VLSI IMPLEMENTATION OF LOSSLESS JPEG2000 IMAGE CODING SYSTEM

A MASTER'S THESIS

in

Electrical - Electronics Engineering
Atılım University

by

MURAT DİLAVER VURAL

AUGUST 2009

VLSI IMPLEMENTATION OF LOSSLESS JPEG2000 IMAGE CODING SYSTEM

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

OF

ATILIM UNIVERSITY

BY

MURAT DİLAVER VURAL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

THE DEPARTMENT OF ELECTRICAL - ELECTRONICS ENGINEERING

AUGUST 2009

Approval of the Graduate School of Natural and Applied Sciences, Atılım University.

Prof. Dr. Abdurrahim Özgenoğlu
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Ali Kara
Head of Department

This is to certify that we have read the thesis "VLSI IMPLEMENTATION OF LOSSLESS JPEG2000 IMAGE CODING SYSTEM" submitted by "Murat Dilaver Vural" and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Dr. Ruşen Öktem
Co-supervisor

Asst. Prof. Dr. Hakan Tora
Supervisor

Examining Committee Members

Asst. Prof. Dr. Hakan Tora _____
Asst. Prof. Dr. Mehmet Efe Özbek _____
Asst. Prof. Dr. Nergiz Ercil Çağıltay _____

Date: August 14, 2009

I declare and guarantee that all data, knowledge and information in this document has been obtained, processed and presented in accordance with academic rules and ethical conduct. Based on these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Murat Dilaver VURAL

ABSTRACT

VLSI IMPLEMENTATION OF LOSSLESS JPEG2000 IMAGE CODING SYSTEM

Vural, Murat Dilaver

M.S., Electrical and Electronics Engineering Department

Supervisor: Asst. Prof. Dr. Hakan Tora

Co-Supervisor: Dr. Ruşen Öktem

August 2009, 87 pages

This thesis discusses lossless JPEG2000 Image Coding System and implements a VLSI application of the system. The system is implemented using Verilog hardware description language and high-level synthesis tools. Implemented system is synthesized using logic synthesis tools and results are compared with reference applications. The implemented system can be realized either on an FPGA device or on an ASIC device once the memories stripped. The implemented system is designed to run in streaming mode, except the entropy coder part.

Keywords: JPEG2000, Lossless image coding, VLSI Implementation

ÖZ

KAYIPSIZ JPEG2000 GÖRÜNTÜ KODLAMA SİSTEMİ VLSI UYGULAMASI

Vural, Murat Dilaver

Yüksek Lisans, Elektrik ve Elektronik Muhendisligi Bölümü

Tez Yöneticisi: Yrd. Doç. Dr. Hakan Tora

Ortak Tez Yöneticisi: Dr. Ruşen Öktem

Ağustos 2009, 87 sayfa

Bu tez, kayıpsız JPEG2000 Görüntü Kodlama Sistemi'ni inceler ve sistemin bir VLSI uygulamasını gerçekleştirir. Sistem Verilog donanım tanımlama dili ve yüksek seviye sentezleme araçları kullanılarak gerçekleştirilmiştir. Gerçekleştirilen sistem mantık sentezi araçları ile sentezlenmiş ve sonuçları referans uygulamalar ile karşılaştırılmıştır. Uygulanan sistem FPGA aygıtlar veya hafıza kısımları ayrıldıktan sonra ASIC aygıtlar üzerinde gerçekleştirilebilir

Anahtar Kelimeler: JPEG2000, Kayıpsız görüntü kodlama, VLSI uygulaması.

To My Family

ACKNOWLEDGMENTS

I would like to thank Dr. Ruşen Öktem for her guidance and keen insight throughout the work. I would like to thank Asst. Prof. Dr. Hakan Tora for his contributions in this thesis.

Also, I am grateful to Dr. Levent Öktem for his support and ideas throughout the work.

I also would like to thank former Synplicity, Inc and Synopsys, Inc for their support during the implementation of this thesis.

I would like to thank every committee member of this thesis for reviewing the study.

Finally, special thanks to my wife for her endless support, understandings, and continual patience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

VLSI	-	Very-large-scale integration
RLC	-	Run-Length Coding
DWT	-	Discrete wavelet Transform
CWT	-	continuous wavelet transform
DTWT	-	discrete time wavelet transform
QMF	-	quadrature mirror filter
FIR	-	finite impulse response
DCT	-	Discrete Cosine Transform
FPGA	-	field-programmable gate array
ASIC	-	application-specific integrated circuit
RCT	-	Reversible Color Transform
ICT	-	Irreversible Color Transform
BPC	-	Bit-Plane Coding
EBCOT	-	Embedded Block Coding with Optimal Truncation
SPP	-	Significance Propagation Pass
MRP	-	Magnitude Refinement Pass
CUP	-	Cleanup Pass
ZC	-	Zero Coding
SC	-	Sign Coding
MRC	-	Magnitude Refinement Coding
RAM	-	Random Access Memory
Hz	-	Hertz
FDRE	-	D Flip-Flop with Clock Enable and Synchronous Reset

CHAPTER 1

INTRODUCTION

Image compression is an application of data compression. Its aim is to reduce the redundancy in an image data and try to achieve efficiency in storing or transmitting images by representing the image in a size closer to its entropy.

In general, image compression can be classified into two general categories, which are lossy compression and lossless compression. Lossy compression is a good choice if the source image is natural image or a photo, where a minor loss in fidelity to the original image in order to achieve better compression ratios, is acceptable. Lossless compression is a good choice where any information in the image is of the great importance, since lossy compression methods may introduce artifacts due to the compression. Examples of this kind image are medical images or technical drawings, etc.

There exists many methods for both lossy and lossless image compression and they are selected for the applications they are best suited in. Some of the lossy compression methods are Chroma subsampling, transform based coding and fractal compression [1]. Besides the entropy coding algorithms such as Huffman coding and Arithmetic Coding, run length coding and dictionary based algorithms can also be listed as lossless compression methods.

Figures like compression performance and complexity take important part in selection of a particular compression method. Another concern for selecting a compression method is the royalties that selected compression method may or may not have.

JPEG200 is one of the existing image compression standards. JPEG2000 is based on wavelet transform and it uses a modified form of arithmetic coder, that is MQ-Coder [2], as its entropy coder.

JPEG2000 offers much flexibility over commonly known image compression methods such as JPEG. These flexibilities include;

- The maximum number of components (color channels, e.g.) possible to use within the image,
- Maximum image size that can be coded,
- Ability to describe truncating points during decoding process,
- Ability to select resolution at many levels during decoding.

Image compression methods can be run on variety of platforms such as generic-PCs, embedded systems with or without an operating system. In some computationally demanding applications such as JPEG2000 image compression algorithm, designers may choose to off-load the computationally intensive part. For example, in a digital camera, there can be a general purpose CPU that manages the overall operations of the device and a separate IC that compresses the image.

In this thesis, HDL implementation of the essential parts of the lossless JPEG2000 image compression system is presented. The implementation is based on the definitions of the JPEG2000 Image Coding System, Part1, Final Committee Draft [2].

The organization of this thesis is follows; in Chapter 2, we present a limited survey of source coding algorithms related to image compression standards. In Chapter 3, we study discrete wavelet transform and its efficient implementation by means of *lifting*. Chapter 4 discusses the parts of the JPEG2000 image coding system in detail. In Chapter 5, we discuss the VLSI implementation of the parts of the JPEG2000 image coding system. Finally, we present and discuss the results of the VLSI implementation in Chapter 6. Conclude of the thesis is presented in Chapter 7 by proposing further enhancements on the implementation presented and pointing out the shortcomings of the implementation.

CHAPTER 2

SOURCE CODING ALGORITHMS

This chapter reviews some of the source coding algorithms used in image compression. Some of the source coding algorithms in the literature include Huffman coding, arithmetic coding and its variations, Ziv-Lempel coding based algorithms. In this chapter, we review Run Length Coding, Huffman coding, Arithmetic coding and Binary Arithmetic coding.

Selection of a particular algorithm in an application usually depends on the characteristics of the data to be coded. Each one of them has its own advantages and disadvantages, in terms of complexity, performance and suitability in applications.

In lossy image compression methods, source coding algorithms are used after the transformation and the quantization part. In lossless image compression methods, that may or may not have transformation stage, images are compressed using source coding algorithms.

2.1 Run Length Coding

Pixels in a typical image can be highly correlated. In a smooth region of an image, neighboring pixels may have identical values or change in value can be minimal. For binary images, pixel values can be observed as consecutive runs of 1s or 0s in a neighborhood. In a grayscale or a color image, pixel value may not be identical but change in values can be slowly varying, thus minimal value change may take place.

For example a data sequence is given below to simply demonstrate how RLC works.

D = [WWWWWWBWWWWWWWWWBBWWWWWWWWWWWWBWW] where W represents white and B represents black in a binary image.

The data sequence consists of 30 Ws and 4Bs, summing up to 34 bits of data. When RLC is applied to the data sequence, result would be 6W1B9W2B13W1B2W.

Depending on the number of occurrences (6, 1, 9 e.g.), total number of bits in the encoded bit stream would be

$$\text{Occurrence coding width } X\ 7 \tag{2.1}$$

For the given example, maximum number of occurrence is 13, which can be coded with 4 bits. Replacing this value in Eq. 1.1, total number of encoded bits would be 4 X 7 = 28 bits, meaning 6 less bits out of 34 bits of data.

In some cases such as gray scale or color images, runs of similar pixels may not be apparent. In these situations, the data can be preprocessed before applying RLC to the source data.

For example, in the data given below, the correlation between data may not be apparent at first and there exists no runs of identical values.

D = [52 58 64 70 76 82 88 100 112 124 132 156 176 196 216 236 232 228 224 220 216 212 208 204 200 196 192]

However if we find the difference between two consecutive data pairs, there would be identical runs. We may define the difference run as

$$e(i) = d(i) - d(i-1) \tag{2.2}$$

Applying Eq. 2.2 to the above data sequence, the result of difference run would be

e = [6 6 6 6 6 6 12 12 12 8 24 20 20 20 20 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4]

After preprocessing, input data shows more consecutively redundant data, thus it is more appropriate for RLC.

2.2 Huffman Coding

Huffman coding is a technique for generating the shortest code length for a given source symbol set [3]. Basically, Huffman coding maps the source data symbols into new set of symbols, which are calculated in Huffman coding process.

Huffman coding is based on two observations;

- More frequent symbols can be coded with shorter code words than the less frequent in symbols in the source data set.
- Two least frequent symbols of the source symbol set will have code words of the same size, but, differing only in the least significant bits.

Let's assume that we have a source symbol set $\{s_1, s_2 \ldots s_m\}$ with associated probability values $\{p_1, p_2 \ldots p_m\}$ for each symbol. The Huffman codes are mapped into a binary tree, which is also called a *Huffman Tree*.

Huffman coding technique is described in the steps below.

1. A set of nodes is generated $N = [N_1, N_2, \ldots, N_m]$ and each not is associated with a source symbol (s_1, s_2, etc.). Each node is labeled with the corresponding probability value (p_1, p_2, etc.).
2. A new node is generated and the parent nodes of this node will be the two least probable symbols in the current node set.
3. Label of the newly generated node will be the sums of the probability values of its parent nodes.
4. One of the branches of the newly generated node will be labeled as 1, and the other one as 0.
5. Node set is updated by replacing two least probable symbols with newly generated parent node. If the number of remaining nodes is more than 1, step 2 is executed.
6. Binary tree is traversed from the root to the leaf nodes to produce the code word for the corresponding symbol [3].

In theory, Huffman coding offers a coding length close to the entropy of the source data. However, in practice, this may not be the case. Huffman coding has some limitations.

First of all, Huffman coding requires that statistics of the source data is known to both the encoder and the decoder. Along with the encoded data stream, probability table for the source data should also be sent to the decoder. If the data set is not sufficiently large, overhead of adding probability tables can be significant. If the statistics are not also forwarded to the decoder, a static probability table can be used

to both encode and decode the source data. However, Huffman coding will not reach its in-theory effectiveness since a generalized probability table may not be close to the ideal values for every kind of source.

Another limitation of Huffman Coding is that Huffman Coding reaches its maximum effectiveness if the source probability values are exactly negative powers of 2 (2^{-1}, 2^{-2}, e.g.). If not, encoder can achieve poor coding efficiency [4].

One another limitation is that, if the probability of a particular symbol is too small then the code generated for this particular symbol may be very large to fit into a basic storage unit or word of the computing platform. This limitation actually depends on the platform Huffman coding is running on.

2.3 Arithmetic Coding

Arithmetic coding is a variable-length source coding algorithm. In Huffman coding, each source code symbol is substituted with another symbol as the result of Huffman coding. Arithmetic coding does not use this approach. In arithmetic coding a number of input symbols are represented by a real number (0.0 ->1.0).

Arithmetic coding works by defining a range and then, dividing this range according to the probability values of the source alphabet. Range is updated in the process according to the next symbol in the source data. If a sufficiently large source data set is encoded, arithmetic coding can reach to the Shannon's entropy limit given the statistics about the data set is accurate.

Moreover, arithmetic coding is more efficient when the source data set has a limited alphabet. For example, in case of a bi-level alphabet, Huffman coding even can not achieve any compression.

2.3.1 Encoding Algorithm of Arithmetic coding

Consider a four symbol alphabet A = {a, b, c, d}, with fixed number of probabilities p (a) = 0.2, p (b) = 0.3, p (c) = 0.1 and p (d) = 0.4. Probabilities, cumulative probabilities and initial ranges of the sample alphabet are given in Table 1.

Table 1 Sample Probability Model

Index	Symbol	Probability	Cumulative Prob.	Range
1	a	0.2	0.2	[0.0, 0.2)
2	b	0.3	0.5	[0.2, 0.5)
3	c	0.1	0.6	[0.5, 0.6)
4	d	0.4	1.0	[0.6, 1.0)

Assume we would like to encode *"badcac"* using the probity estimates tabulated in the sample probability model given in Table 1.

In Figure 1, vertical bars represent the range, which is in turn divided to sub-ranges according to the probability model of the source data.

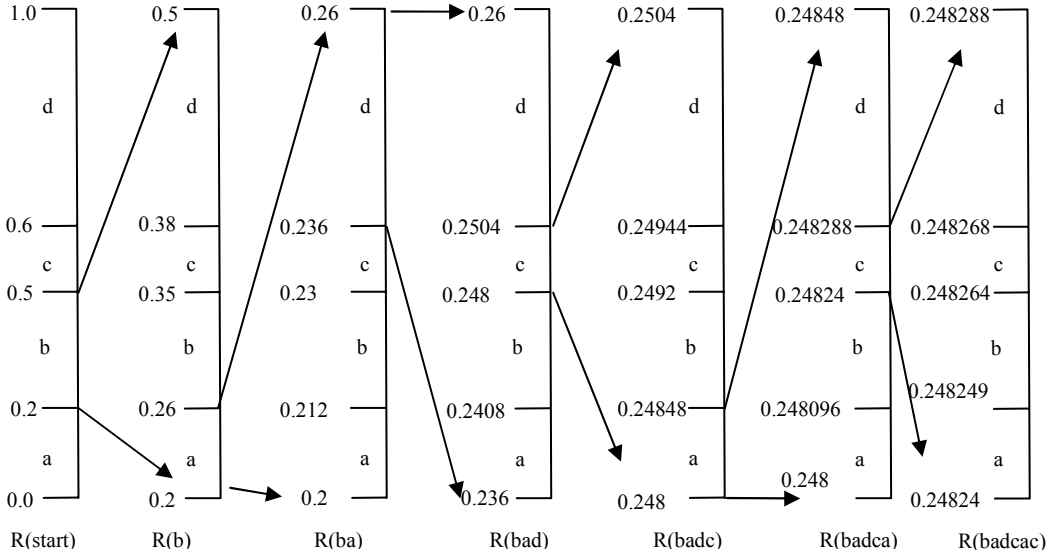

Figure 1 Arithmetic Coding Algorithm

Range, denoted as R, is [0.0, 1.0) when starting encoding and decoding. As the symbols in source data gets processed, range gets narrower.

First symbol in the source data is *b*, which falls into to the sub-range [0.2, 0.5) in the starting range. Thus, range is narrowed down to [0.2, 0.5). Our new range, R(b), is partitioned according to the probability model again. Next symbol in the source is *a* and our new range becomes [0.2, 0.26). Again the range is partitioned. Next symbol is *d* and this symbol falls into the range, [0.236, 0.26), which becomes our new range R (bad). Carrying on these operations till the last symbol as given in Figure 1, we get

the range R(badcac) at the end of the operations. At the end of the operation, we have the half-open interval [0.24824, 0.248288) which is denoted by R(badcac). The decoder doesn't need to know the end point of the last range. Selecting any number between the final range and sending this data to the decoder will be sufficient enough to reconstruct the input message *"badcac"*. One important note here is that, decoder needs to know probability model of the sample data. For the sake of completeness of the example, 0.24825 will be used as our encoder output.

2.3.2 Decoding Algorithm of Arithmetic coding

As pointed out in the encoding algorithm of arithmetic encoder, decoder has to know the probability model of the encoder and start with the range [0.0, 1.0). This range will be partitioned into sub-ranges according to the probability model. Encoded signal given the encoder example in Section 2.3.1, 0.24825, falls into the range [0.2, 0.5) and encoder immediately understands that the first symbol is *b*. Then the range is narrowed down and the next range is [0.2, 0.5). The encoded number 0.24825 falls into the range [0.2, 0.26) and the decoder generates the second symbol *a*. Range is partitioned again and 0.24825 falls into the range [0.236, 0.26) and decoder generates the symbol *d*. New range is [0.236, 0.26) and 0.24825 falls into the range [0.248, 0.24944) and the decoder generates the symbol *c*. Range is partitioned again and 0.24825 falls into the range [0.248, 0.24848) and the decoder generates the symbol *a*. Range is partitioned again. Our encoded data falls into the range [0.24824, 0.248288) and the decoder generates the symbol *c*.

At this point decoder successfully decoded the input symbols using only the encoded data, 0.24825, and probability model of the source data. However, if decoder is not stopped by some means; it may continue to decode the data.

One method to solve the termination problem is that encoder may we be aware of the size of the code block, which is six for this example and encoder can stop when termination number is reached.

Another method is that, one of the symbols can be the terminator signal. In the given example, none of the symbols can be used as a terminator symbol since all the symbols are repeated more than once in the code block. However, another symbol, *t*

can be appended at the end of the source data and the decoder can be aware of the meaning of this symbol and terminate the decoding operation when terminating symbol is decoded.

2.3.3 Disadvantages of Arithmetic Coding

Beside its advantages, arithmetic encoding may not be the perfect solution for every application. For example, Huffman coding is more error resilient than arithmetic coding. In a Huffman coded bitstream, if a single bit gets corrupted, only one symbol will be corrupted in the original data set. However, in arithmetic coding, if a single bit is corrupted, depending on the length of the data coded, more than one source symbol can be incorrectly decoded.

Another disadvantage of arithmetic coding is its computational complexity. Arithmetic encoding requires one or more multipliers, depending on the implementation, in order to compute new interval values. Since multiplication is a costly operation in terms of hardware and software implementations, arithmetic coding puts on more load on the computation platform.

However, a multiplication free algorithm is also available in literature. MQ-Coder [5], a special form of arithmetic encoder, that does not use multipliers and uses integer arithmetic instead of real numbers to operate, is used in JPEG2000.

Other limitations of arithmetic coding are listed below.

- Since any value between the final interval can be used as the encoded data, encoded message is not unique.
- Encoder does not produce any output until the last data in the code block is processed. This has its own effect on the decoder.
- As the size of the code block gets larger, precision required to represent the final data grows. As we have described in the definition of the arithmetic encoder, the result is a real number and the precision of the final value is limited to the precision of the platform that encoder runs.
- Arithmetic coding extensively uses multiplication operation to compute the new intervals during both encoding and decoding.

- Unlike Huffman coding, arithmetic coding is very error prone to transmission errors. Even if one bit of the final results gets corrupted, entire code block coded with arithmetic coding may get corrupted.

