1999

# Embed[d]ed Zerotree Codec

Karma Wangdi
*Edith Cowan University*

# Edith Cowan University

# Copyright Warning

# Edith Cowan University

Library/Archives

[redacted]

*I certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any institution of higher education and that to the best of my knowledge and belief does not contain any material previously published or written by another person except where due reference has been made in the text.*

Signature [redacted]

Date 28/2/2000

# USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

# Embeded Zerotree Codec

A Thesis Submitted in Partial Fulfilment of the Requirements for the Award of Bachelor of Engineering (Electronic Systems) with First Class Honours

Karma Wangdi

Faculty of Communications, Heath and Science
Edith Cowan University
Western Australia

Date of submission: 9 November 1999

## TABLE OF CONTENTS

# FIGURES AND TABLES

## List of Figures

## List of Tables

## ACKNOWLEDGEMENT

Project Supervisor: Dr.Ganesh Kothapalli

Project Examiners: Professor A.Bouzerdoum
Dr. Ganesh Kothapalli

I would like to express my sincere gratitude to my supervisor, Dr. Ganesh Kothapalli for his continued advice and guidance throughout the course of this project. He spent a considerable amount of time out of his busy schedule to ensure the success of my project. I would also like to thank Dr.Stefan Lachowicz who supervised me during the first half of my project. He provided me with abundant amount of literature related to the project and also provided helpful guidance.

I am also very thankful to Mr. Kenneth Ang and Geoffrey Alagoda, Ph.D. students in the engineering department at Edith Cowan University. Kenneth not only provided me with valuable consultation on the EZW algorithm, he also lent me a lot of recent books on the topic and gave advice on how to improve my thesis. Geoff also provided me with very useful consultation.

My thanks are also due to Mr. Richard Geissler, at the University of Ulm, Germany, who kindly and promptly clarified the many doubts I had onVHDL.

Lastly, I would like to thank my family and friends who have inspired me and given me unfaltering support throughout my education career.

**Karma Wangdi**

## ABSTRACT

This thesis discusses the findings of the final year project involving the VHDL (V= Very High Speed Integrated Circuit, Hardware Description Language) design and simulation of an EZT (Embedded Zero Tree) codec.

The basis of image compression and the various image compression techniques that are available today have been explored. This provided a clear understanding of image compression as a whole. An in depth understanding of wavelet transform theory was vital to the understanding of the edge that this transform provides over other transforms for image compression. Both the mathematics of it and how it is implemented using sets of high pass and low pass filters have been studied and presented.

At the heart of the EZT codec is the EZW (Embedded Zerotree Wavelet) algorithm, as this is the algorithm that has been implemented in the codec. This required a thorough study and understanding of the algorithm and the various terms used in it.

A generic single processor codec capable of handling any size of zerotree coefficients of images was designed. Once the coding and decoding strategy of this single processor had been figured out, it was easily extended to a codec with three parallel processors. This parallel architecture uses the same coding and decoding methods as in the single processor except that each processor in the parallel processing now handles only a third of the coefficients, thus promising a much speedier codec as compared to the first one.

Both designs were then translated into VHDL behavioral level codes. The codes were then simulated and the results were verified.

Once the simulations were completed the next aim for the project, namely synthesizing the design, was embarked upon. Of the two logical parts of the encoder, only the significance map generator has been synthesized.

# INTRODUCTION

Digital images play a crucial role of disseminating rich information in today's Information Age. Of the various types of data transferred over our networks, notably the Internet, image comprises the bulk of the traffic. Current estimates indicate that image data transfer take up over 90% of the volume on the Internet.

As ubiquitous and as informative as they are, digital images are also the most data intensive, requiring huge storage space and longer transmission and access time.

In order to utilize these digital images there are clearly needs for effective image compression techniques to reduce the number of bits required to represent them. A wide range of compression techniques has been developed over the years, and novel approaches continue to emerge.

The use of wavelet transform in image compression has captured the imagination and the talent of researchers all over the world in recent times. This wavelet image compression technique promises performance improvements over all the compression methods currently available. So promising is this technique of image compression that wavelet image coders are among the leading coders submitted for consideration in the upcoming JPEG200 standard, which will replace the current JPEG standard for image compression.

The most revolutionary thing about wavelet transform is that when applied to a digital image it executes a multiresolution analysis on the image. In other words, wavelet transform essentially processes the image in much the same manner as the human visual system.

The EZW (embedded zerotree wavelet) algorithm is one image compression algorithm based on this new technique. It is claimed to be a remarkably effective image compression algorithm. This algorithm has the property that bits in a bitstream are generated in the order of importance, yielding a fully embedded code. This property then allows the encoder to terminate the encoding at any point, thereby allowing a bit rate or a distortion rate to be met exactly. Also given a bitstream the decoder can

terminate decoding at any point and still produce the same image that would have been encoded at the bit rate corresponding to the truncated bit stream.

From EZW algorithm two codec (encoder and decoder) architectures have evolved in this project. The single processor architecture is the main architecture in the sense that the other is just a slight adaptation of the single processor to take advantage of the inherent parallelism present in the zerotree data structure.

The VHDL codes for the designs have been written at a behavioral level. Simulation and synthesis tools used were from Synopsys Inc.

# PROJECT DEFINITION

## Aim

The main aim of the project was to design, simulate and if possible, synthesize, an EZT codec implementing the EZW algorithm. The objectives that this main aim of the project translated to were:

- An understanding of image compression in general and that of wavelet image compression in particular.
- A study of wavelet transform and how it relates to image compression.
- A thorough understanding of the EZW algorithm and the various terms used in it.
- A good mastery of the VHDL language
- Design and simulation of a codec using VHDL behavioral level code.
- Synthesis of the codec.

## Scope

The project has both a research component and a VLSI implementation component in the form of VHDL implementation.

- The first task was to understand image compression as a whole and the various features of images that have been exploited to achieve image compression. Models of image compression were also studied.
- A good grasp of wavelet transform theory was paramount in understanding the EZW algorithm, which is at the heart of the project. How this transform relates to image compression was then investigated.

- Of outmost importance was the thorough understanding of the EZW algorithm itself. IEEE transactions were a major source. Consultation with a PhD student working on a similar but advanced topic proved very helpful and productive.

- Since the codec was to be implemented in VHDL, a good knowledge of the language was crucial as well. It was learnt in tandem with the reading of back ground materials.

- VHDL tools used for simulation and synthesis were from Synopsys Inc. PeakVHDL was also used as an alternative simulation tool in the course of the project.

# 1. INTRODUCTION TO IMAGE COMPRESSION

## 1.1 Introduction

In many different fields, digitized images are replacing conventional analog images as photographs or x-rays. The volume of data required to describe such images greatly slow transmission and makes storage prohibitively costly. Image compression is the general term that addresses the problem of reducing the amount of data required to represent a digital image. The following paragraphs present some principles and techniques of image compression that are currently being used.

## 1.2 Basis of Image Compression

Image compression is based on two features of data; namely data redundancy and data irrelevancy. Hence the great variety of compression algorithms mainly differ in their approaches to extracting and exploiting these two features of data redundancy and irrelevancy. (Topiwala, 1998)

### 1.2.1 Data Redundancy

The term data compression refers to the process of reducing the amount of data required to represent a given quantity of information. Various amounts of data may be used to represent the same amount of information. When there are more data than actually required to represent a given information the data is said to contain data redundancy.

Data redundancy is a central issue in digital image compression. Three basic data redundancies can be identified and exploited in digital image compression. They are:

- Coding redundancy
- Interpixel redundancy and
- Psychovisual redundancy

#### 1.2.1.a. Coding Redundancy

In general, coding redundancy is present when the codes assigned to a set of events have not been selected to take full advantage of probabilities of the events. It is almost always present when an image's gray levels are represented with straight or natural binary code. In this, the underlying basis for coding redundancy is that images are typically composed of objects that have a regular and somewhat predictable morphology and reflectance, and are generally sampled so that the objects being depicted are much larger than the picture elements. The natural consequence is that, in most images, certain gray levels are more probable than others. If a natural binary coding of gray levels is used, the same number of bits are assigned to both the most and the least probable values, thus resulting in coding redundancy. (Gonzalez and Woods,1993).

#### 1.2.1.b Interpixel Redundancy

Interpixel redundancy is directly related the interpixel correlations that exit within an image. Because the value of any given pixel can be reasonably predicted from the values of the its neighbor's, the information carried by the individual pixels is relatively small. In other words much of the visual contribution of a single pixel to an image is redundant. A variety of names, including spatial redundancy, geometric redundancy and interframe redundancy, have been coined to refer to these interpixel dependencies. The term Interpixel redundancy encompasses them all. (Gonzalez and Woods,1993). Interpixel redundancies are removed by using suitable transforms.

#### 1.2.1.c Psychovisual Redundancy

Psychovisual redundancy is the result of the nature of the human eye. The eye does not respond with equal sensitivity to all visual information. Certain information simply has less relative importance than other information in normal visual processing. This information is said to be psychovisually redundant. It can be eliminated without significantly impairing the quality of image perception. (Gonzalez and Woods, 1993).

### 1.2.2 Data Irrelevancy

An important example of data irrelevancy occurs in the visualization of gray scale images of high dynamic range, e.g., 12 bits or more. It is an experimental fact that for monochrome images, 6 to 8 bits of dynamic range is the limit of human visual sensitivity; any extra bits do not add perceptual value and can be eliminated. (Topiwala, 1998).

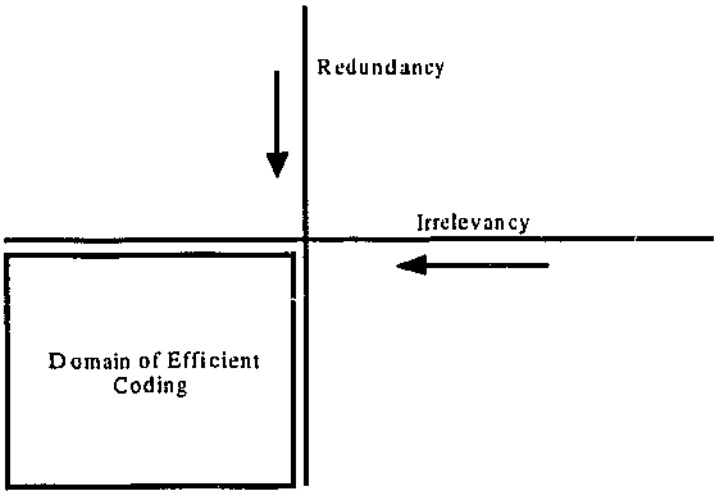



Figure 1.1 The operational space of compression algorithm design

## 1.3 Image Compression Models

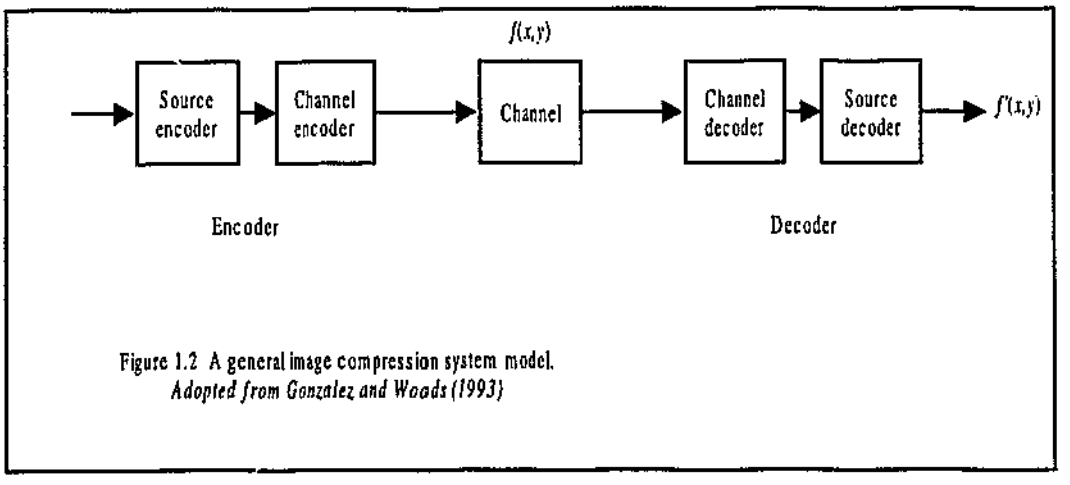



Figure 1.2 A general image compression system model.
*Adopted from Gonzalez and Woods (1993)*

As figure 1.2 shows, an image compression system consists of two distinct structural blocks: an *encoder* and a *decoder*. An input image *f(x,y)* is fed into the encoder which

creates a set of symbols from the input data. After transmission over the channel, the encoded representation is fed to the decoder, where a reconstructed output image $f'(x,y)$ is generated. In general $f'(x,y)$ may or may not be an exact replica of $f(x,y)$. If it is, the system is error free or information preserving; if not some level of distortion is present in the reconstructed image.

Both the encoder and the decoder shown in figure 1.2 consist of two relatively independent functions or subblocks. The encoder is made up of a *source encoder* which removes input redundancies, and a *channel encoder* which increases the noise immunity of the source encoder's output. As would be expected, the decoder includes a *channel decoder* followed by a *source decoder.* If the channel between the encoder and the decoder is noise free (not prone to error), the channel encoder and decoder are omitted, and the general encoder and decoder become the source encoder and decoder, respectively.

### 1.3.1 The Source Encoder and Decoder

The source encoder is responsible for reducing or eliminating any coding, interpixel, or pyschovisual redundancies in the input image. The specific application and associated

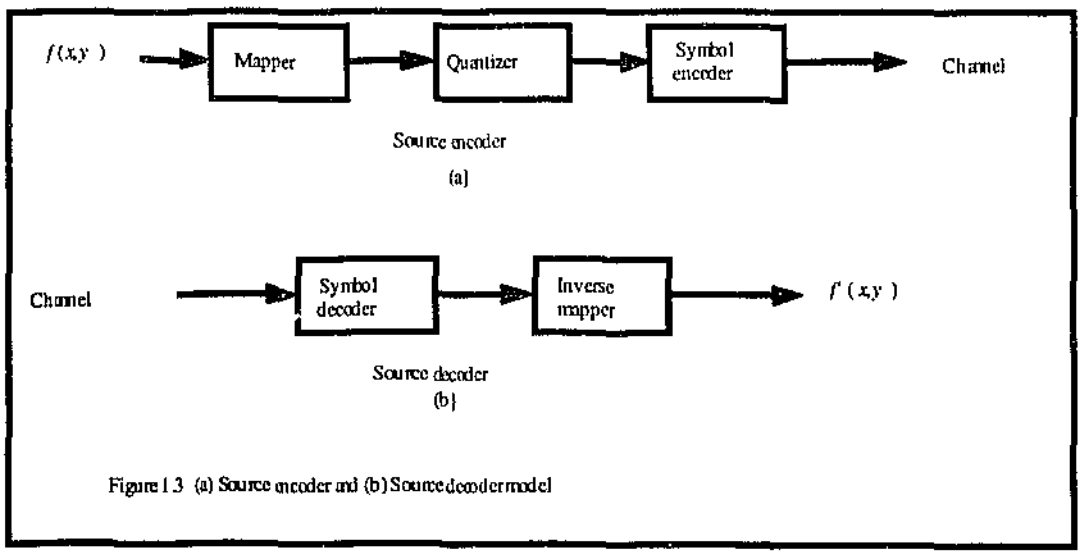



Figure 1.3 (a) Source encoder and (b) Source decoder model

fidelity requirements dictate the best encoding approach to use in any given situation. Normally, the approach can be modeled by a series of three independent operations.

Figure 1.3(a) shows how each operation is designed to reduce one of the three redundancies mentioned in section 1.2. Figure 1.3(b) depicts the corresponding source decoder.

In the first stage of the source encoding, process the *mapper* transforms the input data into a (usually nonvisual) format designed to reduce interpixel redundancies in the input image. This operation is generally reversible and may or may not reduce directly the amount of data required to represent the image. Run length coding which will be explained in a later section is an example of a mapping that directly results in data compression in this initial stage of the overall source encoding process. The representation of an image by a set of transform coefficients is an example of the opposite case. Here the major mapper transforms the image into an array of coefficients, making its interpixel redundancies more accessible for compression in later stages of the encoding process.

The second stage, or *quantizer* block in figure 1.2(a), reduces the accuracy of the mapper's output in accordance with some pre-established fidelity criteria. This stage reduces the pyschovisual redundancies of the input image. This operation is irreversible. Thus it must be omitted when error-free compression is desired. This block is what distinguishes between a lossy and a lossless compression. In the third and the final stage of the source encoding process, the *symbol coder* creates a fixed or variable length code to represent the quantizer output and maps the output in accordance with the code. The term symbol coder distinguishes this operation from the overall source encoding process. In most cases, a variable length code is used to represent the mapped and quantized data set. It assigns the shortest code words to the most frequently occurring output values and thus reduces coding redundancy. This operation is reversible. Upon completion of the symbol-coding step, the input image has been processed to remove each of the three redundancies.

Figure 1.3 shows the source encoding process as three successive operations, but all three operations are not necessarily included in every compression system. For

example the quantizer must be omitted when error-free compression is desired. In addition, some compression techniques normally are modeled by merging blocks that are physically separate in Figure 1.2(a). For instance, in predictive compression systems, a topic discussed in a later section, the mapper and the quantizer are often represented by a single block that simultaneously performs both operations.

## 1.4 Types of Image Compression Techniques

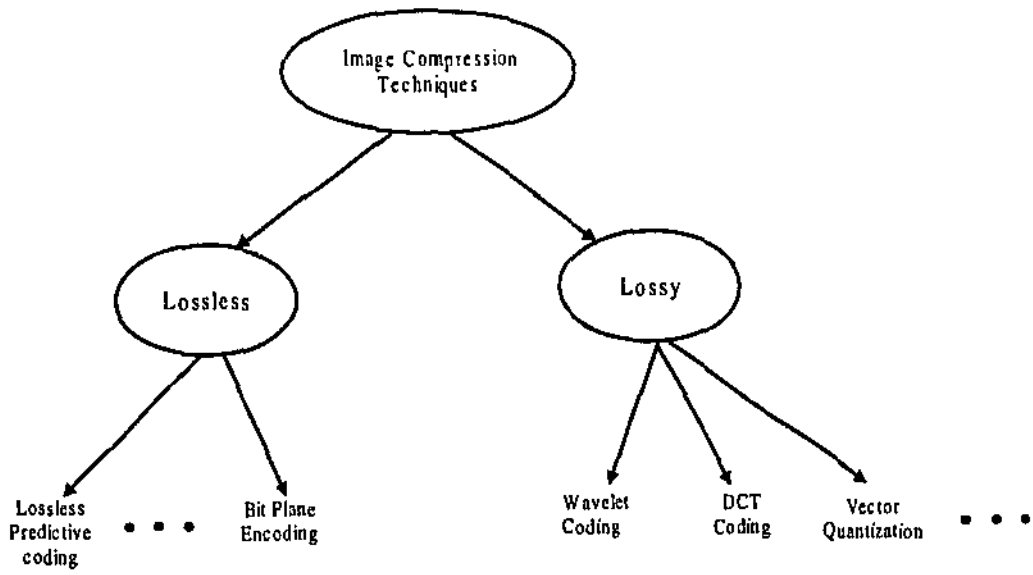



figure 1.4 Image Compression techniques

There are many different approaches to image compression, but they can all be categorized into tw_ fundamental groups: lossy compression techniques and lossless compression techniques.

In lossless compression (also known as bit-preserving or reversible compression), the reconstructed image after compression is numerically identical to the original image on a pixel-by-pixel basis. Since no information is compromised only a modest amount of compression is achieved. In other words the compression ratio (CR) is small. The compression ratio is defined as:

*CR = No of bits for original image / No of bits for compressed image*

In lossy compression (also known as irreversible compression), the reconstructed image contains degradations relative to the original. As a result much higher compression can be achieved as compared to the lossless compression. The compression ratio is high.

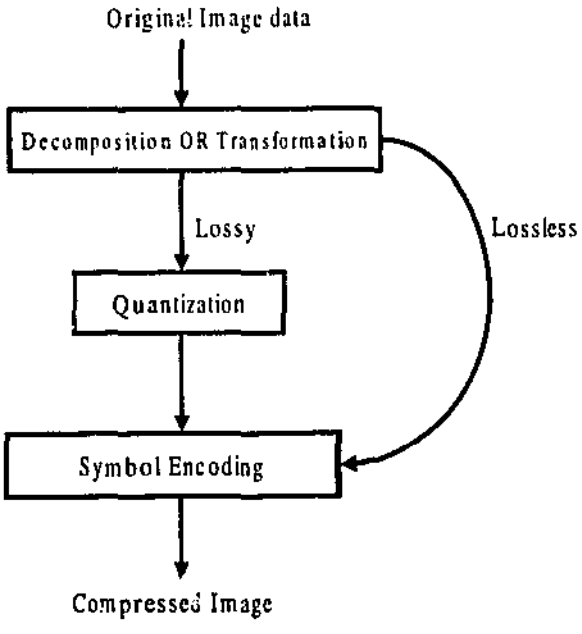



Figure 1.5 A general compression framework

Image compression techniques can also be divided into transform and non transform coding. In transform compression the image data are transformed into transform coefficients by applying some transform functions, such as DCT (discrete cosine transform and wavelet transform, and the resulting coefficients are encoded. In non-transform compression no such transformation is applied. Wavelet coding and JPEG are examples of transform coding while PCM and DPCM are examples of non-transform compression techniques.

Figure 1.5 shows a general compression framework. It includes three components: image decomposition or transformation, quantization, and symbol generation. As we can see from the figure, the primary difference between lossy and lossless schemes is the inclusion of the quantization stage in the lossy compression technique, while it is absent in the lossless scheme.

### 1.4.1 Lossless Compression Techniques

In this section we look at two main lossless compression techniques. They are Bit Plane Encoding and Lossless Predictive Coding.

#### 1.4.1.a Bit Plane Encoding

Bit plane encoding is a lossless and a non-transform compression technique. Consider an N x N image in which a pixel value is represented by k bits. By selecting a single bit from the same position in the binary representation of each pixel, an N x N binary image called a bit plane can be formed. For example we can select the most significant bit of each pixel value to generate an N x N binary image representing the most significant bit

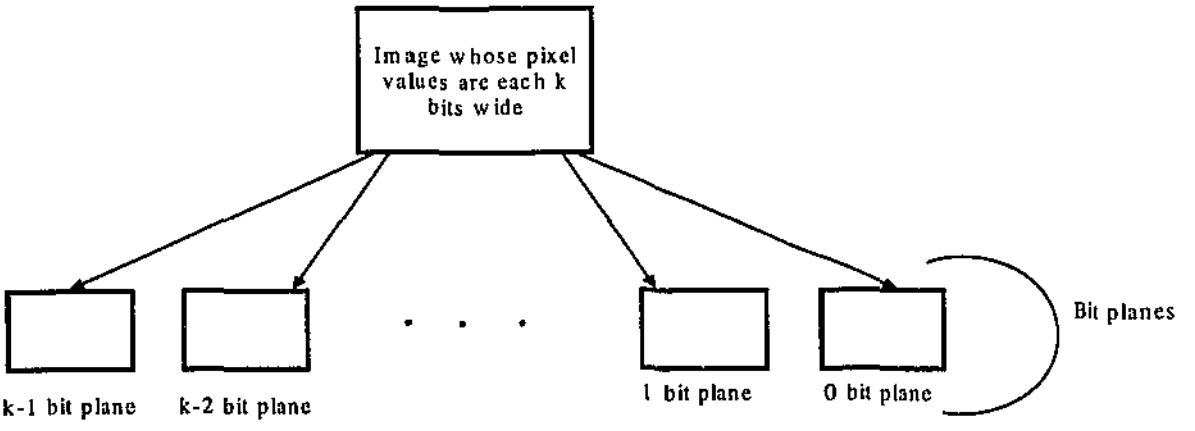



Figure 1.6 An image whose pixel values are k bits wide decomposed into k bit planes

k-1 bit plane contains the most significant bits form all the pixel values in order similarly k-2 contains the next most significant bits from all the pixel values in order and so on.

plane. Repeating this process for the other bit positions, the original image can be decomposed into a set of k, N x N bit planes (numbered 0 for the least significant bit (LSB) plane through k-1 for the most significant (MSB) plane. Each bit plane is then encoded efficiently using a lossless binary compression technique like Run Length Encoding and Arithmetic Encoding.

### 1.4.1.b Lossless Predictive Coding

For typical images, the values of adjacent pixels are highly correlated; that is, a great deal of information about a pixel value can be obtained by inspecting its neighbouring pixels. This property is exploited in predictive coding techniques where an attempt is made to predict the value of a given pixel based on the values of the surrounding pixels. The new information of a pixel is defined as the difference between the actual and the predicted value of the pixel.

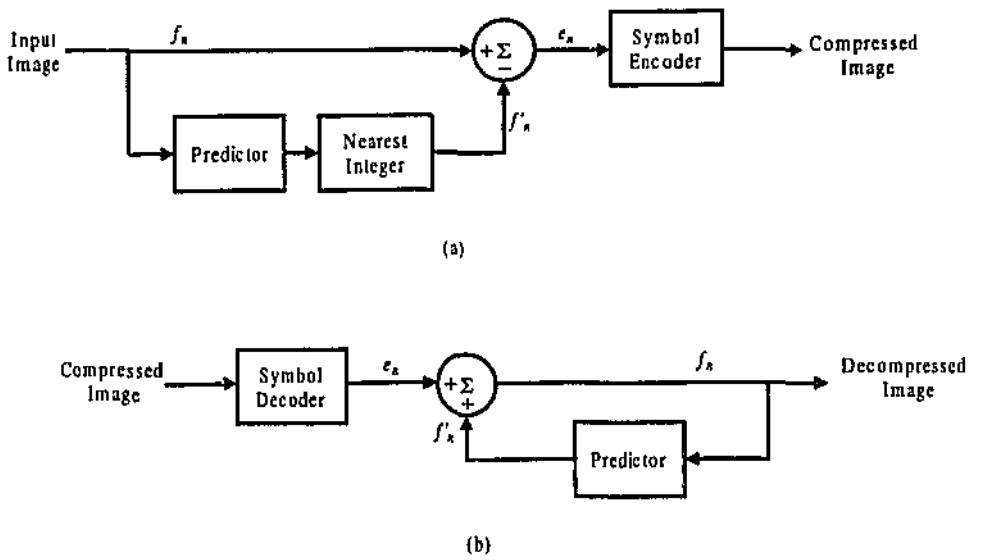



Figure 1.7 A lossless predictive coding model: (a) encoder; (b) decoder.
*Adopted from Gonzalez and Woods, 1993*

Figure 1.7 shows the basic components of a lossless predictive coding system. The system consists of an encoder and a decoder, each containing an identical predictor. As each successive pixel of the input image, denoted $f_n$, is introduced to the encoder, the predictor generates the anticipated value of that pixel based on some number of inputs. The output of the predictor is then rounded to the nearest integer, denoted $f'_n$, and used to form the prediction error

$$e_n = f_n - f'_n$$

which is coded using a variable length code. The decoder reconstructs $e_n$ from the received variable length code words and performs the inverse operation.

$$f_n = e_n + f'_n$$

### 1.4.2 Lossy Compression Techniques

There are a lot of lossy compression techniques available and as a result it isn't feasible to cover all of them. We will instead look at some of the most prominent ones in this section.

#### 1.4.2.a Discrete Cosine Transform Coding

Discrete Cosine Transform (DCT) is a popular transform image compression technique. The JPEG image format uses DCT method. (Topiwala,1998) In DCT the image is divided into blocks or rectangular arrays of pixels. Most existing systems use blocks of regular size, such as 8 x 8 or 16 x 16 pixels. Larger block sizes lead to more efficient coding, but require more computational power.

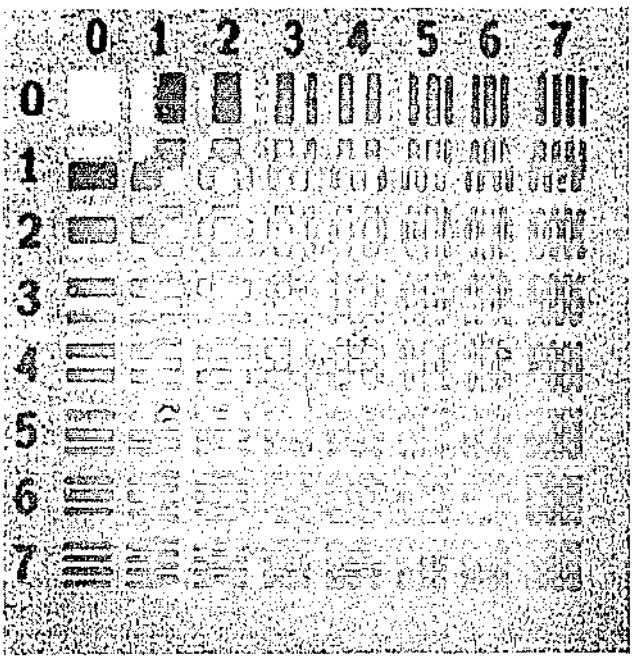



Figure 1.8 8 × 8 DCT basis Functions
*Adopted from Rabbani and Jones (1991)*

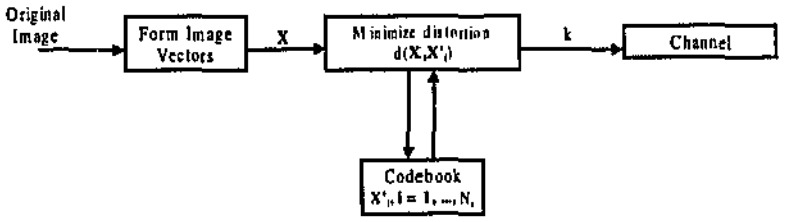
The DCT is applied to each block that converts a block of pixels into a block of DCT coefficients of the same dimensions. These coefficients represent the spatial frequency components that make up an appropriate basis function. The basis function for 8 × 8 DCT are shown in figure 1.8. The top left function is the basis function of the 'dc' coefficient and represents zero spatial frequency. Along the top row the basis functions

have increasing horizontal spatial frequency content. Down the left column, the functions have increasing vertical spatial frequency content, and along the diagonal the functions have combination of vertical and horizontal spatial frequencies.

