JPEG2000 ENCODER IMPLEMENTATION USING VHDL

A Dissertation Submitted in partial fulfillment of the requirements for the award of the degree of

MASTER OF ENGINEERING
(Electronics &Communication Engineering)

Submitted by:
SREENIVAS BACHCHU
College Roll No. 07/E&C/04
University Roll No. 8737

Under the guidance of
Mrs. RAJESHWARI PANDEY

DEPARTMENT OF ELECTRONICS &COMMUNICATION ENGINEERING
DELHI COLLEGE OF ENGINEERING
BAWANA ROAD, DELHI-110042
(UNIVERSITY OF DELHI)

JUNE 2006

CERTIFICATE

It is to certify that the work that is being presented in this Dissertation entitled **"JPEG2000 ENCODER IMPLEMENTATION USING VHDL**", in partial fulfillment of the requirement for the award of the degree of Master of Engineering in Electronics &Communication submitted by ***Sreenivas Bachchu*** (07/E&C/04) to the Department of Electronics &Communication Engineering, Delhi College of Engineering, is the record of the student's own work that is carried out under my supervision and guidance.

Mrs. Rajeshwari Pandey
Dept. of Electronics & Communication Engineering
Delhi College of Engineering

ACKNOWLEDGEMENTS

It is a great pleasure to have the opportunity to extend my heartiest felt gratitude to everybody who helped me throughout the course of this project.

I would like to express my heartiest felt regards to **Mrs. Rajeshwari Pandey,** Lecturer Department of Electronics and Communications Engineering for her constant motivation and support for the entire duration of this project. It is my privilege and honor to have worked under her supervision. Her invaluable guidance and helpful discussions in every stage of this project really helped me in materializing the project.

I would like to thank **Prof. ASOK BATTACHARYYA**, Head of the department, Electronics Communications Engineering, for providing facilities in doing this project.

I am thankful to my friends and classmates for their unconditional support and motivation during this project.

Sreenivas Bachchu

M.E. (Electronics Communication Engg)

College Roll No. 07/E&C/04

Delhi University Roll No. 8737

ABSTRACT

The new still compression image standard, JPEG2000, has emerged with a number of significant features that would allow it to be used efficiently over a wide variety of images. The scalability of the new standard is intended to allow trading off between compression rates and quality of images. Due to multi-resolution nature of wavelet transforms, they have been adopted by the JPEG2000 standard as the transform of choice.

In this Dissertation, an implementation for a reconfigurable fully scalable Integer Wavelet Transform (IWT) unit that satisfies the specifications of the JPEG2000 standard has been presented. The implementation is based on the lifting scheme, which is the most computation efficient implementation of the discrete wavelet transform.

This project aims to provide modules written in the Very High Speed Integrated Circuit Hardware Description Language (VHDL) that can be used to accelerate an existing software implementation of JPEG2000.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 WHAT IS A DIGITAL IMAGE?

An image may be defined as a two dimensional function, ***f (x, y)***, where x and y are spatial (plane) coordinates, and the amplitude of f at any pair of coordinates *(x, y)* is called the *intensity or gray level* of the image at that point. When x, y and the amplitude of ***f*** are all finite discrete quantities then we call the image a digital image.

1.2 WHY DO WE NEED COMPRESSION?

The figures in Table 1 show the qualitative transition from simple text to full-motion video data and the disk space, transmission bandwidth, and transmission time needed to store and transmit such uncompressed data. It clearly illustrates the need for sufficient storage space, large transmission bandwidth, and long transmission time for image, audio, and video data. At the present state of technology, the only solution is to compress multimedia data before its storage and transmission, and decompress it at the receiver for play back. For example, with compression ratio of 32:1, the space, bandwidth, and transmission time requirements can be reduced by a factor of 32, with acceptable quality.

TABLE 1 Multimedia data types and uncompressed storage space, transmission bandwidth, and transmission time required. The prefix kilo-denotes a factor of 1000 rather than 1024.

Multimedia Data	Size/Duration	Bits/Pixel or Bits/Sample	Uncompressed Size
A page of text	11" x 8.5"	Varying resolution	16-32 Kbits
Telephone quality speech	1 sec	8 bps	64 Kbits
Grayscale Image	512 x 512	8 bpp	2.1 Mbits
Color Image	512 x 512	24 bpp	6.29 Mbits
Medical Image	2048 x 1680	12 bpp	41.3 Mbits
Full-motion Video	640 x 480, 10 sec	24 bpp	2.21 Gbits

1.3 PRINCIPLES OF IMAGE COMPRESSION

A common characteristic of most images is that the neighboring pixels are correlated and therefore contain redundant information. The foremost task then is to find less correlated representation of the image. Two fundamental components of compression are redundancy and irrelevancy reduction. **Redundancy reduction** aims at removing duplication from the signal source (image/video). **Irrelevancy reduction** omits parts of the signal that will not be noticed by the signal receiver, namely the Human Visual System (HVS). In general, three types of redundancy can be identified:

- **Interpixel (Spatial) Redundancy** or correlation between neighboring pixel values.
- **Coding Redundancy** usage of more code symbols than needed.
- **Psychovisual redundancy** more information than HVS can process.

1.4 GOAL OF IMAGE COMPRESSION

Image compression research aims at reducing the number of bits needed to represent an image by removing the spatial and spectral redundancies as much as possible while maintaining and acceptable quality and intelligibility.

1.5 TYPICAL ENVIRONMENT FOR IMAGE COMPRESSION

A typical environment for image compression is shown in Figure 1.1. The digital image is encoded by an image coder. The output of the image coder is a string of bits that represents the source image. The channel coder transforms string of bits to a form suitable for transmission over a communication channel. At the receiver, the received signal is transformed back into a string of bits by a channel decoder. The image decoder reconstructs the image from the string of bits. In contrast to the communication environment in Figure 1.1, no communication channel is involved in application of image coding for purpose of storage.

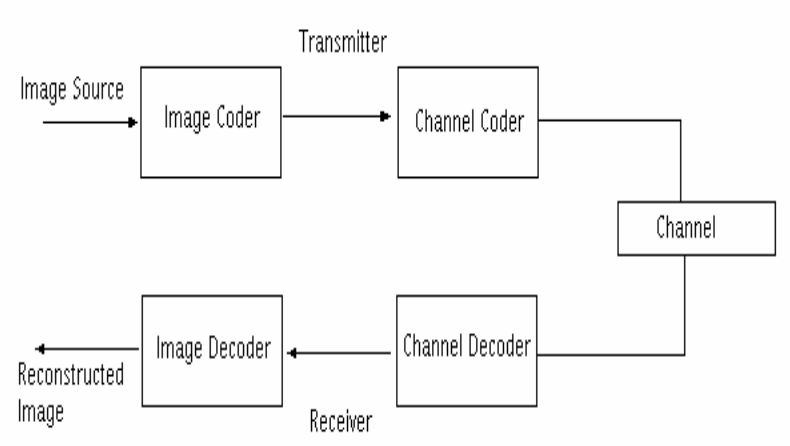


FIG. 1.1 Typical Environment for Image Coding

The image coder in Figure 1.1 consists of three closely connected components viz. (a) Source Encoder or Linear Transforms, (b) Quantizer, and (c) Entropy Encoder, shown in Figure 1.2. Compression is accomplished by applying a linear transform to decorrelate the image data, quantizing the resulting transform coefficients and entropy coding the quantized values.

1.5.1 *Source Encoder*

The source encoder is responsible for reducing or eliminating any coding, interpixel, or psychovisual redundancies in the input image. A variety of linear transforms are available like Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT), and Discrete Wavelet Transform (DWT) for this purpose. This operation is reversible and may or may not reduce directly the amount of data required to represent the image.

1.5.2 *Quantizer*

A Quantizer reduces the precision of the values generated from the encoder and therefore reduces the number of bits required to save the transform co-coefficients. This process is lossy. Quantization can be performed on each individual

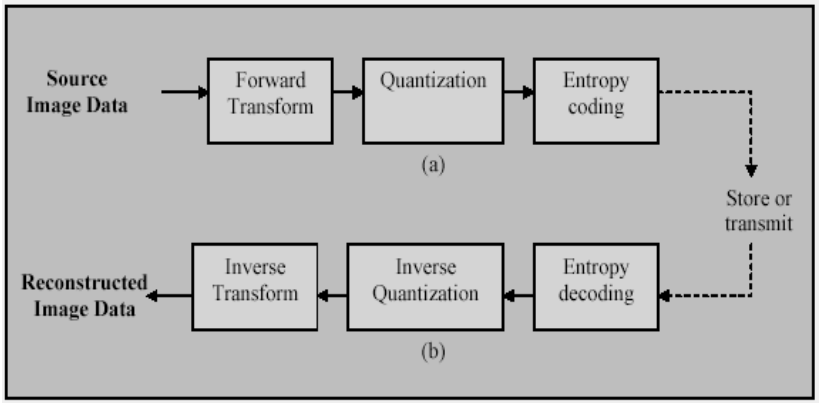


FIG. 1.2 Typical Structured Image Compression System

coefficient i.e. Scalar Quantization (SQ) or it can be performed on a group of coefficients together i.e. Vector Quantization (VQ).

1.5.3 *Entropy Encoder*

An entropy encoder does further compress the quantized values. This is done to achieve even better overall compression. The various commonly used entropy encoders are the Huffman encoder, arithmetic encoder, and simple run-length encoder. For better performance with compression, it's important to have the best of all the three components.

1.6 TYPES OF IMAGE COMPRESSION TECHNIQUES

There are different schemes for classifying compression techniques. Two of these schemes, described in this report, are:

1.6.1 *Lossless vs. Lossy compression***:**

The first categorization is based on the information content of the reconstructed image. They are 'lossless compression' and 'lossy compression' schemes. In lossless compression, the reconstructed image after compression is numerically identical to the original image on a pixel-by-pixel basis. However, only a modest amount of compression is achievable in this technique. In lossy compression

on the other hand, the reconstructed image contains degradation relative to the original, because redundant information is discarded during compression. As a result, much higher compression is achievable, and under normal viewing conditions, no visible loss is perceived (visually lossless). Performance analysis of these schemes is shown in Fig. 1.3 below.

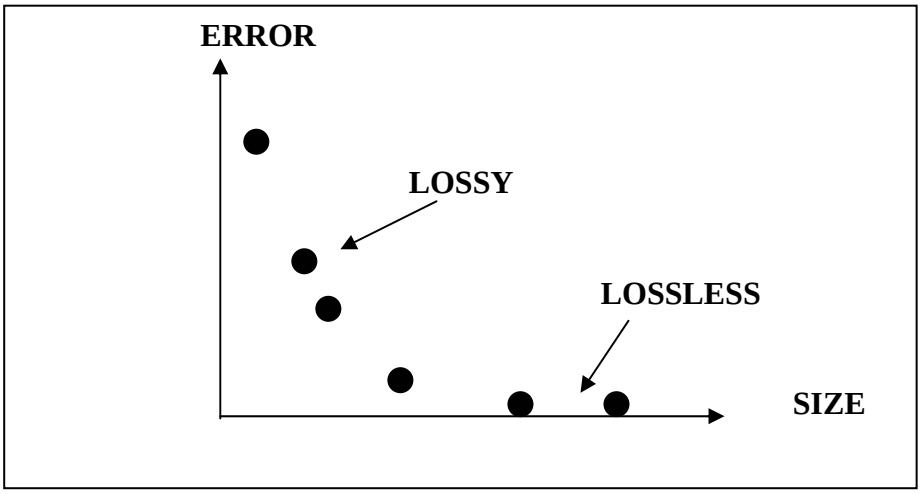


FIG. 1.3 Performance Analysis

1.6.2 *Predictive Vs. Transform coding.*

The second categorization of various coding schemes is based on the 'space' where the compression method is applied. These are 'predictive coding' and 'transform coding'. In predictive coding, information already sent or available is used to predict future values, and the difference is coded. Since this is done in the image or spatial domain, it is relatively simple to implement and is readily adapted to local image characteristics.

Transform coding, on the other hand, first transforms the image from its spatial domain representation to a different type of representation using some well-known transforms mentioned later, and then codes the transformed values (coefficients). Some of these are: Karhunen-Loeve Transform (KLT), Discrete Fourier Transform (DFT), Discrete Cosine Transform (DCT), and Discrete Wavelet Transform (DWT). The primary advantage is that, it provides greater data compression compared to predictive methods, although at the expense of greater computations.

1.7 ORGANIZATION OF DISSERTATION

The remainder of the thesis is organized as follows. Chapter 2.Presents an overview of the Discrete Wavelet Transform. And various filter banks. Chapter3 deals with brief overview of project and background of JPEG2000. Chapter 4 discusses about software implementation of various stages in JPEG 2000. Chapter 5 discusses the results and hardware performance. Chapter 6 gives various the applications of JPEG2000. Chapter 7 concludes with recommendations on the best filter bank structure and coefficient quantization scheme for an FPGA DWT implementation, and makes suggestions for future work.

CHAPTER 2

PRINCIPLES OF WAVELET THEORY

2.1 HISTORICAL PERSPECTIVE

Wavelets are mathematical functions that cup data into different frequency components, and then study each component with a resolution matched to its scale. This idea is not new. Approximation using superposition of functions has existed since early 1800's when Joseph Fourier discovered that he could superpose sines and cosines to represent other functions. However in wavelet analysis, the scale that we use to look at plays a special role. In the history of mathematics, wavelet analysis shows many different origins:

2.1.1 *Pre 1930*

Before 1930, main branch of mathematics leading to wavelets began with Joseph Fourier with his theories of frequency analysis. He asserted that any 2Π periodic function f(x) is the sum

$$a_0 + \sum_{k=1}^{\infty} a_k \cos kx + b_k \sin kx$$

of its Fourier series. The coefficients a_0, a_k, and b_k are calculated by

$$a_0 = \frac{1}{2\pi}\int_0^{2\pi} f(x)dx \qquad a_k = \frac{1}{\pi}\int_0^{2\pi} f(x)\cos(kx)dx \qquad b_k = \frac{1}{\pi}\int_0^{2\pi} f(x)\sin(kx)dx$$

After 1807, by exploring means of functions, Fourier series convergence and orthogonal systems, gradually led from their previous notion of *frequency analysis* to the notion of *scale analysis. i.e.,* analyzing f(x) by creating mathematical structures that vary in scale.

2.1.2 *The 1930s*

In 1930s several groups working independently researched the representation of functions using scale varying basis functions. By using a scale varying basis function called the Haar basis function. Paul Levy, a physicist, found the Haar function superior to the Fourier function. Littlewood, Paley and Stein computed the energy of a function f(x) as:

$$Energy = \frac{1}{2}\int_{0}^{2\pi}\left|f(x)\right|^{2} dx$$

Their work provided David Marr with an effective algorithm for numerical image processing using wavelets in the 1980s.

2.1.3 *1960-1980*

Between 1960 and 1980 mathematicians studied the simplest elements of a function space called atoms, with the goal of finding atoms for a common function and finding the "assembly rules" that allow the reconstruction of all the elements of the function space using these atoms. In 1980 Grossman and Morlet broadly defined wavelets in context of quantum physics.

2.1.4 *Post 1980*

In 1985, Stephane Mallat gave wavelets an additional jump-start through his work in digital signal processing. He discovered some relationships between quadrature mirror filter, pyramid algorithms and orthonormal wavelet bases. Inspired Y. Meyer constructed the first non-trivial wavelets. Unlike the Haar wavelets the Meyer wavelets are continuously differentiable; however they do not have compact support. A couple of years later, Ingrid Daubechies used Mallat's work to construct a set of wavelet orthonormal basis functions that are perhaps the most elegant and have become the corner stone of wavelet applications today.

2.2 WHY USE WAVELETS?

It is well known from Fourier theory that a signal can be expressed as the sum of a, possibly infinite, series of sines and cosines. This sum is also referred to as a Fourier expansion. The big disadvantage of a Fourier expansion however is that it has **only frequency resolution and no time resolution**. This means that although we might be able to determine all the frequencies present in a signal, we do not know when they are present. To overcome this problem in the past decades several solutions have been developed which are more or less able to represent a signal in the time and frequency domain at the same time.

The idea behind these time-frequency joint representations is to cut the signal of interest into several parts and then analyze the parts separately. The problem here is that cutting the signal corresponds to a convolution between the signal and the cutting window. The underlying principle of the phenomena just described is due to ***Heisenberg's uncertainty principle***, which, in signal processing terms, states that it is impossible to know the exact frequency and the exact time of occurrence of this frequency in a signal.

2.3 SHORT COMINGS OF EXISTING TRANSFORMS

A comparison of the existing transforms with the wavelet is given.

2.3.1 *Fourier Analysis*

Signal analysts already have at their disposal an impressive arsenal of tools. Perhaps the most well known of these is *Fourier analysis*, which breaks down a signal into constituent sinusoids of different frequencies. Another way to think of Fourier analysis is as a mathematical technique for *transforming* our view of the signal from time-based to frequency-based.

FIG 2.1 Fourier Transform Analysis

For many signals, Fourier analysis is extremely useful because the signal's frequency content is of great importance. So why do we need other techniques, like wavelet analysis?

Fourier analysis has a serious drawback. In transforming to the frequency domain, time information is lost. When looking at a Fourier transform of a signal, it is impossible to tell *when* a particular event took place. If the signal properties do not change much over time — that is, if it is what is called a *stationary* signal — this drawback isn't very important. However, most interesting signals contain numerous nonstationary or transitory characteristics: drift, trends, abrupt changes, and beginnings and ends of events. These characteristics are often the most important part of the signal, and Fourier analysis is not suited to detecting them.

2.3.2 *Short-Time Fourier Analysis*

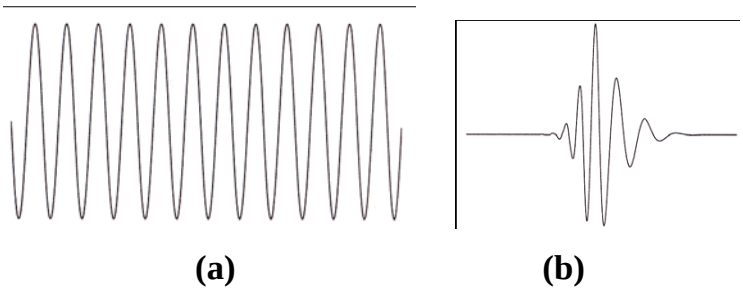
In an effort to correct this deficiency, Dennis Gabor (1946) adapted the Fourier transform to analyze only a small section of the signal at a time — a technique called *windowing* the signal. Gabor's adaptation, called the *Short-Time Fourier Transform* (STFT), maps a signal into a two-dimensional function of time and frequency.

FIG. 2.2 Short-Time Fourier Analysis

The STFT represents a sort of compromise between the time- and frequency-based views of a signal. It provides some information about both when and at what frequencies a signal event occurs. However, you can only obtain this information with limited precision, and that precision is determined by the size of the window. While the STFT compromise between time and frequency information can be useful, the drawback is that once you choose a particular size for the time window, that window is the same for all frequencies. Many signals require a more flexible approach one where we can vary the window size to determine more accurately either time or frequency.

The Wavelet Transform provides a time-frequency representation of the signal. It was developed to overcome the short coming of the Short Time Fourier Transform (STFT), which can also be used to analyze non-stationary signals. While STFT gives a constant resolution at all frequencies, the Wavelet Transform uses multi-resolution technique by which different frequencies are analyzed with different resolutions.

A wave is an oscillating function of time or space and is periodic. In contrast, wavelets are localized waves. They have their energy concentrated in time or space and are suited to analysis of transient signals. While Fourier Transform and STFT use waves to analyze signals, the Wavelet Transform uses wavelets of finite energy. The wavelet analysis is done similar to the STFT analysis.



(a) **(b)**

FIG 2.3 Demonstrations of (a) a Wave and (b) a Wavelet

The signal to be analyzed is multiplied with a wavelet function just as it is multiplied with a window function in STFT, and then the transform is computed for each segment generated. However, unlike STFT, in Wavelet Transform, the width of

the wavelet function changes with each spectral component. The Wavelet Transform, at high frequencies, gives good time resolution and poor frequency resolution, while at low frequencies; the Wavelet Transform gives good frequency resolution and poor time resolution.

2.3.3 *The Continuous Wavelet Transform and the Wavelet Series*

The Continuous Wavelet Transform (CWT) is provided by equation 2.1, where x(t) is the signal to be analyzed. ψ(t) is the mother wavelet or the basis function. All the wavelet functions used in the transformation are derived from the mother wavelet through translation (shifting) and scaling (dilation or compression).

$$X_{WT}(\tau, s) = \frac{1}{\sqrt{|s|}} \int x(t).\psi^* \left(\frac{t-\tau}{s} \right) dt \tag{2.1}$$

The mother wavelet used to generate all the basis functions is designed based on some desired characteristics associated with that function. The translation parameter τ relates to the location of the wavelet function as it is shifted through the signal. Thus, it corresponds to the time information in the Wavelet Transform. The scale parameter *s* is defined as |1/frequency| and corresponds to frequency information. Scaling either dilates (expands) or compresses a signal. Large scales (low frequencies) dilate the signal and provide detailed information hidden in the signal, while small scales (high frequencies) compress the signal and provide global information about the signal. Notice that the Wavelet Transform merely performs the convolution operation of the signal and the basis function. The above analysis becomes very useful as in most practical applications, high frequencies (low scales) do not last for a long duration, but instead, appear as short bursts, while low frequencies (high scales) usually last for entire duration of the signal.

The Wavelet Series is obtained by discretizing CWT. This aids in computation of CWT using computers and is obtained by sampling the time-scale plane. The sampling rate can be changed accordingly with scale change without violating the Nyquist criterion. Nyquist criterion states that, the minimum sampling rate that allows

reconstruction of the original signal is 2ω radians, where ω is the highest frequency in the signal. Therefore, as the scale goes higher (lower frequencies), the sampling rate can be decreased thus reducing the number of computations.

2.4 THE DISCRETE WAVELET TRANSFORM

The Wavelet Series is just a sampled version of CWT and its computation may consume significant amount of time and resources, depending on the resolution required. The Discrete Wavelet Transform (DWT), which is based on sub-band coding, is found to yield a fast computation of Wavelet Transform. It is easy to implement and reduces the computation time and resources required.

The foundations of DWT go back to 1976 when techniques to decompose discrete time signals were devised similar work was done in speech signal coding which was named as sub-band coding. In 1983, a technique similar to sub-band coding was developed which was named pyramidal coding. Later many improvements were made to these coding schemes which resulted in efficient multi-resolution analysis schemes.

In CWT, the signals are analyzed using a set of basis functions which relate to each other by simple scaling and translation. In the case of DWT, a time-scale representation of the digital signal is obtained using digital filtering techniques. The signal to be analyzed is passed through filters with different cutoff frequencies at different scales.

2.5 DWT AND FILTER BANKS

2.5.1 *Multi-Resolution Analysis using Filter Banks*

Filters are one of the most widely used signal processing functions. Wavelets can be realized by iteration of filters with rescaling. The resolution of the signal,

which is a measure of the amount of detail information in the signal, is determined by the filtering operations, and the scale is determined by upsampling and downsampling (subsampling) operations.

The DWT is computed by successive lowpass and highpass filtering of the discrete time-domain signal as shown in figure 2.4. This is called the Mallat algorithm or Mallat-tree decomposition. Its significance is in the manner it connects the continuous-time multi resolution to discrete-time filters. In the figure, the signal is denoted by the sequence x[n], where n is an integer. The low pass filter is denoted by G_0 while the high pass filter is denoted by H_0. At each level, the high pass filter produces detail information; d[n], while the low pass filter associated with scaling function produces coarse approximations, a[n].

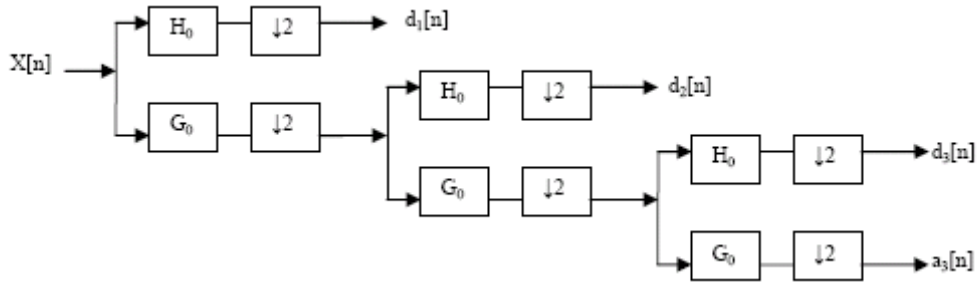


FIG 2.4: Three-level wavelet decomposition tree.

At each decomposition level, the half band filters produce signals spanning only half the frequency band. This doubles the frequency resolution as the uncertainty in frequency is reduced by half. In accordance with Nyquist's rule if the original signal has a highest frequency of ω, which requires a sampling frequency of 2ω radians, then it now has a highest frequency of ω/2 radians. It can now be sampled at a frequency of ω radians thus discarding half the samples with no loss of information. This decimation by 2 halves the time resolution as the entire signal is now represented by only half the number of samples. Thus, while the half band low pass filtering removes half of the frequencies and thus halves the resolution, the decimation by 2 doubles the scale.

With this approach, the time resolution becomes arbitrarily good at high frequencies, while the frequency resolution becomes arbitrarily good at low frequencies. The filtering and decimation process is continued until the desired level is reached. The maximum number of levels depends on the length of the signal. The DWT of the original signal is then obtained by concatenating all the coefficients, a[n] and d[n], starting from the last level of decomposition.

