

2006

A study of the design and real-time implementation of a semi-generic integer-to-integer discrete wavelet transform

Michael P. Flaherty
University of Dayton

Follow this and additional works at: https://ecommons.udayton.edu/graduate_theses

Recommended Citation

Flaherty, Michael P., "A study of the design and real-time implementation of a semi-generic integer-to-integer discrete wavelet transform" (2006). *Graduate Theses and Dissertations*. 2678.
https://ecommons.udayton.edu/graduate_theses/2678

This Thesis is brought to you for free and open access by the Theses and Dissertations at eCommons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of eCommons. For more information, please contact mschlangen1@udayton.edu, ecommons@udayton.edu.

A STUDY OF THE DESIGN AND REAL-TIME
IMPLEMENTATION OF A SEMI-GENERIC
INTEGER-TO-INTEGER DISCRETE
WAVELET TRANSFORM

A Thesis

Submitted to

the Engineering School of the

UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for

The Degree

Master of Science in Electrical Engineering

by

Michael P. Flaherty, B.E.E

UNIVERSITY OF DAYTON

Dayton, Ohio

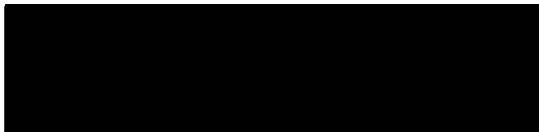
May 2006

A STUDY OF THE DESIGN AND REAL-TIME
IMPLEMENTATION OF A SEMI-GENERIC
INTEGER-TO-INTEGER DISCRETE WAVELET TRANSFORM

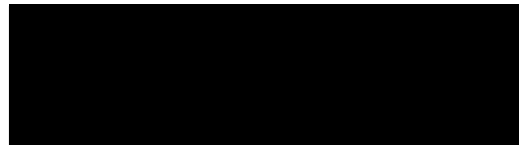
APPROVED BY:




Dr. Frank A. Scarpino, Ph.D.
Adviser Committee Chairman
Electrical & Computer Engineering




Dr. Russell Hardie, Ph.D.
Committee Member
Electrical & Computer Engineering



Dr. John Weber, Ph.D.
Committee Member
Electrical & Computer Engineering



Dr. Donald L. Moon, Ph.D.
Associate Dean, School of Engineering



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

ABSTRACT

A STUDY OF THE DESIGN AND REAL-TIME IMPLEMENTATION OF A SEMI-GENERIC INTEGER-TO-INTEGER DISCRETE WAVELET TRANSFORM

Name: Michael P. Flaherty
University of Dayton, 2006

Advisor: Dr. Frank A. Scarpino

JPEG2000 became an international standard in December of 2000 as the newest and most state of the art still image compression standard available according to the Joint Photographic Experts Group (JPEG). This compression standard is based on the discrete wavelet transform(DWT), instead of the more commonly used discrete cosine transform (DCT) which is part of the original JPEG standard. The result of which has been ever increasing amounts of research in the field of wavelet transforms. This research paper presents a method for the real-time implementation of several types of wavelet transforms including the CDF(9,7) and the LeGall(5,3) transforms which make up the core kernels of the JPEG2000 standard[21].

In this thesis, a forward discrete wavelet transform is designed and implemented on the Vertex 2 Pro package 7 FPGA. It is intended for use in a hardware accelerated JPEG2000 implementation. This implementation would have many potential applications including use in digital cameras, motion JPEG2000 applications, or high altitude surveillance schemes.

This thesis details a design that effectively uses the designs internal memory to allow for a single read - single write design. The single read - single write capability of this provides much more flexibility. By having to input the data only once, a potential implementor has only to worry about feeding the data to a "black box" in the right order and retrieving the data in the correct order. It also make the design very flexible because no specific memories or memory interfaces have to be used. These features allows a user to incorporate this design without the need to understand the implementation details.

To Myself..
...for all of your hard work.

ACKNOWLEDGMENTS

I would like to thank anyone and everyone who had to listen to me talk about this for months on end. However, I would especially like to thank the following people.

- **Dr. Frank Scarpino, PhD:**

I would like to thank you for all of your guidance over the course of my time in grad school. You have not only given sage advice on academic matters, but provided me with valuable life lessons.

- **Dr. Russell Hardie, PhD and Dr. John Weber, PhD:**

Thank you to Dr. Hardie and Dr. Weber for your advice and input on the writing of my thesis and for being on my thesis committee. I know you have busy schedules, so thank you for allowing me to take up some of your valuable time.

- **Dr. Eric Balster, PhD:**

I would like to thank you for providing me as much insight, if not more, than my actual advisor, and easily answering as many questions. Thank you for all of your time and patience.

- **Dave Mundy & Thaddeus Marrara:**

I would like to thank you both for listening to me complain and ask an endless number questions I knew you didn't know the answer to either, when I wasn't understanding. And for letting me bounce ideas off of you constantly.

- **Kerry Hill, Al Scarpelli, and Rob Ewing:**

I would like to thank you all, as well as the rest of the researchers and managers at Wright-Patterson Air Force Base, whose financial support and use of their resources made this thesis and my continuing education possible.

- **B. Jane Flaherty:**

Thank you to my mother for all of you support throughout my life. You have given me the opportunities that allowed me to go to grad school and do research like this. Thank you.

- **Kristin Schleppi:**

Thank you for your infinite patience and support throughout my time in graduate school. You supported me every way possible. Without you pushing me to do better, I probably never would have finished.

TABLE OF CONTENTS

	Page
Approval	ii
Abstract	iii
Dedication	v
Acknowledgments	vi
List of Tables	x
List of Figures	xi
 Chapters:	
1. Introduction	1
1.1 Image Compression	1
1.1.1 JPEG Image Compression Standard	4
1.1.2 JPEG2000 Image Compression Standard	4
1.2 Real-Time Implementation	6
1.2.1 Motion JPEG2000	6
1.2.2 Cost of Hardware Implementations	7
1.2.3 Importance	8
1.3 Innovative Contribution	9
1.4 Thesis Organization	10
2. Wavelet Overview	11
2.1 The Wavelet Transform	12
2.1.1 The Discrete Wavelet Image Transform	14
2.2 Benefits of the Wavelet Transform	16

3.	Lifting Overview	19
3.1	Origins of Lifting	19
3.2	Benefits of Lifting	19
3.2.1	Integer-to-Integer Lifting	22
3.3	Deriving Lifting Equivalent Filters	22
4.	Lifted Wavelet Algorithm	30
4.1	Overview	30
4.2	Implementation	32
4.2.1	Column Processor	33
4.2.2	Row Processor	36
4.2.3	Symmetric Extensions	39
4.3	Creating Generality	40
5.	Hardware Implementation	41
5.1	Synthesis Constraints	41
5.2	Implementing a Lifting Step	42
5.2.1	Implementing the Column Lifting Step	43
5.2.2	Implementing the Row Lifting Step	44
5.2.3	Semi Generality	44
5.3	Implementing the Row Processor Memory	45
5.3.1	Memory Solution	45
6.	Conclusions & Future Work	47
Appendices:		
A.	50
Bibliography		69

LIST OF TABLES

Table	Page
2.1 Dabchicks Orthogonal Wavelets.	16
2.2 LeGall(5,3) & CDF(9,7) Bi-orthogonal Wavelets.	16
4.1 Memory Conservation Examples	37
5.1 Synthesis Constraints	42

LIST OF FIGURES

Figure	Page
1.1 A General Image Compression Process	2
2.1 One MR Level Haar Transform	14
2.2 Three MR Level Haar Transform	15
3.1 Wavelet Filter Bank Approach	20
3.2 Wavelet Polyphase Representation	20
3.3 Lifting Filter Structure	26
4.1 Wavelet Column and Row Processors	32
4.2 Column Processor	33
4.3 Row Processor	37
5.1 Column Lifting Step	43
5.2 Row Lifting Step	44

CHAPTER 1

Introduction

JPEG2000 became an international standard in December 2000 as one of the newest and most state of the art still image compression standards available, according to the Joint Photographic Experts Group (JPEG). This compression standard is based on the discrete wavelet transform(DWT), instead of the more commonly used discrete cosine transform (DCT) which is part of the more popular original JPEG standard. The result of this has been ever increasing amounts of research in the field of wavelet transforms.

This research paper presents a design and implementation for the real-time implementation of several types of wavelet transforms including the CDF(9,7) and the LeGall(5,3) transforms which make up the core kernels of the JPEG2000 standard[21]. First, however, it is important look at why we need to compress images, and what are the current defacto standards for doing so. Consideration is also given to the importance of real-time processing. Finally, what are the costs involved in implementing the transforms with real-time processing constraints in mind.

1.1 Image Compression

Image compression is vital to filling the ever-increasing demand to store, transmit, and process images. Image compression allows us to store more pictures in less space,

transmit pictures to other locations faster and with better quality, and process or refine images in different ways, as they are compressed them.

Image compression does all of these things by reducing the amount of data that is necessary to reconstruct the images into what the human eye perceives as a visual image. Compression is accomplished by using a set of steps that are used in almost all image compression standard. These basic steps of typical image compression and decompression schemes are expressed in Figure 1.1.

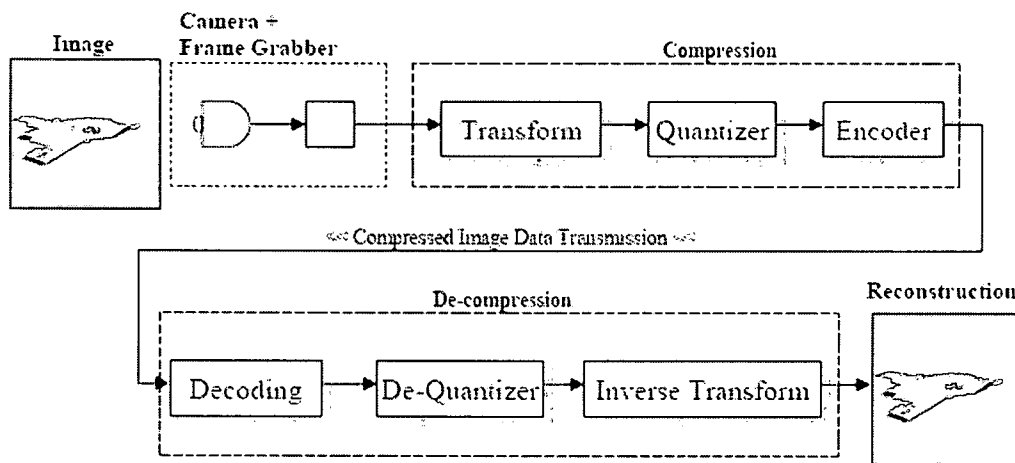


Figure 1.1: A General Image Compression Process

In Figure 1.1 it is seen that the first step is the image transform. The image transform is a mathematical function that allows the spatial reorganization of the data into a form that is, more compressible than the original data. The process of compression can be lossy¹ but, the loss is typically set so that the quality degradation is visually negligible. It is this step which the discrete wavelet transform (DWT), which is the topic of this thesis, falls under. However, the most common present image

¹The word lossy is used here to mean a process by which data can be reconstructed reasonably close to the original but is not perfect in all cases.

transform is the discrete cosine transform (DCT), which is used in such standards as JPEG, MPEG-1&2, and MPEG-4 [12]. The next step that comes in Figure 1.1 is the quantization step. It is in this step where a compression scheme becomes either lossless, in which a picture can be perfectly reconstructed, or lossy, in which the image can be approximately reconstructed. In the quantization step the designer/developer decides how much data can be disregarded before the distortion in the image becomes so great that the image quality is no longer tolerable. The final step in the compression process is the encoding of the data. In this step the data is truly "shrunk". It is in the encoding process that the data is translated into another representation using one of a number of different encoding schemes that allows the new data representation to take up less storage space by exploiting not only the spatial redundancies that were created in the image transform step, but many other types of redundancies that exist in images as well. One method of doing exploiting these redundancies is using a table of symbols that takes the statistical frequency of a data value into account. This is called Huffing Coding and it is a common encoding scheme that is used in the JPEG, M-JPEG, MPEG standards, as well as other image compression standards [24, 12]. Once the image is compressed it can then be stored, transmitted or sometimes manipulated, after which the image can be reconstructed by performing the inverse operations of the compression process in reverse as can be seen in Figure 1.1. The following subsections give a overview of compression standards for completeness.

1.1.1 JPEG Image Compression Standard

The JPEG file format was developed by the Joint Photographic Experts Group, and was the first and most common compressed, true-color image file format ². The JPEG format is used to display images on web pages, to store satellite imagery in military applications, and is embedded in most digital camera microprocessors. The JPEG compression scheme follows the generic compression model represented in Figure 1.1. In the JPEG standard the image transform used is the block-based discrete cosine transform (DCT) which sorts an image into 8x8 pixel blocks of data, and then transforms those blocks into one DC coefficient which is placed in the upper-left most position of the block and 63 higher frequency coefficients, which make up the rest of the positions in the block. The quantization is then done by dividing the data block by one of a set of a few 8x8 quantization matrices. The data is then passed to the encoding step using a "zigzag" pattern which helps to facilitate the encoder used. Finally, the data is encoded using either entropy or arithmetic encoding or sometimes both. Just as in the generic process described in the previous section, to reconstruct the image the reverse processes are run in reverse order. JPEG is arguably the most widely accepted image compression format available today. However, in 2000 the JPEG group that developed JPEG, created a new standard, JPEG2000, which it intends to supersede its original standard.

1.1.2 JPEG2000 Image Compression Standard

The image transform that is used in JPEG2000 is the DWT. The basics of how a DWT work are further developed in Chapter 2, but generally the DWT performs

²A true color image is usually considered one in which there are at least 24-bits per pixels in its uncompressed format.

on the entire image rather than breaking the picture into smaller 8x8 blocks as the DCT does. This does make processing the DWT more complex but in general the DWT has been observed to yield a restored image that has better quantitative quality as well as being more pleasing to the eye. JPEG2000 and wavelet transforms in general has been shown to yield approximately 25% more compression than that of the original JPEG standard[1]. Moving to the next step in the compression model, quantization, the JPEG2000 standard quantization step is optional because of the way that the encoding step process encodes the data. If quantization is used, it is quantized uniformly over each wavelet sub-band by simply dividing that sub-band by some specified amount. Also, the encoder used in JPEG2000 has the option of performing quantization as part of its processing method. For this reason combined with the fact that the JPEG2000 standard also offers a lossless mode in which no quantization is done, that the quantization step in JPEG2000 becomes optional. The final step in the compression model is the encoder which as stated is called EBCOT (Embedded Block Coding with Optimal Truncation). This is an arithmetic encoder that is extremely complex and process intensive, but extremely effective. The advantages that are brought to JPEG2000 by this step are many, as are the improvements the entire JPEG2000 standard brings over the original JPEG standard. These improvements include a progressive data stream, ROI (Region of Interest) capabilities, resolution scalability, SNR (Signal-to-Noise Ratio) scalability, and much more. For further reading any part of the JPEG2000 standard see [1, 9, 21].