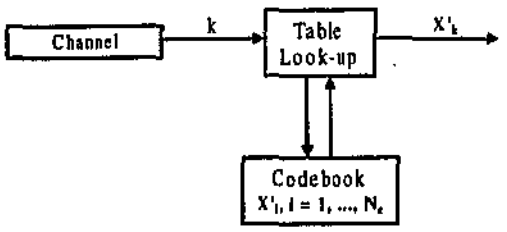
The resulting coefficients are then quantized using uniform quantization step sizes. The quantized DCT coefficients are then scanned in a specific diagonal order, starting from the "dc" or 0-frequency component, then run-length coded, and finally entropy coded according to either Huffman or arithmetic coding. (Topiwala, 1998)

#### 1.4.2.b Vector Quantization

In vector quantization (VQ) method, the original image is first decomposed into n-dimensional image vectors. The vectors can be generated in a number of different ways. For example, an $n = l \times m$ block of pixel values can be ordered to form an n-dimensional vector.




(a) Transmitter




(b) Receiver

Figure 1.9 Vector Quantization block diagram
*Adopted from Rabbali and Jones (1991)*

Each image vector, **X**, is then compared with a collection of representative templates or codevectors $\mathbf{X'}_i$ taken from a previously generated codebook. The code vectors are also of dimension *n*. The best match codevector is chosen using a minimum distortion rule.

After a minimum distortion codevector has been selected, its index *k* is transmitted. At the receiver this index is used as an entry to a duplicate codebook to reproduce the original codevector

#### 1.4.2.c Wavelet Coding

Wavelet coding is one of the most recent techniques of image compression that has been developed. This technique uses the wavelet transform to remove the spatial correlation that exists in images. The wavelet transform coefficients that are obtained contain the information in a compact smaller number of coefficients. These coefficients are then quantized and then efficiently coded using suitable coding algorithms.

Image coding using using wavelet transform exhibits several desirable qualities. Since the wavelt transform executes a multiresolution analysis on the image, it essentially processes the image in much the same way as the human visual system does. The importance of the resulting transform coefficients to the reconstructed image is then easily evaluated for coding purpose. The wavelet transform enjoys a considerable amount of design freedom in the choice of the basis wavelet. By proper choice of the analyzing wavelet, the wavelet transform can be tailored to a specific style of implementation.

Since this project work focus on the EZW algorithm of wavelet coding, wavelet coding method is very welt explained in the chapters ahead.

#### 1.4.2.d Lossy Plus Lossless Residual Coding

Another way of image compression is to mix both the lossy and the lossless techniques. One such techique is the Lossy Plus Lossless Residual coding. Lossy Plus Lossless Residual coding is used in application where it suffices to send a lossy version of the image first and then the lossless version afterwards as needed. One such application might exist in the medical field, where two physicians are discussing a possible patient referral from remote locations. One of the physicians may wish to transmit a digital radiograph over the phone line, and in the interest of a short transmission time, a lossy

(but high quality) version of the iage is sent. If the referral is accepted, the remaining difference (residual) image required to perfectly reconstruct the original image could be sent.

In general a lossy plus a lossless residual encoding scheme consists of the following steps:

- Generate a low bit rate image through the use of an efficient lossy scheme.
- Form a residual by computing the difference between the lossy reconstruction and the original image.
- Encode the residual using an appropriate lossless technique.

# 2. WAVELET TRANSFORM AND IMAGE COMPRESSION

## 2.1 Introduction

Wavelet transforms, as an alternative to the Fourier and related transforms, for application to practical engineering problems have been the focus of intensive research in recent years. The concept of Wavelet itself was introduced quite recently in 1984 by Goupillaud, Grossman and Morlet as a new mathematical tool for multiresolution decomposition of continuous-time signals. This mathematical tool for multiresolution analysis of signals has been investigated and applied in various fields including geophyiscs, image analysis for the purpose of segmentation, pattern recognition and coding. The incentive for this is its ability to provide a multiresolution or multiscale analysis of signals with flexible space-frequency localization.

In the field of image compression wavelets have captured the imagination and talents of researchers all over the world. A number of researches that wavelet transform holds considerable promise in image compression. The most revolutionary thing about wavelet transform is that since it executes a multiresolution analysis on the image, it essentially processes the image in much the same manner as the human visual system. (Coffey and Etter,1995).

## 2.2 Mathematical Representation of Wavelets

Wavelets are functions generated from one single function, the mother wavelet $\psi$ by dilation and translation. Grossman and Morlet (Grossman and Morlet, 1984) introduced this function $\psi$ which dilated by a scaling factor $a$ and translated by $b$ enables the analysis, processing, and synthesis of a signal.

$$\psi_{a,b}(x) = |a|^{-1/2}\psi\left(\frac{x-b}{a}\right) \quad (a,b)\in \mathbf{R}^2,\ a\neq 0 \qquad (1\text{-}1)$$

It is assumed that $x$ is a one dimensional variable. The mother wavelet $\psi$ must satisfy the following admissibility condition.

$$\int \frac{|\Psi(\omega)|^2}{|\omega|} d\omega < \infty \qquad (1\text{-}2)$$

where $\Psi$ denotes the Fourier transform of $\psi$. Moreover if $\psi$ has sufficient decay, then (1-2) is equivalent to

$$\int_{-\infty}^{+\infty} \psi(x)dx = 0 \qquad (1\text{-}3)$$

which means that the wavelet $\psi$ exhibits at least a few oscillations, and that there is a large choice of functions for $\psi$.

The basic idea of the wavelet transform is to represent an arbitrary function $f$ as a superposition of wavelets. This function $f$ can then be decomposed at different scale or resolution levels. One way to achieve such a decomposition involves writing $f$ as an integral of $\psi_{a,b}$ over $a$ and $b$ using appropriate weighting coefficients. In practice, however, it is preferable to express $f$ as a discrete sum rather than as an integral. The coefficients $a$ and $b$ are discretized such that:

$$a = a_0^m \text{ and } b = nb_0a_0^m \text{ with } (m,n) \in \mathbf{Z}^2 \text{ and } a_0 > 1,\ b_0 > 0 \text{ fixed.}$$

The wavelet is then defined as follows:

$$\psi_{m,n}(x) = \psi_{a_0^m, nb_0a_0^m}(x) = a_0^{-m/2}\psi\left(a_0^{-m}x - nb_0\right) \qquad (1\text{-}4)$$

and the wavelet decomposition of $f$ becomes

$$f = \sum_{m,n} c_{m,n}(f)\,\psi_{m,n} \qquad (1\text{-}5)$$

For large positive values of $m$ ($a$>1), the $\psi$ function is highly dilated and large values for the translation step $b$ are well adapted to this dilation. This corresponds to low frequency or narrow band-wavelets. For large values of $m$ ($a$ <1), the $\psi$ function is highly concentrated and the translation step $b$ takes small values. These functions correspond to high frequency or wide band wavelets.

Y. Meyer showed that there are $\psi$ functions for $a^0 = 2$ and $b^0 = 1$, such that the functions $\psi_{m,n}(x)$ make up an orthonormal basis belonging to $L^2(R)$, where

$$\psi_{m,n}(x) = 2^{-m/2}\,\psi\left(2^{-m}x - n\right) \qquad (m,n) \in Z^2 \tag{1-6}$$

(Z is the set of all integers and R is the set of all real numbers and N is the set of all Natural numbers)

The wavelet coefficients $C_{m,n}(f)$ are determined using the following relation:

$$c_{m,n}(f) = \langle f, \psi_{m,n} \rangle = \int f(x)\,\overline{\psi}_{m,n}(x)\,dx \tag{1-7}$$

The oldest known basis of this type was constructed by Haar. In this case, the function $\psi(x)$ is equal to 1 over the interval [0,1/2], -1 over [1/2, 1] and 0 elsewhere. Different bases corresponding to more regular wavelets were later constructed by Stromberg, Meyer, Lemarie, Battle, and Daubechies.

The existence of orthonormal wavelet bases is conditioned by the following regularity property: $|\Psi(\omega)|$ must decrease more rapidly than $C(1 + |\omega|)^{-\varepsilon-0.5}$ for $\omega \to \infty$ and for $\varepsilon > 0$ ;where $C$ is a constant.
Wavelets which exhibit this regularity property necessarily verify:

$$\int x^r \psi(x)\,dx = 0 \tag{1-8}$$

This equation determines the number of vanishing moments of $\psi$ and thus enables evaluation of the oscillations of the wavelet $\psi$

## 2.3 Continuous Wavelet Transform

To perform wavelet transform on a time-domain signal, the signal is passed through various high-pass and low-pass filters, which filter out either high frequency or low frequency portions of the signals. Figure 2.1 shows how a wavelet transform is performed on a signal $x(t)$ which has a maximum frequency of B Hz.

This is how it works: Suppose we have a signal that has frequencies up to 1000 Hz. In the first stage the signal is spilt into two parts by passing the signal through a high-pass and a low-pass filter. Which result in two different versions of the same signal; portion of signal corresponding to 0-500Hz (low-pass portion) and 500-1000Hz(high-pass portion). Then we take either portion or both and pass them through another set of high and low pass filters. This operation is called decomposition.

Assuming that we took the low pass output from the first filtering and performed the second decomposition, we have three sets of data now, each corresponding to the same signal frequencies 0-250 Hz, 250-500Hz and 500-1000Hz. If we take the low pass portion and pass it through low and high pass filters, we now have four sets of signals corresponding to 0-125Hz, 125-250Hz, 250-500Hz and 500-1000Hz.

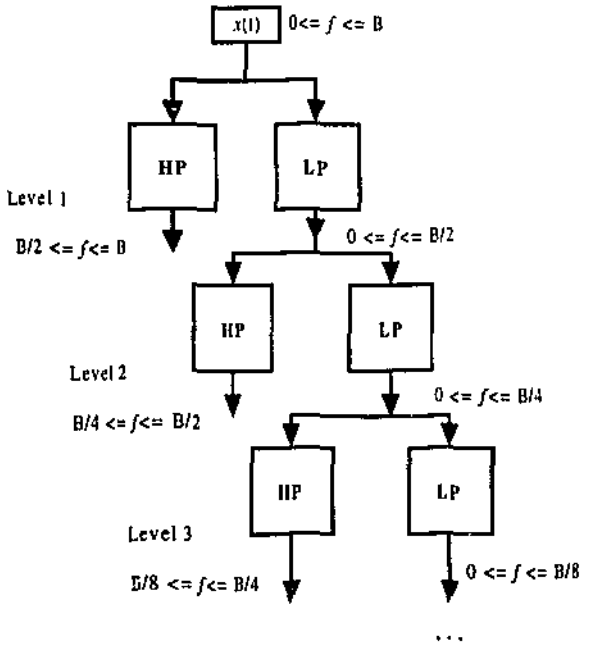



Figure 2.1 Wavelet transform as passing a signal through sets of high pass(HP) and low pass(LP) filters
B Hz is the maximum signal content of an analog signal x(t)

In this way we continue to decompose a signal until we have decomposed the signal into a certain predefined level. At the end we have a bunch of signals, which actually represent the same original signal, but all corresponding to different frequency bands. If the resulting signal is plotted on a 3-D graph, we will have time in one axis, frequency in the other and amplitude in the third. This will show us which frequencies exist at which time. In other words the resulting signal can be resolved both in time as well as in frequency.

However there is a principle known as the uncertainty principle which states that " we cannot exactly know what frequency exists at what time instance, but we can only know what frequency bands exist at what time intervals". This is analogous to the uncertainty principle in quantum Physics ascribed to Heisenberg that states " the momentum and the position of an electron can not be determined simultaneously". This is a problem of resolution, and it is the main reason why researchers have switched from STFT (short time Fourier transform) to WT (wavelet tarnsform). STFT also resolves a signal in both frequency and time, but it gives a fixed resolution at all times whereas WT gives variable resolution.

### 2.3.1 Resolution from Continuous Wavelet Transform

Higher frequencies are better resolved in time, and lower frequencies are better resolved in frequency. This means that a certain high frequency component can be located better in time (with less relative error) than a low frequency component. On the contrary, a low frequency component can be located better in frequency compared to high frequency component.

The grid in the figure 2.2 is interpreted as follows: The top row shows that higher

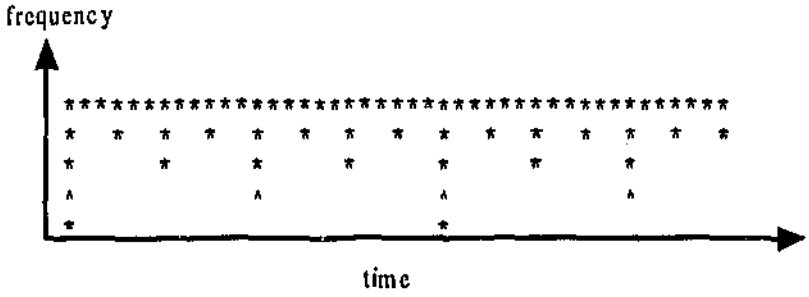



Figure 2.2. Time and frequency resolution from Continuous Wavelet Transform

*Adapted from Robi Polikar, Wavelet tutorial, 1994*

frequencies we have more samples corresponding to smaller intervals of time. In other words, higher frequencies can be resolved better in time. The bottom rows correspond to low frequencies, and there is less number of points to characterize the signal. Therefore low frequencies are not well resolved in time.

## 2.4 Discrete Wavelet Transform

The Discrete Wavelet Transform (DWT) is the analog of the continuous wavelet transform (CWT) presented in the previous section, in the discrete time domain. A time-scale (scale and frequency have inverse relationship) representation of a digital signal is obtained using digital filtering techniques. As in the continuous time the digital signal is passed through a series of low and high pass filters.

The procedure starts with passing the signal (sequence, since discrete) through a half band digital low pass filter with impulse response $h[n]$ and a half band digital high pass filter with impulse response $g[n]$. Filtering a signal is a mathematical operation of convolution of the signal with the impulse response of the filter. The convolution in discrete time is defined as follows.

$$x[n]*h[n] = \sum_{k=-\infty}^{\infty} x[k]\cdot h[n-k]$$

The pair of high pass and low pass filters are not independent but are related by

$$g[L-1-n] = (-1)^n \cdot h[n]$$

A half band low pass filter removes all frequencies that are above half of the highest frequency in the signal, while the high pass filter removes all the frequency components that are below half of the highest frequency in the signal. The resultant sequence is passed through another set of high and low pass filters and the process continues until a certain desired level of decomposition is done. The algorithm is shown in figure 2.3.

As an example, suppose that $x[n]$ has 512 sample points,spanning a frequency of zero to p radians. At the first decomposition level, the sequence is passed through high pass and low pass filters, followed by subsampling by 2. The high pass filter has 256 points, but it only spans the frequency range p/2 to p radians. These 256 samples constitute the first level of DWT coefficients. The output of the low pass filter has 256 points, and it spans the frequency band from 0 to p/2 radians. This sequence is then passed through similar low pass and high pass filters for further decomposition. The output of the second low pass filter followed by subsampling has 128 samples spanning a frequency band of 0 to p/4 radians, and the output of the second high pass filter followed by

subsampling has 128 samples spanning a frequency band p/4 to p/2. The second high pass filter samples constitute the second level of DWT coefficients. This signal has half the time resolution, but twice the frequency resolution of the first level signal. In other words, time resolution has decreased by a factor of 4, and frequency resolution has increased by a factor of 4 compared to the original signal. The low pass filter output is

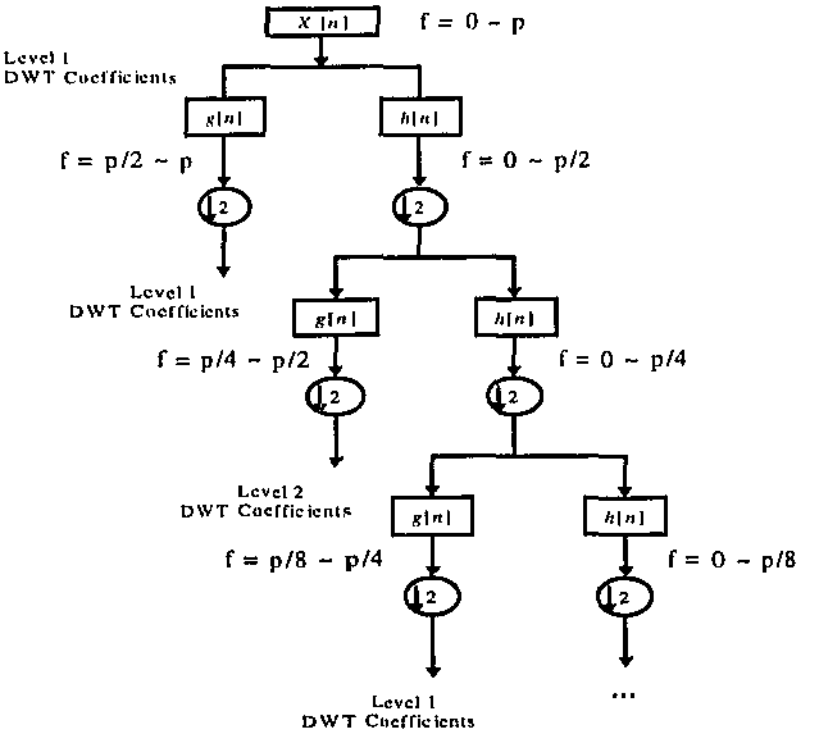



Figure 2.3 Discrete Wavelet transform of $X[n]$
$g[n]$ -> High pass filter
$h[n]$ -> Low pass filter

*Adopted from Robi Polikar, wavelet tutorial, 1994*

then filtered once again for further decomposition. This process continues until two sample is left. For this specific example there would be 9 levels of decomposition, each having half the number of samples of the previous level. The DWT of the original signal is then obtained by concatenating all coefficients starting from the last level of decomposition (remaining one sample, in this case). The DWT will then have the same number of coefficients as the original signal.

### 2.4.1 Resolution from Discrete Wavelet Transform

In the discrete time case, the time resolution of the signal works the same as in the continuous time case, but with one exception. The frequency information has different

resolutions at every stage too. Lower frequencies are better resolved in frequency, whereas higher frequencies are not. Figure 2.4 shows the time – frequency resolution offered by discrete time wavelet transform.

Looking at figure 2.4 it is noticed how the spacing between subsequent frequency components increase as frequency increases. (Robi Polikar, 1994)

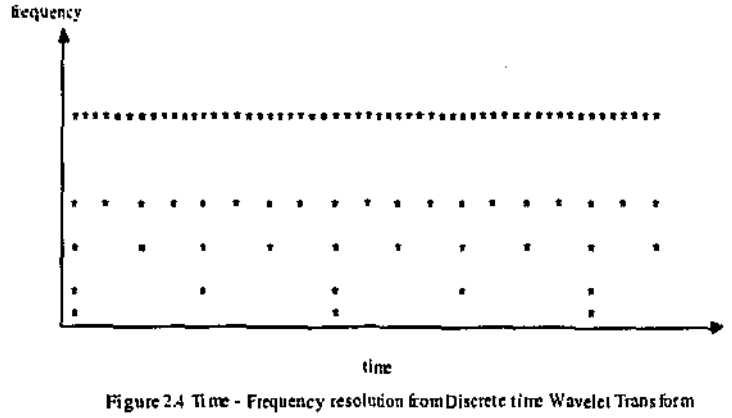



Figure 2.4 Time - Frequency resolution from Discrete time Wavelet Transform

## 2.5 Wavelet Transform and Digital Image Compression

Wavelet transform has opened up a whole new prospect for efficient image compression and wavelet transform technique of image compression has gained a lot of popularity in the last couple of years. Following are some of the most important reasons.

### 2.5.1 Data Compression

When a DWT is performed on a signal, frequencies that are most prominent in the original signal appear as high amplitudes in that region of the DWT signal that includes those particular frequencies. The frequency bands that are not very prominent in the original signal have very low amplitudes and that part of the DWT signal can be discarded without any major loss of information, thus allowing data reduction. In other words, wavelet transform concentrates the original signal values into a relatively small number of large magnitude coefficients.(Relue,1994) Figure 2.5 illustrates the data reduction obtained.

In practice all but a few percent of the wavelet coefficients can be set to zero. (Relue,1994). Selection of the coefficients can be done in two ways:

- An arbitrary threshold can be established as the cutoff point
- The coefficients can be ranked to allow selecting of an arbitrary percentage of the highest values for retention.

Typically, 5 percent of the values are retained, but good results can be obtained with smaller percentages. In image compression it is important to note that the zeroed coefficients cannot be thrown away. The position of the zeroed coefficients must still be known for reconstruction. (Relue,1994)

Once the data have been compressed by the removal of low value coefficients, more compression can be obtained by quantizing the non-zero wavelet coefficients.

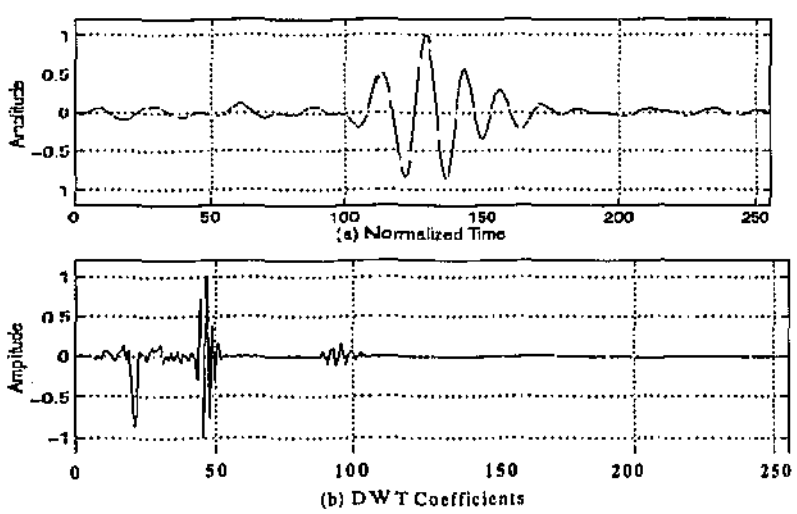



Figure 2.5 Data Reduction in DWT

(a) signal
(b) DWT coefficients

### 2.5.2 Better Frequency Resolution

The bulk of the information in images is found in lower frequency bands. We have already seen in figure2.4 that discrete wavelet transform provides better frequency resolution at lower frequency. This means that we can have better resolution of our images using wavelet transform

### 2.5.3 Noise Immunity

The compression process using wavelet transform has an interesting side effect. Since most of the noise in an image has low energy value, it will be suppressed when reconstructing the compressed data. Figure 2.6 shows a sine wave with 50 percent noise added, and the reconstructed since wave from 3 percent of the original data using Daubechies 2-D transform. The original since wave is very easy to distinguish in the reconstruction. (Savla, 1998)

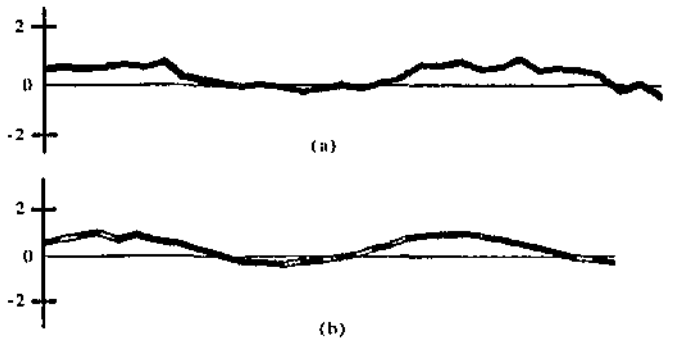



Figure 2.6 The Original (Noisy) and Transformed Sine Curve

(a) Original(noisy)
(b) Transformed

*Adapted from Savla, 1998*


Figure 2.7 The Reconstructed low-noise signal
*Adopted from Savla, 1998*

The reconstruction of low noise signal is generally very good. Simple waveforms such as a since wave can be done with 3 percent of the data as illustrated in Figure 2.7

# 3. THE EZW ALGORITHM

## 3.1 Introduction

In the last chapter we studied the wavelet transform and how it is performed in both continuous and discrete time domains. We then observed the results of wavelet transform on signals and particularly related the results to image coding. And we found that wavelet transform holds considerable promise for image compression.

In this chapter we present the EZW (Embedded Zerotree Wavelet) algorithm, which is an image compression algorithm formulated by J.M.Shapiro.

The EZW algorithm has the property that the bit streams are generated in the order of importance and all information is contained within the code thereby yielding a fully embedded code. Using this algorithm the encoder can terminate the encoding at any point thereby allowing the target rate or target distortion metric to be met. Also, given a bit stream the decoder can cease decoding at any point in the bit stream and still produce exactly the same image that would have been encoded at the bit rate corresponding to the truncated bit stream. (Shapiro, 1993)

## 3.2 Features of the Embedded Coder

The EZW algorithm contains the following features:

- A discrete wavelet transform which provides a compact multiresolution representation of the image

- Zerotree coding which provides a compact multiresolution representation of significance maps, which are binary maps indicating the position of the significant coefficients. Zerotrees allow the successful prediction of significant coefficients across scales to be efficiently represented as part of exponentially growing trees.

- Successive approximation which provides a compact multiresolution representation of the significant coefficients and facilitates the embedding algorithm

- A prioritization protocol whereby the ordering of importance is determined, in order, by precision, magnitude, scale, and spatial location of the wavelet coefficients. Larger coefficients are deemed more important than smaller coefficients regardless of their scale.

- Adaptive multilevel arithmetic coding which provides a fast and efficient method for entropy coding of symbols, and requires no training or prestored tables.

- The algorithm runs sequentially and stops whenever a bit rate is met.

Figure 3.0 shows a generic transform coder.

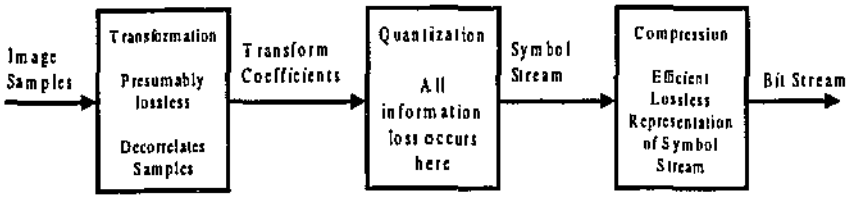



Figure 3.0 A generic transform Coder

## 3.3 2 - D Discrete Wavelet Transform of Image

Before the algorithm can be employed, the image is 2-D discrete wavelet transformed and wavelet coefficients for the image are obtained. It is these wavelet coefficients that are encoded using the algorithm.

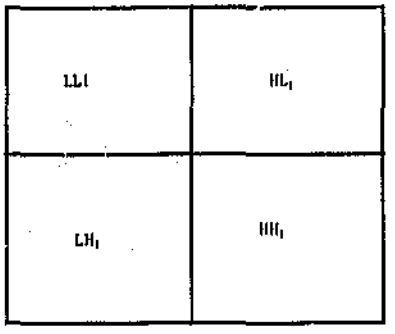



Figure 3.1 A one - scale 2-D wavelet decomposition of an Image.

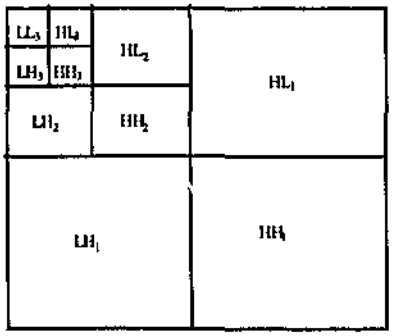



Figure 3.2 A three scale 2-D wavelet decomposition of an Image.

Figure 3.1 shows the result of first stage of wavelet transform of an image. Because we are dealing with digital image compression here, whenever we say wavelet transform, it should be understood as the discrete wavelet transform.

As explained in the previous chapter, the four subbands in figure 3.1 arise from the separable application of vertical and horizontal filters. The subbands $LH_1$, $HL_1$ and $HH_1$ represent the finest scale wavelet coefficients. To obtain the next coarse scale of wavelet coefficients, the subband $LL_1$ is further decomposed. To obtain the third scale level of wavelet coefficients, as shown in figure 3.2 the subband $LL_2$ is further decomposed. As could be obvious, in the figure, the first letter of the L and H combination refers to the horizontal filter outcome and the second letter refers to the vertical filter outcome. The subscript indicates the number of scales. For example $HL_2$ indicates that it is the outcome of the high pass horizontal filter and the low pass vertical filter of scale 2.

Since we know that $-\pi \leq \omega \leq \pi$ for discrete signals, in figure 3.1 the low frequencies represent a bandwidth approximately corresponding to $0 \leq |\omega| \leq \pi/2$ while the high frequencies correspond to $\pi/2 \leq |\omega| \leq \pi$. With each level of decomposition these bandwidths get halved from the previous ones.

## 3.4 The Zerotree Data Structure

The wavelet coefficients in one subband have a parent child relationship with the wavelet coefficients in other subbands when significance with regard to particular threshold value is concerned. This gives rise to a new data structure called the zerotree which improves the coding of the wavelet coefficients. A wavelet coefficient $x$ is said to be insignificant with respect to a threshold T if $|x| < T$. The zerotree is based on the hypothesis that if a wavelet coefficient at a coarse level is insignificant with respect to a given threshold T, then all wavelet coefficients of the same orientation in the same spatial location at finer scales are likely to be insignificant with respect to T. This hypothesis is found to be often true. (Shapiro,1993). Figure 3.3 shows the parent child relationship of the coefficients in the subbands.

The coefficient at the coarse scale is called the parent, and all coefficients corresponding to the same spatial location at the next finer scale of similar orientation are called children. For a given parent the set of all coefficients at all finer scales of similar orientation corresponding to the same location are called descendents. Similarly for a given child, the set of all coefficients at all coarser scales of similar orientation corresponding to the same location are called ancestors. With the exception of the lowest frequency subband, all parents have four children. For the lowest frequency

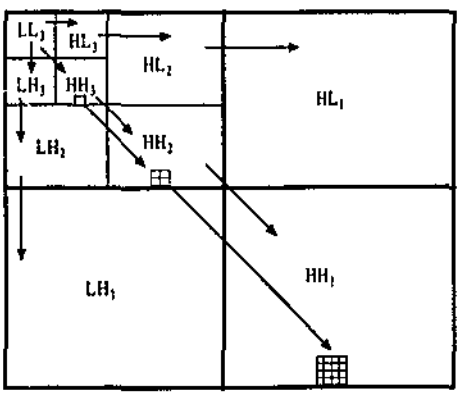



Figure 3.3 Parent-child relationship of a three scale 2-D wavelet Coefficients
*Adopted from Shapiro, 1993*

subband, the parent-child relationship is defined such that each parent node has three children.