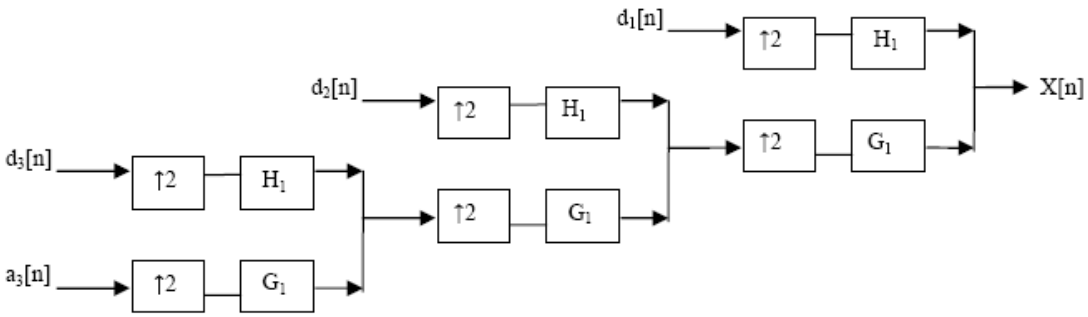


FIG 2.5: Three-level wavelet reconstruction tree.

Figure 2.5 shows the reconstruction of the original signal from the wavelet coefficients. Basically, the reconstruction is the reverse process of decomposition. The approximation and detail coefficients at every level are upsampled by two, passed through the low pass and high pass synthesis filters and then added. This process is continued through the same number of levels as in the decomposition process to obtain the original signal. The Mallat algorithm works equally well if the analysis filters, G_0 and H_0, are exchanged with the synthesis filters, G_1 and H_1.

2.5.2 *Conditions for Perfect Reconstruction*

In most Wavelet Transform applications, it is required that the original signal be synthesized from the wavelet coefficients. To achieve perfect reconstruction the analysis and synthesis filters have to satisfy certain conditions. Let $G_0(z)$ and $G_1(z)$ be the low pass analysis and synthesis filters, respectively and $H_0(z)$ and $H_1(z)$ the high pass analysis and synthesis filters respectively. Then the filters have to satisfy the following two conditions as given in:

$$G_0(-z).G_1(z) + H_0(-z).H_1(z) = 0 \tag{2.2}$$

$$G_0(z).G_1(z) + H_0(z).H_1(z) = 2z^{-d} \tag{2.3}$$

The first condition implies that the reconstruction is aliasing-free and the second condition implies that the amplitude distortion has amplitude of one. It can be observed that the perfect reconstruction condition does not change if we switch the analysis and synthesis filters.

There are a number of filters which satisfy these conditions. But not all of them give accurate Wavelet Transforms, especially when the filter coefficients are quantized. The accuracy of the Wavelet Transform can be determined after reconstruction by calculating the Signal to Noise Ratio (SNR) of the signal. Some applications like pattern recognition do not need reconstruction, and in such applications, the above conditions need not apply.

2.5.3 *Classification of wavelets*

We can classify wavelets into two classes: (a) orthogonal and (b) biorthogonal. Based on the application, either of them can be used.

(a)*Features of orthogonal wavelet filter banks*

The coefficients of orthogonal filters are real numbers. The filters are of the same length and are not symmetric. The low pass filter, G_0 and the high pass filter, H_0 are related to each other by

$$H_0(z) = z^{-N} G_0(z^{-1}) \tag{2.4}$$

The two filters are alternated flip of each other. The alternating flip automatically gives double-shift orthogonality between the lowpass and highpass filters, i.e., the scalar product of the filters, for a shift by two is zero. i.e., ΣG[k] H [k-

2l] = 0, where k,lЄZ. Filters that satisfy equation 2.4 are known as Conjugate Mirror Filters (CMF). Perfect reconstruction is possible with alternating flip.

Also, for perfect reconstruction, the synthesis filters are identical to the analysis filters except for a time reversal. Orthogonal filters offer a high number of vanishing moments. This property is useful in many signal and image processing applications. They have regular structure which leads to easy implementation and scalable architecture.

(b)*Features of biorthogonal wavelet filter banks*

In the case of the biorthogonal wavelet filters, the low pass and the high pass filters do not have the same length. The low pass filter is always symmetric, while the high pass filter could be either symmetric or anti-symmetric. The coefficients of the filters are either real numbers or integers.

For perfect reconstruction, biorthogonal filter bank has all odd length or all even length filters. The two analysis filters can be symmetric with odd length or one symmetric and the other antisymmetric with even length. Also, the two sets of analysis and synthesis filters must be dual. The linear phase biorthogonal filters are the most popular filters for data compression applications.

2.6 CLASSIFICATION OF WAVELETS

There are a number of basis functions that can be used as the mother wavelet for Wavelet Transformation. Since the mother wavelet produces all wavelet functions used in the transformation through translation and scaling, it determines the characteristics of the resulting Wavelet Transform. Therefore, the details of the particular application should be taken into account and the appropriate mother wavelet should be chosen in order to use the Wavelet Transform effectively.

Figure 2.6 illustrates some of the commonly used wavelet functions. Haar wavelet is one of the oldest and simplest wavelet. Therefore, any discussion of wavelets starts with the Haar wavelet. Daubechies wavelets are the most popular

wavelets. They represent the foundations of wavelet signal processing and are used in numerous applications. These are also called Maxflat wavelets as their frequency responses have maximum flatness at frequencies 0 and π. This is a very desirable

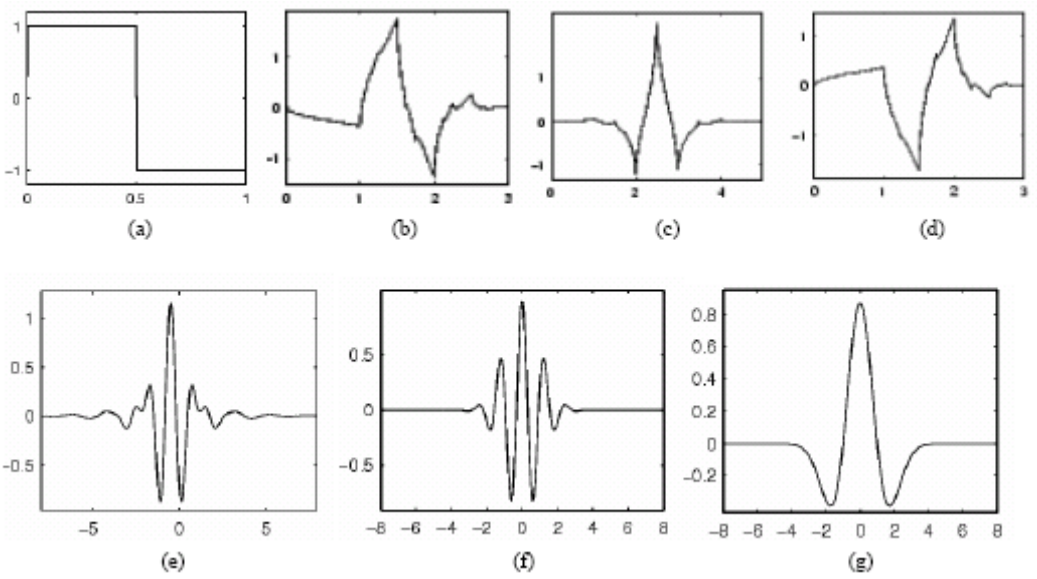


FIG 2.6: Wavelet families (a) Haar (b) Daubechies4 (c) Coiflet1 (d) Symlet2 (e) Meyer (f) Morlet (g) Mexican Hat.

property in some applications. The Haar, Daubechies, Symlets and Coiflets are compactly supported orthogonal wavelets. These wavelets along with Meyer wavelets are capable of perfect reconstruction. The Meyer, Morlet and Mexican Hat wavelets are symmetric in shape. The wavelets are chosen based on their shape and their ability to analyze the signal in a particular application.

2.6.1 *Haar Wavelet*

It is one of the oldest wavelet functions, which was designed way back in 1930. It is defined as:

The Haar Scaling Function:

Let $\phi : R \rightarrow R$ be defined by

$$\phi(t) = \begin{cases} 1 & t \in [0,1) \\ 0 & t \notin [0,1) \end{cases} \tag{2.5}$$

And ϕji : R→R as

$$\phi_i^j(t) = \sqrt{2^j}\,\phi(2^j t - i),\ j = 0,1,\ldots \ \&\ i = 0,1,\ldots,2^j - 1 \tag{2.6}$$

Then the Haar wavelet is defined as:

ψ : R→R defined as:

$$\psi(t) = \begin{cases} 1 & t \in [0, 1/2) \\ -1 & t \in [1/2, 1) \\ 0 & t \notin [0, 1) \end{cases} \tag{2.7}$$

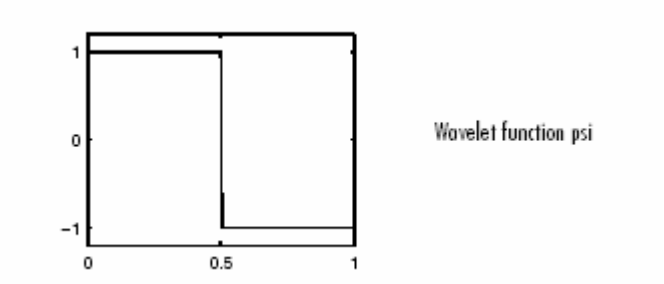


FIG 2.7 - Haar Wavelet Function

2.6.2 *Morlet Wavelet*

In practice the Morlet wavelet is defined as the product of a complex exponential wave and a Gaussian envelope.

The ψ is the wavelet value at non-dimensional time η, and ωo is the wave number. This is the basic wavelet function, but we now need some way to change the overall size as well as slide the entire wavelet along in time. We thus define the "scaled wavelets".

The s is the "dilation" parameter used to change the scale, and n is the translation parameter used to slide in time. The factor of s-1/2 is a normalization to keep the total energy of the scaled wavelet constant.

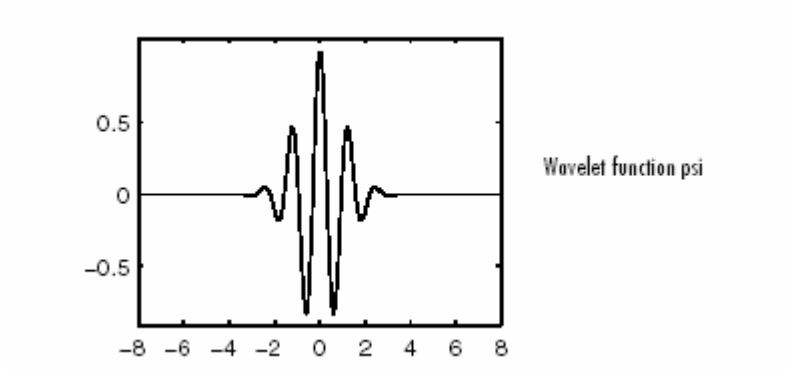


FIG 2.8 - Morlet Wavelet Function

2.6.3 *Daubechies Wavelet*

Ingrid Daubechies, one of the brightest stars in the world of wavelet research, invented what are called compactly supported orthonormal wavelets — thus making discrete wavelet analysis practicable.

The names of the Daubechies family wavelets are written dbN, where N is the order, and db the "surname" of the wavelet. The db1 wavelet, as mentioned above, is the same as Haar wavelet. Here are the wavelet functions psi of the next nine members of the family:

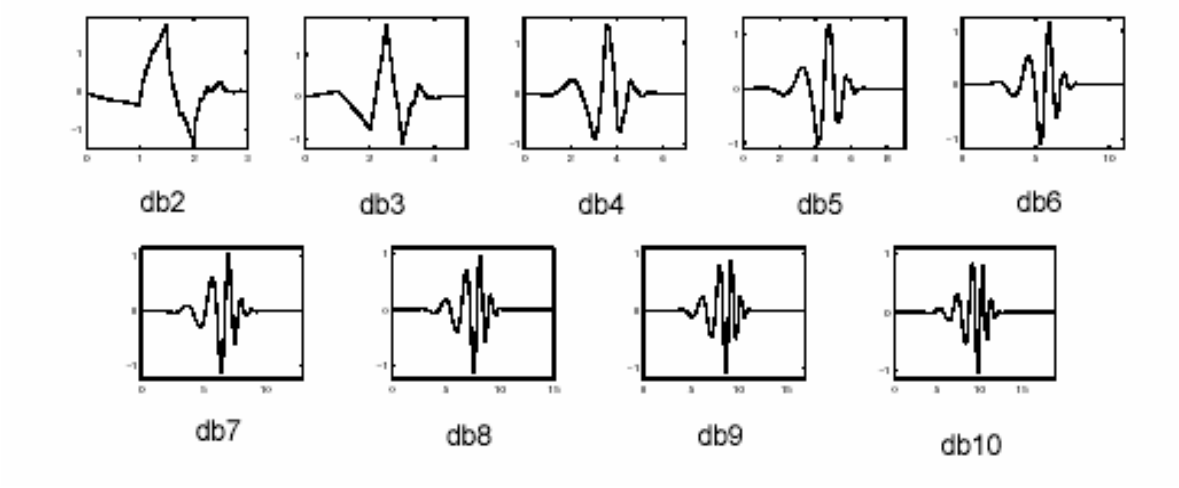


FIG 2.9 - Daubechies Wavelet Functions

2.6.4 Mexican Hat Wavelet

This wavelet has no scaling function and is derived from a function that is proportional to the second derivative function of the Gaussian probability density function.

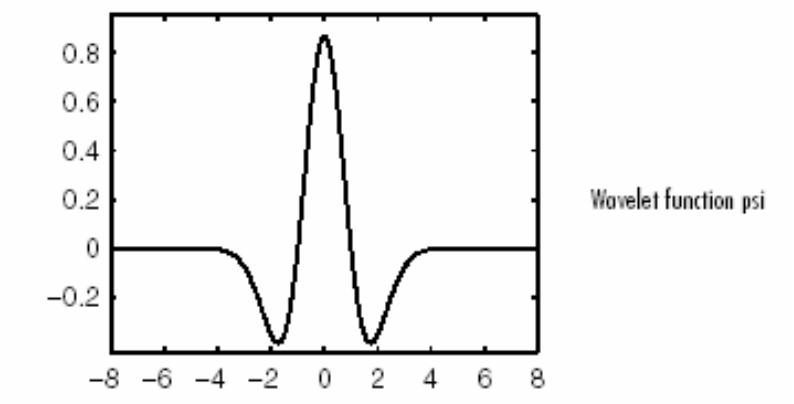


FIG 2.10 - Mexican Hat Functions

2.6.5 *Advantages of wavelets*

- Wavelets provide an efficient decomposition of signals prior to compression.
- Wavelets are adjustable and adaptable. Because there is not just one wavelet, they can be designed to fit individual applications. They are ideal for adaptive systems that adjust themselves to suit the signal.
- They are computationally inexpensive, perhaps one of the few really useful linear transform with a complexity that is O (N) as compared to Fourier transform, which is N log (N).
- Wavelet coding schemes at higher compression avoid blocking artifacts.
- Coefficient vector truncation is near optimal for data compression. The size of the wavelet expansion coefficients drops off rapidly for a large class of signals. This property is called being an unconditional basis and it is why wavelets are so effective in signal and image compression, denoising and detection.

Applications of wavelets include compressing images, such as x-rays and fingerprint collection, and de-noising data, found in many areas including geology, meteorology, astronomy and economics.

2.7 DWT DIFFERENT REALIZATIONS ALTERNATIVES

2.7.1 Theoretical Background

This section discusses different realization alternatives for the DWT, with a special focus in the lifting scheme-based realization that was chosen for our implementation. Historically, the wavelet transform has gained widespread acceptance in fields of signal processing and image coding. In the wavelet transform, dilations and translations of a mother wavelet are used to perform a spatial/frequency analysis of the input signal. Different realizations alternatives for the DWT algorithm can be found in the literature, figure 9 shows three different realizations alternatives. These are namely, the Mallat filtebank realization, lattice structure, and the lifting scheme-based realization.

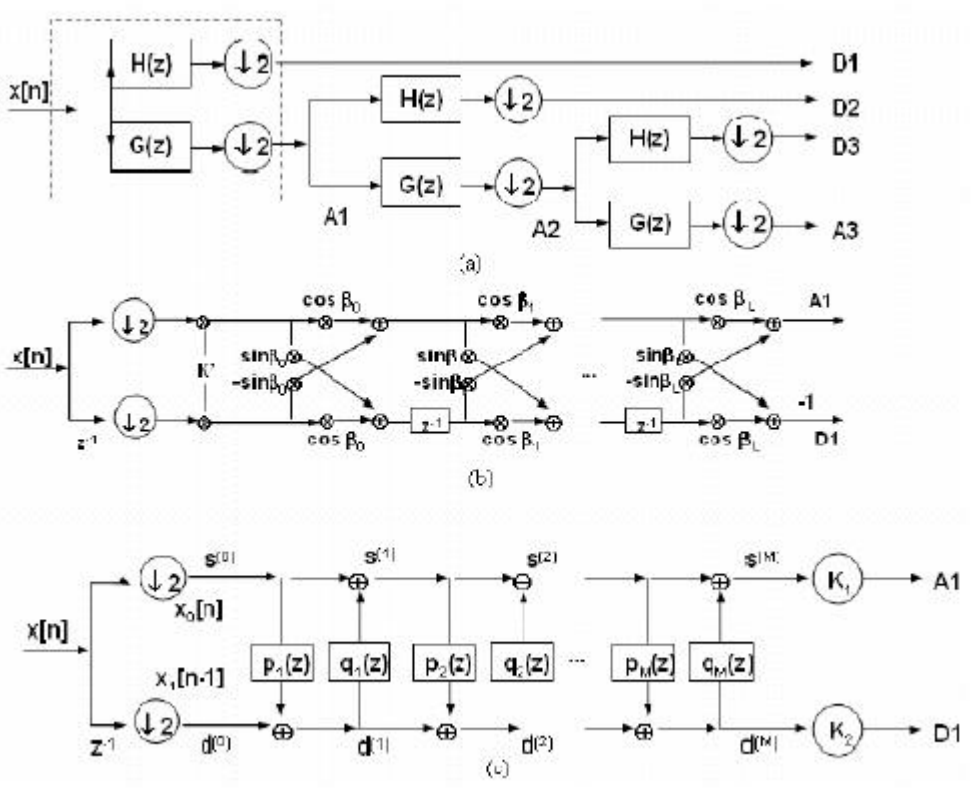


FIG 2.11: Different Realizations Alternatives for DWT. (a) Mallat FilterBank, (b) Lattice Structure, and (c)Lifting scheme.

- Mallat Filter bank realization

The DWT has been traditionally implemented by means of the Mallat filter bank scheme. The algorithm includes two main steps: signal decimating and filtering with a pair of Quadrature Mirror Filters (QMFs). Figure 2.12 shows a one stage wavelet filter bank analysis (left side) and synthesis (right side). The filter bank can be realized using FIR filters. The process consists of performing a series of dot products between the two filter masks and the signal.

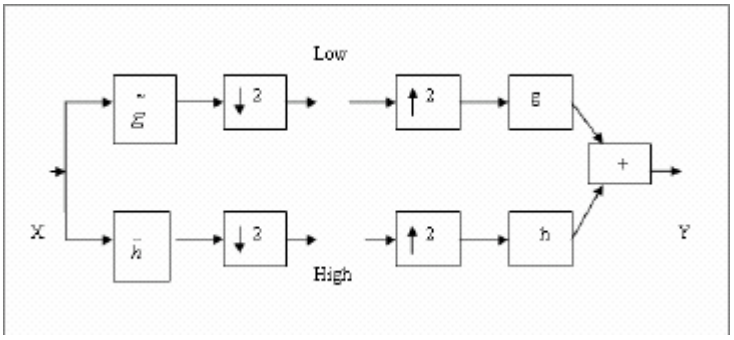


FIG 2.12. A one stage wavelet filter bank analysis (left side) and synthesis (right side)

- Lifting Scheme-based realization

Sweldens has proposed an alternative framework, called Lifting Scheme (LS), to compute the DWT. The lifting scheme is a computationally efficient way of implementing DWT. It has the advantage of reduced complexity, compared to the conventional convolution based scheme, as illustrated in Table 2. Moreover, all the operations within the lifting steps can be performed in parallel, hence the possibility of a fast implementation.

Table 2. Complexity of lifting scheme vs. convolutional wavelet transform for different wavelet filters

Filter	MUls,Shifts		Additions	
	Convolutional	Lifting	Convolutional	Lifting
(5,3)	4	2	6	4
C(13,7)	8	4	14	8
(9,7)	9	5	14	8

Wavelet Transform from Filter Bank to Lifting Scheme Implementation:

In our design for the DWT module, we have chosen the lifting scheme approach for the realization of the DWT. Thus, next we will show some details about the mathematical properties of this scheme.

Starting from the Malat filter bank realization, shown in figure 2.12, the conditions for perfect reconstruction are given by as:

$$h(z)\widetilde{h}(z^{-1}) + g(z)\widetilde{g}(z^{-1}) = 2 \tag{2.8}$$

$$h(z)\widetilde{h}(-z^{-1}) + g(z)\widetilde{g}(-z^{-1}) = 0 \tag{2.9}$$

Polyphase representation:

Figure 2.12 shows that the wavelet decomposition is performed by filtering then downsampling. Clearly, it would be more efficient to do the downsampling before the filtering.

Employing the modified FIR filter in the wavelet transform, we can end up with a new matrix representation:

$$\begin{pmatrix} \lambda(z) \\ \gamma(z) \end{pmatrix} = \widetilde{P}(z) \begin{pmatrix} \chi_e(z) \\ z^{-1}\chi_o(z) \end{pmatrix} \tag{2.10}$$

Where $\widetilde{p}\,(z)$ is the polyphase matrix:

(2.11)

$$\widetilde{P}(z) = \begin{pmatrix} \widetilde{h}_0(z) & \widetilde{h}_0(z) \\ \widetilde{g}_0(z) & \widetilde{g}_0(z) \end{pmatrix}$$

The perfect reconstruction condition is:

$$\widetilde{P}(z^{-1})P(z) = I \tag{2.12}$$

*Lifting Representation***:**

Ingrid Daubchies have proved that, if we start with a complementary filter pair (h,g), the polyphase matrix P(z) can always be factored into a number of lifting steps, which are easier to implement:

$$P(z) = \begin{pmatrix} K_1 & 0 \\ 0 & K_2 \end{pmatrix} \prod_{i=1}^{m} \left\{ \begin{pmatrix} 1 & s_i(z) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ t_i(z) & 1 \end{pmatrix} \right\} \tag{2.13}$$

Lifting Scheme representation for the 5/3 Le Gall wavelet filters:

In our design for the hardware lifting scheme-based DWT module, we have chosen the (5/3) Le Gall wavelet filters. The JPEG2000 standard committee has recommended using the Le Gall wavelet filters for the integer mode operation. The following shows how the lifting equations for the Le Gall 5/3 filter along with the block diagram can be derived.

Starting with the 5/3 wavelet filters transform equations:

$$\tilde{h}(z) = -\frac{1}{8}z^{-2} + \frac{1}{4}z^{-1} + \frac{3}{4} + \frac{1}{4}z - \frac{1}{8}z^{2}$$
$$\tilde{g}(z) = \frac{1}{4}z^{-2} - \frac{1}{2}z^{-1} + \frac{1}{4} \tag{2.14}$$

The factorized polyphase matrix for the le Gall filter is:

$$\tilde{P}(z) = \begin{pmatrix} 1 & 0 \\ 0 & \frac{-1}{2} \end{pmatrix} \begin{pmatrix} 1 & \frac{1}{4} + \frac{1}{4}z \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \frac{-1}{2}z^{-1} - \frac{1}{2} & 1 \end{pmatrix} \tag{2.15}$$

The lifting equations can then be directly obtained:

$$y_{2i+1} = -0.5\left(x_{2i} + x_{2i+2} + x_{2i+1}\right) \tag{2.16}$$

$$y_{2i} = 0.25\left(y_{2i+1} + y_{2i+3}\right) + x_{2i} \text{ where } 0 \le i < N/2 \tag{2.17}$$

Using the above equations, one can sketch how the lifting scheme based realization for the 5/3 leGall filters would look like. Figure 2.13 represents the basic building block of the 1D-DWT using the Le Gall wavelet filters. Since all coefficients are multiplies of 2, all multiplications and divisions can be replaced by shifting operations. From the figures 2.13, we can see an interesting property of the lifting

representation. Every time we apply a predict or update lifting step we add something to one stream. All the samples in the stream are replaced by new samples and at any time we need only the current streams to update sample values.

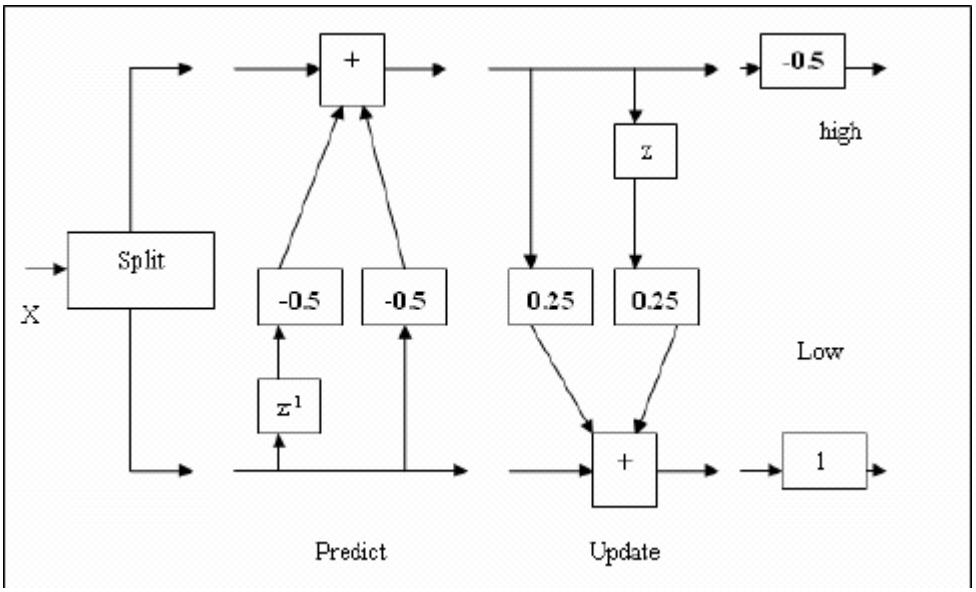


FIG 2.13 The implementation of the wavelet transform of Le Gall filters

In lifting literature, the two basic steps are the predict step, and the update step. The idea behind this terminology is that lifting of the high-pass subband with the low- pass subband can be seen as prediction of the odd samples from the even samples.