2.4 Binary Arithmetic Coding

As described in Section 2.3, arithmetic coding has limitations. To overcome these limitations, further enhancements are added. One of them is binary arithmetic coding (BAC).

Binary arithmetic coding is an incremental coding process. Thus the problem that encoder does not produce any output during encoding until all the symbols are processed is solved. Encoder produces an output just after it encodes the first signal.

As explained in the arithmetic coding section, if the source alphabet is large and the number symbols in a code block is also large, final result may be too small to be represented effectively in a digital system. Incremental coding property of binary arithmetic coding also solves the problem of the output precision limitation, since it does not rely on the precision of the real values on a particular platform that coding system works on.

BAC also uses range calculations and range updates during encoding and decoding process. However, update protocol of the BAC is slightly different.

BAC has four cases for updating the range.

- Case 1: Current range [L, H) entirely falls into to lower half of the interval ($H<0.5$), encoder generates a binary bit 0 and rescales the range by a factor 2 [2L, 2H).
- Case 2: Current range [L,H) entirely falls into the upper half of the interval ($L\geq0.5$), encoder generates binary bit 1 and rescales the range as [2(L-0.5),2(H-0.5))
- Case 3: Current range falls into the range ($L\geq0.25$ and $H< 0.75$), a special counter SPCL_COUNT is incremented and the range is rescaled as [2*(L-0.25), 2*(H-0.25)). If Case 1 or the Case 2 occurs, encoder checks the value that SPCL_COUNT holds. If SPCL_COUNT is greater than 0, encoder

generates the value coming from Case 1 or Case 2. Encoder generates binary 1s or 0s (the value generated by Case 1 or Case 2) as many as the value of SPCL_COUNT.

- For the other cases, encoder doesn't generate any output. This property leads to low bit rate of the output data.

In binary arithmetic coding, we can replace the range [0.0, 1.0) by [0, MAX_VALUE), where max value is an integer value. This integer number is determined according to the platform that encoder/decoder is being run. For example, MQ coder, which is selected by JPEG2000 standard, uses 32 bit registers to keep track of the interval. This gives a MAX_VALUE of 2^{32} -1, which is 4.294.967.295.

Moreover, instead of using cumulative probabilities for partitioning the ranges, frequencies of the alphabet members scaled up within the range need to be used.

Cumulative probabilities can be replaced with cumulative frequencies using Eq. 2.3.

$$CumFreq(i) = \sum_{k=1}^{i} N_k \tag{2.3}$$

where N_k is the number of times a symbol occurs in the source data of a length TC. Then, cumulative probability can be defined as

$$F(i) = \frac{\sum_{k=1}^{i} N_k}{TC} = \frac{CumFreq(i)}{TC} \tag{2.4}$$

When updating ranges, Eq. 2.5 can be used.

$$\begin{aligned} L &= L + \frac{(H - L + 1) * CumFreq(i-1)}{TC} \\ H &= L + \frac{(H - L + 1) * CumFreq(i)}{TC} - 1 \end{aligned} \tag{2.5}$$

As given in the description of binary arithmetic coder, ranges often need to be compared. In this cases, if $L \geq \frac{MAX_{VALUE}+1}{2}$, the range [L, H) falls into the upper half of the interval [0, MAX_VALUE).

JPEG2000 uses a variation of binary arithmetic coder called MQ coder which uses context values and decision bits to compress data [2]. MQ coder is also called context-based adaptive arithmetic coder.

CHAPTER 3

DISCRETE WAVELET TRANSFORM

Fourier analysis represents signals in terms of infinite numbered sinusoidal functions and its harmonics. Fourier transform is also known to be very effective in analysis of time-invariant (stationary) signals [4].

However, Fourier transform omits a parameter of great importance in the source signal; time information. For a stationary signal, time information may not hold a great deal of importance. On the contrary, as in the field of image processing, a signal may accompany non-stationary and/or transitory characteristics.

One way to overcome this shortcoming of Fourier transform is a technique called windowing. In 1946, Nobel laureate Dennis Gabor, adapted the Fourier transform to transform only a small portion of the source signal at time. Gabor's adaptation is called Short Time Fourier Transform (STFT). STFT generates a function of two variables, time and frequency. Precision of the generated output is a function of the transform window size [6].

Wavelet transform is the next step after windowed Fourier transform. Basically, wavelet transform uses variable sized windows. If the output intended to be more precise in the low frequency spectrum, windows of long intervals are used and vice versa for high frequency components.

As explained above, Fourier transform uses sinusoidal functions and harmonics. Wavelet transform, on the other hand, uses a different form of basis function, called to be the *mother wavelet*. Wavelet transform uses translations and scaling of mother wavelet to represent an arbitrary signal.

Not only the energy of wavelets is concentrated in time, but also they still have wave-like properties. These two properties allow wavelet transform to be an efficient tool for analyzing time-variant and transient signals.

Mallat proposed a theory for analyzing signals called multiresolution decomposition [7]. Multiresolution decomposition uses wavelets and represents them using Pyramid algorithm which is discussed in upcoming sections.

Assuming the mother wavelet, also called to be *prototype*, is defined as $\psi(t)$, other wavelets $\psi_{a,b}(t)$ to be used with wavelet transform can be defined as

$$\psi_{a,b}(t) = \frac{1}{\sqrt{|a|}}\psi\left(\frac{t-b}{a}\right) \tag{3.1}$$

where a and b represent dilations and translations in time domain, respectively. Similarly, the variables a and b also represent scaling and shifts in frequency domain respectively. Both a and b are defined as real numbers. In terms of $\psi_{a,b}(t)$, mother wavelet can be also represented as $\psi_{1,0}(t)$.

Using the definition of a wavelet in Eq. 3.1, wavelet transform of $f(t)$ can be defined as

$$W(a,b) = \int_{-\infty}^{+\infty} \psi_{a,b}(t) f(t) dt \tag{3.2}$$

and the inverse of the transform to find the $f(t)$ from $W(a,b)$ can be defined as

$$f(t) = \frac{1}{C}\int_{a=-\infty}^{+\infty}\int_{b=-\infty}^{+\infty} \frac{1}{|a|^2} W(a,b)\psi_{a,b}(t) da db \tag{3.3}$$

where

$$C = \int_{-\infty}^{+\infty} \frac{|\Psi(\omega)|^2}{|\omega|} d\omega \tag{3.4}$$

$\Psi(\omega)$ is defined as the Fourier transform of the mother wavelet $\psi_{1,0}(t)$.

If $f(t)$, a and b are said to be continuous function and variables, $W(a,b)$ is called as the continuous wavelet transform (CWT) of $f(t)$.

3.1 Discrete Wavelet Transform

Many real life applications of wavelet transform require discrete counterpart of wavelet transform. In order to achieve this, there is a need for discretizing the continuous variables. The continuous variables a and b can be discretized by defining

$$a = a_0^m \tag{3.5}$$

and

$$b = nb_0a_0^m \tag{3.6}$$

where m and n are integer valued variables. If a and b are substituted in Eq. 3.1, we get

$$\psi_{m,n}(t) = a_0^{-\frac{m}{2}}\psi\left(\frac{t - nb_0a_0^m}{a_0^m}\right) \tag{3.7}$$

and $\psi_{m,n}(t)$ can be used to represent the discrete wavelets of the mother wavelet $\psi(t)$.

As there are many choices for the values of a_0 and b_0, one particular couple of values is very important among them. If a_0 and b_0 are selected as 2 and 1, which corresponds to $a = 2^m$ and $b = n2^m$, this way of sampling is called as *dyadic sampling* and corresponding decompositions of the input signal is called as *dyadic decomposition*. In other words, *dyadic sampling* means that sampling intervals differ by a factor of two.

Using the values above for a and b and substituting into Eq. 3.7, we get a family of orthonormal basis functions as given in Eq. 3.8.

$$\psi_{m,n}(t) = 2^{-\frac{m}{2}}\psi\left(\frac{t}{2^m} - n\right) \tag{3.8}$$

When wavelet transform is applied to a signal $f(t)$, the output of the transform is called as the *wavelet coefficients*. Wavelet coefficients can be defined as

$$c_{m,n}(f) = a_0^{-\frac{m}{2}}\int f(t)\psi(a_0^{-m}t - nb_0)dt \tag{3.9}$$

For dyadic decomposition, we replace the values of a_0 and b_0, then we get

$$c_{m,n}(f) = 2^{-\frac{m}{2}} \int f(t)\psi(2^{-m}t - n)dt \tag{3.10}$$

The original signal $f(t)$ can be reconstructed using the wavelet coefficients generated using Eq. 3.9 in general or using Eq. 3.10 for dyadic decomposition.

So far, in all equations $f(t)$ is assumed to be continuous. This form of transform is called as *discrete time wavelet transform* (DTWT). However, $f(t)$ needs to be discrete if $f(t)$ is to be processed with a digital computer. In case of both $f(t)$ and variables a and b are discrete, transform is called as *discrete wavelet transform* (DWT) [7].

3.2 Multiresolution Analysis of Signals

The main idea of multiresolution analysis is to approximate a function $f(t)$ at different resolution levels. Multiresolution analysis of signals takes two special functions into account. The *mother wavelet* $\psi(t)$, as discussed in Section 3.1, and a *scaling function* $\phi(t)$. Scaling function $\phi(t)$ may have dilated and translated versions. These versions can be defined as

$$\phi_{m,n}(t) = 2^{-\frac{m}{2}}\phi\left(\frac{t}{2^m} - n\right) \tag{3.11}$$

For a fixed value of m, the set of scaling functions generated by Eq. 3.11 are orthonormal [4].

Thus, using the linear combination of scaling and its translation function $\phi(t)$, a set of functions can be generated using Eq. 3.12.

$$f(t) = \sum_n \alpha_n \phi_{m,n}(t) \tag{3.12}$$

3.3 Implementation by filters and Pyramid Algorithm

Multiresolution analysis decomposes an input signal into two parts. One of them is the approximation of the input signal. This signal contains finer to coarser resolution. The other signal is the details information lost during the approximation.

It is proposed that wavelet decomposition of signals can be realized using quadrature mirror filter (QMF) pairs which constitute a pyramidal filter structure where filters are FIR filters [7].

For a signal $f(t)$, wavelet coefficients can be expressed as

$$c_{m,n}(f) = \sum_k g_{2n-k} a_{m-1,k}(f)$$
$$a_{m,n}(f) = \sum_k h_{2n-k} a_{m-1,k}(f) \tag{3.13}$$

where g and h are the high pass and low pass filters, respectively. Eq. 3.13 is recursive and is used to compute wavelet coefficients at different levels. Eq. 3.13 is also called Mallat's Pyramid algorithm [7].

Many orthonormal basis functions are infinitely supported, meaning filter pairs can have infinitely long taps. However, this application would not be practical in real life applications. Longer filter lengths degrades the performance of real life applications.

Shorter tap-length filters can be achieved by relaxing orthonormality and using biorthogonal basis functions. However, in this case, filters used in forward transform (analysis filters) and inverse transform (synthesis filters) can be different. Also, for perfect reconstruction, which is a requirement for many applications such as lossless image compression in JPEG2000, synthesis filters and analysis filters need to satisfy

$$g'_n = (-1)^n h_{-n+1}$$
$$g_n = (-1)^n h'_{-n+1} \tag{3.14}$$
$$\sum_n h_n h'_{n+2k} = \delta_{k,0}$$

where (h', g') are the synthesis filter coefficients and (h, g) are analysis filter coefficients [8], respectively.

Also, if $(h', g') = (h, g)$, wavelet filters are called to be orthogonal. If not, they are called as biorthogonal [4].

As given [7], wavelet transform can be implemented using FIR filter pairs and multi-level transforms would constitute a pyramidal filter structure. An example filter structure is given in Figure 2.

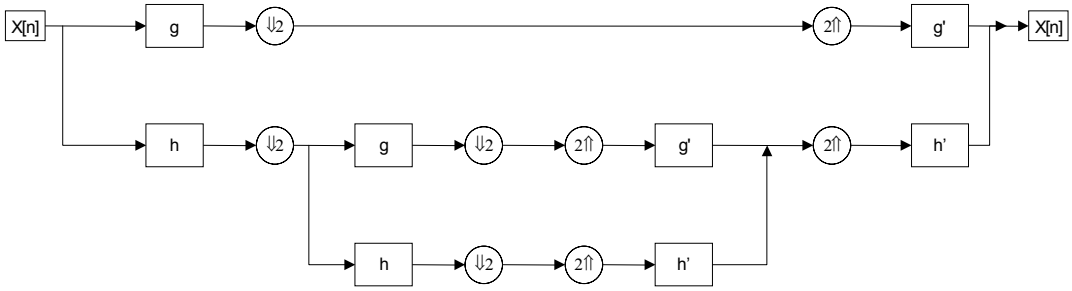

Figure 2 Pyramidal Filter Structure

subsampled by two. Output of filter h is low-pass sub-band, denoted as $y_L[n]$ and output of filter g is high-pass subband, which is denoted as $y_H[n]$, respectively. In arithmetic terms, $y_L[n]$ and $y_H[n]$ can be expressed as

$$y_L[n] = \sum_{i=0}^{\tau_L - 1} h(i)x(2n - i) \tag{3.15}$$

$$y_H[n] = \sum_{i=0}^{\tau_H - 1} g(i)x(2n - i) \tag{3.16}$$

where $\tau_H$and $\tau_L$are the lengths of the high-pass and low-pass filters, respectively. If a further level of transform is to be applied, same procedure is again carried out. In Figure 2, two level of decomposition is applied.

During inverse transform, decomposed signals are filtered again, but using h', g' as filters, however, inputs of these filters are upsampled by inserting zeroes before being filtered. After filtering, outputs of the filters are added together to construct the input signal at a certain level of transformation.

3.4 Lifting Implementation of Wavelet Transform

Wavelet transforms are basically implemented as FIR filters. However, DWT via FIR filtering may not be desirable in terms of performance requirements. In JPEG,

images are transformed using fast implementation of Discrete Cosine Transform (DCT) on a block basis and each block is 8x8 in size. However, in JPEG2000, images are transformed in tile basis, where tile sizes are much larger then JPEG. For instance, tile sizes are selected as 256x256 pixels in the implementation of this thesis. As the tile sizes get larger, memory requirements gets higher.

FIR filters are generally implemented using shift registers, multipliers and adders. In hardware applications, such as FPGAs, multipliers and memories are precious resources. Besides, multiplication is costly operation on many platforms.

Fortunately, a technique called lifting based wavelet transform or simply called *lifting*, helps to solve this problem properly.

The main idea behind lifting is breaking up the filters into smaller filters, and then converting them into a sequence of upper and lower triangular matrices [9]. In order to analyze lifting, we need to establish some preliminary information used in lifting.

3.4.1 Euclidean Algorithm

Assume we have two Laurent polynomials $a(z)$ and $b(z)$. The greatest common divisor (gcd) of these two polynomials can be calculated using Euclidean algorithm.

In [4], the pseudo-code for Euclidean algorithm is given as

```
begin
    k = 0
    a_k(z) =  a(z)
    b_k(z) =  b(z)
    while b_k(z) ≠ 0
   begin
        a_{k+1} = b_k(z)
        b_{k+1} = a_k(z) % b_k(z)
        q_{k+1} = a_k(z) / b_k(z)
        k = k +1
    end
    gcd =  b_k(z)
end
```

where, $q(z)$ is the quotient of the division operation. It should be noted that, Euclidean algorithm holds if b$(z) \neq 0$ and $|a(z)| \geq |a(z)|$ where $|a(z)|$ means the degree of a Laurent polynomial.

Euclidian algorithm states that greatest common divisor of $a(z)$ and $b(z)$ is a_n, where n is the smallest integer for the condition $b_n(z) = 0$.

As given in [4], using Euclidian algorithm, we can write that

$$\begin{bmatrix} a_{i+1}(z) \\ b_{i+1}(z) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q_i(z) \end{bmatrix} \begin{bmatrix} a_i(z) \\ b_i(z) \end{bmatrix} \tag{3.17}$$

which can be rewritten as

$$\begin{bmatrix} a_i(z) \\ b_i(z) \end{bmatrix} = \begin{bmatrix} q_i(z) & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_{i+1}(z) \\ b_{i+1}(z) \end{bmatrix} \tag{3.18}$$

Eq. 3.18 can be generalized a as

$$\begin{bmatrix} a_i(z) \\ b_i(z) \end{bmatrix} = \prod_{k=1}^{n} \begin{bmatrix} q_i(z) & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_n(z) \\ 0 \end{bmatrix} \tag{3.19}$$

3.4.2 Perfect Reconstruction

When a signal is transformed to another domain (frequency domain, e.g.) as in Fourier transform, it is expected that if the inverse of the transform is applied, we would get the exact same input signal supplied to the forward transform if there exists no additional processing in the transform domain. Also, this requires that there isn't any loss because of the digital representation of the signal in the transform domain in case of discrete transforms. This is also required for DWT.

In [9], it is given that a filter bank provides perfect reconstruction if the rules in Eq. 3.20 are satisfied.

$$\begin{aligned} h'(z)h(z^{-1}) + g'(z)\, g(z^{-1}) &= 2 \\ h'(z)h(-z^{-1}) + g'(z)\, g(-z^{-1}) &= 0 \end{aligned} \tag{3.20}$$

where $h(z)$ and $g(z)$ are z-transform of the analysis filters h and g, and $h'(z)$ and $g'(z)$ are z-transform of the synthesis filters h' and g', respectively.

3.4.3 Polyphase Representation

A filter $h(z)$ can be expressed as

$$h(z) = h_e(z^2) + z^{-1}h_o(z^2) \tag{3.21}$$

where h_e is the filter with even indexed coefficients of filter $h(z)$ and h_o is the filter with odd indexed coefficients of the filter $h(z)$. It should be noted that $h_o(z^2)$ is supplied with one sample time delayed data, which can be interpreted as that signal is delayed one sample time in fast clock domain.

Using Eq. 3.21, we can say that the filter $h(z)$ can be splitted into two smaller parts, h_e and h_o. Then a *polyphase matrix* for a filter $h(z)$ can defined as

$$P(z) = \begin{bmatrix} h_e(z) & g_e(z) \\ h_o(z) & g_o(z) \end{bmatrix} \tag{3.22}$$

Let's assume that $h(z)$ is the filter used in the analysis bank, and $h'(z)$ is its complementary filter that is used in synthesis bank. Then we can also define the polyphase matrix for synthesis filter $h'(z)$ as

$$P'(z) = \begin{bmatrix} {h'}_e(z) & {g'}_e(z) \\ {h'}_o(z) & {g'}_o(z) \end{bmatrix} \tag{3.23}$$

$P'(z)$ is called as the dual of the polyphase matrix $P(z)$ [4]. In addition to Eq. 3.20, polyphase matrices $P(z)$ and $P'(z)$ are said to satisfy perfect reconstruction if they hold Eq. 3.24.

$$P'(z)P(z^{-1})^T = I \tag{3.24}$$

where I is the identity matrix of size 2x2.

Wavelet transform can be defined in terms of a polyphase matrix. For instance, analysis filter bank can be represented as

$$\begin{bmatrix} y_L(z) \\ y_H(z) \end{bmatrix} = P(z) \begin{bmatrix} x_e(z) \\ z^{-1}x_o(z) \end{bmatrix} \tag{3.25}$$

Same representation can also be defined for synthesis filter bank, which is given in Eq. 3.26.

$$\begin{bmatrix} x_e(z) \\ z^{-1}x_o(z) \end{bmatrix} = P'(z) \begin{bmatrix} y_L(z) \\ y_H(z) \end{bmatrix} \tag{3.26}$$

3.4.4 Lifting

As defined in [9], lifting has two types. One is called *primal lifting* and the other one is called *dual lifting*.

3.4.4.1 Primal Lifting

When the determinant of polyphase matrix $P(z)$ is 1, ($|P(z)| = 1, e.g.$), filter pair is called to be complementary to its inverse transform filter pair [4]. In [9], lifting theory states that a filter pair (g, h) is complementary to any other FIR filter, such as g^{new}, which is complementary to h. Then, $g^{new}$is of the form

$$g^{new}(z) = g(z) + h(z)s(z^2) \tag{3.27}$$

where $s(z^2)$ is a Laurent polynomial. The notation for the new filter is selected as $g^{new}(z)$ since the new filter is to be used for low-pass subband.