In Figure 3.3 the arrow points from the subband of the parents to the subbands of the children. $LL_3$ the lowest frequency subband is at the top left. Also shown are the children and the descendents of $HH_3$.

## 3.5 The Number of Zerotrees

Depending on the number of scales of decomposition that is performed we get different number of zerotrees of the wavelet coefficients. If there is only one coefficient remaining in the $LL_n$ for an n-scale decomposition then we get one tree. Otherwise we get multiple trees. This is important to understand because in some cases we don't need to decompose so that there is just one coefficient left in the lowest frequency band $LL_n$. As a rule of thumb, for image size like $512 \times 512$, only five or six scales of decomposition

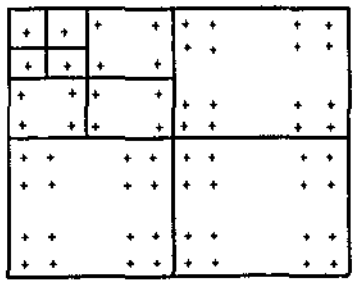


Figure 3.4 A 3-scale decomposition of an 8 x 8 wavelet coefficients
Consist of just one tree as $LL_3$ has just one coefficient
+represents a wavelet coefficient

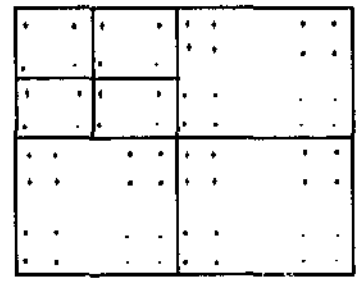


Figure 3.5 A 2-scale decomposition of an 8 x 8 wavelet coefficients
Consists of four trees as $LL_2$ contains four coefficients
Each of the four different symbols represents a different tree

are performed. If we decompose the above 512 x 512 to five scales, we end up with eight coefficients in the $LL_5$ subband, and so we have eight trees. Infact the number of trees is equal to the number of coefficients in the lowest frequency subband ($LL_n$). In Figure 3.4 shows an 8 x 8 wavelet coefficients from three-scale decomposition so that there is only one tree, and Figure 3.5 shows the same 8 x 8 wavelet coefficients but from

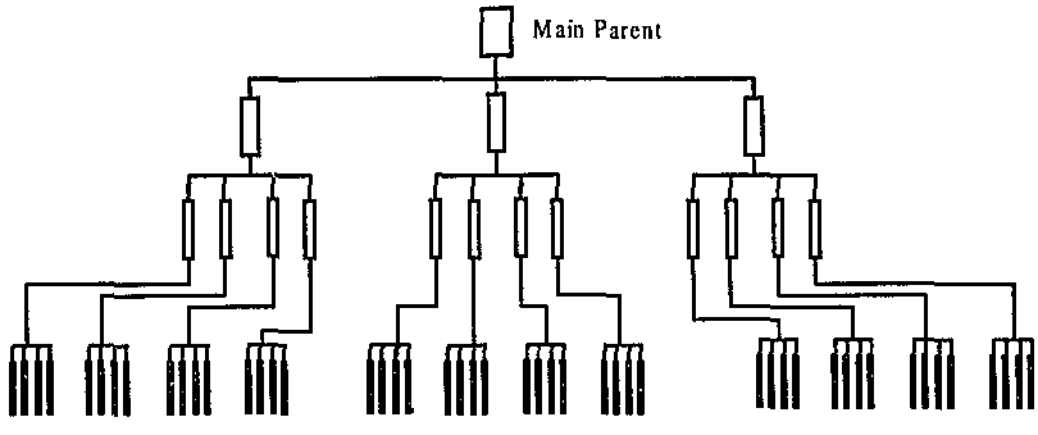



Figure 3.6 A tree structure of wavelet coefficients
A 3 scale decomposition of wavelt trasform on an 8 x 8 image

Note how the first main parent has just three children while the rest have four children each. Ofcourse the leaves don't have any children.

only two-scale decomposition and thus resulting in four trees. The members/coefficients of each different tree are differentiated by representing them by *, +, °, and -. Figure 3.6 shows one tree structure in a tree-like representation.

## 3.6 The Significance Map

The significance map contains the significance information of coefficients in a tree. It contains information whether a coefficient's descendents are significant or not and whether its ancestors are significant or not. This significance information or significance map is very useful when encoding a coefficient code as zerotree root or an isolated zero. The idea will become clear when we discuss section 3.8

## 3.7 Scanning of Coefficients

To process the coefficients, the scanning of coefficients is performed in such a way that no child node is scanned before its parent. For an n-scale transform, the scan begins at the lowest frequency subband, denoted as $LL_n$, and scans subbands $HL_n$, $LH_n$, and $HH_n$, at which point it moves on to the scale n-1, and so on. The scanning pattern for 3-scale wavelet transform coefficients is shown in Figure 3.6. We note that each coefficient within a given subband is scanned before any coefficient in the next subband.

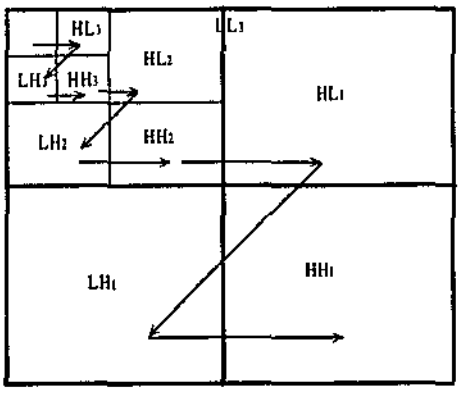



Figure 3.7 Scanning order of the subbands for encoding a significance map

## 3.8 Encoding a Wavelet Coefficient

Given a threshold level T we now discuss how a coefficient is encoded. A coefficient can be any one of the following four:

- A positive significant
- A negative significant
- A Zerotree root, and

- An Isolated zero

A coefficient $x$ is significant with respect to a threshold value T if $|x| \geq T$. A significant coefficient is positive significant if it is positive and negative significant if it is negative.

A coefficient $x$ is a zerotree root if itself and all its descendents are insignificant with respect to a threshold T.
A coefficient $x$ is an isolated zero if it is itself insignificant but at least one of its descendents is significant.

Accordingly four different symbols, one for each of the four kinds of coefficients that can be encountered can be assigned. Four such symbols can be as shown in Table 3.1

| Symbol | Meaning |
|---|---|
| POS | Positive |
| NEG | Negative |
| ZTR | Zerotree Root |
| IZ | Isolated Zero |

Table 3.1 Symbols and their meanings for coding a wavelet coefficient

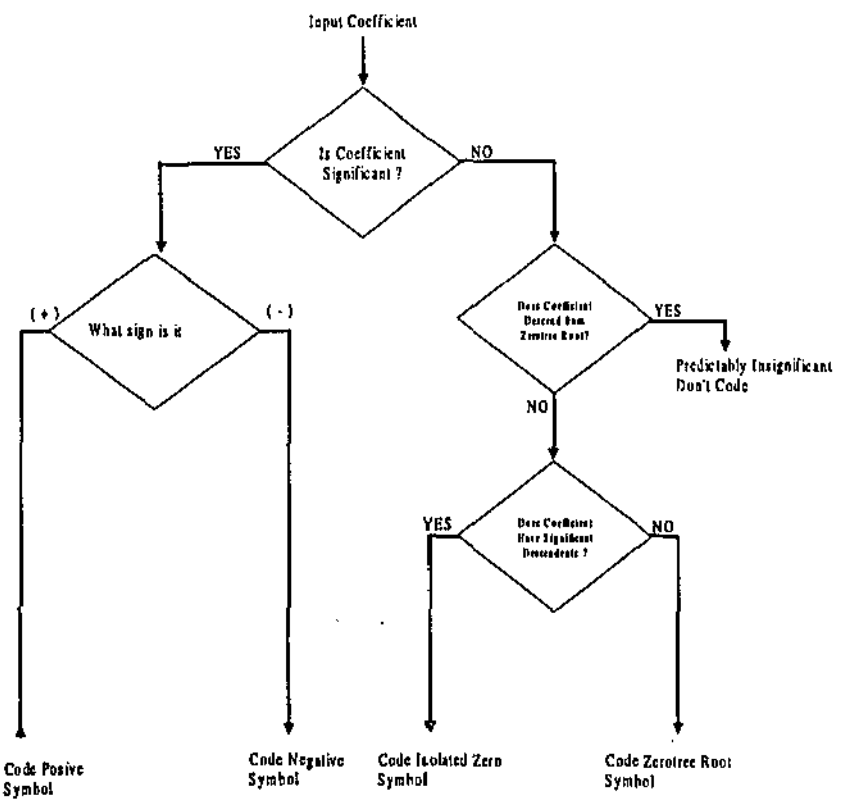



Figure 3.8 Flow Chart for encoding a coefficient of the significance map

In this project these symbols and their meanings are used. The flow chart for encoding a coefficient is shown in Figure 3.8

## 3.9 Successive Approximation Quantization

To perform the embedded coding, successive–approximation quantization (SAQ) is applied. The SAQ sequentially applies a sequence of thresholds $T_0$ to $T_{n-1}$ to determine the significance, where the thresholds are chosen so that $T_i = T_{i-1}/2$. The initial threshold is chosen such that $T_0 > | X_j | /2$ where $X_j$ is the maximum of all the transform coefficients.

During the encoding and decoding, two separate lists of wavelet coefficients are maintained. These lists are called the dominant list and the subordinate list. At any point in the process, the dominant list contains the coordinates of the coefficients that have not yet been found significant in the same relative order as the initial scan. The scan is such that the subbands are ordered, and within each subband, the set of coefficients is ordered. Thus using the ordering of the subbands as shown in figure 3.6, all coefficients in a given subband appear on the initial dominant list prior to coefficients in the next subband. The subordinate list contains the magnitudes of those coefficients that have been found to be significant. For each threshold the list is scanned once.

During the dominant pass, scanning the coefficients with coordinates on the dominant list, i.e. scanning the coefficients that have not been found significant, are compared to the threshold $T_i$ to determine their significance, and sign if they are found to be significant. The significance map is zerotree coded as described in section 3.8. Each time a coefficient is encoded as significant, (positive or negative significant), its magnitude is appended to the subordinate list. Then the coefficient in the wavelet transform array is set to zero, so that the significant coefficient does not prevent the occurrence of a zerotree on future dominant passes at smaller thresholds.

After a dominant pass a subordinate pass is performed. During this subordinate pass the subordinate list is scanned and the specifications of the magnitudes available to the decoder are refined to an additional bit of precision. Specifically, during a subordinate pass the width of the width of the quantizer step size, which defines the uncertainty interval of the true magnitude of the coefficient, is halved. For each magnitude on the

subordinate list, this refinement can be encoded using a binary bit " symbol to indicate that the true value falls in the upper half of the old uncertainty level. We should note that prior to this refinement, the width of the uncertainty level is exactly equal to the current threshold. After the completion of the subordinate pass the magnitudes of the subordinate lists are sorted in decreasing magnitude, to the extent that the decoder has the information to perform the same sort.

The process continues to alternate between the dominant and the subordinate passes where the threshold is halved before each dominant pass. In principle one could divide by any factor other than 2. The factor of 2 is chosen because it has nice interpretations in terms of bit plane encoding and numerical precision in a familiar base 2, and good coding results were obtained. (Shapiro,1993)

In the decoding operation, each decoded symbol, both during the dominant and the subordinate passes, refines and reduces the width of the uncertainty level in which the true value of the coefficient may occur. The center of the uncertainty interval is used as the reconstruction value.

The encoding stops when some target stopping condition is met, such as when a bit budget is exhausted. The encoding can cease at any time and the resulting bit stream contains all lower rate encodings. Further more decoding cans top at any point. However terminating the decoding of an embedded bit stream at a specific point in the bit stream produces the same image that would have resulted had that point been the encoding target rate. This ability to cease encoding and decoding anywhere is extremely useful in systems that are either rate-constrained or distortion-constrained.

## 3.10 Experimental Results Obtained by Shapiro

- The compression performance of this algorithm was found to be competitive with virtually all known techniques.

- The precise rate control that is achieved with this algorithm is a distinct advantage.

- The performance of the EZW coder was compared to widely available version of JPEG. JPEG does not allow the user to select a bit rate but instead allow the user to choose a "quality factor"

- A "Barbara" black and white picture was first encoded using a file size of 12,866 bytes. The PSNR (peak signal to noise ratio) in this case was found to be 26.99 dB. To the same "Barbara" picture EZW algorithm was applied with the same target file as above of exactly 12,866 bytes. The resulting PSNR was 29.39 dB, which is significantly higher than for the JPEG. The EZW encoder was then applied to the same picture using the target PSNR of 26.99 dB. The resulting file size was 8820 bytes.

- When encoding or decoding is terminated during the middle of a pass, there are no artifacts produced that would indicate where the termination occurred.

- A "Lena" image was coded at high compression ratio of 512:1. The image quality was poor but still recognizable. This is not the case with conventional block coding schemes, where at such a high compression ratio, there would not be enough bits to even encode the DC coefficients of each block. (Shapiro,1993)

# 4. ENCODER AND DECODER DESIGN

## 4.1 Introduction

This chapter is divided into three parts. In Part A, the design of a single processor encoder is presented. Part B deals with the design of single processor decoder. In part C the design of codec using three parallel processors to process a zerotree of wavelet coefficients is presented using the principles of the single processor codec design.

The design of the codec described in this chapter is very generic and can be used to encode and decode any size zerotree of wavelet coefficients. In keeping with the specification of the project, the coefficient values are assumed to be between –128 and + 127, which is for an 8 bit implementation. However the ideas can be applied for higher value coefficients.

## A. Encoder Design

## 4.2 Single Processor Architecture

Here we look at the architecture of the single processor encoder. The encoder that uses three parallel processors is explained afterwards. This is helpful because it becomes

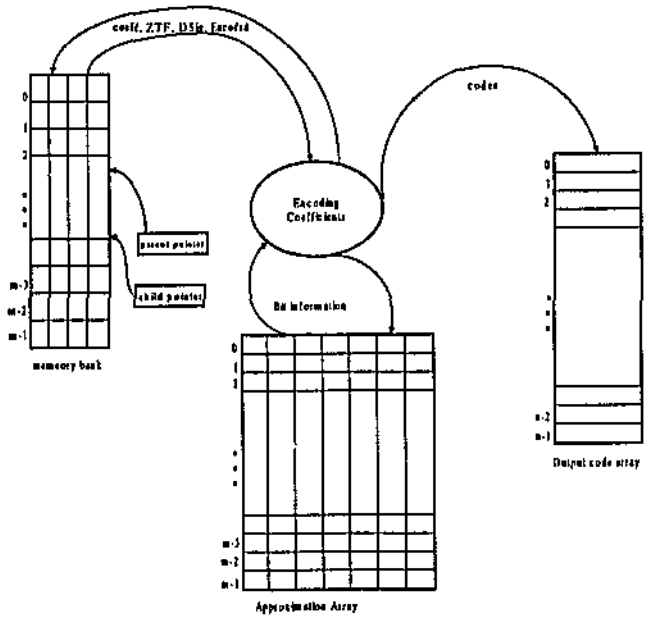



Figure 4.1 The Encoder

fairly easy to grasp the idea once we have discussed the single processor encoder. Figure 4.1 shows a generic encoder

### 4.2.1 Mapping Coefficients to Memory Bank

Let the tree size be *m*. For an *n x n* image that has been wavelet transformed to a single coefficient for the lowest frequency subband, $m = n^2$. If the image has been wavelet

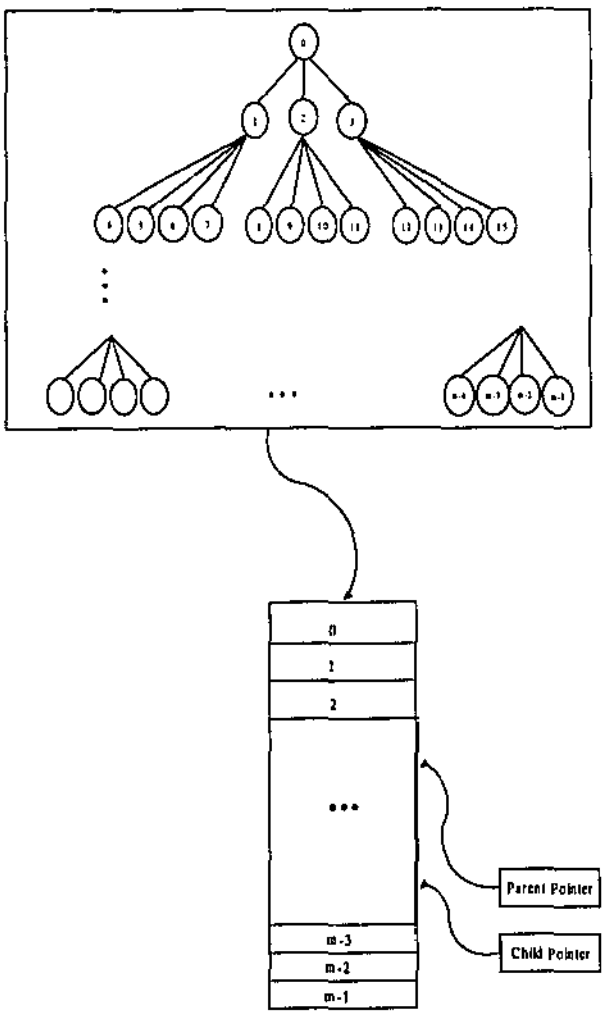



Figure 4.2 One to one mapping of the coefficients in a tree to a memory bank

transformed so that there are *k* coefficients in the lowest frequency subband then the number of trees is *k* and so each tree size $m = n^2/k$.

As figure 4.2 shows these *m* coefficients are then mapped into a memory bank, from 0 to *m* -1. The main parent goes to the index 0 memory element followed by its three children which are in turn followed by their children and so on. The coefficient number in the tree and the indices of the memory bank has one to one correspondence. For example the coefficient number 0 goes into the memory element whose index is 0, coefficient 1 goes into the memory element 1 and so on. The ordering of the coefficients is dictated by the scanning order that was shown in figure 3.3 in chapter 3.

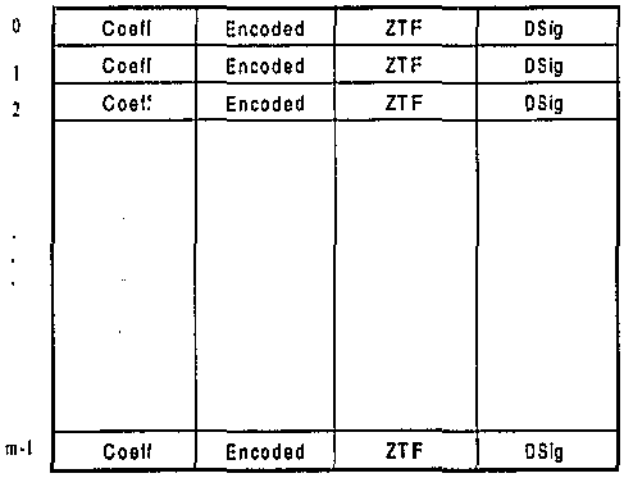



Figure 4.3 The memory bank with the four fields

### 4.2.2 Fields of a Memory Element

Each element of the memory bank is not just a single field containing the coefficient alone, but a record of four fields. The fields are Coeff, DSig, ZTF, and Encoded. The memory bank actually looks as shown in figure 4.3. The fields facilitate the encoding process by containing important information with regard to the coefficient. The functions and meanings of these fields are as follows:

- The Coeff field where the coefficient is actually stored.

- The DSig field is used to store information to indicate if any of the descendents of the coefficient is significant with respect to a given threshold. A binary '1' is to indicate 'yes' and the '0' is for 'no'.

- The ZTF field is used to store the binary information to indicate if ancestor or parent of the coefficient has been found to be a zerotree root. In other words it is used to indicate if the coefficient is an element of a zerotree. Here too '1' is to indicate that the coefficient is an element of the zerotree while a '0' indicates that it is not.

- The Encoded field is also used to store binary information to indicate if the coefficient has already been encoded. A '1' is meant to indicate yes and a '0' for no'

### 4.2.3 Pointers

As shown in figure 4.1 we use two pointers called the parent pointer and the child pointer. As the names indicate, the parent pointer is used to point to the parent coefficient while the child pointer is used to point to the child coefficient.

## 4.3 Choice of Thresholds

The specification for the encoder is for an 8-bit implementation. As such the coefficients can vary from −128 to 127. So the absolute value of the coefficients vary from 0 to 128. Instead of choosing the first threshold $T_i = |X_j|/2$ where $|X_j|$ is the maximum of all coefficients, we choose 64 as the first threshold as it is half of the maximum possible threshold. The other thresholds then become 32, 16, 8, 4, 2 and 1. This choice of thresholds leads to a very simple and effective way of encoding the coefficients, as we will see later. It is especially useful when performing successive approximation.

Subsequently we start encoding using the first threshold, 64. Then encode the whole tree against 32 then for 16 and so on until we have finished coding against threshold value 1.

## 4.4 Encoding the Coefficients

The encoding of the coefficients against any threshold (T) is achieved by three operations. These three operations are:

- Significance Map Generation

- Assignment of Codes, and
- Successive Approximation

Figure 4.4 shows these three steps, for encoding for a threshold. As it is obvious significance map generation is the first one performed. After it has been completed, the two remaining steps, assignment of codes and successive approximation quantization iterate to encode all the coefficients for the given threhold.

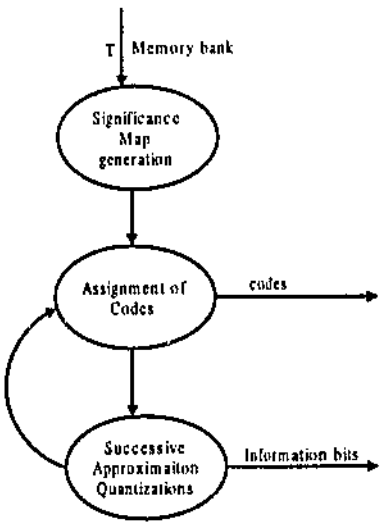



Figure 4.4 Encoding coefficients of memory bank for threshold T

### 4.4.1 Significance Map Generation

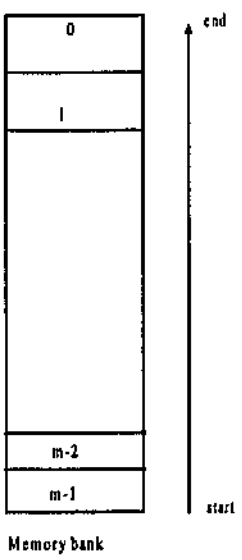



Figure 4.5 Significance map generation of the memorybank
Showing direction of processing

Significance map generation is the first thing that we do in encoding the coefficients. By performing this process we collect information about the significance of the descendents of each coefficient. This information is used when codes are assigned to the coefficients.

We start from the bottom of the memory bank and move our way up as shown in figure 4.5. The child pointer is made to point to the last coefficient $(m - 1)th$ coefficient, that is, the child pointer contains the value $m$-1. The parent pointer is made to point to the last parent, which is the parent of the coefficient that the child pointer is pointing at. The last parent is the last coefficient of the first one-fourth of all the coefficients in the tree. Therefore

$i = (m/4) - 1$ = index of the last parent.

With the parent pointer pointing at the last parent in the tree and the child pointer pointing at its last of the four children we do the following:

The absolute value of the coefficient pointed to the child pointer is compared to the threshold. If $|Coeff| \geq T$, i.e. significant then a '1' is written in the DSig field of the parent coefficient. Since the descendent significance for that parent coefficient is determined there is no need to check for the significance of other three children. So we move to the next parent by decrementing the parent pointer by one.

```
For i in 0 to 3 loop
    If abs Menbnk(child + i).Coeff ≥ T then  -- Membnk is memory bank
        Membnk(parent).DSig := '1';          -- child is child pointer to the last child
        Eyesno:= '1';                        -- was the loop exited or not
        Exit;                                -- parent is pointer
    End If;
End loop;
If EYesNo = '0' then
    Membnk(parent).DSig := '0';
end If;
parent := parent - 1;
child := child - 4;
```

The child pointer is decremented by 4 to point to the last child of the new parent. We are assuming that we are dealing with parent coefficients that have only one level of descendents, i.e. they only have children. This means that the children themselves do not have children in their turn or they are the leaves of the tree. For parent coefficients with only one level of children

$DSig_{Parent}$ = Significance(child1 **OR** child2 **OR** child3 **OR** child4)

In this case a possible pseudo-code would look like:
If the child coefficient is found to be insignificant, the significance of the next child and the other children is checked until we find a significant child. When all the four children are checked and if all of them are found to be insignificant, a '0' is written in the DSig field of the parent coefficient. This is repeated for all the parents that have only one level of descendents.

When we reach higher level of parents, that is coefficients that have more than one level of descendents, their DSig field is determined by the significance of both levels of descendents. In other words, the DSig of the parent is determined by the DSig fields and as well as the significance of all its four children. As a result we need to check both the DSig fields and the children coefficients.

If we find either DSig field containing '1' or the coefficient to be significant for any one of its four children a '1' is written in the DSig field of the parent coefficient. We then proceed to determine the descendent significance of the next parent by decrementing the parent pointer by 1 and the child pointer by 4.

If DSig field of all the four children contain a '0' and if all the children are insignificant (the absolute values of all the children are less than the threshold T), then a '0' is written in the DSig field of the parent coefficient and we proceed to determine the DSig field for the next parent. For parent coefficients with more than one level of children

$$DSig_{Parent} = DSig(child1\ OR\ chidl2\ OR\ child3\ OR\ child4) \\ OR\ Significance(chidl1\ OR\ child2\ OR\ child3\ OR\ child4)$$

In this way we continue to determine the DSig of the coefficients till we determine the DSig of the main parent. We must remember that the main parent has only three children, so the child pointer must be decremented by 3 when we move to the main

```
For i in 0 to 3 loop
    If (abs Membnk(child + i).Coeff ≥ T OR Membnk(child+i).DSig = '1' ) then  -- Membnk is memory bank
        Membnk(parent).DSig := '1';          -- child is child pointer to the last child
        Eyesno:= '1';                        -- was the loop exited or not
        Exit;
    End If;
End loop;
If EYesNo = '0' then
    Membnk(parent).DSig := '0';
end If;
    parent := parent - 1;
    child := child - 4;
```

parent, instead of decrementing by 4 as was the case with other parents.
The pseudo-code we could do something like this:

We realize that there is a need for us to differentiate between the leaves and the higher level coefficients. To this effect the DSig fields of all the coefficients are initialized to 'u' (unknown) prior to significance map generation. So before we check the significance of the child we check if the DSig field of the child contains a 'u'. If it does, then it is a leaf coefficient. The DSig fields of parents get written with '0' or '1' before they are pointed to by the child pointer. So if the DSig of child coefficient does not contain a 'u' then it is not a leaf.

At this point the memory bank contains enough information so that we can start assigning codes to the coefficients.

The Flow chart for determining the DSig field of a parent coefficient is shown in Figure 4.5.1.

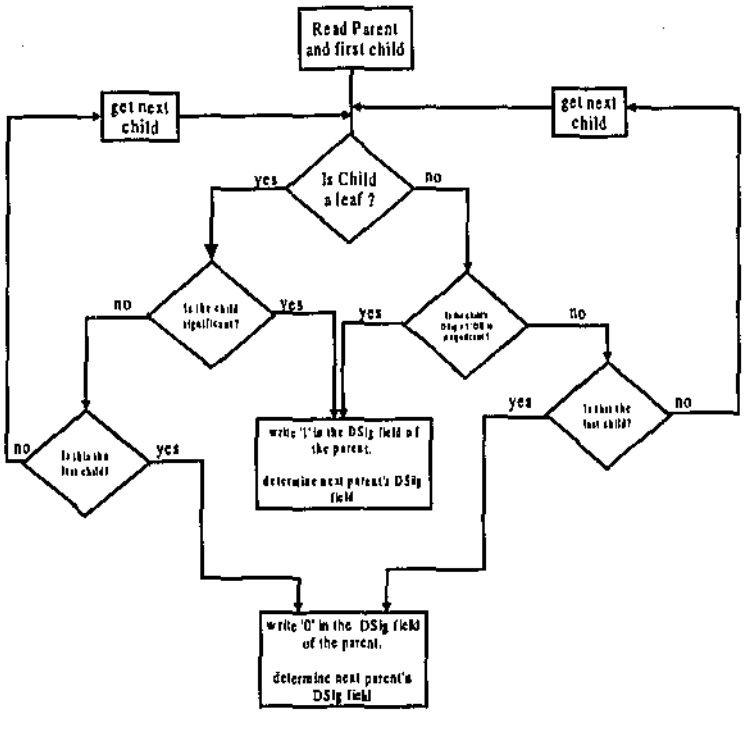



Figure4.5.1 Flow chart to determine the DSig field of a parent coefficient (significance map generation)

### 4.4.2 Assignment of Codes

Once the significance map generation is completed for a given threshold the memory bank contains all the information required for the assignment of codes to begin for that threshold.

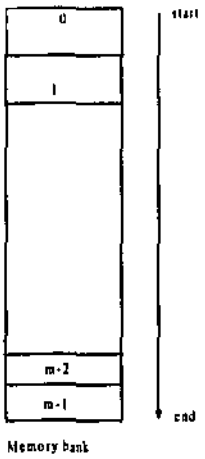



Figure 4.6 Assignment of Codes
Showing direction of processing

As discussed in chapter 4 four different codes are assigned. Once again, they are:

- ZTR, for zerotree root
- POS, for positive significant
- NEG, for negative significant, and
- IZ, for isolated zero

To assign the codes we start from the main parent and move downwards as figure 4.6 shows. Here too we use our child pointer and the parent pointer. While the parent

pointer points to the parent, the child pointer points to the first child of the four children (three in case of the main parent)

We use an array to store the codes for the coefficients and information about the approximate values in case of significant codes. This is the array that will contain the embedded codes for the coefficients of the memory bank at the end of the encoding process. In other words this is the array that will contain the result of the encoding.

The coefficient that is pointed to by the parent pointer is the one that is assigned code to. To code a coefficient its absolute value is compared to the threshold.