- Integer Wavelet Transform (IWT)

The lifting scheme-based realization allows having what is called integer-to integer transform, or simply Integer Wavelet transform. The transform coefficients of the IWT are exactly represented by finite precision numbers, thus allowing for truly lossless encoding. This would help us reduce number of bits for the sample storage and to use simpler filtering units. Integer wavelet transforms is achieved by rounding off the output of $s_i(z)$ and $t_i(z)$ filters, in figure 2.13, before addition or subtraction. This would lead to a very beneficial class of transforms, in terms of computational complexity and memory requirements. Yet, they are highly dependant on the choice of the factorization of the polyphase matrix. Equation xx shows the lifting steps for the 5/3 Le Gall Integer Wavelet Transform. The rational coefficients allow the transform to be

reversible, i.e., invertible in finite precision analysis, hence giving a chance for performing lossless compression.

$$y_{2i+1} = -0.5\left(x_{2i} + x_{2i+2} + x_{2i+1}\right)$$
$$y_{2i} = 0.25\left(y_{2i+1} + y_{2i+3}\right) + x_{2i} \text{ where } 0 \le i < N/2$$

2.7.2 *Proposed Design*

In this project, a scalable lifting scheme based Integer wavelet transform unit was implemented. 5/3 Le Gall wavelet filters were employed. Design Acceleration has been achieved by techniques like exploiting the parallel nature of sub units, pipelining, and re-usability of data. There were two major design stages towards the final proposed architecture.

- *Parallel Operation*

First, starting from the sequential realization of the algorithm, we tried to further exploit the parallel nature of the algorithm through parallel operation of independent units. Second, the designed was further optimized by introducing pipeline stages. The following highlights the major features of the proposed architecture.

Table 1 shows the number of operations required by the 5/3 filter for each predict or update stage. In our design, we introduce parallel processing of independent sub units. Input samples are accessed through a four-sample wide window, allowing two concurrent predict operations and two concurrent update operations, as shown in figure 2.14.

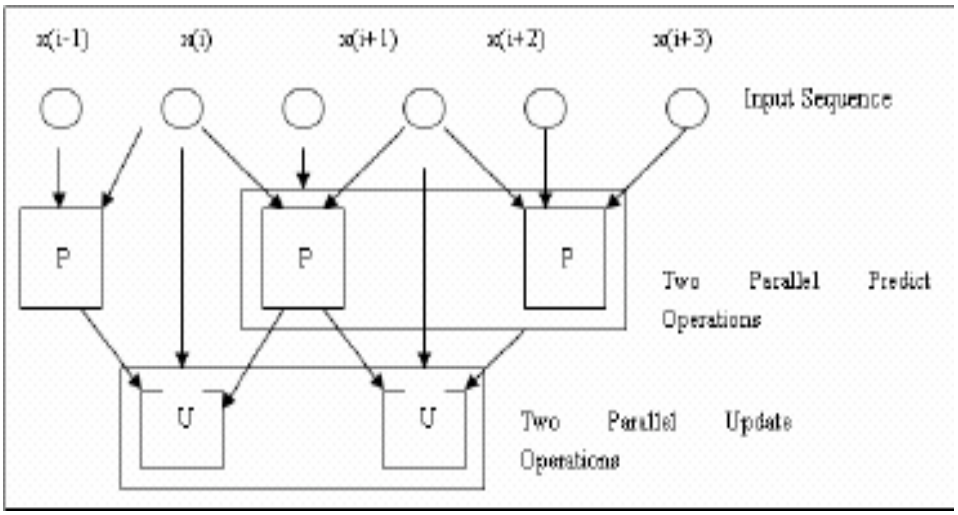


FIG 2.14 Parallel Processing of input data samples

Registers were used for temporary storage, and reusability of temporary data. Figures 2.15 and 2.16 show the block diagram for the update and predict modules respectively.

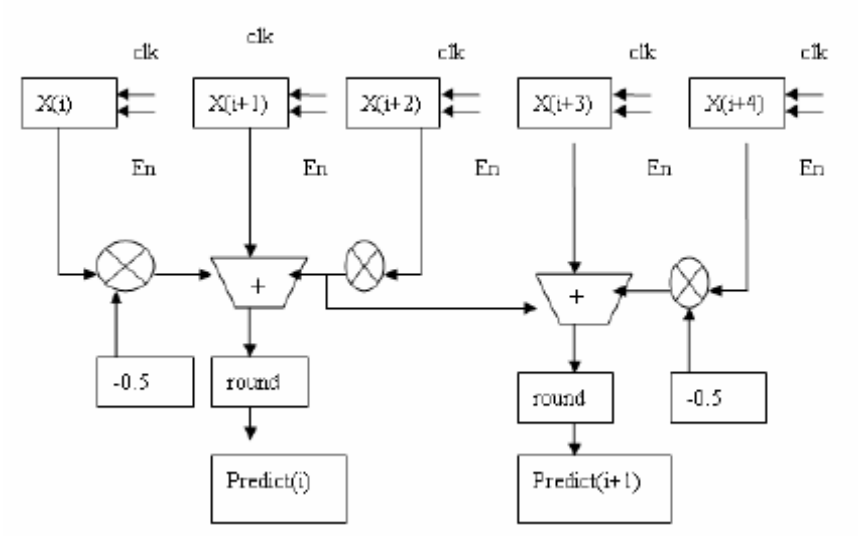


FIG 2.15 Predict Filter module with two concurrent values being calculated.

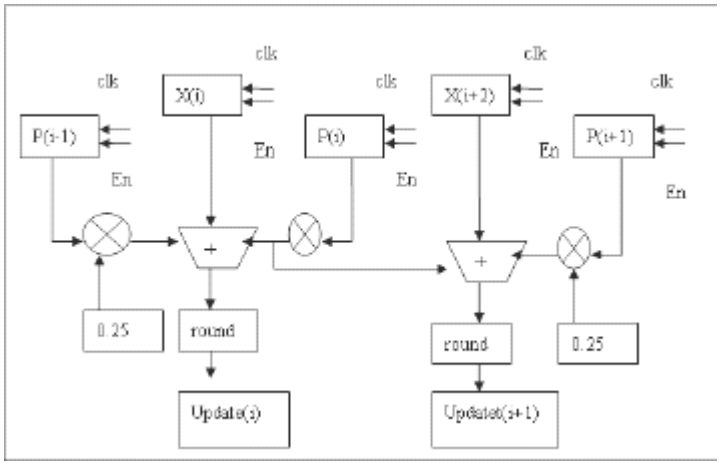


FIG 2.16 Update Filter Module with two concurrent values being calculated.

- *Pipelining the Optimized DWT*

By pipelining the DWT, the highpass and lowpass coefficients can be computed during the time the memory is being accessed, rather than having to wait until the reads are complete, computing the coefficients, and then writing them. Due to the nature of the lifting scheme DWT, the low pass coefficients depend not only on the high pass coefficient in the stage immediately before them, but also the pixel values of higher stages. As a result, it is necessary to ensure that the pipeline data is valid at all times.

By examining the coefficient equations, we can determine which values are necessary to compute the coefficients:

$$L_N = f\left(H_{N-1}, P_{2N-1}, H_N\right) \tag{2.19}$$

$$H_N = f\left(P_{2N-1}, H_{2N,} P_{2N+1}\right) \tag{2.20}$$

The smallest window we can look at is two pixels. In this design, two pixels are read in and two coefficients are computed. When outputting L_N and H_N, L_{N+1} and H_{N+1} are being computed. This requires P_{2N-1}, P_{2N} and P_{2N+1} to be available. P_{2N+1} and P_{2N+2} are read in, and then fed directly into another set of registers resulting in P2N and P_{2N-1}, since two pixels are read in every clock cycle. Note that the design ensures that L_N and H_N become valid during the same clock cycle.

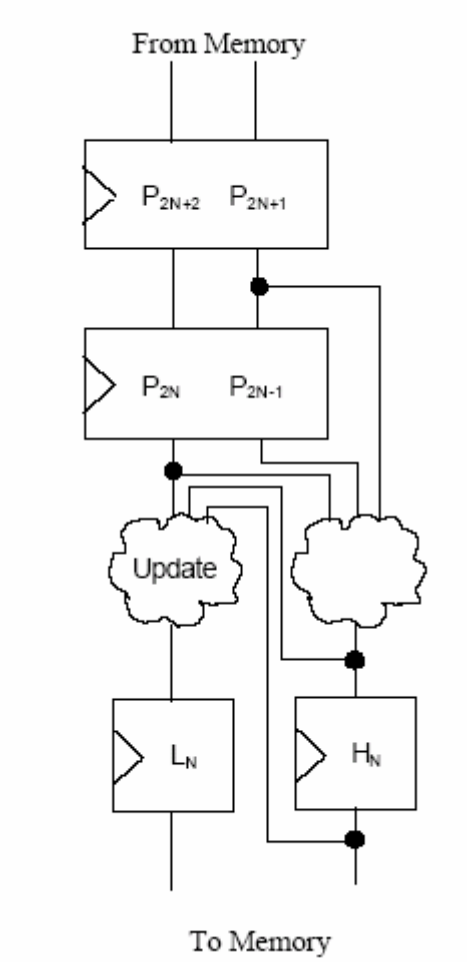


FIG 2.17 Pipelined DWT with a window of 2 pixels

For a window of four pixels, the concept is the same, only more work is done in parallel.

$$L_N = f(H_{N-1}, P_{2N-1}, H_N)$$
$$L_{N+1} = f(H_{N-1}, P_{2N-1}, H_N)$$
$$H_N = g(P_{2N-1}, P_{2N}, P_{2N+1})$$
$$H_{N+1} = g(P_{2N+1}, P_{2N+2}, P_{2N+3})$$

When L_N, L_{N+1}, H_{N+1} and H_{N+2} are available on the outputs, L_{N+2}, L_{N+1}, H_{N+3}, and H_{N+4} are being computed. This requires P_{2N+3} and P_{2N+5} through P_{2N+9} to be available. Pixels P_{2N+6} through P_{2N+9} are read in each clock cycle, and two pixels are delayed to give the other two required pixels.

By delaying H_{N+2} one stage before outputting it, we can have H_N, L_N, H_{N+1} and L_{N+1} become valid all on the same clock cycle, which makes the control system easier to design.

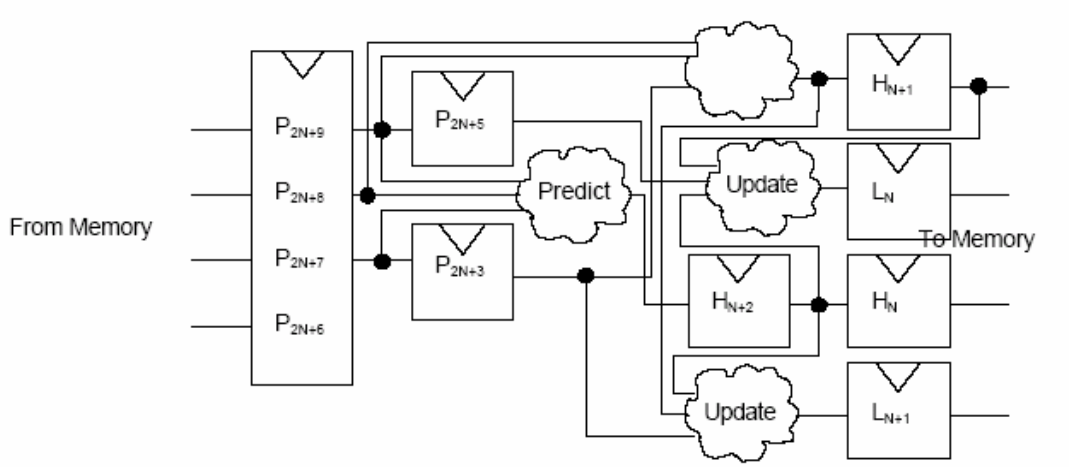


FIG 2.18 Pipelined DWT with a window of 4 pixels

Notice that the critical path from P_{2N+6} or P_{2N+7} to L_{N+1} is through two stages of combinational logic. If the clock frequency needs to be increased, the pipeline can be redesigned with another stage to break up the path at the expense of an increased latency. This is left as an exercise to the reader.

Filling the Pipeline:

The wavelet transform specifies that L_1 should be calculated using H_1, P_1, and H_1. This is equivalent to inputting the beginning of the pixel stream as P_3, P_2, P_1, P_2, P_3, rather than beginning with P_1. This will end up generating one extra high pass coefficient, but the control system simply ignores it.

Emptying the Pipeline:

Similarly to filling the pipeline, when reading in the last pixels, the 2nd last and 3rd last pixels are repeated in reverse order after the last pixel. IE, the end of the pixel stream will look like P_{N-3}, P_{N-2}, P_{N-1}, P_N, P_{N-1}, P_{N-2}. In some cases this will have the

effect of calculating extra coefficients. The control system simply keeps track of how many values it has written, and stops after it has written all the expected coefficients. Any extra values are ignored. For an even number of input pixels, there will be an equal number of high pass and low pass coefficients. For an odd number, the algorithm results in one more low pass coefficient than the number of high pass coefficients.

Further Improvements (better memory organization)

If dual-port RAM is available or there is more than one bank of RAM, reads and writes can be carried out simultaneously, approximately halving the number of clock cycles needed for execution.

Memory access is the major bottleneck to the design. The pipeline requires 4 clock cycles for to read in new values, but only one clock cycle to compute the new value. By intelligent storage of the image in memory, even faster speeds may be obtained. Simply storing more consecutive pixels per word will not work, as the wavelet transform must be able to read the memory in either rows or columns.

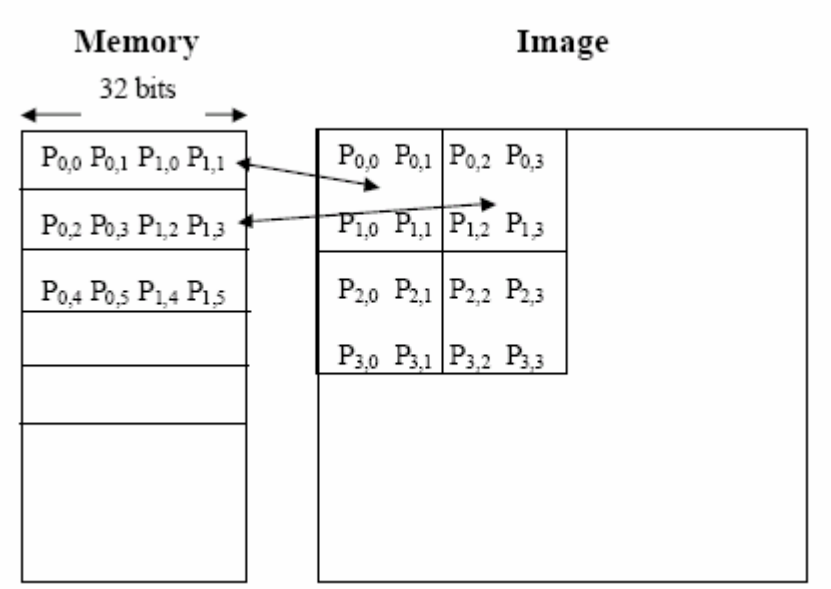


FIG 2.19 Improved memory organization

However, if the pixels are stored in square blocks, not only is it possible to always read multiple pixels per read, but it allowing the system to work on multiple rows or columns in parallel. For example, the first 32-bit word in memory could represent the first two 8-bit pixels of the first row, and the first two pixels of the second row.

When reading rows, the first two words of memory are read in, and the first two bytes of each word are used to calculate the first two high-pass coefficients and the first two low-pass coefficients of the first row. Meanwhile, the first four coefficients of the second row are being processed in parallel.

Another complication comes from the fact that when writing the coefficients out, they are not written to consecutive locations. However, since we are calculating two rows in parallel, we will end up with one full block of high-pass coefficients, and one full block of low-pass coefficients. This method imposes the limitation that the image height and width must be a power of two.

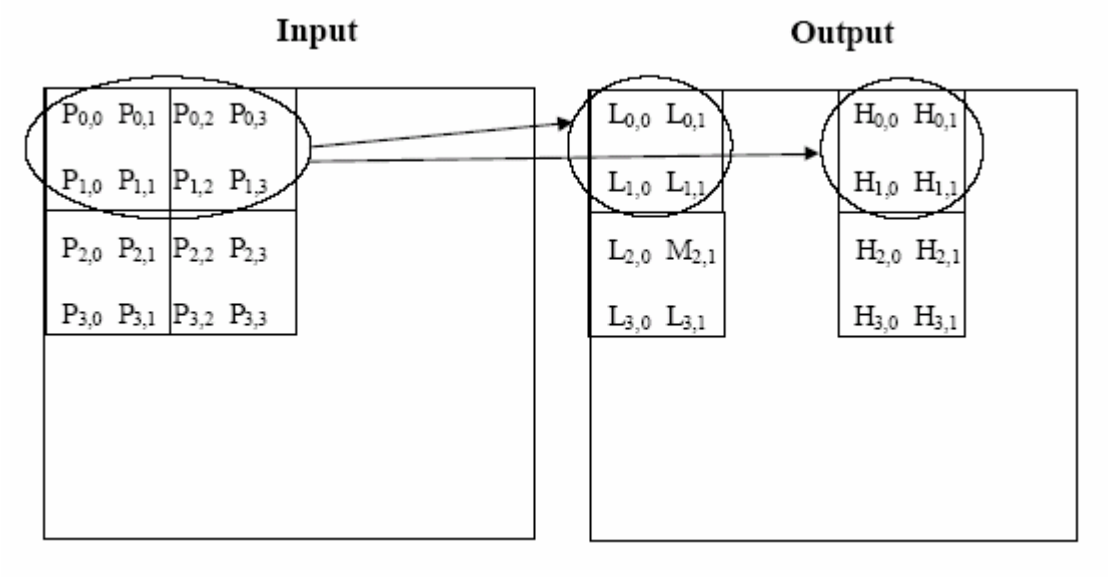


FIG 2.20 Writing coefficients with improved memory organization

Although there was not enough time to implement the design, it is conceptually faster than the existing design, and should allow for a speedup of 4x by requiring half the number of cycles to read in four pixels, and by allowing two columns or rows to be processed in parallel. The pipeline for the DWT calculation

would stay the same, but it would be instantiated twice. Only the control system and memory controller would need to be changed in order to process blocks. Furthermore, in the same manner 128-bit words could be used to access 16 pixels at a time, and provide an even greater speedup.

The 2D-DWT was obtained by performing the designed 1D-DWT on both the rows and columns. Figure 2.21 shows how would the 2D-DWT module operate on an input image and what would be the final output.

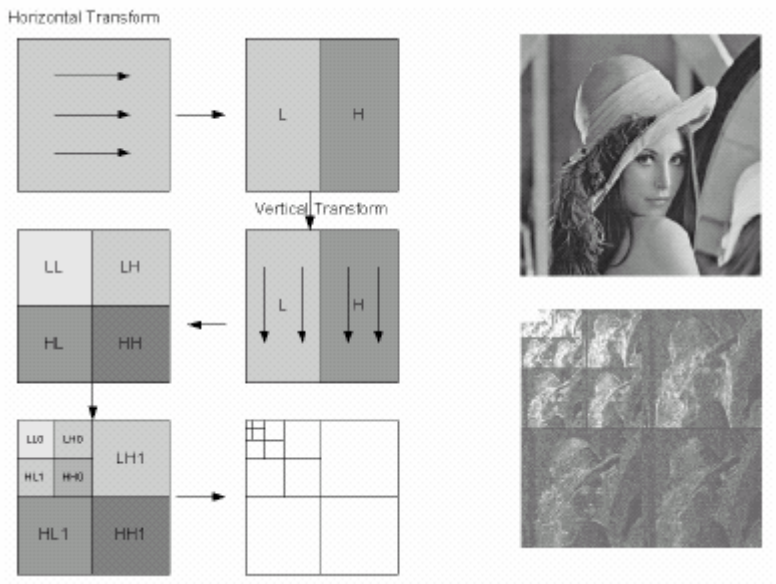


FIG 2.21 2-D Recursive Pyramid DWT

2.7.3 *Precision Issues*

As for the conventional DWT realizations, partial transform results need to be represented with a high precision. This raises storage and complexity problems. On the other side, Integer wavelet transforms (IWT) results in integer intermediate results. Thus, it is possible to use integer arithmetic without encountering rounding problems. Consequently, based upon precision studies from the literature for the 5/3 filter IWT a fixed precision of 8 bits per pixel were selected. The error introduced by this precision has been proved, through comparing software and hardware implementations, to be within an accepted range to most applications.

CHAPTER 3

JPEG2000 COMPRESSION

3.1 BACKGROUND OF JPEG 2000

Currently, the most common form of image compression is known as JPEG. This standard was developed by the Joint Photographic Expert Group (hence the name JPEG) in the late 1980's, However, in 1997 the JPEG committee decided that the needs and requirements of imagery applications in today's world point to the need for a new standard. For example, neither Microsoft Internet Explorer nor Netscape Navigator, two popular Internet browsers, currently support JPEG 2000 file formats. As JPEG 2000 becomes more widely known and its superior features recognized, it will take its place as the main player in the image compression industry.

Some of the features that this standard possesses are the following:

- **Superior low bit-rate performance:** This standard offers performance superior to the current standards at low bit-rates (e.g. below 0.25 bpp for highly detailed grey-scale images).
- **Continuous-tone and bi-level compression:** It is desired to have a coding standard that is capable of compressing both continuous-tone and bi-level images.
- **Lossless and lossy compression:** It is desired to provide lossless compression naturally in the course of progressive decoding.
- **Progressive transmission by pixel accuracy and resolution:** Progressive transmission that allows pictures to be reconstructed with increasing pixel accuracy or spatial resolution.
- **ROI capability** or Region of Interest. The use of wavelets allows one to be able to select a certain area of an image to view at a high quality, while leaving the rest of the image at a lower quality.
- **Robustness to bit-errors:** It is desirable to consider robustness to bit-errors while designing the codestream.

3.2 COMPARISON BETWEEN JPEG AND JPEG 2000

JPEG 2000 offers numerous advantages over the old JPEG standard, and several of these advantages will be discussed. One main advantage in that JPEG 2000 offers both lossy and lossless compression in the same file stream, while JPEG usually only utilizes lossy compression. JPEG does have a lossless compression engine, but it is separate from the lossy engine, and is not used very often. Thus, when high quality is a concern, JPEG 2000 proves to be a much better compression tool. Because of the way the compression engine works, JPEG 2000 promises a higher quality final image, even when using lossy compression.

Since the JPEG 2000 format includes much richer content than existing JPEG files, the bottom line effect is the ability to deliver much smaller files that still contain the same level of detail as larger original JPEG files. The JPEG 2000 files can also handle up to 256 channels of information as compared to the current JPEG standard.

A second advantage of JPEG 2000 over JPEG is that JPEG 2000 is able to offer higher compression ratios for lossy compression. For lossy compression, data has shown that JPEG 2000 can typically compress images from 20% to 200% more than JPEG.

Another advantage of JPEG 2000 is its ability to display images at different resolutions and sizes from the same image file. With JPEG, an image file was only able to be displayed a single way, with a certain resolution. Because JPEG 2000 is based on wavelets, the wavelet stream can be only partially decompressed if the user only wants a low-resolution image, while the full resolution image can be views if this is desired.

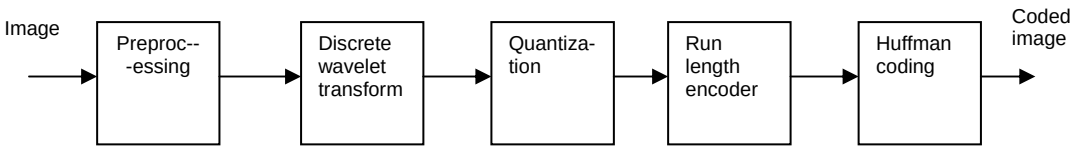


FIG 3.1 Block diagram for JPEG compression

3.2.1 *Reading The Original Image*

The original image, which is unprocessed, is stripped off with its header information. Only the necessary information is retained while reading the image header. These are: num_cols, num_rows: number of columns, rows in the image vector. The image size (img_size) is then calculated which is equal to the product of number of columns and rows. The information after the header inside the image is the pixel intensities. These values are then read in an array of integers (int_data*)*. The image data is ready for further processing.

3.2.2 *Wavelet Transforms Routine*

The first step, the wavelet transform routine process, is a modified version of the biorthogonal wavelet. The basic concept behind wavelet transform is to hierarchically decompose an input signal into a series of successively lower resolution reference signals and their associated detail signals. At each level, the reference signal and their associated detail signal contain the information needed to reconstruct the reference signal at the next higher resolution level.

The wavelet transform routine employs a lifting scheme to simplify the wavelet transform implementation. Therefore, it only requires integer adds and shifts. The computation of the wavelet filter is performed according to the following equations.

$$D_0 = D_0 + D_0 - S_0 - S_0 \qquad (3\text{-}1)$$

$$S_0 = S_0 + (2*D_0/8) \qquad (3\text{-}2)$$

$$D_i = D_i + D_i - S_i - S_{i+1} \qquad (3\text{-}3)$$

$$S_i = S_i + ((D_{i-1}+D_i)/8) \qquad (3\text{-}4)$$

In the above equations, $\boldsymbol{D_i}$ and $\boldsymbol{S_i}$ are odd and even pixels taken from one row or column, respectively. In image compression, one row or column of an image is regarded as a signal.