$g^{new}(z)$ can be rewritten in polyphase form, such as

$$\begin{aligned} g^{new}(z) = \{g_e(z^2) + z^{-1}g_o(z^2)\} + \\ \{h_e(z^2) + z^{-1}h_o(z^2)\}s(z^2) \end{aligned} \tag{3.28}$$

Then, polyphase matrix $P^{new}(z)$ can be written as

$$\begin{aligned} P^{new}(z) &= \begin{bmatrix} h_e(z) & g_e(z) + h_e(z)s(z) \\ h_o(z) & g_o(z) + h_o(z)s(z) \end{bmatrix} \\ P^{new}(z) &= \begin{bmatrix} h_e(z) & g_e(z) \\ h_o(z) & g_o(z) \end{bmatrix} \begin{bmatrix} 1 & s(z) \\ 0 & 1 \end{bmatrix} \end{aligned} \tag{3.29}$$

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

Let us assume $P^{new}(z)$ has a dual, $P^{new\prime}(z)$. We can apply Eq. 3.24 to Eq. 3.29, and then find Eq. 3.30.

$$P^{new\prime}(z) = P^{new}(z)\begin{bmatrix}1 & 0\\ -s(z^{-1}) & 1\end{bmatrix} \tag{3.30}$$

Thus, we can write a new low-pass filter using Eq. 3.30. Instead of polyphase form, we can write

$$h^{new\prime}(z) = h'(z) - g'(z)s(z^{-2}) \tag{3.31}$$

As seen in Eq. 3.31, low-pass subband is *lifted* by using the high-pass subband, which is called *primal lifting* [4].

3.4.4.2 Dual Lifting

In primal lifting, low-pass subband is lifted with the help of high-pass subband. In dual lifting, high-pass subband is lifted using the low-pass subband [4].

As in primal lifting, we first define the new filter as

$$h^{new\prime}(z) = h(z) - g(z)t(z^2) \tag{3.32}$$

where $t(z^2)$ is a Laurent polynomial.

Carrying out the same procedures that is done for primal lifting through equations Eq.3.28 to Eq. 3.30, we can find that

$$g^{new\prime}(z) = g'(z) - h'(z)t(z^{-2}) \tag{3.33}$$

3.4.1.3 Factorization

We can find the greatest common divisor of $h(z)$ and $g(z)$ using the Euclidean algorithm given in Eq. 3.19.

$$\begin{bmatrix} h(z) \\ g(z) \end{bmatrix} = \prod_{k=1}^{n} \begin{bmatrix} q_i(z) & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix} \tag{3.34}$$

where R is the greatest common divisor of $h(z)$ and.

In Section 3.4.4.1, we have shown that we can find a new complementary filter $g^{new}(z)$, if (h, g) is a complementary filter pair. This new filter pair can be represented in polyphase matrix form as

$$P^{new}(z) = \prod_{k=1}^{n} \begin{bmatrix} q_i(z) & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} L & 0 \\ 0 & 1/L \end{bmatrix} \tag{3.35}$$

where $q_i(z)$ is the quotient of the division of $h(z)$ by $g(z)$.

As given in [4], polyphase matrix in Eq. 3.35 can be rewritten using the properties of the Euclidean algorithm such as

$$P^{new}(z) = \prod_{k=1}^{n/2} \begin{bmatrix} 1 & q_{2i-1}(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ q_{2i}(z) & 0 \end{bmatrix} \begin{bmatrix} L & 0 \\ 0 & 1/L \end{bmatrix} \tag{3.36}$$

Lifting factorization states that we can factorize $P(z)$ into finite numbered sequence of alternating upper and lower triangular matrices [9]. Combining Eq. 3.34 to Eq. 3.36, alternating upper and lower triangular matrices can be written as

$$P(z) = \left\{ \prod_{k=1}^{l} \begin{bmatrix} 1 & s_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ t_i(z) & 1 \end{bmatrix} \begin{bmatrix} L & 0 \\ 0 & 1/L \end{bmatrix} \right\} \tag{3.37}$$

for upper triangular matrix (low-pass subband, primal lifting) and

$$P(z) = \left\{ \prod_{k=1}^{l} \begin{bmatrix} 1 & 0 \\ t_i(z) & 1 \end{bmatrix} \begin{bmatrix} 1 & s_i(z) \\ 0 & 0 \end{bmatrix} \begin{bmatrix} L & 0 \\ 0 & 1/L \end{bmatrix} \right\} \tag{3.38}$$

for lower triangular matrix (high-pass subband, dual lifting) [9]. K is a constant scaling factor applied to the subband elements after lifting operation.

Lifting steps given in Eq. 3.37 and 3.38 are also called as *update* and *predict* steps, respectively [4].

3.4.4.4 Lazy Wavelet Transform

One special form of wavelet transform is called as lazy wavelet transform. In lazy wavelet transform, $h(z) = h'(z) = g(z) = g'(z) = 1$, which means input signal is divided into even and odd parts only.

3.4.4.5 Lifting Algorithm

DWT by means of lifting algorithm can be calculated in three steps, which are

1. Applying lazy wavelet transform to input data,
2. Executing *update* and *predict* steps,
3. Applying the scaling factors (L and $1/L$).

Inverse DWT is also calculated in similar but slightly different steps, which are

1. Executing *predict* and *update* steps,
2. Upsampling by inserting 0's,
3. Merging the upsampled signals by adding them together.

Example realizations for both forward transform (analysis) and inverse transform (synthesis) are given in Figure 3 and Figure 4, respectively.

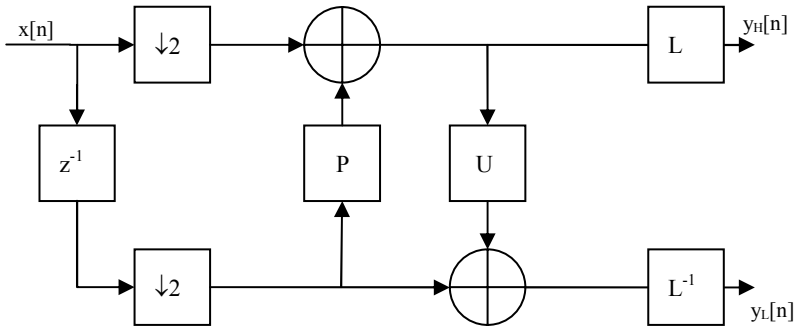

Figure 3 Lifting Realization of 1D Analysis Filter

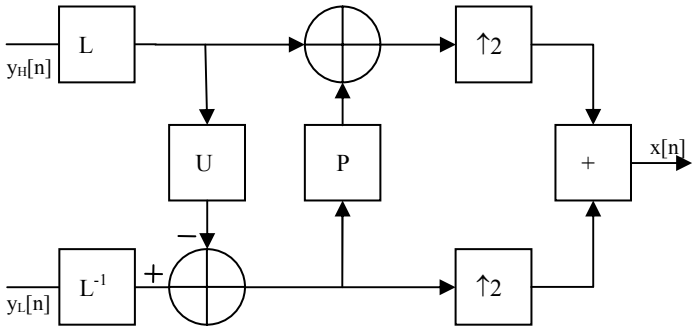

Figure 4 Lifting Realization of 1D Synthesis Filter

3.4.4.6 Lifting and Perfect Reconstruction

The Wavelet transform implemented by means of lifting is reversible if the input data is integer. If the input data is integer, it is also possible to keep the data in integer format by introducing a rounding function into the filtering operation. These types of wavelet transforms are called as integer wavelet transforms [10].

For instance, DWT used in the implementation of this thesis uses flooring function during filtering to round the integer data.

3.5 Two-Dimensional Wavelet Transform

Two-dimensional wavelet transform is calculated first by applying a one-dimensional wavelet transform row-wise, then applying the transform column-wise [4].

Let us assume we have an image of size $m\,x\,n$, where m and n are nonnegative integers. If we apply one dimensional wavelet transform to this image, results would be two images representing the output of high-pass and low-pass filters and they are denoted as H and L, respectively. These images would have a size of $m\,x\,\frac{n}{2}$ and they would look like two thinner images as shown in Figure 5.

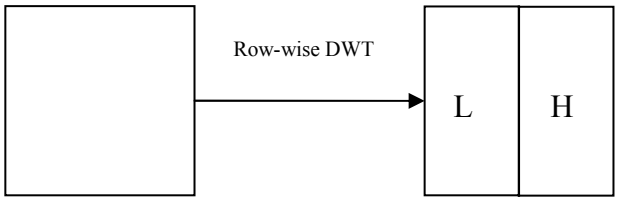

Figure 5 One-dimensional DWT

After applying one-dimensional row-wise, wavelet transform is applied again column-wise. The result would be four images of the size $\frac{m}{2}\ x\ \frac{n}{2}$ as shown in Figure 6.

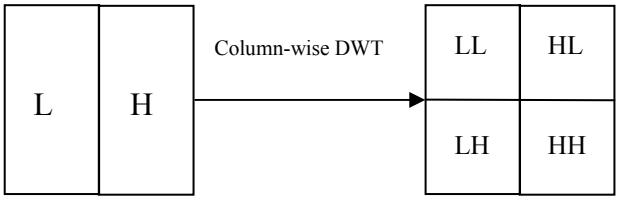

Figure 6 Two-Dimensional DWT

3.6 Advantages of Lifting

Wavelet transform can be implemented using convolution based approach, namely using FIR filters. However, lifting based approach offers many benefits over convolution based, namely FIR filter based approach.

Following items can be listed as the key advantages of lifting over filter based implementations.

- Depending on the filter length, lifting based approach offers up to 50% savings in computational operations.
- Lifting implementation is highly parallel, thus more suitable hardware based implementations are possible.
- Integer to integer transform is suitable for lossless image compression.
- Since integer to integer transform offers perfect construction, no boundary extension, which is studied in chapter 4, is required. This property alone is an important reason to go with lifting implementation.

CHAPTER 4

JPEG2000 STANDARD

JPEG image coding standard, which is considered to be successful and used more than a decade. However, technology has progresses tremendously and requirements from an image coding standard also changed significantly.

JPEG2000 image coding standard offers more than one mode to process and compress a source image. In general, JPEG2000 image coding system can be classified into two, lossless and lossy parts. Focus of this thesis is on the lossless part of the standard.

Encoding process of JPEG2000 lossless part can be briefly described as in Figure 7.

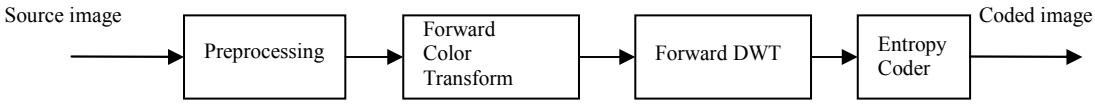

Figure 7 Encoder structure

Similarly, the decoder structure can be described as in Figure 8.

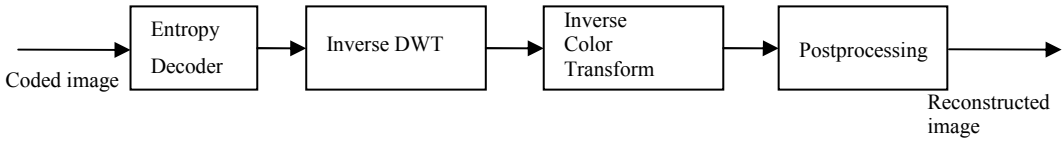

Figure 8 Decoder structure

Details of each processing block of JPEG200 image coding system are described in the following sections. Moreover, some features of JPEG2000 image coding system such as quantization, region of interest coding and rate control system are also explained, briefly. However, we would like to present a brief explanation of the advancements of JPEG2000 image coding standard over JPEG image coding standard first.

JPEG2000 is considered to be a very complex system for image compression [11] and it requires more memory and computational power than JPEG.

4.1 Advancements over JPEG

JPEG2000 outperforms JPEG in some areas. Some of these areas are listed below.

- Superior low bit rate performance of JPEG2000 is very useful when the data is transmitted via limited bandwidth communication channels.
- JPEG2000 offers a large dynamic range. JPEG2000 supports up to 38 bits per pixel dynamic range for each component. Dynamic range is limited to 8 bits in JPEG. It should be noted that standard recommends up to 16 bits dynamic range. This feature is very crucial for some applications such as medical imaging and scientific applications.
- JPEG2000 offers large image sizes. Standard defines the maximum image size as $(2^{32}-1)$ x $(2^{32}-1)$ pixels.
- Although many applications deal with just three image components (red, green, blue), some applications such as astronomical imaging and satellite imaging deals more than three image components. Besides RGB channels, images may contain multispectral components such as X-ray, infrared spectrum images. JPEG2000 supports 2^{14} image components, which makes it suitable for satellite and astronomical imaging.
- Fixed output sizes for the compressed image can be assigned in JPEG2000. For example, embedded hardware such as photocopier or an image scanner may have a limited amount of memory for a compressed image. JPEG2000 can assign a target value for the compressed image in these conditions.
- JPEG2000 codes the image by bit-planes. Thus, it is possible to achieve progressive transmission by transmitting most significant planes first, then transmitting the rest.
- Perfect reconstruction wavelet transform can be used in both lossy and lossless mode. This feature allows an image to be decoded both lossy and lossless. Thus a single unified architecture can be used to decode both lossy and lossless images.

4.2 Image Preprocessing

Image preprocessing in JPEG2000 involves three parts. These parts are tiling, DC level shifting and intercomponent transform.

4.2.1 Tiling

As described in Section 4.1, JPEG2000 support image sizes up to (2^{32}-1) x (2^{32}-1) pixels. If the image is too large to be processed as a whole, standard allows the image to be divided into equal sized, non-overlapping rectangle blocks. Each divided block is a *tile*. Tiles can have arbitrary sizes. Even all the image can be considered as one tile.

Practically, tile sizes can be determined using three parameters. These parameters are image size, number of wavelet transform levels, and the platform that the coder runs on. In VLSI implementations, on-chip memory is the main limiting factor for the tile sizes. Ultimately, tile size is capped by the underlying platform. In this thesis, tile sizes are selected as 256 x 256.

4.2.2 DC Level Shifting

Many image formats store the intensity values for image components as unsigned integers. However, standard requires that intensity values must have a dynamic range centered around zero and represented in two's complement representation. In arithmetic terms, DC level shifting can be defined as given in Eq. 4.1.

$$I'(x,y) = I(x,y) - 2^{s-1} \tag{4.1}$$

$I'(x,y)$ is the DC level shifted value of $I(x,y)$ at coordinates (x,y) and s is the precision of the image. For an 8 bit image, DC level shifting is carried out by subtracting 2^7 from each value the image.

It should be noted that if the source image already uses signed integers, DC level shifting is not required.

4.2.3 Color Space Transform

Purpose of the color space transform is to reduce the correlation amongst the components in multicomponent images, such as color images. In a typical image, there are three chroma components, red, green and blue (RGB). In each component, intensity values are given for the corresponding component.

JPEG2000 standard supports maximum 2^{14} components in a coded image [2]. However, standard itself is colorblind and does not care which component means what color. For convenience, the first three components are called red, green and blue components [4].

JPEG2000 Part 1 defines two multicomponent transforms; Reversible Color Transform (RCT) and Irreversible Color Transform (ICT). RCT can be used in both lossy and lossless implementations. However, ICT can only be used in lossy implementations.

RCT transforms images from RGB color space to YUV color space. In YUV color space, Y stands for *luma* and U and V stand for *chrominance* components. As its names suggests, RCT allows images to be perfectly reconstructed by using the inverse of the RCT.

Forward RCT is defined in [2] as

$$Y = \left\lfloor \frac{R + 2G + B}{4} \right\rfloor$$

$$U = B - G \tag{4.2}$$

$$V = R - G$$

where R, G and B represent the pixel values of red, green and blue components, respectively. $\lfloor \lambda \rfloor$ represents flooring operation on λ.

Inverse of the transform is defined as

$$G = Y - \left\lfloor \frac{U + V}{4} \right\rfloor \tag{4.3}$$

$$R = V + G$$

$$B = U + G$$

where Y, U and V represents the luma and chrominance components generated by forward RCT, respectively.

4.3 Discrete Wavelet Transform

In JPEG2000, wavelet transform step is composed of two parts;

- Discrete Wavelet Transform
- Quantization

In DWT part, input image is decomposed into several sub-bands at different resolution levels. In quantization step, wavelet coefficients are quantized against a quantization parameter. Quantization step is only applied in lossy compression.

JPEG2000 standard uses two dimensional wavelet transform to decompose input images into sub-bands at different resolution levels. As described in Section 3.5, one level of wavelet transform generates subbands LL1, LH1, HL1 and HH1, where suffixed number represents the transform level. Subband LL1 can also be further decomposed, producing subbands LL2, LH2, HL2 and HH2. Further decomposition for LL2 is also possible as shown in Figure 9.

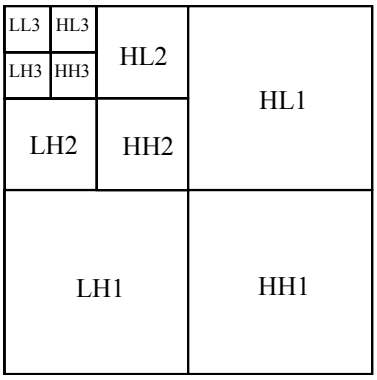

Figure 9 Three-level Wavelet Decomposition

JPEG2000 standard supports maximum 32 levels of decomposition [2]. However, in practice, there is not much gain in compression ratios after four or five levels of decomposition [4].

JPEG2000 standard defines two wavelet filters for decomposing images and generating the wavelet coefficients [2]. They are;

- Daubechies (9, 7) biorthogonal spline filter [8],
- Le Gall (5, 3) spline filter [12].

Daubechies (9, 7) filter is only used in lossy coding of images. However, Le Gall (5, 3) filter can be used in both lossy and lossless coding of images.

The numbers in parentheses are the tap lengths of low-pass and high pass filters, respectively. For example, Le Gall (5, 3) wavelet filter has a five taps low-pass filter and a 3 taps high-pass filter. Low-pass and high-pass filter coefficients of Le Gall (5, 3) wavelet filter are given in Table 2 and Table 3.

Table 2 Le Gall (5, 3) Low-pass Filter Coefficients

Tap	**Coefficient**
h^{-2}	-1/8
h^{-1}	1/4
h^{0}	3/4
h^{1}	1/4
h^{2}	-1/8

Table 3 Le Gall (5, 3) High-pass Filter Coefficients

Tap	**Coefficient**
g^{-1}	-1/2
g^{0}	1
g^{1}	-1/2

In polyphase matrix representation, Le Gall (5, 3) wavelet filter can be expressed as in Eq. 4.4

$$P(z) = \begin{bmatrix} 1 & \frac{1}{4}(1+z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \frac{-1}{2}(1+z^{-1}) & 1 \end{bmatrix} \quad (4.4)$$

4.4 Quantization

After generation of wavelet coefficients in discrete wavelet transform part, coefficients can be quantized to increase the compression ratio. In fact, quantization of wavelet coefficients is one of the major reasons for loss in fidelity.

JPEG2000 uses a uniform scalar quantizer to quantize the wavelet coefficients. Each subband can have different quantization step. Quantizer is defined in the standard [2] as

$$q_b(u,v) = sign\big(a_b(u,v)\big)\left(\frac{a_b(u,v)}{\Delta_b}\right) \quad (4.5)$$

where Δ_b is the step size for the subband b,$a_b(u,v)$ is the wavelet coefficient and $q_b(u,v)$ is the quantized wavelet coefficient.

Quantization step is only part of lossy part of JPEG2000 image coding standard. Since the focus of this thesis is on the lossless part, we would not implement this part of the standard.