If the coefficient is significant, it sign is determined by comparing it with zero. If it found to be positive, a POS is stored in the output array and the array index is incremented by one. If it is found to be negative significant, a NEG code is stored in the output array and the array index is incremented by one. Whenever a POS or a NEG is coded, a '1' is written in the Encoded field of the coefficient, to indicate the coefficient has been coded as significant so that future coding of this coefficient does not take place. A pseudo-code for coding significant code is:

```
(when significant)

If parent.coeff < 0 then      -- parent is the element pointed to by the parent pointer
   CodeRA(n) := NEG;          -- CodeRA is the array to where we store the codes
Else
   CodeRA(n) := POS;
End If;
n := n+1;
parent.Encoded := '1';
-- perform successive approx
parent.coeff := 0;
```

The assignment of a significant code is followed by successive approximation, which is discussed in the next section. After successive approximation is completed the coefficient is replaced by a zero. We then move to assign code for the next coefficient by incrementing the parent pointer by 1 and incrementing the child pointer by 4 (3 when the parent being incremented is the main parent).

If the coefficient is found to be insignificant for the threshold at hand we proceed to find out if it is a zerotree root or (ZTR) or an isolated zero (IZ). This is the time when our significance map proves useful. At this point go back to the significance map and check the DSig field of the parent coefficient. Any of the three conditions would be satisfied.

a. If the DSig field contains a '1' it means that at least one of its descendents is significant, so it is an isolated zero. Subsequently an isolated zero code (IZ) is stored in the output array and the array is incremented by one. We then proceed to determine the code for the next coefficient by incrementing the parent pointer by one and the child pointer by 4 (3 when the parent being incremented to is the first parent).

b. If DSig field contains a '0' instead of a '1', it means that none of its descendents is significant. So the coefficient is a zerotree root (ZTR). Therefore a ZTR code is stored in the output array and the array is incremented by one. A '1' is then written in the ZTF fields of all the four/three children of the parent coefficient.

$ZTF_{child} = ZTF_{Parent}$

In this way the ZTF information gets passed onto from parents to their children which in turn gets passed to children of the children until it reaches the leaves. This information is used to make sure that we do not code the descendents of ZTR as they are all insignificant and need not be coded.

c. If the DSig field contains a 'u' then it is a leaf. A leaf insignificant is also coded as a ZTR. When decoding we can easily distinguish a leaf coefficient from other coefficients.

A pseudo-code for encoding an insignificant coefficient is:

```
(when insignificant)

If parent.DSig = '1' then       -- parent is the element pointed to by the parent pointer
   CodeRA(n) := IZ;             -- CodeRA is the array to where we store the codes
Elsif parent.DSig = '0' then
   CodeRA(n) := ZTR;
   Children.ZTF := '1';         -- for all the children
Else                            -- parent.DSig = 'u'
   CodeRA(n) := ZTR;
End If;
n := n+1;
parent := parent + 1;
child := child + 4;
```

There are two things we need check before we begin to assign code to a coefficient.

The first one is that we should not code a coefficient that has already been coded as significant. The Encoded field is used for this purpose. As we saw above, a '1' is written in the Encoded field of the coefficient that has been coded as significant. Once written this information is preserved for the rest of the encoding process. So before going on to determine the code for a coefficient its encoded field is checked. Only if a '0' is found in

this field we proceed with the next step of assigning code to the coefficient. Otherwise we go to determine the code for the next coefficient

The second thing is to check the ZTF field. If a '1' is found in the ZTF field of the coefficient then it is a descendent of a ZTR, so it need not be coded. Only if a '0' is found in the ZTF field of the coefficient do we proceed with determining the code for the coefficient at hand.

So we don't code a coefficient if either of the ZTF and Encoded fields contains a '1'.

If the main parent is found to be a zerotree root then all other coefficients are insignificant for the threshold at hand and there is no need to code any further for that threshold. We then proceed to encode for the next lower threshold.

In this manner we assign codes to coefficients in the memory bank for a given threshold.

The flow chart for assigning a to a coefficient is shown in figure 3.8 in the last chapter.

### 4.4.3 Successive Approximation Quantization

Successive approximation quantization is the process by which information on the values of the coefficients is embedded with the codes instead of passing the whole coefficient itself. When we have finished discussing this topic we can appreciate how the precision of the coefficient values are improved with each threshold level codes. If the decoder is provided with all the codes then it can reconstruct the exact coefficient values. Otherwise it will only be able to reconstruct an approximate value of the coefficient depending on the number of the codes it receives. Let's see how this happens.

Given the size of the tree it is possible to tell against which threshold a code has been coded. That is given a code, say POS, we can tell whether it has been positive significant against 64 or 32 or 16 or other threshold values. This information is implicitly contained in the encoded codes.

| Coefficient | reconstructed value from Knowledge of Threshold |
|---|---|
| 128 >= \|Coefficient\| >= 64 | ± 96 |
| 63 >= \|Coefficient\| >= 32 | ± 48 |
| 31 >= \|Coefficient\| >= 16 | ± 24 |
| 15 >= \|Coefficient\| >= 8 | ± 12 |
| 7 >= \|Coefficient\| >= 4 | ± 6 |
| 3 >= \|Coefficient\| >= 2 | ± 3 |
| \|Coefficient\| = 1 | ± 1 |
| Coefficient = 0 | 0 |

Figure 4.7 Coefficients and their reconstructed values from knowing the thresholds

With the thresholds we use, 64, 32, 16, 8, 4, 2, and 1, and using the centre of the uncertainty interval as the reconstruction value the coefficients can be reconstructed as shown in figure 4.7

In order to be able to reconstruct the exact original coefficients from the codes the knowledge of threshold itself isn't enough. So we need to send additional information on the coefficient value with the codes. This is what successive approximation quantization exactly does.

Whenever a POS or NEG is coded during the assignment of codes we subtract the threshold value from the absolute value of the coefficient. We then store the value of the remainder in the two dimensional array and send the information on this remainder to

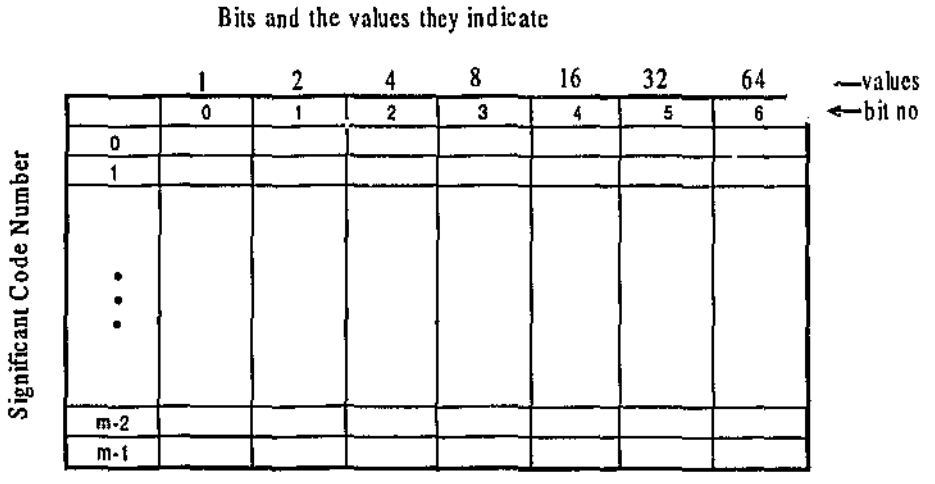



Figure 4.8 The two dimensional array for storing information ont the coefficient values 7 bits for each coefficient

the decoder. Since the maximum absolute value of a coefficient can be 128 and since the max threshold is 64 we can have a maximum remainder of 64 (128–64). As a result we use 7 bits for each coefficient. So the two dimensional array becomes $m$-1 × 7 of bits as shown in figure 4.8. In figure 4.8 the codes are numbered and these codes are the significant codes (POS or NEG) number. The numbering is such that code number 0 is the first significant code encoded, code 1 the second one and so on.

Further explanation can be more effectively achieved by taking an example.

Let us say that we started our encoding with threshold 64 and got a POS as the first significant code. Let us further assume that this coefficient has a value of 120. As soon as we code this POS we build the additional information. The remainder is (120-64 = 56). This 56 can be broken down into (56 = 32 + 16 + 8). Because it is the first significant code we store this information in the first row of the 2 dimensional array. We write a '0' in the bit places corresponding to 64, 4, 2 and 1 and write a '1' in the bit places corresponding to values 32, 16 and 8. After this the two dimensional array looks like shown in figure 4.9.

Bits and the values they indicate

| Significant Code Number | 1 | 2 | 4 | 8 | 16 | 32 | 64 ←values |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 ←bit no |
| 0 | '0' | '0' | '0' | '1' | '1' | '1' | '0' |
| 1 | | | | | | | |
| • | | | | | | | |
| • | | | | | | | |
| • | | | | | | | |
| m-2 | | | | | | | |
| m-1 | | | | | | | |

Figure 4.9 The two dimensional array with information for the first significant code

Since our threshold is 64 and a remainder of 64 is possible we store the bit information contained in the bit place corresponding 64 right after the code in the output code array. Here right after we store POS, '0' is stored in the output array.

Let us assume that our next significant code is a NEG and assume that the coefficient is – 85. Obviously this is also against threshold 64. So the remainder of the coefficient for which we need to provide additional information is (85 –64 = 21). 21 can be written as

(16 + 4 + 1). Since this is the second significant code we store the information in the seven bit positions of the second row in the two dimensional array. A '1' is written in the bit positions corresponding to the values 16, 4 and 1 and a '0' is written in other bit positions. Similarly right after the code NEG, we store a '0', for 64, in the output code array since the threshold is still 64 and a remainder of 64 is possible.

Now let us say that our third significant code is a POS and the coefficient is value is 47. Obviously the threshold is 32, since it is less than 64 and greater than or equal to 32. The remainder is (47-32 = 15). 15 can be written as (8 + 4 + 2 + 1). Like before we store this information in the third row of the 2 dimensional array, storing '1' in the bit positions corresponding to 8, 4, 2, and 1 and '0' in bit position corresponding to 16. Since the threshold here is 32 when we subtract 32 from |coefficient| the maximum remainder that we will get is 31. This is because when encoding has been finished for threshold 64, the maximum absolute value of the unencoded coefficients will be 63. Otherwise it will have been significant against threshold 64 and coded as significant earlier. So we do not need to write anything in the bit positions corresponding to 64 and 32 in the 2 dimensional array.

Bits and the values they indicate

| Significant Code Number | 1 | 2 | 4 | 8 | 16 | 32 | 64 | ←values |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | ←bit no |
| 0 | '0' | '0' | '0' | '1' | '1' | '1' | '0' | |
| 1 | '1' | '0' | '1' | '0' | '1' | '0' | '0' | |
| 3 | '1' | '1' | '1' | '1' | '0' | | | |
| • • • | | | | | | | | |
| m-1 | | | | | | | | |

Figure 4.10 The two dimensional array with information for the first significant code

For threshold less than 64 the maximum |coefficient| is 2T – 1; T being the threshold. The maximum remainder is (2T–1) –T = T–1. So we do not have bit information corresponding to values >= T. Hence we do not need to write anything in those bit positions of the 2 dimensional array. Also we do not need to provide information on the approximate values in these bit positions. Figure 4.10 shows our 2 dimensional array after successive approximation qunatization of this third significant has been completed.

There is an additional thing that we do when we code the first significant for lower thresholds than 64. Right after the code is stored in the output code array the bit information in the two dimensional array corresponding to the current threshold value for all the significant codes encoded for higher threshold values are stored in the output code array. In our example when we store the third code (POS for 47) it is the first significant code for 32. So after storing the code we look at the bit positions corresponding to the value 32 in the 2 dimensional array for already coded coefficients for 64 (row 0 and 1). Referring to our two dimensional array in figure 4.8 we find a '1' for row 0 and '0' for row 1. So we store these bit information in the output code array right after the POS code is stored. For the other codes, for the same threshold 32 we only need to store the code and do not need to send any information on the coefficient values. However we still have to fill the two dimensional array for the code with the additional information for the coefficient value to be sent when lower level thresholds are encoded.

At this point, the output code array according to our example would look like the one shown in figure 4.11

To generalize, whenever we code the first significant for a threshold (less than 64) we send the bit information on that threshold for all the codes that have been coded significant against higher thresholds. That is, when we code for the first significant for threshold 16 we store the code in the output array and then store all the bits

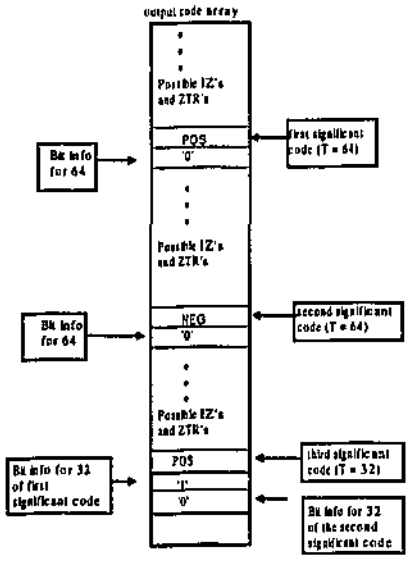



Figure 4.11 The output code array after three significant code coding from the example

corresponding to value 16 in the 2 dimensional array for all the significant codes that have been coded for 64 and 32. Similarly when we code the first significant for threshold 8, we store the code and then store all the bit information corresponding to value 8 in the 2 dimensional array for all previously coded significant codes for 16, 32 and 64.

Similarly we code for the rest of the thresholds.

We observe that with each lower level of threshold more precision information is added as we code for each lower level threshold. Finally when we have coded for the last threshold 1, the output array has enough information so that the decoder can be able to reconstruct the original coefficient.

At this point the encoding is completed.

### 4.4.4 Updating the DSig, ZTF, Encoded and Coeff fields

The DSig fields are initialized to 'u' before significance map generation starts. That is it is initialized to 'u' at the start of coding for every threshold.

The ZTF fields are also initialzed to '0' at the beginning of coding for every threshold.

The Encoded field is initialized only once, at the beginning of the encoding process, to '0'. Once set it remains set for the remainder of the encoding process.

The Coeff field is set to 0 once the coefficient has been coded as significant. This prevents the significant coefficient from preventing the occurrence of zerotree roots.

### 4.4.5 Summary

Encoding is achieved through three operations: significance map generation, assignment of codes and successive approximation.

Starting with threshold 64 we first generate the significance map. Then we assign codes and perform successive approximations as required for each coefficient and repeat code generation and successive approximation till we have coded for the last coefficient in the memory bank. Encoding is completed for threshold 64.

We then code for the next lower threshold (here 32) by repeating the above process, then for 16, then for 8, then 4, then 2, and finally with 1.

This way when the last threshold has been coded for, the encoding is complete. We get in our output array the codes and the information on the coefficient values as the result of encoding.

## B. Decoder Design

### 4.5 Introduction

At the end of the decoding we will obtain the same number of coefficients that has originally been encoded, and in the same order as we found them before encoding in part A.

### 4.6 Assumption on Codes

We assume that the codes to be decoded are stored in an array, like the result of the encoding described in part A. In practical application the codes can be reaching the decoder one after another in real time, with the most significant codes (those coded for higher thresholds) first followed by the less significant ones. To repeat what have been already said in chapter 3, codes that are coded for higher threshold values are considered more significant than those coded for lesser threshold values. With our array assumption, the most significant codes are stored in the beginning followed by less significant ones. For the purpose of explanation we call this array the code array.

### 4.7 Architecture

In the decoder too we use a record of four fields to store vital information. The fields are Coeff field, ZTF field, Decoded field and the Sign field. The Coeff field is used to store the decoded ceofficient while the other three are used to store information about the coefficient to facilitate the decoding process. Since the size of the tree was *m*, we use an array of size *m* (from 0 to *m*-1) of the four-field record. The array of record is shown in figure 4.12.

The four fields and their meanings are:

- Coeff: The Coeff field is used to store the decoded coefficient

- ZTF: The ZTF field is used to store binary information to indicate whether a zerotree root has already been found. A '1' in the field is to indicate that the coefficient is an

element of a zerotree. That is, an ancestor of it has been found to be a zerotree root. A '0' indicates otherwise.

- Decoded: The Decoded field is used to store binary information to indicate whether or not the coefficient has been decoded already. A '1' means 'yes' while a '0' means 'no'

- Sign: The Sign field is used to store the sign of the coefficient. '1' indicates that the sign is negative while '0' indicates that it is negative.

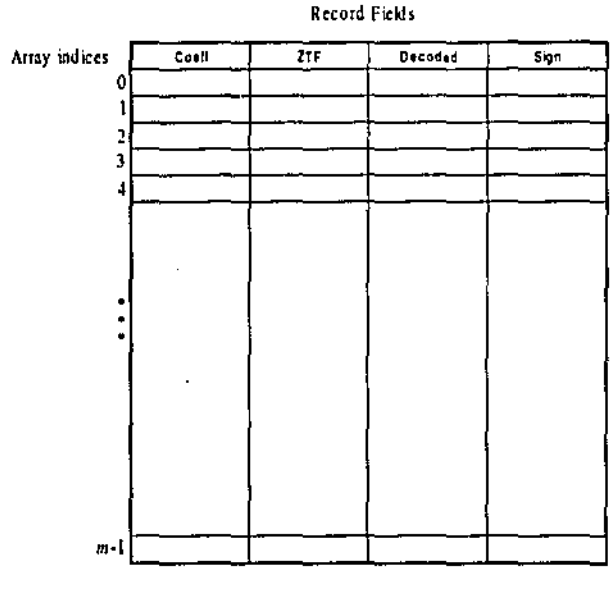



Figure 4.12 Array of record fields, used for decoding

As in the encoder, here too we make use of two pointers: a parent pointer and a child pointer. We also use an array to store the indices of the coefficients that have already

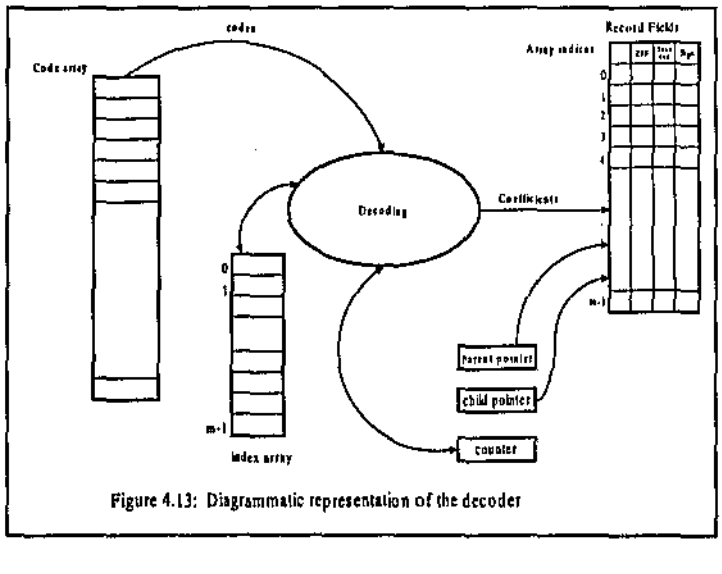



Figure 4.13: Diagrammatic representation of the decoder

been decoded. A counter is also used to store the number of codes that have been decoded. A diagrammatic representation of the decoder as a whole is shown in figure4.13.

## 4.8 Decoding Process

The decoding is dictated by the encoding. As such it is a matter of how elegantly the reverse of encodign can be performed. The following paragraphs shows how we achieve this.

### 4.8.1 Preparation

At the beginning of the decoding process all the four fields of the record array are initialized. The Coeff field is given a value 0 and the ZTF, Decoded and Sign fields are all set to '0'.

Since the record array is to contain the coefficients in the exact order as they were before encoding, the same parent-child relationship holds between indices of the array in which the decoded coefficients will be stored. That is, in the record array, the Coeff field of index 0 will contain the main parent, Coeff field of indices 1,2, and 3 will contain the three children of the main parent, and Coeff field of indices 4, 5, 6 and 7 will contain the four children of coefficient in index 1, and so on.

The parent pointer is made to point to the main parent and the child pointer is made to point to the first child of the main parent.

*Parent Pointer =0;*
*Child Pointer = 1;*

The decoding is done for the coefficient pointed to by the parent pointer. We start from the main parent and move downward through to the bottom of the record array. First we decode for the highest threshold, 64, then for 32, then for 18 and so on. For each threshold the following steps are involved

### 4.8.2 Checking the ZTF and Decoded Fields

Before we read the code from the code array to decode, we check the Decoded and the ZTF fields of the parent (index pointed to by the parent pointer) of the record array which will contain the decoded coefficients. The flow chart for performing this check is shown in figrure 4.13.1

If the Decoded field contains '1' it means that the coefficient has already been decoded for. So we move to the next parent by incrementing the parent pointer by one and the child pointer by four (three if the current parent is the main parent). If the Decoded field contains a '0' instead then we check for the ZTF field

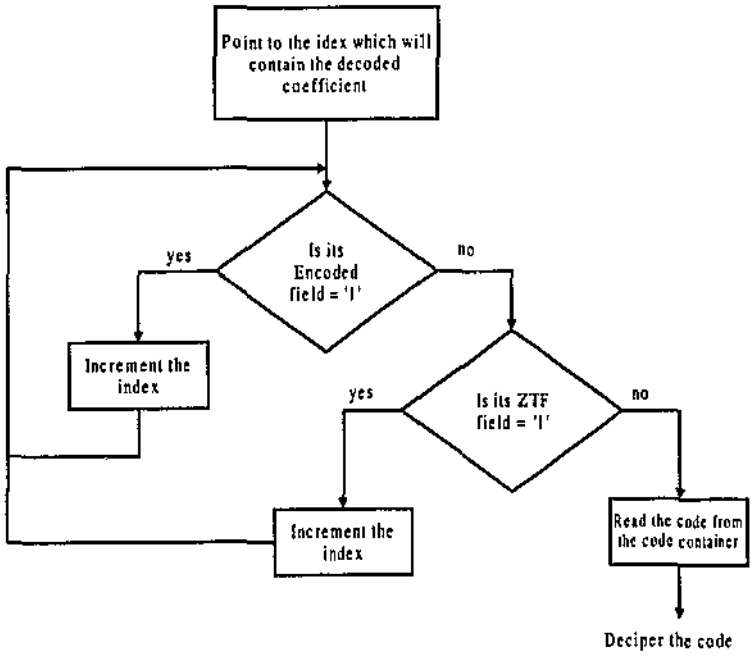



Figure 4.13.1 Flowchart for Checking the Decoded and ZTF fields before reading the code form the code container

If the ZTF field contains a '1', it means that it is an element of a zerotree root found earlier, so we do not have to decode for this coefficient as there is no code for this coefficient for the current threshold. The zerotree found ahead message is then passed onto its children by writing '1' in the ZTF field of all its children. We then move to the next coefficient by incrementing the parent pointer by one and the child pointer by four (three if the current parent is the main parent). The checking operation could translate into the a possible pseudo-code:

```
If parent.Decoded = '1' then
        parent pointer := parent pointer + 1;
        child pointer := child pointer + 4 (or 3);
Else
        If parent.ZTF = '1' then
             For i in 0 to 3 (or 2) loop
                  child(i).ZTF := '1'; -- the child pointer gives the index of the first child
             End Loop;                 -- child(0) is the first child child(i) is the second child and so on
             parent pointer := parent pointer + 1;
             child pointer := child pointer + 4 (or 3);
        Else
            -- Read the code ...

       End If;
End If;
-- repeat the process
```

### 4.8.3 Deciphering the Code and Reconstructing the Coefficients

If both the Decoded field and the ZTF field are found to contain '0' then the code that is about to be read from the code array is for the coefficient pointed to by the parent pointer. So the code is read from the code array and decoded.

The flow chart for decipdhering a code and reconstructing the coefficients is shown in figure 4.13.2

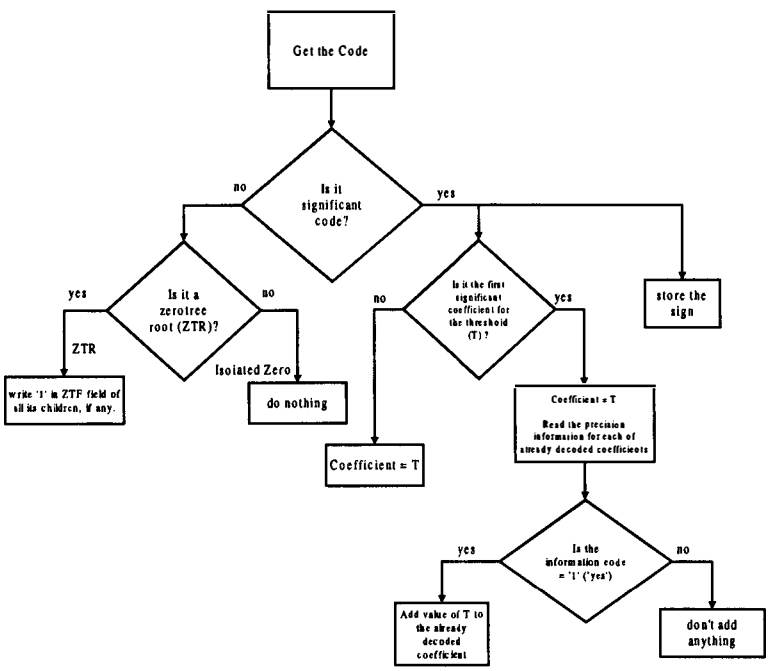



Figure4.13.2 Flow chart for Deciphering a code and Reconstructing the coefficients

The code can be any of the four different codes: NEG, POS, IZ and ZTR. We process each of them as follows:

a. If the code is a ZTR there are three cases that can arise.

If the parent pointer is pointing to the main parent, the rest of the coefficients are insignificant against the current threshold. There are no codes for them in the code array for the current threshold. So we move to decode for the next threshold level. And start again from the main parent by setting the child pointer and the parent pointer are set as

*parent pointer := 0;*

*child pointer := '1';*

and start the process all over again, but for a the next level of threshold value. Actually the parent and child pointers are unchanged. For example if threshold that we found the main parent as the ZTR was 64, the next threshold is 32.

If the parent pointer neither points to the main parent or to a leaf coefficient, then the ZTF fields of its four children are set to '1', and we move to decode for the next coefficient by incrementing the parent pointer by 1 and the child pointer by four.

An equivalent pseudo-code code would be:

```
( When code is ZTR)

If parent = main parent then
        --change to the next threshold
        -- read the next code
elsIf parent /= leaf then
        for i in 0 to 3 loop
             child(i).ZTF := '1';
        end loop;
        child pointer := child pointer + 4;
        parent pointer := parent pointer + 1;
else    -- leaf coefficient
        parent pointer := parent pointer + 1;
end if;

-- decode for the next coefficient
```

If the parent pointer points to the leaf coefficients we do not need to do anything. We increment the parent pointer by 1 and go to decode for the next coefficient. Note that we do not increment the child pointer because the coefficient pointed to by the parent pointer does not have any child as it is a leaf coefficient itself.

b. If the code is an isolated zero (IZ) the coefficient is insignificant, so we move to decode for the next coefficient by incrementing the parent pointer by 1 and the child pointer by 4 (or 3 if the current parent is the main parent).

c. If the code is a POS, the coefficient is significant with respect to the current threshold. This means that the absolute value of the coefficient had been greater than or equal to the current threhold. So we add the threshold value to the value contained in the Coeff field (Coeff := Threshold). We then set the Decoded field to '1' to indicate that the coefficient has been decoded. Another thing that we do here is to store the array index of this decoded coefficient in the index array that is indicated in figure 4.13.

To repeat, the index array is used to store the array indices of the decoded coefficients. The indices are stored in order, first the index of the coefficient that was decoded first, then that of the second, then that of the third and so on. The contents of this index array serve as pointers to the coefficients when we decode for lower thresholds to construct more precise coefficients that are decoded for higher thresholds. Since we have initialized the sign fields to '0' which indicates a positive coefficient we do not need to set the sign field.

If the current threshold is 64 then the next code in the code array is the precision information for decoded coefficient. It is read, and if a '1' is found a precision of 64 (Coeff:= Ceoff + 64) is added to the coefficient, and no precision is added if a '0' is found.

If the threshold is lower than 64 (i.e. 32, 16, 8, 4,2 and 1), and if the coefficient decoded is the first for the threshold, then the codes in the code array after the currently decoded POS are not codes but information on the precision of the coefficients that have been decoded before. There are as many bits following the currently decoded code as the number of already decoded coefficients. The number of decoded coefficients is contained in the counter. The first bit information is for the first decoded coefficient, the second bit information for the second decoded coefficient and the third bit for the third decoded coefficient and so on. A '1' means that the corresponding coefficient has an additional precision value of T

(current threshold) so T is added to it. A '0' means that the corresponding coefficient does not have additional T precision, so we do not add T to it. The information for the number of bit information to be read is given by the counter, while the pointers to the decoded coefficients are given by the index array.

An equivalent pseudo-code for this would be:

```
( When POS)

parent.Coeff := T;  -- T is the threshold
If T = 64 then
     Read the next code (bit information) from the code array
     If bit info read = '1' then
          parent.Coeff := parent.Coeff + 64;
     Else
          Null;
ElsIf Coefficient decoded = the first for the threshold then
    add precision (by T) to the other already decoded coefficients
Else
    Null;
End If;

parent pointer := parent pointer + 1;
child pointer := child pointer + 4;  (if parent is not a leaf)
```

d. If the code is a NEG, it is also a significant coefficient so we do exactly the same things that we do when the code is POS, except for the sign field. A '1' is stored in the Sign field to indicate that the sign is negative.

In this way we first decode for T = 64 then for T = 32, then 16, then 8 and so on until we have decoded for 1. Note how precision is added with the decoding for each lower threshold. When we have finished for threshold = 1 we will have got exactly the same coefficients (in the Coeff field of the record array) and in the same order as they were before they were encoded.

In above we have not discussed the issue of bit truncation. If the bitstream is truncated we need to add additional precision value to all the decoded coefficients. How much to be added depends on where the truncation occurs.

For example, if the current threshold is 32 and the last code obtained is a POS and this POS is not the first significant coefficient for the current threshold (32), It may or may not have an additional (64-32) = 32. So the uncertainty of 16 is added to it. So it is reconstructed to 48 (32 + 16). All the previously encoded coefficients are also given an additional value of 16.

If this POS is the first significant for the current threshold it is itself given an additional value of 16 (center of uncertainty). For those decoded before it, the uncertainty is the center of 64 and 128, so 32 is the center of uncertainty. As such all those coefficients are added by 32.

To generalize when truncation occurs, the centre of uncertainty is added to the already decoded coefficients.

This completes the decoding.

### 4.8.4 Updating the Decoded, ZTF, Coeff and Sign Fields

The Decoded field is initialized to '0' only once in the whole decoding process. That is done at the start of the decoding. Once set it remains set throughout the process. The ZTF field is set to '0' at the start of every new threshold value. This is because a ZTF set to '1' is relevant only for the threshold that it was set.