Calculation of the wavelet transform requires the pixels to be taken from one row or column at a time. In Equations (1) – (4). D_i should be calculated first before processing S_i. Therefore, the odd pixel should be processed first, then the even pixel due to the data dependency. There are a total of three levels based on the 3-level decomposition wavelet transform algorithm. In each level, the rows are processed first and then the columns. Each level's signal length (amount of each row/column pixels) is half of the previous level. Equations (1) – (4) are grouped into a function called forward-wavelet. Figure 3.2 illustrate the three levels of wavelet transform implementation.

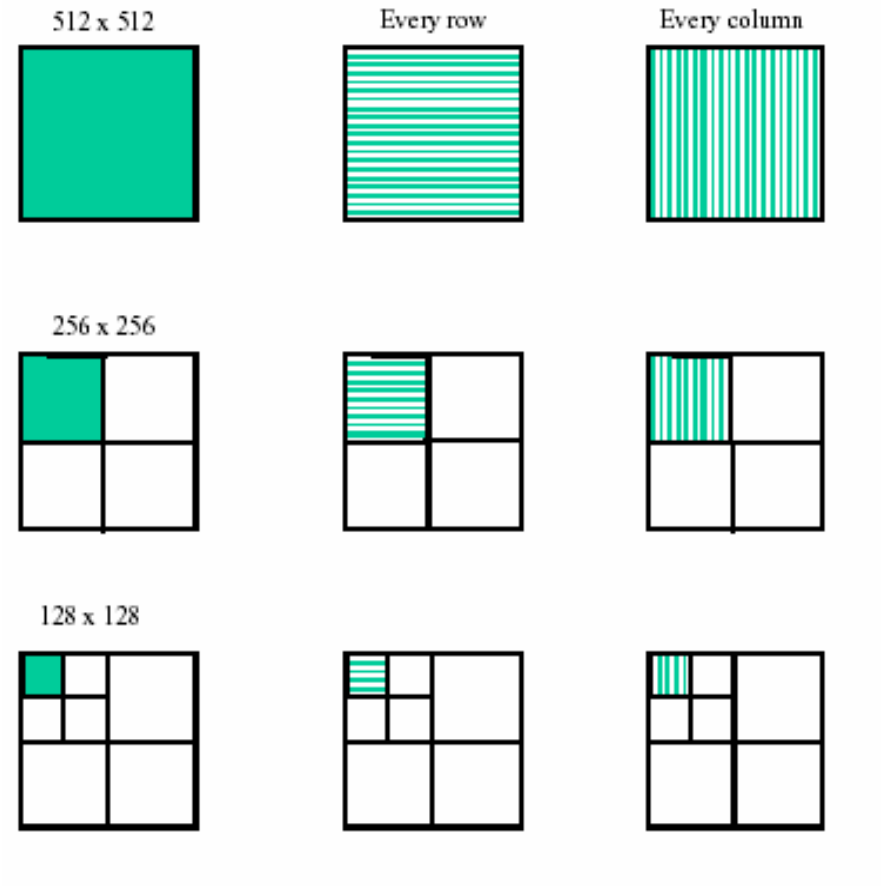


FIG. 3.2 Wavelet transform implementation.

3.2.3 *Quantization Routine*

After obtaining the three levels of the wavelet transform, the quantization routine follows. During the quantization routine, the image is divided into 10 blocks; the first four will be 64 x 64 pixels (4096 pixels), then three will be 128 x 128 (16384 pixels), and the remaining three of 256 x 256 pixels (65536 pixels). Every block executes the same quantization process. Figure 3.3 illustrates this as a block diagram. Before processing each block, some parameters should be prepared. First is the blockthresh, which is a threshold value below which all the intensities in the transformed image are given the zero mark. An array is used to hold these 10 block blockthreshes *values*:

Blockthresh [10] = {0, 39, 27, 104, 79, 51, 191, 99999, 99999, 99999}

For example: Since we want to retain the values in lower numbered blocks, Block 1‘s *blockthresh* is 0, Block 10’s *blockthresh* is 99999. The next values that need to be calculated are the sixteen thresholds for each block, thresh1~thresh10. The formula to calculate these values is given in equation.

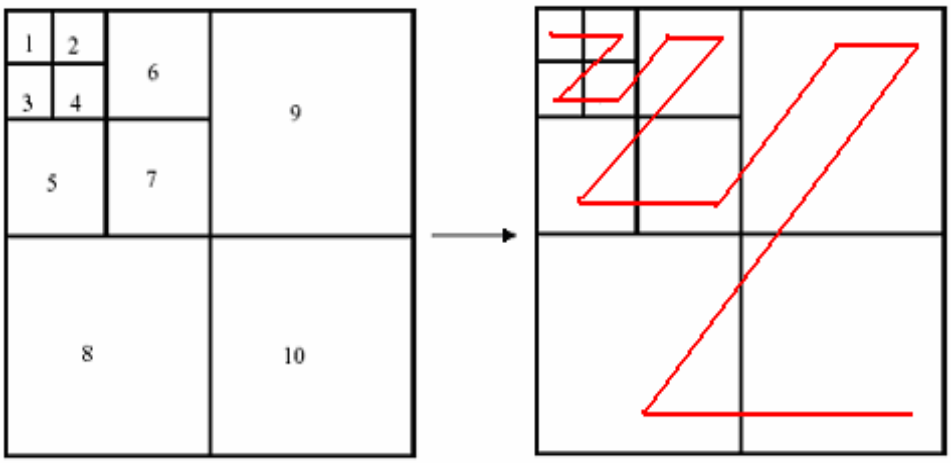


FIG. 3.3 Quantization and Run-Length encoding block diagram

3.2.4 *Run-Length Encoding Routine (RLE)*

Run-length encoding is the next routine following the quantization process. The basic concept is to code each contiguous group of 0's encountered in a scan of a row by its length and to establish a convention for determining the value of the run. This is shown as an example in Figure 3.4

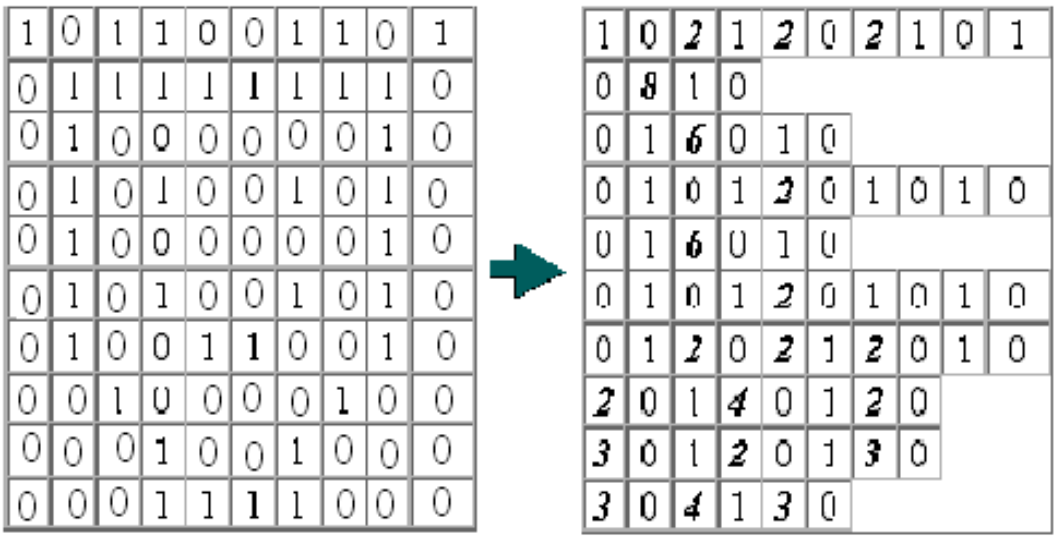


FIG. 3.4 An example of Run Length Encoding

The purpose of this step is to compress the image size based on the pixel values from quantization, which are between integer values *0* to *16*. The image can be compressed as much as up to 10% of the original after the run length encoding. As in the quantization routine, the image is also divided into 10 blocks. Each block will run the same run-length encoding algorithm..

Because values in blocks 8, 9, and 10 after quantization are all equal to 16, these three blocks will not be processed.

3.2.5 *Entropy Coding Routine*

The last routine in the Image Wavelet Compression algorithm is entropy coding. This process is based on the calculation results from run-length encoding.

Using the Huffman encoding algorithm for the entropy coding, the resulting image file can be compressed as much as up to 2-3% of the original image size. In the algorithm, two 256 size integer arrays are used to hold the parameters for a fixed Huffman encoding tree. These parameters are expected to work reasonably well with most images.

The method starts by building a list of all the alphabet symbols in descending order of their probabilities. It then constructs a tree, with a symbol at every leaf, from the bottom up. This is done in steps, where at each step the two symbols with smallest probabilities are selected, added to the top of the partial tree, deleted from the list, and replaced with an auxiliary symbol representing both of them. When the list is reduced to just one auxiliary symbol (representing the entire alphabet), the tree is complete.

The tree is then traversed to determine the codes of the symbols. This is best illustrated by an example. Given five symbols with probabilities as shown in Fig 3.5a, they are paired in the following order:

1. $a4$ is combined with $a5$ and both are replaced by the combined symbol $a45$, whose probability is 0.2.

2. There are now four symbols left, $a1$, with probability 0.4, and $a2$, $a3$, and $a45$, with probabilities 0.2 each. We arbitrarily select $a3$ and $a45$, combine them and replace them with the auxiliary symbol $a345$, whose probability is 0.4.

3. Three symbols are now left, $a1$, $a2$, and $a345$, with probabilities 0.4, 0.2, and 0.4, respectively. We arbitrarily select $a2$ and $a345$, combine them and replace them with the auxiliary symbol $a2345$, whose probability is 0.6.

4. Finally, we combine the two remaining symbols, $a1$ and $a2345$, and replace them with $a12345$ with probability 1. The tree is now complete. It is shown in Figure 2.14a “lying on its side” with the root on the right and the five leaves on the left. To assign the codes, we arbitrarily assign a bit of 1 to the top edge and a bit of 0 to the bottom edge, of every pair of edges. This results in the codes 0, 10, 111, 1101, and 1100. The assignment of bits to the edges is arbitrary.

The average size of this code is 0.4 × 1 + 0.2 × 2 + 0.2 × 3 + 0.1 × 4 + 0.1 ×4 = 2.2bits/symbol, but even more importantly, the Huffman code is not unique. Some of the steps above were chosen arbitrarily, since there were more than two symbols with smallest probabilities. Fig 3.5b shows how the same five symbols can be combined differently to obtain a different Huffman code (11, 01, 00, 101, and 100). The average size of this code is 0.4 × 2 + 0.2 × 2 + 0.2 × 2 + 0.1 × 3 + 0.1 ×3 = 2.2 bits/symbol.

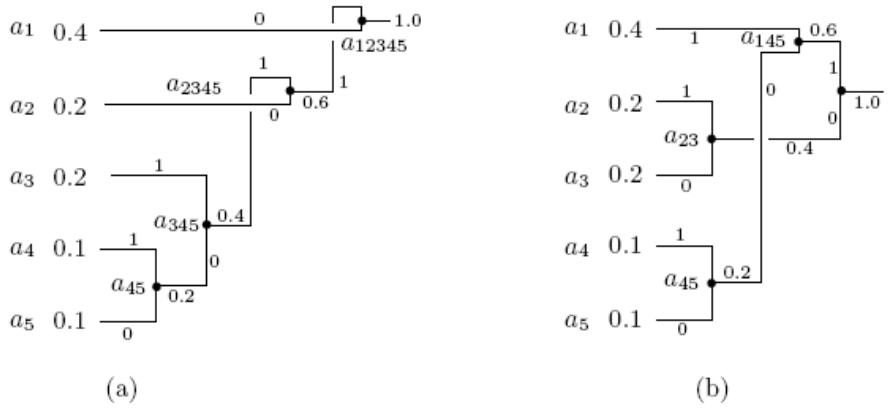


FIG 3.5 Huffman Codes.

CHAPTER 4

SOFTWARE IMPLEMENTATION

This chapter discusses the implementation in VHDL of the JPEG2000 encoder. Once written in VHDL, the encoder design can be synthesized and programmed into the FPGA.A number of background issues to the implementation are discussed first. The implementation process involved several steps. These included an examination of the JPEG2000 standard

4.1 BACKGROUND TO THE IMPLEMENTATION

The implementation of the arithmetic encoder was broadly influenced by a number of background issues. These included the decision to use the VHDL language, the role of the arithmetic encoder in JPEG2000, and the fundamental requirements of the VHDL implementation. These topics are briefly discussed below.

4.1.1 *Rationale for Using the VHDL Language*

The implementation of the JPEG2000 arithmetic encoder was written in the Very High Speed Integrated Circuit Hardware Description Language (VHDL). Following are the important reasons why this language was used.

VHDL is widely used for creating designs that will be programmed into FPGA devices. It is a general and versatile language in which to design a digital system. It is true that there are sections of the arithmetic encoder design produced that are tied to FPGA architecture. However, using VHDL allows as much of the design as possible to be portable to other synthesis tools, other FPGA cards or even other FPGA architectures.

Implementing the JPEG2000 encoder in VHDL therefore results in a much simpler design flow than if an alternate language or design entry technique had been used.

4.2 IMPLEMENTATION REQUIREMENTS

The primary requirement of the VHDL implementation was one of compliance. The code written was required to conform to the JPEG2000 standard for the arithmetic entropy encoding stage. Additionally, it was required that the VHDL design be written in synthesizable code.

The Quartus II v5.1 software package is used for implementing the JPEG2000 Encoder. The fixed-point formats are chosen to avoid truncation and round-off, given the quantized values of the coefficients.

First module of JPEG2000 implementation: The VHDL code presented here implements DWT using Le Gall 5/3 filter bank. The filtering process is "lifted" as described above. This is more efficient then conventional approach using convolution. Also, integer arithmetic is used; hence no need for precision with more then 8 bits is required. The integer arithmetic is possible when filtering coefficients are represented as fractions. In some cases, such as this, the least significant information (bit) cancels out during calculations so its presence can be neglected. Additionally, the coefficients of Le Gall filter bank filters in "lifted" filtering are multiples of 2, so division and multiplication are replaced by shifting.

The project is divided into two parts, first, containing a top level entity: DWT2D. DWT2D is used for direct transformation. The entities require external memory source,for original data and coefficients are stored. The entity Memory is used to simulate memory, where in actual implementation signals from this entity would be used to produce real memory control signals.

Entity DWT perform one-dimensional DWT of the signal, also stored in external memory. These two are based on 1D_SD and 1D_SR filtering process from the JPEG2000 standard.

In second stage, Wavelet coefficients from memory are read from the lower half of the embedded memory. The block (sub-band) minimum and maximum is also read from the memory. The packed bit stream output is written to the upper memory,

and the bit stream length is written to memory location 0. The control software, reads the embedded memory and generates the compressed image file. Before reading the wavelet coefficients, the maximum and minimum of coefficients in each sub-band are read from the lower memory. The coefficients are then read and processed for each sub-band, starting with the lowest frequency band. Memory read has a latency of 2 clock cycles. The two intermediate states, *Read and Write* can be used to write back the output, if output is available. Each memory read brings in two wavelet coefficients. Consider the worst case, where the two coefficients gets expanded to 18 bits each. There are two memory write cycles before the next read. Whenever a memory write is performed, the memory address register is incremented.

The output is written as a continuous stream, starting with the lowest sub-band. Thus the output is effectively in Mallat ordering and can be progressively transmitted and decoded.

At the end of the second stage, the upper memory contains the packed bit stream. The total count of the bit stream approximated to the nearest WORD is written to memory location 0. To reconstruct the data from the bit stream, the following information is needed.

The four quadrants of the final stage of wave-letting can be located at the first four 128*128 byte blocks. The three quadrants of the next stage can be located at at next three blocks sized at 256*256 bytes each. Each quadrant (sub-band) is quantized separately. The dynamic range of each of the quadrant should be known to reconstruct the original stream. The output file written has all the information needed to reconstruct the image.

The format of the output file generated is shown in figure 4.1

Number of Bytes	(4 bytes)
Block0 MIN/MAX	(8 bytes)
Block1 MIN/MAX	(8 bytes)
Block2 MIN/MAX	(8 bytes)
Block3 MIN/MAX	(8 bytes)
Block4 MIN/MAX	(8 bytes)
Block5 MIN/MAX	(8 bytes)
Block6 MIN/MAX	(8 bytes)
Block0	(4 bytes)
Block1	(4 bytes)
Block2	(4 bytes)
Block3	(4 bytes)
Block4	(4 bytes)
Block5	(4 bytes)
Block6	(4 bytes)
Bit Stream	(Variable)

FIG4.1: Output File Format

CHAPTER 5

RESULTS

This section discusses the experimental results obtained using the developed algorithm for the Image Codec. The results are presented according to their qualitative evaluation, the relative size of the compressed file formats and the intelligibility of the reconstructed image.

5.1 Bitrate(bpp) Vs. PSNR(db)

The characteristics of the three different frames are displayed in able 3. A software decoder available was used to reconstruct the encoded image in order to compare with the original. Noise figures from a software encoder are quoted. The PSNR and RMSE metrics are computed as per the equation given below. Percentage compression is the ratio of compressed image size to the original image size (512x512 pixels). Bits per pixel (bpp) is the ratio of image size in bits to number of pixels. The corresponding reconstructed and compressed images are also shown in figures 5.1, 5.2 and 5.3.

PSNR and RMSE Equations**:**

$$MSE = \frac{1}{512 \times 512} \sum_{x=1}^{512} \sum_{y=1}^{512} \left[p(x,y) - p^{1}(x,y) \right]^2$$

$$RMSE = \sqrt{MSE}$$

PSNR= 20log10 (255/RMSE)

Table 3: compression ratios and noise measurements

IMAGE	ORIGINAL SIZE	COMPRESSED SIZE	COMPRESSION RATIO	BPP	RMSE	PSNR
BARBARA	262141	28775	9.11	.878	7.274	30.894
GOLD HILL	262025	29708	8.82	.906	15.237	25.017
BRAIN	261209	30024	8.7	.916	8.027	30.038

(a)

(b)

(c)

FIG 5.1 (a) Original Barbara (b) Compressed Barbara (b) Reconstructed Barbara

(a)

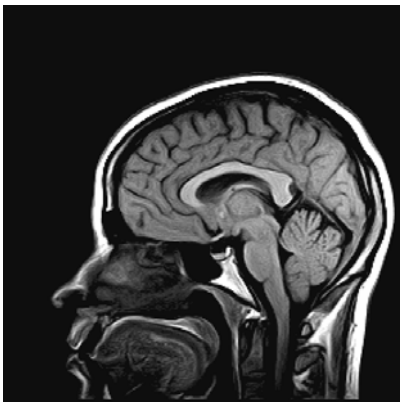
(b)

(c)

FIG 5.2 (a) Original Gold Hill (b) Compressed Gold Hill (c) Reconstructed Gold Hill

(a)

(b)



(c)

FIG 5.3 (a) Original Brain image (b) Compressed Brain image (c) Reconstructed Brain image

The quantized rational designs are smaller, faster, and require less energy than the quantized irrational designs. The primary reason is that the rational coefficients have narrower bit widths, which implies narrower adders that are faster and require less energy. (For example, the first coefficient, -1.5, requires only three bits.) In the lifting structure, intermediate signals grow in bit width after each lifting stage, with the amount of growth determined by the corresponding coefficient. In the rational designs, the bit width grows more slowly, with bit widths near the input of the system being particularly small.

CHAPTER 6

APPLICATIONS

The JPEG-2000 standard provides a set of features that are of importance to many high-end and emerging applications by taking advantage of new technologies.

6.1 IMAGE COMPRESSION

Wavelet Transforms are used to compress the pictures for storage in their data bank. The previously chosen Discrete Cosine Transform (DCT) did not perform well at high compression ratios. It produced severe blocking effects which made it impossible to follow the ridge lines in the fingerprints after reconstruction. This did not happen with Wavelet Transform due to its property of retaining the details present in the data.

In DWT, the most prominent information in the signal appears in high amplitudes and the less prominent information appears in very low amplitudes. Data compression can be achieved by discarding these low amplitudes. The wavelet transforms enables high compression ratios with good quality of reconstruction. At present, the application of wavelets for image compression is one the hottest areas of research. Recently, the Wavelet Transforms have been chosen for the JPEG 2000 compression standard.

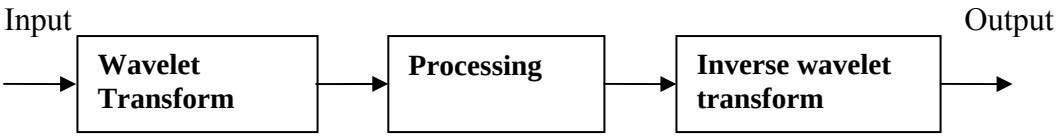


FIG 6.1*:* Steps followed in signal processing

Processing may involve compression, encoding, denoising etc. The processed signal is either stored or transmitted. For most compression applications, processing involves quantization and entropy coding to yield a compressed image. During this process, all the wavelet coefficients that are below a chosen threshold are discarded. These discarded coefficients are replaced with zeros during reconstruction at the other

end. To reconstruct the signal, the entropy coding is decoded, then quantized and then finally Inverse Wavelet Transformed.

6.2 SPEECH COMPRESSION

The idea behind speech compression using wavelets is primarily linked to the relative scarceness of the wavelet domain representation for the signal. Wavelets concentrate speech information (energy and perception) into a few neighboring coefficients. Therefore as a result of taking the wavelet transform of a signal, many coefficients will either be zero or have negligible magnitudes. Data compression is then achieved by treating small valued coefficients as insignificant data and discarding them. The process of compressing a speech signal using wavelets involves number of different stages.

Wavelets with more vanishing moments provide better reconstruction quality, as they introduce less distortion into the processed speech and concentrate more signal energy in a few neighboring coefficients. However the computational complexity of the DWT increases with the number of vanishing moments and hence for real time applications it is not practical to use wavelets with an arbitrarily high number of vanishing moments.

6.3 OPTICAL FREQUENCY DIVISION MULTIPLEXING (OFDM)

Orthogonal Frequency Division Multiplexing (OFDM) is a strong candidate for high-speed data transmission in wireless channels. OFDM is also a capable technology for the 4th generation of global wireless communications systems. In this modulation scheme transmission is carried out in parallel on the different frequencies. This technique is desirable for the transmission of the digital data through the multi path fading wireless channels. Since by the parallel transmission, the deleterious effect of fading is spread over many bits. Therefore, instead of a few adjacent bits completely destroyed by the fading, it is more likely that several bits only be slightly affected by the channel. In spite of these advantages OFDM has some challenges. For example it has a large Peak-to-Average Power Ratio (PAPR) which is a main limitation for this technology.

The reduction of PAPR of OFDM signal is done by using wavelets. Accordingly a proper wavelet for parallel data transmission with low PAPR is designed. The time shifted and scaled wavelets construct an orthonormal set. PAPR of this orthogonal wavelet based parallel transmission method is evaluated and is compared with the conventional OFDM system.

6.4 ELECTRO CARDIOGRAM (ECG)

High performance compression methodologies for the transmission of medical data and images play an important part in high speed, cost effective and efficient networked telemedicine services. However, the recent developments of commercial compression tools and technologies for multimedia, Internet, and other applications, are not paralleled with similar work on application-specific compression methodologies for telemedical systems and, in particular, for wireless telemedicine applications.

In recent years, wavelet-based compression techniques and tools have received significant attention, especially for different biomedical signal-processing applications. The main goal is to achieve sufficiently high compression ratios (CRs) without affecting the diagnostic characteristics of the ECG signal. The technique used to do so is the Wavelet High Order Statistics-based Coding (WHOSC) using the discrete wavelet transform concept. The advantage of using WHOSC is that it extends the scheme proposed for ECG data compression, with an enhanced coding scheme based on the combination of DWT and higher order statistics (HOS). Hence, the robustness of HOS is especially useful in noisy environments.

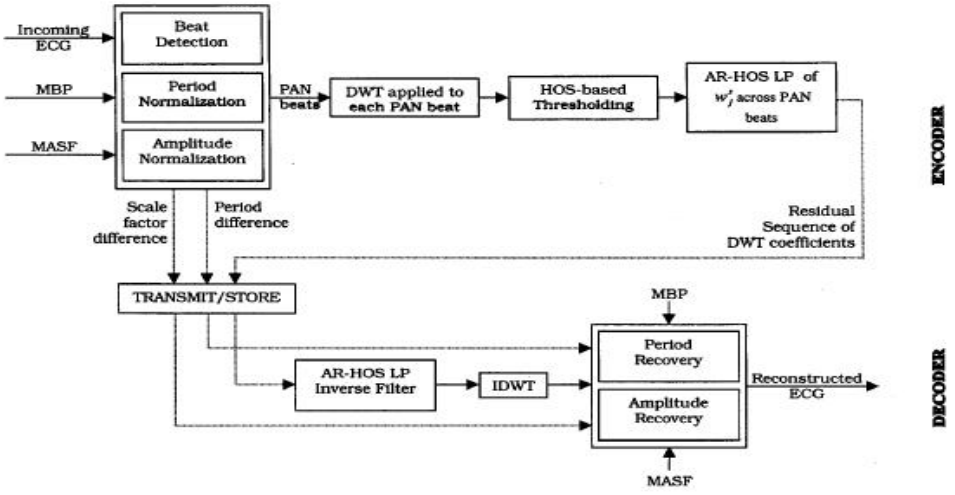


FIG 6.2 Block Diagram of WHOSC Algorithmic Approach

Since wavelets could efficiently represent the non stationarities and the time-localized components of an ECG (ST segment) the ECG signal could adequately be coded with a Wavelet based coding scheme. Thus, a DWT of the PAN beats is performed using Mallat's multiresolution analysis.