1.2 Real-Time Implementation

The term “real-time” has many definitions but its basic meaning is typically the same. “Real-time” refers to the ability to process incoming data fast enough that one set³ of data is finished being processed before the next one is ready given some timing constraints. Real-time implementations for video are considered to be able to process approximately 30 frames (or pictures) a second using a popular video display standard called NTSC [12]. In order to meet this standard requirements, the process must be able to output one picture every $0.0\bar{3}$ seconds in order to be able to keep up in real time. This is can sometimes be faster than a average PC can handle using software implementations, especially with the growing complexities of the compression algorithms (i.e. JPEG2000) and other image processing algorithms. With most of these highly complex algorithms, real-time processes need to either be implemented completely in hardware or in some cases merely hardware-accelerated⁴. Using Field Programable Gate Arrays (FPGA's) in conjunction with a newer type of programming called HDL (Hardware Description Language) algorithms can easily and much more rapidly be developed and implemented on hardware while at the same time at a much lower cost in terms of time, money and risk than in the past.

1.2.1 Motion JPEG2000

In order to create the appearance of motion images are captures one after the other in sequence to create the appearance of motion. This is the way that all video

³A set might be anything as small as a 16-bit audio sample to as large as 300 KB image or larger.

⁴Hardware-accelerated refers to a process that has some of the lower level processes performed in hardware allowing for speed gains, while still retaining the flexibility that software provides to other complex processes of an algorithm.

is created, however in many of the video compression schemes there is a temporal component, or analysis of the video across multiple pictures over time. This is not the case with M-JPEG2000. Motion JPEG2000 (or M-JPEG2000) is simply a standard that uses the JPEG2000 Image compression standard to compress the images one after another. In M-JPEG2000 pictures are compressed as independent entities exactly as would be done if only one picture were being compressed. However, in M-JPEG2000 there is the need to be able to compress each picture in a much shorter amount of time for real-time applications. As stated in Section 1.2, to create a real-time video stream of JPEG2000 compressed images, each image must be compressed in 0.03 seconds. It is in this situation that a hardware or a hardware-accelerated implementation of JPEG2000 is useful. It is with the hardware-accelerated JPEG2000 implementation that the research and development of a hardware implemented DWT is conducted for this thesis.

1.2.2 Cost of Hardware Implementations

The costs of hardware implementations are much less than they have been in the past, but they still need to be considered. Hardware implementations typically take more time to develop and implement than most software implementation. This is in part because HDL languages are so new and developers are still learning to use them. It also takes more time to program a chip and run a test algorithm than it does to simply compile some software, so the debugging process takes longer than in software. Lastly, there is not always a one-to-one equivalence between the resources available in different hardware architectures, so the same HDL script may synthesize correctly and within the resource constraints of one chip and not on another. Additional hardware

also means additional power in most cases. This is a consideration that must be taken into account if a hardware implementation is to be used. Hardware implementations also need access to memory. Memory must be supplied from the overall system resources or additional memory must be given to the algorithm separately. There may also need to be some shared memory, and for a hardware-accelerated design, it can lead to a much more complex overall hardware design. Finally, there is monetary cost. Chips cost money, the extra time it takes to develop a hardware implementation costs money, and additional power needs cost money as well. There are many costs involved in the development of a hardware implementation and these costs must be weighed against the potential benefits in order to determine if a hardware implementation is right design decision.

1.2.3 Importance

Because the DWT is an ineffective means of compression by itself other processes must be used in conjunction with the DWT. This means, for example, that if a wavelet compression scheme, like JPEG2000, is to be used for creating video, not only does the DWT have to happen in our 0.03 seconds, but the quantization and encoding steps do as well. Also, as stated above, the EBCOT process is extremely complex [20]. This makes it very important to speed up each individual process as much as possible in order to be able to perform the compression in real-time. Real-Time implementations then become important to the DWT and JPEG2000 in order to reduce the amount of time that the DWT takes so that the entire process can remain quick. While the EBCOT is complex and therefore difficult to implement in hardware, the DWT is less complex by comparison. The DWT only requires two processing passes over the

image, where as the EBCOT could require as many as 69 passes over the image to complete. As a result, the DWT would make a much more likely candidate to have its operations performed in hardware in order to achieve an overall hardware-accelerated design. By having a having the DWT performed in hardware not only would the DWT be performed faster that it would be in a software implementation, but by having dedicated hardware for the DWT it could be performed concurrently with the other processes. This could make a hardware implementation of the DWT invaluable in the right context.

1.3 Innovative Contribution

While the idea of hardware implementation of a discrete wavelet transform has been explored, there are several points in this paper where ideas and work are new. The first of which is the idea of building a generic wavelet in hardware. While this design has only some generality to it, meaning it allows more than one type of wavelet but not all wavelets, at the date of this writing, a literature search yields no documentation dealing with the subject. Second, the idea for the memory management of the wavelet data, that allows this design to be single read - single write was presented in a paper by S. Barua and others in 2005, the design focused on row-wise operations as the initial pass on the data[4]. While this design is much more memory efficient than an external memory implementation discussed in Chapter 4, the design is more memory efficient if the data is processed column-wise initially, if the statistical prevalence of images to be wider than they are tall is taken into account. In the implementation of the design presented in this paper, the data is processed column-wise initially. Finally, the design presented in the Barua paper was never developed into

an implementation and verified as a usable method. In the implementation of this design, that verification is achieved.

1.4 Thesis Organization

This thesis is organized into 6 chapters. In Chapters 1 through 3, the relevant topics that are needed for the design and implementation of the discrete wavelet transform are introduced. Chapter 4 introduces the specific method of implementation of the DWT algorithm. In it an overall abstracted view of how the design will work is taken. Also, the design choices that are made, and the reasoning behind them, are discussed. In Chapter 5 the method of the implementation of the algorithm is discussed, including all of the hardware resources that are used and design decisions are discussed and justified. Also discussed are the difficulties faced and how the implementation changed as a result. Finally, in Chapter 6 the conclusions are stated as well as possibilities for future additions to this design and further research that is possible in this area.

CHAPTER 2

Wavelet Overview

In this chapter, the wavelet transform is discussed, as is the general process by which a wavelet transform is performed. In addition, an integration example is given in JPEG2000. It is also important to note that while video and the wavelet transform are being discussed in this paper, there is a form of wavelet transform that exists called the 3-D or temporal transform. However, since this is not the subject of this research it will not be discussed.

In image compression there are two types of redundancy that are typically exploited: spectral and spatial[16]. Spectral redundancy is usually utilized by the performance of a color transform in a compression step called preprocessing which is done before the image transform. For more information on color transforms see [1, 9, 21]. The DWT (Discrete Wavelet Transform), on the other hand, fits under the heading of image transform and its purpose is to be a reversible process that spatially reconfigures the data in order to decrease its spatial entropy (or increase its spatial redundancy) which makes the data much more compressible than its original form. This is why image transforms and the wavelet transform in particular are so important to the image compression process.

2.1 The Wavelet Transform

The first thing to note about the DWT is that there is more than one set of DWT filters, unlike the DCT which has only one type of forward transform and therefore only one image transform. On the other hand there are nearly an infinite number of different wavelet transforms. This is the reason that a generic wavelet, or in the case of this design a “semi-generic” wavelet, is needed. The reason there are so many different wavelets is because the effective process behind wavelet transforms is relatively generic. The basic concept of the forward DWT is to low-pass $\tilde{h}(z)$ and high-pass $\tilde{g}(z)$ filter image data in either the vertical or horizontal direction and then in the direction not initially chosen. This is done in order to “push” all of the energy in an image into a smaller area. Once this is done the data is down-sampled by a factor of 2. This can be repeated on the quadrant of the image that was low-pass filtered twice until a satisfactory entropy is achieved. This process can be seen in Figures 2.1 and 2.2. The down-sampling is done so the number of filtered data points is the same as that of the original data. Even though the data has been down-sampled the data can be reconstructed perfectly with the inverse DWT [21]. This is done with a specific set of filters called wavelet filters. In these sets of wavelet filters the low-pass filter is called the scaling function and the high-pass filter is called the wavelet function. An example of this process will follow shortly.

The discrete high-pass filter can be determined from the discrete low-pass filter using,

$$g(z) = z^{-n}h(-z^{-1}). \quad (2.1)$$

Previously the wavelet process was discussed in terms of entropy. Another way to describe the wavelet process is in terms of energy. The DWT compacts and preserves energy[23]. An example of this is the simplest and first wavelet transform developed, the Haar Transform[22]. The discrete transform equation for the Haar low-pass filter, also called the scaling function is given by,

$$\tilde{h}(z) = \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}z^{-1} \quad (2.2)$$

and if we apply Equation 2.1 to 2.2 the Haar high-pass filter, also called the wavelet function is yielded as,

$$\tilde{g}(z) = \frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}z^{-1}. \quad (2.3)$$

For example, if a signal is given as $f = \{4, 6, 10, 12, 8, 6, 5, 5\}$ and the data set f is run through the Haar filters the resulting data is

$$\begin{aligned} f_h &= \{5\sqrt{2}, 8\sqrt{2}, 11\sqrt{2}, 10\sqrt{2}, 7\sqrt{2}, 5.5\sqrt{2}, 5\sqrt{2}, ?\} \\ f_g &= \{-\sqrt{2}, -2\sqrt{2}, -1\sqrt{2}, 2\sqrt{2}, 1\sqrt{2}, 0.5\sqrt{2}, 5\sqrt{2}, ?\}.^5 \end{aligned}$$

Next, the signals are down-sampled by 2 and combined, low-pass on the left and high-pass on the right. This yields,

$$f_{gh} = \{5\sqrt{2}, 11\sqrt{2}, 7\sqrt{2}, 5\sqrt{2} | -\sqrt{2}, -\sqrt{2}, \sqrt{2}, 0\}$$

which is the discrete wavelet transformed data. Now, if the energy of the data is compared, defining energy as,

$$E = \sum_{i=1}^n f(n)^2 \quad (2.4)$$

and Equation 2.4 is applied to the original data set, the energy of that set is $E = 446$.

If then, Equation 2.4 is applied to the new data set f_{gh} the total energy there also

⁵The last values in these sets are undetermined because the type of extension has not been discussed or determined.

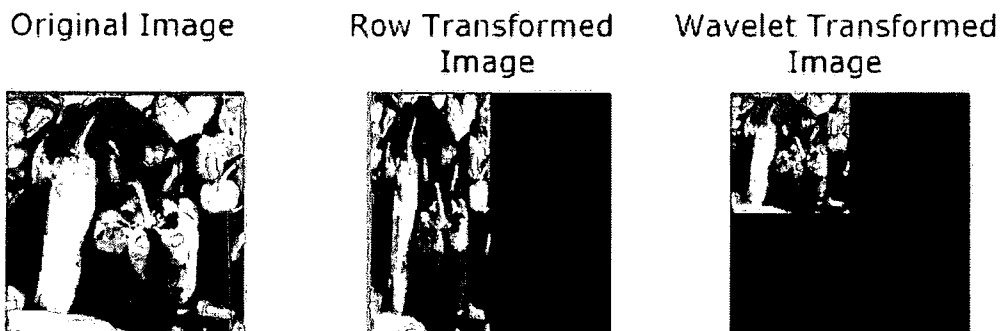


Figure 2.1: One MR Level Haar Transform

yields $E = 446$. This shows energy preservation. If we then look at the energy on either side of data set f_{gh} it is seen that $E = 440$ on the left half and $E = 6$ on the right. This shows energy compaction[23]. Another illustration of this can be seen in Figure 2.1 where the image at the far right appears to be squeezed onto the left half of the center image. While there is still some data on the right half it is at very low energy and therefore is barely detectable by the eye. However, the majority of the energy has been moved to the left half and so that half can be seen plainly.

2.1.1 The Discrete Wavelet Image Transform

In order to extend the previous example into two dimensions so that an image can be transformed, the process is repeated as in the previous example, first in one of the two image dimension, either across the rows or down the columns. Then the process is repeated in the direction that was not covered by the first pass. The transform would then look something like Figure 2.1. This process creates a one multi-resolution (MR) level transform. In order to compact the energy of the image even further, and by doing so make the image even more compressible, the same two dimensional process is again

repeated on the bulk of the energy which is located in the upper left of transformed image as seen in Figure 2.1. This successive transforming can be seen in Figure 2.2.

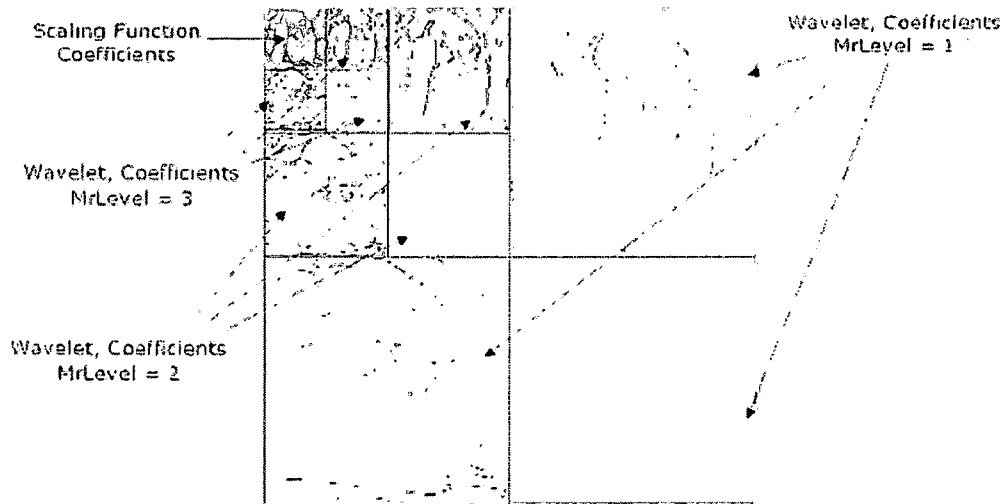


Figure 2.2: Three MR Level Haar Transform

Wavelet Types

While in the examples in this chapter have only used the Haar Wavelet, it is important to note that, as stated previously, there are nearly an infinite number of possible wavelet transforms. One special distinction between that presented, and the CDF(9,7) and the LeGall(5,3), which are the wavelets of focus for this research, is that the Haar is a orthogonal wavelet. This means that it has the energy preservation property discussed above, as well as having all of the same coefficients for both the scaling function and the wavelet function, although the coefficients are in different order and have differing signs. In Table 2.2 a set of wavelets developed by Ingrid Daubechies illustrates these coefficients.