4.5 Region of Interest Coding

One of the unique features is the support for Region of Interest (ROI) coding. As described in Section 4.2.2.1, Le Gall wavelet filter can be used in both lossy and lossless image coding. Using ROI coding, a region in the source image can be defined as the interested region, and this region can be given a high priority in terms of fidelity to the original image.

JPEG2000 uses a method called MAXSHIFT for ROI coding [2]. Since we do not implement ROI coding in this thesis, we will not go into the details of MAXSHIFT implementation in JPEG2000 standard.

4.6 Rate Control

JPEG2000 offers a feature for determining the target size, thus bit-rate for the encoded image. One method for rate control is to apply post compression rate-

distortion algorithm [13] after compressing the whole image. One another method is to change the quantization step size Δ_b. This method requires Tier 1 coder, which is most the computationally intensive part of the standard, to run again for the new quantization step size.

4.7 Entropy Coding

Entropy coding in JPEG2000 consists of two parts; fractional bit-plane coding (BPC) and binary arithmetic coding. JPEG2000 adopts EBCOT fractional bit-plane algorithm [13]. For BAC, JPEG2000 uses MQ-Coder given in [5].

Basically, EBCOT decomposes the image into its bit planes and generate intermediate data for the MQ-Coder. These intermediate data are *context* information and *decision* bits. Context information can take 19 different values and decision bit can be either 1 or 0. MQ-Coder takes context and decision bits and generates the compressed data.

In the following sections, we present detailed information about EBCOT and MQ-Coder algorithms.

4.7.1 EBCOT

EBCOT stands for *Embedded Block Coding with Optimized Truncation of the embedded bit-streams* [13]. EBCOT algorithm decomposes codeblocks into bit planes and applies the algorithm on these bit planes in three passes. These passes are;

Significance Propagation Pass (SPP): In this pass, bit positions that have a magnitude of 1 first time are coded.

Magnitude Refinement Pass (MRP): In this pass, bit positions that have not been coded in SPP and have had magnitude of 1 in the previous bit planes are coded.

Cleanup Pass (CUP): Bit positions that have not been coded in the previous passes are coded in Cleanup Pass.

In each pass, several operations are carried out. These operations are described in detail in Section 4.6.1.2. However, before going into the details of the algorithm, some terms are needed to be described.

4.7.1.1 Terms Used in EBCOT

- **Code-Block (*y*):** A codeblock is a two dimensional array that holds wavelet coefficients.
- **Sign Array (χ):** χ is two-dimensional array of the same size of the code-block. It holds the sign information of the wavelet coefficients. Values in χ are 1 if the corresponding wavelet coefficient is negative, is 0 otherwise.
- **Magnitude Array (ν):** ν holds absolute value of the wavelet coefficients. Since EBCOT treats the coefficient values in bit-planes, another notation, ν^P is also used throughout the algorithm. In this notation, ν^P means the value of ν at bit-plane P.
- **Scan Pattern:** EBCOT in JPEG2000 standard uses a special scan mode to scan the wavelet coefficients in a code-block. This mode is called **vertical causal mode**. In vertical causal mode, coefficients are read from the code-block by 4 x N parts. where N is the length of the dimension of a code block. These parts are also called *stripes*. For example, a code block of 16 x 16 elements would be read in 4 x 16 parts, totaling in an eight stripe structure. In each stripe, elements are read from columns one by one. Reading order of the coefficients in vertical causal mode is shown in Figure 10.

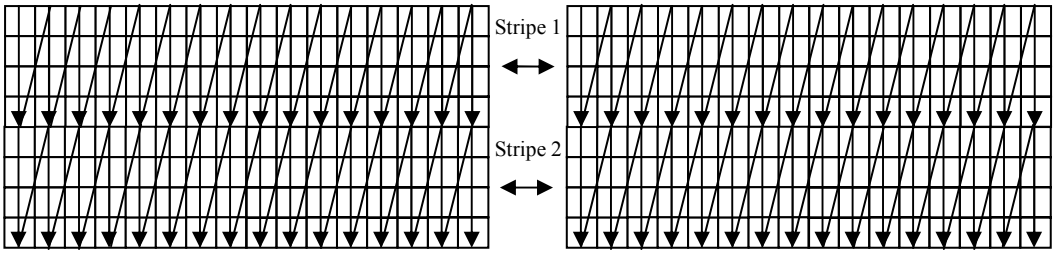

Figure 10 Vertical Causal Mode Scan Pattern

State Variables σ, σ' and η: State variables are two-dimensional arrays of the same size of the codeblock. Their values can either be 0 or 1 and initially they are

set to zero. Values of σ, σ` cannot be changed back to zero until all the code block is coded. However, η is reset after each bit plane is coded. Values of the state variables can be interpreted as;

- If σ[m,n] is 1, it means that first the non-zero element of ν[m,n] has been coded. m and n are the spatial locations of the current element in the code-block.
- If σ'[m,n] is 1, it means that *magnitude refinement coding*, which is to be explained, is applied to ν[m,n].
- If η[m,n] is 1, it means that zero coding, which is to be explained, is applied to ν[m,n].

- **Preferred Neighborhood:** For an element *y*[m,n], if one of its eight adjacent neighbors has a σ value of 1, it is said to be in a preferred neighborhood.
- **Zero Coding Tables.** The *context* values generated in *zero coding* operation are based on the σ values of the neighboring locations. For an element X, neighbors are defined in Figure 11. *D*, *H*, and *V* stand for diagonal, horizontal, and vertical neighbors, respectively.

D_0	V_0	D_1
H_0	X	H_1
D_3	V_1	D_2

Figure 11 Neighbors used in Zero Coding

Using σ values of the diagonal, horizontal, and vertical neighbors, context values are generated based on the subband being coded. For each subband, corresponding context values are defined by the standard. For example, if LL subband being coded, context values will be generated using Table 4, where x stands for *don't care*. Similar tables are defined in the standard [2] for the other subbands.

Table 4 Context Table LL and LH subbands

ΣH	ΣV	ΣD	CX
2	X	X	8
1	≥ 1	X	7
1	0	≥ 1	6
1	0	0	5
0	2	X	4
0	1	X	3
0	0	≥ 2	2
0	0	1	1
0	0	0	0

Index variables m and n can be out of range, such as m or n are negative or greater than the size of codeblock during coding process of boundaries of the codeblock. In these cases, σ and χ arrays are always assumed to be zero [4].

4.7.1.2 Coding Operations in EBCOT

EBCOT applies four different kinds of operations in three coding passes. These operations are Zero Coding (ZC), Sign Coding (SC), Magnitude Refinement Coding (MRC) and Run-length Coding (RLC).

- **Zero Coding:** In zero coding, *decision* (D) bit is simply the value of the $\nu^P[m,n]$. *Context* (CX) is calculated in two steps. First the diagonal, vertical and horizontal neighbors of the current bit in the σ state array are summed up individually (ΣD, ΣV, ΣH, e.g.). Then, CX values are selected from the predefined tables. There are three tables for four different subbands. For LL and LH subbands, Table 4 is used. For HL and HH subbands, Table 5 and Table 6 are used, respectively.

Table 5 Context Table HL subbands

ΣH	ΣV	ΣD	CX
X	2	X	8
≥ 1	1	X	7
0	1	≥ 1	6
0	1	0	5
2	0	X	4
1	0	X	3
0	0	≥ 2	2
0	0	1	1
0	0	0	0

Table 6 Context Table HH subbands

Σ(H+V)	ΣD	CX
X	≥ 3	8
≥ 1	2	7
0	2	6
≥ 2	1	5
1	1	4
0	1	3
≥2	0	2
1	0	1
0	0	0

- **Sign Coding (SC):** As for ZC, in SC also, CX and D values are calculated in two steps. First, vertical and horizontal reference values are calculated. Horizontal reference value is calculated using Eq. 4.6.

$$H = min\Big[-1, max\Big(-1, \sigma[m, n-1] * \big(1 - 2\chi(m, n-1) + \sigma[m, n+1] * (1 - 2\chi(m, n+1) +)\big)\Big)\Big] \quad (4.6)$$

Vertical reference value is calculated using Eq. 4.7.

$$V = min\Big[-1, max\Big(-1, \sigma[m-1, n] * \big(1 - 2\chi(m-1, n) + \sigma[m+1, n] * (1 - 2\chi(m+1, n) +)\big)\Big)\Big] \quad (4.7)$$

The results of H and V can be 0, 1 or -1.

- 0 means that both neighbors are insignificant or significant but have opposite signs.
- 1 means but neighbors are significant
- -1 means that one or more neighbors are significant but with negative signs.

After calculation of H and V reference values, context value *CX* is selected from Table 7. An intermediate variable $\hat{\chi}$ is also selected from Table 7. This variable is used in turn to calculate the decision bit D value.

Table 7 Sign Coding Reference Table

H	V	$\hat{\chi}$	CX
1	1	0	13
1	0	0	12
1	-1	0	11
0	1	0	10
0	0	0	9
0	-1	1	10
-1	1	1	11
-1	0	1	12
-1	-1	1	13

Decision bit D is calculated by using Eq. 4.8

$$D = \hat{\chi} \otimes \chi[m,n] \tag{4.8}$$

where $\chi[m,n]$ is the value of the current bit in sign array χ.

- **Magnitude Refinement Coding (MRC):** In MRC, *decision* bit D is simply the value of v^P[m,n], as in ZC. *Context* value CX is calculated using Eq. 4.9 and Table 8 in a two step process.

$$\begin{aligned}\tau = \sigma[m-1,n] + \sigma[m+1,n] + \sigma[m-1,n-1] \\ + \sigma[m-1,n+1] + \sigma[m+1,n-1] \\ + \sigma[m+1,n+]\end{aligned} \tag{4.9}$$

Table 8 Magnitude Refinement Coding Reference Table

σ`[m,n]	τ	CX
1	X	16
0	≥1	15
0	0	14

- **Run-Length Coding (RLC):** SC, ZC, and MRC generate a CX and D value pair for one bit position at a time. However, RLC works in a different fashion. RLC gets four consecutive bit positions and codes them as a whole. Number of bits coded in RLC depends on the position of the first bit with a value of 1. If all the bits are zero, then all four bits are coded.

  In RLC, either one or three CX and D pairs are generated. If all of the bits are zeroes, then one pair is generated. If all of the bits are not zeroes, then three pairs are generated. For example, if all four bits are zeroes, (CX, D) pair will have values of (17, 0), respectively. If all four bits are not zeroes, then first pair will be (17, 1), which means there is bit of value 1. Then, position of first bit 1 is signaled. CX value during position signaling is 18 and D values are the position of the first bit 1. As an example, for a scan pattern 0011, RLC will first generate (17, 1) then (18,1) and (18,0). D values 1 and 0 means that the first bit 1 is at the 3rd position. Position to decision values can be decoded as illustrated in Table 9.

Table 9 Position to Decision bit Value Conversion

Position	D_2	D_3
1	0	0
2	0	1
3	1	0
4	1	1

4.7.1.3 Coding Passes in EBCOT

There are three coding passes, significance propagation pass, magnitude refinement pass and cleanup pass, applied to each bit-plane except the most significant bit-plane. Only cleanup pass is applied to the most significant bit-plane.

In each coding pass, bit-plane is traversed according to the scan pattern and then the next pass starts. Coding passes are applied in the following order; SPP, MRP, CUP.

- **Significance Propagation Pass (SPP):** If the current scan position is in the preferred neighborhood and $\sigma[m,n]$ is zero, SPP applies zero coding to the current scan position. If not, scan position is incremented and the next position is selected. Also, if zero coding is applied $\eta[m,n]$ is set to 1. After zero coding is applied, it is checked whether current position requires sign coding as well. If $\nu^P[m,n]$ is 1, sign coding is also applied and $\sigma[m,n]$ is set to 1. Flowchart of the SPP is illustrated in Figure 12.
- **Magnitude Refinement Pass (MRP):** If $\sigma[m,n]$ of the current position is 1 and $\eta[m,n]$ is 0, magnitude refinement coding is applied to the current position. If MRC is applied, then $\sigma`[m,n]$ is also set to 1. Flowchart of the MRP is illustrated in Figure 13.
- **Cleanup Pass (CUP):** Cleanup Pass is the most complex pass in EBCOT algorithm. CUP first checks whether $\sigma[m,n]$ and $\eta[m,n]$ are both 0's or not. If any of them are not 0, current bit position is by-passed and next bit position is selected. If both $\sigma[m,n]$ and $\eta[m,n]$ are 0's, then the algorithm checks a condition to determine whether Run-length Coding (RLC) or Zero Coding (ZC) is to be applied. The condition for RLC is;
    1. m is a multiple of four, including 0. This means that RLC is applied if the current position is the first row in a stripe.
    2. σ is 0 for all the four consecutive positions on the same column.
    3. σ is 0 for all the adjacent neighbors of the four consecutive positions following the current position.

  If any one of the above conditions does not apply, then ZC is applied to the current position. It should be noted that, RLC may process more than one bit position. If RLC processes more than one bit location, position pointer must be taken care of accordingly.

  After RLC or SC, another check is done for SC. If $\nu^P[m,n]$ is 1, then SC is also applied to the last coded bit position. SC sets $\sigma[m,n]$ to 1. Flowchart of the CUP is illustrated in Figure 14.

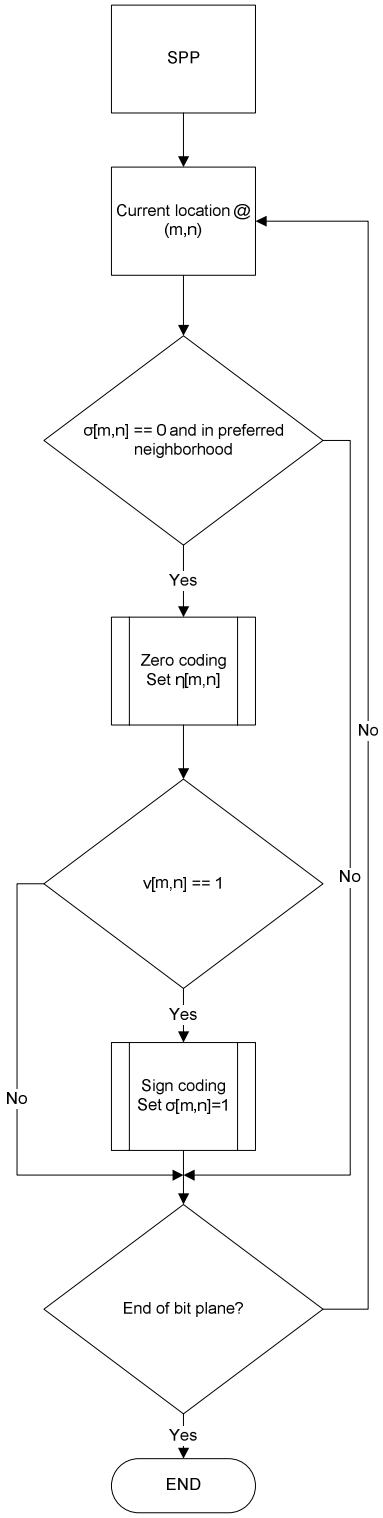

Figure 12 Significance Propagation Pass flowchart

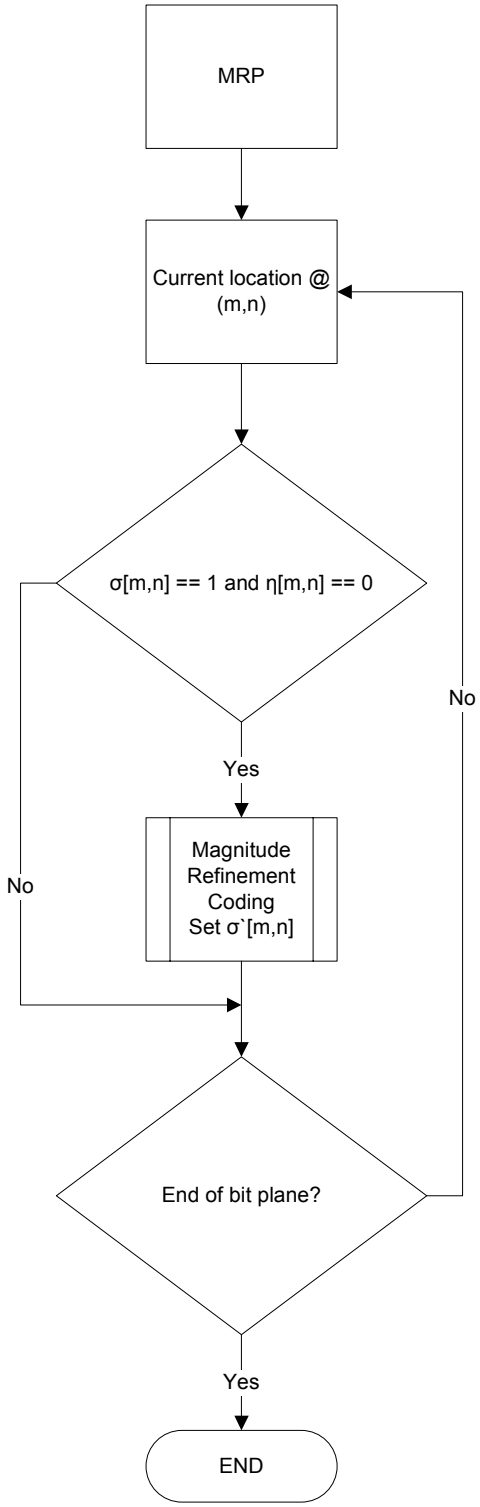

Figure 13 Magnitude Refinement Pass flowchart

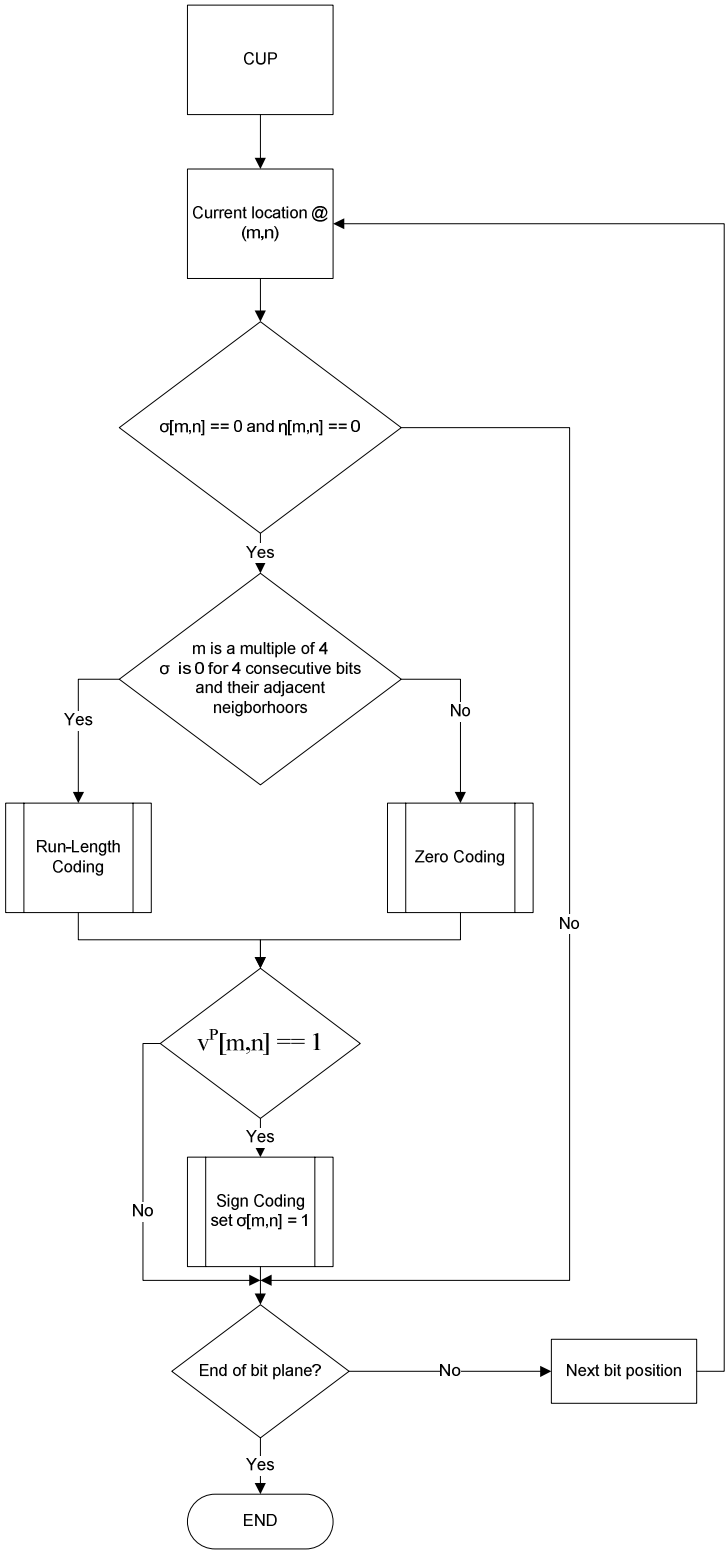

Figure 14 Cleanup Pass flowchart

4.7.2 MQ-Coder

MQ-Coder [5] is the part of the coding system that generates the compressed data using the *context* information (CX) and *decision* bit (D) pairs generated by EBCOT.