6.5 VISUAL FREQUENCY WEIGHTING

The human visual system plays an important role in the perceived image quality of compressed images. System designers and users should be able to take advantage of the current knowledge of visual perception, i.e. to utilize models of visual system's varying sensitivity to spatial frequencies, as measured in the contrast sensitivity function (CSF). Since the CSF weight per sub-band in the wavelet transform.

Two types of visual frequency weighting are supported by the JPEG 2000. The fixed Visual Weighting (FVW) and the Visual Progressive Coding (VPC). In FVW, only one set of CSF weights is chosen and applied in accordance with the viewing conditions. In VPC, different weights are used at the various stages of the

embedded coding. This is because during a progressive transmission stage, the image is viewed at various distances.

6.6 ERROR RESILIENCE

JPEG 2000 uses a variable length coder (arithmetic coder) to compress the quantized wavelet coefficients. Variable length coding is known to be prone to channel or transmission error. To improve the performance of transmitting compressed images over error prone channels, error resilient bit stream syntax and tools can be included in JPEG 2000. The error resilience tools deal with channel errors using the following approaches: data partitioning and resynchronization, error detection and concealment, and Quality of Service (QoS) transmission based on priority. Error resilience is achieved at the entropy coding level and at the packet level.

Entropy coding of the quantized coefficients is performed within code-blocks. Since encoding and decoding of the code-blocks are independent processes, bit error in the bit-stream of a code-block will be restricted within that code-block. To increase error resilience, termination of the arithmetic coder is allowed after every coding pass and the contexts may be reset after each coding pass, this allows the arithmetic decoder to continue the decoding process even if an error has occurred.

The “lazy coding” mode is also useful for error resilience. The relates to the optional arithmetic coding bypass, in which bits are used as raw bits in to the bit-stream without arithmetic coding. This prevents the error propagation types to which variable length coding is susceptible.

6.6 NEW FILE FORMAT WITH IPR CAPABILITIES

An optional file format for the JPEG 2000 compressed image data is defined in the standard. This format has got provisions for both image and metadata and specified mechanisms to indicate image properties, such as the tone scale or color space of the image, to recognize the existence of intellectual property right (IPR)

information in the file and to include metadata (as for example vendor specific information). Metadata give the opportunity to the reader to extract information about the image, to the reader to extract information about the image, without having to decode it thus allowing fast text based search in a database.

In summary, the file format contains the size of the image, the bit depth is not constant across all components, the color space of the image, the palette which maps a single component in index space to a multimedia –component image, the type and ordering of the components within the code-stream, the resolution of the image, the resolution at which the image should be displayed , the code-stream, the intellectual property information about the image , a tool by which vendors can add XML formatted information to the JP2 file, etc.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 CONCLUSIONS

In this dissertation, we introduced architectural design for hardware acceleration of the Discrete Wavelet Transform based on the Lifting scheme. The design can be used to enhance the performance of the multimedia tools as JPEG2000 by hardware acceleration. The unit is meant to be integrated in custom-computing platforms, as a reconfigurable functional unit.

After analyzing the transform algorithm, a design was introduced to calculate the wavelet coefficients efficiently. The design utilizes different techniques as pipelining, data reusability, in parallel operating units

Implementing the design:

VHDL hardware description language was used to describe the behavior of the design. Subsequently, the design was simulated in Modelsim for functional correctness.

Reconfigurable hardware is also suited for applications with rapidly changing requirements. In effect, the same piece of silicon can be reused.

7.2 FUTURE WORK AND SUGGESTIONS

The lessons learned from this work will help us enhance similar implementations in the future. Few of the improvements that we now foresee are listed below:

1. In order to obtain realistic timing data the VHDL code can implement in Xilinx Spartan FPGA series.

2. Build a corresponding wavelet transform decoder on the FPGA and demonstrate the adaptability of the encoder-decoder pair. The encoder would need to signal the decoder on which codec is being used.

3. Multiple levels of DWT computation present the problem of growing signal bit widths. Starting with 8-bit image data, the input to each successive DWT level will have wider bit widths, making it impractical to save all bits of precision in memory till the final decomposition level. Intermediate DWT coefficients will have to be truncated after every level, so that data read from and written to memory will have fixed bit widths. For the lifting implementation, where bit widths grow after every filter within a single lifting stage, intermediate signals may have to be truncated within a level.

4. Currently the software operates for grayscale images of size 512 X 512 in the PGM format only. A more realistic version could be implemented for images of all types and of any size incorporating boundary treatment by using the classical extension methods such as periodic extension.

APPENDIX A

A.1 Dwt2d.vhd

```
LIBRARY IEEE;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.MemManagerPkg.ALL;
PACKAGE DWT2DPkg IS
CONSTANT CST_CTRL2D_DATA_BUS_WIDTH: INTEGER:= 16;
CONSTANT CST_MAX_DECOMP_LEVELS: INTEGER:= 7;
CONSTANT CST_DWT2D_CTRL_SIG_BUS_WIDTH        : INTEGER := 2;
SUBTYPE TYPE_DWT2D_CTRL_SIGNAL IS
STD_LOGIC_VECTOR(CST_DWT2D_CTRL_SIG_BUS_WIDTH - 1 DOWNTO 0);
TYPE TYPE_DWT2D_CTRL_SIGNAL_ENUM IS (
TCTRL_DWT2D_WIDTH,
TCTRL_DWT2D_HEIGHT,
TCTRL_DWT2D_TMP_DATA_OFFSET,
TCTRL_DWT2D_LEVELS);
FUNCTION Dwt2dCtrlVector(x : IN TYPE_DWT2D_CTRL_SIGNAL_ENUM) RETURN
TYPE_DWT2D_CTRL_SIGNAL;
COMPONENT dwt2d
PORT (
clk                 : IN STD_LOGIC;
mem_data            : INOUT TYPE_DATA;
mem_enable          : OUT STD_LOGIC;
mem_rw              : OUT STD_LOGIC;
mem_adr                     : OUT TYPE_ADR;
ctrl_sig   : IN TYPE_DWT2D_CTRL_SIGNAL;
ctrl_data: IN STD_LOGIC_VECTOR(CST_CTRL2D_DATA_BUS_WIDTH - 1 DOWNTO 0);
ready               : OUT STD_LOGIC;
reset               : IN STD_LOGIC
);
END COMPONENT;
END DWT2DPkg;
PACKAGE BODY DWT2DPkg IS
FUNCTION Dwt2dCtrlVector(x : IN TYPE_DWT2D_CTRL_SIGNAL_ENUM) RETURN
TYPE_DWT2D_CTRL_SIGNAL IS
BEGIN
RETURN
TYPE_DWT2D_CTRL_SIGNAL(TO_UNSIGNED(TYPE_DWT2D_CTRL_SIGNAL_ENUM'POS(x
), CST_DWT2D_CTRL_SIG_BUS_WIDTH));
END;
END DWT2DPkg;
LIBRARY IEEE;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.MemManagerPkg.ALL;
USE WORK.DWTPkg.ALL;
USE WORK.DWT2DPkg.ALL;
ENTITY dwt2d IS
PORT (
clk: IN STD_LOGIC;
mem_data: INOUT TYPE_DATA := (others => 'Z');
mem_enable: OUT STD_LOGIC := 'Z';
mem_rw: OUT STD_LOGIC := 'Z';
mem_adr: OUT TYPE_ADR := (others => 'Z');
ctrl_sig            : IN TYPE_DWT2D_CTRL_SIGNAL;
ctrl_data           : IN STD_LOGIC_VECTOR(CST_CTRL2D_DATA_BUS_WIDTH - 1 DOWNTO
0);
ready                       : OUT STD_LOGIC := '1';
```

```
reset                           : IN STD_LOGIC
        );
END dwt2d;
ARCHITECTURE Main OF dwt2d IS
TYPE TYPE_DWT2D_STATE IS (
TS2D_READY,
TS2D_INIT_HORIZ,
TS2D_START_HORIZ,
TS2D_CONTINUE_HORIZ,
TS2D_INIT_VERT,
TS2D_START_VERT,
TS2D_CONTINUE_VERT,
TS2D_NEXT_LEVEL,
TS2D_TERMINATE
);
SIGNAL state : TYPE_DWT2D_STATE := TS2D_READY;
SIGNA  dwt_ctrl_sig: TYPE_DWT_CTRL_SIGNAL;
SIGNAL dwt_ctrl_data: STD_LOGIC_VECTOR(CST_CTRL_DATA_BUS_WIDTH - 1 DOWNTO
0);
SIGNAL dwt_ready: STD_LOGIC := '1';
SIGNAL width, height : UNSIGNED(CST_CTRL2D_DATA_BUS_WIDTH - 1 DOWNTO 0);
SIGNAL tmpDataOffset : UNSIGNED(CST_CTRL2D_DATA_BUS_WIDTH - 1 DOWNTO 0);
SIGNAL levels : INTEGER RANGE CST_MAX_DECOMP_LEVELS DOWNTO 1;
SIGNAL src_cur : TYPE_ADR;
SIGNAL dst_cur : TYPE_ADR;
SIGNAL counter : INTEGER RANGE 0 TO 8;
BEGIN
dwtEng : dwt
PORT MAP (
clk => clk,
mem_data => mem_data,
mem_enable => mem_enable,
mem_rw => mem_rw,
mem_adr => mem_adr,
ctrl_sig => dwt_ctrl_sig,
ctrl_data => dwt_ctrl_data,
ready => dwt_ready,
reset => reset
);
PROCESS(clk, reset)
BEGIN
IF reset = '1' THEN
ready <= '1';
state <= TS2D_READY;
ELSE
IF clk'event AND clk = '1' THEN
CASE state IS
WHEN TS2D_READY =>
CASE ctrl_sig IS
WHEN Dwt2dCtrlVector(TCTRL_DWT2D_WIDTH) =>width <= UNSIGNED(ctrl_data);
WHEN Dwt2dCtrlVector(TCTRL_DWT2D_HEIGHT) =>height <=UNSIGNED(ctrl_data);
WHEN Dwt2dCtrlVector(TCTRL_DWT2D_TMP_DATA_OFFSET) =>tmpDataOffset <=
UNSIGNED(ctrl_data);
WHEN Dwt2dCtrlVector(TCTRL_DWT2D_LEVELS) =>levels <=
TO_INTEGER(UNSIGNED(ctrl_data));
state <= TS2D_INIT_HORIZ;
ready <= '0';
WHEN OTHERS =>
END CASE;
WHEN TS2D_INIT_HORIZ =>
```

```
src_cur <= (others => '0');
dst_cur <= (others => '0');
counter <= 0;
state <= TS2D_START_HORIZ;
WHEN TS2D_START_HORIZ =>CASE counter IS
WHEN 0 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_SRC_START_OFFSET);
dwt_ctrl_data <= STD_LOGIC_VECTOR(RESIZE(src_cur * CST_BUF_WIDTH,
CST_CTRL_DATA_BUS_WIDTH));
WHEN 1 => dwt_ctrl_sig <= DwtCtrlVector(TCTRL_SRC_STEP);
dwt_ctrl_data <= STD_LOGIC_VECTOR(TO_UNSIGNED(1,CST_CTRL_DATA_BUS_WIDTH));
WHEN 2 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_DST_START_OFFSET_LO);
dwt_ctrl_data <= STD_LOGIC_VECTOR(RESIZE(dst_cur * CST_BUF_WIDTH + tmpDataOffset,
CST_CTRL_DATA_BUS_WIDTH));
WHEN 3 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_DST_START_OFFSET_HI);
dwt_ctrl_data <= STD_LOGIC_VECTOR(RESIZE(dst_cur * CST_BUF_WIDTH +
SHIFT_RIGHT(width, 1) + tmpDataOffset, CST_CTRL_DATA_BUS_WIDTH));
WHEN 4 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_DST_STEP_LO);
dwt_ctrl_data <= STD_LOGIC_VECTOR(TO_UNSIGNED(1,CST_CTRL_DATA_BUS_WIDTH));
WHEN 5 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_DST_STEP_HI);
dwt_ctrl_data <= STD_LOGIC_VECTOR(TO_UNSIGNED(1,CST_CTRL_DATA_BUS_WIDTH));
WHEN 6 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_ELEM_COUNT);
dwt_ctrl_data <= STD_LOGIC_VECTOR(width);
WHEN 7 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_IGNORE);
state <= TS2D_CONTINUE_HORIZ;
WHEN OTHERS =>
END CASE;
counter <= counter + 1;
WHEN TS2D_CONTINUE_HORIZ =>
IF dwt_ready = '1' THEN
IF src_cur = height - 1 THEN
state <= TS2D_INIT_VERT;
ELSE
src_cur <= src_cur + 1;
dst_cur <= dst_cur + 1;
counter <= 0;
state <= TS2D_START_HORIZ;
END IF;
WHEN TS2D_INIT_VERT =>
src_cur <= (others => '0');
dst_cur <= (others => '0');
counter <= 0;
state <= TS2D_START_VERT;
WHEN TS2D_START_VERT =>
CASE counter IS
WHEN 0 =>
dwt_ctrl_sig <= DwtCtrlVector(TCTRL_SRC_START_OFFSET);
dwt_ctrl_data <= STD_LOGIC_VECTOR(RESIZE(src_cur + tmpDataOffset,
CST_CTRL_DATA_BUS_WIDTH));
WHEN 1 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_SRC_STEP);
dwt_ctrl_data <= STD_LOGIC_VECTOR(TO_UNSIGNED(CST_BUF_WIDTH,
CST_CTRL_DATA_BUS_WIDTH));
WHEN 2 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_DST_START_OFFSET_LO);
dwt_ctrl_data <= STD_LOGIC_VECTOR(RESIZE(dst_cur,CST_CTRL_DATA_BUS_WIDTH));
WHEN 3 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_DST_START_OFFSET_HI);
dwt_ctrl_data <= STD_LOGIC_VECTOR(RESIZE((dst_cur + SHIFT_RIGHT(height, 1) *
CST_BUF_WIDTH), CST_CTRL_DATA_BUS_WIDTH));
WHEN 4 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_DST_STEP_LO);
dwt_ctrl_data <= STD_LOGIC_VECTOR(TO_UNSIGNED(CST_BUF_WIDTH,
CST_CTRL_DATA_BUS_WIDTH));
WHEN 5 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_DST_STEP_HI);
```

```
dwt_ctrl_data <= STD_LOGIC_VECTOR(TO_UNSIGNED(CST_BUF_WIDTH,
CST_CTRL_DATA_BUS_WIDTH));
WHEN 6 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_ELEM_COUNT);
dwt_ctrl_data <= STD_LOGIC_VECTOR(height);
WHEN 7 =>dwt_ctrl_sig <= DwtCtrlVector(TCTRL_IGNORE);
state <= TS2D_CONTINUE_VERT;
WHEN OTHERS =>
END CASE;
counter <= counter + 1;
WHEN TS2D_CONTINUE_VERT =>
IF dwt_ready = '1' THEN
IF src_cur = width - 1 THEN
state <= TS2D_NEXT_LEVEL;
ELSE
src_cur <= src_cur + 1;
dst_cur <= dst_cur + 1;
counter <= 0;
state <= TS2D_START_VERT;
END IF;
END IF;
WHEN TS2D_NEXT_LEVEL =>IF levels = 1 THEN
state <= TS2D_TERMINATE;
ELSE
width <= SHIFT_RIGHT(width, 1);
height <= SHIFT_RIGHT(height, 1);
levels <= levels - 1;
state <= TS2D_INIT_HORIZ;
END IF;
WHEN TS2D_TERMINATE =>ready <= '1';
state <= TS2D_READY;
END CASE;
END IF;
END IF;
END PROCESS;
END Main;
```

A.2 Dwt.vhd

```
LIBRARY IEEE;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.MemManagerPkg.ALL;
PACKAGE DWTPkg IS
CONSTANT CST_CTRL_DATA_BUS_WIDTH : INTEGER := 16;
CONSTANT CST_DWT_CTRL_SIG_BUS_WIDTH : INTEGER := 3;
SUBTYPE TYPE_DWT_CTRL_SIGNAL IS
STD_LOGIC_VECTOR(CST_DWT_CTRL_SIG_BUS_WIDTH - 1 DOWNTO 0);
TYPE TYPE_DWT_CTRL_SIGNAL_ENUM IS (
TCTRL_SRC_START_OFFSET, TCTRL_SRC_STEP, TCTRL_DST_START_OFFSET_HI,
TCTRL_DST_START_OFFSET_LO, TCTRL_DST_STEP_HI, TCTRL_DST_STEP_LO,
TCTRL_ELEM_COUNT, TCTRL_IGNORE);
FUNCTION DwtCtrlVector(x : IN TYPE_DWT_CTRL_SIGNAL_ENUM) RETURN
TYPE_DWT_CTRL_SIGNAL;
COMPONENT dwt
PORT (
clk             : IN STD_LOGIC;
mem_data        : INOUT TYPE_DATA;
mem_enable      : OUT STD_LOGIC;
```

```
mem_rw          : OUT STD_LOGIC;
mem_adr                 : OUT TYPE_ADR;
ctrl_sig  : IN TYPE_DWT_CTRL_SIGNAL;
ctrl_data: IN STD_LOGIC_VECTOR(CST_CTRL_DATA_BUS_WIDTH - 1 DOWNTO 0);
ready           : OUT STD_LOGIC;
reset           : IN STD_LOGIC
);
END COMPONENT;
END DWTPkg;
PACKAGE BODY DWTPkg IS
FUNCTION DwtCtrlVector(x : IN TYPE_DWT_CTRL_SIGNAL_ENUM) RETURN
TYPE_DWT_CTRL_SIGNAL IS
BEGIN
RETURN
TYPE_DWT_CTRL_SIGNAL(TO_UNSIGNED(TYPE_DWT_CTRL_SIGNAL_ENUM'POS(x),
CST_DWT_CTRL_SIG_BUS_WIDTH));
END;
END DWTPkg;
LIBRARY IEEE;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.MemManagerPkg.ALL;
USE WORK.DWTPkg.ALL;
ENTITY dwt IS
PORT (
clk             : IN STD_LOGIC;
mem_data        : INOUT TYPE_DATA := (others => 'Z');
mem_enable      : OUT STD_LOGIC := 'Z';
mem_rw          : OUT STD_LOGIC := 'Z';
mem_adr                 : OUT TYPE_ADR := (others => 'Z');
ctrl_sig  : IN TYPE_DWT_CTRL_SIGNAL;
ctrl_data: IN STD_LOGIC_VECTOR(CST_CTRL_DATA_BUS_WIDTH - 1 DOWNTO 0);
ready           : OUT STD_LOGIC := '1';
reset           : IN STD_LOGIC
);
END dwt;
LIBRARY IEEE;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.MemManagerPkg.ALL;
USE WORK.DWTPkg.ALL;
ARCHITECTURE Main of dwt IS
TYPE TYPE_STATE IS (
TS_READY,
TS_CALCULATE,
TS_GET_X_ODD,
TS_GET_X_EVEN_A,
TS_CALC_HI_B,
TS_CYC_START,
TS_WRITE_LO,
TS_INCREMENT,
TS_CHECK_TERMINATION,
TS_SHUFFLE,
TS_RELOAD,
TS_CALC_HI_B_CYC,
TS_FINAL_PASS,
TS_WRITE_LO_FINAL,
TS_STORE_HI_A,
TS_WRITE_LO_LAST,
TS_TERMINATE);
```

```
FUNCTION filter_5_3_even(x, yp, yn : SIGNED(CST_BUF_DATA_BUS_WIDTH - 1 DOWNTO 0))
RETURN SIGNED IS
BEGIN
RETURN x + SHIFT_RIGHT(yp + yn + 2, 2);
END;
FUNCTION filter_5_3_odd(x, xp, xn : SIGNED(CST_BUF_DATA_BUS_WIDTH - 1 DOWNTO 0))
RETURN SIGNED IS
BEGIN
RETURN x - SHIFT_RIGHT(xp + xn, 1);
END;
SIGNAL state : TYPE_STATE;
SIGNAL x_odd, x_even_a, x_even_b, hi_a, hi_b, lo : SIGNED(CST_BUF_DATA_BUS_WIDTH - 1
DOWNTO 0);
SIGNAL src_start_offset, src_step, dst_start_offset_hi, dst_start_offset_lo, dst_step_lo, dst_step_hi,
elem_count: UNSIGNED(CST_CTRL_DATA_BUS_WIDTH - 1 DOWNTO 0);
SIGNAL access_bus : STD_LOGIC := '0';
SIGNAL counter : INTEGER RANGE 0 TO CST_MEM_READ_CLK_CYC;
SIGNAL i : INTEGER;
SIGNAL          data      : TYPE_DATA := (others => '0');
SIGNAL          enable    : STD_LOGIC := '0';
SIGNAL          rw                  : STD_LOGIC := '0';
SIGNAL          m_adr     : TYPE_ADR := (others => '0');
BEGIN
mem_data          <= (others => 'Z') WHEN access_bus = '0' ELSE data;
mem_enable        <= 'Z' WHEN state = TS_READY ELSE enable;
mem_rw            <= 'Z' WHEN state = TS_READY ELSE rw;
mem_adr                   <= (others => 'Z') WHEN state = TS_READY ELSE m_adr;
PROCESS(clk, reset)
VARIABLE adr : UNSIGNED(CST_CTRL_DATA_BUS_WIDTH - 1 DOWNTO 0);
VARIABLE srcAdr : UNSIGNED(CST_CTRL_DATA_BUS_WIDTH - 1 DOWNTO 0);
VARIABLE dstAdrHi : UNSIGNED(CST_CTRL_DATA_BUS_WIDTH - 1 DOWNTO 0);
VARIABLE dstAdrLo : UNSIGNED(CST_CTRL_DATA_BUS_WIDTH - 1 DOWNTO 0);
VARIABLE dstAdr : UNSIGNED(CST_CTRL_DATA_BUS_WIDTH - 1 DOWNTO 0);
VARIABLE h : SIGNED(CST_BUF_DATA_BUS_WIDTH - 1 DOWNTO 0);
VARIABLE lo : SIGNED(CST_BUF_DATA_BUS_WIDTH - 1 DOWNTO 0);
BEGIN
IF reset = '1' THEN
ready <= '1';
state <= TS_READY;
access_bus <= '0';
ELSE
IF clk'event AND clk = '1' THEN
CASE state IS
WHEN TS_READY =>CASE ctrl_sig IS
WHEN DwtCtrlVector(TCTRL_SRC_START_OFFSET) =>src_start_offset <=
UNSIGNED(ctrl_data);
WHEN DwtCtrlVector(TCTRL_SRC_STEP) =>src_step <= UNSIGNED(ctrl_data);
WHEN DwtCtrlVector(TCTRL_DST_START_OFFSET_LO) =>dst_start_offset_lo <=
UNSIGNED(ctrl_data);
WHEN DwtCtrlVector(TCTRL_DST_START_OFFSET_HI) =>dst_start_offset_hi <=
UNSIGNED(ctrl_data);
WHEN DwtCtrlVector(TCTRL_DST_STEP_LO) =>dst_step_lo <= UNSIGNED(ctrl_data);
WHEN DwtCtrlVector(TCTRL_DST_STEP_HI) =>dst_step_hi <= UNSIGNED(ctrl_data);
WHEN DwtCtrlVector(TCTRL_ELEM_COUNT) =>elem_count <= UNSIGNED(ctrl_data);
state <= TS_CALCULATE;
ready <= '0';
WHEN OTHERS =>
END CASE;
WHEN TS_CALCULATE =>srcAdr := src_start_offset + SHIFT_LEFT(src_step, 1);
ReadMemReq(enable, rw, m_adr, srcAdr);
```

```
CheckReadClkCyc(counter, enable);
IF (IsReadCycOver(counter) = '1') THEN
state <= TS_GET_X_ODD;
END IF;
x_even_b <= SIGNED(mem_data);
WHEN TS_GET_X_ODD =>
srcAdr := src_start_offset + src_step;
ReadMemReq(enable, rw, m_adr, srcAdr);
CheckReadClkCyc(counter, enable);
IF (IsReadCycOver(counter) = '1') THEN
state <= TS_GET_X_EVEN_A;
END IF;
x_odd <= SIGNED(mem_data);
WHEN TS_GET_X_EVEN_A =>
srcAdr := src_start_offset;
ReadMemReq(enable, rw, m_adr, srcAdr);
CheckReadClkCyc(counter, enable);
IF (IsReadCycOver(counter) = '1') THEN
state <= TS_CALC_HI_B;
END IF;
x_even_a <= SIGNED(mem_data);
WHEN TS_CALC_HI_B =>
h := filter_5_3_odd(x_odd, x_even_a, x_even_b);
hi_b <= h;
hi_a <= h;
i <= 1;
state <= TS_CYC_START;
srcAdr := src_start_offset + src_step;
dstAdrHi := dst_start_offset_hi;
dstAdrLo := dst_start_offset_lo;
WHEN TS_CYC_START =>WriteMemReq(enable, rw, m_adr, dstAdrHi, data, TYPE_DATA(hi_b),
access_bus);
CheckWriteClkCyc(counter, enable, access_bus);
IF (IsWriteCycOver(counter) = '1') THEN
state <= TS_WRITE_LO;
END IF;
lo := filter_5_3_even(x_even_a, hi_a, hi_b);
WHEN TS_WRITE_LO =>
WriteMemReq(enable, rw, m_adr, dstAdrLo, data, TYPE_DATA(lo), access_bus);
CheckWriteClkCyc(counter, enable, access_bus);
IF (IsWriteCycOver(counter) = '1') THEN
state <= TS_INCREMENT;
END IF;
WHEN TS_INCREMENT =>
i <= i + 2;
srcAdr := srcAdr + SHIFT_LEFT(src_step, 1);
dstAdrHi := dstAdrHi + dst_step_hi;
dstAdrLo := dstAdrLo + dst_step_lo;
state <= TS_CHECK_TERMINATION;
WHEN TS_CHECK_TERMINATION =>
IF i >= TO_INTEGER(elem_count - 1) THEN
state <= TS_FINAL_PASS;
ELSE
state <= TS_SHUFFLE;
END IF;
WHEN TS_SHUFFLE =>
hi_a <= hi_b;
x_even_a <= x_even_b;
ReadMemReq(enable, rw, m_adr, srcAdr);
CheckReadClkCyc(counter, enable);
```

```
IF (IsReadCycOver(counter) = '1') THEN
state <= TS_RELOAD;
END IF;
x_odd <= SIGNED(mem_data);
WHEN TS_RELOAD =>
adr := srcAdr + src_step;
ReadMemReq(enable, rw, m_adr, adr);
CheckReadClkCyc(counter, enable);
IF (IsReadCycOver(counter) = '1') THEN
state <= TS_CALC_HI_B_CYC;
END IF;
x_even_b <= SIGNED(mem_data);
WHEN TS_CALC_HI_B_CYC =>
hi_b <= filter_5_3_odd(x_odd, x_even_a, x_even_b);
state <= TS_CYC_START;
WHEN TS_FINAL_PASS =>
IF (elem_count(0) = '1') THEN
lo := filter_5_3_even(x_even_b, hi_b, hi_b);
state <= TS_WRITE_LO_FINAL;
ELSE
ReadMemReq(enable, rw, m_adr, srcAdr);
CheckReadClkCyc(counter, enable);
x_odd <= SIGNED(mem_data);
hi_a <= filter_5_3_odd(SIGNED(mem_data), x_even_b, x_even_b);

IF (IsReadCycOver(counter) = '1') THEN
state <= TS_STORE_HI_A;
END IF;
END IF;
WHEN TS_WRITE_LO_FINAL =>
WriteMemReq(enable, rw, m_adr, dstAdrLo, data, TYPE_DATA(lo), access_bus);
CheckWriteClkCyc(counter, enable, access_bus);
IF (IsWriteCycOver(counter) = '1') THEN
state <= TS_TERMINATE;
END IF;
WHEN TS_STORE_HI_A =>lo := filter_5_3_even(x_even_b, hi_b, hi_a);
WriteMemReq(enable, rw, m_adr, dstAdrHi, data, TYPE_DATA(hi_a), access_bus);
CheckWriteClkCyc(counter, enable, access_bus);
IF (IsWriteCycOver(counter) = '1') THEN
state <= TS_WRITE_LO_LAST;
END IF;
WHEN TS_WRITE_LO_LAST=>
WriteMemReq(enable, rw, m_adr, dstAdrLo, data, TYPE_DATA(lo), access_bus);
CheckWriteClkCyc(counter, enable, access_bus);
IF (IsWriteCycOver(counter) = '1') THEN
state <= TS_TERMINATE;
END IF;
WHEN TS_TERMINATE =>
ready <= '1';
state <= TS_READY;
END CASE;
END IF;
END IF;
END PROCESS;
END Main;
```