Wavelet Name	Coefficients
Haar (D2)	Scaling: 0.707 0.707 Wavelet: -0.707 0.707
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

Table 2.1: Dabchicks Orthogonal Wavelets.

Bi-orthogonal wavelets have the same traits and follow the same rules as orthogonal wavelets, except for the two distinctions made previously for the orthogonal wavelets. Bi-orthogonal wavelets do not always preserve energy as is the case of the LeGall(5,3), however, in some instances they do, as is the case for the CDF(9,7) [8]. Also, the coefficients are not the same in bi-orthogonal filters. In fact the filters are not the same length in most cases. This is illustrated in Table 2.2 where the coefficients for the LeGall(5,3) and the CDF(9,7) are listed.

Wavelet Name	Coefficients
LeGall(5,3)[21]	Scaling: -0.125 0.25 0.75 0.25 -0.125 Wavelet: -0.5 1 -0.5
CDF(9,7)[7]	Scaling: 0.037 0.023 -0.110 -0.377 0.852 -0.377 -0.110 0.023 0.037 Wavelet: -0.064 -0.040 0.418 0.788 0.418 -0.040 -0.064

Table 2.2: LeGall(5,3) & CDF(9,7) Bi-orthogonal Wavelets.

2.2 Benefits of the Wavelet Transform

The DWT has many benefits over the DCT, the first of which is quality of the images it can produce[15]. There are two aspects to this topic. The first is the sheer

variety of wavelets available. With so many wavelets to choose from, a wavelet can almost always be found that allows for better image compression than the DCT. This must also mean that if better compression can be achieved at the same quality, then it must be possible to get a better quality image at the same compression ratio. The second part of this argument is the method transform. Because the wavelet transform encompasses the entire image rather than a 8x8 block at a time there are no blocking artifacts. This, arguably, makes the image seem better at higher compression rates than the DCT because the eye is better equipped to deal with a blurry images rather than blocky ones [3].

The wavelet transform also lends itself to making an image resolution scalable. In fact, this trait, combined with a progressive encoding, allows JPGE2000 to have resolution scalability as well a progressive resolution capability, which allows the image to be reconstructed with only the first fraction of the compressed data and then increase the resolution as more data is access, as part of its feature set.

Finally, because there are many different wavelets available an emerging field is that of multivalent compression. In this technique a combination of wavelets are used, a different one for each multiresolution level. This research is showing promise in making wavelet combinations that are allowing for even more compression than that of the single wavelet compression schemes that are already out performing the DCT based compression schemes[14].

Drawbacks

While there are so many benefits of the DWT and wavelet analysis, the drawbacks must be mentioned as well. The first of which is the number of wavelets available. Because there are so many wavelets and no closed form solution for finding which one

is the best for a particular set of data, the only option left is experimental iteration. It could take some time to find a wavelet that will give a substantial improvement over more established compression schemes in certain cases. It also makes finding combinations that work well in the emerging multivalent field all that more difficult. A second minor drawback of the wavelet transform is the fact that the entire image is processed together. While this makes the image arguably easier to look at it, it also makes the computational complexity and memory requirements more demanding. This drawback is discussed further in later chapters.

CHAPTER 3

Lifting Overview

3.1 Origins of Lifting

The term “Lifting” is used to describe the factorization of a discrete wavelet filter, was coined by Wim Sweldens who discovered and introduced the subject in the 1990’s[10, 17, 19]. Sweldens uses a factorized filter structure to implement the discrete wavelet transform filters in order to obtain a more compact representation of the data. This structure has been refined and is now what is known as lifting[6]. Lifting has many benefits for effectively implementing wavelet filters. Some of these benefits are briefly discussed in the following section.

3.2 Benefits of Lifting

A method of performing a discrete wavelet transform is introduced in Chapter 2 and one method of achieving the transform is discussed. This method is called the filter bank approach, and a graphical representation can be seen in Figure 3.1. It is named as such because the image data is passed through a series of FIR filters called a filter bank. However, the filter bank approach is a generic method for implementing any type of filter and does not taking into account redundancies in wavelet filters. Therefore the filter bank approach is not the most efficient method for the implementation of the discrete wavelet filters. In using the filter bank method, all

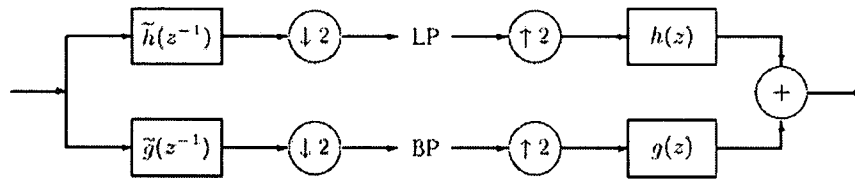


Figure 3.1: Wavelet Filter Bank Approach

of the data is run independently through each of the two forward transform filters, seen in Figure 3.1, $\tilde{h}(z^{-1})$ and $\tilde{g}(z^{-1})$. Once that is done, the data is then twice the size of the original, so it then has to be down-sampled decimating by 2 and throwing half of the output data, and half of the processing time is wasted. The wasted time and data are the most significant reasons the filter bank approach to implementing wavelet filters is so inefficient.

Lifting takes into account the redundancies present in wavelet filters and eliminates some of the inefficiencies discussed with the filter bank approach. First it allows the down-sampling to take place before the filter processing. It also allows parallel processing of the data. Utilizing these improvements over the filter bank approach, a lifting implementation reduces the number of operations needed to perform a forward DWT to nearly half[11, 18]. This helps to significantly speed up the DWT. The lifting implementation is represented in Figure 3.2 where P represents the polyphase matrix, which is the standard mathematical representation of a lifting filter[6, 10].

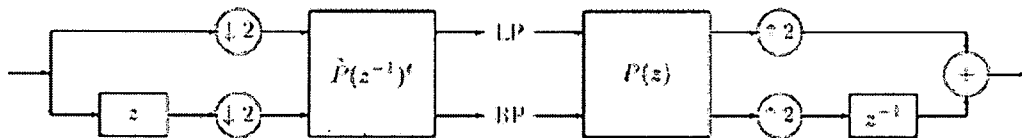


Figure 3.2: Wavelet Polyphase Representation

Another advantage that lifting has over the filter bank implementation is processing order. Utilizing the filter bank approach, the data must be run through the low-pass filtration process and then the high-pass process, or the two processes can be run simultaneously at the cost of additional resources. However, if the lifting approach is used then the data is run through the filtering process only once and the lifting filter outputs both the low and high-pass data at the same time. This result is again illustrated in Figure 3.2. Lifting has the added benefit of allowing in-place processing[17]. The benefit of in-place processing is that because the input data needs to be read once there is no need for the input data to be retained as with the filter bank approach. In-place processing allows for the input data to be immediately overwritten by the output data and essentially halves the memory requirement for performing a DWT.

A third benefit of lifting allows for pipelining of the filtering process. Pipelining is achieved as a result of the factorization that takes place in order to obtain the lifting filter from the originals. As a result of this process, lifting factors the wavelet FIR filters into a series of smaller filters that produces the same results as the originals. These smaller filters are called steps [6]. A series of smaller filters is beneficial because it allows for pipelining of the filtration process. Pipelining in this case is a process in which the filter can move onto another data point before the previous one has finished. The benefit is that it allows for a more efficient use of resources. This becomes especially important in either a custom ASIC or FPGA hardware implemented design.

3.2.1 Integer-to-Integer Lifting

One final benefit that can only be achieved through the lifting process is the creation of integer-to-integer DWT filters[5]. This is advantageous when attempting hardware implementations of DWT's. Integer outputs are advantageous because floating point numbers are complex elements to introduce to a hardware system, as are the operations needed to do floating point operations. On the other hand, if an integer-to-integer implementation is used, the hardware complexity and resources needed to implement these same operations are reduced[2, 5].

With the integer-to-integer approach, however, there are several drawbacks that result. These drawback occur because of the method used to achieve the output integers. The integerization of the lifting filters will be discussed in more detail in later sections, however essentially a non-linearity is introduced to the transform[5]. The introduction of the non-linearity results in the loss of separability⁶. Therefore the order of the one dimensional transforms must be predetermined and run in opposite orders in the forward and inverse DWT's. An additional drawback is that any compression scheme that uses the integer-to-integer approach must use the same approach during the decompression process or perfect reconstruction will not be achieved.

3.3 Deriving Lifting Equivalent Filters

The lifting filters are determined by factoring the wavelet filter coefficients. This factorization is done by putting the wavelet filters into a polyphase matrix, and factoring that matrix into alternating upper and lower triangular matrices each of

⁶Separability in the of case the DWT refers to the transforms ability to have the one dimensional transforms performed in either order in both the forward and inverse transforms and still achieve perfect reconstruction.

which represents a new smaller filter. Once the factorization is done the wavelet can then be implemented using the smaller filters that each upper and lower triangular matrices represent.

In order to begin the lifting derivation, one must obtain the polyphase matrix representation of the wavelet filter pair,

$$P(z) = \begin{bmatrix} h_e(z) & g_e(z) \\ h_o(z) & g_o(z) \end{bmatrix} \quad (3.1)$$

where $P(z)$ is the polyphase matrix, $h_e(z)$ & $h_o(z)$ are the even and odd coefficients of the inverse low-pass filter, respectively, and $g_e(z)$ & $g_o(z)$ are the even and odd coefficients of the inverse high-pass filter, respectively. Figure 3.2 it shows that $P(z)$ is associated with the synthesis wavelet filter. The analysis or decomposition matrix, $\tilde{P}(z^{-1})^t$, can then be obtained directly from the synthesis matrix, and it is more convenient to find the synthesis filter. The even and odd coefficient equations for the scaling filter are obtained by,

$$h(z) = h_e(z^2) + z^{-1}h_o(z^2), \quad (3.2)$$

where in Figure 3.1 it can be seen that $h(z)$ is the synthesis scaling function. Also, Laurent polynomials $h_e(z)$ and $h_o(z)$ can be obtained more directly using,

$$h_e(z^2) = \frac{h(z) + h(-z)}{2}, \quad (3.3)$$

$$h_o(z^2) = \frac{h(z) - h(-z)}{2z^{-1}}. \quad (3.4)$$

The synthesis wavelet function, $g(z)$ can be decomposed in the same manner.

The Euclidean algorithm is now introduced as a method of finding the greatest common divisor of the two Laurent polynomials $h_e(z)$ and $h_o(z)$. While the Euclidean

algorithm⁷ was originally developed to find the greatest common divisor of natural numbers, it can be extended to Laurent polynomials⁸ with only the caveat that the solution is not unique[6]. In order to use the Euclidean algorithm let $h_e(z) = a(z)$ and $h_o(z) = b(z)$ where $|a(z)| \geq |b(z)|$ and iterate through equations 3.5 and 3.6, where initially let $a_0(z) = a(z)$ and $b_0(z) = b(z)$, until $b_n(z) = 0$. As a result $a_n(z) = gcd(a(z), b(z))$.

$$a_{i+1}(z) = b_i(z) \quad (3.5)$$

$$b_{i+1} = a_i(z) \% b_i(z) \quad (3.6)$$

where % is the modulus operator. While not part of the Euclidean algorithm, it is important in deriving the lifting filter to keep track of the quotient $q(z)$. In order to do this another equation is added to the iterations in the Euclidean algorithm and it is,

$$q_{i+1} = a_i(z)/b_i(z). \quad (3.7)$$

It should also be noted that $q(z)$ can take many values, which is why the solution of the Euclidean Algorithm is not unique. This allows for a certain freedom when developing lifting filters which also allows for a variety of lifting filter equivalents for a given wavelet filter pair. It is up to the developer to decide which is best for a given design[6].

Once all of the quotients $q_n(z)$ are obtained a representation of the filter can be shown as,

$$\begin{bmatrix} h_e(z) \\ h_o(z) \end{bmatrix} = \prod_{i=1}^n \begin{bmatrix} q_i(z) & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} K \\ 0 \end{bmatrix} \quad (3.8)$$

⁷Euclidean algorithm is an algorithm to determine the greatest common divisor of two integers.

⁸Laurent Polynomial is a polynomial which has the form of the Laurent Series in which functions are represented as power series.

where $K = a_n(z)$. An intermediate representation of the polyphase matrix $P^0(z)$ can then be found by a small change in the constant matrix so that,

$$P^0(z) = \begin{bmatrix} h_e(z) & g_e^0(z) \\ h_o(z) & g_o^0(z) \end{bmatrix} = \prod_{i=1}^n \begin{bmatrix} q_i(z) & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix} \quad (3.9)$$

which can be shown to be equivalent to,

$$P^0(z) = \begin{bmatrix} h_e(z) & g_e^0(z) \\ h_o(z) & g_o^0(z) \end{bmatrix} = \prod_{i=1}^{n/2} \begin{bmatrix} 1 & q_{2i-1}(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ q_{2i}(z) & 0 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix}. \quad (3.10)$$

Finally, the complete factored representation of the polyphase matrix $P(z)$ can be found by,

$$P(z) = P^0(z) \begin{bmatrix} 1 & s(z) \\ 0 & 1 \end{bmatrix}. \quad (3.11)$$

However, it is difficult to find $s(z)$ using this representation. Instead, by noticing that both the intermediate and the original filter pairs are complementary,⁹ the filters are then reconstructed from the polyphase matrices using equation 3.2. $s(z)$ can then be found using,

$$g^0(z) = g(z) + h(z)s(z^2). \quad (3.12)$$

This still leaves a problem. Using 3.11 the polyphase representation,

$$P(z) = \prod_{i=1}^{n/2} \begin{bmatrix} 1 & q_{2i-1}(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ q_{2i}(z) & 0 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix} \begin{bmatrix} 1 & s(z) \\ 0 & 1 \end{bmatrix}. \quad (3.13)$$

where for the easiest implementation the constant gains will be moved to the last step. In order to move the last matrix inside of the constant matrix the Laurent polynomial inside $s(z)$ must be scaled by,

$$s_m(z) = K^2 s(z). \quad (3.14)$$

⁹Complementary in this case refers to any filter pair whose polyphase matrix has determinate 1.

This yields a final polyphase matrix factorization of,

$$P(z) = \begin{bmatrix} h_e(z) & g_e(z) \\ h_o(z) & g_o(z) \end{bmatrix} = \prod_{i=1}^m \begin{bmatrix} 1 & s(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ t(z) & 0 \end{bmatrix} \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix} \quad (3.15)$$

where $m = n/2 + 1$ because of the extra step generated in equation 3.11. Once this procedure is complete the filter construction then looks like that seen in Figure 3.3, which is a more detailed representation of the left half of Figure 3.2.