As explained in Section 2.4, arithmetic coding requires the probability values of the source data. JPEG2000 standard provides a predetermined table of probability values (Qe) and probability estimation/mapping process [2].

MQ-Coder is implemented using two 32-bit registers, named as A and C. Register A holds the current interval of the coder. Register C holds partial codeword during coding process. Structures of register A and C are listed in Table 10.

Table 10 Register Structures in MQ-Coder

32-bit Register	MSB	LSB
C	0000 cbbb bbbb bsss	xxxx xxxx xxxx xxxx
A	0000 0000 0000 0000	aaaa aaaa aaaa aaaa

In Table 10;

- *a* means fractional bits in register A.
- *x* means fractional bits in register C.
- *s* means space bits.
- *b* means bits for ByteOut procedure.
- *c* means the carry bit.

The procedures of MQ-Coder are; Initialization(), CodeMPS(), CodeLPS(), renormalization() and ByteOut(). ByteOut() procedure may call two sub-procedures, bitStuffing() and noBitStuffing(). The overall flow of EBCOT algorithm is presented in Figure 15.

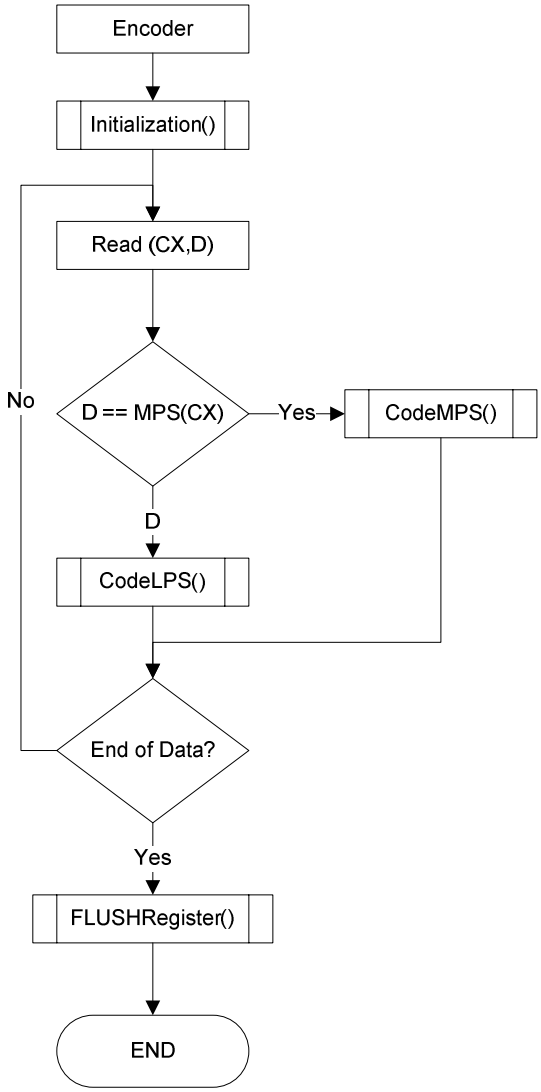

Figure 15 MQ-Coder Encoder Algorithm

The descriptions of the procedures used during coding process are follows.

- **Initialization():** In this procedure, registers A and C are set to their initial values defined by the standard. Some other variables that are used during coding process are also initialized to their initial values. Pseudo-code of the algorithm which is described in [4] is as follows.

Initialization()

{

```
        A = 0x00008000
        C = 0x00000000
        BPST = 0
        BP = BPST -1
        B = 0
        CT = 12
        if (B == 0xFF)
                CT = 13
        reset I(CX) to initial values
        reset MPS(CX) to initial values
}
```

Variables that are used in Initialization() procedure are B, BPST, BP and CT. BP is the compressed data buffer pointer. B is the byte in the compressed data buffer which is pointed by BP. BPST is the pointer that shows first position that encoder will write on. Initial value of BPST is 0 as described in [14].

- **CodeMPS():** In this procedure, probability estimate of the current context is added to the C register. Then current interval is recalculated. After calculation of the new interval, the procedure determines whether conditional exchange is required or not. Next index is selected by using the lookup table NMPS(I(CX)). The pseudo code for the procedure is as follows.

```
CodeMPS()
begin
qe = Qe(I(CX))
A = A – qe
if (A < 0x80000)
        if(A < qe)
                A = qe
        else
                C = C + qe
                I(CX) = NMPS(I(CX))
                renormalization();

else
        C = C + qe
```

```
end
```

- **CodeLPS():** The procedure first adjusts the interval than checks the value of the new interval against the probability value of the current context value. Depending on the index value of the current context, the sense of MPS may change depending on the value of SWITCH flag for the context value. In CodeLPS(), renormalization() procedure is always called. The pseudo code for the procedure is as follows.

```
CodeLPS()
begin
qe = Qe(I(CX))
A = A – qe
if (A => qe)
        A = qe
else
        C = C + qe

if (switch(I(CX)))
        MPS(CX) = 1 – MPS(CX)

I(CX) = NLPS(I(CX))
renormalization ()
end
```

- **renormalization():** The purpose of renormalization() procedure is to ensure that interval is above 0x8000 in every cycle of the coding. This is done by left-shifting both A and C registers until interval gets greater than 0x8000. A special counter which is initialized in Initialization() procedure, CT, is decremented after every left-shift operation. When CT gets to zero, ByteOut() procedure is called. The pseudo code for the procedure is as follows.

```
renormalization()
begin
        while (A < 0x8000)
                A = A <<1
                C = C <<1
```

```
                    CT = CT <<1
                    if (CT == 0)
                        ByteOut()
            end
```

- **ByteOut():** MQ-Coder generates the compressed data in this procedure. Depending on the conditions, it may call either bitStuffing() or noBitStuffing() procedures. The pseudo code for the procedure is as follows.

```
ByteOut()
begin
    if (B == 0xFF)
        bitStuffing()
    else
        if (C < 0x08000000)
            noBitStuffing()
        else
            B = B + 1
            if (B = 0xFF)
                C = C & 0x07FFFFFF
                bitStuffing()
            else
                noBitStuffing()
end
```

- **bitStuffing():** The carry bit *c* and the most significant 7 bits *bs* as shown in Table 10 are moved into the byte B in this procedure. The pseudo code for the procedure is as follows.

```
bitStuffing()
begin
    BP = BP + 1
    B = C >> 20
    C = C & 0x000FFFFF
    CT = 7
end
```

- **noBitStuffing():** In this procedure, the 8 bs bits as shown in Table 10 are moved into the byte B. The pseudo code for the procedure is as follows.

```
nobitStuffing()
begin
        BP = BP + 1
        B = C >> 19
        C = C & 0x0007FFFF
        CT = 8
end
```

- **FLUSHRegister():** The last step of the encoding operation is calling FLUSHRegister() procedure. In this procedure, register C is stuffed with as many 1 bits as possible. The pseudo code for the procedure is as follows.

```
FLUSHRegister()
begin
        TempC = C + A
        C = C | 0x0000FFFF
        if (C => TempC)
                C = C – 0x00008000
        C = C << CT
        ByteOut()
        C = C << CT
        ByteOut()
        C = C << CT
        if (B == 0xFF)
                discard B
        else
                BP = BP +1
end
```

CHAPTER 5

VLSI IMPLEMENTATION OF JPEG2000

VLSI designs generally targets meeting the timing constraints using minimum resources. There is always a trade-off between resource usage and timing closure. Techniques like unfolding aims to help meeting the timing constraints at the expense of resource usage. In this thesis, the first goal is to get the correct functionally. After that, the implementation aims to get a system as fast as possible.

The main advantage of VLSI designs is their ability to run concurrent processes at the same time. For example, RCT part of this thesis needs to generate three new color space value at the same time.

The system is implemented using Verilog HDL and Synopsys Synplify DSP. Functionality of the implemented system is verified using Synopsys VCS RTL simulator. A software decoder is implemented in C and Python programming languages to verify the functionality of the system. For RTL synthesis, Synopsys Synplify Pro is used.

5.1 VLSI Architecture for Image Preprocessing

As explained in Section 4.2, image preprocessing consists of three parts; Tiling, DC Level Shifting and Reversible Color Transform.

For an image of size *M x N*, Tiling requires that at least *Tile Height x N* samples of a raster scan image is available in memory. Then, image is read from the memory in the intended format. In this thesis, it is assumed that image is supplied in tiles, thus no special work is done for tiling the image. Also, tile size is selected as 256 x 256 during the implementation of the thesis.

5.1.1 DC Level Shifting

As given in Section 4.2.2., DC Level shifting is done by subtracting a constant value that depends on the bit width of the image's intensity values. In this thesis, it is

assumed that input images are 8-bit for each channel. Thus, using Eq. 4.1, the constant value that needs to be subtracted from the intensity values is 128.

VLSI implementation of DC Level Shifting is straightforward and it uses only adders in subtract or configuration. Block diagram of the implementation for a single color channel is given in Figure 16.

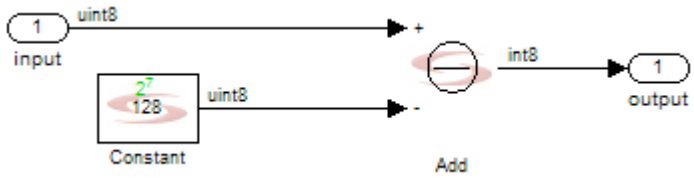

Figure 16 DC Level Shifter

5.1.2 Reversible Color Transform

VLSI implementation of Reversible Color Transform simply follows its mathematical definition. RCT calculates three component channels, Y, U and V in parallel. The overall system is presented in Figure 17.

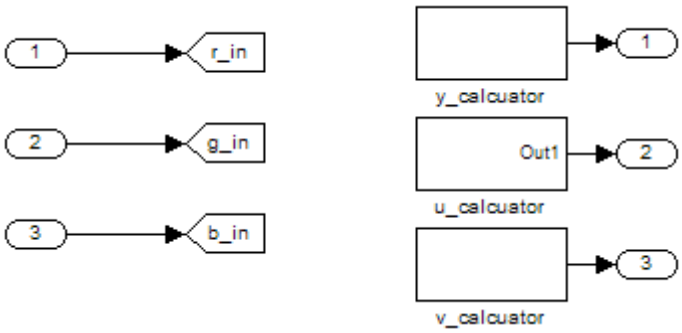

Figure 17 Overall System of RCT

Calculation of Y channel involves division by 2 and 4 and then summation. Division by 4 and 2 are simply done by left shifting the input by 2 bits and 1 bit, respectively. The subsystem that calculates Y component is presented in Figure 18.

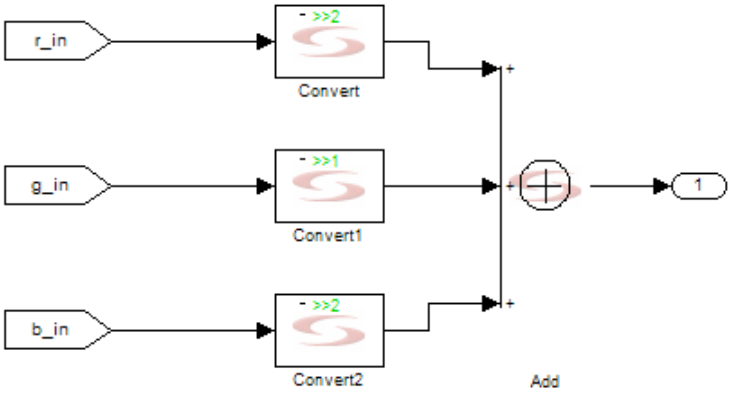

Figure 18 *Y* Component Calculation Subsystem

Calculation of U and V components simply involves subtracting one component from another as given in Eq. 4.2. The subsystems that calculate U and V components are illustrated in Figure 19 and Figure 20, respectively.

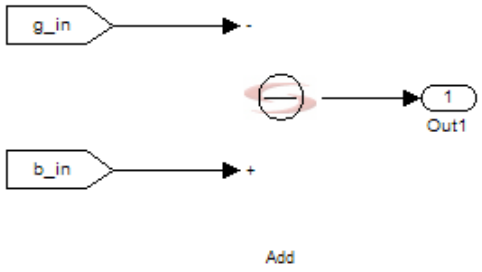

Figure 19 *U* Component Calculation Subsystem

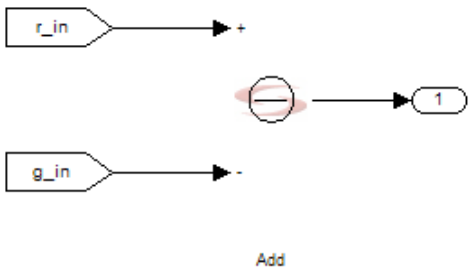

Figure 20 *V* Component Calculation Subsystem

This implementation does not require any memory element and only contains combinatorial logic elements to generate the output. Thus, in terms of hardware resource, RCT implementation could be called as an efficient implementation.

5.1 VLSI Architecture for DWT

VLSI implementations of DWT could be classified into two; convolution approach and lifting approach. As discussed in Section 3.6, lifting implementation of DWT offers many advantages over traditional FIR based implementations. One of these advantages is parallelism of the lifting implementation, which is an important figure for VLSI implementations. Moreover, JPEG2000 standard also encourages lifting based approach. Hence, lifting based approach is chosen in DWT implementation of this thesis.

JPEG2000 standard [2] defines 1D reversible filtering (Le Gall 5/3) as

$$Y(2n+1) = X_{ext}(2n+1) - \left\lfloor \frac{X_{ext}(2n) + X_{ext}(2n+2)}{2} \right\rfloor \qquad (5.1)$$

and

$$Y(2n) = X_{ext}(2n) - \left\lfloor \frac{Y(2n-1) + Y(2n+1)}{4} \right\rfloor \qquad (5.2)$$

for high-pass filter output and low-pass filter output, respectively. In order to calculate the result of Eq. 5.2, the result of Eq. 5.1 needs to be calculated. Thus, there is a data dependency issue for lifting implementation.

In this thesis, direct mapping of data-flow of lifting steps which is proposed in [15] and illustrated in [13] is used. The 1D DWT architecture is shown in Figure 21, Figure 22 and Figure 23.

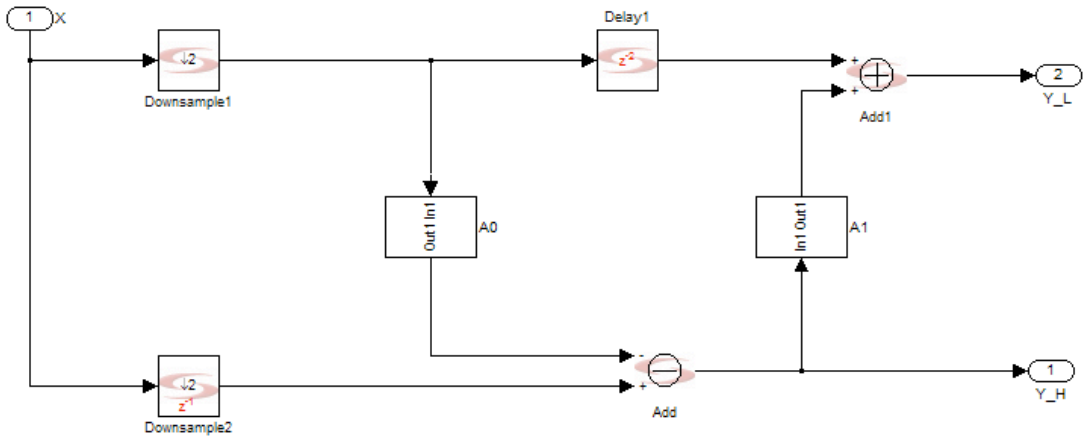

Figure 21 VLSI Architecture of 1D DWT

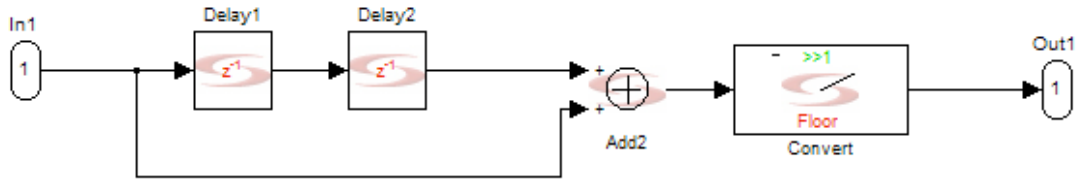

Figure 22 Predict Section of 1D DWT

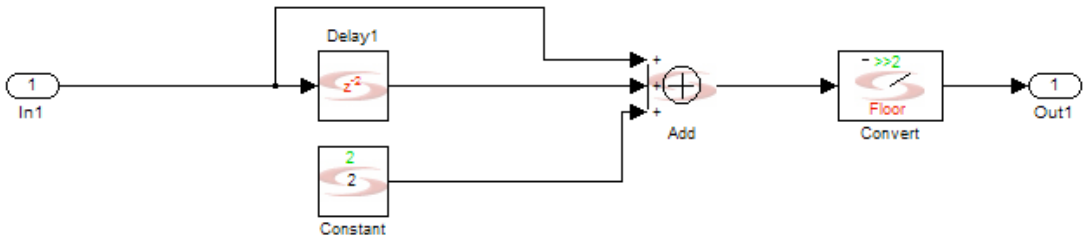

Figure 23 Update Section of 1D DWT

Le Gall (5, 3) spline filter has only two stages, thus optimized architectures are not strictly required. However, Daubechies (9, 7) biorthogonal spline filter has two intermediate stages, which may require optimized architectures depending on the resources.

Le Gall (5, 3) spline filter is an integer to integer transform and has only trivial multiplication and division operations which can be implemented using bit-shifting operations. On the other hand, Daubechies (9, 7) biorthogonal spline filter is a real transform and requires arithmetic multipliers, which are very precious in VLSI implementations. In [16], it has been proposed that a multiplier used in lifting scheme can be shared by time-multiplexing the multipliers, which is called folding.

Also, one another optimization that can be used is pipelining the lifting scheme. In convolution based approach, there exists a shift register and an adder that sums up the shift register tap values. Critical path of the filter is usually at the adder section, since multi-bit, multi-input adders have long critical paths. However, lifting-scheme avoids the long shift registers and number of registers that would break the critical path is limited. Thus, there occurs a long critical path in direct mappings of data flow of lifting scheme in Daubechies (9, 7) biorthogonal spline filter. In [15] and [17], pipelined architectures for breaking the critical paths are proposed.

In addition to the architectural changes, enough retiming registers can be added to the output ports of the lifting scheme, and let the logic synthesis software to pipeline the design.

JPEG2000 standard defines DWT as 1D transform and lets the designer to decide how to apply 2D extension to the transform. As discussed in Section 3.5, 2D DWT is calculated by first applying 1D transform row-wise, then applying 1D transform column-wise. This operation requires matrix transpose operation over streaming data. Matrix transpose operation is implemented using double-buffering scheme [18].