A.3 MemManager.vhd

```
LIBRARY IEEE;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
PACKAGE MemManagerPkg IS
CONSTANT CST_BUF_WIDTH: INTEGER:= 64;
CONSTANT CST_BUF_HEIGHT: INTEGER:= 64;
CONSTANT CST_BUF_WIDTH_BUS_WIDTH: INTEGER:= 8;
CONSTANT CST_BUF_HEIGHT_BUS_WIDTH: INTEGER:= 8;
CONSTANT CST_BUF_DATA_BUS_WIDTH: INTEGER:= 8;
CONSTANT CST_MEM_READ_CLK_CYC: INTEGER:= 2;
CONSTANT CST_MEM_WRITE_CLK_CYC: INTEGER:= 2;
CONSTANT CST_MEM_ADR_BUS_WIDTH: INTEGER:= 13;
SUBTYPE TYPE_DATA IS STD_LOGIC_VECTOR(CST_BUF_DATA_BUS_WIDTH - 1
DOWNTO 0);
SUBTYPE TYPE_ADR IS UNSIGNED(CST_MEM_ADR_BUS_WIDTH - 1 DOWNTO 0);
PROCEDURE ReadMemReq(
SIGNAL enable : OUT STD_LOGIC;
SIGNAL rw : OUT STD_LOGIC;
SIGNAL adr : OUT TYPE_ADR;
ARIABLE address : UNSIGNED
);
PROCEDURE CheckReadClkCyc(
SIGNAL cyc_counter : INOUT INTEGER RANGE 0 TO CST_MEM_READ_CLK_CYC;
SIGNAL enable : OUT STD_LOGIC
);
FUNCTION IsReadCycOver(SIGNAL cyc_counter : IN INTEGER RANGE 0 TO
CST_MEM_READ_CLK_CYC) RETURN STD_LOGIC;
PROCEDURE WriteMemReq(
SIGNAL enable : OUT STD_LOGIC;
SIGNAL rw : OUT STD_LOGIC;
SIGNAL adr : OUT TYPE_ADR;
VARIABLE address : UNSIGNED;
SIGNAL dst_data : OUT TYPE_DATA;
value : IN TYPE_DATA;
SIGNAL access_bus : OUT STD_LOGIC
);
PROCEDURE CheckWriteClkCyc(
SIGNAL cyc_counter : INOUT INTEGER RANGE 0 TO CST_MEM_READ_CLK_CYC;
SIGNAL enable : OUT STD_LOGIC;
SIGNAL access_bus : OUT STD_LOGIC
);

FUNCTION IsWriteCycOver(SIGNAL cyc_counter : IN INTEGER RANGE 0 TO
CST_MEM_READ_CLK_CYC) RETURN STD_LOGIC;
COMPONENT Memory
GENERIC (
load_offset    INTEGER := 0;
dump_offset                : INTEGER := 0;
width                      : INTEGER := 0;
height                     : INTEGER := 0;
inputFileName    : STRING := "";
outputFileName   : STRING := ""
);
PORT (
adr        : IN TYPE_ADR;
rw                 : IN STD_LOGIC;
data       : INOUT TYPE_DATA;
```

```
enable    : IN STD_LOGIC;
dump      : IN STD_LOGIC;
clk       : IN STD_LOGIC
);
END COMPONENT;
END MemManagerPkg;
LIBRARY IEEE;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
PACKAGE BODY MemManagerPkg IS
PROCEDURE ReadMemReq(
SIGNAL enable : OUT STD_LOGIC;
SIGNAL rw : OUT STD_LOGIC;
SIGNAL adr : OUT TYPE_ADR;
VARIABLE address : UNSIGNED) IS
BEGIN
enable <= '1';
rw <= '1';
adr <= TYPE_ADR(RESIZE(address, CST_MEM_ADR_BUS_WIDTH));
END;
PROCEDURE CheckReadClkCyc(
SIGNAL cyc_counter : INOUT INTEGER RANGE 0 TO CST_MEM_READ_CLK_CYC;
SIGNAL enable : OUT STD_LOGIC) IS
BEGIN
IF cyc_counter = CST_MEM_READ_CLK_CYC THEN
cyc_counter <= 0;
enable <= '0';
ELSE
cyc_counter <= cyc_counter + 1;
END IF;
END;
FUNCTION IsReadCycOver(SIGNAL cyc_counter : IN INTEGER RANGE 0 TO
CST_MEM_READ_CLK_CYC) RETURN STD_LOGIC IS
BEGIN
IF cyc_counter = CST_MEM_READ_CLK_CYC THEN
RETURN '1';
ELSE
RETURN '0';
END IF;
END;
PROCEDURE WriteMemReq(
SIGNAL enable : OUT STD_LOGIC;
SIGNAL rw : OUT STD_LOGIC;
SIGNAL adr : OUT TYPE_ADR;
VARIABLE address : UNSIGNED;
SIGNAL dst_data : OUT TYPE_DATA;
value : IN TYPE_DATA;
SIGNAL access_bus : OUT STD_LOGIC) IS
BEGIN
access_bus <= '1';
enable <= '1';
rw <= '0';
adr <= TYPE_ADR(RESIZE(address, CST_MEM_ADR_BUS_WIDTH));
dst_data <= value;
END;
PROCEDURE CheckWriteClkCyc(
SIGNAL cyc_counter : INOUT INTEGER RANGE 0 TO CST_MEM_READ_CLK_CYC;
SIGNAL enable : OUT STD_LOGIC;
SIGNAL access_bus : OUT STD_LOGIC) IS
BEGIN
```

```
IF cyc_counter = CST_MEM_WRITE_CLK_CYC THEN
access_bus <= '0';
enable <= '0';
cyc_counter <= 0;
ELSE
cyc_counter <= cyc_counter + 1;
END IF;
END;
FUNCTION IsWriteCycOver(SIGNAL cyc_counter : IN INTEGER RANGE 0 TO
CST_MEM_READ_CLK_CYC) RETURN STD_LOGIC IS
BEGIN
IF cyc_counter = CST_MEM_WRITE_CLK_CYC THEN
RETURN '1';
ELSE
RETURN '0';
END IF;
END;
END MemManagerPkg;
LIBRARY IEEE;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.MemManagerPkg.ALL;
USE STD.TEXTIO.ALL;
USE IEEE.std_logic_textio.all;
ENTITY Memory IS
GENERIC (
load_offset             : INTEGER;
dump_offset             : INTEGER;
width           : INTEGER;
height          : INTEGER;
inputFileName           : STRING;
outputFileName  : STRING
);
PORT (
adr     : IN TYPE_ADR;
rw      : IN STD_LOGIC;
data    : INOUT TYPE_DATA := (others => 'Z');
enable  : IN STD_LOGIC := '0';
dump    : IN STD_LOGIC := '0';
clk     : IN STD_LOGIC
);
END Memory;
ARCHITECTURE Main OF Memory IS
TYPE TYPE_MATRIX IS ARRAY(0 TO CST_BUF_WIDTH * CST_BUF_HEIGHT * 2 - 1) OF
TYPE_DATA;
PROCEDURE FillMemory(SIGNAL memory : OUT TYPE_MATRIX; fileName : STRING; picWidth
: INTEGER; picHeight : INTEGER) IS
FILE ifile : TEXT IS IN fileName;
VARIABLE l_in : LINE;
VARIABLE d : STD_LOGIC_VECTOR(CST_BUF_DATA_BUS_WIDTH - 1 DOWNTO 0);
BEGIN
FOR ty IN 0 TO picHeight - 1 LOOP
readline(ifile, l_in);
FOR tx IN 0 TO picWidth - 1 LOOP
hread(l_in, d);
memory(tx + ty * CST_BUF_WIDTH + load_offset) <= TYPE_DATA(d);
END LOOP;
END LOOP;
END;
```

```
PROCEDURE DumpMemory(SIGNAL memory : IN TYPE_MATRIX; filename : STRING; picWidth
: INTEGER; picHeight : INTEGER) IS
FILE ofile : TEXT IS OUT fileName;
VARIABLE l_out : LINE;
VARIABLE d : STD_LOGIC_VECTOR(CST_BUF_DATA_BUS_WIDTH - 1 DOWNTO 0);
BEGIN
FOR ty IN 0 TO picHeight - 1 LOOP
FOR tx IN 0 TO picWidth - 1 LOOP
d := STD_LOGIC_VECTOR(memory(tx + ty * CST_BUF_WIDTH + dump_offset));
hwrite(l_out, d);
write(l_out, ' ');
END LOOP;
writeline(ofile, l_out);
END LOOP;
END;
SIGNAL mem : TYPE_MATRIX;
BEGIN
PROCESS(clk, enable, rw, dump)
VARIABLE mem_init : BOOLEAN := FALSE;
BEGIN
IF NOT mem_init THEN
mem_init := TRUE;
IF inputFileName'LENGTH > 0 THEN
FillMemory(mem, inputFileName, width, height);
END IF;
END IF;
IF dump'event AND dump = '1' THEN
IF outputFileName'LENGTH > 0 THEN
DumpMemory(mem, outputFileName, width, height);
END IF;
END IF;
IF clk'event AND clk = '1' THEN
IF enable = '1' THEN
IF rw = '1' THEN
data <= mem(TO_INTEGER(UNSIGNED(adr)));
ELSE
mem(TO_INTEGER(UNSIGNED(adr))) <= data;
END IF;
ELSE
data <= (others =>'Z');
END IF;
END IF;
END PROCESS;
END Main;
```

A.4 testDWT2D.vhd

```
LIBRARY IEEE;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.MemManagerPkg.ALL;
USE WORK.DWT2DPkg.ALL;
ENTITY testDWT2D IS
END testDWT2D;
ARCHITECTURE Main of testDWT2D IS
SIGNAL adr : TYPE_ADR := (others => '0');
SIGNAL clk : STD_LOGIC := '0';
SIGNAL enable : STD_LOGIC := '1';
SIGNAL data : TYPE_DATA;
```

```
SIGNAL dump : STD_LOGIC;
SIGNAL rw : STD_LOGIC := '1';
SIGNAL ctrl_sig : TYPE_DWT2D_CTRL_SIGNAL;
SIGNAL ctrl_data             : STD_LOGIC_VECTOR(CST_CTRL2D_DATA_BUS_WIDTH - 1
DOWNTO 0);
SIGNAL ready : STD_LOGIC;
SIGNAL reset : STD_LOGIC;
CONSTANT PERIOD : TIME := 50 ns;
BEGIN
mem : Memory
GENERIC MAP (
width => CST_BUF_WIDTH,
height => CST_BUF_HEIGHT,
inputFileName => "TestData.txt",
outputFileName => "DWTResult.txt"
)
PORT MAP (
adr => adr,
rw => rw,
data => data,
enable => enable,
dump => dump,
clk => clk
);
dwtL : dwt2d
PORT MAP (
clk => clk,
mem_data => data,
mem_enable => enable,
mem_rw => rw,
mem_adr => adr,
ctrl_sig  => ctrl_sig,
ctrl_data => ctrl_data,
ready => ready,
reset => reset
);
clk <= NOT clk AFTER PERIOD/2;
PROCESS
VARIABLE counter : INTEGER := 0;
BEGIN
reset <= '1';
WAIT FOR PERIOD;
reset <= '0';
ctrl_sig <= Dwt2dCtrlVector(TCTRL_DWT2D_WIDTH);
ctrl_data <=
STD_LOGIC_VECTOR(TO_UNSIGNED(CST_BUF_WIDTH,CST_CTRL2D_DATA_BUS_WIDT
H));
WAIT FOR PERIOD;
ctrl_sig <= Dwt2dCtrlVector(TCTRL_DWT2D_HEIGHT);
ctrl_data <=
STD_LOGIC_VECTOR(TO_UNSIGNED(CST_BUF_HEIGHT,CST_CTRL2D_DATA_BUS_WIDT
H));
WAIT FOR PERIOD;
ctrl_sig <= Dwt2dCtrlVector(TCTRL_DWT2D_TMP_DATA_OFFSET);
ctrl_data <= STD_LOGIC_VECTOR(TO_UNSIGNED(CST_BUF_WIDTH * ST_BUF_HEIGHT,
CST_CTRL2D_DATA_BUS_WIDTH));
WAIT FOR PERIOD;
ctrl_sig <= Dwt2dCtrlVector(TCTRL_DWT2D_LEVELS);
ctrl_data <= STD_LOGIC_VECTOR(TO_UNSIGNED(2,CST_CTRL2D_DATA_BUS_WIDTH));
WAIT UNTIL ready = '1';
```

```
dump <= '1';
WAIT FOR PERIOD * 3;
assert false
report "Simulation Finished"
severity Failure;
END PROCESS;
END Main;
```

A.5 testMemManager.vhd

```
LIBRARY IEEE;
USE IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.MemManagerPkg.ALL;
ENTITY testMemManager IS
END testMemManager;
ARCHITECTURE Main of testMemManager IS
SIGNAL adr : TYPE_ADR := (others => '0');
SIGNAL clk : STD_LOGIC := '0';
SIGNAL enable : STD_LOGIC := '1';
SIGNAL data : TYPE_DATA;
SIGNAL dump : STD_LOGIC;
SIGNAL rw : STD_LOGIC := '1';
BEGIN
mem : Memory
GENERIC MAP (
width => CST_BUF_WIDTH,
height => CST_BUF_WIDTH,
inputFileName => "TestData.txt",
outputFileName => "ResultData.txt"
)
PORT MAP (
adr => adr,
rw => rw,
data => data,
enable => enable,
dump => dump,
clk => clk
);
clk <= NOT clk AFTER 20 ns;
PROCESS(clk)
VARIABLE counter : INTEGER := 0;
VARIABLE address : INTEGER := 20;
BEGIN
IF clk'event AND clk = '0' THEN
IF counter = 10 THEN
dump <= '1';
ELSE
IF counter = 11 THEN
ASSERT FALSE REPORT "End of simulation" SEVERITY FAILURE;
END IF;
adr <= TYPE_ADR(TO_UNSIGNED(address, CST_MEM_ADR_BUS_WIDTH));
enable <= '1';
rw <= '1';
END IF;
counter := counter + 1;
address := address + 1;
END IF;
END PROCESS;
END Main;
```

A.6 quantizer.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use ieee.std_logic_signed.all;
entity QUANT is
port(
QUANTclk : in std_logic;
QUANTen  : in std_logic;
QUANTmax : in  std_logic_vector (15 downto 0);
QUANTmin : in  std_logic_vector (15 downto 0);
QUANTin  : in  std_logic_vector (15 downto 0);
QUANTout : out std_logic_vector (3 downto 0));
end QUANT;
architecture structural of QUANT is
subtype std4 is std_logic_vector  ( 3 downto 0);
subtype std16 is std_logic_vector (15 downto 0);
subtype std20 is std_logic_vector (19 downto 0);
signal r      : std16;
signal r_by_2 : std20;
signal r_by_4 : std20;
signal r_by_8 : std20;
signal r_by_16: std20;
signal in1    : std16;
signal in2    : std16;
signal in3    : std16;
signal in4    : std16;
signal cmp1   : std20;
signal cmp2   : std20;
signal cmp3   : std20;
signal cmp4   : std20;
signal level1 : std4;
signal level2 : std4;
signal level3 : std4;
signal level4 : std4;
begin
r <= (QUANTmax - QUANTmin);
run : process(QUANTclk)
begin
if(rising_edge(QUANTclk)) then
if(QUANTen = '1') then
r_by_2 <= (r(15) & r & "000");
r_by_4 <= (r_by_2(19) & r_by_2(19 downto 1));
r_by_8 <= (r_by_4(19) & r_by_4(19 downto 1));
r_by_16 <= (r_by_8(19) & r_by_8(19 downto 1));
in4 <= in3;
in3 <= in2;
in2 <= in1;
in1 <= (QUANTin - QUANTmin);
if(SIGNED(in1) > SIGNED(r_by_2(19 downto 4))) then
level1 <= "1000";
cmp1 <= (r_by_2 + r_by_4);
else
level1 <= "0000";
cmp1 <= (r_by_2 - r_by_4);
end if;
if(SIGNED(in2 & '0') > SIGNED(cmp1(19 downto 3))) then
level2 <= (level1 or "0100");
cmp2 <= (cmp1 + r_by_8);
else
```

```
level2 <= level1;
cmp2 <= (cmp1 - r_by_8);
end if;
if(SIGNED(in3 & "0000") > SIGNED(cmp2(19 downto 0))) then
level3 <= (level2 or "0010");
cmp3 <= (cmp2 + r_by_16);
else
level3 <= level2;
cmp3 <= (cmp2 - r_by_16);
end if;
cmp4 <= cmp3;
if(SIGNED(in4 & "0000") > SIGNED(cmp3(19 downto 0))) then
level4 <= (level3 or "0001");
else
level4 <= level3;
end if;
end if;
end if;
end process run;
QUANTout <= level4;
end structural;
```

A.7 rle.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_signed.all;
entity RLE is
port(
RLEclk     : in std_logic;
RLEreset   : in std_logic;
RLEen      : in std_logic;
RLEflush   : in std_logic;
RLEin      : in std_logic_vector (15 downto 0);
RLEzeroth  : in std_logic_vector (15 downto 0);
RLEout     : out std_logic_vector ( 7 downto 0);
RLErunning : out std_logic;
RLEspellEnd: out std_logic);
end RLE;
architecture structural of RLE is
signal z1    : std_logic;
signal z2    : std_logic;
signal z3    : std_logic;
signal z4    : std_logic;
signal z5    : std_logic;
signal s240  : std_logic;
signal count : std_logic_vector ( 7 downto 0) := "00010000";
begin
run : process(RLEreset, RLEclk)
begin
if(RLEreset = '1') then
count <= "00001111";
z1 <= '0';
z2 <= '0';
z3 <= '0';
z4 <= '0';
z5 <= '0';
elsif(rising_edge(RLEclk)) then
```

```
if(RLEen = ’1’) then
if((SIGNED(RLEin) < SIGNED(RLEzeroth)) and
(SIGNED(RLEin) > SIGNED(-RLEzeroth)) and
(RLEflush = ’0’)) then
z1 <= ’1’;
else
z1 <= ’0’;
end if;
z2 <= z1;
z3 <= z2;
z4 <= z3;
z5 <= z4;
s240 <= ’0’;
 if(z4 = ’0’) then
count <= "00001111";
else
if(count = "11111110") then
s240 <= ’1’;
end if;
if(count = "11111111") then
count <= "00010000";
else
count <= UNSIGNED(count) + 1;
end if;
end if;
end if;
end if;
end process run;
RLEout <= count;
RLErunning <= z5;
RLEspellEnd <= (z5 and not(z4)) or s240;
end structural;
```

A.8 huffman.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity HUFF is
port(
HUFFclk : in std_logic;
HUFFin : in std_logic_vector (7 downto 0);
HUFFlout : out std_logic_vector (4 downto 0);
HUFFdout : out std_logic_vector (17 downto 0));
end HUFF;
architecture structural of HUFF is
signal tmp : std_logic_vector(7 downto 0);
begin
run : process (HUFFclk)
begin
if(rising_edge(HUFFclk)) then
tmp <= HUFFin;
case tmp is
when "00000000" => HUFFdout<="111010010XXXXXXXXX"; HUFFlout<="01001";
when "00000001" => HUFFdout<="0110011XXXXXXXXXXX"; HUFFlout<="00111";
when "00000010" => HUFFdout<="111000XXXXXXXXXXXX"; HUFFlout<="00110";
when "00000011" => HUFFdout<="01101XXXXXXXXXXXXX"; HUFFlout<="00101";
when "00000100" => HUFFdout<="0000XXXXXXXXXXXXXX"; HUFFlout<="00100";
when "00000101" => HUFFdout<="1101XXXXXXXXXXXXXX"; HUFFlout<="00100";
when "00000110" => HUFFdout<="100XXXXXXXXXXXXXXX"; HUFFlout<="00011";
```

```
when "00000111" => HUFFdout<="1111XXXXXXXXXXXXX"; HUFFlout<="00100";
when "00001000" => HUFFdout<="010XXXXXXXXXXXXXX"; HUFFlout<="00011";
when "00001001" => HUFFdout<="001XXXXXXXXXXXXXX"; HUFFlout<="00011";
when "00001010" => HUFFdout<="0111XXXXXXXXXXXXX"; HUFFlout<="00100";
when "00001011" => HUFFdout<="10101XXXXXXXXXXXXX"; HUFFlout<="00101";
when "00001100" => HUFFdout<="111011XXXXXXXXXXXX"; HUFFlout<="00110";
when "00001101" => HUFFdout<="101001XXXXXXXXXXXX"; HUFFlout<="00110";
when "00001110" => HUFFdout<="0001110XXXXXXXXXXX"; HUFFlout<="00111";
when "00001111" => HUFFdout<="10110111XXXXXXXXXX"; HUFFlout<="01000";
when "00010000" => HUFFdout<="1100XXXXXXXXXXXXX"; HUFFlout<="00100";
when "00010001" => HUFFdout<="10111XXXXXXXXXXXXX"; HUFFlout<="00101";
when "00010010" => HUFFdout<="00010XXXXXXXXXXXXX"; HUFFlout<="00101";
when "00010011" => HUFFdout<="101100XXXXXXXXXXXX"; HUFFlout<="00110";
when "00010100" => HUFFdout<="000110XXXXXXXXXXXX"; HUFFlout<="00110";
when "00010101" => HUFFdout<="1110011XXXXXXXXXXX"; HUFFlout<="00111";
when "00010110" => HUFFdout<="1010000XXXXXXXXXXX"; HUFFlout<="00111";
when "00010111" => HUFFdout<="0110000XXXXXXXXXXX"; HUFFlout<="00111";
when "00011000" => HUFFdout<="11101011XXXXXXXXXX"; HUFFlout<="01000";
when "00011001" => HUFFdout<="10110100XXXXXXXXXX"; HUFFlout<="01000";
when "00011010" => HUFFdout<="01100011XXXXXXXXXX"; HUFFlout<="01000";
when "00011011" => HUFFdout<="00011111XXXXXXXXXX"; HUFFlout<="01000";
when "00011100" => HUFFdout<="111001011XXXXXXXXX"; HUFFlout<="01001";
when "00011101" => HUFFdout<="111010011XXXXXXXXX"; HUFFlout<="01001";
when "00011110" => HUFFdout<="101101101XXXXXXXXX"; HUFFlout<="01001";
when "00011111" => HUFFdout<="111001001XXXXXXXXX"; HUFFlout<="01001";
when "00100000" => HUFFdout<="101000111XXXXXXXXX"; HUFFlout<="01001";
when "00100001" => HUFFdout<="011001011XXXXXXXXX"; HUFFlout<="01001";
when "00100010" => HUFFdout<="011000101XXXXXXXXX"; HUFFlout<="01001";
when "00100011" => HUFFdout<="1110101011XXXXXXXX"; HUFFlout<="01010";
when "00100100" => HUFFdout<="000111101XXXXXXXXX"; HUFFlout<="01001";
when "00100101" => HUFFdout<="1110101001XXXXXXXX"; HUFFlout<="01010";
when "00100110" => HUFFdout<="1011010100XXXXXXXX"; HUFFlout<="01010";
when "00100111" => HUFFdout<="1011011000XXXXXXXX"; HUFFlout<="01010";
when "00101000" => HUFFdout<="1110010100XXXXXXXX"; HUFFlout<="01010";
when "00101001" => HUFFdout<="1011010111XXXXXXXX"; HUFFlout<="01010";
when "00101010" => HUFFdout<="1011010110XXXXXXXX"; HUFFlout<="01010";
when "00101011" => HUFFdout<="1110100000XXXXXXXX"; HUFFlout<="01010";
when "00101100" => HUFFdout<="1010001100XXXXXXXX"; HUFFlout<="01010";
when "00101101" => HUFFdout<="1010001101XXXXXXXX"; HUFFlout<="01010";
when "00101110" => HUFFdout<="1010001010XXXXXXXX"; HUFFlout<="01010";
when "00101111" => HUFFdout<="11101000100XXXXXXX"; HUFFlout<="01011";
when "00110000" => HUFFdout<="0110010100XXXXXXXX"; HUFFlout<="01010";
when "00110001" => HUFFdout<="11100100001XXXXXXX"; HUFFlout<="01011";
when "00110010" => HUFFdout<="11100101010XXXXXXX"; HUFFlout<="01011";
when "00110011" => HUFFdout<="0110001001XXXXXXXX"; HUFFlout<="01010";
when "00110100" => HUFFdout<="0110010001XXXXXXXX"; HUFFlout<="01010";
when "00110101" => HUFFdout<="10110101010XXXXXXX"; HUFFlout<="01011";
when "00110110" => HUFFdout<="11101000010XXXXXXX"; HUFFlout<="01011";
when "00110111" => HUFFdout<="11100100000XXXXXXX"; HUFFlout<="01011";
when "00111000" => HUFFdout<="01100101011XXXXXXX"; HUFFlout<="01011";
when "00111001" => HUFFdout<="01100101010XXXXXXX"; HUFFlout<="01011";
when "00111010" => HUFFdout<="10100010001XXXXXXX"; HUFFlout<="01011";
when "00111011" => HUFFdout<="111001000101XXXXXX"; HUFFlout<="01100";
when "00111100" => HUFFdout<="101101100101XXXXXX"; HUFFlout<="01100";
when "00111101" => HUFFdout<="111010001110XXXXXX"; HUFFlout<="01100";
when "00111110" => HUFFdout<="111010101011XXXXXX"; HUFFlout<="01100";
when "00111111" => HUFFdout<="10100010110XXXXXXX"; HUFFlout<="01011";
when "01000000" => HUFFdout<="01100010001XXXXXXX"; HUFFlout<="01011";
when "01000001" => HUFFdout<="111010100000XXXXXX"; HUFFlout<="01100";
when "01000010" => HUFFdout<="01100100111XXXXXXX"; HUFFlout<="01011";
```

