use

   ; Load in two destination pointers, one at the start of the storage
   ; space in memory and one halfway through (assume the storage space
   ; holds a whole frame) These pointers store intermediary results.
   LDSI image2%:, R2       ; Intermediary results, "top half" of result matrix
   LDSI (image2+128)%:, R5 ; Intermediary results, "bottom half"
                           ; of result matrix

   ; Load in two vector registers with sparse addresses, i.e. the
   ; addresses in memory to pull data from in a mul&accumulate operation
   ; These addresses are all in the same column.
   LDV spaddr1%:,VR5       ; Sparse addresses for "top half"
   LDV spaddr1%:,VR6       ; Sparse addresses for "bottom half"
                           ; - start at same location

   ; ** THE WAVELET TRANSFORM, STAGE 1/2 **
   ; Outer loop advances the row of X  in X*Y
   ; Inner loop advances the column of Y in X*Y

   LDSI rows%:, R4         ; Initialize the row counter
sg1out: LDSI cols%:, R3    ; Initialize the column counter

   ; Compute a segment of a matrix multiply with a single row * column
   ; Via the sparse register VR5, we ignore coefs that are zero
   ; We compute the "top half" and "bottom half" inside the same loop.
   ; Loop and advance, same row, next column...
sg1in: MLACS VR5, VR1, R1, R2     ; Multiply Mem[VR5(x)] by VR1(x), shift
                           ; right by Reg[R1] positions, and accumulate
                           ; Save result in Mem[R2]
   ALUSSADD R2, R10, R2    ; Increment destination pointer (R2) by 1
                           ; (stored in R10)
   ALUVSADD VR5,R10,VR5    ; Increment elements of sparse register by
                           ; 1 to advance to next column

   ; Bottom-half intermediary results
   MLACS VR6, VR3, R1, R5  ; Multiply Mem[VR6(x)] by VR3(x), shift right
                           ; by Reg[R1] positions, and accumulate
                           ; Save results in Mem[R5]
   ALUSSADD R5, R10, R5    ; Increment destination pointer (R5) by 1
                           ; (stored in R10)
   ALUVSADD VR6,R10,VR6    ; Increment elements of sparse register by 1
                           ; to advance to next column

   ; Loop control
   ALUSSSUB R3, R10, R3    ; Decrement the column counter by 1
   BRTNEQ R3, R11, sg1in%: ; Does column counter (R3) != 0  If TRUE,
                           ; branch and continue stage 1 inner loop

   ALUVSADD VR5,R12,VR5    ; Increment elements of sparse register by
                           ;  <image width> to skip 2 rows down in same column
   ALUVSADD VR6,R12,VR6    ; Increment elements of sparse register by
                           ;  <image width> to skip 2 rows down in same column
   ALUSSSUB R4, R10, R4    ; Decrement the rows counter by 1
   BRTNEQ R4, R11, sg1out%: ; Does rows counter (R4) != 0. If TRUE, branch
                           ; and continue stage 1 outer loop


   ; The middle and bottom rows of this stage use the same coefs in a
   ; different order and with a difference sparse address (because the
   ; transform coefs wrap halfway around). The destination counters can
   ; keep counting in the sequence started in the previous loops
```

```
sg1btm:      LDSI cols%:, R3     ; Initialize the column counter

   ; Load in two vector registers with sparse addresses
   LDV spaddr2%:,VR5       ; Sparse addresses in "left side"
   LDV spaddr2%:,VR6       ; Sparse addresses in "right side" - start at same
location

   ; Compute a segment of a matrix multiply with a single row * column
   ; Via the sparse register VR5, we ignore coefs that are zero
   ; We compute the "top half" and "bottom half" inside the same loop.
   ; Loop and advance, same row, next column...
   MLACS VR5, VR1, R1, R2  ; Multiply Mem[VR5(x)] by VR1(x), shift right
                           ;  by Reg[R1] positions, and accumulate
                           ; Save result in Mem[R2]
   ALUSSADD R2, R10, R2    ; Increment destination pointer (R2) by 1
                           ; (stored in R10)
   ALUVSADD VR5,R10,VR5    ; Increment elements of sparse register by
                           ; 1 to advance to next column

   ; Bottom-half intermediary results
   MLACS VR6, VR3, R1, R5  ; Multiply Mem[VR6(x)] by VR3(x), shift right
                           ; by Reg[R1] positions, and accumulate
                           ; Save results in Mem[R5]
   ALUSSADD R5, R10, R5    ; Increment destination pointer (R5) by 1
                           ; (stored in R10)
   ALUVSADD VR6,R10,VR6    ; Increment elements of sparse register by 1
                           ; to advance to next column