The Sign field is set to '0' in the beginning to indicate positive. It is set to '1' when a negative coefficient is decoded, and it remains set throughout.

The Coeff field is set to 0 at the start. As the decoding happens it gets filled in with the real coefficients.

## C. Parallel Processor Architecture

### 4.9 Introduction

This architecture is not a completely different architecture to what we have discussed above. In fact it is only a slight adaptation of the single processor architecture, trying to make use of the advantage of inherent parallelism present in the zerotree of wavelet coefficients. A faster codec is thus envisaged from using this architecture. Bae and Prasana also propose this architecture.

### 4.10 Observation of Inherent Parallelism

If we look at the parent-child dependencies of the subbands in a zerotree we realize that there are three main branches, with the main ancestor common to all three of them. They are shown in figure 4.14 as branch_A, branch_B and branch_C. These three branches are each independent of the other, while the way in which coefficients within one branch relate among themselves is the same as the way in which the coefficients in another branch relate among themselves. This then points to us that if we have three separate but same processors we can process the entire tree using three such processors in parallel, each processor handling a main branch. This translates to a processor having to handle only one third of the coefficients it would have had to handle otherwise, thus promising a speedier codec. We can make use of this inherent parallelism in the design of the codec.

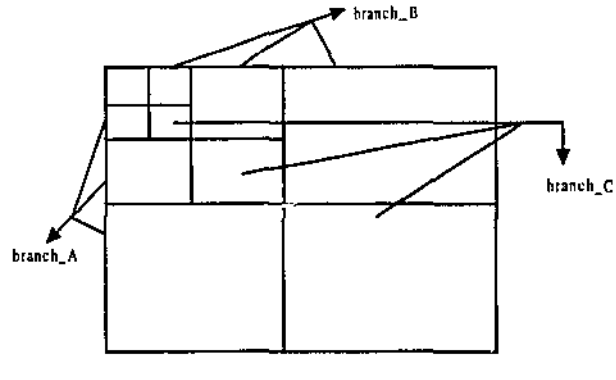



Figure 4.14 The three main branches of a tree

## 4.11 Architecture

The coefficients in each of the three branches are mapped into three memory banks as shown in the figure 4.15. The first coefficient of a branch is mapped into the element 0 of the memory bank, the 2nd coefficient is mapped into element 1, the 3rd into element 2 of the memory bank, and so on until the last coefficient is mapped into the last element of the memory bank. The main parent is mapped as the first coefficient in each of the three memory banks. So the first coefficient is the same for all the three memory banks.

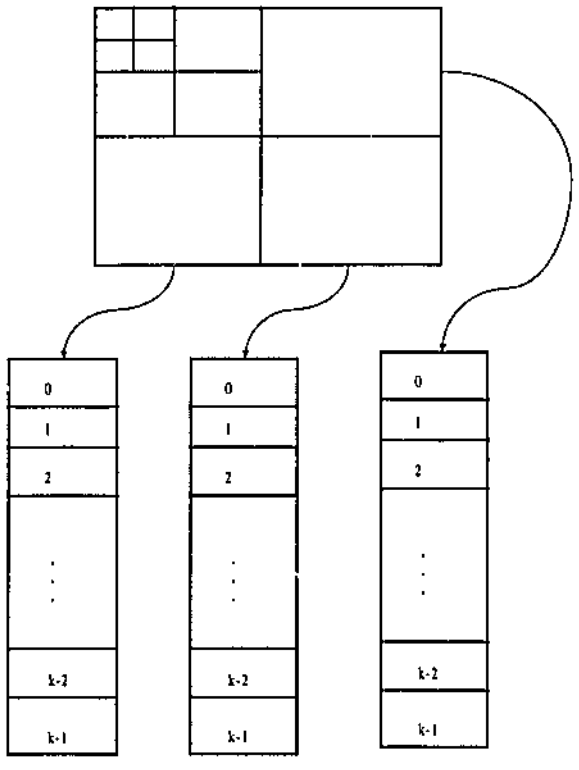



Figure 4.15 Mapping of the coefficients of three branches into 3 memory banks

Each memory bank contains one third of the total number of coefficients in the tree.

Having mapped the coefficients into the three memory banks each of the memory banks is encoded and decoded as exactly same as that for single processor encoder and decoder in part A and B above. And each of them can be processed in parallel by three identical processors.

Since the encoding and decoding strategies used are the same as in the single processor architecture they will not be discussed here again as it would just be unnecessary repetition. Instead the difference will be pointed out.

There is one important difference to note here. This happens in the encoder. When we have finished generating the significance map for a given threshold. The DSig value of the main ancestor (the first coefficient) as determined in a memory bank is not its true value. Being the main ancestor of all the three memory banks in three separate encoders, its descendent significance can be determined only after having information of its descendents in all the three memory banks. However the DSig field of the main ancestor in a memory bank contains all the information required from that memory bank, to determine the exact descendent significance of the main ancestor. If we can get together the DSig information for the main parent (first coefficient) from each of the memory banks then we will be able to determine the true DSig value of the main parent. The encoding can not proceed until the true DSig value is determined and passed to the three processors.

This is what we do. We have two other processes working on the other two memory banks in parallel. The content of the DSig field of the main ancestor in each memory bank is sent to a different processor (we call it DSig process here) to determine the exact descendent significance of the main ancestor. The actual descendent significance of the main ancestor is '0' only if all the DSig value from all the three encoder is '0'. If either one of them is a '1' then the DSig of the main ancestor is a '1'. In the processor where this actual DSig of the main ancestor is determined, the three DSig field values from three parallel encoders are ORed.

$$\text{True } DSig_{\text{main parent}} = DSig_{\text{main parent}}(A) \text{ OR } DSig_{\text{main parent}}(B) \text{ OR } DSig_{\text{main parent}}(C)$$

The result is the true descendent significance of the main ancestor for the particular threshold. This information is passed back to the three parallel encoders/processors. Once the true DSig value of the main parent is received, the next stage of assigning codes can begin. After sending the DSig value of the main parent as determined from the coefficients in a given memory bank to the DSig process, the processing is halted till it gets the true DSig value of the of the main parent from the DSig process. The data

transmission employs a simple handshaking mechanism. Figure 4.16 shows how these four processors relate.

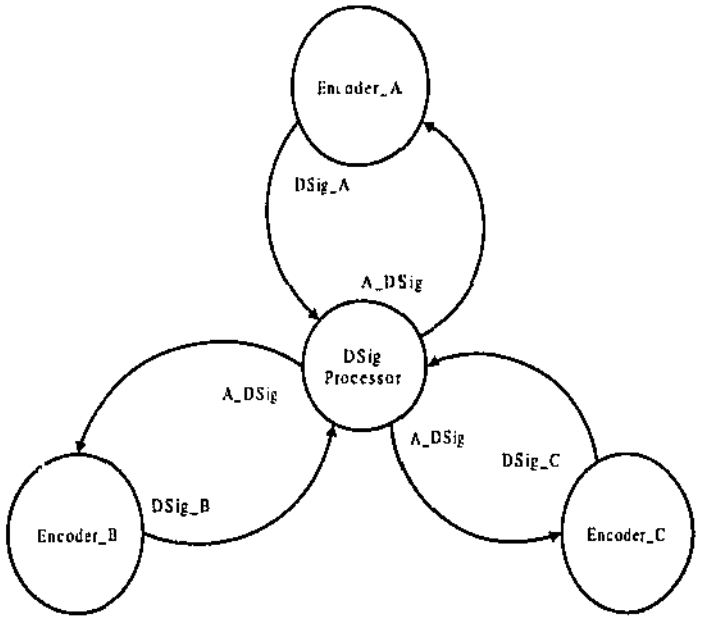



Figure 4.16 : The three encoders and the DSig processor

DSig_A, DSig_B and DSig_C are DSig of the main ancestor from the three encoders A_DSig (actual descendent significance) of the main ancestor determined by the DSig processor. Encoder_A, Encoder_B and Encoder_C are the three processors working in parallel

Besides this difference the encoding and decoding using the three parallel processors is the same as was described in the case of the single processor architecture. The obvious difference here is that instead of having to process (encode or decode) the whole tree the processors here process only a third of the coefficients. And owing to the nature of the EZW algorithm this parallel processing should give a very fast codec. At least three times as fast as the first one.

Another difference is that the first coefficient in each memory bank has just one child coefficient.

Decoding is exactly the same as the single processor except for the main parent having only one child in each memory bank. There is no communication needed between the decoders.

To conclude this chapter diagrammatic representation of the single processor architecture and the parallel processor codec are presented.

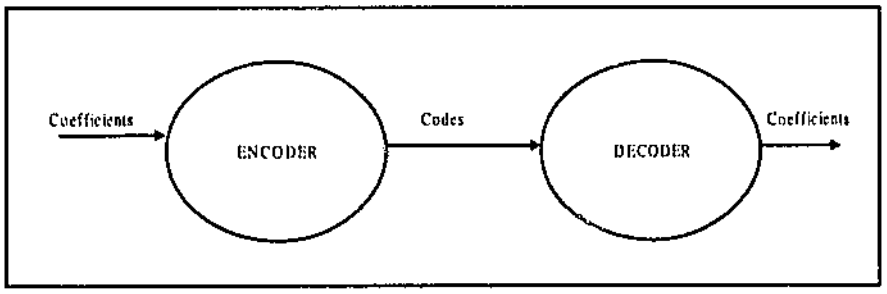



Figure 4.17 Single processor codec

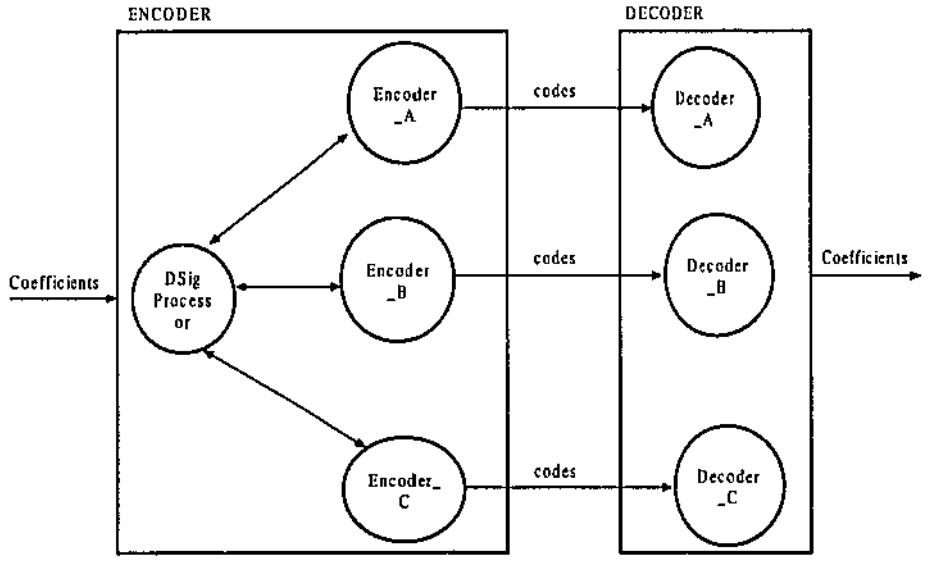



Figure 4.18 : The Parallel architecture codec

# 5. SIMULATION AND SYNTHESIS

## A. Simulation

### 5.1 Introduction

The VHDL coding was done at the behavioral level and the programs were simulated. Both the single processor codec and the codec with three parallel processors were simulated and results were verified.

With the parallel processor codec, the coefficients were input into the encoder and the resulting codes were fed into the corresponding decoder. It was checked if it gave the same coefficients as those input into the encoder.

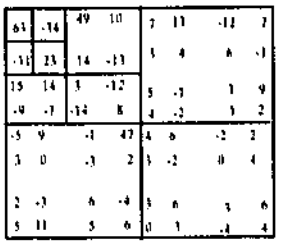
With the single processor codec extensive tests were performed. It was not only checked to see if the decoder reconstructed the same coefficients in the same order as that was input into the encoder, it was also checked to see how it produced more and more precise coefficients when the first number of codes given were increased.

All the tests were checked against those computed by hand, and it was found that the codec of both architectures, parallel processor and single processor, were working correctly as expected.

### 5.2 Test Data

The test data used was a standard data obtained from Shapiro's original paper for 8 × 8 coefficient. So the design was checked for 8 × 8 coefficients. There was another reason why this data set was chosen.

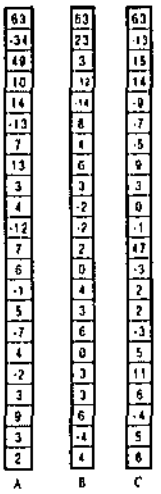
In keeping with the specification of the project, to assume static input data, data as contained in a memory bank in the required order, the interface was not designed as it would have hugely side tracked the project work. As a result inputs were written into the array elements by hand in the program. What this translated to was an enormous



Shapiro's test data for 8x8 coefficients

| | | | | | |
|---|---|---|---|---|---|
| 0 | 63 | 22 | 6 | 44 | 6 |
| 1 | -34 | 23 | -1 | 45 | -4 |
| 2 | -31 | 24 | 5 | 46 | 5 |
| 3 | 23 | 25 | -7 | 47 | 6 |
| 4 | 49 | 26 | 4 | 48 | 4 |
| 5 | 10 | 27 | -2 | 49 | 6 |
| 6 | 14 | 28 | 3 | 50 | 3 |
| 7 | -13 | 29 | 9 | 51 | -2 |
| 8 | 15 | 30 | 3 | 52 | -2 |
| 9 | 14 | 31 | 2 | 53 | 2 |
| 10 | -9 | 32 | -5 | 54 | 0 |
| 11 | -7 | 33 | 9 | 55 | 4 |
| 12 | 3 | 34 | 3 | 56 | 3 |
| 13 | -12 | 35 | 0 | 57 | 6 |
| 14 | -14 | 36 | -1 | 58 | 0 |
| 15 | 8 | 37 | 47 | 59 | 3 |
| 16 | 7 | 38 | -3 | 60 | 3 |
| 17 | 13 | 39 | 2 | 61 | 6 |
| 18 | 3 | 40 | 2 | 62 | -4 |
| 19 | 4 | 41 | -3 | 63 | 4 |
| 20 | -12 | 42 | 5 | | |
| 21 | 7 | 43 | 11 | | |

Shapiro's data for the single processor codec, arranged in order



Shapiro's data for the 3-processor codec Arranged in order.

A : for processor A
B : for processor B
C : for processor C

Figure 5.1

number of inputs to be written by hand within the code. Just for an 8 × 8 coefficients the number of codes obtained was about 300, which is almost 5 times the number of coefficients. All these 300 codes had to be input by hand inside the code to check for the decoder. If a 64 × 64 coefficients were tested, assuming a linear relationship between the number of coefficients (in reality the number of codes obtained would be more than that obtained from this assumption), there would be 4096 inputs for the encoder and 20,480 codes to be input into the decoder, all by hand. These numbers are truly prohibitively large. Even for a 16 × 16 coefficients, the codes would number more than 1300. As a result tests were performed using the 8 × 8 coefficients data.

But we should remember that the codec designed was a generic one. So the argument is if it works for an 8 × 8 coefficients it should also work for any image

size. In fact had there been sufficient time to do the interface design this claim would have been proved indeed.

The standard test data is shown below.

## 5.3 Simulation Result for Single Processor Codec

The results presented below were all checked against hand worked results, and it was found both results tallied as desired.

### 5.3.1 Encoder Simulation

**Input = from figure 5.1, Out put code = Table 5.0**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| ZTR | ZTR | POS | ZTR | POS | ZTR | NO | NEG | YES | NO |
| POS | ZTR | ZTR | ZTR | ZTR | ZTR | NO | POS | NO | NO |
| NEG | ZTR | POS | ZTR | POS | ZTR | YES | POS | YES | YES |
| IZ | ZTR | ZTR | ZTR | POS | POS | YES | POS | NO | YES |
| ZTR | ZTR | ZTR | ZTR | POS | ZTR | YES | POS | NO | NO |
| POS | ZTR | NEG | ZTR | ZTR | POS | NO | ZTR | NO | YES |
| ZTR | ZTR | ZTR | ZTR | POS | ZTR | YES | Z1R | YES | YES |
| ZTR | ZTR | ZTR | ZTR | NEG | ZTR | YES | NEG | NO | NO |
| ZTR | ZTR | ZTR | ZTR | POS | ZTR | NO | POS | YES | YES |
| ZTR | ZTR | ZTR | ZTR | ZTR | POS | YES | POS | YES | YES |
| IZ | ZTR | ZTR | NEG | ZTR | EG | YES | NEG | YES | NO |
| ZTR | ZTR | ZTR | YES | ZTR | POS | NO | POS | YES | NO |
| ZTR | ZTR | ZTR | NO | ZTR | POS | YES | NEG | YES | YES |
| ZTR | POS | ZTR | NO | NEG | YES | NO | NEG | NO | YES |
| ZTR | YES | POS | YES | ZTR | YES | NO | POS | YES | NO |
| ZTR | NO | ZTR | YES | ZTR | NO | NO | ZTR | NO | NO |
| ZTR | NO | ZTR | YES | ZTR | YES | YES | POS | YES | NO |
| ZTR | YES | ZTR | NO | ZTR | YES | NO | ZTR | YES | YES |
| POS | YES | POS | YES | ZTR | YES | NO | POS | NO | YES |
| ZTR | NO | ZTR | YES | ZTR | YES | YES | POS | YES | YES |
| ZTR | POS | ZTR | YES | ZTR | YES | NO | NEG | YES | POS |
| NEG | POS | ZTR | YES | POS | NO | YES | YES | NO | ZTR |
| YES | NEG | ZTR | NO | POS | YES | NO | NO | NO | NEG |
| NO | POS | ZTR | YES | NEG | YES | YES | YES | YES | ZTR |
| YES | POS | ZTR | YES | POS | NO | YES | YES | NO | ZTR |
| NO | NEG | ZTR | NO | POS | NO | NO | YES | NO | |
| POS | ZTR | ZTR | NO | POS | YES | NO | YES | NO | |
| POS | ZTR | POS | NO | POS | NO | POS | NO | NO | |
| ZTR | NEG | ZTR | POS | ZTR | NO | POS | NO | NO | |
| ZTR | NEG | ZTR | IZ | ZTR | NO | ZTR | YES | NO | |

Table 5.0 Result from encoding (singel processor arachitecture)

## 5.3.2 Decoder Simulation

| index | Reconstructed Coefficients | index | Reconstructed Coefficients |
|---|---|---|---|
| 0 | 30 | 33 | 0 |
| 1 | -30 | 34 | 0 |
| 2 | -24 | 35 | 0 |
| 3 | 0 | 36 | 0 |
| 4 | 40 | 37 | 40 |
| 5 | 0 | 38 | 0 |
| 6 | 0 | 39 | 0 |
| 7 | 0 | 40 | 0 |
| 8 | 0 | 41 | 0 |
| 9 | 0 | 42 | 0 |
| 10 | 0 | 43 | 0 |
| 11 | 0 | 44 | 0 |
| 12 | 0 | 45 | 0 |
| 13 | 0 | 46 | 0 |
| 14 | 0 | 47 | 0 |
| 15 | 0 | 48 | 0 |
| 16 | 0 | 49 | 0 |
| 17 | 0 | 50 | 0 |
| 18 | 0 | 51 | 0 |
| 19 | 0 | 52 | 0 |
| 20 | 0 | 53 | 0 |
| 21 | 0 | 54 | 0 |
| 22 | 0 | 55 | 0 |
| 23 | 0 | 56 | 0 |
| 24 | 0 | 57 | 0 |
| 25 | 0 | 58 | 0 |
| 26 | 0 | 59 | 0 |
| 27 | 0 | 60 | 0 |
| 28 | 0 | 61 | 0 |
| 29 | 0 | 62 | 0 |
| 30 | 0 | 63 | 0 |
| 31 | 0 | | |
| 32 | 0 | | |

Table 5.1 Reconstructed from first 21 codes

| Index | Reconstructed Coefficients | Index | Reconstructed Coefficients |
|---|---|---|---|
| 0 | 52 | 33 | 0 |
| 1 | -36 | 34 | 0 |
| 2 | -20 | 35 | 0 |
| 3 | 20 | 36 | 0 |
| 4 | 52 | 37 | 36 |
| 5 | 12 | 38 | 0 |
| 6 | 0 | 39 | 0 |
| 7 | 0 | 40 | 0 |
| 8 | 0 | 41 | 0 |
| 9 | 0 | 42 | 0 |
| 10 | 0 | 43 | 0 |
| 11 | 0 | 44 | 0 |
| 12 | 0 | 45 | 0 |
| 13 | 0 | 46 | 0 |
| 14 | 0 | 47 | 0 |
| 15 | 0 | 48 | 0 |
| 16 | 0 | 49 | 0 |
| 17 | 0 | 50 | 0 |
| 18 | 0 | 51 | 0 |
| 19 | 0 | 52 | 0 |
| 20 | 0 | 53 | 0 |
| 21 | 0 | 54 | 0 |
| 22 | 0 | 55 | 0 |
| 23 | 0 | 56 | 0 |
| 24 | 0 | 57 | 0 |
| 25 | 0 | 58 | 0 |
| 26 | 0 | 59 | 0 |
| 27 | 0 | 60 | 0 |
| 28 | 0 | 61 | 0 |
| 29 | 0 | 62 | 0 |
| 30 | 0 | 63 | 0 |
| 31 | 0 | | |
| 32 | 0 | | |

Table 5.2 Reconstructed from first 44 codes

| Index | Reconstructed Coefficients | Index | Reconstructed Coefficients |
|---|---|---|---|
| 0 | 58 | 33 | 10 |
| 1 | -34 | 34 | 0 |
| 2 | -26 | 35 | 0 |
| 3 | 18 | 36 | 0 |
| 4 | 50 | 37 | 42 |
| 5 | 10 | 38 | 0 |
| 6 | 10 | 39 | 0 |
| 7 | -10 | 40 | 0 |
| 8 | 10 | 41 | 0 |
| 9 | 10 | 42 | 0 |
| 10 | -10 | 43 | 10 |
| 11 | -6 | 44 | 0 |
| 12 | 0 | 45 | 0 |
| 13 | -10 | 46 | 0 |
| 14 | -10 | 47 | 0 |
| 15 | 10 | 48 | 0 |
| 16 | 0 | 49 | 0 |
| 17 | 10 | 50 | 0 |
| 18 | 0 | 51 | 0 |
| 19 | 0 | 52 | 0 |
| 20 | -10 | 53 | 0 |
| 21 | 0 | 54 | 0 |
| 22 | 0 | 55 | 0 |
| 23 | 0 | 56 | 0 |
| 24 | 0 | 57 | 0 |
| 25 | 0 | 58 | 0 |
| 26 | 0 | 59 | 0 |
| 27 | 0 | 60 | 0 |
| 28 | 0 | 61 | 0 |
| 29 | 10 | 62 | 0 |
| 30 | 0 | 63 | 0 |
| 31 | 0 | | |
| 32 | 0 | | |

Table 5.3 Reconstructed from first 100 codes

| Index | Reconstructed Coefficients | Index | Reconstructed Coefficients |
|---|---|---|---|
| 0 | 63 | 33 | 9 |
| 1 | -35 | 34 | 3 |
| 2 | -31 | 35 | 0 |
| 3 | 23 | 36 | 0 |
| 4 | 49 | 37 | 47 |
| 5 | 11 | 38 | -3 |
| 6 | 15 | 39 | 3 |
| 7 | -13 | 40 | 3 |
| 8 | 15 | 41 | -3 |
| 9 | 15 | 42 | 5 |
| 10 | -9 | 43 | 11 |
| 11 | -7 | 44 | 7 |
| 12 | 3 | 45 | -5 |
| 13 | -13 | 46 | 5 |
| 14 | -15 | 47 | 7 |
| 15 | 9 | 48 | 5 |
| 16 | 7 | 49 | 7 |
| 17 | 13 | 50 | 3 |
| 18 | 3 | 51 | -3 |
| 19 | 5 | 52 | -3 |
| 20 | -13 | 53 | 3 |
| 21 | 7 | 54 | 0 |
| 22 | 7 | 55 | 5 |
| 23 | -2 | 56 | 3 |
| 24 | 5 | 57 | 7 |
| 25 | -7 | 58 | 0 |
| 26 | 5 | 59 | 3 |
| 27 | -3 | 60 | 3 |
| 28 | 3 | 61 | 7 |
| 29 | 9 | 62 | -5 |
| 30 | 3 | 63 | 5 |
| 31 | 3 | | |
| 32 | -5 | | |

Table 5.4 Reconstructed from first 230 coefficients

Table 5.1 shows the result of reconstruction of coefficients when the decoder is given the first 21 codes. Table 5,2 shows the result of reconstruction when the decoder is given the first 44 codes. Tables 5.3 and 5.4 are results more codes being given. The results show how the precision of the reconstructed coefficients improve as more and more codes are given. This mimics the bit truncation that would happen for rate constrained codec and how the codec would still be able to reconstruct approximate coefficients. Finally in table 5.5 we see perfect reconstruction as all the codes are given.

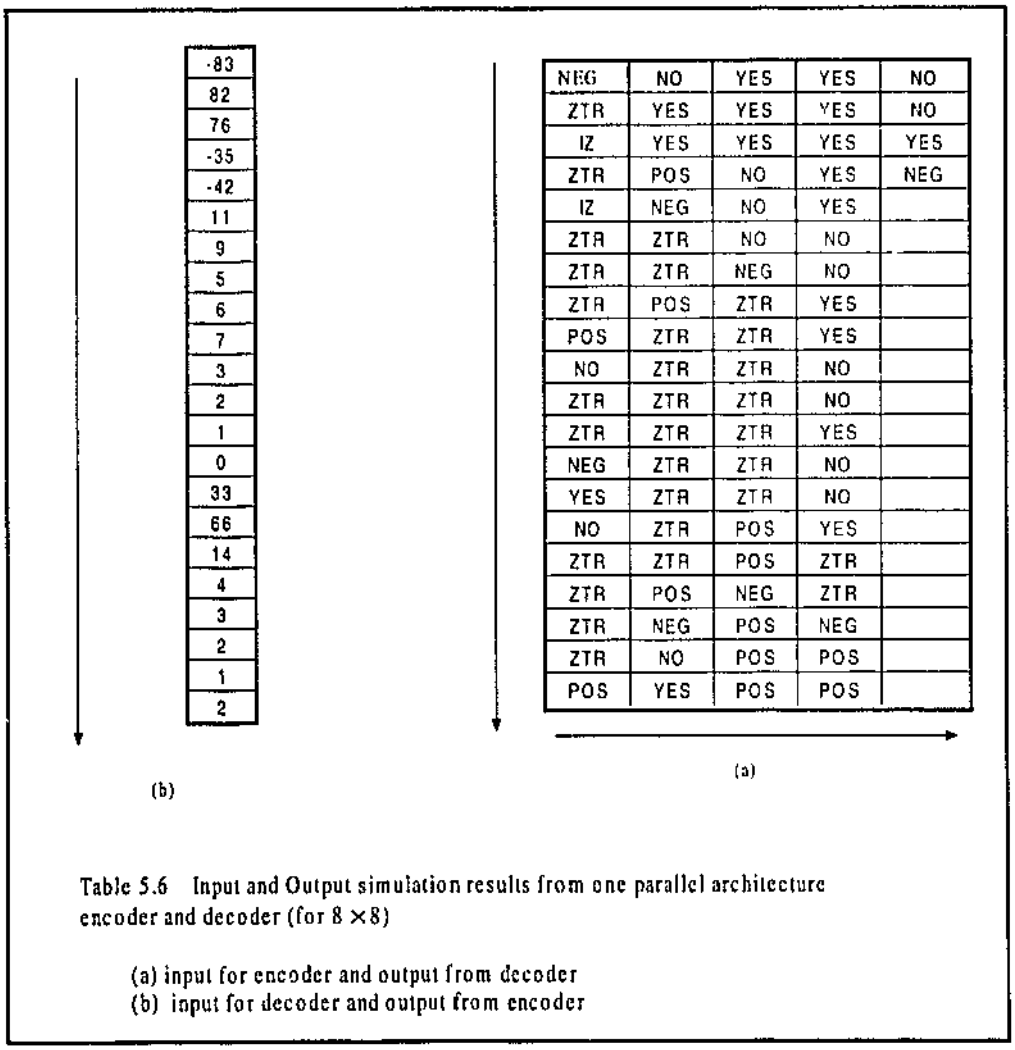
## 5.4 Simulation Result for Parallel Processor Codec

Similarly, as in the single processor codec tests were performed for the parallel procesor codec. The test was performed to see if the encoder and decoder encoded and decoded properly for the entire coefficients and entire codes provided. The result was found to agree with the one computed by hand. Rate constraint was not checked as each processor performs exactly the same as that of the single processor above. In other words each processor here is a smaller version of the above single processor codec.