```
when "01000011" => HUFFdout<="1110010101l0XXXXXX"; HUFFlout<="01100";
when "01000100" => HUFFdout<="0110010000l0XXXXXX"; HUFFlout<="01100";
when "01000101" => HUFFdout<="1110010101l1XXXXXX"; HUFFlout<="01100";
when "01000110" => HUFFdout<="1110100001l1XXXXXX"; HUFFlout<="01100";
when "01000111" => HUFFdout<="01100100100XXXXXXX"; HUFFlout<="01011";
when "01001000" => HUFFdout<="1011011001000XXXXX"; HUFFlout<="01101";
when "01001001" => HUFFdout<="0110010010l1XXXXXX"; HUFFlout<="01100";
when "01001010" => HUFFdout<="1011010101l0XXXXXX"; HUFFlout<="01100";
when "01001011" => HUFFdout<="1110100011l1XXXXXX"; HUFFlout<="01100";
when "01001100" => HUFFdout<="1010001000l1XXXXXX"; HUFFlout<="01100";
when "01001101" => HUFFdout<="1110100011l0XXXXXX"; HUFFlout<="01101";
when "01001110" => HUFFdout<="1110100011l0XXXXXX"; HUFFlout<="01100";
when "01001111" => HUFFdout<="1110100001100XXXXX"; HUFFlout<="01101";
when "01010000" => HUFFdout<="1110101000101XXXXX"; HUFFlout<="01101";
when "01010001" => HUFFdout<="0110010011001XXXXX"; HUFFlout<="01101";
when "01010010" => HUFFdout<="01100100101001XXXX"; HUFFlout<="01110";
when "01010011" => HUFFdout<="0110010011010XXXXX"; HUFFlout<="01101";
when "01010100" => HUFFdout<="1110100010111XXXXX"; HUFFlout<="01101";
when "01010101" => HUFFdout<="11101000111110XXXX"; HUFFlout<="01110";
when "01010110" => HUFFdout<="1110010001100XXXXX"; HUFFlout<="01101";
when "01010111" => HUFFdout<="1110101000111XXXXX"; HUFFlout<="01101";
when "01011000" => HUFFdout<="1011011001101XXXXX"; HUFFlout<="01101";
when "01011001" => HUFFdout<="1011011001110XXXXX"; HUFFlout<="01101";
when "01011010" => HUFFdout<="11101010100100XXXX"; HUFFlout<="01110";
when "01011011" => HUFFdout<="1011011001111XXXXX"; HUFFlout<="01101";
when "01011100" => HUFFdout<="1010001001111XXXXX"; HUFFlout<="01101";
when "01011101" => HUFFdout<="0110010010101XXXXX"; HUFFlout<="01101";
when "01011110" => HUFFdout<="1110100010100XXXXX"; HUFFlout<="01101";
when "01011111" => HUFFdout<="0110001000011XXXXX"; HUFFlout<="01101";
when "01100000" => HUFFdout<="1010001011100XXXXX"; HUFFlout<="01101";
when "01100001" => HUFFdout<="1110100001101XXXXX"; HUFFlout<="01101";
when "01100010" => HUFFdout<="1110101010000XXXXX"; HUFFlout<="01101";
when "01100011" => HUFFdout<="1110010001101XXXXX"; HUFFlout<="01101";
when "01100100" => HUFFdout<="11100100011100XXXX"; HUFFlout<="01110";
when "01100101" => HUFFdout<="1011010101111XXXXX"; HUFFlout<="01101";
when "01100110" => HUFFdout<="11100100010000XXXX"; HUFFlout<="01110";
when "01100111" => HUFFdout<="1010001011101XXXXX"; HUFFlout<="01101";
when "01101000" => HUFFdout<="1010001001000XXXXX"; HUFFlout<="01101";
when "01101001" => HUFFdout<="11101010100110XXXX"; HUFFlout<="01110";
when "01101010" => HUFFdout<="111010101001011XXX"; HUFFlout<="01111";
when "01101011" => HUFFdout<="1110100010101XXXXX"; HUFFlout<="01101";
when "01101100" => HUFFdout<="1110101010001XXXXX"; HUFFlout<="01101";
when "01101101" => HUFFdout<="10110110010011XXXX"; HUFFlout<="01110";
when "01101110" => HUFFdout<="1010001001101XXXXX"; HUFFlout<="01101";
when "01101111" => HUFFdout<="11101010101001XXXX"; HUFFlout<="01110";
when "01110000" => HUFFdout<="1010001001010XXXXX"; HUFFlout<="01101";
when "01110001" => HUFFdout<="11101010100111XXXX"; HUFFlout<="01110";
when "01110010" => HUFFdout<="0110001000001XXXXX"; HUFFlout<="01101";
when "01110011" => HUFFdout<="10100010000011XXXX"; HUFFlout<="01110";
when "01110100" => HUFFdout<="11101010001000XXXX"; HUFFlout<="01110";
when "01110101" => HUFFdout<="11100100010001XXXX"; HUFFlout<="01110";
when "01110110" => HUFFdout<="11101010101000XXXX"; HUFFlout<="01110";
when "01110111" => HUFFdout<="1010001001011XXXXX"; HUFFlout<="01101";
when "01111000" => HUFFdout<="0110001000000XXXXX"; HUFFlout<="01101";
when "01111001" => HUFFdout<="1110101000110XXXXX"; HUFFlout<="01101";
when "01111010" => HUFFdout<="0110010011000XXXXX"; HUFFlout<="01101";
when "01111011" => HUFFdout<="11101010001001XXXX"; HUFFlout<="01110";
when "01111100" => HUFFdout<="1011010101110XXXXX"; HUFFlout<="01101";
when "01111101" => HUFFdout<="0110010011011XXXXX"; HUFFlout<="01101";
when "01111110" => HUFFdout<="1010001001100XXXXX"; HUFFlout<="01101";
```

```
when "01111111" => HUFFdout<="1110010001001XXXXX"; HUFFlout<="01101";
when "10000000" => HUFFdout<="01100100001010XXXX"; HUFFlout<="01110";
when "10000001" => HUFFdout<="1011011001100XXXXX"; HUFFlout<="01101";
when "10000010" => HUFFdout<="1010001001110010XXX"; HUFFlout<="01111";
when "10000011" => HUFFdout<="1110101010101XXXX"; HUFFlout<="01110";
when "10000100" => HUFFdout<="1110100011111110XXX"; HUFFlout<="01111";
when "10000101" => HUFFdout<="1110010001101XXXX"; HUFFlout<="01110";
when "10000110" => HUFFdout<="0110010000101 1XXXX"; HUFFlout<="01110";
when "10000111" => HUFFdout<="1010001001001XXXXX"; HUFFlout<="01101";
when "10001000" => HUFFdout<="1110100010110 0XXXX"; HUFFlout<="01110";
when "10001001" => HUFFdout<="1110101010010100XX"; HUFFlout<="10000";
when "10001010" => HUFFdout<="1110101010101 1XXXX"; HUFFlout<="01110";
when "10001011" => HUFFdout<="1110100010110 1XXXX"; HUFFlout<="01110";
when "10001100" => HUFFdout<="1010001000000XXXXX"; HUFFlout<="01101";
when "10001101" => HUFFdout<="1110101000001XXXXXX"; HUFFlout<="01100";
when "10001110" => HUFFdout<="1010001011111XXXXXX"; HUFFlout<="01100";
when "10001111" => HUFFdout<="0110001000010XXXXX"; HUFFlout<="01101";
when "10010000" => HUFFdout<="1011011001001 01XXX"; HUFFlout<="01111";
when "10010001" => HUFFdout<="0110010000100 00XXX"; HUFFlout<="01111";
when "10010010" => HUFFdout<="1110101010010 1011X"; HUFFlout<="10001";
when "10010011" => HUFFdout<="0110010000110 11000"; HUFFlout<="10010";
when "10010100" => HUFFdout<="0110010000110 11001"; HUFFlout<="10010";
when "10010101" => HUFFdout<="0110010000100 01010"; HUFFlout<="10010";
when "10010110" => HUFFdout<="1010001000001 0111X"; HUFFlout<="10001";
when "10010111" => HUFFdout<="1110010001111 0010X"; HUFFlout<="10001";
when "10011000" => HUFFdout<="0110010010100 00XXX"; HUFFlout<="01111";
when "10011001" => HUFFdout<="1110010001111 0011X"; HUFFlout<="10001";
when "10011010" => HUFFdout<="0110010000100 01000"; HUFFlout<="10010";
when "10011011" => HUFFdout<="1010001001110 110XX"; HUFFlout<="10000";
when "10011100" => HUFFdout<="1110010001111 0100X"; HUFFlout<="10001";
when "10011101" => HUFFdout<="1110010001111 0101X"; HUFFlout<="10001";
when "10011110" => HUFFdout<="0110010000110 100XX"; HUFFlout<="10000";
when "10011111" => HUFFdout<="0110010010100 010XX"; HUFFlout<="10000";
when "10100000" => HUFFdout<="0110010000100 01001"; HUFFlout<="10010";
when "10100001" => HUFFdout<="0110010000110 01100"; HUFFlout<="10010";
when "10100010" => HUFFdout<="1010001001110 111XX"; HUFFlout<="10000";
when "10100011" => HUFFdout<="0110010000110 01101"; HUFFlout<="10010";
when "10100100" => HUFFdout<="0110010010100 011XX"; HUFFlout<="10000";
when "10100101" => HUFFdout<="0110010000110 11010"; HUFFlout<="10010";
when "10100110" => HUFFdout<="0110010000110 11011"; HUFFlout<="10010";
when "10100111" => HUFFdout<="0110010000110 10100"; HUFFlout<="10010";
when "10101000" => HUFFdout<="1110010001111 110XX"; HUFFlout<="10000";
when "10101001" => HUFFdout<="0110010000011 000XX"; HUFFlout<="10000";
when "10101010" => HUFFdout<="0110010000011 001XX"; HUFFlout<="10000";
when "10101011" => HUFFdout<="0110010000110 10101"; HUFFlout<="10010";
when "10101100" => HUFFdout<="0110010000110 000XXX"; HUFFlout<="01111";
when "10101101" => HUFFdout<="0110010000011 010XX"; HUFFlout<="10000";
when "10101110" => HUFFdout<="0110010000010 10100"; HUFFlout<="10010";
when "10101111" => HUFFdout<="1010001000001 00XXX"; HUFFlout<="01111";
when "10110000" => HUFFdout<="0110010000011 011XX"; HUFFlout<="10000";
when "10110001" => HUFFdout<="0110010000010 10101"; HUFFlout<="10010";
when "10110010" => HUFFdout<="0110010000110 010XX"; HUFFlout<="10000";
when "10110011" => HUFFdout<="1010001001110 011XX"; HUFFlout<="10000";
when "10110100" => HUFFdout<="0110010000010 10110"; HUFFlout<="10010";
when "10110101" => HUFFdout<="0110010000010 10111"; HUFFlout<="10010";
when "10110110" => HUFFdout<="0110010000110 10110"; HUFFlout<="10010";
when "10110111" => HUFFdout<="0110010000110 10111"; HUFFlout<="10010";
when "10111000" => HUFFdout<="0110010000100 011XX"; HUFFlout<="10000";
when "10111001" => HUFFdout<="0110010000110 01110"; HUFFlout<="10010";
when "10111010" => HUFFdout<="1011011001001 000XX"; HUFFlout<="10000";
```

```
when "10111011" => HUFFdout<="011001000011011110"; HUFFlout<="10010";
when "10111100" => HUFFdout<="011001000011011111"; HUFFlout<="10010";
when "10111101" => HUFFdout<="011001000011101110"; HUFFlout<="10010";
when "10111110" => HUFFdout<="0110010000111010XX"; HUFFlout<="10000";
when "10111111" => HUFFdout<="011001000011101111"; HUFFlout<="10010";
when "11000000" => HUFFdout<="011001000011101100"; HUFFlout<="10010";
when "11000001" => HUFFdout<="011001000011101101"; HUFFlout<="10010";
when "11000010" => HUFFdout<="011001000001001110"; HUFFlout<="10010";
when "11000011" => HUFFdout<="011001000001001111"; HUFFlout<="10010";
when "11000100" => HUFFdout<="011001000001001000"; HUFFlout<="10010";
when "11000101" => HUFFdout<="011001000001001001"; HUFFlout<="10010";
when "11000110" => HUFFdout<="011001000001001100"; HUFFlout<="10010";
when "11000111" => HUFFdout<="011001000001001101"; HUFFlout<="10010";
when "11001000" => HUFFdout<="0110010000011100XX"; HUFFlout<="10000";
when "11001001" => HUFFdout<="0110010000011101XX"; HUFFlout<="10000";
when "11001010" => HUFFdout<="011001000010011010"; HUFFlout<="10010";
when "11001011" => HUFFdout<="011001000010011011"; HUFFlout<="10010";
when "11001100" => HUFFdout<="1110010001110110X"; HUFFlout<="10001";
when "11001101" => HUFFdout<="011001000010011000"; HUFFlout<="10010";
when "11001110" => HUFFdout<="011001000001011100"; HUFFlout<="10010";
when "11001111" => HUFFdout<="011001000001011101"; HUFFlout<="10010";
when "11010000" => HUFFdout<="011001000001001010"; HUFFlout<="10010";
when "11010001" => HUFFdout<="1110010001110111X"; HUFFlout<="10001";
when "11010010" => HUFFdout<="111001000111110XXX"; HUFFlout<="01111";
when "11010011" => HUFFdout<="011001000001001011"; HUFFlout<="10010";
when "11010100" => HUFFdout<="011001000010010100"; HUFFlout<="10010";
when "11010101" => HUFFdout<="0110010000011111XXX"; HUFFlout<="01111";
when "11010110" => HUFFdout<="0110010000001000XXX"; HUFFlout<="01111";
when "11010111" => HUFFdout<="011001000010010101"; HUFFlout<="10010";
when "11011000" => HUFFdout<="1110010001111111X"; HUFFlout<="10001";
when "11011001" => HUFFdout<="0110010000111101XX"; HUFFlout<="10000";
when "11011010" => HUFFdout<="0110010000010100XX"; HUFFlout<="10000";
when "11011011" => HUFFdout<="1010001001110000XXX"; HUFFlout<="01111";
when "11011100" => HUFFdout<="0110010000111110XX"; HUFFlout<="10000";
when "11011101" => HUFFdout<="1011011001001001XX"; HUFFlout<="10000";
when "11011110" => HUFFdout<="101000100000101000"; HUFFlout<="10010";
when "11011111" => HUFFdout<="101000100000101001"; HUFFlout<="10010";
when "11100000" => HUFFdout<="0110010000111111XX"; HUFFlout<="10000";
when "11100001" => HUFFdout<="1110010001111111X"; HUFFlout<="10001";
when "11100010" => HUFFdout<="101000100000101010"; HUFFlout<="10010";
when "11100011" => HUFFdout<="101000100000101011"; HUFFlout<="10010";
when "11100100" => HUFFdout<="1110100011111111XXX"; HUFFlout<="01111";
when "11100101" => HUFFdout<="011001000010010110"; HUFFlout<="10010";
when "11100110" => HUFFdout<="0110010000010110XX"; HUFFlout<="10000";
when "11100111" => HUFFdout<="0110010000100111XX"; HUFFlout<="10000";
when "11101000" => HUFFdout<="011001000010010111"; HUFFlout<="10010";
when "11101001" => HUFFdout<="011001000011100110"; HUFFlout<="10010";
when "11101010" => HUFFdout<="011001000011100111"; HUFFlout<="10010";
when "11101011" => HUFFdout<="011001000011100100"; HUFFlout<="10010";
when "11101100" => HUFFdout<="0110010000100100XX"; HUFFlout<="10000";
when "11101101" => HUFFdout<="011001000011100101"; HUFFlout<="10010";
when "11101110" => HUFFdout<="011001000010011001"; HUFFlout<="10010";
when "11101111" => HUFFdout<="0110010000111000XX"; HUFFlout<="10000";
when "11110000" => HUFFdout<="011001000011110010"; HUFFlout<="10010";
when "11110001" => HUFFdout<="1110010001111000XX"; HUFFlout<="10000";
when "11110010" => HUFFdout<="011001000011110011"; HUFFlout<="10010";
when "11110011" => HUFFdout<="101000100000101100"; HUFFlout<="10010";
when "11110100" => HUFFdout<="101000100000101101"; HUFFlout<="10010";
when "11110101" => HUFFdout<="011001000011110000"; HUFFlout<="10010";
when "11110110" => HUFFdout<="011001000011001111"; HUFFlout<="10010";
```

```
when "11110111" => HUFFdout<="011001000011110001"; HUFFlout<="10010";
when "11111000" => HUFFdout<="011001000001011110"; HUFFlout<="10010";
when "11111001" => HUFFdout<="1010001001110010XX"; HUFFlout<="10000";
when "11111010" => HUFFdout<="011001000001011111"; HUFFlout<="10010";
when "11111011" => HUFFdout<="011001000011011100"; HUFFlout<="10010";
when "11111100" => HUFFdout<="11101010100101010X"; HUFFlout<="10001";
when "11111101" => HUFFdout<="011001000011011101"; HUFFlout<="10010";
when "11111110" => HUFFdout<="011001000010001011"; HUFFlout<="10010";
when "11111111" => HUFFdout<="000111100XXXXXXXXX"; HUFFlout<="01001";
when others => HUFFdout <="XXXXXXXXXXXXXXXXXX"; HUFFlout <="XXXXX";
end case;
end if;
end process;
end structural;
```

A.9 shifter.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity SFTR is
port(
SFTRclk   : in std_logic;
SFTRen    : in std_logic;
SFTRdatin : in std_logic_vector (17 downto 0);
SFTRlenIn : in std_logic_vector ( 4 downto 0);
SFTRout   : out std_logic_vector (31 downto 0);
SFTRoutEn : out std_logic );
end SFTR;
architecture structural of SFTR is
function comparator17(c32: std_logic_vector(4 downto 0))
return std_logic_vector is
variable ret : std_logic_vector(16 downto 0);
begin
ret(16) := c32(4) or c32(3) or c32(2) or c32(1) or c32(0);
ret(15) := c32(4) or c32(3) or c32(2) or c32(1);
ret(14) := c32(4) or c32(3) or c32(2) or (c32(1) and c32(0));
ret(13) := c32(4) or c32(3) or c32(2);
ret(12) := c32(4) or c32(3) or (c32(2) and (c32(1) or c32(0)));
ret(11) := c32(4) or c32(3) or (c32(2) and c32(1));
ret(10) := c32(4) or c32(3) or (c32(2) and c32(1) and c32(0));
ret( 9) := c32(4) or c32(3);
ret( 8) := c32(4) or (c32(3) and (c32(2) or c32(1) or c32(0)));
ret( 7) := c32(4) or (c32(3) and c32(2)) or (c32(3) and c32(1));
ret( 6) := c32(4) or (c32(3) and c32(2)) or (c32(3) and c32(1) and c32(0));
ret( 5) := c32(4) or (c32(3) and c32(2));
ret( 4) := c32(4) or (c32(3) and c32(2) and (c32(1) or c32(0)));
ret( 3) := c32(4) or (c32(3) and c32(2) and c32(1));
ret( 2) := c32(4) or (c32(3) and c32(2) and c32(1) and c32(0));
ret( 1) := c32(4);
ret( 0) := (c32(4) and c32(3)) or (c32(4) and (c32(2) or c32(1) or c32(0)));
return ret;
end function comparator17;
constant prop_delay : time := 5 ns;
subtype std32 is std_logic_vector (31 downto 0);
signal tmp : std_logic_vector(5 downto 0):="000000";
signal stage0_len : std_logic_vector(4 downto 0):="00000";
signal stage1_len : std_logic_vector(4 downto 0):="00000";
signal stage2_len : std_logic_vector(4 downto 0):="00000";
signal stage3_len : std_logic_vector(4 downto 0):="00000";
signal stage4_len : std_logic_vector(4 downto 0):="00000";
```

```
signal timeout : std_logic_vector(1 downto 0):="00";
signal write_ready1 : std_logic := '0';
signal write_ready2 : std_logic := '0';
signal write_ready3 : std_logic := '0';
signal write_ready4 : std_logic := '0';
signal write_ready5 : std_logic := '0';
signal stage1 : std32 :="00000000000000000000000000000000";
signal stage2 : std32 :="00000000000000000000000000000000";
signal stage3 : std32 :="00000000000000000000000000000000";
signal stage4 : std32 :="00000000000000000000000000000000";
signal stage5 : std32 :="00000000000000000000000000000000";
signal stage5_d: std_logic_vector(31 downto 15):="00000000000000000";
begin
tmp <= ('0' & stage0_len) + ('0' & SFTRlenIn);
SFTRoutEn <= write_ready5 and (timeout(1) or timeout(0));
run : process(SFTRclk)
variable stage5_tmp : std_logic_vector (31 downto 0);
variable mask : std_logic_vector (31 downto 15);
variable load_db : std_logic;
begin
if(rising_edge(SFTRclk)) then
f(SFTRen = '1') then
timeout <= "11" after prop_delay;
write_ready1 <= tmp(5) after prop_delay;
write_ready2 <= write_ready1 after prop_delay;
write_ready3 <= write_ready2 after prop_delay;
write_ready4 <= write_ready3 after prop_delay;
write_ready5 <= write_ready4 after prop_delay;
stage0_len <= tmp(4 downto 0) after prop_delay;
stage1_len <= stage0_len after prop_delay;
stage2_len <= stage1_len after prop_delay;
stage3_len <= stage2_len after prop_delay;
stage4_len <= stage3_len after prop_delay;
  if(stage0_len(4) = '1') then
stage1(31 downto 30) <= SFTRdatin(1 downto 0) after prop_delay;
stage1(29 downto 16) <= (others => '0') after prop_delay;
stage1(15 downto 0) <= SFTRdatin(17 downto 2) after prop_delay;
else
stage1(31 downto 14) <= SFTRdatin after prop_delay;
stage1(13 downto 0) <= (others => '0') after prop_delay;
end if;
if(stage1_len(3) = '1') then
stage2(31 downto 24) <= stage1(7 downto 0) after prop_delay;
stage2(23 downto 0) <= stage1(31 downto 8) after prop_delay;
else
stage2 <= stage1 after prop_delay;
end if;
if(stage2_len(2) = '1') then
stage3(31 downto 28) <= stage2(3 downto 0) after prop_delay;
stage3(27 downto 0) <= stage2(31 downto 4) after prop_delay;
else
stage3 <= stage2 after prop_delay;
end if;
if(stage3_len(1) = '1') then
stage4(31 downto 30) <= stage3(1 downto 0) after prop_delay;
stage4(29 downto 0) <= stage3(31 downto 2) after prop_delay;
else
stage4 <= stage3 after prop_delay;
end if;
if(stage4_len(0) = '1') then
```

```
stage5_tmp(31) := stage4(0);
stage5_tmp(30 downto 0):= stage4(31 downto 1);
else
stage5_tmp := stage4;
end if;
if( ((stage3_len(4) or stage3_len(3) or stage3_len(2) or
stage3_len(1) or stage3_len(0)) = '1') and
((stage2_len(4) and stage2_len(3) and stage2_len(2) and
stage2_len(1) and stage2_len(0)) = '0') and
 (write_ready4 = '1') ) then
load_db := '1';
else
load_db := '0';
end if;
mask := comparator17(stage3_len);
if(load_db = '1') then
stage5_d <= (mask and stage5_tmp(31 downto 15)) after prop_delay;
stage5 <= ((stage5(31 downto 15) or
(not(mask) and stage5_tmp(31 downto 15))) &
(stage5(14 downto 0) or stage5_tmp(14 downto 0))) after prop_delay;
else
stage5_d <= (others => '0') after prop_delay;
if(write_ready5 = '1') then
stage5 <= ((stage5_tmp(31 downto 15) or stage5_d(31 downto 15)) &
(stage5_tmp(14 downto 0))) after prop_delay;
else
stage5 <= ((stage5_tmp(31 downto 15) or
stage5(31 downto 15) or
stage5_d(31 downto 15)) &
(stage5(14 downto 0) or
stage5_tmp(14 downto 0))) after prop_delay;
end if;
end if;
 else
timeout(1) <= timeout(0) after prop_delay;
timeout(0) <= '0' after prop_delay;
end if;
end if;
end process run;

SFTRout( 7 downto 0) <= (stage5(24) & stage5(25) & stage5(26) & stage5(27) & stage5(28) &
stage5(29) & stage5(30) & stage5(31));
SFTRout(15 downto 8) <= (stage5(16) & stage5(17) & stage5(18) & stage5(19) & stage5(20) &
stage5(21) & stage5(22) & stage5(23));
SFTRout(23 downto 16) <= (stage5( 8) & stage5( 9) & stage5(10) & stage5(11) & stage5(12) &
stage5(13) & stage5(14) & stage5(15));
SFTRout(31 downto 24) <= (stage5(0) & stage5(1) & stage5(2) & stage5(3) & stage5(4) & stage5(5) &
stage5(6) & stage5(7));
end structural;
configuration SFTR_default of SFTR is
for structural
end for;
end SFTR_default;
```