In double-buffering scheme implemented in this thesis, there exists two buffers. Buffers are written in sequential order and read in transposed order. Instead of implementing two separate buffer elements, two buffers are combined together. The architecture used for transposing the data is shown in Figure 24.

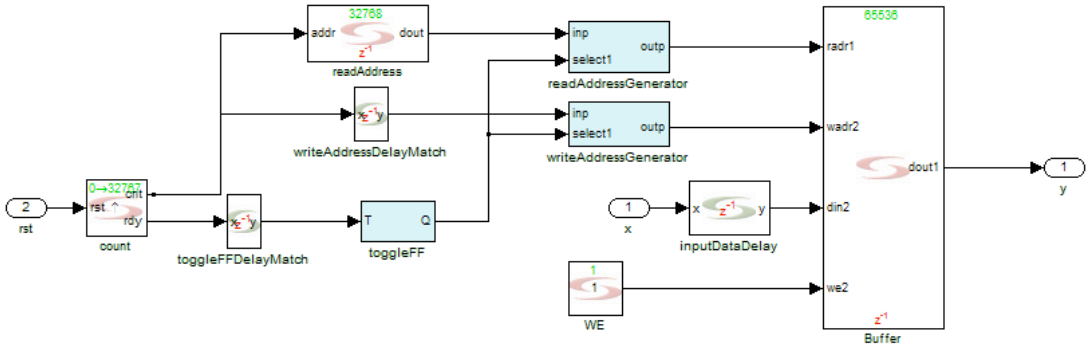

Figure 24 Double-Buffering Transposer

5.2 VLSI Architecture for EBCOT

EBCOT algorithm, unlike DWT and RCT, is a sequential algorithm rather than a parallel algorithm. Thus a state-machine based design is more suited for this kind of application. In this thesis, EBCOT implementation is largely based on [19], which uses a state machine based approach.

The EBCOT implementation consists of a controller, memory modules and combinational logic units. Following are the building blocks of the system.

1. Five special purpose shift-registers; σ-reg, η-reg, σ`-reg, ν -reg and χ-reg for processing the state variables. These special purpose shift-registers have different lengths. Details of these registers are explained in succeeding sections.
2. Separate combinational logic units for calculating CX and D values for SC, ZC and MRC operations. RLC is not implemented in a separate logic unit and it is handled by the controller itself.
3. Local memory modules; σ-MEM, η-MEM, σ`-MEM of size 32 x 4. They hold the state variables. Magnitude and sign arrays are also stored in ν –MEM and χ-MEM memory modules of the size 256 x 256.
4. Context and data multiplexer (MUX) chooses the correct CX and D data from the corresponding combinational logic unit (SC, ZC and MRC).
5. A controller that controls the special purpose shift-registers and MUX. Controller also loads the wavelet coefficients to local ν –MEM and χ-MEM memories and keeps track of the coding process using counter registers.

5.2.1 Registers

EBCOT algorithm requires the state values of the current position and neighboring locations. Also, inaccessible locations such as borders need to be assumed zero [13]. Thus, shift-register based method proposed in [19] is used.

- σ-reg is 16 taps shift-register that holds the state variables of three stripes during coding process. Structure of σ-reg is given in Figure 25.

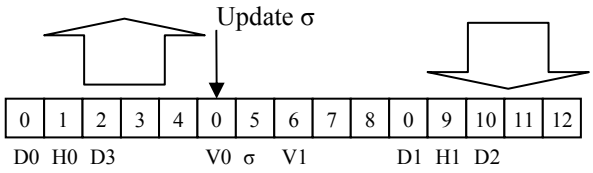

Figure 25 σ-reg Structure

σ-reg is capable of shifting 5 bits and 1 bit at a time. Update bit is set any time σ value needs to be set. Also, σ-reg generates five other signals based on the values of σ-reg. These signals are; RLC_C which indicates RLC condition is satisfied, nhood0 which indicates all the adjacent neighbors are zeros. The other three signals V_w, H_w and D_w are the vertical, horizontal and diagonal neighbor values, respectively.

In [19], σ-reg is said to have the capability to shift 5 bits and 1 bit at one clock cycle. This means that σ-reg is a barrel-shifter [20]. However, implementing a barrel-shifter is a costly operation in VLSI. For instance, in an application note of Xilinx, barrel-shifters are implemented using multipliers [21]. Since the shift amounts are fixed, an ordinary register with the necessary glue logic in a feedback structure that shifts different amounts and reassigns the new value to the register is implemented. The pseudo-code for the glue logic is given below.

```
If (wen)
   sigmaCurrentValue = memINBit4
   sigmaCurrentValue = memINBit3
   sigmaCurrentValue = memINBit2
   sigmaCurrentValue = memINBit1
endif
```

```
if (update)
   sigmaNextValue = bitset(sigmaCurrentValue,6,1)
endif

if (shift5bit)
   sigmaNextValue = bitshift(sigmaCurrentValue,-5)
endif

if (shift1bit)
   sigmaNextValue = bitshift(sigmaCurrentValue,-1)
endif
```

- η-reg and σ`-reg are 8 bits shift-registers with extra border spacing. They have the capability to shift 1 bit at one clock cycle. Also, bit position 3 is used to set the register. Borrowing from the same idea used for σ-reg, an ordinary register with 1 bit shift capability is implemented.
- χ-reg is very similar to σ-reg. Unlike σ-reg, χ-reg does not have extra border spacing. Thus, it has a size of 12 bits. Also, χ-reg is capable of shifting 4 bits and 1 bit at one clock cycle. The structure of χ-reg is given in Figure 26.

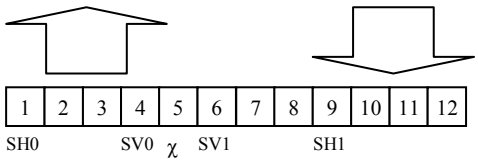

Figure 26 χ-reg Structure

- ν –reg is a simple 4 bits register. Only a zero detector is attached to its output that generates 1 if all the values are 0s. This signal is referred as *All0s*. Also, a special encoder (zero index encoder) is placed at the output to determine the first bit 1 in the stripe. The output of the encoder is used in RLC. The pseudo-code for the zero index encoder is presented below. Default output is set to zero to prevent latch instantiation in logic synthesis tools.

```
if (input ==  1)
   output = 0
elseif (input ==  2)
   output = 1
elseif (input ==  3)
   output = 0
elseif (input ==  4)
   output = 2
```

```
elseif (input ==  5)
   output = 0
elseif (input ==  6)
   output = 1
elseif (input ==  7)
   output = 0
elseif (input ==  8)
   output = 3
elseif (input ==  9)
   output = 0
elseif (input == 10)
   output = 1
elseif (input == 11)
   output = 0
elseif (input == 12)
   output = 2
elseif (input == 13)
   output = 0
elseif (input == 14)
   output = 1
elseif (input == 15)
   output = 0
else
   output = 0
endif
```

5.2.2 Combinational Logic Units

SC, ZC, and MRC units generate the CX and D values during coding process. RLC is handled inside the controller. Details of the coding operations are explained in Section 4.7.1.2.

- Sign coding operation first calculates horizontal and vertical contributions and then generates the CX and D value. The pseudo-code for sign coding combinational logic unit is given below.

```
H = min (1,max (-1, (H0* (1-2*SH0) ) + (H1 * (1-2* SH1) ) ) )
V = min(1, max (-1, (V0* (1-2*SV0) ) + (V1 * (1-2*SV1 ) ) ) )

if (H == 1 && V == 1)
   chiPrime = 0
   CX = 13
elseif (H == 1 && V == 0)
   chiPrime = 0
   CX = 12
elseif (H == 1 && V == -1)
   chiPrime = 0
   CX = 11
elseif (H == 0 && V == 1)
   chiPrime = 0
```

```
    CX = 10
elseif (H == 0 && V == 0)
    chiPrime = 0
    CX = 9
elseif (H == 0 && V == -1)
    chiPrime = 1
    CX = 10
elseif (H == -1 && V == 1)
    chiPrime = 1
    CX = 11
elseif (H == -1 && V == 0)
    chiPrime = 1
    CX = 12;
elseif (H == -1 && V == -1)
    chiPrime = 1
    CX = 13
endif

D = chiPrime^chi11
```

- Zero coding operation generates different values for the subband being coded. V_w, H_w and D_w values from σ-reg are also used. The pseudo-code for zero coding operation is given below.

```
if (subband == 0)
    # subband LL or LH
    if (H_w == 2)
        CX = 8
    elseif (H_w == 1 && V_w >= 1)
        CX = 7
    elseif (H_w == 1 && V_w == 0 && D_w >= 1)
        CX = 6
    elseif (H_w == 1 && V_w == 0 && D_w == 0)
        CX = 5
    elseif (H_w == 0 && V_w == 2)
        CX = 4
    elseif (H_w == 0 && V_w == 1)
        CX = 3
    elseif (H_w == 0 && V_w == 0 && D_w >= 2)
        CX = 2
    elseif (H_w == 0 && V_w == 0 && D_w == 1)
        CX = 1
    elseif (H_w == 0 && V_w == 0 && D_w == 0)
        CX = 0
    end
elseif (subband == 1)
    # subband HL
    if (V_w == 2)
        CX = 8
    elseif (H_w >= 1 && V_w == 1)
        CX = 7;
```

```
    elseif (H_w == 0 && V_w == 1 && D_w >= 1)
      CX = 6
    elseif (H_w == 0 && V_w == 1 && D_w == 0)
      CX = 5
    elseif (H_w == 2 && V_w == 0)
      CX = 4
    elseif (H_w == 1 && V_w == 0)
      CX = 3
    elseif (H_w == 0 && V_w == 0 && D_w >= 2)
      CX = 2
    elseif (H_w == 0 && V_w == 0 && D_w == 1)
      CX = 1
    elseif (H_w == 0 && V_w == 0 && D_w == 0)
    CX = 0
    end
  elseif(subband == 2)
    # subband HH
    if(D_w >= 3)
      CX = 8
    elseif( (H_w + V_w) >= 1 && D_w == 2)
      CX = 7
    elseif( (H_w + V_w) == 0 && D_w == 2)
      CX = 6
    elseif( (H_w + V_w) >= 2 && D_w == 1)
      CX = 5
    elseif( (H_w + V_w) == 1 && D_w == 1)
      CX = 4
    elseif( (H_w + V_w) == 0 && D_w == 1)
      CX = 3
    elseif( (H_w + V_w) >= 2 && D_w == 0)
      CX = 2
    elseif( (H_w + V_w) == 1 && D_w == 0)
      CX = 1
    elseif( (H_w + V_w) == 0 && D_w == 0)
      CX = 0
    endif
  endif
```

- Magnitude refinement coding is a quite straightforward operation. It concatenates the value of σ`-reg and nhood0 signal generated in σ-reg bitwise. The pseudo-code for magnitude refinement coding operation is given below.

$nhood0 = \sigma_{00} \mid \sigma_{01} \mid \sigma_{02} \mid \sigma_{10} \mid \sigma_{12} \mid \sigma_{20} \mid \sigma_{21} \mid \sigma_{22}$
$CX[4] = \sigma`_{11}$
$CX[3] = ! \sigma`_{11}$
$CX[2] = ! \sigma`_{11}$
$CX[1] = ! \sigma`_{11}$
$CX[0] = ! \sigma`_{11} \ \& \ nhood0$

5.2.3 Local Memory Modules

As explained in Section 4.7.1, EBCOT algorithm keeps track of three state variables for each location of a codeblock. In the VLSI implementation of EBCOT algorithm, these state variables are kept in three separate memory modules σ-MEM, η-MEM and σ`-MEM. Each memory module is of size 32 x 4.

Magnitude and sign arrays, which are generated from the input wavelet coefficients, are also stored in ν –MEM and χ-MEM memory modules of size 256 x 256.

Implementation of these memory modules requires one vital feature that needs to be tackled. In vertical causal mode, EBCOT algorithm needs to read four memory locations (one stripe) in one clock cycle. Thus, each memory module is divided into four parts. Each part holds a part of the whole data. For example, RAM1 holds the data of the first rows and RAM2 holds the data of the second rows. The overall structure of ν–MEM and χ-MEM is shown in Figure 27 and that of the σ-MEM, η-MEM and σ`-MEM is illustrated Figure 28.

Dividing the whole memory requires handling of the address bus to select and use the right memory module. For σ-MEM, η-MEM and σ`-MEM, this operation is straightforward, since the overall memory size is 32x4. RAM modules use the index of one dimension of the 2D array as address bus of the memory modules. However, ν–MEM and χ-MEM requires a special circuitry to handle the address bus and enable signals of the RAM modules. *calculateRWAddress* module in Figure 27, generates the address bus signal of the RAM modules. *wenGenerator* module generates the enable signals for RAM modules. *calculateRWAddress* module extracts three most significant bits of the index of X dimension, multiplies it by 32 and adds the result to Y index. *wenGenerator* finds the remainder by 4 of X index. The result of remainder operation is connected to a multiplexer that selects 0b0001, 0b0010, 0b0100 or 0b1000. The output bits of multiplexer are extracted and AND operation is carried out with the write enable signal of the ν–MEM and χ-MEM modules. The output of AND gates are connected to RAM modules. The structures of calculateRWAddress, Remainderby4 and wenGenerator modules are shown in Figure 29, Figure 30 and Figure 31, respectively.

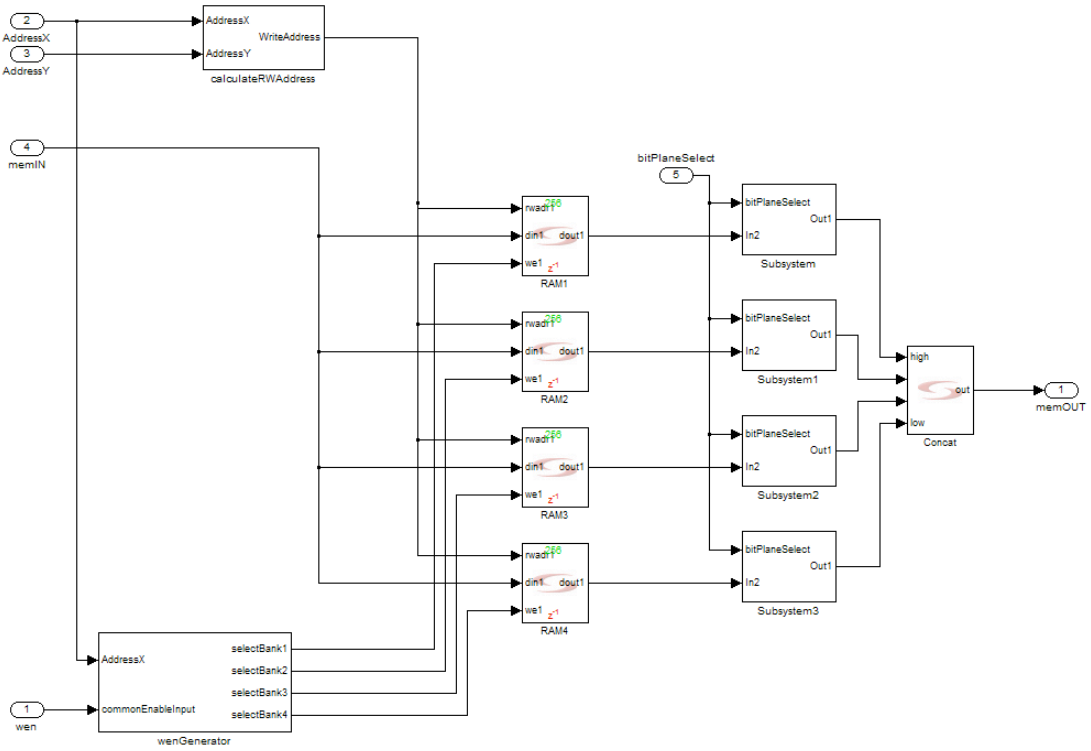

Figure 27 ν–MEM and χ-MEM Structure

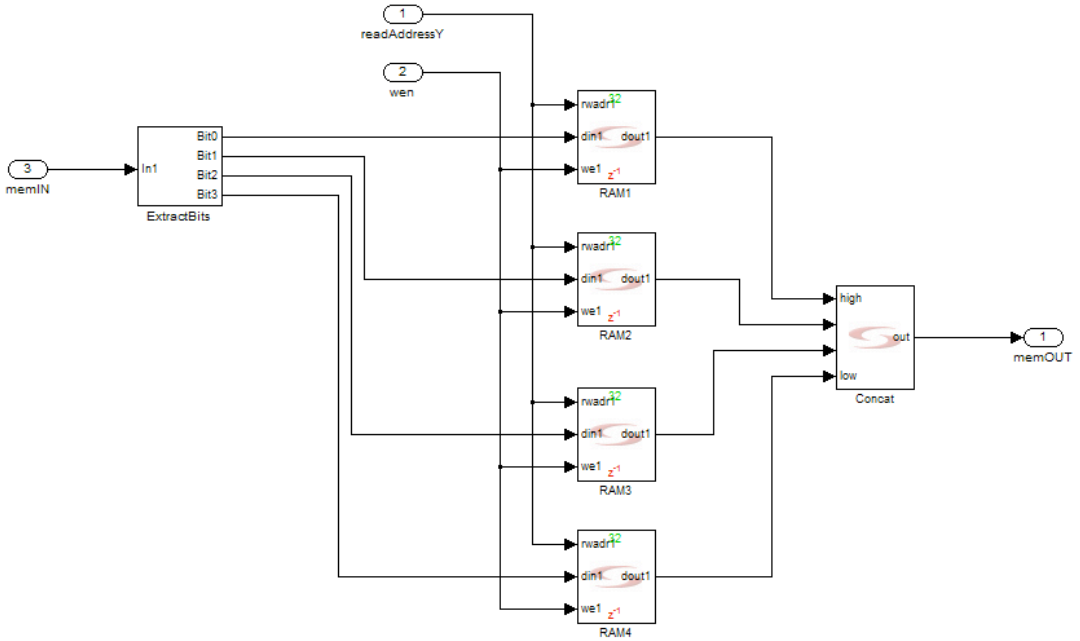

Figure 28 σ-MEM, η-MEM and σ`-MEM Structure

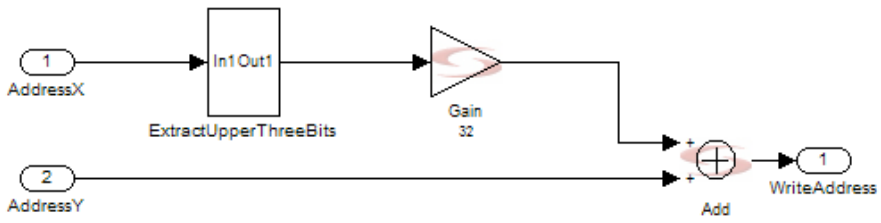

Figure 29 calculateRWAddress Structure

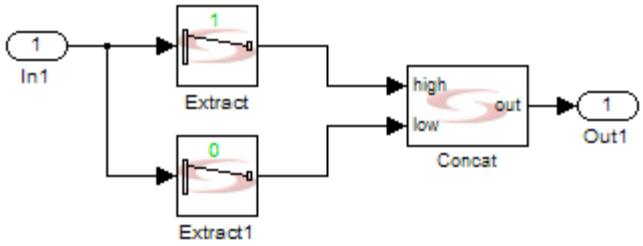

Figure 30 Remainderby4 Structure

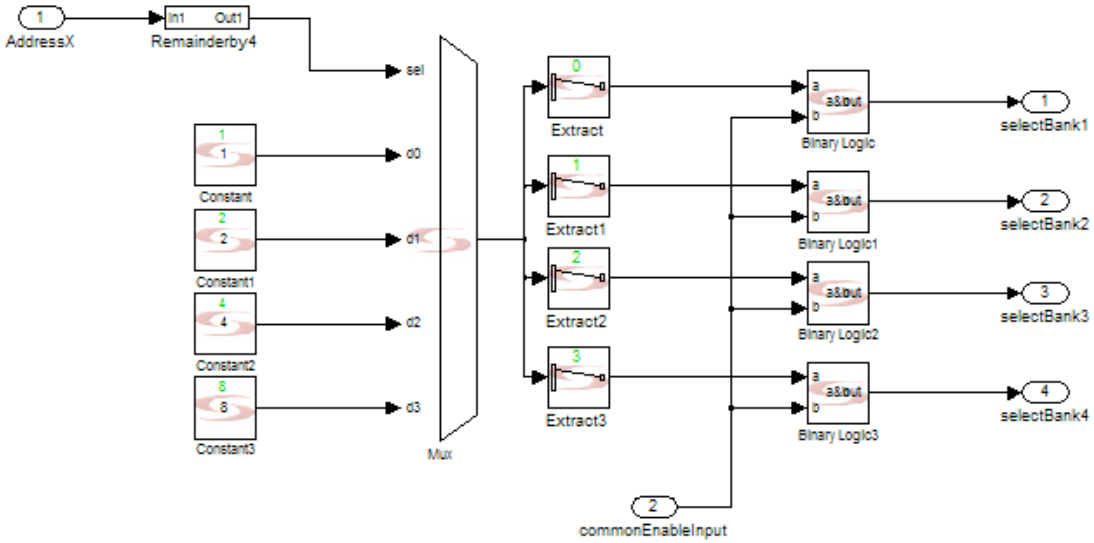

Figure 31 wenGenerator Structure

5.2.4 Context and Data Multiplexer

Context and Data Multiplexer module consists of two multiplexers for selecting the CX and D values generated by Combinational Logic Units. The input output relationships of the Context and Data Multiplexer module are listed in Table 11.