Only the result for one subband has been shown here, in table 5.6. A hypothetical set of coefficients was chosen. However the results were checked against those worked out manually; they were the same. When the codes were input to the decoder (parallel) it reproduced the original coefficients. Thus proving that it works correctly.

| Index | Reconstructe dCoefficient | Index | Reconstructed CoefficN |
|---|---|---|---|
| 0 | -36 | 33 | -3 |
| 1 | -34 | 34 | -3 |
| 2 | -31 | 35 | 0 |
| 3 | 23 | 36 | -1 |
| 4 | 49 | 37 | 47 |
| 5 | 10 | 38 | -3 |
| 6 | 14 | 39 | 2 |
| 7 | -13 | 40 | 2 |
| 8 | 15 | 41 | -3 |
| 9 | 14 | 42 | 5 |
| 10 | -9 | 43 | 11 |
| 11 | -7 | 44 | 6 |
| 12 | 3 | 45 | -4 |
| 13 | -12 | 46 | 5 |
| 14 | -14 | 47 | 6 |
| 15 | 8 | 48 | 4 |
| 16 | 7 | 49 | 6 |
| 17 | 13 | 50 | 3 |
| 18 | 3 | 51 | -2 |
| 19 | 4 | 52 | -2 |
| 20 | -12 | 53 | 2 |
| 21 | 7 | 54 | 0 |
| 22 | 6 | 55 | 4 |
| 23 | -1 | 56 | 3 |
| 24 | 5 | 57 | 6 |
| 25 | -7 | 58 | 0 |
| 26 | 4 | 59 | 3 |
| 27 | -2 | 60 | 3 |
| 28 | 3 | 61 | -6 |
| 29 | 9 | 62 | -4 |
| 30 | 3 | 63 | 4 |
| 31 | 2 | | |
| 32 | -5 | | |

Table 5.5 Perfect Reconstruction from all the codes



| |
|---|
| -83 |
| 82 |
| 76 |
| -35 |
| -42 |
| 11 |
| 9 |
| 5 |
| 6 |
| 7 |
| 3 |
| 2 |
| 1 |
| 0 |
| 33 |
| 66 |
| 14 |
| 4 |
| 3 |
| 2 |
| 1 |
| 2 |

(b)

| | | | | |
|---|---|---|---|---|
| NEG | NO | YES | YES | NO |
| ZTR | YES | YES | YES | NO |
| IZ | YES | YES | YES | YES |
| ZTR | POS | NO | YES | NEG |
| IZ | NEG | NO | YES | |
| ZTR | ZTR | NO | NO | |
| ZTR | ZTR | NEG | NO | |
| ZTR | POS | ZTR | YES | |
| POS | ZTR | ZTR | YES | |
| NO | ZTR | ZTR | NO | |
| ZTR | ZTR | ZTR | NO | |
| ZTR | ZTR | ZTR | YES | |
| NEG | ZTR | ZTR | NO | |
| YES | ZTR | ZTR | NO | |
| NO | ZTR | POS | YES | |
| ZTR | ZTR | POS | ZTR | |
| ZTR | POS | NEG | ZTR | |
| ZTR | NEG | POS | NEG | |
| ZTR | NO | POS | POS | |
| POS | YES | POS | POS | |

(a)

Table 5.6 Input and Output simulation results from one parallel architecture encoder and decoder (for 8 ×8)

(a) input for encoder and output from decoder
(b) input for decoder and output from encoder

# B. Synthesis

## 5.5 Introduction

A lot of time and effort was put in to synthesize the behavioral design of the codec. However only a few logical blocks could be synthesized and those will be presented here.

The main problem was that even if the behavioral code is in a "synthesizable" construct the Synopsys synthesis tool fails to synthesize for some reason. For instance several times it took so long (once even more than 3 days) to read the file and at the end it either gave a "not enough memory space" or just crashed the machine. An attempt was then made to synthesize the encoder not as one whole program block but in two logical parts. When this was performed the first block worked and was synthesized. The second block still gave the same problem.

There were a few but important behavioral synthesis issues encountered which required the original code for the encoder to be converted into a "synthesizable" construct. Those issues and the changes made in the program deserve mention here.

## 5.6 Issues Encountered in Behavioral Synthesis

- Asynchronous design is difficult to synthesize. In other words the synthesis requires a clock in the design for the synthesis to work properly. A global clock to synchronize the encoder parts was introduced.

- Multi-dimensional arrays are not accepted for synthesis. So the original array of record used in the encoder design had to be replaced by single arrays and the associated source codes had to be changed accordingly.

- While-loops were also found to be unacceptable for synthesis. They were replaced by for-loops, with definite number of loops. If-then-Exit statements were used within the for-loops to evaluate the previous while loop conditions.

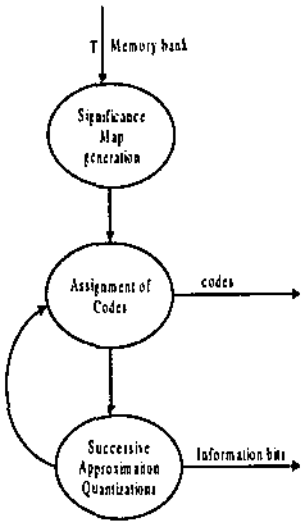



Figure 4.5 Encoding coefficients of memory bank for threshold T

- The wait statement as such is not accepted for synthesis. It is accepted only as clock edge waits. (wait until clock'event and c' ck ='1')

## 5.7 Synthesizing the Significance Map Generator

Figure 4.4 has been presented again to indicate which component the Significance map generator is in the encoding process.

### 5.7.1 The Synthesis Process

- Synopsys synthesis tool is invoked by using the command

  DESIGN_ANALYZER at the unix prompt.

- The .vhd file and all the packages used are then read using the file-read command from the menu

- The gate level schematic can be viewed changing the level up-down from the menu.

- Report on various aspects of the synthesized designed can also be generated using the generate-report command.

### 5.7.2 Synthesized Significance Map Generator Schematics.

Figure 5.2 I level schemaatic of Significance map generator

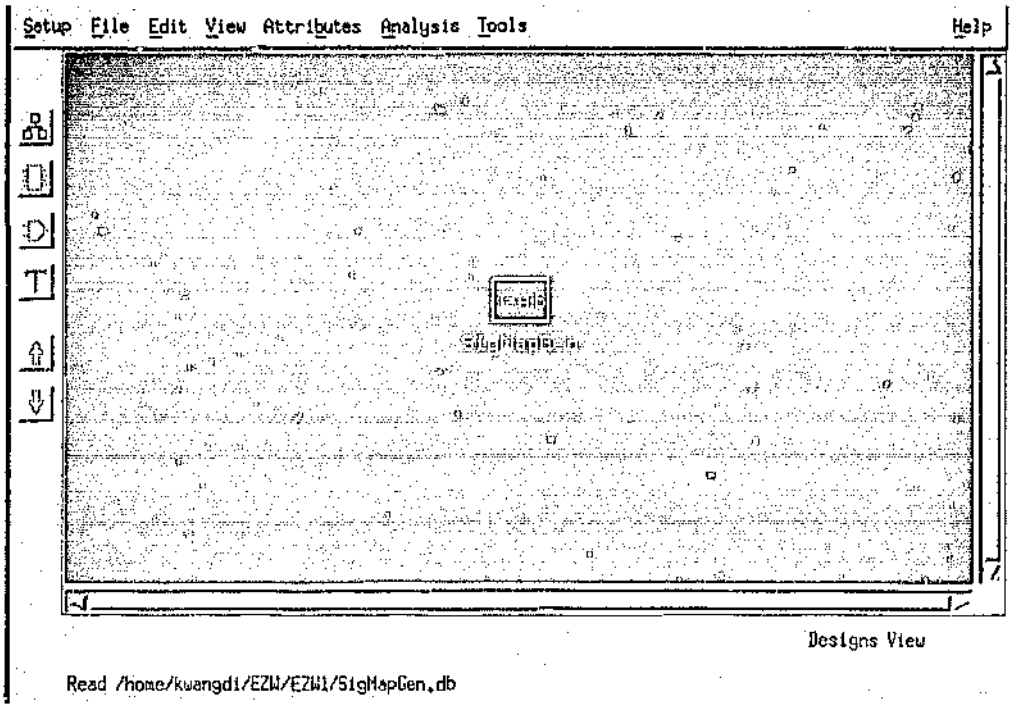



Figure 5.3 II level Schemaatic of Significance map generator

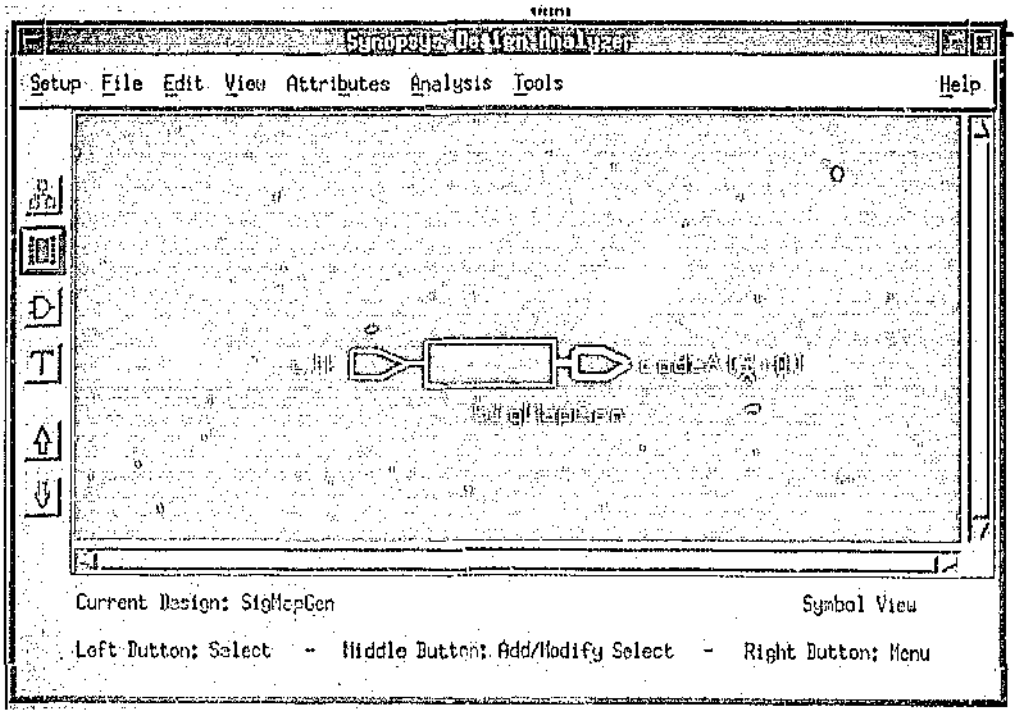


Figure 5.4 Gate Level Schemaatic of the whole Significance map generator

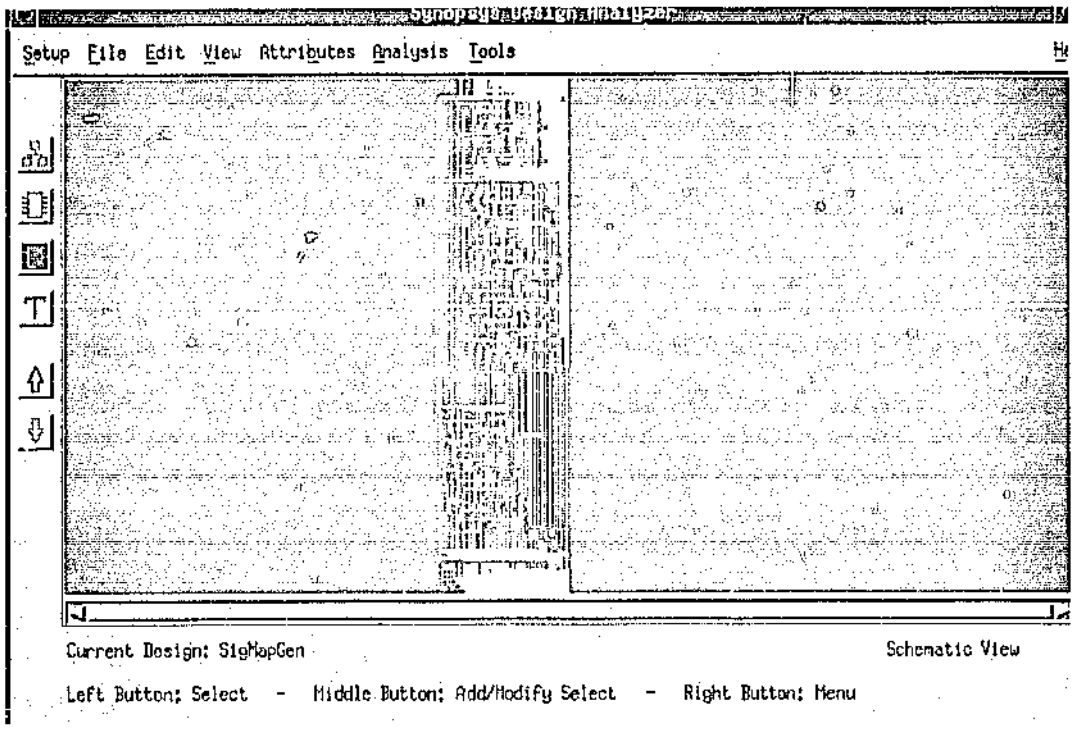
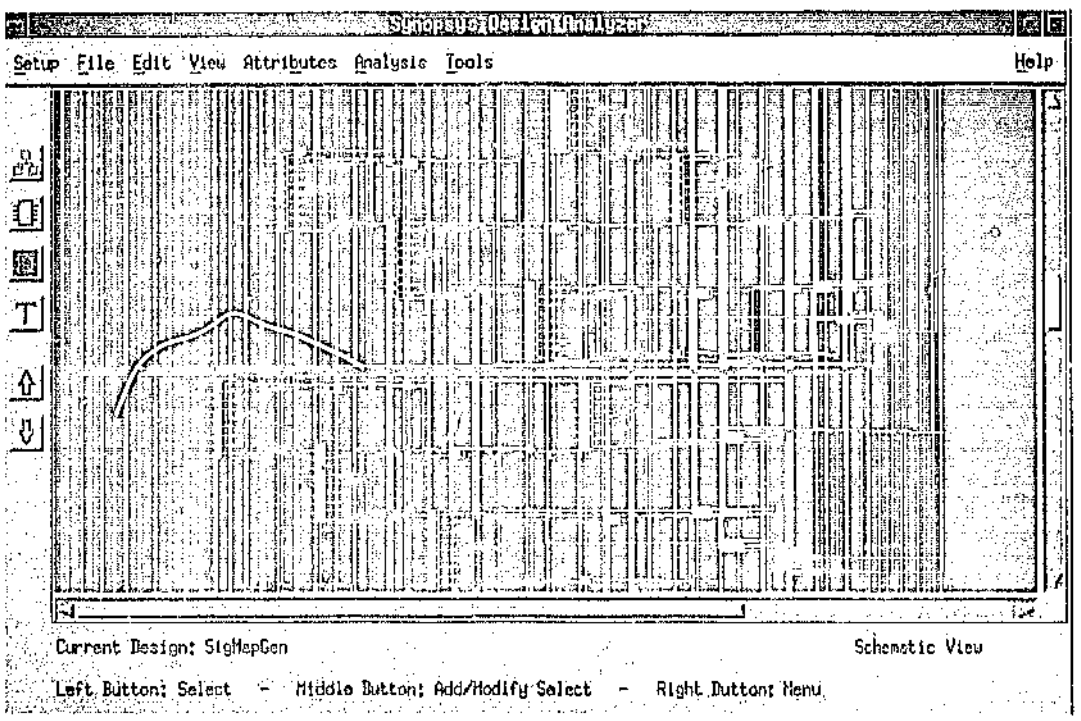



Figure 5.5 Magnified part of the significance map generator



# 6. CONCLUSION

To repeat here, the main aim of the project was to design, simulate and if possible, synthesize, an EZT codec implementing the EZW algorithm. The objectives that this main aim of the project translated to were:

- Understanding of image compression in general and that of wavelet image compression in particular.
- Study of wavelet transform and how it relates to image compression.
- A thorough understanding of the EZW algorithm and the various terms used in it.
- A good mastery of the VHDL language
- Design and simulation of a codec using VHDL behavioral level code.
- Synthesis of the codec.

## 6.1 Project Achievements and Contribution

The original contributions of this project are as follows.

- The most important contribution by this project has been the design of the codec itself. In the codec design, original strategies have been devised to generate the significance map, assign codes and perform successive approximation quantization, the three main steps in the EZW algorithm. The method used to decode the codes has also been an original contribution.

- The EZW algorithm as such from Shapiro's paper deals at an advanced level understanding of image coding and wavelet transform. Consultation with people on authority on the topic in the engineering department at ECU, a simpler and concise algorithm has been presented.

- Another important contribution by this project has been the presentation of a clear and concise explanation on this rather mathematically rigorous topic of wavelet transform. The mathematics of it was first presented, and this was followed by

discussion on how this transform is practically performed using sets of high pass and low pass filters, both in continuous time and in the discrete domains. The merits of wavelet transforms were then discussed in relation to image compression.

- The design, simulation and synthesis were done using the VHDL simulation tools and synthesis tools from Synopsys Inc. These tools were well explored. A number of issues pertaining to behavioral synthesis as encountered have been properly documented, together with the commands that were used to invoke the necessary simulation and synthesis tools. Anyone new to Synopsys will find this contribution very handy.

- The project has also made a good study of the numerous other image compression techniques that are available today and the basis on which each of these techniques have evolved.

## 6.2 Comments and Recommendations for Future Research

One of the great strengths of the codec designed is its ability to meet any bit rate or distortion rate exactly. But then it is not enough for a codec to just satisfy any bit rate if it can not reproduce a good enough picture. An obvious question that comes to mind is, how many bits can be truncated before the reconstructed image begins to show perceptible distortion? To answer this question a series of tests could be performed on this codec by applying suitable wavelet transform on a standard test image, obtain the coefficients, feed the coefficients to the codec and get reconstructed value of coefficients for different bit rates. The output from the decoder for different bit rates can then be inverse transformed and the different reconstructed versions for different bit rates of the same image could be compared with the original. This would give an idea of the performance of the EZT codec, besides the numerical calculation of PSNR (peak signal to noise ratio). A software program like Matlab would be suitable for this purpose.

The present implementation of the codec uses static data. That is, inputs to the encoder and the decoder are provided within the code at the beginning of processing. A suitable interface could be added to the current design so that inputs can be obtained from an outside source. This would take the realization of the encoder and the decoder as single stand-alone chips one last step closer. The tests suggested above can also be implemented very easily with this added interface.

The synthesized codec can then be implemented using FPGA.

Even thus show of promise by the multiresolution image analysis appears to be well matched to the low-level characteristics of human vision. As this approach is developed further to incorporate additional aspects of human vision, such as spectral response characteristics, masking, pattern primitives and the like, the future of image compression looks anything but much more promising and ex [illegible].

This project has contributed successfully to this state of the art technique of wavelet image compression, not only from a VLSI front but also from a research point of view.