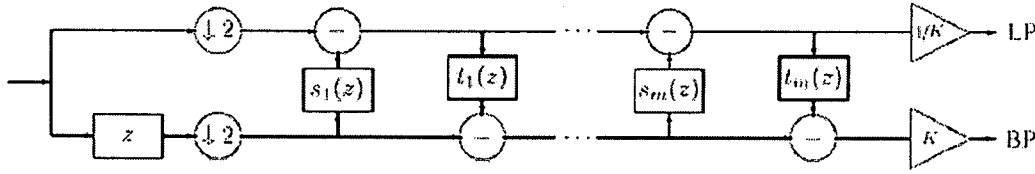


Figure 3.3: Lifting Filter Structure

An Example

To show a useful example, the factorization of the D4 (Daubechies 4) wavelet is demonstrated. The equations for the D4 synthesis filters are as follows,

$$h(z) = \frac{1 + \sqrt{3}}{4\sqrt{2}} + \frac{3 + \sqrt{3}}{4\sqrt{2}}z^{-1} + \frac{3 - \sqrt{3}}{4\sqrt{2}}z^{-2} + \frac{1 - \sqrt{3}}{4\sqrt{2}}z^{-3}, \quad (3.16)$$

$$g(z) = -\frac{1 - \sqrt{3}}{4\sqrt{2}}z^2 + \frac{3 - \sqrt{3}}{4\sqrt{2}}z - \frac{3 + \sqrt{3}}{4\sqrt{2}} + \frac{1 + \sqrt{3}}{4\sqrt{2}}z^{-1}. \quad (3.17)$$

Here is it useful to recognize that the D4 is a orthogonal set of filters based on stipulations set in Chapter 2. Using Equations 3.3 and 3.4 on equation 3.16 the left two elements of the polyphase matrix are defined by,

$$h_e(z^2) = \frac{\frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3+\sqrt{3}}{4\sqrt{2}}z^{-1} + \frac{3-\sqrt{3}}{4\sqrt{2}}z^{-2} + \frac{1-\sqrt{3}}{4\sqrt{2}}z^{-3} + \frac{1+\sqrt{3}}{4\sqrt{2}} - \frac{3+\sqrt{3}}{4\sqrt{2}}z^{-1} + \frac{3-\sqrt{3}}{4\sqrt{2}}z^{-2} - \frac{1-\sqrt{3}}{4\sqrt{2}}z^{-3}}{2}$$

$$h_e(z^2) = \frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3-\sqrt{3}}{4\sqrt{2}}z^{-2}$$

$$h_e(z) = \frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3-\sqrt{3}}{4\sqrt{2}}z^{-1}$$

$$h_o(z^2) = \frac{\frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3+\sqrt{3}}{4\sqrt{2}}z^{-1} + \frac{3-\sqrt{3}}{4\sqrt{2}}z^{-2} + \frac{1-\sqrt{3}}{4\sqrt{2}}z^{-3} - (\frac{1+\sqrt{3}}{4\sqrt{2}} - \frac{3+\sqrt{3}}{4\sqrt{2}}z^{-1} + \frac{3-\sqrt{3}}{4\sqrt{2}}z^{-2} - \frac{1-\sqrt{3}}{4\sqrt{2}}z^{-3})}{2z^{-1}}$$

$$h_o(z^2) = \frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1-\sqrt{3}}{4\sqrt{2}}z^{-2}$$

$$h_o(z) = \frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1-\sqrt{3}}{4\sqrt{2}}z^{-1}$$

Thus far the polyphase matrix looks like,

$$P(z) = \begin{bmatrix} \frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3-\sqrt{3}}{4\sqrt{2}}z^{-1} & g_e(z) \\ \frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1-\sqrt{3}}{4\sqrt{2}}z^{-1} & g_o(z) \end{bmatrix}.$$

The right two elements of the polyphase matrix can then be found by using Equations 3.3 and 3.4 on equation 3.17.

$$g_e(z^2) = \frac{-\frac{1-\sqrt{3}}{4\sqrt{2}}z^2 + \frac{3-\sqrt{3}}{4\sqrt{2}}z - \frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1+\sqrt{3}}{4\sqrt{2}}z^{-1}(-\frac{1-\sqrt{3}}{4\sqrt{2}}z^2 + \frac{3-\sqrt{3}}{4\sqrt{2}}z - \frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1+\sqrt{3}}{4\sqrt{2}}z^{-1})}{2}$$

$$g_e(z^2) = -\frac{1-\sqrt{3}}{4\sqrt{2}}z^2 - \frac{3+\sqrt{3}}{4\sqrt{2}}$$

$$g_e(z) = -\frac{1-\sqrt{3}}{4\sqrt{2}}z - \frac{3+\sqrt{3}}{4\sqrt{2}}$$

$$g_o(z^2) = \frac{\frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3+\sqrt{3}}{4\sqrt{2}}z^{-1} + \frac{3-\sqrt{3}}{4\sqrt{2}}z^{-2} + \frac{1-\sqrt{3}}{4\sqrt{2}}z^{-3} - (\frac{1+\sqrt{3}}{4\sqrt{2}} - \frac{3+\sqrt{3}}{4\sqrt{2}}z^{-1} + \frac{3-\sqrt{3}}{4\sqrt{2}}z^{-2} - \frac{1-\sqrt{3}}{4\sqrt{2}}z^{-3})}{2z^{-1}}$$

$$g_o(z^2) = \frac{3-\sqrt{3}}{4\sqrt{2}}z^2 + \frac{1+\sqrt{3}}{4\sqrt{2}}$$

$$g_o(z) = \frac{3-\sqrt{3}}{4\sqrt{2}}z + \frac{1+\sqrt{3}}{4\sqrt{2}}$$

The Polyphase matrix representation of the filter pair then is given by,

$$P(z) = \begin{bmatrix} \frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3-\sqrt{3}}{4\sqrt{2}}z^{-1} & -\frac{1-\sqrt{3}}{4\sqrt{2}}z - \frac{3+\sqrt{3}}{4\sqrt{2}} \\ \frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1-\sqrt{3}}{4\sqrt{2}}z^{-1} & \frac{3-\sqrt{3}}{4\sqrt{2}}z + \frac{1+\sqrt{3}}{4\sqrt{2}} \end{bmatrix}.$$

This equation is all that is needed to begin using the Euclidian Algorithm. By setting $a_0(z) = h_e(z)$ and $b_0(z) = h_o(z)$ and iterating through equations 3.5, 3.6 and 3.7 of the Euclidean algorithm, the iterations give,

$$a_1(z) = \frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1-\sqrt{3}}{4\sqrt{2}}z^{-1}$$

$$\begin{aligned}
b_1(z) &= \left(\frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3-\sqrt{3}}{4\sqrt{2}} z^{-1} \right) \% \left(\frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1-\sqrt{3}}{4\sqrt{2}} z^{-1} \right) = \frac{1+\sqrt{3}}{\sqrt{2}} \\
q_1(z) &= \frac{\frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3-\sqrt{3}}{4\sqrt{2}} z^{-1}}{\frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1-\sqrt{3}}{4\sqrt{2}} z^{-1}} = -\sqrt{3} \\
a_2(z) &= \frac{1+\sqrt{3}}{\sqrt{2}} \\
b_2(z) &= \left(\frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1-\sqrt{3}}{4\sqrt{2}} z^{-1} \right) \% \left(\frac{1+\sqrt{3}}{\sqrt{2}} \right) = 0 \\
q_2(z) &= \frac{\frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1-\sqrt{3}}{4\sqrt{2}} z^{-1}}{\frac{1+\sqrt{3}}{\sqrt{2}}} = \frac{\sqrt{3}}{4} + \frac{\sqrt{3}-2}{4} z^{-1}.
\end{aligned}$$

Once the Euclidean algorithm is finished the results can be used to construct the intermediate matrix factorization of the left two original Laurent polynomials and the right two intermediate Laurent polynomials, as in Equation 3.13. This intermediate polyphase matrix is found to be,

$$P^0(z) = \begin{bmatrix} 1 & -\sqrt{3} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \frac{\sqrt{3}}{4} + \frac{\sqrt{3}-2}{4} z^{-1} & 0 \end{bmatrix} \begin{bmatrix} \frac{1+\sqrt{3}}{\sqrt{2}} & 0 \\ 0 & \frac{\sqrt{2}}{1+\sqrt{3}} \end{bmatrix}.$$

The product of the matrices in 3.3 can be used in 3.11 to yield,

$$\begin{bmatrix} \frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3-\sqrt{3}}{4\sqrt{2}} z^{-1} & -\frac{1-\sqrt{3}}{4\sqrt{2}} z - \frac{3+\sqrt{3}}{4\sqrt{2}} \\ \frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1-\sqrt{3}}{4\sqrt{2}} z^{-1} & \frac{3-\sqrt{3}}{4\sqrt{2}} z + \frac{1+\sqrt{3}}{4\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3-\sqrt{3}}{4\sqrt{2}} z^{-1} & \frac{\sqrt{3}-3}{\sqrt{2}} \\ \frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1-\sqrt{3}}{4\sqrt{2}} z^{-1} & \frac{\sqrt{3}-1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 & s(z) \\ 0 & 1 \end{bmatrix}.$$

Now, by using Equation 3.2 to reconstruct the filter pairs in both the original and intermediate polyphase matrices the filter equations can then be used in Equation 3.12,

$$\begin{aligned}
&\frac{\sqrt{3}-3}{\sqrt{2}} + \frac{\sqrt{3}-1}{\sqrt{2}} z^{-1} = \\
&-\frac{1-\sqrt{3}}{4\sqrt{2}} z^2 + \frac{3-\sqrt{3}}{4\sqrt{2}} z - \frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{1+\sqrt{3}}{4\sqrt{2}} z^{-1} + \left(\frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3+\sqrt{3}}{4\sqrt{2}} z^{-1} + \frac{3-\sqrt{3}}{4\sqrt{2}} z^{-2} + \frac{1-\sqrt{3}}{4\sqrt{2}} z^{-3} \right) s(z^2)
\end{aligned}$$

and then solving for $s(z^2)$ we obtain,

$$s(z^2) = \frac{\frac{1-\sqrt{3}}{4\sqrt{2}} z^2 - \frac{3-\sqrt{3}}{4\sqrt{2}} z + \frac{\sqrt{3}-3}{\sqrt{2}} + \frac{3+\sqrt{3}}{4\sqrt{2}} + \frac{\sqrt{3}-1}{\sqrt{2}} z^{-1} - \frac{1+\sqrt{3}}{4\sqrt{2}} z^{-1}}{\frac{1+\sqrt{3}}{4\sqrt{2}} + \frac{3+\sqrt{3}}{4\sqrt{2}} z^{-1} + \frac{3-\sqrt{3}}{4\sqrt{2}} z^{-2} + \frac{1-\sqrt{3}}{4\sqrt{2}} z^{-3}} = (2 - \sqrt{3}) z^2$$

$s(z)$ can be obtained. Finally, by using Equation 3.14 on $s(z)$ the result can be taken inside the constant matrix K ,

$$s_m(z) = K^2 s(z) = \left(\frac{\sqrt{3}+1}{\sqrt{2}} \right)^2 s(z) = (2 + \sqrt{3}) s(z) = (2 + \sqrt{3})(2 - \sqrt{3}) z = z$$

to find the last triangular matrix for $P(z)$. This yields the final factorization for the filter pair as,

$$P(z) = \begin{bmatrix} 1 & -\sqrt{3} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \frac{\sqrt{3}}{4} + \frac{\sqrt{3}-2}{4}z^{-1} & 1 \end{bmatrix} \begin{bmatrix} 1 & z \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1+\sqrt{3}}{\sqrt{2}} & 0 \\ 0 & \frac{\sqrt{2}}{1+\sqrt{3}} \end{bmatrix}.$$

This example will be continued in Chapter 4.

CHAPTER 4

Lifted Wavelet Algorithm

4.1 Overview

The hardware design of the lifted wavelet algorithm developed in this chapter is created to be a single read - single write discrete wavelet transform design. By employing a single read - single write design method, the overall design becomes much more flexible. Portability is increased as a result of not having to work with specific connections to specific memories or memory controllers. The data can simply be passed into the design and collected at the output. Memory utilization is lower because there is no intermediate data that needs to be stored and the original data can be directly over written with the transform data. The speed is increased as a result of the reduced of memory accesses, as well as not having to process the entire image twice, once in each of the two dimensions. Also, the data flow is more direct, allowing an implementer to disregard the data is coming from and going to, be it from memory or directly from another process. Finally, with the additional design goal of a semi-generic wavelet, which allows for a wide variety of different wavelet filters, the proposed design becomes system independent.

However, the single read - single write implementation is a challenging implementation for a wavelet transform. The difficulty lies in one of the core benefits of the

wavelet transform, that it operates over an entire image rather than breaking it up into smaller more manageable pieces as the block-based DCT does[12]. This fact, combined with the two dimensional nature of an image makes it difficult to order the data of an image in such a way that allows the DWT to filter each data element twice, as is needed for the wavelet transform, without processing it in one direction, writing it out to memory and then processing that intermediate data in the other direction, as is the more straightforward approach in implementing wavelet transforms. This is called the intermediate data approach in the rest of this paper. The intermediate data approach requires that each pixel value of an image be accessed in memory and passed through the filters twice. An implementation for the intermediate data approach consists a technique that is capable of performing the one dimensional wavelet transform in either of the two dimensions, processing the image in one direction, writing and entire image worth of intermediate data to another entire image worth of addition memory, and then processing the intermediate data in the second dimension. Using the intermediate data method requires two memory accesses for each pixel, one for each directional dimension in the transform. This method slows the overall process by doubling the number of memory accesses as well as creating an inefficient amount of data that is never directly used.

The single read - single write method has a much more straightforward application, although it may be more complex for the design of DWT. The benefits of the single read - single write method, as discussed at the beginning of this chapter, result from each of the data elements needing to be accessed from memory only once, ordered so that the data can then be passed directly to a second filter and then written back to memory, directly over the original data. Writing over the original data eliminates

the need for the additional memory for the intermediate data, as well as the accesses needed to read and write from that memory. Therefore, the memory needs of the single read - single write method are nearly halved. reduced memory needs will be discussed in greater detail later in the chapter.