Table 11Context and Data Multiplexer outputs

$cntrl_{cx}$	**CX**	**D**
0	X	X
1	ZC_CX	ν
2	SC_CX	SC_D
3	MC_CX	ν
4	17	0
5	17	1
6	18	ZI[MSB]
7	18	ZI[LSB]

Pseudo-code for the Context and Data Multiplexer is given below.

```
if (cntrl_cx == 1)
   CX = ZCContext
   D  = nuCurrent
elseif (cntrl_cx == 2)
   CX = SCContext
   D = SCData
elseif (cntrl_cx == 3)
   CX = MRCContext
   D  = nuCurrent
elseif (cntrl_cx == 4)
   CX = 17
   D  = 0
elseif (cntrl_cx == 5)
   CX = 17
   D  = 1
elseif (cntrl_cx == 6)
   CX = 18
   D  = ZI_1
elseif (cntrl_cx == 7)
   CX = 18
   D  = ZI_0
else
   CX = X
   D  = X
endif
```

The output of the Context and Data Multiplexer is set to zero for any other condition to prevent latch inference in logic synthesis tools.

5.2.5 EBCOT Controller

The controller is responsible for loading the wavelet coefficients into local memory modules ν–MEM and χ-MEM, processing the data, updating state variables and selecting the appropriate CX and D from Context and Data Multiplexer.

The controller is implemented as a state machine of 24 different states. These states are divided into five phases.

1. Initialization phase,
2. ZC and SC control phase,
3. MRC control phase,
4. RLC control phase,
5. Termination phase.

Details of these phases are explained in preceding sections. Numbers in the boxes given in flowcharts represent the state numbers used during coding.

5.2.5.1 Initialization Phase

In this phase, wavelet coefficients are loaded into the ν–MEM and χ-MEM memory modules and all the pointers to memory modules are reset. Then, the first stripe is read into σ-reg, χ-reg and shifter 5 and 4 bits, respectively. After the shifting operation, based on the pass being processed, next state is chosen. The flowchart of the initialization phase is shown in Figure 32.

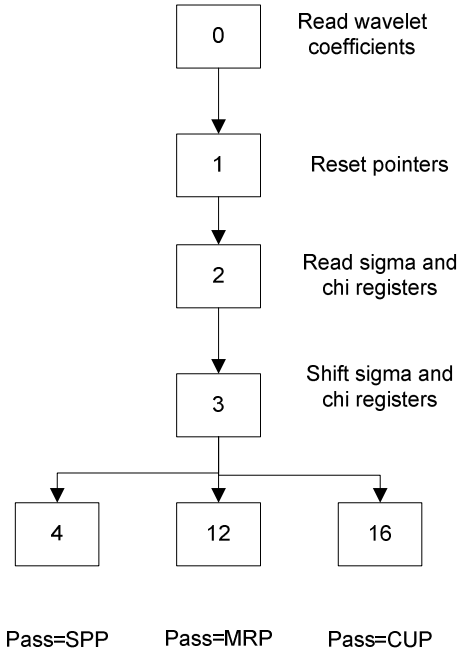

Figure 32 Initialization Phase Flowchart

5.2.5.2 ZC and SC Control Phase

In this phase, data from corresponding memories are read into σ-reg, χ-reg and ν-reg. Then, the conditions whether SC and ZC are required or not are checked. If required, corresponding control signal for Context and Data Multiplexer is generated. This process is done for each element in the stripe. After processing all the elements of the current stripe, conditions are checked whether to enter the termination phase or not. The flowchart of ZC and SC Control Phase is given in Figure 33.

5.2.5.3 MRC Control Phase

In MRC phase, σ-reg, η-reg, σ`-reg, and ν-reg are read first. Then, based on the values of σ and η, either the control signal is generated or next bit position is selected. If a bit position is coded in MRC phase, then σ`-reg is also updated. After processing the entire elements in a stripe, termination phase is invoked. The flowchart of MRC Control Phase is given in Figure 34.

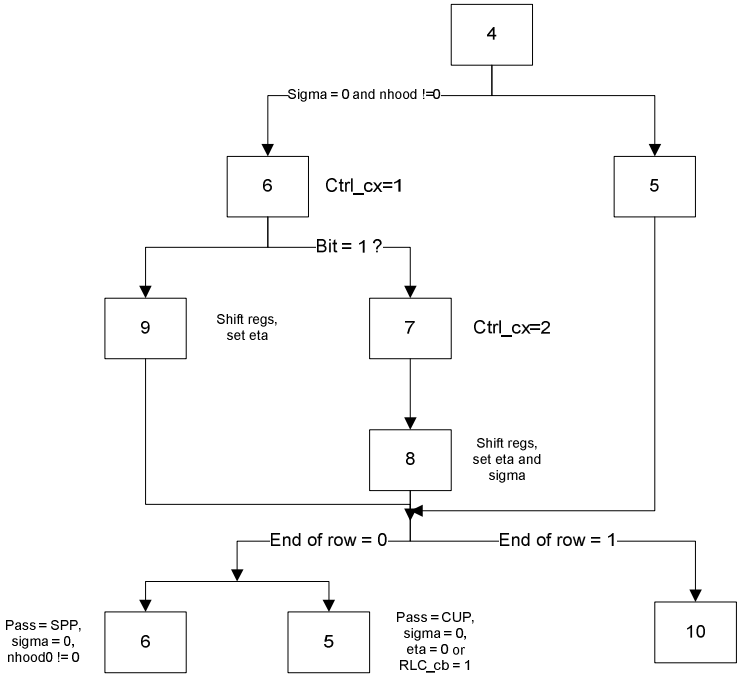

Figure 33 ZC and SC Control Phase Flowchart

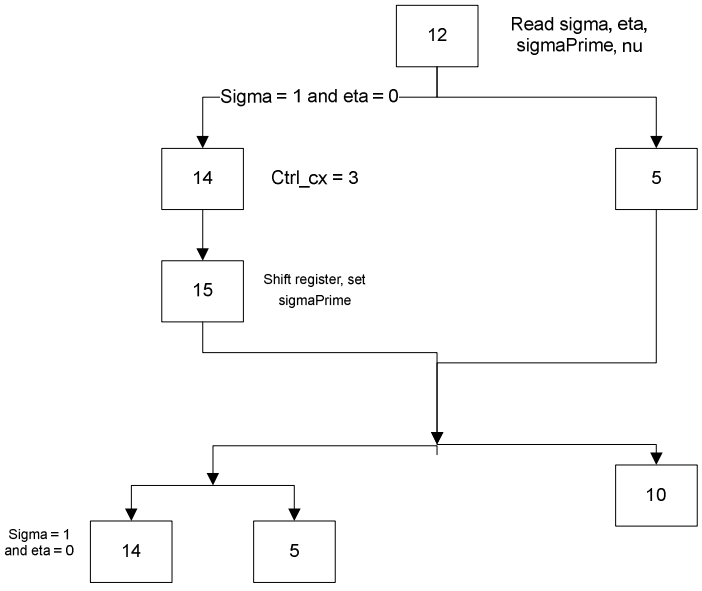

Figure 34 MRC Control Phase Flowchart

5.2.5.4 RLC Control Phase

RLC phase start by reading data into σ-reg, η -reg, χ -reg, and ν-reg. Then, states of σ, η and RLC_c are checked for RLC condition. If RLC condition is satisfied, RLC coding starts. If not, the next state becomes ZC and SC Coding Phase. RLC coding

first checks whether all the elements of the stripe is 0s or not. If they are all 0s, then σ-reg and χ –reg are shifter 5 bits and 4 bits, respectively. If all the elements are not zeros, controller sends the necessary control signals to Context and Data Multiplexer to generate the CX and D pairs for RLC coding. As shown in Section 5.2.4, CX and D values are hard-coded into Context and Data Multiplexer. The operation of RLC Control phase is shown on Figure 35.

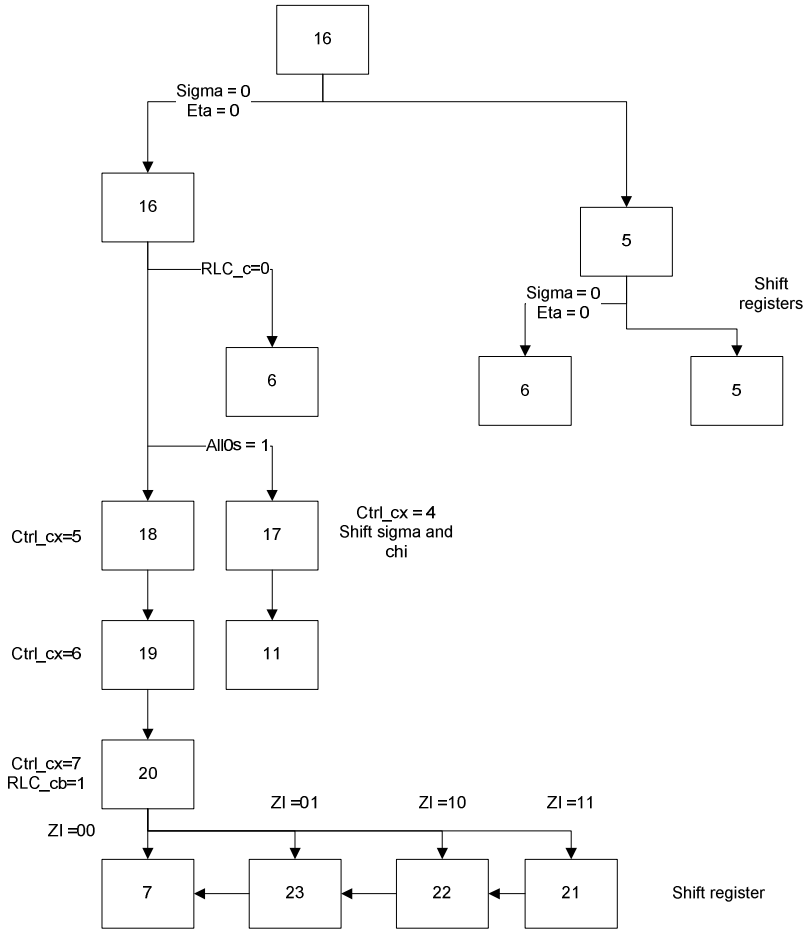

Figure 35 RLC Control Phase Flowchart

5.2.5.5 Termination Phase

Termination phase is invoked at the end of each stripe. First, σ-reg is shifted by 1 bit. Then, values in σ-reg, η-reg, σ`-reg are written back to σ-MEM, η-MEM, σ`-MEM, respectively. If the current pass is CUP, coding ends. If not, depending on the current pass, next state is selected. The flowchart for termination phase is given in Figure 36.

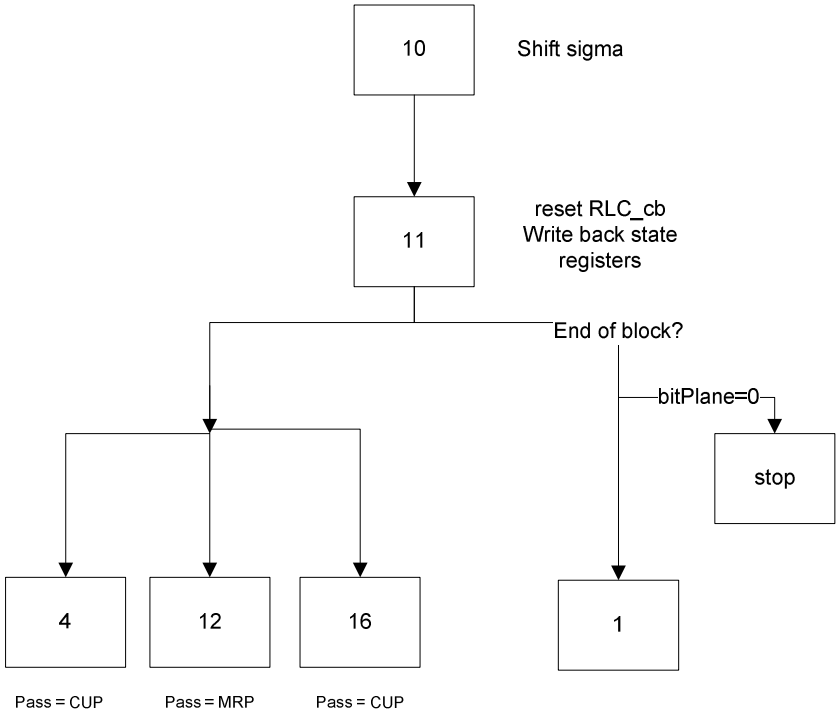

Figure 36 Termination Phase Flowchart

5.3 VLSI Architecture for MQ-Coder

MQ-Coder algorithm just like EBCOT algorithm, consists of sequential operations based on the conditions in the algorithm. Thus, a state-machine based approach is also suitable for MQ-Coder. Details of the MQ-Coder algorithm is described in Section 4.7.2.

MQ-Coder implementation follows the work proposed in [22]. The implementation consists of four registers, an update logic, an adder module, a look-up table for Qe values and a controller circuit. The overall structure of MQ-Coder is presented in Figure 37. Details of each part are listed in the following sections.

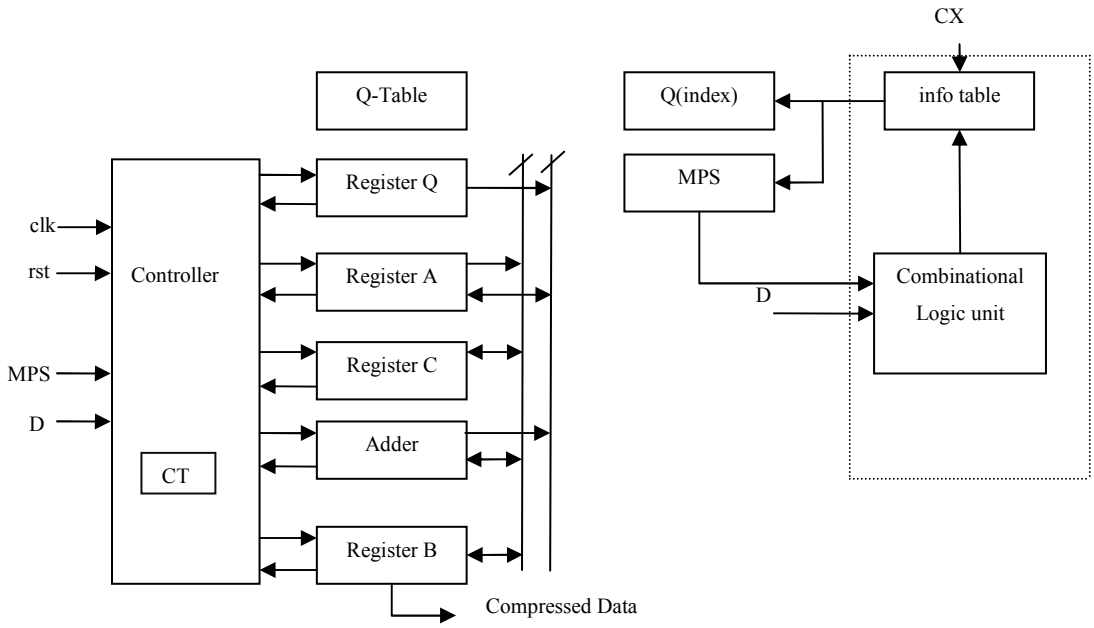

Figure 37 MQ-Coder Structure

5.3.1 Building Blocks of MQ-Coder

5.3.1.1 Registers

MQ-Coder implementation use four special registers, Register A, register C, Register B and Register Q. The details of these register are explained below.

- Register A is a 16-bit register that holds the current interval value during coding operation. Register A is capable of shifting left by 1 bit and most significant bit of Register A (A[15], e.g.) for deciding whether to call *Renormalization* routine or not.

  Since Register A needs only shifting by 1 bit at one clock cycle, an ordinary shift-register is used for Register A. The structure of Register A is illustrated in Figure 38.

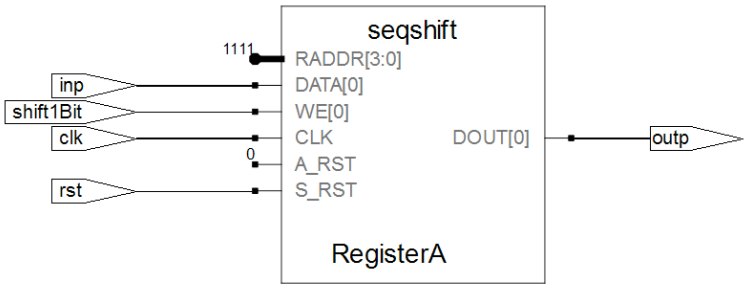

Figure 38 Register A Structure

- Register C is a 32-bit register that holds the coded data at any step of coding operation. Like Register A, Register C also needs to be capable of shifting left by 1 bit. The 28th bit of Register C (C[27], e.g.) is used to determine whether a carry needs to be added to Register B. Thus, two shift-registers in series are used. First shift-register holds the first 28 bits and the second shift-register holds the remaining 4 bits. The structure of Register C is shown in Figure 39.

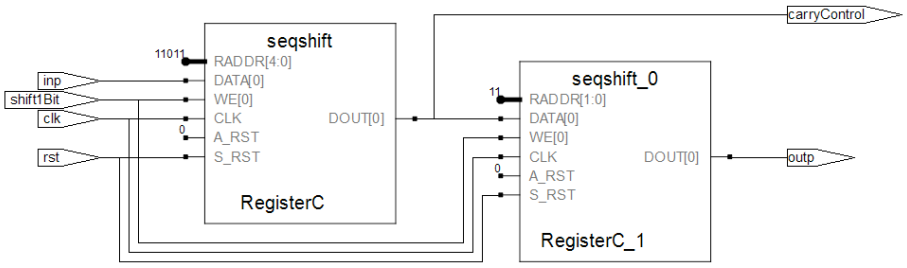

Figure 39 Register C Structure

- Register B is a special purpose 8-bit register. The compressed bit stream is output from Register B. However, there is a detector logic at the end of the Register B that detects whether all the bits in Register B are 1s or not. The structure of Register B is shown in Figure 40.