# Appendix

```
--************************************************************
--        SINGLE PROCESSOR ENCODER FOR SIMULATION
--************************************************************
------------------------------------------------------------
------------------------------------------------------------
-- Embedded Zerotree Wavelet Algorithm.
-- Image size of 8 x 8 wavelet coefficients
-- 8 bit implementation (coefficients range from -128 to 127)
------------------------------------------------------------
------------------------------------------------------------

library IEEE;
library WORK;
use WORK.TypePKGsyn.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;


Entity FullEncoder is

   Port(clk : in std_logic; codeA: out CodeType) ;

End FullEncoder;


Architecture Behavioural of FullEncoder is

Type CoeffRec is
     record
       ZTF:        Bit;
       Encoded:    Bit;
       Coeff:   CoeffType;
     end record;

-- DSig is to indicate if the coefficient has atleast one
-- significant coefficient for that threshold, or whether it
-- is a leaf coefficient.

-- ZTF to indicate whether or not a zerotree has been found
-- earlier.

-- Encoded to indicate if a coefficient has been encoded for
--any prevoius threshold.
-- Coeff to contain the coefficient.


Type RAOfRec is Array (0 to lenth) of CoeffRec;

-- to contain the input coefficients and information on each
-- coefficient
```

```
Type OutPutRA is Array (0 to 400) of CodeType;

-- to contain the code for the coefficients from the encoding
-- process

Type AllQntRA is Array (0 to 63, 0 to 6) of CodeType;

-- to contain the quantization values of all the significant
-- codes

Type FirstRA is Array (0 to 5) of Bit;

-- To indicate if the coefficient is the first significant
-- coefficient for a given threshold

begin  -- start of the Architecture

   ProcessTree: Process
   variable T          : integer range 1 to 64;
   variable Contnr       : RAOfRec;          -- threshold
   variable Parent       : IndexType;        -- parent pointer
   variable Child      : IndexType;          -- child pointer
   variable OutCode      : OutPutRA;         -- output codes
   variable m          : Integer range 0 to 400;
                    -- to index the output codes
   variable ApproxValue : AllQntRA;
variable SigNo         : Integer range 0 to 64;
                  -- count no of significant codes encoded.
   variable copy       : Integer range 0 to 64;
   variable First        : FirstRA;

   begin  -- start the process.


-- *** GetCoeff ***
-- data from Shapiro

Contnr(0).Coeff:=63;Contnr(1).Coeff:=-34;Contnr(2).Coeff:=-31;
Contnr(3).Coeff:=23;Contnr(4).Coeff:=49;Contnr(5).Coeff:=10;
Contnr(6).Coeff:=14;Contnr(7).Coeff:=-13;Contnr(8).Coeff:=15;
Contnr(9).Coeff:=14;Contnr(10).Coeff:=-9;Contnr(11).Coeff:=-7;
Contnr(12).Coeff:=3;Contnr(13).Coeff:=-12;Contnr(14).Coeff:=-14;
Contnr(15).Coeff:=8;Contnr(16).Coeff:=7;Contnr(17).Coeff:=13;
Contnr(18).Coeff:=3;Contnr(19).Coeff:=4;Contnr(20).Coeff:=-12;
Contnr(21).Coeff:=7;Contnr(22).Coeff:=6;Contnr(23).Coeff:=-1;
Contnr(24).Coeff:=5;Contnr(25).Coeff:=-7;Contnr(26).Coeff:=4;
Contnr(27).Coeff:=-2;Contnr(28).Coeff:=3;Contnr(29).Coeff:=9;
Contnr(30).Coeff:=3;Contnr(31).Coeff:=2;Contnr(32).Coeff:=-5;
Contnr(33).Coeff:=9;Contnr(34).Coeff:=3;Contnr(35).Coeff:=0;
Contnr(36).Coeff:=-1;Contnr(37).Coeff:=47;Contnr(38).Coeff:=-3;
Contnr(39).Coeff:=2;Contnr(40).Coeff:=2;Contnr(41).Coeff:=-3;
Contnr(42).Coeff:=5;Contnr(43).Coeff:=11;Contnr(44).Coeff:=6;
Contnr(45).Coeff:=-4;Contnr(46).Coeff:=5;Contnr(47).Coeff:=6;
Contnr(48).Coeff:= 4;Contnr(49).Coeff:= 6;Contnr(50).Coeff:= 3;
Contnr(51).Coeff:= -2;Contnr(52).Coeff:= -2;Contnr(53).Coeff:= 2;
Contnr(54).Coeff:=0;Contnr(55).Coeff:=4;Contnr(56).Coeff:=3;
Contnr(57).Coeff:=6;Contnr(58).Coeff:=0;Contnr(59).Coeff:=3;
```

```
Contnr(60).Coeff:=3;Contnr(61).Coeff:=6;Contnr(62).Coeff:=-4;
Contnr(63).Coeff:= 4;
 For i in 0 to lenth loop -- initialize the variables
   Contnr(i).DSig := u;
   Contnr(i).ZTF:= '0';
   Contnr(i).Encoded := '0';
     End loop;

First := "000000";
          -- no first significant coefficients encoded.
SigNo := 0; -- no of the significants found is zero.
m := 0;      -- point to the first element of the output array.

For l in 6 downto 0 loop      -- main loop that does all the 7
--significance map generation and
-- of coefficients and the
-- subsequent encoding.

   T       := Thresh(l); -- get appropraite threshold.
   Parent  := 15;       -- last parent coefficient
   Child     := lenth;      -- last coefficient (63rd).

-------------------------------------------------
-- SIGNIFICANCE MAP GENERATION
-------------------------------------------------

   While Parent >=1 loop   -- do until the main parent (0th
                       -- coefficient).
     for j in 0 to 3 loop
-- Check the four children
-- for significance and exit if one of
-- them is found to be significant

   If Contnr(Child-j).DSig = u then  -- child is a leaf.     If
abs(Contnr(Child - j).coeff)>= T then
                                --significant
       Contnr(Parent).DSig := '1';
--record descendent significant
-- information.
              Exit;     -- don't need to check other children
            Else
              Contnr(Parent).DSig := '0';
            End If;
          ElsIf Contnr(Child - j).Dsig = '1' then
            Contnr(Parent).DSig := '1';
            Exit;
          Else      -- Contnr(Child - j).DSig = '0'
          If abs(Contnr(Child - j).coeff)>= T then
            Contnr(Parent).DSig := '1';
            Exit;
          Else
            Contnr(Parent).DSig := '0';
          End If;
        End If;
     End Loop;           --for one parent
     Child := Child - 4;-- for the next parent
     Parent := Parent - 1;
```

```
   End Loop;              -- all parents

--we have reached the first parent and its three children
   For j in 0 to 2 Loop  -- main parent has only 3 children
     If Contnr(child -j).DSig = '1' then
       Contnr(Parent).DSig := '1';
       Exit;
     Else
       If abs(Contnr(child-j).Coeff) >= T then
         Contnr(parent).DSig := '1';
         Exit;
       Else
         Contnr(parent).DSig := '0';
       End If;
     End If;
   End Loop;

-------------------------------------------------------------
-- SIGNIFICANCE MAP GENERATION IS COMPLETED FOR ONE THRESHOLD
--ASSIGN CODES
-------------------------------------------------------------

   Parent := 0;  -- start from the main parent
   Child := 1;   -- child pointer point to the first child

   Loop   -- till all the descendents have been encoded, happens
        -- when parent = 63 is over
     If Contnr(parent).ZTF = '1' then  -- zeortree element
       If parent > 15 then        -- leaf coefficient
         Null;
       Else
         For i in 0 to 3 loop
           Contnr(Child + i).ZTF := '1';
-- pass the information that a
-- Zerotree root has been found
-- ahead to the children.
         End loop;
       End If;
     ElsIf
       Contnr(parent).Encoded = '1' then  -- already encoded.
         Null;
     Else                              -- not encoded
       If Abs(Contnr(parent).Coeff) >= T then -- significant
         If Contnr(parent).Coeff < 0 then
           OutCode(m) := NEG;              -- code negative
           m := m + 1;
         Else
           OutCode(m):= POS;               -- code positive
           m := m + 1;
         End If;
-------------------------------------------------------------
-- PERFORM SUCCESSIVE APPROXIMATION QUANTIZATION FOR SIG. CODE
-------------------------------------------------------------
         copy := Abs(contnr(parent).Coeff) - Thresh(l);
         Case l is
           When 0 =>  -- Threshold is 1
             Null;
```

```
            When 6 =>  -- Threshold is 64
              For k in 1 downto 0 loop
                    -- fill the approximation array
                If (copy - Thresh(k)) < 0 then
                  ApproxValue(SigNo,k) := NO;
                Else
                  ApproxValue(SigNo,k) := YES;
                  copy := copy - Thresh(k);
                End If;
              End Loop;
              OutCode(m):= ApproxValue(SigNo,1);
              m := m+1;
            When Others =>  -- Other thresholds
              For k in (1-1) downto 0 loop
                If (copy - Thresh(k)) < 0 then
                  ApproxValue(SigNo,k) := NO;
                Else
                  ApproxValue(SigNo,k) := YES;
                  copy := copy - Thresh(k);
                End If;
              End Loop;
          End Case;

          -- Add precision to the already encoded coefficients

          Case 1 is
            When 6 =>  -- threshold is 64
              Null;
            When Others =>  -- other thresholds
              If First(1) = '0' then
                If SigNo = 0 then
                  Null;
                Else
                  for r in 0 to (SigNo) loop
                    OutCode(m) := ApproxValue(r,1);
                    m := m + 1;
                  End loop;
                End If;
                First(1):='1';
              End If;
            End Case;

            -- increment the significant no
            SigNo := SigNo + 1;
            Contnr(parent).Coeff := 0;
          --set encoded coefficient to 0 to
-- prevent non occurrence of ZTR
--because of it.
            Contnr(parent).Encoded := '1';

        -- code insignificant
        Else
          If Contnr(parent).DSig = '1' then
            OutCode(m) := IZ;  -- code isolated zero
            m := m + 1;
          Else
            OutCode(m) := ZTR; -- code zerotree root
```

```
            m := m + 1;
            If parent = 0 then
                Exit;
            Else
              If Contnr(parent).DSig = '0' then
                -- pass ZTF information to children
                For j in 0 to 3 loop;
                  Contnr(child+j).ZTF := '1';
                End Loop;
              Else  -- it is a leaf insignificant
                Null;
              End If;
            End If;
          End If;
        End If;

        If Parent >= 63 then  -- stop when all the coeffcients have
                        -- been encoded.
          Exit;
        Else
          If parent = 0 then
            child := child + 3;  -- 3 children for the main
                                 -- parent, all other parents 4
                                 -- each.
          Else
            Child := Child + 4;
          End If;
          parent := parent + 1;
        End If;
      End loop;

      -- one threshold loop
      -- set the significance (DSig) fields to 0,but not for the
      -- leaves

      For k in 0 to lenth loop
        If Contnr(k).DSig = u then
          Null;
        Else
          Contnr(k).DSig := '0';
        End If;
        Contnr(k).ZTF := '0';
      End Loop;

    End Loop; -- all the 7 threshods encoding loop

    -- display the code

    For n in 0 to m-1 loop
      CodeA <= OutCode(n);
      wait for 5 ns;
    End Loop;
    Wait;
    End Process;
    End of process EncodeTree;
```

```
End Behavioural;


-------------------------------------------
-- Conifiguration
-------------------------------------------

Configuration CFGFull_Coder of FullEncoder is
   For
        Behaviouralsyn
   End For;
End CFGFull_Coder;

-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
--            END
-------------------------------------------------------------------------------
```

```
--*****************************************************
--      SINGLE PROCESSOR DECODER FOR SIMULATION
--*****************************************************
-------------------------------------------------------
-- Decodes the codes from the single previous encoder.
--The result will be a single tree of 64 coefficients
-- that existed before the encoding was performed
-------------------------------------------------------------
-------------------------------------------------------------
```

```
library IEEE;
library WORK;
use WORK.TypePKGsyn.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

Entity FullDecoder is
Port(outcoeff: out CoeffType; addrs: out Integer);
End FullDecoder;

Architecture Behavioural of FullDecoder is

Type CoeffRec is
  Record
    Coeff :     CoeffType;
    ZTF:        Bit;
    Decoded:  Bit;
  End Record;
-- Coeff to contain te decoded coefficient
-- ZTF to indicate zerotree root found

Type CoeffRA is Array (0 to lenth) of CoeffRec;

Type CodeRA is Array (0 to 300) of CodeType;

Type FrstIndictr is Array (0 to 5) of Bit;

Type AddrSign is
    record
      Addr : Integer range 0 to 63;
      Sign : Bit;
End Record;

  -- addr to store the index of the coefficient
  -- sign to store the sign of the coefficient

Type RAddrSign is Array (0 to lenth) of AddrSign;
  -- to contain the information on the decoded codes

begin

DecodeProcess: Process

   variable P       : Integer range 0 to 300;
                      -- index the code container
   variable CoeffContnr : CoeffRA;  -- contain the coeff and
                          -- the associated fields
   variable CodeContnr  : CodeRA;    -- code container
   variable parent    : Integer range 0 to lenth;
                          -- parent pointer
   variable child     : Integer range 0 to lenth;
                        -- child pointer
   variable DcodedInfo  : RAddrSign;
                        -- to contain sign and index of decoded
                        -- coefficients.
```

```
   variable SigNo     : Integer range 0 to 64;
-- no of significant coefficients
-- decoded.
   variable First     : FrstIndictr;
-- first sig for a threshold decoded
                        -- information
   variable Exited    : Boolean;
   variable copyl     : integer range 0 to 6;
                        -- to indicate   threshold

begin  -- process starts
-- get the codes

CodeContnr(0)   :=   ZTR;CodeContnr(1)   :=   POS;CodeContnr(2)   :=   NEG;
CodeContnr(3) := IZ;  CodeContnr(4) := ZTR; CodeContnr(5)      :=      POS;
CodeContnr(6) := ZTR; CodeContnr(7) := ZTR; CodeContnr(8)      :=      ZTR;
CodeContnr(9) := ZTR; CodeContnr(10) := IZ; CodeContnr(11)     :=      ZTR;
CodeContnr(12) := ZTR;  CodeContnr(13) := ZTR;  CodeContnr(14) := TR;
CodeContnr(15) := ZTR;  CodeContnr(16) := ZTR;  CodeContnr(17)   :=   ZTR;
CodeContnr(18) := POS;  CodeContnr(19) := ZTR;  CodeContnr(20)   :=   ZTR;
CodeContnr(21) := NEG;  CodeContnr(22):= YES; CodeContnr(23):=         NO;
CodeContnr(24):= YES; CodeContnr(25):= NO;  CodeContnr(26):=          POS;
CodeContnr(27):= POS; CodeContnr(28):= ZTR; CodeContnr(29):=          ZTR;
CodeContnr(30):= ZTR; CodeContnr(31) := ZTR;  CodeContnr(32) := ZTR;
CodeContnr(33) := ZTR;  CodeContnr(34) := ZTR;  CodeContnr(35)   :=   ZTR;
CodeContnr(36) := ZTR;  CodeContnr(37) := ZTR;  CodeContnr(38)   :=   ZTR;
CodeContnr(39) := ZTR;  CodeContnr(40) := ZTR;  CodeContnr(41)   :=   ZTR;
CodeContnr(42) := ZTR;  CodeContnr(43) := POS;  CodeContnr(44)   :=   YES;
CodeContnr(45) := NO; CodeContnr(46) := NO; CodeContnr(47)     :=      YES;
CodeContnr(48) := YES;  CodeContnr(49) := NO; CodeContnr(50)    :=     POS;
CodeContnr(51) := POS;  CodeContnr(52) := NEG;  CodeContnr(53)   :=   POS;
CodeContnr(54) := POS;  CodeContnr(55) := NEG;  CodeContnr(56)   :=   ZTR;
CodeContnr(57) := ZTR;  CodeContnr(58) := NEG;  CodeContnr(59)   :=   NEG;
CodeContnr(60) := POS;  CodeContnr(61) := ZTR;  CodeContnr(62)   :=   POS;
CodeContnr(63) := ZTR;  CodeContnr(64) := ZTR;  CodeContnr(65)   :=   NEG;
CodeContnr(66) := ZTR;  CodeContnr(67) := ZTR;  CodeContnr(68)   :=   ZTR;
CodeContnr(69) := ZTR;  CodeContnr(70) := ZTR;  CodeContnr(71)   :=   ZTR;
CodeContnr(72) := ZTR;  CodeContnr(73) := ZTR;  CodeContnr(74)   :=   POS;
CodeContnr(75) := ZTR;  CodeContnr(76) := ZTR;  CodeContnr(77)   :=   ZTR;
CodeContnr(78) := POS;  CodeContnr(79) := ZTR;  CodeContnr(80)   :=   ZTR;
CodeContnr(81) := ZTR;  CodeContnr(82) := ZTR;  CodeContnr(83)   :=   ZTR;
CodeContnr(84) := ZTR;  CodeContnr(85) := ZTR;  CodeContnr(86)   :=   ZTR;
CodeContnr(87) := POS;  CodeContnr(88) := ZTR;  CodeContnr(89)   :=   ZTR;
CodeContnr(90) := ZTR;  CodeContnr(91) := ZTR;  CodeContnr(92)   :=   ZTR;
CodeContnr(93) := ZTR;  CodeContnr(94) := ZTR;  CodeContnr(95)   :=   ZTR;
CodeContnr(96) := ZTR;  CodeContnr(97) := ZTR;  CodeContnr(98)   :=   ZTR;
CodeContnr(99) := ZTR;  CodeContnr(100) := NEG; CodeContnr(101)  :=   YES;
CodeContnr(102) := NO;  CodeContnr(103) := NO;  CodeContnr(104)  :=   YES;
CodeContnr(105) := YES; CodeContnr(106) := YES; CodeContnr(107)   :=   NO;
CodeContnr(108) := YES; CodeContnr(109) := YES; CodeContnr(110)  :=   YES;
CodeContnr(111) := YES; CodeContnr(112) := NO;  CodeContnr(113)  :=   YES;
CodeContnr(114) := YES; CodeContnr(115) := NO;  CodeContnr(116)  :=   YES;
CodeContnr(117) := YES; CodeContnr(118) := NO;  CodeContnr(119)   :=   NO;
CodeContnr(120) := NO;  CodeContnr(121) := POS; CodeContnr(122)   :=   IZ;
CodeContnr(123) := POS; CodeContnr(124) := ZTR; CodeContnr(125)  :=   POS;
CodeContnr(126) := POS; CodeContnr(127) := POS; CodeContnr(128)  :=   ZTR;
CodeContnr(129) := POS; CodeContnr(130) := NEG; CodeContnr(131)  :=   POS;
```

```
CodeContnr(132) := ZTR; CodeContnr(133) := ZTR; CodeContnr(134) := ZTR;
CodeContnr(135) := ZTR; CodeContnr(136) := NEG; CodeContnr(137) := ZTR;
CodeContnr(138) := ZTR; CodeContnr(139) := ZTR; CodeContnr(140) := ZTR;
CodeContnr(141) := ZTR; CodeContnr(142) := ZTR; CodeContnr(143) := ZTR;
CodeContnr(144) := POS; CodeContnr(145) := POS; CodeContnr(146) := NEG;
CodeContnr(147) := POS; CodeContnr(148) := POS; CodeContnr(149) := POS;
CodeContnr(150) := POS; CodeContnr(151) := ZTR; CodeContnr(152) := ZTR;
CodeContnr(153) := ZTR; CodeContnr(154) := ZTR; CodeContnr(155) := ZTR;
CodeContnr(156) := POS; CodeContnr(157) := ZTR; CodeContnr(158) := POS;
CodeContnr(159) := ZTR; CodeContnr(160) := ZTR; CodeContnr(161) := ZTR;
CodeContnr(162) := POS;CodeContnr(163) := NEG;  CodeContnr(164) := POS;
CodeContnr(165) := POS; CodeContnr(166) := YES; CodeContnr(167) := YES;
CodeContnr(168) := NO;  CodeContnr(169) := YES; CodeContnr(170) := YES;
CodeContnr(171) := YES; CodeContnr(172) := YES; CodeContnr(173) := YES;
CodeContnr(174) := NO;  CodeContnr(175) := YES; CodeContnr(176) := YES;
CodeContnr(177) := NO;  CodeContnr(178) := NO;  CodeContnr(179) := YES;
CodeContnr(180) := NO;  CodeContnr(181) := NO;  CodeContnr(182) := NO;
CodeContnr(183) := NO;  CodeContnr(184) := NO;  CodeContnr(185) := YES;
CodeContnr(186) := YES; CodeContnr(187) := YES; CodeContnr(188) := NO;
CodeContnr(189) := YES; CodeContnr(190) := YES; CodeContnr(191) := NO;
CodeContnr(192) := YES; CodeContnr(193) := NO;  CodeContnr(194) := NO;
CodeContnr(195) := NO;  CodeContnr(196) := YES; CodeContnr(197) := NO;
CodeContnr(198) := NO;  CodeContnr(199) := YES; CodeContnr(200) := NO;
CodeContnr(201) := YES; CodeContnr(202) := NO;  CodeContnr(203) := YES;
CodeContnr(204) := YES; CodeContnr(205) := NO;  CodeContnr(206) := NO;
CodeContnr(207) := POS;CodeContnr(208) := POS;  CodeContnr(209):=ZTR
CodeContnr(210) := NEG; CodeContnr(211) := POS; CodeContnr(212) := POS;
CodeContnr(213) := POS; CodeContnr(214) := POS;CodeContnr(215) := ZTR;
CodeContnr(216) := ZTR; CodeContnr(217) := NEG; CodeContnr(218) := POS;
CodeContnr(219) := POS; CodeContnr(220) := NEG; CodeContnr(221) := POS;
CodeContnr(222) := NEG; CodeContnr(223) := NEG; CodeContnr(224) := POS;
CodeContnr(225) := ZTR; CodeContnr(226) := POS; CodeContnr(227) := ZTR;
CodeContnr(228) := POS; CodeContnr(229) := POS; CodeContnr(230) := NEG;
CodeContnr(231) := YES; CodeContnr(232) := NO;  CodeContnr(233) := YES;
CodeContnr(234) := YES; CodeContnr(235) := YES; CodeContnr(236) := YES;
CodeContnr(237) := NO;  CodeContnr(238) := NO;  CodeContnr(239):=  YES;
CodeContnr(240):= YES;  CodeContnr(241):= NO; CodeContnr(242):=    YES;
CodeContnr(243):= NO; CodeContnr(244):= NO; CodeContnr(245):=      NO;
CodeContnr(246) := YES; CodeContnr(247) := NO;  CodeContnr(248) := YES;
CodeContnr(249) := YES; CodeContnr(250) := YES; CodeContnr(251) := YES;
CodeContnr(252) := YES; CodeContnr(253) := NO;  CodeContnr(254) := YES;
CodeContnr(255) := NO;  CodeContnr(256) := YES; CodeContnr(257) := YES;
CodeContnr(258) := NO;  CodeContnr(259) := YES; CodeContnr(260) := YES;
CodeContnr(261) := NO;  CodeContnr(262) := NO;  CodeContnr(263) := YES;
CodeContnr(264) := NO;  CodeContnr(265) := NO;  CodeContnr(266) := NO;
CodeContnr(267) := NO;  CodeContnr(268) := NO;  CodeContnr(269) := NO;
CodeContnr(270) := NO;  CodeContnr(271) := NO;  CodeContnr(272) := YES;
CodeContnr(273) := YES; CodeContnr(274) := NO;  CodeContnr(275) := YES;
CodeContnr(276) := YES; CodeContnr(277) := NO;  CodeContnr(278) := YES;
CodeContnr(279) := YES; CodeContnr(280) := NO;  CodeContnr(281) := NO;
CodeContnr(282) := YES; CodeContnr(283) := YES; CodeContnr(284) := NO;
CodeContnr(285) := NO;CodeContnr(286) := NO;CodeContnr(287) := YES;
CodeContnr(288) := YES; CodeContnr(289) := YES; CodeContnr(290) := POS;
CodeContnr(291) := ZTR;CodeContnr(292) := NEG;  CodeContnr(293) := ZTR;
CodeContnr(294) := ZTR;

-- initialize
```

```
First := "000000";

For m in 0 to lenth loop
    CoeffContnr(m).Coeff := 0;
    CoeffContnr(m).ZTF := '0';
    CoeffContnr(m).Decoded := '0';
    If m <= 5 then
       First(m) := '0';
    End If;
End Loop;

exited := False;    -- bit truncation
P := 0;             -- index the codecontr
SigNo := 0;

For l in 6 downto 0 Loop  -- start decoding from the main parent
    parent:= 0;
    child := 1;
    copyl := 1;

    While parent <= lenth loop  -- decoding for one threshold
      If Coeffcontnr(parent).Decoded = '1' then
                                -- already decoded?
       Null;                      -- no decoding again
       Else
          If Coeffcontnr(parent).ZTF = '1' then
            If parent <= 15 then       -- element of zerotree
              For j in 0 to 3 loop
                              -- pass ZTF info. to children
                Coeffcontnr(child+j).ZTF := '1';
              End Loop;
            End If;
       Else
            Case Codecontnr(p) IS
              When S =>         -- bit truncated
                Exited := True;
                Exit;
              When ZTR =>
                If parent = 0 then
                  p := p + 1;
                  Exit;
-- don't need to code any more for
-- the current threhold
                Else
                  If parent <= 15 then  -- it is not a leaf
                              -- so mark its children
                       For k in 0 to 3 loop
                         CoeffContnr(child + k).ZTF := '1';
                    End Loop;
                  End If;
                  P:= P + 1;
                    -- prepare to read the next code
       End If;
              When POS | NEG =>
                CoeffContnr(parent).Decoded := '1';
                  DcodedInfo(SigNo).Addr := parent;
```

```
                    If Codecontnr(p) = POS then
                      DcodedInfo(parent).sign := '1';
                      Else
                      DcodedInfo(parent).sign := '0';
                      End If;
                    CoeffContnr(parent).Coeff := Thresh(l);
                    P:= P + 1;  -- ready to read the next code
                    -- add precision to the coefficients
                    Case l is
                      When 6 =>  threshold is 64
                        If CodeContnr(p) = S then
                                    -- bit truncated
                          Exited:= true;
                                 -- indicate bit truncated
                          Exit;
                        ElsIf CodeContnr(p) = YES then
                      CoeffContnr(parent).Coeff :=
                      CoeffContnr(parent).Coeff +
                                  Thresh(l);
                        -- add 64 to the coefficient
                        End If;
                        P:= P +1;  -- next code ready to be read
                      When Others =>
                        If Codecontnr(p) = s then  -- bit truncation
                          Exited := true;
                          Exit;
                        ElsIf First(l) = '0' then
-- this is the first significant of
-- the threshold

-- so add precision to the previously
-- found coefficients .
                          If SigNo = 0 then
                            Null;
                          Else
                            For i in 0 to SigNo loop
                                  If Codecontnr(p + i) = YES then

     CoeffContnr(DcodedInfo(i)
                          .Addr).Coeff :=
       CoeffContnr(DcodedInfo(i)
                          .Addr).Coeff + (Thresh(l));
                            Else
                              Null;
            End If;
                          End Loop;
                          p := p + SigNo+1;
                        End If;
                        First(l) := '1';
                        End If;
                    End Case;
                  SigNo := SigNo + 1;
                When Others =>
                    p := P + 1;  -- prepare to read the next code
              End Case;
            End If;
          End If;
```

```
        -- house keeping

        If parent >= 63 then
          Exit;  -- all have been decoded for a threshold
        Else
          If parent = 0 then
            child := child + 3;
          Else
            child := child + 4;
          End If;
          parent := parent + 1;
        End If;
      End Loop; -- single threshold
      If Exited = True then
        Exit;  -- bit truncated
      Else
        For l in 0 to lenth loop
          Coeffcontnr(l).ZTF := '0';  -- reset the ZTF field
        End Loop;
      End If;
End Loop;  -- decoding for all thresholds

If Exited = True then
           -- add the uncertainty precision to the decoded coefficients
   If copyl/= 0 then  -- threshold is not 1
     For i in 0 to SigNo loop
     (CoeffContnr(DcodedInfo(i).addr).Coeff) :=
     CoeffContnr(DcodedInfo(i).addr).Coeff + Thresh(copyl-1);
     end loop;
   Else
     For i in 0 to SigNo loop
       CoeffContnr(DcodedInfo(i).addr).Coeff :=
       CoeffContnr(DcodedInfo(i).addr).Coeff + 1;
     end loop;
   End If;
End If;

-- correct the signs

For i in 0 to lenth loop
   If DcodedInfo(i).Sign = '1' then
     Null;
   Else
     CoeffContnr(i).Coeff := -(CoeffContnr(i).Coeff);
   End If;
End Loop;


--display result

   For m in 0 to lenth loop
    OutCoeff <= CoeffContnr(m).Coeff;
     addrs <= m;
     wait for 5 ns;
   End Loop;
```

```
  wait;

End DecodeProcess;

End Behavioural;


Configuration CFGNew1 of FullDecoder is

  For
       Behavioural

  End for ;

End CFGNew1;


------------------------------------------------------------------------
------------------------------------------------------------------------
--                        END
```

```
--*******************************************************
--      ENCODER FOR SYNTHESIS
--*******************************************************

------------------------------------------------------------------------
------------------------------------------------------------------------
-- The code here is same as the one for synthesis.  But as pointed out
-- in the synthesis topic multidimensional arrays (arrays of records)
-- have been converted to simple on dimensional array.  Clocking is
-- introduced for providing synchronization, while loops have been
-- converted to for loops and wait statements as such have been emoved.
-- Otherwise the logic is
-- obviously the same.

------------------------------------------------------------------------
------------------------------------------------------------------------
```

```
library IEEE;
library WORK;
use WORK.TypePKGsyn.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

Entity FullEncoder is
   Port(clk : in Bit; codeA: out CodeType) ;

End FullEncoder;

Architecture Behaviouralsyn of FullEncoder is

Type OutPutRA is Array (0 to 400) of CodeType;
-- to contain the code for the coefficients from the encoding process
Type FirstRA is Array (0 to 5) of Bit;
Type CoeffRA is array (0 to lenth) of CoeffType;
Type DSigRA is array (0 to lenth) of SigType;
Type ZTFRA is array (0 to lenth) of Bit;
Type EncodedRA is array (0 to lenth) of Bit;
Type ApproxRA is array (0 to lenth) of CodeType;

begin

EncodeTree: Process  --(CLK)
variable T: integer range 1 to 64;
variable Parent: IndexType;
variable Child: IndexType;
variable OutCode : OutPutRA;
variable m : Integer range 0 to 2048;
   variable q : Integer range 0 to 2048;
   variable SigNo : Integer range 0 to 64; -- Significant coefficient
number.
   variable copy : Integer range 0 to 64;
variable First : FirstRA; -- to indicate whether the significant
coefficient
                          --is the first in the level
   variable Coeff : CoeffRA;
   variable ZTF : ZTFRA;
   variable Encoded : EncodedRA;
   variable DSig : DSigRA;
   variable Zero: ApproxRA;
   variable One: ApproxRA;
   variable Two: ApproxRA;
   variable Three: ApproxRA;
   variable Four: ApproxRA;
   variable Five: ApproxRA;
   variable Six: ApproxRA;

begin
   wait until clk'event and clk = '1';
-- *** GetCoeff ***
Coeff(0):=63;Coeff(1):=-34;Coeff(2):=-31;Coeff(3):=23;Coeff(4):=9;
Coeff(5):=10;Coeff(6):=14;Coeff(7):=-13;Coeff(8):=15;Coeff(9):=14;
Coeff(10):=-9;Coeff(11):=-7;Coeff(12):=3;Coeff(13):=-12;Coeff(14):=-14;
Coeff(15):=8;Coeff(16):=7;Coeff(17):=13;Coeff(18):=3;Coeff(19):=4;
Coeff(20):=-12;Coeff(21):=7;Coeff(22):=6;Coeff(23):=-1;Coeff(24):=5;
```

```
Coeff(25):=-7;Coeff(26):=4;Coeff(27):=-2;Coeff(28):=3;Coeff(29):=9;
Coeff(30):=3;Coeff(31):=2;Coeff(32):=-5;Coeff(33):=9;Coeff(34):=3;
Coeff(35):=0;Coeff(36):=-1;Coeff(37):=47;Coeff(38):=-3;Coeff(39):=2;
Coeff(40):=2;Coeff(41):=-3;Coeff(42):=5;Coeff(43):=11;Coeff(44):=6;
Coeff(45):=-4;Coeff(46):=5;Coeff(47):=6;Coeff(48):=4;Coeff(49):=6;
Coeff(50):=3;Coeff(51):=-2;Coeff(52):=-2;Coeff(53):=2;Coeff(54):=0;
Coeff(55):=4;Coeff(56):=3;Coeff(57):=6;Coeff(58):=0;Coeff(59):=3;
Coeff(60):=3;Coeff(61):=6;Coeff(62):=-4;Coeff(63):=4;

For i in 0 to 63 Loop --lenth loop
  DSig(i) := u;
  ZTF(i):= '0';
  Encoded(i) := '0';
End loop;
-- The DSig, ZTF and Encoded mean the same as in the simulation code
-- except that here they are held in separate arrays.

wait until clk'event and clk = '1';
First := "000000"; --  initialize  the  array  that  indicates  if  the
significant
                   --coefficient found is the first in the level
SigNo := 0;        -- indicates the no of the significants found
m := 0;            -- to serve as an index for the output array
For l in 6 downto 0 loop   -- main loop that does all the 7 sig map
generation of
                           -- coefficients and the subsequent encoding
  T := Thresh(l);  -- choose proper threshold
  Parent := 15;
  Child := lenth;
-------------------------------------------------------------------------
-- GENERATE SIGNIFICANE MAP
-------------------------------------------------------------------------

    For parent in 15 downto 1 loop -- for all parents
      For j in 0 to 3 loop         --   check   four   children   for
significance
        If DSig(child-j) = u then
          If abs(Coeff(child - j))>= T then
            DSig(parent) := '1';
            Exit;
          Else
            DSig(parent) := '0';
          End If;
        ElsIf Dsig(child-j) = '1' then
          DSig(parent) := '1';
          exit;
        Else
          If abs((child -j))>= T then
            DSig(parent) := '1';
            Exit;
          Else
            DSig(parent):= '0';
          End If;
        End If;
      End Loop;   --for one parent
      Child := Child - 4;
    End Loop;  -- for all but one last parent
```

```
   -- we have reached the first parent and its three children in the
subband ***
     For j in 0 to 2 Loop  -- for three children
       If DSig(child -1) = '1' then
         DSig(parent) := '1';
         Exit;
       ElsIf abs(Coeff(child-j)) >= T then
         DSig(parent) := '1';
         Exit;
       Else
         DSig(parent) := '0';
       End If;
     End Loop;
-------------------------------------------------------------------
-- SIGNIFICANCE MAP GENERATION FOR A THRESHOLD COMPLETED
-- DEGIN ASSIGNING CODES
-------------------------------------------------------------------
Child := 1;
   wait until clk'event and clk = '1';
     For parent in 0 to lenth Loop  -- encode all coefficients
       If ZTF(parent) = '1' then
            If parent > 15 then
              Null;
            Else
               For i in 0 to 3 loop
                   ZTF(child +i) := '1';
-- pass the information that a
                                                -- Zerotree root has been
found
-- ahead to the childr
                End loop;
                wait until clk'event and clk = '1';
           End If;
         ElsIf  Encoded(parent) = '1' then
-- if the coefficient has been coded before
-- don't code it.
           Null;
         ElsIf Abs(Coeff(parent))>= T then  -- significant
           If Coeff(parent) < 0 then
             OutCode(m) := NEG;    -- code negative
             m := m + 1;
           Else
             OutCode(m):= POS;     -- code positive
             m := m + 1;
           End If;
-------------------------------------------------------------------
-- START SUCCESSIVE APPROXIMATION QUANTIZATION
-------------------------------------------------------------------
     copy := Abs(Coeff(parent)) - Thresh(1);
       Case 1 is
         When 0 =>  -- threshold is 1
           Null;
         When 6 =>  -- threshold is 64
         wait until clk'event and clk = '1';
           For k in 1 downto 0 loop
             If (copy - Thresh(k)) < 0 then  -- no remainder
```

```
      Case k is
        When 0 =>
          Zero(SigNo):= NO;
        When 1 =>
          One(SigNo) := NO;
        When 2 =>
          Two(SigNo) := NO;
        When 3 =>
          Three(SigNo) := NO;
        When 4 =>
          Four(SigNo) := NO;
        When 5 =>
          Five(SigNo) := NO;
        When 6 =>
          Six(SigNo) := NO;
        When Others =>
          Null;
        End Case;
      Else
        Case k is
          When 0 =>
            Zero(SigNo):= YES;
          When 1 =>
            One(SigNo) := YES;
          When 2 =>
            Two(SigNo) := YES;
          When 3 =>
            Three(SigNo) := YES;
          When 4 =>
            Four(SigNo) := YES;
          When 5 =>
            Five(SigNo) := YES;
          When 6 =>
            Six(SigNo) := YES;
          When Others =>
            Null;
          End Case;
          copy := copy - Thresh(k);
        End If;
      End Loop;
      OutCode(m):= Six(SigNo);
      m := m+1;
    When Others =>
      For k in (l-1) downto 0 loop
        If (copy - Thresh(k)) < 0 then
          Case k is
            When 0 =>
              Zero(SigNo):= NO;
            When 1 =>
              One(SigNo) := NO;
            When 2 =>
              Two(SigNo) := NO;
            When 3 =>
              Three(SigNo) := NO;
            When 4 =>
              Four(SigNo) := NO;
            When Others =>
```

```
                Null;
              End Case;
            Else
              Case k is
                When 0 =>
                  Zero(SigNo):= YES;
                When 1 =>
                  One(SigNo) := YES;
                When 2 =>
                  Two(SigNo) := YES;
                When 3 =>
                  Three(SigNo) := YES;
                When 4 =>
                  Four(SigNo) := YES;
                When Others =>
                  Null;
                End Case;
                copy := copy - Thresh(k);
              End If;
            End Loop;
          End Case;
-- tag the approximate values of the coefficients
Case l is
  When 6 =>
    Null;
  When Others =>
    If First(l) = '0' then
      If SigNo = 0 then
        Null;
      Else
        wait until clk'event and clk = '1';
        for r in 0 to lenth loop  --(SigNo) loop
          If r > SigNo then
            Exit;
          Else
            Case l is
              When 0 =>
                OutCode(m) := Zero(r);
              When 1 =>
                OutCode(m) := One(r);
              When 2 =>
                OutCode(m) := Two(r);
              When 3 =>
                OutCode(m) := Three(r);
              When 4 =>
                OutCode(m) := Four(r);
              When 5 =>
                OutCode(m) := Two(r);
              When Others =>
                Null;
            End Case;
            m := m +1;
          End If;
          End Loop;
        End If;
      First(l):='1';
    End If;
```

```
            End Case;
            SigNo := SigNo + 1;
            Coeff(parent) := 0;
            Encoded(parent) := '1';
        -- code insignificant
          Else
            If DSig(parent) = '1' then
              OutCode(m) := IZ;
              m := m + 1;
            Else
              OutCode(m) := ZTR;
              m := m + 1;
              If parent = 0 then
                Exit;
              ElsIf  DSig(parent) = '0' then
              wait until clk'event and clk = '1';
                For j in 0 to 3 loop
                  ZTF(child +j):= '1';
                End Loop;
              wait until clk'event and clk = '1';
              Else  -- it is a leaf insignificant
                Null;
              End If;
            End If;
          End If;

          If parent >= 15 then
            Null;
          ElsIf parent = 0 then
            child := child + 3;
          Else
            Child := Child + 4;
          End If;
           End Loop;
      -- one threshold loop
      -- set the significance (DSig)found earlier threshold to 0
       For k in 0 to lenth loop
        If DSig(k) = u then
          Null;
        Else
          DSig(k) := '0';
        End If;
        ZTF(k) := '0';
      End Loop;
End Loop; -- all the 7 arranging and encoding loop
wait until clk'event and clk = '1';

 -- display the code
For n in 0 to 65 loop  --(m-1)
  CodeA <= OutCode(n);
End Loop;
wait until clk'event and clk = '1';

End Process;
-- end of EncodeTree process
End Behaviouralsyn;
```

```
-------------------------------------------------------------
-- CONFIGURATION
-------------------------------------------------------------
Configuration  CFGFull_Coder of FullEncoder is
  For
    Behaviouralsyn
  End For;
End CFGFull_Coder;

-------------------------------------------------------------
-------------------------------------------------------------
--                          END
-------------------------------------------------------------
```

```
--*************************************************************
--       3 PARALLEL PROCESSOR ENCODER
--*************************************************************
-------------------------------------------------------------
-------------------------------------------------------------
-- The same strategy of significance map generation, code assignment
-- and successive approximation quantization is used.  There are three
-- fundamental differences from the single processor encoder.
-- The first parent in each of the three processors has only one child
-- And to determine the descendent significance of the first parent
-- information is required from the other two processors. And each
-- processor proceses only a third of the coefficients, i.e. one
-- processor encodes one subband of the three Processor for subband A's
-- code is shown in full.  Since the other two
-- are the same we show only how the three relate to the fourth
-- Processor and donot repeat the code for the other two processors in
-- the interest of space. They are just the repetition of the first one
```

```
-- The same types and variable used earlier in the single processor are
-- used here as well.  For the three processors the variables and
-- signals are appended with A, B, C so as to differentiate their use
-- in three processors
-------------------------------------------------------------------------
-------------------------------------------------------------------------


library IEEE;
library WORK;
use WORK.TypePKG.all;
Use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
-- Embedded Zerotree Wavelet Algorithm.
-- Image 8 by 8
-- 8 bit implementation

Entity FullEncoder is
  Port(codeA, codeB, codeC : out CodeType) ;
End FullEncoder;


Architecture Behavioural1 of FullEncoder is

  Type CoeffRec is
    record
    DSig: SigType;
    -- '1' for yes, '0' for no and 'u'
-- for no children (leaves)
    ZTF: Bit;
    Encoded: Bit;
    Coeff: CoeffType;
  End Record;


-- variables have the same meaning as before in the single processor
case
-- DSig is to indicate if the descendents have been found to be
significant
-- ZTF for zerotree root found ahead or at the present level
-- Encoded to indicate if it has been encoded for any prevoius
threshold
-- Coeff to contain the actual coefficient

Type RAOfRec is Array (0 to lenth) of CoeffRec;

-- to contain the input coefficients and information on each
coefficient
Type OutPutRA is Array (0 to 150) of CodeType;
-- to contain the code for the coefficients from the encoding process

Type AllQntRA is Array (0 to 21, 0 to 6) of CodeType;
-- to contain the quantization values of all the codes
Type FirstRA is Array (0 to 5) of Bit;

Signal A, B, C : Bit;
```

```
Signal Acode, Bcode, Ccode: CodeType;

signal MainParent: CoeffType;

signal ASig, BSig, CSig: Bit;

signal AYes, BYes, CYes: Bit;

signal DoneA, DoneB, DoneC : Bit;

signal FromA, fromB, fromC: Bit;

signal ADone, BDone, CDone : Bit;

signal TCopy: Integer range 1 to 64;


begin


ControlProcesses: Process
begin
  -- control the start of the three processes
     A <= '1';
     B <= '1';
     C <= '1';
     Wait Until Ayes = '1' and Byes = '1' and Cyes = '1';
     A <= '0';
     B <= '0';
     C <= '0';
  End Process;

Subband_A: Process  -- to process subband A

variable T: integer range 1 to 64;

variable Contnr: RAOfRec;

variable Parent: IndexType;

variable Child: IndexType;

variable OutCode : OutPutRA;

variable m : Integer range 0 to 150;
variable ApproxValue : AllQntRA;
variable SigNo : Integer range 0 to 21; -- Significant coefficient
number.
variable copy : Integer range 0 to 128;
variable First : FirstRA;
-- to indicate whether the significant coefficient is the
-- first in the level

begin
```

```
  -- process subband_A

-- Initialization
For i in 0 to lenth loop
   Contnr(i).DSig := u;
   COntnr(i).ZTF:= '0';
   Contnr(i).Encoded := '0';
End loop;

First := "000000";   -- initialize the array that indicates if the
significant coefficient found is the first in the level
Contnr(0).Coeff:=   -83;Contnr(1).Coeff:=   -31;Contnr(2).Coeff:=   15;
Contnr(3).Coeff:= 14;Contnr(4).Coeff:= -9;Contnr(5).Coeff:= -7;
Contnr(6).Coeff:=     -5;     Contnr(7).Coeff:=     9;Contnr(8).Coeff:=
3;Contnr(9).Coeff:=     0;Contnr(10).Coeff:=     -1;Contnr(11).Coeff:=
47;Contnr(12).Coeff:=   -3;   Contnr(13).Coeff:=   2;Contnr(14).Coeff:=
2;Contnr(15).Coeff:=   -3;   Contnr(16).Coeff:=   5;Contnr(17).Coeff:=
11;Contnr(18).Coeff:=   6;   Contnr(19).Coeff:=   -4;Contnr(20).Coeff:=
5;Contnr(21).Coeff:= 6;

--  Inputs/Coefficients ready
m := 0; -- to serve as an index for the output array
SigNo := 0;  -- indicates the no of the significants found

For l in 6 downto 0 loop  -- main loop that does all the encoding
T := Thresh(l);
Parent:= 5;
Child := lenth;
-------------------------------------------------------------
-- GENERATE SIGNFICANCE MAP
-------------------------------------------------------------
While Parent >= 1 loop
for j in 1 to 4 loop
     If Contnr(Child - j+ 1).DSig = u then
       If abs(Contnr(Child - j + 1).coeff)>= T then
         Contnr(Parent).DSig := '1';
         Exit;
       Else
         Contnr(Parent).DSig := '0';
       End If;
     ElsIf Contnr(Child - j + 1).Dsig = '1' then
       Contnr(Parent).DSig := '1';
       exit;
     Else -- Contnr(Child - j + 1).DSig = '0'
       If abs(Contnr(Child - j + 1).coeff)>= T then
         Contnr(Parent).DSig := '1';
         Exit;
       Else
         Contnr(Parent).DSig := '0';
         End If;
       End If;

     End Loop;
     Child := Child - 4;
     Parent := Parent - 1;
   End Loop;
   -- we have reached the first parent and its only child in the subband
```

```
  If Contnr(1).DSig = '1' then
    Contnr(0).Dsig := '1';
  Else
    If abs(Contnr(1).Coeff) >= T then
      Contnr(0).DSig := '1';
    Else
      Contnr(0).DSig := '0';
    End If;
  End If;
  ------------------------------------------------------------
-- SIGNIFICANCE MAP GENERATION COMPLETED
-- ASSIGN CODES
  ------------------------------------------------------------
  -- send zeroth (0) coefficient and the significantce of its first
descendents
-- to another process to
  -- code the main parent

    If Contnr(0).Encoded = '0' then
    -- do the folowing only if the main parent hadn't been encoded
      MainParent <= Contnr(0).Coeff;
      If Contnr(0).DSig = '1' then
        ASig <= '1';  -- main parent from processor A has
              -- descendent significant
      Else
        ASig <= '0';  -- main parent does not have any significant
              -- descendent in subband A
      End If;
      TCopy <= T;
      FromA <= '1'; -- subband A has reached to process the main
            -- parent
      Wait until ADone = '1';
      FromA <= '0';
      OutCode(m) := ACode;
      m := m + 1;
      Case Acode is
        When POS | NEG =>
        -- start quantization for main parent
        copy := Abs(contnr(0).Coeff) - Thresh(1);
          For k in (1-1) downto 0 loop
            If (copy - Thresh(k)) < 0 then
              ApproxValue(SigNo,k) := NO;
            Else
              ApproxValue(SigNo,k) := YES;
              copy := copy - Thresh(k);
            End If;
          End Loop;
        -- tag the approximate values of the coefficients
        Case 1 is
          When 6 =>
            --OutCode(m):= ApproxValue(SigNo,1);
            --m := m + 1;
            Null;
          When Others =>
            If First(1) = '0' then
              for r in 0 to (SigNo-1) loop
            OutCode(m) := ApproxValue(r,1);
```

```
            m := m + 1;
          End loop;
          First(l):='1';
        End If;
        End Case;
        -- increment the no of significants found
        If SigNo < 21 then
          SigNo := SigNo + 1;
        End If;
        Contnr(0).Encoded := '1';
        Contnr(0).Coeff := 0;

      When ZTR =>
        Contnr(0).ZTF := '1';
      When Others =>
        Null;
      End Case;
    End If;
    -- encode the rest
    If Contnr(0).ZTF = '1' then
      Null;
    Else -- only if the main parent is not a zerotree root
      If Contnr(1).Encoded = '1' then
        Null;
      Else
        If Abs(Contnr(1).coeff)>= T then
          If Contnr(1).Coeff < 0 then
            OutCode(m) := NEG;
          Else
            OutCode(m):= POS;
          End If;
          m:= m + 1;

      copy := Abs(contnr(1).Coeff) - Thresh(l);
      For k in (l-1) downto 0 loop  -- prepare the table of
                  -- coefficients
        If copy - Thresh(k) < 0 then
          ApproxValue(SigNo,k) := NO;
        Else
          ApproxValue(SigNo,k) := YES;
          copy := copy - Thresh(k);
        End If;
      End Loop;
      -- tag the approximate values of the coefficients
      Case l is
        When 6 =>
          OutCode(m):= ApproxValue(SigNo,l);
          m := m + 1;
        When Others =>
          If First(l) = '0' then
            for r in 0 to (SigNo-1) loop
              OutCode(m) := ApproxValue(r,l);
              m := m + 1;
            End loop;
            First(l):='1';
          End If;
        End Case;
```

```
            If SigNo < 21 then
              SigNo := SigNo + 1;
            End If;

            Contnr(1).Coeff := 0;
            Contnr(1).Encoded := '1';

          Else
            If Contnr(1).DSig = '1' then
              OutCode(m) := IZ;
              m := m + 1;
             ElsIf Contnr(1).DSig = '0' then
              OutCode(m) := ZTR;
              Contnr(1).ZTF := '1';
              m := m + 1;
            Else
              Null;
            End If;
          End If;
        End If;
      -- start coding the other descendents from 2 and all

      If Contnr(1).ZTF = '1' then
        Null;
      Else
        parent := 1;
        child := 2;
        Loop   --till all the descendents have been encoded, happens
when
          -- parent = 5 is over
          If Contnr(Parent).ZTF = '1' then
            For i in 1 to 4 loop
              Contnr(Child + i -1).ZTF := '1';  -- pass the
              -- information that a Zerotree
              -- root has been found ahead to
              -- the children
            End loop;
          Else
            For i in 1 to 4 loop
              If Contnr(Child + i -1).Encoded = '1' then
                -- if the coefficient has been coded before don't
                -- code it
                Null;
              Else
          If Abs(Contnr(Child + i - 1).Coeff) >= T then

                If Contnr(Child + i - 1).Coeff < 0 then
                     OutCode(m) := NEG;
                  Else
                     OutCode(m):= POS;
                  End If;
                  m:= m + 1;

            -- start quantization
            copy := Abs(contnr(Child + i -  1).Coeff) - Thresh(l);
            Case l is
```

```
                When 0 =>
                  Null;
                When Others =>
                  For k in (l-1) downto 0 loop
                    If (copy - Thresh(k)) < 0 then
                      ApproxValue(SigNo,k) := NO;
                    Else
                      ApproxValue(SigNo,k) := YES;
                      copy := copy - Thresh(k);
                    End If;
                  End Loop;
                    End Case;
                  -- tag the approximate values of the coefficients
                  Case l is
                    When 6 =>
                      OutCode(m):= ApproxValue(SigNo,1);
                      m := m + 1;
                    When Others =>
                      If First(l) = '0' then
                         for r in 0 to (SigNo-1) loop
                         OutCode(m) :=
                         ApproxValue(r,l);
                           m := m + 1;
                         End loop;
                         First(l):='1';
                      End If;
                  End Case;
                  If SigNo < 21 then
                    SigNo := SigNo + 1;
                  End If;
                  Contnr(Child + i - 1).Coeff := 0;
                  Contnr(Child + i - 1).Encoded := '1';
                Else
                  If Contnr(Child + i - 1).DSig = '1' then
                    OutCode(m) := IZ;
                    m := m + 1;
                   ElsIf Contnr(Child + i - 1).DSig = '0' then
                    OutCode(m) := ZTR;
                    Contnr(Child + i - 1).ZTF := '1';
                    m := m + 1;
                  Else -- if it is a leaf coefficient
                    OutCode(m):= ZTR;
                    m := m + 1;
                  End If;
                End If;
              End If;
            End Loop;
          End If;
          If Parent >= 5 then
            Exit;
          Else
            Child := Child + 4;
            Parent := Parent + 1;
          End If;
        End loop;
      End If;
    End If;
```

```
-- reset the fields
For k in 1 to (lenth + 1) loop
     If Contnr(k-1).DSig = u then
       null;
     Else
       Contnr(k-1).DSig := '0';
     End If;
       Contnr(k-1).ZTF := '0';
     End Loop;

    End Loop; -- all the 7 arranging and encoding loop

-- display the code

   For n in 0 to m-1 loop
        CodeA <= OutCode(n);
        wait for 5 ns;
   End Loop;
   Ayes <= '1';
   Wait until A = '1';

End Process; -- end of process Subband_A




Subband_B: Process


-------------------------------------------------------------------
-------------------------------------------------------------------
-- SAME AS PROCESSOR FOR SUBBAND_A, INPUTS ARE FROM SUBBAND_B
-- GENERATES FIGNIFICANCE MAP, SENDS THE MAIN PARENT'S DESCENDENDT
-- INFORMATION TO A FOURTH PROCESSOR AND ENCODES AS DONE IN SUBBAND_A
-------------------------------------------------------------------
-------------------------------------------------------------------

Subband_C: Process

-------------------------------------------------------------------
-------------------------------------------------------------------
-- SAME AS PROCESSOR FOR SUBBAND_A AND B, BUT INPUTS FROM SUBBAND_C
-- GENERATES FIGNIFICANCE MAP, SENDS THE MAIN PARENT'S DESCENDENDT
-- INFORMATION TO A FOURTH PROCESSOR AND ENCODES AS DONE IN SUBBAND_A
-------------------------------------------------------------------
-------------------------------------------------------------------


-------------------------------------------------------------------
-------------------------------------------------------------------
-- PROCESS THAT DETERMINES THE DESCENDENT SIGNIFICANCE OF THE MAIN
-- PARENT AND SENDS THE INFORMATION TO THE INDIVIDUAL PROCESSORS
-------------------------------------------------------------------
-------------------------------------------------------------------
```

```
DetMain: Process

   variable Thold : Integer range 1 to 64;
   variable CopyCoeff: CoeffType;

   begin

      Wait Until FromA = '1' and FromB = '1' and FromC = '1';
                  -- get information from all the
                  -- three processes of the th~ ~ subbands
      ADone <= '0';
      BDone <= '0';       -- keep the processors waiting
      CDone <= '0';
      CopyCoeff := MainParent;
      If Abs(CopyCoeff) >= Thold then
         If CopyCoeff < 0 then
            ACode <= NEG;
            BCode <= NEG;
            CCode <= NEG;
         Else
            ACode <= POS;
            BCode <= POS;
            CCode <= POS;
         End IF;

      Else
         If ASig = '1' or BSig = '1' or CSig = '1' then
            ACode <= IZ;
            BCode <= IZ;
            CCode <= IZ;
         Else
            ACode <= ZTR;
            BCode <= ZTR;
            CCode <= ZTR;
         End If;
      End If;

      ADone <= '1';
      BDone <= '1';
      CDone <= '1';
   End Process;
End Behavioural1;


Configuration CFGFull_Coder of FullEncoder is

   For
      Behavioural1
   End For;
End CFGFull_Coder;

-------------------------------------------------------------------
--                          END
-------------------------------------------------------------------
-------------------------------------------------------------------
```

```
--************************************************** **************
--         PARALLEL ARCHITECTURE DECODER
--
************************************************************************
***
------------------------------------------------------------------------
------------------------------------------------------------------------


-- The three parallel decoders are exactly the same.  So, only one is
-- shown. As explained in chapter 4 this decoder is a smaller version
-- of the single processor decoder, though it does not necessarily be.
-- It is atleast as implemented in this project.  The only main
-- difference is that the first parent has only one child here, whereas
-- in the single processor codec we had three children.  Rest, is the
-- same.  The variable names used here are same as those used in the
-- other decoder.  Hence only modest explanations are provided here as
-- they are just the repetition of what we have been explained
```

```
-- thoroughly before
--------------------------------------------------------------------------
--------------------------------------------------------------------------



library IEEE;
library WORK;
use WORK.TypePKG.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

Entity FullDecoder is

Port(codeout: out CodeType) ;
End FullDecoder;

Architecture Behavioural of FullDecoder is

Type CoeffRec is
  Record
    Coeff : CoeffType;
    ZTF : Bit;
    Decoded : Bit;
  End Record;


Type CoeffRA is Array (0 to lenth) of CoeffRec;
Type CodeRA is Array (0 to 264) of CodeType;
Type FrstIndictr is Array (0 to 5) of Bit;
Type AddrSign is
     record
       Addr : Integer range 0 to 21;
       Sign : Bit;
End Record;

Type RAddrSign is Array (0 to lenth) of AddrSign;   -- to contain the
information on the decoded codes

begin

DecodeProcess: Process

   variable P       : Integer range 0 to 264;
   variable CoeffContnr    : CoeffRA;
   variable CodeContnr     : CodeRA;
   variable parent    : Integer range 0 to 6;
   variable child     : Integer range 0 to lenth;
   variable DcodedInfo     : RAddrSign;
   variable SigNo     : Integer range 0 to 22;   --significant number
   variable First       : FrstIndictr;

   begin

CodeContrr(0):=NEG;      CodeContnr(1):=ZTR;      CodeContnr(2):=      IZ;
CodeContnr(3):=ZTR;      CodeContnr(4):=IZ;      CodeContnr(5):=      ZTR;
CodeContnr(6):=ZTR;    CodeContnr(7):=    ZTR;    CodeContnr(8):=    POS;
```

```
CodeContnr(9):=NO; CodeContnr(10) := ZTR;  CodeContnr(11)     :=     ZTR;
CodeContnr(12):=NEG;  CodeContnr(13):=  YES;      CodeContnr(14)  :=  NO;
CodeContnr(15):= ZTR; CodeContnr(16) := ZTR;  CodeContnr(17)    :=    ZTR;
CodeContnr(18):= ZTR; CodeContnr(19) := POS;  CodeContnr(20)    :=    NO;
CodeContnr(21):= YES;CodeContnr(22) := YES;CodeContnr(23)     :=     POS;
CodeContnr(24):= NEG;CodeContnr(25) := ZTR;CodeContnr(26)     :=     ZTR;
CodeContnr(27):= POS;CodeContnr(28) := ZTR;CodeContnr(29)     :=     ZTR;
CodeContnr(30):= ZTR;CodeContnr(31) := ZTR;CodeContnr(32)     :=     ZTR;
CodeContnr(33):= ZTR;CodeContnr(34) := ZTR;CodeContnr(35)     :=     ZTR;
CodeContnr(36):= POS;CodeContnr(37) := NEG;CodeContnr(38)     :=     NO;
CodeContnr(39):= YES;CodeContnr(40) := YES;CodeContnr(41)     :=     YES;
CodeContnr(42):= YES;CodeContnr(43) := NO; CodeContnr(44)     :=     NO;
CodeContnr(45):= NO;CodeContnr(46) := NEG; CodeContnr(47)     :=     ZTR;
CodeContnr(48):= ZTR;CodeContnr(49) := ZTR;CodeContnr(50)     :=     ZTR;
CodeContnr(51):= ZTR;CodeContnr(52) := ZTR;CodeContnr(53)     :=     ZTR;
CodeContnr(54):= POS;CodeContnr(55) := POS;CodeContnr(56)     :=     NEG;
CodeContnr(57):= POS;CodeContnr(58) := POS;CodeContnr(59)     :=     POS;
CodeContnr(60):= YES;CodeContnr(61) := YES;CodeContnr(62)     :=     YES;
CodeContnr(63):= YES;CodeContnr(64) := YES;CodeContnr(65)     :=     NO;
CodeContnr(66)  := NO;CodeContnr(67) := YES;CodeContnr(68)    :=     YES;
CodeContnr(69):= NO;CodeContnr(70) := NO;  CodeContnr(71)     :=     YES;
CodeContnr(72):= NO;CodeContnr(73) := NO;  CodeContnr(74)     :=     YES;
CodeContnr(75):= ZTR;CodeContnr(76) := ZTR;CodeContnr(77)     :=     NEG;
CodeContnr(78):= POS;CodeContnr(79) := POS;CodeContnr(80)     :=     NEG;
CodeContnr(81):= ZTR;CodeContnr(82) := NEG;CodeContnr(83)     :=     YES;
CodeContnr(84):= YES;CodeContnr(85) := YES;CodeContnr(86)     :=     YES;
CodeContnr(87):= NO;CodeContnr(88) := YES; CodeContnr(89)     :=     YES;
CodeContnr(90):= YES;CodeContnr(91) := YES;CodeContnr(92)     :=     YES;
CodeContnr(93):= YES;CodeContnr(94) := NO; CodeContnr(95)     :=     NO;
CodeContnr(96)  := YES;CodeContnr(97) := NO;CodeContnr(98)    :=     YES;
CodeContnr(99):=  YES;CodeContnr(100)   :=  NO;CodeContnr(101)  :=  NO;
CodeContnr(102):= YES;CodeContnr(103) := NEG; CodeContnr(104) := NEG;

-- initialize

For m in 0 to lenth loop
    CoeffContnr(m).Coeff := 0;
    CoeffContnr(m).ZTF := '0';
    CoeffContnr(m).Decoded := '0';
    If m <= 5 then
  First(m) := '0';
    End If;
End Loop;

P := 0;  -- index the codecontr
SigNo := 0;
For l in 6 downto 0 Loop
    parent:= 0;
    child := 0;

   For j in 0 to lenth loop  -- for one threshold
        If Coeffcontnr(j).Decoded = '1' then
       null;

  Else
      If Coeffcontnr(j).ZTF = '1' then
```

```
          null;

       Else

          Case Codecontnr(p) IS
         When ZTR =>
                   If ((j = 0) OR (j = 1)) then

          parent:= 0;

          child := 0;

          p := p + 1;

          Exit;
        Else
          If ((parent /= 0) AND (parent <= 5)) then
               -- it is not a leaf
               -- so mark its children
              For k in 0 to 3 loop
            CoeffContnr(child + k).ZTF := '1';
       End Loop;
          End If;
        End If;
         When POS | NEG =>
            CoeffContnr(j).Decoded := '1';
            SigNo := SigNo + 1;
        DcodedInfo(SigNo-1).Addr := j;
        If Codecontnr(p) = POS then
         DcodedInfo(j).sign := '1';
        Else
          DcodedInfo(j).sign := '0';
        End If;
        CoeffContnr(j).Coeff := (Thresh(l));
        If l <= 5 then
          If First(l) = '0' then
            -- this is the first significant in the level
-- so add precision to the previously found
-- coefficients
              For i in 1 to SigNo-1 loop
               If Codecontnr(p + i) = YES then
                   addrs <= DcodedInfo(i-1).addr;
                   wait for 5 ns;
              addrs <= 22;
              wait for 5 ns;
                  CoeffContnr(DcodedInfo(i-1).Addr).Coeff :=
           CoeffContnr(DcodedInfo(i-1).Addr).Coeff +
                 (Thresh(l));
               Else
              Null;                                    End If;
           End Loop;
           First(l) := '1';
           p := p + SigNo -1;
          End If;
        End If;

         When Others =>
```

```
            Null;
    End Case;
    p := P + 1;  -- prepare to read the next code
      End If;
  End If;
  -- house keeping
  If j >= 21 then

      Exit;

  Else

      If j >= 1 then
          If parent = 0 then
        Parent := 2;
        child := 6;
    Else
        Case parent IS
      When 6 =>
                Null;
      When 5 =>
          Parent := parent + 1;
      When Others =>
          Parent := parent + 1;
          child := child + 4;
        End Case;
    End If;
      End If;
  End If;
    End Loop;
    For l in 0 to lenth loop
        Coeffcontnr(l).ZTF := '0';
    End Loop;

End Loop;

-- correct the signs

For i in 0 to lenth loop
  If DcodedInfo(i).Sign = '1' then
    Null;
  Else
    CoeffContnr(i).Coeff := - (CoeffContnr(i).Coeff);
  End If;
End Loop;


--display result

  For m in 0 to lenth loop

    OutCoeff <= CoeffContnr(m).Coeff;
    wait for 5 ns;
  End Loop;

  Wait;
```

```
End Process;


End Behavioural;


Configuration CFGNew1 of FullDecoder is

   for Behavioural
   End for ;

End CFGNew1;

 ------------------------------------------------------------------------
------------------------------------------------------
               END
------------------------------------------------------------------------
------------------------------------------------------------------------
```

```
--************************************************************************
-- Significance Map Generator: Code that has been Successfully
-- Synthesized
--************************************************************************
------------------------------------------------------------------------
------------------------------------------------------------------------
-- This the one logical portion of the encoder that has been
-- synthesized. Significance Generation code has appeared before. Of
-- noteworthy is the replacement of records by arrays, introduction of
-- clocking, use of only for loops, and no wait statement as such being
-- used.  It is only used with rising edge of a clock. Otherwise, here
-- too the codes are similar to those found earlier
------------------------------------------------------------------------
------------------------------------------------------------------------


library IEEE;
library WORK;
```

```
use WORK.TypePKGsyn.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

-- Embedded Zerotree Wavelet Algorithm.
-- Image 8 by 8
-- 8 bit implementation

Entity SigMapGen is
   Port(clk : in Bit; codeA: out CodeType) ;
End SigMapGen;


Architecture Behave of SigMapGen is

Type OutPutRA is Array (0 to 200) of CodeType;
-- to contain the code for the coefficients from the encoding process


Type FirstRA is Array (0 to 5) of Bit;
Type CoeffRA is array (0 to lenth) of CoeffType;
Type DSigRA is array (0 to lenth) of SigType;
Type ZTFRA is array (0 to lenth) of Bit;
Type EncodedRA is array (0 to lenth) of Bit;
Type ApproxRA is array (0 to lenth) of CodeType;

begin

Arrange: Process  --(CLK)
variable T: integer range 1 to 64;

variable Parent: IndexType;

variable Child: IndexType;

variable OutCode : OutPutRA;

variable m : Integer range 0 to 2048;
variable q : Integer range 0 to 2048;
variable SigNo : Integer range 0 to 64;
-- Significant coefficient number.
variable copy : Integer range 0 to 64;
variable First : FirstRA; -- to indicate whether the significant
coefficient is the first in the level
variable Coeff : CoeffRA;
variable ZTF : ZTFRA;
variable Encoded : EncodedRA;
variable DSig : DSigRA;
variable Zero: ApproxRA;
variable One: ApproxRA;
variable Two: ApproxRA;
variable Three: ApproxRA;
variable Four: ApproxRA;
variable Five: ApproxRA;
variable Six: ApproxRA;
```

```
begin

wait until clk'event and clk = '1';

-- *** GetCoeff here ***
  For i in 0 to 63  Loop --lenth loop
    DSig(i) := u;
    ZTF(i):= '0';
    Encoded(i) := '0';
End loop;
wait until clk'event and clk = '1';
First := "000000";   -- initialize the array that indicates if the
significant        -- coefficient found is the first in the level
  SigNo := 0;  -- indicates the no of the significants found
  m := 0; -- to serve as an index for the output array
  Parent := 15;
  Child := lenth;
    For parent in 15 downto 1 loop
    for j in 0 to 3 loop
      If DSig(child-j) = u then
        If abs(Coeff(child - j))>= T then
          DSig(parent) := '1';
          Exit;
        Else
          DSig(parent) := '0';
        End If;
      ElsIf Dsig(child-j) = '1' then
        DSig(parent) := '1';
        exit;
      Else
        If abs((child -j))>= T then
          DSig(parent) := '1';
          Exit;
        Else
          DSig(parent):= '0';
        End If;
      End If;
    End Loop;   --for one parent
    Child := Child - 4;
  End Loop;  -- all parents

  -- we have reached the first parent and its three children in the
subband
  For j in 0 to 2 Loop
       If DSig(child -1) = '1' then
       DSig(parent) := '1';
       Exit;
    ElsIf abs(Coeff(child-j)) >= T then
      DSig(parent) := '1';
      Exit;
    Else
      DSig(parent) := '0';
    End If;
  End Loop;
End Process;
End Behave;
```

```
Configuration CFG_Arranger of SigMapGen is
   For
     Behave
   End For;
End CFG_Arranger;

--------------------------------------------------------------------------
--------------------------------------------------------------------------
--                       END
--------------------------------------------------------------------------
--------------------------------------------------------------------------
```