4.2 Implementation

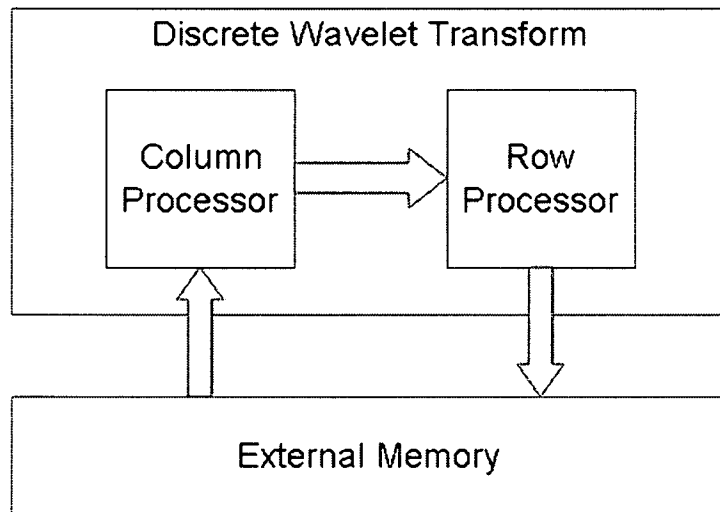


Figure 4.1: Wavelet Column and Row Processors

In order to achieve a single read - single write architecture, the DWT is broken down into two main blocks, a column processor and a row processor. This sets up a data flow that will allow the data to be moved from the column processor to the row processor directly, without returning to memory. This design strategy achieves the desired single read - single write effect. The method by which the data is allowed to be passed from one block to another without an intermediate write is a variation on memory management for the lifted wavelet[4]. The memory management variation described in the paper is implemented between the column and row processors,

in the current design it became more effective with with addition of the generality requirements for wavelets that the variation's implementation is moved into the row processor. The reason for the move will be discussed in more detail in section 4.2.2.

While it is stated that the order in which rows or columns are processed doesn't matter for the wavelet transform, that is not entirely true for the design. The order in which the processors are positioned is primarily determined by the JPEG2000 Standard. Floating point wavelet transforms are separable, which means that the order in which these processes are performed is of no consequence[3]. However, the integer-to-integer lifting approach introduces a non-linearity to the transform to achieve integer outputs, a loss of separability is the result, as noted in Chapters 2 and 3[5]. Therefore, the column processor is applied first because it is called for in the JPEG2000 Standard[1]. The row processor then naturally follows giving a design that is represented in Figure 4.1.

4.2.1 Column Processor

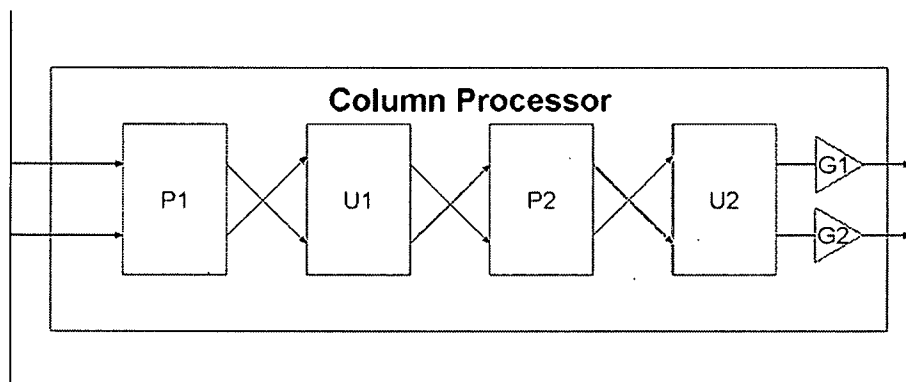


Figure 4.2: Column Processor

In Chapter 3, the polyphase matrix, $P(z)$, is presented and the de-construction of wavelet filters into polyphase matrices is discussed. It is important to understand how these polyphase matrices are used to construct filters, for a better understanding of how the column processor works. Internal to the column processor is a series of lifting steps. A step in the lifting process is a 2×2 matrix that results from the decomposition of the polyphase matrix as shown at the end of Chapter 3. These steps are alternately predict and update filters, and are indicated in Figure 4.2 by the P# or U#. If the example of the D4 polyphase matrix decomposition discussed at the end of Chapter 3 is picked up where it was left at the end of Chapter 3 where,

$$P(z) = \begin{bmatrix} 1 & -\sqrt{3} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \frac{\sqrt{3}}{4} + \frac{\sqrt{3}-2}{4}z^{-1} & 1 \end{bmatrix} \begin{bmatrix} 1 & z \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1+\sqrt{3}}{\sqrt{2}} & 0 \\ 0 & \frac{\sqrt{2}}{1+\sqrt{3}} \end{bmatrix}$$

is extended as an example, the conversion of matrix to filter equations is,

$$\begin{aligned} d_l^{(1)} &= x_{2l+1} - \sqrt{3}x_{2l} \\ s_l^{(1)} &= x_{2l} + \frac{\sqrt{3}}{4}d_l^{(1)} + \frac{\sqrt{3}-2}{4}d_{l+1}^{(1)} \\ d_l^{(2)} &= d_l^{(1)} + s_{l-1} \\ s_l &= \frac{\sqrt{3}+1}{\sqrt{2}}s_l^{(1)} \\ d_l &= \frac{\sqrt{2}}{1+\sqrt{3}}d_l^{(2)} = \frac{\sqrt{3}-1}{\sqrt{2}}d_l^{(2)}, \end{aligned}$$

where x_{2l} represents the even input image coefficients, and x_{2l+1} represents the odd input image coefficients. In these equations, d is the detail, or high-pass, coefficients and s represents the smooth, or low-pass, coefficients. The detail coefficients are the output of the update steps which are represented by the U# in Figure 4.2 and the smooth coefficients are the output of the predict steps which are represented by the P# steps also in Figure 4.2. In this example the equations shown lead to 3 lifting steps with gain constants.

Continuing the example above the equations are used to determine the two parallel data paths through each lifting step. that are seen in Figure 4.2. The equations for each step are,

$$\begin{aligned}
 \text{P1:} \quad & d_l^{(1)} = x_{2l+1} - \sqrt{3}x_{2l} \\
 & s_l^{(1)} = x_{2l} \\
 \text{U1:} \quad & d_l^{(2)} = d_l^{(1)} \\
 & s_l^{(2)} = s_l^{(1)} + \frac{\sqrt{3}}{4}d_l^{(1)} + \frac{\sqrt{3}-2}{4}d_{l+1}^{(1)} \\
 \text{P2:} \quad & d_l^{(3)} = d_l^{(2)} + s_{l-1}^{(2)} \\
 & s_l^{(3)} = s_l^{(2)}.
 \end{aligned}$$

While the example given has only three lifting steps and is not in one-to-one correspondence with the figure shown, it is useful to note that this design requires there be an even number of lifting steps. The even step requirement can easily be achieved by setting the last lifting step with the equations,

$$\begin{aligned}
 \text{U2:} \quad & d_l^{(4)} = d_l^{(3)} \\
 & s_l^{(4)} = s_l^{(3)}.
 \end{aligned}$$

This is the equivalent to adding a 2×2 identity matrix to the polyphase decomposition equation seen at the end of Chapter 3. By adding this last matrix, the last step is a pass through and adds a minimal amount of latency while meeting the even number of steps requirement. This allows wavelets filters whose decomposition leads to only three steps to still be implemented using this design. The reasons for the even step design requirement will be further discussed in Chapter 5. The final two equations are simply multipliers that come at the end, which can be seen represented by G1 and G2 shown in Figure 4.2.

In summary, the column processor receives the image data in a column-wise fashion. Reading the data in two pixels at a time from top to bottom and left to right. The data is then passed through a structure similar to the one outlined in this section and then out of the column processor and onto the row processor. It is useful to note that the column processor could be used to both the column processing and the row processing in a implementation similar to the intermediate data method described in section 4.1. In that case, the image data would initially be read into the column processor in the same fashion as before, however, it would be read to a secondary memory, and then the image stored in that secondary image data would then be read back into the column processor in row-wise fashion from left to right and then from top to bottom. The output of the column processor could then be read over the original data and ready for a second MR Level to be performed. This again illustrates the memory savings that the DWT design helps to create.

4.2.2 Row Processor

The row processor design is quite a bit different than the column processor. Internal to the row processor is a significant amount of memory, which is needed to re-order the data and then pass it into the row ordered lifting steps. The row processor architecture can be seen represented in figure 4.3. Specifically, there must be enough memory internal to the row processor to hold $8 \times \text{Image Height} \times 2 * (\# \text{ of lifting steps}) + 1$, in bits. The memory internal to the row processor is necessary in order to, reorder the image data so that the column ordered data from the column processor can be redistributed allowing the row processor to perform row processing. It is also for because of this internal memory that in all previous references to the single read

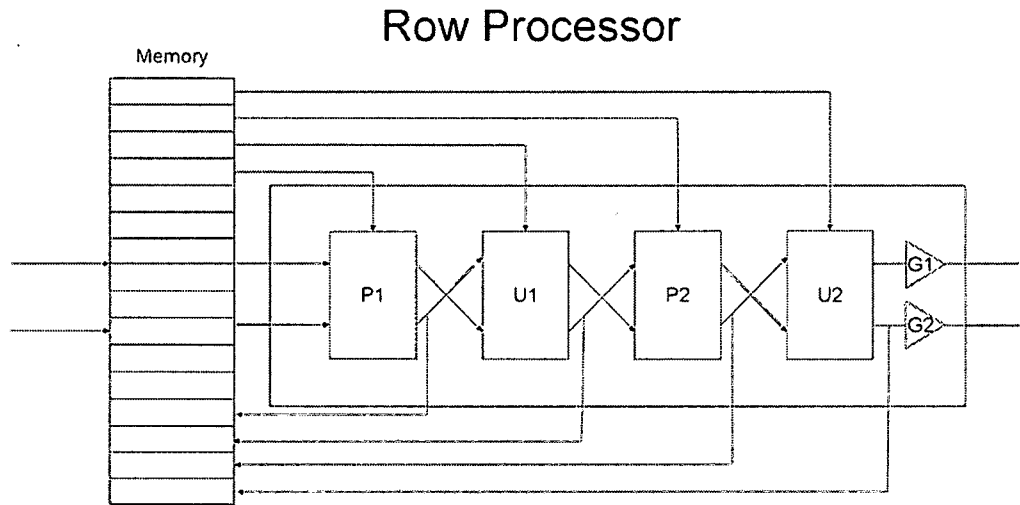


Figure 4.3: Row Processor

- single write method, it was stated that the memory was “nearly” halved. In order to get a better idea of how much memory is saved, Table 4.1 contains examples of memory usage in the row processor for both external and internal memory designs.

Wavelet Filter	Square Image Dimensions	Single read - Single Write vs. Intermediate Memory
Five-Three (minimum width 5)	512 pixels	0.5% memory utilization
Nine-Seven (minimum width 9)	512 pixels	1.3% memory utilization
Five-Three (minimum width 5)	1024 pixels	0.2% memory utilization
Nine-Seven (minimum width 9)	1024 pixels	0.6% memory utilization

Table 4.1: Memory Conservation Examples

The memory internal to the row processor is used to store a minimum amount of column data, n columns worth, coming from the column processor so that the

row processor can perform the first row-wise filter operation, where n is the length of the larger of the two wavelet filters. The trick to this technique is that instead of proceeding to the next filter position down the row, the first filter operation is performed on the first row-wise portions of the next row down. By iterating this process, the wavelet coefficients of the first row position of all of the rows is completed and the rows can be processed while minimally reordering the data from a column order. This method requires enough of the columns to be buffered to match the width of the filter, allowing the first lifting step to retrieve memory from three different columns. Each additional step obtains two of its inputs for the step before, and the third from addition column memory. By using this amount of memory there are enough columns memories' to allow access to all of the steps while the column ahead of the filter is being filled with the next column of image data, and the two behind to allow for the lag of the following steps, starting over at the top. It is this process that allows the data to be read into the column processor and then passed directly into the row processor without being intermediately stored or dramatically reordered.

However, by using this method two things about the row processor fundamentally change compared to what was described in the column processor. First, there is the need for much more memory than was allocated to the column processor. While the row processor memory is still much smaller than needed to perform the intermediate memory reordering, it is still more than was used in the column processor, and needs to be noted. The second fundamental difference between the column and row processors is the need for three inputs for each step, instead of the two inputs per step that are reflected in the column processor diagram in Figure 4.2. Three inputs are needed because in the column processor the data is being passed in in spacial order. As a

result, the column processor is allowed to store one of its past inputs as one of the filter inputs. Because the row processor is processing its data in column order instead of row order, the exact data that needs for each of the three filter inputs is needed to be passed to the each of the filters. The memory usage for re-ordering the data is reflected with the loops from memory to each of the row step in the row processor representation in Figure 4.3. A method by which so many simultaneous reads and writes maybe performed is described in Chapter 5.

4.2.3 Symmetric Extensions

Symmetric extensions are widely regarded as the best method for handling the image edges while utilizing the wavelet transform to achieve compression[13]. While using a standard filter bank approach dealing with the filter extensions can be a bottleneck in the filter process. The bottleneck results because when the filter bank is centered on the outermost edge of an image, the filter taps that are extended beyond the image have to be filled in order to achieve perfect reconstruction[13]. In the case of image transforms it will be symmetric data to the other side of the filter, thus "symmetric" extensions. While using filter banks to perform the image transform, extra time must be taken to fill those extra taps, or extra resources must be used to handle the condition. This is not the case when using the lifted wavelet approach. In the lifted wavelet approach there is never a loss of time and and only and extra multiplexor and 16-bit register to handle the edge conditions. More effective edge handling results because in the lifted wavelet filters the number of taps tied directly to the original data is minimal, only three in this design. Three taps allow the output of this filter that is tied to the original data to only reflect one data point, instead

of as many data points as half of a filter length using the filter bank method. This reduces not only the data that has to be reflected but squeezes the the inconsistency of an edge to a very narrow part of the filter. Therefore, in each step the step filter needs to be recognized, by means of an external signal, if it is on the edge of an image. If it is, then one of its three taps off the edge of the image is ignored and the other is simply doubled to handle the symmetric extensions.

4.3 Creating Generality

The method of creating generality in this filter is the same as would be done in a standard filter bank approach, by changing the filter coefficients of the smaller filters. Allowing the filter coefficients to change allows a virtually infinite number of possible filters of length n , where n is the filter length, using the filter bank method. However, in the lifted wavelet approach it is not the length of the filter that is limited, in fact steps can be added to increase the filter length indefinitely. It is the types of wavelet filter that can be created that is limited. This is because it is not mathematically possible to have all filters of any length decompose into only 3 tap filters. However, the lifting steps in this design have been developed to be low latency. Low latency in this case means spatially there will never be more than one look forward or one look behind to obtain data and the total smaller filter widths are limited to 3 taps each. Because the smaller filters of the lifted design are pipelined the first filter is looking at data that is further ahead spatially than that of the last filter. Pipelining allows the filter lengths to be expanded indefinitely by substituting spatial length for temporal length using pipelining of the filters.