A.10 memory.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
architecture Memory_Access of PE1_Logic_Core is
```

```
subtype std16 is std_logic_vector (15 downto 0);
component QUANT is
port (
QUANTclk  : in std_logic;
QUANTen   : in std_logic;
QUANTmax  : in std_logic_vector (15 downto 0);
QUANTmin  : in std_logic_vector (15 downto 0);
QUANTin   : in std_logic_vector (15 downto 0);
QUANTout  : out std_logic_vector ( 3 downto 0));
end component;
component RLE is
port (
RLEclk     : in std_logic;
RLEreset   : in std_logic;
RLEen      : in std_logic;
RLEflush   : in std_logic;
RLEin      : in std_logic_vector (15 downto 0);
RLEzeroth  : in std_logic_vector (15 downto 0);
RLEout     : out std_logic_vector ( 7 downto 0);
RLErunning : out std_logic;
RLEspellEnd: out std_logic);
end component;
component HUFF is
port (
HUFFclk  : in std_logic;
HUFFin   : in std_logic_vector (7 downto 0);
HUFFlout : out std_logic_vector (4 downto 0);
HUFFdout : out std_logic_vector (17 downto 0));
end component;
component SFTR is
port (
SFTRclk   : in std_logic;
SFTRen    : in std_logic;
SFTRdatin : in std_logic_vector (17 downto 0);
SFTRlenIn : in std_logic_vector ( 4 downto 0);
SFTRout   : out std_logic_vector (31 downto 0);
SFTRoutEn : out std_logic);
end component;
type MemoryStates is(
WaitforBus,
ReadBlock1MinMax_001,
ReadBlock1MinMax_011,
ReadBlock1MinMax_111, -- got block1 min/max
ReadBlock2MinMax_111, -- got block2 min/max
ReadBlock3MinMax_111, -- got block3 min/max
ReadBlock4MinMax_111, -- got block4 min/max
ReadBlock5MinMax_111, -- got block5 min/max
ReadBlock6MinMax_110, -- got block6 min/max
ReadBlock7MinMax_100, -- got block7 min/max
ReadBlockData_001,
ReadBlockData_010,
ReadBlockData_100,
WriteData,
WriteDataCount,
WriteBlock12,
WriteBlock34,
WriteBlock56,
WriteBlock7,
MemInterrupt,
MemDone
```

```
);
signal Mem_PState : MemoryStates;
signal Mem_NState : MemoryStates;
 signal ReadCntrROW : std_logic_vector(8 downto 0);
signal ReadCntrCOL : std_logic_vector(7 downto 0);
signal eReadCntrROW : std_logic_vector(8 downto 0);
signal eReadCntrCOL : std_logic_vector(7 downto 0);
signal ROW_limit : std_logic_vector(8 downto 0);
signal COL_limit : std_logic_vector(7 downto 0);
signal ROW_skip : std_logic_vector(8 downto 0);
signal COL_skip : std_logic_vector(7 downto 0);
signal ladj : std_logic_vector(6 downto 0);
signal WriteCntr : std_logic_vector(16 downto 0);
signal RLE_Count1 : std_logic_vector(15 downto 0);
signal RLE_Count2 : std_logic_vector(15 downto 0);
signal RLE_Count3 : std_logic_vector(15 downto 0);
signal RLE_Count4 : std_logic_vector(15 downto 0);
signal RLE_Count5 : std_logic_vector(15 downto 0);
signal RLE_Count6 : std_logic_vector(15 downto 0);
signal RLE_Count7 : std_logic_vector(15 downto 0);
signal Block1Min : std16;
signal Block1Max : std16;
signal Block2Min : std16;
signal Block2Max : std16;
signal Block3Min : std16;
signal Block3Max : std16;
signal Block4Min : std16;
signal Block4Max : std16;
signal Block5Min : std16;
signal Block5Max : std16;
signal Block6Min : std16;
signal Block6Max : std16;
signal Block7Min : std16;
signal Block7Max : std16;
constant Block1Th : std16 := "0000000000000000"; -- 0 x 2
constant Block3Th : std16 := "0000000000110110"; -- 27 x 2
constant Block2Th : std16 := "0000000001001110"; -- 39 x 2
constant Block4Th : std16 := "0000000011010000"; -- 104 x 2
constant Block6Th : std16 := "0000000001100100"; -- 50 x 2
constant Block5Th : std16 := "0000000010011110"; -- 79 x 2
constant Block7Th : std16 := "0000000101111110"; -- 191 x 2
signal QUANTen : std_logic;
signal QUANTmax : std_logic_vector(15 downto 0);
signal QUANTmin : std_logic_vector(15 downto 0);
signal QUANTin : std_logic_vector(15 downto 0);
signal QUANTin2 : std_logic_vector(15 downto 0);
signal QUANTout : std_logic_vector( 3 downto 0);
signal QUANTout2 : std_logic_vector( 3 downto 0);
signal RLEflush : std_logic;
signal RLEen : std_logic;
signal RLEin : std_logic_vector(15 downto 0);
signal RLEzeroth : std_logic_vector(15 downto 0);
signal RLEout : std_logic_vector( 7 downto 0);
signal RLErunning : std_logic;
signal RLEspellEnd : std_logic;
signal RLErunning1 : std_logic;
signal RLEspellEnd1 : std_logic;
signal RLErunning2 : std_logic;
signal RLEspellEnd2 : std_logic;
signal HUFFin : std_logic_vector( 7 downto 0);
```

```
signal HUFFlout : std_logic_vector( 4 downto 0);
signal HUFFdout : std_logic_vector(17 downto 0);
signal SFTRen : std_logic;
signal SFTRdatin : std_logic_vector(17 downto 0);
signal SFTRlenIn : std_logic_vector( 4 downto 0);
signal SFTRout : std_logic_vector(31 downto 0);
signal SFTRoutEn : std_logic;
signal readComplete : std_logic;
signal nStages : std_logic_vector(2 downto 0);
signal nStages1 : std_logic_vector(2 downto 0);
signal nStages_1 : std_logic_vector(2 downto 0);
signal nStages_2 : std_logic_vector(2 downto 0);
signal nStages_3 : std_logic_vector(2 downto 0);
begin
quantizer : QUANT
PE_Pclk,
QUANTen,
QUANTmax,
QUANTmin,
QUANTin,
QUANTout);
rle : RLE
port map (
PE_Pclk,
PE_Reset,
RLEen,
RLEflush,
RLEin,
RLEzeroth,
RLEout,
RLErunning,
RLEspellEnd);
huffman : HUFF
port map (
PE_Pclk,
HUFFin,
HUFFlout,
HUFFdout);
bitpacker : SFTR
port map (
PE_Pclk,
SFTRen,
SFTRdatin,
SFTRlenIn,
SFTRout,
SFTRoutEn);
quantizer_in : process(Mem_PState, PE_MemData_InReg, QUANTin2)
begin
if (Mem_PState = ReadBlockData_100) then
QUANTin <= PE_MemData_InReg(31 downto 16);
else
QUANTin <= QUANTin2;
end if;
end process quantizer_in;
RLEin <= QUANTin;
RLEen <= QUANTen;
with RLErunning select HUFFin <= -- Input to huffman:
RLEout when '1', -- from RLE, when RLE
("0000" & QUANTout) when others; -- from QUANT, else
SFTRdatin <= HUFFdout;
```

```
SFTRlenIn <= HUFFlout;
with nStages1 select QUANTmax <=
Block1Max when "000",
Block2Max when "001",
Block3Max when "010",
Block4Max when "011",
Block5Max when "101",
Block6Max when "110",
Block7Max when "111",
(others => 'X') when others;
with nStages1 select QUANTmin <=
Block1Min when "000",
Block2Min when "001",
Block3Min when "010",
Block4Min when "011",
Block5Min when "101",
Block6Min when "110",
Block7Min when "111",
(others => 'X') when others;
with nStages1 select RLEzeroth <=
Block1Th when "000",
Block2Th when "001",
Block3Th when "010",
Block4Th when "011",
Block5Th when "101",
Block6Th when "110",
Block7Th when "111",
(others => '0') when others;
st_update : process (PE_Pclk, PE_Reset)
begin
if (PE_Reset = '1') then
Mem_PState <= WaitforBus;
readComplete <= '0';
nStages <= "000";
nStages1 <= "000";
nStages_1 <= "100";
nStages_2 <= "100";
nStages_3 <= "100";
QUANTin2 <= (others => '0');
QUANTout2 <= (others => '0');
ReadCntrROW <= "000000000";
ReadCntrCOL <= "00000000";
eReadCntrROW <= "000000000";
eReadCntrCOL <= "00000000";
ladj <= (others => '0');
RLErunning1 <= '0';
RLEspellEnd1 <= '0';
RLErunning2 <= '0';
RLEspellEnd2 <= '0';
ROW_limit <= "111111000";
COL_limit <= "11111000";
ROW_skip <= "000001000";
COL_skip <= "00001000";
WriteCntr <= "0000000000000000";
RLE_Count1 <= "0000000000000000";
RLE_Count2 <= "0000000000000000";
RLE_Count3 <= "0000000000000000";
RLE_Count4 <= "0000000000000000";
RLE_Count5 <= "0000000000000000";
RLE_Count6 <= "0000000000000000";
```

```
RLE_Count7 <= "0000000000000000";
Block1Min <= (others => '0');
Block1Max <= (others => '0');
Block2Min <= (others => '0');
Block2Max <= (others => '0');
Block3Min <= (others => '0');
Block3Max <= (others => '0');
Block4Min <= (others => '0');
Block4Max <= (others => '0');
Block5Min <= (others => '0');
Block5Max <= (others => '0');
Block6Min <= (others => '0');
Block6Max <= (others => '0');
Block7Min <= (others => '0');
Block7Max <= (others => '0');
elsif (rising_edge(PE_Pclk)) then
Mem_PState <= Mem_NState;
nStages1 <= nStages;
RLErunning1 <= RLErunning;
RLEspellEnd1 <= RLEspellEnd;
RLErunning2 <= RLErunning1;
RLEspellEnd2 <= RLEspellEnd1;
if (Mem_PState = ReadBlock1MinMax_111) then
Block1Max <= PE_MemData_InReg(31 downto 16);
Block1Min <= PE_MemData_InReg(15 downto 0);
end if;
if (Mem_PState = ReadBlock2MinMax_111) then
Block2Max <= PE_MemData_InReg(31 downto 16);
Block2Min <= PE_MemData_InReg(15 downto 0);
end if;
if (Mem_PState = ReadBlock3MinMax_111) then
Block3Max <= PE_MemData_InReg(31 downto 16);
Block3Min <= PE_MemData_InReg(15 downto 0);
end if;
if (Mem_PState = ReadBlock4MinMax_111) then
Block4Max <= PE_MemData_InReg(31 downto 16);
Block4Min <= PE_MemData_InReg(15 downto 0);
end if;
if (Mem_PState = ReadBlock5MinMax_111) then
Block5Max <= PE_MemData_InReg(31 downto 16);
Block5Min <= PE_MemData_InReg(15 downto 0);
end if;
if (Mem_PState = ReadBlock6MinMax_110) then
Block6Max <= PE_MemData_InReg(31 downto 16);
Block6Min <= PE_MemData_InReg(15 downto 0);
end if;
if (Mem_PState = ReadBlock7MinMax_100) then
Block7Max <= PE_MemData_InReg(31 downto 16);
Block7Min <= PE_MemData_InReg(15 downto 0);
end if;
if (Mem_PState = ReadBlockData_100) then
QUANTin2 <= PE_MemData_InReg(15 downto 0);
QUANTout2 <= QUANTout; -- DEBUG
end if;
if (Mem_PState = ReadBlockData_001) then
ladj(6) <= not(readComplete);
ladj(5) <= ladj(6);
ladj(4) <= ladj(5);
ladj(3) <= ladj(4);
ladj(2) <= ladj(3);
```

```
ladj(1) <= ladj(2);
ladj(0) <= ladj(1);
end if;
if (((Mem_PState = WriteData) or (Mem_PState = ReadBlockData_010)) and
(SFTRoutEn = '1')) then
WriteCntr <= WriteCntr + 1;
end if;
if (((Mem_PState = WriteData) or (Mem_PState = ReadBlockData_010)) and
((RLErunning = '0') or (RLEspellEnd = '1'))) then
if(nStages_3="000") then
RLE_Count1 <= RLE_Count1 + 1;
end if;
if(nStages_3="001") then
RLE_Count2 <= RLE_Count2 + 1;
end if;
if(nStages_3="010") then
RLE_Count3 <= RLE_Count3 + 1;
end if;
if(nStages_3="011") then
RLE_Count4 <= RLE_Count4 + 1;
end if;
if(nStages_3="101") then
RLE_Count5 <= RLE_Count5 + 1;
end if;
if(nStages_3="110") then
RLE_Count6 <= RLE_Count6 + 1;
end if;
if(nStages_3="111") then
RLE_Count7 <= RLE_Count7 + 1;
end if;
end if;

if(nStages(1) = '1') then
eReadCntrCOL <= ReadCntrCOL + ('0' & COL_skip(7 downto 1));
else
eReadCntrCOL <= ReadCntrCOL;
end if;
if(nStages(0) = '1') then
eReadCntrROW <= ReadCntrROW + ('0' & ROW_skip(8 downto 1));
else
eReadCntrROW <= ReadCntrROW;
end if;
if (Mem_PState = ReadBlockData_100) then
nStages_1 <= nStages;
nStages_2 <= nStages_1;
nStages_3 <= nStages_2;
ReadCntrCOL <= ReadCntrCOL + COL_skip;
if (ReadCntrCOL = COL_limit) then
ReadCntrROW <= ReadCntrROW + ROW_skip;
end if;
if((ReadCntrROW = ROW_limit) and -- End of current
(ReadCntrCOL = COL_limit)) then -- block
if (nStages = "011") then
nStages <= "101";
elsif (nStages = "111") then
nStages <= "100";
else
nStages <= nStages + 1;
end if;
if (nStages(1 downto 0) = "11") then
```

```
ROW_skip <= (’0’ & ROW_skip(8 downto 1));
COL_skip <= (’0’ & COL_skip(7 downto 1));
ROW_limit <= (’1’ & ROW_limit(8 downto 1));
COL_limit <= (’1’ & COL_limit(7 downto 1));
end if;
if (nStages = "111") then
readComplete <= ’1’;
end if;
end if;
end if;
end if;

end process st_update;
PE_MemAddr_OutReg(21 downto 18) <= (others => ’0’);
mem_state: process(Mem_PState,ladj,PE_MemBusGrant_n,eReadCntrROW,eReadCntrCOL,
WriteCntr,nStages, nStages1,RLE_Count1, RLE_Count2, RLE_Count3, RLE_Count4,
RLE_Count5, RLE_Count6, RLE_Count7,RLErunning2, RLEspellEnd2,SFTRoutEn,
SFTRout,PE_InterruptAck_n)
Begin
PE_InterruptReq_n <= ’1’;
PE_MemWriteSel_n <= ’1’;
PE_MemStrobe_n <= ’1’;
PE_MemBusReq_n <= ’0’;
QUANTen <= ’0’;
SFTRen <= ’0’;
RLEflush <= ’0’;
PE_MemAddr_OutReg(17 downto 0) <= (others => ’0’);
PE_MemData_OutReg(31 downto 0) <= (others => ’0’);
case Mem_PState is
when WaitforBus =>if(PE_MemBusGrant_n = ’0’) then
Mem_NState <= ReadBlock1MinMax_001;
else
Mem_NState <= WaitforBus;
end if;
when ReadBlock1MinMax_001 =>PE_MemStrobe_n <= ’0’;
Mem_NState <= ReadBlock1MinMax_011;
PE_MemAddr_OutReg(17 downto 0) <= "10000000000001000";
when ReadBlock1MinMax_011 =>PE_MemStrobe_n <= ’0’;
Mem_NState <= ReadBlock1MinMax_111;
PE_MemAddr_OutReg(17 downto 0) <= "10000000000001001";
when ReadBlock1MinMax_111 =>PE_MemStrobe_n <= ’0’;
Mem_NState <= ReadBlock2MinMax_111;
PE_MemAddr_OutReg(17 downto 0) <= "10000000000001010";
when ReadBlock2MinMax_111 =>PE_MemStrobe_n <= ’0’;
Mem_NState <= ReadBlock3MinMax_111;
PE_MemAddr_OutReg(17 downto 0) <= "10000000000001011";
when ReadBlock3MinMax_111 =>PE_MemStrobe_n <= ’0’;
Mem_NState <= ReadBlock4MinMax_111;
PE_MemAddr_OutReg(17 downto 0) <= "10000000000000101";
when ReadBlock4MinMax_111 =>PE_MemStrobe_n <= ’0’;
Mem_NState <= ReadBlock5MinMax_111;
PE_MemAddr_OutReg(17 downto 0) <= "10000000000000110";
when ReadBlock5MinMax_111 =>PE_MemStrobe_n <= ’0’;
Mem_NState <= ReadBlock6MinMax_110;
PE_MemAddr_OutReg(17 downto 0) <= "10000000000000111";
when ReadBlock6MinMax_110 =>Mem_NState <= ReadBlock7MinMax_100;
when ReadBlock7MinMax_100 =>Mem_NState <= ReadBlockData_001;
when ReadBlockData_001 =>PE_MemStrobe_n <= ’0’;
Mem_NState <= ReadBlockData_010;
PE_MemAddr_OutReg(17) <= ’0’;
```

```
PE_MemAddr_OutReg(16 downto 8) <= eReadCntrROW;
PE_MemAddr_OutReg( 7 downto 0) <= eReadCntrCOL;
when ReadBlockData_010 =>Mem_NState <= ReadBlockData_100;
PE_MemWriteSel_n <= '0';
SFTRen <= ((ladj(3) or ladj(0)) and (not(RLErunning2) or RLEspellEnd2));
PE_MemStrobe_n <= not(SFTRoutEn);
PE_MemData_OutReg(31 downto 0) <= SFTRout;
PE_MemAddr_OutReg(17) <= '1';
PE_MemAddr_OutReg(16 downto 0) <= WriteCntr;
when ReadBlockData_100 =>
Mem_NState <= WriteData;
QUANTen <= '1';
when WriteData =>PE_MemWriteSel_n <= '0'; -- for writing
PE_MemStrobe_n <= not(SFTRoutEn);
QUANTen <= '1';
SFTRen <= ((ladj(3) or ladj(0)) and (not(RLErunning2) or RLEspellEnd2));
if(nStages /= nStages1) then
LEflush <= '1';
end if;
PE_MemAddr_OutReg(17) <= '1';
PE_MemAddr_OutReg(16 downto 0) <= WriteCntr;
if ((ladj(6) = '0') and (ladj(0) = '0')) then
Mem_NState <= WriteDataCount;
else
Mem_NState <= ReadBlockData_001;
end if;
PE_MemData_OutReg(31 downto 0) <= SFTRout;
when WriteDataCount =>
PE_MemWriteSel_n <= '0'; -- for writing
PE_MemStrobe_n <= '0';
Mem_NState <= WriteBlock12;
PE_MemData_OutReg(31 downto 17) <= (others => '0');
PE_MemData_OutReg(16 downto 0) <= WriteCntr;
PE_MemAddr_OutReg(17 downto 0) <= "000000000000000000";
when WriteBlock12 =>PE_MemWriteSel_n <= '0'; -- for writing
PE_MemStrobe_n <= '0';
Mem_NState <= WriteBlock34;
PE_MemData_OutReg(31 downto 16) <= RLE_Count1;
PE_MemData_OutReg(15 downto 0) <= RLE_Count2;
PE_MemAddr_OutReg(17 downto 0) <= "000000000000000001";
when WriteBlock34 =>PE_MemWriteSel_n <= '0';
PE_MemStrobe_n <= '0';
Mem_NState <= WriteBlock56;
PE_MemData_OutReg(31 downto 16) <= RLE_Count3;
PE_MemData_OutReg(15 downto 0) <= RLE_Count4;
PE_MemAddr_OutReg(17 downto 0) <= "000000000000000010";
when WriteBlock56 =>PE_MemWriteSel_n <= '0'; -- for writing
PE_MemStrobe_n <= '0';
Mem_NState <= WriteBlock7;
PE_MemData_OutReg(31 downto 16) <= RLE_Count5;
PE_MemData_OutReg(15 downto 0) <= RLE_Count6;
PE_MemAddr_OutReg(17 downto 0) <= "000000000000000011";
when WriteBlock7 =>PE_MemWriteSel_n <= '0'; -- for writing
PE_MemStrobe_n <= '0';
Mem_NState <= MemInterrupt;
PE_MemData_OutReg(31 downto 16) <= "0000000000000000";
PE_MemData_OutReg(15 downto 0) <= RLE_Count7;
PE_MemAddr_OutReg(17 downto 0) <= "000000000000000100";
when MemInterrupt =>PE_MemBusReq_n <= '1'; -- Give up bus
PE_InterruptReq_n <= '0'; -- Interrupt host
```

```
if(PE_InterruptAck_n = ’0’) then
Mem_NState <= MemDone;
else
Mem_NState <= MemInterrupt;
end if;
when MemDone =>PE_MemBusReq_n <= ’1’;
Mem_NState <= MemDone;
end case;
end process mem_state;
PE_MemHoldReq_n <= ’1’;
PE_Left_OE <= ( others => ’0’ );
PE_Right_OE <= ( others => ’0’ );
PE_FifoSelect <= "00"; -- Deselect fifo
PE_Fifo_WE_n <= ’1’;
PE_FifoPtrIncr_EN <= ’0’;
end Memory_Access;
```

REFERENCES

1. G. Fernandez, S. Periaswamy, and W. Sweldens,"LIFTPACK: A software package for wavelet transforms using lifting". Wavelet Applications in Signal and Image Processing IV, Proc. SPIE 2825, 1996, http://www.cse.sc.edu/~fernande/liftpack.

2. "JPEG2000 Image Coding System", JPEG 2000 final committee draft version 1.0, March 2000 (available from http://www.jpeg.org/public/fcd15444-1.pdf).

3. SIAM, J.Math. Anal "The lifting scheme: A construction of second generation wavelets", vol. 29, no. 2, pp. 511–546, March 1998.

4 Nazeeh Aranki, Wenqing Jiang Antonio Ortega, "FPGA-Based Parallel Implementation for the Lifting Discrete Wavelet Transform", Jet propulsion Laboratory.

5. R. Mateos, **A.** Gardel, **A.** Hernandez, I. Bravo, C. Garcia, **"Lossless Implementation in VHDL of an Image Wavelet Transform",** IEEE JOURNAL 2003.

6. G. Dimitroulakos , N. D. Zervas, N. Sklavos and C.E Goutis, **"**An Efficient Vlsi Implementation For Forward And Inverse Wavelet Transform For Jpeg2000", Proceedings of 14th IEEE International Conference on Digital Signal Processing (DSP'02),Greece, July 1-3, 2002.

7. http://www.jpeg.org.

8. http://www.wavelet.org.

9. Kishore Andra, Chaitali Chakrabarti, Tinku Acharya, "Efficient Implementation Of A Set Of Lifting Based Wavelet Filters", Intel Corporation, Chandler, Arizona, USA.

10. Marco Grangetto, Enrico Magli, Maurizio Martina, and Gabriella Olmo, "Optimization and Implementation of the Integer Wavelet Transform for Image Coding", IEEE Transactions On Image Processing, vol. 11, no. 6, June 2002.

11. C. Valens, 1999-2004. **"**The Fast Lifting Wavelet Transform"**.**

12. Martin Vetterli and Cormac Herley, "Wavelets and filter banks theory and design", IEEE Transactions On Image Processing, vol. 40, no. 9, September 1992.

13 VHDL Primer by J Bhaskar.

14 Data compression by David Salomon, Springer Publishers

15 Mike Goldsmith, VHDL Tutorial [Online] available:
http://www.asic.uwaterloo.ca/groups/digital/mgoldsmith/VHDL_Tutorial_1.pdf

16 "VHDL Tutorial," [Online] available:
http://www.vhdlonline.de/tutorial/englisch/t_219.htm