# Bibliography

[1] A.Ayurbuch, D.Lazar and M.israeli, "Image Compression Using Wavelet Transform and Multiresolution Decomposition", *IEEE, 1996.*

[2] A. Laine (1993). Wavelet Theory and Application.
Boston: Kluwer Academic Publishers.

[3] A.M.Rassau, K.Eshragian, H.Cheung, S.W.Lachowicz, T.C.B.Yu, W.A.Crossland and T.D.Wilkinson, "Smart Pixel Implementation of a 2-D Parallel Nucleic Wavelet Transform for Mobile Multimedia Communications" Copy provided by Dr.Stefan Lachowicz.

[4] A.N. Netravali & G.B. Haskell (1989). Digital Pictures; Representation and Compression. New York: Plenum Press

[5] A. Savla (1998). Gallium Arsenide Implementation of a Triangular FIR Filter for Discrete Wavelet Transforms. Engineering Report, Edith Cowan University

[6] A.S.Lewis and G.Knowles, "Image Compression Using the 2-D Wavelet Transform", *IEEE Transactions on Image Processing, vol.1, no 2, April, 1992.*

[7] C.Chakrabarti and C.Mumford, "Efficient Realizations of Encoders and Decoders Based on the 2-D Discrete Wavelet Transform", *IEEE, 1999.*

[8] C.K. Chui (1992). An Introduction to Wavelets.
Boston: Academic Press Inc.

[9] D.W. Knapps (1996). Behavioral Synthesis
New Jersey: Prentice Hall PTR.

[10] E. Hernandez, G. Weiss (1996). A First Course on Wavelets.
New York: CRC Press.

[11] G.Alagoda. "An Integration of Zerotrees into the Intelligent Pixel Architecture". Copy provided by Dr.Stefan Iachowicz.

[12] J.Bae and V.K.Prasana. "A Fast and Area-Efficient VLSI Architecture for Embedded Image Coding". Copy provided by Dr. Stefan Iachowicz.

[13] J.C. Russ (1995). The Image Processing Handbook
Boca Raton: CRC Press

[14] J.M Shapiro, "Embedded Image Coding Using Zerotrees of Wavelet Coefficients" *IEEE Transactions on Signal Processing, vol.41, no.12, Dec 1993.* Copy provided by Dr.Stefan Lachowicz.

[15] K.C. Chang (1997). Digital Design and Modelling with VHDL and Synthesis
California: IEEE Computer Society Press.

[16] M.A.Coffey and D.M.Etter, "Image Coding with the Wavelet Transform", *IEEE, 1995.*

[17] M.Antonini, M.Barlaud, P.Mathieu and I.Daubechies, "Image Coding Using Wavelet Transform", *IEEE Transactions on Image Processing, vol.1, no.2, April, 1992.*

[18] M.Barlaud(1994). Wavelets in Image Communication.
Amsterdam: ELSEVIER

[19] M.K.Mandal, S.Panchanathan and T.Aboulnasr, "Wavelets for Image Compression" *IEEE 1994.*

[20] M. Prokein (1998). Tutorial: Preactical Excercises with Synopsys Tools (SGE/VSS).

[21] M.Shahshahani, "Wavelets and Image Compression", *IEEE, 1992.*

[22] M.Vetterli, & J.Kovacevic (1995). Wavelet and Subband Coding
New jersey: Prentice Hall PTR

[23] P.N. Topiwala (1998). Wavelet Image and Video Coding.
Massachusetts: Kluwer Academic Publishers

[24] R.C. Gonzalez, R.E. Woods (1993). Digital Image Processing
Massachusetts: Addison-Wesley Publishing Company.

[25] R. Geissler, S. Bulach (1998). VHDL Manual. WWW Site: http://mikro.e-technik.uni-ulm.de/vhdl/anl-engl.vhd/html/vhdl-all-e.html., University of Ulm.

[26] R. Polikar (1999). Wavelet Tutorial.
WWW Site: http://www.public.iastate.edu/~rpolikar/WAVELETS/WTpart1.html.,
Iowa State University.

[27] R.Rabbani, & P.W. Jones (1991). Digital Image Compression Techniques.
Bellingham: Spie Optical Engineering Press.

[28] S.T. Solari, (1997). Digital Video & Audio Compression.
New York: McGraw Hill.

[29] V. Bhaskaran (1997). Image and Video Compression Standards: Algorithms and Architectures. Boston: Kluwer Academic Publishers.

[30] W. Kou (1995). Digital Image Compression.
Boston: Kluwer Academic Publishers.

[31] Y. Meyer (1993). Wavelets: Algorithms and Applications.
Philadelphia: Society for Industrial and Applied Mathematics.

[32] Y.X.Zhong,"Advances in coding and Compression", IEEE Communications magazine, vol.31, no.7, July,1993.

[33] Z. Navabi (1993). VHDL: Analysis and Modelling of Digital Systems. New York: McGraw Hill PTR.