CHAPTER 5

Hardware Implementation

While in chapter 4 the general principles and concepts that are used in the design of the lifted wavelet filter are discussed. This chapter discusses the design specifics of the actual hardware Implementation. These specifics include architecture diagrams for each of the blocks that are discussed in chapter 4 as well as detailed descriptions of each. Also, discussed are the design difficulties that effected the outcome of the overall design of each block. These are important because it is necessary to note how the original design concepts change due to specific implementation difficulties.

5.1 Synthesis Constraints

Synthesis constraints are design requirements that are not in the designers control. That is because the resources that are available on each platform will be different. This is further exacerbated because the design is intended to be generic enough to be used across many different platforms. Thus, the result is usually varying timing results as well as resource costs. The design is then an effective implementation on some devices, while on others, has unreasonable resource utilization. This makes it very difficult to design with synthesis requirements in mind. One method of managing this problem is to simply chose a specific platform and try and meet specific design requirement on that platform. Then if similar or more advanced platforms are used

in the future the resulting requirements should either be met or exceeded. This is the approach that is taken in the design of this DWT implementation and the constraints that are attempted to be met are seen in Table 5.1.

Constraint	Quantity
Timing	100 MHz Clock
Area	Less than 50% gates
Memory	Less than 76 KB*

Table 5.1: Synthesis Constraints

*This is the maximum internal block ram of the V2P7 Xilinx FPGA.

5.2 Implementing a Lifting Step

The most basic element of the lifted wavelet design is that lifting step. As described in Chapter 4, the lifting step is a three tap FIR filter. However, in order to get this simple three tap filter to operate as generically as possible the design is extended to accommodate several design specifics, including: whether it is a predict or update step as described in Section 4.2.1, creating the capability to change the filter coefficients, and the symmetric extension capability. There are also two different implementation of the lifting step, one for the column processor and the second for the row processor. This stems from the fact that, as mentioned in Section 4.2.2, the column processor lifting steps only have to accommodate two data inputs. This is because the data is read into the DWT in column-wise order. In the row processor, however, the lifting steps have to accommodate three inputs allowing for the input from memory in order to facilitate the reordering of the data that is necessary to perform the row-wise operations. The differences can be more clearly illustrated in Figures 4.2 and 4.3 with a more in depth analysis in Figures 5.1 and 5.2.

5.2.1 Implementing the Column Lifting Step

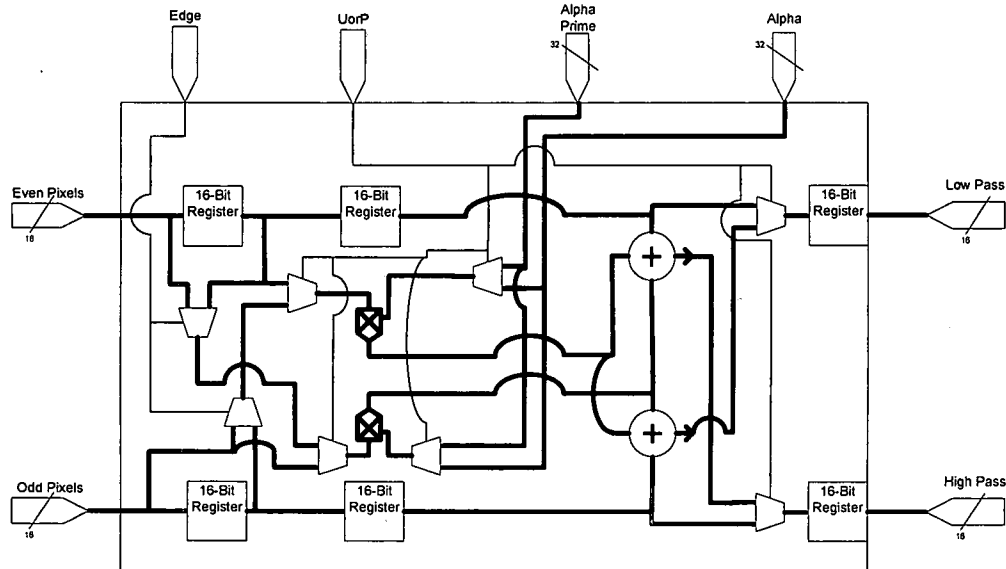


Figure 5.1: Column Lifting Step

The design of the lifting step changes to meet the synthesis constraints listed in Table 5.1. Specifically the timing constraint listed in Table 5.1. In the development of this design the Virtex 2 Pro package 7 with speed grade 5 is used. This is significant because if this design is built on that platform then all of the synthesis constraints listed should be met.

In Figure 5.1 more register can be seen than may seem necessary. The reason for the extra register is to allow for less path delay during each clock cycle. For instance, because the data is registered at the end of the multiplier, the data is not forced to continue through the adder which would elongate the data path and slow the overall timing of the design. In the column lifting step this modification was made from an earlier design to meet the timing requirement set in Table 5.1.

5.2.2 Implementing the Row Lifting Step

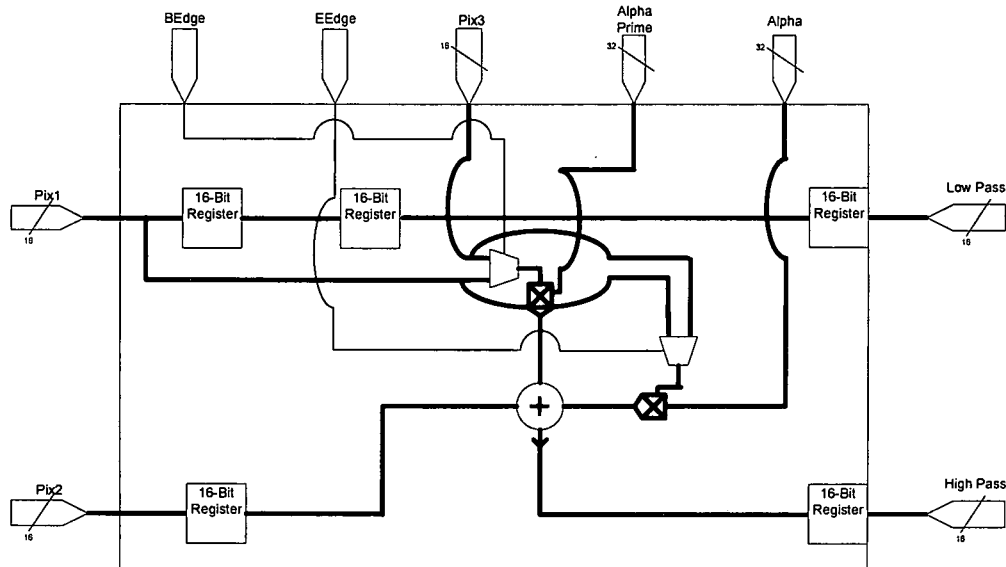


Figure 5.2: Row Lifting Step

The row step also has additional registers added to it in order to shorten data paths, allowing for the system timing requirements, set in Table 5.1, to be met. Note that the row processor design is far less complex than that of the column processor, illustrated in Figures 5.1 and 5.2. The lower complexity is the result of the addition of the third input. By allowing a third input the filter is a more direct implementation of the three tap filter equations that the hardware is meant to mimic. The more direct implementation makes the design simpler by forcing fewer conditions that need to be handled on the design.

5.2.3 Semi Generality

At this point the reason for designing a semi-generic filter instead of a completely generic one is addressed. It is address in this chapter instead of Chapter 4 because

there reason for this design change is purely form a practical hardware implementation point of view. It is the limiting of the smaller filters to three tap filters that forces this design to only be semi-general. As stated in Section 4.3 it is not mathematically possible to factor all possible wavelet filters in to three filter taps, and at this time, it is much more difficult to implement hardware filters that are variable in length. Hence one of the reasons lifting was developed, the filters are limited to fixed lengths.

Filter lengths of 3 where chosen for many reasons. First, filters of length two or three are the smallest filter size possible, any less is merely DC scaling. Filter length of three allows for some versatility allowing for both look-ahead's and delays. Finally, the three tap filters are the smallest lifting steps that allow for the implementation of both the Five-Three and the Nine-Seven wavelets, the two key kernels required for the implementation of JPEG2000.

5.3 Implementing the Row Processor Memory

One implementation problem that is essentially ignored in chapter 4 is memory accesses that are required for implementation of this design. The conceptual descriptions in chapter 4 calls for one write and one read to memory for every step in the wavelet transform, with and additional two reads for the first step. In the case of the Nine-seven wavelet, which was the highest length filter that was tested this would amount to six reads and four writes per system clock cycle. This is clearly not possible using any single memory implementation available today.

5.3.1 Memory Solution

The solution to the single clock read - write dilemma lies in the type of memory being used. By allowing each column of image data to be contained in a separate

memory, the problem is simplified to a maximum of two reads and one write per memory per clock cycle. In forcing the design to conform to this constraint, the reads and writes necessary are possible. While there are memories that have the functionality to implement the design at this point, the target FPGA has only dual port block rams internal to its design, which will only allow one read and one write on a single clock. Therefore, in order to further simplify memory requirements for the design each column of image data instead of being one memory, is two dual port memories. The first will memory will contain the first half on the column and the second the second. By using two memories per column it allows two reads and two writes per column of image data per clock. The number of reads and writes now allowed are more than enough to be able to implement the design, and because there is never the need to read twice or write twice to the same half of a column of image data, no functionality is lost.

CHAPTER 6

Conclusions & Future Work

In this thesis, a design and implementation for a semi-generic lifted discrete wavelet transform is presented. The design's primary application, as mentioned previously, is intended to be one of the key pieces of an hardware accelerated JPEG2000 design and implementation. However, the proposed design can be used in any design that requires the discrete wavelet transform where speed or dedicated hardware is an issue. One possible implementation includes, use on digital cameras. Currently most digital cameras use the original JPEG standard format to store the digital images when they are taken. In the future if the JPEG2000 standard truly becomes the replacement the original JPEG2000 then this design can be used to help implement the replacement dedicated hardware to do that. A second, implementation which was mentioned in this thesis is an implementation of Motion JPEG2000. Which would basically take a series of JPEG2000 image to create a video stream without cross compressing the image. In this instance the proposed design would help to meet of increase the speed at which an individual image could be compressed. This in turn would increase the potential frame rates that that implementation could achieve.

In the future there are several improvements that could be made to the proposed design to make it more effective for implementation. The first of which is to allow the design to complete more than one multi-resolution level. As the design stands now

only one multi-resolution level can be performed per implementation. This means that in order to do a five MR Level wavelet transform there would have to be five instantiations of this design in the completed design. This could become cumbersome in an design that calls for multiple MR Levels. However, because this design was started with the possibility of that in mind the transition to a design that would accommodate this capability should take very little effort.

Another possible design improvement is it incorporate the capability to change then filter coefficients on the fly. This would allow the type of wavelet filter that is used to be change electronically though software or simple changing a few register values, instead of having to re-flash the FPGA. This would potentially allow for different wavelet transform to be used for different MR Levels in the transformation of an image. Another alternative for this capability is to be able to switch between the two core wavelet kernels using the same instance of this design instead of having to have two instances to perform that task as is now necessary.

Most of the changes to the design presented in this chapter would be very easy to implement. This is because during the design stages these capacities were considered and going to be implemented should time have allowed. However, because time did not allow but the ground work has been laid for this capacities the changes to the overall design should be few. The desired synthesis constraints may however not hold as no testing was done that included these changes.

One final improvement that was never considered at the conception of this design would be to allow for variable step filter length. While this design change would involve more work than some of the other proposed in this chapter, by adding the flex ability of letting the lifting step filters be any length this design would no longer

be only semi-generic. Variable lifting step filter length would allow for any wavelet filter to be decomposed into lifting step and implemented using this design. This would make the overall design much more flexible. This is something that during the process of designing and implementing this design, it is now believed is not only possible but would take only moderately more work than those already proposed here.

There are also many more potential applications for this design than just that of the originally intended purpose of JPEG2000. One of those applications that might be aided by this design is signal processing in the wavelet domain. Dr. Eric Blaster has conducted some research in this area and he has found effective de-noising algorithms in the wavelet domain. These algorithms help to produce much less noisy images than the originals. He has also developed a video compression technique in the wavelet domain which finds motion in video streams and updates those area of an image at the tradition 30 frames a second while updating the stationary portions of a image at lower rates like 5 frames per second.

Any or all of the design improvement detailed in this chapter would increase the flexibility of this design. In the future with improvements in FPGA design as well as to the tools that are used to synthesis this design all of the proposed improvements may be able to be made with negligible resources and timing loss to the entire design.