   ; Loop control
   ALUSSSUB R3, R10, R3    ; Decrement the column counter by 1
   BRTNEQ R3, R11, sg1btm%: ; Does column counter (R3) != 0  If TRUE, branch
                           ; and continue stage 1 inner loop


 ; ** THE WAVELET TRANSFORM, STAGE 2/2 **

   ; Load in two destination pointers, one at the start of the storage space
   ; in memory and one halfway through (assume the storage space holds a
   ; whole frame) These pointers store intermediary results
   ; In this second stage we overwrite the original untransformed image
   LDSI image1%:, R2       ; Intermediary results, "left half" of result matrix
   LDSI (image1+128)%:, R5 ; Intermediary results, "right half" of result matrix

   ; Load in immediate value for later pointer arithmetic
   LDSI 238%:, R13         ; Used for sparse addresses
   LDSI 223%:, R7          ; Used for destination address

   ; Load in two vector registers with sparse addresses, i.e. the
   ; addresses in memory to pull data from in a mul&accumulate operation
   ; These addresses are all in the same column.
   LDV spaddr3%:,VR5       ; Sparse addresses for "left half"
   LDV spaddr3%:,VR6       ; Sparse addresses for "right half" - start at same
location

   LDSI cols%:, R3         ; Initialize the column counter
sg2out:LDSI rows%:, R4           ; Initialize the row counter

   ; Compute a segment of a matrix multiply with a single row * column
   ; Via the sparse register VR5, we ignore coefs that are zero
   ; We compute the "left half" and "right half" inside the same loop.
   ; Loop and advance, same column, next row...
sg2in: MLACS VR5, VR1, R1, R2     ; Multiply Mem[VR5(x)] by VR1(x),
                           ; shift right by Reg[R1] positions, and accumulate
```

```
                          ; Save result in Mem[R2]
   ALUSSADD R2, R12, R2   ; Increment destination pointer (R2) by
                          ; <image width> (stored in R12)
   ALUVSADD VR5,R12,VR5   ; Increment elements of sparse register by
                          ; <image width> to advance to next row

   ; Right-half intermediary results
   MLACS VR6, VR3, R1, R5 ; Multiply Mem[VR6(x)] by VR3(x), shift right
                          ; by Reg[R1] positions, and accumulate
                          ; Save results in Mem[R5]
   ALUSSADD R5, R12, R5   ; Increment destination pointer (R5) by
                          ; <image width> (stored in R12)
   ALUVSADD VR6,R12,VR6   ; Increment elements of sparse register by
                          ; <image width> to advance to next row

   ; Loop control
   ALUSSSUB R4, R10, R4   ; Decrement the rows counter by 1
   BRTNEQ R4, R11, sg2in%: ; Does rows counter (R4) != 0  If TRUE, branch
                          ; and continue stage 1 inner loop

   ALUVSSUB VR5,R12,VR5   ; Decrement elements of sparse register by
                          ;  238 (image width*height-length of 1 row - 2)
                          ; to return to first row, but 2 columns to the right
   ALUVSSUB VR6,R12,VR6   ; Decrement elements of sparse register by 238
   ALUSSSUB R2, R7, R2    ; Decrement destination address by 223 to
                          ;  return to first row, but 1 column over
   ALUSSSUB R5, R7, R2    ; Decrement destination address by 223 to return
                          ;  to first row, but 1 column over
   ALUSSSUB R3, R10, R3   ; Decrement the columns counter by 1
   BRTNEQ R3, R11, sg2out%: ; Does rows counter (R3) != 0. If TRUE, branch
                          ; and continue stage 1 outer loop

   ; The middle and bottom rows of this stage use the same coefs in a
   ; different order and with a difference sparse address (because the
   ; transform coefs wrap halfway around). The destination counters can keep
   ; counting in the sequence started in the previous loops

sg2btm:      LDSI rows%:, R4          ; Initialize the rows counter

   ; Load in two vector registers with sparse addresses
   LDV spaddr4%:,VR5             ; Sparse addresses in "left side"
   LDV spaddr4%:,VR6             ; Sparse addresses in "right side" -
                                 ; start at same location