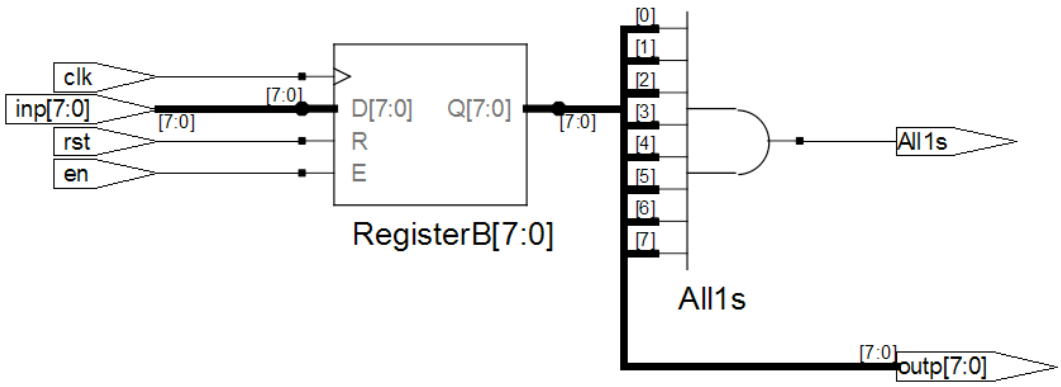

Figure 40 Register B Structure

- Register Q is a regular 16-bit register that holds the probability estimation q_e obtained from Q-table.

5.3.1.2 Update Logic

Update Logic consists of two parts; the info table and the combinational logic circuit. Details are explained below.

- The info table is essentially a RAM block of size 19. The initial values of info table is provided by JPEG2000 standard. Input for the table is a 5-bit CX value and the output of the table is an index value (I(CX). Another input for the info table is the new index value generated by the combinational logic unit of Update Logic.

  The structure of info table implementation is shown in Figure 41. It should be noted that output of the info table is not registered, thus the implementation is an asynchronous RAM block. The output is not registered, data appears at the output of the RAM block when the address data is provided. This selection helps sample time delays of the Update Logic.

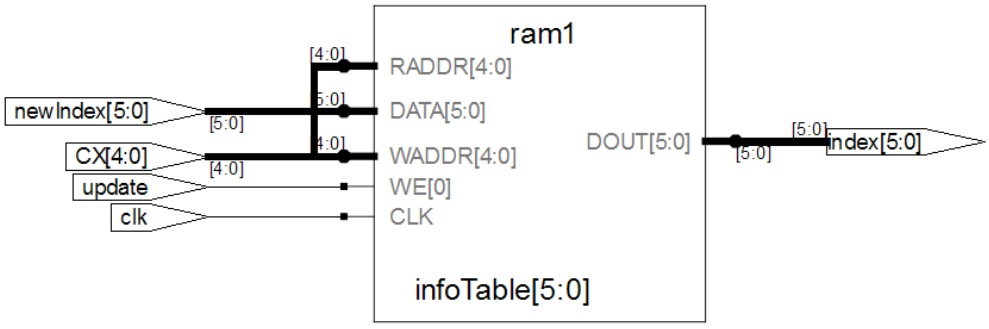

Figure 41 Info Table Structure

- The combinational logic unit calculates the new index value and updates the corresponding location in the info table. The calculation involves selecting the right value out of the Q-Table using three parameters; current index value, sense of MPS and input symbol generated by EBCOT.

5.3.1.3 Adder Module

MQ-Coder algorithm requires addition, subtraction and comparison operations when calculating new interval values and checking the value of Register A. Thus, a single unit capable of running these operations is implemented. Since, the largest of the values is 32-bit, adder module uses 32-bit adders.

Adder Module consists of two adders. One of the adders is used for addition and subtraction operations. Another adder is used for comparison operation. The structure of Adder Module is shown in Figure 42.

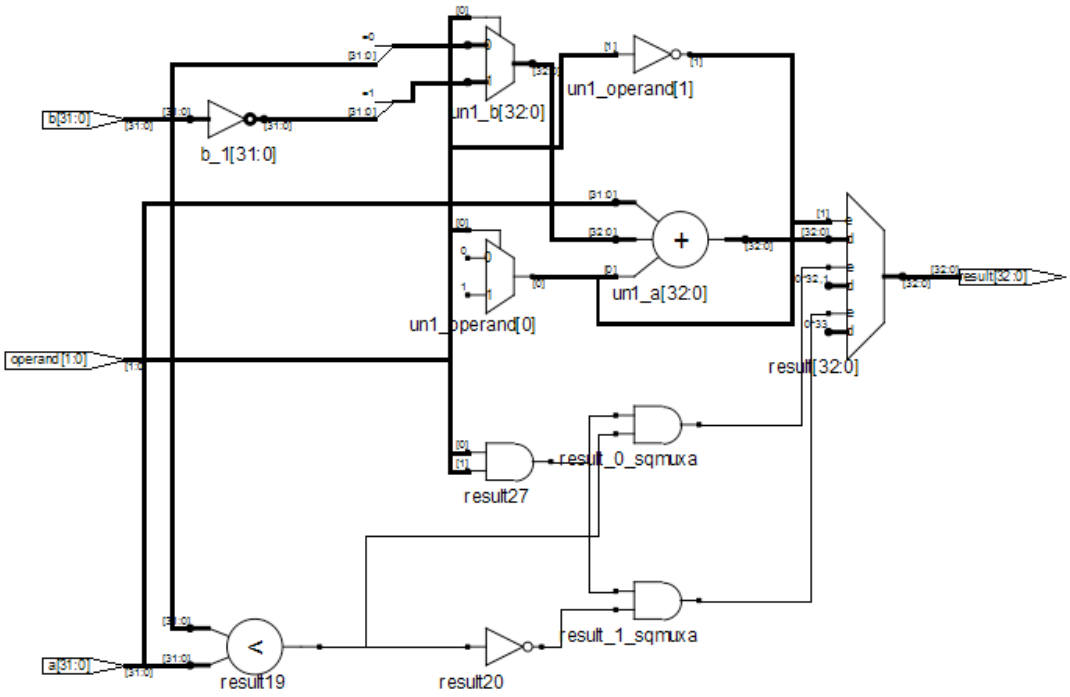

Figure 42 Adder Module Structure

5.3.1.3 Q-Table

Q-table is provided by JPEG2000 standard and it includes predefined probability estimates, Q_e, for CX values. Moreover, Q-Table includes new index values used by Update Logic.

Q-Table uses index value generated out of the info table as its input and generates four values for the index value provided. Thus, Q-Table actually comprises four tables. Since, the values of Q-Table is not changed during coding process, Q-Table is implemented as four ROM blocks. The contents of Q-Table are provided by JPEG 2000 standard [2].

5.3.1.4 MQ-Coder Controller

Controller of MQ-Coder is a state machine that controls all the signals for memories and tables. Controller also has a counter register CT of initial value of 12. Controller checks the value of CT register at every clock cycle. When CT reaches down to zero, compressed code is taken out of Register B, and new data is loaded from Register C.

CHAPTER 6

RESULTS & DISCUSSIONS

In this chapter, the results of the implementation are presented and compared with the available implementations in the literature.

6.1 Results

The aim of this work is to implement a VLSI implementation of JPEG2000 image coding system. Thus, majority of the experiments are carried out on the VLSI aspects of the study. Though, compression performance of the lossless image coding system is also experimented and presented in this thesis.

6.1.1 Compression Performance

Compression performance of the implemented system is experimented using three different color test images; lena, mandrill and aircraft. Compression performance of the implemented system is compared with reference software implementation [23], provided in JPEG2000 Image Coding System, Part 5.

The code sizes generated by the implementation presented in this study, reference implementation and corresponding source images are listed in Table 12.

Table 12 Compressed Data Sizes of Implementations

	lena (byte)	**mandrill (byte)**	**aircraft (byte)**
Original Image	786432	786432	786432
Reference Implementation	450453	597447	384289
Experimented Results	444670	591448	378145

Compression ratios are presented in Table 13. It should be noted that the reference implementation generates a slightly larger compressed data. Reference implementation puts in all the signals and markers needed for any decoder implementation. However, the implementation presented in this study only generates the raw compressed data.

Table 13 Compression Ratios of The Implemented System and Reference Implementation

	lena (byte)	**mandrill (byte)**	**aircraft (byte)**
Reference Implementation	1.7459:1	1.3163:1	2.0465:1
Experimented Results	1.7686:1	1.3297:1	2.0797:1

6.1.2 Results of VLSI Implementation

Results of the implementation are estimated for Xilinx Virtex 5 XC5VSX240T FPGA device. Synopsys Synplify Pro is used for logic synthesis, resource usage analysis, and timing analysis. In order to increase the accuracy of the estimation, all the input and output ports of the individual parts of the system are registered. Results of each section of the implementation are presented in the following sections.

6.1.2.1 DC Level Shifter

DC level shifter only employs adders. Thus, DC Level Shifter Implementation reaches to the limits of the target device. Timing performance and resource usage information of DC Level Shifter are listed in Table 14 and Table 15.

Table 14 Timing results of DC Level Shifter

Timing Results	
Estimated Frequency:	1134.9 MHz
Estimated Period	0.881ns

Table 15 Resource Usage Results of DC Level Shifter

Resource Usage	
FDRE	48 uses
Total LUTs	0 (0%)

6.1.2.2 Reversible Color Transform

Target operation frequency of RCT implementation is set to 1 GHz. Following are the results of logic synthesis.

Table 16 Timing Results of RCT

Timing Results	
Estimated Frequency:	412.8 MHz
Estimated Period	2.371 ns

Table 17 Resource Usage Results of RCT

Resource Usage	
FDRE	48 uses
Total LUTs	36 (0%)

6.1.2.2 Discrete Wavelet Transform

Following results include 3-level DWT with matrix transpose sections. Matrix transpose sections require large amount of memories. Thus, resource usage of the 3-level 2D DWT is expected to be very high. Timing results and resource usage information are presented in Table 18 and Table 19.

Table 18 Timing Results of 3-level 2D DWT

Timing Results	
Estimated Frequency:	388.5 MHz
Estimated Period	2.574 ns

Table 19 Resource Usage Results of 3-level 2D DWT

Resource Usage	
FDRE	990 uses
Total LUTs	1152 (0%)
Block Rams	94 of 516 (18%)

Another useful information about the DWT implementation would be the performance of 1-level 1D transform. Timing performance and resource usage information of 1-level 1D DWT are presented in Table 20 and Table 21, respectively.

Table 20 Timing Results of 1-level 1D DWT

Timing Results	
Estimated Frequency:	504.7 MHz
Estimated Period	1.982 ns

Table 21 Resource Usage Results of 1-level 1D DWT

Resource Usage	
FDRE	102 uses
Total LUTs	Total LUTs: 58 (0%)

6.1.2.3 EBCOT

EBCOT is the most computationally demanding part of the implementation. Most of the overhead of the implementation comes from the controller of the implementation. The timing performance and resource usage of EBCOT implementation are listed in Table 22 and Table 23.

Table 22 Timing Results of EBCOT

Timing Results	
Estimated Frequency:	233.5 MHz
Estimated Period	4.283 ns

Table 23 Resource Usage Results of EBCOT

Resource Usage	
FDRE	121 uses
Total LUTs	366 (0%)
Block Rams	2 of 516 (0%

6.1.2.3 MQ-Coder

Like EBCOT, most of the performance capping part of the MQ-Coder implementation is the controller. However, memory utilization of MQ-Coder is less than EBCOT. Timing results and resource utilization of MQ-Coder are presented in Table 24 and Table 25, respectively.

Table 24 Timing Results of MQ-Coder

Timing Results	
Estimated Frequency:	248.8 MHz
Estimated Period	4.019 ns

Table 25 Resource Usage Results of MQ-Coder

Resource Usage	
FDRE	46 uses
Total LUTs	244 (0%)
Block Rams	1 of 516 (0%)

6.2 Discussions

Resource usage of a VLSI system is of great importance. However, in most of the cases, timing constraints of an implementation take priority. Also, every target architecture does not have the same architectural elements and resource usage of the systems tend to change with the logic synthesis tools used. The logic synthesis tool used in this study targets timing performance over area utilization.

Another concern for comparison is the recentness of the published implementations. FPGA and ASIC industry leap ahead each year and performance of the devices that are available gets better. Also, another concern is that there is not unified decision on the target architecture. While some of the implementations target FPGA devices, some target ASIC devices. Thus, comparison results are absolute and reference implementations may or may not perform on the target architecture selected for this study.

DWT section of the implementation has its critical path inside the transpose part. Thus, it's more meaningful to compare the 1-level 1D transform against other implementations. A similar work is carried out for both filter bank and lifting implementations in [15]. Achieved speed for this implementation 350 nm ASIC device is 50 MHz. Hence, nearly a tenfold increase in speed can be observed.

EBCOT implementation, which is the most costly part of the implementation, is compared to [24], one of the most recent implementations with sufficient data to

compare. The results show that the achievable speed is almost doubled in this study. However, the target device used in the reference implementation use a 180 nm ASIC device. A recent ASIC device should be expected to perform better than the FPGA target device selected in this study.

MQ-Coder implementation is compared to the direct implementation of the cited implementation [22]. In the reference implementation a Virtex II Pro FPGA device is used and 112 MHz target speed is achieved. Taking the speed advancements between Virtex II and Virtex 5 devices into account, the results are comparable.

As shown by the results of the implementation presented in Section 6.1, a VLSI implementation of JPEG2000 image coding system requires large amount of memory. For instance in the first level of DWT, transpose operation requires a RAM block of 2 x frame size, which is 128 kWord. It should be noted that this is the number required for each component. Thus, an image coding system of three components (red, green and blue) would require a 384 kWord of memory just for the transpose operation in the first level of DWT. There exists some work in the literature that use less amount of memory for transpose operation. For example, single buffered implementation that only requires a memory as much as the frame size [25] can be used. However, it is a patented implementation.

JPEG2000 is the next generation image coding standard and is expected to be employed in many imaging applications. Many of these applications will be executed on embedded systems where time to complete encoding and decoding operations are crucial. VLSI implementations offer a great solution for this. This thesis addresses these expectations and presents results based on a recent target device available.

In general, VLSI systems are designed using hardware description languages, such as Verilog HDL and VHDL. This thesis takes a step more and schematic level entry tools and high-level synthesis tools are also used. The main advantage of this methodology is that it allows faster functional verification of the designed system. This methodology also allows easy porting of the design from one platform to another. For example, ASIC designs require the large memory banks to be removed and memory IPs to be used instead. High level synthesis tool used in this thesis

allows easy extraction of memory banks from the designed system based on size criterions.

JPEG2000 image coding system consists of many optional parts and is highly configurable. The configurability of the system makes it suitable for many applications. However, this flexibility comes at a price and the overall system is considered to be quite complex.

Lossless JPEG2000 offers a slight increase in compression ratios against other lossless image coding systems. Considering the complexity of the system, decision for choosing lossless JPEG2000 over any other lossless image coding standards is an important trade-off. For instance, NASA chose JPEG-LS over JPEG2000 for low complexity in its Spirit rover, which sends images from Mars.

Another concern for JPEG2000 standard is the license and patent issues. JPEG committee stated that they agreed over 20 organizations for making JPEG2000 license free. However, JPEG committee also states that there may be submarine patents in the technology and urges the implementers to carry out their own searches.

As a future work, configurability of the system will be exploited. These works will include a DWT module reprogrammable number of transform levels and frame sizes, selective binary arithmetic coding mode will be implemented.

REFERENCES

[1] R. C. Gonzales and R. E. Woods, *Digital Image Processing*, Second Edition ed. Upper Saddle River, NJ, USA: Prentice Hall, 2002.

[2] ISO/IEC, 15444-1, Information Technology - JPEG2000 Image Coding System, Part 1: Core Coding System, Final Commitee Draft, 2000.

[3] D. A Huffman, "A Method for the Construction of Minimum-Redundancy Codes", in *Proceedings of the I.R.E, Vol. 40, No. 9*, pp. 1098-1101, 1952.

[4] T. Acharya and P. T. Tsai, *JPEG2000 Standard for Image Compression*. Hoboken, NJ, USA: Wiley-Interscience, 2005.

[5] ISO/IEC, 14492-1, Lossy/Lossless Coding of Bi-level Images, Final Committee Draft, 2000.

[6] M. Misiti, Y. Misiti, G. Oppenheim, and J. M. Poggi, *Wavelet Toolbox™ 4 User's Guide*.: The MathWorks, Inc., 2007.

[7] S. G. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 7, pp. 674 - 693, July 1989.

[8] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image Coding Using Wavelet Transform", *IEEE Transactions on Image Processing*, vol. 1, no. 2, pp. 205 - 220, April 1992.

[9] I. Daubechies and W. Sweldens, "Factoring Wavelet Transforms Into Lifting Steps", *The Journal of Fourier Analysis and Applications*, vol. 4, pp. 247-269, 1998.

[10] R. Calderbank, I. Daubechies, W. Sweldens, and B. Yeo, "Wavelet Transforms that Map Integers to Integers", *Applied and Computational Harmonic Analysis*, vol. 5, pp. 332-369, July 1998.

[11] A. N. Skodras, C. A. Christopoulos, and T. Ebrahimi, "JPEG2000: The Upcoming Still Image Compression Standard", in *Proceedings of the 11th Portuguese Conference on Pattern Recognition*, Porto, Portugal, pp. 359-366, 2000.

[12] D. Le Gall and A. Tabatabai, "Sub-band Coding of Digital Images Using Symmetric Short Kernel Filters and Arithmetic Coding Techniques", in *Proceedings of IEEE International Conference Acoustics, Speech and Signal Processing*, New York, pp. 761-764, 1988.

[13] D. Taubman, "High Performance Scalable Image Compression with EBCOT", *IEEE Transactions on Image Processing*, vol. 9, no. 7, pp. 1158-1170, July 2000.

[14] D. S. Taubman and M. W. Marcellin, *JPEG2000 Image Compression Fundamentals, Standards and Practice*. Norwell, Massachusetts, USA: Kluwer Academic Publishers, 2002.

[15] J. M. Jou, Y. Shiau, and C. Liu, "Efficient VLSI architectures for the biorthogonal wavelet transform by filter bank and lifting scheme", in *IEEE International Symposium on Circuits and Systems ISCAS 2001.*, Sydney, Australia, pp. 529-532, 2001.

[16] C. Lian, K. Chen, H. Chen, and L. Chen, "Lifting Based Discrete Wavelet Transform Architecture for JPEG2000", in *IEEE International Symposium on Circuits and Systems, ISCAS 2001*, Sydney, Australia, pp. 445-448, 2001.

[17] K. Andra, C. Chakrabarti, and T. Acharya, "A VLSI architecture for lifting-based forward and inverse wavelet transform", *IEEE Transactions on Signal Processing*, vol. 50, no. 4, pp. 966-977, April 2002.

[18] R. B. Sheeparamatti, B. G. Sheeparamatti, Manjula Bharamagoudar, and Nayan Ambali, "Simulink Model for Double Buffering", in *32nd Annual Conference on IEEE Industrial Electronics, IECON 2006*, Paris, France, pp. 4593-4597, Nov 2006.

[19] K. Andra, T. Acharya, and C. Chakrabarti, "Efficient VLSI Implementation of Bit Plane Coder of JPEG2000", in *Proceedings of the SPIE International Symposium on Optical Science and Technology*, San Diego, pp. 246-257, 2001.

[20] J. F. Wakerly, *Digital Design Principles & Practices*, 3rd ed. NJ, US: Prentice Hall, 2000.

[21] P. Gigliotti, Implementing Barrel Shifters Using Multipliers - Xilinx Application Note 195, 2004.

[22] T. Saidani, M. Atri, and R. Tourki, "Implementation of JPEG 2000 MQ-Coder", in *3rd International Conference on Design and Technology of Integrated Systems in Nanoscale Era. DTIS 2008.*, Tozeur, Tunisia, pp. 1-4, 2008.

[23] M. D. Adams. (2007) The JasPer Project Home Page. [Online]. http://www.ece.uvic.ca/~mdadams/jasper/

[24] L. Liu, N. Chen, L. Zhang, and Z. Wang, "VLSI Architecture of EBCOT Tier-2 Encoder for JPEG2000", in *6th International Conference On ASIC, ASICON 2005.*, Shanghai, China, pp. 173 - 176, 2005.

[25] L. J. D'Luna, "Matrix Transpose Memory Device", United States Patent 5177704, April 26, 1991.