APPENDIX A

HDL Code

DWT.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DWT is
generic(
    BitWidth    : integer := 9;
    MemDepth    : integer := 512;
    Steps       : integer := 4;
    PicHeight   : integer := 334;
    PicWidth    : integer := 600
);
    Port(
        clk      : in std_logic;
        VSync    : in std_logic;
        HSync    : in std_logic;
        EvenPix  : in std_logic_vector(15 downto 0);
        OddPix   : in std_logic_vector(15 downto 0);
        DataValid : out std_logic;
        LowPass  : out std_logic_vector(15 downto 0);
        HighPass : out std_logic_vector(15 downto 0)
    );
end DWT;

architecture behavioral of DWT is
    ----Column Input-----
    signal CValid : std_logic := '0';
    signal CLowPass : std_logic_vector(15 downto 0);
    signal CHighPass : std_logic_vector(15 downto 0);

    component ColumnProcessor
        generic(Steps, PicHeight, PicWidth : integer);
        Port(
            clk      : in std_logic;
            VSync    : in std_logic;
            HSync    : in std_logic;
            EvenPix  : in std_logic_vector(15 downto 0);
            OddPix   : in std_logic_vector(15 downto 0);
            DataValid : out std_logic := '0';
            LowPass  : out std_logic_vector(15 downto 0);
            HighPass : out std_logic_vector(15 downto 0)
        );
    end component;

    component RowProcessor
        generic(BitWidth, MemDepth, Steps, PicHeight, PicWidth : integer);
        Port(
            clk      : in std_logic;
            InputValid : in std_logic;
            EvenPix  : in std_logic_vector(15 downto 0);
            OddPix   : in std_logic_vector(15 downto 0);

```



```

        OutputValid : out std_logic := '0';
        LowPass      : out std_logic_vector(15 downto 0);
        HighPass      : out std_logic_vector(15 downto 0)
    );
end component;

begin

    Column : ColumnProcessor
    generic map(Steps, PicHeight, PicWidth)
    port map(
        clk => clk,
        VSync => VSync,
        HSync => HSync,
        EvenPix => EvenPix,
        OddPix => OddPix,
        DataValid => CValid,
        LowPass => CLowPass,
        HighPass => CHighPass
    );

    Row : RowProcessor
    generic map(BitWidth, MemDepth, Steps, PicHeight, PicWidth)
    port map(
        clk => clk,
        InputValid => CValid,
        EvenPix => CLowPass,
        OddPix => CHighPass,
        OutputValid => DataValid,
        LowPass => LowPass,
        HighPass => HighPass
    );

end behavioral;

```

ColumnProcessor.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity ColumnProcessor is
    generic(Steps, PicHeight, PicWidth : integer);
    Port(
        clk          : in std_logic;
        VSync        : in std_logic;
        HSync        : in std_logic;
        EvenPix       : in std_logic_vector(15 downto 0);
        OddPix        : in std_logic_vector(15 downto 0);
        DataValid     : out std_logic := '0';
        LowPass       : out std_logic_vector(15 downto 0);
        HighPass      : out std_logic_vector(15 downto 0)
    );
end ColumnProcessor;

architecture behavioral of ColumnProcessor is

    type Coefficients is array (0 to 2*Steps-1) of std_logic_vector(31 downto 0);
    constant Coeff : Coefficients := (
        --
        x"ff800000",x"ff800000",x"00400000",x"00400000");
        x"FE69F31A",x"FE69F31A",x"FFF26FE6",x"FFF26FE6",
        x"00E20675",x"00E20675",x"007189AA",x"007189AA");
        --
        x"FFD3F8BB",x"00000000",x"FF215090",x"00000000",
        --
        x"00264C79",x"00000000",x"01000000",x"00000000");

    constant UorP      : std_logic_vector(Steps-1 downto 0) :=
        CONV_STD_LOGIC_VECTOR(44739242,Steps);

    signal ColCount    : integer range 0 to PicWidth;
    signal RowCount    : integer range 0 to PicHeight/2-1;
    signal Bedge       : std_logic_vector(2*(Steps+1)+1 downto 0) :=
        CONV_STD_LOGIC_VECTOR(0,2*(Steps+1)+2);
    signal Eedge       : std_logic_vector(2*(Steps+1)+1 downto 0) :=
        CONV_STD_LOGIC_VECTOR(0,2*(Steps+1)+2);
    signal Edg         : std_logic_vector(Steps-1 downto 0);
    signal EndEdge     : std_logic := '0';
    signal Valid       : std_logic := '0';

    type data is array (0 to Steps-1) of std_logic_vector(15 downto 0);
    signal LowOut      : data;
    signal HighOut     : data;

    type ColState is (Idle, Run);
    signal CSt         : ColState := Idle;

```

```

--Column Instance
component Column
  Port (
    clk      : in std_logic;
    UorP     : in std_logic;
    Edg      : in std_logic;
    EvenPix  : in std_logic_vector(15 downto 0);
    OddPix   : in std_logic_vector(15 downto 0);
    Alpha    : in std_logic_vector(31 downto 0);
    AlphaP   : in std_logic_vector(31 downto 0);
    LowPass  : out std_logic_vector(15 downto 0);
    HighPass : out std_logic_vector(15 downto 0)
  );
end component;

begin
LowPass <= LowOut(Steps-1);
HighPass <= HighOut(Steps-1);
DataValid <= Valid;
C: for N in 0 to Steps/2-1 generate
  Edg(2*N) <= Eedge(N*5);
  Edg(2*N+1) <= Bedge(N*5+2);
end generate C;

S0 : Column
port map(
  clk => clk,                -- Clock
  UorP => UorP(0),           -- P Select
  Edg => Edg(0),             -- Edge Calc
  EvenPix => EvenPix,        -- EvenPix
  OddPix => OddPix,          -- OddPix
  Alpha => Coeff(0),         -- Alpha
  AlphaP => Coeff(1),        -- AlphaP
  LowPass => LowOut(0),      -- LowPass
  HighPass => HighOut(0)     -- HighPass
);

G: for N in 1 to Steps-1 generate
  S1 : Column
  port map(
    clk => clk,                -- Clock
    UorP => UorP(N),           -- P Select
    Edg => Edg(N),             -- Edge Calc
    EvenPix => LowOut(N-1),    -- EvenPix
    OddPix => HighOut(N-1),    -- OddPix
    Alpha => Coeff(2*N),       -- Alpha
    AlphaP => Coeff((2*N)+1),  -- AlphaP
    LowPass => LowOut(N),      -- LowPass
    HighPass => HighOut(N)     -- HighPass
  );
end generate G;

```

```

process(clk)
begin
    if(clk'event and clk = '1') then
        Bedge <= Bedge(2*(Steps+1) downto 0) & HSync;
        Eedge <= Eedge(2*(Steps+1) downto 0) & EndEdge;
        case CSt is
            when Idle =>
                RowCount <= 1;
                EndEdge <= '0';
                Valid <= '0';
                if HSync = '1' and VSync = '1' then
                    CSt <= Run;
                end if;
            when Run =>
                if RowCount = PicHeight/2-2 then
                    EndEdge <= '1';
                    RowCount <= RowCount + 1;
                elsif RowCount = PicHeight/2-1 then
                    EndEdge <= '0';
                    RowCount <= 0;
                    ColCount <= ColCount + 1;
                else
                    RowCount <= RowCount + 1;
                end if;
            end case;
            if ColCount = 0 and RowCount = 5*Steps/2-1 then
                Valid <= '1';
            elsif ColCount = PicWidth and RowCount = 5*Steps/2-1 then
                Valid <= '0'; CSt <= Idle;
            else
                Valid <= Valid;
            end if;
        end if;
    end Process;
end behavioral;

```

Column.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity Column is
    Port (
        clk          : in std_logic;
        UorP          : in std_logic;
        Edg          : in std_logic;
        EvenPix       : in std_logic_vector(15 downto 0);
        OddPix        : in std_logic_vector(15 downto 0);
        Alpha         : in std_logic_vector(31 downto 0);
        AlphaP        : in std_logic_vector(31 downto 0);
        LowPass       : out std_logic_vector(15 downto 0);
        HighPass      : out std_logic_vector(15 downto 0)
    );
end Column;

architecture Behavioral of Column is

    constant Half : std_logic_vector(47 downto 0) := x"0000008000000";
    signal Mult1 : std_logic_vector(47 downto 0);
    signal Mult2 : std_logic_vector(47 downto 0);
    signal Delay1 : std_logic_vector(15 downto 0);
    signal Delay2 : std_logic_vector(15 downto 0);
    signal Reg1 : std_logic_vector(15 downto 0);
    signal Reg2 : std_logic_vector(15 downto 0);

begin
    process(clk) begin
        if (clk'EVENT AND clk = '1') then
            Delay1 <= EvenPix; Reg1 <= Delay1;
            Delay2 <= OddPix; Reg2 <= Delay2;
            if (UorP) = '0' then
                Mult1 <= Alpha * Delay1;
                if (Edg = '1') then
                    Mult2 <= AlphaP * Delay1;
                else
                    Mult2 <= AlphaP * EvenPix;
                end if;
                LowPass <= Reg1;
                HighPass <= Reg2 + To_StdLogicVector(
                    (To_bitvector(Mult1 + Mult2 + Half) sra 24))(15 downto 0);
            else
                if (Edg = '1') then
                    Mult1 <= AlphaP * OddPix;
                else
                    Mult1 <= AlphaP * Delay2;
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

```
    Mult2 <= Alpha * OddPix;
    HighPass <= Delay2;
    LowPass <= Delay1 + To_StdLogicVector(
        (To_bitvector(Mult1 + Mult2 + Half) sra 24))(15 downto 0);
    end if;
end if;
end process;
end Behavioral;
```

RowProcessor.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity RowProcessor is
    generic(BitWidth, MemDepth, Steps, PicHeight, PicWidth : integer);
    Port(
        clk          : in std_logic;
        InputValid    : in std_logic;
        EvenPix       : in std_logic_vector(15 downto 0);
        OddPix        : in std_logic_vector(15 downto 0);
        OutputValid    : out std_logic := '0';
        LowPass       : out std_logic_vector(15 downto 0);
        HighPass      : out std_logic_vector(15 downto 0)
    );
end RowProcessor;

architecture behavioral of RowProcessor is

    ----Seek Function-----
    function Seek(B: integer; D: integer; O: character; W: integer) return integer is
        variable R: integer;
        constant P: integer := D/2;
    begin
        case O is
            when 'A' =>
                if (B + 1) = W/2 then R := P;
                elsif (B + 1) > (P + W/2-1) then R := 0;
                else R := B + 1; end if;
            when 'P' =>
                if (B + D) > W-1 then R := B + D - W;
                else R := B + D; end if;
            when 'M' =>
                if (B - D) < 0 then R := B - D + W;
                else R := B - D; end if;
            when others => R := B;
        end case;
        return R;
    end Seek;

    ----Control Signals-----
    signal ColumnIdx    : integer range 0 to Steps+2 := 0;
    signal RowIdx       : integer range 0 to PicHeight/2-1 := 0;

    type RowState is (Idle, Run);
    signal RSt         : RowState := Idle;

    signal ReadRowIdx   : integer range 0 to MemDepth/2+PicHeight/2-1 := 0;

    type WrBksRow is array (0 to Steps-2) of integer range 0 to MemDepth/2+PicHeight/2-1;

```



```

type dataOut is array (0 to Steps-1) of std_logic_vector(15 downto 0);
signal Pass      : dataOut;
signal Alter     : dataOut;

signal Bedg : std_logic_vector(Steps-1 downto 0):=conv_std_logic_vector(0,Steps);
signal Eedg : std_logic_vector(Steps-1 downto 0):=conv_std_logic_vector(0,Steps);

type Rdlys is array (0 to 2) of integer range 0 to (Steps+2);
type ReadIn is array (0 to Steps+1) of Rdlys;
signal MemRead      : ReadIn;

type Wdlys is array (0 to 3+2*(Steps-2)) of integer
                                range 0 to MemDepth/2+PicHeight/2-1;
signal WtDelay      : Wdlys;

type DlySig is array (0 to 2*Steps-3) of std_logic_vector(15 downto 0);
type NumDly is array (0 to Steps-2) of DlySig;
signal RdDelay      : NumDly;

signal OutDly1      : std_logic_vector(15 downto 0);
signal OutDly2      : std_logic_vector(15 downto 0);

component Row
  Port (
    clk          : in std_logic;
    BEdg         : in std_logic;
    EEdg         : in std_logic;
    Pix1         : in std_logic_vector(15 downto 0);
    Pix2         : in std_logic_vector(15 downto 0);
    Pix3         : in std_logic_vector(15 downto 0);
    Alpha        : in std_logic_vector(31 downto 0);
    AlphaP       : in std_logic_vector(31 downto 0);
    Pass         : out std_logic_vector(15 downto 0);
    Alter        : out std_logic_vector(15 downto 0)
  );
end component;

-----

begin
----Memory-----
GMem: for N in 0 to Steps+2 generate
  M0: OneColumn
    generic map(BitWidth, MemDepth)
    port map(
      clk          => clk,
      RdAddress    => ReadAdd(N),
      WtAddress1   => WriteAdd(2*N),
      WtAddress2   => WriteAdd(2*N+1),
      DataOut      => ReadOut(N),
      DataIn1      => WriteIn(2*N),
      DataIn2      => WriteIn(2*N+1),

```

```

        WtEnable1  => be(N),
        WtEnable2  => ce(N)
    );
end generate;
-----Row Operations-----
S0 : Row
    port map(
        clk => clk,                -- Clock
        BEdg => BEdg(0),            -- Edge Calc
        EEdg => EEdg(0),            -- Edge Calc
        Pix1 => Pix(0),
        Pix2 => Pix(1),
        Pix3 => Pix(2),
        Alpha => Coeff(0),          -- Alpha
        AlphaP => Coeff(1),         -- AlphaP
        Pass => Pass(0),
        Alter => Alter(0)
    );
GRow: for N in 1 to Steps-1 generate
    S1 : Row
        port map(
            clk => clk,                -- Clock
            BEdg => BEdg(N),          -- Edge Calc
            EEdg => EEdg(N),          -- Edge Calc
            Pix1 => Pix(N+2),
            Pix2 => Pass(N-1),
            Pix3 => Alter(N-1),
            Alpha => Coeff(2*N+(N mod 2)), -- Alpha
            AlphaP => Coeff(2*N+1-(N mod 2)), -- AlphaP
            Pass => Pass(N),
            Alter => Alter(N)
        );
end generate;
-----Input Control-----
InCtrl: process (clk)
begin
    if(clk'event and clk = '1') then
        InValid <= InputValid;
        if (InputValid = '1') then
            RowIdx <= Seek(RowIdx,1,'P',PicHeight/2);
            if (RowIdx = PicHeight/2-1) then
                ColumnIdx <= Seek(ColumnIdx,1,'P',Steps+3);
            end if;
        else
            RowIdx <= 0; ColumnIdx <= 0;
        end if;
    end if;
end process;
-----Read Control-----
RdCtrl: process (clk)
begin

```

```

if(clk'event and clk = '1') then
  case Rst is
    when Idle =>
      ReadRowIdx <= 0; Valid <= '0';
      ColCount <= Seek(0,Steps/2,'M',PicWidth);
      for N in 0 to Steps/2-1 loop
        BEdg(2*N) <= '0';BEdg(2*N+1) <= '1';
        EEdg(2*N) <= '0';EEdg(2*N+1) <= '0';
      end loop;
      for N in 0 to Steps + 1 loop
        MemRead(N)(0) <= Seek(2,N,'M',Steps+3);
      end loop;
      if ColumnIdx = 2 then
        RSt <= Run;
      end if;
    when Run =>
      ReadRowIdx <= Seek(ReadRowIdx,MemDepth,'A',PicHeight);
      if ReadRowIdx = MemDepth/2 + PicHeight/2 - 1 then
        MemRead(0)(0) <= Seek(MemRead(0)(0),2,'P',Steps+3);
        MemRead(1)(0) <= Seek(MemRead(1)(0),2,'P',Steps+3);
        for N in 0 to Steps-1 loop
          MemRead(N+2)(0) <= MemRead(N)(0);
        end loop;
      end if;
      for N in 0 to Steps + 1 loop
        for M in 0 to 1 loop
          MemRead(N)(M+1) <= MemRead(N)(M);
        end loop;
      end loop;
      for N in 0 to Steps/2-1 loop
        if MemRead(2*N)(0) = 4 and ReadRowIdx = (N+1)*4 then
          BEdg(2*N+1) <= '0';
        end if;
        if ColCount = PicWidth/2-1-Steps/2+N and
           ReadRowIdx = 4*(N+1)-2 then
          EEdg(2*N) <= '1';
        end if;
      end loop;
      if ReadRowIdx = Steps*2+2 then
        ColCount <= Seek(ColCount,1,'P',PicWidth);
        if ColCount = PicWidth/2-1 then
          Valid <= not Valid;
          RSt <= Idle;
        elsif ColCount = PicWidth-1 then
          Valid <= not Valid;
        end if;
      end if;
    end case;
  end if;
end process;
-----Write Control-----

```