   ; Compute a segment of a matrix multiply with a single row * column
   ; Via the sparse register VR5, we ignore coefs that are zero
   ; We compute the "top half" and "bottom half" inside the same loop.
   ; Loop and advance, same row, next column...
   MLACS VR5, VR1, R1, R2 ; Multiply Mem[VR5(x)] by VR1(x), shift right
                          ; by Reg[R1] positions, and accumulate
                          ; Save result in Mem[R2]
   ALUSSADD R2, R12, R2   ; Increment destination pointer (R2) by
                          ; <image width> (stored in R12)
   ALUVSADD VR5,R12,VR5   ; Increment elements of sparse register by
                          ; <image width> to advance to next column

   ; Bottom-half intermediary results
   MLACS VR6, VR3, R1, R5 ; Multiply Mem[VR6(x)] by VR3(x), shift right
                          ; by Reg[R1] positions, and accumulate
                          ; Save results in Mem[R5]
   ALUSSADD R5, R12, R5   ; Increment destination pointer (R5) by
                          ; <image width> (stored in R12)
   ALUVSADD VR6,R12,VR6   ; Increment elements of sparse register by
```

```
                          ; <image width> to advance to next column

   ; Loop control
   ALUSSSUB R4, R10, R4    ; Decrement the rows counter by 1
   BRTNEQ R4, R11, sg2btm%: ; Does rows counter (R4) != 0  If TRUE, branch
                          ;and continue stage 1 inner loop


; Some empty space to create a clear boundary in the assembler output
 DW 0%:
 DW 0%:
 DW 0%:
 DW 0%:
 DW 0%:
 DW 0%:
 DW 0%:


; ** DATA **
; Memory module 2
; Data only, no instructions
ORG H#00            ; Orign = 0

; Wavelet Transform Coefs - Stored in memory
; For negative numbers, we express -x as (0-x) ("feature" of this assembler)
;  1 and 2 are same coefs, just shifted to compute "bottom" row of region
;  3 and 4 are same coefs, just shifted to compute "bottom" row of region
xform1:      DW (0-8)%:
    DW 14%:
    DW 54%:
    DW 31%:
xform2:             DW 54%:
    DW 31%:
    DW (0-8)%:
    DW 14%:
xform3:             DW (0-31)%:
    DW 54%:
    DW (0-14)%:
    DW (0-8)%:
xform4:             DW (0-14)%:
    DW (0-8)%:
    DW (0-31)%:
    DW 54%:


; Memory addresses pointing to data used in sparse Mul/Accum (MLACS)
operations.
; These addresses are loaded into VR5 and VR6 which are used in the MLACS
operations.

; Stage 1 sparse addresses:

; Generic addresses used for "most" of the multiply
spaddr1:     DW (image1+0)%:
    DW (image1+16)%:
    DW (image1+32)%:
    DW (image1+48)%:
; Specific addresses used for the final row of multiplies in each
; half of the image (because the wavelet coefs wrap)
spaddr2:     DW (image1+0)%:
    DW (image1+16)%:
    DW (image1+224)%:
    DW (image1+240)%:
```

```
; Stage 2 sparse addresses:

; Generic addresses used for "most" of the multiply
spaddr3:     DW (image2+0)%:
    DW (image2+1)%:
    DW (image2+2)%:
    DW (image2+3)%:

; Specific addresses used for the final row of multiplies in each
; half of the image (because the wavelet coefs wrap)
spaddr4:     DW (image2+0)%:
    DW (image2+113)%:
    DW (image2+127)%:
    DW (image2+128)%:

; Image to be transformed
; Overwritten in second stage of the transform with final results
; Replace these pixels with those from the Matlab wrapper program output.
image1:      <Lots of DW entries to create blank space for incoming image>
             <Incoming image comes from Matlab wrapper program>
             <…>
             <…>

; Some empty space to make an obvious gap when browsing memory
; between original and transformed blocks
empty:       DW 0%:
    DW 0%:
    DW 0%:
    DW 0%:
    DW 0%:
    DW 0%:
    DW 0%:

; Empty space to be filled with first-stage transform results
image2:             DW 0%:
```

Appendices

A-3 Processor VHDL Design Files

Files included:

1. cpu.vhd
2. memory.vhd
3. reg_file_scalar.vhd
4. reg_file_vector.vhd
5. control_execute.vhd
6. alu.vhd
7. alu_mlacs.vhd
8. pc.vhd
9. ir.vhd
10. loop_counter.vhd
11. fetch.vhd
12. simulated_input.vhd