```

WtCtrl: process (clk)
begin
    if (clk'event and clk = '1') then
        if InputValid = '1' or Valid = '1' then
            if RowIdx = 0 and (InputValid = '1' or InValid = '1') then
                be(ColumnIdx) <= '1'; be(Seek(ColumnIdx,1,'M',Steps+3)) <= '0';
                ce(ColumnIdx) <= '1'; ce(Seek(ColumnIdx,1,'M',Steps+3)) <= '0';
            end if;
        else
            be <= conv_std_logic_vector(0,Steps+3);
            ce <= conv_std_logic_vector(0,Steps+3);
        end if;
        case Rst is
            when Idle =>
                for N in 0 to Steps - 2 loop
                    WrRow(N) <= Seek(0,2,'M',PicHeight);
                    WrCol(N) <= Seek(Steps+2,N,'M',Steps+3);
                end loop;
                for N in 0 to 3+2*(Steps-2) loop
                    WtDelay(N) <= MemDepth/2+PicHeight/2-1-N;
                end loop;

                when Run =>
                    WtDelay(0) <= ReadRowIdx;
                    for N in 0 to 2+2*(Steps-2) loop
                        WtDelay(N+1) <= WtDelay(N);
                    end loop;
                    for N in 0 to Steps - 2 loop
                        WrRow(N) <= WtDelay(3+2*N);
                    end loop;
                    for N in 0 to Steps - 2 loop
                        if WrRow(N) = 0 then
                            be(WrCol(N)) <= '1';
                            be(Seek(WrCol(N),2,'M',Steps+3)) <= '0';
                        elsif WrRow(N) = MemDepth/2 + PicHeight/2-1 then
                            WrCol(N) <= Seek(WrCol(N),2,'P',Steps+3);
                        end if;
                    end loop;
                end case;
        end if;
    end process;
    ----Read-----
    Rd: process (clk)
    begin
        if (clk'event and clk = '1') then
            -- Step One Inputs
            for N in 0 to 2 loop
                Pix(N) <= ReadOut(MemRead(2-N)(2));
                ReadAdd(MemRead(N)(0)) <= conv_std_logic_vector(ReadRowIdx,BitWidth);
            end loop;
            -- Delays

```

```

    for N in 0 to Steps-2 loop
        RdDelay(N)(0) <= ReadOut(MemRead(N+3)(2));
        ReadAdd(MemRead(N+3)(0)) <= conv_std_logic_vector(ReadRowIdx, BitWidth);
    end loop;
    for N in 0 to Steps-2 loop
        for M in 0 to 2*N loop
            RdDelay(N)(M+1) <= RdDelay(N)(M);
        end loop;
    end loop;
    -- Following Steps
    for N in 0 to Steps-2 loop
        Pix(N+3) <= RdDelay(N)(2*N+1);
    end loop;
end if;
end process;
-----Write-----
Wt: process (clk)
begin
    if (clk'event and clk = '1') then
        -- For Inputs from Column Processor
        WriteAdd(2*ColumnIdx) <= conv_std_logic_vector(RowIdx, BitWidth);
        WriteAdd(2*ColumnIdx+1) <= '1' & conv_std_logic_vector(RowIdx, BitWidth-1);
        WriteIn(2*ColumnIdx) <= EvenPix;
        WriteIn(2*ColumnIdx+1) <= OddPix;
        -- Write Back
        for N in 0 to Steps - 2 loop
            WriteAdd(2*WrCol(N)) <= conv_std_logic_vector(WrRow(N), BitWidth);
            WriteIn(2*WrCol(N)) <= Alter(N);
        end loop;
        OutDly1 <= Alter(Steps-2);
        OutDly2 <= OutDly1;
    end if;
end process;
-----Outputs-----
OutputValid <= Valid;
LowPass <= Alter(Steps-1);
HighPass <= OutDly2;
-----
end behavioral;

```

Row.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity Row is
    Port (
        clk          : in std_logic;
        BEdg         : in std_logic;
        EEdg         : in std_logic;
        Pix1          : in std_logic_vector(15 downto 0);
        Pix2          : in std_logic_vector(15 downto 0);
        Pix3          : in std_logic_vector(15 downto 0);
        Alpha         : in std_logic_vector(31 downto 0);
        AlphaP        : in std_logic_vector(31 downto 0);
        Pass          : out std_logic_vector(15 downto 0);
        Alter         : out std_logic_vector(15 downto 0)
    );
end Row;

architecture Behavioral of Row is

    constant Half : std_logic_vector(47 downto 0) := x"0000008000000";

    signal Mult1      : std_logic_vector(47 downto 0);
    signal Mult2      : std_logic_vector(47 downto 0);
    signal Delay       : std_logic_vector(15 downto 0);
    signal PassDelay   : std_logic_vector(15 downto 0);

begin
    process(clk) begin
        if (clk'EVENT AND clk = '1') then
            PassDelay <= Pix1;
            Pass <= PassDelay;
            Delay <= Pix2;
            if BEdg = '1' then
                Mult1 <= Alpha * Pix3;
                Mult2 <= AlphaP * Pix3;
            elsif EEdg = '1' then
                Mult1 <= Alpha * Pix1;
                Mult2 <= AlphaP * Pix1;
            else
                Mult1 <= Alpha * Pix1;
                Mult2 <= AlphaP * Pix3;
            end if;
            Alter <= Delay + To_StdLogicVector(
                (To_bitvector(Mult1 + Mult2 + Half) sra 24))(15 downto 0);
        end if;
    end process;
end Behavioral;
```

OneColumn.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity OneColumn is
    generic(BitWidth, MemDepth : integer);
    port (
        clk          : in std_logic;
        RdAddress    : in std_logic_vector(BitWidth-1 downto 0);
        WtAddress1   : in std_logic_vector(BitWidth-1 downto 0);
        WtAddress2   : in std_logic_vector(BitWidth-1 downto 0);
        DataOut      : out std_logic_vector(15 downto 0);
        DataIn1      : in std_logic_vector(15 downto 0);
        DataIn2      : in std_logic_vector(15 downto 0);
        WtEnable1    : in std_logic;
        WtEnable2    : in std_logic
    );
end OneColumn;

architecture behavioral of OneColumn is

    component Mem
        generic(BitWidth, MemDepth : integer);
        port (
            addra      : in std_logic_VECTOR(BitWidth-1 downto 0);
            addrb      : in std_logic_VECTOR(BitWidth-1 downto 0);
            clka       : in std_logic;
            clkb       : in std_logic;
            dinb       : in std_logic_VECTOR(15 downto 0);
            douta      : out std_logic_VECTOR(15 downto 0);
            web        : in std_logic
        );
    end component;

    signal be          : std_logic;
    signal ce          : std_logic;
    signal test        : std_logic;
    signal betest      : std_logic;
    signal cetest      : std_logic;
    signal din1test    : std_logic;
    signal din2test    : std_logic;
    signal din1        : std_logic_vector(15 downto 0);
    signal din2        : std_logic_vector(15 downto 0);
    signal dout1       : std_logic_vector(15 downto 0);
    signal dout2       : std_logic_vector(15 downto 0);
    signal waddr1      : std_logic_vector(BitWidth-2 downto 0);
    signal waddr2      : std_logic_vector(BitWidth-2 downto 0);

begin
    betest <= (WtEnable1 and not WtAddress1(BitWidth-1)) or
```

```

        (WtEnable2 and not WtAddress2(BitWidth-1));
be <= '1' when betest = '1' else '0';

cetest <= (WtEnable1 and WtAddress1(BitWidth-1)) or
        (WtEnable2 and WtAddress2(BitWidth-1));
ce <= '1' when cetest = '1' else '0';

din1test <= (WtEnable2 and not WtAddress2(BitWidth-1));
din1 <= DataIn2 when din1test = '1' else DataIn1;

din2test <= (WtEnable1 and WtAddress1(BitWidth-1));
din2 <= DataIn1 when din2test = '1' else DataIn2;

waddr1 <= WtAddress2(BitWidth-2 downto 0) when din1test = '1'
        else WtAddress1(BitWidth-2 downto 0);
waddr2 <= WtAddress1(BitWidth-2 downto 0) when din2test = '1'
        else WtAddress2(BitWidth-2 downto 0);

DataOut <= dout2 when test = '1' else dout1;

M0: Mem
generic map(BitWidth-1, MemDepth/2)
port map(
    addra => RdAddress(BitWidth-2 downto 0),
    addrb => waddr1(BitWidth-2 downto 0),
    clka => clk,
    clkb => clk,
    dinb => din1,
    douta => dout1,
    web => be
);

M1: Mem
generic map(BitWidth-1, MemDepth/2)
port map(
    addra => RdAddress(BitWidth-2 downto 0),
    addrb => waddr2(BitWidth-2 downto 0),
    clka => clk,
    clkb => clk,
    dinb => din2,
    douta => dout2,
    web => ce
);

process(clk)
begin
    if(clk'event and clk = '1') then
        test <= RdAddress(BitWidth-1);
    end if;
end process;
end behavioral;

```


Mem.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Mem is
    generic(BitWidth, MemDepth : integer);
    port (
        addra      : in std_logic_VECTOR(BitWidth-1 downto 0);
        addrb      : in std_logic_VECTOR(BitWidth-1 downto 0);
        clka       : in std_logic;
        clkb       : in std_logic;
        dinb       : in std_logic_VECTOR(15 downto 0);
        douta      : out std_logic_VECTOR(15 downto 0);
        web        : in std_logic
    );
end mem;

architecture behavioral of mem is

    type Column is array (0 to MemDepth-1) of std_logic_vector(15 downto 0);
    signal Pixels : Column;

begin
    process (clka)
    begin
        if(clka'event and clka = '1') then
            douta <= Pixels(conv_integer(addra));
        end if;
    end process;
    process (clkb)
    begin
        if(clkb'event and clkb = '1' and web = '1') then
            Pixels(conv_integer(addrb)) <= dinb;
        end if;
    end process;
end behavioral;
```

BIBLIOGRAPHY

- [1] Tinku Acharya and Ping-Sing Tsai. *JPEG2000 Standard for Image Compression*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2005.
- [2] Michael D. Adams and Faouzi Kossentini. "Reversible Integer-to-Integer Wavelet Transforms for Image Compression: Performance Evaluation and Analysis". *IEEE Transactions on Image Processing*, 9(4), 2000.
- [3] Eric J. Balster. "VIDEO COMPRESSION AND RATE CONTROL METHODS BASED ON THE WAVELET TRANSFORM". 2004.
- [4] S. Barua, J.E. Carletta, K.A. Kotteri, and A.E.Bell. "An Efficient Architecture for Lifting-based Two-Dimensional Discrete Wavelet Transforms". *INTEGRATION, the VLSI journal*, 38:341-352, 2005.
- [5] R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo. "Wavelet transforms that map integers to integers". *Appl. Comput. Harmon. Anal.*, 5(3):332-369, 1998.
- [6] I. Daubechies and W. Sweldens. "Factoring Wavelet Transforms into Lifting Steps". *J. Fourier Anal. Appl.*, 4(3):245-267, 1998.
- [7] Pascal Getreuer. "Filter Coefficients to Popular Wavelets". 2004.
- [8] Jaideva C. Goswami and Andrew K. Chan. *Fundamentals of Wavelets*. John Wiley & Sons, Inc., New York, New York, 1999.
- [9] Gaetano Impoco. Jpeg2000 - a short tutorial. 2004.
- [10] A. Jensen and A. la Cour-Harbo. *Ripples in Mathematics: The Discrete Wavelet Transform*. Springer, New York, New York, 2001.
- [11] Wenqing Jiang and Antonio Ortega. "Lifting Factorization-Based Discrete Wavelet Transform Architecture Design". *IEEE Transactions on Circuits and Systems for Video Technology*, 11(5):651-657, May 2001.
- [12] J.J. Hwang K.R. Rao. *Techniques & Standards for Image, Video & Audio Coding*. Prentice Hall PTR, Upper Saddle River, New Jersey, 1996.
- [13] S. Li and W. Li. "Shape-Adaptive Discrete Wavelet Transforms for Arbitrarily Shaped Visual Object Coding". *IEEE Transactions on Circuits and Systems for Video Technology*, 10(5):725-743, August 2000.

R002588762

- [14] Michael B. Martin and Amy E. Bell. "New Image Compression Techniques Using Multiwavelets and Multiwavelet Packets". *IEEE TRANSACTIONS ON IMAGE PROCESSING*, 10(4).
- [15] F. McMahon. "JPEG2000". *Digital Output*, June. 2002.
- [16] S.R. Subramanya. "Image Compression Techniques". *IEEE Potentials*, 20(1), 2001.
- [17] W. Sweldens. "The Lifting Scheme: A New Philosophy in Biorthogonal Wavelet Constructions". In A. F. Laine and M. Unser, editors, *Wavelet Applications in Signal and Image Processing III*, pages 68–79. Proc. SPIE 2569, 1995.
- [18] W. Sweldens. "Wavelets and the lifting scheme: A 5 minute tour". *Z. Angew. Math. Mech.*, 76 (Suppl. 2):41–44, 1996.
- [19] W. Sweldens. "The lifting scheme: A construction of second generation wavelets". *SIAM J. Math. Anal.*, 29(2):511–546, 1997.
- [20] David Taubman. "High Performance Scalable Image Compression with EBCOT". *IEEE Transactions on Image Processing*, 9(7):1158–1170, July 2000.
- [21] David S. Taubman and Michael W. Marcellin. *JPEG2000 Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, Boston, Mass., 2002.
- [22] William F. Turri. "DESIGN AND IMPLEMENTATION OF A SUPER-EFFICIENT WAVELET TRANSFORM FOR COMPRESSION OF COLOR IMAGES". 2002.
- [23] James S. Walker. *"A Primer On Wavelets and their Scientific Applications"*. Chapman & Hall/CRC, Boca Raton, FL, 1999.
- [24] Gregory K. Wallace. "The JPEG Still Picture Compression Standard". *IEEE Transactions on Consumer Electronics*, 1991.